||Jai Sri Gurudev ||

BGSKH Education Trust (R.) – A unit of Sri Adichunchanagiri Shikshana Trust(R.)

BGS College of Engineering and Technology (BGSCET)

Mahalakshmipuram, West of Chord Road, Bengaluru-560086

(Approved by AICTE, New Delhi and Affiliated to VTU, Belagavi)

3^{rd} SEMESTER

LECTURE NOTES

2023-2024

TABLE OF CONTENTS

||Jai Sri Gurudev ||
BGSKH Education Trust (R.) – A unit of Sri Adichunchanagiri Shikshana Trust(R.)

BGS College of Engineering and Technology (BGSCET)
Mahalakshmipuram, West of Chord Road, Bengaluru-560086
(Approved by AICTE, New Delhi and Affiliated to VTU, Belagavi)

DATA STRCTURE AND APPLICATIONS Notes

For Third Semester B.E [VTU/CBCS, 2023-2024 Syllabus]

Subject Code: **BCS304**

Data Structures
Module - 1

Data Structure is the organization of the data in a way so that it can be used efficiently.

DS is used to implement an Abstract data type

ADT tells us what is to be done and

DS tells us how to do it

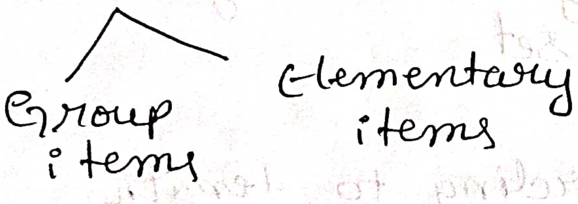
Eg:- In order to implement Stack ADT we can use an array DS or linked list DS.

→ The Logical or mathematical model of a particular organization of data is called Data structure.

Basic Terminology :-

Data : Data are simply values or set of values.

Data items : It refers to a single unit of value



Data items that are divided into sub-items are called Group items.

Eg:- An Employee Name can be divided into three subitems first name, middle name and last name.

Data items that are not able to divide into sub items are called elementary items

Ex- SSN, Tele.NO etc

Entity :- An Entity is something that has certain attributes or properties which may be assigned values. The values may be either numeric or non-numeric.

Ex).				
Attributes -	Name	Age	Sex	SSN
Values -	Jyothi	22	F	1345

Entities with similar attributes form an Entity set.

Field :- is a single Elementary unit of information representing an attribute of an Entity.

Record :- is the collection of field values of a given Entity.

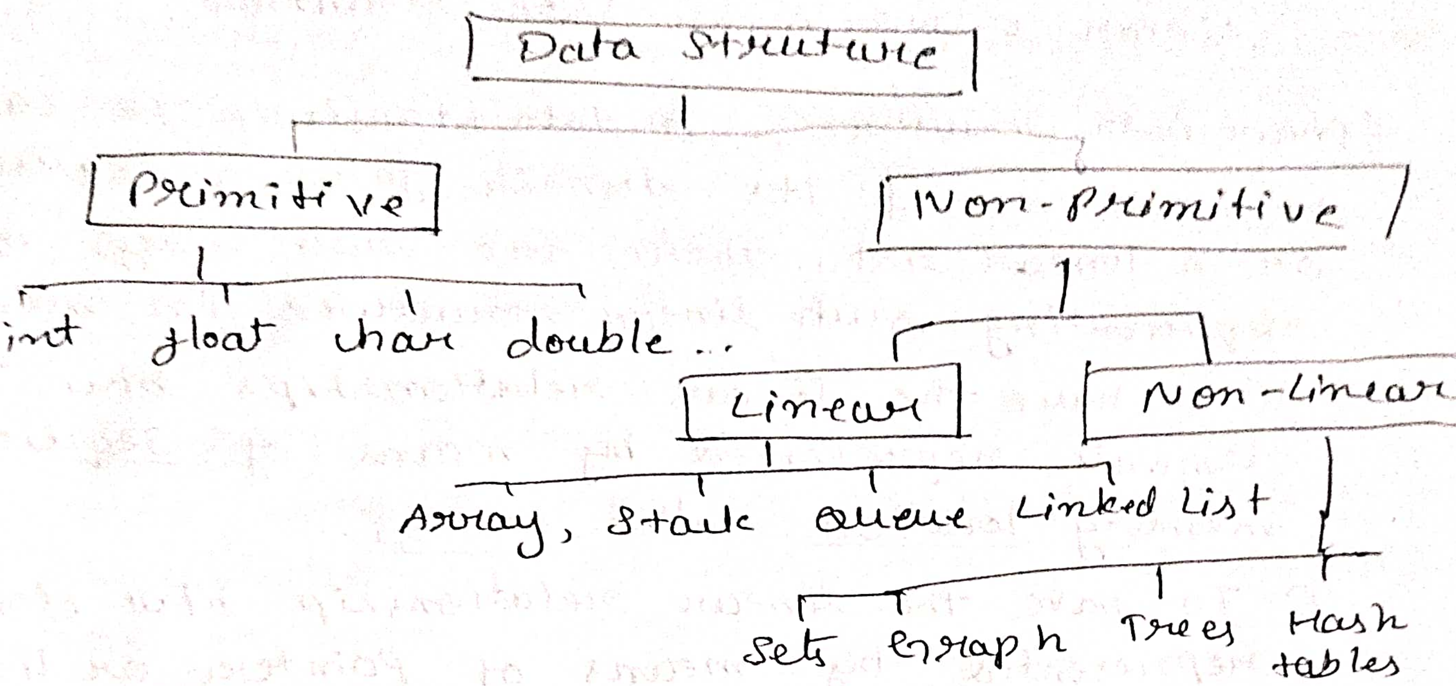
File :- is the collection of records of the Entities in a given Entity set.

Records are classified according to length

Fixed-length records :- All records contain the same data items with the same amount of space assigned to Each data item.

Variable-length record :- file records may contain different lengths.

Classification of Data Structures



Primitive Data Structure :- The fundamental data types which are supported by a programming language. Ex:- integer, real, char and Boolean

→ There consists of characters that cannot be divided & hence called Simple data type.

→ Data structures that can be manipulated directly by the machine instructions are primitive data structures (+, -, *, /)

Non-primitive Data Structure :- There are created by using primitive data structures.
Ex:- linked list, Stack, trees etc

→ Data structures cannot be manipulated directly by the machine instructions.

using insertion deletion

Based on the structure and arrangement of data further classified into
Linear & Non linear data structure.

Linear Data Structure :- A data structure is said to be linear if its elements form a sequence or a linear list. There are two ways of representing such linear structure in memory

① To have the linear relationships b/w the elements represented by means of sequential memory location. called arrays.

② To have the linear relationship b/w element represented by means of pointers or links called linked list.

Common Egs - Arrays, Stack, Queues, Linked List.

Non-linear Data Structure :- Data are not arranged in sequence or a linear. Insertion & deletion of data is not possible in linear fashion. The structure is mainly used to represent data containing a hierarchical relationship between elements.

Eg Graph, trees, Hash tables, Sets.

Data Structures operations

① **Traversing** :- Accessing each record / node Exactly once so that certain items in the record may be processed. This accessing & processing is also called **visiting** the record.

② **Searching** :- Finding the location of the desired node with a given key value or finding the locations of all such nodes which satisfy one or more conditions.

③ **Inserting** :- Add a new node/record to the structure.

④ **Deleting** :- Removing a record /node from the structure.

⑤ **Creating** :- The process of repeatedly adding various data items into list.

⑥ **Sorting** :- Arranging the records in some logical order

⑦ **Merging** :- Combining the records in two different sorted files into a single sorted file.

Pointers

A pointer is a variable which contains the address in memory of another variable.
The two most important operator used are

& – The unary operator which gives the address of a variable.

* – The indirection or Dereference operator gives the content of the object pointed to by a pointer.

Declaration.

```
int i, *pi;
```

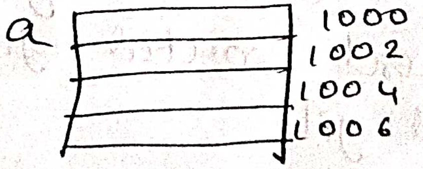
Here, i is the integer variable
pi is a pointer to an integer

```
pi = &i;
```

&i returns the address of i & assigns to the value of pi

Accessing variable value using Pointer

Let int a be a integer variable and &a represents address of a. This address can be stored in another variable. i.e. Pointer.

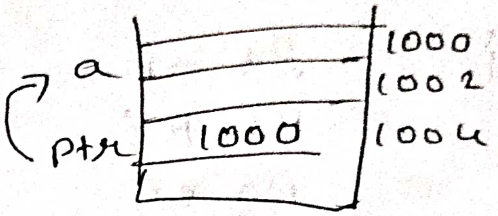


Here address of a is 1000

i.e &a = 1000.

To store this address, we need a pointer Variable. i.e.. int *ptr where ptr is a integer pointer

```
∴   int a;
    int *ptr = &a
```



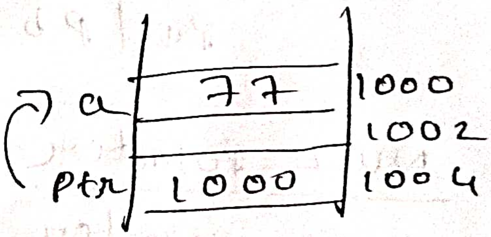
Now we can assign Values to a using

① a and i.e a=77

② ptr *ptr=77

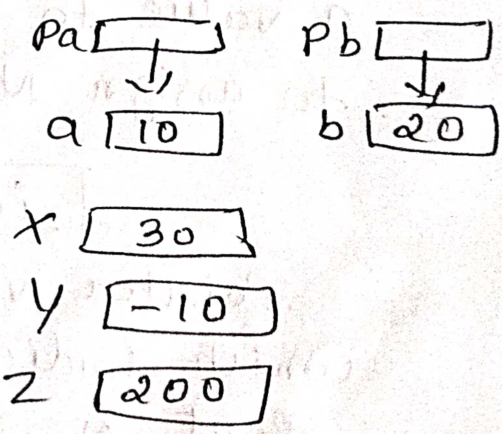
a=77 or *ptr=77 both we can use

```
ie  int a ;
    int *ptr=&a;
    *ptr =7 / a=77
```



Arithmetic operations such as addition, sub, mul & div using pointers.

```
int a=10, b=20
int *pa=&a, *pb=&b;
int x = *pa + *pb;
int y = *pa - *pb;
int z = *pa * *pb;
```



compare whether one pointer is greater or lesser or equal or not equal are allowed.

pa > pb , pa < pb, pa == pb , pa != pb.

If pa & pb are pointers & i is integer

32-bit	64-bit
char-1	1
short-2	2
int 4byte	4
long - 4byte	8

pa ∓ i *(sizeof(datatype))

pa - i = valid

pa + i = valid

pa * i = invalid } ptr variable cannot be used in Mul & Div

pa / i = invalid

pa - pb = valid - gives no of items b/w pa & pb.

pa + pb = Invalid

pa * pb = Invalid

pa / pb : Invalid.

n + 2
3 + 2
5

NULL Pointer

NULL pointer is a special pointer value that points to '\0' (no where) in the memory. If it is too early in the code to assign a value to the pointer, then it is better to assign NULL ('\0' or 0) to the pointer.

$\boxed{\text{int } *p = \text{NULL}}$

Pointer variable p is a NULL pointer. which indicates p does not point to any part of the memory.

The error condition can be checked using

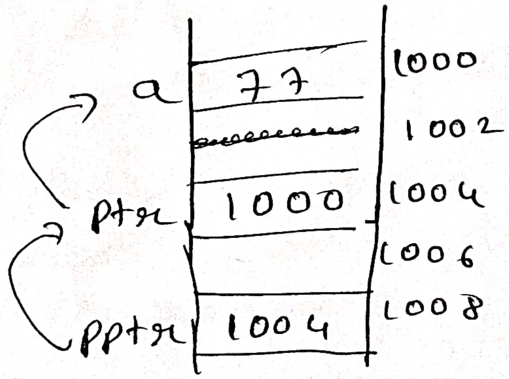
```
if ( P == NULL)
Printf( "P does not point to any memory\n");
else
Printf( "Access the value of P\n");
```

Pointers to Pointer

A pointer can point to another pointer variable. A variable which contains address of a pointer variable is called pointer to – pointer.

Ex-

```
int a ;
int *ptr = &a;
int **pptr = &ptr;
```



a=7 or *ptr=77 or **pptr=77 all are correct way to access "a"

Memory allocation functions

Memory can be allocated for variables using two different techniques:

1. Static allocation (User declarations & definitions)
2. Dynamic allocation (Using predefined functions)

Static memory allocation

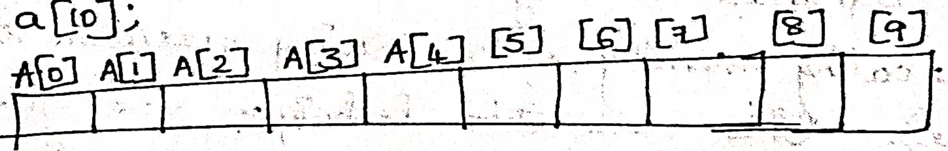
If memory space to be allocated for various variables is decided during compilation time itself, then the memory space cannot be expanded to accomodate more data or cannot be reduced to accommodate less data.

In this technique, once the size of the memory space to be allocated is fixed, it cannot be altered during execution time.

This is called "Static Memory allocation".

For Example Consider the following declaration:

int a[10];



During compilation, the compiler will allocate 10 locations (each location consisting of 4 bytes say) for the variable a.

In the worst case, 10 elements space can be Inserted. Inserting less than 10 elements leads to underutilization of allocated space, and more than 10 elements cannot be Inserted.

Disadvantages

1. The memory space to be allocated is fixed during compilation. Hence the memory space cannot be altered during execution time.
2. Leads to underutilization if more memory space is allocated
3. Leads to overflow if less memory is allocated.

All the above disadvantages can be overcome using dynamic memory allocation.

Dynamic Memory allocation

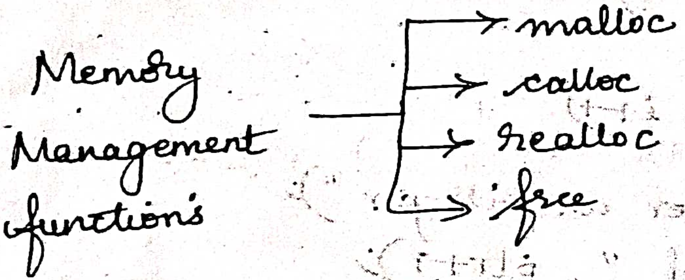
Dynamic memory allocation is the process of allocating memory space during execution time (ie run time). This allocation technique uses predefined functions to allocate and release memory for data during execution time

Dynamic allocation will be used commonly in the following data structures.

dynamic Arrays
Linked lists, trees.

What are the various memory management functions in C?

The various predefined memory management functions that are used to allocate or deallocate memory as shown below



malloc() function: It allocates the required memory space during execution time.

SYNTAX:

```
datatype *p;

p=(data_type*)malloc(size);
```

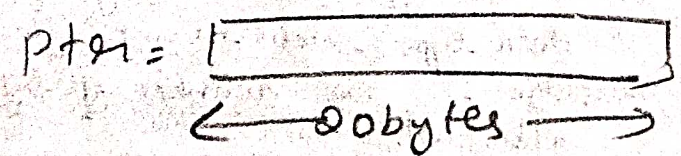
where,
→ p is a pointer variable
→ data_type can be any of the basic C data_type such as int, float, char, etc.
→ size is the number of bytes to be allocated.

Example:

```
#include<stdio.h>
#include<stdlib.h>

int main()
{
  int i,n;
  printf(" Enter the number of integers: ");
  scanf("%d ", &n);
  int *ptr = (int *)malloc (n* sizeof(int));
  if (ptr== NULL)
  {
     printf(" Memory not available");
     exit(1);
  }
  for (i=0; i<n ; i++)
   {
     printf(" Enter an integer ");
     scanf("%d ", ptr+i);
   }
   for (i=0; i<n ;i++)
     printf("%d ",*(ptr +i));
     return0;
}
```

i		
4*0	ptr	— 1000
		— 1004
4*1		
		— 1008
4*2		— 1012
4*3		

int * ptr = (int *) malloc (5 * sizeof(int));



20 bytes memory block is dynamically allocated to ptr

NOTE: It is not possible to write a generalized. function to allocate memory for n elements of any datatype in 'c.

But we can write a generalized macro in place of a generalized function.

The macro definition to allocate memory for n elements of any datatype can be written as shown below:

```
#define MALLOC(p,n,type)                          \
   p=(type *)malloc(n*sizeof(type));              \
   if (p==NULL)                                   \
   {
       printf("Insufficient memory \n");          \
       exit(0);                                   \
   }.
```

→ Macros are normally written in a single line.

→ Since, the body of the macro consists of more number of statements, for readability purpose we may write the body of macro in multiple lines. The symbol backslash (denoted by \) is must at the end of each line.

→ The symbol \ is an instruction given to the preprocessor that all the lines following \ are continuation of the first line of that macro definition.

program showing the usage of macro MALLOC

```
#include<stdio.h>.
#include<stdlib.h>.
#define MALLOC(p,n,type)          \
      p =(type *)malloc(n* sizeof(type)); \
      if (p== NULL)  \
      {                  \
          printf("Insufficient memory\n"); \
          exit(0);          \
      }
void main()
{
      int *p1,*p2;
      int sum;
      MALLOC(p1, 1,int);
      MALLOC(p2, 1,int);

     *p1 = 10;
     *p2 = 20;
     sum =*p1+*p2;
     printf("%d +%d =%d\n",*p1,*p2, sum);
}
```

calloc() function

calloc function allocate the required memory size during execution time and automatically initialize memory with 0's.

SYNTAX :

```
datatype *p;
...
...
p=(datatype *)calloc(n,size);
.....
.....
```

where,

P is a pointer variable

datatype can be any of the basic datatype

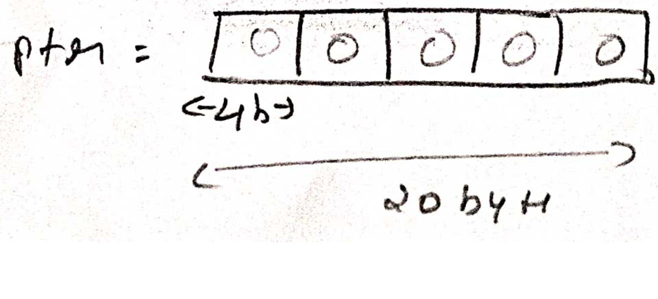
n is the no. of blocks to be allocated.

size is the number of bytes in each block.

* Address of the first byte of allocated space is returned, if memory is successfully allocated.
* The allocated memory is initialized automatically to 0's
* NULL is returned if memory allocation fails.
* The declaration/prototype of calloc() is available in stdlib.h

4bytes

```
int *ptr = (int*) calloc(5, sizeof(int));
```



5 blocks of 4bytes
Each is dynamically allocated to ptr

Macro to allocate memory for one or more Items of any datatype.

```
#define CALLOC(p,n,type)                          \
    p=(type*) calloc(n, sizeof(type));             \
    if(p==NULL)                                    \
    {                                              \
        printf("Insufficient memory\n"}            \
        exit(0);                                   \
    }
```

Example: Program showing the usage of macro CALLOC.

```
#include<stdio.h>
#include <stdlib.h>
#define CALLOC(p,n,type)                   \
 p=(type*) calloc(n,sizeof(type));         \
 if(p==NULL)                               \
                                           \
 {
    printf("Insufficient memory\n");\
                                   \
    exit(0);
 }
void main()
{
    int *p1,*p2;
    int sum;
    CALLOC(p1,1,int);
    CALLOC(p2,1,int);
   *p1=10;
   *p2=20;
    sum=*p1+*p2
    printf("%d+%d=%d\n",
           *p1,*p2,sum);
}
```

3) realloc() function.

Before using this function, the memory should have been allocated using malloc() or calloc().

Sometimes, the allocated memory may not be sufficient and we may require additional memory space.

Sometimes, the allocated memory ~~kes~~ may be much larger and we want to ~~re~~ reduce the size of allocated memory.

In both situations, the size of allocated memory can be changed using realloc() and the process is called reallocation of memory

→ realloc() changes the size of the block by extending or deleting the allocated memory at the end of the ~~fo~~ block.

→ If the existing allocated memory can be extended, ptr value will not be changed.

→ If the existing allocated memory cannot be extended, this function allocates a completely new block and copies the contents of existing memory block into new memory block and then deletes the old memory block.

SYNTAX:

```
#include<stdlib.h>
. . . . .
. . . . .
ptr=(data-type *) realloc(ptr, size);
if (ptr== NULL)
{
   printf ("Insufficient memory\n");
   return;
}
```

where, ptr is a pointer to a block of previously allocated memory either using malloc() or calloc().
size is new size of the block.

NOTE:

On successful allocation, the function returns the address of first byte of allocated memory.

If specified size of memory cannot be allocated, the condition is called "overflow of memory". In such cases, the function returns NULL.

Example: C program showing ~~the~~ usage of realloc()fn.

```
#Include< stdio.h>
#include<stdlib.h>
#include<string.h>
void main()
{
    char *str;
    str= (char *) malloc(10);
    strcpy (str, "Computer");
    printf ("String = %s\n", str);
    str=(char *) realloc(str,40);
    strcpy(str, "Computer Science and Engineering");
    printf ("String = %s\n ", str);
}
```

If we frequently re-allocate the memory space using various data types, then we can write a macro as shown below.

MACRO to re-allocate memory for one or more items of any datatype

```
#define REALLOC(p, n, type)
    p =(type *) realloc (p, n*sizeof(type));
    if (p==NULL)
    {
        printf("Insufficient memory\n");
        exit(0);
    }.
```

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int i ;
    int   *ptr = (int *) malloc(2*sizeof(int));
    if(ptr==NULL)
    {
        printf("Memory not available");
        exit(1);
    }
   printf(" Enter two numbers:\n");
   for(i=0; i<2; i++)
   {
        scanf("%d", ptr+i)
     }
   ptr=(int *) realloc(ptr, 4*sizeof(int))
   if(ptr==NULL)
   {    printf("Memory not available");
        exit(1);
   }
   pf(" Enter 2 more integers:\n");
   for (i=2; i<4; i++)
       scanf("%d",ptr+i);
  return 0;
}.
```

4) free(ptr)

This function is used to de-allocate (or free) the allocated block of memory which is allocated by using the function malloc() or calloc().

It is responsibility of the programmer to de-allocate memory whenever it is not required by the program/ application and initialize pointer variable to NULL.

The syntax is as shown below

```
#include<stdlib.h>
  . . . .
  . . . .
 MALLOC(ptr, 2,int);
  . . . .
  . . . .
  free(ptr);
  ptr=NULL ;
  . . .
  . . .
```

Here ptr is a pointer to a memory block.

NOTE: 1) Even after freeing the memory, the pointer value stored in ptr is not changed.

It still contains the address of the first byte of the memory block allocated earlier.

If the pointer ptr is used even after the memory has been released. it is a logical error.

These logical errors are very difficult to debug and correct

So Immediately after freeing the memory space, it is better to Initialize to NULL as shown below

```
free(ptr);
ptr = NULL;
```

2) The problem wherein memory is reserved dynamically but not accessible to any of the program is called memory leakage or dangling reference. So care should be taken while allocating and deallocating the memory.

It is the responsibility of the programmer to allocate the memory and deallocate the memory when no longer required

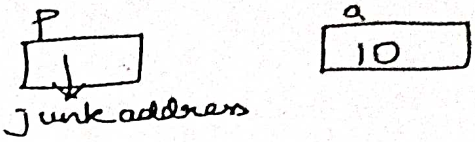
Pointers can be dangerous

The pointers are dangerous in following situations

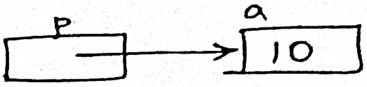
1) We may attempt to access an area of memory that is out of range of our program.

For Example; Consider the statements

```
int *p;
int a=10;
```



```
p=&a;
```



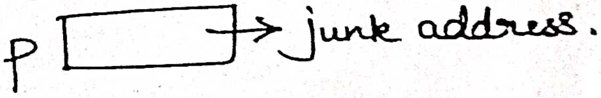
```
printf("%d ",*p);  //output =10
printf("%d ",*(p-1));  //accessing memory which is
                          out of range.
```

Here, *(p-1) refers to the value in previous location which does not exist. So, we are accessing memory location which is outside our program area which eventually may crash the program.

2) The pointer may not contain address of legitimate variable (memory location).

For Ex:, Consider the following program segment:

```
int *p;
```



The pointer p contains Invalid address. So de-referencing p results in unpredictable output & may even crash the program.

3) we may de-reference a NULL pointer. When a NULL pointer is de-referenced, some computers may return 0 and some computers may return data in location zero producing serious error. So we should never try to dereference NULL pointer.

4) In many computers, the size of int datatype and size of pointer is same.

So a valid integer value can be interpreted as a pointer which is very serious error.

For Example, when a function returns an integer value, the returned value can be interpreted as a pointer. This is very serious error.

Abstract Data type (ADT)

ADT is mathematical model for data types, defined by its behavior from the point of view of a user of the data,
Specifically in terms of possible values, possible operations on data of this type.

Array ADT

ADT Array

object : A set of pairs <index, value> where for each value of index there is a value from the set item.
Index is a finite ordered set of one or more dimensions, Eg- $\{0 \ldots n-1\}$ for one dimensio
$\{(0,0),(0,1),(0,2),(1,0),(1,1),(1,2),(2,0),(2,1),(2,2)\}$ for two dimensions etc

functions :-
for all $A \in Array$, $i \in index$, $x \in item$, j, $size \in$

Array Create (j, list) ::= return an array of j dimensions where list is a j-tuple whose ith element is t size of ith dimension. Items are undefined.

```
Item Retrieve (A, i) ::= if (i ∈ index) return
                         the item associated with
                         index value i in array A.
                         else return error.

Array Store (A, i, x) ::= if (i in index)
                          return an array that is
                          identical to array A Except
                          the new pair <i, x> has
                          been inserted
                          else return error.

End Array.
```

ADT is like a blue print which specifies what are the minimum requirements and operations should be defined.

ADT
- Minimum requirements (object)
- Operations (functions).

Arrays

An Array is defined as, an ordered set of similar data items. All the data items of an array are stored in consecutive memory locations.

→ The data items of an array are of same type & Each data items can be accessed using the same name but different index value

→ An array is a set of pairs, <index, value> such that each index has a value associated with it. It can be called as corresponding or a mapping.

Ex:-

<index, value>	
<0, 25>	list [0] = 25
<1, 15>	list [1] = 15
<2, 20>	list [2] = 20
<3, 17>	list [3] = 17
<4, 35>	list [4] = 35

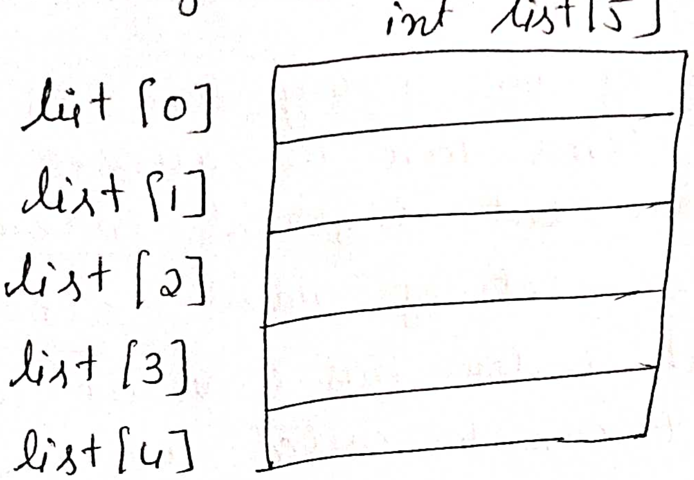
Here list is the name of array.

Array in C

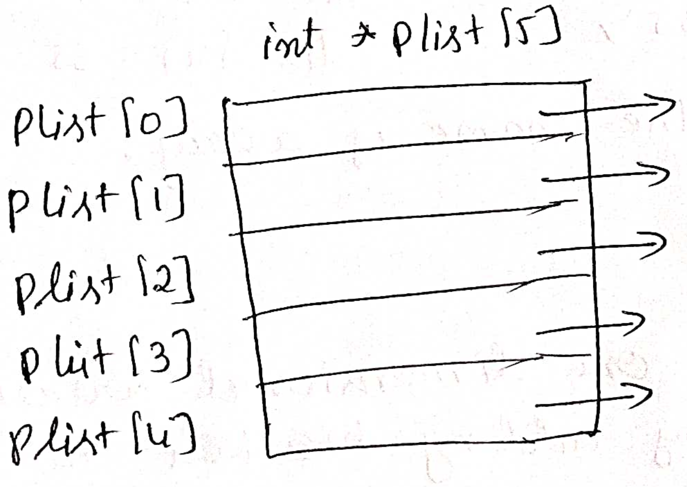
Declaration :- One dimensional array in C is declared by adding brackets to the name of variable.

Ex. int list[5], *plist[5];

→ The array list[5] .defines 5 integers and in C array start at index 0.



→ the array *plist[5]. defines an array of 5 pointers to integers. where plist[0] to plist[4] are five array elements which contains a pointer to an integer.



Implementation

the address of first element of an array is called Base Address .

Ex - For list [5] the address of list [0] is called base address .

-> If the memory address of list [i] need to compute by the compiler , then the size of the int would get by sizeof(int) , the memory address of list [i] is as follows;

$$\boxed{list[i] = \alpha + i * sizeof(int)}$$

where α = base address

list [0]	list [1]	list [2]	list [3]	list [4]
2000	2004	2008	2012	2016

$$list[3] = \alpha + 3 * sizeof(int)$$

$$= 2000 + 3 * 4$$

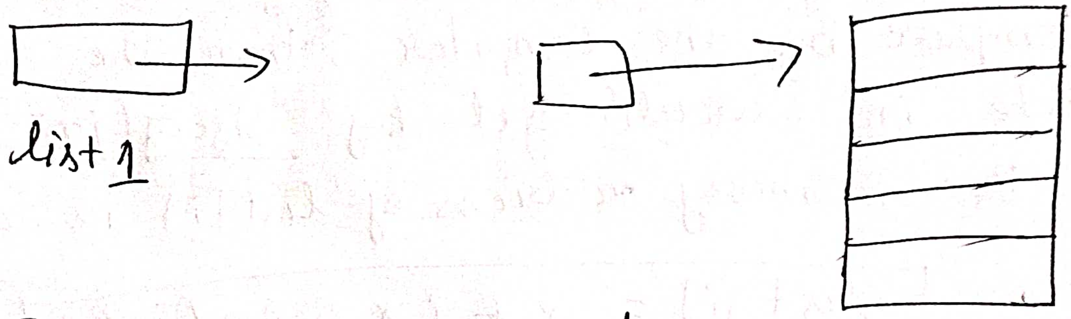
$$= 2000 + 12$$

$$list[3] = 2012$$

Difference between int * list1 and int list2[5]

→ The variables list1 and list2 are both pointers to an int but list2[5] reserves 5 memory locations for holding integers.

int *list1 int list2[5]



Note:- In C the offset i do not multiply with size of type to get to appropriate element of the array.

$$\therefore \boxed{list2 + i = \&list2[i]}$$

and

$$\boxed{*(list2 + i) = list2[i]}$$

A function that prints out both the address of the ith element of the array & the value found at that address

```
void printAV (int *ptr, int rows)
{
    int i ;
    Printf ("Address\t Contents \n");
    for (i=0; i<rows ; i++)
    Printf ( "%u %d \n", ptr+i , *(ptr+i));
    Printf ("\n");
}
```

Here we have to invoke function.

Let int a[5] = {0, 1, 2, 3, 4} be array so we are invoking function using

printAV (&a[0] , 5);

where &a[0] = base address is assigned to pointer variable ptr & rows will be assigned to 5.

O/P.

Address	Contents
1000	0
1004	1
1008	2
1012	3
1016	4

One Dimensional array

```
# include < stdio.h>
# include < stdlib.h>

void main()
{
    int *ptr;
    int size;
  printf(" Enter size of elements ");
  scanf ("%d", &size);

   ptr= (int *) malloc ( size * sizeof (int));
    if ( ptr = = NULL)
   {
     printf ("Memory not allocated \n");
   }
   else
   {
     printf (" Memory allocated successfully");
      printf (" Enter the array elements ");
                  for (i=0; i<size; i++)

        scanf ("%d", &ptr[i]);

  }
printf("The elements of array are:");
for (int k=0; k<size ; k++)
    printf ("%d", ptr[k]);

}
```

Here we are printing array elements using ptr[k] not using *

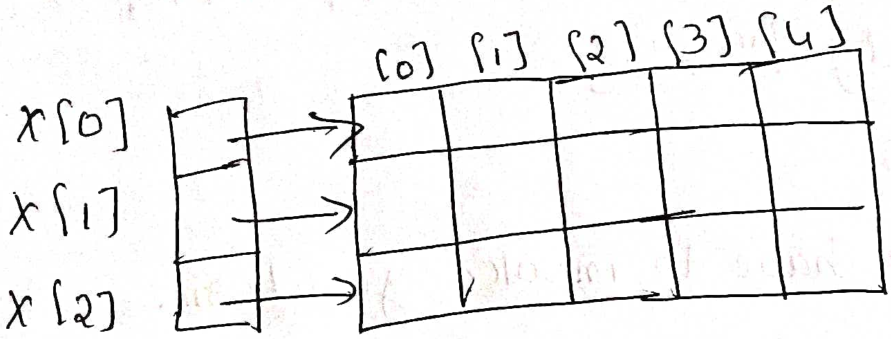
because

$\boxed{*(ptr+i) = ptr[i]}$ if ptr is your base address of array.

Two dimensional arrays

C uses array-of-arrays representation to represent a multidimensional array. The two dimensional arrays is represented as a one-dimensional array in which Each element is itsely a one-dimensional array.

Ex. int x [3][5]



To allocate memory for 2D array we need **Pointer to Pointer variable**.

while allocating row memory we should specify

```
(int **) malloc (rows * sizeof (int *))
```

it points to column

while allocating column we should use

```
(int *) malloc (cols * sizeof(int))
```

2D Array

```
#include <stdio.h>
#include <stdlib.h>

void main ()
{
    int **array;
    int rows, cols, i, j;
   printf ("Enter the no of rows");
   scanf ("%d", &rows);
   printf ("Enter the no of columns");
   scanf ("%d", &cols);
  array = (int **) malloc (rows * sizeof (int*));
                                  ↑ row allocation
    if (array == NULL)
    {
      printf ("Memory allocation failed");
       return (0);
    }
    for (i=0; i<rows; i++)   // for Each row
                               create column space
  {
   array [i] = (int *) malloc (cols * sizeof (int));
                                  ↑ column allocation
    if (array [i] == NULL
  {
    printf ("Memory allocation failed");
  } return (0);
Printf ("Enter the elements of 2D array \n");
  for (i=0; i<rows ; i++)
  {
    for (j=0; j<cols ; j++)
    {
      Scanf ("%d", &array[i][j]);
    }
  }

Printf ("2D array elements are \n");
 for (i=0; i<rows ; i++)
 {
  for (j=0 ; j<cols ; j++)
  {
   Printf ("%d", array[i][j]);
  }
  Printf ("\n");
 }
}
}
```

For 2D array we have to use pointer to pointer variable there fore we declared array as ** array.

Structures

Structures

Definition

- structure is a collection of related data items, of similar / different data types, having a single name".

Syntax :

```
               keyword                identifier
struct          struct_name
   {
      data_type     member1;     }  structure
      data_type     member2;     }  members
         :             :      :
   };
```

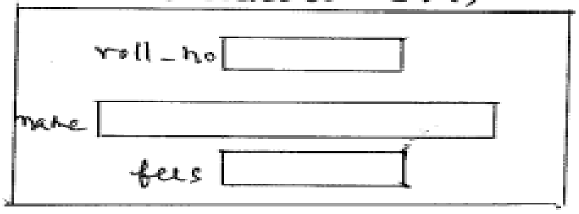
Example

```
      struct   student
        {
           int      roll_no;
           char     name[20];
           float    fees;
        };
```

The variables that are used to store the data are called members of the structure or fields of the structure. In the above structure, roll_no, name and fees are members of the structure.

Representation :

struct student std;



Ex: struct {

char name[10];

int age;

```
                float salary;
        } Person;
```

The above example creates a **structure** and variable name is **Person** and that has three fields:

name = a name that is a characterarray
age = an integer value representing the age of the person
salary = a float value representing the salary of the individual

Assign values to fields

To assign values to the fields, use **.** (dot) as the structure member operator. This operator is used to select a particular member of the structure

Ex:

```
strcpy(Person.name,"james");
Person.age =10;
Person.salary = 35000;
```

Type-Defined Structure

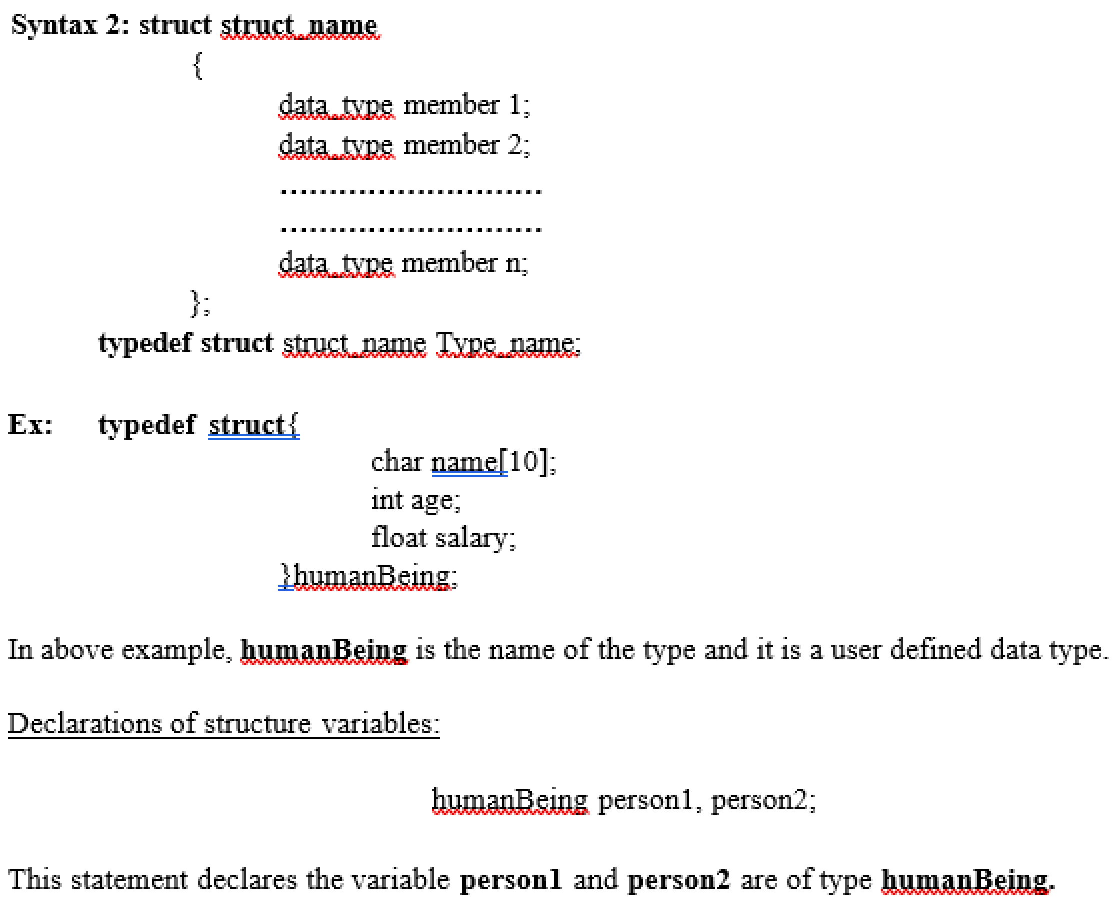
The structure definition associated with keyword **typedef** is called Type-Defined Structure.

Syntax 1: typedef struct

```
        {
                data_type member 1;
                data_type member 2;
                ………………………
                ………………………
                data_type member n;
}Type_name
```

Where,

- **typedef** is the keyword used at the beginning of the definition and by using typedef user defined data type can beobtained.
- **struct** is the keyword which tells structure is defined to the complier
- The members are declare with their data_type
- **Type_name** is not a variable, it is user defined data_type.



Syntax 2: struct struct_name

```
                {
                        data_type member 1;
                        data_type member 2;
                        ...........................
                        ...........................
                        data_type member n;
                };
        typedef struct struct_name Type_name;
```

Ex:

```
        typedef struct{
                        char name[10];
                        int age;
                        float salary;
                }humanBeing;
```

In above example, **humanBeing** is the name of the type and it is a user defined data type.

Declarations of structure variables:

humanBeing person1, person2;

This statement declares the variable **person1** and **person2** are of type **humanBeing.**

Structure Operation

The various operations can be performed on structures and structure members.

1. Structure Equality Check:

Here, the equality or inequality check of two structure variable of same type or dissimilar type is not allowed

```
        typedef struct{
                        char name[10];
                        int age;
                        float salary;
                }humanBeing;
        humanBeing person1, person2;
```

if (person1 = = person2) is invalid.

The **valid function** is shown below

```
#define FALSE 0
#define TRUE 1
```

```
if (humansEqual(person1,person2))
        printf("The two human beings are the same\n");
else
        printf("The two human beings are not the same\n");


int humansEqual(humanBeing person1, humanBeing person2)
    { /* return TRUE if person1 and person2 are the same human being otherwise
                return FALSE */
        if (strcmp(person1.name, person2.name))
                return FALSE;
        if (person1.age != person2.age)
                return FALSE;
        if (person1.salary != person2.salary)
                return FALSE;
        return TRUE;
      }
```

Program: Function to check equality of structures

2. Assignment operation on Structure variables:

person1 = person2

The above statement means that the value of every field of the structure of person 2 is assigned as the value of the corresponding field of person 1, but this is invalid statement.

Valid Statements is given below:

```
strcpy(person1.name, person2.name);
person1.age = person2.age;
person1.salary = person2.salary;
```

Structure within a structure:

There is possibility to embed a structure within a structure. There are 2 ways to embed structure.

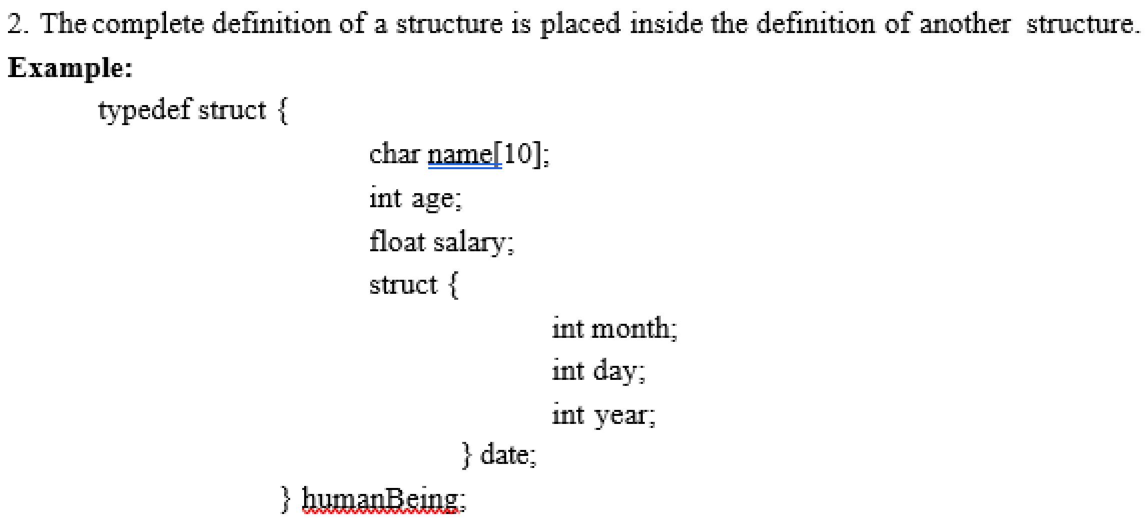
1. The structures are defined separately and a variable of structure type is declared inside the definition of another structure. The accessing of the variable of a structure type that are nested inside another structure in the same way as accessing other member of that structure

Example: The following example shows two structures, where both the structure are defined separately.

```
typedef struct {
                    int month;
                    int day;
                    int year;
             }date;

typedef struct {
                    char name[10];
                    int age;
                    float salary;
                    date dob;

                     } humanBeing;
                     humanBeing person1;
```

A person born on February 11, 1944, would have the values for the date struct set as:

```
person1.dob.month = 2;
person1.dob.day = 11;
person1.dob.year = 1944;
```



2. The complete definition of a structure is placed inside the definition of another structure.

Example:

```
typedef struct {
                    char name[10];
                    int age;
                    float salary;
                    struct {
                                int month;
                                int day;
                                int year;
                          } date;
              } humanBeing;
```

SELF-REFERENTIAL STRUCTURES

A self-referential structure is one in which one or more of its components is a pointer to itself. Self-referential structures usually require dynamic storage management routines (malloc and free) to explicitly obtain and release memory.

Consider as an example:

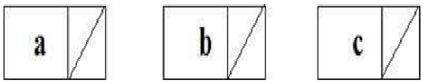
```
          typedef struct {
                              char data;
                              struct list *link ;
                    } list;
```

Each instance of the structure **list** will have two components **data** and **link**.

- **Data:** is a single character,
- **Link:** link is a pointer to a list structure. The value of link is either the address inmemory of an instance of list or the null pointer.

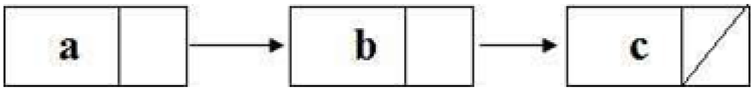
Consider these statements, which create three structures and assign values to their respective fields

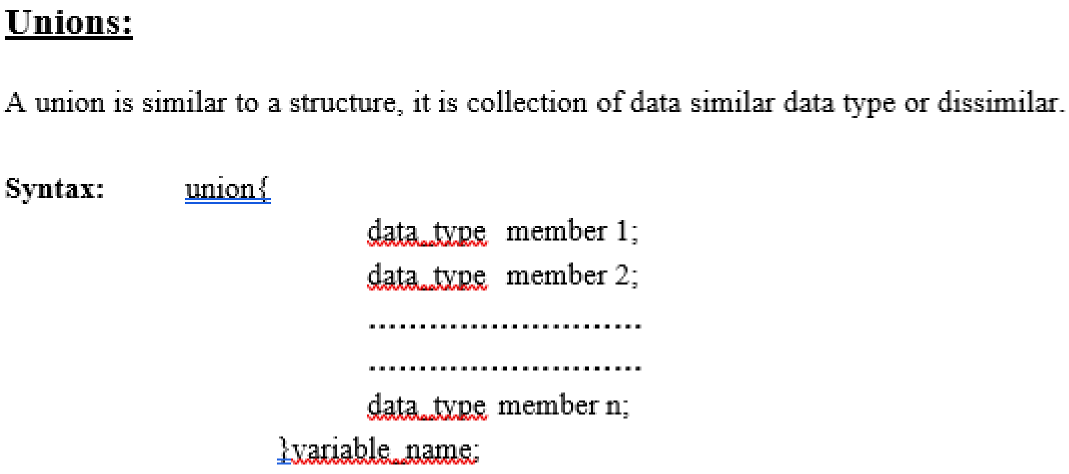
```
List item1, item2, item3;
item1.data=’a’;
item1.data=’b’;
item1.data=’c’;
item1.link= item2.link= item3.link=NULL;
```



Structures item1, item2 and item3 each contain the data item **a, b,** and **c** respectively, and the null pointer. These structures can be attached together by replacing the **null link** field in item 2 with one that points to item 3 and by replacing the null link field in item 1 with one that points to item 2.

```
item1.link = &item2;
item2.1ink = &item3;
```





Unions:

A union is similar to a structure, it is collection of data similar data type or dissimilar.

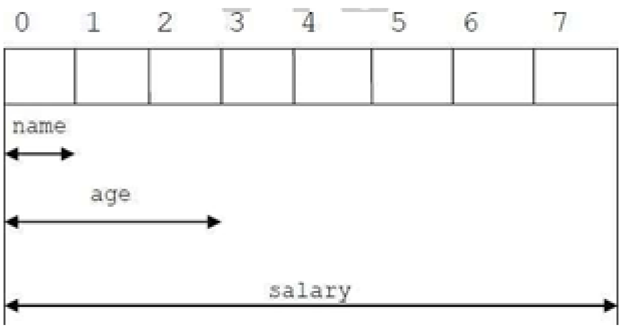
Syntax:

```
union{
        data_type  member 1;
        data_type  member 2;
        ............................
        ............................
        data_type member n;
  }variable_name;
```

Union Declaration:

A union declaration is similar to a structure, but the fields of a union must share their memoryspace. This means that only one field of the union is "active" at any given time.

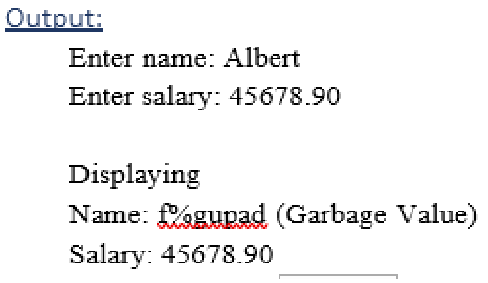
```
union{
        char name;
        int age;
        float salary;
  }u;
```



The major difference between a union and a structure is that unlike structure members which are stored in separate memory locations, all the members of union must share the same memory space. This means that only one field of the union is "active" at any given time.

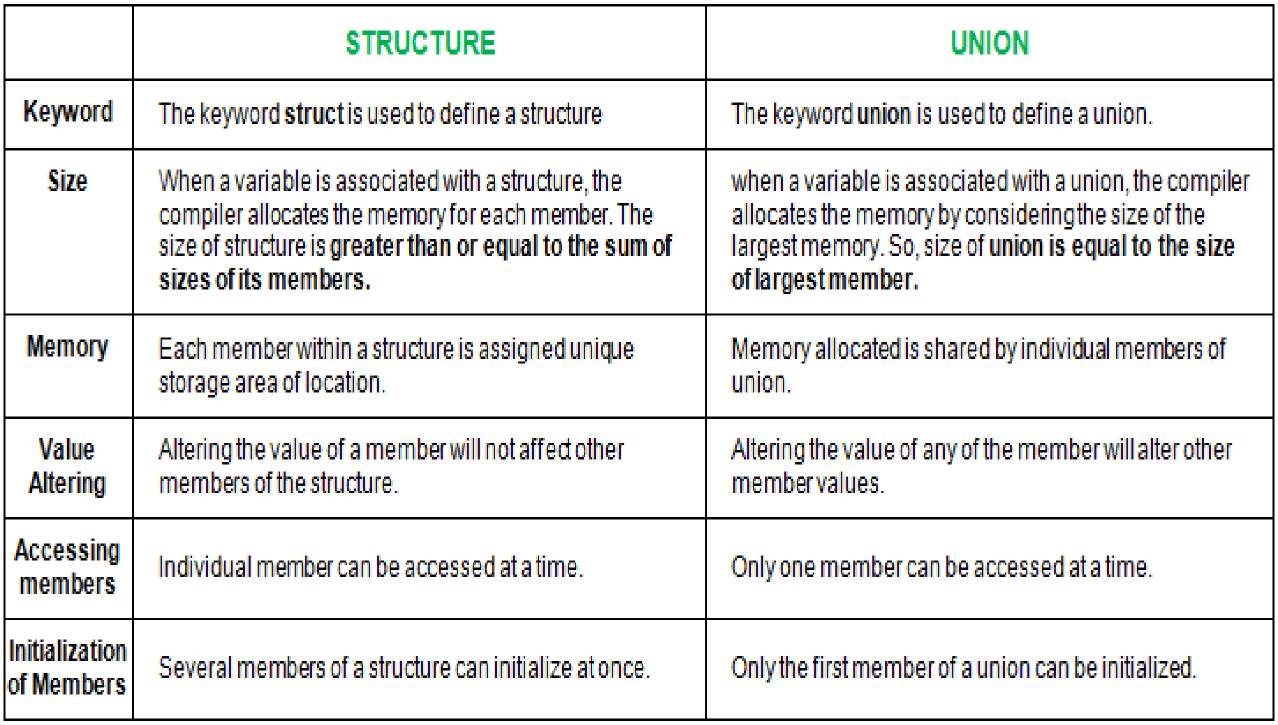
Example:

```
#include <stdio.h>
union job {
                    char name[32];
                    float salary;
                    int worker_no;
                  }u;

int main( ){
                    printf("Enter name:\n");
                    scanf("%s", &u.name);
                    printf("Enter salary: \n");
                    scanf("%f", &u.salary);
                    printf("Displaying\n  Name :%s\n",u.name);
                    printf("Salary: %.1f",u.salary);
                    return 0;
          }
```

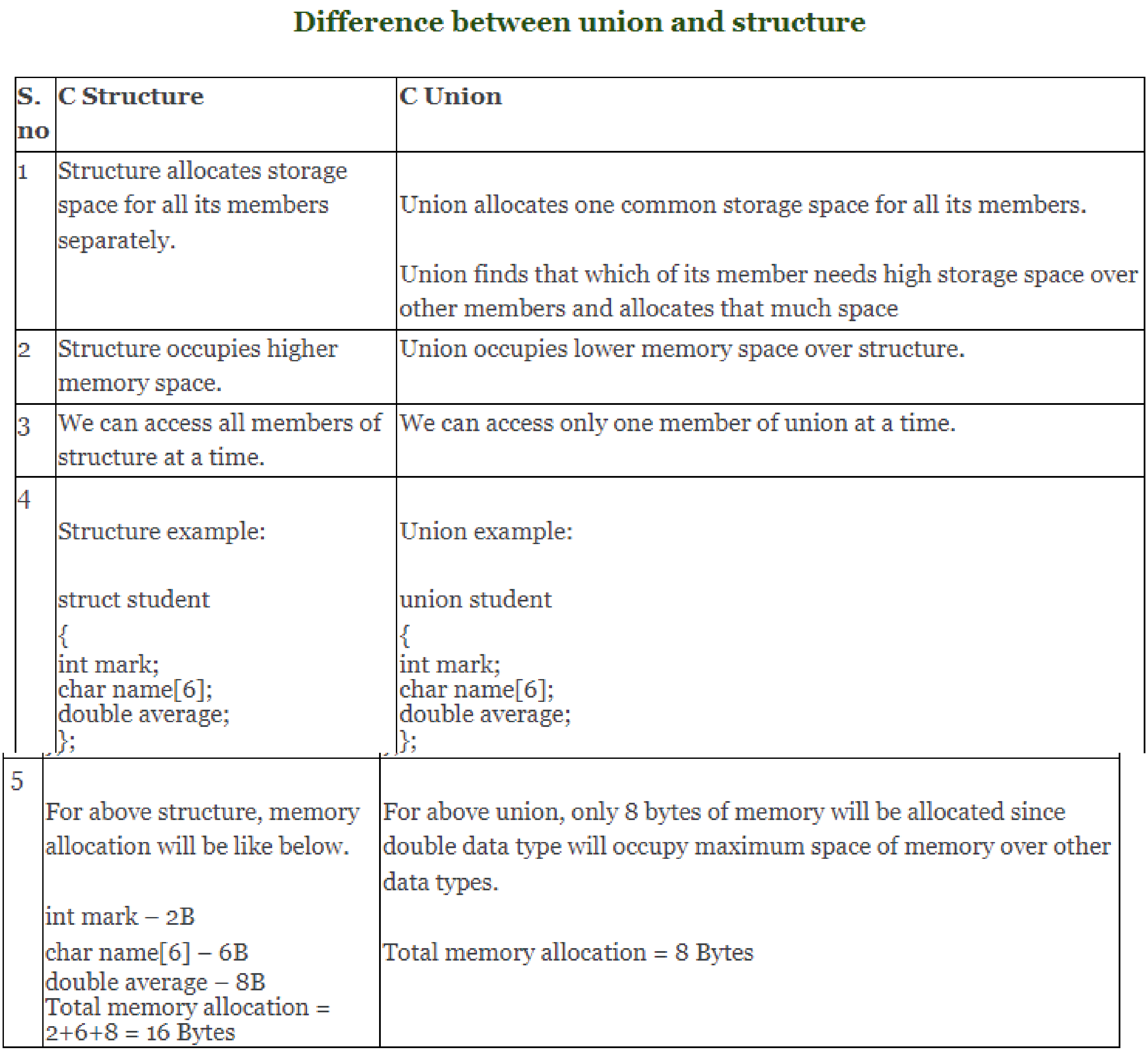


Output:

```
Enter name: Albert
Enter salary: 45678.90

Displaying
Name: f%gupad (Garbage Value)
Salary: 45678.90
```



	STRUCTURE	UNION
Keyword	The keyword **struct** is used to define a structure	The keyword **union** is used to define a union.
Size	When a variable is associated with a structure, the compiler allocates the memory for each member. The size of structure is **greater than or equal to the sum of sizes of its members.**	when a variable is associated with a union, the compiler allocates the memory by considering the size of the largest memory. So, size of **union is equal to the size of largest member.**
Memory	Each member within a structure is assigned unique storage area of location.	Memory allocated is shared by individual members of union.
Value Altering	Altering the value of a member will not affect other members of the structure.	Altering the value of any of the member will alter other member values.
Accessing members	Individual member can be accessed at a time.	Only one member can be accessed at a time.
Initialization of Members	Several members of a structure can initialize at once.	Only the first member of a union can be initialized.



Difference between union and structure

S. no	C Structure	C Union
1	Structure allocates storage space for all its members separately.	Union allocates one common storage space for all its members. Union finds that which of its member needs high storage space over other members and allocates that much space
2	Structure occupies higher memory space.	Union occupies lower memory space over structure.
3	We can access all members of structure at a time.	We can access only one member of union at a time.

4	Structure example:
	struct student
	{
	int mark;
	char name[6];
	double average;
	};
		Union example:
		union student
		{
		int mark;
		char name[6];
		double average;
		};

5	For above structure, memory allocation will be like below. int mark – 2B char name[6] – 6B double average – 8B Total memory allocation = 2+6+8 = 16 Bytes	For above union, only 8 bytes of memory will be allocated since double data type will occupy maximum space of memory over other data types. Total memory allocation = 8 Bytes

Polynomials :-

A Polynomial is sum of terms where each term has a form :

$$a\,x^{e} \xrightarrow{} \text{exponent}$$

↳ variable

↳ coefficient

* The largest exponent (Leading exponent) of a Polynomial is called its degree.

Ex: $6x^{25} + 5x^{10} + 35$

This polynomial is sum of three terms. Since 25 is the largest exponent, the degree of this Polynomial is 25. The last term 35 can also be written $35x^{0}$.

* The Abstract DataType : An ADT Polymonial is the one that shows various operations that can be Performed on polynomials. These Operations are implemented as functions subsequently in the Program:

ADT Polynomial is :

Objects : $p(x) = a_1 x^{e_1} + \ldots\ldots a_n x^n$ where a_i and e_i are coefficients & exponents respectively ; are integers >= 0.

Functions : Parameters used : poly , Poly 1 , Poly 2 ∈ Polynomials,
coef ∈ coefficients
expon ∈ exponents

Boolean IsZero(Poly) ::= if polynomial poly does not exists return true else return false.

Coefficient coef (Poly, expon)::= if expon ∈ Poly, then return coefficient else return 0.

Exponent LeadExp(poly) ::= return the largest exponent of poly.

Polynomial Zero() ::= return empty polynomial.

Polynomial Attach(Poly, coeff, expon)::= If expon ∈ poly then return error else insert term <coeff, expon> into poly & return poly.

Polynomial Remove(poly, expon) ::= If expon ∈ Poly, delete the term <coeff, expon> & return Poly. else return error.

Polynomial Add(Poly1, Poly2) ::= return Poly1 + Poly2

Polynomial mult(Poly1, Poly2) ::= return poly1 * poly2

Example : Let $a(x) = 25x^6 + 10x^5 + 6x^2 + 9$. Calculate :

i) LeadExp(a) = 6

(ii) coef(a , LeadExp (a)) = 25

(iii) IsZero(a) = False

(iv) Attach (a, 15, 3) ⇒ $25x^6 + 10x^5 + \boxed{15x^3} + 6x^2 + 9$
↑ insert

(v) Remove (a, 6) ⇒ $10x^5 + 6x^2 + 9$

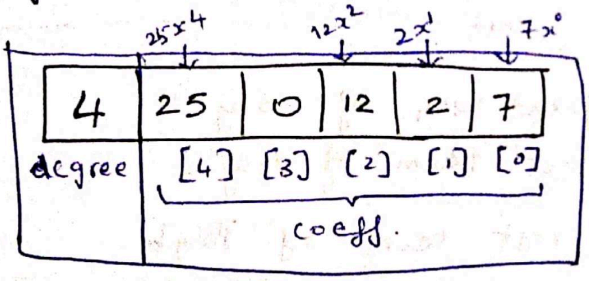
→ Polynomial Representation: ① Polynomials can be represented in C using a structure as shown below:

```
#define   MAX_DEGREE   101
typedef struct
{
      int degree;
      float coeff[MAX_DEGREE];
} Polymomial;
```

Polynomial a, b;

* Using the variable a, the degree of the polynomial can be accessed using a.degree and the coefficients can be accessed using a.coeff[i] for i = 0, 1, 2 . . .

Ex: The polynomial $a(x) = 25x^4 + 12x^2 + 2x + 7$ with degree 4 can be represented as shown:



4	25	0	12	2	7
degree	[4]	[3]	[2]	[1]	[0]

Advantages: ① Simple
② If few terms with zero coefficients are present, it uses less space.

Disadv: If most of the terms are not present, then we store 0's in corresponding coeff's. & occupy more space. This can be overcomed using array of structures.

② Another method using Array of structures :-

* can be used to store several Polynomials.
* This method is used to save space.
* Each Term of a polynomial with 2 fields : coeff and expo can be stored in array location P[0], P[1], P[2]....

```
#define MAX-DEG 100
typedef struct
{
        float coef;
        int expo;
} polynomial;

Polynomial P[MAX-DEG];
```

Ex: $A(x) = 2x^{1000} + 5$ and $B(x) = x^4 + 10x^3 + 3x^2 + 1$

can be represented as:

	$2x^{1000}$	$5x^0$	x^4	$10x^3$	$3x^2$	x^0	
Coef	2	5	1	10	3	1	- - -
Expo	1000	0	4	3	2	0	- - -
	[0]	[1]	[2]	[3]	[4]	[5]	[6] ↑
	↑ StartA	↑ endA	↑ StartB			↑ endB	avail.

* Start A - index of first term of poly a

end A - index of last term of poly a

StartB - index of first term of poly b

end B - index of last term of poly b.

avail - index of next free space, where a term of a polynomial can be stored.

Disadv: If more no. of non-zero coefficients are present, then it occupies twice the memory occupied by previous method.

→ Polynomial Addition :-

① Design an algorithm to add 2 polynomials using ADT Polynomial [c = a + b].

* The addition of 2 polynomials can be done ~~using~~ by considering various cases as shown:

Case 0: Powers of two terms to be added are equal.

⇒ Assume that the 1st term of foll. polynomials are to be added:

$a = 25x^6 + 10x^5 + 6x^2 + 9$ ---> LeadExp(a) = 6

$b = 15x^6 + 5x^4 + 4x^3$ ---> LeadExp(b) = 6

$c = 40x^6$

∵ LeadExp(a) and LeadExP(b) are same, they can be added:

sum = coef(a, LeadExp(a)) + coef(b, LeadExp(b))

= 25 + 15

= 40 // Now add into poly C

* This can be done by calling the attach fn as:

if(sum != 0) Attach(C, sum, LeadExp(a));

* Now, move to the next term of poly a, and poly b, by removing the added terms ie;

a = Remove(a, LeadExp(a)); ie; $25x^6$

b = Remore(b, LeadExp(b)); ie; $15x^6$

∴ $a = 10x^5 + 6x^2 + 9$ and

$b = 5x^4 + 4x^3$

⇒ The complete code can be written as:

```
if (LeadExp(a) == LeadExP(b))
{
    sum = coef(a, LeadExp(a)) + coef(b, LeadExp(b));
    if(sum != 0) Attach(C, sum, LeadExp(a));
    a = Remove(a, LeadExp(a));
    b = Remore(b, LeadExp(b));
}
```

Case 1: Power of 1st term of poly a is greater than power of 1st term of poly b. ie;

$a = 10x^5 + 6x^2 + 9$ ---> LeadExp(a) = 5

$b = 5x^4 + 4x^3$ ---> LeadExp(b) = 4

$C = 10x^5$

∵ $10x^5 > 5x^4$, we attach $10x^5$ into C. and then we remove it before moving to the next of poly a ie;

→ complete code:

```
if (LeadExp(a) > LeadExp(b))
{
    Attach(c, coef(a, LeadExp(a)), LeadExp(a));
    a= Remove(a, LeadExp(a));
}
```

default: If Ist term of Poly B > Ist term of Poly A

```
if (LeadExp(b) > LeadExp(a))
{
    Attach(c, coef(b, LeadExp(b)), LeadExp(b));
    b= Remove(b, LeadExp(b));
}
```

* All the above cases have to be repeated until one or both polynomials become empty. If one of the poly becomes empty, copy the remaining terms into c.

* The above logic can be written using a COMPARE() macro ie; compares whether the exponents of a and b are same, greater than or lesser than as shown below:

```
c= Zero();
while (!IsZero(a) && !IsZero(b))
{
    switch(COMPARE(LeadExp(a), LeadExp(b))
    {
        case 0: sum= coef(a, leadExp(a)) + coef(b, LeadExp(b));
                if(sum!=0) Attach(c, sum, LeadExp(a));
                a= Remove(a, LeadExp(a));
                b= Remove(b, LeadExp(b));
                break;
case 1: Attach(C, coef(a, LeadExp(a)), LeadExp(a));
        a= Remove(a, LeadExp(a));
        break;
default: Attach(c, coef(b, LeadExp(b), LeadExp(b));
         b= Remove(b, LeadExp(b));

   }

 }
// Insert any remaining items of a or b into c.
```

② To add Two polynomials using array of structures :-

```
void polyAdd(Polynomial *P, int startA, int endA, int StartB,
                 int endB , int * startC, int *endc)
{    *startC= avail;   // get index of 1st term of resulting poly
   while(startA <= endA && startB<= endB)
   {
      switch(COMPARE(P[startA].expon, P[startB].expon))
      {
        case 0: coef = P[startA].coef + P[startB]. coef;
                if(coef != 0) attach(coefs, P[startA].expon, P);
                startA++;  startB++ ; break;
        case 1 : attach(P[startA].coef , P[startA].expon, P);
                StartA++;  break;
        default : attach(P[startB].coef , P[startB].expo, P);
                startB++;
      }
   }
 while (startA <= endA) //Add remaining terms of Poly A.
 { attach (P[startA].coef , P[startA].expo , P);
    start A++;
 }
 while (startB <= endB)   // Add remaining terms of Poly B.
 { attach (P[startB].coef , P[startB].expo , P);
    startB++;
 }
    * endC=avail-1;
}
```

Sparse Matrices :-

* A sparse matrix is a matrix which has more number of zero elements or a very few non-zero elements.

* It can be a 1D, 2D etc

Ex:

$$a\ \begin{array}{|c|c|c|c|c|c|}\hline 0 & 10 & 0 & 0 & 12 & 0 \\ \hline \multicolumn{1}{c}{0} & \multicolumn{1}{c}{1} & \multicolumn{1}{c}{2} & \multicolumn{1}{c}{3} & \multicolumn{1}{c}{4} & \multicolumn{1}{c}{5} \end{array}$$

and

$$\begin{array}{c|c|c|c|c|} & 0 & 1 & 2 & 3 \\ \hline 0 & 10 & 0 & 0 & 40 \\ \hline 1 & 11 & 0 & 20 & 0 \\ \hline 2 & 0 & 0 & 0 & 0 \\ \hline 3 & 20 & 0 & 0 & 50 \\ \hline \end{array}$$

* The Abstract Data Type: An ADT sparse matrix is the one that shows the various operations that can be performed on sparse matrices. It consists of various objects (ie: variables) and functions as shown below:

ADT Array is

Objects: A set of triples ⟨row, col, val⟩ where val is the data stored in a[row][col].

Functions: Parameters used: a, b ∈ Array, MaxRow, maxCol ∈ index, n ∈ integer.

SparseMatrix Create(maxRow, maxCol) ::= Creates a 2D array with row & cols.

SparseMatrix Transpose(a) ::= Returns the transpose of a given matrix. The row elements are changed to column elements & vice-versa

SparseMatrix Add(a, b) ::= If size of two matrices are same, the fn performs sum of a and b else return 0.

SparseMatrix Multiply(a, b) ::= Let mxn and pxq are the size of matrix A & B resp.

If n and p are same then multiplicati is Possible. Otherwise return 0.

The matrix C:

$$c[i][j] = \sum_{k=0}^{n-1} (a[i][k] * b[k][j])$$

for $i = 0$ to $m-1$ and $j = 0$ to $q-1$

End Array.

*Disadvantages of Sparse matrices: A sparse matrix contains many 0's. If we are manipulating only non-zero values, then we are wasting the memory space by storing unecessary zero values.

→ This can be overcomed by storing only non-zero values.

* Sparse matrix Representation: A sparse matrix can be created using the array of triples as:

```
#define MAX 100
typedef struct
{
      int row;
      int col;
      int val;
} TERM;
TERM  a[MAX_TERMS];
```

Example: Represent the given matrix using triples in a 1D array.

* The non-zero elements of the matrix along with row & col position, starting from a[1].

	row	col	val	
a[0]	5	4	8	
a[1]	0	0	10	row 0
a[2]	0	3	40	
a[3]	1	0	11	row 1
a[4]	1	2	22	
a[5]	3	0	20	row 3
a[6]	3	3	50	
a[7]	4	1	15	row 4
a[8]	4	3	25	

	0	1	2	3
0	10	0	0	40
1	11	0	22	0
2	0	0	0	0
3	20	0	0	50
4	0	15	0	25

* The various information can be accessed as shown:

→ The size of matrix : a[0]. row, a[0].col

→ The no of non-zero values : a[0].val

→ The row index of a non-zero element ; a[j].row

→ The column index of a non-zero element ; a[j].col

→ The index of non-zero element : a[j].val

* Function to read the sparse matrix as a triple

```
void   readspmatrix(TERM a[ ], int m, int n)
{
        int  i, j, k, item;
        a[0].row = m, a[0].col = n, k = 1;
         for(i=0; i<m; i++)
          {
             for(j=0; j<n; j++)
              {
                  scanf("%d", & item);
                  if (item ==0) continue;
                  a[k].row = i, a[k].col = j, a[k].val = item;
                  k++;
              }
          }
         a[0].val = k-1;
}
```

* Transpose of a matrix: A matrix which is obtained by changing row elements into column elements and vice-versa is called transpose of a matrix.

⇒ How to represent a sparse matrix into 1D array with triples as well as tranfoim it into a transpose matrix.

Sparse matrix.

	rows	col	val
a[0]	5	4	8
a[1]	0	0	10
a[2]	0	3	40
a[3]	1	0	11
a[4]	1	2	22
a[5]	3	0	20
a[6]	3	3	50
a[7]	4	1	15
a[8]	4	3	25

Transpose ⇒

Transpose Sparse matrix

	row	col	val
b[0]	4	5	8
b[1]	0	0	10
b[2]	0	1	11
b[3]	0	3	20
b[4]	1	4	15
b[5]	2	1	22
b[6]	3	0	40
b[7]	3	3	50
b[8]	3	4	25

$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 10 & 0 & 0 & 40 \\ 1 & 11 & 0 & 22 & 0 \\ 2 & 0 & 0 & 0 & 0 \\ 3 & 20 & 0 & 0 & 50 \\ 4 & 0 & 15 & 0 & 25 \end{array}$$

⇓ Transpose

$$\begin{array}{c|ccccc} & 0 & 1 & 2 & 3 & 4 \\ \hline 0 & 10 & 11 & & 20 & \\ 1 & & & & & 15 \\ 2 & & 22 & & & \\ 3 & 40 & & & 50 & 25 \end{array}$$

* Function to find the transpose of given Sparse Matrix

```
void  transpose (TERM  a[], TERM  b[]).
{
        int  i, j, k;
        b[0]. row  =  a[0]. col;
        b[0]. col  =  a[0]. row;
        b[0]. val  =  a[0]. val;
         k=1;   // Position  of  1st  non-zero element
         for (i= 0 ;  i<  a[0].col ; i++)

              for(j = 1;  j <= a[0]. row;  j++)
                  if (a[j].col = = i)
                  {      b[k].row = a[j].col;
                         b[k]. col  =  a[j].row;
                         b[k]. val  =  a[j] . val;
                         k++;
                  }

}
```

strings :-

Set of zero or more characters are called strings.

ADT String is

object :- a finite set of zero or more characters.

Functions :-
for all s, t ∈ string, i, j, m ∈ non-negative integers.

String NULL(m) ::= return a string
whose maximum length is
m characters, but is
initially set to NULL. we
write NULL as " ".

Integer Compare (s, t) ::= if s equals
t return 0
else if s precedes t
(s<t) return -1
else return +1.

Boolean IsNull (s) ::= if (Compare(s, NULL
return False else
return True.

Integer Length (s) ::= if (Compare(s, NULL))

return the no of characters
in s else return 0

String Concat (s,t) ::= if (compare(t, NULL))
return a String whose elements are those of s followed by those of t.
else return s.

String Substr (s, i, j) ::= if ((j>0) && (i+j-1) < Length(s))
return the String containing the characters of s at positions i, i+1, ---, i+j-1
else return NULL.

End String.

⇒ Strings in C

In C we represent strings as character arrays terminated with the null character \0 .

```
# define MAX_SIZE 100
char s [MAX_SIZE] = { "dog" };
char t [MAX_SIZE] = { "house" };
```

s[0]	s[1]	s[2]	s[3]
d	o	g	\0

t[0]	t[1]	t[2]	t[3]	t[4]	t[5]
h	o	u	s	e	\0

Technically, we can declare

```
char s[] = {"dog"};
char t[] = {"house"};
```

c String junctions

① char *strcat (char *dest, char *src)
⇒ It concatenate dest and src strings return result in dest.

② char *strncat (char *dest, char *src, int n)
⇒ contatenate dest and n characteres from src ; return result in dest.

③ char *strcmp (char *str1, char *str2)
⇒ It compare two strings
return 0 if str1 = str2
return -1 if str1 < str2
return +1 if str1 > str2.

④ char *strncmp (char *str1, char *str2, int n
⇒ It compare first n characteres
return 0 if str1 = first n (str2)
-1 if str1 < first n (str2)
+1 if str1 > first n (str2).

⑤ char *strcpy (char *dest, char *src)
⇒ It copy src into dest
return dest.

⑥ char *strncpy (char *dest, char *src, int n)
⇒ copy n characteres from src into dest & return dest.

⑦ size_t strlen (char *s)
⇒ it return the length of s

(8) char *strchr(char *s, int c)
⇒ it return pointer to the first occurrence of c in s.
return NULL if not present.

(9) char *strrchr(char *s, int c)
⇒ it return pointer to last occurrence of c in s
return NULL if not present.

(10) char *strstr(char *s, char *pat
⇒ it return pointer to start of pat (pattern) in s

String insertion

we want to insert str2 into str1 starting at ith position of str1

Ex- str1 = amobile
str2 = uto

at last we have to get automobile

s →	a	m	o	b	i	l	e	\0

t →	u	t	o	\0

let take temp, empty string

temp ⟶ | \0 |

⇒ copy the first i characters from s into temp.

strncpy (temp, s, i)   i = 1
∴ strncpy (temp, s, 1)

temp → | a | \0 |

⇒ ~~[illegible]~~ concatenate temp with t

strcat (temp, t)

⇒ temp → | a | u | t | o | \0 |

⇒ Now we append remaining of s to temp. Since strncat copied the first i characters, the remaining string is at address (s+i)

∴ strcat (temp, (s+i))

temp → | a | u | t | o | m | o | b | i | l | e | \0 |

```
void strnins (char *s , char *t ,
                            int i)
{ /*insert string t into string s
       at position i */
   char string [MAX_SIZE] , *temp ;
       temp = string ;

   if ( i<0 && i > strlen(s))
 {
    fprintf ( stderr, "position is out
                of bounds \n");
    Exit ( EXIT_FAILURE );
  }
   if  ( ! strlen (s))
       strcpy  (s, t) ;
   else  if  (strlen (t))
   {
       strncpy (temp, s, i) ;
       strcat  (temp, t) ;
       strcat  (temp, (s+i)) ;
       strcpy  (s, temp) ;
   }
 }
```

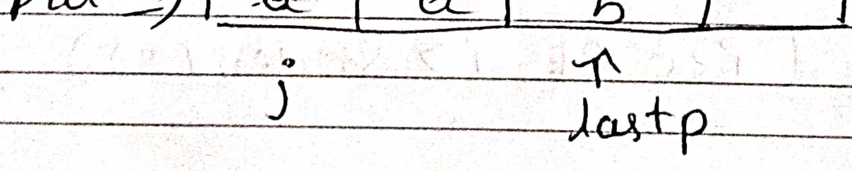
Pattern Matching

① Pattern matching by checking End indices first (nfind)

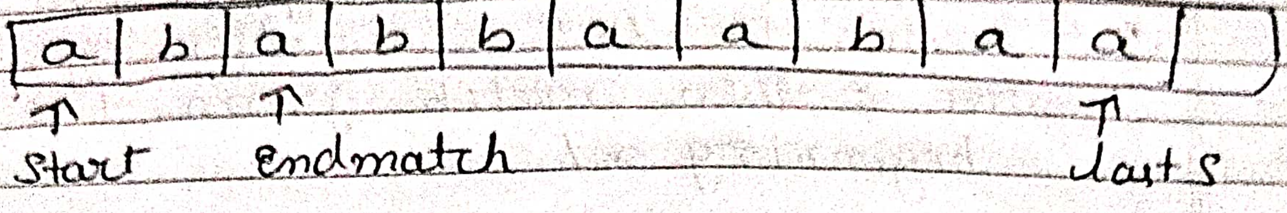
If string = " ababbaabaa " &
  Pat = "aab".

End of the String = lasts
End of the Pat = lastp

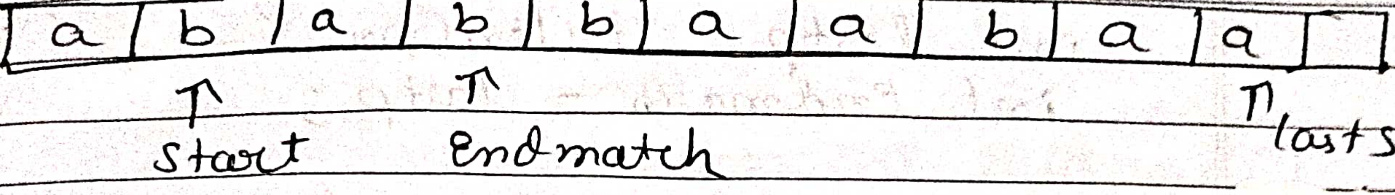
Compare string [end match] & pat [lastp]
If matches use i & j to move through two strings.



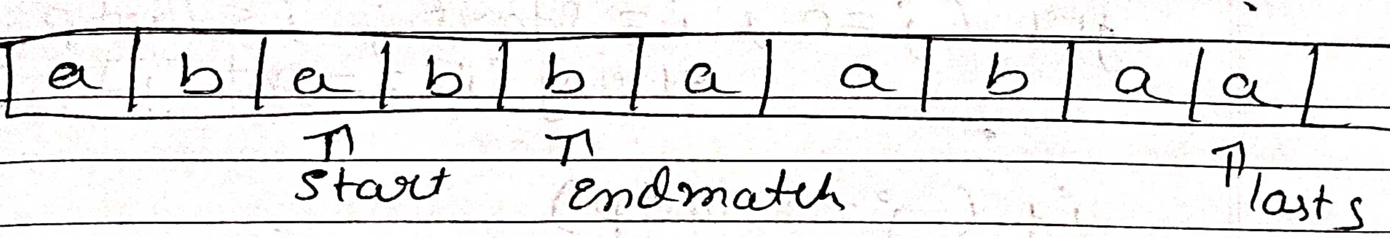
String



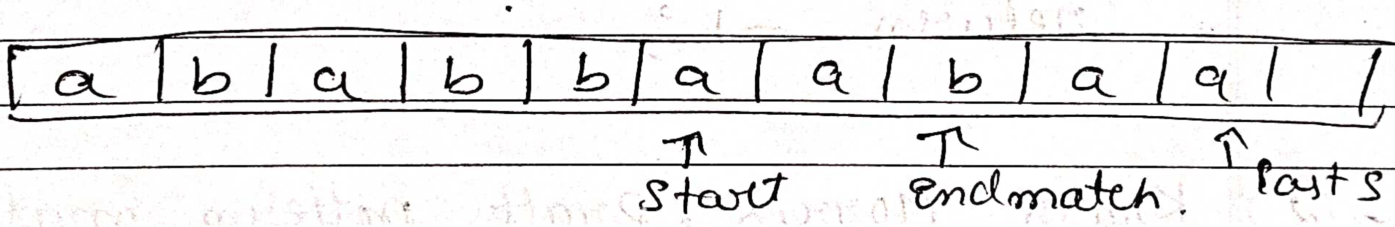
b ≠ a
so increment start.



b = b next increment Start
a = a next increment start.
a ≠ b so reset Start &
End match.



⋮



match
so return Start

```
int nfind (char *string , char *pat)
{ /* match the last character of pattern
   first & then match from the
      beginning */

     int  i, j, start = 0;
     int lasts = strlen(string) - 1;
     int lastp = strlen(pat) - 1;
     int endmatch = lastp;

  for (i=0; endmatch <= lasts ; endmatch++
                  start ++)
  {
        if (string [endmatch] == pat [lastp])
     for (j=0 ; i = start ; j < lastp &&
                  string [i] == pat [j] ; i++, j++ )
          ;
        if ( j == lastp)
         return  start ;
  }
  return  -1;
```

(2) Knuth, Morris, Pratt pattern matching

Compares characters by character from left to right. But whenever the mismatch occurs it uses prefix table to step characters comparison while matching.

Prefix table is also called Longest Proper prefix which is also suffix (LPS)

Create LPS table

① Define a one dimensional array with the size equal to the length of pattern (LPS [size])

② Define variable i & j, set i = 0, j = 1 and LPS[0] = 0.

③ Compare the characters at pattern[i] and pattern [j]

④ If both are matched then set LPS[j]=i+ and increment both i & j value by one then go to Step ③

⑤ If both are not matched then check the value of variable i. If it is '0' then set LSP[j] = 0 & increment j. if not zero '0' then set i = LPS[i-1 then go to Step ③.

⑥ Repeat all steps until all values of LSP[] are filled.

Ex- Pattern =

a	b	c	d	a	b	d
0	1	2	3	4	5	6

① => length = 7

LPS =

[0]	[1]	[2]	[3]	[4]	[5]	[6]

② => i = 0 , j = 1 , LPS(0) = 0

	0	1	2	3	4	5	6
LPS =	0						
	i	j					

③ => Pattern [$\overset{0}{i}$] Pattern [$\overset{1}{j}$]

a ≠ b

go to step ⑤ & check LPS[$\underset{0}{i}$] = 0

so LSP[j] = 0

LSP[1] = 0 & j++ ∴ j = 2

	0	1	2	3	4	5	6
∴ LPS =	0	0					
	i		j				

Again check Pattern [0] Pattern [2]

a ≠ c

goto Step ⑤ & LPS [0] = 0

so LSP[j] = 0

LSP[2] = 0 & j++ => j = 3

	0	1	2	3	4	5	6
∴ LPS =	0	0	0				
	i			j			

again check Pattern [0] Pattern [3]

a ≠ d

goto Step ⑤ & LPS[0] = 0

so LSP[j] = 0

LSP[3] = 0 & j++ => j = 4

	0	1	2	3	4	5	6
∴ LPS =	0	0	0	0			
	i				j		

again Pattern [0] Pattern [4]
a = a

goto step (4)
LPS [j] = i+1 = 0+1 = 1

i++ , j++
= 1 5

LPS =

0	1	2	3	4	5	6
0	0	0	0	1		
	i				j	

again pattern [1] pattern [5]
b = b

goto step (4)

LPS [j] = LPS [5] = i+1 = 1+1 = 2

i++ , j++
2 6

LPS =

0	1	2	3	4	5	6
0	0	0	0	1	2	
		i				j

again Pattern [2] Pattern [6]
c ≠ d

goto step (5) check LPS [$\overset{2}{i}$] = 0
LPS [j] = 0, LPS [6] = 0

j++ End

∴ LPS =

0	0	0	0	1	2	0

KMP algo

Text : abcabcdababcdabcdabde
Pattern: abcdabd

① create LPS for pattern

	0	1	2	3	4	5	6
∴ LPS =	0	0	0	0	1	2	0

② Start comparing 1st character of pattern with first character of text.

Text :	a	b	c	a	b	c	d	a	b	a	---
	↕	↕	↕	↕							
pattern	a	b	c	d	a	b	d				
j	0	1	2								

a=a so compare next character i++, j++
b=b so compare next character ..
c=c so compare next character ..
a≠d mismatch
so decrement j i.e. j=2
check LPS table for [2]
LPS[2]=0 so start comparing
from 0 pattern[0] i.e. j=0

	i			i											
Text =	a	b	c	a	b	c	d	a	b	a	b	c	d	a	--
				↕	↕	↕	↕	↕	↕	↕					
pattern =				a	b	c	d	a	b	d					
				j 0	1	2	3	4	5	6					

a=a next character i++, j++
b=b " "
c=c " "

d = d next char i++, j++

a = a " "

b = b " "

a ≠ d mismatch

so j--, 6-- = 5

check LPS[5] => 2

∴ Reset j = 2; & continue.

Patt[2] compared with t[9] i.e c ≠ a so

decrement j i.e j-- => j=1 look LPS[1] = 0

so start with j=0

									i											
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
a	b	c	a	b	c	d	a	b	a	b	c	d	a	b	c	d	a	b	d	e

a	b	c	d	a	b	d
0	1	2	3	4	5	6
j						

a = a

b = b

c = c ∴ i++, j++

d = d ∴ i++, j++

a = a ∴ i++, j++

b = b ∴ i++, j++

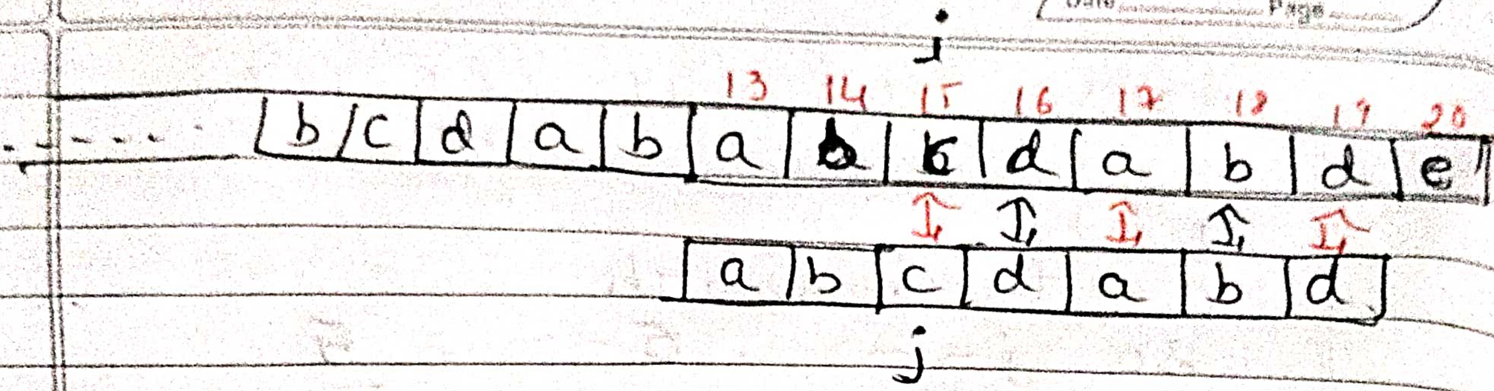
c ≠ d mismatch

so j--, 6--, 5

check LPS[5] => 2

∴ Reset j = 2 & continue.

Now i=15 j=2



$c = c$ ∴ $i{+}{+}, j{+}{+}$

$d = d$ ∴ $i{+}{+}, j{+}{+}$

$a = a$ ∴ $i{+}{+}, j{+}{+}$

$b = b$ ∴ $i{+}{+}, j{+}{+}$

$d = d$ ∴ $i{+}{+}, j{+}{+}$

∴ Pattern found at Position 13

we found position by $(i - \text{length}(\text{Pattern})$

$\Rightarrow (19 - 7) + 1$

$= 12 + 1$

$= 3$

Stacks

A Stack is an ordered list in which insertions (push) and deletion (PoP) are made at one end called the top.

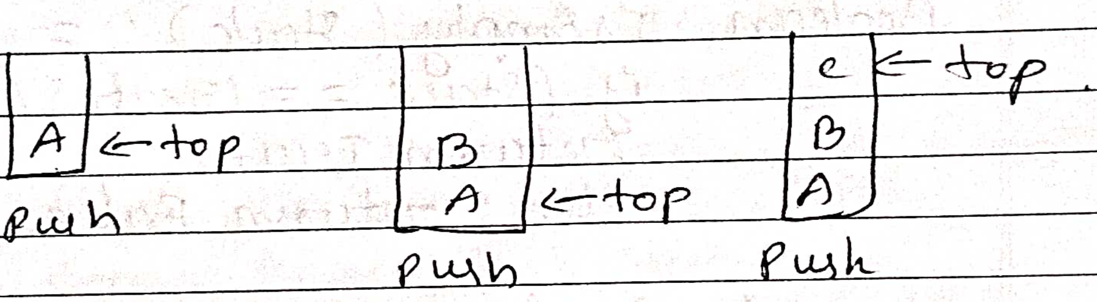
Stack $S = (a_0, a_1, \text{----} a_{n-1})$

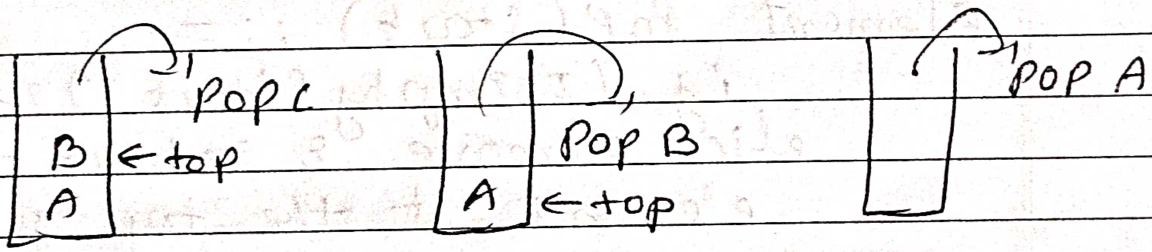
$a_0 \Rightarrow$ is bottom element

$a_{n-1} \Rightarrow$ is top element.

Stack is also known as ~~F~~ Last-in-First-out (LIFO) list.

Ex:-





ADT Stack is

object : a finite ordered list with zero or more elements.

functions :-

parameter :- Stack ∈ stack , item ∈ element

maxStackSize ∈ +ve integer

```
Stack CreateS(maxStackSize) ::=
        create an empty stack whose
        maximum size is maxStackSize

Boolean IsFull(Stack, maxStackSize) ::=
        if (no of elements in Stack == maxStackSize)
          return TRUE
        else return False.

Stack Push(Stack, item) ::=
        if (IsFull(Stack)) StackFull
    else insert item into top of stack
        and return.

Boolean IsEmpty(Stack) ::=
        if (Stack == CreateS(maxStackSize))
        return True
        else return False.

Element Pop(Stack) ::=
        if (IsEmpty(Stack)) return
    else remove & return the
        element at the top of the Stack.

End Stack.
```

To create Stack we have to define structure.

↓

```
#define MAX_STACK_SIZE 100
typedef struct
{
    int key;
} element;

element Stack [MAX_STACK_SIZE];
int top = -1;

Boolean IsEmpty (Stack) ::= top<0;

Boolean IsFull (Stack) ::= top>=MAX_
                           STACK_SIZE-1;

void Push (element item)
{
    if (top>=MAX_STACK_SIZE-1)
        StackFull ();
    Stack [++top] = item;
}

element POP()
{
    if (top==-1)
    return StackEmpty();
    return Stack [top--];
}

void StackFull()
{
fPrintf (Stderr, "Stack is full");
    Exit (EXIT_FAILURE);
}
```

Stack Using Dynamic arrays

```
Stack CreateS() ::= typedef struct
{
        int key;
 } element;
element  *stack;
MALLOC(Stak, sizeof (*stack));
 int capacity = 1;
 int  top = -1;

 Boolean IsEmpty(Stack) ::= top<0;

 Boolean IsFull(Stack) ::= top>=capacity-1;

void Push(element item)
{
     if (top >= capacity-1)
        stackFull();
     stack[++top] = item;
}
```

The new code for stackFULL attempts to increase the capacity of the array stack so that we can add an additional element to the stack.

In array doubling, we double array capacity whenever it becomes necessary to increase the capacity of the array.

code for Stackfull when doubling is used.

```
void StackFull ( )
{
   REALLOC ( Stack, 2 * Capacity *
                    sizeof ( * Stack ) )
   capacity * = 2;
}.
```

Application of Stack

① Conversion of infix to postfix Expression

② Conversion of infix to prefix Expression.

③ evaluate postfix Expression.

$\underbrace{a \overset{\text{operator}}{+} b}_{\text{operands}}$ = Infix

$\underbrace{a\ b}_{\text{operands}}\ + \rightarrow \text{operator}$ = Postfix

$\underset{\text{operator}}{+}\ ab$ — operands = Prefix

we convert expressions using preferences.

① parenthesis ( ) — 1st preference
② Exponent ^ or $ → Right → left
③ Multiplication & division → Left → Right
④ addition & subtraction → L → R

Precedence value of Symbols in i/p & Stack

Symbols	I/P precedence (G)	stack precedence (P)	
+, −	1	2	L−R
*, /	3	4	L−R
$, ^	6	5	R−L
operands	7	8	L−R
(	9	0	L−R
)	0	—	
#	—	−1	

Conversion from infix to Postfix

① Scan every i/p symbol from left to right

② when operator comes push into the stack increment top value.

③ If next symbol is operand directly push to output → Postfix

④ If next symbol is operator check & compare the precedence of operator with top of stack operator if greater then push to Stack. & increment top.

⑤ If Symbol precedence is less than precedence of top of the stack then pop the top value & push to output i.e postfix. & push Symbol to stack.

⑥ If Symbol is equal to top of the stack i.e precedence then pop top of the stack & push to Postfix and push Symbol to stack.

⑦ when closing braces ')' appears pop all the contents in Stack till open braces '(' & push to Postfix.

eg. a+b*c

Symbol	Stack	Top	Postfix (output)
a		-1	a
+	+	0	a
b	+	0	ab
*	$\overset{2}{+}\ \overset{>4}{*}$	1	ab
c	+ *	1	abc
end		-1	abc*+

~~order~~

① a is operand so directly push to postfix
② + is operator so push to Stack
③ b is operand so directly push to postfix

④ * is operator so check

Precedence (*) with Precedence (+)

4 > 2

So push * to stack

⑤ c is operand so directly push to postfix.

Now we got End so pop the stack values & push to Postfix

∴ abc*+

② a*(b+c)*d

Symbol	Stack	top	Postfix
a		-1	a
*	*	0	a
(	* (	1	a
b	* (	1	ab
+	* (+	2	ab
c	* (+	2	abc
)	*	0	abc+
*	*	0	abc+*
d	*	0	abc+*d
end		-1	abc+*d*

(1) a ⇒ operand ⇒ Push to Postfix

(2) * ⇒ operator ⇒ push to stack

(3) (⇒ For open braces just push to stack & all following operators to be pushed to stack until you get closing braces ')'.
so push '(' & top +*

(4) b ⇒ operand ⇒ push to Postfix

(5) + ⇒ operator but still we have to push till we get ')'.

(6) c = operand ⇒ Push to postfix

(7)) = so now pop stack till we get opening brace '('.

(8) * = operator ⇒ so check
prece(*) = prec(*)
so pop & push

(9) d = operand ⇒ push to Postfix

(10) end so pop stack.

⇒ abc+*d*

A+(B*C-(D/E∧F)*G)*H

Symbol	Stack	top	Postfix
A		-1	A
+	+	0	
(	+(	1	
B	+(	1	AB
*	+(*	2	AB
C	+(*	2	ABC
-	+(-	2	ABC*
(	+(-(	3	
D	+(-(	3	ABC*D
/	+(-(/	4	ABC*D
E	+(-(/	4	ABC*DE
∧	+(-(/∧	5	ABC*DE
F	+(-(/∧	5	ABC*DEF
)	+(-	2	ABC*DEF∧/
*	+(-*	3	ABC*DEF∧/
G	+(-*	3	ABC*DEF∧/G
)	+	0	ABC*DEF∧/G*-
*	+ *	1	ABC*DEF∧/G*-
H	+ *	1	ABC*DEF∧/G*-H
End	sopop		

$\boxed{\text{ABC*DEF∧/G*-H*+}}$

Evaluation of Postfix Expression

① Scan the given postfix expression from left to right.

② If Symbol is operand or no push to Stack.

③ If Symbol is operator Pop two operands from stack

1^{st} poped no = op1
2^{nd} poped no = op2

④ Perform op2 operand op1 & save the result

⑤ Push the result to stack then continue with Symbols.

E.g. 2 3 1 * + 9 −

Symbol	Stack	op1	op2	Result
2	2			
3	23			
1	231			
*	23	1	3	3*1=3
+	5	3	2	2+3=5
9	59			
−	−4	9	5	5−9=−4

∴ −4 is answer

(2) 6 2 / 3 − 4 2 * +

Symbol	Stack	op1	op2	R= op2 (op) op
6	6			
2	6 2			
/	3	2	6	6 / 2 = 3
3	3 3			
−	0	3	3	3 − 3 = 0
4	0 4			
2	0 4 2			
*	0 8	2	4	4 * 2 = 8
+		8	0	0 + 8 = 8

Result = 8

(3) 6 5 1 − 4 * 2 3 ^ / +

Symbol	Stack	op1	op2	R=op2 □ op1
6	6			
5	6 5			
1	6 5 1			
−	6 4	1	5	5−1=4
4	6 4 4			
*	6 16	4	4	4*4=16
2	6 16 2			
3	6 16 2 3			
^	6 16 8	3	2	2^3 = 8
/	6 2	8	16	16/ 8 = 2
+		2	6	6+2 = 8

||Jai Sri Gurudev ||
BGSKH Education Trust (R.) – A unit of Sri Adichunchanagiri Shikshana Trust(R.)
BGS College of Engineering and Technology (BGSCET)
Mahalakshmipuram, West of Chord Road, Bengaluru-560086
(Approved by AICTE, New Delhi and Affiliated to VTU, Belagavi)

Question Bank

Subject: Data Structure and Applications **Class:** AIDS

Subject code: BCS304 **Faculty:** Mrs. Jyothi R

Course Outcomes

CO 1. Explain different data structures and their applications.
CO 2. Apply Arrays, Stacks and Queue data structures to solve the given problems.
CO 3. Use the concept of linked list in problem solving.
CO 4. Develop solutions using trees and graphs to model the real-world problem.
CO 5. Explain the advanced Data Structures concepts such as Hashing Techniques and Optimal Binary Search Trees

MODULE 1		
Sl. No.	**Questions**	**CO**
Topic:	**Introduction**	
1.	Define Data structures. Classify the data structures with examples. And explain the operations of Data structures	CO1
Topic:	**Pointers**	
2	Define Pointers and Pointer Variable. How to declare and initialize pointers in C, explain with examples,	CO1
3	Can we have multiple pointers to a Variable? How pointers can be dangerous	CO1
Topic:	**Dynamic Memory Allocation**	
4	Explain four dynamic memory allocation functions with Syntax and examples	CO1
Topic:	**Arrays**	
5	What is array? Give Abstract data type (ADT) for array. How array are declarations and implemented in C	CO2
6	Write a basic C program to demonstrate the basic operations of Array	CO2
7	Write a C program to read and display two dimensional array by using Dynamic memory allocation	CO2
Topic:	**Structures and Unions**	
8	What is structure? Explain various operations that can be performed on structures.	CO1
9	How does a structure differ from a union? Explain with example.	CO1
10	How to declare user defined datatype using structures with an example program.	CO1
11	What is self-referential structures? Explain with example.	CO1
Topic:	**Polynomials**	

12	What is a polynomial? Apply ADT to represent two polynomials and write a function to add the polynomials.	CO1
13	Write addition of polynomial using arrays of structures. $A(x)= 3x^{23} + 3x^4 + 4x^2 + 15$. $B(x)= x^5 + 20x^3+ 2$. Solve by explaining all 3 cases.	CO1
Topic:	**Sparse Matrix**	
14	Define sparse matrix. Explain its representation with its ADT.	CO1
15	Write a function to read a sparse matrix and transpose a sparse matrix. Explain with example.	CO1
Topic:	**Strings**	
16	Define strings. Explain any four string handling functions supported by ‘C’ with syntax and example	CO1
17	Write the ADT of strings	CO1
18	Define pattern Matching. Write the Knuth Morris Pratt pattern matching algorithm and apply same pattern ‘abcdabcy’ in the text: ‘abcxabcdabxabcdabcdabcy’	CO1
19	Write a function to insert a string into another string at position ‘i’ and explain with example.	CO1
20	Explain nfind string matching algorithm and find a pattern “aab” in the string “ababbbaabaa”.	CO1
Topic:	**Stacks**	
21	Give Abstract datatype(ADT) of Stack	CO2
22	Define Stack. Explain the different operations that can be performed on stack with suitable ‘C’ functions and explain	CO2
23	Convert the following infix expression into postfix expression using stack 1. A + (B * C - (D / E ^ F) * G) * H 2. ((H * ((((A + ((B + C) * D)) * F) * G) * E)) + J) 3. A * (B + D) / E – F * (G + H / K)	CO2

24	Write an algorithm to evaluate a postfix expression. Trace the algorithm for the expression showing the stack contents 6 5 1 – 4 * 2 3 ^ / +. 5 4 6 + * 4 9 3 / + *	CO2

Module - 2

Queues

A Queue is an ordered list in which insertions (insert) and deletions (delete) take place at different Ends.

The End at which new elements are added is called the rear (r)

The End from which old elements are deleted is called the front. (f)

Queue is also called FIFO (First in First out)

```
add A      add B      add C.      add D
  A         A B        A B C       A B C D
  ↑         ↑ ↑        ↑   ↑       ↑     ↑
 r,f        f r        f   r       f     r


delete A     delete B
 B C D        C D         . . . .
 ↑   ↑        ↑ ↑
 f   r        f r
```

ADT Queue is

objects: a finite ordered list with zero
or more elements.

functions :-
parameters queue ∈ Queue , item ∈ elemen
maxQueueSize ∈ +ve integer

Queue CreateQ (maxQueueSize) ::=
create an empty queue whose
maximum size is maxQueueSize

Boolean IsFull (queue, maxQueueSize) ::=
if (no of elements in queue = =
maxQueueSize)
return TRUE
else return FALSE

Queue AddQ (queue, item) ::=
if (IsFull (queue)) queue Full
else insert item at rear of queue &
return queue.

Boolean IsEmptyQ (queue) ::=
if (queue == CreateQ (maxQueueSize
return TRUE
else return FALSE

Element DeleteQ (queue) ::=
if (IsEmptyQ (queue)) return
else remove and return
the item at front of queue.

End Queue

```
Queue createQ (maxQueueSize) ::=

#define MAX_QUEUE_SIZE 100
 typedef struct
 {
     int key;
 } element;

element queue [MAX_QUEUE_SIZE];
 int rear = -1;
 int front = -1;

Boolean IsEmptyQ(queue) ::= front == rear

Boolean IsFullQ(queue) ::=
                   rear == MAX_QUEUE_SIZE-1

 void addq (element item)
{
    if (rear == MAX_QUEUE_SIZE-1)
     queueFull();
    queue [++rear] = item;
}


 element deleteq()
{
    if (front == rear)
    return queueEmpty();
    return queue[++front];
}
```

Ex job scheduling :- jobs are Entered or inserted & deleted or served in sequential queue.

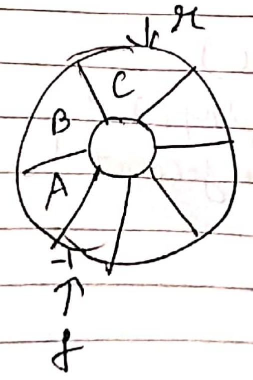
front	rear	Q[0]	Q[1]	Q[2]	Q[3]	Comments
-1	-1					Q is Em
-1	0	J1				J1 added
-1	1	J1	J2			J2 added
-1	2	J1	J2	J3		J3 add
0	2	~~J1~~	J2	J3		J1 deleted
1	2		~~J2~~	J3		J2 dele
2	2					J3 delet

so now front = rear
2 = 2 so Queue Empty

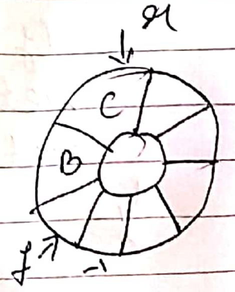
Circular Queue

If we join the End of queue with front it will form circular queue.

The Position next to position MAX-QUEUE-SI -1 is 0 &
The position that precedes 0 is MAX-QUEUE-SIZE -1.



if we delete



```
void addcq (element item)
{
    rear = (rear +1) % MAX_QUEUE_SIZE ;
    if (front == rear)
        queueFull () ;
    queue [rear] = item ;
}

element deletecq ()
{
    element item ;
    if ( front == rear )
        return queueEmpty () ;
    front = (front +1) % MAX_QUEUE_SIZE ;
    return queue[front] ;
}
```

C Program for Queue (linear)

```
#include < stdio.h>
#define Q_SIZE 5
int ch, r, f, i, item, q[Q_SIZE];

void insertrear ();
void deletefront ();
void display ();

void main ()
{
    f = r = -1
    for (;;)
{
    Printf (" 1: Insert \n 2: Delete \n 3: Display
            \n 4: Exit \n");
Printf ("enter your choice \n");
printf ("%d", &ch);
 switch (ch)
{
 case 1 : insertrear();
          break;
 case 2 : delete front();
          break;
 case 3 : diplay();
          break;
 case 4 : Exit (0);
          break;
 default : printf ("Invalid choice")
}
}
}

void insertrear()
{
  int item, r = -1;
  if (r == Q_SIZE - 1)
  {
   Printf ("Queue overflow \n");
   return;
  }
   printf ("Enter the item \n");
   scanf ("%d", &item);
        q[++r] = item;
}
```

```
void deletefront ()
{
        f = -1 ;
   if    (r == f)
 {
     printf (" Queue is empty \n");
       return;
 }
  printf (" Deleted element is = %d",
                                [++q]) ;
}

 void display()
{
    int i ;
    if ( f==r )
  {
     printf ( "Queue is empty \n");
       return;
  }
  printf (" Queue contents are \n");
     for ( i=f ; i<=r ; i++)
      printf ("%d \n", q[i]);
}
```

Circular queues using Dynamic arrays

Before inserting, check for sufficient space in the queue. But once the queue is Full, we can increase the size of queue using realloc().

① create() : Let assume Q-SIZE = 1

```
int QSIZE=1;
int *q, r=f=-1, count =0;
```

② InsertQ() :

```
void insertQ()
{
    if (count == QSIZE)
    {
    printf("Q Full - Increase Size by1");
    QSIZE ++;
    q = (int *) realloc (QSIZE, sizeof(int));
    }
  if (f == r)
  {
   for (i = QSIZE - 2; i>f ; i--)
   {
       q[i+1] = q[i];
   }
   ++f;
  }
   r = (r+1) % QSIZE;
   q[r] = item;
```

```
    count ++;
}
```

delete() & diplay are same

Implement cirular queue using array in C

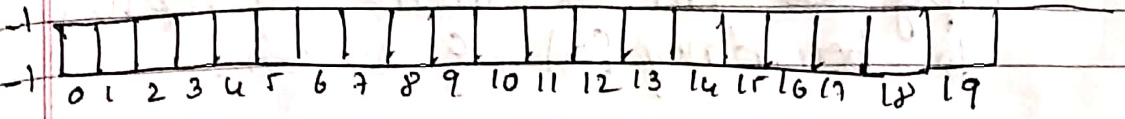
```
#define SIZE 5
int q[SIZE], f = r = -1, count = 0;

void insertcq()
{
    int item;
    if (count == size)
    {
        Printf ("Q is full");
        return;
    }
    r = (r+1) % SIZE;
    Printf ("Enter the item \n");
    Scanf ("%d", &item);

      q[r] = item;
      count ++;
}

void deletecQ()
{
    if (count ==0)
    {
        Printf ("Q is Empty \n");
        return;
    }
```

```
        f = (f+1) % SIZE ;
printf (" Deleted element = %d ", q [f]).;
        count --;
}

      void display ()
{
      int i, j;
      if (count == 0)
    {
       printf ("Q is empty");
       return;
     }

      j = f;

     printf ("contents of queue are \n");
     for (i = 1; i <= count ; i++)
      {
         [illegible] j = (j+1) % SIZE;
          printf ("%d \n", q [j]);
      }
}
```

multiple Stacks and Queues

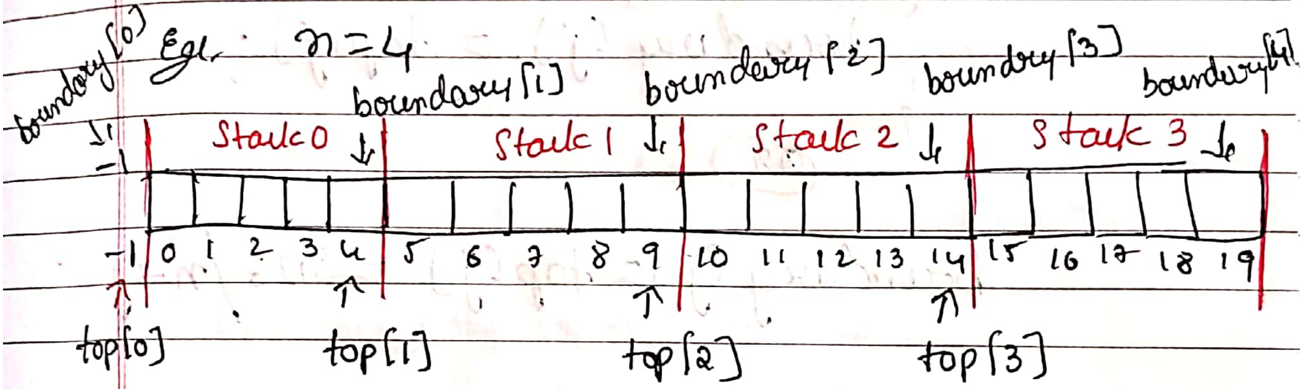
Implement multiple stacks using arrays.

```
#define SIZE 20
int S[SIZE];
```



Divide array into n stack.

Eg. n=4



* to Find the initial value of top of Each Stack i.e top[0], top[1], top[2] & top[3] we use expression.

$$top[j] = SIZE/n * j - 1$$

where j = 0 to n.

i.e j=0 top[0] = 20/4 * 0 - 1 = -1
j=1 top[1] = 20/4 * 1 - 1 = 4
j=2 top[2] = 20/4 * 2 - 1 = 9
j=3 top[3] = 20/4 * 3 - 1 = 14

∴ we can write code as

```
for( j=0 ; j<=n ; j++)
{
    top[j] = SIZE/n * j - 1;
}
```

* To find the boundary of Each stack copy initial value of top[0] to boundary [0] & so on.

i.e

```
for (j=0 ; j<=n ; j++)
{
    boundary [j] = top[j];
}
```

OR

```
boundary [j] = top[j] = SIZE/n*j-1;
```

write C prog to demonstrate multiple stacks and Queues

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 20
 int S [SIZE];
#define MAX_STACKS 10
 int boundary [MAX_STACKS];
 int top [MAX_STACKS];
 int n, i, j, item, ch;

 Void Push()
{
   if (top[i]== boundary [i+1])
  {
```

```
printf ("stack is full in %d", i);
return;
}
   s[++top[i]] = item;
}

void pop()
{
  if (top[i]==boundary[i])
  {
   printf ("Stack is empty", i);
   return;
  }
   printf ("Item deleted = %d", s[top[i]--]);
}

void display()
{
  if (top[i] == boundary[i])
  {
   printf (" stack %d is empty", i);
   return;
  }
   for (j= boundary[i]+1 ; j<= top[i];
                                 j++)
   printf ("%d\n", s[j]);
}

void main()
{
  printf ("enter no of queues \n");
  scanf ("%d", &n);
  for (j=0; j<=n; j++)
```

```
boundary[j] = top[j] = SIZE/n * j - 1 ;
    for(;;)
{
  Printf("Stack number: ");
    for(j=0; j<n; j++)
      printf("%d", j);
  printf("Enter Stack no : To perform operati ");
  scanf("%d",&i);

  Print(" 1:Push\n 2:POP\n 3:diplay\n
                4:Exit \n");
  printf("enter your choice\n");
    scanf("%d", &ch);
    switch (ch)
   {
      case 1: printf("enter item\n");
              scanf("%d", &item);
              Push();
              break;
      case 2: POP();
              break;
      case 3: diplay();
              break;
      default: exit(0);
   }
  }
}
```

Linked list

Disadvantages of array

→ Array items are stored continously next to each other

→ Insertion & deletion is difficult job.

To overcome we use dynamic memory allocation using linked list.

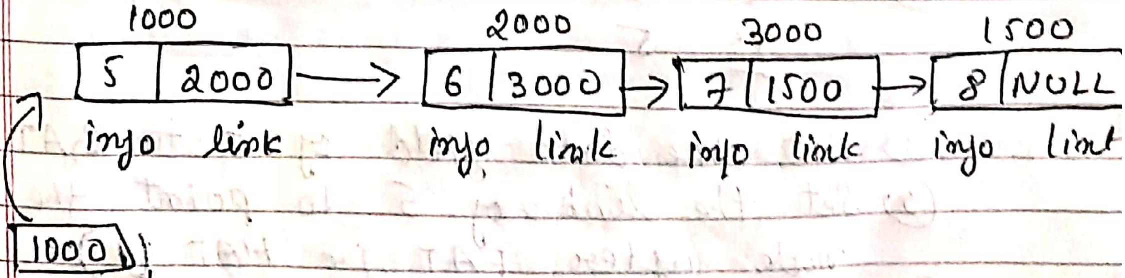
Linked list is a ~~non~~- linear, collection of data elements whose order is not given by their physical placement in memory.

It is collection of nodes

Each node has 2 fields

① info field : To store data or information to be manipulated.

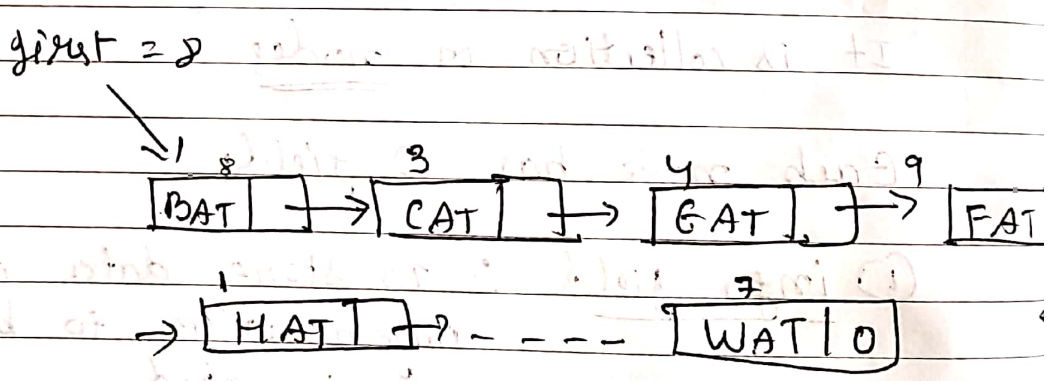
② next (link) field :- contains the address of the next node



first/head
contains address of Ist node

	data	link
1	HAT	15
2		
3	CAT	4
4	EAT	9
5		
6		
7	WAT	0
8	BAT	3
9	FAT	1
10		
11	VAT	7

Let first = 8



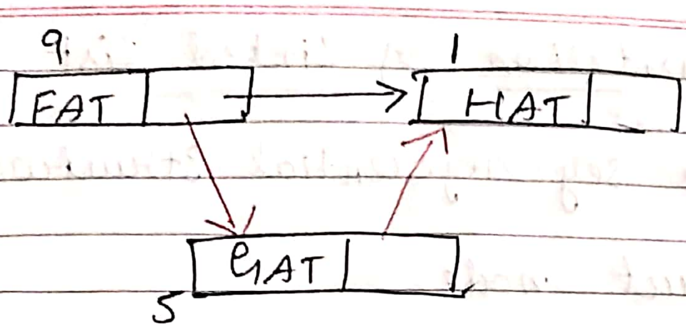
Now How to add GAT Node

① Let a node that is currently unused i.e 5

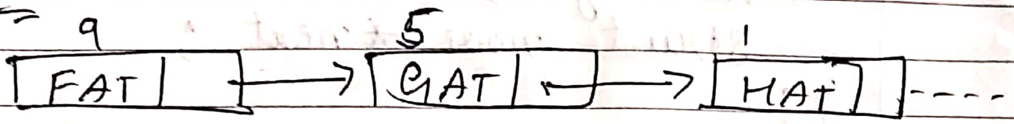
② Set the data field of 5 to GAT

③ Set the link of 5 to point the node after FAT i.e HAT (1)

④ Set the link field of ~~Nod~~ node FAT to 5 i.e

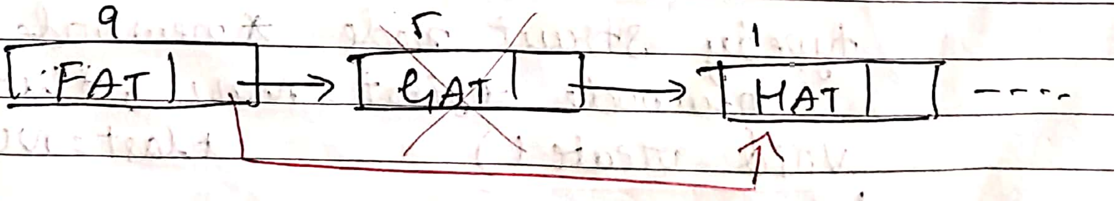


i.e



1	HAT	15
2		
3	CAT	4
4	EAT	9
5	GAT	1
6		
7	WAT	0
8	BAT	3
9	FAT	5
10		
11	VAT	7
	⋮	⋮

Delete GAT node



link [9] = 1

Representation of Linked List.

we use self-referential structures

```
struct node
{
    int info;
    struct node *next;
};
```

Info :- field is integer.

next field is a pointer type &
pointer should be a structure
type. Because next fied holds
the address of next node.

Implementation

```
typedef struct ....
 {
     int info;
     struct node *next;
 }node;

typedef struct node    newnode;
  newnode *first = NULL, *cur = NUL
void create()               *last = NULL;
{
    new node  ;  *temp;

    temp = (new node*)malloc(sizeof(newno));
    printf ("enter the information to
     be stored in");
  Scanf("%d", &temp->info);
     temp->next = NULL;
}

void insertfront()
{
      if (first == NULL)   //list is empty
    {
           create();
          first = temp;
    }
    else
    {
          create();
         temp->next = first;
           first = temp;
      }
}

void insertend()                rear End
{
      if (first==NULL)
     {
          create();
           first = temp; last = first;
      }
      else
      {       create();
           cur = first;
          while (cur->next != NULL)
            cur = cur->next;
```

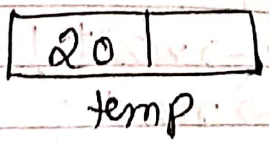
```
        cur-> next = temp;
          last = temp;
}
}
void display()
{
   struct node *temp1;
      temp1 = first;

   if (temp1 == NULL)
   {
      Printf (" List is Empty\n");
      return;
   }
   printf (" The contents of list are\n"
   while (temp1 != NULL)
   {
      printf ("%d\n", temp->info);
         temp1 = temp1->next;
   }
}
```

Eg:

insert from front End

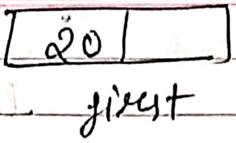
insert 20 → first = NULL so
create() →

temp→info = 20
temp → next = NULL

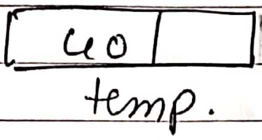


assign first = temp

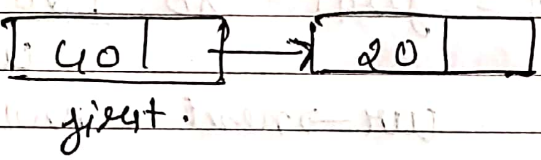
so



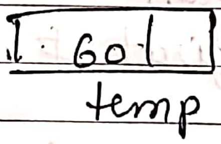
Next insert 40 ⇒ first ≠ NULL
so go to else
create() ⇒ temp → info = 40
temp → next = NULL

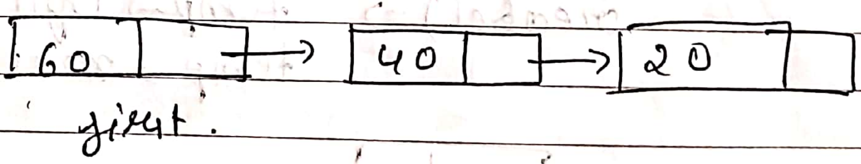


temp → next = first

∴ 

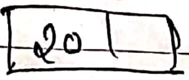
insert 60 ⇒ first ≠ NULL
so go to else
create() ⇒ temp → info = 60
temp → next = NULL



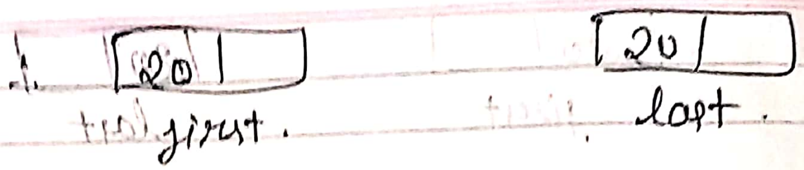
∴ 

Ex- Insert from Rear End

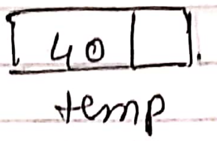
insert 20 ⇒ first = NULL so
create ⇒
temp → info = 20
temp → next = NULL



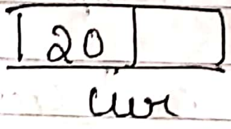
assign first = temp



insert 40 ⇒ first ≠ NULL.
so go to else
create ⇒ temp → info = 40
temp → next = NULL



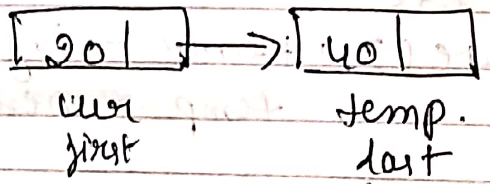
cur = first so



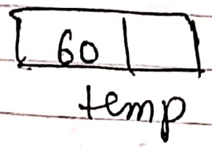
cur → next = NULL

so

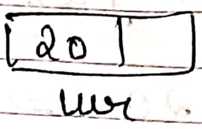
cur → next = temp.



insert 60 ⇒ first ≠ NULL
so go to else
create() ⇒ temp → info = 60
temp → next = NULL

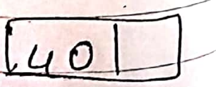


cur = first so



cur → next ≠ NULL so

cur = cur → next =

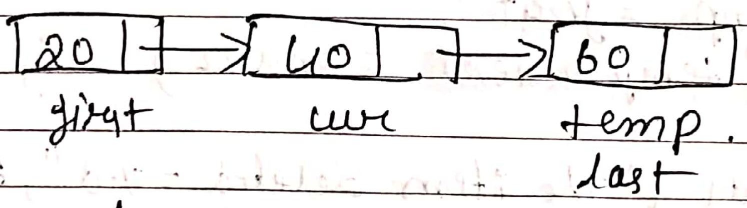


so now [40 |]
cur

cur → next = NULL

so

cur → next = temp



insert 80 ⇒ first ≠ NULL
so goto else
create() ⇒ temp → info = 80
temp → next = NULL

[80 |]
temp.

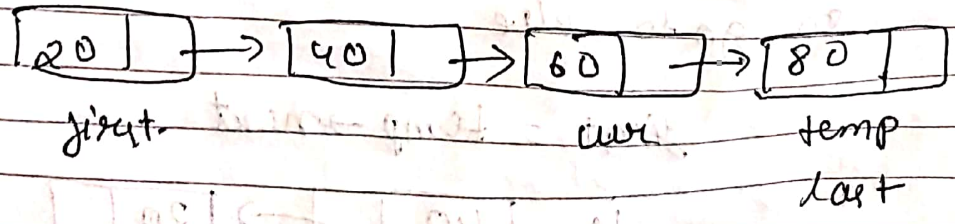
cur = first ⇒ [20 |]
cur

cur → next ≠ NULL so
Now cur = cur → next ⇒ [40 |]
cur

cur → next ≠ NULL so
Now cur = cur → next ⇒ [60 |]
cur

cur → next = NULL
so

cur → next = temp

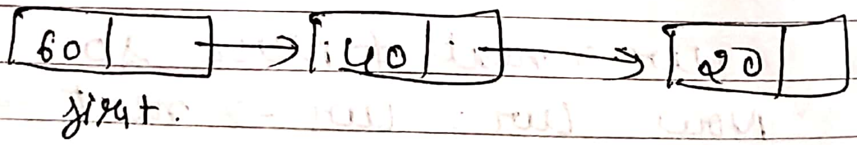


(from)

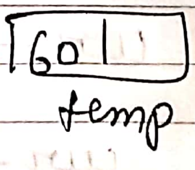
Delete a node at the begining (front)

```
void deletefront ( )
{
  struct node *temp;
    temp = first;
  if ( temp→next ==NULL) // if only on
                              node
  {
   printf ("The item deleted = %d", temp→in
     free (temp);
      first = last = NULL;
      return;
  }
  else
  {
      first = temp→next; //make 2nd node
                           as first
   printf ("The item deleted = %d", temp→i
      free (temp);
   }
}
```

Ex:- Delete



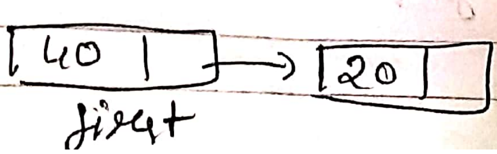
temp = first =>



tem→next ≠ NULL
so go to else

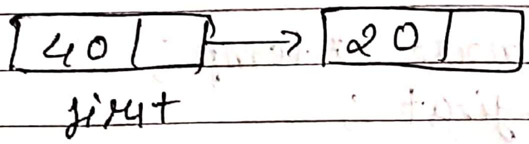
first = temp→next

ie

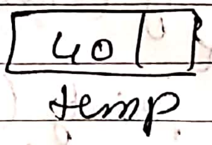


The item deleted = 60

* Delete

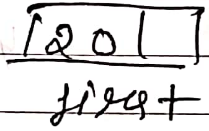


temp = first =>



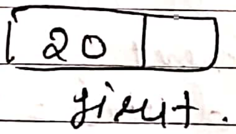
temp → next ≠ NULL
so goto else

first = temp → next =>

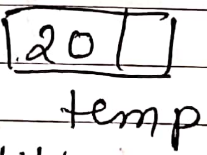


The item deleted = 40

* Delete



temp = first =>

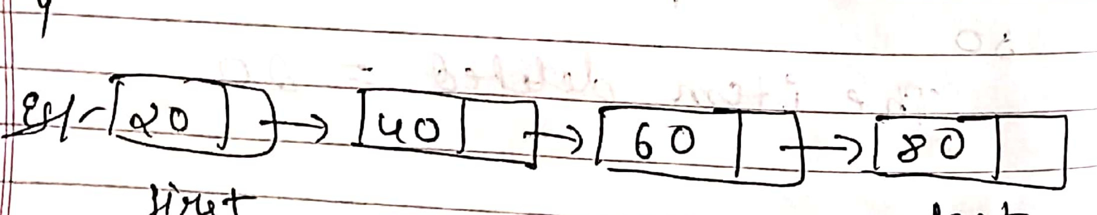


temp → next = = NULL
so
The item deleted = 20

Delete from Rear end

```
void deleterear ()
{
    struct node *temp;
    temp = first;
    if (temp->next == NULL) //only 1 nod
    {
    printf ("The item deleted is = %d", temp->info);
     free (temp);
     first = last = NULL;
      return;
    }

   else
   {
       while (temp->next != last) //search
                          for last but one nod
         temp = temp->next;
   printf ("The item deleted = %d", last->info);
        free (last).
        temp->next = NULL;
         last = temp;
   }
}
```

Ex:-



temp = first = [20]

temp -> next ≠ NULL so go to else

temp → next != last i.e [80|]

so temp = temp → next

i.e [40|]→ Before
temp.

temp → next != last

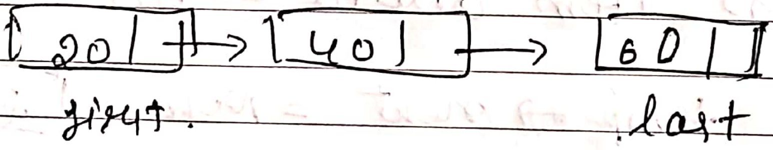
so temp = temp → next i.e [60|]→
temp

now temp → nere == last

so The item deleted = 80

temp → next = NULL i.e [60|0]

last = temp i.e



* **Delete**

temp = first = [20|]

temp → next ≠ NULL so
go to else

temp → next != last so

temp = [40|]

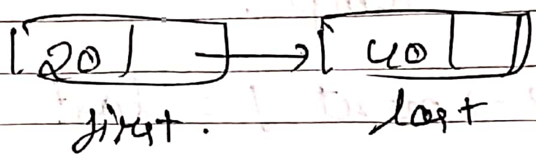
temp → next = last so

The item deleted = 60

temp → NULL ; = [40 / 10]

last = temp

i.e



Delete

temp = first ⇒ [20 |]

temp → next ≠ NULL , so go to else

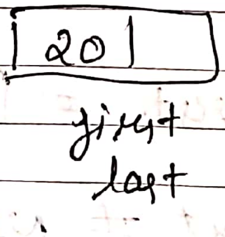
temp → next == last

so item deleted = 40

temp → next = NULL. [20 | ~~40~~]

last = temp.

i.e



Delete

temp = first [20 |]

temp → next == NULL

so

item deleted = 20

Stacks using Linked Lists

Stack is data structure where elements are inserted at one End and deleted from Same End (top).

Stack can be implemented using

insertfront () deletefront () display ()	or	insertEnd () deleteEnd () display ()

C Program to implement all the Stack operations using linked list

```
#include <stdio.h>
#include <stdlib.h>
 void Push();
 void POP();
 void display();

 struct node
{
   int info;
   struct node *next;
};
    struct node *head;

 void main()
{
   int ch=0;
   printf("** Stack operations using linked
          list **\n");
```

```
while(ch != 4)
{
Printf("Choose one from below\n");
printf(" 1: Push \n 2: POP \n 3: Display \n
          4: Exit \n");
printf(" Enter your choice \n");
 Scanf(" %d", &ch);

 Switch (ch)
 {
  case 1:  Push(); break;
  case 2:  POP (); break;
  case 3:  display(); break;
  case 4:  printf(" Exiting ");
              break;
  default: printf(" Enter valid choice");
 }
}
}

void push()
{
  int info;
struct node *temp = (struct node *)
          malloc(sizeof(struct node));
 if (temp == NULL)
 {
 Printf(" not able to push the element \n");
 }
 else
 {
  Printf(" Enter the element to push \n");
  Scanf(" %d", &info);
if (head == NULL)
{
    temp -> info = info ;
    temp -> next = NULL ;
    head = temp ;
}
else
{
    temp -> info = info ;
    temp -> next = head ;
    head = temp ;
}
printf ("Item/Element Pushed");
}
}

void POP()
{
    int item ;
    struct node *temp ;
    if (head == NULL)
    {
        Printf ("underflow");
    }
    else
    {
        item = head -> info ;
        temp = head ;
        head = head -> next ;
        printf (" The item deleted = %d", item);
        free (temp);
        printf (" item popped ");
    }
}
```

```
void diplay()
{
    int i;
    struct node *temp;
    temp = head;
    if (temp == NULL)
   {
     printf("Stack is Empty \n");
   }
   else
   {
    printf("printing Stack element \n");
    while (temp!=NULL)
    {
      printf("%d", temp->info);
      temp = temp->next;
    }
   }
}
```

Queues using linked list

Queue is data structure where insertion takes place at one End and deletion at other End, so Queue can be implemented using

insert front()
delete rear()
display()

or

insert rear()
delete front()
display()

C program to implement Queue

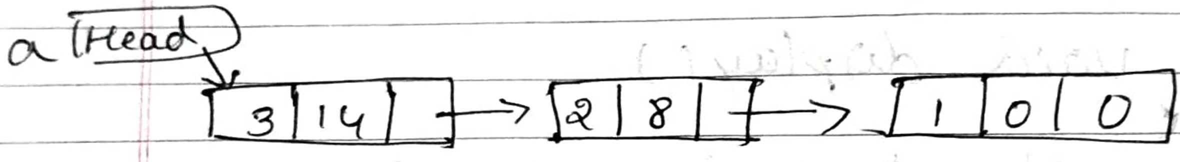
```
#include <stdio.h>
# include <stdlib.h>
struct node
{
   int info;
   struct node *next;
};
struct node *first, *cur, *last;
void insert();
void delete();
void display();

Void main()
{
// Same as Stack use insert, delet instead
}        of Push & POP

void insert()
{
   struct node *temp;
    int item;
   temp = (struct node *) malloc (sizeof (-
                                  - struct node));
   if (temp == NULL)
  {
     Printf ("overflow\n");
     return
  }
   else
  {
     Printf ("Enter the item to be inserted\n");
     Scanf ("%d", &item);
    temp -> info = item;
    if (first == NULL)
   {
        first = temp;
        last = temp;
        first -> next = NULL;
        last -> next = NULL;
   }
    else
    {
              cur = first;
        while (cur -> next != NULL)
          cur = cur -> next;
          cur -> next = temp;
              last = temp;
     }
  }
}

    void delete()
  {
     struct node *temp; int item = temp->info;
     if (first == NULL)
    {
       printf("underflow\n");
       return;
    }
     else
    {
         temp = first;
       if (temp -> next == NULL)
      {
        printf("The item deleted = %d", item);
         free(temp);
   first = last = NULL;
    return;
  }
  else
  {
   first = temp -> next;
   printf ("Item deleted = %d", temp->info)
    free(temp);
  }
 }
}
    void display()
   {
     struct node *temp;
     temp = first;
    if (first == NULL)
    {
     Printf (" Empty queue \n");
    }
    else
    {
    Printf (" Queue contents are \n");
     while (temp != NULL)
     {
      Printf ("%d\n", temp->info);
       temp = temp->next;
     }
    }
}
```

Polynomial Representation using Linked list

Node structure

co-efficient	Exponent	address of next node

Eg- $3x^{14} + 2x^{8} + 1$



```
struct node
{
    int coeff;
    int Exp;
    Struct node *next;
};

Struct node *polynomial;
```

Addition of two polynomial using linked list

```
#include <stdio.h>
#include <stdlib.h>

Struct node
{
    float coeff;
    int Expo;
    Struct node *link;
};
```

```
void main()
{
  struct node *a = NULL;
  struct node *b = NULL;
  printf("Enter the first polynomial \n");
    a = create (a);
  printf("enter the second polynomial \n");
    b = create (b);
  polyadd (a, b);
}

struct node *create (struct node *head)
{
  int n, i;
  float coeff;
   int expo;
  printf(" enter the no of terms");
   scanf ("%d", &n);
  for (i=0 ; i<n; i++)
  {
   printf(" Enter the co-eff for term %d", i+1);
  scanf ("%f", &coeff);
  printf ("enter the exponent for term %d", i+1);
   scanf ("%d", &expo);

    head = insert (head, coeff, expo);
  }
  return head;
}
```

```
struct node *insert (struct node * head,
                    float co, int Ex)
{
    struct node *temp;
    struct node *newP = malloc(sizeof(
                          struct node));
    newP -> coeff = co;
    newP -> expo = Ex;
    newP -> link = NULL;
//if no node (or) given exponent is greater than first one
  if((head ==NULL) || (Ex > head -> expo))
  {
     newp -> link = head;
      head = newp;
  }
    else
  {
      temp = head;
    while((temp -> link != NULL) &&
              (temp -> link -> expo >= Ex))

          temp = temp -> link;
         newp -> link = temp -> link;
          temp -> link = newp;
  }
    return head;
}


   void print (struct node *head)
 {
       If (head == NULL)
         printf ("no polynomial \n");
       else
       {
```

```
 struct node *temp = head;
  while (temp!= NULL)
 {
   printf("(%.1f x^%d)", temp->coeff,
                          temp->expo);
      temp = temp->link;
      if (temp!= NULL)
        printf ("+");
      else
        printf ("\n");
    }
 }
}

 void  Polyadd (struct node *a ,
                struct node *b )
 {
   struct node *ptr1 = a;
   struct node *ptr2 = b;
   struct node * c = NULL;
 while( ptr1!=NULL && ptr2!=NULL)
 {       // exponents are equal
   if ( ptr1->expo == ptr2->expo)
  {
     c = insert(c, ptr1->coeff + ptr2->coeff,
                   ptr1->expo);
     ptr1 = ptr1->link;
     ptr2 = ptr2->link;
   }
  else if (ptr1->expo > ptr2->expo)
  {
    c = insert(c, ptr1->coeff, ptr1->expo);
    ptr1 = ptr1->link;
  }
```

```
elseif (ptr1->expo < ptr2->expo)
{
   C = insert (c, ptr2->coeff, ptr2->expo);
    ptr2 = ptr2->link;
 }
}

  while (ptr1 != NULL)
  {
     C = insert (c, ptr1->coeff, ptr1->expo);
     ptr1 = ptr1->link;
  }

  while (ptr2 != NULL)
  {
     C = insert (c, ptr2->coeff, ptr2->expo);
      ptr2 = ptr2->link;
   }
   printf ("~~Added~~ Two Polynomials\n");
     print (a);
     print (b);
   printf ("Added or Result of Polynomial
                addition\n");
       print (c);
}
```

Adding two Polynomials represented as circular lists with header nodes

↓

lab program

||Jai Sri Gurudev ||
BGSKH Education Trust (R.) – A unit of Sri Adichunchanagiri Shikshana Trust(R.)
BGS College of Engineering and Technology (BGSCET)
Mahalakshmipuram, West of Chord Road, Bengaluru-560086
(Approved by AICTE, New Delhi and Affiliated to VTU, Belagavi)

Question Bank

Subject: Data Structure and Applications **Class:** AIDS

Subject code: BCS304 **Faculty:** Mrs. Jyothi R

Course Outcomes

CO 1. Explain different data structures and their applications.
CO 2. Apply Arrays, Stacks and Queue data structures to solve the given problems.
CO 3. Use the concept of linked list in problem solving.
CO 4. Develop solutions using trees and graphs to model the real-world problem.
CO 5. Explain the advanced Data Structures concepts such as Hashing Techniques and Optimal Binary Search Trees

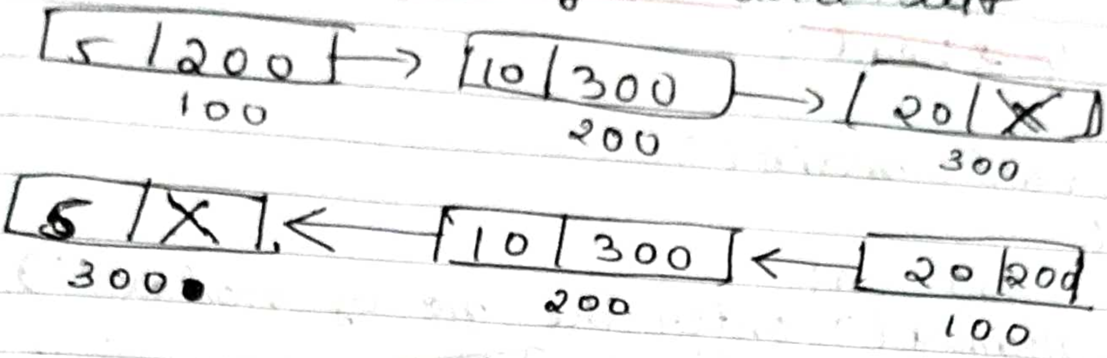
MODULE 2

Sl. No.	Questions	CO
Topic:	**Queues, Circular Queues, Using Dynamic Arrays, Multiple Stacks and queues**	
1.	Define Queue. Write QINSERT, QDELETE and QDISPLAY procedures for queues using arrays	CO2
2	Develop a C function to implement insertion, deletion and display operations of circular queue	CO2
3	Explain the ADT of Queue with its functions	CO2
4	Implement circular queue using dynamic arrays	CO2
5	Write C program to demonstrate multiple stacks and queues	CO2
Topic:	**Singly Linked, Lists and Chains, Representing Chains in C, Linked Stacks and Queues, Polynomials**	
6	Differentiate between array and Linked list	CO3
7	Develop a C functions to implement using Single linked list 1)Insert and Delete node from front 2) Insert and Delete node from rear end 3)Display/ printing the contents	CO3
8	Develop a C program to implement all Stack operations using Linked list	CO3
9	Develop a C program to implement all Queue operations using Linked list	CO3
10	Write C function to add two polynomials using Single linked list	CO3
11	Write C function to add two polynomials using Circular linked list	CO3

Module - 3

Operations on linked list

① Inverting a singly linked list



Initialize pointers prev as NULL, curr as head and next as NULL

/* before changing next pointer of current, store the next node */

next = curr → next

/*change next pointer of current node */

curr → next = prev

/* move prev and curr one step ahead */

Prev = curr

curr = next.

C Program to Reverse / Invert Single linked list

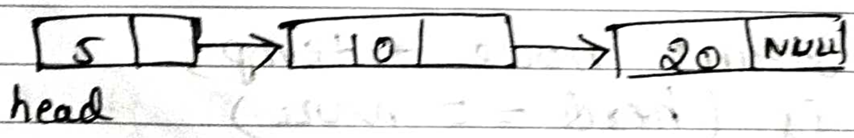
```
# include <stdio.h>
# include <stdlib.h>
 void  create list (int n);
 void  reverse_list();
 void  displaylist();
```

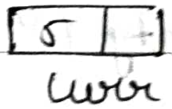
```
// create node
struct node
{
    int data;
    struct node *next;
} *head;

void main()
{
    int n;
  printf(" enter the no of nodes");
  scanf("%d", &n);
  create list (n);
  printf ("The list is \n");
   displaylist();
  printf (" The Reversed linked list is \n");
    reverse_list();
    display list();
}

 void createlist (int n)
{
 struct node *newnode, *temp;
 int data, i;
 head = (struct node *) malloc (sizeof
                          (struct node));
//when the list is empty
  if (head == NULL)
 {
  printf ("unable to allocate memory \n");
 }
 else
 {
 Printf (" enter the data of node 1: ");
 scanf ("%d", &data);
```

```
head -> data = data;
head -> next = NULL;
 temp = head;
 for ( i=2 ; i<=n; i++)
{
newnode = (struct node *) malloc
                  (sizeof (struct node));
 if (newnode == NULL)
 {
 Printf ("unable to allocate memory");
  break;
 }
 else
 {
 printf ("enter the data of node %d", i);
  scanf ("%d", &data);
 newnode -> data = data;
 newnode -> next = NULL;
  temp -> next = newnode;
   temp = temp -> next;  //make newnode
                           as temp
 }
}
}
}

 void displaylist ()
{
  struct node *temp;
  if (head == NULL)
  {
   printf ("List is empdy");
  }
  else
  {
   temp = head;
   while (temp != NULL)
   {
    printf("%d\t", temp->data);
     temp = temp->next;
   }
   printf("\n");
  }
}
```

```
   void reverse_list()
 {
   struct node * curr = head;
 struct node *prev = NULL, *next = NULL;
   while (curr != NULL)
  {  // store next
     next = curr->next;
     // reverse current node's pointer
     curr->next = prev;
    // move pointer
      prev = curr;
      curr  = next;
   }
   head = prev;
 }
```

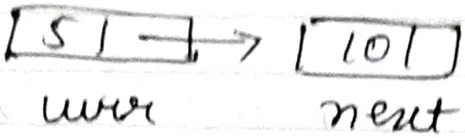
Eg



⇒ curr = head so 

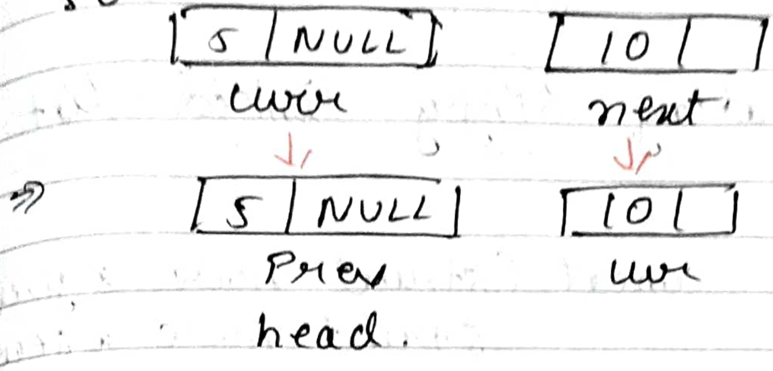
curr != NULL true so

next = curr -> next

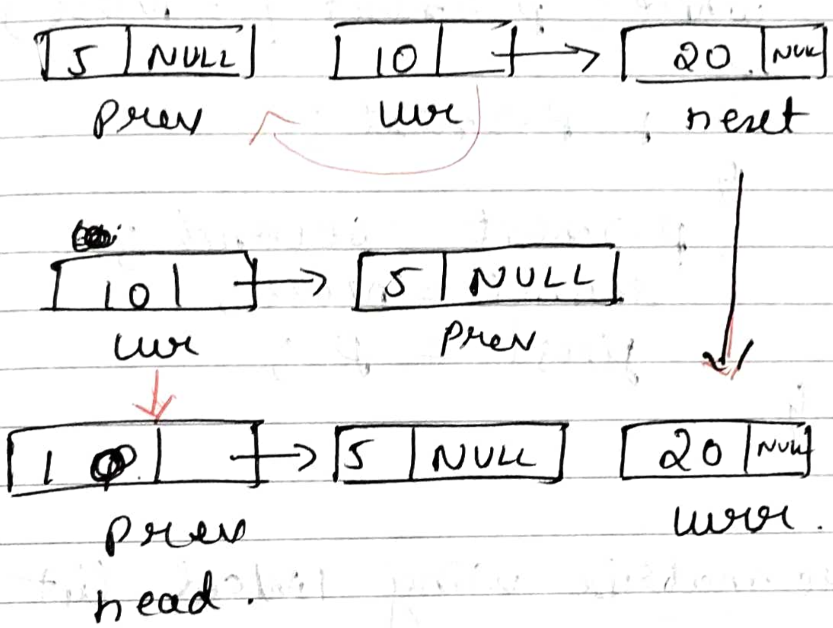


curr -> next = Prev = NULL

so

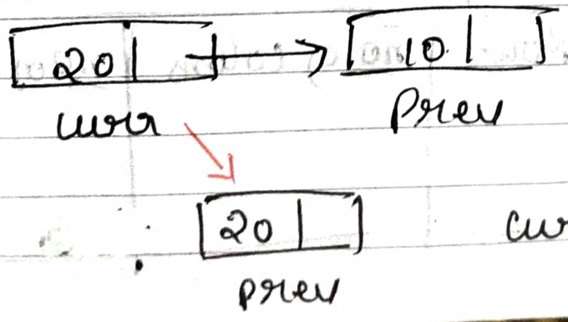


⇒ while curr != NULL true. so

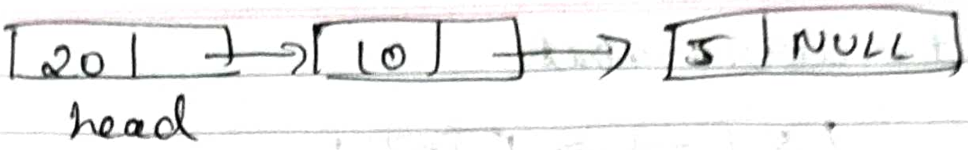


⇒ while curr != NULL true so

curr -> next = NULL = next



curr = NULL = next



//

while (curr ==NULL) so stops //

② concatenating Singly linked list

```
void  concat (struct node *first,
                  struct node * second)
{
   struct node *p = first ;
   while( p->next != NULL)
   {
        p=p->next ;
   }
      p->next = second ;
      second = NULL;
      first = p ;
}
```

Sparse matrix using linked list

we have three Structures

① Head Node

TR	TC	TNZ	
no of Rows	no of colums	no of value	Pointer Pointing to 1st row

② Row node

Row no	Pointer to next row	pointer to column

→ non zero value

PN	PNR	PC

③ column node

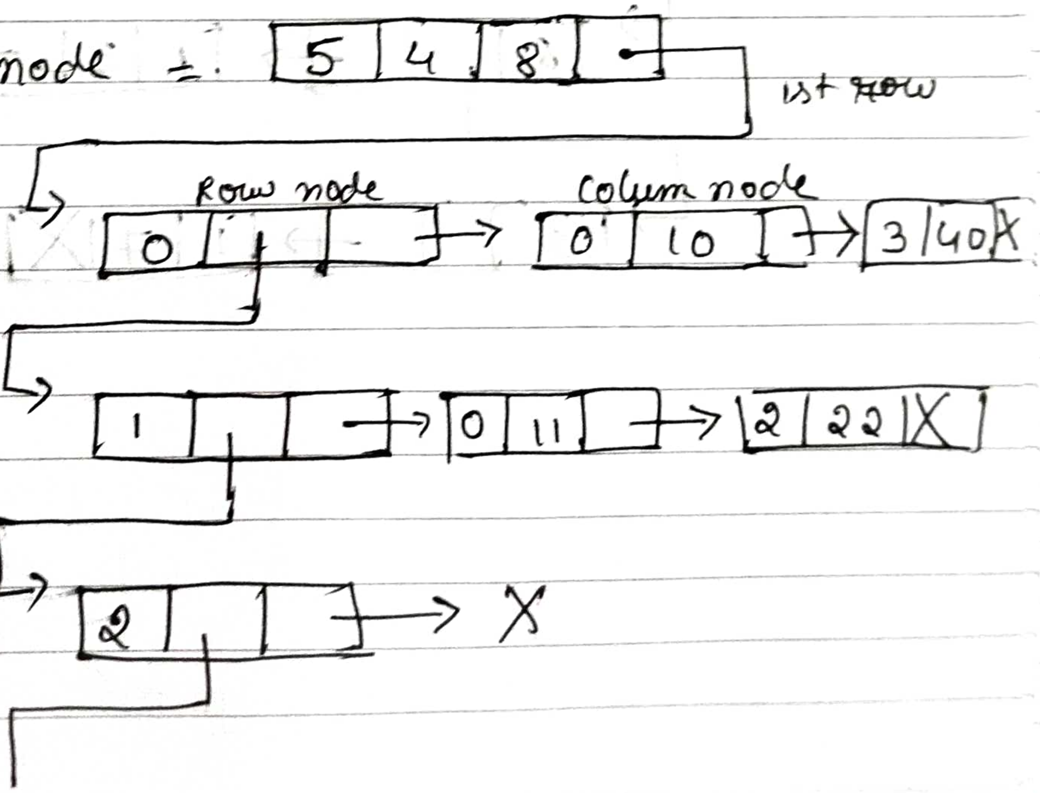
column no	value	Pointer to next column having non-zero in same row

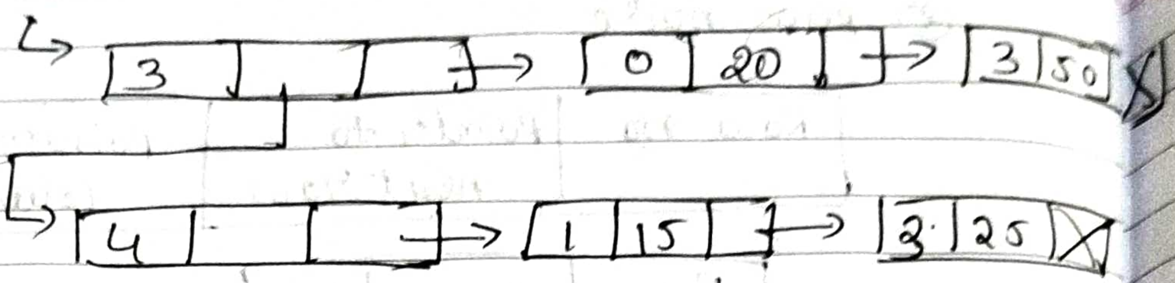
CN	Value	PNC

Ex

$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 10 & 0 & 0 & 40 \\ 1 & 11 & 0 & 22 & 0 \\ 2 & 0 & 0 & 0 & 0 \\ 3 & 20 & 0 & 0 & 50 \\ 4 & 0 & 15 & 0 & 25 \end{array}$$

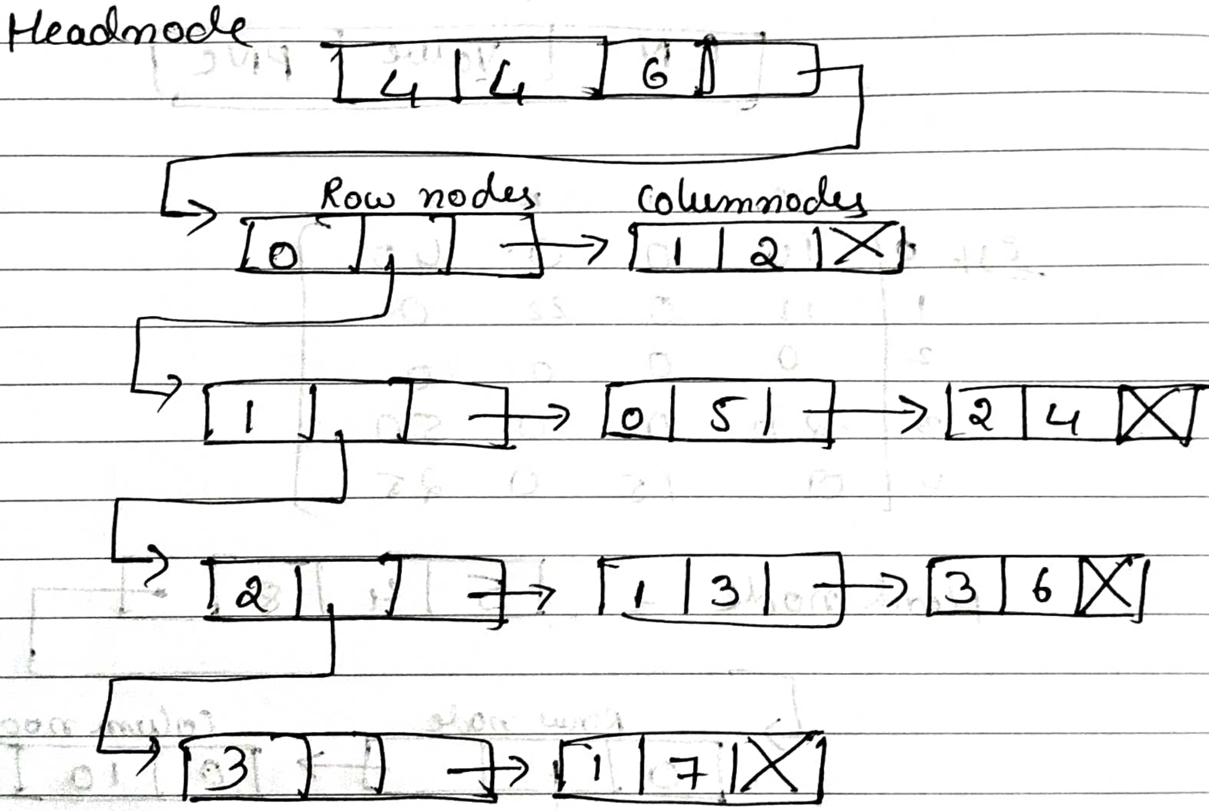
Head node :=





eg:-

$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 2 & 0 & 0 \\ 1 & 5 & 0 & 4 & 0 \\ 2 & 0 & 3 & 0 & 6 \\ 3 & 0 & 7 & 0 & 0 \end{array}$$



C program for Sparse Matrix Representation using Linked list

```
#include<stdio.h>
#include<stdlib.h>
#define R 4
#define C 5

// Node to represent row - list
struct row_list
{
    int row_number;
    struct row_list *link_down;
    struct value_list *link_right;
};

// Node to represent triples
struct value_list
{
    int column_index;
    int value;
    struct value_list *next;
};

// Function to create node for non - zero elements
void create_value_node(int data, int j, struct row_list **z)
{
    struct value_list *temp, *d;

    // Create new node dynamically
    temp = (struct value_list*)malloc(sizeof(struct value_list));
    temp->column_index = j+1;
    temp->value = data;
    temp->next = NULL;

    // Connect with row list
    if ((*z)->link_right==NULL)
        (*z)->link_right = temp;
    else
    {
        // d points to data list node
        d = (*z)->link_right;
        while(d->next != NULL)
            d = d->next;
        d->next = temp;
    }
}
```

```
// Function to create row list
void create_row_list(struct row_list **start, int row,
                                        int column, int Sparse_Matrix[R][C])
{
        // For every row, node is created
        for (int i = 0; i < row; i++)
        {
                struct row_list *z, *r;

                // Create new node dynamically
                z = (struct row_list*)malloc(sizeof(struct row_list));
                z->row_number = i+1;
                z->link_down = NULL;
                z->link_right = NULL;
                if (i==0)
                        *start = z;
                else
                {
                        r = *start;
                        while (r->link_down != NULL)
                                r = r->link_down;
                        r->link_down = z;
                }

                // Firstly node for row is created,
                // and then traversing is done in that row
                for (int j = 0; j < 5; j++)
                {
                        if (Sparse_Matrix[i][j] != 0)
                        {
                                create_value_node(Sparse_Matrix[i][j], j, &z);
                        }
                }
        }
}

//Function display data of LIL
void print_LIL(struct row_list *start)
{
        struct row_list *r;
        struct value_list *z;
        r = start;

        // Traversing row list
        while (r != NULL)
        {
                if (r->link_right != NULL)
                {
```

```
            printf("row=%d \n", r->row_number);
            z = r->link_right;

            // Traversing data list
            while (z != NULL)
            {
                printf("column=%d value=%d \n",
                    z->column_index, z->value);
                z = z->next;
            }
        }
        r = r->link_down;
    }
}

//Driver of the program
int main()
{
    // Assume 4x5 sparse matrix
    int Sparse_Matrix[R][C] =
    {
        {0 , 0 , 3 , 0 , 4 },
        {0 , 0 , 5 , 7 , 0 },
        {0 , 0 , 0 , 0 , 0 },
        {0 , 2 , 6 , 0 , 0 }
    };

    // Start with the empty List of lists
    struct row_list* start = NULL;

    //Function creating List of Lists
    create_row_list(&start, R, C, Sparse_Matrix);

    // Display data of List of lists
    print_LIL(start);
    return 0;
}
```

Menu Driven Program in C to implement all the operations of Doubly linked list

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
    struct node *prev;
    struct node *next;
    int data;
};
struct node *head;
void insertion_beginning();
void insertion_last();
void insertion_specified();
void deletion_beginning();
void deletion_last();
void deletion_specified();
void display();
void search();
void main ()
{
int choice =0;
    while(choice != 9)
    {
        printf("\n*********Main Menu*********\n");
        printf("\nChoose one option from the following list ...\n");
        printf("\n===============================================\n");
        printf("\n1.Insert in begining\n2.Insert at last\n3.Insert at any random location\n4.Delete from Beginning\n  5.Delete from last\n6.Delete the node after the given data\n7.Search\n8.Show\n9.Exit\n");
```

```
      printf("\nEnter your choice?\n");
      scanf("\n%d",&choice);
      switch(choice)
      {
         case 1:  insertion_beginning();  break;
         case 2:  insertion_last();  break;
         case 3:  insertion_specified(); break;
         case 4:  deletion_beginning(); break;
         case 5:  deletion_last();  break;
         case 6:   deletion_specified(); break;
         case 7:   search();  break;
         case 8:   display();  break;
         case 9:   exit(0);  break;
         default:   printf("Please enter valid choice..");
      }
   }
}
void insertion_beginning()
{
  struct node *ptr;
  int item;
  ptr = (struct node *)malloc(sizeof(struct node));
  if(ptr == NULL)
  {
     printf("\nOVERFLOW");
  }
  else
  {
   printf("\nEnter Item value");
   scanf("%d",&item);
  if(head==NULL)
  {
     ptr->next = NULL;
     ptr->prev=NULL;
     ptr->data=item;
     head=ptr;
  }
  else
  {
     ptr->data=item;
     ptr->prev=NULL;
     ptr->next = head;
     head->prev=ptr;
     head=ptr;
  }
  printf("\nNode inserted\n");
}
}
void insertion_last()
{
  struct node *ptr,*temp;
  int item;
  ptr = (struct node *) malloc(sizeof(struct node));
  if(ptr == NULL)
  {
     printf("\nOVERFLOW");
  }
  else
  {
     printf("\nEnter value");
     scanf("%d",&item);
      ptr->data=item;
     if(head == NULL)
     {
         ptr->next = NULL;
         ptr->prev = NULL;
         head = ptr;
     }
     else
     {
        temp = head;
        while(temp->next!=NULL)
        {
            temp = temp->next;
        }
        temp->next = ptr;
        ptr ->prev=temp;
        ptr->next = NULL;
        }
     }
   printf("\nnode inserted\n");
  }
void insertion_specified()
{
  struct node *ptr,*temp;
  int item,loc,i;
  ptr = (struct node *)malloc(sizeof(struct node));
  if(ptr == NULL)
  {
     printf("\n OVERFLOW");
  }
  else
  {
     temp=head;
     printf("Enter the location");
     scanf("%d",&loc);
     for(i=0;i<loc;i++)
     {
         temp = temp->next;
         if(temp == NULL)
         {
             printf("\n There are less than %d elements", loc);
             return;
         }
     }
     printf("Enter value");
     scanf("%d",&item);
     ptr->data = item;
     ptr->next = temp->next;
     ptr -> prev = temp;
     temp->next = ptr;
     temp->next->prev=ptr;
     printf("\nnode inserted\n");
  }
}
void deletion_beginning()
{
  struct node *ptr;
  if(head == NULL)
  {
      printf("\n UNDERFLOW");
  }
  else if(head->next == NULL)
  {
      head = NULL;
      free(head);
      printf("\nnode deleted\n");
  }
  else
  {
      ptr = head;
      head = head -> next;
      head -> prev = NULL;
      free(ptr);
      printf("\nnode deleted\n");
  }
}
void deletion_last()
{
  struct node *ptr;
  if(head == NULL)
  {
      printf("\n UNDERFLOW");
  }
  else if(head->next == NULL)
  {
      head = NULL;
      free(head);
      printf("\nnode deleted\n");
  }
  else
  {
      ptr = head;
      if(ptr->next != NULL)
      {
          ptr = ptr -> next;
      }
      ptr -> prev -> next = NULL;
      free(ptr);
      printf("\nnode deleted\n");
  }
}
void deletion_specified()
{
  struct node *ptr, *temp;
  int val;
  printf("\n Enter the data after which the node is to be deleted : ");
  scanf("%d", &val);
  ptr = head;
  while(ptr -> data != val)
  ptr = ptr -> next;
  if(ptr -> next == NULL)
  {
      printf("\nCan't delete\n");
  }
  else if(ptr -> next -> next == NULL)
  {
      ptr ->next = NULL;
  }
  else
  {
      temp = ptr -> next;
      ptr -> next = temp -> next;
      temp -> next -> prev = ptr;
      free(temp);
      printf("\nnode deleted\n");
  }
}
void display()
{
  struct node *ptr;
  printf("\n printing values...\n");
  ptr = head;
  while(ptr != NULL)
  {
      printf("%d\n",ptr->data);
      ptr=ptr->next;
  }
}
void search()
{
  struct node *ptr;
  int item,i=0,flag;
  ptr = head;
  if(ptr == NULL)
  {
      printf("\nEmpty List\n");
  }
  else
  {
      printf("\nEnter item which you want to search?\n");
      scanf("%d",&item);
      while (ptr!=NULL)
      {
          if(ptr->data == item)
          {
              printf("\nitem found at location %d ",i+1);
              flag=0;
              break;
          }
          else
          {
              flag=1;
          }
          i++;
          ptr = ptr -> next;
      }
      if(flag==1)
      {
          printf("\nItem not found\n");
      }
  }
}
```

C program to implement all the operations on Circular Doubly linked list

```
include<stdio.h>
#include<stdlib.h>
struct node
{
   struct node *prev;
   struct node *next;
   int data;
};
struct node *head;
void insertion_beginning();
void insertion_last();
void deletion_beginning();
void deletion_last();
void display();
void search();
void main ()
{
int choice =0;
   while(choice != 9)
   {
      printf("\n*********Main Menu*********\n");
      printf("\nChoose one option from the following list ...\n");
      printf("\n========================================
====\n");
      printf("\n1.Insert in Beginning\n2.Insert at last\n3.Delete from Beginning\n4.
Delete from last\n5.Search\n6.Show\n7.Exit\n");
      printf("\nEnter your choice?\n");
```

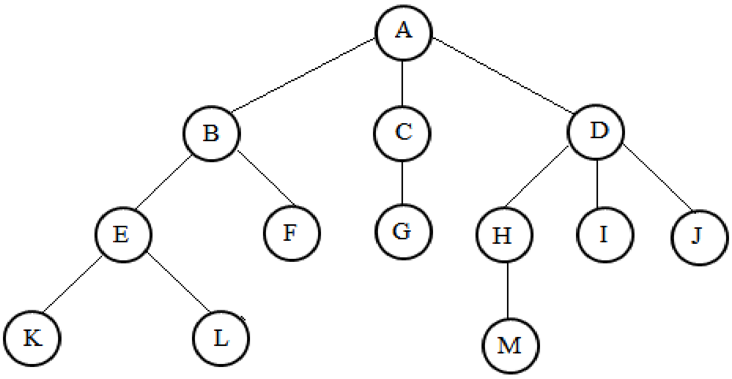
```
    scanf("\n%d",&choice);
    switch(choice)
    {
      case 1:   insertion_beginning();  break;
      case 2:  insertion_last();   break;
       case 3:  deletion_beginning();  break;
      case 4:  deletion_last();  break;
       case 5:   search();  break;
     case 6:    display();  break;
       case 7:   exit(0);  break;
     default:  printf("Please enter valid choice..");
    }
  }
}
void insertion_beginning()
{
  struct node *ptr,*temp;
  int item;
  ptr = (struct node *)malloc(sizeof(struct node));
  if(ptr == NULL)
  {
    printf("\nOVERFLOW");
  }
  else
  {
   printf("\nEnter Item value");
   scanf("%d",&item);
   ptr->data=item;
  if(head==NULL)
  {
      head = ptr;
      ptr -> next = head;
      ptr -> prev = head;
   }
   else
   {
       temp = head;
    while(temp -> next != head)
    {
        temp = temp -> next;
    }
    temp -> next = ptr;
    ptr -> prev = temp;
    head -> prev = ptr;
    ptr -> next = head;
    head = ptr;
   }
   printf("\nNode inserted\n");
}
}
void insertion_last()
{
   struct node *ptr,*temp;
   int item;
   ptr = (struct node *) malloc(sizeof(struct node));
   if(ptr == NULL)
   {
       printf("\nOVERFLOW");
   }
   else
   {
       printf("\nEnter value");
       scanf("%d",&item);
        ptr->data=item;
       if(head == NULL)
       {
           head = ptr;
           ptr -> next = head;
           ptr -> prev = head;
       }
       else
       {
          temp = head;
          while(temp->next !=head)
          {
              temp = temp->next;
          }
          temp->next = ptr;
          ptr ->prev=temp;
          head -> prev = ptr;
      ptr -> next = head;
        }
   }
     printf("\nnode inserted\n");
}

void deletion_beginning()
{
    struct node *temp;
    if(head == NULL)
    {
        printf("\n UNDERFLOW");
    }
    else if(head->next == head)
    {
        head = NULL;
        free(head);
        printf("\nnode deleted\n");
    }
    else
    {
        temp = head;
        while(temp -> next != head)
        {
            temp = temp -> next;
        }
        temp -> next = head -> next;
        head -> next -> prev = temp;
        free(head);
        head = temp -> next;
    }
}
void deletion_last()
{
    struct node *ptr;
    if(head == NULL)
    {
        printf("\n UNDERFLOW");
    }
    else if(head->next == head)
    {
        head = NULL;
        free(head);
        printf("\nnode deleted\n");
    }
    else
    {
        ptr = head;
        if(ptr->next != head)
        {
            ptr = ptr -> next;
        }
        ptr -> prev -> next = head;
        head -> prev = ptr -> prev;
        free(ptr);
        printf("\nnode deleted\n");
    }
}

void display()
{
    struct node *ptr;
    ptr=head;
    if(head == NULL)
    {
        printf("\nnothing to print");
    }
    else
    {
        printf("\n printing values ... \n");
        while(ptr -> next != head)
        {

            printf("%d\n", ptr -> data);
            ptr = ptr -> next;
        }
        printf("%d\n", ptr -> data);
    }

}

void search()
{
    struct node *ptr;
    int item,i=0,flag=1;
    ptr = head;
    if(ptr == NULL)
    {
        printf("\nEmpty List\n");
    }
    else
    {
        printf("\nEnter item which you want to search?\n");
        scanf("%d",&item);
        if(head ->data == item)
        {
        printf("item found at location %d",i+1);
        flag=0;
        }
        else
        {
        while (ptr->next != head)
        {
            if(ptr->data == item)
            {
                printf("item found at location %d ",i+1);
                flag=0;
                break;
            }
            else
            {
                flag=1;
            }
            i++;
            ptr = ptr -> next;
        }
        }
        if(flag != 0)
        {
            printf("Item not found\n");
        }
    }

}
```

MODULE 3: TREES

DEFINITION

A *tree* is a finite set of one or more nodes such that

- There is a specially designated node called *root*.
- The remaining nodes are partitioned into n >= 0 disjoint set $T_1,\ldots,T_n$, where each of these sets is a tree. $T_1,\ldots,T_n$ are called the *subtrees* of the root.

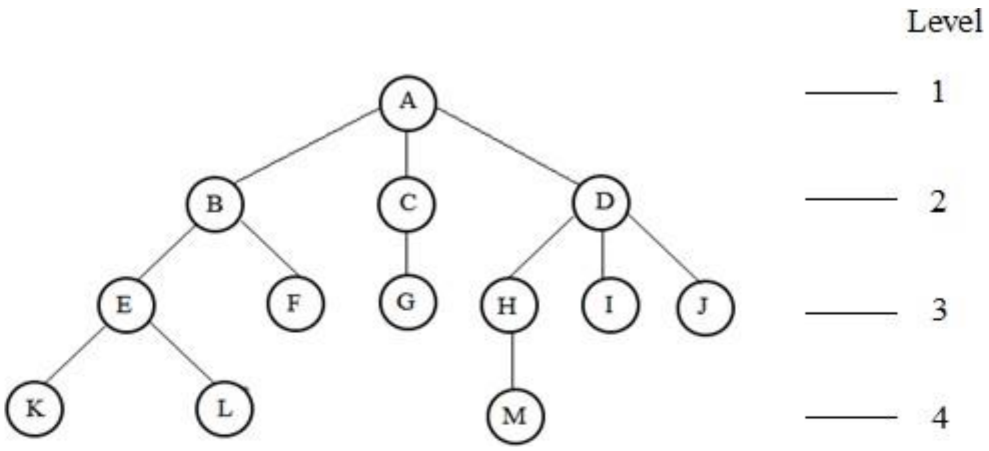


Every node in the tree is the root of some subtree

TERMINOLOGY

- ***Node:*** The item of information plus the branches to other nodes
- ***Degree:*** The number of subtrees of a node
- ***Degree of a tree:*** The maximum of the degree of the nodes in the tree.
- ***Terminal nodes (or leaf):*** nodes that have degree zero or node with no successor
- ***Nonterminal nodes***: nodes that don't belong to terminal nodes.
- ***Parent and Children:*** Suppose N is a node in T with left successor S1 and right successor S2, then N is called the Parent (or father) of S1 and S2. Here, S1 is called left child (or Son) and S2 is called right child (or Son) of N.
- ***Siblings:*** Children of the same parent are said to be siblings.
- ***Edge:*** A line drawn from node N of a T to a successor is called an edge
- ***Path:*** A sequence of consecutive edges from node N to a node M is called a path.
- ***Ancestors of a node:*** All the nodes along the path from the root to that node.
- ***The level of a node:*** defined by letting the root be at level zero. If a node is at level *l*, then it children are at level *l+1*.
- ***Height (or depth):*** The maximum level of any node in the tree

Example



A is the root node

B is the parent of E and F

C and D are the sibling of B

E and F are the children of B

K, L, F, G, M, I, J are external nodes, or leaves

A, B, C, D, E, H are internal nodes

The level of E is 3

The height (depth) of the tree is 4

The degree of node B is 2

The degree of the tree is 3

The ancestors of node M is A, D, H

The descendants of node D is H, I, J, M

Representation of Trees

There are several ways to represent a given tree such as:

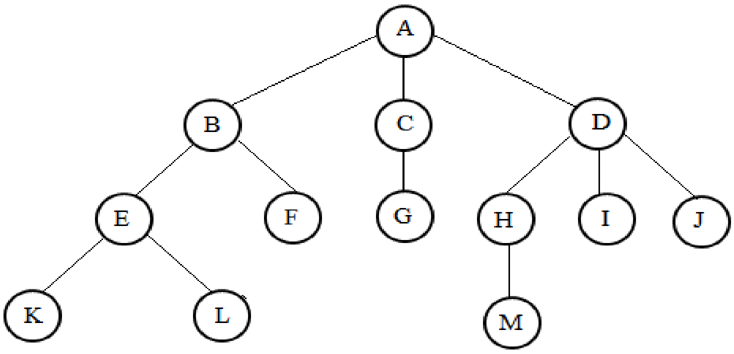


Figure (A)

1. List Representation
2. Left Child- Right Sibling Representation
3. Representation as a Degree-Two tree

List Representation:

The tree can be represented as a List. The tree of **figure (A)** could be written as the list.

(A (B (E (K, L), F), C (G), D (H (M), I, J)))

- The information in the root node comes first.
- The root node is followed by a list of the subtrees of that node.

Tree node is represented by a memory node that has fields for the data and pointers to the tree node's children

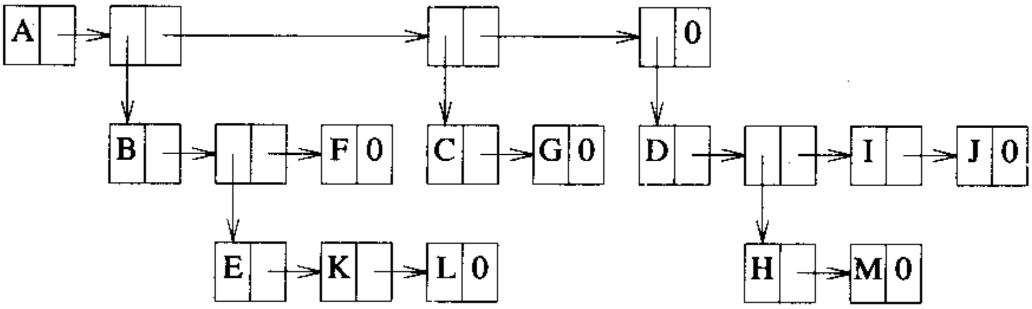
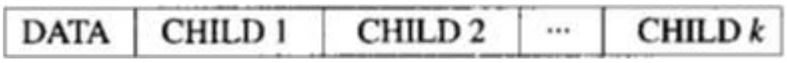


Figure (B): List representation of the tree of figure (A)

Since the degree of each tree node may be different, so memory nodes with a varying number of pointer fields are used.

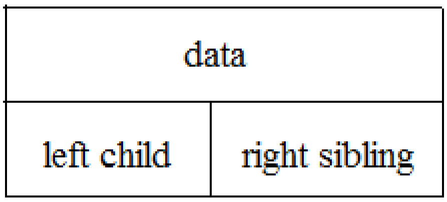
For a tree of degree k, the node structure can be represented as below figure. Each child field is used to point to a subtree.



DATA	CHILD 1	CHILD 2	···	CHILD *k*

Left Child-Right Sibling Representation

The below figure show the node structure used in the left child-right sibling representation



data	
left child	right sibling

Figure (c): Left child right sibling node structure

To convert the tree of Figure (A) into this representation:

1. First note that every node has at most one leftmost child
2. At most one closest right sibling.

Ex:

- In Figure (A), the leftmost child of A is B, and the leftmost child of D is H.
- The closest right sibling of B is C, and the closest right sibling of H is I.
- Choose the nodes based on how the tree is drawn. The left child field of each node points to its leftmost child (if any), and the right sibling field points to its closest right sibling (if any).

Figure (D) shows the tree of Figure (A) redrawn using the left child-right sibling representation.

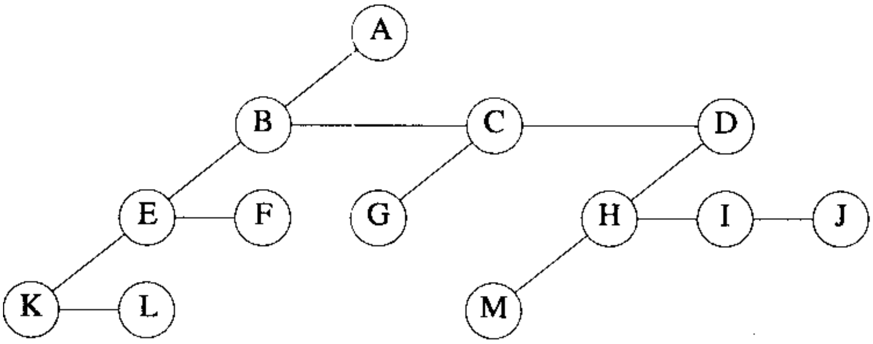


Figure (D): Left child-right sibling representation of tree of figure (A)

<u>Representation as a Degree-Two Tree</u>

To obtain the degree-two tree representation of a tree, simply rotate the right-sibling pointers in a left child-right sibling tree clockwise by 45 degrees. This gives us the degree-two tree displayed in Figure (E).

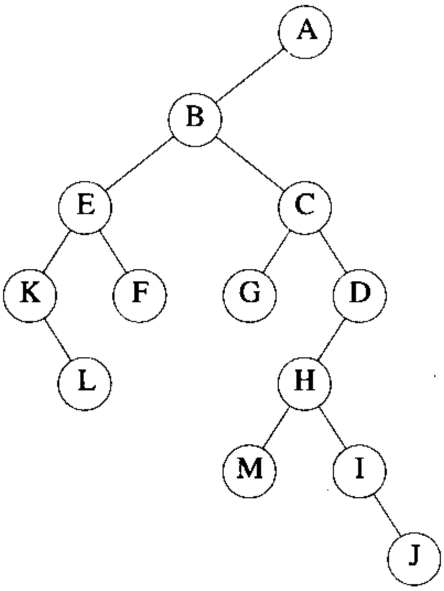


Figure (E): degree-two representation

In the degree-two representation, a node has two children as the left and right children.

BINARY TREES

Definition: A binary tree T is defined as a finite set of nodes such that,

- T is empty or
- T consists of a root and two disjoint binary trees called the left subtree and the right subtree.

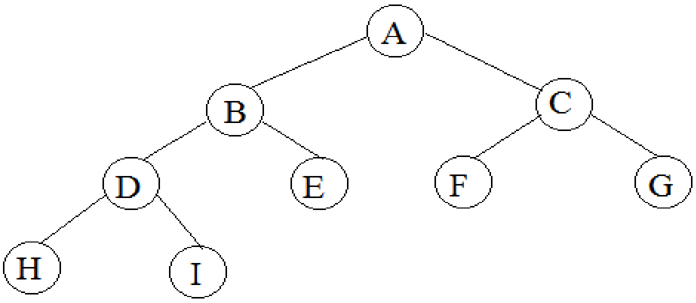


Figure: Binary Tree

Different kinds of Binary Tree

1. Skewed Tree

A skewed tree is a tree, skewed to the left or skews to the right.

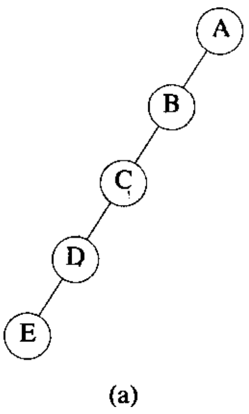
or

It is a tree consisting of only left subtree or only right subtree.

- A tree with only left subtrees is called Left Skewed Binary Tree.
- A tree with only right subtrees is called Right Skewed Binary Tree.

2. Complete Binary Tree

A binary tree T is said to complete if all its levels, except possibly the last level, have the maximum number node 2^i, $i \geq 0$ and if all the nodes at the last level appears as far left as possible.



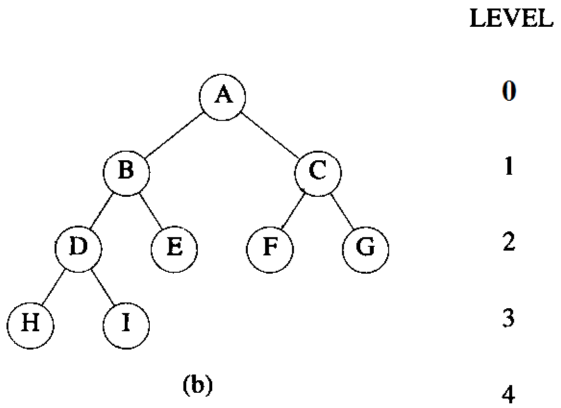


Figure (a): Skewed binary tree   Figure (b): Complete binary tree

3. Full Binary Tree

A full binary tree of depth ‘k’ is a binary tree of depth k having $2^k - 1$ nodes, $k \geq 1$.

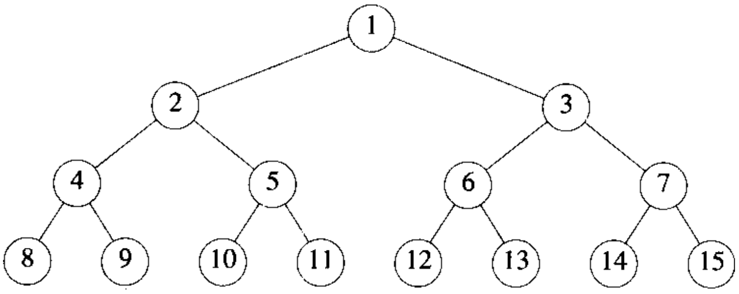
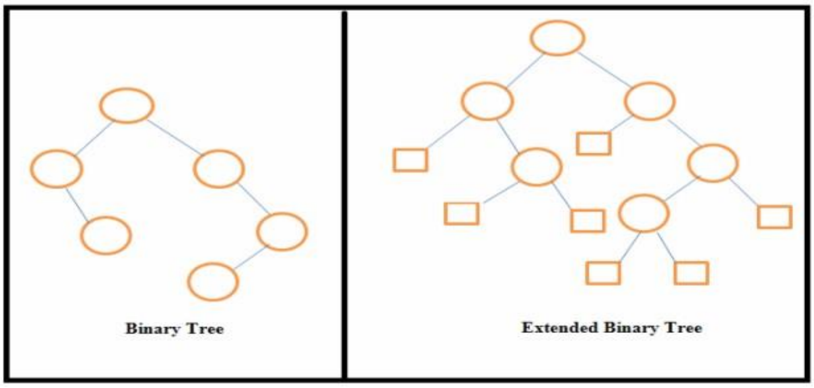


Figure: Full binary tree of level 4 with sequential node number

4. Extended Binary Trees or 2-trees

An *extended binary tree* is a transformation of any binary tree into a complete binary tree. This transformation consists of replacing every null subtree of the original tree with “special nodes.” The nodes from the original tree are then *internal nodes*, while the special nodes are *external nodes*.

For instance, consider the following binary tree.



The following tree is its extended binary tree. The circles represent internal nodes, and square represent external nodes.

Every internal node in the extended tree has exactly two children, and every external node is a leaf. The result is a complete binary tree.

PROPERTIES OF BINARY TREES

Lemma 1: [Maximum number of nodes]:

(1) The maximum number of nodes on level i of a binary tree is 2^{i-1}, $i \geq 1$.

(2) The maximum number of nodes in a binary tree of depth k is $2^k - 1$, $k \geq 1$.

Proof:

(1) The proof is by induction on i.

Induction Base: The root is the only node on level i = 1. Hence, the maximum number of nodes on level i =1 is $2^{i-1} = 2^0 = 1$.

Induction Hypothesis: Let i be an arbitrary positive integer greater than 1. Assume that the maximum number of nodes on level i -1is 2^{i-2}

Induction Step: The maximum number of nodes on level i -1 is 2^{i-2} by the induction hypothesis. Since each node in a binary tree has a maximum degree of 2, the maximum number of nodes on level i is two times the maximum number of nodes on level i-1, or 2^{i-1}

(2) The maximum number of nodes in a binary tree of depth k is

$$\sum_{i=0}^{k} (\text{maximum number of nodes on level } i) = \sum_{i=0}^{k} 2^{i-1} = 2^{k-1}$$

Lemma 2: [Relation between number of leaf nodes and degree-2 nodes]:

For any nonempty binary tree, T, if n_0 is the number of leaf nodes and n_2 the number of nodes of degree 2, then $n_0 = n_2 + 1$.

Proof: Let ***n_1*** be the number of nodes of degree one and ***n*** the total number of nodes.

Since all nodes in T are at most of degree two, we have

$$n = n_0 + n_1 + n_2 \qquad (1)$$

Count the number of branches in a binary tree. If B is the number of branches, then

$$n = B + 1.$$

All branches stem from a node of degree one or two. Thus,

$$B = n_1 + 2n_2.$$

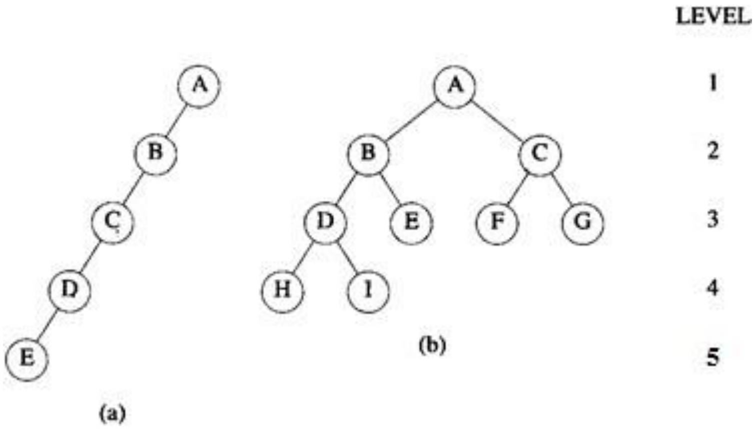
Hence, we obtain

$$n = B + 1 = n_1 + 2n_2 + 1 \qquad (2)$$

Subtracting Eq. (2) from Eq. (1) and rearranging terms, we get

$$\mathbf{n_0 = n_2 + 1}$$

Consider the figure:



Here, For Figure (b) $n_2=4$, $n_0= n_2+1= 4+1=5$
Therefore, the total number of leaf node=5

BINARY TREE REPRESENTATION

The storage representation of binary trees can be classified as

1. Array representation
2. Linked representation.

Array representation:

- A tree can be represented using an array, which is called sequential representation.
- The nodes are numbered from 1 to n, and one dimensional array can be used to store the nodes.
- Position 0 of this array is left empty and the node numbered *i* is mapped to position *i* of the array.

Below figure shows the array representation for both the trees of figure (a).

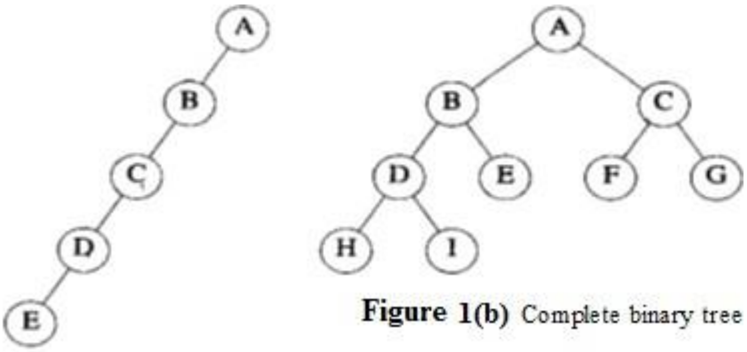
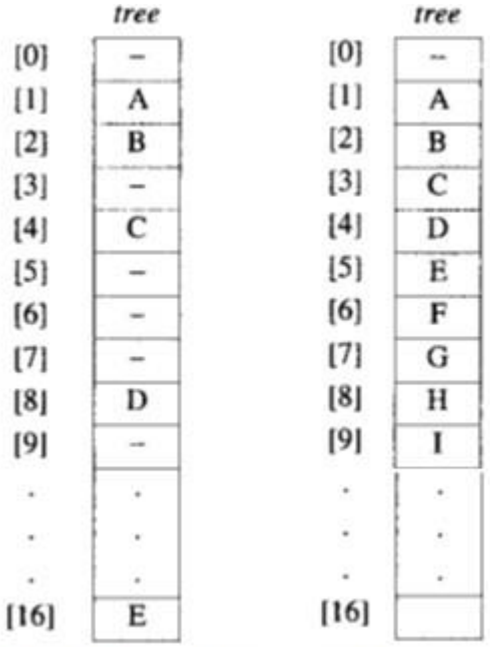


Figure 1(b) Complete binary tree

Figure 1(a) Skewed binary tree



	tree	tree
[0]	–	–
[1]	A	A
[2]	B	B
[3]	–	C
[4]	C	D
[5]	–	E
[6]	–	F
[7]	–	G
[8]	D	H
[9]	–	I
.	.	.
.	.	.
.	.	.
[16]	E	

(a). Tree of figure 1(a) **(b). Tree of figure 1(b)**

- For complete binary tree the array representation is ideal, as no space is wasted.
- For the skewed tree less than half the array is utilized.

Linked representation:

The problems in array representation are:

- It is good for complete binary trees, but more memory is wasted for skewed and many other binary trees.
- The insertion and deletion of nodes from the middle of a tree require the movement of many nodes to reflect the change in level number of these nodes.

These problems can be easily overcome by linked representation

Each node has three fields,

- LeftChild - which contains the address of left subtree
- RightChild - which contains the address of right subtree.
- Data - which contains the actual information

C Code for node:

```
typedef struct node *treepointer;
typedef struct {
                    int data;
                    treepointer leftChild, rightChild;
              }node;
```

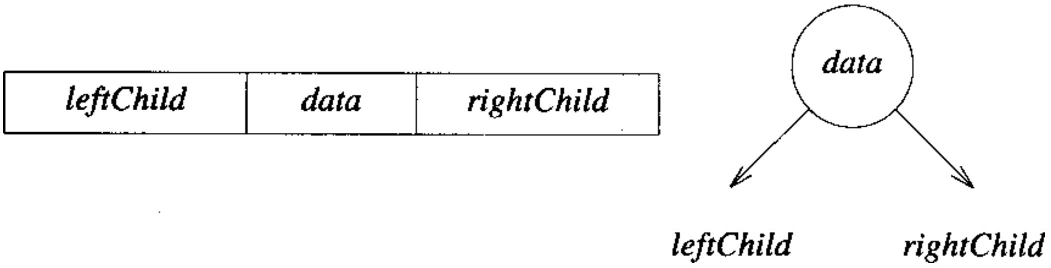


Figure: Node representation

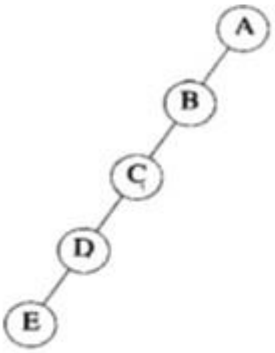


Figure 1(a) Skewed binary tree

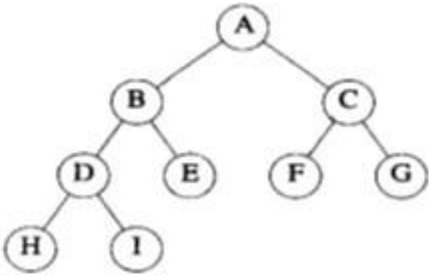
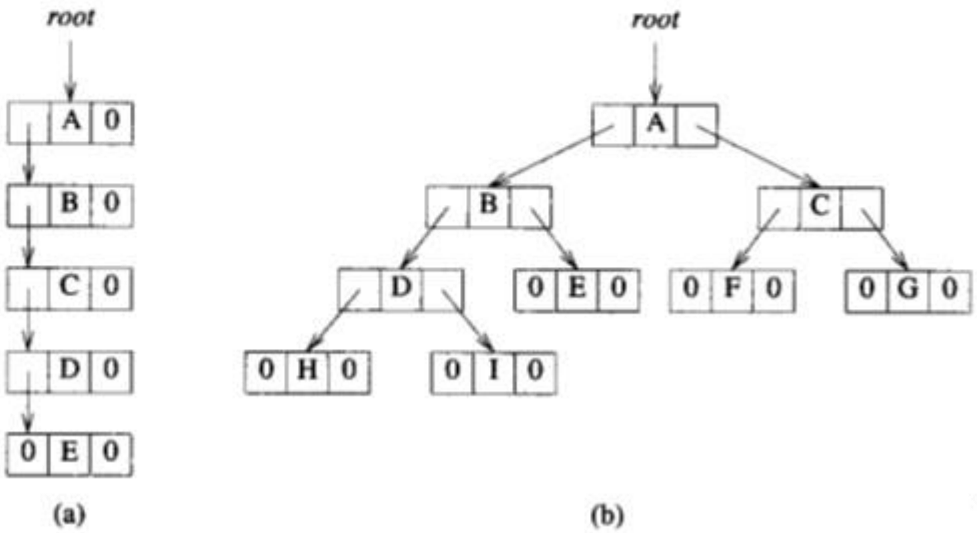


Figure 1(b) Complete binary tree



Linked representation of the binary tree

BINARY TREE TRAVERSALS

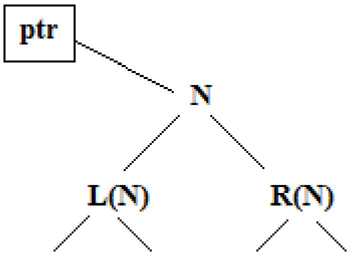
Visiting each node in a tree exactly once is called tree traversal

The different methods of traversing a binary tree are:

1. Preorder
2. Inorder
3. Postorder
4. Iterative inorder Traversal
5. Level-Order traversal

1. Inorder: Inorder traversal calls for moving down the tree toward the left until you cannot go further. Then visit the node, move one node to the right and continue. If no move can be done, then go back one more node.

Let ptr is the pointer which contains the location of the node N currently being scanned.
L(N) denotes the leftchild of node N and R(N) is the right child of node N



Recursion function:

The inorder traversal of a binary tree can be recursively defined as

- Traverse the left subtree in inorder.
- Visit the root.
- Traverse the right subtree in inorder.

```
void inorder(treepointerptr)
{
        if (ptr)
        {
                inorder   (ptr→leftchild);
                printf   (“%d”,ptr→data);
                inorder  (ptr→rightchild);
           }
    }
```

2. Preorder: Preorder is the procedure of visiting a node, traverse left and continue. When you cannot continue, move right and begin again or move back until you can move right and resume.

Recursion function:

The Preorder traversal of a binary tree can be recursively defined as

- Visit the root
- Traverse the left subtree in preorder.
- Traverse the right subtree in preorder

```
void preorder (treepointerptr)
{
        if (ptr)
        {
                printf (“%d”,ptr→data)
                preorder (ptr→leftchild);
                preorder (ptr→rightchild);
           }
  }
```

3. Postorder: Postorder traversal calls for moving down the tree towards the left until you can go no further. Then move to the right node and then visit the node and continue.

Recursion function:

The Postorder traversal of a binary tree can be recursively defined as

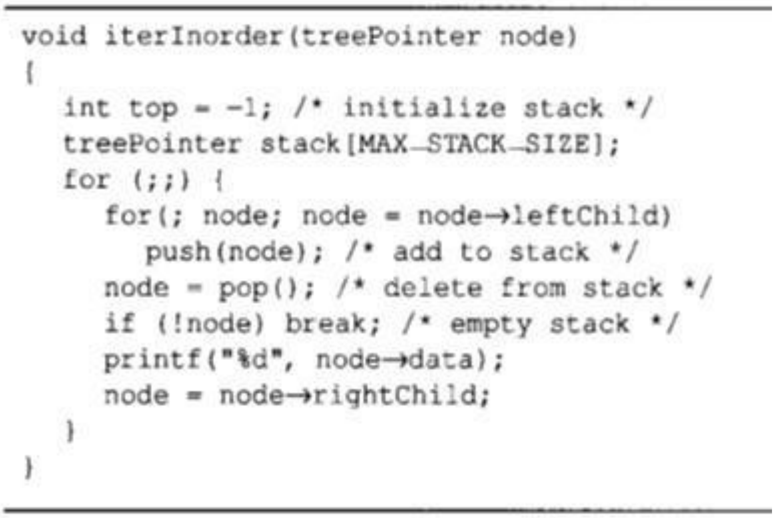
- Traverse the left subtree in postorder.
- Traverse the right subtree in postorder.
- Visit the root

```
void  postorder(treepointerptr)
{
        if (ptr)
        {
                postorder (ptr→leftchild);
                postorder (ptr→rightchild);
                printf (“%d”,ptr→data);
          }
   }
```

4. Iterative inorder Traversal:

Iterative inorder traversal explicitly make use of stack function.

The left nodes are pushed into stack until a null node is reached, the node is then removed from the stack and displayed, and the node's right child is stacked until a null node is reached. The traversal then continues with the left child. The traversal is complete when the stack is empty.



```
void iterInorder(treePointer node)
{
   int top = -1; /* initialize stack */
   treePointer stack[MAX_STACK_SIZE];
   for (;;) {
      for(; node; node = node→leftChild)
         push(node); /* add to stack */
      node = pop(); /* delete from stack */
      if (!node) break; /* empty stack */
      printf("%d", node→data);
      node = node→rightChild;
   }
}
```

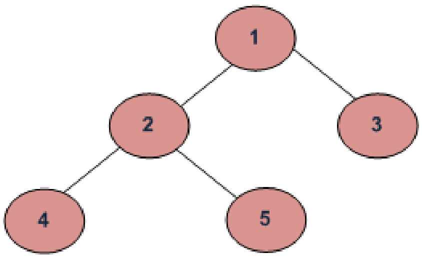
Program : Iterative inorder traversal

5. Level-Order traversal:

Visiting the nodes using the ordering suggested by the node numbering is called level ordering traversing.

The nodes in a tree are numbered starting with the root on level 1 and so on.

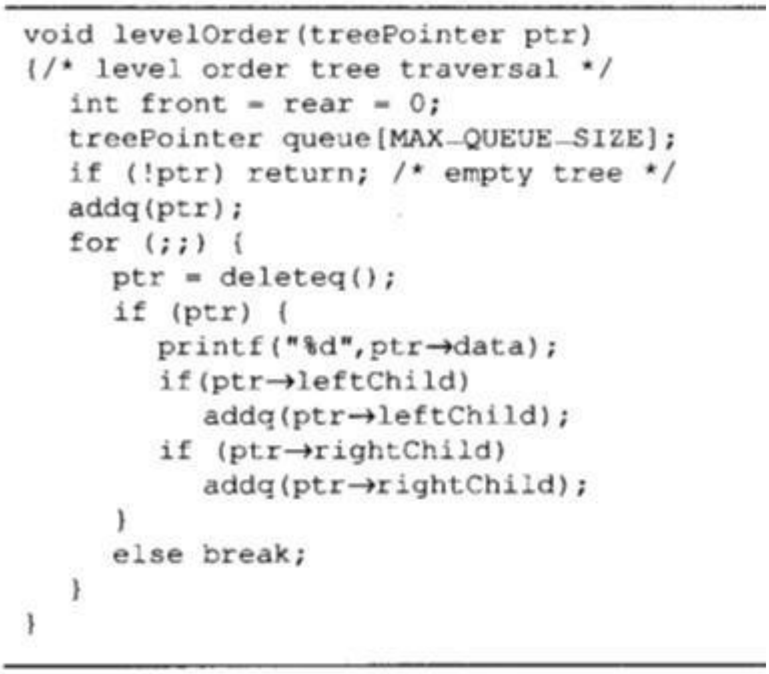
Firstly visit the root, then the root's left child, followed by the root's right child. Thus continuing in this manner, visiting the nodes at each new level from the leftmost node to the rightmost node.



Level order traversal: 1 2 3 4 5

Initially in the code for level order add the root to the queue. The function operates by deleting the node at the front of the queue, printing the nodes data field and adding the nodes left and right children to the queue.

Function for level order traversal of a binary tree:



```
void levelOrder(treePointer ptr)
{/* level order tree traversal */
   int front = rear = 0;
   treePointer queue[MAX_QUEUE_SIZE];
   if (!ptr) return; /* empty tree */
   addq(ptr);
   for (;;) {
      ptr = deleteq();
      if (ptr) {
         printf("%d",ptr→data);
         if(ptr→leftChild)
            addq(ptr→leftChild);
         if (ptr→rightChild)
            addq(ptr→rightChild);
      }
      else break;
   }
}
```

Program : Level-order traversal of a binary tree

ADDITIONAL BINARY TREE OPERATIONS

1. Copying a Binary tree

This operations will perform a copying of one binary tree to another.

C function to copy a binary tree:

```
treepointer copy(treepointer original)
{  if(original)
     {   MALLOC(temp,sizeof(*temp));
         temp→leftchild=copy(original→leftchild);
         temp→rightchild=copy(original→rightchild);
         temp→data=original→data;
         return temp;
      }
    return NULL;
}
```

2. Testing Equality

This operation will determin the equivalance of two binary tree. Equivalance binary tree have the same strucutre and the same information in the corresponding nodes.

C function for testing equality of a binary tree:

```
int equal(treepointer first,treepointer second)
{
        return((!first && !second) || (first && second && (first→data==second→data)
        && equal(first→leftchild,second→leftchild) && equal(first→rightchild,
        second→rightchild))
}
```

This function will return TRUE if two trees are equivalent and FALSE if they are not.

3. The Satisfiability problem

- Consider the formula that is constructed by set of variables: x_1, x_2, …, x_n and operators $\wedge$(*and*), $\vee$(*or*), $\neg$ (*not*).
- The variables can hold only of two possible values, *true* or *false*.
- The expression can form using these variables and operators is defined by the following rules.
    - A variable is an expression
    - If x and y are expressions, then $\neg x$, $x \wedge y$, $x \vee y$ are expressions
    - Parentheses can be used to alter the normal order of evaluation ($\neg > \wedge > \vee$)

Example: $x_1 \vee (x_2 \wedge \neg x_3)$ If x_1 and x_3 are *false* and x_2 is *true*

$= \mathit{false} \vee (\mathit{true} \wedge \neg \mathit{false})$

$= \mathit{false} \vee \mathit{true}$

$= \mathit{true}$

The satisfiablity problem for formulas of the propositional calculus asks if there is an assignment of values to the variable that causes the value of the expression to be *true.*

Let's assume the formula in a binary tree

$(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_3) \vee \neg x_3$

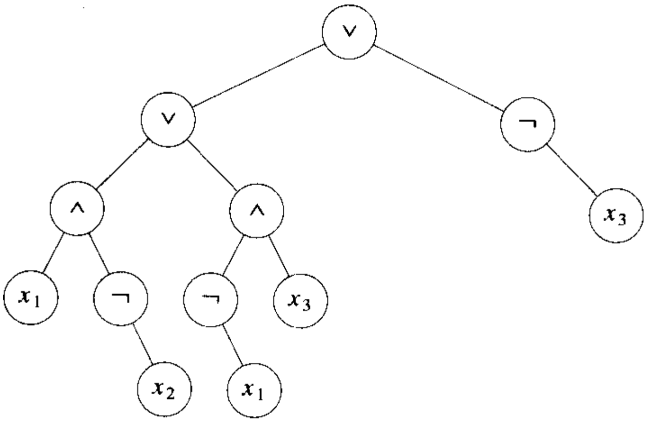


Figure : Propositional formula in a binary tree

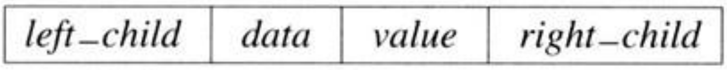
The inorder traversal of this tree is

$$x_1 \wedge \neg x_2 \vee \neg\, x_1 \wedge x_3 \vee \neg x_3$$

The algorithm to determine satisfiablity is to let (x_1, x_2, x_3) takes on all the possible combination of true and false values to check the formula for each combination.

For *n* value of an expression, there are 2^n possible combinations of *true* and *false*
For example n=3, the eight combinations are (t,t,t), (t,t,f), (t,f,t), (t,f,f), (f,t,t), (f,t,f), (f,f,t), (f,f,f).
The algorithm will take O(g 2^n), where g is the time to substitute values for x_1, x_2,… x_n and evaluate the expression.

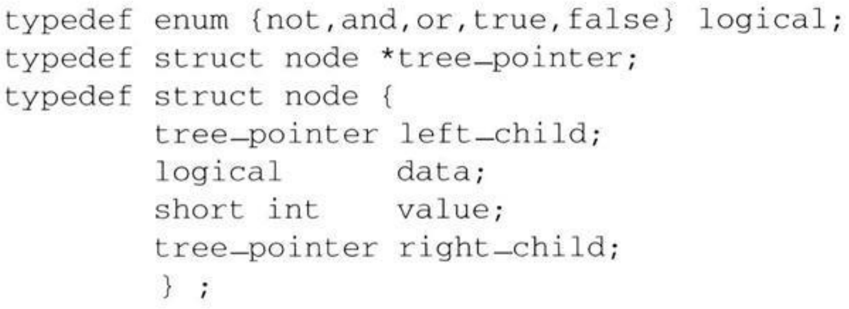
Node structure:
For the purpose of evaluation algorithm, assume each node has four fields:



left–child	*data*	*value*	*right–child*

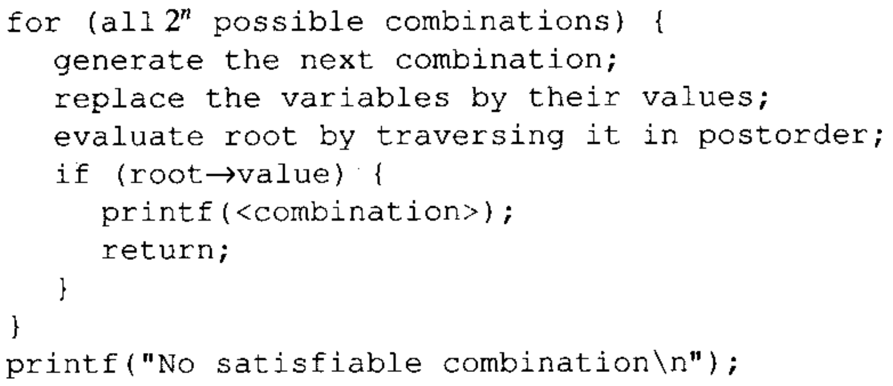
Figure : Node structure for the satisfiability problem

Define this node structure in C as:



```
typedef enum {not,and,or,true,false} logical;
typedef struct node *tree_pointer;
typedef struct node {
        tree_pointer left_child;
        logical      data;
        short int    value;
        tree_pointer right_child;
        } ;
```

Satisfiability function: The first version of Satisfiability algorithm



```
for (all 2^n possible combinations) {
   generate the next combination;
   replace the variables by their values;
   evaluate root by traversing it in postorder;
   if (root→value) {
      printf(<combination>);
      return;
   }
}
printf("No satisfiable combination\n");
```

THREADED BINARY TREE

The limitations of binary tree are:

- In binary tree, there are n+1 null links out of 2n total links.
- Traversing a tree with binary tree is time consuming.

These limitations can be overcome by threaded binary tree.

In the linked representation of any binary tree, there are more null links than actual pointers. These null links are replaced by the pointers, called **threads**, which points to other nodes in the tree.

To construct the threads use the following rules:

1. Assume that **ptr** represents a node. If ptr→leftChild is null, then replace the null link with a pointer to the inorder predecessor of ptr.
2. If ptr →rightChild is null, replace the null link with a pointer to the inorder successor of ptr.

Ex: Consider the binary tree as shown in below figure:

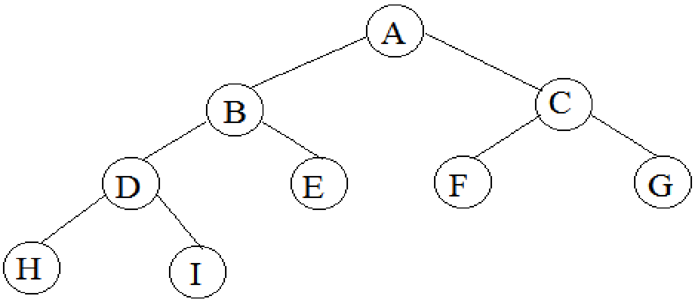


Figure A: Binary Tree

There should be no loose threads in threaded binary tree. But in **Figure B** two threads have been left dangling: one in the left child of *H,* the other in the right child of G.

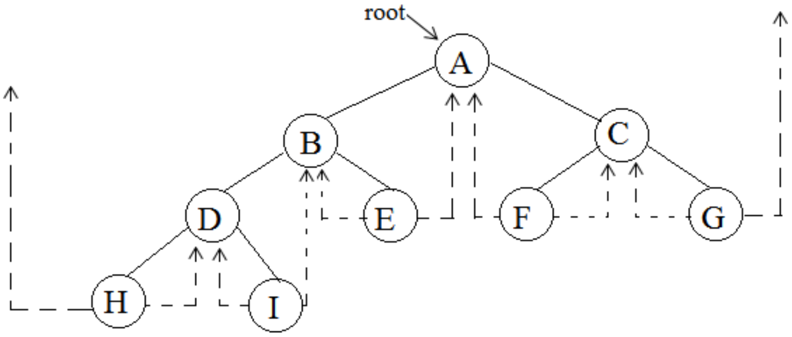


Figure B: Threaded tree corresponding to Figure A

In above figure the new threads are drawn in broken lines. This tree has 9 node and 10 0-links which has been replaced by threads.

When trees are represented in memory, it should be able to distinguish between threads and pointers. This can be done by adding two additional fields to node structure, ie., *leftThread* and *rightThread*

- If `ptr→leftThread = TRUE,` then `ptr→leftChild` contains a thread, otherwise it contains a pointer to the left child.
- If `ptr→rightThread = TRUE, then ptr→rightChild` contains a thread, otherwise it contains a pointer to the right child.

Node Structure:

The node structure is given in C declaration

```
typedef  struct  threadTree  *threadPointer
typedef struct{
               short  int leftThread;
               threadPointer  leftChild;
               char data;
               threadPointer  rightChild;
               short  int rightThread;
          }threadTree;
```

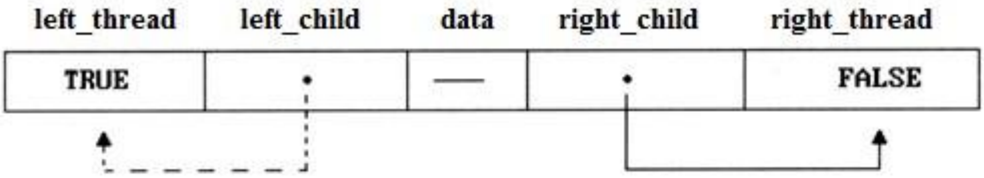
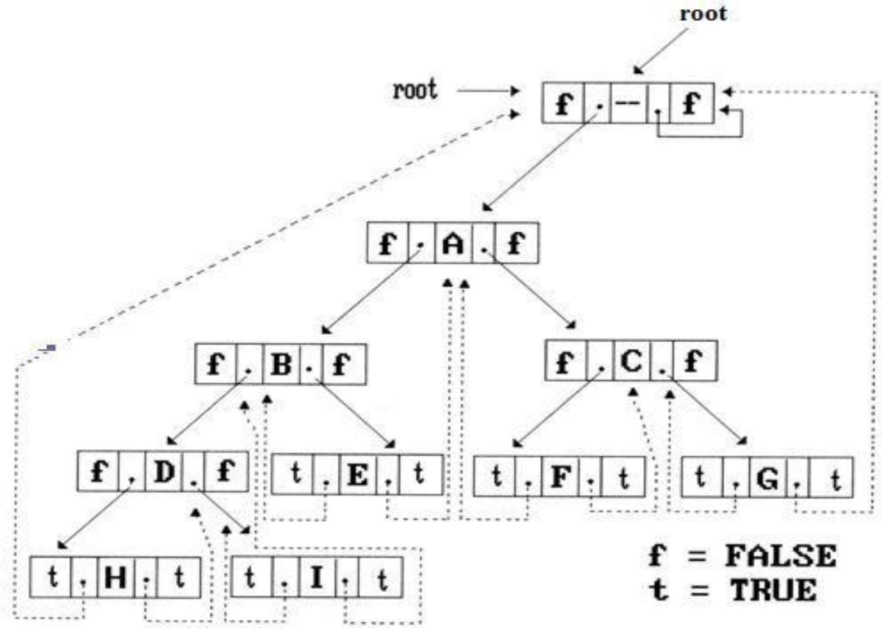


Figure An empty threaded tree

The complete memory representation for the tree of figure is shown in Figure C



The variable ***root*** points to the header node of the tree, while **root →leftChild** points to the start of the first node of the actual tree. This is true for all threaded trees. Here the problem of the loose threads is handled by pointing to the head node called ***root***.

Inorder Traversal of a Threaded Binary Tree

- By using the threads, an inorder traversal can be performed without making use of a stack.
- For any node, **ptr**, in a threaded binary tree, if **ptr→rightThread =TRUE,** the inorder successor of **ptr** is **ptr →rightChild** by definition of the threads. Otherwise we obtain the inorder successor of **ptr** by following a path of **left-child links** from the **right-child** of **ptr** until we reach a node with **leftThread = TRUE.**
- The function insucc **( )** finds the inorder successor of any node in a threaded tree without using a stack.

```
threadedpointer  insucc(threadedPointer  tree)
{        /* find the inorder successor of tree in a threaded binary tree */
          threadedpointer temp;
         temp = tree→rightChild;
         if (!tree→rightThread)
            while (!temp→leftThread)
                     temp = temp→leftChild;
         return temp;
```

Program: Finding inorder successor of a node

To perform inorder traversal make repeated calls to insucc ( ) function

```
void tinorder (threadedpointer tree)
{
Threadedpointer temp = tree;
for(; ;){
                temp = insucc(temp);
                if (temp == tree)     break;
                printf("%3c", temp→data
      }
}
```

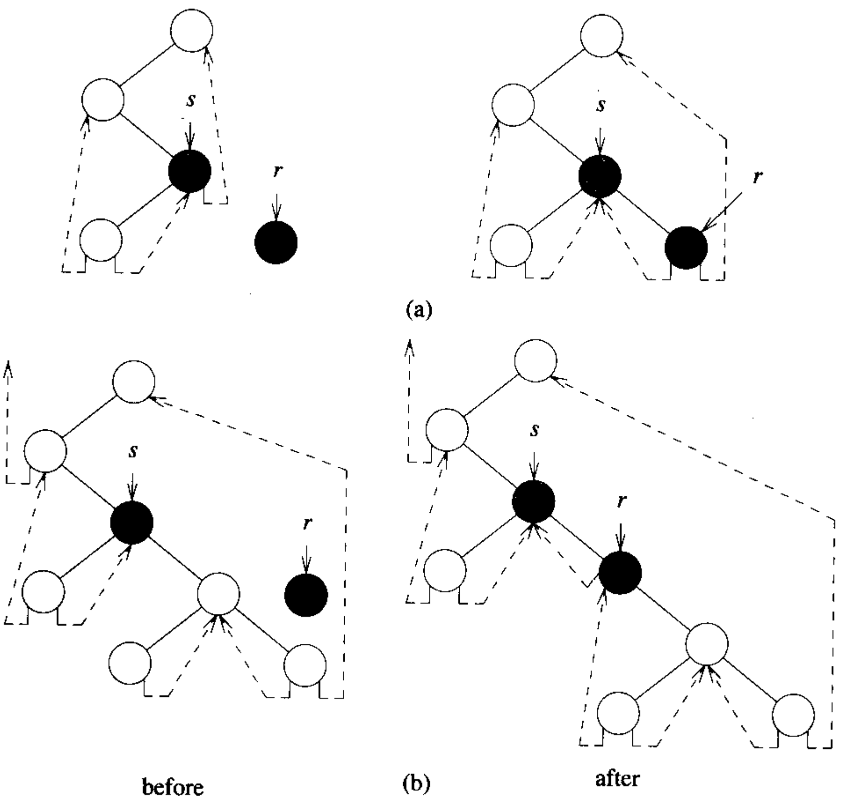
Program: Inorder traversal of a threaded binary tree

Inserting a Node into a Threaded Binary Tree

In this case, the insertion of **r** as the right child of a node **s** is studied.

The cases for insertion are:

- If ***s*** has an **empty** right subtree, then the insertion is simple and diagrammed in Figure
- If the right subtree of ***s*** is **not empty**, then this right subtree is made the right subtree of ***r*** after insertion. When this is done, r becomes the inorder predecessor of a node that has a **leftThread == true** field, and consequently there is a thread which has to be updated to point to r. The node containing this thread was previously the inorder successor of s.



```
void insertRight(threadedPointer Sf threadedPointer r)
{     /* insert r as the right child of s */
        threadedpointer   temp;
        r→rightChild  = parent→rightChild;
        r→rightThread = parent→rightThread;
        r→leftChild = parent;
        r→leftThread = TRUE;
        s→rightChild = child;
        s→rightThread  = FALSE;
        if (!r→rightThread) {
                temp = insucc(r);
                temp→leftChild  = r;
         }
}
```

||Jai Sri Gurudev ||
BGSKH Education Trust (R.) – A unit of Sri Adichunchanagiri Shikshana Trust(R.)

BGS College of Engineering and Technology (BGSCET)

Mahalakshmipuram, West of Chord Road, Bengaluru-560086
(Approved by AICTE, New Delhi and Affiliated to VTU, Belagavi)

Question Bank

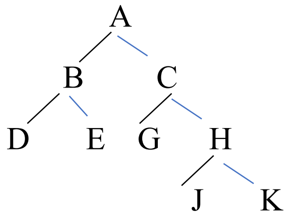
Subject: Data Structure and Applications **Class:** AIDS

Subject code: BCS304 **Faculty:** Mrs. Jyothi R

Course Outcomes

CO 1. Explain different data structures and their applications.
CO 2. Apply Arrays, Stacks and Queue data structures to solve the given problems.
CO 3. Use the concept of linked list in problem solving.
CO 4. Develop solutions using trees and graphs to model the real-world problem.
CO 5. Explain the advanced Data Structures concepts such as Hashing Techniques and Optimal Binary Search Trees

MODULE 3		
Sl. No.	**Questions**	**CO**
Topic:	**LINKED LISTS: Additional List Operations, Sparse Matrices, Doubly Linked List**	
1.	Develop a C function to Invert/Reverse a Single linked list using example	CO3
2	Develop a C function to Concatenate single linked list	CO3
3	Show diagrammatic linked representation for the following sparse matrix $\begin{matrix} 0 & 1 & 2 \\ 3 & 0 & 3 \\ 0 & 0 & 0 \end{matrix}$ $\begin{matrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{matrix}$ $\begin{matrix} 0 & 0 & 4 & 0 & 0 \\ 6 & 5 & 0 & 0 & 0 \\ 0 & 3 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 2 \end{matrix}$ $\begin{matrix} 3 & 0 & 0 & 0 \\ 5 & 0 & 0 & 6 \\ 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 8 \\ 0 & 0 & 9 & 0 \end{matrix}$	CO3
4	Differentiate Single Linked List(SLL) and Double Linked List(DLL)	CO3
5	Write C program to develop Double linked list with following functions 1)Insert a node at front end of the list 2)Delete a node from front end of the list 3)Display	CO3
6	Write C program to develop Double linked list with following functions 1)Insert a node at rear end of the list 2)Delete a node from rear end of the list 3)Searching a node with given key value	CO3
7	Write C program to develop Double linked list with following functions 1)Insert a node at specific location of the list 2)Delete a node from specific location of the list	CO3
8	Write a C program to represent and add two polynomials using singly circular linked list	CO3
Topic:	**TREES: Introduction, Binary Trees, Binary Tree Traversals, Threaded Binary Trees**	
9	Define Tree. With the examples explain the terminologies of tree.	CO4

10	Explain the representation of trees	CO4
11	What is Binary tree and explain its properties with proof	CO4
12	Taking a binary tree as example show the representation of binary tree with both array and linked list ways.	CO4
13	Write a C functions for of binary tree.	CO4
14	Consider a following tree T write Inorder, Preorde and Postorder traversals along with its functions.  	CO4
15	Explain five types of Binary tree	CO4
16	Explain the Threaded Binary trees along with function for insertion and inorder traversal	CO4

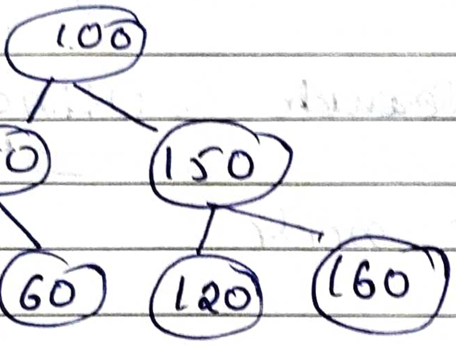
Module - 4

Binary Search tree (BST)

→ BST is a binary tree in which for each node x in the tree, elements in the left subtree are less than info(x) and elements in the right subtree are greater than info(x).

→ Every node in the tree should satisfy this condition.

Ex



Common operations

① Traversal :- Inorder, Preorder & postorder.

② Searching :- Search for a specific item in tree

③ Insertion :- Insert an item into B.S.T

④ Deletion :- Deleting a node from B.S.T

Searching a BST for an item

- Begin the search from the root node.
- Compare K with key value in root. If K equals the root's key, then search Successfull.
- If K is less than root's key, then search the left subtree of the root.
- If K is larger than root's key, then search the right subtree

~~node search~~ using Recursion

```
struct node
{
    int info;
    struct node * llink;
    struct node * rlink;
};
typedef struct node * NODE;

  NODE search (int item, NODE root)
{
      if (root == NULL)
          return root;      //item not found
      if (item == roo->info)
      return root;
       if (item < root->info)
      return search (item, root->llink);
      else
      return search (item, root->rlink);
}
```

search in BST using Iteration

```
NODE search( int item, NODE root)
{
    NODE cur;
    if (root = = NULL)
      return NULL ;
    cur = root ;
    while( cur ! = NULL)
   {
      if (item = = cur->info)
        return cur;
      if (item < cur->info)
        cur = cur->llink;
     else
        cur = cur->rlink;
   }
   return NULL;        //key not found
}
```

Inserting an element (node) into BST

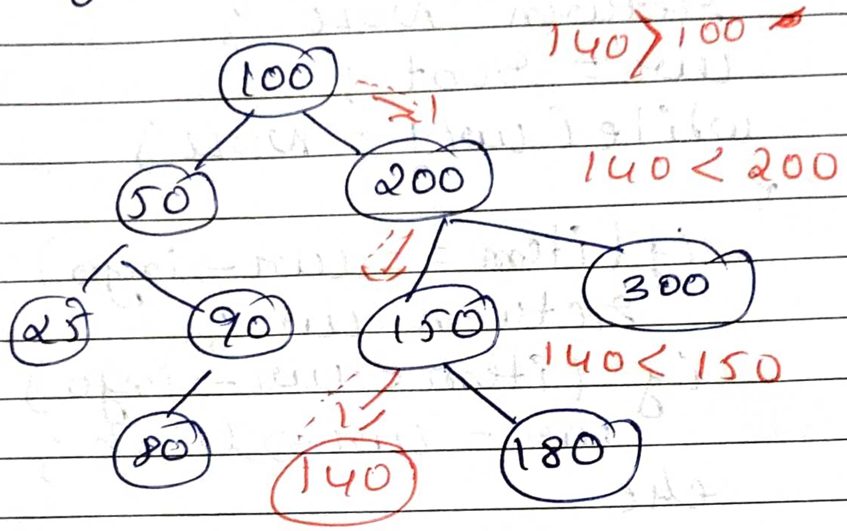
-> check whether new node k is not there in BST.

-> First create a node using malloc function and assign a key value (info) to it.

-> If tree is empty then newly

created node becomes the first node in the tree.

→ To insert check with root value if lesser travel towards left if greater travel towards right and get exact position & insert.

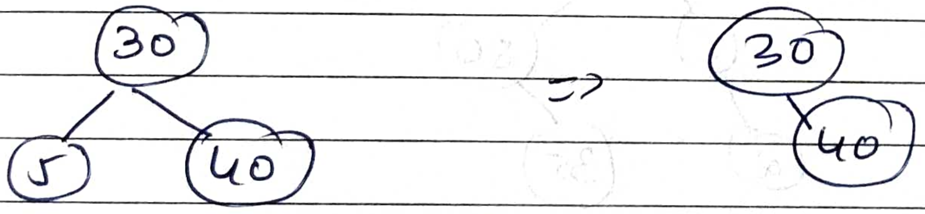


Insert 140

```
NODE insert (int item, NODE root)
{
   NODE temp, cur, prev;
  temp = (NODE *) malloc (sizeof(NODE));
    temp->info = item;
   temp->llink = temp->rlink = NULL;
   if (root == NULL)
     return temp;
     prev = NULL;
      cur = root;
    while (cur != NULL)
    {
          prev = cur;
          if ( item == cur->info)
{
  Printf("Duplicate items are not
                 allowed");
    free(temp);
    return root;
  }
  if (item < cur->info)
    cur = cur->llink;
  else
    cur = cur->rlink;
}
  if (item < prev->info)
    prev->llink = temp;
  else
    prev->rlink = temp;
  return root;
}
```

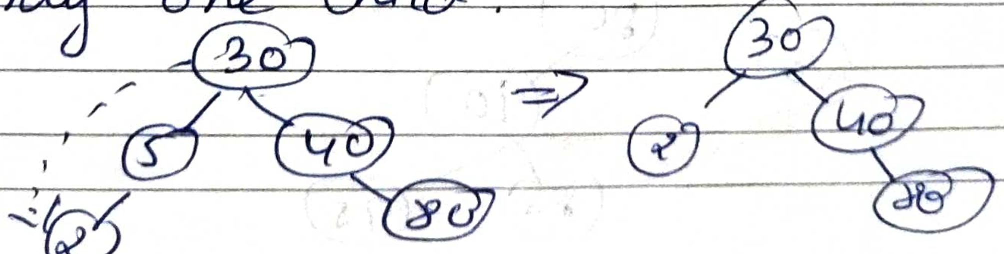
Deletion from a BST

① Delete a leaf node is simple



Delete 5 delete cur node 5 & set Leftnode of 30 to NULL.

② Deletion of a non-leaf node that has only one child.



Delete 5 . so change the pointer link from parent node 30 to 2

Parent →llink = cur →llink.

③ Delete non-leaf node that has 2 children

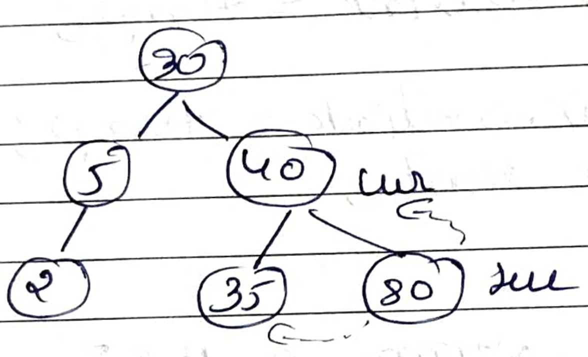
⇒ obtain inorder successor of node to be deleted.

⇒ Attach left subtree of node to left of inorder succ.

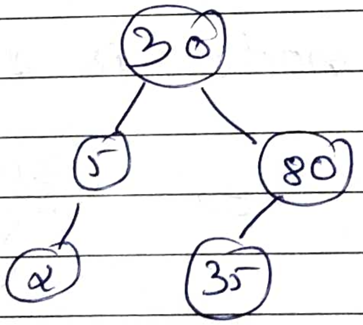
⇒ obtain right subtree of the node

⇒ attach q to parent.

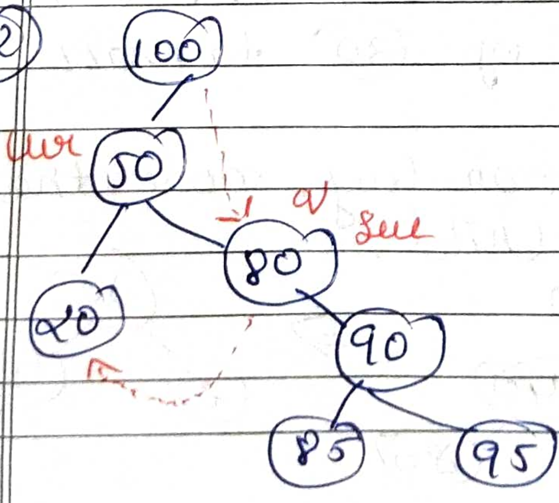
①



Delete 40.

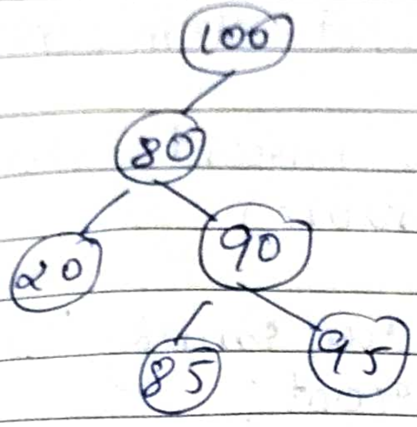


②

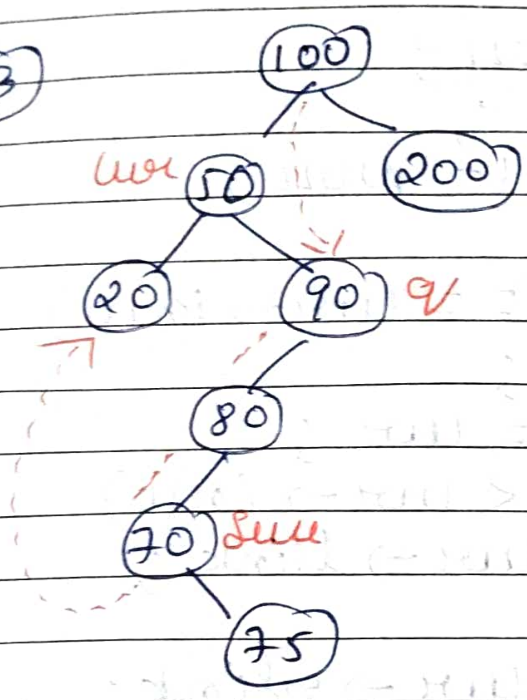


Delete 50

10

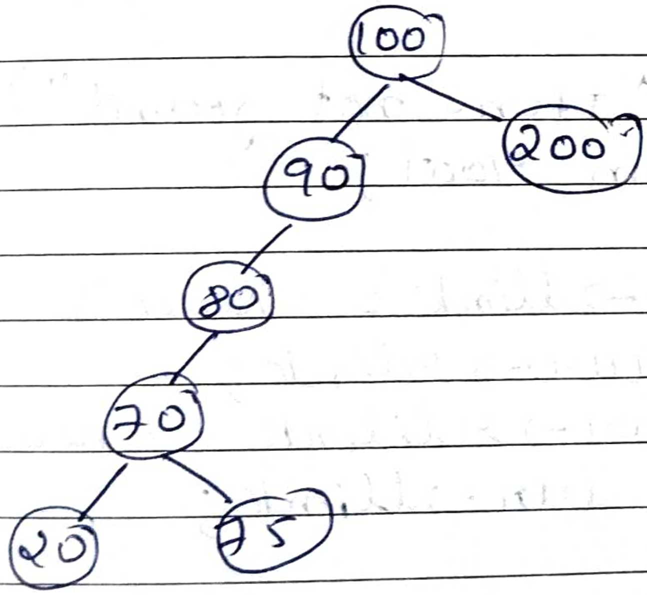


3



Delete 50.

11



```
NODE delete(int item, NODE root)
{
  NODE cur, parent, suc, q;
  if (root == NULL)
  {
    printf("tree empty");
    return root;
  }
  parent = NULL;
  cur = root;
  while (cur != NULL)
  {
    if (item == cur->info)
       break;
    parent = cur;
    if (item < cur->info)
       cur = cur->llink;
    else
       cur = cur->rlink;
  }
  if (cur == NULL)
  {
    printf("Item not found");
    return root;
  }
   if (cur->llink == NULL)
      q = cur->rlink;
  else if (cur->rlilink == NULL)
      q = cur->llink;
  else
  {
     suc = cur->rlink;
while (suc->llink != NULL)
        suc = suc->llink;
    suc->llink = cur->llink;
      q = cur->rlink;
}
  if (parent == NULL)
   return q;

 if (cur == parent->llink)
      parent->llink = q;

 else
      parent->rlink = q;
   free (cur);
   return root;
}
```

other operations on BST

① To find maximum value in a BST

```
NODE max(NODE root)
{
    NODE cur;
    if (root == NULL)
     return root;
     cur = root;
    while (cur->rlink != NULL)
       cur = cur->rlink;
    return cur;
}
```

② To find minimum value ↑ same change
[rlink to llink]

③ To find height of the tree.

$$\text{Height(root)} = \begin{cases} -1 \\ 1 + \text{max}(\text{height(root} \to \text{llink)}, \text{height(root} \to \text{rlink))} \end{cases}$$

```
int height (NODE root)
{
  if (root == NULL)
    return -1;
  else
  return (1 + max (height (root -> llink),
                   height (root -> rlink)));
}

int max (int a, int b)
{
    return (a > b) ? a : b;
}
```

④ To count the nodes in a tree

```
void count ( NODE root)
{
    if (root == NULL)
     return;
    count (root -> llink);
    count ++;
    count (root -> rlink);
}
```

(5) To count the leaf nodes in BST

```
void countleaf (NODE root)
{
    if (root = = NULL)
    return;
    countleaf (root -> llink);
   if (root -> llink = = NULL  &&
        root -> rlink = = NULL)
    count ++;
    countleaf (root -> rlink);
}
```

Selection tree

Selection tree is a tree used to select a winner in a knockout tournament.
Leaves of the tree represent n players entering the tournament and each internal node represents a winner of a match.

2 types
- → winner tree
- → Loser tree.

(1) winner tree :- It is a complete binary tree with n leaf nodes and n-1 internal nodes. Each internal node records the winner of the match.

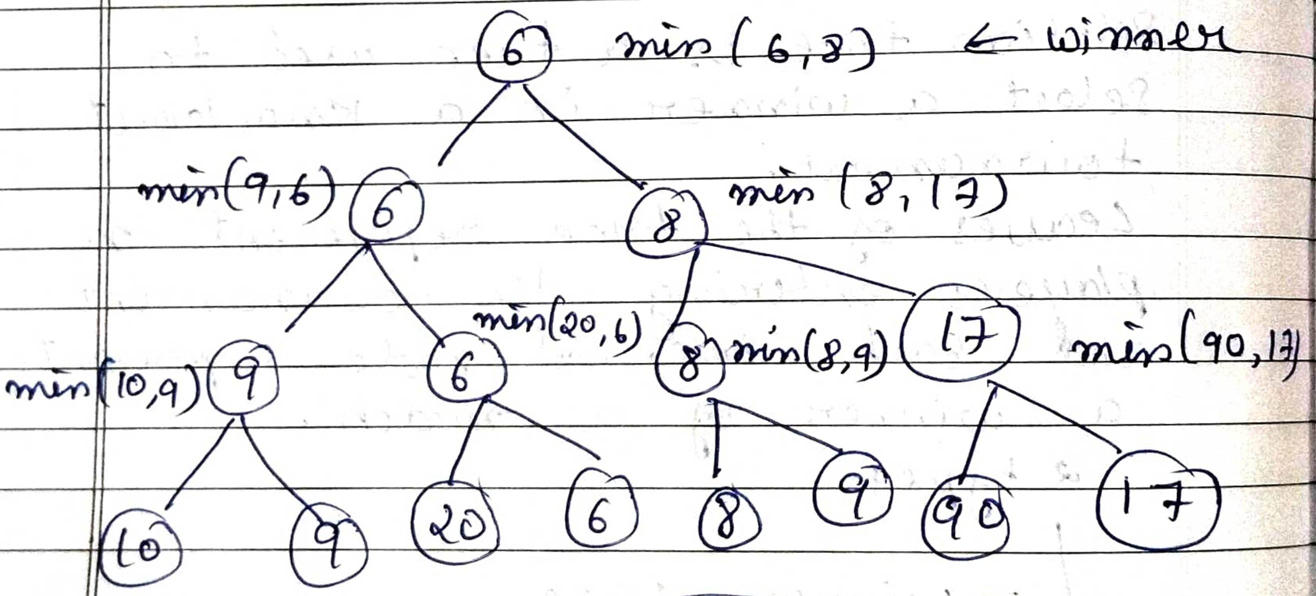
It can be either min winner tree or max winner tree.

→ In min winner. Each node represents the smaller of its two children.

→ In max winner Each node represents the larger of its 2 children

Ex Score values
10, 9, 20, 6, 8, 9, 90, 17

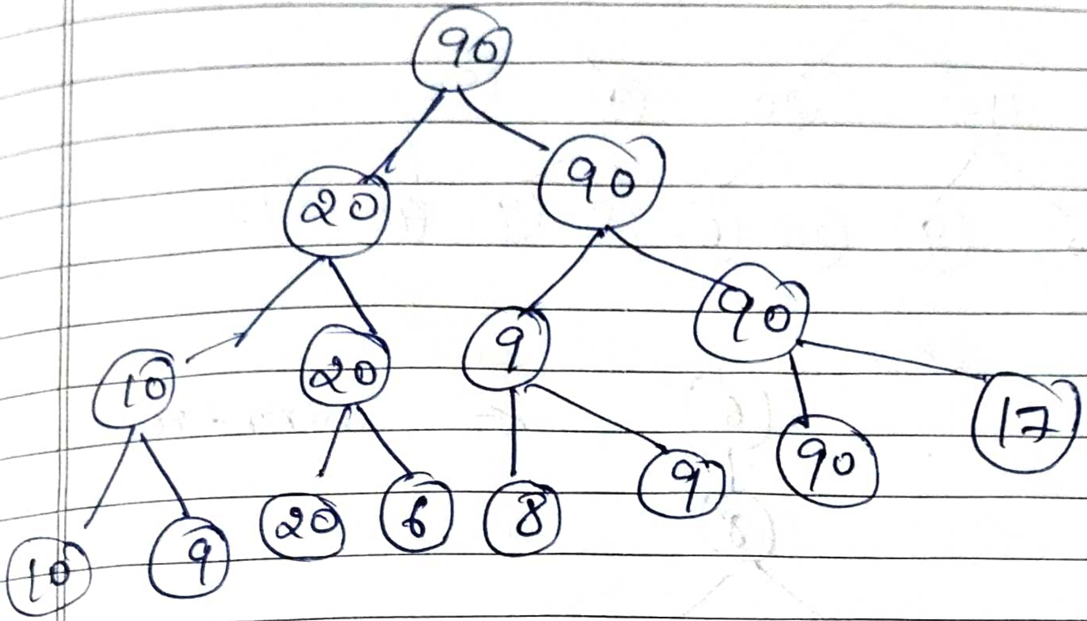
Min tree



8 players with Scores

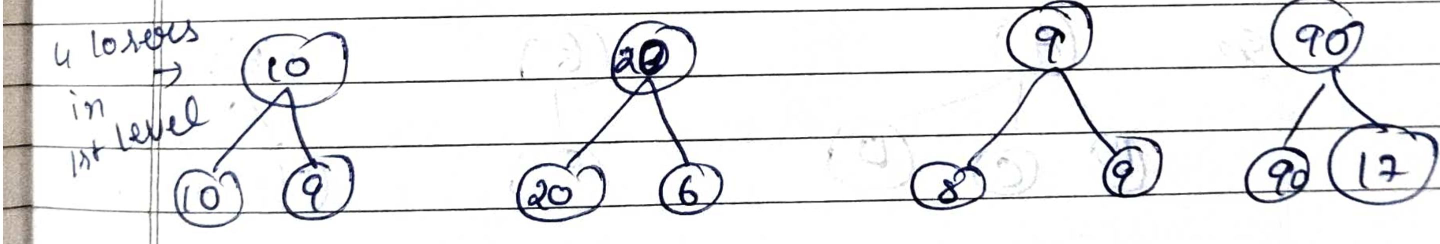
Player with min Score wins.

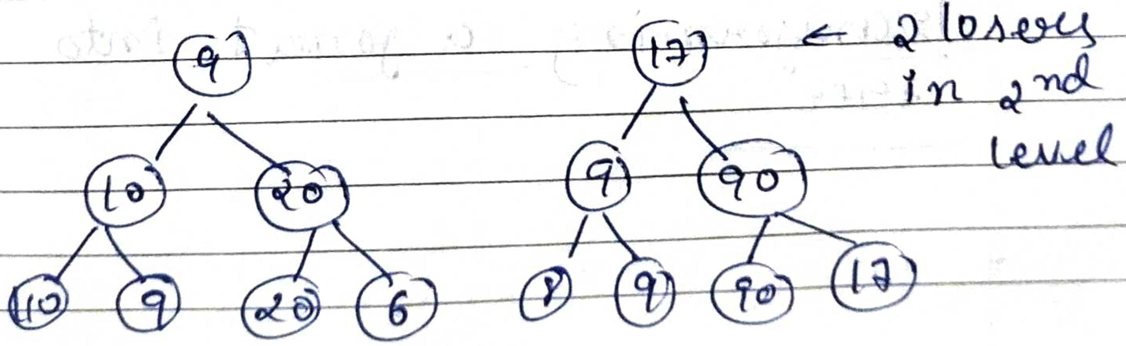
Max tree

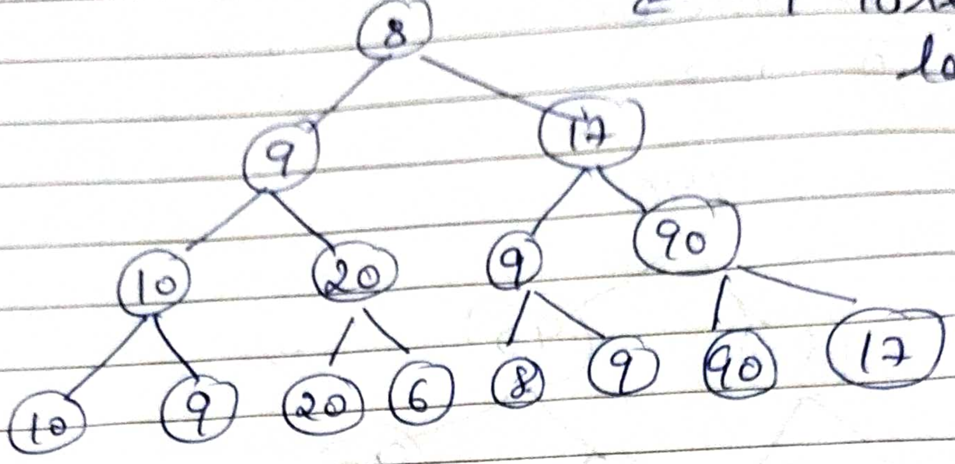


Loser tree

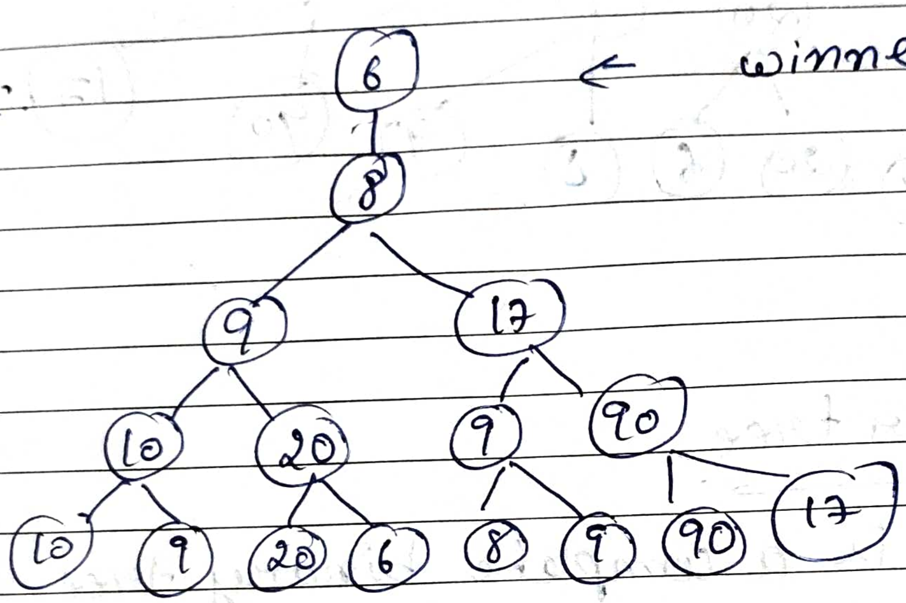
It is a complete binary tree with n leaf & n-1 internal nodes. Each internal node records the loser of the match. The final player who has not lost any match is the winner of the tournament.







← 1 loser in last level

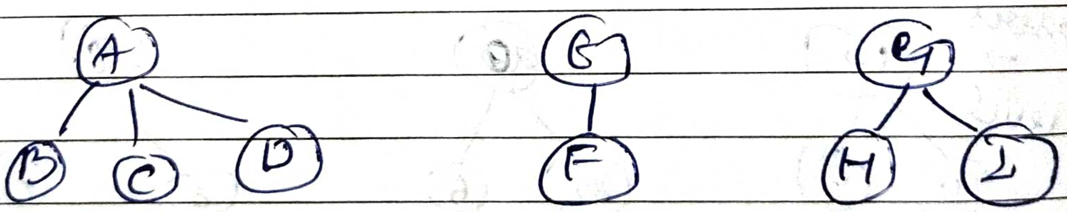


← winner

Forest

The collection of zero or more trees is called forest.

Eg



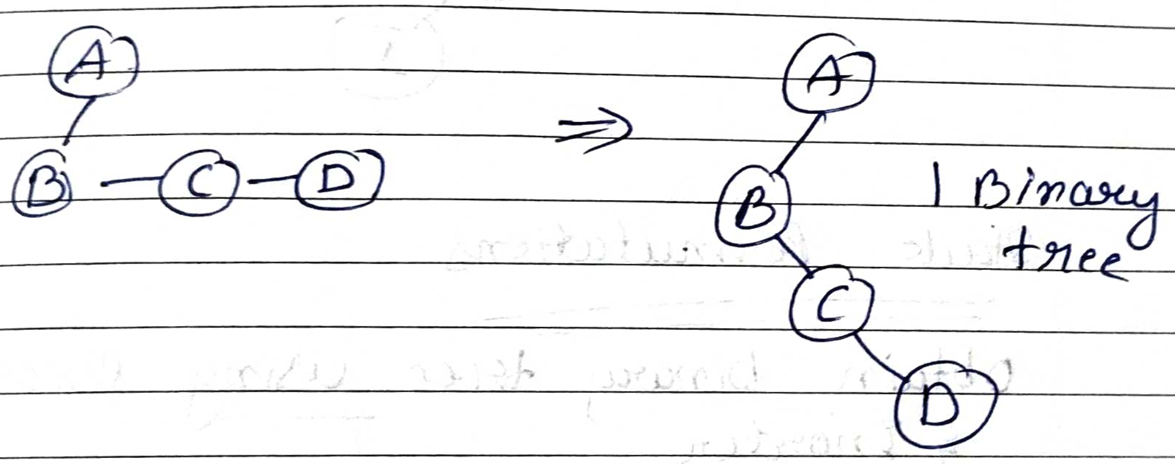
Transforming a forest into binary tree

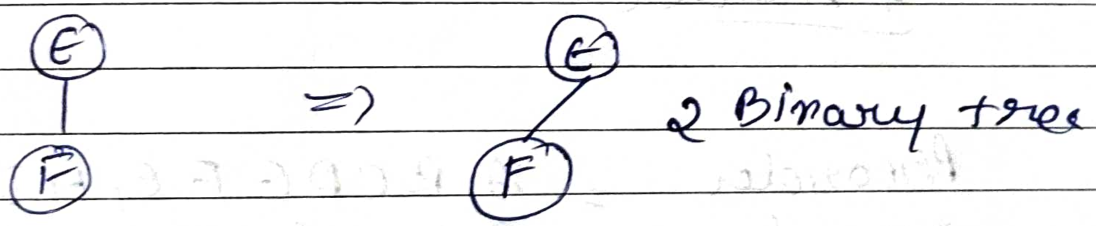
① convert each tree in the forest into binary tree (using left-child right sibling).

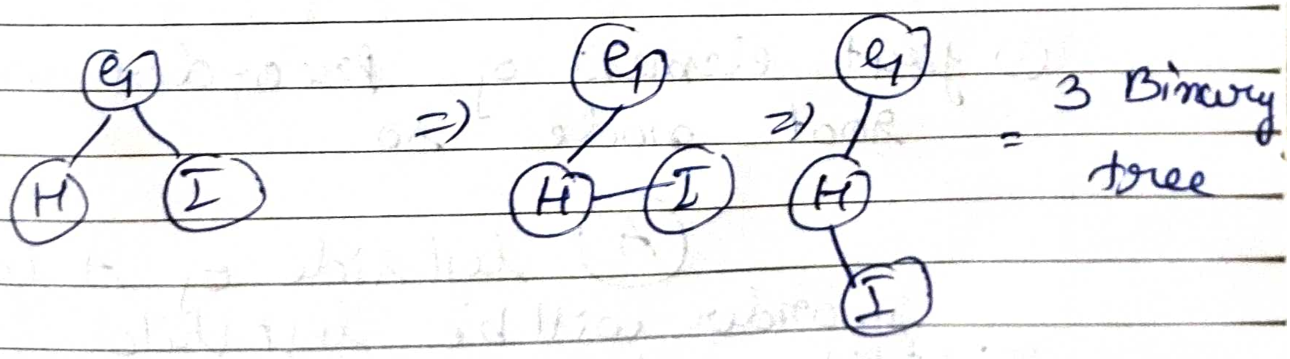
② The root node of the first binary tree is the root for entire tree

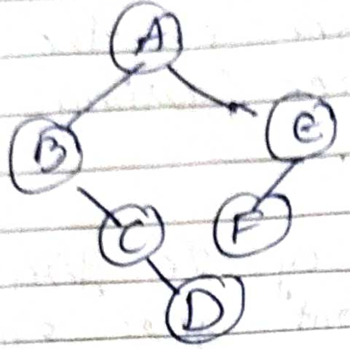
③ Root of second binary tree is attached as right child of root of first binary tree.

④ root of third binary tree is attached as right child of root of second binary tree & so on.

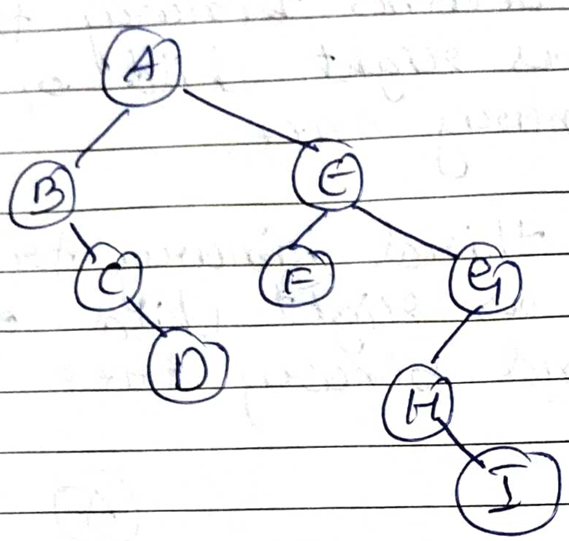








attach 2nd to right of A.



attach 3rd to right of E.

Stack Permutations

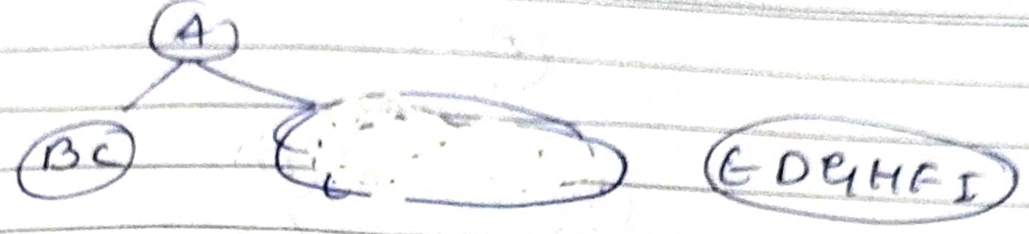
obtain binary tree using preorder & Inorder

Preorder = A B C D E F G H I
Inorder = B C A E D G H F I

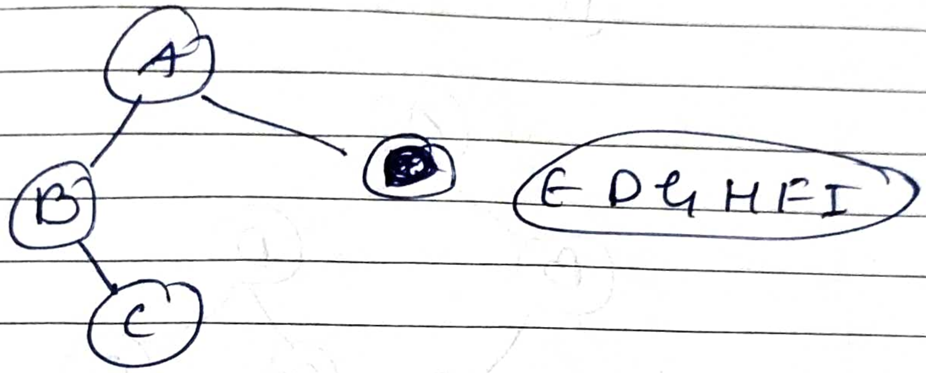
① first element of preorder will be root node so.

A left side of A in Inorder will be left child
right side of A in inorder will be right child

∴

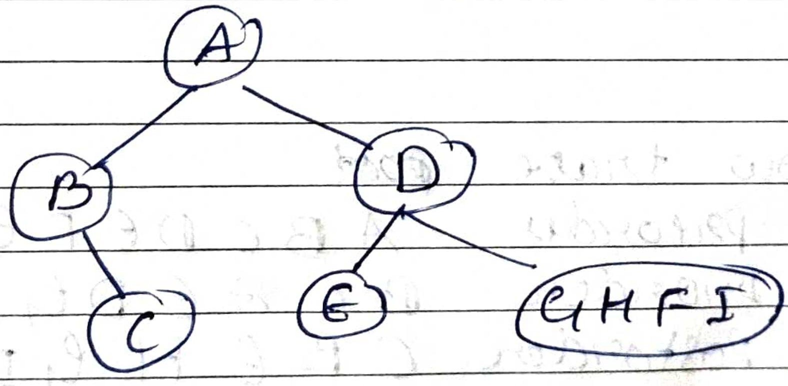


Now see in preorder in BC which comes first so (B) so be will be root & C comes after B in inorder so C will be right child of (B)



∴ Inorder figure E DGC is already [illegible]

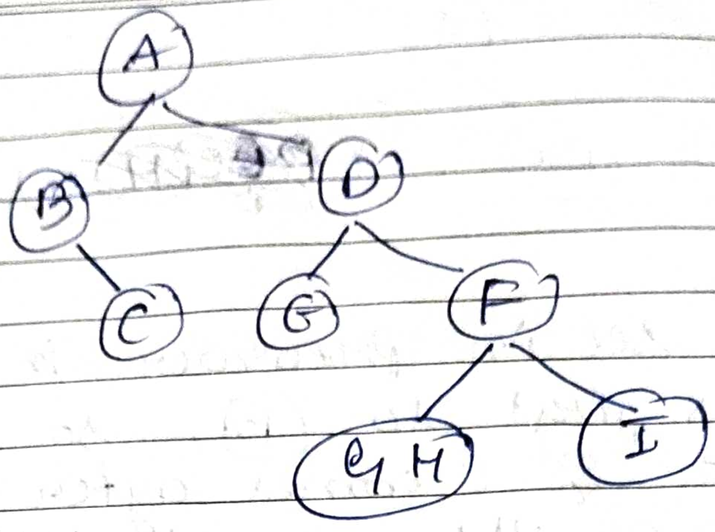
⇒ In(EDGHFI) you should see which comes first in preorder so D



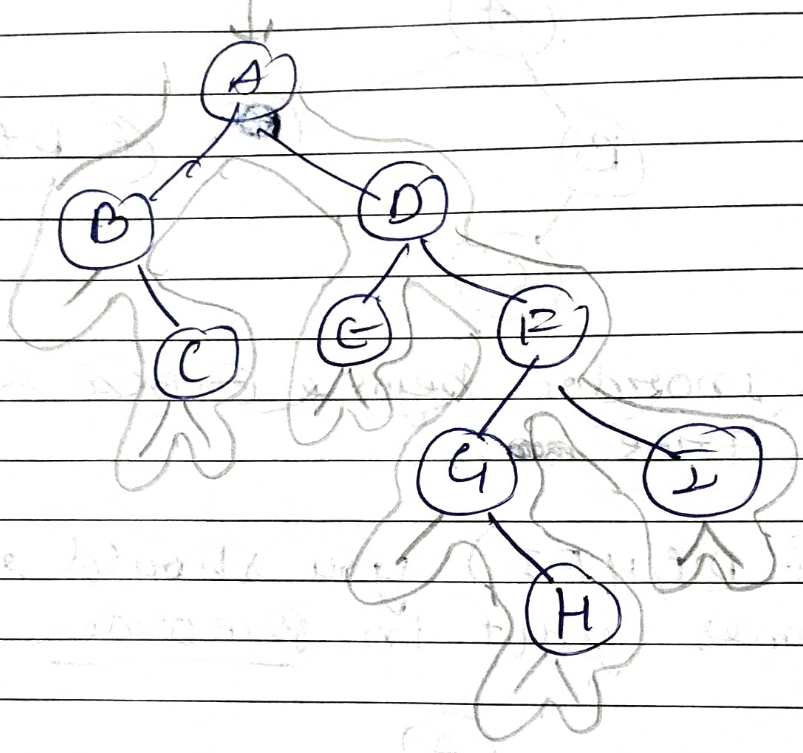
⇒ left of D in Inorder is E & right GHFI

Now in GHFI ⇒ F comes 1st so F becomes root.

hild.



ii,



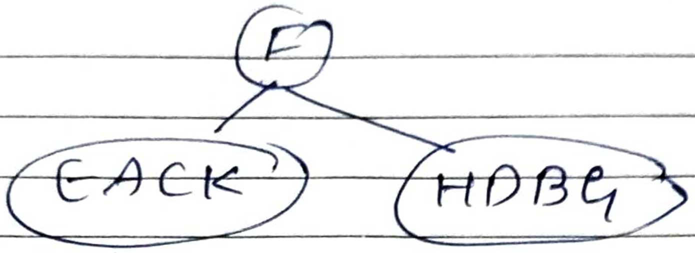
Now trase ~~post~~

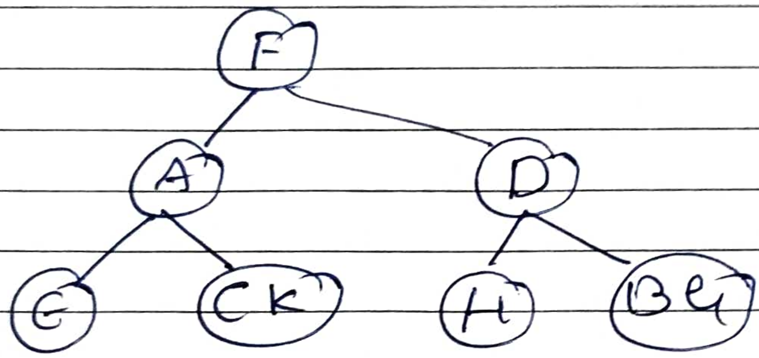
Preorder A B C D E F G H I

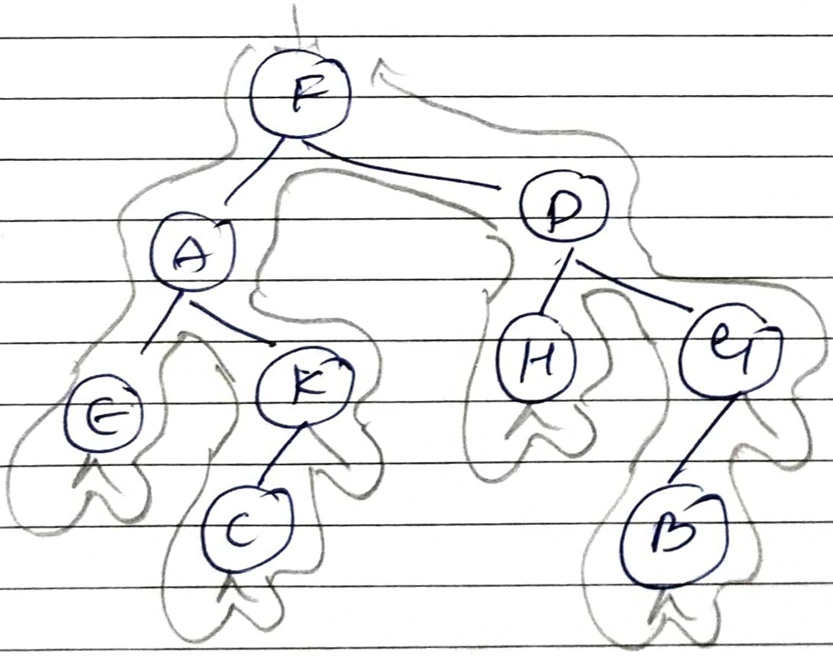
Inorder B C A E D G H F I

Postorder C B E H G I F D A

Inorder = EACKFHDBG
Preorder = FAEKCDHGB







Preorder FAEKCDHGB
Inorder EACKFHDBG
Postorder ECKAHBGDF

GRAPHS

The Graph ADT Introduction

Definition

Graph representation

Elementary graph operations BFS, DFS

Introduction to Graphs

Graph is a non linear data structure; A map is a well-known example of a graph. In a map various connections are made between the cities. The cities are connected via roads, railway lines and aerial network. We can assume that the graph is the interconnection of cities by roads. Euler used graph theory to solve Seven Bridges of Königsberg problem. Is there a possible way to traverse every bridge exactly once – Euler Tour

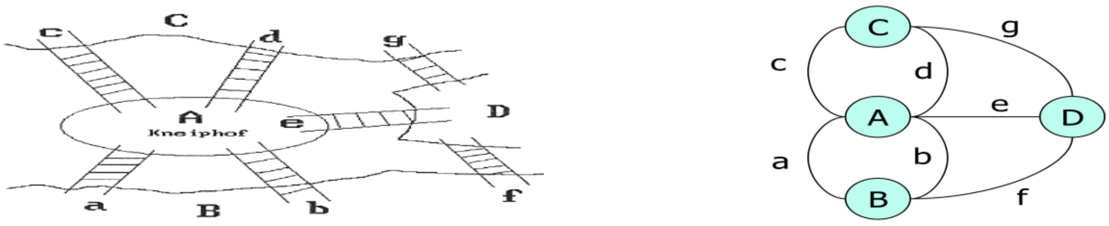


Figure: Section of the river Pregal in Koenigsberg and Euler's graph.

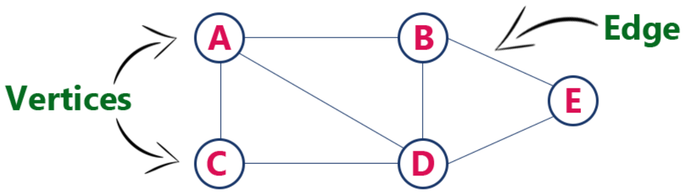
Defining the degree of a vertex to be the number of edges incident to it, Euler showed that there is a walk starting at any vertex, going through each edge exactly once and terminating at the start vertex iff the degree of each, vertex is even. A walk which does this is called Eulerian. There is no Eulerian walk for the Koenigsberg bridge problem as all four vertices are of odd degree.

A graph contains a set of points known as nodes (or vertices) and set of links known as edges (or Arcs) which connects the vertices.

A graph is defined as Graph is a collection of vertices and arcs which connects vertices in the graph. A graph G is represented as G = (V , E), where V is set of vertices and E is set of edges.

Example: graph G can be defined as G = (V , E) Where V = {A,B,C,D,E} and

E = {(A,B),(A,C)(A,D),(B,D),(C,D),(B,E),(E,D)}. This is a graph with 5 vertices and 6 edges.



Graph Terminology

1.**Vertex** : An individual data element of a graph is called as Vertex. Vertex is also known as node. In above example graph, A, B, C, D & E are known as vertices.

2.**Edge** : An edge is a connecting link between two vertices. Edge is also known as Arc. An edge is represented as (starting Vertex, ending Vertex).

In above graph, the link between vertices A and B is represented as (A,B).

Edges are three types:

1.Undirected Edge - An undirected edge is a bidirectional edge. If there is an undirected edge between vertices A and B then edge (A , B) is equal to edge (B , A).

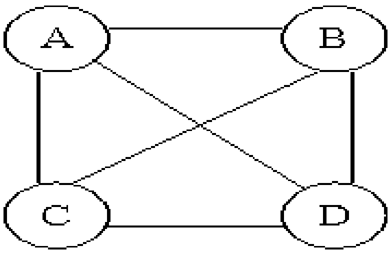
2.Directed Edge - A directed edge is a unidirectional edge. If there is a directed edge between vertices A and B then edge (A , B) is not equal to edge (B , A).

3.Weighted Edge - A weighted edge is an edge with cost on it.

Types of Graphs

1.Undirected Graph

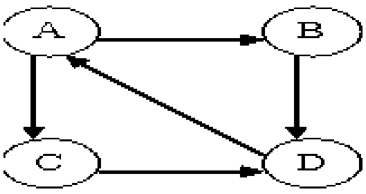
A graph with only undirected edges is said to be undirected graph.



Undirected Graph.

2.Directed Graph

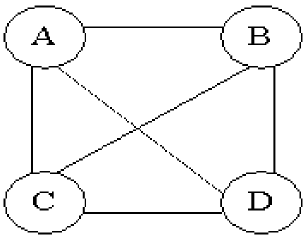
A graph with only directed edges is said to be directed graph.



Directed Graph.

3.Complete Graph

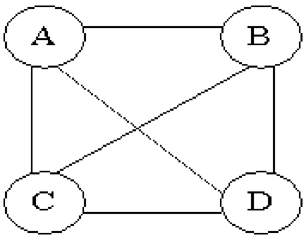
A graph in which any V node is adjacent to all other nodes present in the graph is known as a complete graph. An undirected graph contains the edges that are equal to edges = n(n-1)/2 where n is the number of vertices present in the graph. The following figure shows a complete graph.



A complete graph.

4.Regular Graph

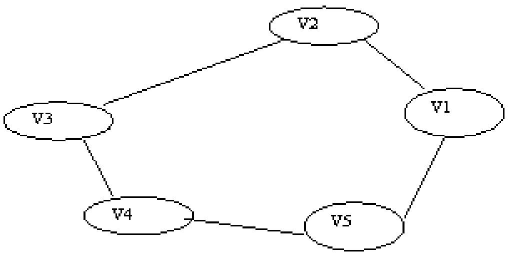
Regular graph is the graph in which nodes are adjacent to each other, i.e., each node is accessible from any other node.



A regular graph

5.Cycle Graph

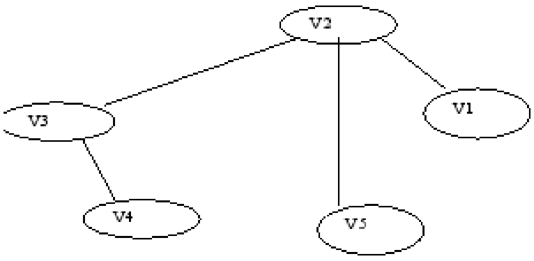
A graph having cycle is called cycle graph. In this case the first and last nodes are the same. A closed simple path is a cycle.



A cycle graph

6.Acyclic Graph

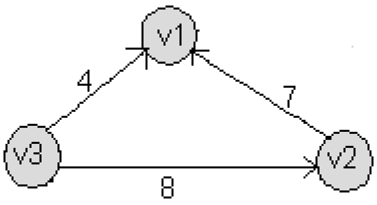
A graph without cycle is called acyclic graphs.



A acyclic graph

7. Weighted Graph

A graph is said to be weighted if there are some non negative value assigned to each edges of the graph. The value is equal to the length between two vertices. Weighted graph is also called a network.



A weighted graph

Outgoing Edge

A directed edge is said to be outgoing edge on its orign vertex.

Incoming Edge

A directed edge is said to be incoming edge on its destination vertex.

Degree

Total number of edges connected to a vertex is said to be degree of that vertex.

Indegree

Total number of incoming edges connected to a vertex is said to be indegree of that vertex.

Outdegree

Total number of outgoing edges connected to a vertex is said to be outdegree of that vertex.

Parallel edges or Multiple edges

If there are two undirected edges to have the same end vertices, and for two directed edges to have the same origin and the same destination. Such edges are called parallel edges or multiple edges.

Self-loop

An edge (undirected or directed) is a self-loop if its two endpoints coincide.

Simple Graph

A graph is said to be simple if there are no parallel and self-loop edges.

Adjacent nodes

When there is an edge from one node to another then these nodes are called adjacent nodes.

Incidence

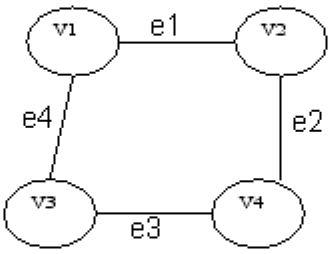
In an undirected graph the edge between v1 and v2 is incident on node v1 and v2.

Walk

A walk is defined as a finite alternating sequence of vertices and edges, beginning and ending with vertices, such that each edge is incident with the vertices preceding and following it.

Closed walk

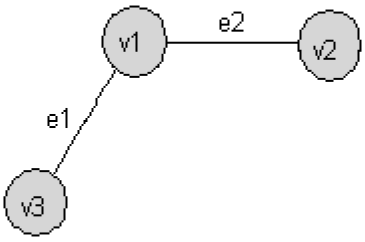
A walk which is to begin and end at the same vertex is called close walk. Otherwise it is an open walk.



If e1,e2,e3,and e4 be the edges of pair of vertices (v1,v2),(v2,v4),(v4,v3) and (v3,v1) respectively ,then v1 e1 v2 e2 v4 e3 v3 e4 v1 be its closed walk or circuit.

Path

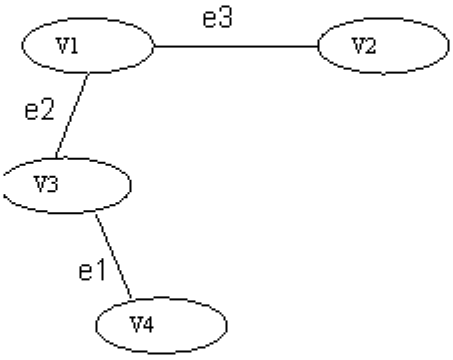
A open walk in which no vertex appears more than once is called a path.



If e1 and e2 be the two edges between the pair of vertices (v1,v3) and (v1,v2) respectively, then v3 e1 v1 e2 v2 be its path.

Length of a path

The number of edges in a path is called the length of that path. In the following, the length of the path is 3.

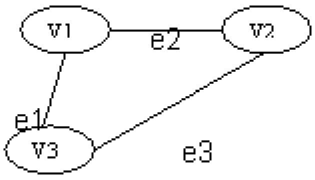


An open walk Graph

Circuit

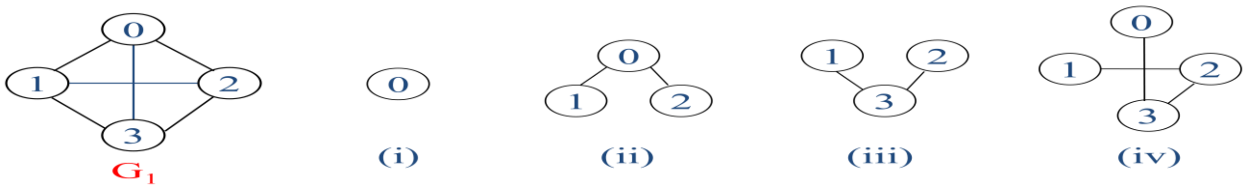
A closed walk in which no vertex (except the initial and the final vertex) appears more than once is called a circuit.

A circuit having three vertices and three edges.



Sub Graph

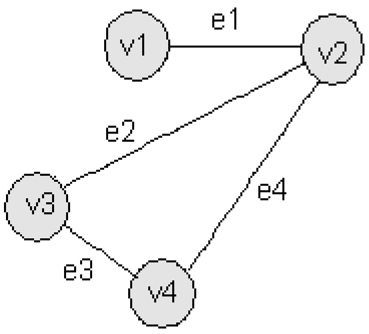
A graph S is said to be a sub graph of a graph G if all the vertices and all the edges of S are in G, and each edge of S has the same end vertices in S as in G. A subgraph of G is a graph G’ such that V(G’) ⊆ V(G) and E(G’) ⊆ E(G)

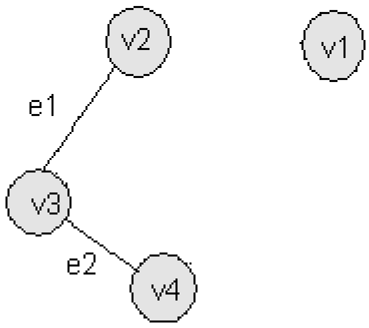


G_1 (i) (ii) (iii) (iv)

Connected Graph

A graph G is said to be connected if there is at least one path between every pair of vertices in G. Otherwise,G is disconnected.





A connected graph G

A disconnected graph G

This graph is disconnected because the vertex v1 is not connected with the other vertices of the graph.

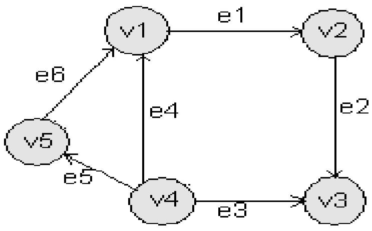
Degree

In an undirected graph, the number of edges connected to a node is called the degree of that node or the degree of a node is the number of edges incident on it.

In the above graph, degree of vertex v1 is 1, degree of vertex v2 is 3, degree of v3 and v4 is 2 in a connected graph.

Indegree

The indegree of a node is the number of edges connecting to that node or in other words edges incident to it.



In the above graph,the indegree of vertices v1, v3 is 2, indegree of vertices v2, v5 is 1 and indegree of v4 is zero.

Outdegree

The outdegree of a node (or vertex) is the number of edges going outside from that node or in other words the

ADT of Graph:

Structure Graph is

objects: a nonempty set of vertices and a set of undirected edges, where each edge is a pair of vertices

functions: for all *graph* ∈ *Graph*, *v*, v_1 and v_2 ∈ *Vertices*

Graph Create()::=return an empty graph

Graph InsertVertex(*graph*, *v*)::= return a graph with *v* inserted. *v* has no edge.

Graph InsertEdge(*graph*, *v1*,*v2*)::= return a graph with new edge between *v1* and *v2*

Graph DeleteVertex(*graph*, *v*)::= return a graph in which *v* and all edges incident to it are removed

Graph DeleteEdge(*graph*, *v1*, *v2*)::=return a graph in which the edge (*v1*, *v2*) is removed

Boolean IsEmpty(*graph*)::= if (*graph*==*empty graph*) return TRUE else return FALSE

List Adjacent(*graph*,*v*)::= return a list of all vertices that are adjacent to *v*

Graph Representations

Graph data structure is represented using following representations

1. **Adjacency Matrix**
2. **Adjacency List**
3. **Adjacency Multilists**

1.Adjacency Matrix

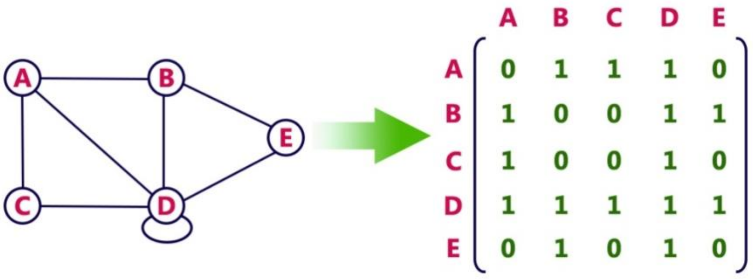
In this representation, graph can be represented using a matrix of size total number of vertices by total number of vertices; means if a graph with 4 vertices can be represented using a matrix of 4X4 size.

In this matrix, rows and columns both represent vertices.

This matrix is filled with either 1 or 0. Here, 1 represents there is an edge from row vertex to column vertex and 0 represents there is no edge from row vertex to column vertex.

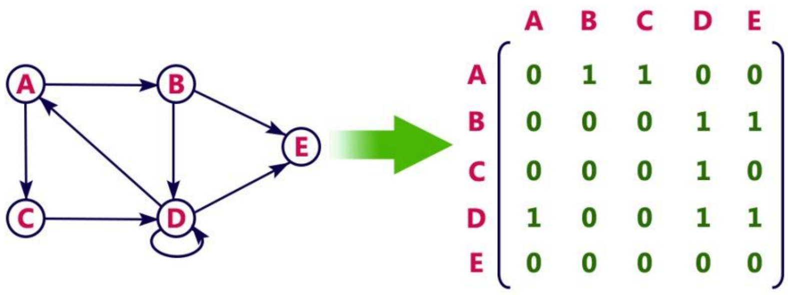
Adjacency Matrix : let G = (V, E) with n vertices, n ≥ 1. The adjacency matrix of G is a 2-dimensional n × n matrix, A, A(i, j) = 1 iff $(v_i, v_j) \in E(G)$ ($\langle v_i, v_j \rangle$ for a diagraph), A(i, j) = 0 otherwise.

example :   for undirected graph



$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 1 & 1 & 0 \\ B & 1 & 0 & 0 & 1 & 1 \\ C & 1 & 0 & 0 & 1 & 0 \\ D & 1 & 1 & 1 & 1 & 1 \\ E & 0 & 1 & 0 & 1 & 0 \end{array}$$

For a Directed graph



The adjacency matrix for an undirected graph is symmetric; the adjacency matrix for a digraph need not be symmetric.

Merits of Adjacency Matrix:

From the adjacency matrix, to determine the connection of vertices is easy

The degree of a vertex is $\sum_{j=0}^{n-1} adj\_mat[i][j]$

For a digraph, the row sum is the out_degree, while the column sum is the in_degree

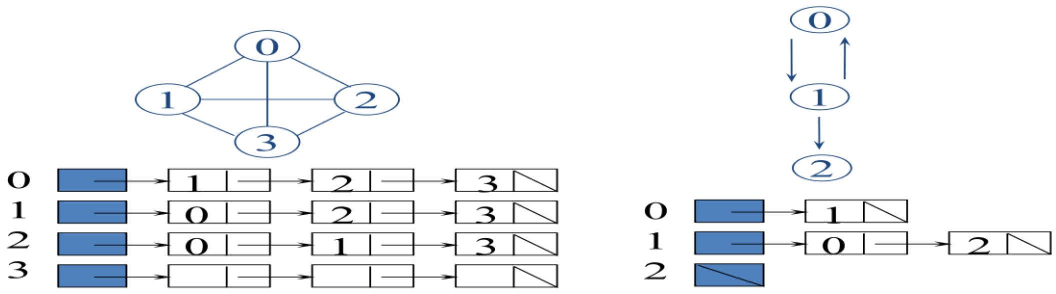
$$ind(vi) = \sum_{j=0}^{n-1} A[j,i] \qquad outd(vi) = \sum_{j=0}^{n-1} A[i,j]$$

The space needed to represent a graph using adjacency matrix is n^2 bits. To identify the edges in a graph, adjacency matrices will require at least $O(n^2)$ time.

2. Adjacency List

In this representation, every vertex of graph contains list of its adjacent vertices. The n rows of the adjacency matrix are represented as n chains. The nodes in chain I represent the vertices that are adjacent to vertex i.

It can be represented in two forms. In one form, array is used to store n vertices and chain is used to store its adjacencies. Example:

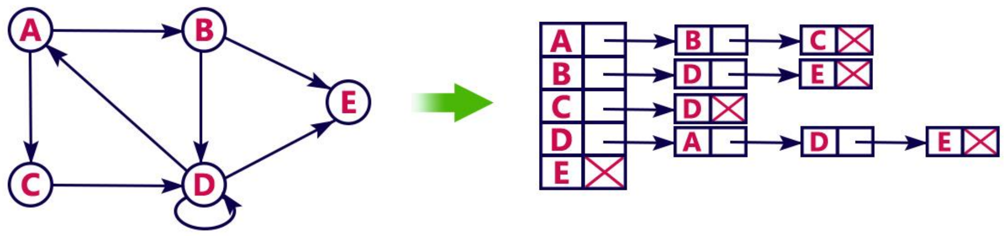


So that we can access the adjacency list for any vertex in O(1) time. Adjlist[i] is a pointer to to first node in the adjacency list for vertex i. Structure is

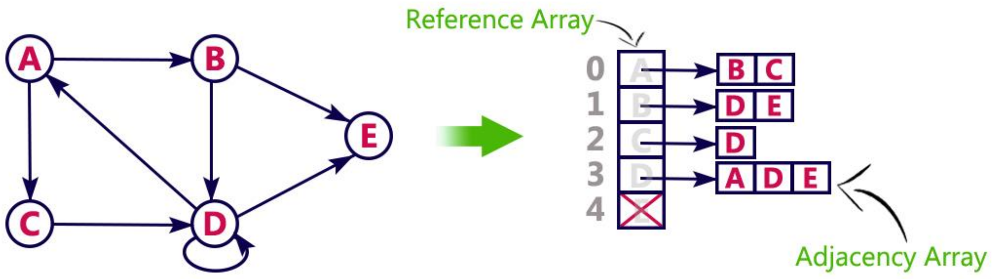
```
#define MAX_VERTICES 50
typedef struct node *node_pointer;
typedef struct node {
   int vertex;
   struct node *link;
};
node_pointer graph[MAX_VERTICES];
int n=0; /* vertices currently in use */
```

Another type of representation is given below.

example: consider the following directed graph representation implemented using linked list

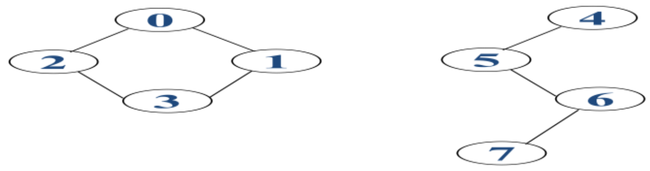


This representation can also be implemented using array



Sequential representation of adjacency list is

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
9	11	13	15	17	18	20	22	23	2	1	3	0	0	3	1	2	5	6	4	5	7	6

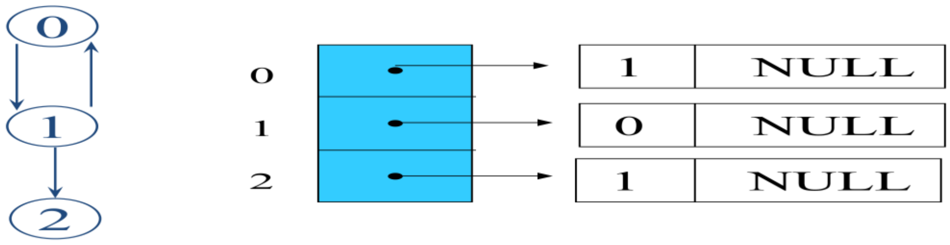


Graph

Instead of chains, we can use sequential representation into an integer array with size n+2e+1. For 0<=i<n, Array[i] gives starting point of the list for vertex I, and array[n] is set to n+2e+1. The adjacent vertices of node I are stored sequentially from array[i].

For an undirected graph with n vertices and e edges, linked adjacency list requires an array of size n and 2e chain nodes. For a directed graph, the number of list nodes is only e. the out degree of any vertex may be determined by counting the number of nodes in its adjacency list. To find in-degree of vertex v, we have to traverse complete list.

To avoid this, inverse adjacency list is used which contain in-degree.



Determine in-degree of a vertex in a fast way.

3.Adjacency Multilists

In the adjacency-list representation of an undirected graph each edge (u, v) is represented by two entries one on the list for u and the other on tht list for v. As we shall see in some situations it is necessary to be able to determin ie ~ nd enty for a particular edge and mark that edg as having been examined. This can be accomplished easily if the adjacency lists are actually maintained as multilists (i.e., lists in which nodes may be shared among several lists). For each edge there will be exactly one node but this node will be in two lists (i.e. the adjacency lists for each of the two nodes to which it is incident).

For adjacency multilists, node structure is

```
typedef struct edge *edge_pointer;
typedef struct edge {
   short int marked;
   int vertex1, vertex2;
   edge_pointer path1, path2;
};
edge_pointer graph[MAX_VERTICES];
```

marked	vertex1	vertex2	path1	path2

Lists: vertex 0: N0->N1->N2, vertex 1: N0->N3->N4
vertex 2: N1->N3->N5, vertex 3: N2->N4->N5

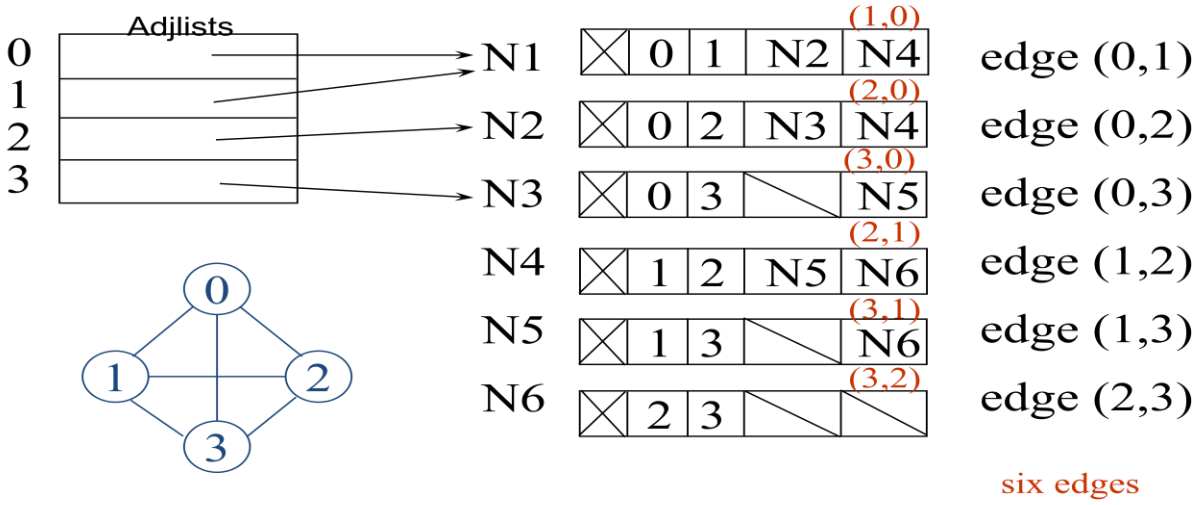
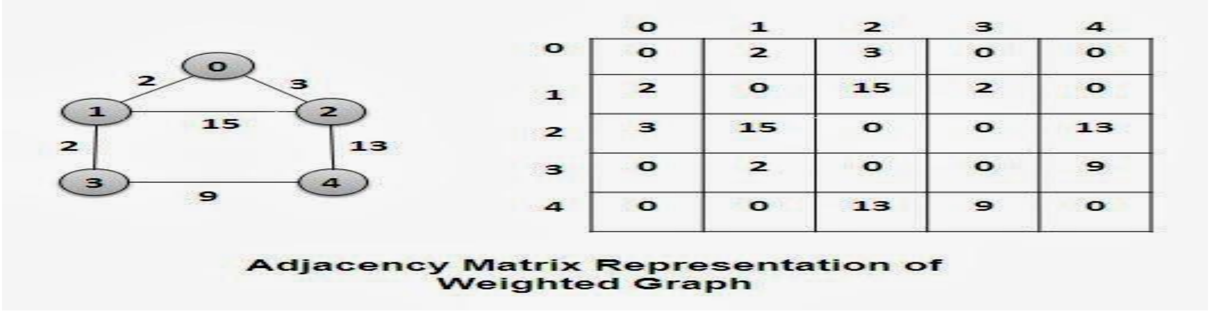


Figure: Adjacency multilists for given graph

4. Weighted edges

In many applications the edges of a graph have weights assigned to them. These weights may represent the distance from one vertex to another or the cost of going from one; vertex to an adjacent vertex In these applications the adjacency matrix entries A [i][j] would keep this information too. When adjacency lists are used the weight information may be kept in the list'nodes by including an additional field weight. A graph with weighted edges is called a network.



	0	1	2	3	4
0	0	2	3	0	0
1	2	0	15	2	0
2	3	15	0	0	13
3	0	2	0	0	9
4	0	0	13	9	0

Adjacency Matrix Representation of Weighted Graph

ELEMENTARY GRAPH OPERATIONS

Given a graph G = (V E) and a vertex v in V(G) we wish to visit all vertices in G that are reachable from v (i.e., all vertices that are connected to v). We shall look at two ways of doing this: depth-first search and breadth-first search. Although these methods work on both directed and undirected graphs the following discussion assumes that the graphs are undirected.

Depth-First Search

- Begin the search by visiting the start vertex v
  - If v has an unvisited neighbor, traverse it recursively
  - Otherwise, backtrack
- Time complexity
  - Adjacency list: O(|E|)
  - Adjacency matrix: O($|V|^2$)

We begin by visiting the start vertex v. Next an unvisited vertex w adjacent to v is selected, and a depth-first search from w is initiated. When a vertex u is reached such that all its adjacent vertices have been visited, we back up to the last vertex visited that has an unvisited vertex w adjacent to it and initiate a depth-first search from w. The search terminates when no unvisited vertex can be reached from any of the visited vertices.

DFS traversal of a graph, produces a **spanning tree** as final result. **Spanning Tree** is a graph without any loops. We use **Stack data structure** with maximum size of total number of vertices in the graph to implement DFS traversal of a graph.

We use the following steps to implement DFS traversal...

Step 1: Define a Stack of size total number of vertices in the graph.

Step 2: Select any vertex as starting point for traversal. Visit that vertex and push it on to the Stack.

Step 3: Visit any one of the adjacent vertex of the verex which is at top of the stack which is not visited and push it on to the stack.

Step 4: Repeat step 3 until there are no new vertex to be visit from the vertex on top of the stack.

Step 5: When there is no new vertex to be visit then use back tracking and pop one vertex from the stack.

Step 6: Repeat steps 3, 4 and 5 until stack becomes Empty.

Step 7: When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

This function is best described recursively as in Program.

```
#define FALSE 0
#define TRUE 1
 int visited[MAX_VERTICES];
void dfs(int v)
{
  node_pointer w;
  visited[v]= TRUE;
  printf("%d", v);
  for (w=graph[v]; w; w=w->link)
    if (!visited[w->vertex])
      dfs(w->vertex);
}
```

Consider the graph G of Figure 6.16(a), which is represented by its adjacency lists as in Figure 6.16(b). If a depth-first search is initiated from vertex 0 then the vertices of G are visited in the following order: **0 1 3 7 4 5 2 6.** Since DFS(O) visits all vertices that can be reached from 0 the vertices visited, together with all edges in G incident to these vertices form a connected component of G.

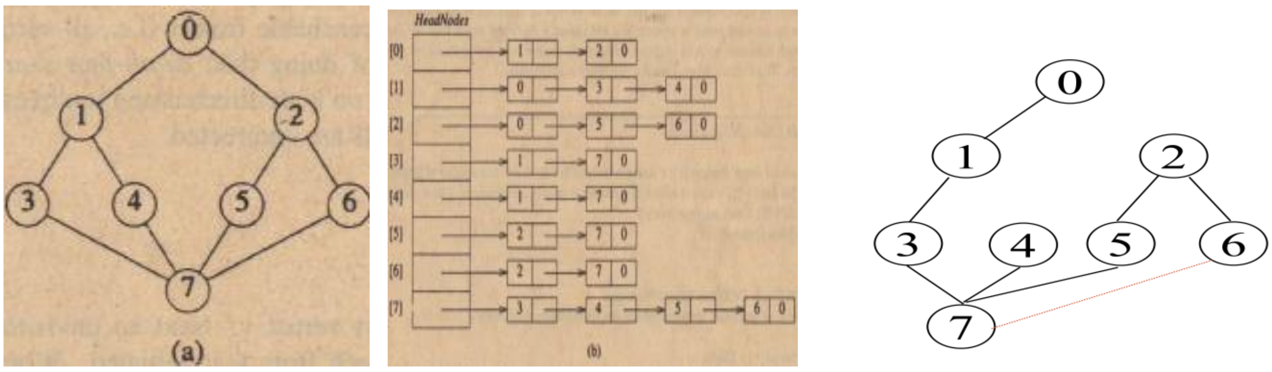


Figure: Graph and its adjacency list representation, DFS spanning tree

Analysis or DFS:
When G is represented by its adjacency lists, the vertices w adjacent to v can be determined by following a chain of links. Since DFS examines each node in the adjacency lists at most once and there are 2e list nodes the time to complete the search is O(e). If G is represented by its adjacency matrix then the time to determine all vertices adjacent to v is O(n). Since at most n vertices are visited the total time is O(n^2).

Breadth-First Search
In a breadth-first search, we begin by visiting the start vertex v. Next all unvisited vertices adjacent to v are visited. Unvisited vertices adjacent to these newly visited vertices are then visited and so on. Algorithm BFS (Program 6.2) gives the details.

```
typedef struct queue *queue_pointer;
typedef struct queue {
    int vertex;
    queue_pointer link;
};
void addq(queue_pointer *,
        queue_pointer *, int);
int deleteq(queue_pointer *);
void bfs(int v)
{
  node_pointer w;
  queue_pointer front, rear;
  front = rear = NULL;
  printf("%d", v);
  visited[v] = TRUE;
  addq(&front, &rear, v);
while (front) {
    v= deleteq(&front);
    for (w=graph[v]; w; w=w->link)
      if (!visited[w->vertex]) {
        printf("%d", w->vertex);
        addq(&front, &rear, w->vertex);
        visited[w->vertex] = TRUE;
      }
  }
}
```

Steps:

BFS traversal of a graph, produces a spanning tree as final result. Spanning Tree is a graph without any loops. We use Queue data structure with maximum size of total number of vertices in the graph to implement BFS traversal of a graph.

We use the following steps to implement BFS traversal...

Step 1: Define a Queue of size total number of vertices in the graph.

Step 2: Select any vertex as starting point for traversal. Visit that vertex and insert it into the Queue.

Step 3: Visit all the adjacent vertices of the vertex which is at front of the Queue which is not visited and insert them into the Queue.

Step 4: When there is no new vertex to be visit from the vertex at front of the Queue then delete that vertex from the Queue.

Step 5: Repeat step 3 and 4 until queue becomes empty.

Step 6: When queue becomes Empty, then produce final spanning tree by removing unused edges from the graph

Analysis Of BFS:

Each visited vertex enters the queue exactly once. So the while loop is iterated at most n times If an adjacency matrix is used the loop takes O(n) time for each vertex visited. The total time is therefore, O(n^2). If adjacency lists are used the loop has a total cost of $d_0 + \ldots + d_{n-1} = O(e)$, where d is the degree of vertex i. As in the case of DFS all visited vertices together with all edges incident to them, form a connected component of G.

3.Connected Components

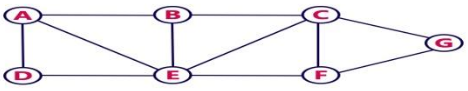
If G is an undirected graph, then one can determine whether or not it is connected by simply making a call to either DFS or BFS and then determining if there is any unvisited vertex. The connected components of a graph may be obtained by making repeated calls to either DFS(v) or BFS(v); where v is a vertex that has not yet been visited. This leads to function Connected(Program 6.3), which determines the connected components of G. The algorithm uses DFS (BFS may be used instead if desired). The computing time is not affected. Function connected –Output outputs all vertices visited in the most recent invocation of DFS together with all edges incident on these vertices.

```
void connected(void){
   for (i=0; i<n; i++) {
       if (!visited[i]) {
           dfs(i);
  printf("\n");        }     } }
```

Analysis of Components:

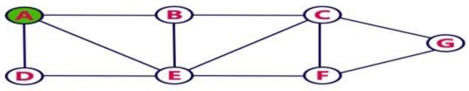
If G is represented by its adjacency lists, then the total time taken by dfs is O(e). Since the for loops take O(n) time, the total time to generate all the Connected components is O(n+e). If adjacency matrices are used,then the time required is O(n^2)

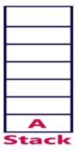
Consider the following example graph to perform DFS traversal



Step 1:

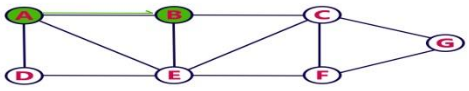
- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.

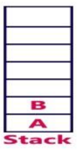




Step 2:

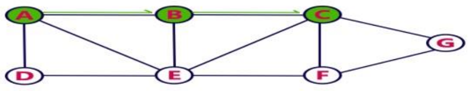
- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.

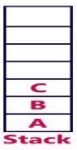




Step 3:

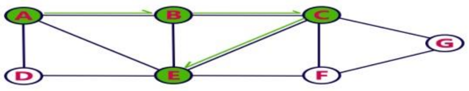
- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.

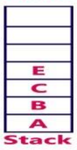




Step 4:

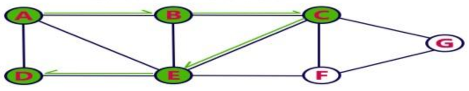
- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack

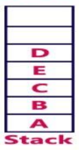




Step 5:

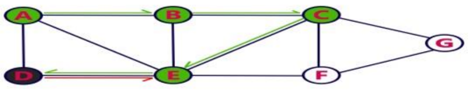
- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack

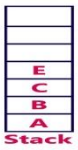




Step 6:

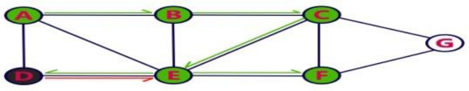
- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.

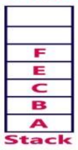




Step 7:

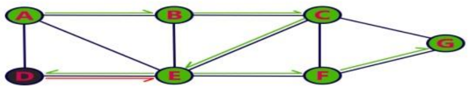
- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.

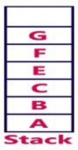




Step 8:

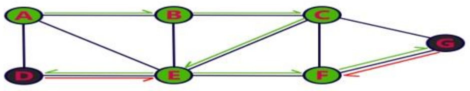
- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.

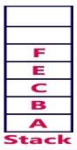




Step 9:

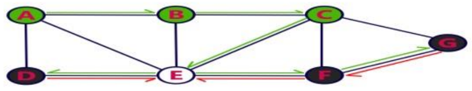
- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.

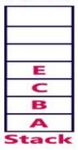




Step 10:

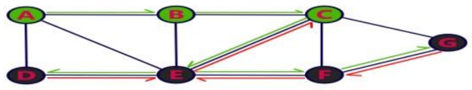
- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.

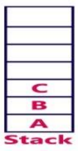




Step 11:

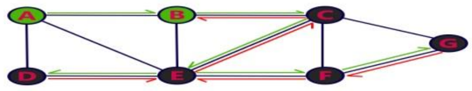
- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.

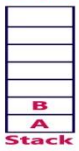




Step 12:

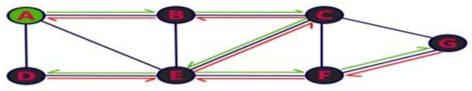
- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.

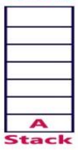




Step 13:

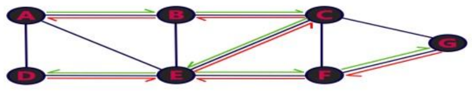
- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.

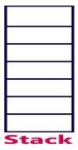




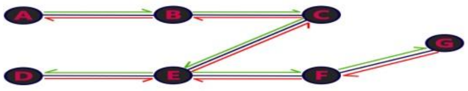
Step 14:

- There is no new vertiex to be visited from A. So use back track.
- Pop A from the Stack.

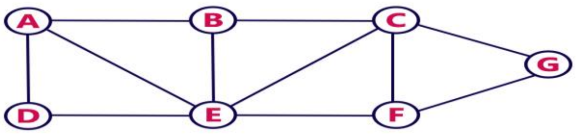




- Stack became Empty. So stop DFS Treversal.
- Final result of DFS traversal is following spanning tree.

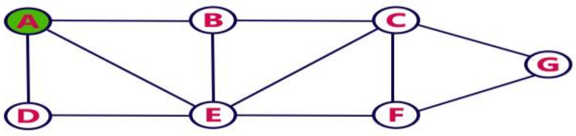


Consider the following example graph to perform BFS traversal



Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.

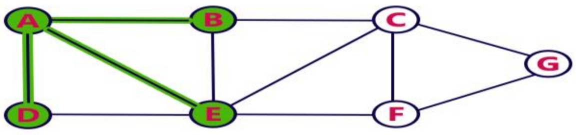


Queue

A						

Step 2:

- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..

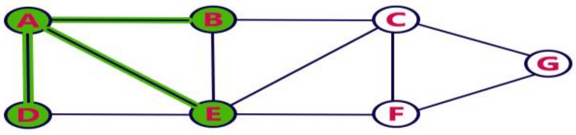


Queue

	D	E	B			

Step 3:

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.

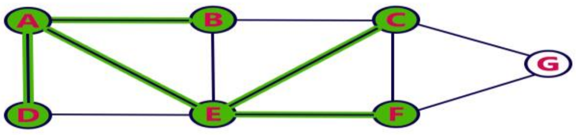


Queue

		E	B			

Step 4:

- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.

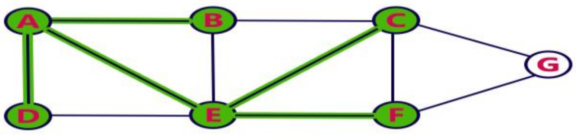


Queue

			B	C	F	

Step 5:

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.

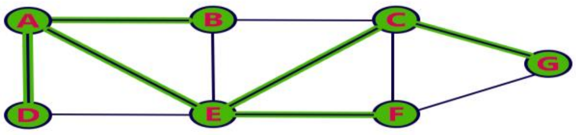


Queue

				C	F	

Step 6:

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.

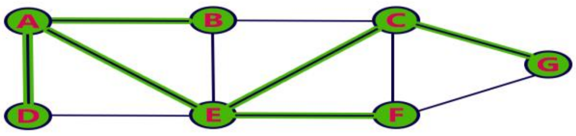


Queue

					F	G

Step 7:

- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.

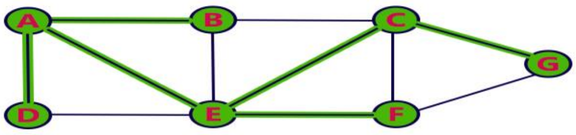


Queue

						G

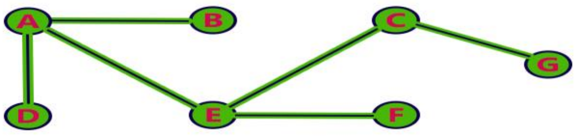
Step 8:

- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



Queue

- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...



http://m.kkhsou.in/EBIDYA/CSC/MODIFY_intro_graph.html

||Jai Sri Gurudev ||
BGSKH Education Trust (R.) – A unit of Sri Adichunchanagiri Shikshana Trust(R.)
BGS College of Engineering and Technology (BGSCET)
Mahalakshmipuram, West of Chord Road, Bengaluru-560086
(Approved by AICTE, New Delhi and Affiliated to VTU, Belagavi)

Question Bank

Subject: Data Structure and Applications **Class:** AIDS

Subject code: BCS304 **Faculty:** Mrs. Jyothi R

Course Outcomes

CO 1. Explain different data structures and their applications.
CO 2. Apply Arrays, Stacks and Queue data structures to solve the given problems.
CO 3. Use the concept of linked list in problem solving.
CO 4. Develop solutions using trees and graphs to model the real-world problem.
CO 5. Explain the advanced Data Structures concepts such as Hashing Techniques and Optimal Binary Search Trees

MODULE 4

Sl. No.	**Questions**	**CO**
1.	Draw a binary serach tree for the following input of elements 40 10 79 90 12 54 11 9 50 and also write a Recursive Search function for Binary Search tree	CO4
2.	Construct a Binary tree by using the following in-order and preorder traversal Inorder: BCAEDGHFI Preorder: ABCDEFGHI	CO4
3.	Write a Recursive Search and Iterative Search function for Binary Search tree	CO4
4.	Write a function to perform Delete from BST and Insert an element into BST and explain with example	CO4
5.	Construct a Binary tree by using the following in-order and preorder traversal Inorder: 42516738 Preorder: 45267831	CO4
6.	Write a function to find **i)** Maximum element in BST **ii)** Minimum element in BST **iii)** Height of BST **iv)** Count the no of nodes in BST **v)** count the no of leaf nodes in BST	CO4
7.	Write short notes on: i) Transforming a Forest into a Binary Tree ii) Forest Traversals. iii) Selection Tress iv) Array and linked representation of binary trees	CO4
8.	Construct a Binary tree by using the following in-order and preorder traversal Inorder: EACKFHDBG Preorder: FAEKCDHGB also perform postorder traversal of the obtained tree	CO4
9.	Draw a binary serach tree for the following input of elements 14, 15, 4, 9, 7, 18, 3, 5, 16, 20 and also perform Inorder, Preorder and Postorder Traversal	CO4

MODULE-5

HASHING: Introduction, Static Hashing, Dynamic Hashing

PRIORITY QUEUES: Single and double ended Priority Queues, Leftist Trees

INTRODUCTION TO EFFICIENT BINARY SEARCH TREES: Optimal Binary Search Trees

HASHING:

We have already seen that the time required to access any element in the array irrespective of its position is same. The physical location of ith item in the array can be calculated by multiplying the size of each element of the array with *i* and adding to the base address as shown below:

$$Loc(A_i) = Base(A) + w*(i – lb)$$

Here,
i - is the index,
w - is the size of each element of the array and
lb - is the lower bound. Using this formula, the address of any item at any specified position *i* can be obtained and from the address obtained, we can access the item. The similar idea is used in *hashing* to store and retrieve the data.So, the hashing technique is essentially independent of n where n is number of elements.

Definition:
A function can be used to provide a mapping between large original data and the smaller table by transforming a key information into an index to the table. The index value returned by this function is called hash value.
The function that transforms a data into hash value to a table is called hash function.

Definition:
The data can be stored in the form of a table using arrays with the help of hash function which gives a hash value as an index to access any element in the table. This table on which insertion, deletion and retrieve operations takes place with the help of hash value is called hash table**.**

A hash table can be implemented as an array ht[0..m-1]. The size of the hash table is limited and so it is necessary to mapthegivendata into this fairly restricted set of integers. The hash function assigns an integer value from 0 to m-1 to keys and these values which act as index to the hash table are called hash addresses or hash values.

What is hashing?
This process of mapping large amounts of data into a smaller table using hash function, hash value and hash table is called hashing.

The two types of hashing techniques are:

1. Static hashing
2. Dynamic hashing

Static hashing :
What is static hashing?

Definition: This process of mapping large amounts of data into a table whose size is fixed during compilation time is called static hashing. In static hashing, all the data items are stored in a fixed size tablet called hash table. The hash table is implemented as an array ht[0..m-1] with 0 as the low index and m – 1 as the high index. Each item in the table can be stored in ht[0], ht[1],…….ht[m – 1] where m is the size of the table usually with a prime number such as 5, 7, 11 and so on. The items are inserted into the table based on the hash value obtained from the hash function.

Hash table

Now, let us take the following example, where identifiers are inserted into the hashtable.

Example : Construct a hash table ht for storing various identifiers.

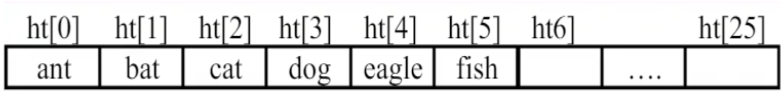
Solution: We need to store identifiers in a hash table.
Let us assume all identifiers starts with only letters from 'A' to 'Z' having the range 0 to 25. Since, there are 26letters, let us have a hash table ht whose size is m = 26. Assume identifiers to be inserted into hash table ht are: ant, dog, cat, bat, eagle, fish and ape. To insert an identifier into the hash table, we require hash function.

Let us define hash function as:
h(x) = toupper(x[0]) – 65 // range is 0 to 25 for'a' to 'z'

where 65 is the ASCII value of 'A'.
Now, let us complete hash value for each of the identifier using the above hash function:

- Hash value of identifier "ant"='A' – 65 = 65 – 65 = 0. So, insert "ant" into ht[0]
- Hash value of identifier "dog"= 'D' – 65 = 68 – 65 = 3. So, insert "dog" into ht[3]
- Hash value of identifier "cat" = 'C' – 65 = 67 – 65 = 2. So, insert "cat" into ht[2]
- Hash value of identifier "bat" = 'B' – 65 = 66 – 65 = 1. So, insert "bat" into ht[1]
- Hash value of identifier "eagle" = 'E' – 65 = 69 – 65 = 4. So, insert "eagle" into ht[4]
- Hash value of identifier "f ish" = 'F' – 65 = 70 – 65 = 5. So, insert "eagle" into ht[5]
- Hash value of identifier "a pe" = 'A' – 65 = 65 – 65 = 0. So, insert "a pe" into ht[0]

Now, the hash table for first six identifiers is shown below:



ht[0]	ht[1]	ht[2]	ht[3]	ht[4]	ht[5]	ht6]		ht[25]
ant	bat	cat	dog	eagle	fish		….	

- The seventh identifier “ape” whose hash value is 0 cannot be inserted into ht[0] because, an identifier is already placed in ht[0]. This condition is called over flow or collision.

- When there is no overflow, the time required to insert, delete or search depends only on the time required to compute the hash function and the time to search on location in ht[i]. Hence, the insert, delete and search times are independent of n which is the number of items.

- Most of the time collision cannot be avoided and in the worst case all keys may have the same hash value. In such situation all the keys may be stored in only one cell in the form of a list and so, it is required to search all n keys in the worst case.

- But, with appropriately chosen size of the hash table and using a good hash function, this phenomenon will not occur and under reasonable assumptions, the expected time to search for an element in a hash table will be O (1).

- So, in practical situation, hashing is extremely effective where insertion, deletion and searching takes place frequently.

The various hashing techniques using which collision can be avoided are:

- Open addressing
- Chaining

OPEN ADDRESSING:

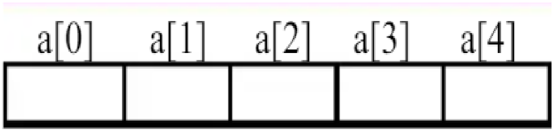
In open addressing hashing, the amount of space available for storing various data is fixed at compile time by declaring a fixed array for the hash table. So, all the keys are stored in this fixed hash table itself without the use of linked lists. In such a table, collision can be avoided by finding another, unoccupied location in the array. The collision can be avoided using linear probing.

Let us create a hash table. To create a hash table, we need the following:

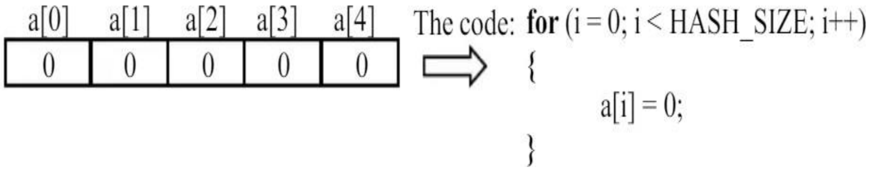
- Initial hash table.
- Select the hashing function.
- Find the index of each location in the hash table.

Create initial hash table:

Assume the size of the hash table is HASH_SIZE = 5 and the hash table can be pictorially represented as shown below:



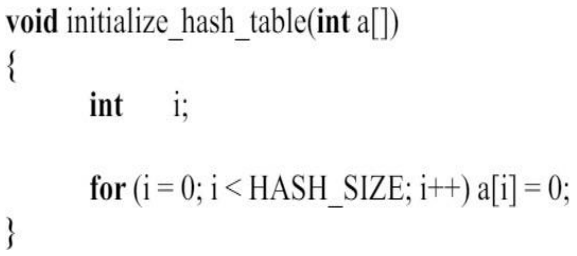
The empty hash table is indicated by storing 0 values in each location as shown below:



The code:
```
for (i = 0; i < HASH_SIZE; i++)
{
        a[i] = 0;
}
```

The function to create initial hash table can be written as shown below:

Create initial hash table



```
void initialize_hash_table(int a[])
{
        int     i;

        for (i = 0; i < HASH_SIZE; i++) a[i] = 0;
}
```

Identify the hash function

H(k) = k % m wherem is HASH_SIZE. The equivalent codecan be written as shown below:

Compute hash value using the function:H(k) = k % m

```
int H(int k)
{
        return k % HASH_SIZE;
}
```

Find the index for hash table given hash value

This can be done using the following code:

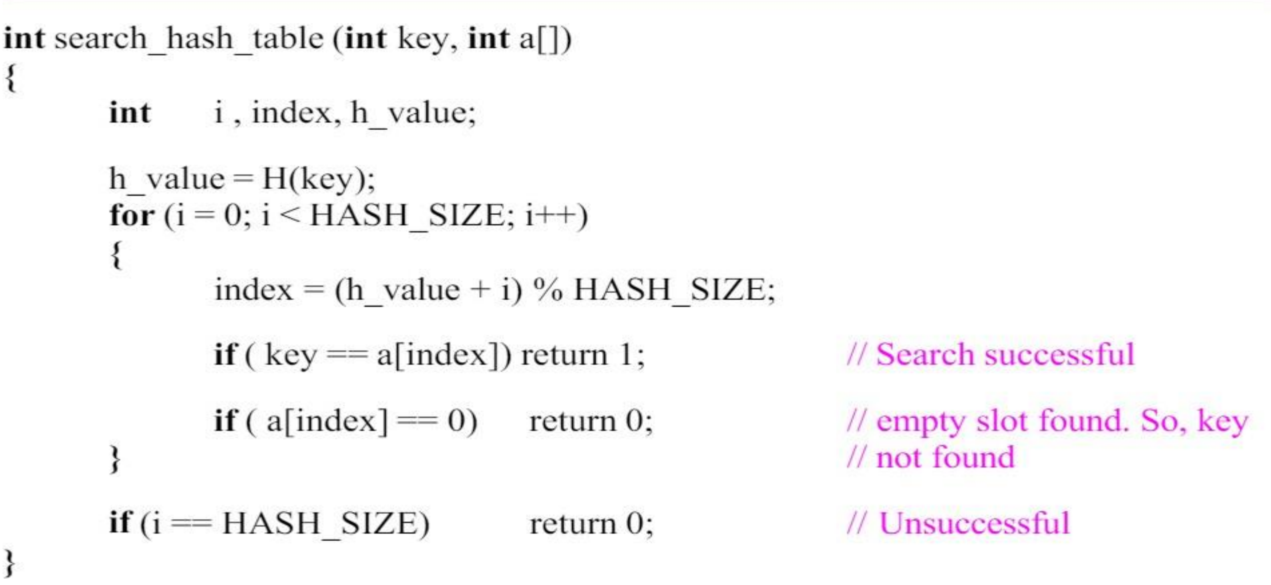
```
for (i = 0; i < HASH_SIZE; i++)

{
        printf ("%d ", i);                  // 0 1 2 3 4
}
```

Insert an item into hash table using linear probing

This is achieved by accessing each item in the hash table. Each item in the hash table can be accessed using the following code:

```
h_value = H(item);
for (i = 0; i < HASH_SIZE; i++)
{
        index = (h_value + i) % HASH_SIZE;
        a[index];
}
```

Search for an item in the hash table



```
int search_hash_table (int key, int a[])
{
        int     i , index, h_value;

        h_value = H(key);
        for (i = 0; i < HASH_SIZE; i++)
        {
                index = (h_value + i) % HASH_SIZE;

                if ( key == a[index]) return 1;             // Search successful

                if ( a[index] == 0)     return 0;           // empty slot found. So, key
        }                                                   // not found

        if (i == HASH_SIZE)         return 0;               // Unsuccessful
}
```

HASH FUNCTIONS:

overview of four different types of uniform hash functions:

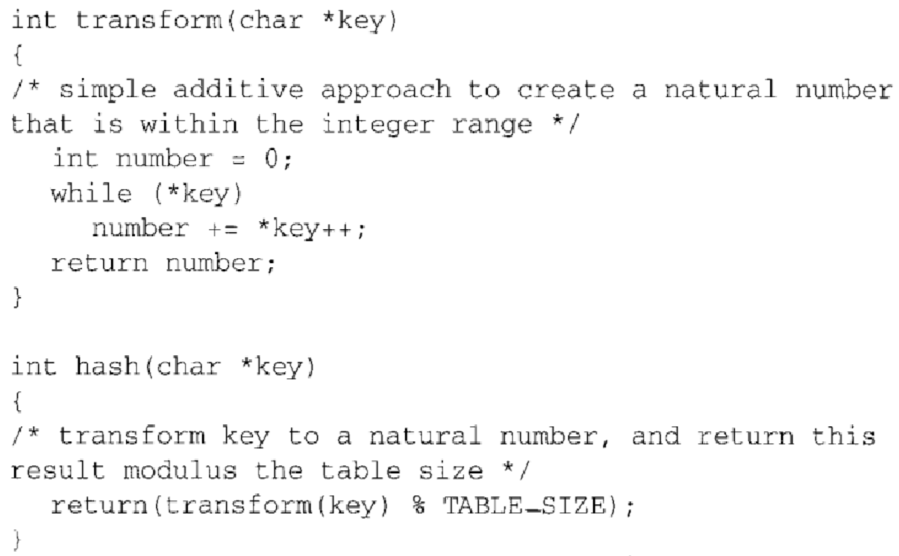
Mid-square: This method involves squaring the identifier and using a portion of the resulting square's bits to obtain the hash address. It is effective because it usually depends on all the characters in the identifier, reducing collisions.

Division: This method utilizes the modulus operator, where the identifier is divided by a chosen number M, and the remainder is used as the hash address. However, the choice of M is crucial, and it's recommended to avoid biases by selecting a prime number for M.

Folding: This technique partitions the identifier into parts and combines them to obtain the hash address. There are two methods: shift folding and folding at the boundaries, each with its own way of combining the parts.

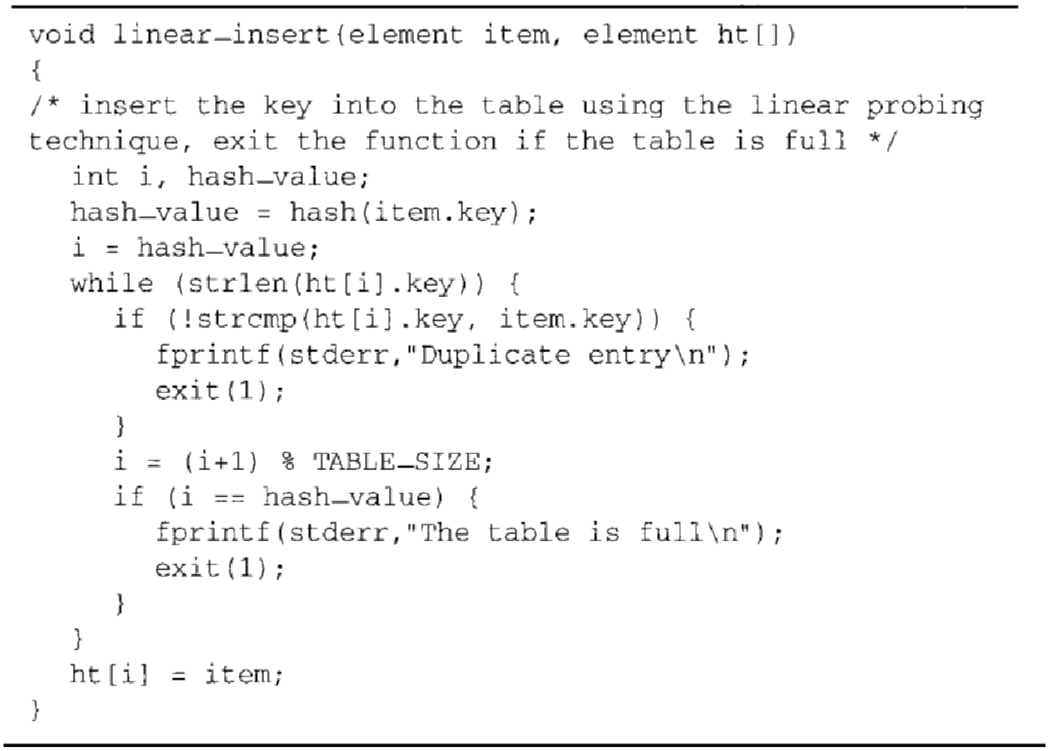
Digit Analysis: This approach is suitable for static files where all identifiers are known in advance. It involves transforming identifiers into numbers, examining their digits, and deleting digits with skewed distributions until the remaining digits provide a suitable hash address range.

Creation of a hash function



```
int transform(char *key)
{
/* simple additive approach to create a natural number
that is within the integer range */
  int number = 0;
  while (*key)
    number += *key++;
  return number;
}

int hash(char *key)
{
/* transform key to a natural number, and return this
result modulus the table size */
  return(transform(key) % TABLE_SIZE);
}
```

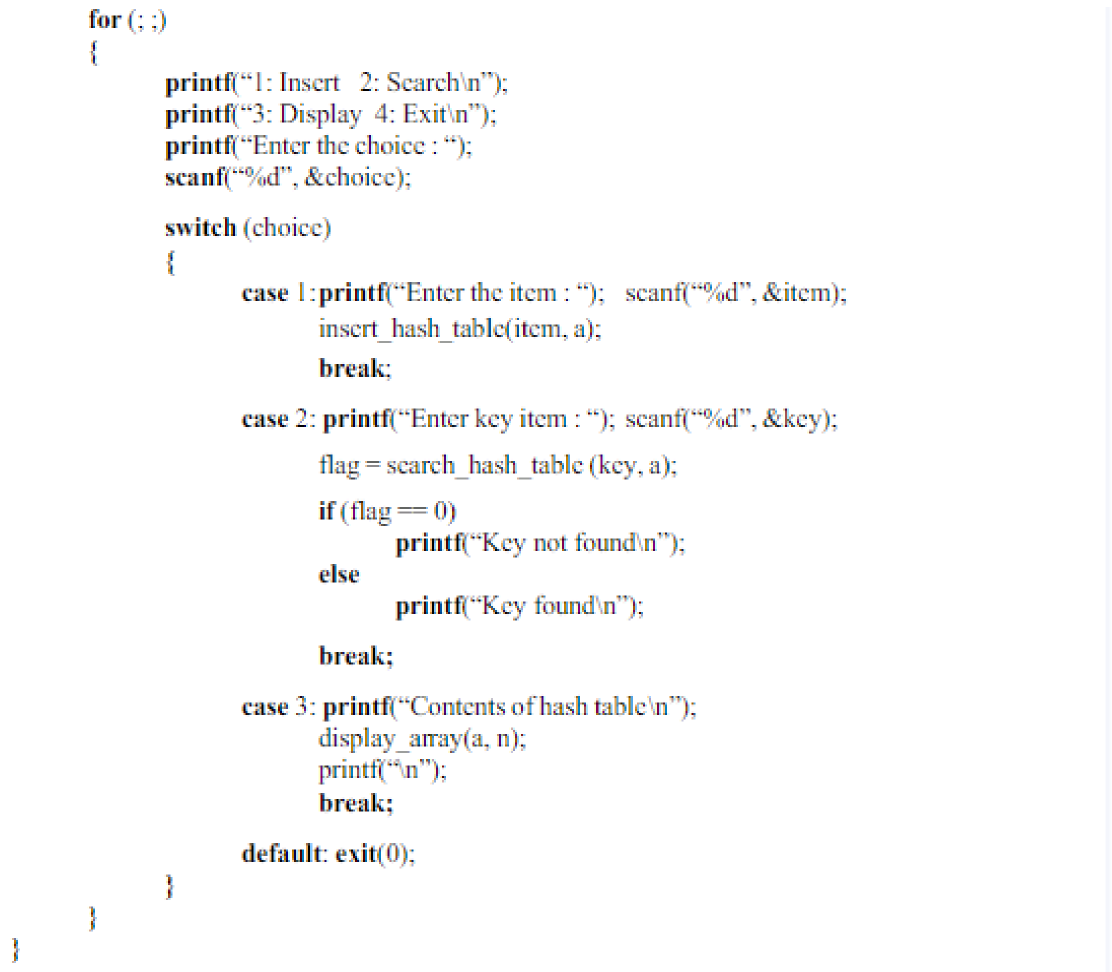
Linear insert into a hash table



```
void linear_insert(element item, element ht[])
{
/* insert the key into the table using the linear probing
technique, exit the function if the table is full */
   int i, hash_value;
   hash_value = hash(item.key);
   i = hash_value;
   while (strlen(ht[i].key)) {
      if (!strcmp(ht[i].key, item.key)) {
         fprintf(stderr,"Duplicate entry\n");
         exit(1);
      }
      i = (i+1) % TABLE_SIZE;
      if (i == hash_value) {
         fprintf(stderr,"The table is full\n");
         exit(1);
      }
   }
   ht[i] = item;
}
```

C program to create hash table and to search for key

```
#include <stdio.h>
#include <stdlib.h>

#define HASH_SIZE 5

void main()
{
        int a[10], item, key, choice, flag;

        initialize_hash_table(a);         // Create initial hash table
```



```
        for ( ; ;)
        {
                printf("1: Insert   2: Search\n");
                printf("3: Display  4: Exit\n");
                printf("Enter the choice : ");
                scanf("%d", &choice);

                switch (choice)
                {
                        case 1:printf("Enter the item : ");   scanf("%d", &item);
                                insert_hash_table(item, a);
                                break;

                        case 2: printf("Enter key item : "); scanf("%d", &key);
                                flag = search_hash_table (key, a);
                                if (flag == 0)
                                        printf("Key not found\n");
                                else
                                        printf("Key found\n");

                                break;

                        case 3: printf("Contents of hash table\n");
                                display_array(a, n);
                                printf("\n");
                                break;

                        default: exit(0);
                }
        }
}
```

DYNAMIC HASHING:

Challenges with Traditional Hashing in DBMS: Traditional hashing methods require static allocation of memory for the hash table, which can be inefficient. Allocating too much memory wastes space, while allocating too little requires restructuring the entire file when data exceeds the table's capacity, leading to time-consuming operations.

Dynamic Hashing Solution: Dynamic hashing, or extendible hashing, addresses these challenges by allowing the hash table to accommodate dynamically increasing and decreasing file sizes without penalties. It maintains fast retrieval times while adapting to changing data volumes.

File Structure: In dynamic hashing, a file (F) consists of records (R), each identified by a key field (K). Records are stored in buckets or pages, with each page typically having a capacity of p. Minimizing page accesses is crucial, as pages are often stored on disk, and retrieving them into memory dominates any operation.

Space Utilization: The efficiency of dynamic hashing is measured by the ratio of the number of records (n) to the total space (mp), where m is the number of pages. Maximizing space utilization is essential for optimal performance.

<u>Dynamic hashing using directories with the provided example of identifiers and their binary representations:</u>

1. **Page Structure:** We have four pages indexed by the 2-bit sequence: 00, 01, 10, 11, each capable of holding up to two identifiers.

2. **Identifier Representation:** Each identifier consists of two characters, with each character represented by 3 bits. For example:

- Identifier "aO" is represented as 100 000
- Identifier "al" is represented as 100 001
- Identifier "bO" is represented as 101 000
- And so on.

3. **Placement of Identifiers**: Using the two low-order bits of each identifier, we determine the page address for each identifier. For example:

- "aO" and "bO" have the same low-order bits (00), so they are placed on the first page (index 00).
- "c2" has the low-order bits 10, so it goes on the third page (index 10).
- "al" and "bl" have the low-order bits 01, so they go on the second page (index 01).
- "c3" has the low-order bits 11, so it goes on the fourth page (index 11).

4. **Trie Structure:** We construct a trie where each node represents a bit position, and branching occurs based on the value of that bit. For example:

- At the root node, we branch based on the least significant bit.
- At the next level, we branch based on the second least significant bit, and so on.

5. **Traversal**: To locate an identifier, we follow its bit sequence through the trie, branching accordingly at each node until reaching a leaf node containing a pointer to the corresponding page.

6. **Efficient Retrieval:** Organizing identifiers in this manner allows for efficient retrieval, as the trie structure enables direct traversal based on the binary representation of the identifiers.

7. **Leaf Nodes:** Only the leaf nodes of the trie contain pointers to pages, indicating where the identifiers are stored in the table.

C program that provides many of the details for implementing the directory version of dynamic hashing.

```
#include  <stdio.h>

#include <stdlib.h>

#include <string.h>


#define MAX_IDENTIFIERS 100

#define MAX_PAGES 4

#define MAX_IDENTIFIERS_PER_PAGE 2


// Structure to represent an identifier

typedef struct {

    char name[3]; // Assuming each identifier has two characters

} Identifier;


// Structure to represent a page

typedef struct {

    Identifier identifiers[MAX_IDENTIFIERS_PER_PAGE];

    int num_identifiers;

} Page;


// Structure to represent a directory

typedef struct {

    Page pages[MAX_PAGES];

} Directory;


// Hash function to determine the page index for an identifier
```

```
int hash(char identifier[3]) {
    // Use the two low-order bits of the identifier
    return (identifier[0] & 0x03);
}

// Function to insert an identifier into the directory
void insert_identifier(Directory *directory, char identifier[3]) {
    int index = hash(identifier);
    Page *page = &directory->pages[index];

    // Check if page is full
    if (page->num_identifiers < MAX_IDENTIFIERS_PER_PAGE) {
        strcpy(page->identifiers[page->num_identifiers].name, identifier);
        page->num_identifiers++;
    } else {
        // Page is full, handle overflow or split
        // Implementation specific, not included in this pseudo code
        printf("Page is full. Overflow or split needed.\n");
    }
}

// Function to search for an identifier in the directory
void search_identifier(Directory *directory, char identifier[3]) {
    int index = hash(identifier);
    Page *page = &directory->pages[index];

    // Search for the identifier in the page
    for (int i = 0; i < page->num_identifiers; i++) {
        if (strcmp(page->identifiers[i].name, identifier) == 0) {
            printf("Identifier %s found in page %d.\n", identifier, index);
            return;
        }
    }

    printf("Identifier %s not found.\n", identifier);
}

int main() {
    Directory directory;

    // Initialize directory pages
    for (int i = 0; i < MAX_PAGES; i++) {
        directory.pages[i].num_identifiers = 0;
    }

    // Example of inserting identifiers
    insert_identifier(&directory, "aO");
    insert_identifier(&directory, "bO");
    insert_identifier(&directory, "c2");
    insert_identifier(&directory, "al");
    insert_identifier(&directory, "bl");
    insert_identifier(&directory, "c3");

    // Example of searching for identifiers
    search_identifier(&directory, "aO");
    search_identifier(&directory, "c2");
    search_identifier(&directory, "d1"); // Identifier not inserted


    return 0;
}
```

Leftist tree:

To define a leftist tree, we first introduce the concept of an extended binary tree. An extended binary tree is a binary tree where all empty binary subtrees have been replaced by square nodes, called external nodes. The original nodes of the binary tree are called internal nodes.

1. Extended Binary Tree:

- An extended binary tree is a binary tree where all empty subtrees are replaced by square nodes, called external nodes.
- Internal nodes are the original nodes of the binary tree.

2. Shortest Path Length:

- For any node X in an extended binary tree, let shortest(X) denote the length of the shortest path from X to an external node.

3. Properties of Shortest Path Length:

- The shortest path length satisfies the following recurrence:

shortest(X) =

$$\begin{cases} 0 & \text{if X is an external node (square node)} \\ 1 & 1 + \min(\text{shortest(left-child(X))}, \text{shortest(right-child(X))}) \quad \text{otherwise} \end{cases}$$

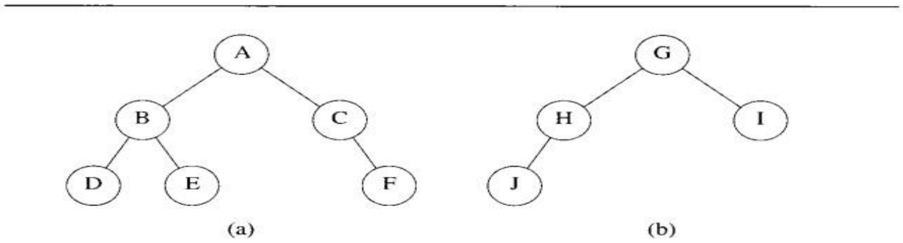
4. Leftist Tree:

- A leftist tree is a binary tree that satisfies the leftist property, which states that the shortest path length of any node's right subtree is always greater than or equal to the shortest path length of its left subtree.
- Formally, for any node X in a leftist tree, the leftist property is given by:

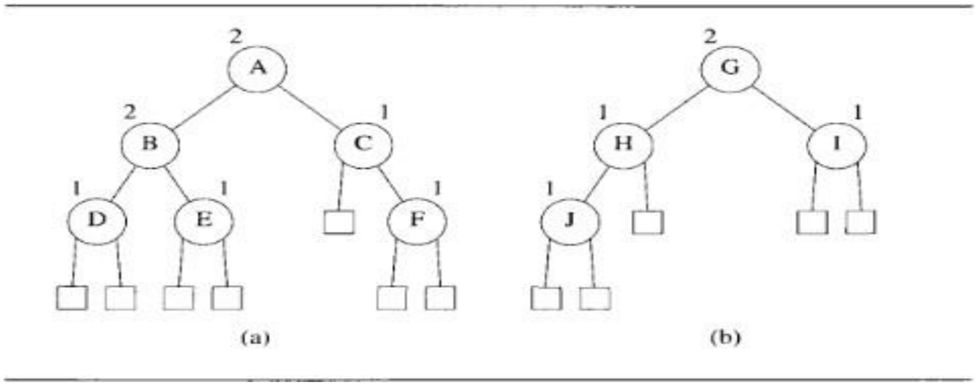
Shortest(right-child(X)) >= shortest(left-child(x))

5. Combine Operation:

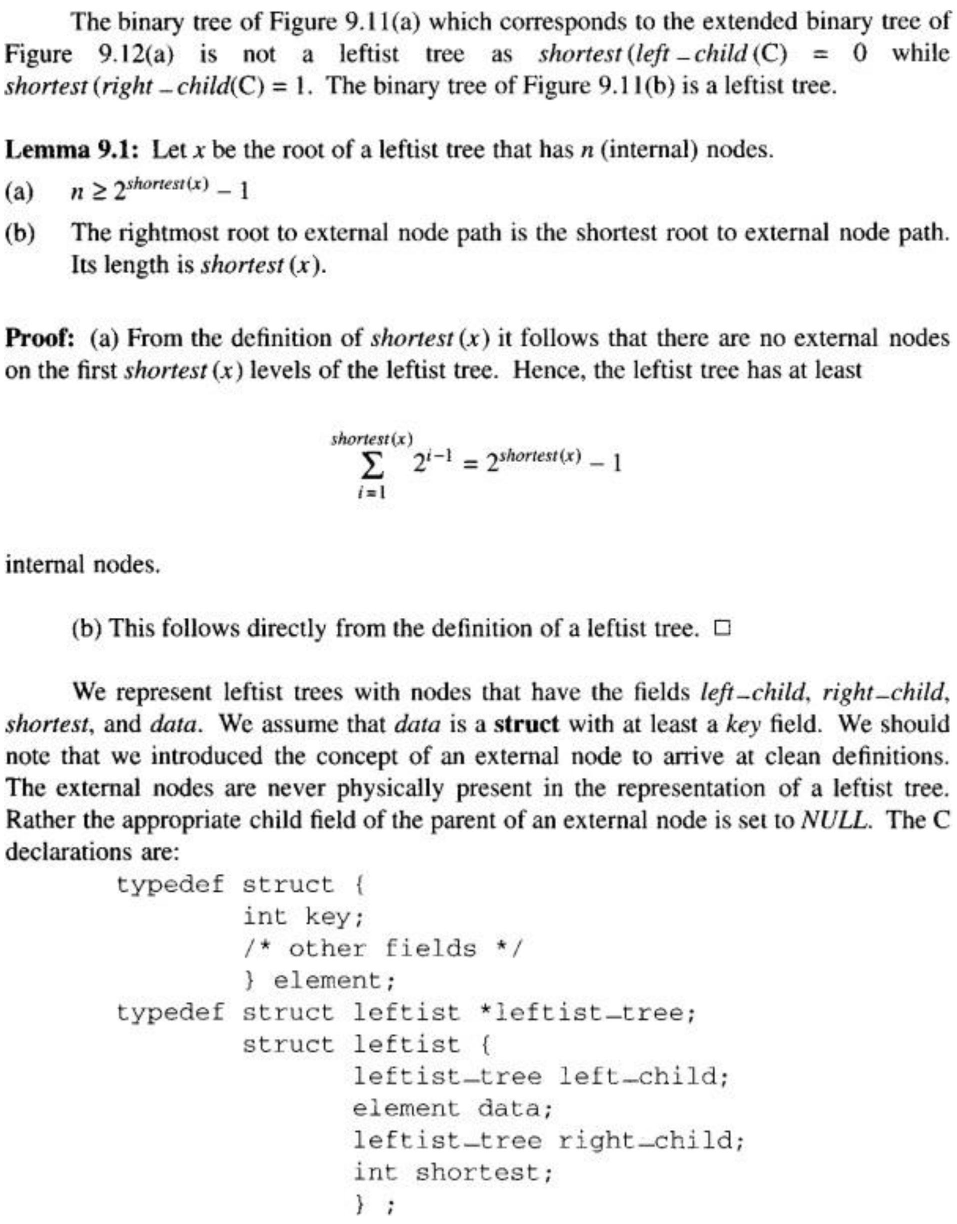
- The combine operation for leftist trees merges two leftist trees into a single leftist tree.
- During the combine operation, the trees are merged such that the leftist property is maintained.
- The combine operation in a leftist tree takes logarithmic time, making it efficient for merging priority queues.



Two binary trees



. Extended binary trees



The binary tree of Figure 9.11(a) which corresponds to the extended binary tree of Figure 9.12(a) is not a leftist tree as $shortest(left\_child(C)) = 0$ while $shortest(right\_child(C) = 1$. The binary tree of Figure 9.11(b) is a leftist tree.

Lemma 9.1: Let x be the root of a leftist tree that has n (internal) nodes.

(a) $n \geq 2^{shortest(x)} - 1$

(b) The rightmost root to external node path is the shortest root to external node path. Its length is $shortest(x)$.

Proof: (a) From the definition of $shortest(x)$ it follows that there are no external nodes on the first $shortest(x)$ levels of the leftist tree. Hence, the leftist tree has at least

$$\sum_{i=1}^{shortest(x)} 2^{i-1} = 2^{shortest(x)} - 1$$

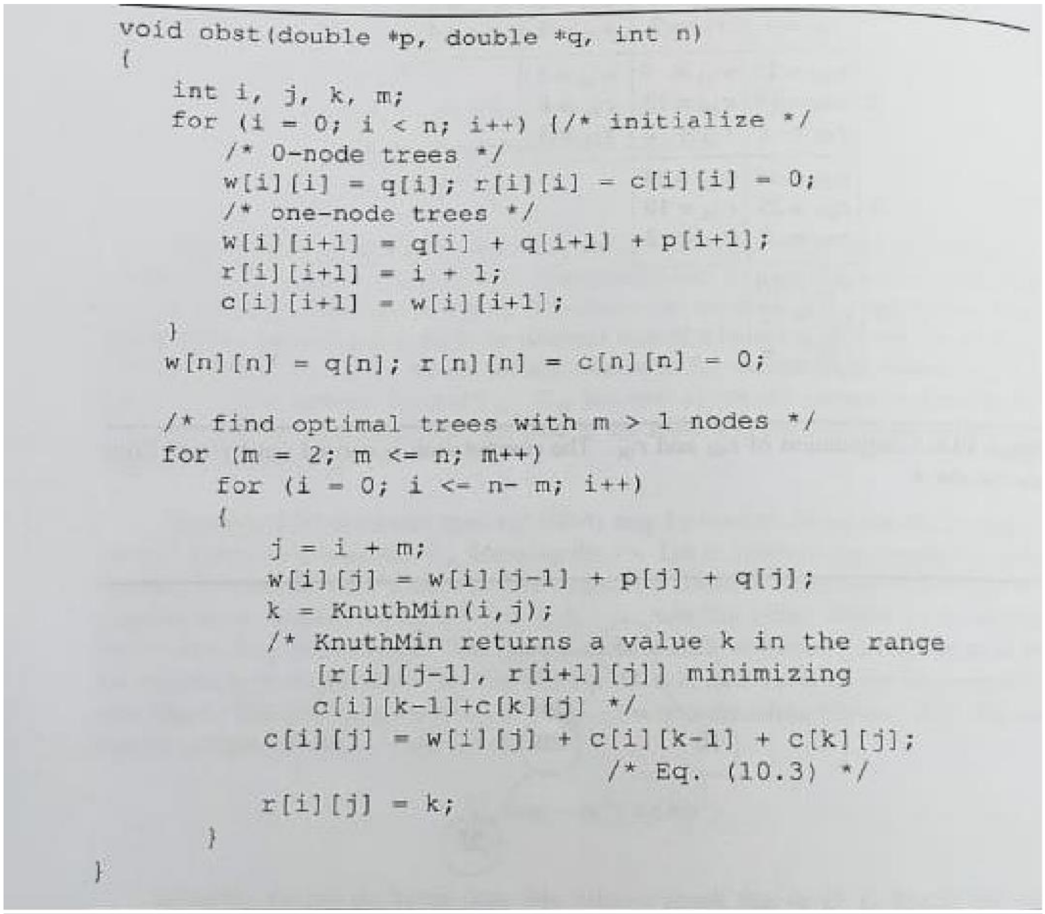
internal nodes.

(b) This follows directly from the definition of a leftist tree. □

We represent leftist trees with nodes that have the fields *left_child*, *right_child*, *shortest*, and *data*. We assume that *data* is a **struct** with at least a *key* field. We should note that we introduced the concept of an external node to arrive at clean definitions. The external nodes are never physically present in the representation of a leftist tree. Rather the appropriate child field of the parent of an external node is set to *NULL*. The C declarations are:

```
typedef struct {
        int key;
        /* other fields */
        } element;
typedef struct leftist *leftist_tree;
        struct leftist {
               leftist_tree left_child;
               element data;
               leftist_tree right_child;
               int shortest;
               } ;
```

Finding an optimal binary search tree



```
void obst(double *p, double *q, int n)
{
  int i, j, k, m;
  for (i = 0; i < n; i++) {/* initialize */
    /* 0-node trees */
    w[i][i] = q[i]; r[i][i] = c[i][i] = 0;
    /* one-node trees */
    w[i][i+1] = q[i] + q[i+1] + p[i+1];
    r[i][i+1] = i + 1;
    c[i][i+1] = w[i][i+1];
  }
  w[n][n] = q[n]; r[n][n] = c[n][n] = 0;

  /* find optimal trees with m > 1 nodes */
  for (m = 2; m <= n; m++)
    for (i = 0; i <= n- m; i++)
    {
      j = i + m;
      w[i][j] = w[i][j-1] + p[j] + q[j];
      k = KnuthMin(i,j);
      /* KnuthMin returns a value k in the range
         [r[i][j-1], r[i+1][j]] minimizing
         c[i][k-1]+c[k][j] */
      c[i][j] = w[i][j] + c[i][k-1] + c[k][j];
                              /* Eq. (10.3) */
      r[i][j] = k;
    }
}
```

||Jai Sri Gurudev ||
BGSKH Education Trust (R.) – A unit of Sri Adichunchanagiri Shikshana Trust(R.)
BGS College of Engineering and Technology (BGSCET)
Mahalakshmipuram, West of Chord Road, Bengaluru-560086
(Approved by AICTE, New Delhi and Affiliated to VTU, Belagavi)

Question Bank

Subject: Data Structure and Applications **Class:** AIDS

Subject code: BCS304 **Faculty:** Mrs. Jyothi R

Course Outcomes

CO 1. Explain different data structures and their applications.
CO 2. Apply Arrays, Stacks and Queue data structures to solve the given problems.
CO 3. Use the concept of linked list in problem solving.
CO 4. Develop solutions using trees and graphs to model the real-world problem.
CO 5. Explain the advanced Data Structures concepts such as Hashing Techniques and Optimal Binary Search Trees

MODULE 5

Sl. No.	**Questions**	**CO**
1.	What is Hashing? Explain how Hashing table is constructed	CO5
2.	Explain the creation of hash function.	CO5
3.	Discuss Linear insert into hash table	CO5
4.	Write a C program to create hash table and to search for key	CO5
5.	Explain DYNAMIC HASHING	CO5
6.	Dynamic hashing using directories with the provided example of identifiers and their binary representations	CO5
7.	C program that provides many of the details for implementing the directory version of dynamic hashing	CO5
8.	Explain Leftist tree.	CO5
9.	Write a function to Finding an optimal binary search tree	CO5

||Jai Sri Gurudev ||

BGSKH Education Trust (R.) – A unit of Sri Adichunchanagiri Shikshana Trust(R.)

BGS College of Engineering and Technology

Mahalakshmipuram, West of Chord Road, Bengaluru-560086

(Approved by AICTE, New Delhi and Affiliated to VTU, Belagavi)

Object Oriented Programming with JAVA

NOTES

For third Semester BE [VTU/CBCS, 2023-24 Syllabus]

Subject Code:

B	C	S	3	0	6	A

Syllabus

Object Oriented Programming with JAVA		Semester	3
Course Code	BCS306A	CIE Marks	50
Teaching Hours/Week (L: T:P: S)	2:0:2	SEE Marks	50
Total Hours of Pedagogy	28 Hours of Theory + 20 Hours of Practical	Total Marks	100
Credits	03	Exam Hours	03
Examination type (SEE)	Theory		

Note - Students who have undergone “ Basics of Java Programming-BPLCK105C/205C” in first year are not eligible to opt this course

Course objectives:

- To learn primitive constructs JAVA programming language.
- To understand Object Oriented Programming Features of JAVA.
- To gain knowledge on: packages, multithreaded programing and exceptions.

Teaching-Learning Process (General Instructions)

These are sample Strategies, which teachers can use to accelerate the attainment of the various course outcomes and make Teaching –Learning more effective

- Use Online Java Compiler IDE: https://www.jdoodle.com/online-java-compiler/ or any other.
- Demonstration of programing examples.
- Chalk and board, power point presentations
- Online material (Tutorials) and video lectures.

Module-1

An Overview of Java: Object-Oriented Programming (Two Paradigms, Abstraction, The Three OOP Principles), Using Blocks of Code, Lexical Issues (Whitespace, Identifiers, Literals, Comments, Separators, The Java Keywords).

Data Types, Variables, and Arrays: The Primitive Types (Integers, Floating-Point Types, Characters, Booleans), Variables, Type Conversion and Casting, Automatic Type Promotion in Expressions, Arrays, Introducing Type Inference with Local Variables.

Operators: Arithmetic Operators, Relational Operators, Boolean Logical Operators, The Assignment Operator, The ? Operator, Operator Precedence, Using Parentheses.

Control Statements: Java’s Selection Statements (if, The Traditional switch), Iteration Statements (while, do-while, for, The For-Each Version of the for Loop, Local Variable Type Inference in a for Loop, Nested Loops), Jump Statements (Using break, Using continue, return).

Chapter 2, 3, 4, 5

Module-2

Introducing Classes: Class Fundamentals**,** Declaring Objects**,** Assigning Object Reference Variables**,** Introducing Methods**,** Constructors**,** The this Keyword**,** Garbage Collection.

Methods and Classes: Overloading Methods, Objects as Parameters, Argument Passing, Returning Objects, Recursion, Access Control, Understanding static, Introducing final, Introducing Nested and Inner Classes.

Chapter 6, 7

Module-3

Inheritance: Inheritance Basics, Using super, Creating a Multilevel Hierarchy, When Constructors Are Executed, Method Overriding, Dynamic Method Dispatch, Using Abstract Classes, Using final with Inheritance, Local Variable Type Inference and Inheritance, The Object Class.

Interfaces: Interfaces, Default Interface Methods, Use static Methods in an Interface, Private Interface

Methods.
Chapter 8, 9

Module-4

Packages: Packages, Packages and Member Access, Importing Packages.
Exceptions: Exception-Handling Fundamentals**,** Exception Types**,** Uncaught Exceptions**,** Using try and catch, Multiple catch Clauses, Nested try Statements, throw, throws, finally, Java's Built-in Exceptions, Creating Your Own Exception Subclasses, Chained Exceptions**.**
Chapter 9, 10

Module-5

Multithreaded Programming: The Java Thread Model, The Main Thread, Creating a Thread, Creating Multiple Threads, Using isAlive() and join(), Thread Priorities, Synchronization, Interthread Communication, Suspending, Resuming, and Stopping Threads, Obtaining a Thread's State.
Enumerations, Type Wrappers and Autoboxing: Enumerations (Enumeration Fundamentals**,** The values() and valueOf() Methods), Type Wrappers (Character, Boolean, The Numeric Type Wrappers), Autoboxing (Autoboxing and Methods, Autoboxing/Unboxing Occurs in Expressions, Autoboxing/Unboxing Boolean and Character Values).
Chapter 11, 12

Course outcome (Course Skill Set)

At the end of the course, the student will be able to:

1. Demonstrate proficiency in writing simple programs involving branching and looping structures.
2. Design a class involving data members and methods for the given scenario.
3. Apply the concepts of inheritance and interfaces in solving real world problems.
4. Use the concept of packages and exception handling in solving complex problem
5. Apply concepts of multithreading, autoboxing and enumerations in program development

Programming Experiments (Suggested and are not limited to)

1. Develop a JAVA program to add TWO matrices of suitable order N (The value of N should be read from command line arguments).
2. Develop a stack class to hold a maximum of 10 integers with suitable methods. Develop a JAVA main method to illustrate Stack operations.
3. A class called Employee, which models an employee with an ID, name and salary, is designed as shown in the following class diagram. The method raiseSalary (percent) increases the salary by the given percentage. Develop the Employee class and suitable main method for demonstration.
4. A class called MyPoint, which models a 2D point with x and y coordinates, is designed as follows:

   - Two instance variables x (int) and y (int).
   - A default (or "no-arg") constructor that construct a point at the default location of (0, 0).
   - A overloaded constructor that constructs a point with the given x and y coordinates.
   - A method setXY() to set both x and y.
   - A method getXY() which returns the x and y in a 2-element int array.
   - A toString() method that returns a string description of the instance in the format "(x, y)".
   - A method called distance(int x, int y) that returns the distance from this point to another point at the given (x, y) coordinates
   - An overloaded distance(MyPoint another) that returns the distance from this point to the given MyPoint instance (called another)
   - Another overloaded distance() method that returns the distance from this point to the origin (0,0)

   Develop the code for the class MyPoint. Also develop a JAVA program (called TestMyPoint) to test all the methods defined in the class.
5. Develop a JAVA program to create a class named shape. Create three sub classes namely: circle, triangle and square, each class has two member functions named draw () and erase (). Demonstrate olymorphism concepts by developing suitable methods, defining member data and main program.

6. Develop a JAVA program to create an abstract class Shape with abstract methods calculateArea() and calculatePerimeter(). Create subclasses Circle and Triangle that extend the Shape class and implement the respective methods to calculate the area and perimeter of each shape.
7. Develop a JAVA program to create an interface Resizable with methods resizeWidth(int width) and resizeHeight(int height) that allow an object to be resized. Create a class Rectangle that implements the Resizable interface and implements the resize methods
8. Develop a JAVA program to create an outer class with a function display. Create another class inside the outer class named inner with a function called display and call the two functions in the main class.
9. Develop a JAVA program to raise a custom exception (user defined exception) for DivisionByZero using try, catch, throw and finally.
10. Develop a JAVA program to create a package named mypack and import & implement it in a suitable class.
11. Write a program to illustrate creation of threads using runnable class. (start method start each of the newly created thread. Inside the run method there is sleep() for suspend the thread for 500 milliseconds).
12. Develop a program to create a class MyThread in this class a constructor, call the base class constructor, using super and start the thread. The run method of the class starts after this. It can be observed that both main thread and created child thread are executed concurrently.

Assessment Details (both CIE and SEE)

The weightage of Continuous Internal Evaluation (CIE) is 50% and for Semester End Exam (SEE) is 50%. The minimum passing mark for the CIE is 40% of the maximum marks (20 marks out of 50) and for the SEE minimum passing mark is 35% of the maximum marks (18 out of 50 marks). A student shall be deemed to have satisfied the academic requirements and earned the credits allotted to each subject/ course if the student secures a minimum of 40% (40 marks out of 100) in the sum total of the CIE (Continuous Internal Evaluation) and SEE (Semester End Examination) taken together.

CIE for the theory component of the IPCC (maximum marks 50)

- IPCC means practical portion integrated with the theory of the course.
- CIE marks for the theory component are **25 marks** and that for the practical component is **25 marks**.
- 25 marks for the theory component are split into **15 marks** for two Internal Assessment Tests (Two Tests, each of 15 Marks with 01-hour duration, are to be conducted) and **10 marks** for other assessment methods mentioned in 22OB4.2. The first test at the end of 40-50% coverage of the syllabus and the second test after covering 85-90% of the syllabus.
- Scaled-down marks of the sum of two tests and other assessment methods will be CIE marks for the theory component of IPCC (that is for **25 marks)**.
- The student has to secure 40% of 25 marks to qualify in the CIE of the theory component of IPCC.

CIE for the practical component of the IPCC

- **15 marks** for the conduction of the experiment and preparation of laboratory record, and **10 marks** for the test to be conducted after the completion of all the laboratory sessions.
- On completion of every experiment/program in the laboratory, the students shall be evaluated including viva-voce and marks shall be awarded on the same day.
- The CIE marks awarded in the case of the Practical component shall be based on the continuous evaluation of the laboratory report. Each experiment report can be evaluated for 10 marks. Marks of all experiments' write-ups are added and scaled down to **15 marks**.
- The laboratory test **(duration 02/03 hours)** after completion of all the experiments shall be conducted for 50 marks and scaled down to **10 marks.**
- Scaled-down marks of write-up evaluations and tests added will be CIE marks for the laboratory component of IPCC for **25 marks**.
- The student has to secure 40% of 25 marks to qualify in the CIE of the practical component of the IPCC.

SEE for IPCC

Theory SEE will be conducted by University as per the scheduled timetable, with common question papers for the course (**duration 03 hours**)

1. The question paper will have ten questions. Each question is set for 20 marks.
2. There will be 2 questions from each module. Each of the two questions under a module (with a maximum of 3 sub-questions), **should have a mix of topics** under that module.
3. The students have to answer 5 full questions, selecting one full question from each module.
4. Marks scored by the student shall be proportionally scaled down to 50 Marks

The theory portion of the IPCC shall be for both CIE and SEE, whereas the practical portion will have a CIE component only. Questions mentioned in the SEE paper may include questions from the practical component.

Suggested Learning Resources:

Textbook

1. Java: The Complete Reference, Twelfth Edition, by Herbert Schildt, November 2021, McGraw-Hill, ISBN:9781260463422

Reference Books

1. Programming with Java, 6th Edition, by E Balagurusamy, Mar-2019, McGraw Hill Education, ISBN: 9789353162337.
2. Thinking in Java, Fourth Edition, by Bruce Eckel, Prentice Hall, 2006(https://sd.blackball.lv/library/thinking_in_java_4th_edition.pdf)

Web links and Video Lectures (e-Resources):

- Java Tutorial: https://www.geeksforgeeks.org/java/
- Introduction To Programming In Java (by Evan Jones, Adam Marcus and Eugene Wu): https://ocw.mit.edu/courses/6-092-introduction-to-programming-in-java-january-iap-2010/
- Java Tutorial: https://www.w3schools.com/java/
- Java Tutorial: https://www.javatpoint.com/java-tutorial

Activity Based Learning (Suggested Activities)/ Practical Based learning

1. Installation of Java (Refer: https://www.java.com/en/download/help/index_installing.html)
2. Demonstration of online IDEs like geeksforgeeks, jdoodle or any other Tools
3. Demonstration of class diagrams for the class abstraction, type visibility, composition and inheritance

Assessment Method

- Programming Assignment / Course Project

MODULE 1

Chapter 2: An Overview of Java:

Object-Oriented Programming:

- Object oriented programming (OOP) is the core of Java programming.
- Java is a general purpose, object- oriented programming language developed by Sun Microsystems. It was invented by James Gosling and his team and was initially called as Oak.
- The most important feature that made Java very popular is the Platform Independent approach.
- It was the first programming language that did not tie-up with any particular operating system (or hardware) rather Java programs can be executed anywhere and on any system.
- Java was designed for the development of the software for consumer electronic devices like TVs, VCRs, etc.

Two Paradigms:

- Every program contains two components - code and data.
- Two approaches are there to solve the problem and in program writing:

  Procedure oriented and object oriented.

Procedure Oriented:

- Procedure oriented programs are written based on –what's happening around, where the code acts on data. Ex: C language
- Problems increases in procedure oriented as the program grows larger and more complex.

Object Oriented:

- Object oriented programs are written based on –Who is being affected around, which manages the increasing complexity.
- It organizes program around data and well-defined interfaces of that data.
- Characterized as data controlling access to code. Ex: C++, JAVA, Small Talk etc.

Abstraction:

- Data abstraction refers to providing only essential information to the outside world and hiding their background details i.e., to represent the needed information in program without presenting the details.

  Example: we see a car as a single object with unique behavior, not as a collection of parts.
- Manage complexity through hierarchical classifications.
- Break complex systems into more manageable pieces.

- For instance, a car is composed of subsystems (steering, brakes, sound system) and further specialized units (e.g., radio, CD player).

- Traditional process-oriented programs can be transformed into objects through abstraction.
- A sequence of process steps becomes a collection of messages between objects, each with unique behavior.

The Three OOP:

The three important features of OOP are:

- Encapsulation
- Inheritance
- Polymorphism

Encapsulation

- Encapsulation is the mechanism that binds together code and data it manipulates, and keeps both safe from outside interference and misuse.
- In Java the basis of encapsulation is the class. A class defines the state and behavior (data & code) that will be shared by set of objects.
- Each object contains the structure and behavior defined by the class. The data defined by the class are called instance variables(member variables), the code that operates on that data are called methods(member functions).

Inheritance

- Inheritance is the process by which one object acquires the properties of another object. This is important as it supports the concept of hierarchical classification.
- By the use of inheritance, a class has to define only those qualities that make it unique. The general qualities can be derived from the parent class or base class.
- Ex: A child inheriting properties from parents.

Polymorphism

- Polymorphism (meaning many forms) is a feature that allows one interface to be used for a general class of actions. The specific action determined by the exact nature of the situation. This concept is often expressed as — one interface, multiple methods

Example: + can be used for addition of 2 numbers and also concatenation of 2 strings.

```
System.out.println(2+4); // outputs 6 as answer
    System.out.println(“Hello” +”world”); // outputs Hello world as answer
```

Polymorphism, Encapsulation and Inheritence work Together

- The 3 principles of OOP Polymorphism, Encapsulation and Inheritence combines together to make the programming robust and scalable.
- Encapsulation allows to migrate the implementation without disturbing the code that depends on class.
- Polymorphism allows to create clean, sensible, readable, resilient code.
- Inheritance mainly deals with the code reusability.

Using blocks of code

- Java supports **code blocks -** which means that two or more statements are grouped into blocks of code.
- Opening and closing braces is used to achieve this.
- Each block is treated as logical unit.
- Whenever two or more statements has to be linked blocks can be used.
- Example:class Example

```
{
        public static void main(String args[])
        {
              int a=10;
              if(a>0)
             { // begin of block
                System.out.println(–a is positive number‖);
                System.out.println(– inside block‖);
              }// end of block
        }
}
```

Lexical issues:

Java programs are a collection of whitespace, identifiers, literals, comments, operators, separators, and keywords.

Whitespace:

- Java is a free from language- means no need to follow any indentation rules.
- Whitespace is a space, tab, or newline.

Identifiers

Identifiers are used to name things, such as classes, variables, and methods.

Rules to frame identifiers

uppercase and lowercase letters numbers, or the underscore, dollar sign

must not begin with a number,

Java is case sensitive (VALUE is different from Value)

Valid identifiers names

AvgTemp   count   a4 $test   this_is_ok

Invalid identifier names

2count   high-temp   Not/ok

Literals:

A constant value in Java is created by using a *literal* representation of it. Literals in Java are sequence of characters that represents constant values to be stored invariables. Java language specifies five major types of Literals.
They are:

Integer Literals – eg. 10

Floating-point Literals. Eg. 12.7

Character Literals. Eg . 'v'

String Literals. Eg "hello"

Boolean Literals. Eg true

Comments:

There are three types of comments defined by Java.

Single line comments: this type of comment begins with // and ends at the end of current line

Ex: // Welcome to java Programming

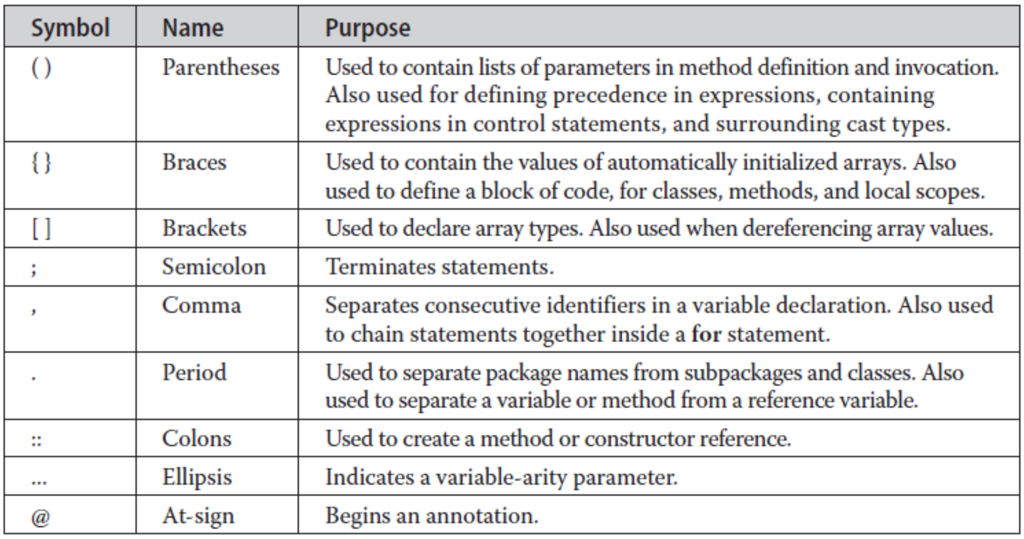
Multiline comment: this type of comment begins with /* and ends with */

Ex: /* Welcome to Java Programming */

Documentation Comment: this type of comment is used to produce an HTML file that documents your program. The documentation comment begins with /** and ends with */

Separators:

Separators are the symbols that indicates where group of code are divided and arranged. Some of the operators are:



Symbol	Name	Purpose
( )	Parentheses	Used to contain lists of parameters in method definition and invocation. Also used for defining precedence in expressions, containing expressions in control statements, and surrounding cast types.
{ }	Braces	Used to contain the values of automatically initialized arrays. Also used to define a block of code, for classes, methods, and local scopes.
[ ]	Brackets	Used to declare array types. Also used when dereferencing array values.
;	Semicolon	Terminates statements.
,	Comma	Separates consecutive identifiers in a variable declaration. Also used to chain statements together inside a **for** statement.
.	Period	Used to separate package names from subpackages and classes. Also used to separate a variable or method from a reference variable.
::	Colons	Used to create a method or constructor reference.
...	Ellipsis	Indicates a variable-arity parameter.
@	At-sign	Begins an annotation.

Java character set:

- The smallest unit of Java language are its character set used to write Java tokens. This character is defined by Unicode character set that tries to create character for a large number of character worldwide.
- The Unicode is a 16-bit character coding system and currently supports 34,000 defined characters derived from 24 languages of worldwide.

Key Words:

Java program is basically a collection of classes. A class is defined by a set of declaration statements and methods containing executable statements. Most statement contains an expression that contains the action carried out on data. The compiler recognizes the tokens for building up the expression and statements. Smallest individual units of programs are known as tokens. Java language includes five types of tokens. They are

Reserved Keyword

Identifiers

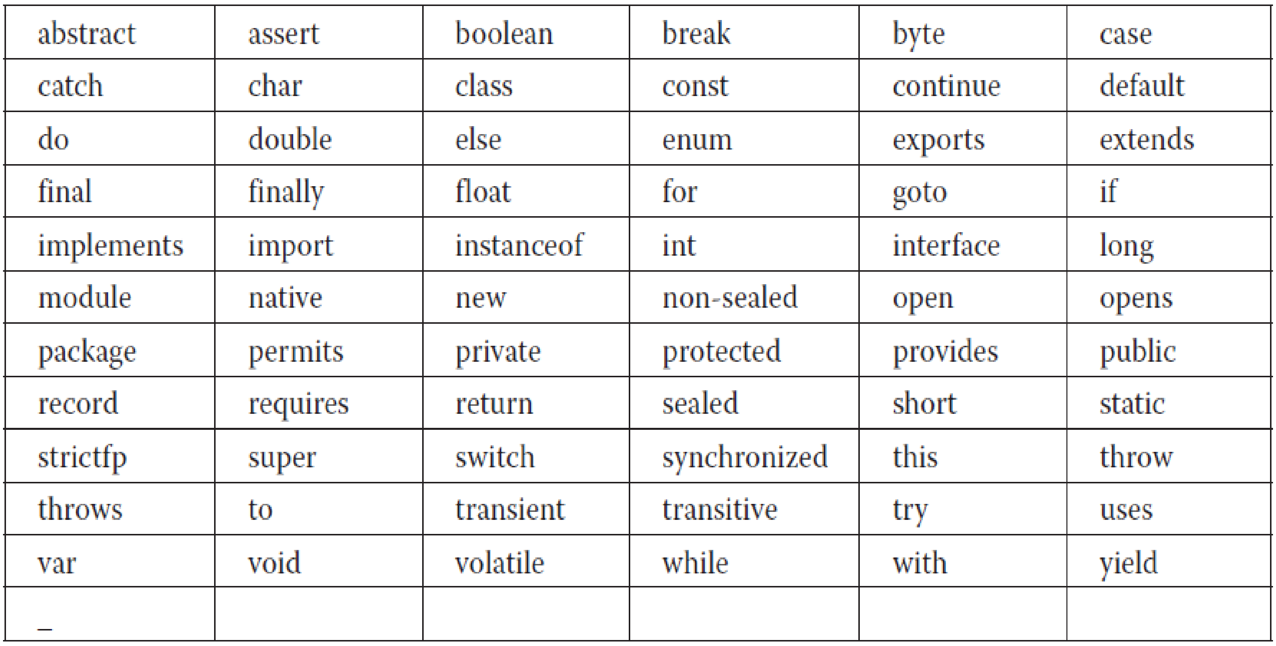
Literals.

Operators

Separators.

Reserved keyword:

- Java language has 50 words as reserved keywords. They implement specific feature of the language. The keywords combined with operators and separators according to syntax build the Java language.



abstract	assert	boolean	break	byte	case
catch	char	class	const	continue	default
do	double	else	enum	exports	extends
final	finally	float	for	goto	if
implements	import	instanceof	int	interface	long
module	native	new	non-sealed	open	opens
package	permits	private	protected	provides	public
record	requires	return	sealed	short	static
strictfp	super	switch	synchronized	this	throw
throws	to	transient	transitive	try	uses
var	void	volatile	while	with	yield
_					

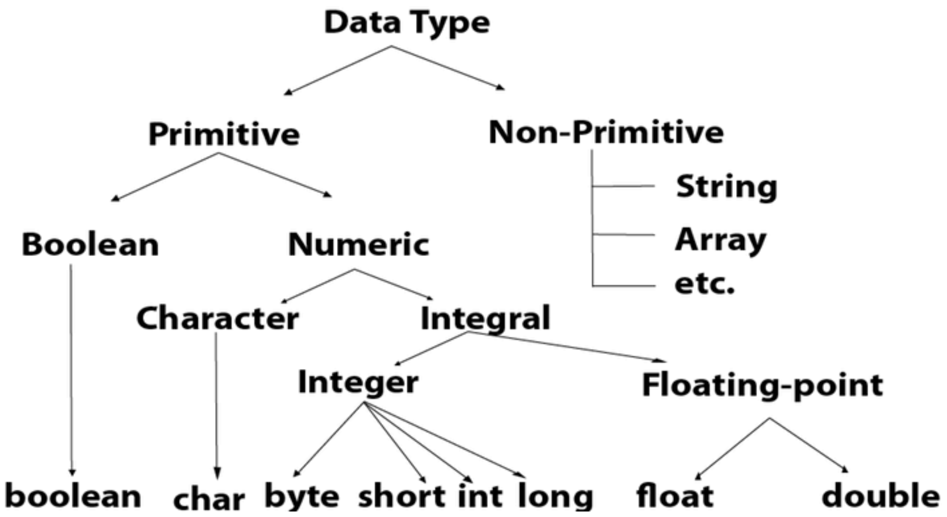
Chapter 3: Data Types, Variables and Arrays

Java is a strongly typed language:

- The strongly typed nature of Java gives it the robustness and safety for it.
- Every variable and expression has strictly defined type.
- Assignments, parameter passing or explicit value passing are checked for type compatibility.
- Java compiler checks all expressions and parameters to ensure type compatibility

Data types

The various data types supported in java is as follows



Java defines eight *primitive* types of data:

byte, **short**, **int**, **long**, **char**, **float**, **double**, and boolean

The primitive types are also commonly referred to as *simple* types, and both terms will be used in this book. These can be put in four groups:

• **Integers** This group includes **byte**, **short**, **int**, and **long**, which are for whole valued signed numbers.

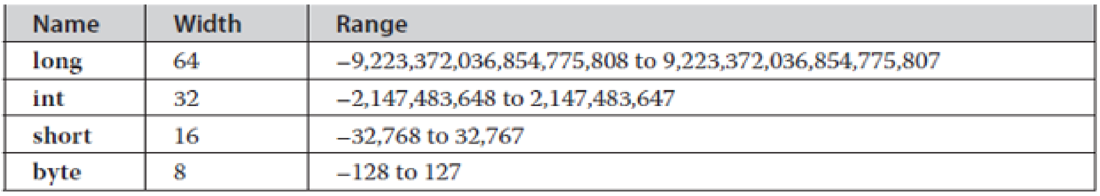
• **Floating-point numbers** This group includes **float** and **double**, which represent numbers with fractional precision.

• **Characters** This group includes **char**, which represents symbols in a character set, like letters and numbers.

• **boolean** This group includes **boolean**, which is a special type for representing true/false values.

Integers

- Java defines four integer types: **byte**, **short**, **int**, and **long**.
- All of these are signed, positive and negative values. Java does not support unsigned, positive-only integers.



Name	Width	Range
long	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	32	–2,147,483,648 to 2,147,483,647
short	16	–32,768 to 32,767
byte	8	–128 to 127

byte

- The smallest integer type is **byte**.
- This is a signed 8-bit type that has a range from –128 to127.
- Variables of type **byte** are especially useful when you_re working with a stream of data from a network or file.
- Byte variables are declared by use of the **byte** keyword.
- For example, the following declares two **byte** variables called **b** and **c**: byte b, c;

short

- **short** is a signed 16-bit type.
- It has a range from –32,768 to 32,767.
- It is probably the least-used Java type.

examples of **short** variable declarations: short s; short t;

int

- **int** is a signed 32-bit type.
- It has a range from –2,147,483,648 to 2,147,483,647.
- **byte** and **short** values are used in an expression, they are *promoted* to **int** when the expression is
- evaluated.

long

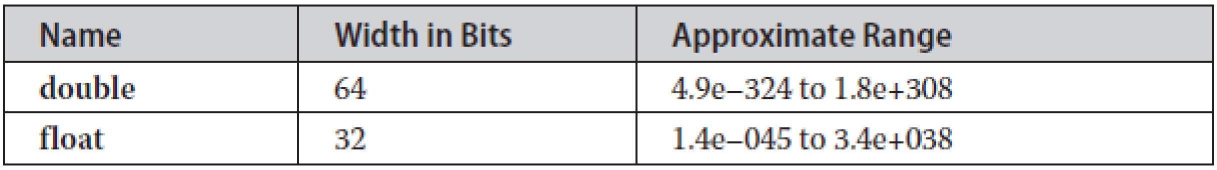
- **long** is a signed 64-bit type
- and is useful where an **int** type is not large enough to hold the desired value.
- The range of a **long** is –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.
- it useful when big, whole numbers are needed.

For example,

```
//Program that computes the number of miles that light will travel in a specified number of days:
// Compute distance light travels using long variables.
class Light {
public static void main(String[] args) {
int lightspeed;
long days;
long seconds;
long distance;
// approximate speed of light in miles per second
lightspeed = 186000;
days = 1000; // specify number of days here
seconds = days * 24 * 60 * 60; // convert to seconds
distance = lightspeed * seconds; // compute distance
System.out.print("In " + days);
System.out.print(" days light will travel about ");
System.out.println(distance + " miles.");
}
}
```

Floating-Point Types

- Floating-point numbers, also known as *real* numbers, are used when evaluating expressions that require fractional precision.
- For example, calculations such as square root, or transcendental such as sine and cosine, result in a value whose precision requires a floating-point type.
- There are two kinds of floating-point types, **float** and **double**, which represent single- and double-precision numbers, respectively.



Name	Width in Bits	Approximate Range
double	64	4.9e–324 to 1.8e+308
float	32	1.4e–045 to 3.4e+038

float

- The type **float** specifies a *single-precision* value that uses 32 bits of storage.

double

- Double precision, as denoted by the **double** keyword, uses 64 bits to store a value.
- Double precision is actually faster than single precision on some modern processors that have been optimized for high-speed mathematical calculations.

Eg. double variables to compute the area of a circle:

```
// Compute the area of a circle.
class Area {
public static void main(String[] args) {
double pi, r, a;
r = 10.8; // radius of circle
pi = 3.1416; // pi, approximately
a = pi * r * r; // compute area
System.out.println("Area of circle is " + a);
}
}
```

Characters

- In Java, the data type used to store characters is **char**.
- In C/C++, **char** is 8 bits wide. But Java uses Unicode to represent characters.
- *Unicode* defines a fully international character set that can represent all of the characters found in all human languages.
- It is a unification of dozens of character sets, such as Latin, Greek Arabic, Cyrillic,Hebrew, Katakana, Hangul, and many more. For this purpose, it requires 16 bits.
- Thus, in Java **char** is a 16-bit type. The range of a **char** is 0 to 65,536.

Eg.:

```
// Demonstrate char data type.
class CharDemo {
public static void main(String[] args) {
char ch1, ch2;
ch1 = 88; // code for X
ch2 = 'Y';
System.out.print("ch1 and ch2: ");
System.out.println(ch1 + " " + ch2);
}
}
```

This program displays the following output:

```
ch1 and ch2: X Y
```

Booleans:

Java has a simple type called **boolean** for logical values. It can have only one of two possible values. They are true or false.

Eg.

```
// Demonstrate boolean values.
class BoolTest {
public static void main(String[] args) {
boolean b;
b = false;
System.out.println("b is " + b);
b = true;
System.out.println("b is " + b);
// a boolean value can control the if statement
if(b) System.out.println("This is executed.");
b = false;
if(b) System.out.println("This is not executed.");

// outcome of a relational operator is a boolean value
System.out.println("10 > 9 is " + (10 > 9));
}
}
```

The output generated by this program is shown here:

```
b is false
b is true
This is executed.
10 > 9 is true
```

Data Type	Default Value	Default size
boolean	False	1 bit
Char	'\u0000'	2 byte
Byte	0	1 byte
short	0	2 byte
Int	0	4 byte
Long	0L	8 byte
Float	0.0f	4 byte

double	0.0d	8 byte

Literals:

A constant value in Java is created by using a literal representation of it. There are 5 types of literals.

1. Integer Literals.
2. Floating-point Literals.
3. Character Literals.
4. String Literals.
5. Boolean Literals.

Integer literals:

- Any whole number value is an integer literal.
- These are all decimal values describing a base 10 number.
- There are two other bases which can be used in integer literal, octal(base 8) where 0 is prefixed with the value, hexadecimal (base 16) where 0X or 0x is prefixed with the integer value.

Example:

int decimal = 100; int octal = 0144; int hexa = 0x64;

Floating point literals:

- The default type when you write a floating-point literal is double, but you can designate it explicitly by appending the D (or d) suffix
- However, the suffix F (or f) is appended to designate the data type of a floating-point literal as float.
- We can also specify a floating-point literal in scientific notation using Exponent (short E ore), for instance: the double literal 0.0314E2 is interpreted as:

Example:

0.0314 *10^2 (i.e 3.14).

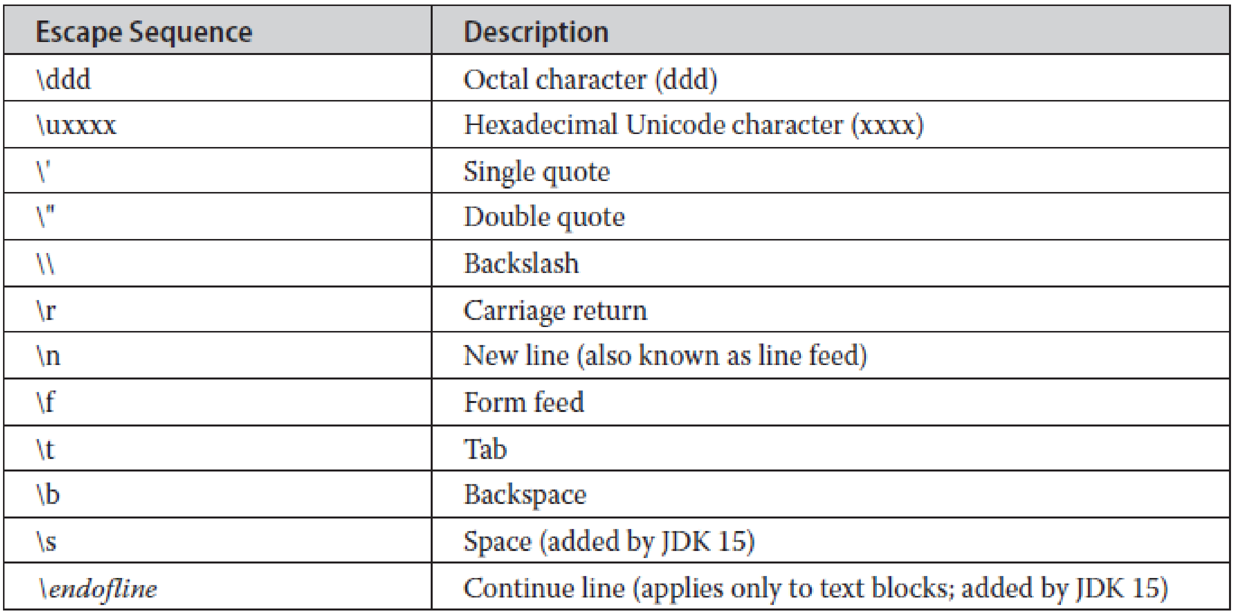
6.5E+32 (or 6.5E32) Double-precision floating-point literal 7D Double-precision floating-point literal

.01f Floating-point literal

Character literals:

- char data type is a single 16-bit Unicode character.
- We can specify a character literal as a single printable character in a pair of single quote characters such as 'a', '#', and '3'.
- You must know about the ASCII character set. The ASCII character set includes 128 characters including letters, numerals, punctuation etc.

Below table shows a set of these special characters.



Escape Sequence	Description
\ddd	Octal character (ddd)
\uxxxx	Hexadecimal Unicode character (xxxx)
\'	Single quote
\"	Double quote
\\\\	Backslash
\r	Carriage return
\n	New line (also known as line feed)
\f	Form feed
\t	Tab
\b	Backspace
\s	Space (added by JDK 15)
endofline	Continue line (applies only to text blocks; added by JDK 15)

Boolean Literals:

- The values true and false are treated as literals in Java programming.
- When we assign a value to a boolean variable, we can only use these two values.
- Unlike C, we can't presume that the value of 1 is equivalent to true and 0 is equivalent to false in Java.
- We have to use the values true and false to represent a Boolean value.

Example
boolean chosen = true;

String Literal

- The set of characters in represented as String literals in Java.
- Always use "double quotes" for String literals.
- There are few methods provided in Java to combine strings, modify strings and to know whether to strings have the same values.

Example:
—hello world‖ —Java‖

Variables:
A variable is an identifier that denotes a storage location used to store a data value. A variable may have different value in the different phase of the program. To declare one identifier as a variable there are certain rules. They are:
1. They must not begin with a digit.

2. Uppercase and lowercase are distinct.

3. It should not be a keyword.

4. White space is not allowed.

Declaring Variable: One variable should be declared before using.
The syntax is
type identifier [= *value*][, *identifier* [= *value*] ...] ; Example:

```
int a,b,c;
float quot, div;
```

Initializing a variable:A variable can be initialize in two ways. They are
(a) Initializing by Assignment statements.

(b) Dynamic Initialisation

Initializing by assignment statements:
□ One variable can be initialize using assignment statements. The syntax is :
Variable-name = Value;
Example: int a=10,b,c=16;
Double pi=3.147;
Dynamic initialization:

- Java allows variables to be initialized dynamically, using expression valid at the time variable is declared.

Example:

```
class Example
{
public static void main(String args[])
{
double a=10, b=2.6; double c=a/b;
System.out.println(—value of c is‖+c);
}
}
```

The Scope and Lifetime of Variables

- Java allows variables to be declared within any block. A block is begun with an opening curly brace and ended by a closing curly brace. A block defines a *scope.*

- A scope determines what objects are visible to other parts of your program. It also determines the lifetime of those objects.

- Many other computer languages define two general categories of scopes: **global** and **local**. However, these traditional scopes do not fit well with Java‗s strict, object-oriented model.

- As a general rule, variables declared inside a scope are not visible (that is, accessible) to code that is defined outside that scope. Thus, when you declare a variable within a scope, you are localizing that variable and protecting it from unauthorized access and/or modification.

```
class Scope
{
public static void main(String args[])
{
int x; // known to all code within main x = 10; if(x == 10) // start new scope
{
int y = 20; // known only to this block
// x and y both known here.
System.out.println("x and y: " + x + " " + y); x = y * 2;
}
// y = 100; // Error! y not known here
// x is still known here. System.out.println("x is " + x);
}
```

}

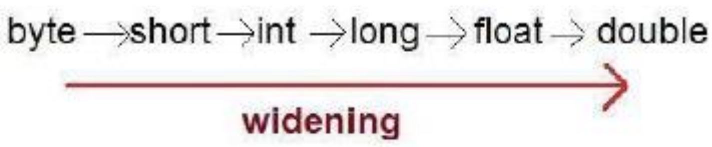
Note:

- ***There should not be two variables with the same name in different scope.***
- ***The variable at outer scope can be accessed in inner scope but vice versa is not possible.***

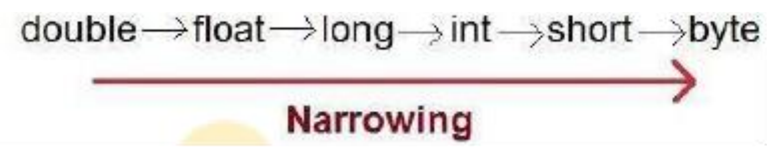
Type Conversion and casting

It is often necessary to store a value of one type into the variable of another type. In these situations, the value that to be stored should be casted to destination type. Assigning a value of one type to a variable of another type is known as **Type Casting. Type** casting can be done in two ways.
In Java, type casting is classified into two types

1. Widening Casting (Implicit)



2. Narrowing casting(Explicitly done)



Widening or Automatic type converion

Automatic Type casting take place when, the two types are compatible
the target type is larger than the source type

Example :

```
public class Test
{
public static void main(String[] args)
{
int i = 100;
long l = i; //no explicit type casting required float f = l; //no explicit type casting required
System.out.println("Int value "+i);
System.out.println("Long value "+l);
 System.out.println("Float value "+f);
}
}
```

Output:

```
Int value 100
Long value 100
Float value 100.0
```

Narrowing or Explicit type conversion

When you are assigning a larger type value to a variable of smaller type, then you need to perform explicit type casting.

Example :

```
public class Test
{
public static void main(String[] args)
{
double d = 100.04;
long l = (long)d; //explicit type casting required int i = (int)l; //explicit type casting required
System.out.println("Double value "+d); System.out.println("Long value "+l); System.out.println("Int
value "+i);
}
}
```

Output :

```
Double value 100.04
Long value 100
Int value 100
```

Automatic type promotion in expressions:

Type conversions also occurs in expressions.

Java automatically promotes each byte, short, or char operand to int when evaluating an expression.

```
byte b = 50;
b = b * 2; // Error! Cannot assign an int to a byte!
```

the operands were automatically promoted to int when the expression was evaluated, the result has also been promoted to int. Thus, the result of the expression is now of type int, which cannot be assigned to a byte without the use of a cast.

byte b = 50;
b = (byte)(b * 2); which yields the correct value of 100.

Java defines several type promotion rules that apply to expressions. They are as follows:

- First, all byte, short, and char values are promoted to int, as just described.
- Then, if one operand is a long, the whole expression is promoted to long.
- If one operand is a float, the entire expression is promoted to float.
- If any of the operands is double, the result is double.

Arrays in Java

Array which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Declaring Array Variables:

To use an array in a program, you must declare a variable to reference the array, and you must specify the type of array the variable can reference.

Syntax for declaring an array variable:

dataType[] arrayRefVar; or dataType arrayRefVar[];

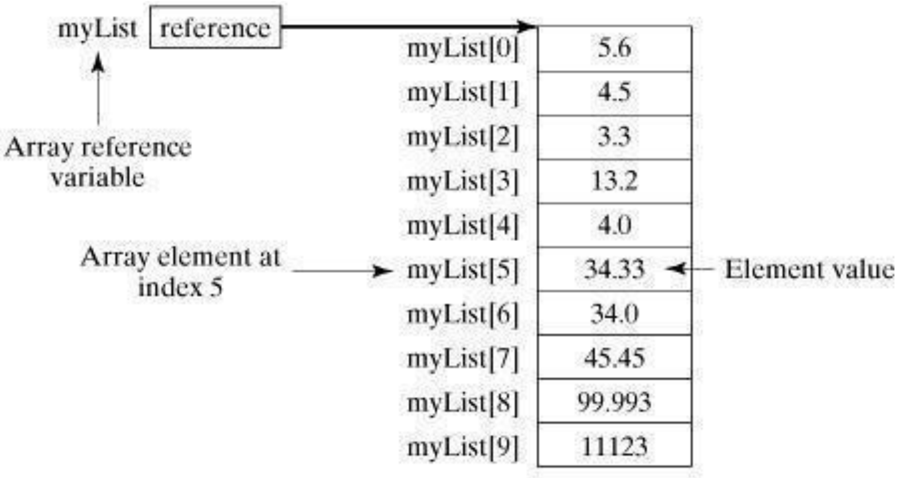
Example:

The following code snippets are examples of this syntax:

int[] myList; or int myList[];

Arrays:
we can create an array by using the new operator with the following syntax:
arrayRefVar = new dataType[arraySize];
The above statement does two things:
It creates an array using new **dataType[arraySize];**
It assigns the reference of the newly created array to the variable **arrayRefVar**.
Declaring an array variable, creating an array, and assigning the reference of the array to the variable can be combined in one statement, as shown below:
dataType[] arrayRefVar = new dataType[arraySize];
Alternatively you can create arrays as follows:
dataType[] arrayRefVar = {value0, value1, ..., valuek};
The array elements are accessed through the **index**. Array indices are 0-based; that is, they start from 0 to **arrayRefVar.length-1**.

Example:
Following statement declares an array variable, myList, creates an array of 10 elements of double type and assigns its reference to myList:
double[] myList = new double[10];
Following picture represents array myList. Here, myList holds ten double values and the indices are from 0 to 9.



Processing Arrays:
When processing array elements, we often use either for loop or foreach loop because all of the elements in an array are of the same type and the size of the array is known.

Example:
Here is a complete example of showing how to create, initialize and process arrays: class TestArray

```
{
public static void main(String[] args)
{
double[] myList = {1.9, 2.9, 3.4, 3.5};
// Print all the array elements for (int i = 0; i < 4; i++)
{
System.out.println(myList[i] + " ");
}
}
```

Multidimensional Arrays

Java does not support multidimensional arrays. However, you can declare and create an array of arrays (and those arrays can contain arrays, and so on, for however many dimensions you need), and access them as you would C-style multidimensional arrays:

```
int coords[] [] = new int[12] [12];

coords[0] [0] = 1; coords[0] [1] = 2;
```

A few words about strings:

- Java supports string type which is an object. It is used to declare string variables
- Array of strings can also be declared.
- A string variable can be assigned to another string variable.
- String variable can also be used as argument. Example:

```
String name1=gautham, name2;
 Name2=name1; // sets name2 withvalue gautham
System.out.println(name2); // string variable passed as parameter.
```

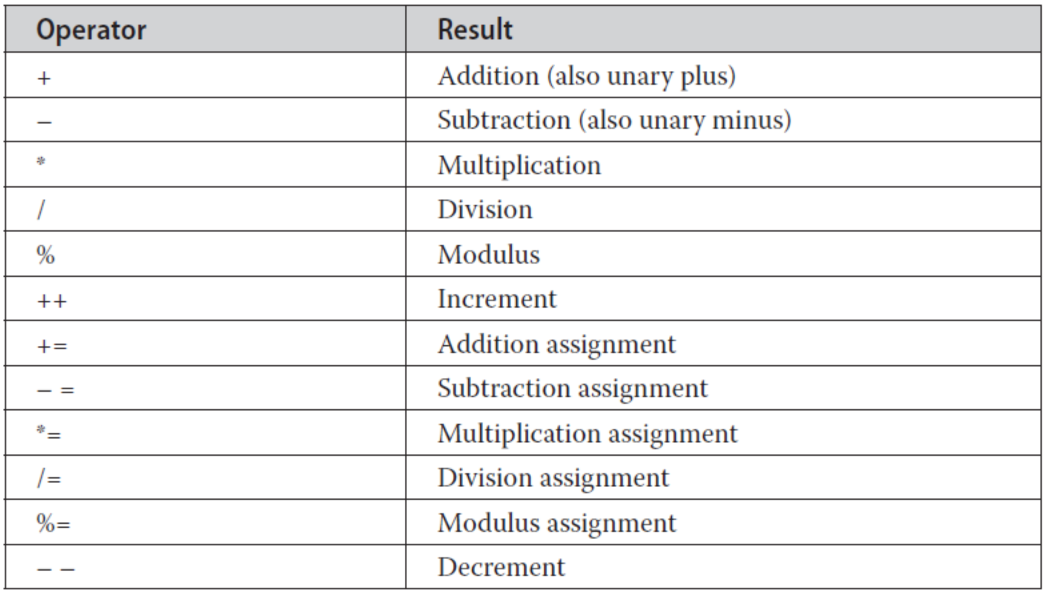
Chapter 4: Operators

Java provides a rich set of operators to manipulate variables. Java operators are divided into the following groups:

- Arithmetic Operators
- Bitwise Operators
- Relational Operators
- Logical Operators
- Assignment Operators

1. Arithmetic Operators

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators:



Operator	Result
+	Addition (also unary plus)
–	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition assignment
– =	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
– –	Decrement

i. Basic Arithmetic operators

The following simple example program demonstrates the arithmetic operators. It also illustrates the difference between floating-point division and integer division.

```
// Demonstrate the basic arithmetic operators.
class BasicMath {
public static void main(String[] args) {
// arithmetic using integers
System.out.println("Integer Arithmetic");
int a = 1 + 1;
int b = a * 3;
int c = b / 4;
int d = c - a;
int e = -d;
System.out.println("a = " + a);
System.out.println("b = " + b);
System.out.println("c = " + c);
System.out.println("d = " + d);
System.out.println("e = " + e);
// arithmetic using doubles
System.out.println("\nFloating Point Arithmetic");
double da = 1 + 1;
double db = da * 3;
double dc = db / 4;
double dd = dc - a;
double de = -dd;
System.out.println("da = " + da);
System.out.println("db = " + db);
System.out.println("dc = " + dc);
System.out.println("dd = " + dd);
System.out.println("de = " + de);
}
}
```

When we run this program, you will see the following output:

```
Integer Arithmetic
a = 2
b = 6
c = 1
d = -1
e = 1
Floating Point Arithmetic
da = 2.0
db = 6.0
dc = 1.5
dd = -0.5
de = 0.5
```

ii. The Modulus Operator

The modulus operator, **%**, returns the remainder of a division operation. It can be applied to floating-point types as well as integer types. The following example program demonstrates the **%**:

```
// Demonstrate the % operator.
class Modulus {
public static void main(String[] args) {
int x = 42;
double y = 42.25;
System.out.println("x mod 10 = " + x % 10);
System.out.println("y mod 10 = " + y % 10);
}
}
```

When we run this program, you will get the following output:

```
x mod 10 = 2
y mod 10 = 2.25
```

iii. Arithmetic Compound Assignment Operators

Java provides special operators that can be used to combine an arithmetic operation with an assignment. Consider a = a + 4;

In Java, you can rewrite this statement as shown here:

a += 4;

This version uses the += *compound assignment operator*. Both statements perform the same action: they increase the value of **a** by 4.

Here is another example,

a = a % 2;

which can be expressed as

a %= 2;

In this case, the **%=** obtains the remainder of **a** /2 and puts that result back into **a**. There are compound assignment operators for all of the arithmetic, binary operators. Thus, any statement of the form

var = var op expression;

can be rewritten as

var op= expression;

```
// Demonstrate several assignment operators.
class OpEquals {
public static void main(String[] args) {
int a = 1;
int b = 2;
int c = 3;
a += 5;
b *= 4;
c += a * b;
c %= 6;
System.out.println("a = " + a);
System.out.println("b = " + b);
System.out.println("c = " + c);
}
}
```

The output of this program is shown here:

```
a = 6
b = 8
c = 3
```

iv. Increment and Decrement

The ++ and the – – are Java's increment and decrement operators. The increment operator increases its operand by one. The decrement operator decreases its operand by one. For example, this statement:

x = x + 1;

can be rewritten like this by use of the increment operator:

x++;

Similarly, this statement:

x = x - 1; is equivalent to x--;

For example:

```
x = 42;
y = ++x;
```

In this case, **y** is set to 43, because the increment occurs *before* **x** is assigned to **y**. Thus, the line **y = ++x**; is the equivalent of these two statements:

```
x = x + 1;
y = x;
```

However, when written like this,

```
x = 42;
y = x++;
```

the value of **x** is obtained before the increment operator is executed, so the value of **y** is 42. **x** is set to 43. Here, the line **y = x++**; is the equivalent of these two statements:

```
y = x;
x = x + 1;
```

The following program demonstrates the increment operator.

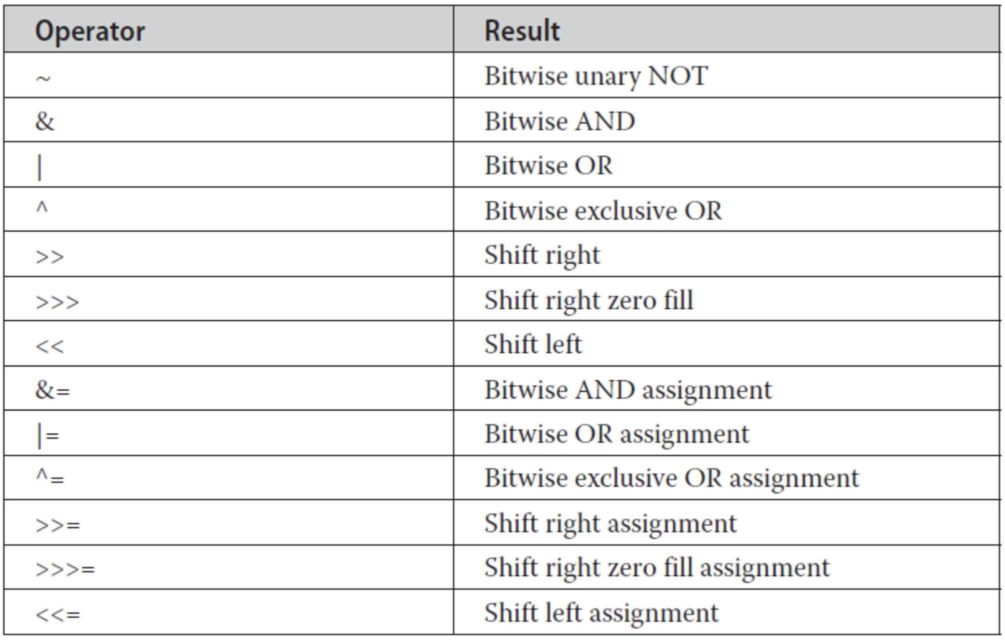
```
// Demonstrate ++.
class IncDec {
public static void main(String[] args) {
int a = 1;
int b = 2;
int c;
int d;
c = ++b;
d = a++;
c++;
System.out.println("a = " + a);
System.out.println("b = " + b);
System.out.println("c = " + c);
System.out.println("d = " + d);
}
}
```

The output of this program follows:

```
a = 2
b = 3
c = 4
d = 1
```

2. The Bitwise Operators

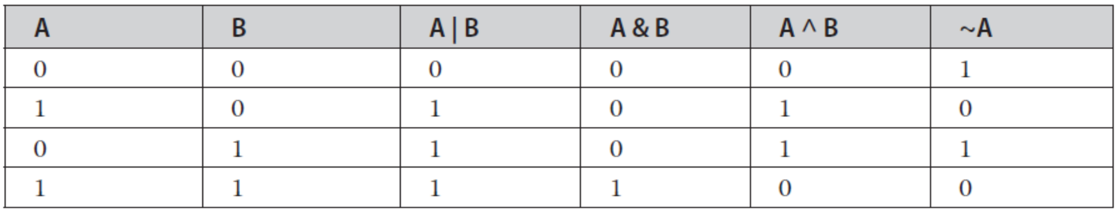
Java defines several *bitwise operators* that can be applied to the integer types: **long**, **int**, **short**, **char**, and **byte**. These operators act upon the individual bits of their operands. They are summarized in the following table:



Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left
&=	Bitwise AND assignment
|=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment

The Bitwise Logical Operators

The bitwise logical operators are **&**, |, ^, and **~**. The following table shows the outcome of each operation.



A	B	A | B	A & B	A ^ B	~A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

The Bitwise NOT

Also called the *bitwise complement*, the unary NOT operator, ~, inverts all of the bits of its operand. For example, the number 42, which has the following bit pattern:

```
 00101010
      becomes
 11010101
   after the NOT operator is applied.
```

The Bitwise AND

The AND operator, **&**, produces a 1 bit if both operands are also 1. A zero is produced in all other cases. Here is an example:

```
 00101010 42
&00001111 15
____________
 00001010 10
```

The Bitwise OR

The OR operator, |, combines bits such that if either of the bits in the operands is a 1, then the resultant bit is a 1, as shown here:

```
 00101010 42
| 00001111 15
____________
 00101111 47
```

The Bitwise XOR

The XOR operator, ^, combines bits such that if exactly one operand is 1, then the result is 1. Otherwise, the result is zero.

```
  00101010 42
^ 00001111 15
___________
  00100101 37
```

The following program demonstrates the bitwise logical operators:

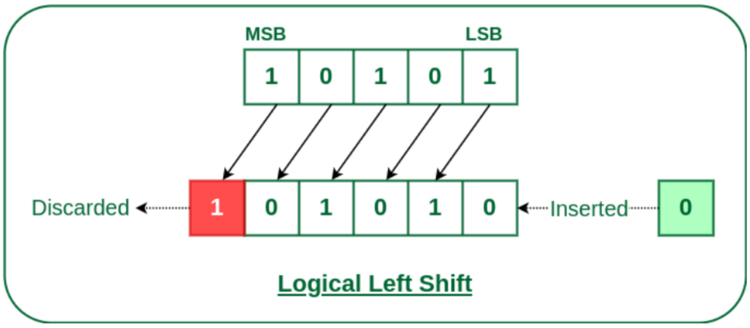
```
// Demonstrate the bitwise logical operators.
class BitLogic {
public static void main(String[] args) {
String[] binary = {
"0000", "0001", "0010", "0011", "0100", "0101", "0110", "0111",
"1000", "1001", "1010", "1011", "1100", "1101", "1110", "1111"
};
int a = 3; // 0 + 2 + 1 or 0011 in binary
int b = 6; // 4 + 2 + 0 or 0110 in binary
int c = a | b;
int d = a & b;
int e = a ^ b;
int f = (~a & b)|(a & ~b);
int g = ~a & 0x0f;
System.out.println(" a = " + binary[a]);
System.out.println(" b = " + binary[b]);
System.out.println(" a|b = " + binary[c]);
System.out.println(" a&b = " + binary[d]);
System.out.println(" a^b = " + binary[e]);
System.out.println("~a&b|a&~b = " + binary[f]);
System.out.println(" ~a = " + binary[g]);
}
}
```

output from this program:

```
a = 0011
b = 0110
a|b = 0111
a&b = 0010
a^b = 0101
~a&b|a&~b = 0101
~a = 1100
```

ii. The Left Shift

The left shift means that shift each of the bits is in binary representation toward the left.



Syntax:

x<<n;

Example program:

```
class GFG {

    public static void main (String[] args) {

        // Number to be shifted
        int x = 5;

        // Number of positions
        int n = 1;

        // Shifting x by n positions towards left using left shift operator
        int answer = x << n;

        System.out.println("Left shift " + x + " by " + n + " positions : " + answer);
}
```

Output:

```
Left shift 5 by 1 positions : 10
```

iii. The Right Shift

The Right Shift Operator moves the bits of a number in a given number of places to the right. The >> sign represents the right shift operator, which is understood as double greater than. When you type x>>n, you tell the computer to move the bits x to the right n places.

When we shift a number to the right, the least significant bits (rightmost) are deleted, and the sign bit is filled in the most considerable place (leftmost).

Syntax:

left_operand >> number

```
class GFG
    {
    public static void main (String[] args) {
        {
        int number = 8;
        // 2 bit signed right shift
        int Ans = number >> 2;

        System.out.println(Ans);
        }
    }
}
```

iv. The Unsigned Right Shift

Unsigned Right Shift Operator moves the bits of the integer a given number of places to the right. The sign bit was filled with 0s. The Bitwise Zero Fill Right Shift Operator is represented by the symbol >>>.

Syntax:

```
left_operand  >>>  number
class GFG
    {
    public static void main (String[] args)
    {
        byte num1 = 8;
        byte num2 = -8;

        System.out.println(num1 >>> 2);
        System.out.println(num2 >>> 2);
    }
}
```

Output:

```
2
1073741822
```

v. Bitwise Operator Compound Assignments

All of the binary bitwise operators have a compound form similar to that of the algebraic operators, which combines the assignment with the bitwise operation. For example, the following two statements, which shift the value in **a** right by four bits, are equivalent:

a = a >> 4;

a >>= 4;

Likewise, the following two statements, which result in **a** being assigned the bitwise expression **a** OR **b**, are equivalent:

a = a | b;

a |= b;

Example program

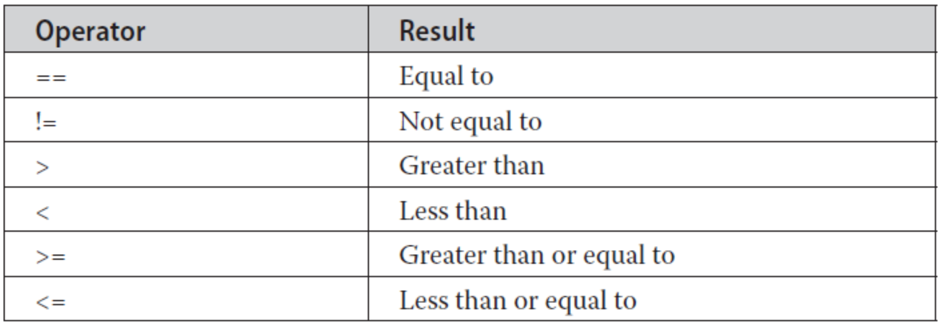
```
class OpBitEquals {
public static void main(String[] args) {
int a = 1;
int b = 2;
int c = 3;
a |= 4;
b >>= 1;
c <<= 1;
a ^= c;
System.out.println("a = " + a);
System.out.println("b = " + b);
System.out.println("c = " + c);
}
}
```

The output of this program is shown here:

```
a = 3
b = 1
c = 6
```

3. Relational Operators

The *relational operators* determine the relationship that one operand has to the other. Specifically, they determine equality and ordering. The relational operators are shown here:



Operator	Result
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

The outcome of these operations is a **boolean** value. The relational operators are most frequently used in the expressions that control the **if** statement and the various loop statements.

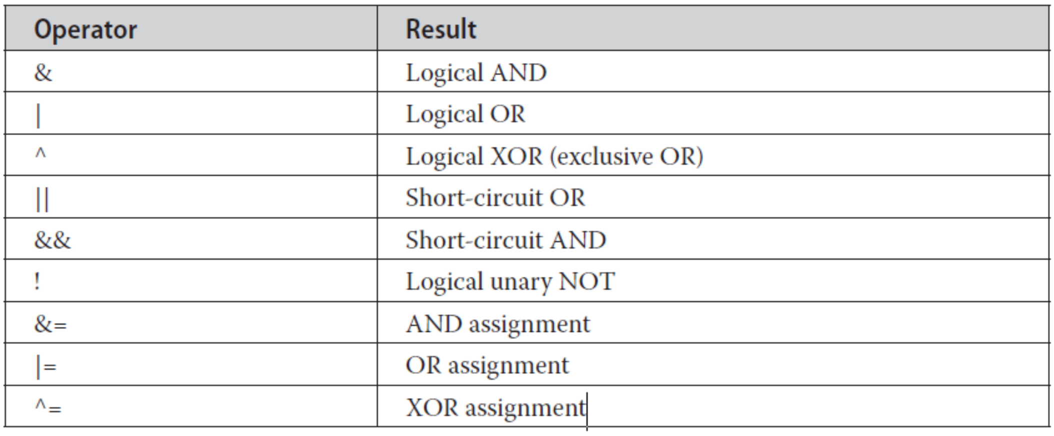
```
public class RelationalOperators {
            public static void main(String[] args) {
                        int a = 10;
                        int b = 20;
                        System.out.println(a == b);
                        System.out.println(a != b);
                        System.out.println(a > b);
                        System.out.println(a < b);
                        System.out.println(a >= b);
                        System.out.println(a <= b);
            }
}
```

Output:

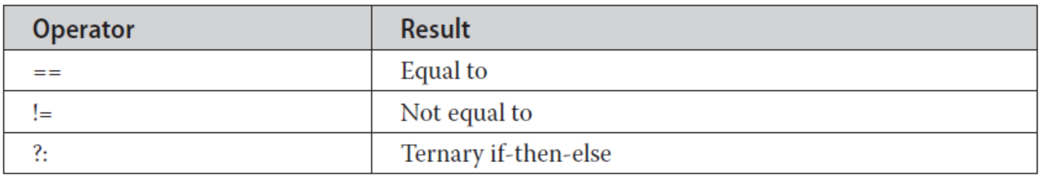
```
False
True
False
True
False
True
```

4. **Boolean Logical Operators**

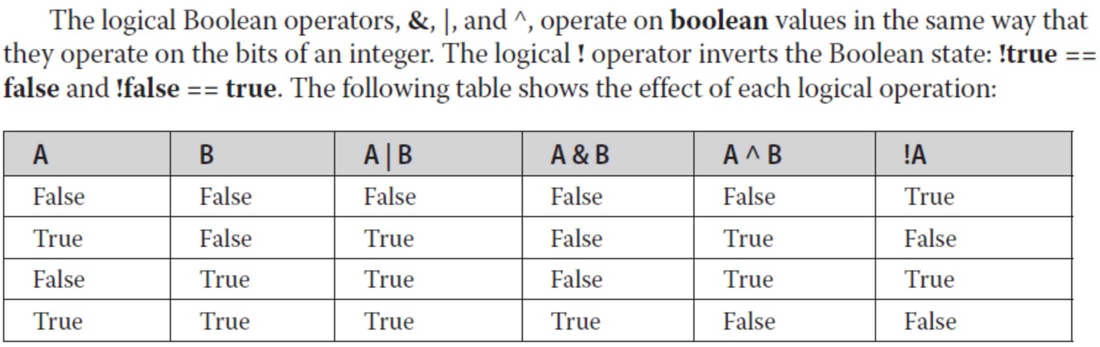
The Boolean logical operators shown here operate only on boolean operands. All of the binary logical operators combine two boolean values to form a resultant boolean value.



Operator	Result
&	Logical AND
|	Logical OR
^	Logical XOR (exclusive OR)
||	Short-circuit OR
&&	Short-circuit AND
!	Logical unary NOT
&=	AND assignment
|=	OR assignment
^=	XOR assignment



Operator	Result
==	Equal to
!=	Not equal to
?:	Ternary if-then-else



The logical Boolean operators, **&**, |, and ^, operate on **boolean** values in the same way that they operate on the bits of an integer. The logical **!** operator inverts the Boolean state: **!true** == **false** and **!false** == **true**. The following table shows the effect of each logical operation:

A	B	A | B	A & B	A ^ B	!A
False	False	False	False	False	True
True	False	True	False	True	False
False	True	True	False	True	True
True	True	True	True	False	False

Here is a program that is almost the same as the **BitLogic** example shown earlier, but it operates on **boolean** logical values instead of binary bits

```
// Demonstrate the boolean logical operators.
class BoolLogic {
public static void main(String[] args) {
boolean a = true;
boolean b = false;
boolean c = a | b;
boolean d = a & b;
boolean e = a ^ b;
boolean f = (!a & b) | (a & !b);
boolean g = !a;
System.out.println(" a = " + a);
System.out.println(" b = " + b);
System.out.println(" a|b = " + c);
System.out.println(" a&b = " + d);
System.out.println(" a^b = " + e);
System.out.println("!a&b|a&!b = " + f);
System.out.println(" !a = " + g);
}
}
```

Output:

```
a = true
b = false
a|b = true
a&b = false
a^b = true
!a&b|a&!b = true
!a = false
```

Short-Circuit Logical Operators

A short circuit operator is a logical operator that lets in a programmer to improve the performance of a program.

The "&&" operator is used for "AND" operations. It tests if each the left-hand side and the proper-hand side of the operator are authentic.

```
public class ShortCircuitAndOperatorExample {
public static void main(String[] args) {
    int a = 10;
    int b = 5;
    if (a > 5 && b < 10) {
      System.out.println("Both conditions are true.");
    } else {
      System.out.println("At least one condition is false.");
    }
  }
}
```

Output:

```
        "Both conditions are true."
```

The "||" operator is used for "OR" operations. It checks if either the left-hand side or the right-hand side of the operator is true. If either operand is true, then the expression is true.

```
public class ShortCircuitOrOperatorExample {
  public static void main(String[] args) {
    int a = 10;
    int b = 5;
    if (a > 5 || b > 10) {
      System.out.println("At least one condition is true.");
```

```
        } else {
            System.out.println("Both conditions are false.");
        }
    }
}
```

Output:

"At least one condition is true."

5. The Assignment Operator

The *assignment operator* is the single equal sign, =

It has this general form:

var = *expression*;

Here, the type of *var* must be compatible with the type of *expression*.

The assignment operator allows us to create a chain of assignments. For example, consider this fragment:

```
int x, y, z;
x = y = z = 100; // set x, y, and z to 100
```

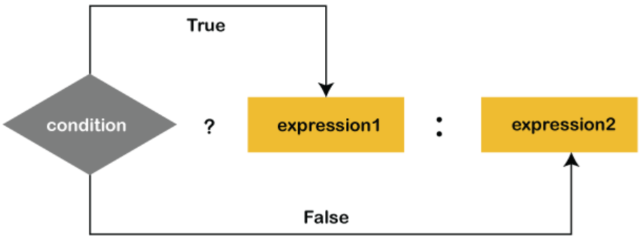
This fragment sets the variables **x**, **y**, and **z** to 100 using a single statement. This works because the = is an operator that yields the value of the right-hand expression. Thus, the value of **z = 100** is 100, which is then assigned to **y**, which in turn is assigned to **x**. Using a "chain of assignment" is an easy way to set a group of variables to a common value.

i. The ? Operator

Java includes a special *ternary* (three-way) *operator* that can replace certain types of if-then else statements. This operator is the **?**. The **?** has this general form:

expression1 ? *expression2* : *expression3*

Here, *expression1* can be any expression that evaluates to a **boolean** value. If *expression1* is **true**, then *expression2* is evaluated; otherwise, *expression3* is evaluated. The result of the **?** operation is that of the expression evaluated. Both *expression2* and *expression3* are required to return the same (or compatible) type, which can't be **void**.



```
public class TernaryOperatorExample
{
public static void main(String args[])
{
int x, y;
x = 20;
y = (x == 1) ? 61: 90;
System.out.println("Value of y is: " +  y);
y = (x == 20) ? 61: 90;
System.out.println("Value of y is: " + y);
}
}
```

Output:

```
        Value of y is: 90
        Value of y is: 61
```

Operator Precedence

Table 4-1 shows the order of precedence for Java operators, from highest to lowest. Operators in the same row are equal in precedence. In binary operations, the order of evaluation is left to right (except for assignment, which evaluates right to left). Although they are technically separators, the **[ ]**, **( )**, and **.** can also act like operators. In that capacity, they would have the highest precedence. Also, notice the arrow operator (->). It is used in lambda expressions.

Using Parentheses

Parentheses raise the precedence of the operations that are inside them. This is often necessary to obtain the result you desire. For example, consider the following expression:

a >> b + 3

This expression first adds 3 to **b** and then shifts **a** right by that result. That is, this expression can be rewritten using redundant parentheses like this:

a >> (b + 3)

If you want to first shift **a** right by **b** positions and then add 3 to that result, you will need to parenthesize the expression like this:

(a >> b) + 3

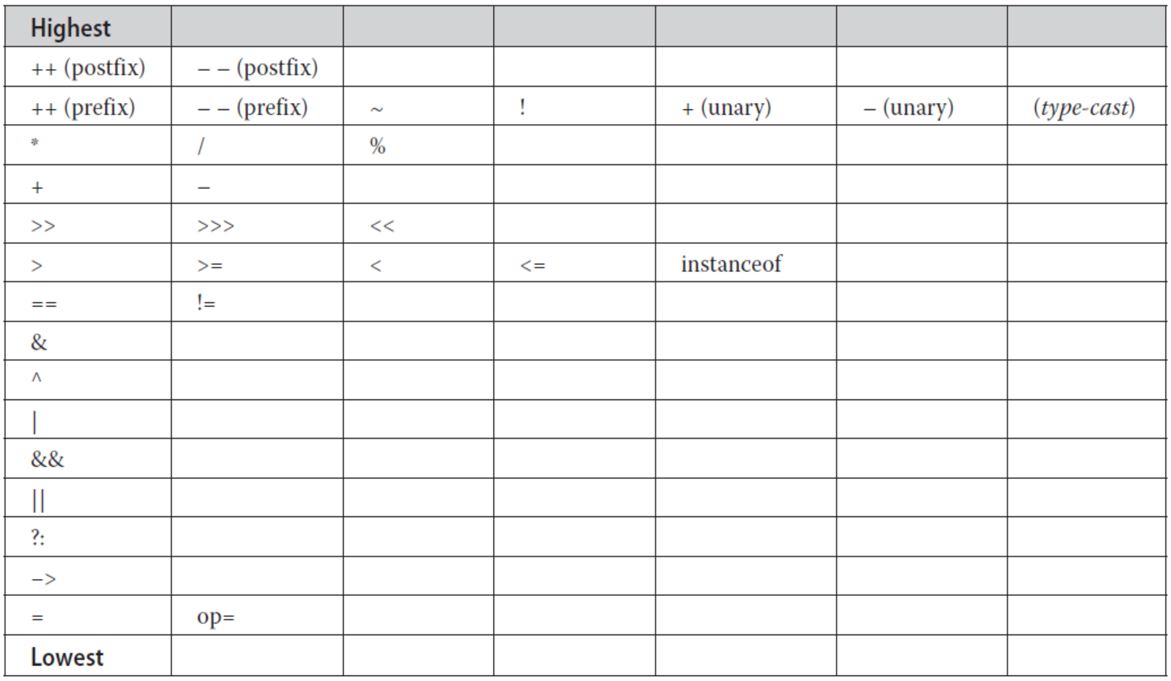
parentheses helps to prevent confusion.

For example, which of the following expressions is easier to read?

a | 4 + c >> b & 7

(a | (((4 + c) >> b) & 7))

parentheses do not degrade the performance of your program. Therefore, adding parentheses to reduce ambiguity does not negatively affect your program.



Highest						
++ (postfix)	– – (postfix)					
++ (prefix)	– – (prefix)	~	!	+ (unary)	– (unary)	(*type-cast*)
*	/	%				
+	–					
>>	>>>	<<				
>	>=	<	<=	instanceof		
==	!=					
&						
^						
|						
&&						
||						
?:						
–>						
=	op=					
Lowest						

Table 4-1 The Precedence of the Java Operators

Chapter 5: Control Statements

- Java’s program control statements can be put into the following categories: selection, iteration, and jump.
- *Selection* statements allow your program to choose different paths of execution based upon the outcome of an expression or the state of a variable.
- *Iteration* statements enable program execution to repeat one or more statements (that is, iteration statements form loops).
- *Jump* statements allow your program to execute in a nonlinear fashion.

Java’s Selection Statements

- Java supports two selection statements: **if** and **switch**.

The if statement

- The **if** statement executes a block of code only if the specified expression is true.
- If the value is false, then the **if** block is skipped and execution continues with the rest of the program.
- You can either have a single statement or a block of code within an **if** statement.
- Note that the conditional expression must be a Boolean expression.

Syntax:

```
if (<conditional expression>) {
<statements>
}
```

Example:

```java
public class Example
{
        public static void main(String[] args) {
                int a = 10, b = 20;
                if (a > b)
                        System.out.println("a > b");
                if (a < b)
                        System.out.println("b > a");
        }
}
```

The if else statement

- The **if** statement is Java’s conditional branch statement. It can be used to route program execution through two different paths.
- Here   is the general form of the **if** statement:

Syntax:

```
if (condition)
statement1; else statement2;
```

```
public class Example {
        public static void main(String[] args) {
                int a = 10, b = 20;
                if (a > b)
                        System.out.println("a > b");
               else
```

- Here, each *statement* may be a single statement or a compound statement enclosed in curly braces (that is, a *block*).
- The *condition* is any expression that returns a **boolean** value. The **else** clause is optional.
- The **if** works like this: If the *condition* is true, then *statement1* is executed. Otherwise, *statement2* (if it exists) is executed.

Example:

```
public class Example {
        public static void main(String[] args) {
                int a = 10, b = 20;
                if (a > b)
                        System.out.println("a > b");
               else
                System.out.println("b > a");

                         }
```

Nested ifs

- A *nested* **if** is an **if** statement that is the target of another **if** or **else**.
- When you nest **if**s, the main thing to remember is that an **else** statement always refers to the nearest **if** statement that is within the same block as the **else** and that is not already associated with an **else**.

Here is an example:

```
if(i == 10) {
                        if(j < 20) a = b;
                        if(k > 100) c = d; // this if is
        else a = c; // associated with this else
}
        else a = d; // this else refers to if(i == 10)
```

The if-else-if Ladder

- A common programming construct that is based upon a sequence of nested **if**s is the *if-else-if ladder*.

It looks like this:

- The **if** statements are executed from the top down.
- As soon as one of the conditions controlling the **if** is **true**, the statement associated with that **if**

is executed, and the rest of the ladder is bypassed.

- If none of the conditions is true, then the final **else** statement will be executed.

Example:

```
class IfElse {
        public  static  void  main(String  args[])  {  int
                month = 4; // April
                String season;
                if(month == 12 || month == 1 || month == 2) season
                        = "Winter";
                else if(month == 3 || month == 4 || month == 5) season
                        = "Spring";
                else if(month == 6 || month == 7 || month == 8) season
                        = "Summer";
                else if(month == 9 || month == 10 || month == 11) season
                        = "Autumn";
                else
                        season = "Bogus Month";
        System.out.println("April is in the " + season + ".");
        }
}
```

The switch statement

- The **switch case** statement is a multi-way branch with several choices. A switch is easier to implement than a series of if/else statements.
- The switch statement begins with a keyword, followed by an expression that equates to a no long integral value.
- Following the controlling expression is a code block that contains zero or more labeled cases.
- Each label must equate to an integer constant and each must be unique.

Working of switch case:

- When the switch statement executes, it compares the value of the controlling expression to the values of each case label.
- The program will select the value of the case label that equals the value of the controlling expression and branch down that path to the end of the code block.
- If none of the case label values match, then none of the codes within the switch statement code block will be executed. Java includes a **default** label to use in cases where there are no matches.
- We can have a nested switch within a case block of an outer switch.

Syntax:

```
switch (<non-long integral expression>) { case

label1: <statement1> ; break;

case label2: <statement2> ; break;

...

case labeln: <statementn> ; break; default:

<statement>

}
```

Example:

```
 public class Example {
        public static void main(String[] args) {
               int a = 10, b = 20, c = 30;
               int status = -1;
               if (a > b && a > c) {
                      status = 1;
               } else if (b > c) { status = 2;
               } else {
                      status = 3;
}
        switch (status) {
               case 1: System.out.println("a is the greatest");
               break;
              case 2:  System.out.println("b is the greatest");                                }
                     break;                                                              }
              case 3: System.out.println("c is the greatest");
                     break;
         default: System.out.println("Cannot be determined");
                      }
```

- The **break** statement is optional. If you omit the **break**, execution will continue on into the next **case**.
- It is sometimes desirable to have multiple **case**s without **break** statements between them.
- For example, consider the following program:

```
 // In a switch, break statements are optional.
   class MissingBreak {
        public static void main(String args[]) { for(int
               i=0; i<12; i++)  switch(i) {
                      case 0:
                      case 1:
                      case 2:
                      case 3:
                      case 4:
                             System.out.println("i is less than 5"); break;
                      case 5:
                      case 6:
                      case 7:
                      case 8:
                      case 9:
```

```
                    System.out.println("i is less than 10"); break;
                default:
                System.out.println("i is 10 or more");
            }
        }
}
```

Nested switch Statements

- You can use a **switch** as part of the statement sequence of an outer **switch**. This is called a *nested* **switch**.
- Since a **switch** statement defines its own block, no conflicts arise between the **case** constants in the inner **switch** and those in the outer **switch**.
- For example, the following fragment is perfectly valid:

```
switch(count) {
        case 1:
            switch(target) { // nested switch case 0:
                    System.out.println("target is zero"); break;
                case 1: // no conflicts with outer switch
                    System.out.println("target is one"); break;
            }
            break;  case  2:
        // ...
```

In summary, there are three important features of the **switch** statement to note:

- The **switch** differs from the **if** in that **switch** can only test for equality, whereas **if** can evaluate any type of Boolean expression. That is, the **switch** looks only for a match between the value of the expression and one of its **case** constants.
- No two **case** constants in the same **switch** can have identical values. Of course, a **switch** statement and an enclosing outer **switch** can have **case** constants in common.
- A **switch** statement is usually more efficient than a set of nested **if**s.

Iteration Statements

The while loop

- The **while** statement is a looping construct control statement that executes a block of code while a condition is true.
- You can either have a single statement or a block of code within the while loop.
- The loop will never be executed if the testing expression evaluates to false.
- The loop condition must be a **boolean** expression.

Syntax:

```
while (<loop condition>) {
<statements>
}
```

Example:

```
public class Example {
        public static void main(String[] args) {
                int count = 1;
                System.out.println("Printing Numbers from 1 to 10");
                while (count <= 10) {
                        System.out.println(count++);
                }
        }
}
```

The do-while loop

- The **do-while** loop is similar to the **while** loop, except that the test is performed at the end of the loop instead of at the beginning.
- This ensures that the loop will be executed at least once.

Syntax:

```
do {
<loop body>
} while (<loop condition>);
```

Example:

```
 public class Example {
        public static void main(String[] args) {
                int count = 1;
                System.out.println("Printing Numbers from 1 to 10");
                do {
                        System.out.println(count++);
                } while (count <= 10);
        }
}
```

The for loop

- The **for** loop is a looping construct which can execute a set of instructions a specified number of times. It's a counter controlled loop.

Syntax:

```
for (<initialization>; <loop condition>; <increment expression>) {
<loop body>
```

Example:

```
 public class Example {
        public static void main(String[] args) { System.out.println("Printing
                Numbers from 1 to 10"); for (int count = 1; count <= 10;
                count++) {
                        System.out.println(count);
                }
        }
}
```

Declaring Loop Control Variables Inside the for Loop

- Often the variable that controls a **for** loop is only needed for the purposes of the loop and is not used elsewhere.
- When this is the case, it is possible to declare the variable inside the initialization portion of the **for**.

```
class ForTick {
    public static void main(String args[]) {
            // here, n is declared inside of the for loop for(int
            n=10; n>0; n--)
                    System.out.println("tick " + n);
    }
}
```

- When you declare a variable inside a **for** loop, there is one important point to remember: the scope of that variable ends when the **for** statement does

Using the Comma

- There will be times when you will want to include more than one statement in the initialization and iteration portions of the **for** loop.

```
class Comma {
    public static void main(String args[]) { int a, b;
            for(a=1, b=4; a<b; a++, b--) {
            System.out.println("a = " + a);
            System.out.println("b = " + b);
            }
    }
}
```

Some for Loop Variations

- The **for** loop supports a number of variations that increase its power and applicability. The reason it is so flexible is that its three parts—the initialization, the conditional test, and the iteration—do not need to be used for only those purposes can be used for any purpose you desire.
- One of the most common variations involves the conditional expression.
- Specifically, this expression does not need to test the loop control variable against some target value. In fact, the condition controlling the **for** can be any Boolean expression. For example, consider the following fragment:

```
boolean done = false; for(int i=1;
!done; i++) {
    // ...
    if(interrupted()) done = true;
}
```

In this example, the **for** loop continues to run until the **boolean** variable **done** is set to **true**. It does not test the value of **i**.

- Here is another interesting **for** loop variation. Either the initialization or the iteration expression or both may be absent, as in this next program:

```
// Parts of the for loop can be empty. class
ForVar {
    public static void main(String args[]) {
            int i;
            boolean done = false; i = 0;
            for( ; !done; ) {
                    System.out.println("i is " + i);
                    if(i == 10) done = true;
                    i++;
            }
    }
}
```

Here, the initialization and iteration expressions have been moved out of the **for**. Thus, parts of the **for** are empty

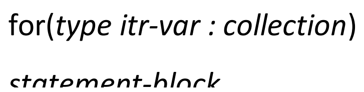
- Here is one more **for** loop variation. You can intentionally create an infinite loop (a loop that never terminates) if you leave all three parts of the **for** empty.
- For example:

```
for( ; ; ) {
// ...
}
```

This loop will run forever because there is no condition under which it will terminate.

The For-Each Version of the for Loop

- Beginning with JDK 5, a second form of **for** was defined that implements a "for- each" style loop.
- The general form of the for-each version of the **for** is shown here:



```
for(type itr-var : collection)
statement-block
```

- Here, *type* specifies the type and *itr-var* specifies the name of an *iteration variable* that will receive the elements from a collection, one at a time, from beginning to end.
- The collection being cycled through is specified by *collection*.
- There are various types of collections that can be used with the **for**, but the only type used in this chapter is the array.

Working:

- With each iteration of the loop, the next element in the collection is retrieved and stored in *itr-var*.
- The loop repeats until all elements in the collection have been obtained.
- Because the iteration variable receives values from the collection, *type* must be the same as (or compatible with) the elements stored in the collection.
- Thus, when iterating over arrays, *type* must be compatible with the base type of the array.

```
class ForEach {
    public static void main(String args[]) {
        int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };    int sum = 0; for(int x : nums) {
            sum += x;
        }
        System.out.println("Summation: " + sum);
    }
}
```

- With each pass through the loop, **x** is automatically given a value equal to the next element in **nums**. Thus, on the first iteration, **x** contains 1; on the second iteration, **x** contains 2; and so on.
- Not only is the syntax streamlined, but it also prevents boundary errors.

For example, this program sums only the first five elements of **nums**: class

```
    ForEach2 {
        public static void main(String args[]) { int sum
            = 0;
            int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
            // use for to display and sum the values for(int x
            : nums) {
                sum += x;
                if(x == 5) break; // stop the loop when 5 is obtained
            }
            System.out.println("Summation of first 5 elements: " + sum);
        }
    }
```

Iterating Over Multidimensional Arrays

- The enhanced version of the **for** also works on multidimensional arrays.
- Remember, however, that in Java, multidimensional arrays consist of *arrays of arrays*. (For example, a two-dimensional array is an array of one-dimensional arrays.)

```
class ForEach3 {
    public static void main(String args[]) { int sum
        = 0;
        int nums[][] = new int[3][5];
        // give nums some values for(int i
        = 0; i < 3; i++)
            for(int j=0; j < 5; j++)
                nums[i][j] = (i+1)*(j+1);
        // use for-each for to display and sum the values for(int
        x[] : nums) {
            for(int y : x) {
                sum += y;
            }
        }
        System.out.println("Summation: " + sum);
    }
}
```

- In the program, pay special attention to this line: for(int
  x[] : nums)
- Notice how **x** is declared. It is a reference to a one-dimensional array of integers.
- This is necessary because each iteration of the **for** obtains the next *array* in **nums**, beginning with the array specified by **nums[0]**.
- The inner **for** loop then cycles through each of these arrays, displaying the values of each element.

Java program to search given key element

```
class Search {
        public static void main(String args[]) {
                int nums[] = { 6, 8, 3, 7, 5, 6, 1, 4 };
                int val = 5;
                boolean found = false;
                // use for-each style for to search nums for val for(int x
                : nums) {
                        if(x == val) {
                                found = true; break;
                        }
                }
                if(found)
                        System.out.println("Value found!");
        }
}
```

Nested Loops

- Like all other programming languages, Java allows loops to be nested.
- That is, one loop may be inside another. For example, here is a program that nests **for** loops: class

```
Nested {
        public static void main(String args[]) { int i, j;
                for(i=0; i<10; i++) {
                        for(j=i; j<10; j++)
                                System.out.print(".");
                        System.out.println();
                }
        }
}
```

Jump Statements

- Java supports three jump statements: **break**, **continue**, and **return**. These statements transfer control to another part of your program.

The break statement

- The **break** statement transfers control out of the enclosing loop (for, while, do or switch statement).
- You use a **break** statement when you want to jump immediately to the statement following the

enclosing control structure.

- You can also provide a loop with a label, and then use the label in your **break** statement.
- The label name is optional, and is usually only used when you wish to terminate the outermost loop in a series of nested loops.

Syntax:

```
break; // the unlabeled form

break <label>; // the labeled form
```

Example for break:

```
public class Example {
        public    static    void    main(String[]    args)    {
                System.out.println("Numbers 1 - 10"); for (int i
                = 1;; ++i) {
                        if (i == 11)
                                break; System.out.println(i +
                        "\t");
                }
        }
}
```

Example for labeled break:

```
class Break {
        public static void main(String args[]) { boolean
                t = true;
                first: {
                        second: {
                                third: {

                                            System.out.println("Before the break.");
                                            if(t) break second; // break out of second block
                                            System.out.println("This won't execute");
                                            }
                                            System.out.println("This won't execute");

                                }
        System.out.println("This is after second block.");
                }
        }
}
```

Running this program generates the following output: Before the break.

This is after second block.

The continue statement

- A **continue** statement stops the iteration of a loop (while, do or for) and causes execution to resume at the top of the nearest enclosing loop.
- You use a **continue** statement when you do not want to execute the remaining statements in the loop, but you do not want to exit the loop itself.
- You can also provide a loop with a label and then use the label in your **continue** statement.
- The label name is optional, and is usually only used when you wish to return to the outermost loop in a series of nested loops.

  Syntax:

  ```
  continue; // the unlabeled form

  continue <label>; // the labeled form
  ```

Example for continue:

```
public class Example {
        public static void main(String[] args) {
                System.out.println("Odd Numbers"); for (int
                i = 1; i <= 10; ++i) {
                        if (i % 2 == 0)
                                continue;
                        System.out.println(i + "\t");
                }
        }
}
```

Example for labeled continue:

```
class ContinueLabel {
        public static void main(String args[]) { outer:
                for (int i=0; i<10; i++) { for(int j=0;
                j<10; j++) {
                                if(j > i) {
                                                System.out.println();
                                                continue outer;
                                        }
                                System.out.println(" " + (i * j));
                        }
                }
                System.out.println();
        }
}
```

The return statement

- The **return** statement exits from the current method, and control flow returns to where the method was invoked.

Syntax:

The **return** statement has two forms:

One that returns a value

return val;

One that doesn't returns a value

return;

Example:

```
public class Example {
        public static void main(String[] args) {
                int res = sum(10, 20);
                        System.out.println(res);
        }
                private static int sum(int a, int b) {
                        return (a + b);
                }
}
```

MODULE 2

Chapter 6: Introducing Classes

- The class is at the core of Java.
- It is the logical construct upon which the entire Java language is built because it defines the shape and nature of an object.

Class Fundamentals

- Class defines a new data type. Once defined, this new type can be used to create objects of that type.
- Thus, a class is a *template* for an object, and an object is an *instance* of a class. Because an object is an instance of a class, you will often see the two words *object* and *instance* used interchangeably.
- Class is a collection of data members and member functions.

The General Form of a Class

- When you define a class, you declare its exact form and nature. You do this by specifying the data that it contains and the code that operates on that data.
- A class is declared by use of the **class** keyword

```
class classname {
type instance-variable1; type instance-variable2;
// ...
type instance-variableN;
type methodname1(parameter-list) {
// body of method
}
type methodnameN(parameter-list) {
// body of method
}
}
```

- The data, or variables, defined within a **class** are called *instance variables.*
- The code is contained within *methods.*

- Collectively, the methods and variables defined within a class are called *members* ofthe class.
- Variables defined within a class are called instance variables because each instance ofthe class (that is, each object of the class) contains its own copy of these variables.
- Thus, the data for one object is separate and unique from the data for another.

A Simple Class

- Here is a class called **Box** that defines three instance variables: **width**, **height**, and ***depth***.

```
        class Box {
                    double width; double height;double depth;

                   }
```

- As stated, a class defines a new type of data.
- In this case, the new data type is called **Box**.
- You will use this name to declare objects of type **Box**.
- It is important to remember that a **class** declaration only creates a template; it does notcreate an actual object

```
                        Box mybox = new Box(); // create a Box object called mybox
```

- After this statement executes, **mybox** will be an instance of **Box**.
- Thus, it will have "physical" reality.
- Thus, every **Box** object will contain its own copies of the instance variables **width**, **height**, and **depth**.
- To access these variables, you will use the *dot* (.) operator.
- The dot operator links the name of the object with the name of an instance variable. For

example, to assign the **width** variable of **mybox** the value 100, you would use the following statement:

```
              mybox.width = 100;
              class Box {
                          double width; double height;double depth;
                         }
// This class declares an object of type Box.class
BoxDemo {
     public static void main(String args[]) {
```

```
                Box mybox = new Box(); double vol;    // assign values to mybox's instance
                  variablesmybox.width = 10;
                  mybox.height = 20;
                  mybox.depth = 15; /
                  vol = mybox.width * mybox.height * mybox.depth;
                  System.out.println("Volume is " + vol);
    }
  }
```

Declaring Objects

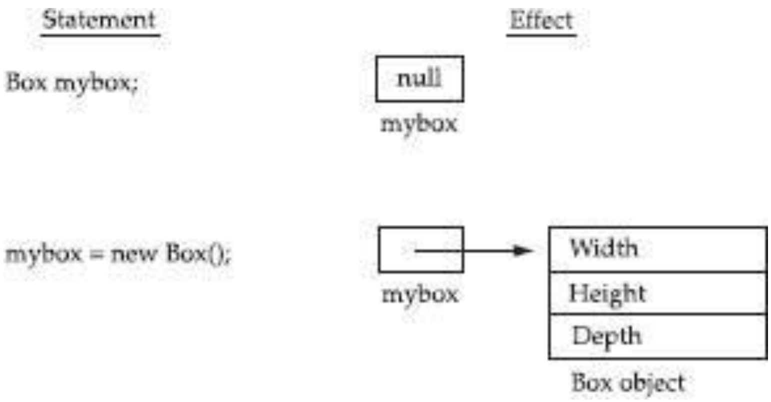
- When you create a class, you are creating a new data type.
- However, obtaining objects of a class is a two-step process.
- First, you must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can *refer* to an object.
- Second, you must acquire an actual, physical copy of the object and assign it to that variable. You can do this using the **new** operator.
- The **new** operator dynamically allocates (that is, allocates at run time) memory for anobject and returns a reference to it.
- This reference is, more or less, the address in memory of the object allocated by **new**.This reference is then stored in the variable.
- Thus, in Java, all class objects must be dynamically allocated.

```
Box mybox = new Box();
```

- This statement combines the two steps just described. It can be rewritten like this toshow each step more clearly:

```
Box mybox; // declare reference to objectmybox
= new Box(); // allocate a Box object
```

- The first line declares **mybox** as a reference to an object of type **Box**.
- After this line executes, **mybox** contains the value **null**, which indicates that it doesnot yet point to an actual object.
- Any attempt to use **mybox** at this point will result in a compile-time error. The nextline allocates an actual object and assigns a reference to it to **mybox**.
- After the second line executes, you can use **mybox** as if it were a **Box** object. But in reality, **mybox** simply holds the memory address of the actual **Box** object.



- Here, *class-var* is a variable of the class type being created. The *classname* is thename of the class that is being instantiated.
- The class name followed by parentheses specifies the *constructor* for the class.
- A constructor defines what occurs when an object of a class is created.
- Constructors are an important part of all classes and have many significant attributes.
- It is important to understand that **new** allocates memory for an object during run time.
- The advantage of this approach is that your program can create as many or as few objects as it needs during the execution of your program.
- However, since memory is finite, it is possible that **new** will not be able to allocate memory for an object because insufficient memory exists.
- If this happens, a run-time exception will occur.
- A class creates a new data type that can be used to create objects.That is, a class creates a logical framework that defines the relationship between its members. When you declare an object of a class, you are creating an instance of that class.
- Thus, a class is a logical construct. An object has physical reality.

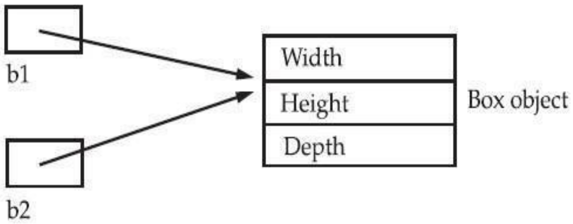
Assigning Object Reference Variables

Object reference variables act differently when an assignment takes place.

Box b1 = new Box();Box

b2 = b1;

This situation is depicted here:

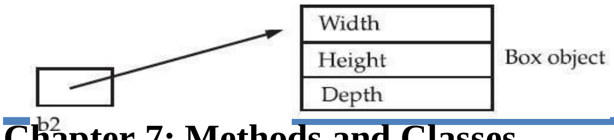


- After this fragment executes, **b1** and **b2** will both refer to the *same* object.
- The assignment of **b1** to **b2** did not allocate any memory or copy any part of theoriginal object.

- It simply makes **b2** refer to the same object as does **b1**.
- Thus, any changes made to the object through **b2** will affect the object to which **b1** is referring, since they are the same object.
- Although **b1** and **b2** both refer to the same object, they are not linked in any otherway.

```
Box b1 = new Box(); Box b2 = b1;
// ...
b1 = null;
```

Here, **b1** has been set to **null**, but **b2** still points to the original object.



Chapter 7: Methods and Classes

Introducing methods:

This is the general form of a method:

```
type name(parameter-list) {
// body of method
}
```

- Here, *type* specifies the type of data returned by the method. This can be any validtype, including class types that you create.
- If the method does not return a value, its return type must be **void**.
- The name of the method is specified by *name*. This can be any legal identifier other than those already used by other items within the current scope.
- The *parameter-list* is a sequence of type and identifier pairs separated by commas. Parameters are essentially variables that receive the value of the *arguments* passed to the method when it is called.
- If the method has no parameters, then the parameter list will be empty.
- Methods that have a return type other than **void** return a value to the calling routine using the following form of the **return** statement:

```
return value;
```

- Here, *value* is the value returned.

Adding a Method to the Box Class

```
class Box {
        double width;    double height; double depth;
        // display volume of a box void
        volume() {
                System.out.print("Volume:");
```

```
                 System.out.println(width * height * depth);
                    }
             }
 class BoxDemo3 {
 public static void main(String args[]) {
       Box mybox1 = new Box(); //  assign  values  to mybox1's instance variables
mybox1.width = 10;
       mybox1.height = 20;
       mybox1.depth = 15; // display volume of first box
mybox1.volume();
   }
 }
```

This program generates the following output, which is the same as the previous version.

```
Volume is 3000.0
Volume is 162.0
```

Look closely at the following two lines of code:

```
mybox1.volume();
```

- The first line here invokes the **volume( )** method on **mybox1**.
- That is, it calls **volume( )** relative to the **mybox1** object, using the object’s name followed by the dot operator.
- Thus, the call to **mybox1.volume( )** displays the volume of the box defined by ***mybox1***,
- There is something very important to notice inside the **volume( )** method: the instance variables **width**, **height**, and **depth** are referred to directly, without preceding them with an object name or the dot operator.
- When a method uses an instance variable that is defined by its class, it does so directly, without explicit reference to an object and without use of the dot operator.
- This is easy to understand if you think about it. Amethod is always invoked relative to some object of its class. Once this invocation has occurred, the object is known.

Returning a Value

- While the implementation of **volume( )** does move the computation of a box’s volumeinside the **Box** class where it belongs, it is not the best way to do it.

```
class Box {
            double width;  double height;double depth;
           // compute and return volumedouble
        volume() {
                return width * height * depth;
 }
}
class BoxDemo4 {
        public static void main(String args[]) { Box
                mybox1 = new Box();double vol;
                // assign values to mybox1's instance variables
                mybox1.width = 10;
                mybox1.height = 20;
                mybox1.depth = 15;
                // get volume of first boxvol =
                mybox1.volume();
                System.out.println("Volume is " + vol);
}
}
```

- As you can see, when **volume( )** is called, it is put on the right side of an assignment statement.
- On the left is a variable, in this case **vol**, that will receive the value returned by ***volume( )***.
- Thus, after vol = mybox1.volume(); executes, the value of **mybox1.volume( )** is3,000 and this value then is stored in **vol**.

There are two important things to understand about returning values:

- The type of data returned by a method must be compatible with the return type specified by the method. For example, if the return type of some method is **boolean**, you could not return an integer.
- The variable receiving the value returned by a method (such as **vol**, in this case) must also be compatible with the return type specified for the method.

Adding a Method That Takes Parameters

- While some methods don’t need parameters, most do. Parameters allow a method tobe generalized.
- That is, a parameterized method can operate on a variety of data and/or be used in a number of slightly different situations

```
int square()
{
```

```
return 10 * 10;
}
```

- While this method does, indeed, return the value of 10 squared, its use is very limited.
- However, if you modify the method so that it takes a parameter, as shown next, thenyou can make **square( )** much more useful.

```
int square(int i)
{
      return i * i;
}
```

Now, **square( )** will return the square of whatever value it is called with. That is, **square( )** is now a general-purpose method that can compute the square of any integervalue, rather than just 10. Here is an example:

```
int x, y;
x = square(5); // x equals   25
x = square(9); // x equals 81
y = 2;
x = square(y); // x equals 4
```

- In the first call to **square( )**, the value 5 will be passed into parameter **i**.
- In the second call, **I** will receive the value 9.
- The third invocation passes the value of **y**, which is 2 in this example.
- As these examples show, **square( )** is able to return the square of whatever data it is passed
- A *parameter* is a variable defined by a method that receives a value when the methodis called. For example, in **square( )**, **i** is a parameter.
- An *argument* is a value that is passed to a method when it is Invoked.
- For example, **square(100)** passes 100 as an argument. Inside **square( )**, the parameter **i** receives that value.
- Thus, a better approach to setting the dimensions of a box is to create a method that takes the dimensions of a box in its parameters and sets each instance variableappropriately.
- This concept is implemented by the following program:

```
  // This program uses a parameterized method. class
  Box {
          double width; double height; double depth; // compute and return volume
          double volume() {
                  return width * height * depth;
}
// sets dimensions of box
```

```
            void setDim(double w, double h, double d)
            {width = w; height = h;depth = d;  }
}
class BoxDemo5 {
            public static void main(String args[]) {
             Box mybox1 = new  Box();
                  Box mybox2 = new Box();
                  double vol;
             mybox1.setDim(10, 20, 15);
                   mybox2.setDim(3, 6, 9);
                      vol = mybox1.volume();
                      System.out.println("Volume is " + vol);
                       mybox2.volume();
                      System.out.println("Volume is " + vol);
  }
}
```

- As you can see, the **setDim( )** method is used to set the dimensions of each box. For example, when mybox1.setDim(10, 20, 15); is executed, 10 is copied into parameter **w**, 20 is copied into **h**, and 15 is copied into **d**.
- Inside **setDim( )** the values of **w**, **h**, and **d** are then assigned to **width**, **height**, and **depth**, respectively.

Constructors

- It can be tedious to initialize all of the variables in a class each time an instance is created.
- Even when you add convenience functions like **setDim( )**, it would be simpler andmore concise to have all of the setup done at the time the object is first created.
- Because the requirement for initialization is so common, Java allows objects toinitialize themselves when they are created.
- This automatic initialization is performed through the use of a constructor.
- A *constructor* initializes an object immediately upon creation.
- It has the same name as the class in which it resides and is syntactically similar to a method.
- Once defined, the constructor is automatically called immediately after the object is created, before the **new** operator completes.
- Constructors look a little strange because they have no return type, not even **void**.This

is because the implicit return type of a class’ constructor is the class type itself.

- It is the constructor’s job to initialize the internal state of an object so that the codecreating an instance will have a fully initialized, usable object immediately.

```
  class Box {
      double width;double
      height;
      double depth;
// This is the constructor for Box.
      Box() {
            System.out.println("Constructing Box");width = 10;
            height = 10;
            depth = 10;
      }
      // compute and return volumedouble
      volume() {
            return width * height * depth;
    }
  }
  class BoxDemo6 {
public static void main(String args[]) {
            // declare, allocate, and initialize Box objectsBox
            mybox1 = new Box();
            Box mybox2 = new Box();
            double vol;
            // get volume of first box
            vol = mybox1.volume();
            System.out.println("Volume is " + vol);
            // get volume of second box
            vol = mybox2.volume();
            System.out.println("Volume is " + vol);
   }
  }
```

When this program is run, it generates the following results:

```
Constructing Box
Constructing BoxVolume is
1000.0
Volume is 1000.0
```

***class-var* = new *classname*( );**

- Now you can understand why the parentheses are needed after the class name. What is actually happening is that the constructor for the class is being called. Thus, in the line

  Box mybox1 = new Box();

- **new Box( )** is calling the **Box( )** constructor **new .**
- When you do not explicitly define a constructor for a class, then Java creates a default constructor for the class

Parameterized Constructors

- While the **Box( )** constructor in the preceding example does initialize a **Box** object, it is not very useful—all boxes have the same dimensions.
- What is needed is a way to construct **Box** objects of various dimensions.
- The easy solution is to add parameters to the constructor class

```
  Box {
          double width; double height;double depth;
          // This is the constructor for Box. Box(double w, double h, double d) {
                  width = w;height = h;depth = d;
}

          // compute and return volumedouble
          volume() {
                  return width * height * depth;
}
  }
  class BoxDemo7 {
  public static void main(String args[]) {
                  // declare, allocate, and initialize Box objects
                  Box mybox1 = new Box(10, 20, 15);
                  Box mybox2 = new Box(3, 6, 9);
                  double vol;

                  vol = mybox1.volume();
                  System.out.println("Volume is " + vol);
                  boxvol = mybox2.volume();
                  System.out.println("Volume is " + vol);
    }
  }
```

The output from this program is shown here:

```
Volume is 3000.0
Volume is 162.0
```

The this keyword

- Sometimes a method will need to refer to the object that invoked it.
- To allow this, Java defines the **this** keyword. **this** can be used inside any method torefer to the *current* object

```
Box(double  w, double h, double  d)
{this.width = w;  this.height=h;this.depth = d; }
```

Uses of this:

- To overcome shadowing or instance variable hiding.To call an overload constructor
- Example program: refer class notes

Instance Variable Hiding

- It is illegal in Java to declare two local variables with the same name inside the sameor enclosing scopes.Interestingly, you can have local variables, including formal parameters to methods,which overlap with the names of the class' instance variables.
- However, when a local variable has the same name as an instance variable, the local variable *hides* the instance variable.

```
// Use this to resolve name-space collisions.
Box(double width, double height, double depth) {
        this.width    =    width;
        this.height   =   height;
       this.depth = depth;
}
```

NOTE: The use of **this** in such a context can sometimes be confusing, and some programmers are careful not to use local variables and formal parameter names that hide instance variables.

Garbage Collection

- Since objects are dynamically allocated by using the **new** operator, you might be wondering how such objects are destroyed and their memory released for later reallocation.
- Java takes a different approach; it handles deallocation for you automatically.
- The technique that accomplishes this is called *garbage collection.*
- It works like this: when no references to an object exist, that object is assumed to beno longer needed, and the memory occupied by the object can be reclaimed.
- There is no explicit need to destroy objects as in C++.
- Garbage collection only occurs sporadically (if at all) during the execution of your program.

- It will not occur simply because one or more objects exist that are no longer used.

The finalize( ) Method

- Sometimes an object will need to perform some action when it is destroyed. For example, if an object is holding some non-Java resource such as a file handle or character font, then you might want to make sure these resources are freed before an object is destroyed.
- To handle such situations, Java provides a mechanism called *finalization.*
- By using finalization, you can define specific actions that will occur when an object isjust about to be reclaimed by the garbage collector.

The **finalize( )** method has this general form:

```
protected void finalize( )
{  // finalization code here
}
```

- Here, the keyword **protected** is a specifier that prevents access to **finalize( )** by code defined outside its class.
- It is important to understand that **finalize( )** is only called just prior to garbage collection.
- It is not called when an object goes out-of-scope, for example. This means that you cannot know when—or even if—**finalize( )** will be executed.
- Therefore, your program should provide other means of releasing system resources,etc., used by the object.
- It must not rely on **finalize( )** for normal program operation.

A Stack Class

```
  class Stack {
          int stck[] = new int[10];int tos;
          // Initialize top-of-stackStack() {
                  top = -1;
}
  // Push an item onto the stackvoid
  push(int item) {
if(top==9)
                  System.out.println("Stack is full.");
else
                   stck[++top] = item;
}
// Pop an item from the stack
int pop() {
```

```
                if(top < 0) {
                        System.out.println("Stack underflow.");return
                        0;
                }
                else
                         return stck[top--];
 }
  }
  class TestStack {
          public static void main(String args[]) {
                  Stack mystack1 = new Stack();
                  Stack mystack2 = new Stack();
                  // push some numbers onto the stack for(int i=0;
                  i<10; i++) mystack1.push(i);
                  for(int i=10; i<20; i++) mystack2.push(i);
                  //    pop    those    numbers    off    the    stack
                  System.out.println("Stack in mystack1:");
                  for(int i=0; i<10; i++)
                          System.out.println(mystack1.pop());
                          System.out.println("Stack  in        mystack2:");
                          for(int i=0; i<10; i++)
                          System.out.println(mystack2.pop());
}
  }
```

This program generates the following output:Stack in mystack1:

```
9
8
7
6
5
4
3
2
1
0
Stack in mystack2:
19
18
17
```

```
16
15
14
13
12
11
10
```

Overloading Methods

- In Java it is possible to define two or more methods within the same class that sharethe same name, as long as their parameter declarations are different.
- When this is the case, the methods are said to be *overloaded*, and the process is referred to as *method overloading.*
- Method overloading is one of the ways that Java supports polymorphism.
- When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method toactually call.
- Thus, overloaded methods must differ in the type and/or number of their parameters.
- While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method.

```
class OverloadDemo12 pt {
void test() {
            System.out.println("No parameters");
}
// Overload test for one integer parameter.
void test(int a) {
            System.out.println("a: " + a);
           }
      // Overload test for two integer parameters.void
      test(int a, int b) {
            System.out.println("a and b: " + a + " " + b);
      }
      // overload test for a double parameterdouble
      test(double a) {
            System.out.println("double a: " + a);return a*a;
    }
  }
  class Overload {
    public static void main(String args[]) {
            OverloadDemo ob = new OverloadDemo();
            double result;
            // call all versions of test()ob.test();
            ob.test(10);
            ob.test(10, 20);
            result = ob.test(123.25);
            System.out.println("Result of ob.test(123.25): " + result);
    }
  }
```

This program generates the following output:

```
No parametersa: 10
a and b: 10 20
double a: 123.25
Result of ob.test(123.25): 15190.5625
```

- When an overloaded method is called, Java looks for a match between the argumentsused to call the method and the method's parameters.
- However, this match need not always be exact. In some cases, Java's automatic type conversions can play a role in overload resolution.

For example, consider the following program:

// Automatic type conversions apply to overloading.

```
class OverloadDemo {
void test() {
                System.out.println("No parameters");
            }
        // Overload test for two integer parameters.void
        test(int a, int b) {
                System.out.println("a and b: " + a + " " + b);
       }
        // overload test for a double parametervoid
        test(double a) {
                System.out.println("Inside test(double) a: " + a);
   }
}
class Overload {
        public static void main(String args[]) {
              OverloadDemo ob = new OverloadDemo();
```

```
                    int i = 88;
                    ob.test();

                    ob.test(10, 20);

                    ob.test(i);    //    this    will    invoke    test(double)
                    ob.test(123.2); // this will invoke test(double)
        }

}
```

This program generates the following output:

```
No parametersa and
b: 10 20
Inside   test(double)   a:   88Inside
test(double) a: 123.2
```

- When **test( )** is called with an integer argument inside **Overload**, no matching methodis found.
- However, Java can automatically convert an integer into a **double**, and thisconversion can be used to resolve the call.
- Therefore, after **test(int)** is not found, Java elevates **i** to **double** and then calls ***test(double)***.
- Of course, if **test(int)** had been defined, it would have been called instead. Java will employ its automatic type conversions only if no exact match is found.
- Method overloading supports polymorphism because it is one way that Javaimplements the “one interface, multiple methods” paradigm.
- When you overload a method, each version of that method can perform any activityyou desire.
- There is no rule stating that overloaded methods must relate to one another.
- However, from a stylistic point of view, method overloading implies a relationship. Thus, while you can use the same name to overload unrelated methods, you should not.

Overloading Constructors

- In addition to overloading normal methods, you can also overload constructor methods. In fact, for most real-world classes that you create, overloaded constructors will be the norm, not the exception.
- To understand why, let’s return to the **Box** class developed in the preceding chapter. Following is the latest version of **Box**:

```
class Box {

        double width; double height;double depth;

         Box(double w, double h, double d) {
              width = w;height = h;depth = d;      }
        // constructor used when no dimensions specified Box()
        {
                width = -1; // use -1 to indicate an uninitialized box
                height = -1;
                depth = -1;
}

        // constructor used when cube is created
        Box(double len) {
                width = height = depth = len;

        // compute and return volume

        double volume() { return width * height * depth; }

        }



class OverloadCons {

 public static void main(String args[]) {
                // create boxes using the various constructors

                Box mybox1 = new Box(10, 20, 15);
                Box mybox2 = new Box();
                Box mycube = new Box(7);
                double vol;
                vol=mybox1.volume();
                System.out.println("Volume of mybox1 is " + vol);
                // get volume of second box
                vol = mybox2.volume();
                System.out.println("Volume of mybox2 is " + vol);

                // get volume of cube

                vol = mycube.volume();
                System.out.println("Volume of mycube is " + vol);

  }

  }
```

The output produced by this program is shown here:Volume of
mybox1 is 3000.0

Volume of mybox2 is -1.0Volume of
mycube is 343.0

Using Objects as Parameters

- So far, we have only been using simple types as parameters to methods. However, it isboth correct and common to pass objects to methods.
- For example, consider the following short program:

```
// Objects may be passed to methods.

class Test {
            int a, b;

     Test(int i, int j) {

              a = i;b = j;
                   }

       // return true if o is equal to the invoking objectboolean
       equals(Test o) {
              if(o.a == a && o.b == b)
              return true;
              else return false;
 }

}


class PassOb {

       public static void main(String args[]) {

       Test ob1 = new Test(100, 22);

       Test ob2 = new Test(100, 22);

       Test ob3 = new Test(-1, -1);
       System.out.println("ob1 == ob2: " + ob1.equals(ob2));
       System.out.println("ob1 == ob3: " + ob1.equals(ob3));
  }

}
```

This program generates the following output:ob1 == ob2: true ob1 == ob3: false

- As you can see, the **equals( )** method inside **Test** compares two objects for equalityand returns the result.
- That is, it compares the invoking object with the one that it is passed.
- If they contain the same values, then the method returns **true**. Otherwise, it returns **false**. Notice that the parameter **o** in **equals( )** specifies **Test** as its type.
- Although **Test** is a class type created by the program, it is used in just the same wayas Java's built-in types.
- One of the most common uses of object parameters involves constructors. Frequently, you will want to construct a new object so that it is initially the same as some existing object.
- To do this, you must define a constructor that takes an object of its class as aparameter

A Closer Look at Argument Passing

- In general, there are two ways that a computer language can pass an argument to a subroutine.
- The first way is *call-by-value*. This approach copies the *value* of an argument into the formal

parameter of the subroutine. Therefore, changes made to the parameter of the subroutine have no effect on the argument.

- The second way an argument can be passed is *call-by-reference.* In this approach, a reference to an argument (not the value of the argument) is passed to the parameter.
- Inside the subroutine, this reference is used to access the actual argument specified in the call. This means that changes made to the parameter will affect the argument used to call the subroutine.
- As you will see, Java uses both approaches, depending upon what is passed.
- In Java, when you pass a primitive type to a method, it is passed by value. Thus, what occurs to the parameter that receives the argument has no effect outside the method. For example, consider the following program:

```
// Primitive types are passed by value.
class Test {
        void meth(int i, int j) {
         i *= 2;
                j /= 2;
  }
 }
 class CallByValue {
        public static void main(String args[]) {Test ob =
                new Test();
                int a = 15, b = 20;
                System.out.println("a and b before call: " +a + " " + b);
                ob.meth(a, b);
                System.out.println("a and b after call: " +a + " " + b);
  }
 }
```

The output from this program is shown here:

```
a and b before call: 15 20a and
b after call: 15 20
```

- When you pass an object to a method, the situation changes dramatically, because objects are passed by what is effectively call-by-reference.
- Thus, when you pass this reference to a method, the parameter that receives it willrefer to the same object as that referred to by the argument.
- This effectively means that objects are passed to methods by use of call-by-reference. Changes to the object inside the method *do* affect the object used as an argument.
- For example, consider the following program:

```
// Objects are passed by reference.
```

```
class Test {
        int a, b;
         Test(int i, int j) {
               a = i;b = j;
         }
// pass an object
void meth(Test o) {
                        o*= 2;
                        o/= 2;
  }
}
class CallByRef {
        public static void main(String args[]) {
         Test ob = new Test(15, 20);
               System.out.println("ob.a and ob.b before call: " +ob.a + " " + ob.b);
               ob.meth(ob);
               System.out.println("ob.a  and  ob.b  after  call:  "  +ob.a + " " + ob.b);
  }
}
```

This program generates the following output:

```
ob.a and ob.b before call: 15 20ob.a
and ob.b after call: 30 10
```

Returning Objects

- A method can return any type of data, including class types that you create. For example, in the following program, the **incrByTen( )** method returns an object in which the value of **a** is ten greater than it is in the invoking object.

```
// Returning an object.
class Test {
             int  a;
       Test(int i) {
               a = i;
                }
Test incrByTen() {
               Test temp = new Test(a+10);return
               temp;
 }
}
class RetOb {
       public static void main(String args[]) {
               Test ob1 = new Test(2);
               Test ob2;

               ob2=ob1.incrByTen();
               System.out.println("ob1.a:"+ob1.a);
               System.out.println("ob2.a:" + ob2.a);

               ob2 = ob2.incrByTen();
               System.out.println("ob2.a after second increase: " + ob2.a);

 }
}
```

The output generated by this program is shown here:

```
ob1.a: 2

ob2.a: 12

ob2.a after second increase: 22
```

Recursion

- Java supports *recursion*. Recursion is the process of defining something in terms of itself. As it relates to Java programming, recursion is the attribute that allows a method to call itself.
- A method that calls itself is said to be *recursive*.
- The classic example of recursion is the computation of the factorial of a number. The factorial of a number *N* is the product of all the whole numbers between 1 and *N*. For example, 3 factorial is 1 × 2 × 3, or 6. Here is how a factorial can be computed by use of a recursive method:

```
// A simple example of recursion.
class Factorial {
         int fact(int n) {
                int result;
                if(n==1) return 1;
                result = fact(n-1) * n;
                return result;
                        }
          }
 class Recursion {

            public static void main(String args[]) {

                Factorial f = new Factorial();
                System.out.println("Factorial  of  3  is  "  +  f.fact(3));
                System.out.println("Factorial  of  4  is  "  +  f.fact(4));
                System.out.println("Factorial of 5 is " + f.fact(5));
  }
}
```

The output from this program is shown here:

```
Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120
```

- When a method calls itself, new local variables and parameters are allocated storageon the stack, and the method code is executed with these new variables from the start.
- As each recursive call returns, the old local variables and parameters are removedfrom the stack, and execution resumes at the point of the call inside the method. Recursive methods could be said to "telescope" out and back.
- Recursive versions of many routines may execute a bit more slowly than the iterative equivalent because of the added overhead of the additional function calls.
- Many recursive calls to a method could cause a stack overrun. Because storage for parameters and local variables is on the stack and each new call creates a new copy of these variables, it is possible that the stack could be exhausted.
- If this occurs, the Java run-time system will cause an exception.

The main advantage to recursive methods is that they can be used to create clearer and simpler versions of several algorithms than can their iterative relatives.

Here is one more example of recursion. The recursive method **printArray( )** prints the first **i** elements in the array **values**.

```
// Another example that uses recursion.

class RecTest {
int values[];
RecTest(int i) {
                values = new int[i];
   }
printArray(int i)  // display array -- recursivelyvoid
{
                if(i==0) return;
                else printArray(i-1);
                System.out.println("[" + (i-1) + "] " + values[i-1]);
   }
}
class Recursion2 {
        public  static  void  main(String  args[])  {
                RecTest ob = new RecTest(5);
                int i;
                        for(i=0; i<5; i++)
                        ob.values[i] = i;
                        ob.printArray(5);
}
 }
```

This program generates the following output:

```
[0] 0
[1] 1
[2] 2
[3] 3
[4] 4
```

Introducing Access Control

- Encapsulation provides another important attribute: *access control.*
- Through encapsulation, you can control what parts of a program can access themembers of a class.
- By controlling access, you can prevent misuse. For example, allowing access to dataonly through a welldefined set of methods, you can prevent the misuse of that data.
- Thus, when correctly implemented, a class creates a “black box” which may be used,but the inner workings of which are not open to tampering
- Java’s access specifiers are **public**, **private**, and **protected**.
- Java also defines a default access level. **protected** applies only when inheritance is involved When a member of a class is modified by the **public** specifier, then that member can be accessed by any other code.
- When a member of a class is specified as **private**, then that member can only be accessed by other members of its class.
- Now you can understand why **main( )** has always been preceded by the **public** specifier. It is called by code that is outside the program—that is, by the Java run-timesystem.
- When no access specifier is used, then by default the member of a class is public within its own package, but cannot be accessed outside of its package.
- An access specifier precedes the rest of a member’s type specification. That is, it mustbegin a member’s declaration statement.
- Here is an example:

```
                        class Test {
                          c  int i;private double j;
p                         private int myMethod(int a, char b) { // ...
u
b
l
i

          int   a;   //   default   access

          public int b; // public access

          private int c; // private access
         // methods to access c

         void setc(int i) { c = i; }

         int getc() { return c;   }

 }

 class AccessTest {

         public static void main(String args[]) {Test ob =
                 new Test();
                 // These are OK, a and b may be accessed directlyob.a =
                 10;
                 ob.b = 20;

                 // This is not OK and will cause an error

                 // ob.c = 100; // Error!

                 //   You   must   access   c   through   its   methods
                 ob.setc(100); // OK
                 System.out.println("a, b, and c: " + ob.a + " " + ob.b + " " + ob.getc());

   }

 }
```

- As you can see, inside the **Test** class, **a** uses default access, which for this example isthe same as specifying **public**. **b** is explicitly specified as **public**.
- Member **c** is given private access. This means that it cannot be accessed by code outside of its class.
- So, inside the **AccessTest** class, **c** cannot be used directly. It must be accessed throughits public methods: **setc( )** and **getc( )**.
- If you were to remove the comment symbol from the beginning of the following line,

  // ob.c = 100; // Error!

Understanding static

- There will be times when you will want to define a class member that will be used independently of any object of that class.
- Normally, a class member must be accessed only in conjunction with an object of itsclass.

- However, it is possible to create a member that can be used by itself, withoutreference to a specific instance.
- To create such a member, precede its declaration with the keyword **static**.
- ***When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object.***
- You can declare both methods and variables to be **static**.
- The most common example of a **static** member is **main( )**. **main( )** is declared as **static** because it must be called before any objects exist.
- ***Instance variables declared as static are, essentially, global variables***.
- **When objects of its class are declared, no copy of a static variable is made. Instead, all instances of the class share the same static variable.**
- Methods declared as **static** have several restrictions:
    - They can only call other **static** methods.
    - They must only access **static** data.
    - They cannot refer to **this** or **super** in any way

The following example shows a class that has a **static** method, some **static** variables, and a **static** initialization block:

```
// Demonstrate static variables, methods, and blocks.class
UseStatic {
        static int a = 3;
        static int b;
        static void meth(int x) {
                System.out.println("x = " + x);System.out.println("a = " + a);
                System.out.println("b = " + b);
           }
        static {
                 System.out.println("Static block initialized.");
                 b = a * 4;
               }
        public static void main(String args[]) {
        meth(42);
      }
}
```

- As soon as the **UseStatic** class is loaded, all of the **static** statements are run.
- First, **a** is set to **3**,

- then the **static** block executes, which prints a message and then initializes **b** to **a * 4** or **12**.
- Then **main( )** is called, which calls **meth( )**, passing **42** to **x**.
- The three **println( )** statements refer to the two **static** variables **a** and **b**, as well as tothe local variable **x**.

Here is the output of the program:

```
Static block initialized.
x = 42
a = 3
b = 12
```

- Outside of the class in which they are defined, **static** methods and variables can be used independently of any object.
- To do so, you need only specify the name of their class followed by the dot operator.
- For example, if you wish to call a **static** method from outside its class, you can do so using the following general form:

classname.method( )

- Here, *classname* is the name of the class in which the **static** method is declared. As you can see, this format is similar to that used to call non-**static** methods through object-reference variables.
- A **static** variable can be accessed in the same way—by use of the dot operator on the name of the class. This is how Java implements a controlled version of global methods and global variables.
- Here is an example. Inside **main( )**, the **static** method **callme( )** and the **static** variable **b** are accessed through their class name **StaticDemo**. class

```
StaticDemo {
        static int a = 42;
        static int b = 99;
        static void callme() {
         System.out.println("a = " + a);
}
 }
class StaticByName {
        public   static   void   main(String   args[])   {
        StaticDemo.callme();
        System.out.println("b = " + StaticDemo.b);
}
 }
```

Here is the output of this program:

```
a = 42
b = 99
```

Introducing final

- A variable can be declared as **final**. Doing so prevents its contents from beingmodified. This means that you must initialize a **final** variable when it is declared.
- For example:

```
final int FILE_NEW = 1;

final int FILE_OPEN = 2;

final int FILE_SAVE = 3;

final int FILE_SAVEAS = 4;

final int FILE_QUIT = 5;
```

- Subsequent parts of your program can now use **FILE_OPEN**, etc., as if they were constants, without fear that a value has been changed.
- It is a common coding convention to choose all uppercase identifiers for **final** variables.
- Variables declared as **final** do not occupy memory on a per-instance basis. Thus, a **final** variable is essentially a constant.
- The keyword **final** can also be applied to methods, but its meaning is substantiallydifferent than when it is applied to variables.

Nested and Inner classes

In Java, we can define a class within another class. Such class is known as nested classes.

For ex:

```
Class outer_class
{
  //…
        Class nested_class
        {
           //…
        }
}
```

If class B is defined within class A, then B does not exist independently of A. A nested class has access to the members, including private members, of the class in which it is nested. The enclosing class does not have access to the members of the nested class.

There are two types of nested classes: static and non-static. A static nested class is one that has the static modifier applied. Because it is static, it must access the non-static members of its enclosing class through an object. That is, it cannot refer to non-static members of its enclosing class directly.

The second type of nested class is the inner class. An inner class is a non-static nested class. It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class do.

The following program illustrates how to define and use an inner class. The class named Outer has one instance variable named outer_x, one instance method named test( ), and defines one inner class called Inner.

```
    // Demonstrate an inner class.class
    Outer
     {
     int outer_x = 100;
     void test()
     {
    Inner inner = new Inner();
    inner.display();
     }

     // this is an inner classclass
    Inner {
    void display()
    {
    System.out.println("display: outer_x = " + outer_x);
    }
    }
     }
    class InnerClassDemo
     {
    public static void main(String[] args)
     {
    Outer outer = new Outer(); outer.test();
    }
}
```

Output from this application is shown here: display: outer_x = 100

In the program, an inner class named Inner is defined within the scope of class Outer. Therefore, any code in class Inner can directly access the variable outer_x. An instance method named display( ) is defined inside Inner. This method displays outer_x on the standard output stream. The main( ) method of InnerClassDemo creates an instance of class Outer and invokes its test( )

method. That method creates an instance of class Inner and the display( ) method is called.

MODULE 3

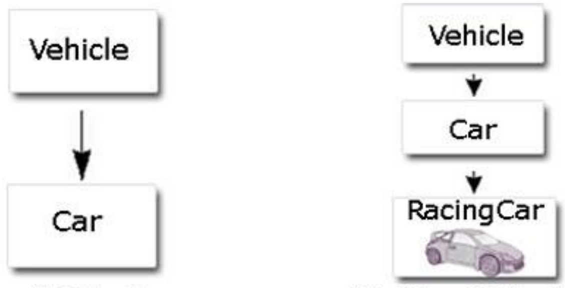
Chapter :8 Inheritance

- Inheritance is the mechanism through which we can derive classes from other classes.
- It is process of deriving features from parent class
- The derived class is called as child class or the subclass or we can say the extended class and theclass from which we are deriving the subclass is called the base class or the parent class.
- Inheritance supports code reusability: Child class can reuse the methods and fields of parent class.
- To derive a class in java the keyword **extends** is used.

Types of Inheritance

- Single level/Simple Inheritance
- Multi level Inheritance
- Multiple Inheritance (Java doesn't support Multiple inheritance but we can achieve this throughInterface)

Pictorial Representation of Simple and Multi level Inheritance



SimpleInheritance **MultilevelInheritance**

Single level/Simple Inheritance

- When a subclass is derived from a parent class then this mechanism is known as simple inheritance.
- In case of simple inheritance there is only a sub class and its parent class. It is also called singleinheritance or one level inheritance.
- The syntax for creating a subclass is simple. At the beginning of your class declaration, use theextends keyword, followed by the name of the class to inherit from:

```
class A
 {
 }
 class B extends A
 {
 }
```

Program to demonstrate single inheritance

```
class A
{
 int x=10;
  void get1()
{    System.out.println("value of x is"+x);   }
}
```

```
class B extends A
{
 int y=20;
 void get2()
              {  System.out.println(x+y);  }
}
class Main
{
     public static void main(String[] args)
 {
       B obj=new B();
       obj.get1();
       obj.get2();
 }
}
Output:
Value of x is 10
30
```

Multilevel Inheritance

- When a sub class is derived from a derived class then this mechanism is known as the multilevel inheritance.
- The derived class is called the subclass or child class for it's parent class and this parent classworks as the child class for it's just above(parent) class.
- Multi level inheritance can go up to any number of level.

```
class A
{
 int x=10;
 void get1()
              {   System.out.println("value of x is"+x);  }
}
class B extends A
{
     int y=20;
     void get2()
              {  System.out.println(x+y);   }
}
class C extends B
{    int z=30;
     void get3()
              {  System.out.println(z+y);   }
}
class Main
{
      public static void main(String[] args)
 {
        C obj=new C();
   obj.get1();
        obj.get2();
        obj.get3();
 }
}
```

Output

```
        Value of x is 10
        30
        50
```

super keyword

The super is java keyword. As the name suggest super is used to access the members of thesuper class. Uses of super keyword are:

- **Use super with datamembers/variables**

  We can use super keyword to access the data member or field of immediate parent class. Itis used if parent class and child class have same fields.

```
class vehicle
{
  int maxspeed=200;
}
class car extends vehicle
{
   int maxspeed =150;
   void display()
   {
     System.out.println(maxspeed);     // 150
     System.out.println(super.maxspeed);  //200
   }
}
class Main
{
   public static void main(String[] args)
  {
      car obj=new car();
      obj.display();
      }
}
```

- **Use super with methods**

  The super keyword can also be used to invoke parent class method. It should be used ifsubclass contains the same method as parent class.

```
class A
{
              void display()
               {  System.out.println("hello");  }
}
class B extends A
{
             void display()
              {
                  super.display(); //call parent class display() hello
                  System.out.println("welcome");
              }
}
```

```
class Main
{
                 public static void main(String[] args)
{                          B obj=new B();
                          obj.display();
}
}
```

- **Use super with constructors**

The super keyword can also be used to invoke the parent class constructor.

```
                class A
              {
                 A( )
                 {         System.out.println("parent class constructor");

                  }
           }


                          class B extends A
               {
                 B( )
                  {
                             super();     // call parent class constructor
                              System.out,println("child class constructor");
                  }
}
class Main
{
                 public static void main(String[] args) {
                          B obj=new B();
 }
}
```

Output:

```
                Parent class constructor
                Child class constructor
```

Method Overriding

- Method overriding in java means a subclass method overriding a super class method.
- Super class method should be non-static. Subclass uses extends keyword to extend the super class.
- If subclass (child class) has the same method as declared in the parent class, it is known as **methodoverriding in Java**.

Rules for Java Method Overriding

- The method must have the same name as in the parent class
- The method must have the same parameter as in the parent class.
- There must be an IS-A relationship (inheritance).

```
//Program to demonstrate method overriding
        class A
                {
                   void display()
                  {   System.out.println("hello");  }
        }
```

```
class B extends A
{
                void display()      //method overriding
                {  System.out.println("welcome");   }
}


class Main
{
             public static void main(String[] args)
{
 B obj=new B();
 obj.display();
}
}
Output: Welcome
```

- When we call display() using the obj object (object of the subclass), the method inside the subclass B is called. This will display welcome as output.
- If we want to access parent class display function, then use super keyword.

When constructors are executed?

When a derived class is extended from the base class, the constructor of the base class isexecuted first followed by the constructor of the derived class.

```
class A
{
             A( )
                {                System.out.println("parent A class constructor");   }
class B extends A
{
             B( )
                {               System.out.println("parent B class constructor");   }

}
class C extends B
{
             C( )
               {
}                              System.out,println("child class constructor");
class Main
{
             public static void main(String[] args)
{
                     C obj=new C();
}
}
Output:
            Parent A class constructor
            Parent B class constructor
            child class constructor
```

In the above program, derived class C is extended from the class B, class B is extended from class A. When object of child class C is created the constructor of the base classes i.e A() and B() is executed first followed by the constructor of the derived class C().

Dynamic method dispatch

- Dynamic method dispatch is the mechanism by which a calling to an overridden method will be done at run time, rather than compile time.
- When an overridden method is called through a superclass reference, Java determines which version(superclass/subclasses) of that method is to be executed based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time.

```
   class mobile
   {
     void display()
      {
          System.out.println("9 inch display");
      }
  }
   class Samsung extends mobile
   {
     void display()
      {
          System.out.println("7 inch display");
      }
  }
   class Apple extends mobile
   {
     void display()
      {
          System.out.println("7.5 inch display");
      }
  }
 class Main
  {
   public static void main(String args[])
    {
         mobile m=new mobile(); S
         amsung s =new Samsung();
         Apple a =new Apple();
         mobile ref;
         ref=m;
          ref.dis play();ref=s;
          ref.display();ref=a;
          ref.display();
    }
 }
```

Output:

```
   9 inch display
   7 inch display
```

7.5 inch display

The above program creates one superclass mobile and it's two subclasses samsung and apple. These subclasses overrides display( ) method.

- Inside the main() method, initially objects of classes are declared.
- Now a reference of type mobile, called ref, is also declared.
- Now we are assigning a reference to each **type of object** to *ref*, one-by-one, and uses that reference to invoke display( ). As the output shows, the version of display( ) executed is determined **by the type of object being referred to at the time of the call.**

Abstraction in Java:

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

There are two ways to achieve abstraction in java

- Abstract class
- Interface

Abstract classes

A class which is declared with the abstract keyword is known as an abstract class. It can have abstract and non-abstract methods (method with the body). It is not possible to create objects of abstract class, its properties needs to be extended to derived class.

syntax:

abstract class A{}

- A method which is declared as abstract and does not have implementation is known as an abstractmethod.

Syntax:

abstract void display();//no method body and abstract

Example of Abstract class that has an abstract method

```
abstract class A
      {
        abstract void display();    // abstract
          method void show()        //non-abstract
          method
          {
              System.out.println("hello");
          }
     }
      class B extends A
                                                                    {
         void display()
          {
              System.out.println("Welcome");
          }
     }
    class main
    {
      public static void main(String args[])
      {
            B obj=new B();
obj.display();
            obj.show();
          }
        }
```

In this example, A is an abstract class that contains one abstract method display() and non-abstract method show(). As display() is an abstract method it does not have implementation part in class A. Its implementation is provided by the class B.

Using final with inheritance

- **Using final to prevent overriding**

  Methods declared as final cannot be overridden.Example:

  ```
  class A
  {
    final void display()
     {   System.out.println(" class A");   }
  }
  class B extends A
  {
   void display()      //error! cant override display() because it is declared as final in
                         parent class
     {   System.out.println(" class A");    }
  }
  ```

- **Using final to prevent inheritance**

  final with class prevents a class from being inheritedExample

  ```
  final class A
  {
   // .......

  }
  class B extends A      // error! cant inherit class A
  {
  }
  ```

Local variable type inference and inheritance

- Type inference refers to the automatic detection of the datatype of a variable, done generally at the compiler time.
- Local variable type inference is a feature in Java 10 that allows the developer to skip the type declaration associated with local variables (those defined inside method definitions, initialization blocks, for-loops, and other blocks like if-else), which is supported by the keyword 'var'.

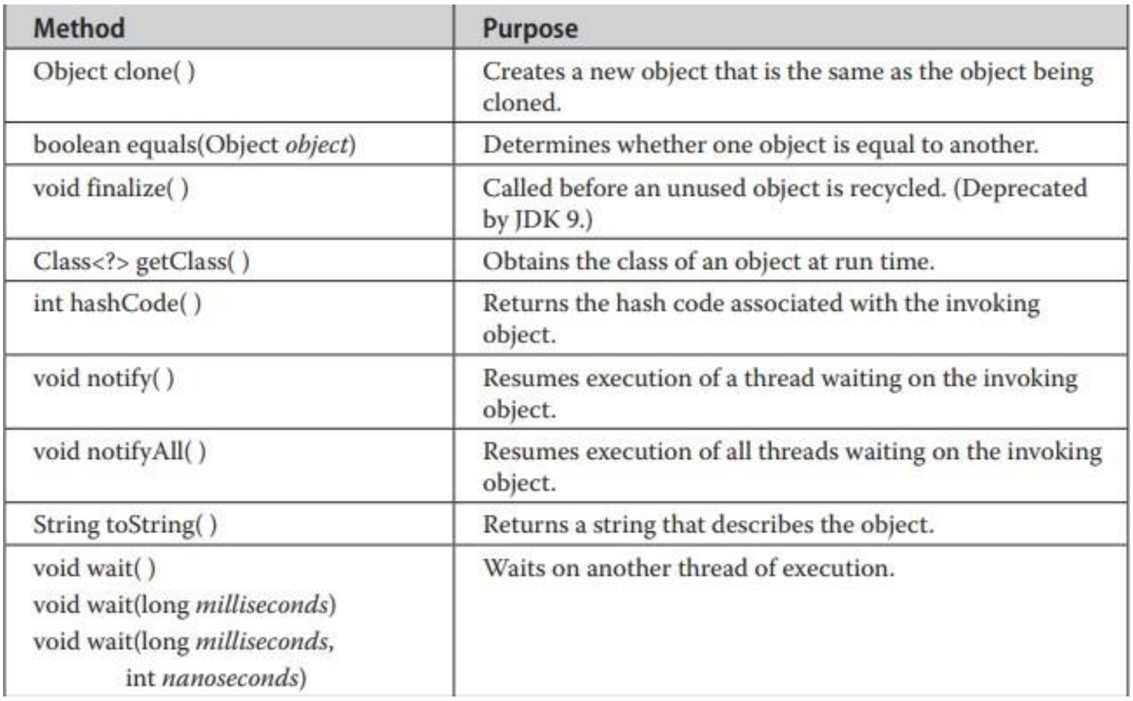
  ```
  // Declaration of a local variable in java 10
  using LVTIclass A {
   public static void main(String a[])
   {
    var x = "Hi there"; System.out.println(x)
  }
  ```

```
}     // Declaring iteration variables in enhanced for loops using LVTI in Javaclass A {
              public static void main(String a[])
              {
               int[] arr = new int[3];
               arr = { 1, 2, 3 };
              for (var x : arr)
                System.out.println(x + "\n");
              }
}
```

The object class

There is one special class, Object, defined by Java. All other classes are subclasses of Object. That is, Object is a super class of all other classes. This means that a reference variable of type Object can refer to an object of any other class. Also, since arrays are implemented as classes, a variable of type Object can also refer to any array.

Object defines the following methods, which means that they are available in every object.



Method	Purpose
Object clone( )	Creates a new object that is the same as the object being cloned.
boolean equals(Object *object*)	Determines whether one object is equal to another.
void finalize( )	Called before an unused object is recycled. (Deprecated by JDK 9.)
Class<?> getClass( )	Obtains the class of an object at run time.
int hashCode( )	Returns the hash code associated with the invoking object.
void notify( )	Resumes execution of a thread waiting on the invoking object.
void notifyAll( )	Resumes execution of all threads waiting on the invoking object.
String toString( )	Returns a string that describes the object.
void wait( ) void wait(long *milliseconds*) void wait(long *milliseconds*, int *nanoseconds*)	Waits on another thread of execution.

Chapter 9: Interfaces

- The interface in Java is *a* mechanism to achieve abstraction. There can be only variables andabstract methods in the Java interface, not the method body.
- It is used to achieve abstraction and multiple inheritances in Java using Interface.

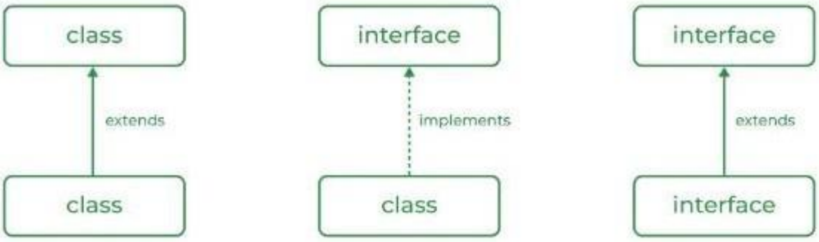
Syntax for Java Interfaces

```
interface {
            // declare variables
           // declare abstract methods
         }
```

- To declare an interface, use the interface keyword.
- It is used to provide total abstraction. That means all the methods in an interface are declaredwith an empty body and are public and all fields are public, static, and final by default.
- A class that implements an interface must implement all the methods declared in the the interface.
- To implement the interface, use the implements keyword.

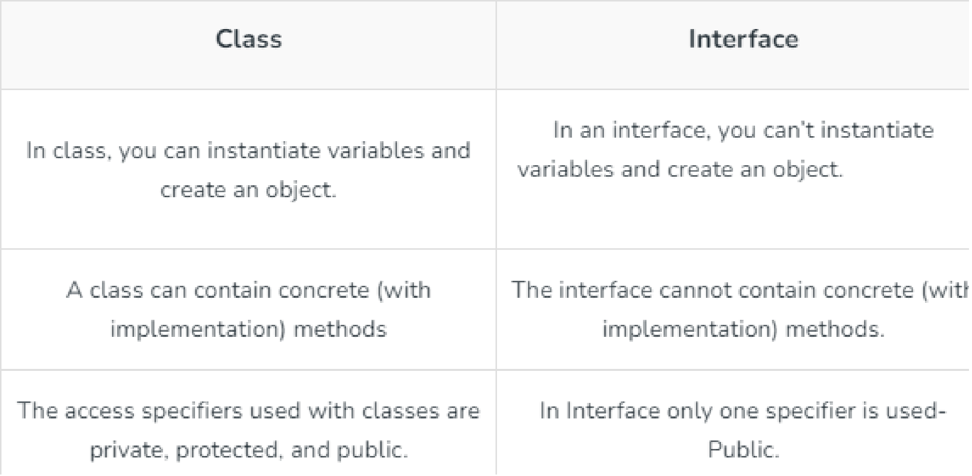
```
  // Java program to demonstrate working of interfaceinterface In {
              int a = 10;
             void display();   //public and abstract
            }
            // A class that implements the interface.class A
            implements In{
            // Implementing the capabilities of interface.public
              void display()
                 {
                     System.out.println("Geek");
                 }
             public static void main(String[] args)
             {
             A obj = new A();
             obj.display();
             System.out.println(a);
}
}
```

In the above program, after creating objects of class A and B, a reference variable of interface In is created.Then Assign objects to reference variable, based on reference variable assigned methods can be accessed.



Relationship between class and interface

Difference between class and interface:



Class	Interface
In class, you can instantiate variables and create an object.	In an interface, you can't instantiate variables and create an object.
A class can contain concrete (with implementation) methods	The interface cannot contain concrete (wit implementation) methods.
The access specifiers used with classes are private, protected, and public.	In Interface only one specifier is used- Public.

Accessing Implementations Through Interface References

```
  interface In {
  void display(int a);
 }
 class A implements In {
    public void display(int p)
    {System.out.println(p);
   }
 }
 class B implements In {
  public void display(int p)
  {
    System.out.println("p squared is " + (p*p));
  }
 }
 public class Main {
  public static void main(String args[])
                {
       A o=new A();
                B ob = new
                B();In i;
                i=o;
                i.display(4);
                i = ob;
                i.display(4);
  }
 }
Output:4
 p squared is 16
```

I

Nested Interfaces

- An interface can be declared as member of a class or another interface. Such an interface is called amember interface or a nested interface.
- While implementing the interface, we mention the interface as **c_name.i_name** where **c_name** isthe name of the class in which it is nested and **i_name** is the name of the interface itself.

Syntax

```
                class abc
               {
                 interface _interface_name {
       ...       }
             }
```

```
// Java program to demonstrate working of interface
inside a class.class Test {
     interface Yes {
      void show();
   }
}
class Testing
   implements
   Test.Yes {
   public void
   show()
   {
      System.out.println("show method of interface");
   }
}
class Main {
   public static void main(String[] args)
   {
      Test.Yes obj;
      Testing t = new Testing();
      obj = t;
      obj.show();
   }
 }
```

Output

```
show method of interface
```

Variables in Interfaces

An interface is a container of abstract methods and static final variables. The interface contains the static final variables. The variables defined in an interface can not be modified by the class that implements the interface, but it may use as it defined in the interface.

- The variable in an interface is public, static, and final by default.
- If any variable in an interface is defined without public, static, and final keywords then, thecompiler automatically adds the same.

- No access modifier is allowed except the public for interface variables.
- Every variable of an interface must be initialized in the interface itself.
- The class that implements an interface cannot modify the interface variable, but it may use as itdefined in the interface.

Example code to illustrate variables in an interface

```
interface In{
int a = 100;
//int b;                       // Error - must be
public class A implements In{
public static void main(String[] args)
{
System.out.println(a);
// a = 150: //Error! Can not be modified
```

Interfaces Can Be Extended

- One interface can inherit another by use of the keyword extends. The syntax is the same as forinheriting classes.
- When a class implements an interface that inherits another interface, it must provideimplementations for all methods required by the interface inheritance chain.

Following is an example:

```
interface A
                {
                        void meth1();
               }
interface B extends A
               {
                        void meth2();
                }
class MyClass implements B
                {
                     public void meth1()
                         {
                               System.out.println("Implement meth1().");
                          }
                    public void meth2()
                         {
                          System.out.println("Implement meth2().");
                          }
}
class Main
{
                public static void main(String[] args)
                {
                          MyClass ob = new MyClass();
                          ob.meth1();
                          ob.meth2();
}
                }
```

Default Interface Methods

- Java provides a facility to create default methods inside the interface. Methods which are defined inside the interface and tagged with default are known as default methods. These methods are non-abstract methods.
- In the following example, In is an interface that contains a default and an abstract method. The concept of default method is used to define a method with default implementation. We can override default method also to provide more specific implementation for the method.

```
              interface In
             {
               void display(String msg);
               //Abstract method default
               void show ()                    // Default method
               {
                System.out.println("Hello, this is default method");
                }
}
class A implements in{
                 void display(String msg){      //
                 implementing abstract method
                 System.out.println(msg);
                }
             public static void
                 main(String[]
                 args) {A obj =
                 new A();
                obj.show();  // calling default method
                obj.display("Work is worship");  // calling abstract method
}
}
```

Output:

```
                 Hello, this is default method
                 Work is worship
```

Use static Methods in an interface

- A static method defined by an interface can be called independently of any object. Thus, no implementation of the interface is necessary, and no instance of the interface is required, in order to call a static method.
- A static method is called by specifying the interface name, followed by a period, followed by the method name.

Here is the general form:

```
                          InterfaceName.staticMethodName
interface In{
            void display(String msg);   // Abstract method static
            void show(String msg)       // static method
              {
               System.out.println(msg);
              }
```

```
        }
        class A implements In{
          public void display(String msg){   // implementing abstract method
              System.out.println(msg);
           }
           public static void
              main(String[]
              args) {A obj =
              new A();
              obj.display("Work is worship");     // calling abstract method
              In.show("Helloooo...");  // calling static method
           }
        }
```

Private Interface methods

In Java 9, we can create private methods inside an interface. Interface allows us to declare privatemethods that help to share common code between non-abstract methods.

```
    interface In{
        default void say()
        {
           saySomething();
        }
        private void saySomething()
         {
          System.out.println("Hello... I'm private method");
         }
    }
    class A implements In {
public static void main(String[] args) {
 A obj=new A();
     In ref;
     ref=obj;
     ref.say();
        }
      }
    Output: Hello... I'm private method
```

MODULE 4

Chapter 9: Packages

Packages:

- A Package can be defined as a grouping of related types(classes, interfaces)
- A package represents a directory that contains related group of classes and interfaces.
- Packages are used in Java in order to prevent naming conflicts.
- Java package provides access protection.

There are two types of packages in Java.

  - Pre-defined Packages(built-in)
  - User defined packages

There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Defining a Package(User defined):

- To create a package : include a **package** command as the first statement in a Java source file.
- Any classes declared within that file will belong to the specified package.
- The **package** statement defines a name space in which classes are stored.
- If you omit the **package** statement, the class names are put into the default package, which has no name.

This is the general form of the **package** statement:

package *pkg* ;

pkg is the name of the package.

For example, the following statement creates a package called **MyPackage**:

package MyPackage;

- Java uses file system directories to store packages.

  For example, the **.class** files for any classes you declare to be part of **MyPackage** must be stored in a directory called **MyPackage**.
- More than one file can include the same **package** statement.
- The **package** statement simply specifies to which package the classes defined in a file belong.

To create a hierarchy of packages.

- separate each package name from the one above it by use of a period.

The general form of a multileveled package statement

package *pkg1*[.*pkg2*[.*pkg3*]];

A package hierarchy must be reflected in the file system of Java development system.

For example, a package declared as

package java.awt.image;

Finding Packages and CLASSPATH:

- Packages are mirrored by directories.
- The Java run-time system know where to look for packages that are created.
  - First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in a subdirectory of the current directory, it will be found.
- Second, you can specify a directory path or paths by setting the **CLASSPATH** environmental variable.
  - Third, you can use the **-classpath** option with **java** and **javac** to specify the path to your classes.

For example, consider the following package specification:

- package MyPack can be executed from a directory immediately above **MyPack** ,
- or the **CLASSPATH** must be set to include the path to **MyPack**,
- or the **-classpath** option must specify the path to **MyPack** when the program is run via **java**.
- When the second two options are used, the class path *must not* include **MyPack**, itself.
- It must simply specify the *path to* **MyPack**.
- For example, in a Windows environment, if the path to **MyPack** is

C:\MyPrograms\Java\MyPack

- Then the class path to **MyPack** is

C:\MyPrograms\Java

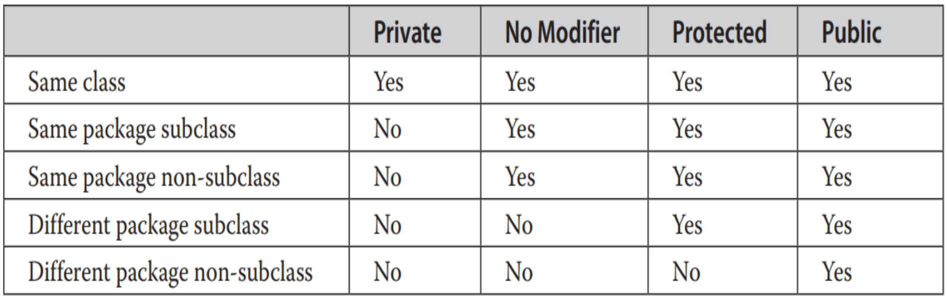
Example: Package demonstration

```
package pack;

public class Addition
{

        int x,y;
        public Addition(int a, int b)
        {

                x=a; y=b;
        }
        public void sum()
        {

                System.out.println("Sum :"+(x+y));
        }

}
```

Save the above file with Addition.java

Access Protection

- Access protection defines actually how much an element (class, method, variable) is exposed to other classes and packages.
- There are four types of access specifiers available in java:
    - Visible to the class only (private).
    - Visible to the package (default). No modifiers are needed.
    - Visible to the package and all subclasses (protected)
    - Visible to the world (publ



	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Example:

The following example shows all combinations of the access control modifiers. This example has two packages and five classes. The source for the first package defines three classes: **Protection**, **Derived**, and **SamePackage**.

Name of the package: pkg1

This file is Protection.java

```
package pkg1;

public class Protection
    {       int n = 1;

            private int n_priv = 2;
            protected int n_prot = 3;
            public int n_publ = 4;

            public Protection()
            {

                System.out.println("base constructor");

                System.out.println("n = " + n);

                System.out.println("n_priv = " + n_priv);
                System.out.println("n_prot = " + n_prot);
                System.out.println("n_publ = " + n_publ);
            }
    }
```

This is file Derived.java:

```
package pkg1;
class Derived extends Protection
{   Derived()

  {

   System.out.println("Same package - derived (from base) constructor");
  System.out.println("n = " + n);

   /* class only
   *  System.out.println("n_priv = "4 + n_priv); */
   System.out.println("n_prot = " + n_prot);
   System.out.println("n_publ = " +n_publ);
  }
}12 pt
```

This is file SamePackage.java package

```
pkg1;
class SamePackage
{     SamePackage()

  {   Protection pro = new Protection();

   System.out.println("same package - other constructor");
   System.out.println("n = " + pro.n);
   /* class only
   *  System.out.println("n_priv = " + pro.n_priv); */
   System.out.println("n_prot = " + pro.n_prot);
   System.out.println("n_publ = " + pro.n_publ);
  }

}
```

Name of the package: pkg2

This is file Protection2.java:

```
package pkg2;
class Protection2 extends pkg1.Protection
{

       Protection2()
       {

         System.out.println("Other    package-Derived  (from  Package  1-Base) Constructor"

         /* class or package only
         • System.out.println("n = " + n); */
         /* class only
         • System.out.println("n_priv = " + n_priv); */
         System.out.println("n_prot = " + n_prot);
         System.out.println("n_publ = " + n_publ);
       }
}
```

This is file **OtherPackage.java**

```
package pkg2;

     class OtherPackage
     {

       OtherPackage()
       {

         pkg1.Protection pro = new pkg1.Protection();
         System.out.println("other package - Non sub class constructor");
         /* class or package only
         • System.out.println("n = " + pro.n); */

         /* class only
         • System.out.println("n_priv = " + pro.n_priv); */
         /* class, subclass or package only
         System.out.println("n_prot = " + pro.n_prot); */
         System.out.println("n_publ = " + pro.n_publ);
       }

     }
```

If you want to try these t two packages, here are two test files you can use. The one for package

pkg1 is shown here:

```
     /* demo package pkg1 */
     package pkg1;
     /* instantiate the various classes in pkg1 */
     public class Demo
     {
       public static void main(String args[])
       {

         Derived obj2 = new Derived();
       SamePackage obj3 = new
       SamePackage();
    }
}
```

The test file for package pkg2 is

```
package pkg2;

/* instantiate the various classes in pkg2 */
public class Demo2
{
    public static void main(String args[])
    {

       Protection2 obj1 = new
       Protection2(); OtherPackage obj2 =
       new OtherPackage();
    }
}
```

Importing Packages

- import statement is used to include certain classes, or entire packages, into visibility.
- Once imported, a class can be directly referred using its name.

This is the general form of the import statement:

import pkg1 [.pkg2].(classname | *);

- pkg1 is the name of a top-level package, and pkg2 is the name of a sub package inside the outer package separated by a dot (.). There is no practical limit on the depth of a package hierarchy, except that imposed by the file system.
- specify either an explicit classname or a star (*), which indicates that the Java compiler should import the entire package.

Example:

```
package pack;

      public class Addition
      {

            int x,y;
            public Addition(int a, int b)
            {

                  x=a; y=b;
            }
            public void sum()
            {

                  System.out.println("Sum :"+(x+y));
            }

      }
```

Save the above file with Addition.java

```
package pack;
public class Subtraction
{
        int x,y;
        public Subtraction(int a, int b)
        {
                x=a; y=b;
        }
        public void diff()
        {
                System.out.println("Difference :"+(x-y));
        }
}
```

Save the above file with Subtraction.java

There are three ways to use package in another package:

- **With fully qualified name.**

```
class UseofPack
{
        public static void main(String arg[])
        {
                pack.Addition a=new
                pack.Addition(10,15); a.sum();
                pack.Subtraction s=new
                pack.Subtraction(20,15);
                s.difference();
        }
}
```

- **import package.classname;**

```
import
pack.Addition;
import
pack.Subtraction;
class UseofPack
{
        public static void main(String arg[])
        {
                 Addition a=new
                Addition(10,15); a.sum();
                Subtraction s=new Subtraction(20,15);
                s.difference();
        }
}
```

- **import package.*;**

```
impor
t
pack.
*;
class
Useof
Pack
{
        public static void main(String arg[])
        {

                 Addition a=new
                Addition(10,15); a.sum();
                Subtraction s=new Subtraction(20,15);
                s.difference();
        }
}
```

Chapter 10: Exception handling

Introduction:

An **Exception**, It can be defined as an abnormal event that occurs during program execution and disrupts the normal flow of instructions. The abnormal event can be an error in the program.

Errors in a java program are categorized into two groups:

- **Compile- time errors** occur when you do not follow the syntax of a programming language.
- **Run-time errors** occur during the execution of a program.

Concepts of Exceptions:

An exception is a run-time error that occurs during the exception of a java program.
Example :If you divide a number by zero or open a file that does not exist, an exception is raised. In java, exceptions can be handled either by the java run-time system or by a user-defined code . When a run-time error occurs, an exception is thrown.

The unexpected situations that may occur during program execution are:

1. Running out of memory
2. Resource allocation errors
3. In ability to find files
4. Problems in network connectivity

Exception handling techniques:

Java exception handling is managed via five keywords they are:

- try
- catch
- throw
- throws
- finally

Exception handling Statement Syntax

Exceptions are handled using a try-catch-finally construct, which has the Syntax:

```
try
{ <code>

}

catch(<exceptiontype1>    <parameter1>)

{ //0 or more <statements>


}finally

{

  //finally block <statements>
}
```

1. **try Block:** The java code that you think may produce an exception is placed with in a try block for a suitable catch block to handle the error.

If no exception occurs the execution proceeds with the finally block else it will look for the matching catch block to handle the error.Again if the matching catch handler is not found execution proceeds with the finally block and the default exception handler throws an exception.

2. **catch Block**: Exceptions thrown during execution of the try block can be caught and handled in a catch block. On exit from a catch block ,normal execution continues and the finally block is executed (Though the catch block throws an exception).

3. **finally Block:** A finally block is always executed, regardless of the cause of exit from the try block, or whether any catch block was executed. Generally, finally block is used for freeing resources, cleaning up, closing connections etc.

***Example**:*

The following is an array is declared with 2 elements. Then the code tries to access the 3rd element of the array which throws an exception.

```
//FileName:ExcepTest.java
importjava.io.*;
public class ExcepTest
{
        public static void main(String args[])
        {
            try
            { Int a[]= new int[2]; System.out.println("Accesselementthree:"+a[3]);   }
            catch(ArrayIndexOutOfBoundsException   e)
            { System.out.println("Exceptionthrown:"+e);  }
            System.out.println("Outoftheblock");
      }
}
```

This would produce following result:

```
Exceptionthrown:java.lang.ArrayIndexOutOfBoundsException:3 Out of the block
```

Multiple catch Blocks:

A try block can be followed by multiple catch blocks .The syntax for multiple catch blocks looks like the following:

```
try
 {

   //code
 }
 catch(ExceptionType1  e1)
 {

   //Catchblock
 }
 catch(ExceptionType2  e2)
 {

   //Catchblock
 }
 catch(ExceptionType3  e3)
 {

   //Catchblock
 }
```

The previous statements demonstrate three catch blocks, but you can have any number of them after a single try.

Example: Here is code segment showing how to use multiple try/catch statements.

```
 class Multi_Catch

 {          public static void main(String args[])
             {try
                         {

                                     int a=args.length;
                                     System.out. println(“a=”+a);
                                     Int b=50/a;
                                      Int c[]={1};
                         }

                         catch(ArithmeticException  e)
                         {
                                  System.out.println("Divisionbyzero");

                         }

                         catch(ArrayIndexOutOfBoundsException  e)

                         {

                                   System.out.println("arrayindexoutofbound");
                         }
         }
 }
```

```
OUTPUT

Division by zero

 array index out of bound
```

Nested try Statements

- Just like the multiple catch blocks, we can also have multiple try blocks . These try blocks may be written independently or we can nest the try blocks within each other, i.e., keep one try-catch block within another try- block.The program structure for nested try statement is:

Syntax

```
try

{
    //statements

    //statements

    try
    {

        //statements

        //statements

    }

    catch(<exception_two>  obj)

    {

        //statements
    }

    //statements

    //statements

}

catch(<exception_two> obj)
{
    //statements

}
```

- Consider the following example in which you are accepting two numbers from the command line. After that, the command line arguments, which are in the string format, are converted to integers.

- If the numbers were not received properly in a number format, then during the conversion a NumberFormatException is raised otherwise the control goes to the next try block. Inside this second try-catch block the first number is divided by the second number, and during the calculation if there is any arithmetic error, it is caught by the inner catch block.

Example

```
class Nested_Try
{      public static void main(String args[])
    {
        try
        {       int a = Integer.parseInt(args [0]);

               int b = Integer.parseInt (args [1]);
               int quot=0;

               try
               {
                   quot = a /b;
                   System.out.println(quot);
               }

               catch(ArithmeticException   e)
               {
                   System.out.println("dividebyzero");
               }

          }
          catch(NumberFormatException   e)
           {

               System.out.println("Incorrectargumenttype");
           }
    }
}
```

The output of the program is:

If the arguments are entered properly in the command prompt like:

```
OUTPUT
Java Nested_Try2 4 6
4
```

If the argument contains a string than the number:

```
Java Nested_Try 2 4 aa
Incorrect argument type
```

If the second argument is entered zero:

```
java Nested_Try2 4 0
divide by zero
```

throw Keyword

- **throw keyword** is used to throw an exception explicitly. Only object of throwable class or its sub classes can be thrown.
- Program execution stops on encountering **throw** statement, and the closest catch statement is checked for matching type of exception.

Syntax: **throw** *ThrowableInstance*

Creating InstanceofThrowableclass

There are two possible ways to get an instance of class Throwable,

1. Using a parameter in catch block.
2. Creating instance with **new** operator.

New NullPointerException("test");

This constructs an instance of NullPointerException with name test.

Example demonstrating throw Keyword

```
class Test
{       static void avg()
            {       try
                        {
                                throw new ArithmeticException("demo");
                         }
                         catch(ArithmeticException  e)
                        {
                                System.out.println("Exceptioncaught");
                         }
            }
            public static void main(String args[])
           {
                        avg();
            }
}
```

In the above example the avg() method throw an instance of Arithmetic Exception, which is successfully handled using the catch statement.

throws Keyword

1) Any method capable of causing exceptions must list all the exceptions possible during its execution, so that anyone calling that method gets a prior knowledge about which exceptions to handle. A method can do so by using the **throws** keyword.

Syntax:

```
type method_name(parameter_list)

throws exception_list

{

        //definition of method
}
```

NOTE: It is necessary for all exceptions, except the exceptions of type **Error** and **Runtime Exception**, or any of their subclass.

Example demonstrating throws Keyword

```
class Test

{

        static void check() throws ArithmeticException
        {

                System.out.println("Insidecheckfunction");
                thrownewArithmeticException("demo");
        }

        public static void main(String args[])
        {

                try
                {   check();   }              catch(ArithmeticException e)
                {

                        System.out.println("caught"+e);
                }

        }

}
```

finally

The finally clause is written with the try- catch statement. It is guaranteed to be executed after a catch block or before the method quits.

Syntax

try

```
{
    //statements
           }

catch(<exception>  obj)
{ //statements    }

finally
{  //statements  }
```

Take a look at the following example which has a catch and a finally block. The catch block catches the Arithmetic Exception which occurs for arithmetic error like divide-by-zero. After executing the catch block the finally is also executed and you get the out put for both the blocks

Example:

```
class Finally_Block

{
    static void division()
    {    try

         {
            int num = 34, den = 0;
            int quot=num/den;
         }

       catch(ArithmeticExceptione)

         {

            System.out.println("Dividebyzero");
         }
         finally
         {

            System.out.println("Inthefinallyblock");
         }
    }
class Mypgm

{

    public static void main(String args[])
    {

             Finally f=new Finally_Block();
             f.division();
    }

}
```

OUTPUT

Divide by zero
In the finally block

Java’s Built in Exceptions

Java defines several exception classes inside the standard package **java.lang**. The most general of these exceptions are subclasses of the standard type **RuntimeException**. Since **java.lang** is implicitly imported in to all Java programs , most exceptions derived from **RuntimeException** are automatically available.

Java defines several other types of exceptions that relate to its various class libraries . Following is the list of Java **UncheckedRuntimeException**

Exception	**Description**
ArithmeticException	Arithmeticerror,suchasdivide-by-zero.
ArrayIndexOutOfBoundsException	Arrayindexisout-of-bounds.
ArrayStoreException	Assignmenttoanarrayelementofaninc ompatibletype.
ClassCastException	Invalidcast.
IllegalArgumentException	Illegalargumentusedtoinvokeamethod.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on anunlockedthread.

Following is the list of **Java Checked Exceptions** Defined in java.lang.

Exception	**Description**
ArithmeticException	Arithmeticerror,suchasdivide-by-zero.
ArrayIndexOutOfBoundsException	Arrayindexisout-of-bounds.
ArrayStoreException	Assignmenttoanarrayelementofaninc ompatibletype.
ClassCastException	Invalidcast.
IllegalArgumentException	Illegalargumentusedtoinvokeamethod.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on anunlockedthread.
IllegalStateException	Environmentorapplicationisinincorrectstat e.
IllegalThreadStateException	Requestedoperationnotcompatiblewithcurrentt hreadstate.
IndexOutOfBoundsException	Sometypeof index isout-of-bounds.

NegativeArraySizeException	Arraycreatedwithanegativesize.
NullPointerException	Invaliduseofanullreference.
NumberFormatException	Invalid conversion of a string to a numericformat.
SecurityException	Attempttoviolatesecurity.
StringIndexOutOfBounds	Attempt to index outside the bounds of astring.
UnsupportedOperationException	Anunsupportedoperationwasencountered.

Creating your own Exception Subclasses

2) Here you can also define your own exception classes by extending **Exception**. These exception can represents specific runtime condition of course you will have to throw them yourself, but once thrown they will behave just like ordinary exceptions.
3) When you define your own exception classes, choose the ancestor carefully. Most custom exception will be part of the official design and thus checked, meaning that they extend Exception but not Runtime Exception.

Example:Throwing User defined Exception

```
     public class MyException extends Exception

     {

        String msg = "";intmarks=50;
        public MyException()
        {
        }

     publicMyException(Stringstr)
    {
    super(str);
    }

    Public String toString()

  {
    if(marks<=40)
       msg = "You have failed";
    if(marks>40)
      msg = "You have Passed";

    return msg;

  }

}
```

```
class test

{
    public static void main(String args[])
        {

                test t = new test();
                t.add();
        }
    Public void add()

     {
          try
           {
               int i=0;

               if(i< 40)

               throw new MyException();

            }
            catch(MyException ee1)

            {

                 System.out.println("Result:"+ ee1);
            }
       }

}
```

OUTPUT

```
Result: You have Passed
```

Chained Exception

- Chained exceptions are the exceptions which occur one after another i.e.most of the time to response to an exception are given by an application by throwing another exception.
- Whenever in a program the first exception causes an another exception, that is termed as **Chained Exception**. Java provides new functionality for chaining exceptions.
- Exception chaining (also known as "nesting exception") is a technique for handling the exception, which occur one after another i.e. most of the time is given by an application to response to an exception by throwing another exception.
- Typically, the second exception is caused by the first exception. There fore chained exceptions help the programmer to know when one exception causes another.
- The **constructors** that support chained exceptions in **Throwable** class are:

  Throwable init Cause
  (Throwable) Throwable (
  Throwable)
  Throwable (String, Throwable)

Throwable getCause()

MODULE 5

Chapter 11: Multithreading

Introduction:

The program in execution is called "**Process**". The program can be structured as set of individual units that can run in parallel. These units can be called as "**Threads**". Multithreading is actually akind of multitasking. The multitasking is either process-based or thread base. Process-based multitasking is nothing but, execution of more than one program concurrently. Thread-based multitasking allows more than one thread within the program run simultaneously orconcurrently.The process is a Heavy Weight Process. The Thread is a Light Weight Process. The context-switching of CPU from one process to other requires more overhead as it different address spaces are involved in it. On the other hand, context-switching is less overhead because of all the threads within the same program. The objective of all forms of the Multitasking including the multithreading is to utilize the idle time of the processor. From here onwards, thread-based multitasking is called "**Multithreading**".

Difference between Multiprocessing and Multithreading

Process-Based Multitasking	Thread-Based Multitasking
This deals with "Big Picture"	This deals with Details
These are Heavyweight tasks	These are Lightweight tasks
Inter-process communication is expensive and limited	Inter-Thread communication is inexpensive.
Context switching from one process to another is costly in terms of memory	Context switching is low cost in terms of memory, because they run on the same address space
This is not under the control of Java	This is controlled by Java

Multithreading in Java

Every program that we have been writing has at least one thread, that is, the "main" thread. Whenever a program starts executing, the JVM is responsible for creating the main thread and calling "main()" method. Along with this main thread, some other threads are also running to carryout the tasks such as "finalization" and "garbage collection". The thread can either die naturally or be forced to die.

- Thread dies naturally when it exits from the "run()" method.

Thread can be forced to die by calling "interrupt()" method.

<u>java.lang.Thread package</u>

Creation of Thread in java is very simple task. There is a class called "Thread", which belongs tothe "java.lang.Thread" package. This package contains one interface also called "Runnable". Both these contain a common method called "**run()**" which is the heart of the thread. The run() methods would have the following syntax:

Syntax:

The methods of the Thread class are as follow:

- **public void run():** is used to perform action for a thread.
- **public void start():** starts the execution of the thread.JVM calls the run() methodon the thread.
- **public void sleep(long miliseconds):** Causes the currently executing thread tosleep (temporarily cease execution) for the specified number of milliseconds.
- **public void join():** waits for a thread to die.
- **public void join(long miliseconds):** waits for a thread to die for the specifiedmiliseconds.
- **public int getPriority():** returns the priority of the thread.
- **public int setPriority(int priority):** changes the priority of the thread.
- **public String getName():** returns the name of the thread.
- **public void setName(String name):** changes the name of the thread.
- **public Thread currentThread():** returns the reference of currently executingthread.
- **public int getId():** returns the id of the thread.
- **public Thread.State getState():** returns the state of the thread.
- **public boolean isAlive():** tests if the thread is alive.
- **public void yield():** causes the currently executing thread object to temporarilypause and allow other threads to execute.
- **public void suspend():** is used to suspend the thread(depricated).
- **public void resume():** is used to resume the suspended thread(depricated).
- **public void stop():** is used to stop the thread(depricated).
- **public boolean isDaemon():** tests if the thread is a daemon thread.
- **public void setDaemon(boolean b):** marks the thread as daemon or user thread.
- **public void interrupt():** interrupts the thread.
- **public boolean isInterrupted():** tests if the thread has been interrupted.
- **public static boolean interrupted():** tests if the current thread has beeninterrupted.

Thread Constructors:

- Thread ()-without arguments, default constructor
- Thread(String str)- Thread contains name given as argument

The Main Thread

Every java program has a thread called "main" thread. When the program execution starts, the JVM creates "main" Thread and calls the "main()" method from within that thread. Along with this JVM also creates other threads for the purpose of the Housekeeping task such as "garbage" collection. The "main" thread Spawns the other Threads. These spawned threads are called"Child Threads". The main thread is always is the last thread to finish execution. We, asProgrammer can also take control of the main thread, using the method "**currentThread()**". The main thread can be controlled by this method. We can also change the name of the Thread using the method "**setName(String name)**".

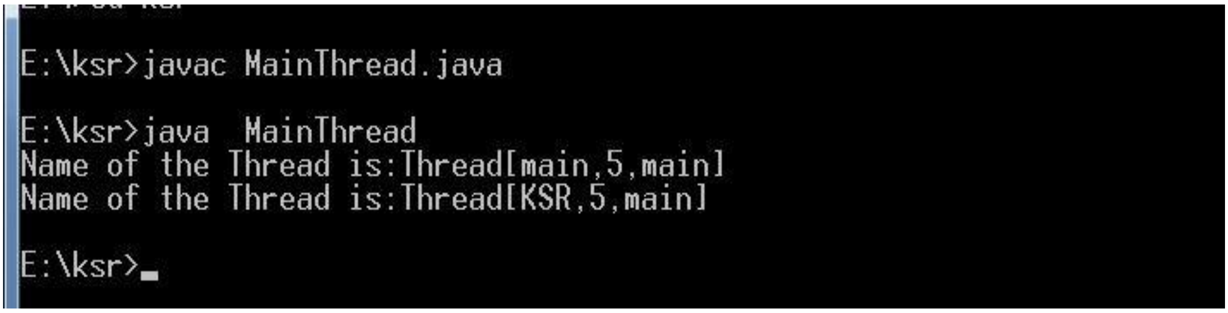
MainThread.java

public static void main(String args[])

Example Program:

```
class MainThread{
        {
        Thread t=Thread.currentThread();
        System.out.println("Name of the Thread
        is:"+t);t.setName("KSR");
        System.out.println("Name of the Thread is:"+t);
        }
}
```

Output:



```
E:\ksr>javac MainThread.java

E:\ksr>java  MainThread
Name of the Thread is:Thread[main,5,main]
Name of the Thread is:Thread[KSR,5,main]

E:\ksr>_
```

Creation of Threads

Creating the threads in the Java is simple. The threads can be implemented in the form ofobject that contains a method "**run()**". The "**run()**" method is the heart and soul of any thread. Itmakes up the entire body of the thread and is the only method in which the thread behavior can be implemented. There are two ways to create thread.

- Declare a class that **extends** the **Thread** class and override the **run()** method.
- Declare a class that implements the **Runnable** interface which contains the **run()** method

- **Creating Thread using The Thread Class**

We can make our thread by extending the Thread class of java.lang.Thread class. This gives usaccess to all the methods of the Thread. It includes the following steps:

- Declare the class as Extending the Thread class.
- Override the "**run()**" method that is responsible for running the thread.
- Create a thread and call the "**start()**" method to instantiate the Thread Execution.

Declaring the class

TheThread class can be declared as

```
        follows:class MyThread
        extends Thread
        {
   ---------------
   ---------------
   ---------------
   ---------------
```

```
          }
```

Overriding the method run()

The run() is the method of the Thread. We can override this as

```
          follows:public void run()
          {
     - - - - - - - - - - -
     - - - - - - - - - - -
     - - - - - - - - - - -


          }
```

Starting the new Thread

To actually to create and run an instance of the thread class, we must write the following:

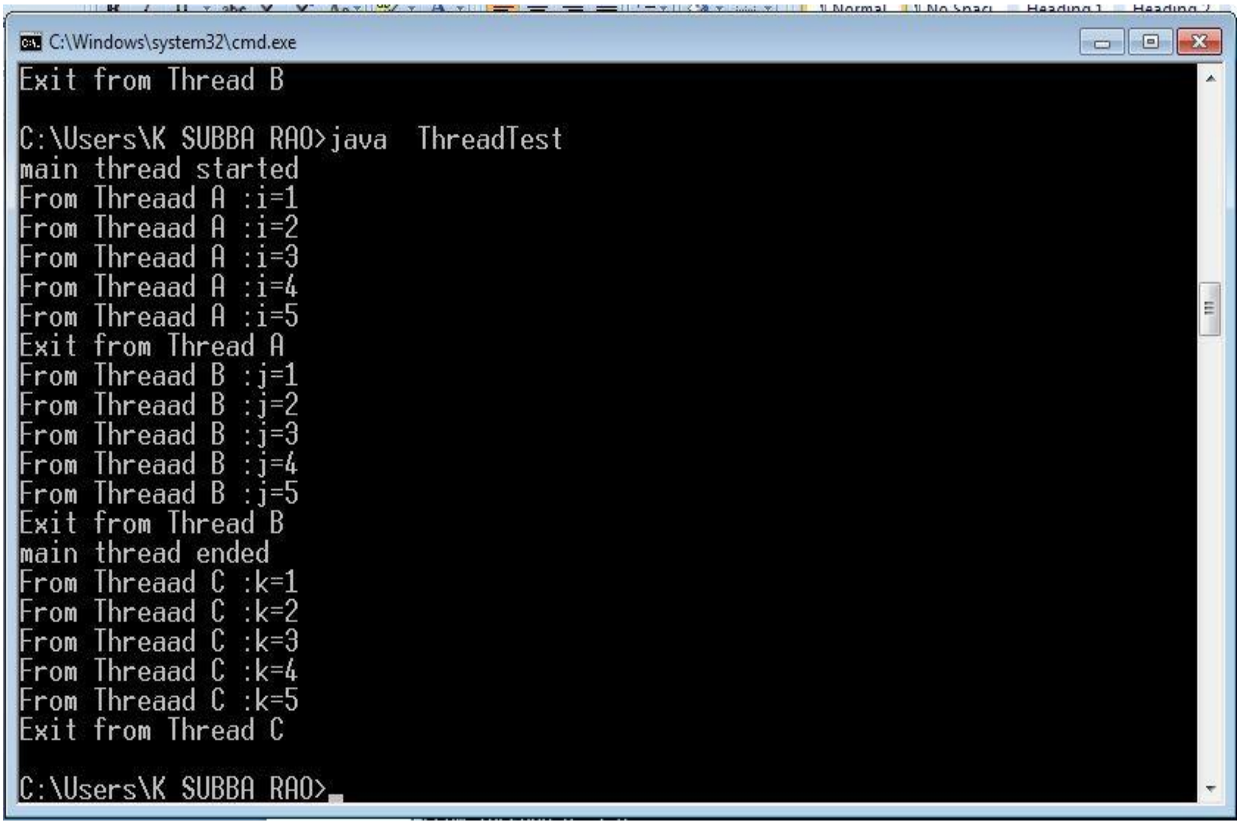
MyThread a=new MyThread(); // creating
the Threada.start();   // Starting
the Thread

Example program:

```
import java.io.*;
import java.lang.*;

class A extends Thread
          public void run()
          {
                    for(int i=1;i<=5;i++)
                    {
                              System.out.println("From Threaad A :i="+i);
                    }
                    System.out.println("Exit from Thread A");
          }
}
class B extends Thread
{
          public void run()
          {
                    for(int j=1;j<=5;j++)
                    {
                              System.out.println("From Threaad B :j="+j);
                    }
                    System.out.println("Exit from Thread B");
          }
}
class C extends Thread
{
          public void run()
          {
                    for(int k=1;k<=5;k++)
                    {
                              System.out.println("From Threaad C :k="+k);
                    }
```

```
                        System.out.println("Exit from Thread C");
            }
}



class ThreadTest
{
            public static void main(String args[])
            {
                        System.out.println("main thread started");
                        A a=new A();
                        a.start();
                         B b=new B();
              b.start();
                        C c=new C();
                        c.start();
                        System.out.println("main thread ended");
            }
}
```

Output: First Run



```
Exit from Thread B

C:\Users\K SUBBA RAO>java  ThreadTest
main thread started
From Threaad A :i=1
From Threaad A :i=2
From Threaad A :i=3
From Threaad A :i=4
From Threaad A :i=5
Exit from Thread A
From Threaad B :j=1
From Threaad B :j=2
From Threaad B :j=3
From Threaad B :j=4
From Threaad B :j=5
Exit from Thread B
main thread ended
From Threaad C :k=1
From Threaad C :k=2
From Threaad C :k=3
From Threaad C :k=4
From Threaad C :k=5
Exit from Thread C

C:\Users\K SUBBA RAO>
```

<u>Creating the Thread using Runnable Interface</u>

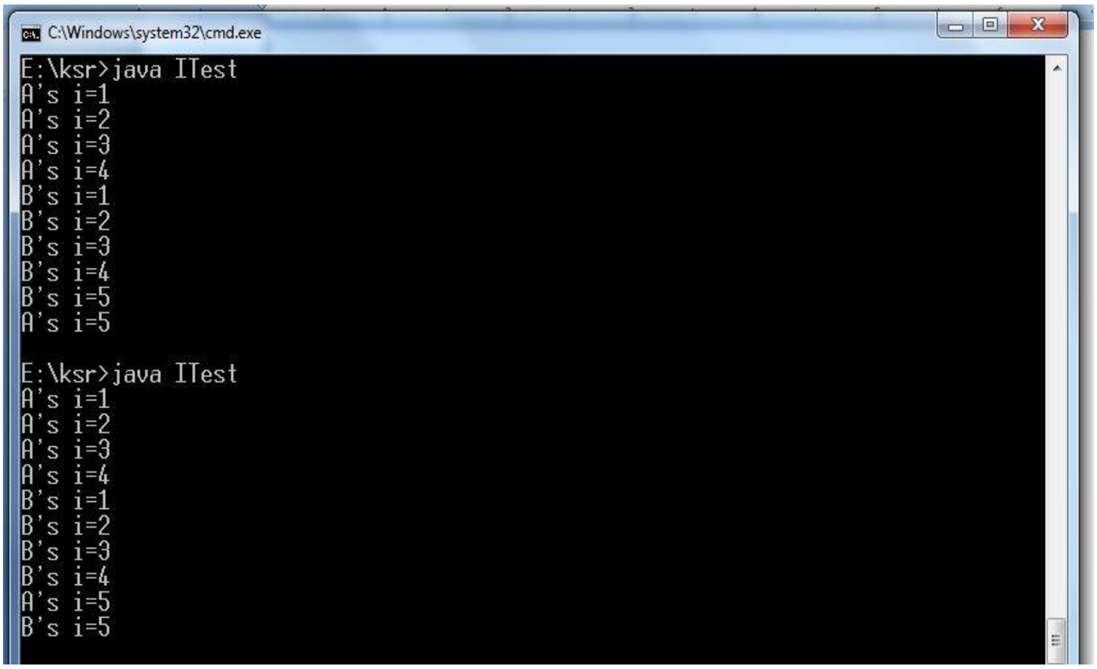
The Runnable interface contains the run() method that is required for implementing the threads in our program. To do this we must perform the following steps:

- Declare a class as implementing the **Runnable** interface
- Implement the **run()** method
- Create a **Thread** by defining an object that is instantiated from this "runnable" class as
- the target of the thread
- Call the thread's **start()** method to run the thread.

Example program:

```
class A implements Runnable
	{
		public void run()
		{
		for(int i=1;i<=5;i++)
		{
			System.out.println("A's i="+i);
		}
		}
	}
	class B implements Runnable
	{
		public void run()
		{
		for(int i=1;i<=5;i++)
		{
			System.out.println("B's i="+i);
		}
		}
	}
	class ITest
	{
		public static void main(String args[])
		{
			A a=new A();
			Thread t1=new
			Thread(a);
			t1.start();
			B b=new B();
			Thread t2=new
			Thread(b);
			t2.start();
		}

	}
```

Output: Threads A and B execution by running the above program two times. (You may see a differentsequence of Output, every time you run this program)



```
E:\ksr>java ITest
A's i=1
A's i=2
A's i=3
A's i=4
B's i=1
B's i=2
B's i=3
B's i=4
B's i=5
A's i=5

E:\ksr>java ITest
A's i=1
A's i=2
A's i=3
A's i=4
B's i=1
B's i=2
B's i=3
B's i=4
A's i=5
B's i=5
```

Advantage of the Multithreading

- It enables you to write very efficient programs that maximizes the CPU utilization and reducesthe idle time.
- Most I/O devices such as network ports, disk drives or the keyboard are much slower than CPU
- A program will spend much of it time just send and receive the information to or from thedevices, which in turn wastes the CPU valuable time.
- By using the multithreading, your program can perform another task during this idle time.
- For example, while one part of the program is sending a file over the internet, another part canread the input from the keyboard, while other part can buffer the next block to send.
- It is possible to run two or more threads in multiprocessor or multi core systems simultaneously.

Thread States

A thread can be in one of the several states. In general terms, a thread can ***running***. It can be ***ready*** to run as soon as it gets the CPU time.A running thread can be ***suspended***, which is a temporary halt to its execution. It can later be ***resumed***. A thread can be ***blocked*** when waiting for the resource. A thread can be ***terminated***.

Single Threaded Program

A Thread is similar to simple program that contains single flow of control. It has beginning, body,and ending. The statements in the body are executed in sequence.

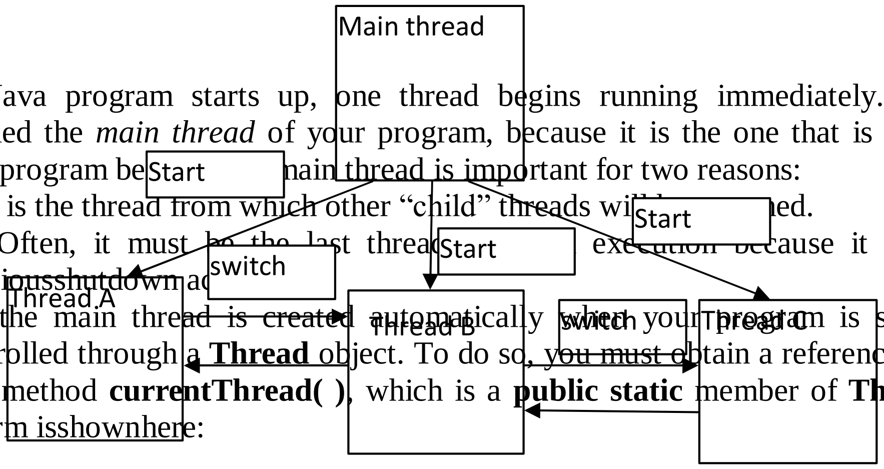
For example:

```
class ABC                          //Begining

        {
                   -----------//bodv
                   ---------
        }                          //ending
```

Multithreaded Program

- A unique property of the java is that it supports the multithreading. Java enables us the multiple flows of control in developing the program.

- Each separate flow of control is thought as tiny program known as "**thread**" that runs in parallel with other threads.
- In the following example when the main thread is executing, it may call thread A, as the Thread A is in execution again a call is mad for Thread B. Now the processor is switched from Thread A to Thread B. After the task is finished the flow of control comes back to the Thread A.
- The ability of the language that supports multiple threads is called "***Concurrency***". Since threads in the java are small sub programs of the main program and share the same address space, they are called "***light weight processes***".



When a Java program starts up, one thread begins running immediately. This is usuallycalled the *main thread* of your program, because it is the one that is executed when yourprogram be[illegible] main thread is important for two reasons:

- It is the thread from which other “child” threads wi[illegible]ned.
- Often, it must be the last thread [illegible] execution because it performs variousshutdown ac[illegible]

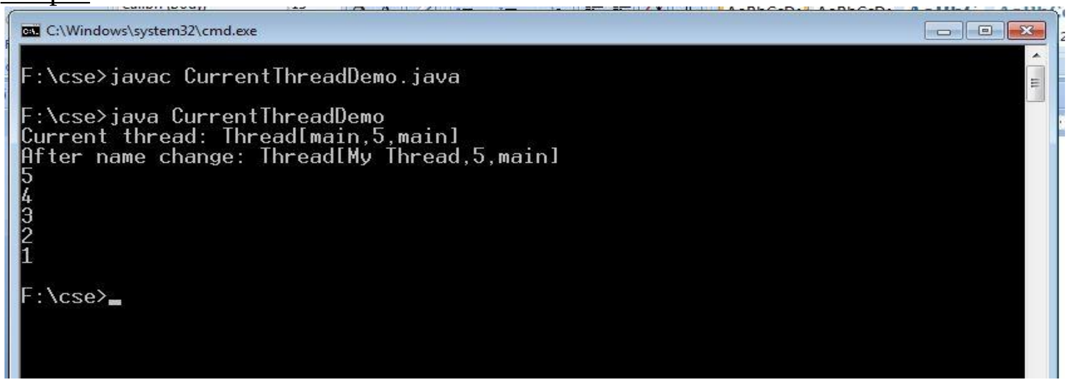
Although the main thread is created automatically when your program is started, it canbecontrolled through a **Thread** object. To do so, you must obtain a reference to it by callingthe method **currentThread( )**, which is a **public static** member of **Thread**. Its general form isshownhere:

static Thread.currentThread( )

This method returns a reference to the thread in which it is called. Once you have a referenceto the main thread, you can control it just like any other thread.
Let’s begin by reviewing the following example:

```
// Controlling the main Thread.class
CurrentThreadDemo
{
public static void main(String args[])
{
        Thread t = Thread.currentThread();
        System.out.println("Current thread: " + t);
        // change the name of the thread
  t.setName("My Thread");
        System.out.println("After name change: " + t);
        try
  {      for(int n = 5; n > 0; n--)
         {
                 System.out.println(n);Thread.sleep(1000);
          }
         }
         catch (InterruptedException e)
         {
                 System.out.println("Main thread interrupted");
         }
  }
}
```

Output



C:\Windows\system32\cmd.exe

```
F:\cse>javac CurrentThreadDemo.java

F:\cse>java CurrentThreadDemo
Current thread: Thread[main,5,main]
After name change: Thread[My Thread,5,main]
5
4
3
2
1

F:\cse>_
```

Creating Multiple Threads

So far, you have been using only two threads: the main thread and one child thread. However,your program can spawn as many threads as it needs. For example, the following program creates three child threads:

```
// Create multiple threads.
class NewThread implements Runnable
{
String name; // name of thread
Thread t;
NewThread(String threadname)
{
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New
        thread: " + t);t.start(); // Start
        the thread
}// This is the entry point for thread.
public void run()
{     try
        {
                for(int i = 5; i > 0; i--)
                {
                        System.out.println(name + ": " + i);
                        Thread.sleep(1000);
                }
        } catch (InterruptedException e)
        {
                System.out.println(name + "Interrupted");
        }
System.out.println(name + " exiting.");
}//end of run  method
}//end of NewThread
class MultiThreadDemo
{
public static void main(String args[])
{
new NewThread("One"); // start threads
new  NewThread("Two");
new NewThread("Three");
try
{
        // wait for other threads to end
        Thread.sleep(10000);
}
 catch (InterruptedException e)
{
        System.out.println("Main thread Interrupted");
}
System.out.println("Main thread exiting.");
}
}
```

When a Thread is ended

It is often very important to know which thread is ended. This helps to prevent the main fromterminating before the child Thread is terminating. To address this problem "Thread" class provides two methods: **1) Thread.isAlive() 2) Thread.join()**.

The general form of the **"isAlive()"** method is as follows:

final boolean isAlive();

This method returns the either "**TRUE**" or "**FALSE**" . It returns "TRUE" if the thread is alive,returns "FALSE" otherwise. While **isAlive( )** is occasionally useful, the method that you will more commonly use towait for athread to finish is called **join( )**, shown here:

final void join( ) throws InterruptedException

This method waits until the thread on which it is called terminates. Its name comes from theconcept of the calling thread waiting until the specified thread *joins* it. Additional forms of**join( )** allow you to specify a maximum amount of time that you want to wait for thespecifiedthread to terminate.

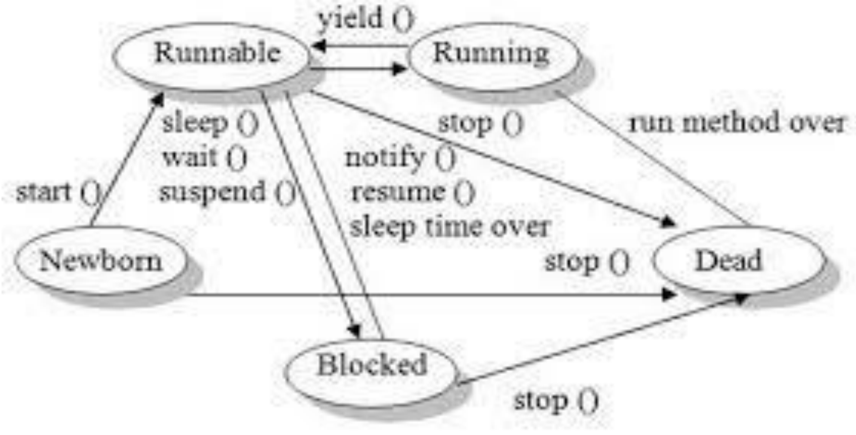
Example Program:

```
// Using join() to wait for threads
to finish.class NewThread
implements Runnable
{
        String name; // name
        of threadThread t;
        NewThread(String threadname)
        {
                name = threadname;
                t = new Thread( name);
                System.out.println("New thread: " +
                t.getName());t.start(); // Start the thread
        }
// This is the entry point for thread.
public void run()
{
try
 {      for(int i = 5; i > 0; i--)
        {
                System.out.println(name + ": " + i);
                Thread.sleep(1000);
        }
}
 catch (InterruptedException e)
        {
                System.out.println(name + " interrupted.");
        }
System.out.println(name + " exiting.");
}
}


class DemoJoin
{
public static void main(String args[])
{
        NewThread     ob1     =     new     NewThread("One");
        NewThread     ob2     =     new     NewThread("Two");
        NewThread ob3 = new NewThread("Three");
        System.out.println("Thread   One   is   alive:   "+   ob1.t.isAlive());
        System.out.println("Thread   Two   is   alive:   "+   ob2.t.isAlive());
        System.out.println("Thread Three is alive: "+ ob3.t.isAlive());
        // wait for threads to finish
        try {
                System.out.println("Waiting for threads to finish.");
                ob1.t.join();
                ob2.t.join();
                ob3.t.join();
        }
         catch (InterruptedException e)
        {
                System.out.println("Main thread Interrupted");
        }
        System.out.println("Thread One is alive: "+ ob1.t.isAlive());
        System.out.println("Thread Two is alive: "+ ob2.t.isAlive());
        System.out.println("Thread Three is alive: "+ ob3.t.isAlive());
        System.out.println("Main thread exiting.");
}
}
```

Life Cycle of a Thread

During the life time of the thread, there are many states it can enter. They include the following:

- Newborn state
- Runnable State
- Running State
- Blocked state
- Dead state



Life-cycle of a thread.

A thread can always in any one of the five states. It can move from one state to other via variety ofways as shown in the fig.

Newborn State: When we create a thread it is said to be in the new born state. At this state we can dothe following:

- schedule it for running using the start() method.
- Kill it using stop() method.

Runnable State: A runnable state means that a thread is ready for execution and waiting for the availability of the processor. That is the thread has joined the queue of the threads for execution. If all the threads have equal priority, then they are given time slots for execution in the round rabin fashion, first-come, first-serve manner. The thread that relinquishes the control will join the queue at the end and again waits for its turn. This is known as time slicing.

Running state: Running state means that the processor has given its time to the thread for it execution. The thread runs until it relinquishes the control or it is preempted by the other higher priority thread. As shown in the fig. a running thread can be preempted using the suspen(), or wait(), or sleep() methods.

Blocked state: A thread is said to be in the blocked state when it is prevented from entering into runnable state and subsequently the running state.

Dead state: Every thread has a life cycle. A running thread ends its life when it has completed execution. It is a natural death. However we also can kill the thread by sending the stop() message to it at any time.

The Thread Priorities

Thread priorities are used by the thread ***scheduler*** to decide when and which thread should beallowed to run. In theory, **higher-priority** threads get more CPU time than **lower- priority**threads. In practice, the amount of CPU time that a thread gets often depends on **several factors** besides its priority. (For example, how an operating system implements ***multitasking***can affect the relative availability of CPU time.) A higher-priority thread can also **preempt**(stop) alower-priority one. For instance, when a lower-priority thread is running and a higher- prioritythread **resumes** (from sleeping or waiting on I/O, for example), it will preempt the lowerprioritythread.

To set a thread's priority, use the **setPriority( )** method, which is a member of **Thread**.This is its general form:

final void setPriority(int *level*)

Here, *level* specifies the new priority setting for the calling thread. The value of *level* must bewithin the range **MIN_PRIORITY** and **MAX_PRIORITY**. Currently, these values are 1 and10, respectively. To return a thread to default priority, specify **NORM_PRIORITY**, which iscurrently 5. These priorities are defined as **static final** variables within **Thread**.

You can obtain the current priority setting by calling the **getPriority( )** method of **Thread**,shownhere:

final int getPriority( )

Example Program:

//setting the priorities for thethread

```
        class PThread1 extends Thread
        {
                public void run()
                {
                        System.out.println(" Child 1 is started");
                }
        }
        class PThread2 extends Thread
        {
                public void run()
                {
                        System.out.println(" Child 2 is started");
                }
        }
        class PThread3 extends Thread
        {
                public void run()
                {
                        System.out.println(" Child 3 is started");
                }
        }

class PTest
{
   public static void main(String args[])
   {
    //setting the priorities to the thread using the setPriority() method
        PThread1 pt1=new PThread1();
                pt1.setPriority(1);
        PThread2 pt2=new PThread2();
                pt2.setPriority(9);
        PThread3 pt3=new PThread3();
        pt3.setPriority(6);
         pt1.start();
         pt2.start();
         pt3.start();

         //getting the priority
         System.out.println("The pt1 thread priority is :"+pt1.getPriority());
    }
}
```

Synchronization

When two or more threads need access to a shared resource, they need some way to ensurethat the resource will be used by only one thread at a time. The process by which this isachieved is called ***synchronization***.

Key to synchronization is the concept of the **monitor** (also called a *semaphore*). A *monitor*is an object that is used as a mutually exclusive lock, or ***mutex***. Only one thread can *own* amonitor at a given time. When a thread acquires a lock, it is said to have ***entered***the monitor.Allother threads attempting to enter the locked monitor will be suspended until the firstthread ***exits***the monitor. These other threads are said to be ***waiting***for the monitor. A threadthat owns a monitor can reenter the same monitor if it so desires.

Let us try to understand the problem without synchronization. Here, in the following example to threads are accessing the same resource (object) to print the Table. The Table class contains one method, printTable(int), which actually prints the table. We are creating two Threads, Thread1 and Thread2, which are using the same instance of the Table Resource (object), to print the table. When one thread is using the resource, no other thread is allowed to access the same resource Table to print the table.

Example without the synchronization:

```
Class Table
{
        void printTable(int n)
        {//method not synchronized
                for(int i=1;i<=5;i++)
              {
                 System.out.println(n*i);
                 try{
                         Thread.sleep(400);
                 }
                 catch(InterruptedException ie)
                 {
                            System.out.println("The Exception is :"+ie);
                     }
              }
```

```
                }  //end of the printTable() method
}

class MyThread1 extends Thread
{
Table t;
            MyThread1(Table mt)
            {
                    t=mt;
            }

            public void run()
            { t.printTable(5);
            }
 }  //end of the Thread1

class MyThread2 extends Thread
{
Table t;
            MyThread2(Table mt)
            {
                    t=mt;
            }
            public void run()
            {
                    t.printTable(100);
            }
}  //end of Thread2

class TestSynchronization1
{
public static void main(String args[])
{
Table obj = new Table();//only one objectMyThread1
t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

The output for the above program will be as follow:

```
    Output: 5
             100
             10
             200
             15
             300
             20
             400
```

```
25
500
```

In the above output, it can be observed that both the threads are simultaneously accessing the Table object to print the table. Thread1 prints one line and goes to sleep, 400 milliseconds, and Thread1 prints its task.

Using the Java synchronized method

If you declare any method as synchronized, it is known as synchronized method.Synchronizedmethod is used to lock an object for any shared resource.When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the threadcompletes its task.

The general form of the synchronized method is:

```
synchronized type method_name(para_list)
{
        //body of the method
}
```

where synchronized is the keyword, method contains the type, and method_name represents thename of the method, and para_list indicate the list of the parameters

Example using the synchronized method

```
Class Table
{           synchronized void printTable(int n)
             {//method not synchronized
                        for(int i=1;i<=5;i++)
                      {
                          System.out.println(n*i);
                          try{
                                  Thread.sleep(400);
                          }
                          catch(InterruptedException ie)
                          {
                                      System.out.println("The Exception is :"+ie);
                              }
                      }
               }  //end of the printTable() method
}

class MyThread1 extends Thread
{
Table t;
            MyThread1(Table t)
            {
                    this.t=t;
            }
            public void run()
```

```
                {
                          t.printTable(5);
                }
 }  //end of the Thread1

class MyThread2 extends Thread
{
Table t;
                MyThread2(Table t)
                {
                    this.t=t;
            }
            public void run()
            {
                    t.printTable(100);
            }
}  //end of Thread2

class TestSynchronization1
{
public static void main(String args[])
{
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

```
Output: 5
        10
        15
        20
        25
              100
              200
              300
              400
              500
```

In the above output it can be observed that when Thread1 is accessing the Table object, Thread2 is not allowed to access it. Thread1 preempts the Thread2 from accessing the printTable() method.

Note:

This way of communications between the threads competing for same resource is called **implicit communication.**
This has one disadvantage due to polling. The polling wastes the CPU time. To save the CPU time, it is preferred to go to the **inter-thread communication.**

Inter-Thread Communication

If two or more Threads are communicating with each other, it is called "inter thread" communication. Using the synchronized method, two or more threads can communicate indirectly. Through, synchronized method, each thread always competes for the resource. This way of competing is called **polling**. The polling wastes the much of the CPU valuable time. The better solution to this problem is, just notify other threads for the resource, when the current thread has finished its task, meanwhile other threads will be doing some useful work.. This is **explicit communication** between the threads.

Java addresses this polling problem, using via **wait()**, **notify()**, and **notifyAll()** methods. These methods are implemented as **final**methods in **Object**, so all classes have them. All three methodscan be called only fromwithin a **synchronized** context.

- **wait( )** tells the calling thread to give up the monitor and go to sleep until someother thread enters the same monitor and calls **notify( )**.
- **notify( )** wakes up a thread that called **wait( )** on the same object.
- **notifyAll( )** wakes up all the threads that called **wait( )** on the same object.
One ofthe threads will be granted access.

These methods are declared within Object, as shown here:

final void wait( ) throws InterruptedException
final void notify( )
final void notifyAll( )
Additional forms of **wait( )** exist that allow you to specify a period of time to wait.

Although **wait( )** normally waits until **notify( )** or**notifyAll( )** is called, there is a possibility that in very rare cases the waiting thread could beawakened due to a *spurious wakeup*.In this case, a waiting thread resumes without **notify( )**or **notifyAll( )** having been called. (In essence, the thread resumes for no apparent reason.)Because of this remote possibility, Sun recommends that calls to **wait( )** should take placewithin a loop that checks the condition on which the thread is waiting. The followingexample shows this technique.

Example program for producer and consumer

```
problemclass Q
{
        int n;
        boolean valueSet = false;//flag
        synchronized int get()
        {
        while(!val
        ueSet)try
        {
                wait();
        }
         catch(InterruptedException e)
        {
                System.out.println("InterruptedException caught");
        }
        System.out.println("Got: " + n);
```

```
    valueSet = false;notify();
    return n;
    } //end of the get() method
    synchronized void put(int n)
    {
    while(valueSet)
    try {
        wait();
    }
    catch(InterruptedException e)
    {
        System.out.println("InterruptedException caught");
    }
    this.n = n;
    valueSet = true;
    System.out.println("Put: " + n);
    notify();
    }//end of the put method
} //end of the class Q

class Producer implements Runnable
{
Q q;
Produce
r(Q q)
{
this.q = q;
new Thread(this, "Producer").start();
}
public void run()
{
int i = 0;
    while(true)
    {
        q.put(i++);
    }
}
}//end of Producer
class Consumer implements Runnable
{
    Q
q;
Consume
r(Q q)
{
this.q = q;
new Thread(this, "Consumer").start();
}
```

```
public void run()
{
        while(true)
        {
                q.get();
        }
}
}//end of Consumer


class PCFixed
{
public static void main(String args[])
{
        Q q = new Q();
        new Producer(q);
        new Consumer(q);
        System.out.println("Press Control-C to stop.");
}
}
```

Suspending, Blocking and StoppingThreads

Whenever we want stop a thread we can stop from running using "**stop()**" method of thread class. It's general form will be as follows:

T**hread.stop();**

This method causes a thread to move from **running** to **dead** state. A thread will also move todead state automatically when it reaches the end of its method.

Blocking Thread

A thread can be temporarily suspended or blocked from entering into the runnable and running state by using the following methods:

sleep()	—blocked for specified time
suspend()	----blocked until further orders
wait()	--blocked until certain condition occurs

These methods cause the thread to go into the blocked state. The thread will return to the runnable state when the specified time is elapsed in the case of **sleep()**, the **resume()** method is invokedin the case of **suspend()**, and the **notify()** method is called in the case of **wait()**.

Example program:

The following program demonstrates these methods:

// Using suspend() and resume().

```
class NewThread implements Runnable
{
        String name; // name of
        threadThread t;
        NewThread(String threadname)
        {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread:
        " + t);t.start(); // Start the thread
}
// This is the entry point for thread.
public void run()
{
        try
        {
                    for(int i = 15; i > 0; i--)
                    {
                    System.out.println(name
                    + ": " + i);
                    Thread.sleep(200);
        }           }
         catch (InterruptedException e)
        {
                System.out.println(name + " interrupted.");
        }
System.out.println(name + " exiting.");
}
}
class SuspendResume
{
public static void main(String args[])
{
NewThread ob1 = new NewThread("One");
NewThread ob2 = new NewThread("Two");try
{
        Thread.sleep(1000);
        ob1.t.suspend();
        System.out.println("Suspending thread One");
        Thread.sleep(1000);
        ob1.t.resume(); System.out.println("Resuming thread One");
        ob2.t.suspend();
        System.out.println("Suspending thread Two");
        Thread.sleep(1000);
        ob2.t.resume(); System.out.println("Resuming thread Two");
}
 catch (InterruptedException e)
{System.out.println("Main thread Interrupted");
}
// wait for threads to
finishtry
{
        System.out.println("Waiting for threads to finish.");
        ob1.t.join();
        ob2.t.join();
}
```

```
 catch (InterruptedException e)
{
            System.out.println("Main thread Interrupted");
}
System.out.println("Main thread exiting.");
}
}
```

Thread Exceptions:

Note that a call to the sleep() method is always enclosed in try/ catch block. This is necessary because the sleep() method throws an exception, which should be caught. If we fail to catch the exception the program will not compile.

it general form will be as follows:try

```
{
            Thread.sleep(1000);
}
cathc(Exception e)
{           --------
------

}
```

Chapter 12: Enumerations

- An enumeration is a list of named constants.
- Java enumerations are class types.
- Each enumeration constant is an object of its enumeration type.
- Each enumeration constant has its own copy of any instance variables defined by theenumeration.
- All enumerations automatically inherit one: **java.lang.Enum**.

```
// An enumeration of apple varieties.
    enum Apple {
            A, B, C, D, E
    }
```

- ➢ The identifiers Jonathan, GoldenDel, and so on, are called enumeration constants.
- ➢ Each is implicitly declared as a public, static, final member of Apple.
- ➢ In the language of Java, these constants are called **self-typed.**

Built-in Methods of ENUM

2 methods: values() and valueOf()

The values( ) and valueOf( ) Methods

All enumerations automatically contain two predefined methods: **values( )** and **valueOf( ).**

Their general forms are shown here:

public static enum-type [ ] values( )

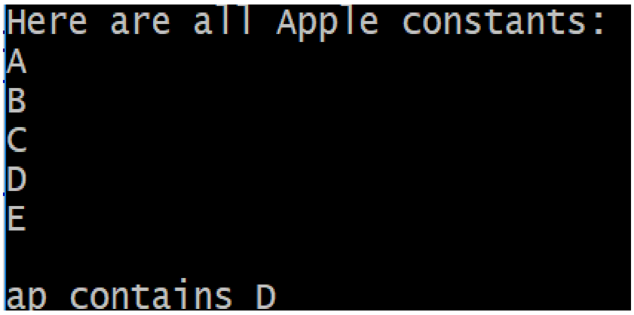
public static enum-type valueOf(String str)

- The **values( )** method returns an array that contains a list of the enumeration constants.
- The **valueOf( )** method returns the enumeration constant whose value corresponds tothe string passed in str.

Enum Example:

```
// An enumeration of apple varieties.
enum Apple {
        A, B, C, D, E
}
class EnumDemo {
   public static void main(String args[]) {
   Apple ap;
        System.out.println("Here are all Apple constants:");
        // use values()
        Apple allapples[] = Apple.values();
```

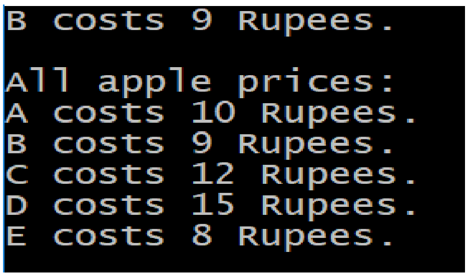
```
            for(Apple a : allapples)
                System.out.println(a);
                System.out.println();
              // use valueOf()
             ap = Apple.valueOf("Winesap");
            System.out.println("ap contains " + ap);
  }
}
```



```
Here are all Apple constants:
A
B
C
D
E

ap contains D
```

Java enumerations with the constructors, instance variables andmethod

```
// Use an enum constructor, instance variable, and method.
enum Apple {
        A(10), B(9), C(12), D(15), E(8);
        private int price; // price of each apple

         ConstructorApple(int p) {
                price = p;
        }
        int getPrice() {
                return price;
        }
}
class EnumConsDemo {
        public static void main(String args[]) {Apple ap;

                // Display price of B.
                System.out.println("B costs " + Apple.B.getPrice() + " cents.\n");

                // Display all apples and prices.
                System.out.println("All apple prices:");
                for(Apple a : Apple.values())
                        System.out.println(a + " costs " + a.getPrice() +" cents.");
                }
}
```



```
B costs 9 Rupees.

All apple prices:
A costs 10 Rupees.
B costs 9 Rupees.
C costs 12 Rupees.
D costs 15 Rupees.
E costs 8 Rupees.
```

Type Wrappers (Autoboxing and autounboxing)

- Type wrappers are classes that encapsulate a primitive type within an object.
- The type wrappers are **Double, Float, Long, Integer, Short, Byte, Character, andBoolean**.

  Type wrappers are related directly to **auto-boxing / auto-unboxing** feature.

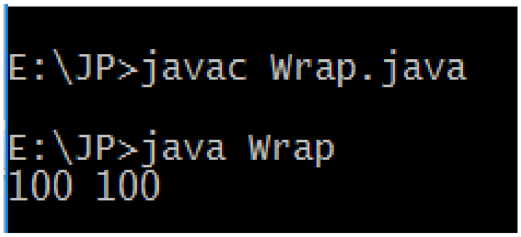
Type Wrapper	Constructor	Method name
Character	Character(char ch)	char charValue( )

Boolean	Boolean(boolean boolValue)
	Boolean(String boolString)
		boolean booleanValue( )

Numeric Type Wrappers		
Byte	Byte(int num)	byte byteValue( )
Short	Short(int num)	short shortValue( )
Long	Long(int num),Long(String str)	long longValue( )
Float	Float(float num)	float floatValue( )
Double	Double(double num)	double doubleValue( )

Demonstrate a type wrapper.

```
class Wrap {
        public static void main(String args[]) {
          // boxing
           Integer iOb = new Integer(100);
          //unboxing
           int i = iOb.intValue();
           System.out.println(i + " " +
           iOb);
        }
}
```



```
E:\JP>javac Wrap.java

E:\JP>java Wrap
100 100
```

boxing: The process of encapsulating a value within an object is called **boxing**.

Unboxing: The process of extracting a value from a type wrapper is called **unboxing**.

Autoboxing and Autounboxing

Beginning with JDK 5, Java does this **boxing** and **unboxing** automatically through auto-boxing and auto-unboxing features.

Autoboxing: Autoboxing is the process by which a primitive type is automatically encapsulated (boxed) intoits equivalent type wrapper.

Autounboxing: Auto-unboxing is the process by which the value of a boxed object is automatically extracted (unboxed) from a type wrapper.

Autoboxing/Autounboxing- benefits

- No manual boxing and unboxing values.
- It prevents errors
- It is very important to generics, which operate only on objects.
- autoboxing makes working with the Collections Framework.

Where it Works

Autoboxing in assignments

```
Integer iOb = 100; // autobox an intint

i = iOb; // auto-unbox
```

Autoboxing in Methods

```
int m(Integer v) {
        return v ; // auto-unbox to int
}
```

Autoboxing in expressions

```
        Integer iOb, iOb2;
         int i;
        iOb = 100;
++iOb;
System.out.println("After ++iOb: " + iOb);        // output: 101

iOb2 = iOb + (iOb / 3);
System.out.println("iOb2 after expression: " + iOb2); // output: 134
```

Autoboxing in switch and loops

```
Integer iOb = 2;

switch(iOb) {
        case 1: System.out.println("one");
        break;
        case 2: System.out.println("two");
        break;
        default: System.out.println("error");
}
```

Autoboxing in assignments

```
Integer iOb = 100; // autobox an intint

i = iOb; // auto-unbox
```

Autoboxing in Methods

```
int m(Integer v) {
        return v ; // auto-unbox to int
}
```

Autoboxing in expressions

```
        Integer iOb, iOb2;
         int i;
        iOb = 100;
++iOb;
System.out.println("After ++iOb: " + iOb);        // output: 101

iOb2 = iOb + (iOb / 3);
System.out.println("iOb2 after expression: " + iOb2); // output: 134
```

Autoboxing in switch and loops

```
        Integer iOb = 2;

        switch(iOb) {
                case 1: System.out.println("one");
                break;
                case 2: System.out.println("two");
                break;
                default: System.out.println("error");
        }
```

||Jai Sri Gurudev ||

BGSKH Education Trust (R.) – A unit of Sri Adichunchanagiri Shikshana Trust(R.)

BGS College of Engineering and Technology

Mahalakshmipuram, West of Chord Road, Bengaluru-560086

(Approved by AICTE, New Delhi and Affiliated to VTU, Belagavi)

OPERATING SYSTEM

NOTES

For third Semester BE [VTU/CBCS, 2023-24 Syllabus]

Subject Code:

B	**C**	**S**	**3**	**0**	**3**

OPERATING SYSTEMS		Semester	3
Course Code	BCS303	CIE Marks	50
Teaching Hours/Week (L:T:P: S)	3:0:2:0	SEE Marks	50
Total Hours of Pedagogy	40 hours Theory + 20 hours practicals	Total Marks	100
Credits	04	Exam Hours	3
Examination nature (SEE)	**Theory**		

Course objectives:

- To Demonstrate the need for OS and different types of OS
- To discuss suitable techniques for management of different resources
- To demonstrate different APIs/Commands related to processor, memory, storage and file system management.

Teaching-Learning Process (General Instructions)

Teachers can use the following strategies to accelerate the attainment of the various course outcomes.

1. Lecturer methods (L) need not to be only traditional lecture method, but alternative effective teaching methods could be adopted to attain the outcomes.
2. Use of Video/Animation to explain functioning of various concepts.
3. Encourage collaborative (Group Learning) Learning in the class.
4. Adopt Problem Based Learning (PBL), which fosters students' Analytical skills, develop design thinking skills such as the ability to design, evaluate, generalize, and analyze information rather than simply recall it.
5. Role play for process scheduling.
6. Demonstrate the installation of any one Linux OS on VMware/Virtual Box

MODULE-1 **8 Hours**

Introduction to operating systems, System structures: What operating systems do; Computer System organization; Computer System architecture; Operating System structure; Operating System operations; Process management; Memory management; Storage management; Protection and Security; Distributed system; Special-purpose systems; Computing environments.

Operating System Services: User - Operating System interface; System calls; Types of system calls; System programs; Operating system design and implementation; Operating System structure; Virtual machines; Operating System debugging, Operating System generation; System boot.

Textbook 1: Chapter – 1 (1.1-1.12), 2 (2.2-2.11)

MODULE-2 **8 Hours**

Process Management: Process concept; Process scheduling; Operations on processes; Inter process communication

Multi-threaded Programming: Overview; Multithreading models; Thread Libraries; Threading issues.

Process Scheduling: Basic concepts; Scheduling Criteria; Scheduling Algorithms; Thread scheduling; Multiple-processor scheduling,

Textbook 1: Chapter – 3 (3.1-3.4), 4 (4.1-4.4), 5 (5.1 -5.5)

MODULE-3 **8 Hours**

Process Synchronization: Synchronization: The critical section problem; Peterson’s solution; Synchronization hardware; Semaphores; Classical problems of synchronization;

Deadlocks: System model; Deadlock characterization; Methods for handling deadlocks; Deadlock prevention; Deadlock avoidance; Deadlock detection and recovery from deadlock.

Textbook 1: Chapter – 6 (6.1-6.6), 7 (7.1 -7.7)

MODULE-4 **8 Hours**

Memory Management: Memory management strategies: Background; Swapping; Contiguous memory allocation; Paging; Structure of page table; Segmentation.

Virtual Memory Management: Background; Demand paging; Copy-on-write; Page replacement; Allocation of frames; Thrashing.

Textbook 1: Chapter -8 (8.1-8.6), 9 (9.1-9.6)

MODULE-5 **8 Hours**

File System, Implementation of File System: File system: File concept; Access methods; Directory and Disk structure; File system mounting; File sharing; **Implementing File system:** File system structure; File system implementation; Directory implementation; Allocation methods; Free space management.

Secondary Storage Structure, Protection: Mass storage structures; Disk structure; Disk attachment; Disk scheduling; Disk management; **Protection**: Goals of protection, Principles of protection, Domain of protection, Access matrix.

Textbook 1: Chapter – 10 (10.1-10.5) ,11 (11.1-11.5),12 (12.1-12.5), 14 (14.1-14.4)

COURSE NAME: OPERATING SYSTEMS

COURSE CODE: BCS303

SEMESTER: 3

MODULE: 1

NUMBER OF HOURS: 8

CONTENTS:

- **Introduction to operating systems:**
    - What operating systems do?
    - Computer System organization
    - Computer System architecture
    - Operating System structure
    - Operating System operations
    - Process management
    - Memory management
    - Storage management
    - Protection and Security
    - Distributed system
    - Special-purpose systems
    - Computing environments.
- **Operating-System Structures:**
    - Operating System Services;
    - User - Operating System interface
    - System calls
    - Types of system calls
    - System programs
    - Operating system design and implementation
    - Operating System structure
    - Virtual machines
    - Operating System generation
    - System boot.

MODULE 1

INTRODUCTION TO OPERATING SYSTEM

What is an Operating System?

An operating system is system software that acts as an intermediary between a user of a computer and the computer hardware. It is software that manages the computer hardware and allows the user to execute programs in a convenient and efficient manner.

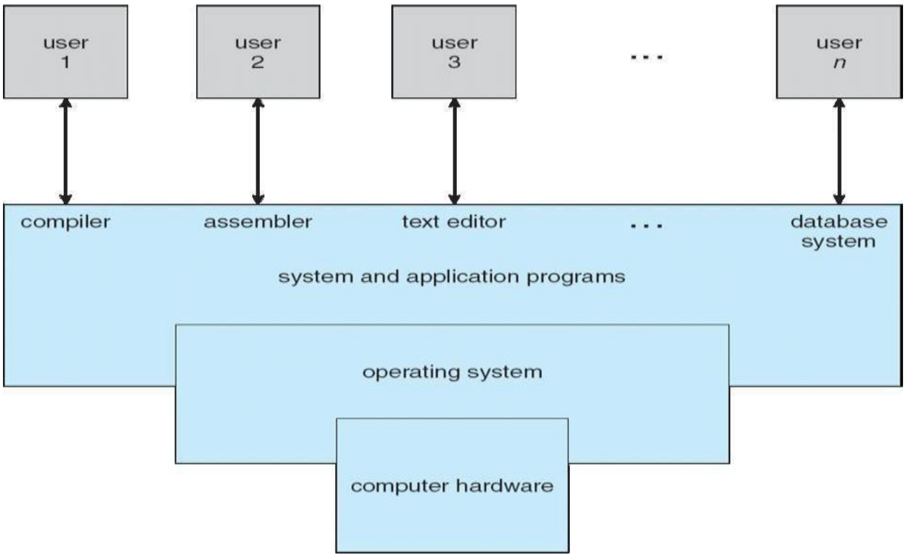
Operating system goals:

- Make the computer system convenient to use. It hides the difficulty in managing the hardware.
- Use the computer hardware in an efficient manner
- Provide and environment in which user can easily interface with computer.
- It is a resource allocator

Computer System Structure (Components of Computer System)

Computer system mainly consists of four components-

- ***Hardware*** **–** provides basic computing resources CPU, memory, I/O devices
- ***Operating system*** **-** Controls and coordinates use of hardware among various applications and users
- ***Application programs*** – define the ways in which the system resources are used to solve the computing problems of the users, Word processors, compilers, web browsers, database systems, video games
- ***Users*** **-** People, machines, other computers



List out the User Views and System views of OS

Operating System can be viewed from two viewpoints– User views & System views

User Views:- The user’s view of the operating system depends on the type of user.

- If the user is using **standalone** system, then OS is designed for ease of use and high performances. Here resource utilization is not given importance.

- If the users are at different **terminals** connected to a mainframe or minicomputers, by sharing information and resources, then the OS is designed to maximize resource utilization. OS is designed such that the CPU time, memory and i/o are used efficiently and no single user takes more than the resource allotted to them.

- If the users are in **workstations**, connected to networks and servers, then the user have a system unit of their own and shares resources and files with other systems. Here the OS is designed for both ease of use and resource availability (files).

- Other systems like embedded systems used in home devie (like washing m/c) & automobiles do not have any user interaction. There are some LEDs to show the status of its work

- Users of **hand held** systems, expects the OS to be designed for ease of use and performance per amount of battery life

System Views:- Operating system can be viewed as a **resource allocator** and **control program**.

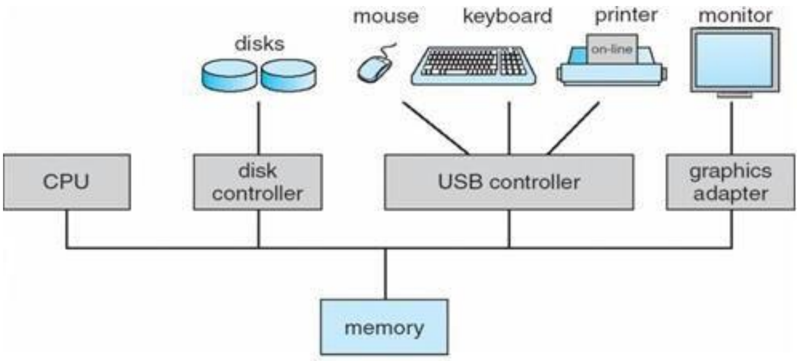
- **Resource allocator** - The OS acts as a manager of hardware and software resources. CPU time, memory space, file-storage space, I/O devices, shared files etc. are the different resources required during execution of a program. There can be conflicting request for these resources by different programs running in same system. The OS assigns the resources to the requesting program depending on the priority.

- Control Program – The OS is a control program and manage the execution of user program to prevent errors and improper use of the computer.

Computer System Organization

Computer-system operation

One or more CPUs, device controllers connect through common bus providing access to shared memory. Each device controller is in-charge of a specific type of device. To ensure orderly access to the shared memory, a memory controller is provided whose function is to synchronize access to the memory. The CPU and other devices execute concurrently competing for memory cycles. Concurrent execution of CPUs and devices competing for memory cycles



When system is switched on, '**Bootstrap'** program is executed. It is the initial program to run in the system. This program is stored in read-only memory (ROM) or in electrically erasable programmable read-only memory (EEPROM). It initializes the CPU registers, memory, device controllers and other initial setups. The program also locates and loads, the OS kernel to the memory. Then the OS starts with the first process to be executed (ie. 'init' process) and then wait for the interrupt from the user.

Switch on ⟶ 'Bootstrap' program

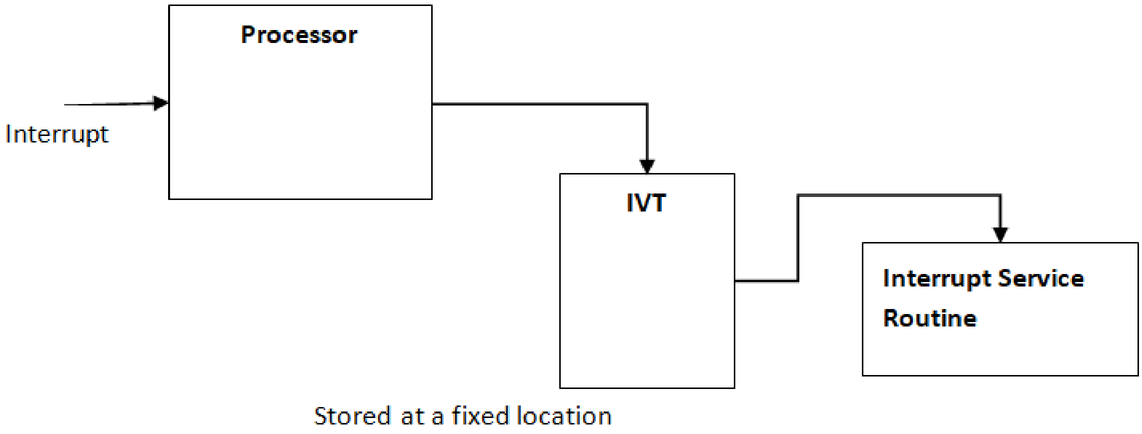
- Initializes the registers, memory and I/O devices
- Locates & loads kernel into memory
- Starts with 'init' process
- Waits for interrupt from user.

Interrupt handling –

- The occurrence of an event is usually signaled by an interrupt. The interrupt can either be from the hardware or the software. Hardware may trigger an interrupt at any time by sending a signal to the CPU. Software triggers an interrupt by executing a special operation called a system call (also called a monitor call).

- When the CPU is interrupted, it stops what it is doing and immediately transfers execution to a fixed location. The fixed location (Interrupt Vector Table) contains the starting address where the service routine for the interrupt is located. After the execution of interrupt service routine, the CPU resumes the interrupted computation.

- Interrupts are an important part of computer architecture. Each computer design has its own interrupt mechanism, but several functions are common. The interrupt must transfer control to the appropriate interrupt service routine



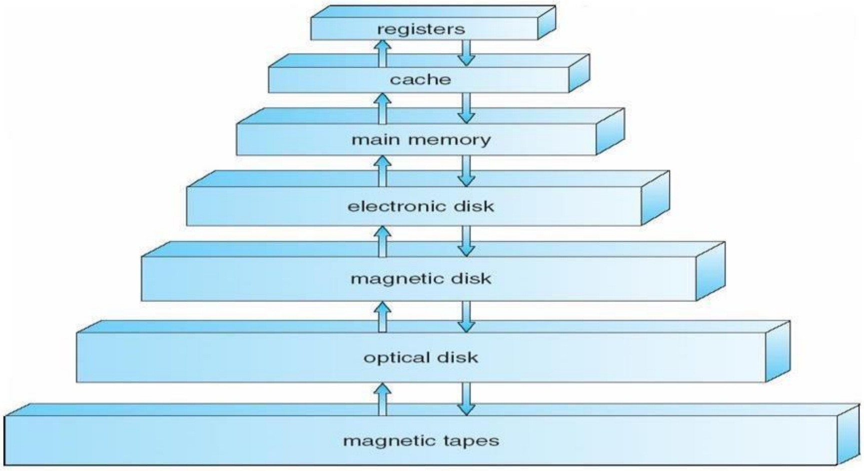
Storage Structure

- Computer programs must be in main memory (**RAM)** to be executed. Main memory is the large memory that the processor can access directly. It commonly is implemented in a semiconductor technology called **dynamic random-access memory (DRAM).** Computers provide Read Only Memory (ROM), whose data cannot be changed.
- All forms of memory provide an array of memory words. Each word has its own address. Interaction is achieved through a sequence of load or store instructions to specific memory addresses.
- A typical instruction-execution cycle, as executed on a system with a **Von Neumann** architecture, first fetches an instruction from memory and stores that instruction in the **instruction register.** The instruction is then decoded and may cause operands to be fetched from memory and stored in some internal register. After the instruction on the operands has been executed, the result may be stored back in memory.
- Ideally, we want the programs and data to reside in main memory permanently. This arrangement usually is not possible for the following two reasons:

  1. Main memory is usually too small to store all needed programs and data permanently.
  2. Main memory is a *volatile* storage device that loses its contents when power is turned off.

- Thus, most computer systems provide **secondary storage** as an extension of main memory. The main requirement for secondary storage is that it will be able to hold large quantities of data permanently.

- The most common secondary-storage device is a **magnetic disk,** which provides storage for both programs and data. Most programs are stored on a disk until they are loaded into memory. Many programs then use the disk as both a source and a destination of the information for their processing.



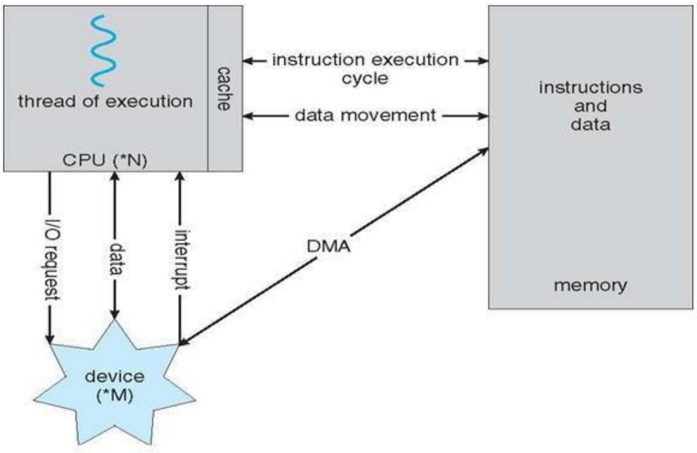
- The wide variety of storage systems in a computer system can be organized in a hierarchy as shown in the figure, according to speed, cost and capacity. The higher levels are expensive, but they are fast. As we move down the hierarchy, the cost per bit generally decreases, whereas the access time and the capacity of storage generally increases.
- In addition to differing in speed and cost, the various storage systems are either volatile or nonvolatile. V**olatile storage** loses its contents when the power to the device is removed. In the absence of expensive battery and generator backup systems, data must be written to **nonvolatile storage** for safekeeping. In the hierarchy shown in figure, the storage systems above the electronic disk are volatile, whereas those below are nonvolatile.
- An **electronic disk** can be designed to be either volatile or nonvolatile. During normal operation, the electronic disk stores data in a large DRAM array, which is volatile. But many electronic-disk devices contain a hidden magnetic hard disk and a battery for backup power. If external power is interrupted, the electronic-disk controller copies the data from RAM to the magnetic disk. Another form of electronic disk is flash memory.

Caching

Important principle, performed at many levels in a computer (in hardware, operating system, software) Information in use copied from slower to faster storage temporarily. Faster storage (cache) checked first to determine if information is there, If it is, information used directly from the cache (fast) .If not, data copied to cache and used their Cache smaller than storage being cached Cache management important design problem Cache size and replacement policy.

I/O Structure

- A large portion of operating system code is dedicated to managing I/O, both because of its importance to the reliability and performance of a system and because of the varying nature of the devices.
- Every device have a device controller, maintains some local buffer and a set of special- purpose registers. The device controller is responsible for moving the data between the peripheral devices. The operating systems have a **device driver** for each device controller.



Computer System Architecture

Categorized roughly according to the number of general-purpose processors used –

Single-Processor Systems –

- Most systems use a single processor. The variety of single-processor systems range from PDAs through mainframes. On a single-processor system, there is one main CPU capable of executing instructions from user processes. It contains special-purpose processors, in the form of device-specific processors, for devices such as disk, keyboard, and graphics controllers.
- All special-purpose processors run limited instructions and do not run user processes. These are managed by the operating system; the operating system sends them information about their next task and monitors their status.
- For example, a disk-controller processor, implements its own disk queue and scheduling algorithm, thus reducing the task of main CPU. Special processors in the keyboard, converts the keystrokes into codes to be sent to the CPU.
- The use of special-purpose microprocessors is common and does not turn a single- processor system into a multiprocessor. If there is only one general-purpose CPU, then the system is a single-processor system.

Q) Explain advantages of multiprocessing

Multiprocessor Systems (parallel systems or **tightly coupled systems) –**
Systems that have two or more processors in close communication, sharing the computer bus, the clock, memory, and peripheral devices are the multiprocessor systems.

Multiprocessor systems have three main advantages:

1. *Increased throughput*- In multiprocessor system, as there are multiple processors execution of different programs take place simultaneously. Even if the number of processors is increased the performance cannot be simultaneously increased. This is due to the overhead incurred in keeping all the parts working correctly and also due to the competition for the shared resources. The speed-up ratio with *N* processors is not *N*, rather, it is less than N. Thus the speed of the system is not has expected.
2. *Economy of scale* - Multiprocessor systems can cost less than equivalent number of many

single-processor systems. As the multiprocessor systems share peripherals, mass storage, and power supplies, the cost of implementing this system is economical. If several processes are working on the same data, the data can also be shared among them.

3. *Increased reliability*- In multiprocessor systems functions are shared among several processors. If one processor fails, the system is not halted, it only slows down. The job of the failed processor is taken up, by other processors.

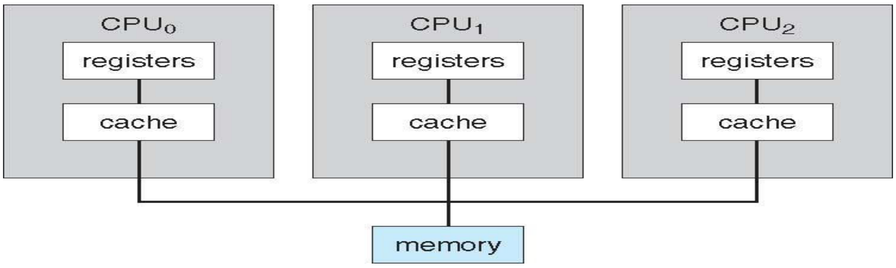
Two techniques to maintain 'Increased Reliability' - graceful degradation & fault tolerant

1. **Graceful degradation** – As there are multiple processors when one processor fails other process will take up its work and the system goes down slowly.
2. **Fault tolerant** – When one processor fails, its operations are stopped, the system failure is then detected, diagnosed, and corrected.

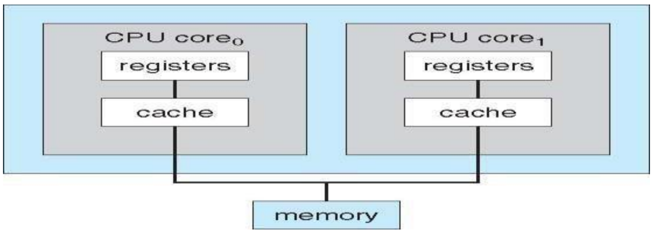
Q) Explain different types of multiprocessor systems and the types of clustering. What are fault tolerant systems?

1. Asymmetric multiprocessing
2. Symmetric multiprocessing

1) *Asymmetric multiprocessing* – **(Master/Slave architecture)** Here each processor is assigned a specific task, by the master processor. A master processor controls the other processors in the system. It schedules and allocates work to the slave processors.
2) *Symmetric multiprocessing (SMP)* – All the processors are considered as peers. There is no master-slave relationship. All the processors have its own registers and CPU, only memory is shared.



The benefit of this model is that many processes can run simultaneously. *N* processes can run if there are *N* CPUs—without causing a significant deterioration of performance. Operating systems like Windows, Windows XP, Mac OS X, and Linux—now provide support for SMP. A recent trend in CPU design is to include multiple compute **cores** on a single chip. The communication between processors within a chip is more faster than communication between two single processors.



Clustered Systems

Clustered systems are two or more individual systems connected together via network and sharing software resources. Clustering provides **high-availability** of resources and services. The service will continue even if one or more systems in the cluster fail. High availability is generally obtained by storing a copy of files (s/w resources) in the system.

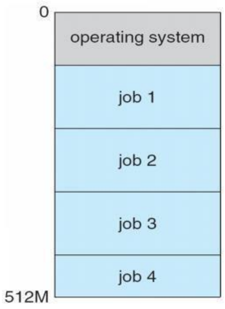
There are two types of Clustered systems – **asymmetric** and **symmetric**

1. *Asymmetric clustering* – one system is in **hot-stand by mode** while the others are running the applications. The hot-standby host machine does nothing but monitor the active server. If that server fails, the hot-standby host becomes the active server.
2. *Symmetric clustering* – two or more systems are running applications, and are monitoring each other. This mode is more efficient, as it uses all of the available hardware. If any system fails, its job is taken up by the monitoring system.

Other forms of clusters include parallel clusters and clustering over a wide-area network (WAN). Parallel clusters allow multiple hosts to access the same data on the shared storage. Cluster technology is changing rapidly with the help of **SAN(storage-area networks)**. Using SAN resources can be shared with dozens of systems in a cluster, that are separated by miles.

Explain multiprogramming and multitasking systems.

One of the most important aspects of operating systems is the ability to multiprogram. A single user cannot keep either the CPU or the I/O devices busy at all times. **Multiprogramming** increases CPU utilization by organizing jobs, so that the CPU always has one to execute.

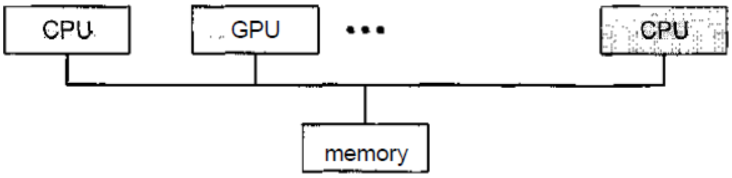


- The operating system keeps several jobs in memory simultaneously as shown in figure. This set of jobs is a subset of the jobs kept in the job pool. Since the number of jobs that can be kept simultaneously in memory is usually smaller than the number of jobs that can be kept in the job pool(in secondary memory). The operating system picks and begins to execute one of the jobs in memory. Eventually, the job may have to wait for some task, such as an I/O operation, to complete. In a non-multiprogrammed system, the CPU would sit idle.
- In a multiprogrammed system, the operating system simply switches to, and executes, another job. When *that* job needs to wait, the CPU is switched to *another* job, and so on.
- Eventually, the first job finishes waiting and gets the CPU back. Thus the CPU is never idle.
- Multiprogrammed systems provide an environment in which the various system resources (for example, CPU, memory, and peripheral devices) are utilized effectively, but they do not provide for user interaction with the computer system.

- In **Time sharing** (or **multitasking) systems,** a single CPU executes multiple jobs by switching among them, but the switches occur so frequently that the users can interact with each program while it is running. The user feels that all the programs are being executed at the same time. Time sharing requires an **interactive** (or **hands-on) computer system,** which provides direct communication between the user and the system. The user gives instructions to the operating system or to a program directly, using a input device such as a keyboard or a mouse, and waits for immediate results on an output device. Accordingly, the **response time** should be short—typically less than one second.

- A time-shared operating system allows many users to share the computer simultaneously. As the system switches rapidly from one user to the next, each user is given the impression that the entire computer system is dedicated to his use only, even though it is being shared among many users.

- A **multiprocessor system** is a computer system having two or more CPUs within a single computer system, each sharing main memory and peripherals. Multiple programs are executed by multiple processors parallel.



Distributed Systems

- Individual systems that are connected and share the resource available in network is called Distributed system. Access to a shared resource increases computation speed, functionality, data availability, and reliability.

- A **network** is a communication path between two or more systems. Distributed systems depend on networking for their functionality. Networks vary by the protocols used, the distances between nodes, and the transport media. TCP/IP is the most common network protocol. Most operating systems support TCP/IP.

- Networks are characterized based on the distances between their nodes. A **local-area network (LAN)** connects computers within a room, a floor, or a building. A **wide-area network (WAN)** usually links buildings, cities, or countries. A global company may have a WAN to connect its offices worldwide. A **metropolitan-area network** (MAN) links buildings within a city. A **small-area network** connects systems within a several feet using wireless technology. Eg. Bluetooth and 802.11.

- A **network operating system** is an operating system that provides features such as file sharing across the network and that allows different processes on different computers to exchange messages. A computer running a network operating system acts autonomously from all other computers on the network, although it is aware of the network and is able to communicate with other networked computers.

Explain dual mode operation in operating system with a neat block diagram

Operating-System Operations

Modern operating systems are **interrupt driven.** If there are no processes to execute, no I/O devices to service, and no users to whom to respond, an operating system will sit quietly, waiting for something to happen. Events are signaled by the occurrence of an interrupt or a trap. A **trap (or** an **exception)** is a software-generated interrupt. For each type of interrupt, separate segments of code in the operating system determine what action should be taken. An interrupt service routine is provided that is responsible for dealing with the interrupt.

Dual-Mode Operation

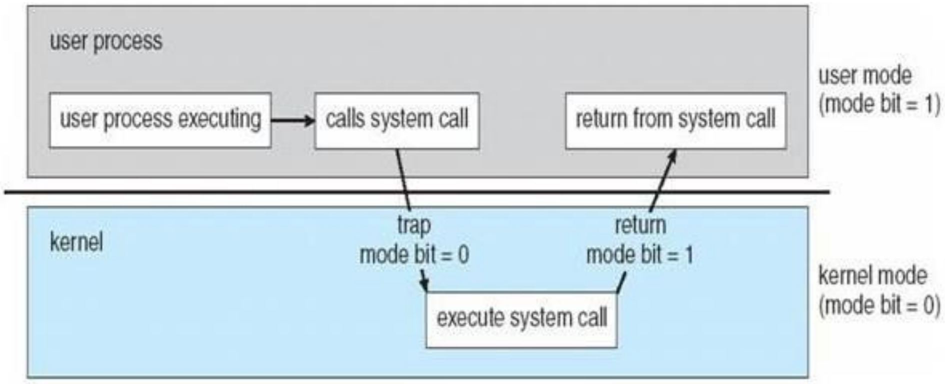
Since the operating system and the user programs share the hardware and software resources of the computer system, it has to be made sure that an error in a user program cannot cause problems to other programs and the Operating System running in the system.

The approach taken is to use a hardware support that allows us to differentiate among various modes of execution.

The system can be assumed to work in two separate **modes** of operation:

1. User mode
2. Kernel mode (supervisor mode, system mode, or privileged mode).

- A hardware bit of the computer, called the **mode bit**, is used to indicate the current mode: kernel (0) or user (1). With the mode bit, we are able to distinguish between a task that is executed by the operating system and one that is executed by the user.
- When the computer system is executing a user application, the system is in user mode. When a user application requests a service from the operating system (via a system call), the transition from user to kernel mode takes place.



At system boot time, the hardware starts in kernel mode. The operating system is then loaded and starts user applications in user mode. Whenever a trap or interrupt occurs, the hardware switches from user mode to kernel mode (that is, changes the mode bit from 1 to 0). Thus, whenever the operating system gains control of the computer, it is in kernel mode.

The dual mode of operation provides us with the means for protecting the operating system from errant users—and errant users from one another.

- The hardware allows privileged instructions to be executed only in kernel mode. If an attempt is made to execute a privileged instruction in user mode, the hardware does not execute the instruction but rather treats it as illegal and traps it to the operating system. The instruction to switch to user mode is an example of a privileged instruction.

- Initial control is within the operating system, where instructions are executed in kernel mode. When control is given to a user application, the mode is set to user mode. Eventually, control is switched back to the operating system via an interrupt, a trap, or a system call.

Process Management

A program under execution is a process. A process needs resources like CPU time, memory, files, and I/O devices for its execution. These resources are given to the process when it is created or at run time. When the process terminates, the operating system reclaims the resources.

The program stored on a disk is a **passive entity** and the program under execution is an **active entity**. A single-threaded process has one **program counter** specifying the next instruction to execute. The CPU executes one instruction of the process after another, until the process completes. A multithreaded process has multiple program counters, each pointing to the next instruction to execute for a given thread.

The operating system is responsible for the following activities in connection with process management:

- Scheduling process and threads on the CPU
- Creating and deleting both user and system processes
- Suspending and resuming processes
- Providing mechanisms for process synchronization
- Providing mechanisms for process communication

Memory Management

Main memory is a large array of words or bytes. Each word or byte has its own address. Main memory is the storage device which can be easily and directly accessed by the CPU. As the program executes, the central processor reads instructions and also reads and writes data from main memory.

To improve both the utilization of the CPU and the speed of the computer's response to its users, general-purpose computers must keep several programs in memory, creating a need for memory management.

The operating system is responsible for the following activities in connection with memory management:

- Keeping track of which parts of memory are currently being used by user.
- Deciding which processes and data to move into and out of memory.
- Allocating and deallocating memory space as needed.

Storage Management

There are three types of storage management

i) File system management
ii) Mass-storage management
iii) Cache management.

File-System Management

- File management is one of the most visible components of an operating system. Computers can store information on several different types of physical media. Magnetic disk, optical disk, and magnetic tape are the most common. Each of these media has its own characteristics and physical organization. Each medium is controlled by a device, such as a disk drive or tape drive, that also has its own unique characteristics.
- A file is a collection of related information defined by its creator. Commonly, files represent programs and data. Data files may be numeric, alphabetic, alphanumeric, or binary. Files may be free-form (for example, text files), or they may be formatted rigidly (for example, fixed fields).
- The operating system implements the abstract concept of a file by managing mass storage media. Files are normally organized into directories to make them easier to use. When multiple users have access to files, it may be desirable to control by whom and in what ways (read, write, execute) files may be accessed.

The operating system is responsible for the following activities in connection with file management:

- Creating and deleting files
- Creating and deleting directories to organize files
- Supporting primitives for manipulating files and directories
- Mapping files onto secondary storage
- Backing up files on stable (nonvolatile) storage media

Mass-Storage Management

- As the main memory is too small to accommodate all data and programs, and as the data that it holds are erased when power is lost, the computer system must provide secondary storage to back up main memory. Most modern computer systems use disks as the storage medium for both programs and data.

- Most programs—including compilers, assemblers, word processors, editors, and formatters—are stored on a disk until loaded into memory and then use the disk as both the source and destination of their processing. Hence, the proper management of disk storage is of central importance to a computer system.

The operating system is responsible for the following activities in connection with disk management:

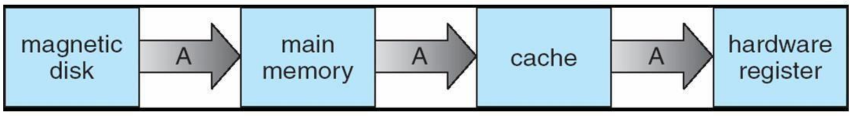
- Free-space management

- Storage allocation
- Disk scheduling

As the secondary storage is used frequently, it must be used efficiently. The entire speed of operation of a computer may depend on the speeds of the disk. Magnetic tape drives and their tapes, CD, DVD drives and platters are **tertiary storage** devices. The functions that operating systems provides include mounting and unmounting media in devices, allocating and freeing the devices for exclusive use by processes, and migrating data from secondary to tertiary storage.

Caching

- **Caching** is an important principle of computer systems. Information is normally kept in some storage system (such as main memory). As it is used, it is copied into a faster storage system—the cache—as temporary data. When a particular piece of information is required, first we check whether it is in the cache. If it is, we use the information directly from the cache; if it is not in cache, we use the information from the source, putting a copy in the cache under the assumption that we will need it again soon.
- Because caches have limited size, **cache management** is an important design problem. Careful selection of the cache size and page replacement policy can result in greatly increased performance.
- The movement of information between levels of a storage hierarchy may be either explicit or implicit, depending on the hardware design and the controlling operating-system software. For instance, data transfer from cache to CPU and registers is usually a hardware function, with no operating-system intervention. In contrast, transfer of data from disk to memory is usually controlled by the operating system.
- In a hierarchical storage structure, the same data may appear in different levels of the storage system. For example, suppose to retrieve an integer A from magnetic disk to the processing program. The operation proceeds by first issuing an I/O operation to copy the disk block on which A resides to main memory. This operation is followed by copying A to the cache and to an internal register. Thus, the copy of A appears in several places: on the magnetic disk, in main memory, in the cache, and in an internal register.



- In a multiprocessor environment, in addition to maintaining internal registers, each of the CPUs also contains a local cache. In such an environment, a copy of A may exist simultaneously in several caches. Since the various CPUs can all execute concurrently, any update done to the value of A in one cache is immediately reflected in all other caches where A resides. This situation is called **cache coherency,** and it is usually a hardware problem (handled below the operating-system level).

I/O Systems

One of the purposes of an operating system is to hide the peculiarities of specific hardware devices from the user. The I/O subsystem consists of several components:

- A memory-management component that includes buffering, caching, and spooling
- A general device-driver interface
- Drivers for specific hardware devices

Only the device driver knows the peculiarities of the specific device to which it is assigned.

Protection and Security

- If a computer system has multiple users and allows the concurrent execution of multiple processes, then access to data must be regulated. For that purpose, mechanisms ensure that files, memory segments, CPU, and other resources can be operated on by only those processes that have gained proper authorization from the operating system.
- If a computer system has multiple users and allows the concurrent execution of multiple processes, then access to data must be regulated. For that purpose, there are mechanisms which ensure that files, memory segments, CPU, and other resources can be operated on by only those processes that have gained proper authorization from the operating system.
- For example, memory-addressing hardware ensures that a process can execute only within its own address space. The timer ensures that no process can gain control of the CPU for a long time. Device-control registers are not accessible to users, so the integrity of the various peripheral devices is protected.

- **Protection** is a mechanism for controlling the access of processes or users to the resources defined by a computer system. This mechanism must provide means for specification of the controls to be imposed and means for enforcement.
- Protection improves reliability. A protection-oriented system provides a means to distinguish between authorized and unauthorized usage. A system can have adequate protection but still be prone to failure and allow inappropriate access.
- Consider a user whose authentication information is stolen. Her data could be copied or deleted, even though file and memory protection are working. It is the job of **security** to defend a system from external and internal attacks. Such attacks spread across a huge range and include viruses and worms, denial-of service attacks etc.
- Protection and security require the system to be able to distinguish among all its users. Most operating systems maintain a list of user names and associated **user identifiers (user IDs).** When a user logs in to the system, the authentication stage determines the appropriate user ID for the user.

Distributed Systems

- A distributed system is a collection of systems that are networked to provide the users with access to the various resources in the network. Access to a shared resource increases computation speed, functionality, data availability, and reliability.
- A **network** is a communication path between two or more systems. Networks vary by the protocols used(TCP/IP,UDP,FTP etc.), the distances between nodes, and the transport media(copper wires, fiber-optic,wireless).
- TCP/IP is the most common network protocol. The operating systems support of protocols also varies. Most operating systems support TCP/IP, including the Windows and UNIX operating systems.
- Networks are characterized based on the distances between their nodes. A **local-area network (LAN)** connects computers within a room, a floor, or a building. A **wide-area network (WAN)** usually links buildings, cities, or countries. A global company may have a WAN to connect its offices worldwide. These networks may run one protocol or several protocols. A **metropolitan-area network** (MAN) connects buildings within a city. BlueTooth and 802.11 devices use wireless technology to communicate over a distance of several feet, in essence creating a **small-area network** such as might be found in a home.
- The transportation media to carry networks are also varied. They include copper wires, fiber strands, and wireless transmissions between satellites, microwave dishes, and radios. When computing devices are connected to cellular phones, they create a network.

Multimedia Systems

- Multimedia data consist of audio and video files as well as conventional files. These data differ from conventional data in that multimedia data—such as frames of video—must be delivered (streamed) according to certain time restrictions (for example, 30 frames per second).
- Multimedia describes a wide range of applications like audio files - MP3, DVD movies, video conferencing, and short video clips of movie previews or news. Multimedia applications may also include live webcasts of speeches or sporting events and even live webcams. Multimedia applications can be either audio or video or combination of both. For example, a movie may consist of separate audio and video tracks.

Handheld Systems

- Handheld systems include personal digital assistants (PDAs), such as Palm and Pocket-PCs, and cellular telephones. Developers of these systems face many challenges, due to the limited memory, slow processors and small screens in such devices.
- The amount of physical memory in a handheld depends upon the device, the operating system and applications must manage memory efficiently. This includes returning all allocated memory back to the memory manager when the memory is not being used. A second issue of concern to developers of handheld devices is the speed of the processor used in the devices. Processors for most handheld devices run at faster speed than the processor in a PC. Faster processors require more power and so, a larger battery is required. Another issue is the usage of I/O devices.

- Generally, the limitations in the functionality of PDAs are balanced by their convenience and portability. Their use continues to expand as network connections become more available and other options, such as digital cameras and MP3 players, expand their utility.

Computing Environments

The different computing environments are -

Traditional Computing

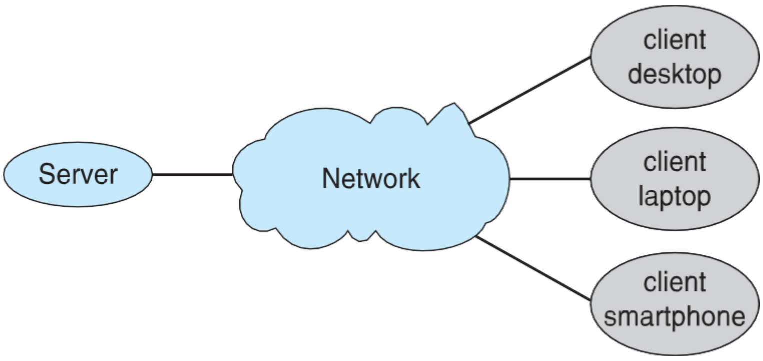
- The current trend is toward providing more ways to access these computing environments. Web technologies are stretching the boundaries of traditional computing. Companies establish **portals,** which provide web accessibility to their internal servers. **Network computers** are essentially terminals that understand web-based computing. Handheld computers can synchronize with PCs to allow very portable use of company information. Handheld PDAs can also connect to **wireless networks** to use the company's web portal. The fast data connections are allowing home computers to serve up web pages and to use networks. Some homes even have **firewalls** to protect their networks.

- In the latter half of the previous century, computing resources were scarce. Years before, systems were either batch or interactive. Batch system processed jobs in bulk, with predetermined input (from files or other sources of data). Interactive systems waited for input from users. To optimize the use of the computing resources, multiple users shared time on these systems. Time-sharing systems used a timer and scheduling algorithms to rapidly cycle processes through the CPU, giving each user a share of the resources.

- Today, traditional time-sharing systems are used everywhere. The same scheduling technique is still in use on workstations and servers, but frequently the processes are all owned by the same user (or a single user and the operating system). User processes, and system processes that provide services to the user, are managed so that each frequently gets a slice of computer time.

Client-Server Computing

Designers shifted away from centralized system architecture to - terminals connected to centralized systems. As a result, many of today's systems act as **server systems to** satisfy requests generated by **client systems.** This form of specialized distributed system, called **client- server** system.



General Structure of Client – Server System

Server systems can be broadly categorized as compute servers and file servers:

- The **compute-server system** provides an interface to which a client can send a request to perform an action (for example, read data); in response, the server executes the action and sends back results to the client. A server running a database that responds to client requests for data is an example of such a svstem.
- The **file-server system** provides a file-system interface where clients can create, update, read, and delete files. An example of such a system is a web server that delivers files to clients running the web browsers.

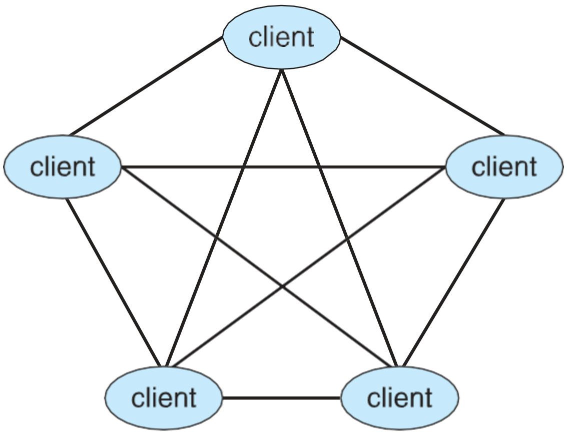
Peer-to-Peer Computing

- In this model, clients and servers are not distinguished from one another; here, all nodes within the system are considered peers, and each may act as either a client or a server, depending on whether it is requesting or providing a service.
- In a client-server system, the server is a bottleneck, because all the services must be served by the server. But in a peer-to-peer system, services can be provided by several nodes distributed throughout the network.
- To participate in a peer-to-peer system, a node must first join the network of peers. Once a node has joined the network, it can begin providing services to—and requesting services from—other nodes in the network.

Determining what services are available is accomplished in one of two general ways:

- When a node joins a network, it registers its service with a centralized lookup service on the network. Any node desiring a specific service first contacts this centralized lookup service to determine which node provides the service. The remainder of the communication takes place between the client and the service provider.

- A peer acting as a client must know, which node provides a desired service by broadcasting a request for the service to all other nodes in the network. The node (or nodes) providing that service responds to the peer making the request. To support this approach, a *discovery protocol* must be provided that allows peers to discover services provided by other peers in the network.



Web-Based Computing

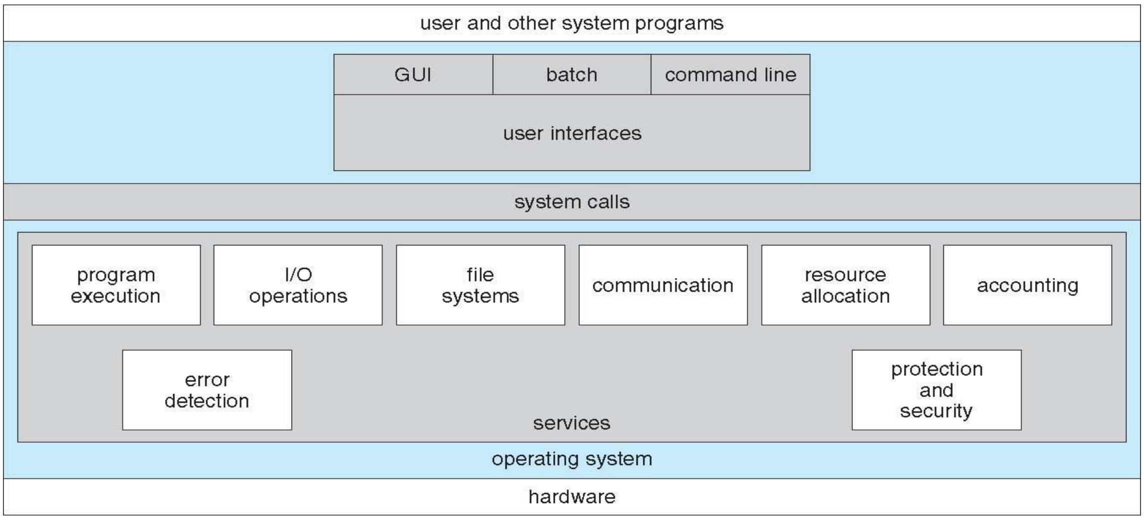
- Web computing has increased the importance on networking. Devices that were not previously networked now include wired or wireless access. Devices that were networked now have faster network connectivity.
- The implementation of web-based computing has given rise to new categories of devices, such as **load balancers,** which distribute network connections among a pool of similar servers. Operating systems like Windows 95, which acted as web clients, have evolved into Linux and Windows XP, which can act as web servers as well as clients. Generally, the Web has increased the complexity of devices, because their users require them to be web-enabled.
- The design of an operating system is a major task. It is important that the goals of the new system be well defined before the design of OS begins. These goals form the basis for choices among various algorithms and strategies.

OPERATING SYSTEM SERVICES

Operating-System Services

Q) List and explain the services provided by OS for the user and efficient operation of system.

An operating system provides an environment for the execution of programs. It provides certain services to programs and to the users of those programs.



OS provide services for the users of the system, including:

- **User Interfaces** - Means by which users can issue commands to the system. Depending on the operating system these may be a **command-line interface** (e.g. sh, csh, ksh, tcsh, etc.), a **Graphical User Interface** (e.g. Windows, X-Windows, KDE, Gnome, etc.), or a **batch command systems**.
  In Command Line Interface (CLI)- commands are given to the system.
  In Batch interface – commands and directives to control these commands are put in a file and then the file is executed.
  In GUI systems- windows with pointing device to get inputs and keyboard to enter the text.
- **Program Execution** - The OS must be able to load a program into RAM, run the program, and terminate the program, either normally or abnormally.
- **I/O Operations** - The OS is responsible for transferring data to and from I/O devices, including keyboards, terminals, printers, and files. For specific devices, special functions are provided (device drivers) by OS.

- **File-System Manipulation** – Programs need to read and write files or directories. The services required to create or delete files, search for a file, list the contents of a file and change the file permissions are provided by OS.
- **Communications** - Inter-process communications, IPC, either between processes running on the same processor, or between processes running on separate processors or separate machines. May be implemented by using the service of OS- like shared memory or message passing.
- **Error Detection** - Both hardware and software errors must be detected and handled appropriately by the OS. Errors may occur in the CPU and memory hardware (such as power failure and memory error), in I/O devices (such as a parity error on tape, a connection failure on a network, or lack of paper in the printer), and in the user program (such as an arithmetic overflow, an attempt to access an illegal memory location).

OS provide services for the efficient operation of the system, including:

- **Resource Allocation** – Resources like CPU cycles, main memory, storage space, and I/O devices must be allocated to multiple users and multiple jobs at the same time.
- **Accounting** – There are services in OS to keep track of system activity and resource usage, either for billing purposes or for statistical record keeping that can be used to optimize future performance.
- **Protection and Security** – The owners of information (file) in multiuser or networked computer system may want to control the use of that information. When several separate processes execute concurrently, one process should not interfere with other or with OS. Protection involves ensuring that all access to system resources is controlled. Security of the system from outsiders must also be done, by means of a password.

User Operating-System Interface

There are several ways for users to interface with the operating system.

i) Command-line interface, or command interpreter, allows users to directly enter commands to be performed by the operating system.

ii) Graphical user interface (GUI), allows users to interface with the operating system using pointer device and menu system.

Command Interpreter

- Command Interpreters are used to give commands to the OS. There are multiple command interpreters known as shells. In UNIX and Linux systems, there are several different shells, like the *Bourne shell, C shell, Bourne-Again shell, Korn shell,* and others.
- The main function of the command interpreter is to get and execute the user-specified command. Many of the commands manipulate files: create, delete, list, print, copy, execute, and so on.

The commands can be implemented in two general ways-

i) The command interpreter itself contains the code to execute the command. For example, a command to delete a file may cause the command interpreter to jump to a particular section of its code that sets up the parameters and makes the appropriate system call.

ii) The code to implement the command is in a function in a separate file. The interpreter searches for the file and loads it into the memory and executes it by passing the parameter. Thus by adding new functions new commands can be added easily to the interpreter without disturbing it.

Graphical User Interfaces

- A second strategy for interfacing with the operating system is through a userfriendly graphical user interface or GUI. Rather than having users directly enter commands via a command-line interface, a GUI allows provides a mouse-based window-and-menu system as an interface.
- A GUI provides a **desktop** metaphor where the mouse is moved to position its pointer on images, or **icons,** on the screen (the desktop) that represent programs, files, directories, and system functions.
- Depending on the mouse pointer's location, clicking a button on the mouse can invoke a program, select a file or directory—known as a **folder**— or pull down a menu that contains commands.

System Calls

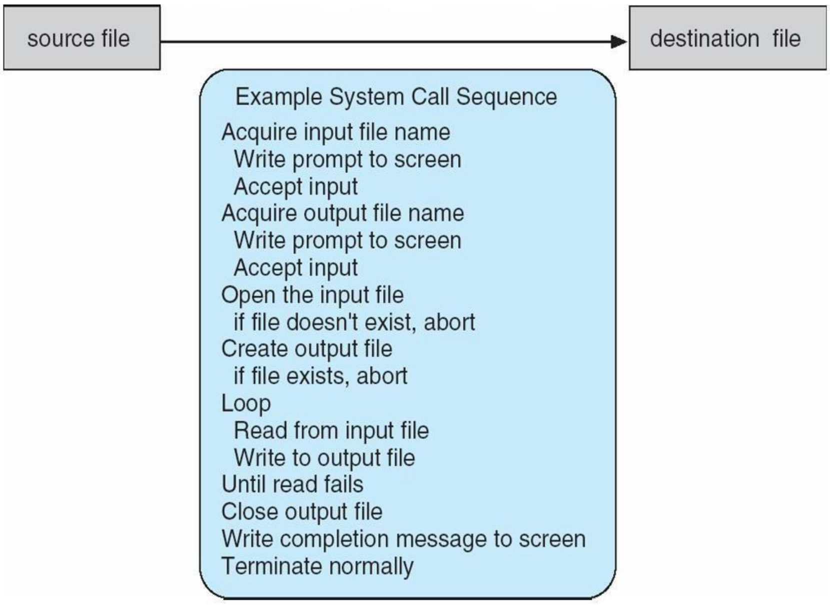
Q) What are system calls? Briefly point out its types.

- System calls is a means to access the services of the operating system generally written in C or C++, although some are written in assembly for optimal performance.
- The below figure illustrates the sequence of system calls required to copy a file content from one file (input file) to another file (output file).

An example to illustrate how system calls are used: writing a simple program to read data from one file and copy them to another file

- There are number of system calls used to finish this task. The first system call is to write a message on the screen (monitor). Then to accept the input filename. Then another system call to write message on the screen, then to accept the output filename.
- When the program tries to open the input file, it may find that there is no file of that name or that the file is protected against access. In these cases, the program should print a message on the console (another system call) and then terminate abnormally (another system call) and create a new one (another system call).

- Now that both the files are opened, we enter a loop that reads from the input file (another system call) and writes to output file (another system call).
- Finally, after the entire file is copied, the program may close both files (another system call), write a message to the console or window (system call), and finally terminate normally (final system call).



```
Example System Call Sequence
Acquire input file name
  Write prompt to screen
  Accept input
Acquire output file name
  Write prompt to screen
  Accept input
Open the input file
  if file doesn't exist, abort
Create output file
  if file exists, abort
Loop
  Read from input file
  Write to output file
Until read fails
Close output file
Write completion message to screen
Terminate normally
```

- Most programmers do not use the low-level system calls directly, but instead use an "Application Programming Interface", API.
- Instead of direct system calls provides for greater program portability between different systems. The API then makes the appropriate system calls through the system call interface, using a system call table to access specific numbered system calls.
- Each system call has a specific numbered system call. The system call table (consisting of system call number and address of the particular service) invokes a particular service routine for a specific system call.
- The caller need know nothing about how the system call is implemented or what it does during execution.

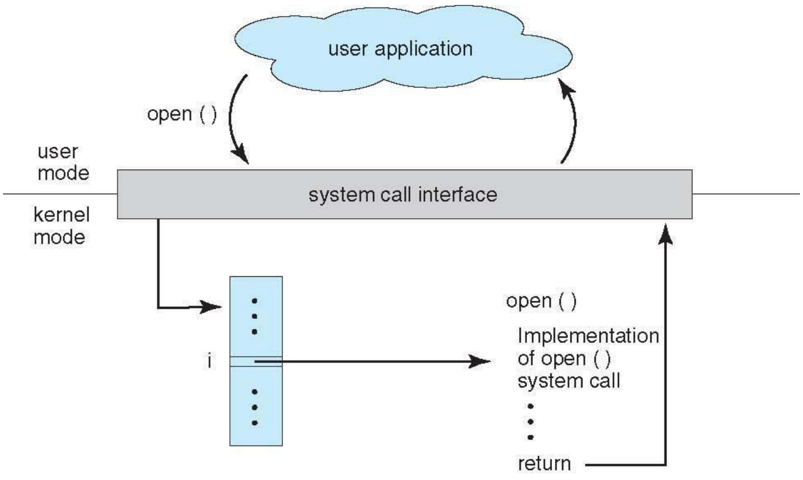


Figure: The handling of a user application invoking the open() system call.

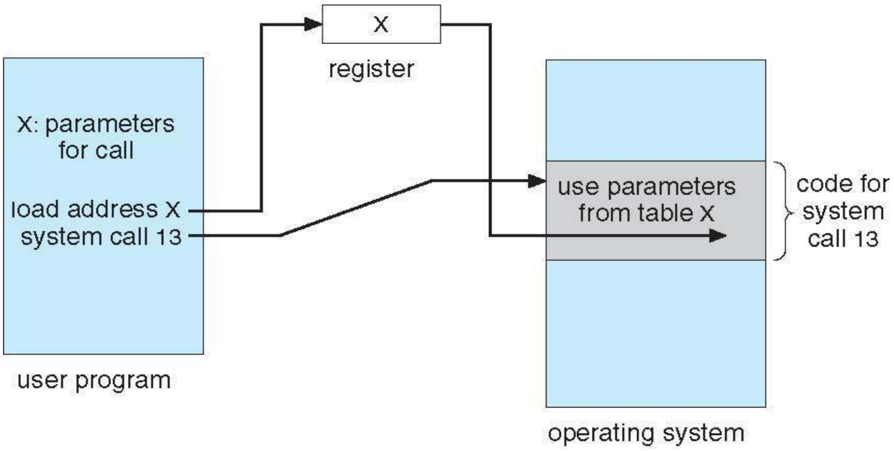


Figure: Passing of parameters as a table.

Three general methods used to pass parameters to OS are –

i) To pass parameters in registers

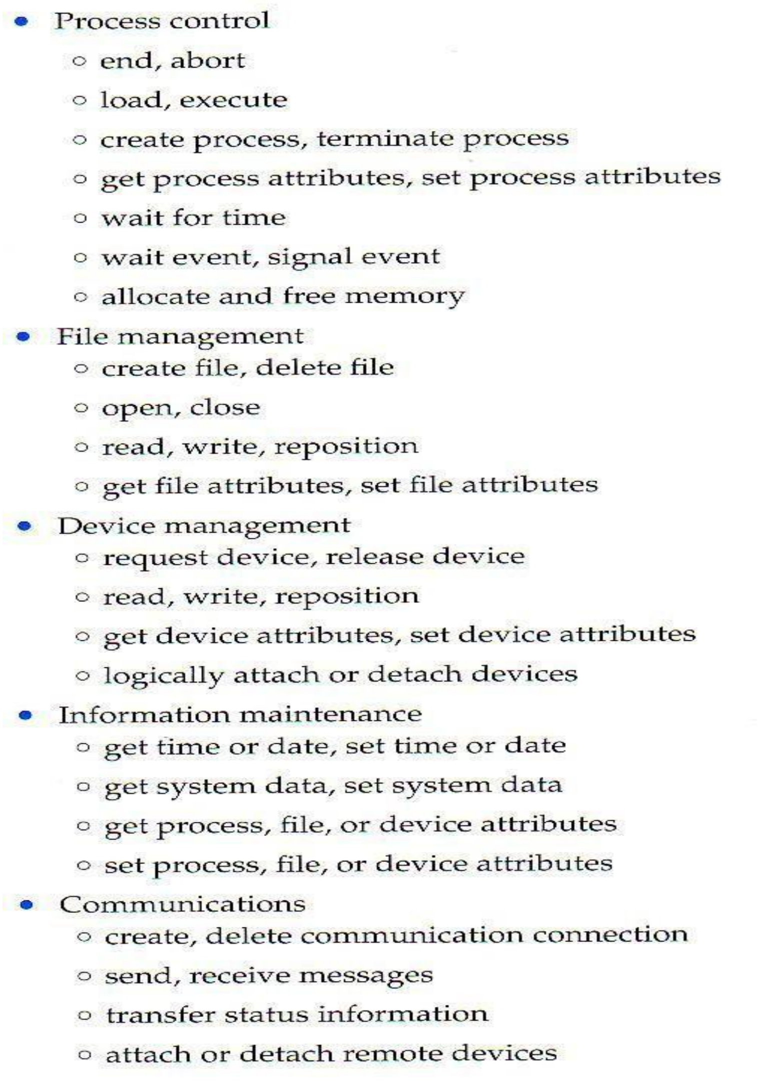
ii) If parameters are large blocks, address of block (where parameters are stored in memory) is sent to OS in the register. (Linux & Solaris).

iii) Parameters can be pushed onto the stack by program and popped off the stack by OS.

Types of System Calls

The system calls can be categorized into six major categories:

1. Process Control
2. File management
3. Device management
4. Information management
5. Communications
6. Protection



- Process control
  - end, abort
  - load, execute
  - create process, terminate process
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory
- File management
  - create file, delete file
  - open, close
  - read, write, reposition
  - get file attributes, set file attributes
- Device management
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices
- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get process, file, or device attributes
  - set process, file, or device attributes
- Communications
  - create, delete communication connection
  - send, receive messages
  - transfer status information
  - attach or detach remote devices

Figure: Types of system calls

1. Process Control

- Process control system calls include end, abort, load, execute, create process, terminate process, get/set process attributes, wait for time or event, signal event, and allocate and free memory.
- Processes must be created, launched, monitored, paused, resumed, and eventually stopped.
- When one process pauses or stops, then another must be launched or resumed
- Process attributes like process priority, max. allowable execution time etc. are set and retrieved by OS.
- After creating the new process, the parent process may have to wait (wait time), or wait for an event to occur(wait event). The process sends back a signal when the event has occurred (signal event)

In DOS, the command interpreter loaded first. Then loads the process and transfers control to it. The interpreter does not resume until the process has completed, as shown in Figure

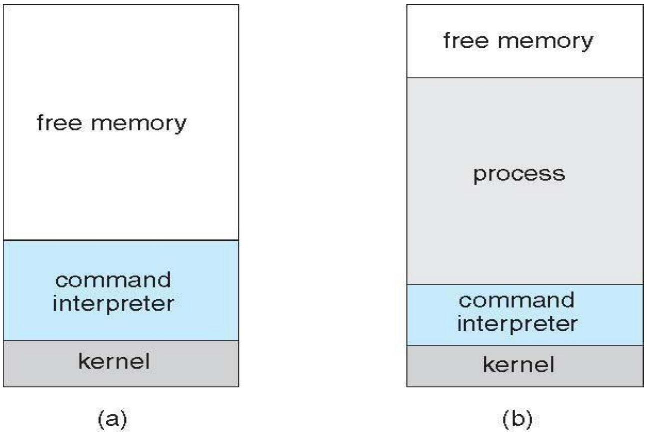


Figure: MS-DOS execution, (a) At system startup, (b) Running a program

Because UNIX is a multi-tasking system, the command interpreter remains completely resident when executing a process, as shown in Figure below.

- The user can switch back to the command interpreter at any time, and can place the running process in the background even if it was not originally launched as a background process.
- In order to do this, the command interpreter first executes a "fork" system call, which creates a second process which is an exact duplicate (clone) of the original command interpreter. The original process is known as the parent, and the cloned process is known as the child, with its own unique process ID and parent ID.
- The child process then executes an "exec" system call, which replaces its code with that of the desired process.
- The parent (command interpreter) normally waits for the child to complete before issuing a new command prompt, but in some cases it can also issue a new prompt right away, without waiting for the child process to complete. (The child is then said to be running "in the background", or "as a background process".).

2. File Management

The file management functions of OS are –

- File management system calls include create file, delete file, open, close, read, write, reposition, get file attributes, and set file attributes.
- After **creating** a file, the file is **opened**. Data is **read** or **written** to a file.
- The file pointer may need to be **repositioned** to a point.
- The file **attributes** like filename, file type, permissions, etc. are set and retrieved using system calls.
- These operations may also be supported for directories as well as ordinary files.

3. Device Management

- Device management system calls include **request device**, **release** device, **read**, **write**, **reposition**, **get/set** device attributes, and logically **attach** or **detach** devices.
- When a process needs a resource, a request for resource is done. Then the control is granted to the process. If requested resource is already attached to some other process, the requesting process has to wait.
- In multiprogramming systems, after a process uses the device, it has to be returned to OS, so that another process can use the device.
- Devices may be physical (e.g. disk drives), or virtual / abstract (e.g. files, partitions, and RAM disks).

4. Information Maintenance

- Information maintenance system calls include calls to get/set the time, date, system data, and process, file, or device attributes.
- These system calls care used to transfer the information between user and the OS. Information like current time & date, no. of current users, version no. of OS, amount of free memory, disk space etc. are passed from OS to the user.

5. Communication

- Communication system calls create/delete communication connection, send/receive messages, transfer status information, and attach/detach remote devices.
- The **message passing** model must support calls to:
  - Identify a remote process and/or host with which to communicate.
  - Establish a connection between the two processes.
  - Open and close the connection as needed.
  - Transmit messages along the connection.
  - Wait for incoming messages, in either a blocking or non-blocking state.
  - Delete the connection when no longer needed.
- The **shared memory** model must support calls to:
  - Create and access memory that is shared amongst processes (and threads.)
  - Free up shared memory and/or dynamically allocate it as needed.

- Message passing is simpler and easier, (particularly for inter-computer communications), and is generally appropriate for small amounts of data. It is easy to implement, but there are system calls for each read and write process.
- Shared memory is faster, and is generally the better approach where large amounts of data are to be shared. This model is difficult to implement, and it consists of only few system calls.

6. Protection

- Protection provides mechanisms for controlling which users / processes have access to which system resources.
- System calls allow the access mechanisms to be adjusted as needed, and for non- privileged users to be granted elevated access permissions under carefully controlled temporary circumstances.

System Programs

Q) List and explain the different categories of system program?

A collection of programs that provide a convenient environment for program development and execution (other than OS) are called system programs or system utilities.

System programs may be divided into five categories:

1. **File management** - programs to create, delete, copy, rename, print, list, and generally manipulate files and directories.
2. **Status information** - Utilities to check on the date, time, number of users, processes running, data logging, etc. System registries are used to store and recall configuration information for particular applications.
3. **File modification** - e.g. text editors and other tools which can change file contents.
4. **Programming-language support** - E.g. Compilers, linkers, debuggers, profilers, assemblers, library archive management, interpreters for common languages, and support for make.
5. **Program loading and execution** - loaders, dynamic loaders, overlay loaders, etc., as well as interactive debuggers.
6. **Communications** - Programs for providing connectivity between processes and users, including mail, web browsers, remote logins, file transfers, and remote command execution.

Operating-System Design and Implementation

Design Goals

- The first problem in designing a system is to define goals and specifications. At the highest level, the design of the system will be affected by the choice of hardware and the type of system: batch, time shared, single user, multiuser, distributed, real time, or general purpose.
- Beyond this highest design level, the requirements may be much harder to specify. The requirements can, however, be divided into two basic groups
    1. User goals (User requirements)
    2. System goals (system requirements)

- **User requirements** are features that users care about and understand like system should be convenient to use, easy to learn, reliable, safe and fast.
- **System requirements** are written for the developers, ie. People who design the OS. Their requirements are like easy to design, implement and maintain, flexible, reliable, error free and efficient.

Mechanisms and Policies

- Policies determine *what* is to be done. Mechanisms determine *how* it is to be implemented.
- Example: in timer- counter and decrementing counter is the mechanism and deciding how long the time has to be set is the policies.
- Policies change overtime. In the worst case, each change in policy would require a change in the underlying mechanism.
- If properly separated and implemented, policy changes can be easily adjusted without re- writing the code, just by adjusting parameters or possibly loading new data / configuration files.

Implementation

- Traditionally OS were written in assembly language.
- In recent years, OS are written in C, or C++. Critical sections of code are still written in assembly language.
- The first OS that was not written in assembly language was the Master Control Program (MCP).
- The advantages of using a higher-level language for implementing operating systems are: The code can be written faster, more compact, easy to port to other systems and is easier to understand and debug.
- The only disadvantages of implementing an operating system in a higher-level language are reduced speed and increased storage requirements.

Operating-System Structure

Simple Structure

- Many operating systems do not have well-defined structures. They started as small, simple, and limited systems and then grew beyond their original scope. Eg: MS-DOS.
- In MS-DOS, the interfaces and levels of functionality are not well separated. Application programs can access basic I/O routines to write directly to the display and disk drives. Such freedom leaves MS-DOS in bad state and the entire system can crash down when user programs fail.
- UNIX OS consists of two separable parts: the kernel and the system programs. The kernel is further separated into a series of interfaces and device drivers. The kernel provides the file system, CPU scheduling, memory management, and other operating-system functions through system calls.

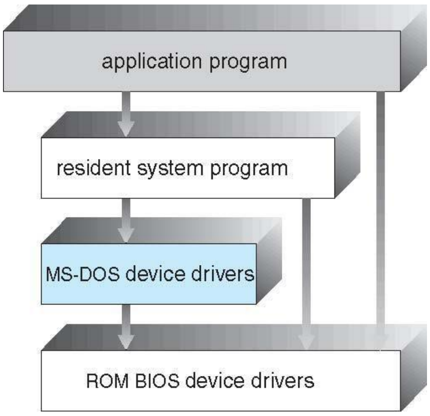


Figure: MS-DOS layer structure.

Layered Approach

- The OS is broken into number of layers (levels). Each layer rests on the layer below it, and relies on the services provided by the next lower layer.
- Bottom layer (layer 0) is the hardware and the topmost layer is the user interface.
- A typical layer, consists of data structure and routines that can be invoked by higher-level layer.
- Advantage of layered approach is simplicity of construction and debugging.
- The layers are selected so that each uses functions and services of only lower-level layers. So simplifies debugging and system verification. The layers are debugged one by one from the lowest and if any layer doesn’t work, then error is due to that layer only, as the lower layers are already debugged. Thus the design and implementation is simplified.
- A layer need not know how its lower level layers are implemented. Thus hides the operations from higher layers.

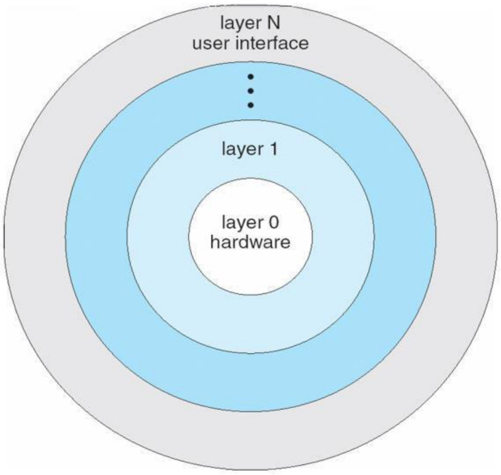


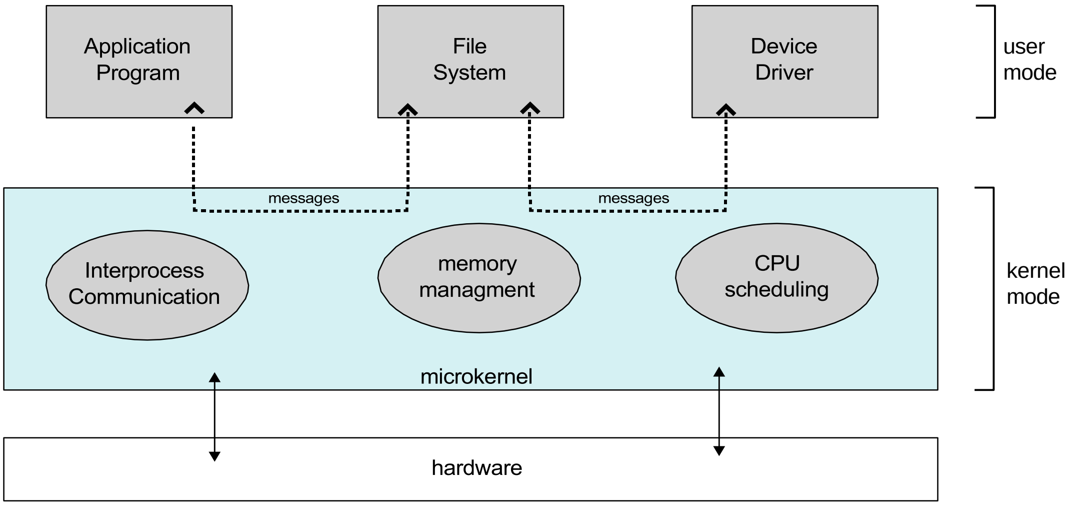
Figure: A layered Operating System

Disadvantages of layered approach:

- The various layers must be appropriately defined, as a layer can use only lower level layers.
- Less efficient than other types, because any interaction with layer 0 required from top layer. The system call should pass through all the layers and finally to layer 0. This is an overhead.

Microkernels

- This method structures the operating system by removing all nonessential components from the kernel and implementing them as system and user-level programs thus making the kernel as small and efficient as possible.
- The removed services are implemented as system applications.
- Most microkernels provide basic process and memory management, and message passing between other services.
- The main function of the microkernel is to provide a communication facility between the client program and the various services that are also running in user space.



Benefit of microkernel –

- System expansion can also be easier, because it only involves adding more system applications, not rebuilding a new kernel.

- Mach was the first and most widely known microkernel, and now forms a major component of Mac OSX.

Disadvantage of Microkernel -

- Performance overhead of user space to kernel space communication

Modules

- Modern OS development is object-oriented, with a relatively small core kernel and a set of ***modules*** which can be linked in dynamically.
- Modules are similar to layers in that each subsystem has clearly defined tasks and interfaces, but any module is free to contact any other module, eliminating the problems of going through multiple intermediary layers.
- The kernel is relatively small in this architecture, similar to microkernels, but the kernel does not have to implement message passing since modules are free to contact each other directly. Eg: Solaris, Linux and MacOSX.

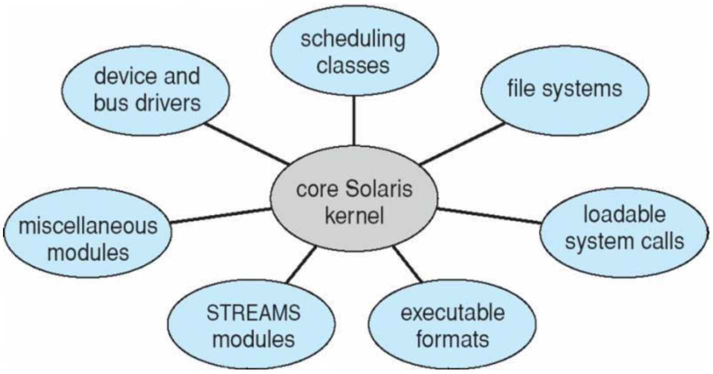


Figure: Solaris loadable modules

- The Max OSX architecture relies on the Mach microkernel for basic system management services, and the BSD kernel for additional services. Application services and dynamically loadable modules (kernel extensions) provide the rest of the OS functionality.
- Resembles layered system, but a module can call any other module.
- Resembles microkernel, the primary module has only core functions and the knowledge of how to load and communicate with other modules.

Virtual Machines

Q) Demonstrate the concept of virtual machine with an example

- The fundamental idea behind a virtual machine is to abstract the hardware of a single computer (the CPU, memory, disk drives, network interface cards, and so forth) into several different execution environments, thereby creating the illusion that each separate execution environment is running its own private computer.

- Creates an illusion that a process has its own processor with its own memory.
- Host OS is the main OS installed in system and the other OS installed in the system are called guest OS.

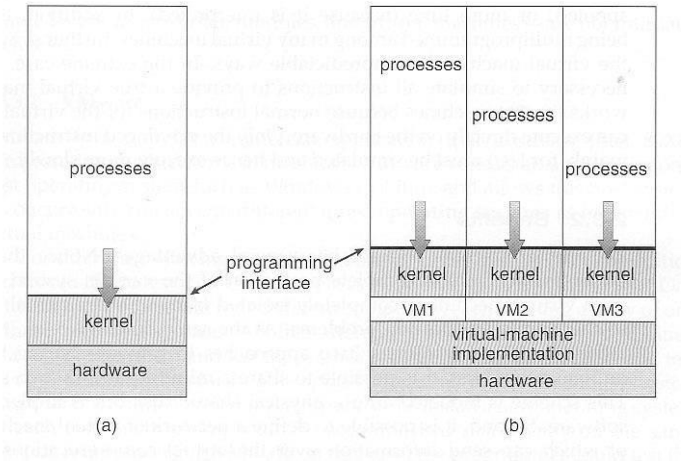


Figure: System modes. (A) Non-virtual machine (b) Virtual machine

Virtual machines first appeared as the VM Operating System for IBM mainframes in 1972.

Implementation

- The virtual-machine concept is useful, it is difficult to implement.
- Work is required to provide an exact duplicate of the underlying machine. Remember that the underlying machine has two modes: user mode and kernel mode.
- The virtual-machine software can run in kernel mode, since it is the operating system. The virtual machine itself can execute in only user mode.

Benefits

- Able to share the same hardware and run several different execution environments(OS).
- Host system is protected from the virtual machines and the virtual machines are protected from one another. A virus in guest OS, will corrupt that OS but will not affect the other guest systems and host systems.
- Even though the virtual machines are separated from one another, software resources can be shared among them. Two ways of sharing s/w resource for communication are:
  - To share a file system volume (part of memory).
  - To develop a virtual communication network to communicate between the virtual machines.
- The operating system runs on and controls the entire machine. Therefore, the current system must be stopped and taken out of use while changes are made and tested. This period is commonly called *system development time.* In virtual machines such problem is eliminated. User programs are executed in one virtual machine and system development is done in another environment.

- Multiple OS can be running on the developer's system **concurrently**. This helps in rapid porting and testing of programmer's code in different environments.
- **System consolidation** – two or more systems are made to run in a single system.

Simulation –
Here the host system has one system architecture and the guest system is compiled in different architecture. The compiled guest system programs can be run in an emulator that translates each instructions of guest program into native instructions set of host system.

Para-Virtualization –

This presents the guest with a system that is similar but not identical to the guest's preferred system. The guest must be modified to run on the para-virtualized hardware.

Examples

VMware

- VMware is a popular commercial application that abstracts Intel 80X86 hardware into isolated virtual machines. The virtualization tool runs in the user-layer on top of the host OS. The virtual machines running in this tool believe they are running on bare hardware, but the fact is that it is running inside a user-level application.
- VMware runs as an application on a host operating system such as Windows or Linux and allows this host system to concurrently run several different **guest operating systems** as independent virtual machines.

In below scenario, Linux is running as the host operating system; FreeBSD, Windows NT, and Windows XP are running as guest operating systems. The virtualization layer is the heart of VMware, as it abstracts the physical hardware into isolated virtual machines running as guest operating systems. Each virtual machine has its own virtual CPU, memory, disk drives, network interfaces, and so forth.

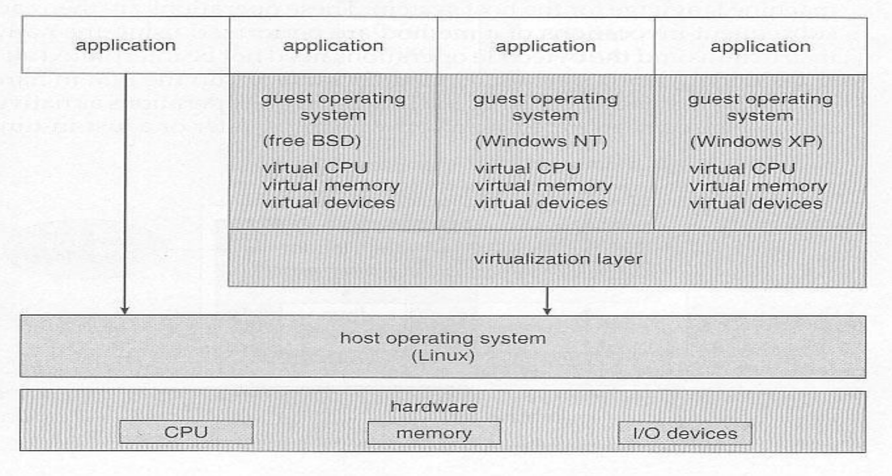


Figure: VMware architecture

The Java Virtual Machine

- Java was designed from the beginning to be platform independent, by running Java only on a Java Virtual Machine, JVM, of which different implementations have been developed for numerous different underlying HW platforms.
- Java source code is compiled into Java byte code in .class files. Java byte code is binary instructions that will run on the JVM.
- The JVM implements memory management and garbage collection.
- JVM consists of class loader and Java Interpreter. Class loader loads compiled .class files from both java program and java API for the execution of java interpreter. Then it checks the .class file for validity.

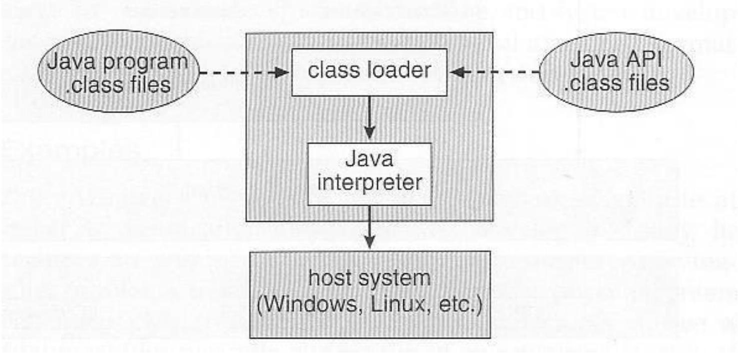


Figure: The JVM

Operating System Generation

- Operating systems are designed to run on any of a class of machines; the system must be configured for each specific computer site.
- SYSGEN program obtains information concerning the specific configuration of the hardware system.
  - *Booting* – starting a computer by loading the kernel.
  - *Bootstrap program* – code stored in ROM that is able to locate the kernel, load it into memory, and start its execution.

System Boot

- Operating system must be made available to hardware so hardware can start it.
- Small piece of code – **bootstrap loader**, locates the kernel, loads it into memory, and starts it Sometimes two-step process where **boot block** at fixed location loads bootstrap loader.
- When power initialized on system, execution starts at a fixed memory location Firmware used to hold initial boot code

MODULE-2
PROCESS MANAGEMENT

Process concept

- A process is a program under execution.
- Its current activity is indicated by PC (Program Counter) and the contents of the processor's registers..

The Process

Process memory is divided into four sections as shown in the figure below:

- The stack is used to store local variables, function parameters, function return values, return address etc.
- The heap is used for dynamic memory allocation.
- The data section stores global and static variables.
- The text section comprises the compiled program code.
- Note that, there is a free space between the stack and the heap. When the stack is full, it grows downwards and when the heap is full, it grows upwards.

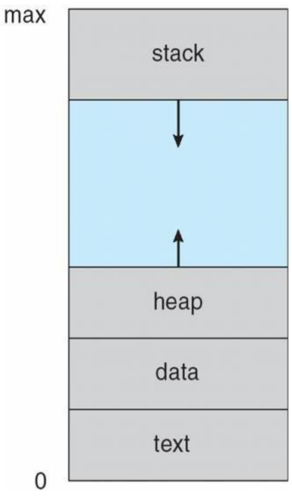


Figure: Process in memory.

Process State

Q) Illustrate with a neat sketch, the process states and process control block.

Process State

A Process has 5 states. Each process may be in one of the following states –

1. **New** - The process is in the stage of being created.
2. **Ready** - The process has all the resources it needs to run. It is waiting to be assigned to the processor.
3. **Running** – Instructions are being executed..
4. **Waiting** - The process is waiting for some event to occur. For example the process may be waiting for keyboard input, disk access request, inter-process messages, a timer to go off, or a child process to finish.
5. **Terminated** - The process has completed its execution.

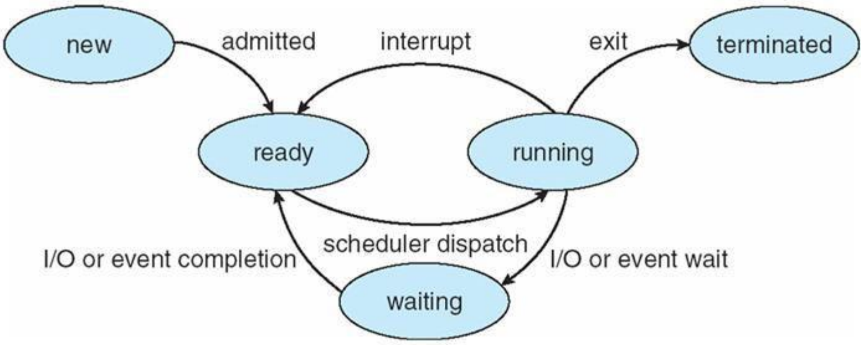


Figure: Diagram of process state

Process Control Block

For each process there is a Process Control Block (PCB), which stores the process-specific information as shown below –

- **Process State** – The state of the process may be new, ready, running, waiting, and so on.
- **Program counter** – The counter indicates the address of the next instruction to be executed for this process.
- **CPU registers** - The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.
- **CPU scheduling information**- This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
- **Memory-management information** – This include information such as the value of the base and limit registers, the page tables, or the segment tables.

- **Accounting information** – This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
- **I/O status information** – This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

The PCB simply serves as the repository for any information that may vary from process to process.

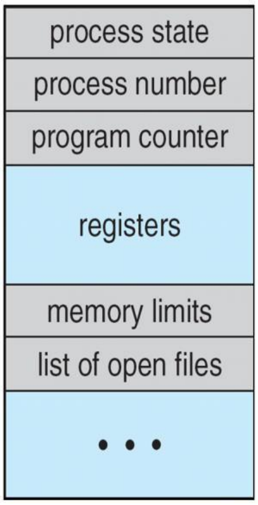


Figure: Process control block (PCB)

CPU Switch from Process to Process

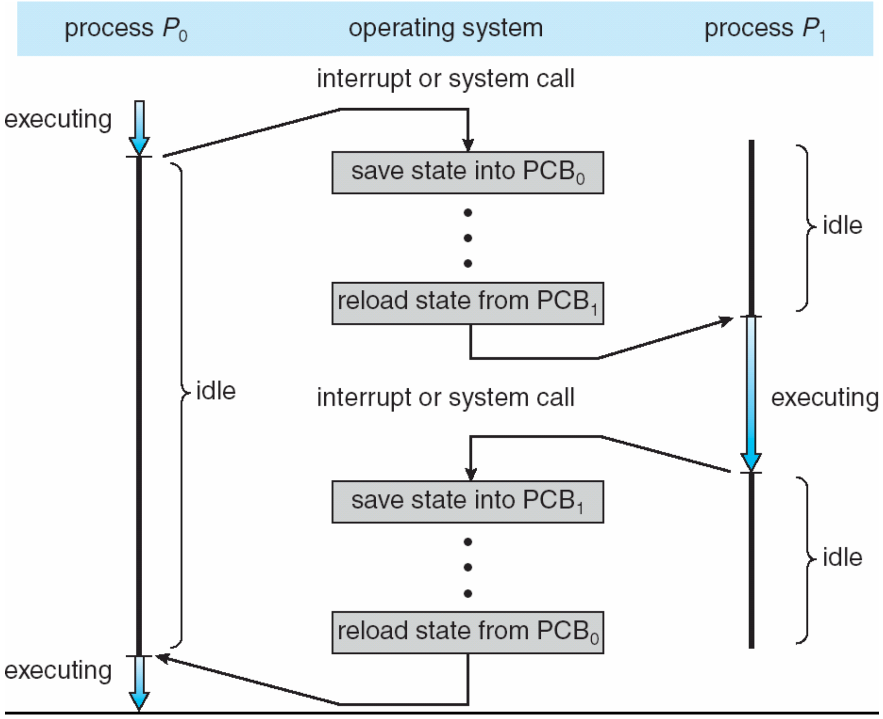


Figure: Diagram showing CPU switch from process to process.

Process Scheduling

Scheduling Queues

- As processes enter the system, they are put into a job queue, which consists of all processes in the system.
- The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the ready queue. This queue is generally stored as a linked list.
- A ready-queue header contains pointers to the first and final PCBs in the list. Each PCB includes a pointer field that points to the next PCB in the ready queue.

Ready Queue and Various I/O Device Queues

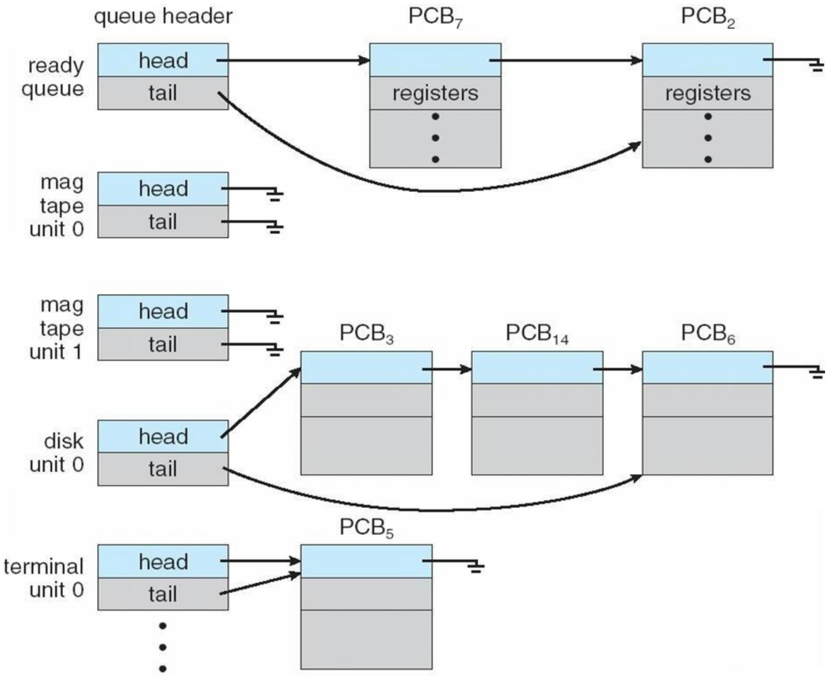


Figure: The ready queue and various I/O device queues

- A common representation of process scheduling is a ***queueing diagram***. Each rectangular box in the diagram represents a queue. Two types of queues are present: the ready queue and a set of device queues. The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.

- A new process is initially put in the ready queue. It waits in the ready queue until it is selected for execution and is given the CPU. Once the process is allocated the CPU and is executing, one of several events could occur:
  - The process could issue an I/O request, and then be placed in an I/O queue.
  - The process could create a new subprocess and wait for its termination.
  - The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.

In the first two cases, the process eventually switches from the waiting state to the ready state, and is then put back in the ready queue. A process continues this cycle until it terminates, at which time it is removed from all queues.

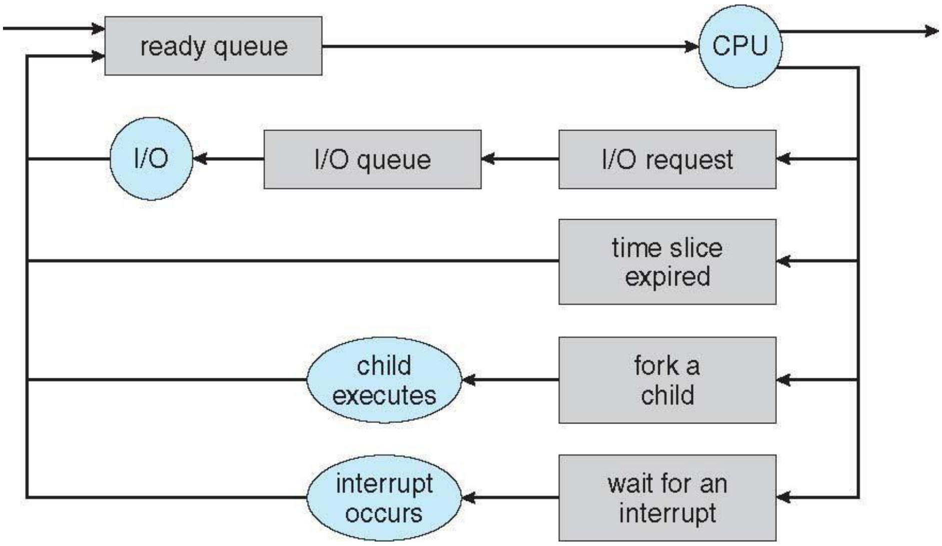


Figure: Queueing-diagram representation of process scheduling.

Schedulers

Schedulers are software which selects an available program to be assigned to CPU.

- A **long-term scheduler or Job scheduler** – selects jobs from the job pool (of secondary memory, disk) and loads them into the memory.
  If more processes are submitted, than that can be executed immediately, such processes will be in secondary memory. It runs infrequently, and can take time to select the next process.

- **The short-term scheduler, or CPU Scheduler** – selects job from memory and assigns the CPU to it. It must select the new process for CPU frequently.
- **The medium-term scheduler** - selects the process in ready queue and reintroduced into the memory.

Processes can be described as either:

- I/O-bound process – spends more time doing I/O than computations,
- CPU-bound process – spends more time doing computations and few I/O operations.

An efficient scheduling system will select a good mix of **CPU-bound** processes and **I/O bound** processes.

- If the scheduler selects **more I/O bound process**, then I/O queue will be full and ready queue will be empty.
- If the scheduler selects **more CPU bound process**, then ready queue will be full and I/O queue will be empty.

Time sharing systems employ a **medium-term scheduler**. It swaps out the process from ready queue and swap in the process to ready queue. When system loads get high, this scheduler will swap one or more processes out of the ready queue for a few seconds, in order to allow smaller faster jobs to finish up quickly and clear the system.

Advantages of medium-term scheduler –

- To remove process from memory and thus reduce the degree of multiprogramming (number of processes in memory).
- To make a proper mix of processes(CPU bound and I/O bound)

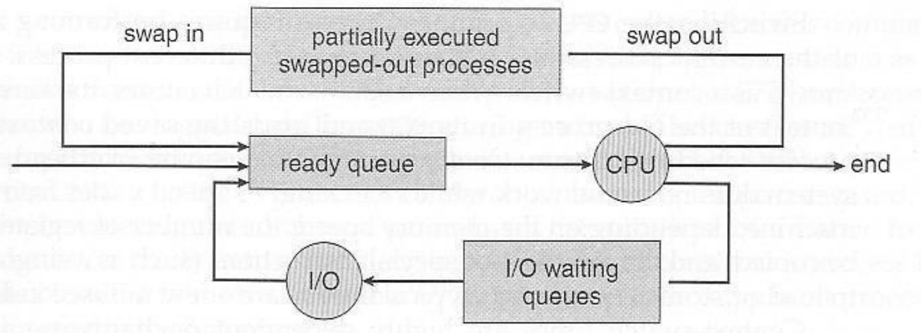


Figure 3.8 Addition of medium-term scheduling to the queueing diagram.

Context switching

- The task of switching a CPU from one process to another process is called context switching. Context-switch times are highly dependent on hardware support (Number of CPU registers).
- Whenever an interrupt occurs (hardware or software interrupt), the state of the currentlyrunning process is saved into the PCB and the state of another process is restored from the PCBto the CPU.
- Context switch time is an overhead, as the system does not do useful work while switching.

Operations on Processes

Q) Demonstrate the operations of process creation and process termination in UNIX

Process Creation

- A process may create several new processes. The creating process is called a **parent** process, and the new processes are called the **children** of that process. Each of these new processes may in turn create other processes. Every process has a unique process ID.
- On typical Solaris systems, the process at the top of the tree is the '**sched**' process with PID of 0. The **'sched'** process creates several children processes – **init**, **pageout** and **fsflush**. Pageout and fsflush are responsible for managing memory and file systems. The init process with a PID of 1, serves as a parent process for all user processes.

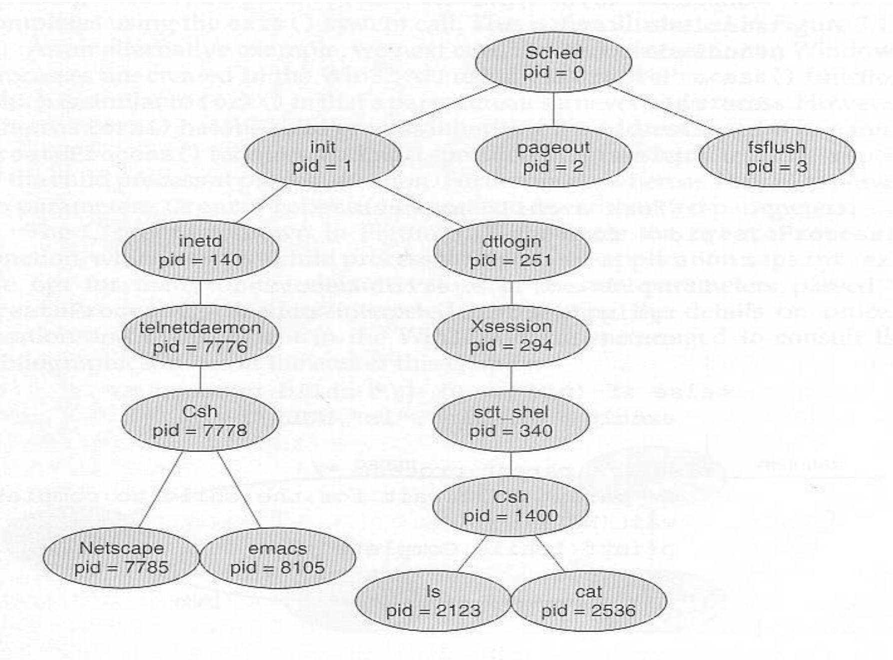


Figure 3.9 A tree of processes on a typical Solaris system.

A process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its task. When a process creates a subprocess, the subprocess may be able to obtain its resources in two ways :

- directly from the operating system
- Subprocess may take the resources of the parent process.

  The resource can be taken from parent in two ways –

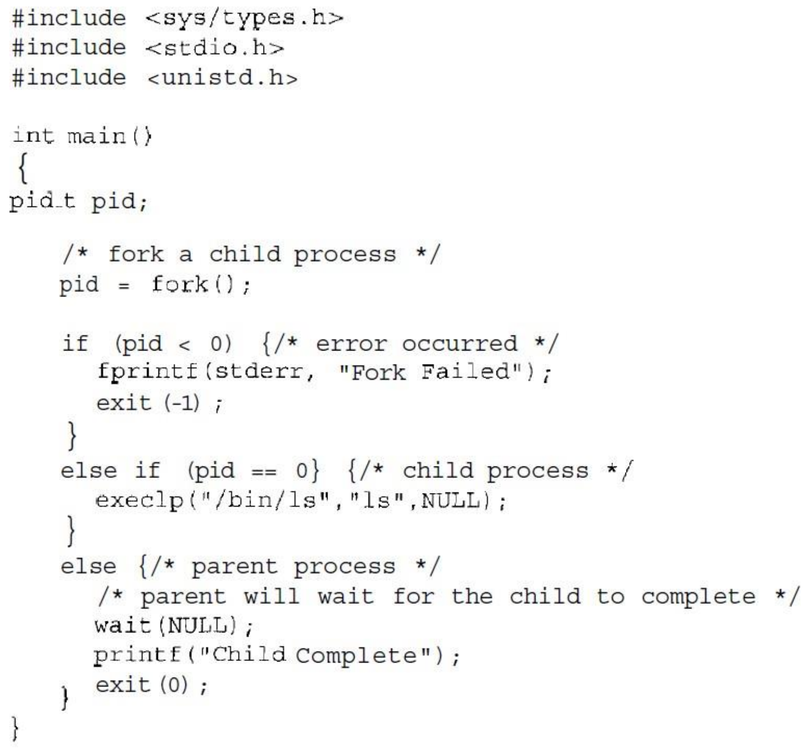
  - The parent may have to partition its resources among its children
  - Share the resources among several children.

There are two options for the parent process after creating the child:

- Wait for the child process to terminate and then continue execution. The parent makes a wait() system call.
- Run concurrently with the child, continuing to execute without waiting.

Two possibilities for the address space of the child relative to the parent:

- The child may be an exact duplicate of the parent, sharing the same program and data segments in memory. Each will have their own PCB, including program counter, registers, and PID. This is the behaviour of the **fork** system call in UNIX.
- The child process may have a new program loaded into its address space, with all new code and data segments. This is the behaviour of the **spawn** system calls in Windows.



```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) {/* error occurred */
      fprintf(stderr, "Fork Failed");
      exit (-1) ;
    }
    else if (pid == 0} {/* child process */
      execlp("/bin/ls","ls",NULL);
    }
    else {/* parent process */
      /* parent will wait for the child to complete */
      wait(NULL);
      printf("Child Complete");
    }  exit (0) ;
}
```

Figure 3.10 C program forking a separate process.

In UNIX OS, a child process can be created by **fork()** system call. The **fork** system call, if successful, returns the PID of the child process to its parents and returns a zero to the child process. If failure, it returns -1 to the parent. Process IDs of current process or its direct parent can be accessed using the getpid( ) and getppid( ) system calls respectively.

The parent waits for the child process to complete with the wait() system call. When the child process completes, the parent process resumes and completes its execution.

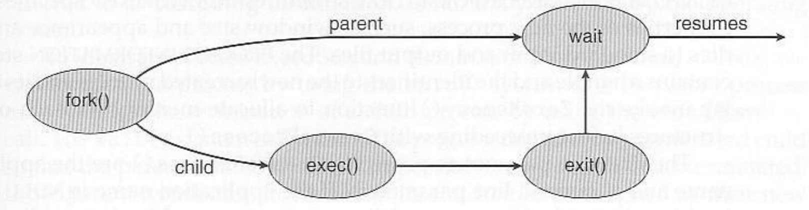


Figure 3.11 Process creation.

In windows the child process is created using the function **createprocess( )**. The createprocess( ) returns 1, if the child is created and returns 0, if the child is not created.

Process Termination

- A process terminates when it finishes executing its last statement and asks the operating system to delete it, by using the **exit**( ) system call. All of the resources assigned to the process like memory, open files, and I/O buffers, are deallocated by the operating system.
- A process can cause the termination of another process by using appropriate system call. The parent process can terminate its child processes by knowing of the PID of the child.
- A parent may terminate the execution of children for a variety of reasons, such as:
  - The child has exceeded its usage of the resources, it has been allocated.
  - The task assigned to the child is no longer required.
  - The parent is exiting, and the operating system terminates all the children. This is called **cascading termination**.

Interprocess Communication

Q) What is interprocess communication? Explain types of IPC.

Interprocess Communication- Processes executing may be either co-operative or independent processes.

- ***Independent Processes*** – processes that cannot affect other processes or be affected by other processes executing in the system.
- ***Cooperating Processes*** – processes that can affect other processes or be affected by other processes executing in the system.

Co-operation among processes are allowed for following reasons –

- **Information Sharing** - There may be several processes which need to access the same file. So the information must be accessible at the same time to all users.
- **Computation speedup** - Often a solution to a problem can be solved faster if the problem can be broken down into sub-tasks, which are solved simultaneously (particularly when multiple processors are involved.)
- **Modularity** - A system can be divided into cooperating modules and executed by sending information among one another.
- **Convenience** - Even a single user can work on multiple task by information sharing.

Cooperating processes require some type of inter-process communication. This is allowed by two models:

1. Shared Memory systems
2. Message passing systems.

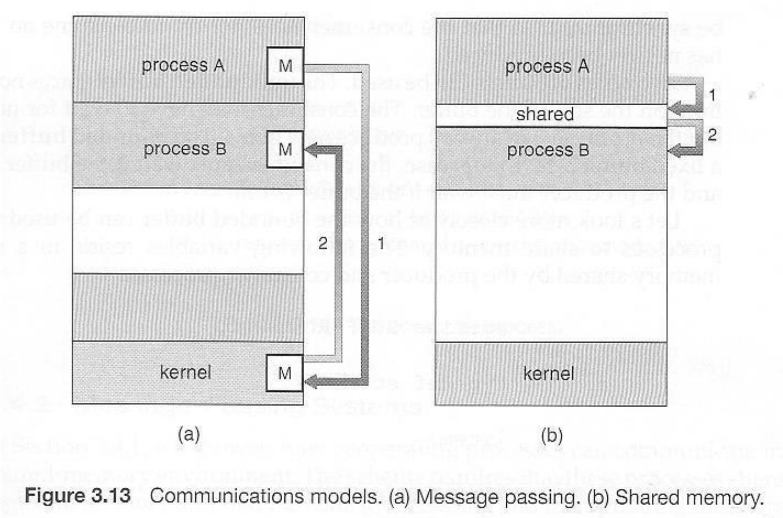


Figure 3.13 Communications models. (a) Message passing. (b) Shared memory.

Sl No	Shared Memory	Message passing
1.	A region of memory is shared by communicating processes, into which the information is written and read	Message exchange is done among the processes by using objects.
2.	Useful for sending large block of data	Useful for sending small data.
3.	System call is used only to create shared memory	System call is used during every read and write operation.
4.	Message is sent faster, as there are no system calls	Message is communicated slowly.

- **Shared Memory** is faster once it is set up, because no system calls are required and access occurs at normal memory speeds. Shared memory is generally preferable when large amounts of information must be shared quickly on the same computer.
- **Message Passing** requires system calls for every message transfer, and is therefore slower, but it is simpler to set up and works well across multiple computers. Message passing is generally preferable when the amount and/or frequency of data transfers is small.

Shared-Memory Systems

- A region of shared-memory is created within the address space of a process, which needs to communicate. Other process that needs to communicate uses this shared memory.
- The form of data and position of creating shared memory area is decided by the process. Generally a few messages must be passed back and forth between the cooperating processes first in order to set up and coordinate the shared memory access.
- The process should take care that the two processes will not write the data to the shared memory at the same time.

Producer-Consumer Example Using Shared Memory

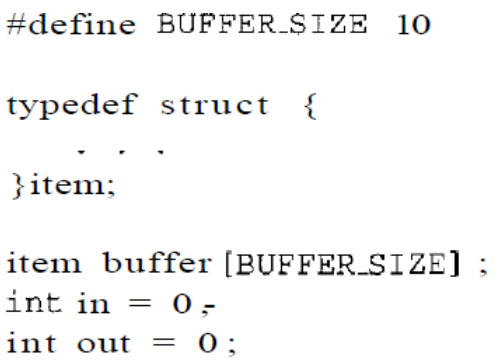
- This is a classic example, in which one process is producing data and another process is consuming the data.
- The data is passed via an intermediary buffer (shared memory). The producer puts the data to the buffer and the consumer takes out the data from the buffer. A producer can produce one item while the consumer is consuming another item. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced. In this situation, the consumer must wait until an item is produced.
- There are two types of buffers into which information can be put –
    - Unbounded buffer
    - Bounded buffer

- **With Unbounded buffer**, there is no limit on the size of the buffer, and so on the data produced by producer. But the consumer may have to wait for new items.

- **With bounded-buffer** – As the buffer size is fixed. The producer has to wait if the buffer is full and the consumer has to wait if the buffer is empty.

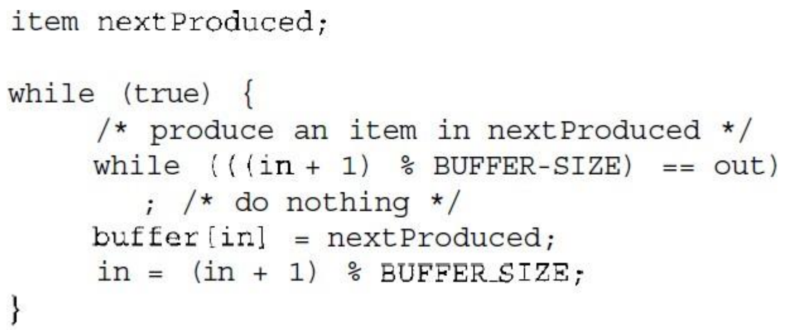
This example uses shared memory as a circular queue. The **in** and **out** are two pointers to the array. Note in the code below that only the producer changes "in", and only the consumer changes "out".

- First the following data is set up in the shared memory area:



```
#define BUFFER_SIZE 10

typedef struct {
    . . .
}item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

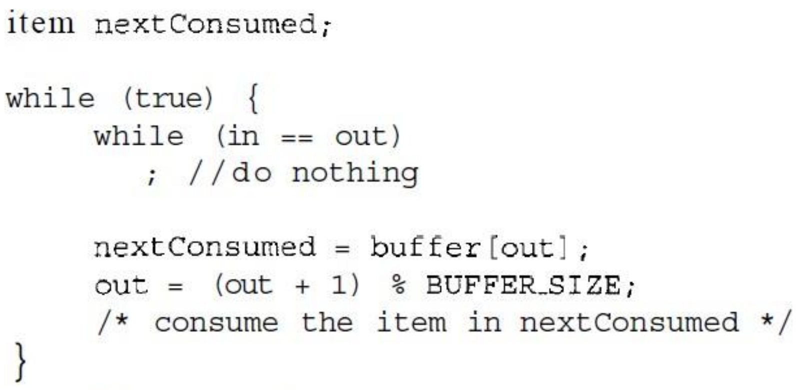
- The producer process –
  Note that the buffer is full when [(in+1)%BUFFER_SIZE == out]



```
item nextProduced;

while (true) {
     /* produce an item in nextProduced */
     while (((in + 1) % BUFFER-SIZE) == out)
        ; /* do nothing */
     buffer[in] = nextProduced;
     in = (in + 1) % BUFFER_SIZE;
}
```

Figure The producer process.

- The consumer process –
  Note that the buffer is empty when [in == out]



```
item nextConsumed;

while (true) {
     while (in == out)
        ; //do nothing

     nextConsumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     /* consume the item in nextConsumed */
}
```

Figure The consumer process.

Message-Passing Systems

A mechanism to allow process communication without sharing address space. It is used in distributed systems.

- Message passing systems uses system calls for "send message" and "receive message".
- A communication link must be established between the cooperating processes before messages can be sent.
- There are three methods of creating the link between the sender and the receiver-
  - Direct or indirect communication (naming)
  - Synchronous or asynchronous communication (Synchronization)
  - Automatic or explicit buffering.

1. Naming

Processes that want to communicate must have a way to refer to each other. They can use either direct or indirect communication.

a) Direct communication the sender and receiver must explicitly know each other's name. The syntax for send() and receive() functions are as follows-

- **send** (*P*, *message*) – send a message to process P
- **receive**(*Q*, *message*) – receive a message from process Q

Properties of communication link :

- A link is established automatically between every pair of processes that wants to communicate. The processes need to know only each other's identity to communicate.
- A link is associated with exactly one pair of communicating processes
- Between each pair, there exists exactly one link.

Types of addressing in direct communication –

- Symmetric addressing – the above described communication is symmetric communication. Here both the sender and the receiver processes have to name each other to communicate.
- Asymmetric addressing – Here only the sender name is mentioned, but the receiving data can be from any system.

  send(P, message) --- Send a message to process ***P***

  receive(id, message). Receive a message from any process

Disadvantages of direct communication – any changes in the identifier of a process, may have to change the identifier in the whole system(sender and receiver), where the messages are sent and received.

<u>b)</u> <u>Indirect communication</u> uses shared mailboxes, or ports.

A mailbox or port is used to send and receive messages. Mailbox is an object into which messages can be sent and received. It has a unique ID. Using this identifier messages are sent and received.

Two processes can communicate only if they have a shared mailbox. The send and receive functions are –

- **send**(*A*, *message*) – send a message to mailbox A
- **receive**(*A*, *message*) – receive a message from mailbox A

<u>Properties of communication link:</u>

- A link is established between a pair of processes only if they have a shared mailbox
- A link may be associated with more than two processes
- Between each pair of communicating processes, there may be any number of links, each link is associated with one mailbox.
- A mail box can be owned by the operating system. It must take steps to –
    - create a new mailbox
    - send and receive messages from mailbox
    - delete mailboxes.

<u>2.</u> <u>Synchronization</u>

The send and receive messages can be implemented as either **blocking** or **non-blocking**.

- **Blocking (synchronous) send -** sending process is blocked (waits) until the message is received by receiving process or the mailbox.
- **Non-blocking (asynchronous) send** - sends the message and continues (doesnot wait)

- **Blocking (synchronous) receive -** The receiving process is blocked until a message is available
- **Non-blocking (asynchronous) receive** - receives the message without block. The received message may be a valid message or null.

<u>3.</u> <u>Buffering</u>

When messages are passed, a temporary queue is created. Such queue can be of three capacities:

- **Zero capacity** – The buffer size is zero (buffer does not exist). Messages are not stored in the queue. The senders must block until receivers accept the messages.
- **Bounded capacity**- The queue is of fixed size(n). Senders must block if the queue is full. After sending 'n' bytes the sender is blocked.
- Unbounded capacity **- The queue is of infinite capacity. The sender never blocks.**

MODULE 2

MULTITHREADED PROGRAMMING

- A thread is a basic unit of CPUutilization.
- It consistsof
    - thread ID
    - PC
    - register-set and
    - stack.
- It shares with other threads belonging to the same process its code-section &data-section.
- A traditional (or heavy weight) process has a single thread ofcontrol.
- If a process has multiple threads of control, it can perform more than one task at a time.

such a process is called **multithreaded process**

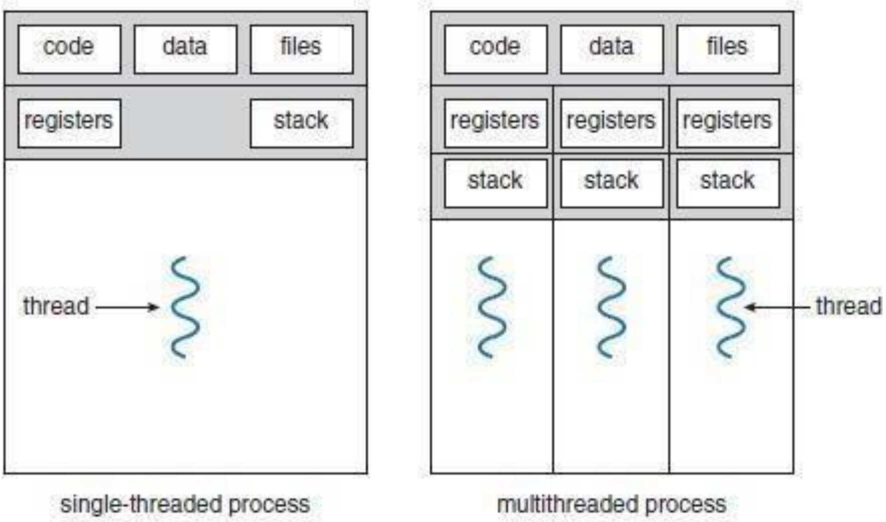


Fig: Single-threaded and multithreaded processes

Motivation for Multithreaded Programming

1. The software-packages that run on modern PCs aremultithreaded.An application is implemented as a separate process with several threads of control. For ex: A word processor mayhave

- first thread for displaying graphics
- second thread for responding to keystrokesand
- Thirdthread for performing grammarchecking.

2. In some situations, a single application may be required to perform several similartasks. For ex: A web-server may create a separate thread for each client requests. This allows the server to service several concurrent requests.

3. RPC servers aremultithreaded.

- When a server receives a message, it services the message using separate concurrent threads.

4. Most OS kernels aremultithreaded;

- Several threads operate in kernel, and each thread performs a specific task, suchasmanaging devices or interrupt handling.

Benefits of Multithreaded Programming

- ***Responsiveness*** A program may be allowed to continue running even if part of it is blocked. Thus, increasing responsiveness to the user.

- ***Resource Sharing*** By default, threads share the memory (and resources) of the process to which they belong. Thus, an application is allowed to have severaldifferent threads of activity within the sameaddress-space.
- ***Economy*** Allocating memory and resources for process-creation is costly. Thus, it is more economical to create and context-switchthreads.
- ***Utilization of Multiprocessor Architectures*** In a multiprocessor architecture, threads may be running in parallel on different processors. Thus, parallelism will beincreased.

MULTITHREADING MODELS

- Support for threads may be provided ateither
  1. The user level, for **user threads** or
  2. By the kernel, for **kernel threads**.
- User-threads are supported above the kernel and are managed withoutkernelsupport. Kernel-threads are supported and managed directly by the OS.
- Three ways of establishing relationship between user-threads &kernel-threads:
  1. Many-to-onemodel
  2. One-to-one modeland
  3. Many-to-manymodel.

Many-to-One Model

- Many user-level threads are mapped to one kernel thread.

  Advantages:
  - Thread management is done by the thread library in user space, so it isefficient.

  Disadvantages:
  - The entire process will block if a thread makes a blockingsystem-call.
  - Multiple threads are unable to run in parallel onmultiprocessors.
- Forexample:
  - Solaris green threads
  - GNU portable threads.

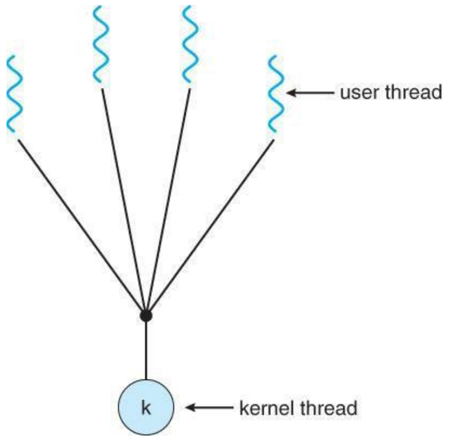


Fig: Many-to-one model

One-to-One Model

- Each user thread is mapped to a kernel thread.

  Advantages:
  - It provides more concurrency by allowing another thread to run when a thread makes a blockingsystem-call.
  - Multiple threads can run in parallel on multiprocessors.

  Disadvantage:
  - Creating a user thread requires creating the corresponding kernel thread.
- For example:
  - Windows NT/XP/2000, Linux

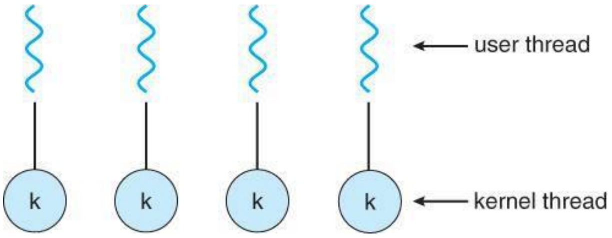


Fig: one-to-one model

Many-to-Many Model

- Many user-level threads are multiplexed to a smaller number of kernel threads.

Advantages:

- Developers can create as many user threads as necessary
- The kernel threads can run in parallel on amultiprocessor.
- When a thread performs a blocking system-call, kernel can schedule another thread for execution.

Two Level Model

- A variation on the many-to-many model is the two level-model
- Similar to M:N, except that it allows a user thread to be bound to kernelthread.
- forexample:
  - HP-UX
  - Tru64 UNIX

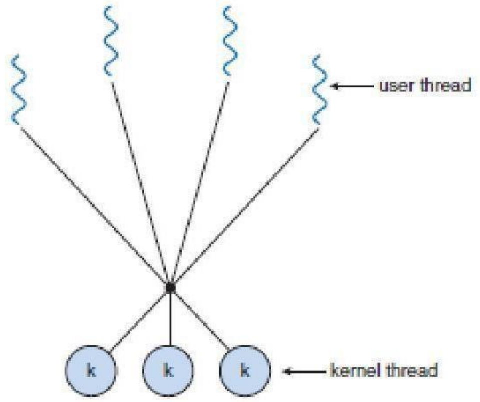


Fig: Many-to-many model

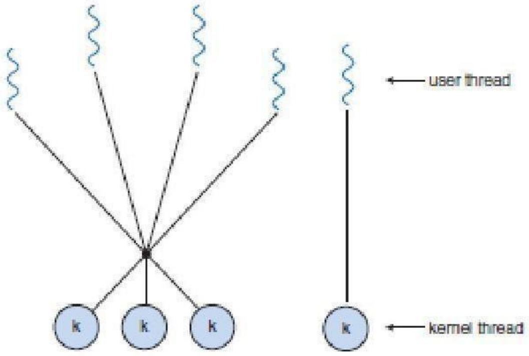


Fig: Two-level model

Thread Libraries

- It provides the programmer with an API for the creation and management ofthreads.

- Two ways of implementation:

1. *First Approach:*

   Provides a library entirely in user space with no kernel support. All code and data structures for the library exist in the user space.

2. *SecondApproach*

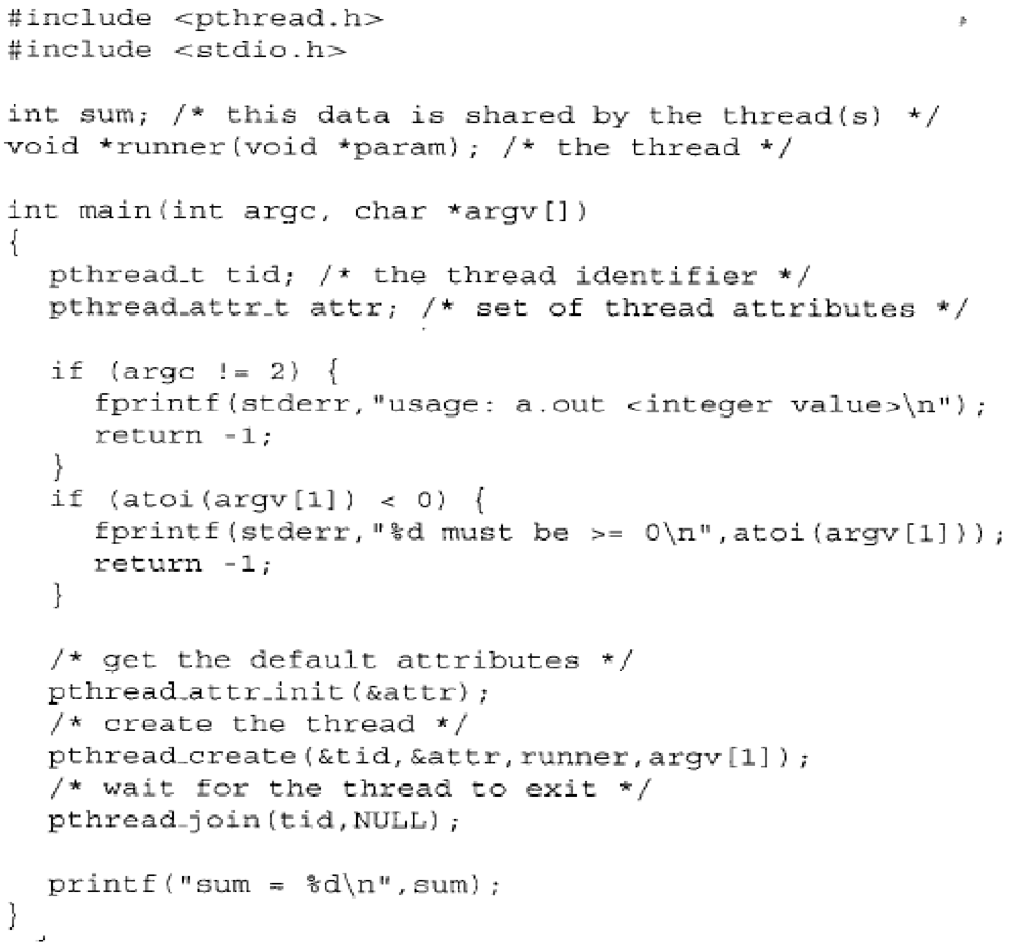
   Provides a library entirely in user space with no kernel support. All code and data structures for the library exist in the user space.

Three main threadlibraries:

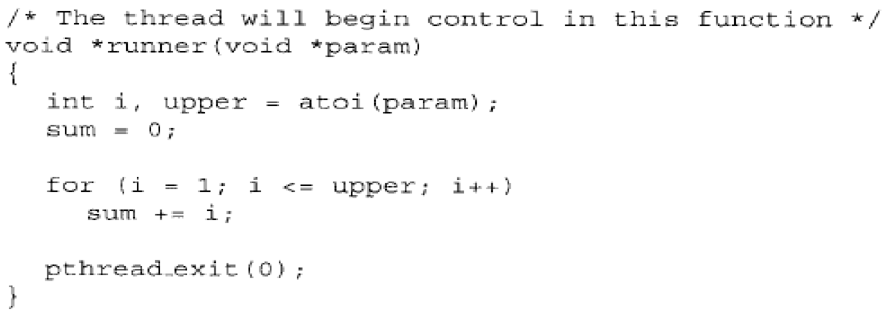
1. POSIXP threads
2. Win32 and
3. Java.

Pthreads

- This is a POSIX standard API for thread creation andsynchronization.
- This is a specification for thread-behavior, not an implementation.
- OS designers may implement the specification in any way theywish.
- Commonly used in: UNIX andSolaris.



```
#include <pthread.h>
#include <stdio.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* the thread */

int main(int argc, char *argv[])
{
  pthread_t tid; /* the thread identifier */
  pthread_attr_t attr; /* set of thread attributes */

  if (argc != 2) {
    fprintf(stderr,"usage: a.out <integer value>\n");
    return -1;
  }
  if (atoi(argv[1]) < 0) {
    fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
    return -1;
  }

  /* get the default attributes */
  pthread_attr_init(&attr);
  /* create the thread */
  pthread_create(&tid,&attr,runner,argv[1]);
  /* wait for the thread to exit */
  pthread_join(tid,NULL);

  printf("sum = %d\n",sum);
}
```



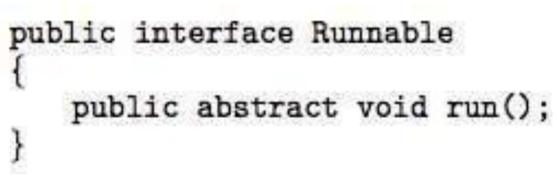
```
/* The thread will begin control in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

Win32 threads

- Implements the one-to-onemapping
- Each threadcontains
  - A threadid
  - Registerset
  - Separate user and kernelstacks
  - Private data storagearea
- The register set, stacks, and private storage area are known as the context of the threads The primary data structures of a thread include:
  - ETHREAD (executive threadblock)
  - KTHREAD (kernel threadblock)
  - TEB (thread environmentblock)

Java Threads

- Threads are the basic model of program-executionin
  - Java program and
  - Java language.
- The API provides a rich set of features for the creation and management of threads.
- All Java programs comprise at least a single thread ofcontrol.
- Two techniques for creating threads:
  1. Create a new class that is derived from the Thread class and override its run() method.
  2. Define a class that implements the Runnable interface. The Runnable interface is defined as follows:



```
public interface Runnable
{
   public abstract void run();
}
```

THREADING ISSUES

fork() and exec() System-calls

- fork() is used to create a separate, duplicateprocess.
- If one thread in a program calls fork(),then
  1. Some systems duplicates all threads and
  2. Other systems duplicate only the thread that invoked the forkO.
- If a thread invokes the exec(), the program specified in the parameter to exec() will replace the entire process including allthreads.

Thread Cancellation

- This is the task of terminating a thread before it hascompleted.
- Target thread is the thread that is to be cancelled
- Thread cancellation occurs in two differentcases:
  1. *Asynchronous cancellation*: One thread immediately terminates the targetthread.
  2. *Deferred cancellation*: The target thread periodically checks whether it should be terminated.

Signal Handling

- In UNIX, a signal is used to notify a process that a particular event hasoccurred.
- All signals follow thispattern:
  1. A signal is generated by the occurrence of a certainevent.
  2. A generated signal is delivered to aprocess.
  3. Once delivered, the signal must behandled.
- A signal handler is used to processsignals.
- A signal may be received either synchronously or asynchronously, depending on thesource.
  1. *Synchronoussignals*
     - Delivered to the same process that performed the operation causing the signal.
     - E.g. illegal memory access and division by 0.
  2. *Asynchronoussignals*
     - Generated by an event external to a running process.
     - E.g. user terminating a process with specific keystrokes<ctrl><c>.

- Every signal can be handled by one of two possiblehandlers:
    1. *A Default SignalHandler*
        - Run by the kernel when handling the signal.
    2. *A User-defined SignalHandler*
        - Overrides the default signal handler.
- In ***single-threaded programs***, delivering signals is simple (since signals are always delivered to a process).
- In ***multithreaded programs***, delivering signals is more complex. Then, the following options exist:
    1. Deliver the signal to the thread to which the signal applies.
    2. Deliver the signal to every thread in process
    3. Deliver the signal to certain threads in the process.
    4. Assign a specific thread to receive all signals for the process.

THREAD POOLS

- The basic idea is to
    - create a no. of threads at process-startup and
    - place the threads into a pool (where they sit and wait for work).
- Procedure:
    1. When a server receives a request, it awakens a thread from the pool.
    2. If any thread is available, the request is passed to it for service.
    3. Once the service is completed, the thread returns to the pool.
- Advantages:
    - Servicing a request with an existing thread is usually faster than waiting to create a thread.
    - The pool limits the no. of threads that exist at any one point.
- No. of threads in the pool can be based on actors such as
    - no. of CPUs
    - amount of memory and
    - expected no. of concurrent client-requests.

THREAD SPECIFIC DATA

- Threads belonging to a process share the data of the process.
- this sharing of data provides one of the benefits of multithreadedprogramming.
- In some circumstances, each thread might need its own copy of certain data. We will call such data **thread-specific data**.
- For example, in a transaction-processing system, we might service each transaction in a separatethread.

- Furthermore, each transaction may be assigned a unique identifier. To associate each thread with its unique identifier, we could use thread-specificdata.

SCHEDULER ACTIVATIONS

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to theapplication.
- Scheduler activations provide **upcalls**a communication mechanism from the kernel to the threadlibrary
- This communication allows an application to maintain the correct number kernel threads
- One scheme for communication between the user-thread library and the kernel is known as **scheduler activation.**

PROCESS SCHEDULING

Basic Concepts

- In a single-processor system,
  - Only one process may run at a time.
  - Other processes must wait until the CPU is rescheduled.
- Objective ofmultiprogramming:
  - To have some process running at all times, in order to maximize CPU utilization.

CPU-I/0 Burst Cycle

- Process execution consists of a cycleof
  - CPU execution and
  - I/O wait
- Process execution begins with a CPU burst, followed by an I/O burst, then another CPU burst, etc…
- Finally, a CPU burst ends with a request to terminateexecution.
- An I/O-bound program typically has many short CPUbursts.
- A CPU-bound program might have a few long CPU bursts.

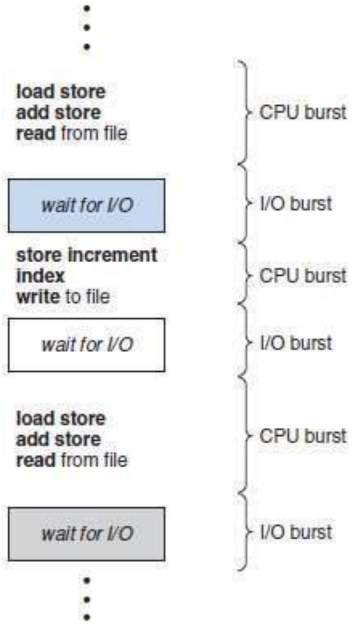


Fig Alternating sequence of CPU and I/O bursts

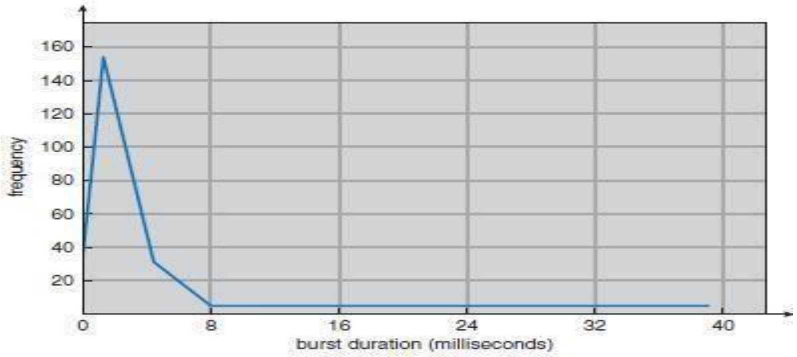


Fig: Histogram of CPU-burst durations

CPU Scheduler

- Thisscheduler
  - selects a waiting-process from the ready-queue and
  - allocates CPU to the waiting-process.
- The ready-queue could be a FIFO, priority queue, tree andlist.
- The records in the queues are generally process control blocks (PCBs) of theprocesses.

CPU Scheduling

- Four situations under which CPU scheduling decisions takeplace:
  1. When a process switches from the running state to the waiting state. For ex; I/O request.
  2. When a process switches from the running state to the ready state. For ex: when an interrupt occurs.
  3. When a process switches from the waiting state to the ready state. For ex: completion of I/O.
  4. When a process terminates.
- Scheduling under 1 and 4 is non- preemptive. Scheduling under 2 and 3 is preemptive.

Non Preemptive Scheduling

•Once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either

- by terminating or
- by switching to the waiting state.

Preemptive Scheduling

- This is driven by the idea of prioritizedcomputation.
- Processes that are runnable may be temporarilysuspended
- Disadvantages:
  1. Incurs a cost associated with access toshared-data.
  2. Affects the design of the OSkernel.

Dispatcher

- It gives control of the CPU to the process selected by the short-termscheduler.
- The functioninvolves:
  1. Switchingcontext
  2. Switching to user mode&
  3. Jumping to the proper location in the user program to restart that program
- It should be as fast as possible, since it is invoked during every process switch.
- ***Dispatch latency*** means the time taken by the dispatcherto
  - stop one process and
  - start another running.

SCHEDULING CRITERIA:

In choosing which algorithm to use in a particular situation, depends upon the properties of the various algorithms.Many criteria have been suggested for comparing CPU-scheduling algorithms. The criteria include the following:

1. **CPU utilization:** We want to keep the CPU as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system).
2. **Throughput:** If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes that are completed per time unit, called throughput. For long processes, this rate may be one process per hour; for short transactions, it may be ten processes per second.

3. **Turnaround time.** This is the important criterion which tells how long it takes to execute that process. The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/0.
4. **Waiting time:** The CPU-scheduling algorithm does not affect the amount of time during which a process executes or does I/0, it affects only the amount of time that a process spends waiting in the ready queue.Waiting time is the sum of the periods spent waiting in the ready queue.
5. **Response time:**In an interactive system, turnaround time may not be the best criterion. Often, a process can produce some output fairly early and can continue computing new results while previous results are being output to the user. Thus, another measure is the time from the submission of a request until the first response is produced. This measure, called response time, is the time it takes to start responding, not the time it takes to output the response. The turnaround time is generally limited by the speed of the output device.

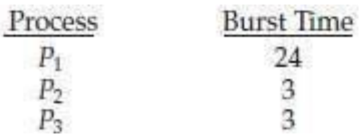
SCHEDULING ALGORITHMS

- CPU scheduling deals with the problem of deciding which of the processes in the ready-queue is to be allocated theCPU.
- Following are some schedulingalgorithms:
  1. FCFS scheduling (First Come FirstServed)
  2. Round Robin scheduling
  3. SJF scheduling (Shortest JobFirst)
  4. SRT scheduling
  5. Priority scheduling
  6. Multilevel Queue schedulingand
  7. Multilevel Feedback Queuescheduling

FCFS Scheduling

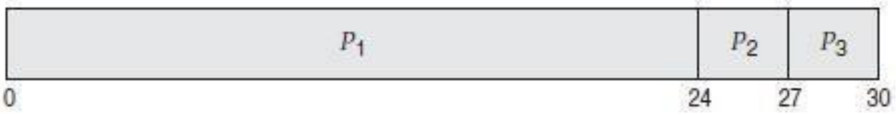
- The process that requests the CPU first is allocated the CPUfirst.
- The implementation is easily done using a FIFOqueue.
- Procedure:
  1. When a process enters the ready-queue, its PCB is linked onto the tail of thequeue.
  2. When the CPU is free, the CPU is allocated to the process at the queue'shead.
  3. The running process is then removed from the queue.

- Advantage:
  1. Code is simple to write & understand.
- Disadvantages:
  1. **Convoy effect:** All other processes wait for one big process to get off theCPU.
  2. Non-preemptive (a process keeps the CPU until it releasesit).
  3. Not good for time-sharingsystems.
  4. The average waiting time is generally notminimal.



Process	Burst Time
P_1	24
P_2	3
P_3	3

- Example: Suppose that the processes arrive in the order P1, P2,P3.
- The Gantt Chart for the schedule is asfollows:

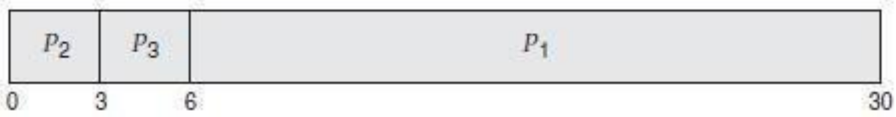


- Waiting time for *P1* = 0; *P2* = 24; *P3* =27

  Average waiting time: (0 + 24 + 27)/3 = 17ms

- Suppose that the processes arrive in the order P2, P3,P1.

- The Gantt chart for the schedule is asfollows:



- Waiting time for P1 = 6;P2 = 0; P3 =3
- Average waiting time: (6 + 0 + 3)/3 = 3ms

SJF Scheduling

- The CPU is assigned to the process that has the smallest next CPUburst.
- If two processes have the same length CPU burst, FCFS scheduling is used to break thetie.
- For long-term scheduling in a batch system, we can use the process time limit specified by the user, as the‘length’
- SJF can't be implemented at the level of short-term scheduling, because there is no way to know the length of the next CPUburst
- Advantage:

  1. The SJF is optimal, i.e. it gives the minimum average waiting time for a given set of processes.

- Disadvantage:

  1. Determining the length of the next CPU burst.

- SJF algorithm may be either 1) non-preemptive or 2)preemptive.

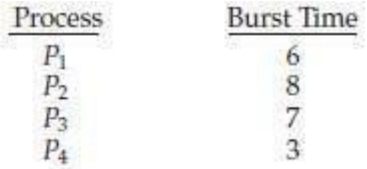
□ *1. Non preemptiveSJF*

The current process is allowed to finish its CPU burst.

□ *2. PreemptiveSJF*

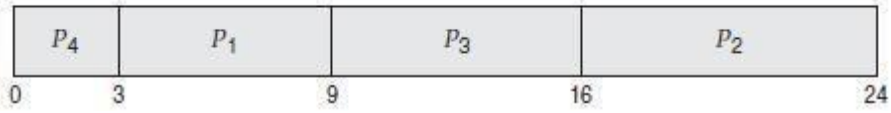
□ If the new process has a shorter next CPU burst than what is left of the executing process, that process is preempted. It is also known as **SRTF** scheduling (Shortest-Remaining-Time-First).

- Example (for non-preemptive SJF): Consider the following set of processes, with the length of the CPU-burst time given inmilliseconds.



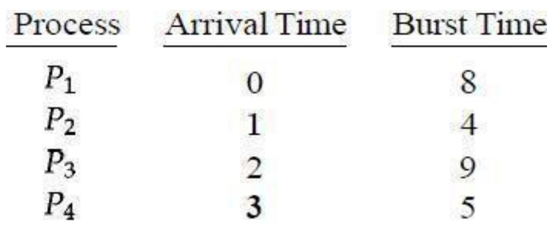
Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- For non-preemptive SJF, the Gantt Chart is asfollows:



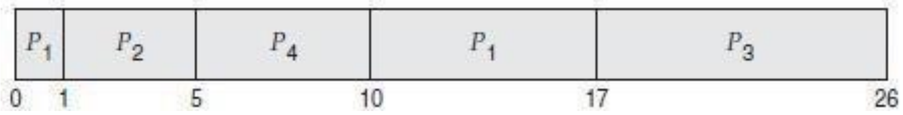
- Waiting time for P1 = 3; P2 = 16; P3 = 9; 4=0 Average waiting time: (3 + 16 + 9 + 0)/4= 7

preemptive SJF/SRTF: Consider the following set of processes, with the length of the CPU- burst time given inmilliseconds.



Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- For preemptive SJF, the Gantt Chart is asfollows:



- The average waiting time is ((10 - 1) + (1 - 1) + (17 - 2) + (5 - 3))/4 = 26/4 =6.5.

Priority Scheduling

- A priority is associated with eachprocess.
- The CPU is allocated to the process with the highestpriority.
- Equal-priority processes are scheduled in FCFSorder.
- Priorities can be defined either internally orexternally.
  1. *Internally-defined* priorities.
     - Use some measurable quantity to compute the priority of a process.
     - For example: time limits, memory requirements, no. f open files.
  2. *Externally-defined* priorities.
     - Set by criteria that are external to the OS For example:
     - importance of the process, political factors
- Priority scheduling can be either preemptive or non-preemptive.

  1.***Preemptive***

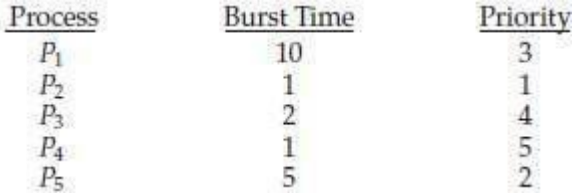
  The CPU is preempted if the priority of the newly arrived process is higher than the priority of the currently running process.

  2. *Non Preemptive*

  The new process is put at the head of the ready-queue

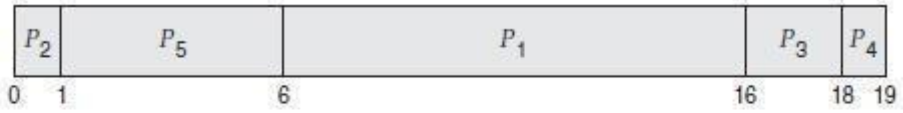
- Advantage:
  - Higher priority processes can be executed first.
- Disadvantage:
  - Indefinite blocking, where low-priority processes are left waiting indefinitely for CPU. Solution: ***Aging*** is a technique of increasing priority of processes that wait in system for a long time.

- Example: Consider the following set of processes, assumed to have arrived at time 0, in the order PI, P2, ..., P5, with the length of the CPU-burst time given inmilliseconds.



Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- The Gantt chart for the schedule is asfollows:

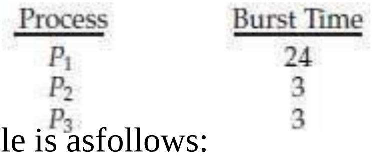


- The average waiting time is 8.2milliseconds.

Round Robin Scheduling

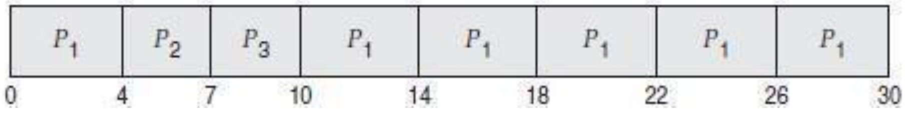
- Designed especially for timesharingsystems.
- It is similar to FCFS scheduling, but with preemption.
- A small unit of time is called a ***time quantum****(or timeslice).*
- Time quantum is ranges from 10 to 100ms.
- The ready-queue is treated as a **circularqueue**.
- The CPUscheduler
  - goes around the ready-queue and
  - allocates the CPU to each process for a time interval of up to 1 time quantum.
- To implement:

  The ready-queue is kept as a FIFO queue of processes
- CPUscheduler
  1. Picks the first process from theready-queue.
  2. Sets a timer to interrupt after 1 time quantumand
  3. Dispatches theprocess.
- One of two things will thenhappen.
  1. The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily.
  2. If the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the OS. The process will be put at the tail of the ready-queue.
- Advantage:
  - Higher average turnaround than SJF.
- Disadvantage:
  - Better response time than SJF.
- Example: Consider the following set of processes that arrive at time 0, with the length of the CPU-burst time given inmilliseconds.



Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart for the schedule is asfollows:



- The average waiting time is 17/3 = 5.66milliseconds.

- *The RR scheduling algorithm is preemptive.*

  No process is allocated the CPU for more than *1* time quantum in a row. If a process' CPU burst exceeds *1* time quantum, that process is preempted and is put back in the ready- queue.

- The performance of algorithm depends heavily on the size of the time quantum.
  1. If time quantum=very large, RR policy is the same as the FCFSpolicy.
  2. If time quantum=very small, RR approach appears to the users as though each of n processes has its own processor running at l/n the speed of the real processor.
- In software, we need to consider the effect of context switching on the performance of RR scheduling
  1. Larger the time quantum for a specific process time, less time is spend on context switching.
  2. The smaller the time quantum, more overhead is added for the purpose of context- switching.

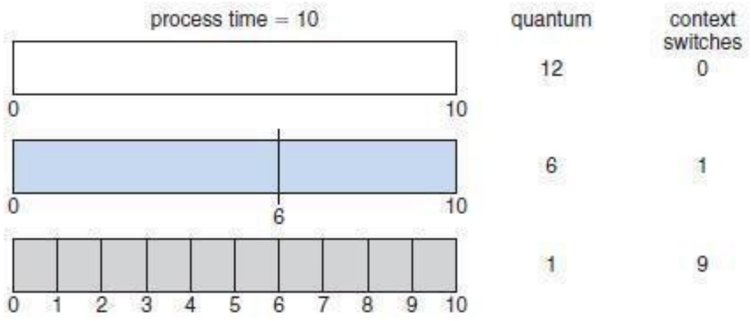
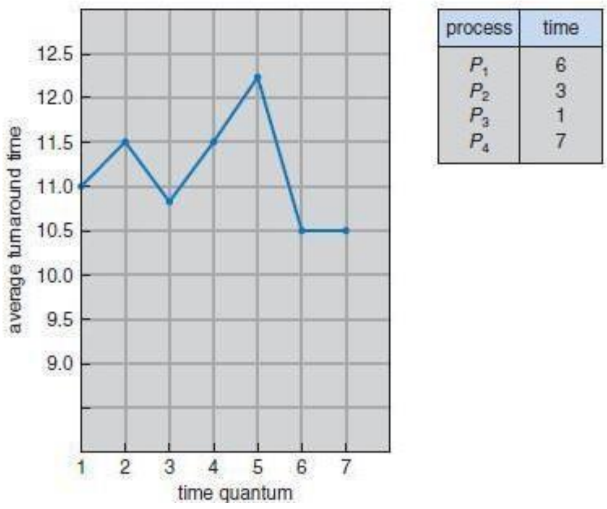


Fig: How a smaller time quantum increases context switches



process	time
P_1	6
P_2	3
P_3	1
P_4	7

Fig: How turnaround time varies with the time quantum

Multilevel Queue Scheduling

- Useful for situations in which processes are easily classified into different groups.
- For example, a common division is made between
  - foreground (or interactive) processes and
  - background (or batch) processes.
- The ready-queue is partitioned into several separate queues (Figure2.19).
- The processes are permanently assigned to one queue based on some property like
  - memory size
  - process priority or
  - process type.
- Each queue has its own scheduling algorithm.

  For example, separate queues might be used for foreground and background processes.

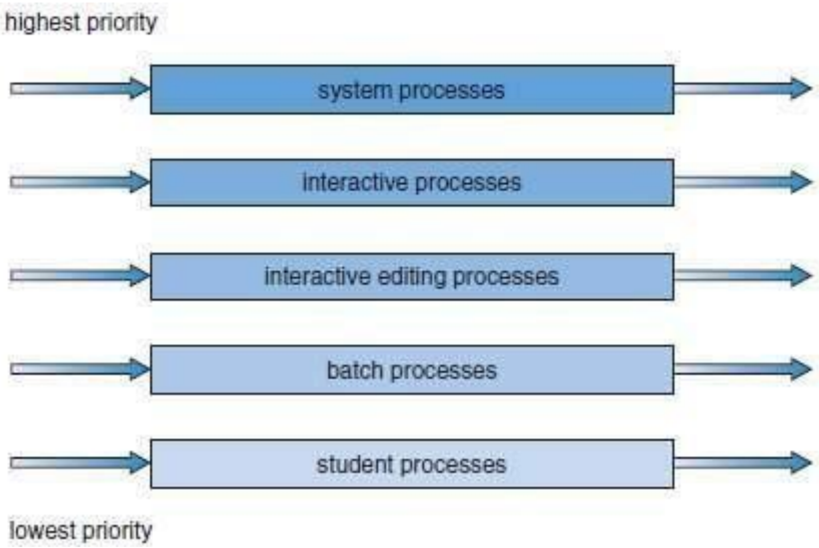
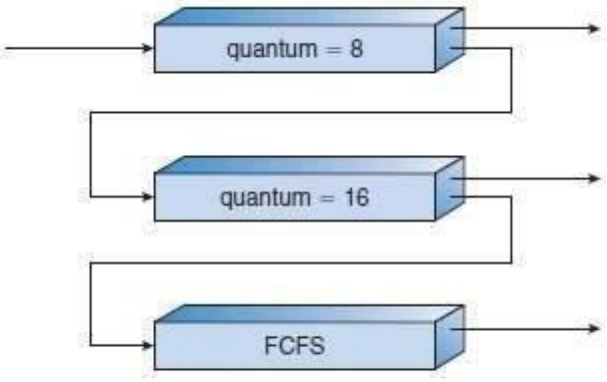


Fig Multilevel queue scheduling

- There must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling.
- For example, the foreground queue may have absolute priority over the background queue.
- **Time slice**: each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR 20% to backgroundin FCFS

Multilevel Feedback Queue Scheduling

- A process may move between queues
- The basic idea: Separate processes according to the features of their CPU bursts. Forexample
    1. If a process uses too much CPU time, it will be moved to a lower-priority queue. This scheme leaves I/O-bound and interactive processes in the higher-priority queues.
    2. If a process waits too long in a lower-priority queue, it may be moved to a higher-priority queue This form of aging prevents starvation.



.

Figure 2.20 Multilevel feedback queues

In general, a multilevel feedback queue scheduler is defined by the followingparameters:

1. The number ofqueues.
2. The scheduling algorithm for eachqueue.
3. The method used to determine when to upgrade a process to a higher priorityqueue.
4. The method used to determine when to demote a process to a lower priorityqueue.
5. The method used to determine which queue a process will enter when that process needs service

MULTIPLE PROCESSOR SCHEDULING

- If multiple CPUs are available, the scheduling problem becomes morecomplex.
- Twoapproaches:

AsymmetricMultiprocessing

The basic idea is:

- A master server is a single processor responsible for all scheduling decisions, I/O processing and other systemactivities.
- The other processors execute only user code.
- Advantage: This is simple because only one processor accesses the system data structures, reducing the need for data sharing.

Symmetric Multiprocessing

The basic idea is:

- Each processor is self-scheduling.
- To do scheduling, the scheduler for eachprocessor
- Examines the ready-queue and
- Selects a process to execute.

Restriction: We must ensure that two processors do not choose the same process and that processes are not lost from the queue.

Processor Affinity

- In SMP systems,
  1. Migration of processes from one processor to another are avoided and
  2. Instead processes are kept running on same processor. This is known as processor affinity.
- Two forms:

  1. ***SoftAffinity***
     - When an OS try to keep a process on one processor because of policy, but cannot guarantee it will happen.
     - It is possible for a process to migrate between processors.
  2. ***Hard Affinity***
     - When an OS have the ability to allow a process to specify that it is not to migrate to other processors. Eg: Solaris OS

Load Balancing

- This attempts to keep the workload evenly distributed across all processors in an SMPsystem.
- Twoapproaches:

  1. ***PushMigration***

     A specific task periodically checks the load on each processor and if it finds an imbalance, it evenly distributes the load to idle processors.
  2. ***PullMigration***

     An idle processor pulls a waiting task from a busy processor.

Symmetric Multithreading

- The basic idea:
  1. Create multiple logical processors on the same physical processor.
  2. Present a view of several logical processors to the OS.
- Each logical processor has its own architecture state, which includes general-purpose and machine-state registers.
- Each logical processor is responsible for its own interrupt handling.
- SMT is a feature provided in hardware, notsoftware.

THREAD SCHEDULING

- On OSs, it is kernel-level threads but not processes that are being scheduled by theOS.
- User-level threads are managed by a thread library, and the kernel is unaware ofthem.
- To run on a CPU, user-level threads must be mapped to an associated kernel-levelthread.

Contention Scope

- Twoapproaches:

  1. Process-Contention scope

  - On systems implementing the many-to-one and many-to-many models, the thread library schedules user-level threads to run on an available LWP.
  - Competition for the CPU takes place among threads belonging to the sameprocess.

  2. System-Contentionscope

  - The process of deciding which kernel thread to schedule on theCPU.
  - Competition for the CPU takes place among all threads in thesystem.
  - Systems using the one-to-one model schedule threads using onlySCS.

Pthread Scheduling

- Pthread API that allows specifying either PCS or SCS during threadcreation.
- Pthreads identifies the following contention scopevalues:
  1. PTHREAD_SCOPEJPROCESS schedules threads using PCSscheduling.
  2. PTHREAD-SCOPE_SYSTEM schedules threads using SCSscheduling.
- Pthread IPC provides following two functions for getting and setting the contention scopepolicy:
  1. pthread_attr_setscope(pthread_attr_t *attr, intscope)
  2. pthread_attr_getscope(pthread_attr_t *attr, int*scop)

MODULE-3
PROCESS SYNCHRONIZATION

- A cooperating process is one that can affect or be affected by other processes executing in the system. Cooperating processes can either directly share a logical address space (that is, both code and data) or be allowed to share data only through files or messages.
- Concurrent access to shared data may result in data inconsistency. To maintain data consistency, various mechanisms is required to ensure the orderly execution of cooperating processes that share a logical address space.

Producer- Consumer Problem

- A Producer process produces information that is consumed by consumer process.
- To allow producer and consumer process to run concurrently, A Bounded Buffer can be used where the items are filled in a buffer by the producer and emptied by the consumer.
- The original solution allowed at most **BUFFER_SIZE - 1** item in the buffer at the same time. To overcome this deficiency, an integer variable ***counter***, initialized to 0 isadded.
- ***counter*** is incremented every time when a new item is added to the buffer and is decremented every time when one item removed from thebuffer.

The code for the ***producer process*** can be modified as follows:

```
while (true) {

        /* produce an item and put in nextProduced*/ while
              (counter == BUFFER_SIZE)
                             ; // do nothing
                         buffer [in] = nextProduced;
                         in = (in + 1) % BUFFER_SIZE;
                         counter++;
}
```

The code for the ***consumer process*** can be modified as follows:

```
while (true){
                  while (counter ==0)
                          ; // donothing
                         nextConsumed =buffer[out];
                          out = (out + 1) % BUFFER_SIZE;
                        counter--;
                            /* consume the item in nextConsumed */
          }
```

- **Race Condition**

  When the producer and consumer routines shown above are correct separately, they may not function correctly when executed concurrently.

- ***Illustration:***

  Suppose that the value of the variable ***counter*** is currently 5 and that the producer and consumer processes execute the statements "counter++" and "counter--" concurrently. The value of the variable counter may be 4, 5, or 6 but the only correct result is counter == 5, which is generated correctly if the producer and consumer execute separately.

The value of ***counter*** may be incorrect as shown below:

The statement counter++ could be implemented as

```
register1= counter
register1 = register1 + 1
counter =register1
```

The statement counter-- could be implemented as

```
register2 =counter
register2 = register2 – 1
count = register2
```

The concurrent execution of "counter++" and "counter--" is equivalent to a sequential execution in which the lower-level statements presented previously are interleaved in some arbitrary order. One such interleaving is

Consider this execution interleaving with “count = 5” initially:

S0: producer execute register1=counter	{register1 = 5}
S1: producer execute register1 = register1+1	{register1 = 6}
S2: consumer execute register2=counter	{register2 = 5}
S3: consumer execute register2 = register2-1	{register2 = 4}
S4: producer execute counter=register1	{count =6}
S5: consumer execute counter=register2	{count =4}

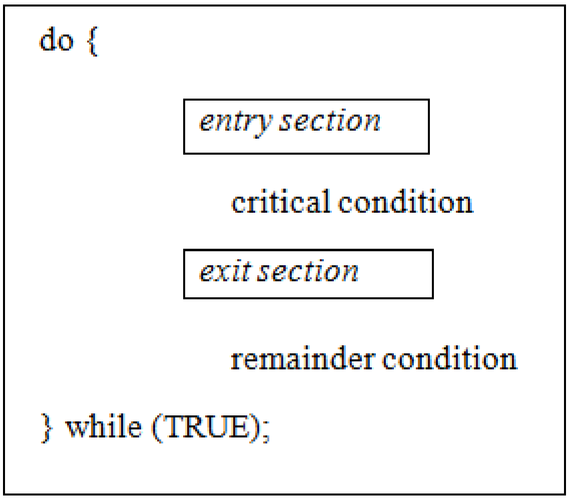
- ***Note:*** It is arrived at the incorrect state "counter == 4", indicating that four buffers are full, when, in fact, five buffers are full. If we reversed the order of the statements at T4 and T5, we would arrive at the incorrect state "counter==6".
- *Definition* ***Race Condition***: A situation where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **Race Condition.**
- To guard against the race condition, ensure that only one process at a time can be manipulating the variable counter. To make such a guarantee, ***the processes are synchronized*** in some way.

The Critical Section Problems

- Consider a system consisting of n processes {Po, P1 , ... ,Pn-1}.
- Each process has a segment of code, called a **critical section** in which the process may be changing common variables, updating a table, writing a file, and soon
- The important feature of the system is that, when one process is executing in its critical section, no other process is to be allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the sametime.
- The critical-section problem is to design a protocol that the processes can use to cooperate.

The general structure of a typical process Pi is shown in below figure.

- Each process must request permission to enter its critical section. The section of code implementing this request is the **entry section**.
- The critical section may be followed by an **exit section**. The remaining code is the **reminder section**.



```
do {
            entry section
                critical condition
            exit section
                remainder condition
} while (TRUE);
```

Figure: General structure of a typical process Pi

A solution to the critical-section problem must satisfy the following **three requirements**:

1. **Mutual exclusion:** If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.

2. **Progress:** If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.

3. **Bounded waiting:** There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

PETERSON'S SOLUTION

- This is a classic software-based solution to the critical-section problem. There are no guarantees that Peterson's solution will work correctly on modern computer architectures
- Peterson's solution provides a good algorithmic description of solving the critical-section problem and illustrates some of the complexities involved in designing software that addresses the requirements of mutual exclusion, progress, and bounded waiting.

Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. The processes are numbered P_o and P_1 or Pi and P_j where j = 1-i

Peterson's solution requires the two processes to share two data items:

```
int         turn;
boolean flag[2];
```

- **turn:** The variable turn indicates whose turn it is to enter its critical section. **Ex:** if turn == i, then process P_i is allowed to execute in its critical section
- **flag:** The flag array is used to indicate if a process is ready to enter its critical section. **Ex:** if flag [i] is true, this value indicates that P_i is ready to enter its critical section.

```
do {
            flag[i] = TRUE;
            turn = j;
            while (flag[j] && turn == j)
                        ; // do nothing
             critical section
            flag[i] = FALSE;

                        remainder section

    } while (TRUE);
```

Figure: The structure of process P_i in Peterson's solution

- To enter the critical section, process Pi first sets flag [i] to be true and then sets turn to the value j, thereby asserting that if the other process wishes to enter the critical section, it can do so.
- If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time. Only one of these assignments will last, the other will occur but will be over written immediately.

- The eventual value of turn determines which of the two processes is allowed to enter its critical section first

To prove that solution is correct, then we need to show that

1. Mutual exclusion is preserved
2. Progress requirement is satisfied
3. Bounded-waiting requirement is met

1. To prove Mutual exclusion

- Each pi enters its critical section only if either flag [j] == false or turn ==i.
- If both processes can be executing in their critical sections at the same time, then flag [0] == flag [1]==true.
- These two observations imply that Pi and Pj could not have successfully executed their while statements at about the same time, since the value of turn can be either 0 or 1 but cannot be both. Hence, one of the processes (Pj) must have successfully executed the while statement, whereas Pi had to execute at least one additional statement ("turn==j").
- However, at that time, flag [j] == true and turn == j, and this condition will persist as long as Pi is in its critical section, as a result, mutual exclusion is preserved.

2. To prove Progress and Bounded-waiting

- A process Pi can be prevented from entering the critical section only if it is stuck in the while loop with the condition flag [j] ==true and turn=== j; this loop is the only one possible.
- If Pj is not ready to enter the critical section, then flag [j] ==false, and Pi can enter its critical section.
- If Pj has set flag [j] = true and is also executing in its while statement, then either turn === i or turn ===j.
    - If turn == i, then Pi will enter the critical section.
    - If turn== j, then Pj will enter the critical section.
- However, once Pj exits its critical section, it will reset flag [j] = false, allowing Pi to enter its critical section.
- If Pj resets flag [j] to true, it must also set turn to i.
- Thus, since Pi does not change the value of the variable turn while executing the while statement, Pi will enter the critical section (progress) after at most one entry by Pj (bounded waiting).

SYNCHRONIZATION HARDWARE

- The solution to the critical-section problem requires a simple tool-a **lock.**
- Race conditions are prevented by requiring that critical regions be protected by locks. That is, a process must acquire a lock before entering a critical section and it releases the lock when it exits the critical section

```
do {
              acquire lock
                     critical section
              release lock
                     remainder section
       } while (TRUE);
```

Figure: Solution to the critical-section problem using locks.

- The critical-section problem could be solved simply in a uniprocessor environment if interrupts are prevented from occurring while a shared variable was being modified. In this manner, the current sequence of instructions would be allowed to execute in order without preemption. No other instructions would be run, so no unexpected modifications could be made to the shared variable.
- But this solution is not as feasible in a multiprocessor environment. Disabling interrupts on a multiprocessor can be time consuming, as the message is passed to all the processors. This message passing delays entry into each critical section, and system efficiency decreases.

TestAndSet ( ) and Swap( ) instructions

- Many modern computer systems provide special hardware instructions that allowto **test** and **modify** the content of a word or to **swap** the contents of two words **atomically**, that is, as one uninterruptible unit.
- Special instructions such as TestAndSet () and Swap() instructions are used to solve the critical-section problem
- The TestAndSet () instruction can be defined as shown in Figure. The important characteristic of this instruction is that it is executed atomically.

Definition:

```
booleanTestAndSet (boolean *target)
 {
     booleanrv = *target;
     *target = TRUE;
     return rv:
 }
```

Figure: The definition of the TestAndSet () instruction.

- Thus, if two TestAndSet () instructions are executed simultaneously, they will be executed sequentially in some arbitrary order. If the machine supports the TestAndSet () instruction, then implementation of mutual exclusion can be done by declaring a Boolean variable lock, initialized to false.

```
do {
                while ( TestAndSet (&lock ))
                        ;  // do nothing
                      //    critical section
                lock =FALSE;
                       //      remaindersection
        } while (TRUE);
```

Figure: Mutual-exclusion implementation with TestAndSet ()

- The Swap() instruction, operates on the contents of two words, it is defined as shown below

Definition:

```
void Swap (boolean *a, boolean *b)
        {
              boolean temp = *a;
              *a = *b;
              *b = temp:
        }
```

Figure: The definition of the Swap ( ) instruction

- Swap() it is executed atomically. If the machine supports the Swap() instruction, then mutual exclusion can be provided as follows.
- A global Boolean variable lock is declared and is initialized to false. In addition, each process has a local Boolean variable key. The structure of process Pi is shown in below

```
do {
                key = TRUE;
                while ( key == TRUE) Swap
                       (&lock, &key );

                          //    critical section
                 lock =FALSE;
                         //      remaindersection
         } while (TRUE);
```

Figure: Mutual-exclusion implementation with the Swap() instruction

- These algorithms satisfy the mutual-exclusion requirement, they do not satisfy the bounded- waiting requirement.
- Below algorithm using the TestAndSet () instruction that satisfies all the critical-section requirements. The common data structures are

```
boolean waiting[n];
boolean lock;
```

These data structures are initialized to false.

```
do {
        waiting[i] = TRUE;
        key = TRUE;
        while (waiting[i] && key)
            key = TestAndSet(&lock);
        waiting[i] = FALSE;

            // critical section j

        = (i + 1) % n;
        while ((j != i) && !waiting[j])
            j = (j + 1) % n;
        if (j == i)
            lock = FALSE;
        else
            waiting[j] = FALSE;
            // remainder section
    } while (TRUE);
```

Figure: Bounded-waiting mutual exclusion with TestAndSet ()

1. To prove the mutual exclusionrequirement

- Note that process Pi can enter its critical section only if either waiting [i] == false or key ==false.
- The value of key can become false only if the TestAndSet( ) isexecuted.
- The first process to execute the TestAndSet( ) will find key== false; all others must wait.
- The variable waiting[i] can become false only if another process leaves its critical section; only one waiting[i] is set to false, maintaining the mutual-exclusion requirement.

2. To prove the progress requirement

Note that, the arguments presented for mutual exclusion also apply here, since a process exiting the critical section either sets lock to false or sets waiting[j] to false. Both allow a process that is waiting to enter its critical section to proceed.

3. To prove the bounded-waiting requirement

- Note that, when a process leaves its critical section, it scans the array waiting in the cyclic ordering (i + 1, i + 2, ... , n 1, 0, ... , i 1).
- It designates the first process in this ordering that is in the entry section (waiting[j] ==true) as the next one to enter the critical section. Any process waiting to enter its critical section will thus do so within n - 1 turns.

SEMAPHORE

- A semaphore is a synchronization tool is used solve various synchronization problem and can be implementedefficiently.
- Semaphore do not require busywaiting.
- A semaphore S is an integer variable that, is accessed only through two standard atomic operations: wait () and signal (). The wait () operation was originally termed P and signal() was calledV.

Definition of wait ():

```
wait (S) {
          while S <= 0
           ; // no-op
    S--;
```

Definition of signal ():

```
signal (S) {
        S++;}
```

- All modifications to the integer value of the semaphore in the wait () and signal() operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.

Binary semaphore

- The value of a binary semaphore can range only between 0 and1.
- Binary semaphores are known as mutex locks, as they are locks that provide mutual exclusion. Binary semaphores to deal with the critical-section problem for multiple processes. Then processes share a semaphore, mutex, initialized to1

Each process Pi is organized as shown in below figure

```
do {
        wait (mutex);
                   // Critical Section
          signal (mutex);
               // remainder section
} while (TRUE);
```

Figure: Mutual-exclusion implementation with semaphores

Counting semaphore

- The value of a counting semaphore can range over an unrestricted domain.
- Counting semaphores can be used to control access to a given resource consisting of a finite number of instances.
- The semaphore is initialized to the number of resources available. Each process that wishes to use a resource performs a wait() operation on the semaphore. When a process releases a resource, it performs a signal()operation.
- When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

Implementation

- The main disadvantage of the semaphore definition requires busywaiting.
- While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code.
- This continual looping is clearly a problem in a real multiprogramming system, where a single CPU is shared among many processes.

- Busy waiting wastes CPU cycles that some other process might be able to use productively. This type of semaphore is also called a **spinlock** because the process "spins" while waiting for thelock.

Semaphore implementation with no busy waiting

- The definition of the wait() and signal() semaphore operations ismodified.
- When a process executes the wait () operation and finds that the semaphore value is not positive, it mustwait.
- However, rather than engaging in busy waiting, the process can block itself. The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then control is transferred to the CPU scheduler, which selects another process toexecute.
- A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal() operation. The process is restarted by a wakeup( ) operation, which changes the process from the waiting state to the ready state. The process is then placed in the readyqueue.

- To implement semaphores under this definition, we define a semaphore as a "C' struct:

```
typedefstruct {
                    int value;
                    struct process *list;
             } semaphore;
```

- Each semaphore has an integer value and a list of processes list. When a process must wait on a semaphore, it is added to the list of processes. A signal() operation removes one process from the list of waiting processes and awakens that process.

- The wait() semaphore operation can now be defined as:

```
wait(semaphore *S) {
 S->value--;
 if (S->value < 0) {
        add this process to S-
        >list; block();
 }}
```

- The signal () semaphore operation can now be defined as

```
signal(semaphore *S) {
            S->value++;
            if (S->value <= 0) {
                  remove a process P
                  from S->list;
                  wakeup(P);
            }
      }
```

- The block() operation suspends the process that invokes it. The wakeup(P) operation resumes the execution of a blocked process P. These two operations are provided by the operating system as basic systemcalls.
- In this implementation semaphore values may be negative. If a semaphore value is negative, its magnitude is the number of processes waiting on thatsemaphore.

Deadlocks and Starvation

- The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. The event in question is the execution of a signal( ) operation. When such a state is reached, these processes are said to be deadlocked.
- To illustrate this, consider a system consisting of two processes, Po and P1, each accessing two semaphores, S and Q, set to the value 1

```
  P0                    P1
wait(S);             wait(Q);
wait(Q);             wait(S);
    .                   .
    .                   .
 signal(S);          signal(Q);
 signal(Q);          signal(S);
```

- Suppose that Po executes wait (S) and then P1 executes wait (Q). When Po executes wait (Q), it must wait until P1 executes signal (Q). Similarly, when P1 executes wait (S), it must wait until Po executes signal(S). Since these signal() operations cam1ot be executed, Po and P1 are deadlocked.

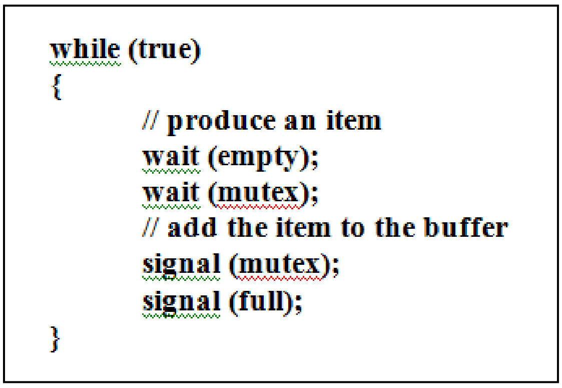
- Another problem related to deadlocks is indefinite blocking or starvation: A situation in which processes wait indefinitely within the semaphore.
- Indefinite blocking may occur if we remove processes from the list associated with a semaphore in LIFO (last-in, first-out) order.

CLASSICAL PROBLEMS OF SYNCHRONIZATION

- Bounded-BufferProblem
- Readers and WritersProblem
- Dining-PhilosophersProblem

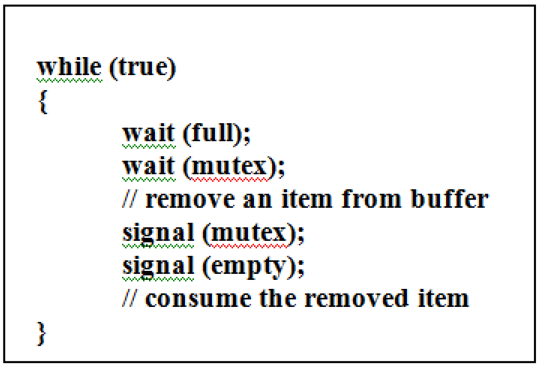
Bounded-Buffer Problem

- N buffers, each can hold one item
- Semaphore **mutex**initialized to the value 1
- Semaphore **full** initialized to the value0
- Semaphore empty initialized to the value N.



```
while (true)
{
        // produce an item
        wait (empty);
        wait (mutex);
        // add the item to the buffer
        signal (mutex);
        signal (full);
}
```

The structure of the producer process:

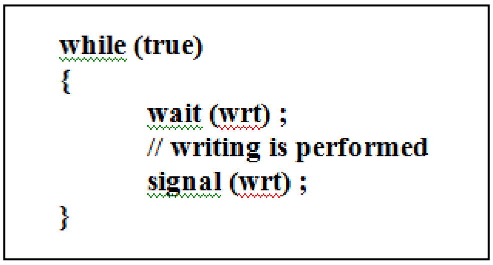


```
while (true)
{
        wait (full);
        wait (mutex);
        // remove an item from buffer
        signal (mutex);
        signal (empty);
        // consume the removed item
}
```

The structure of the consumerprocess:

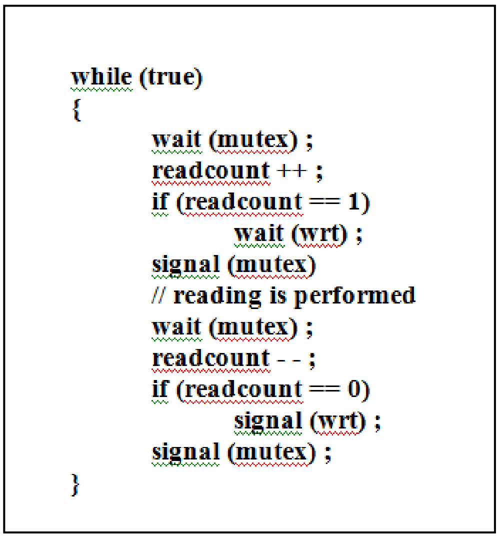
Readers-Writers Problem

- A data set is shared among a number of concurrentprocesses
- Readers – only read the data set; they do **not** perform anyupdates
- Writers – can both read andwrite.
- Problem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the sametime.
  - SharedData
  - Dataset
  - Semaphore **mutex**initialized to 1.
  - Semaphore **wrt**initialized to1.
  - Integer **readcount**initialized to 0.



```
while (true)
{
        wait (wrt) ;
        // writing is performed
        signal (wrt) ;
}
```

The structure of a writerprocess

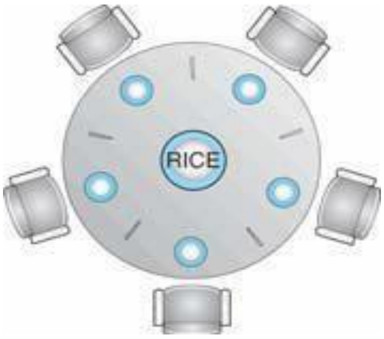


```
while (true)
{
        wait (mutex) ;
        readcount ++ ;
        if (readcount == 1)
                wait (wrt) ;
        signal (mutex)
        // reading is performed
        wait (mutex) ;
        readcount - - ;
        if (readcount == 0)
                signal (wrt) ;
        signal (mutex) ;
}
```

The structure of a readerprocess

Dining-Philosophers Problem

Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five singlechopsticks.



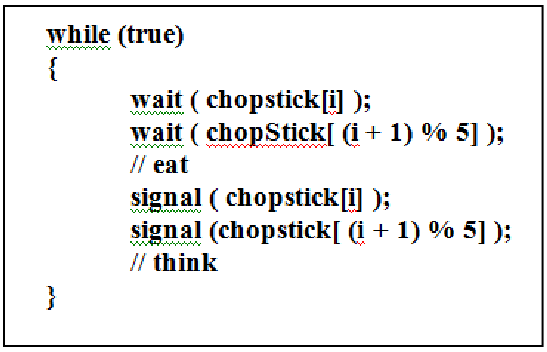
A philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up only one chopstick at a time. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing the chopsticks. When she is finished eating, she puts down both chopsticks and starts thinkingagain.

It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-freemanner.

Solution: One simple solution is to represent each chopstick with a semaphore. A philosopher tries to grab a chopstick by executing a wait() operation on thatsemaphore. She releases her chopsticks by executing the signal() operation on the appropriate semaphores. Thus, the shared data are

semaphore chopstick[5];

where all the elements of chopstick are initialized to 1. The structure of philosopher *i*is shown



```
while (true)
{
        wait ( chopstick[i] );
        wait ( chopStick[ (i + 1) % 5] );
        // eat
        signal ( chopstick[i] );
        signal (chopstick[ (i + 1) % 5] );
        // think
}
```

Several possible remedies to the deadlock problem are replaced by:

- Allow at most four philosophers to be sitting simultaneously at thetable.

- Allowaphilosophertopickupherchopsticksonlyifbothchopsticksareavailable.
- Use an asymmetric solution—that is, an odd-numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an even numbered philosopher picks up her right chopstick and then her leftchopstick.

Problems with Semaphores

Correct use of semaphore operations:

- signal (mutex) …. wait (mutex) : Replace signal with wait andvice-versa
- wait (mutex) … wait(mutex)
- Omitting of wait (mutex) or signal (mutex) (orboth)

MODULE 3

DEADLOCKS

A process requests resources, if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a**Deadlock.**

SYSTEM MODEL

- A system consists of a finite number of resources to be distributed among a number of competing processes. The resources are partitioned into several types, each consisting of some number of identical instances. Memory space, CPU cycles, files, and I/0 devices are examples of resource types.
- A process must request a resource before using it and must release the resource after using it. A process may request as many resources as it requires carrying out its designated task. The number of resources requested may not exceed the total number of resources available in the system.

Under the normal mode of operation, a process may utilize a resource in only the following sequence:

1. **Request:** The process requests the resource. If the request cannot be granted immediately, then the requesting process must wait until it can acquire the resource.
2. **Use:** The process can operate on the resource.
3. **Release:** The process releases the resource.

A set of processes is in a deadlocked state when every process in the set is waiting for an event that can be caused only by another process in the set. The events with which we are mainly concerned here are resource acquisition and release. The resources may be either physical resources or logical resources

To illustrate a deadlocked state, consider a system with three CD RW drives.

Suppose each of three processes holds one of these CD RW drives. If each process now requests another drive, the three processes will be in a deadlocked state.

Each is waiting for the event "CD RW is released," which can be caused only by one of the other waiting processes. This example illustrates a deadlock involving the same resource type.

Deadlocks may also involve different resource types. For example, consider a system with one printer and one DVD drive. Suppose that process Pi is holding the DVD and process P_j is holding the printer. If P_i requests the printer and P_j requests the DVD drive, a deadlock occurs.

DEADLOCK CHARACTERIZATION

Necessary Conditions

A deadlock situation can arise if the following four conditions hold simultaneously in a system:

1. **Mutual exclusion:** At least one resource must be held in a non-sharable mode, that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

2. **Hold and wait:** A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.

3. **No preemption:** Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.

4. **Circular wait:** A set $\{P_0, P_l, \ldots, P_n\}$ of waiting processes must exist such that P_o is waiting for a resource held by P_1, P_1 is waiting for a resource held by P_2, ... , P_{n-1} is waiting for a resource held by P_n and P_n is waiting for a resource held by P_o.

Resource-Allocation Graph

Deadlocks can be described in terms of a directed graph called **System Resource-Allocation Graph**

The graph consists of a set of vertices V and a set of edges E. The set of vertices V is partitioned into two different types of nodes:

- $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the active processes in the system.
- $R = \{R_1, R_2, \ldots, R_m\}$ the set consisting of all resource types in the system.

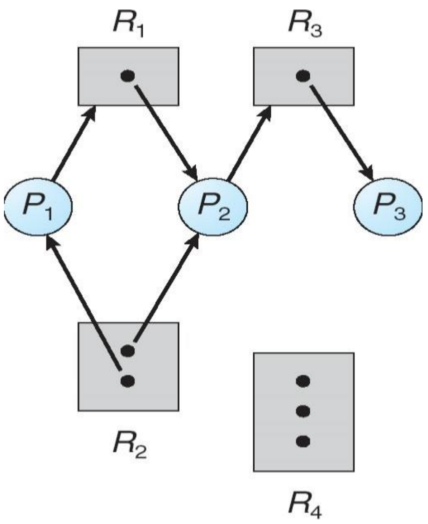
A directed edge from process P_i to resource type R_j is denoted by $P_i \rightarrow R_j$ it signifies that process P_i has requested an instance of resource type R_j and is currently waiting for that resource.
A directed edge from resource type R_j to process P_i is denoted by $R_j \rightarrow P_i$ it signifies that an instance of resource type R_j has been allocated to process $P_{i.}$

- A directed edge $P_i \rightarrow R_j$ is called a Request Edge.
- A directed edge $R_j \rightarrow P_i$ is called an Assignment Edge.

Pictorially each process P_i as a circle and each resource type R_j as a rectangle. Since resource type R_j may have more than one instance, each instance is represented as a dot within the rectangle.
A request edge points to only the rectangle R_j, whereas an assignment edge must also designate one of the dots in the rectangle.

When process P_i requests an instance of resource type $R_{j,}$ a request edge is inserted in the resource-allocation graph. When this request can be fulfilled, the request edge is *instantaneously* transformed to an assignment edge. When the process no longer needs access to the resource, it releases the resource; as a result, the assignment edge is deleted.

The resource-allocation graph shown in Figure depicts the following situation.



The sets P, R and E:

- $P = \{P_1, P_2, P_3\}$
- $R= \{R_1, R_2, R_3, R_4\}$
- $E = \{P_l \rightarrow R_l, P_2 \rightarrow R_3, R_l \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3 \}$

Resource instances:

- One instance of resource type R_1
- Two instances of resource type R_2
- One instance of resource type R_3
- Three instances of resource type R_4

Process states:

- Process P_1 is holding an instance of resource type R_2 and is waiting for an instance of resource type R_1.
- Process P_2 is holding an instance of R_1 and an instance of R_2 and is waiting for an instance of R_3.
- Process P_3 is holding an instance of R_3.

<u>If the graph does contain a cycle, then a deadlock may exist.</u>

- If each resource type has exactly **<u>one instance</u>**, then a cycle implies that a deadlock has occurred. If the cycle involves only a set of resource types, each of which has only a single instance, then a deadlock has occurred. Each process involved in the cycle is deadlocked.
- If each resource type has **<u>several instances</u>**, then a cycle does not necessarily imply that a deadlock has occurred. In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.

To illustrate this concept, the resource-allocation graph depicted in below figure:
Suppose that process P3 requests an instance of resource type R2. Since no resource instance is currently available, a request edge P3 → R2 is added to the graph. At this point, two minimal cycles exist in the system:

1. P1 →R1 → P2 → R3 → P3 → R2→P1
2. P2 →R3 → P3 → R2 → P2

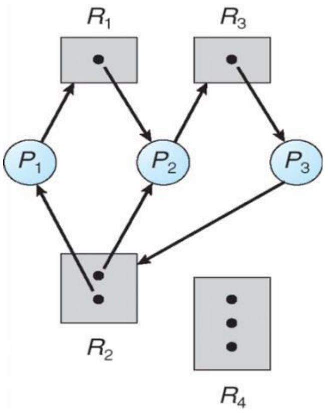


Figure: Resource-allocation graph with a deadlock.

Processes P1, P2, and P3 are deadlocked. Process P2 is waiting for the resource R3, which is held by process P3. Process P3 is waiting for either process P1 or process P2 to release resourceR2. In addition, process P1 is waiting for process P2 to release resource R1.

Consider the resource-allocation graph in below Figure. In this example also have a cycle:
P1→R1→P3→R2→P1

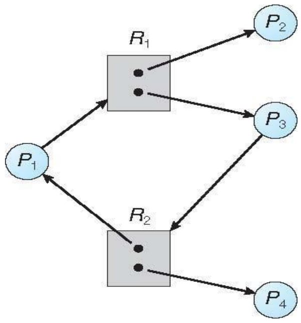


Figure: Resource-allocation graph with a cycle but no deadlock

However, there is no deadlock. Observe that process P4 may release its instance of resource type R2. That resource can then be allocated to P3, breaking the cycle.

METHODS FOR HANDLING DEADLOCKS

The deadlock problem can be handled in one of three ways:

1. Use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlocked state.
2. Allow the system to enter a deadlocked state, detect it, and recover.
3. Ignore the problem altogether and pretend that deadlocks never occur in the system.

To ensure that deadlocks never occur, the system can use either deadlock prevention or a deadlock-avoidance scheme.

Deadlock prevention provides a set of methods for ensuring that at least one of the necessary conditions cannot hold. These methods prevent deadlocks by constraining how requests for resources can be made.

Deadlock-avoidance requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime. With this additional knowledge, it can decide for each request whether or not the process should wait. To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process

If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may arise. In this environment, the system can provide an **algorithm** that examines the state of the system to determine whether a deadlock has occurred and an algorithm to recover from the deadlock.

In the absence of algorithms to detect and recover from deadlocks, then the system is in a deadlock state yet has no way of recognizing what has happened. In this case, the undetected deadlock will result in deterioration of the system's performance, because resources are being held by processes that cannot run and because more and more processes, as they make requests for resources, will enter a deadlocked state. Eventually, the system will stop functioning and will need to be restarted manually.

DEADLOACK PREVENTION

Deadlock can be prevented by ensuring that at least one of the four necessary conditions cannot hold.

Mutual Exclusion

- The mutual-exclusion condition must hold for non-sharable resources. Sharable resources, do not require mutually exclusive access and thus cannot be involved in a deadlock.
- Ex: Read-only files are example of a sharable resource. If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file. A process never needs to wait for a sharable resource.
- Deadlocks cannot prevent by denying the mutual-exclusion condition, because some resources are intrinsically non-sharable.

Hold and Wait

To ensure that the hold-and-wait condition never occurs in the system, then guarantee that, whenever a process requests a resource, it does not hold any other resources.

- One protocol that can be used requires each process to request and be allocated all its resources before it begins execution.
- Another protocol allows a process to request resources only when it has none. A process may request some resources and use them. Before it can request any additional resources, it must release all the resources that it is currently allocated.

Ex:

- Consider a process that copies data from a DVD drive to a file on disk, sorts the file, and then prints the results to a printer. If all resources must be requested at the beginning of the process, then the process must initially request the DVD drive, disk file, and printer. It will hold the printer for its entire execution, even though it needs the printer only at the end.
- The second method allows the process to request initially only the DVD drive and disk file. It copies from the DVD drive to the disk and then releases both the DVD drive and the disk file. The process must then again request the disk file and the printer. After copying the disk file to the printer, it releases these two resources and terminates.

The two main disadvantages of these protocols:

1. Resource utilization may be low, since resources may be allocated but unused for a long period.
2. Starvation is possible.

No Preemption

The third necessary condition for deadlocks is that there be no preemption of resources that have already been allocated.

To ensure that this condition does not hold, the following protocols can be used:

- If a process is holding some resources and requests another resource that cannot be immediately allocated to it, then all resources the process is currently holding are preempted.
- The preempted resources are added to the list of resources for which the process is waiting. The process will be restarted only when it can regain its old resources, as wellas the new ones that it is requesting.

If a process requests some resources, first check whether they are available. If they are, allocate them.
If they are not available, check whether they are allocated to some other process that is waiting for additional resources. If so, preempt the desired resources from the waiting process and allocate them to the requesting process.
If the resources are neither available nor held by a waiting process, the requesting process must wait. While it is waiting, some of its resources may be preempted, but only if another process requests them.
A process can be restarted only when it is allocated the new resources it is requesting and recovers any resources that were preempted while it was waiting.

Circular Wait

One way to ensure that this condition never holds is to impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration.

To illustrate, let R = {R1, R2, ... , Rm} be the set of resource types. Assign a unique integer number to each resource type, which allows to compare two resources and to determinewhether one precedes another in ordering. Formally, it defined as a one-to-one function
F: R ->N, where N is the set of natural numbers.

Example: if the set of resource types R includes tape drives, disk drives, and printers, then the function F might be defined as follows:

F (tape drive) = 1
F (disk drive) = 5
F (printer) = 12

Now consider the following protocol to prevent deadlocks. Each process can request resources only in an increasing order of enumeration. That is, a process can initially request any number of instances of a resource type -R_i. After that, the process can request instances of resource typeR_j if and only if $F(R_j) > F(R_i)$.

DEADLOCK AVOIDANCE

- To avoid deadlocks an additional information is required about how resources are to be requested. With the knowledge of the complete sequence of requests and releases for each process, the system can decide for each request whether or not the process should wait in order to avoid a possible future deadlock
- Each request requires that in making this decision the system consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.
- The various algorithms that use this approach differ in the amount and type of information required. The simplest model requires that each process declare the *maximum number* of resources of each type that it may need. Given this a priori information, it is possible to construct an algorithm that ensures that the system will never enter a deadlocked state. Such an algorithm defines the ***deadlock-avoidance approach.***

Safe State

- **Safe state:** A state is safe if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. A system is in a safe state only if there exists a safe sequence.

- **Safe sequence:** A sequence of processes <P1, P2, ... , Pn> is a safe sequence for the current allocation state if, for each Pi, the resource requests that Pi can still make can be satisfied by the currently available resources plus the resources held by all Pj, with j <i.

In this situation, if the resources that Pi needs are not immediately available, then Pi can wait until all Pj have finished. When they have finished, Pi can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate. When Pi terminates, Pi+1 can obtain its needed resources, and so on. If no such sequence exists, then the system state is said to be unsafe.

A safe state is not a deadlocked state. Conversely, a deadlocked state is an unsafe state. Not all unsafe states are deadlocks as shown in figure. An unsafe state may lead to a deadlock. As long as the state is safe, the operating system can avoid unsafe states

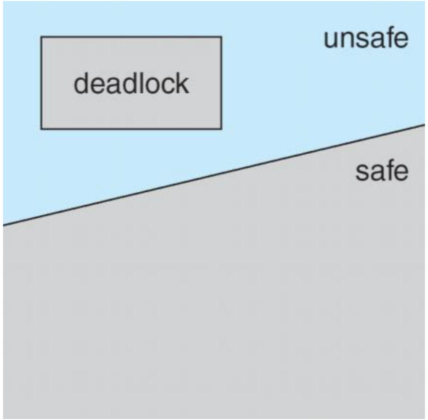


Figure: Safe, unsafe, and deadlocked state spaces.

Resource-Allocation-Graph Algorithm

- If a resource-allocation system has only one instance of each resource type, then a variant of the resource-allocation graph is used for deadlock avoidance.
- In addition to the request and assignment edges, a new type of edge is introduced, called a claim edge.
- A claim edge *Pi* ->*Rj* indicates that process *Pi* may request resource *Rj* at some time in the future. This edge resembles a request edge in direction but is represented in the graph by a **dashed line**.
- When process *Pi* requests resource *Rj*, the claim edge *Pi* ->*Rj* is converted to a request edge. When a resource *Rj* is released by *Pi* the assignment edge *Rj*->*Pi* is reconverted to a claim edge *Pi*->*Rj*.

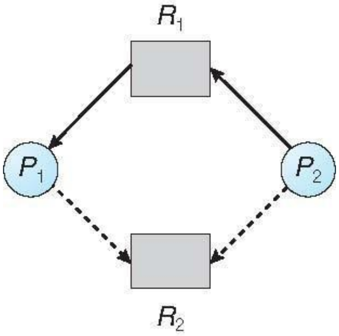


Figure: Resource-allocation graph for deadlock avoidance.

Note that the resources must be claimed a priori in the system. That is, before process *Pi* starts executing, all its claim edges must already appear in the resource-allocation graph.
We can relax this condition by allowing a claim edge *Pi* ->*Rj* to be added to the graph only if all the edges associated with process *Pi* are claim edges.

Now suppose that process *Pi* requests resource *Rj*. The request can be granted only if converting the request edge *Pi* ->*Rj* to an assignment edge *Rj*->*Pi* does not result in the formation of a cycle in the resource-allocation graph.

There is need to check for safety by using a cycle-detection algorithm. An algorithm for detecting a cycle in this graph requires an order of n^2 operations, where n is the number of processes in the system.

- If no cycle exists, then the allocation of the resource will leave the system in a safe state.
- If a cycle is found, then the allocation will put the system in an unsafe state. In that case, process Pi will have to wait for its requests to be satisfied.

To illustrate this algorithm, consider the resource-allocation graph as shown above. Suppose that P2 requests R2. Although R2 is currently free, we cannot allocate it to P2, since this action will create a cycle in the graph.

A cycle, indicates that the system is in an unsafe state. If P1 requests R2, and P2 requests R1, then a deadlock will occur.

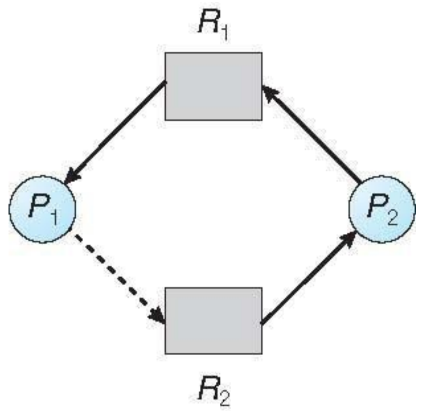


Figure: An unsafe state in a resource-allocation graph

Banker's Algorithm

The Banker’s algorithm is applicable to a resource allocation system with multiple instances of each resource type.

- When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system.
- When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.

To implement the banker's algorithm the following data structures are used.

Let n = number of processes, and m = number of resources types

Available: A vector of length *m* indicates the number of available resources of each type. If available [j] = k, there are k instances of resource type Rj available.

Max: An *n x m* matrix defines the maximum demand of each process. If Max [i,j] = k, then process Pi may request at most k instances of resource type Rj

Allocation: An *n x m* matrix defines the number of resources of each type currently allocated to each process. If Allocation[i,j] = k then Pi is currently allocated k instances of Rj

Need: An n x m matrix indicates the remaining resource need of each process. If Need[i,j] = k, then Pi may need k more instances of Rj to complete its task.

Need [i,j] = Max[i,j] – Allocation [i,j]

Safety Algorithm

The algorithm for finding out whether or not a system is in a safe state. This algorithm can be described as follows:

```
1. Let Work and Finish be vectors of length m and n, respectively. Initialize:
        Work = Available
        Finish [i] = false for i = 0, 1,…,n- 1

2. Find an index i such that both:
        (a) Finish[i] = false
        (b) $Need_i \le Work$
    If no such i exists, go to step 4

3.  $Work = Work + Allocation_i$
        Finish[i] = true
        go to step 2

4. If Finish [i] == true for all i, then the system is in a safe state
```

This algorithm may require an order of m x n^2 operations to determine whether a state is safe.

Resource-Request Algorithm

The algorithm for determining whether requests can be safely granted.
Let Request$_i$ be the request vector for process P$_i$. If Request$_i$ [j] == k, then process P$_i$ wants k instances of resource type R$_j$. When a request for resources is made by process Pi, the following actions are taken:

1. If $Request_i \leq Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim

2. If $Request_i \leq Available$, go to step 3. Otherwise P_i must wait, since resources are not available

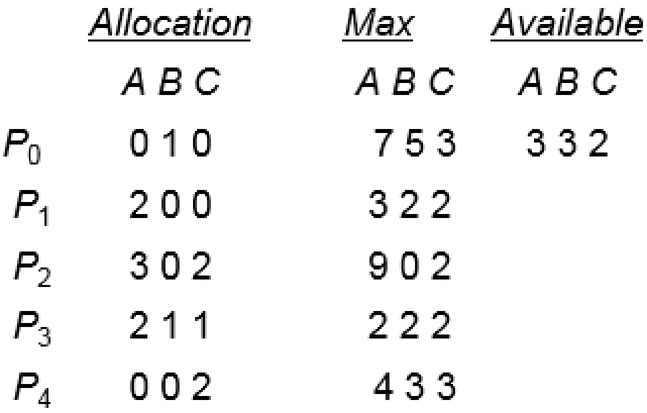
3. Have the system pretend to allocate requested resources to P_i by modifying the state as follows:

$$Available = Available - Request;$$
$$Allocation_i = Allocation_i + Request_i;$$
$$Need_i = Need_i - Request_i;$$

If safe ⇒ the resources are allocated to Pi
If unsafe ⇒ Pi must wait, and the old resource-allocation state is restored

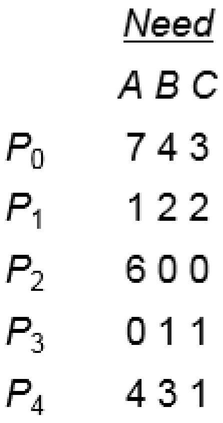
Example

Consider a system with five processes P_o through P_4 and three resource types *A*, *B*, and C. Resource type *A* has ten instances, resource type *B* has five instances, and resource type C has seven instances. Suppose that, at time $T_0$the following snapshot of the system has been taken:



	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

The content of the matrix *Need* is defined to be *Max* - *Allocation*



	Need A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

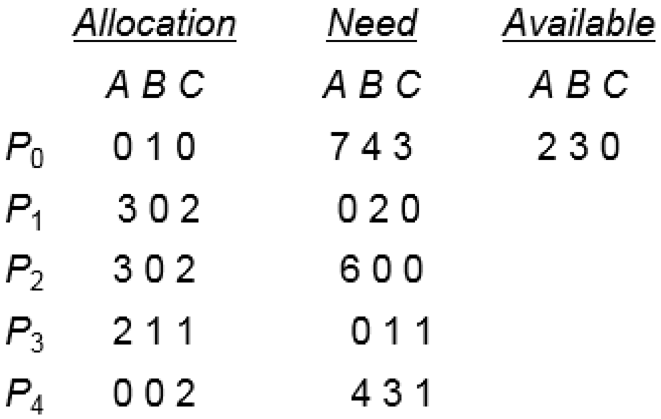
The system is currently in a safe state. Indeed, the sequence <P_1, P_3, P_4, P_2, P_0> satisfies the safety criteria.

Suppose now that process P_1 requests one additional instance of resource type A and two instances of resource type C, so $\text{Request}_1 = (1,0,2)$. Decide whether this request can be immediately granted.

Check that Request ≤ Available

$(1,0,2) \leq (3,3,2) \Rightarrow$ true

Then pretend that this request has been fulfilled, and the following new state is arrived.



	Allocation A B C	Need A B C	Available A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

Executing safety algorithm shows that sequence <P_1, P_3, P_4, P_0, P_2> satisfies safety requirement.

DEADLOCK DETECTION

If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur. In this environment, the system may provide:

- An algorithm that examines the state of the system to determine whether a deadlock has occurred
- An algorithm to recover from the deadlock

Single Instance of Each Resource Type

- If all resources have only a single instance, then define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a ***wait-for*** graph**.**
- This graph is obtained from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.
- An edge from P_i to P_j in a wait-for graph implies that process P_i is waiting for process P_j to release a resource that P_i needs. An edge $P_i \rightarrow P_j$ exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges $P_i \rightarrow R_q$ and $R_q \rightarrow P_i$ for some resource R_q.

Example: In below Figure, a resource-allocation graph and the corresponding wait-for graph is presented.

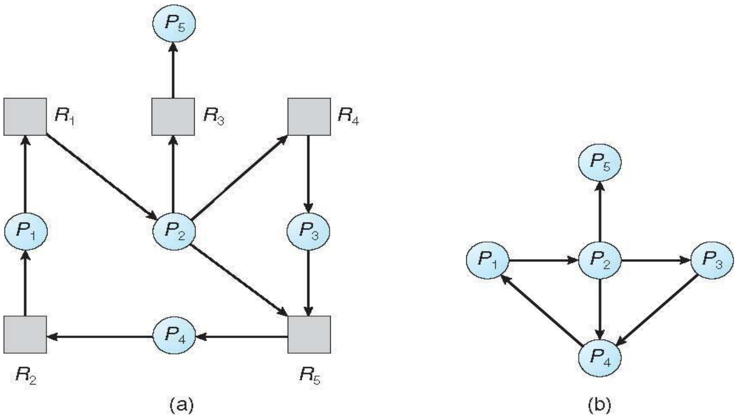


Figure: (a) Resource-allocation graph. (b) Corresponding wait-for graph.

- A deadlock exists in the system if and only if the wait-for graph contains a cycle. To detect deadlocks, the system needs to *maintain* the wait-for graph and periodically *invoke an algorithm* that searches for a cycle in the graph.
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph.

Several Instances of a Resource Type

A deadlock detection algorithm that is applicable to several instances of a resource type. The algorithm employs several time-varying data structures that are similar to those used in the banker's algorithm.

- **Available**: A vector of length *m* indicates the number of available resources of each type.
- **Allocation:** Ann *x m* matrix defines the number of resources of each type currently allocated to each process.
- **Request:** An *n* x *m* matrix indicates the current request of each process. If *Request[i][j]* equals *k*, then process *P;* is requesting *k* more instances of resource type *Rj.*

Algorithm:

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively Initialize:

   (a) Work = Available

   (b) For *i* = 1,2, …, *n*, if $Allocation_i \neq 0$, then *Finish*[i] = false;
   otherwise, *Finish*[i] = *true*

2. Find an index *i*such that both:

   (a) Finish[*i*] == *false*

   (b)$Request_i \leq Work$

   If no such *i* exists, go to step 4

3. $Work = Work + Allocation_i$

   Finish[*i*] = *true*

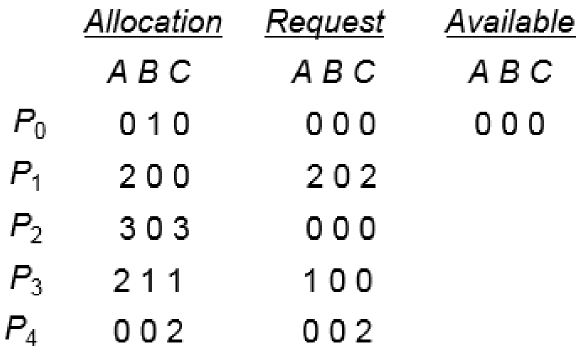
   go to step 2

4. If *Finish*[*i*] == false, for some *i*, $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if *Finish*[*i*] == *false*, then P_i is deadlocked

Algorithm requires an order of O(*m* x *n*$^{2)}$ operations to detect whether the system is in deadlocked state

Example of Detection Algorithm

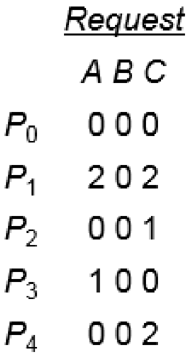
Consider a system with five processes *Po* through *P4* and three resource types *A, B,* and C. Resource type *A* has seven instances, resource type *B* has two instances, and resource type C has six instances. Suppose that, at time T_0, the following resource-allocation state:



	Allocation	*Request*	*Available*
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

After executing the algorithm, Sequence <P_0, P_2, P_3, P_1, P_4> will result in *Finish*[*i*] = true for all *i*

Suppose now that process P2 makes one additional request for an instance of type C. The Request matrix is modified as follows:



	Request
	A B C
P_0	0 0 0
P_1	2 0 2
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

The system is now deadlocked. Although we can reclaim the resources held by process Po, the number of available resources is not sufficient to fulfill the requests of the other processes. Thus, a deadlock exists, consisting of processes P1, P2, P3, and P4.

Detection-Algorithm Usage

The detection algorithm can be invoked on two factors:

1. How *often* is a deadlock likely to occur?
2. How *many* processes will be affected by deadlock when it happens?

If deadlocks occur frequently, then the detection algorithm should be invoked frequently. Resources allocated to deadlocked processes will be idle until the deadlock can be broken.

If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.

RECOVERY FROM DEADLOCK

The system recovers from the deadlock automatically. There are two options for breaking a deadlock one is simply to abort one or more processes to break the circular wait. The other is to preempt some resources from one or more of the deadlocked processes.

Process Termination

To eliminate deadlocks by aborting a process, use one of two methods. In both methods, the system reclaims all resources allocated to the terminated processes.

1. **Abort all deadlocked processes:** This method clearly will break the deadlock cycle, but at great expense; the deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and probably will have to be recomputed later.
2. **Abort one process at a time until the deadlock cycle is eliminated:** This method incurs considerable overhead, since after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

If the partial termination method is used, then we must determine which deadlocked process (or processes) should be terminated. Many factors may affect which process is chosen, including:

1. What the priority of the process is
2. How long the process has computed and how much longer the process will compute before completing its designated task
3. How many and what types of resources the process has used.
4. How many more resources the process needs in order to complete
5. How many processes will need to be terminated?
6. Whether the process is interactive or batch

Resource Preemption

To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken. If preemption is required to deal with deadlocks, then three issues need to be addressed:

1. **Selecting a victim.** Which resources and which processes are to be preempted? As in process termination, we must determine the order of preemption to minimize cost. Cost factors may include such parameters as the number of resources a deadlocked process is holding and the amount of time the process has thus far consumed during its execution.
2. **Rollback.** If we preempt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some needed resource. We must roll back the process to some safe state and restart it from that state. Since it is difficult to determine what a safe state is, the simplest solution is a total rollback: abort the process and then restart it.

3. **Starvation.** How do we ensure that starvation will not occur? That is, how can we guarantee that resources will not always be preempted from the same process?

MODULE-4
MEMORY MANAGEMENT

Main Memory Management Strategies

- Every program to be executed has to be executed must be in memory. The instruction must be fetched from memory before it is executed.
- In multi-tasking OS memory management is complex, because as processes are swapped in and out of the CPU, their code and data must be swapped in and out of memory.

Basic Hardware

- Main memory, cache and CPU registers in the processors are the only storage spaces that CPU can access directly.
- The program and data must be bought into the memory from the disk, for the process to run. Each process has a separate memory space and must access only this range of legal addresses. Protection of memory is required to ensure correct operation. This prevention is provided by hardware implementation.
- Two registers are used - a base register and a limit register. The base register holds the smallest legal physical memory address; the limit register specifies the size of the range.
- For example, the base register holds the smallest legal physical memory address; the limit register specifies the size of the range. For example, if the base register holds 300040 and limit register is 120900, then the program can legally access all addresses from 300040 through 420940 (inclusive).

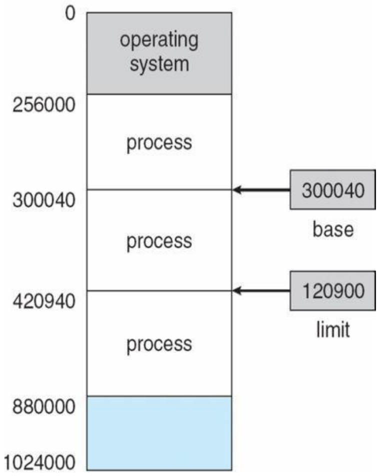


Figure: A base and a limit-register define a logical-address space

- The base and limit registers can be loaded only by the operating system, which uses a special privileged instruction. Since privileged instructions can be executed only in kernel mode only the operating system can load the base and limit registers.

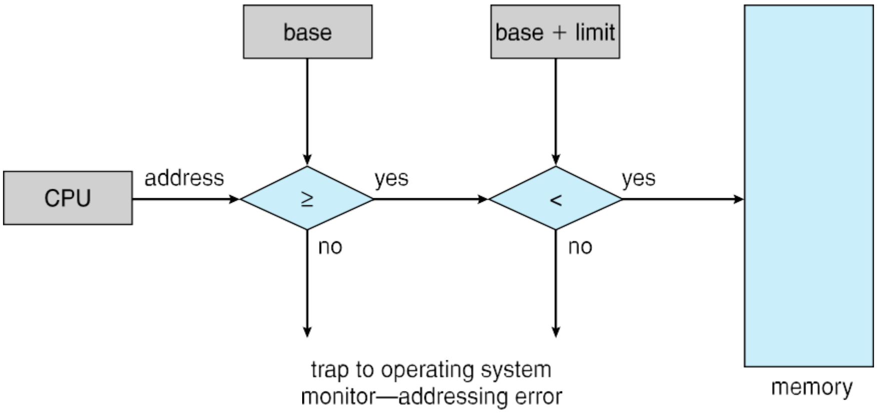


Figure: Hardware address protection with base and limit-registers

Address Binding

- User programs typically refer to memory addresses with symbolic names. These symbolic names must be mapped or bound to physical memory addresses.
- Address binding of instructions to memory-addresses can happen at 3 different stages.

1. **Compile Time** - If it is known at compile time where a program will reside in physical memory, then absolute code can be generated by the compiler, containing actual physical addresses. However, if the load address changes at some later time, then the program will have to be recompiled.
2. **Load Time** - If the location at which a program will be loaded is not known at compile time, then the compiler must generate relocatable code, which references addresses relative to the start of the program. If that starting address changes, then the program must be reloaded but not recompiled.
3. **Execution Time** - If a program can be moved around in memory during the course of its execution, then binding must be delayed until execution time.

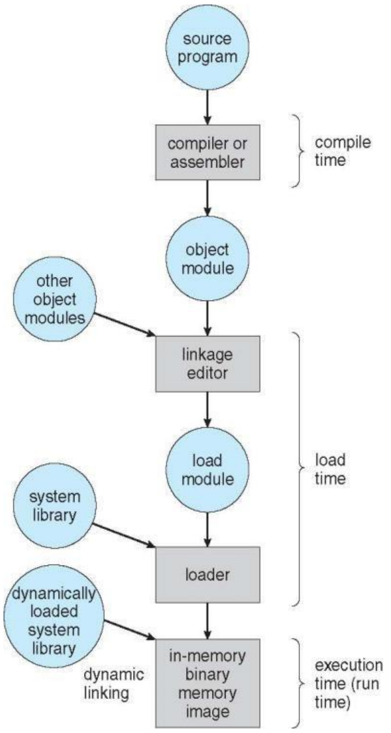


Figure: Multistep processing of a user program

Logical Versus Physical Address Space

- The address generated by the CPU is a logical address, whereas the memory address where programs are actually stored is a physical address.
- The set of all logical addresses used by a program composes the logical address space, and the set of all corresponding physical addresses composes the physical address space.
- The run time mapping of logical to physical addresses is handled by the memory-management unit (MMU).
    - One of the simplest is a modification of the base-register scheme.
    - The base register is termed a relocation register
    - The value in the relocation-register is added to every address generated by a user-process at the time it is sent to memory.
    - The user-program deals with logical-addresses; it never sees the real physical-addresses.

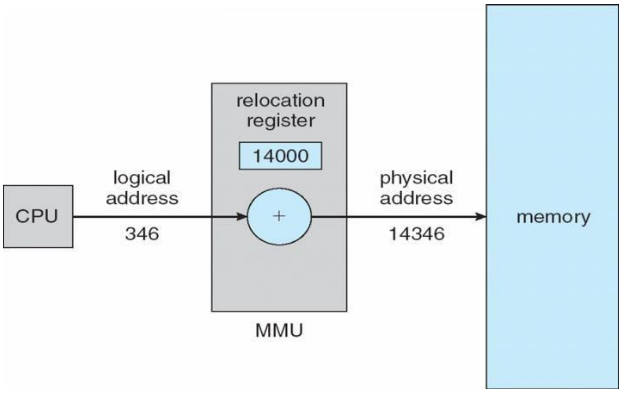


Figure: Dynamic relocation using a relocation-register

Dynamic Loading

- This can be used to obtain better memory-space utilization.
- A routine is not loaded until it is called.

This works as follows:

1. Initially, all routines are kept on disk in a relocatable-load format.
2. Firstly, the main-program is loaded into memory and is executed.
3. When a main-program calls the routine, the main-program first checks to see whether the routine has been loaded.
4. If routine has been not yet loaded, the loader is called to load desired routine into memory.
5. Finally, control is passed to the newly loaded-routine.

Advantages:

1. An unused routine is never loaded.
2. Useful when large amounts of code are needed to handle infrequently occurring cases.
3. Although the total program-size may be large, the portion that is used (and hence loaded) may be much smaller.
4. Does not require special support from the OS.

Dynamic Linking and Shared Libraries

- With **static linking** library modules get fully included in executable modules, wasting both disk space and main memory usage, because every program that included a certain routine from the library would have to have their own copy of that routine linked into their executable code.
- With **dynamic linking**, however, only a stub is linked into the executable module, containing references to the actual library module linked in at run time.
  - The stub is a small piece of code used to locate the appropriate memory-resident library-routine.
  - This method saves disk space, because the library routines do not need to be fully included in the executable modules, only the stubs.
  - An added benefit of dynamically linked libraries (DLLs, also known as shared libraries or shared objects on UNIX systems) involves easy upgrades and updates.

Shared libraries

- A library may be replaced by a new version, and all programs that reference the library will automatically use the new one.
- Version info. is included in both program & library so that programs won't accidentally execute incompatible versions.

Swapping

- A process must be loaded into memory in order to execute.
- If there is not enough memory available to keep all running processes in memory at the same time, then some processes that are not currently using the CPU may have their memory swapped out to a fast local disk called the ***backing store***.
- *Swapping is the process of moving a process from memory to backing store and moving another process from backing store to memory*. Swapping is a very slow process compared to other operations.
- A variant of swapping policy is used for priority-based scheduling algorithms. If a higher-priority process arrives and wants service, the memory manager can swap out the lower-priority process and then load and execute the higher-priority process. When the higher-priority process finishes, the lower-priority process can be swapped back in and continued. This variant of swapping is ***called roll out, roll in.***

Swapping depends upon address-binding:

- If binding is done at load-time, then process cannot be easily moved to a different location.
- If binding is done at execution-time, then a process can be swapped into a different memory-space, because the physical-addresses are computed during execution-time.

Major part of swap-time is transfer-time; i.e. total transfer-time is directly proportional to the amount of memory swapped.

Disadvantages:

1. Context-switch time is fairly high.
2. If we want to swap a process, we must be sure that it is completely idle.
   Two solutions:
   i) Never swap a process with pending I/O.
   ii) Execute I/O operations only into OS buffers.

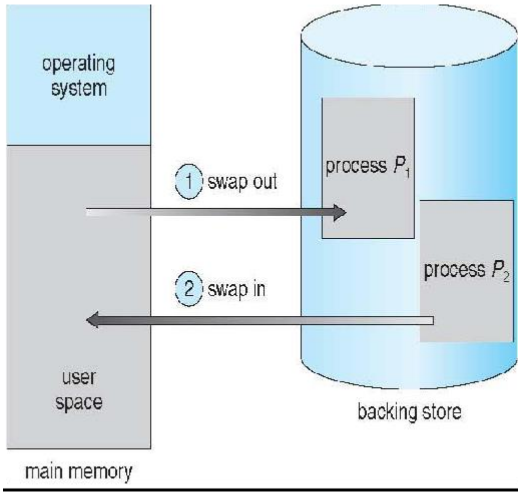


Figure: Swapping of two processes using a disk as a backing store

Example:

Assume that the user process is 10 MB in size and the backing store is a standard hard disk with a transfer rate of 40 MB per second.

The actual transfer of the 10-MB process to or from main memory takes

10000 KB/40000 KB per second = 1/4 second

= 250 milliseconds.

Assuming that no head seeks are necessary, and assuming an average latency of 8 milliseconds, the swap time is 258 milliseconds. Since we must both swap out and swap in, the total swap time is about 516 milliseconds.

Contiguous Memory Allocation

- The main memory must accommodate both the operating system and the various user processes. Therefore we need to allocate the parts of the main memory in the most efficient way possible.
- Memory is usually divided into 2 partitions: One for the resident OS. One for the user processes.
- Each process is contained in a single contiguous section of memory.

1. Memory Mapping and Protection

- Memory-protection means protecting OS from user-process and protecting user-processes from one another.
- Memory-protection is done using
  - Relocation-register: contains the value of the smallest physical-address.
  - Limit-register: contains the range of logical-addresses.
- Each logical-address must be less than the limit-register.
- The MMU maps the logical-address dynamically by adding the value in the relocation-register. This mapped-address is sent to memory
- When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit-registers with the correct values.
- Because every address generated by the CPU is checked against these registers, we can protect the OS from the running-process.
- The relocation-register scheme provides an effective way to allow the OS size to change dynamically.
- Transient OS code: Code that comes & goes as needed to save memory-space and overhead for unnecessary swapping.

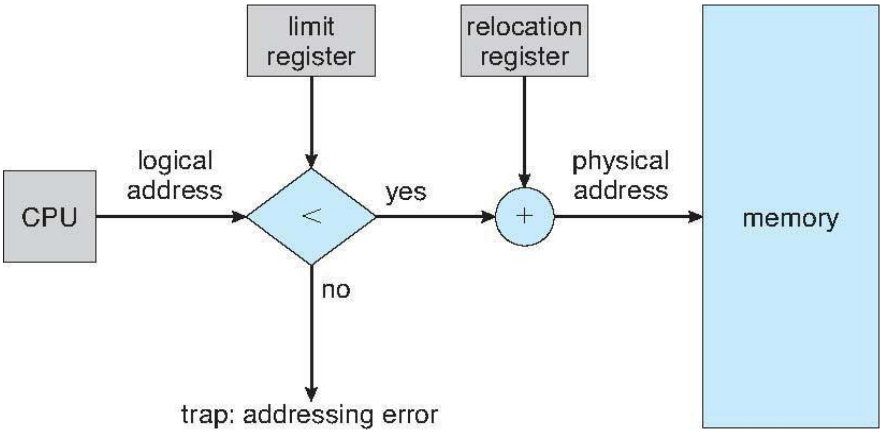


Figure: Hardware support for relocation and limit-registers

2. Memory Allocation

Two types of memory partitioning are:

1. Fixed-sized partitioning
2. Variable-sized partitioning

1. Fixed-sized Partitioning

- The memory is divided into fixed-sized partitions.
- Each partition may contain exactly one process.
- The degree of multiprogramming is bound by the number of partitions.
- When a partition is free, a process is selected from the input queue and loaded into the free partition.
- When the process terminates, the partition becomes available for another process.

2. Variable-sized Partitioning

- The OS keeps a table indicating which parts of memory are available and which parts are occupied.
- A hole is a block of available memory. Normally, memory contains a set of holes of various sizes.
- Initially, all memory is available for user-processes and considered one large hole.
- When a process arrives, the process is allocated memory from a large hole.
- If we find the hole, we allocate only as much memory as is needed and keep the remaining memory available to satisfy future requests.

Three strategies used to select a free hole from the set of available holes:

1. First Fit: Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended.

2. Best Fit: Allocate the smallest hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.

3. Worst Fit: Allocate the largest hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole.

First-fit and best fit are better than worst fit in terms of decreasing time and storage utilization.

3. Fragmentation

Two types of memory fragmentation:

1. Internal fragmentation
2. External fragmentation

1. Internal Fragmentation

- The general approach is to break the physical-memory into fixed-sized blocks and allocate memory in units based on block size.
- The allocated-memory to a process may be slightly larger than the requested-memory.
- The difference between requested-memory and allocated-memory is called internal fragmentation i.e. Unused memory that is internal to a partition.

2. External Fragmentation

- External fragmentation occurs when there is enough total memory-space to satisfy a request but the available-spaces are not contiguous. (i.e. storage is fragmented into a large number of small holes).
- Both the first-fit and best-fit strategies for memory-allocation suffer from external fragmentation.
- Statistical analysis of first-fit reveals that given N allocated blocks, another 0.5 N blocks will be lost to fragmentation. This property is known as the 50-percent rule.

Two solutions to external fragmentation:

- Compaction: The goal is to shuffle the memory-contents to place all free memorytogether in one large hole. Compaction is possible only if relocation is dynamic and done at execution-time
- Permit the logical-address space of the processes to be non-contiguous. This allows a process to be allocated physical-memory wherever such memory is available. Two techniques achieve this solution: 1) Paging and 2) Segmentation.

Paging

- Paging is a memory-management scheme.
- This permits the physical-address space of a process to be non-contiguous.
- This also solves the considerable problem of fitting memory-chunks of varying sizes onto the backing-store.
- Traditionally: Support for paging has been handled by hardware.
- Recent designs: The hardware & OS are closely integrated.

Basic Method of Paging

- The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called frames and breaking logical memory into blocks of the same size called pages.
- When a process is to be executed, its pages are loaded into any available memory frames from the backing store.
- The backing store is divided into fixed-sized blocks that are of the same size as the memory frames.

The hardware support for paging is illustrated in Figure 1.

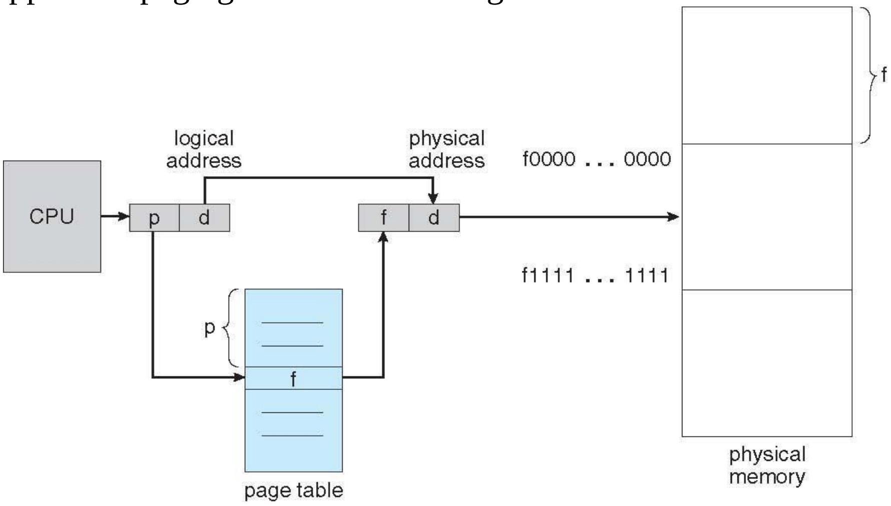


Figure 1: Paging hardware

- Address generated by CPU is divided into 2 parts (Figure 2):
  1. Page-number (p) is used as an index to the page-table. The page-table contains the base-address of each page in physical-memory.
  2. Offset (d) is combined with the base-address to define the physical-address. This physical-address is sent to the memory-unit.
- The page table maps the page number to a frame number, to yield a physical address
- The page table maps the page number to a frame number, to yield a physical address which also has two parts: The frame number and the offset within that frame.
- The number of bits in the frame number determines how many frames the system can address, and the number of bits in the offset determines the size of each frame.

The paging model of memory is shown in Figure 2.

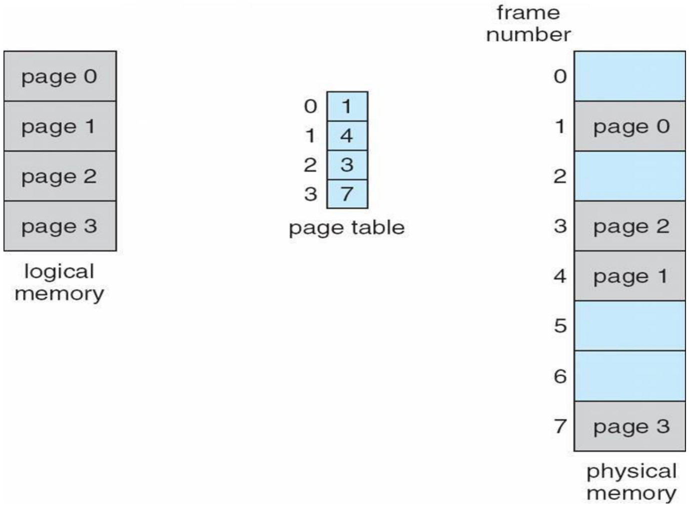


Figure 2: Paging model of logical and physical memory.

- The page size (like the frame size) is defined by the hardware.
- The size of a page is typically a power of 2, varying between 512 bytes and 16 MB per page, depending on the computer architecture.
- The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset.
- If the size of logical address space is 2^m and a page size is 2^n addressing units (bytes or words), then the high-order m – n bits of a logical address designate the page number, and the n low-order bits designate the page offset.

Thus, the logical address is as follows:

page number	page offset
p	d
m -n	n

- When a process requests memory (e.g. when its code is loaded in from disk), free frames are allocated from a free-frame list, and inserted into that process's page table.
- Processes are blocked from accessing anyone else's memory because all of their memory requests are mapped through their page table. There is no way for them to generate an address that maps into any other process's memory space.
- The operating system must keep track of each individual process's page table, updating it whenever the process's pages get moved in and out of memory, and applying the correct page table when processing system calls for a particular process. This all increases the overhead involved when swapping processes in and out of the CPU.

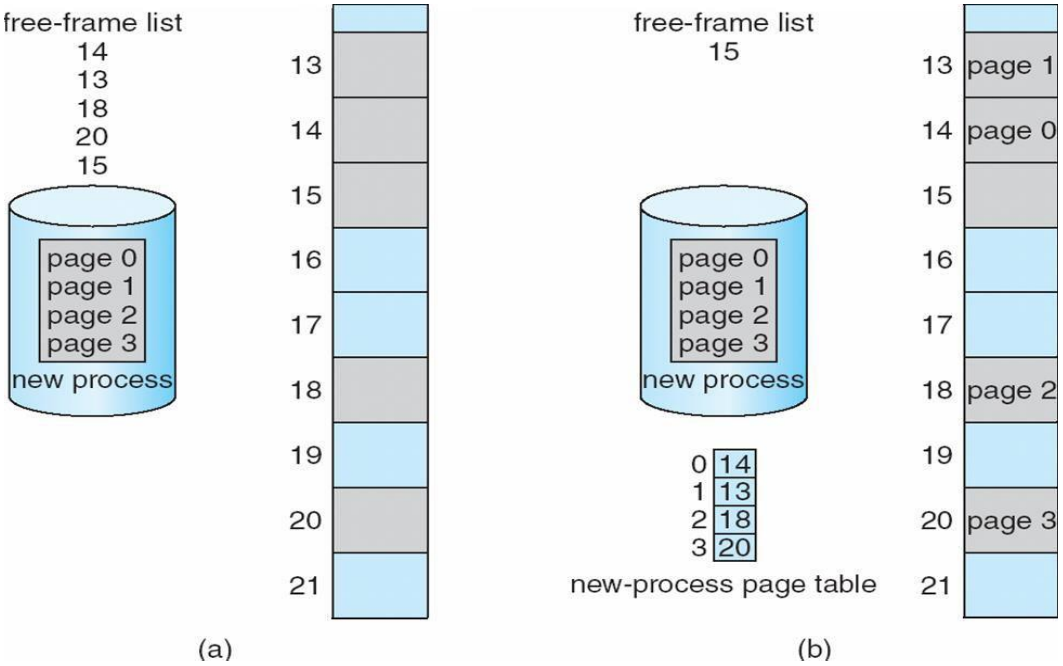


Figure: Free frames (a) before allocation and (b) after allocation.

Hardware Support

Translation Look aside Buffer

- A special, small, fast lookup hardware cache, called a translation look-aside buffer (TLB).
- Each entry in the TLB consists of two parts: a key (or tag) and a value.
- When the associative memory is presented with an item, the item is compared with all keys simultaneously. If the item is found, the corresponding value field is returned. The search is fast; the hardware, however, is expensive. Typically, the number of entries in a TLB is small, often numbering between 64 and 1,024.
- The TLB contains only a few of the page-table entries.

Working:

- When a logical-address is generated by the CPU, its page-number is presented to the TLB.
- If the page-number is found (TLB hit), its frame-number is immediately available and used to access memory
- If page-number is not in TLB (TLB miss), a memory-reference to page table must be made. The obtained frame-number can be used to access memory (Figure 1)

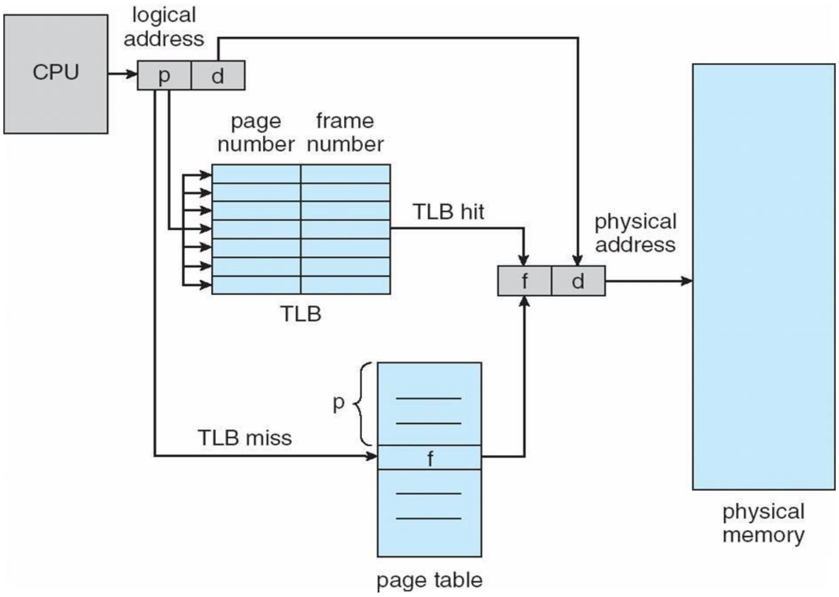


Figure 1: Paging hardware with TLB

- In addition, we add the page-number and frame-number to the TLB, so that they will be found quickly on the next reference.
- If the TLB is already full of entries, the OS must select one for replacement.
- Percentage of times that a particular page-number is found in the TLB is called hit ratio.

Advantage: Search operation is fast.
Disadvantage: Hardware is expensive.

- Some TLBs have wired down entries that can't be removed.
- Some TLBs store ASID (address-space identifier) in each entry of the TLB that uniquely identify each process and provide address space protection for that process.

Protection

- Memory-protection is achieved by protection-bits for each frame.
- The protection-bits are kept in the page-table.
- One protection-bit can define a page to be read-write or read-only.
- Every reference to memory goes through the page-table to find the correct frame-number.
- Firstly, the physical-address is computed. At the same time, the protection-bit is checked to verify that no writes are being made to a read-only page.
- An attempt to write to a read-only page causes a hardware-trap to the OS (or memory protection violation).

Valid Invalid Bit

- This bit is attached to each entry in the page-table.
- Valid bit: “valid” indicates that the associated page is in the process’ logical address space, and is thus a legal page
- Invalid bit: “invalid” indicates that the page is not in the process’ logical address space

Illegal addresses are trapped by use of valid-invalid bit.
The OS sets this bit for each page to allow or disallow access to the page.

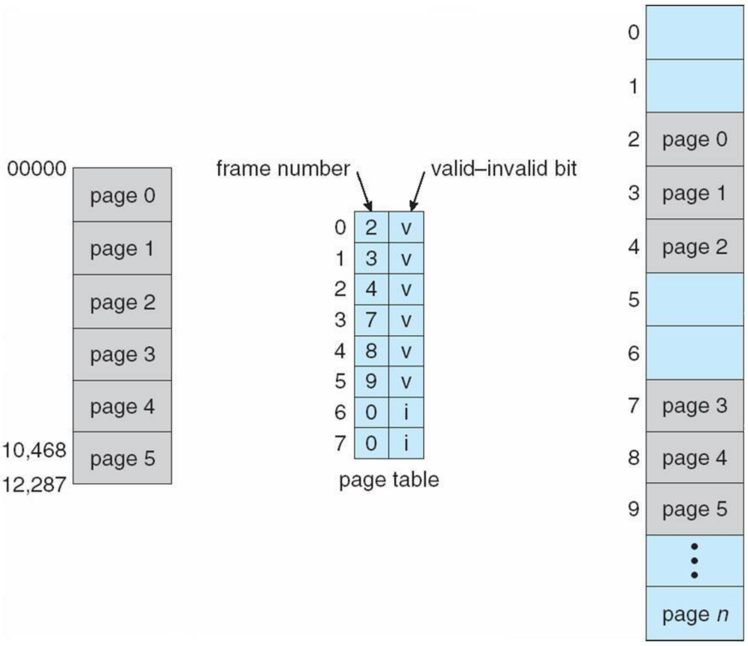


Figure: Valid (v) or invalid (i) bit in a page-table

Shared Pages

- An advantage of paging is the possibility of sharing common code.
- Re-entrant code (Pure Code) is non-self-modifying code, it never changes during execution.
- Two or more processes can execute the same code at the same time.
- Each process has its own copy of registers and data-storage to hold the data for the process's execution.
- The data for 2 different processes will be different.
- Only one copy of the editor need be kept in physical-memory (Figure 5.12).
- Each user's page-table maps onto the same physical copy of the editor, but data pages are mapped onto different frames.

Disadvantage:
Systems that use inverted page-tables have difficulty implementing shared-memory.

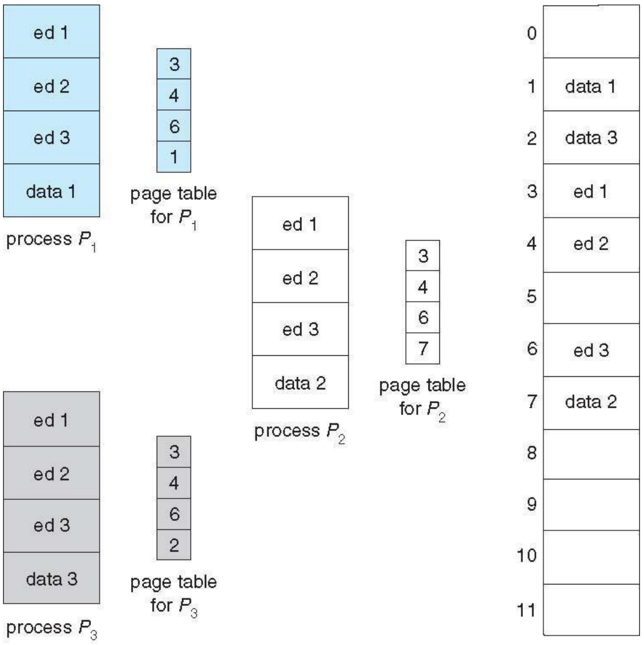


Figure: Sharing of code in a paging environment

Structure of the Page Table

The most common techniques for structuring the page table:

1. Hierarchical Paging
2. Hashed Page-tables
3. Inverted Page-tables

1. Hierarchical Paging

- Problem: Most computers support a large logical-address space (232 to 264). In these systems, the page-table itself becomes excessively large.
- Solution: Divide the page-table into smaller pieces.

Two Level Paging Algorithm:

- The page-table itself is also paged.
- This is also known as a forward-mapped page-table because address translation works from the outer page-table inwards.

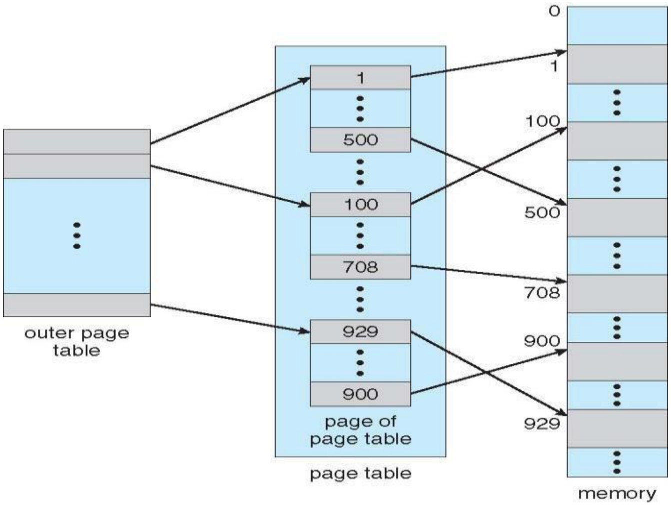


Figure: A two-level page-table scheme

For example:

Consider the system with a 32-bit logical-address space and a page-size of 4 KB.

A logical-address is divided into

→ 20-bit page-number and

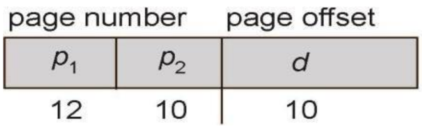
→ 12-bit page-offset.

Since the page-table is paged, the page-number is further divided into

→ 10-bit page-number and

→ 10-bit page-offset.

Thus, a logical-address is as follows:



- where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table

The address-translation method for this architecture is shown in below figure. Because address translation works from the outer page table inward, this scheme is also known as a forward-mapped page table.

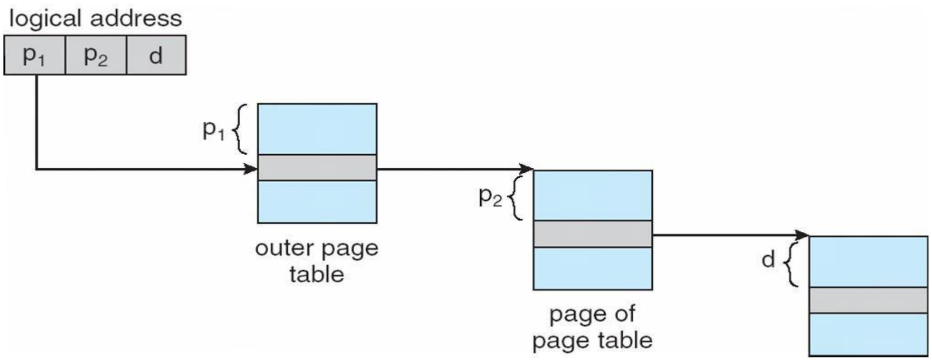


Figure: Address translation for a two-level 32-bit paging architecture

2. Hashed Page Tables

- This approach is used for handling address spaces larger than 32 bits.
- The hash-value is the virtual page-number.
- Each entry in the hash-table contains a linked-list of elements that hash to the same location (to handle collisions).
- Each element consists of 3 fields:
    1. Virtual page-number
    2. Value of the mapped page-frame and
    3. Pointer to the next element in the linked-list.

The algorithm works as follows:

- The virtual page-number is hashed into the hash-table.
- The virtual page-number is compared with the first element in the linked-list.
- If there is a match, the corresponding page-frame (field 2) is used to form the desired physical-address.
- If there is no match, subsequent entries in the linked-list are searched for a matching virtual page-number.

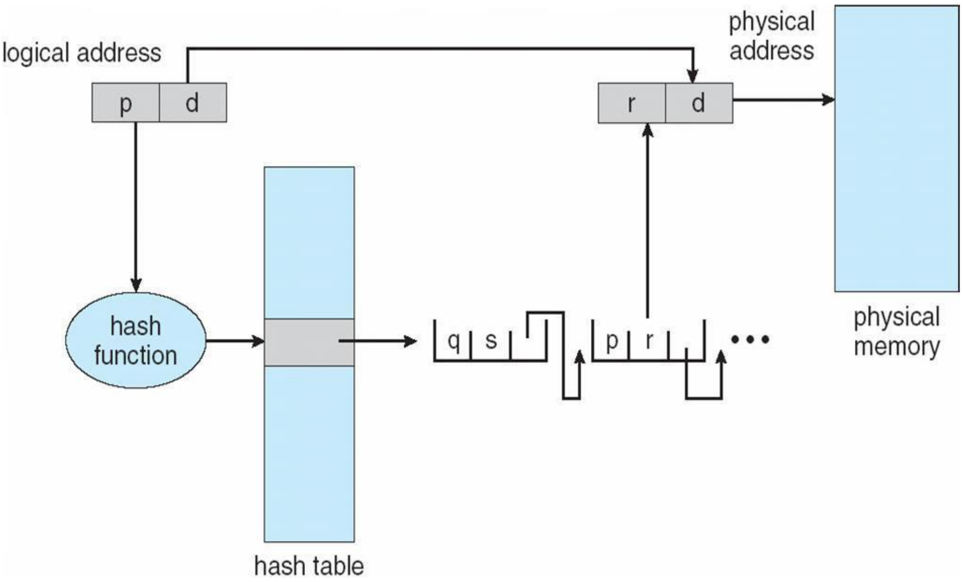


Figure: Hashed page-table

3. Inverted Page Tables

- Has one entry for each real page of memory.
- Each entry consists of virtual-address of the page stored in that real memory-location and information about the process that owns the page.
- Each virtual-address consists of a triplet <process-id, page-number, offset>.
- Each inverted page-table entry is a pair <process-id, page-number>

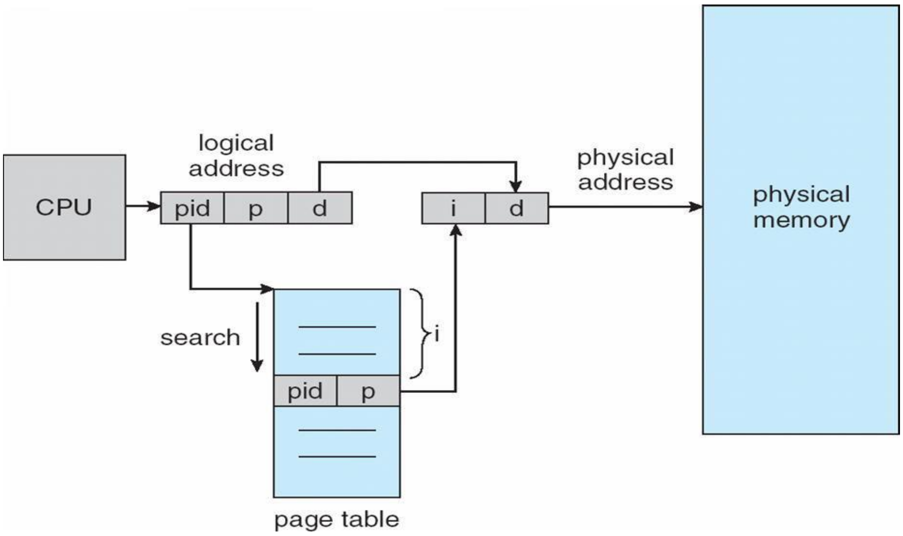


Figure: Inverted page-table

The algorithm works as follows:

1. When a memory-reference occurs, part of the virtual-address, consisting of <process-id, page-number>, is presented to the memory subsystem.
2. The inverted page-table is then searched for a match.
3. If a match is found, at entry i-then the physical-address <i, offset> is generated.
4. If no match is found, then an illegal address access has been attempted.

Advantage:

1. Decreases memory needed to store each page-table

Disadvantages:

1. Increases amount of time needed to search table when a page reference occurs.
2. Difficulty implementing shared-memory

Segmentation

Basic Method of Segmentation

- This is a memory-management scheme that supports user-view of memory (Figure 1).
- A logical-address space is a collection of segments.
- Each segment has a name and a length.
- The addresses specify both segment-name and offset within the segment.
- Normally, the user-program is compiled, and the compiler automatically constructs segments reflecting the input program.
- For ex: The code, Global variables, The heap, from which memory is allocated, The stacks used by each thread, The standard C library

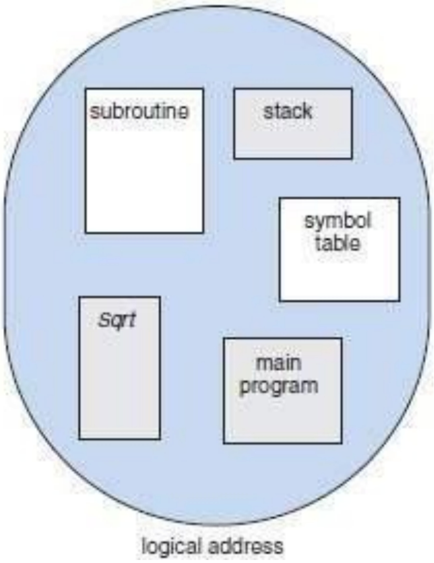


Figure: Programmer's view of a program

Hardware support for Segmentation

- Segment-table maps 2 dimensional user-defined addresses into one-dimensional physical addresses.
- In the segment-table, each entry has following 2 fields:
  1. Segment-base contains starting physical-address where the segment resides in memory.
  2. Segment-limit specifies the length of the segment (Figure 2).
- A logical-address consists of 2 parts:
  1. Segment-number(s) is used as an index to the segment-table
  2. Offset(d) must be between 0 and the segment-limit.
- If offset is not between 0 & segment-limit, then we trap to the OS(logical-addressing attempt beyond end of segment).
- If offset is legal, then it is added to the segment-base to produce the physical-memory address.

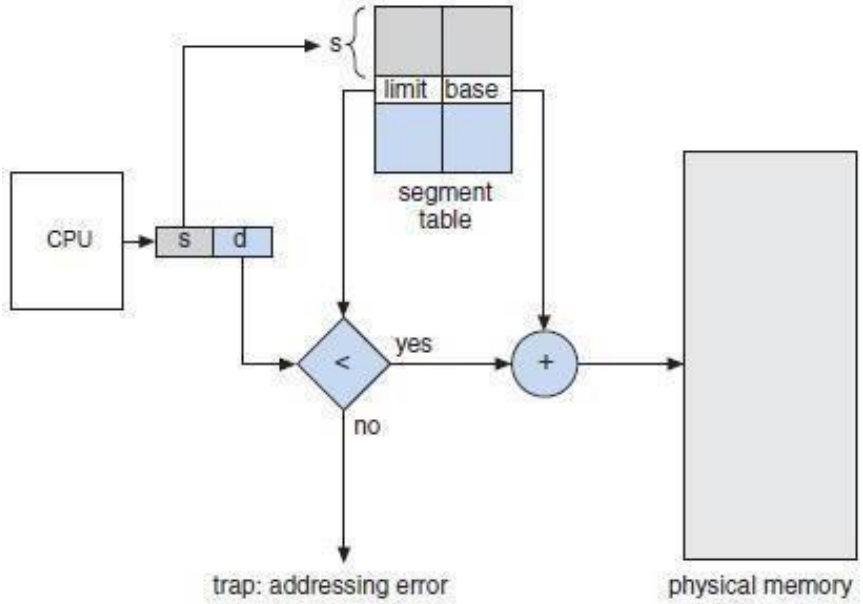


Figure: Segmentation hardware

QUESTION BANK

DEADLOCKS

1. What are deadlocks? What are its characteristics? Explain the necessary conditions for its occurrence.
2. Explain the process of recovery from deadlock.
3. Describe RAG:
   i) With deadlock
   ii) With a cycle but no deadlock
4. What is Resource Allocation Graph (RAG)? Explain how RAG is very useful in describing deadly embrace (dead lock) by considering your own example.
5. With the help of a system model, explain a deadlock and explain the necessary conditions that must hold simultaneously in a system for a deadlock to occur.
6. Explain how deadlock can be prevented by considering four necessary conditions cannot hold.
7. How is a system recovered from deadlock? Explain the different methods used to recover from deadlock.
8. Explain deadlock detection with algorithm and example
9. Define the terms: safe state and safe sequence. Give an algorithm to find whether or not a system is in a safe state.
10. b) Using Banker's algorithm determines whether the system is in a safe state.
   Consider the following snapshot of a system

	Allocation	Max	Need
	A B C	A B C	A B C
Po	012	012	- - -
P_1.	000	750	- - -
P_2	354	356	- - -
P_3.	632	652	- - -
P4-	014	656	- - -

   Available Quantity of resources are 5,2 and 0 for A, B and C respectively.
   Answer the following questions using Banker's algorithm.
   i) Analyze the content of the column need?
   ii) is the system in a safe state? If yes, write the safe sequence.
11. Describe the methods for recovery form Deadlock.
12. Using Banker's algorithm determines whether the system is in a safe state.
   Consider a system described by:

	Allocation	Max	Available
	ABC	ABC	ABC
Po	0 1 0	75 3	3 3 2
PI	2 0 0	3 2 2	
P2	3 0 2	9 0 2	
P3	2 1 1	2 2 2	
P4	0 0 2	4 3 3	

Check whether the system is safe or not. Determine the sequence if it is safe. Further if PI requests (1,0,2), determine if it can be immediately granted.

MEMORY MANAGEMENT

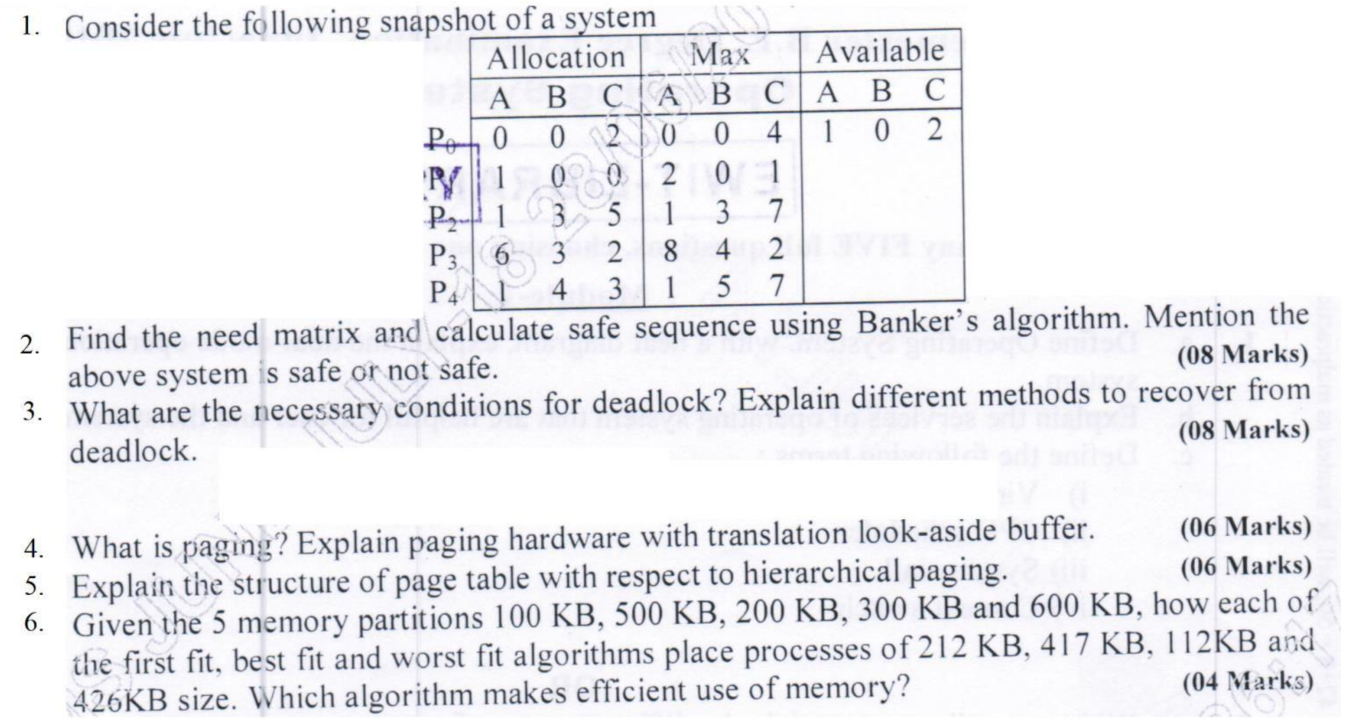
1. Explain the multistep processing of a user program with a neat block diagram.
2. Distinguish between internal and external fragmentation.
3. Explain segmentation with an example.
4. Explain with a diagram, how TLB is used to solve the problem of simple paging scheme.
5. With a supporting paging hardware, explain in detail concept of paging with an example for a 32-byte memory with 4-type pages with a process being 16-bytes. How many bits are reserved for page number and page offset in the logical address. Suppose the logical address is 5, calculate the corresponding physical address, after populating memory and page table.
6. What are the draw backs of contiguous memory allocation?
7. Consider a paging system with the page table stored in memory.
   i. if a memory reference takes 200 nano seconds, how long does a paged memory reference take?
   ii. if we add associative register and 75 percentage of all page table references are found in the associative registers, what is the effective memory access time? (Assume that finding a page table entry in the associative memory/registers takes.
8. Memory partitions of 100KB,500KB, 200KB, 300KB, 600KB (in order) are available. How would first - fit, best - fit and worst - fit algorithms place processes of 212KB, 417KB, 112KB and 426KB (in order). Which algorithm makes the most efficient use of memory ?
9. Consider a paging system with TLB and page table stored in memory. If hit ratio of TLB is 80 percent and it takes 20 nanoseconds to search TLB and 100 nanoseconds to access memory, find the effective memory access time.
10. With a diagram, explain the steps involved in handling a page fault.
11. What is address binding? Explain the concept of dynamic relocation of addresses.
12. For a logical address space of 8 pages of 1024 words mapped to a physical memory of 32 frames, find the number of bits in the logical address and the number of bits in the physical address.

zero time, if the entry is found).

13. Distinguish between:
    - i. Logical address space and physical address space.
    - ii. Internal fragmentation and external fragmentation.
    - iii. Paging and segmentation.
14. Explain with the help of supporting hardware diagram how the TLB improves the performance of a demand paging system.
15. Explain the concept of forward mapped page table.
16. What is fragmentation? Explain two types of memory fragmentation
17. What is swapping? Explain in detail.
18. What do you mean by address binding? Explain with the necessary steps, the binding of instructions and data to memory addresses.

University Questions

June / July 2018

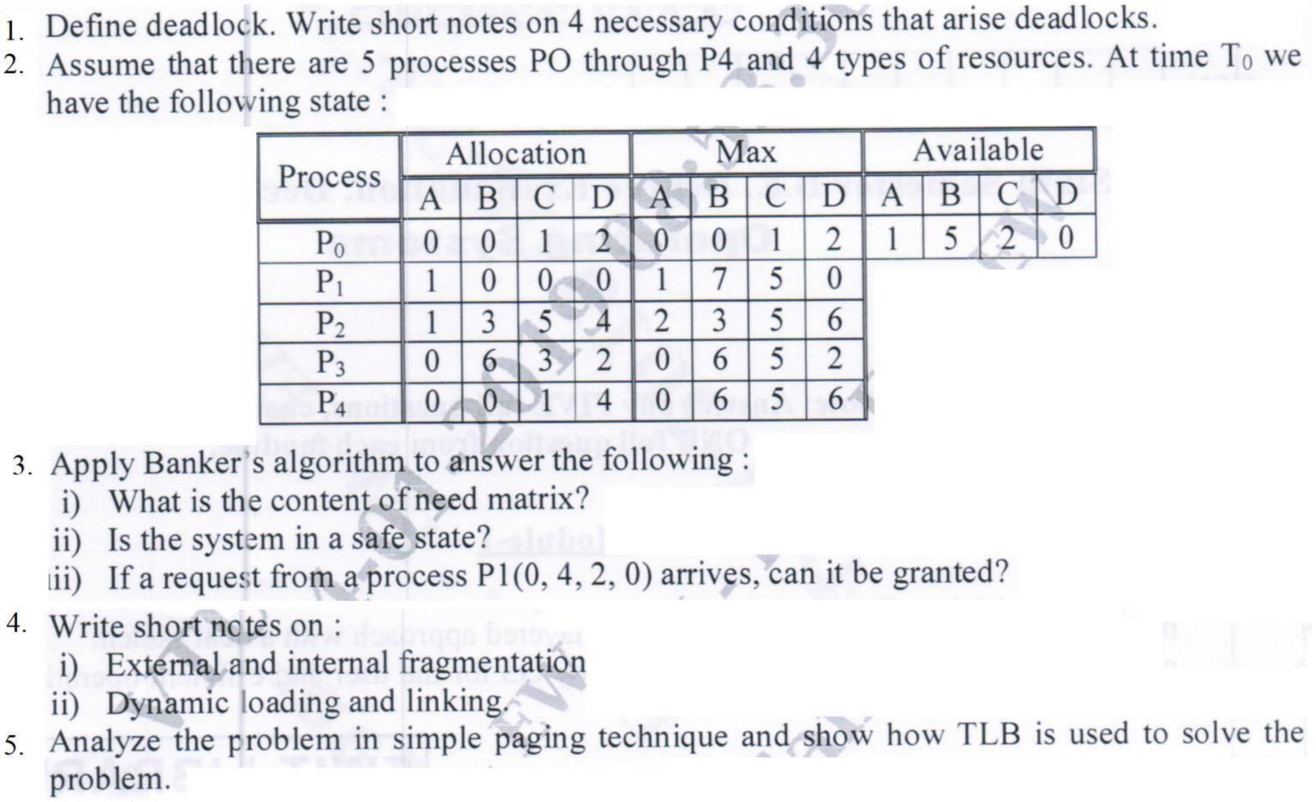


1. Consider the following snapshot of a system

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	0	2	0	0	4	1	0	2
P_1	1	0	0	2	0	1			
P_2	1	3	5	1	3	7			
P_3	6	3	2	8	4	2			
P_4	1	4	3	1	5	7			

2. Find the need matrix and calculate safe sequence using Banker's algorithm. Mention the above system is safe or not safe. **(08 Marks)**
3. What are the necessary conditions for deadlock? Explain different methods to recover from deadlock. **(08 Marks)**
4. What is paging? Explain paging hardware with translation look-aside buffer. **(06 Marks)**
5. Explain the structure of page table with respect to hierarchical paging. **(06 Marks)**
6. Given the 5 memory partitions 100 KB, 500 KB, 200 KB, 300 KB and 600 KB, how each of the first fit, best fit and worst fit algorithms place processes of 212 KB, 417 KB, 112KB and 426KB size. Which algorithm makes efficient use of memory? **(04 Marks)**

Dec. 2018 / Jan 2019

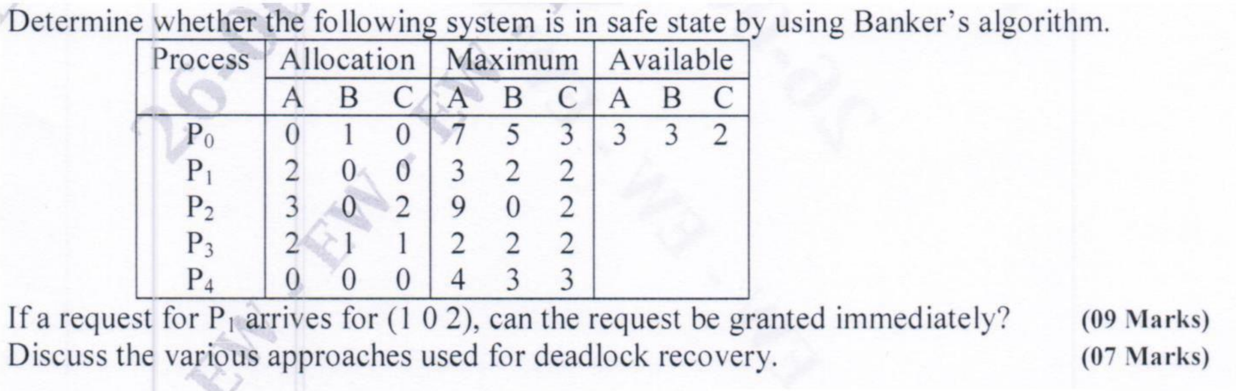


1. Define deadlock. Write short notes on 4 necessary conditions that arise deadlocks.
2. Assume that there are 5 processes PO through P4 and 4 types of resources. At time T_0 we have the following state :

Process	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P_0	0	0	1	2	0	0	1	2	1	5	2	0
P_1	1	0	0	0	1	7	5	0				
P_2	1	3	5	4	2	3	5	6				
P_3	0	6	3	2	0	6	5	2				
P_4	0	0	1	4	0	6	5	6				

3. Apply Banker's algorithm to answer the following :
   i) What is the content of need matrix?
   ii) Is the system in a safe state?
   iii) If a request from a process P1(0, 4, 2, 0) arrives, can it be granted?
4. Write short notes on :
   i) External and internal fragmentation
   ii) Dynamic loading and linking.
5. Analyze the problem in simple paging technique and show how TLB is used to solve the problem.

June / July 2019

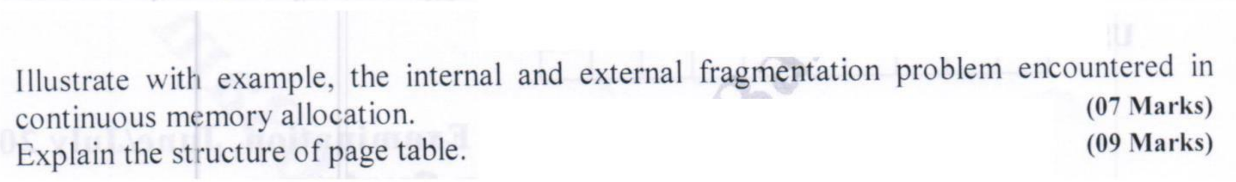


1. Determine whether the following system is in safe state by using Banker's algorithm.

Process	Allocation			Maximum			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	3	3	2
P_1	2	0	0	3	2	2			
P_2	3	0	2	9	0	2			
P_3	2	1	1	2	2	2			
P_4	0	0	0	4	3	3			

If a request for P_1 arrives for (1 0 2), can the request be granted immediately? **(09 Marks)**

2. Discuss the various approaches used for deadlock recovery. **(07 Marks)**



3. Illustrate with example, the internal and external fragmentation problem encountered in continuous memory allocation. **(07 Marks)**

4. Explain the structure of page table. **(09 Marks)**

MODULE 4

VIRTUAL MEMORYMANAGEMENT

- Virtual memory is a technique that allows for the execution of partially loaded process.
- Advantages:
  - A program will not be limited by the amount of physical memory that is available user can able to write in to large virtual space.
  - Since each program takes less amount of physical memory, more than one program could be run at the same time which can increase the throughput and CPU utilization.
  - Less i/o operation is needed to swap or load user program in to memory. So each user program could run faster.

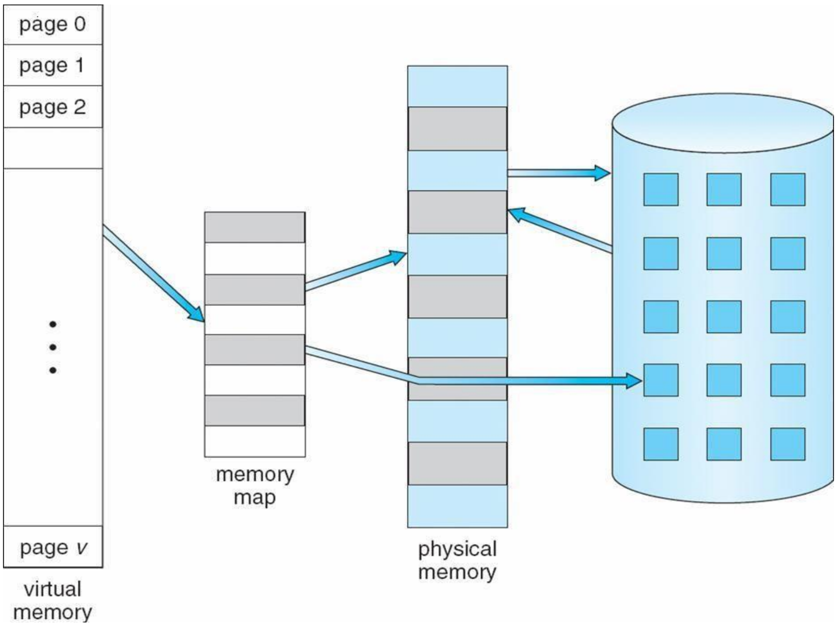


Fig: Virtual memory that is larger than physical memory.

- Virtual memory is the separation of users logical memory from physical memory. This separation allows an extremely large virtual memory to be provided when these is less physical memory.
- Separating logical memory from physical memory also allows files and memory to be shared by several different processes through page sharing.

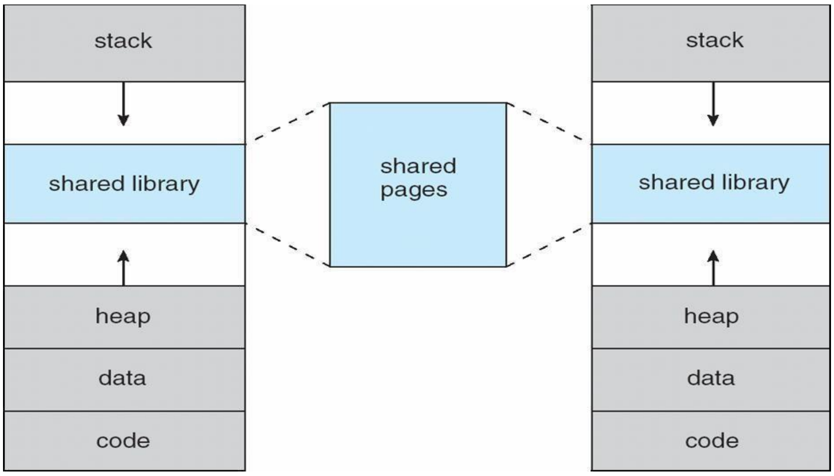


Fig: Shared Library using Virtual Memory

- Virtual memory is implemented using Demand Paging.
- Virtual address space: Every process has a virtual address space i.e used as the stack or heap grows in size.

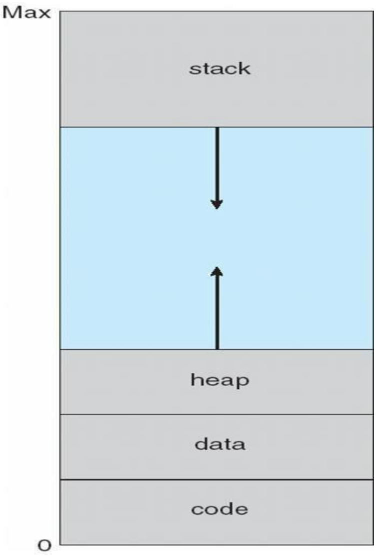


Fig: Virtual address space

DEMAND PAGING

- A demand paging is similar to paging system with swapping when we want to execute a process we swap the process the in to memory otherwise it will not be loaded in to memory.
- A swapper manipulates the entire processes, where as a pager manipulates individual pages of the process.
  - Bring a page into memory only when it is needed
  - Less I/O needed
  - Less memory needed
  - Faster response

- More users
- Page is needed ⇒ reference to it
- invalid reference ⇒abort
- not-in-memory ⇒ bring to memory
- ***Lazy swapper***– never swaps a page into memory unless page will be needed
- Swapper that deals with pages is a **pager.**

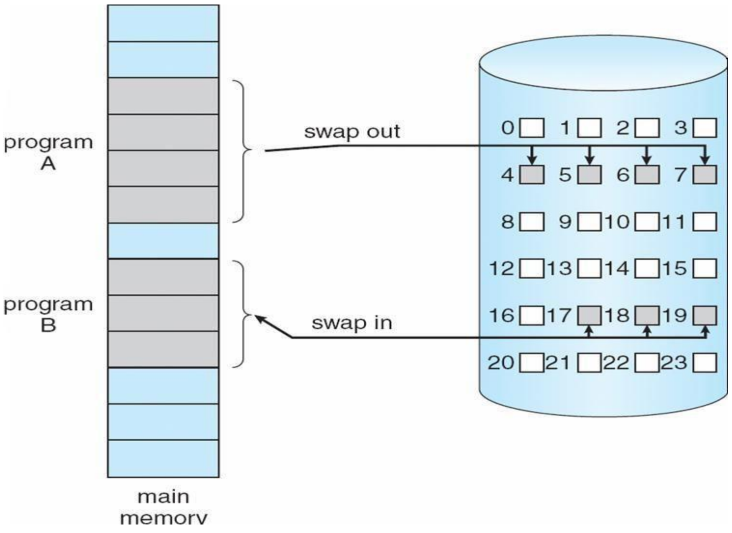
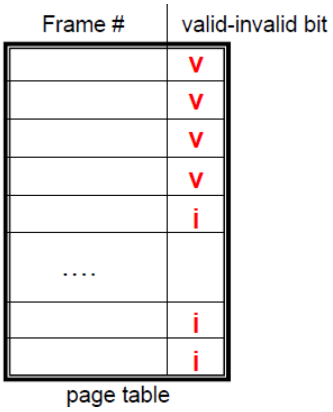


Fig: Transfer of a paged memory into continuous disk space

- **Basic concept:** Instead of swapping the whole process the pager swaps only the necessary pages in to memory. Thus it avoids reading unused pages and decreases the swap time and amount of physical memory needed.
- The valid-invalid bit scheme can be used to distinguish between the pages that are on the disk and that are in memory.
  - With each page table entry a valid–invalid bit is associated
  - (**v** ⇒ in-memory, **i**⇒not-in-memory)
  - Initially valid–invalid bit is set to **i**on all entries
  - Example of a page table snapshot:



Frame #	valid-invalid bit
	v
	v
	v
	v
	i
….	
	i
	i

page table

- During address translation, if valid–invalid bit in page table entry is **I** ⇒ page fault.
- If the bit is valid then the page is both legal and is in memory.
- If the bit is invalid then either page is not valid or is valid but is currently on the disk. Marking

a page as invalid will have no effect if the processes never access to that page. Suppose if it access the page which is marked invalid, causes a page fault trap. This may result in failure of OS to bring the desired page in to memory.

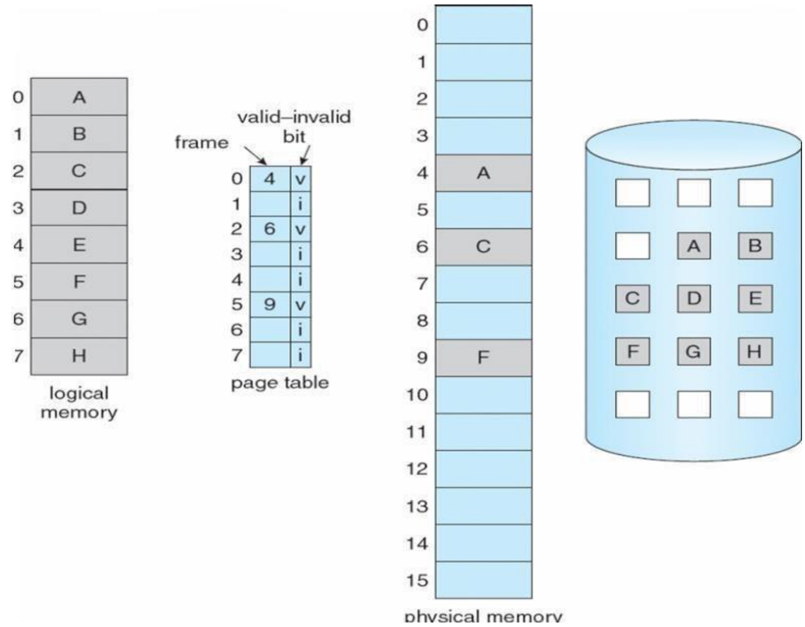


Fig: Page Table when some pages are not in main memory

Page Fault

If a page is needed that was not originally loaded up, then a ***page fault trap*** is generated.

Steps in Handling a Page Fault

1. The memory address requested is first checked, to make sure it was a valid memory request.
2. If the reference is to an invalid page, the process is terminated. Otherwise, if the page is not present in memory, it must be paged in.
3. A free frame is located, possibly from a free-frame list.
4. A disk operation is scheduled to bring in the necessary page from disk.
5. After the page is loaded to memory, the process's page table is updated with the new frame number, and the invalid bit is changed to indicate that this is now a valid page reference.
6. The instruction that caused the page fault must now be restarted from the beginning.

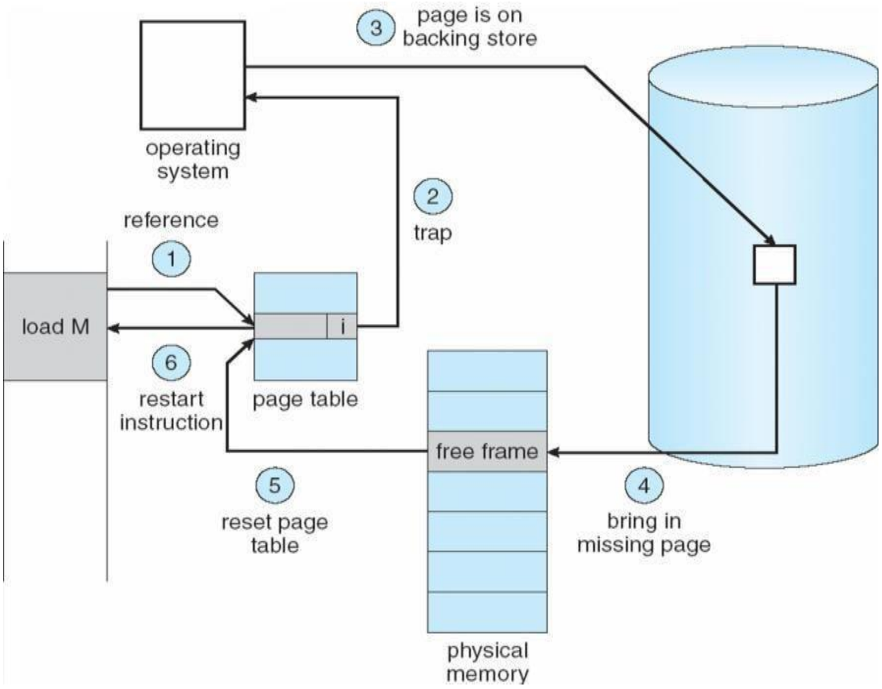


Fig: steps in handling page fault

Pure Demand Paging: Never bring a page into main memory until it is required.

- We can start executing a process without loading any of its pages into main memory.
- Page fault occurs for the non memory resident pages.
- After the page is brought into memory, process continues to execute.
- Again page fault occurs for the next page.

Hardware support: For demand paging the same hardware is required as paging and swapping.

1. Page table:-Has the ability to mark an entry invalid through valid-invalid bit.
2. Secondary memory:-This holds the pages that are not present in main memory.

Performance of Demand Paging: Demand paging can have significant effect on the performance of the computer system.

- Let P be the probability of the page fault (0<=P<=1)
- **Effective access time = (1-P) * ma + P * page fault.**
    - Where P = page fault and ma = memory access time.
- Effective access time is directly proportional to page fault rate. It is important to keep page fault rate low in demand paging.

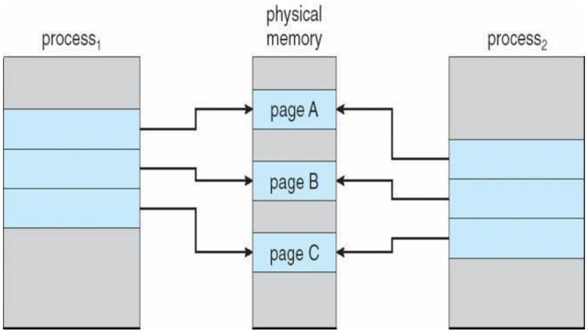
Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8milliseconds
- EAT = (1 – p) x 200 + p (8milliseconds)
  = (1 – p x 200 + p x 8,000,000
  = **200 + p x 7,999,800**
- If one access out of 1,000 causes a page fault, then EAT = 8.2 microseconds. This is a slowdown by a factor of 40.

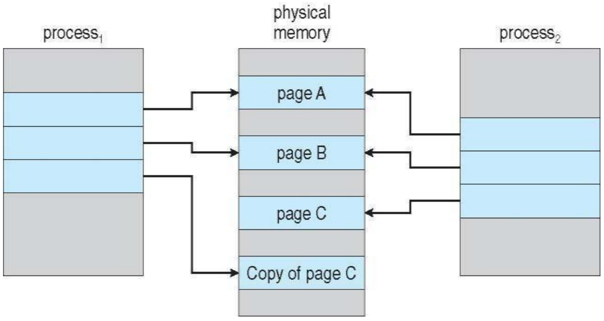
COPY-ON-WRITE

- Technique initially allows the parent and the child to share the same pages. These pages are marked as copy on- write pages i.e., if either process writes to a shared page, a copy of shared page is created.
- *Eg*:-If a child process try to modify a page containing portions of the stack; the OS recognizes them as a copy-on-write page and create a copy of this page and maps it on to the address space of the child process. So the child process will modify its copied page and not the page belonging to parent. The new pages are obtained from the pool of free pages.
- The previous contents of pages are erased before getting them into main memory. This is called **Zero – on fill demand.**

a) Before Process 1 modifies pageC



b) After process 1 modifies page C



PAGEREPLACEMENT

- Page replacement policy deals with the solution of pages in memory to be replaced by a new page that must be brought in. When a user process is executing a page fault occurs.
- The hardware traps to the operating system, which checks the internal table to see that this is a page fault and not an illegal memory access.
- The operating system determines where the derived page is residing on the disk, and this finds that there are no free frames on the list of free frames.
- When all the frames are in main memory, it is necessary to bring a new page to satisfy the page fault, replacement policy is concerned with selecting a page currently in memory to be replaced.
- The page i,e to be removed should be the page i,e least likely to be referenced in future.

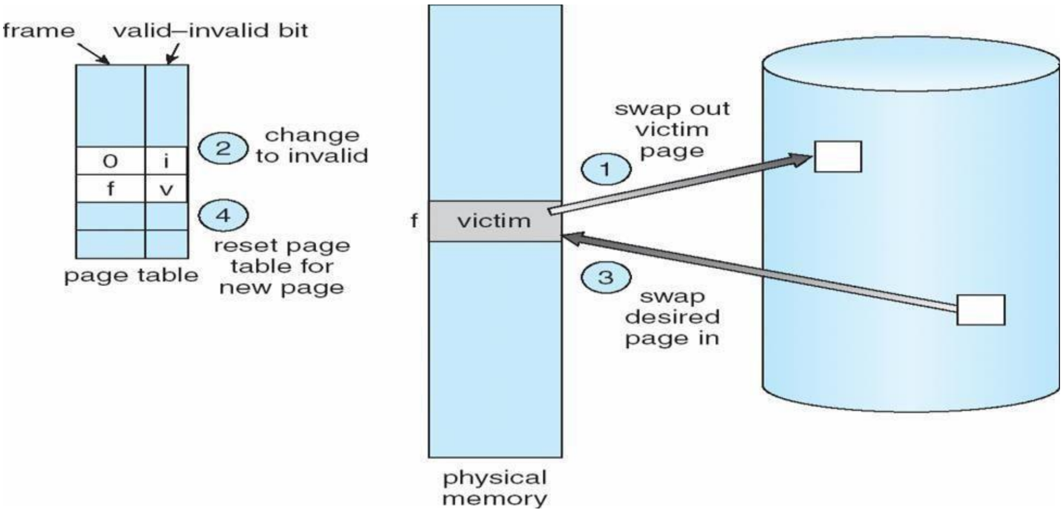


Fig: Page Replacement

Working of Page Replacement Algorithm

1. Find the location of derived page on the disk.
2. Find a free frame x If there is a free frame, use it. x Otherwise, use a replacement algorithm to select a victim.
   - Write the victim page to the disk.
   - Change the page and frame tables accordingly.
3. Read the desired page into the free frame; change the page and frame tables.
4. Restart the user process.

Victim Page

- The page that is supported out of physical memory is called victim page.
- If no frames are free, the two page transforms come (out and one in) are read. This will see the effective access time.
- Each page or frame may have a dirty (modify) bit associated with the hardware. The modify bit for a page is set by the hardware whenever any word or byte in the page is

written into, indicating that the page has been modified.

- When we select the page for replacement, we check its modify bit. If the bit is set, then the page is modified since it was read from the disk.
- If the bit was not set, the page has not been modified since it was read into memory. Therefore, if the copy of the page has not been modified we can avoid writing the memory page to the disk, if it is already there. Sum pages cannot be modified.

Modify bit/ Dirty bit :

- Each page/frame has a modify bit associated with it.
- If the page is not modified (read-only) then one can discard such page without writing it onto the disk**.** Modify bit of such page is set to0.
- Modify bit is set to 1, if the page has been modified. Such pages must be written to the disk.
- Modify bit is used to reduce overhead of page transfers – only modified pages are written to disk

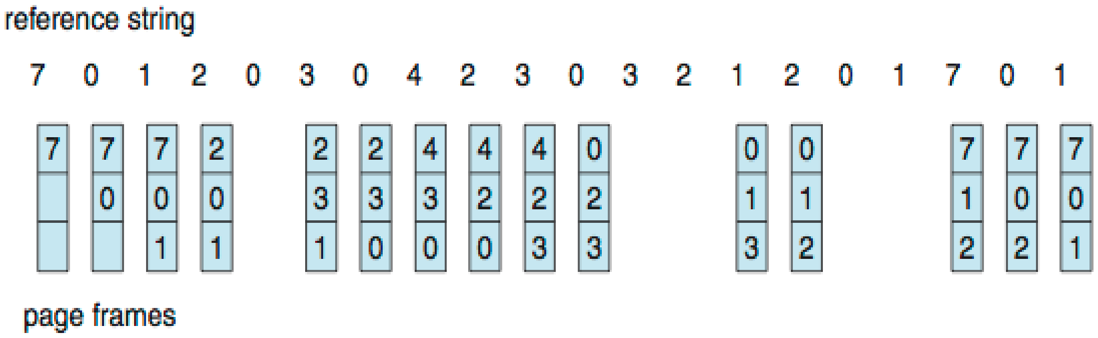
PAGE REPLACEMENT ALGORITHMS

- Want lowest page-fault rate
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
- In all our examples, the reference string is

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

FIFO Algorithm:

- This is the simplest page replacement algorithm. A FIFO replacement algorithm associates each page the time when that page was brought into memory.
- When a Page is to be replaced the oldest one is selected.
- We replace the queue at the head of the queue. When a page is brought into memory, we insert it at the tail of the queue.
- In the following example, a reference string is given and there are 3 free frames. There are 20 page requests, which results in 15 page faults



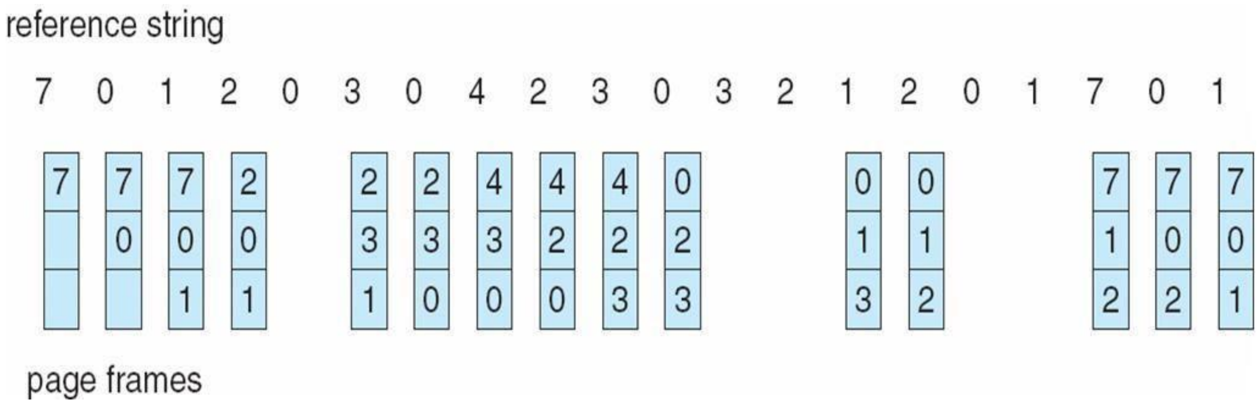
Belady’s Anomaly

- For some page replacement algorithm, the page fault may increase as the number of allocated frames increases. FIFO replacement algorithm may face this problem.

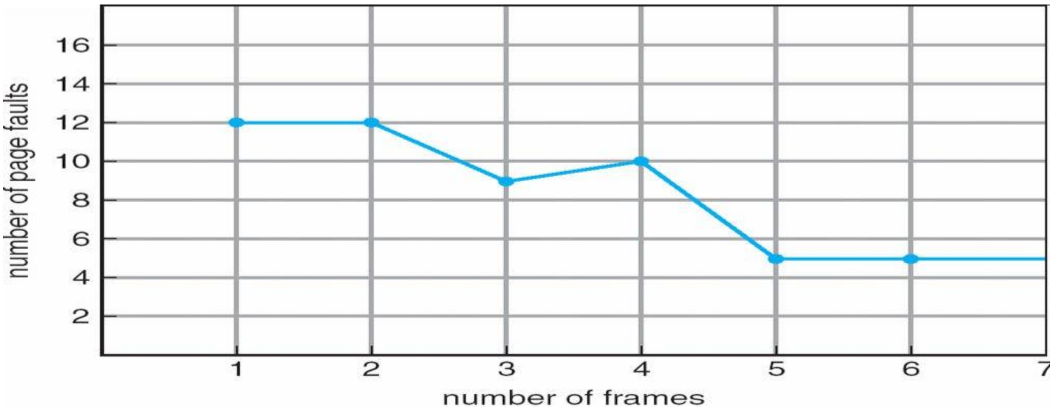
more frames ⇒ more page faults

Example: Consider the following references string with frames initially empty.

- The first three references (7,0,1) cases page faults and are brought into the empty frames.
- The next references 2 replaces page 7 because the page 7 was brought in first. x Since 0 is the next references and 0 is already in memory e has no page faults.
- The next references 3 results in page 0 being replaced so that the next references to 0 causer page fault. This will continue till the end of string. There are 15 faults all together.



FIFO Illustrating Belady’s Anomaly



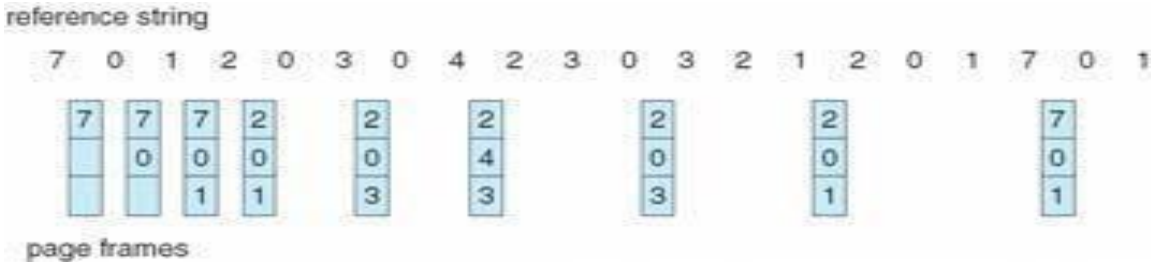
Optimal Algorithm

- Optimal page replacement algorithm is mainly to solve the problem of Belady’s Anomaly.
- Optimal page replacement algorithm has the lowest page fault rate of all algorithms.
- An optimal page replacement algorithm exists and has been called OPT.

The working is simple “Replace the page that will not be used for the longest period of time”
Example: consider the following reference string

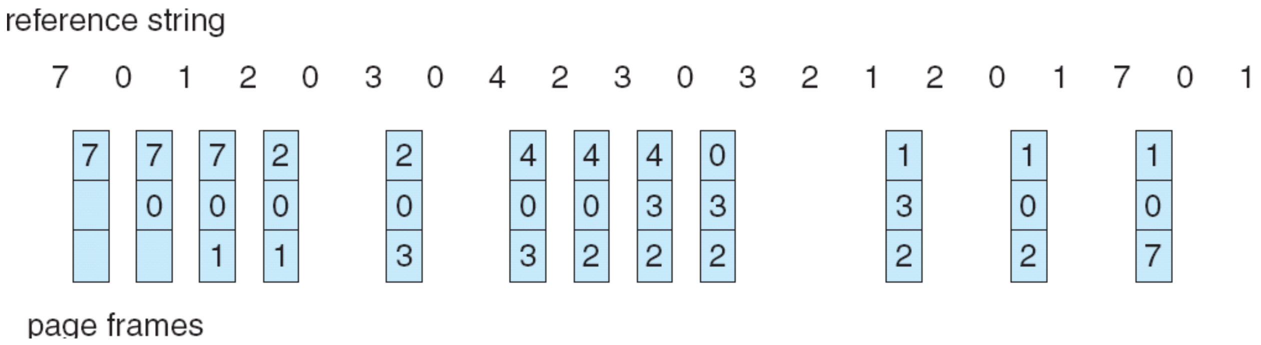
- The first three references cause faults that fill the three empty frames.
- The references to page 2 replaces page 7, because 7 will not be used until reference 18. x The page 0 will be used at 5 and page 1 at 14.
- With only 9 page faults, optimal replacement is much better than a FIFO, which had 15 faults. This algorithm is difficult t implement because it requires future knowledge of reference strings.
- Replace page that will not be used for longest period of time

Optimal Page Replacement



Least Recently Used (LRU) Algorithm

- The ***LRU (Least Recently Used)*** algorithm, predicts that the page that has not been used in the longest time is the one that will not be used again in the near future.
- Some view LRU as analogous to OPT, but here we look backwards in time instead of forwards.



The main problem to how to implement LRU is the LRU requires additional h/w assistance.

Two implementation are possible:

1. **Counters:** In this we associate each page table entry a time -of -use field, and add to the cpu a logical clock or counter. The clock is incremented for each memory reference. When a reference to a page is made, the contents of the clock register are copied to the time-of-use field in the page table entry for that page. In this way we have the time of last reference to each page we replace the page with smallest time value. The time must also be maintained when page tables are changed.
2. **Stack:** Another approach to implement LRU replacement is to keep a stack of page numbers when a page is referenced it is removed from the stack and put on to the top of stack. In this way the top of stack is always the most recently used page and the bottom in least recently used page. Since the entries are removed from the stack it is best implement by a doubly linked list. With a head and tail pointer.

Note: Neither optimal replacement nor LRU replacement suffers from Belady's Anamoly. These are called stack algorithms.

LRU-Approximation Page Replacement

- Many systems offer some degree of hardware support, enough to approximate LRU.
- In particular, many systems provide a ***reference bit*** for every entry in a page table, which is set anytime that page is accessed. Initially all bits are set to zero, and they can also all be cleared at any time. One bit distinguishes pages that have been accessed since the last clear from those that have not been accessed.

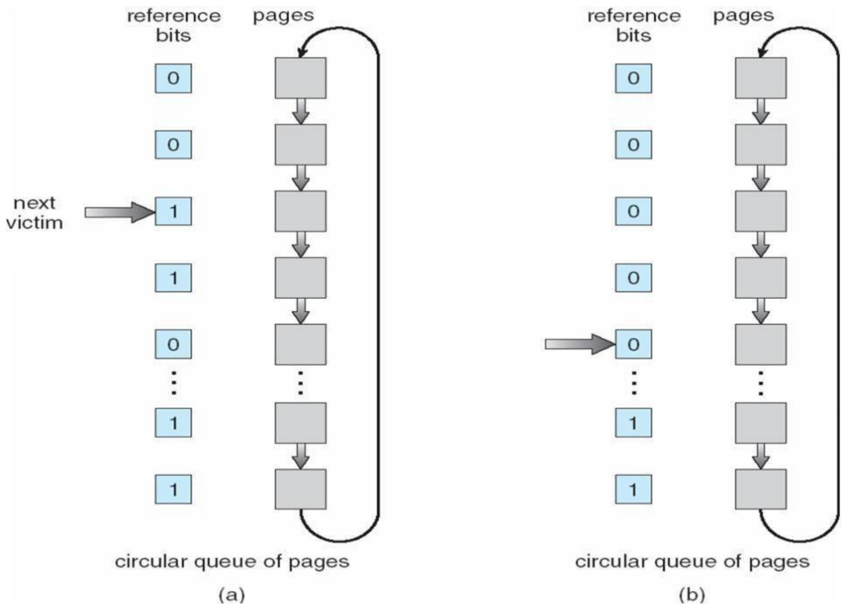
Additional-Reference-Bits Algorithm

- An 8-bit byte (reference bit) is stored for each page in a table in memory.
- At regular intervals (say, every 100 milliseconds), a timer interrupt transfers control to the operating system. The operating system shifts the reference bit for each page into the high-order bit of its 8-bit byte, shifting the other bits right by 1 bit and discarding the low-order bit.
- These 8-bit shift registers contain the history of page use for the last eight time periods.
- If the shift register contains 00000000, then the page has not been used for eight time periods.
- A page with a history register value of 11000100 has been used more recently than one with a value of 01110111.

Second- chance (clock) page replacement algorithm

- The ***second chance algorithm*** is a FIFO replacement algorithm, except the reference bit is used to give pages a second chance at staying in the page table.
- When a page must be replaced, the page table is scanned in a FIFO (circular queue) manner.
- If a page is found with its reference bit as '0', then that page is selected as the next victim.

- If the reference bitvalueis‘1’, then the page is given a second chance and its reference bit value is cleared (assigned as‘0’).
- Thus, a page that is given a second chance will not be replaced until all other pages have been replaced (or given second chances). In addition, if a page is used often, then it sets its reference bit again.
- This algorithm is also known as the ***clock*** algorithm.



Enhanced Second-Chance Algorithm

- The **enhanced second chance algorithm** looks at the reference bit and the modify bit (dirty bit) as an ordered page, and classifies pages into one of four classes:
    1. (0, 0) - Neither recently used nor modified.
    2. (0, 1) - Not recently used, but modified.
    3. (1, 0) - Recently used, but clean.
    4. (1, 1) - Recently used and modified.
- This algorithm searches the page table in a circular fashion, looking for the first page it can find in the lowest numbered category. i.e. it first makes a pass looking for a (0, 0), and then if it can't find one, it makes another pass looking for a(0,1),etc.
- The main difference between this algorithm and the previous one is the preference for replacing clean pages if possible.

Count Based Page Replacement

There is many other algorithms that can be used for page replacement, we can keep a counter of the number of references that has made to a page.

a) **LFU** (least frequently used):

This causes the page with the smallest count to be replaced. The reason for this selection is that actively used page should have a large reference count.

This algorithm suffers from the situation in which a page is used heavily during the initial phase of a process but never used again. Since it was used heavily, it has a large count and remains in memory even though it is no longer needed.

b) **MFU** Algorithm:

based on the argument that the page with the smallest count was probably just brought in and has yet to be used

ALLOCATION OF FRAMES

- The absolute minimum number of frames that a process must be allocated is dependent on system architecture.
- The maximum number is defined by the amount of available physical memory.

Allocation Algorithms

After loading of OS, there are two ways in which the allocation of frames can be done to the processes.

1. **Equal Allocation**- If there are m frames available and n processes to share them, each process gets m / n frames, and the left over's are kept in a free-frame buffer pool.
2. **Proportional Allocation** - Allocate the frames proportionally depending on the size of the process. If the size of process i is S_i, and S is the sum of size of all processes in the system, then the allocation for process P_i is $a_i = m * S_i / S$. where m is the free frames available in the system.

- Consider a system with a 1KB frame size. If a small student process of 10 KB and an interactive database of 127 KB are the only two processes running in a system with 62 free frames.
- with proportional allocation, we would split 62 frames between two processes, as follows

  m=62, S = (10+127)=137

  Allocation for process 1 = 62 X 10/137 ~ 4 Allocation for process 2 = 62 X 127/137 ~57

  Thus allocates 4 frames and 57 frames to student process and database respectively.
- Variations on proportional allocation could consider priority of process rather than just their size.

Global versus Local Allocation

- Page replacement can occur both at local or global level.
- With local replacement, the number of pages allocated to a process is fixed, and page replacement occurs only amongst the pages allocated to this process.
- With global replacement, any page may be a potential victim, whether it currently belongs to the process seeking a free frame or not.
- Local page replacement allows processes to better control their own page fault rates, and leads to more consistent performance of a given process over different system load levels.

- Global page replacement is over all more efficient, and is the more commonly used approach.

Non-Uniform Memory Access (New)

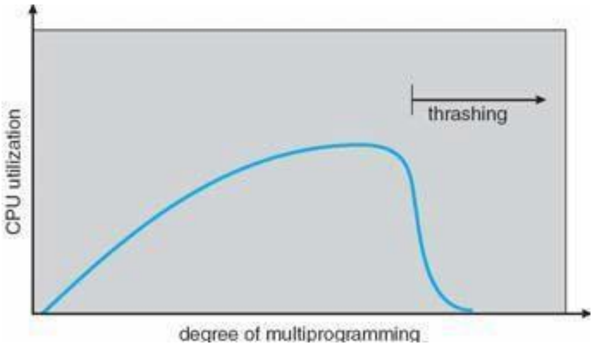
- Usually the time required to access all memory in a system is equivalent.
- This may not be the case in multiple-processor systems, especially where each CPU is physically located on a separate circuit board which also holds some portion of the overall system memory.
- In such systems, CPU s can access memory that is physically located on the same board much faster than the memory on the other boards.
- The basic solution is akin to processor affinity - At the same time that we try to schedule processes on the same CPU to minimize cache misses, we also try to allocate memory for those processes on the same boards, to minimize access times.

THRASHING

- If the number of frames allocated to a low-priority process falls below the minimum number required by the computer architecture then we suspend the process execution.
- A process is thrashing if it is spending more time in paging than executing.
- If the processes do not have enough number of frames, it will quickly page fault. During this it must replace some page that is not currently in use. Consequently it quickly faults again and again.
- The process continues to fault, replacing pages for which it then faults and brings back. This high paging activity is called thrashing. The phenomenon of excessively moving pages back and forth b/w memory and secondary has been called ***thrashing***.

Cause of Thrashing

- Thrashing results in severe performance problem.
- The operating system monitors the cpu utilization is low. We increase the degree of multi programming by introducing new process to the system.
- A global page replacement algorithm replaces pages with no regards to the process to which they belong.



The figure shows the thrashing

- As the degree of multi programming increases, more slowly until a maximum is reached. If the degree of multi programming is increased further thrashing sets in and the cpu utilization drops sharply.

- At this point, to increases CPU utilization and stop thrashing, we must increase degree of multiprogramming.
- we can limit the effect of thrashing by using a local replacement algorithm. To prevent thrashing, we must provide a process as many frames as it needs.

Locality of Reference:

- As the process executes it moves from locality to locality.
- A locality is a set of pages that are actively used.
- A program may consist of several different localities, which may overlap.
- Locality is caused by loops in code that find to reference arrays and other data structures by indices.
- The ordered list of page number accessed by a program is called reference string.
- Locality is of two types :

  1. spatial locality 2. temporal locality

Working set model

- Working set model algorithm uses the current memory requirements to determine the number of page frames to allocate to the process, an informal definition is "the collection of pages that a process is working with and which must be resident if the process to avoid thrashing". The idea is to use the recent needs of a process to predict its future reader.
- The working set is an approximation of programs locality. Ex: given a sequence of memory reference, if the working set window size to memory references, then working set at time t1 is{1,2,5,6,7} and at t2 is changed to {3,4}
- At any given time, all pages referenced by a process in its last 4 seconds of execution are considered to compromise its working set.
- A process will never execute until its working set is resident in main memory.
- Pages outside the working set can be discarded at any movement.
- Working sets are not enough and we must also introduce balance set.
  - If the sum of the working sets of all the run able process is greater than the size of memory the refuse some process for a while.
  - Divide the run able process into two groups, active and inactive. The collection of active set is called the balance set. When a process is made active its working set is loaded.
  - Some algorithm must be provided for moving process into and out of the balance set. As a working set is changed, corresponding change is made to the balance set.
  - Working set presents thrashing by keeping the degree of multi programming as high as possible. Thus if optimizes the CPU utilization. The main disadvantage of this is keeping track of the working set.

Page-Fault Frequency

- When page- fault rate is too high, the process needs more frames and when it is too low, the process may have too many frames.
- The upper and lower bounds can be established on the page-fault rate. If the actual page- fault rate exceeds the upper limit, allocate the process another frame or suspend the process.
- If the page-fault rate falls below the lower limit, remove a frame from the process. Thus, we can directly measure and control the page-fault rate to prevent thrashing.

MODULE-5
FILE SYSTEM

FILE:

- *A* file is a named collection of related information that is recorded on secondary storage.
- The information in a file is defined by its creator. Many different types of information may be stored in a file source programs, object programs, executable programs, numeric data, text, payroll records, graphic images, sound recordings, and so on.

A file has a certain defined which depends on its type.

- *A text* file is a sequence of characters organized into lines.
- *A source* file is a sequence of subroutines and functions, each of which is further organized as declarations followed by executable statements.
- An *object* file is a sequence of bytes organized into blocks understandable by the system's linker.
- An *executable* file is a series of code sections that the loader can bring into memory and execute.

File Attributes

- A file is named, for the convenience of its human users, and is referred to by its name. A name is usually a string of characters, such as *example*.c
- When a file is named, it becomes independent of the process, the user, and even the system that created it.

A file's attributes vary from one operating system to another but typically consist of these:

- **Name:** The symbolic file name is the only information kept in human readable form.
- **Identifier:** This unique tag, usually a number, identifies the file within the file system; it is the non-human-readable name for the file.
- **Type:** This information is needed for systems that support different types of files.
- **Location:** This information is a pointer to a device and to the location of the file on that device.
- **Size:** The current size of the file (in bytes, words, or blocks) and possibly the maximum allowed size are included in this attribute.
- **Protection:** Access-control information determines who can do reading, writing, executing, and so on.
- **Time, date, and user identification:** This information may be kept for creation, last modification, and last use. These data can be useful for protection, security, and usage monitoring.

The information about all files is kept in the directory structure, which also resides on secondary storage. Typically, a directory entry consists of the file's name and its unique identifier. The identifier in turn locates the other file attributes.

File Operations

A file is an abstract data type. To define a file properly, we need to consider the operations that can be performed on files.

1. **Creating a file:** Two steps are necessary to create a file,
   a) Space in the file system must be found for the file.
   b) An entry for the new file must be made in the directory.
2. **Writing a file:** To write a file, we make a system call specifying both the name of the file and the information to be written to the file. Given the name of the file, the system searches the directory to find the file's location. The system must keep a write pointer to the location in the file where the next write is to take place. The write pointer must be updated whenever a write occurs.
3. **Reading a file:**To read from a file, we use a system call that specifies the name of the file and where the next block of the file should be put. Again, the directory is searched for the associated entry, and the system needs to keep a read pointer to the location in the file where the next read is to take place. Once the read has taken place, the read pointer is updated. Because a process is usually either reading from or writing to a file, the current operation location can be kept as a per-process current file-position pointer.
4. **Repositioning within a file:**The directory is searched for the appropriate entry, and the current-file-position pointer is repositioned to a given value. Repositioning within a file need not involve any actual I/0. This file operation is also known as files seek.
5. **Deleting a file:**To delete a file, search the directory for the named file. Having found the associated directory entry, then release all file space, so that it can be reused by other files, and erase the directory entry.
6. **Truncating a file:**The user may want to erase the contents of a file but keep its attributes. Rather than forcing the user to delete the file and then recreate it, this function allows all attributes to remain unchanged but lets the file be reset to length zero and its file space released.

- Other common operations include appending new information to the end of an existing file and renaming an existing file.
- Most of the file operations mentioned involve searching the directory for the entry associated with the named file.
- To avoid this constant searching, many systems require that an open () system call be made before a file is first used actively.
- The operating system keeps a small table, called the **open file table** containing information about all open files. When a file operation is requested, the file is specified via an index into this table, so no searching is required.

- The implementation of the open() and close() operations is more complicated in an environment where several processes may open the file simultaneously
- The operating system uses two levels of internal tables:
    1. A per-process table
    2. A system-wide table

The per-process table:

- Tracks all files that a process has open. Stored in this table is information regarding the use of the file by the process.
- Each entry in the per-process table in turn points to a system-wide open-file table.

The system-wide table

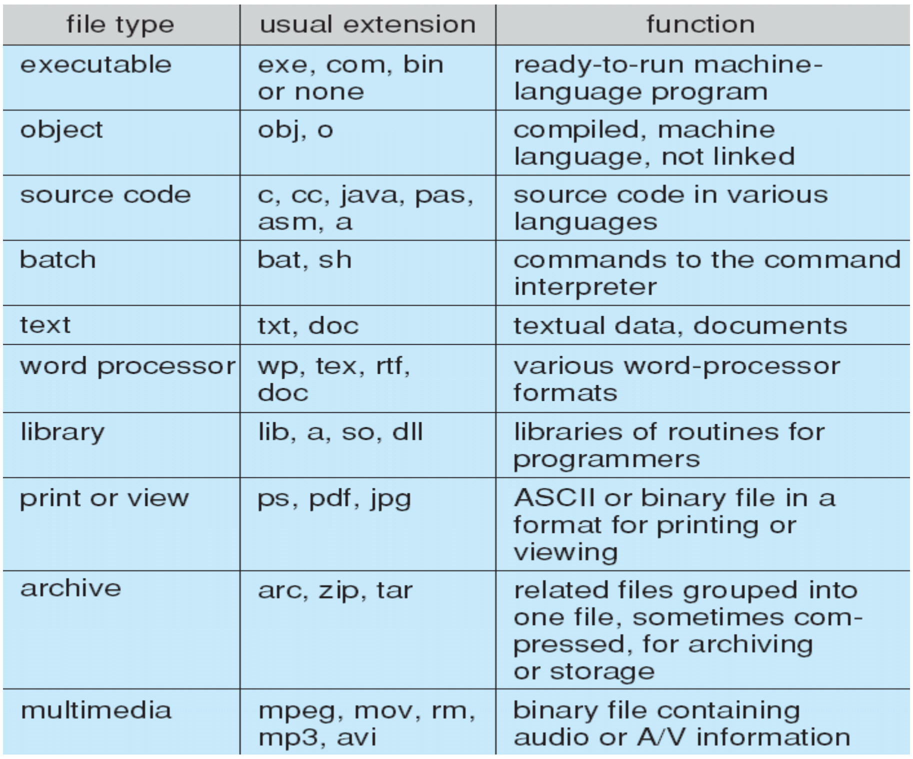
- contains process-independent information, such as the location of the file on disk, access dates, and file size. Once a file has been opened by one process, the system-wide table includes an entry for the file.

Several pieces of information are associated with an open file.

1. **File pointer:** On systems that do not include a file offset as part of the read() and write() system calls, the system must track the last read write location as a current- file-position pointer. This pointer is unique to each process operating on the file and therefore must be kept separate from the on-disk file attributes.
2. **File-open count:** As files are closed, the operating system must reuse its open- file table entries, or it could run out of space in the table. Because multiple processes may have opened a file, the system must wait for the last file to close before removing the open-file table entry. The file-open counter tracks the number of opens and closes and reaches zero on the last close. The system can then remove the entry.
3. **Disk location of the file:**Most file operations require the system to modify data within the file. The information needed to locate the file on disk is kept in memory so that the system does not have to read it from disk for each operation.
4. **Access rights:** Each process opens a file in an access mode. This information is stored on the per-process table so the operating system can allow or deny subsequent I/0 requests.

File Types

- The operating system should recognize and support file types. If an operating system recognizes the type of a file, it can then operate on the file in reasonable ways.
- A common technique for implementing file types is to include the type as part of the file name. The name is split into two parts-**a name and an extension**, usually separated by a period character
- The system uses the extension to indicate the type of the file and the type of operations that can be done on that file.



file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

File Structure

- File types also can be used to indicate the internal structure of the file. For instance source and object files have structures that match the expectations of the programs that read them. Certain files must conform to a required structure that is understood by the operating system. For example: the operating system requires that an executable file have a specific structure so that it can determine where in memory to load the file and what the location of the first instruction is.
- The operating system support multiple file structures: the resulting size of the operating system also increases. If the operating system defines five different file structures, it needs to contain the code to support these file structures.
- It is necessary to define every file as one of the file types supported by the operating system. When new applications require information structured in ways not supported by

the operating system, severe problems may result.

- Example: The Macintosh operating system supports a minimal number of file structures. It expects files to contain two parts: a resource fork and data fork.
    - **The resource fork** contains information of interest to the user.
    - **The data fork** contains program code or data

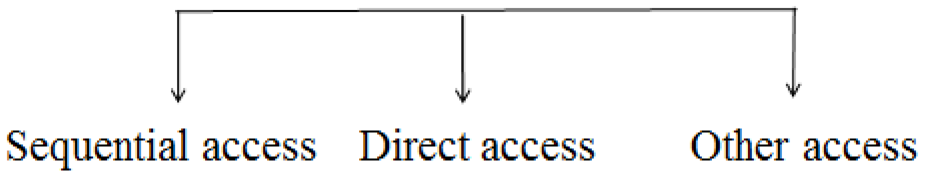
Internal File Structure

- Internally, locating an offset within a file can be complicated for the operating system. Disk systems typically have a well-defined block size determined by the size of a sector.
- All disk I/0 is performed in units of one block (physical record), and all blocks are the same size. It is unlikely that the physical record size will exactly match the length of the desired logical record.
- Logical records may even vary in length. Packing a number of logical records into physical blocks is a common solution to this problem.

ACCESS METHODS

- Files store information. When it is used, this information must be accessed and read into computer memory. The information in the file can be accessed in several ways.

- Some of the common methods are:



1. Sequential methods

- The simplest access method is sequential methods. Information in the file is processed in order, one record after the other.
- Reads and writes make up the bulk of the operations on a file.
- A read operation (next-reads) reads the next portion of the file and automatically advances a file pointer, which tracks the I/O location
- The write operation (write next) appends to the end of the file and advances to the end of the newly written material.
- A file can be reset to the beginning and on some systems, a program may be able to skip forward or backward n records for some integer n-perhaps only for n =1.

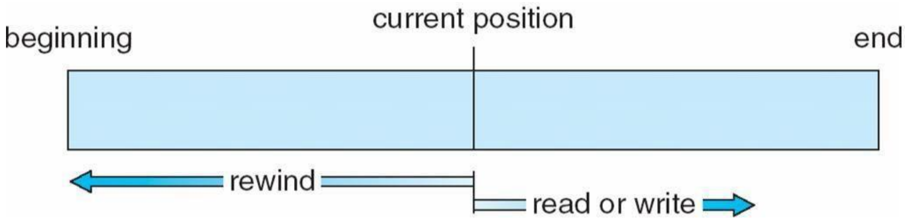
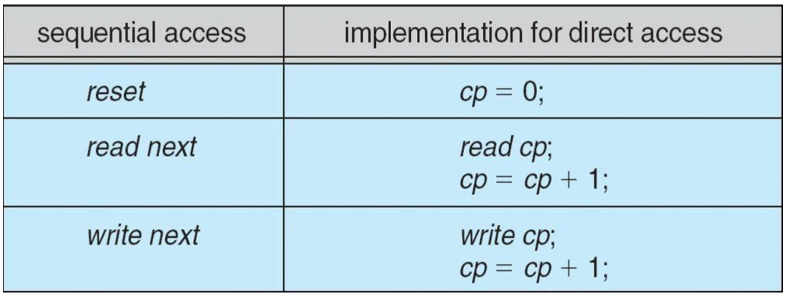


Figure: Sequential-access file.

2. Direct Access

- A file is made up of fixed length logical records that allow programs to read and write records rapidly in no particular order.
- The direct-access method is based on a disk model of a file, since disks allow random access to any file block. For direct access, the file is viewed as a numbered sequence of blocks or records.
- Example: if we may read block 14, then read block 53, and then write block 7. There are no restrictions on the order of reading or writing for a direct-access file.
- Direct-access files are of great use for immediate access to large amounts of information such as Databases, where searching becomes easy and fast.
- For the direct-access method, the file operations must be modified to include the block number as a parameter. Thus, we have read n, where n is the block number, rather than read next, and ·write n rather than write next.
- An alternative approach is to retain read next and write next, as with sequential access, and to add an operation position file to n, where n is the block number. Then, to affect a read n, we would position to n and then read next.



sequential access	implementation for direct access
reset	*cp* = 0;
read next	*read cp*; *cp* = *cp* + 1;
write next	*write cp*; *cp* = *cp* + 1;

Figure: Simulation of sequential access on a direct-access file.

3. Other Access Methods:

- Other access methods can be built on top of a direct-access method. These methods generally involve the construction of an index for the file.
- The **Index**, is like an index in the back of a book contains pointers to the various blocks. To find a record in the file, we first search the index and then use the pointer to access the file directly and to find the desired record.

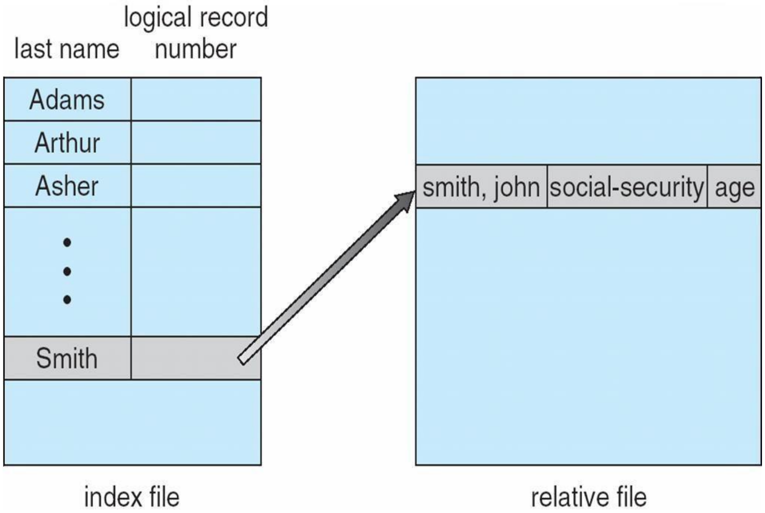


Figure: Example of index and relative files

DIRECTORY AND DISK STRUCTURE

- Files are stored on random-access storage devices, including hard disks, optical disks, and solid state (memory-based) disks.
- A storage device can be used in its entirety for a file system. It can also be subdivided for finer-grained control
- Disk can be subdivided into partitions. Each disks or partitions can be RAID protected against failure.
- Partitions also known as minidisks or slices. Entity containing file system known as a volume. Each volume that contains a file system must also contain information about the files in the system. This information is kept in entries in a **device directory** or **volume table of contents.**

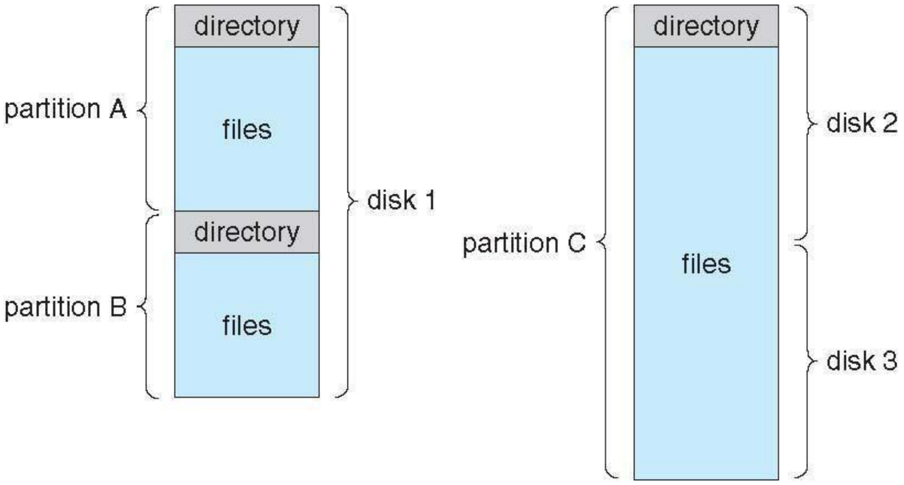


Figure: A Typical File-system Organization

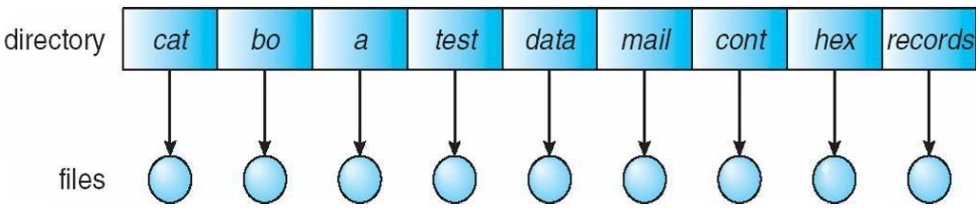
Directory Overview

- The directory can be viewed as a symbol table that translates file names into their directory entries. A directory contains information about the files including attributes location and ownership. To consider a particular directory structure, certain operations on the directory have to be considered:
    - **Search for a file**: Directory structure is searched for a particular file in directory. Files have symbolic names and similar names may indicate a relationship between files. Using this similarity it will be easy to find all whose name matches a particular pattern.
    - **Create a file**: New files needed to be created and added to the directory.
    - **Delete a file:** When a file is no longer needed, then it is able to remove it from thedirectory.
    - **List a directory**: It is able to list the files in a directory and the contents of the directory entry for each file in the list.
    - **Rename a file**: Because the name of a file represents its contents to its users, It is possible to change the name when the contents or use of the file changes. Renaming a file may also allow its position within the directory structure to be changed.
    - **Traverse the file system**: User may wish to access every directory and every file within a directory structure. To provide reliability the contents and structure of the entire file system is saved at regular intervals.

- The most common schemes for defining the logical structure of a directory are described below
    1. Single-level Directory
    2. Two-Level Directory
    3. Tree-Structured Directories
    4. Acyclic-Graph Directories
    5. General Graph Directory

1. Single-level Directory

- The simplest directory structure is the single-level directory. All files are contained in the same directory, which is easy to support and understand

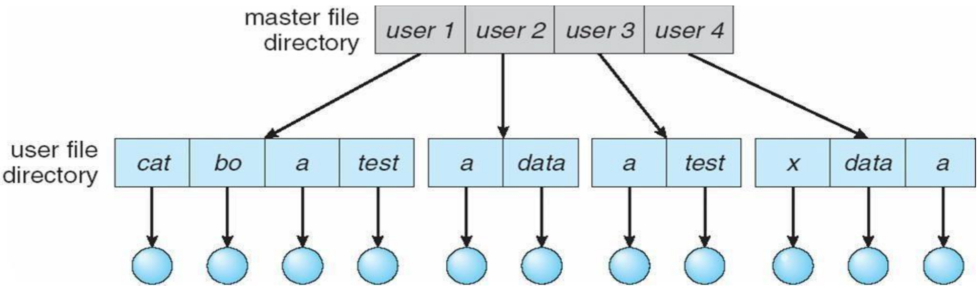


- A single-level directory has significant limitations, when the number of files increases or when the system has more than one user.

- As directory structure is single, uniqueness of file name has to be maintained, which is difficult when there are multiple users.
- Even a single user on a single-level directory may find it difficult to remember the names of all the files as the number of files increases.
- It is not uncommon for a user to have hundreds of files on one computer system and an equal number of additional files on another system. Keeping track of so many files is a daunting task.

2. Two-Level Directory

- In the two-level directory structure, each user has its own **user file directory** (UFD). The UFDs have similar structures, but each lists only the files of a single user.
- When a user refers to a particular file, only his own UFD is searched. Different users may have files with the same name, as long as all the file names within each UFD are unique.
- To create a file for a user, the operating system searches only that user's UFD to ascertain whether another file of that name exists. To delete a file, the operating system confines its search to the local UFD thus; it cannot accidentally delete another user's file that has the same name.
- When a user job starts or a user logs in, the system's Master file directory (MFD) is searched. The MFD is indexed by user name or account number, and each entry points to the UFD for that user.



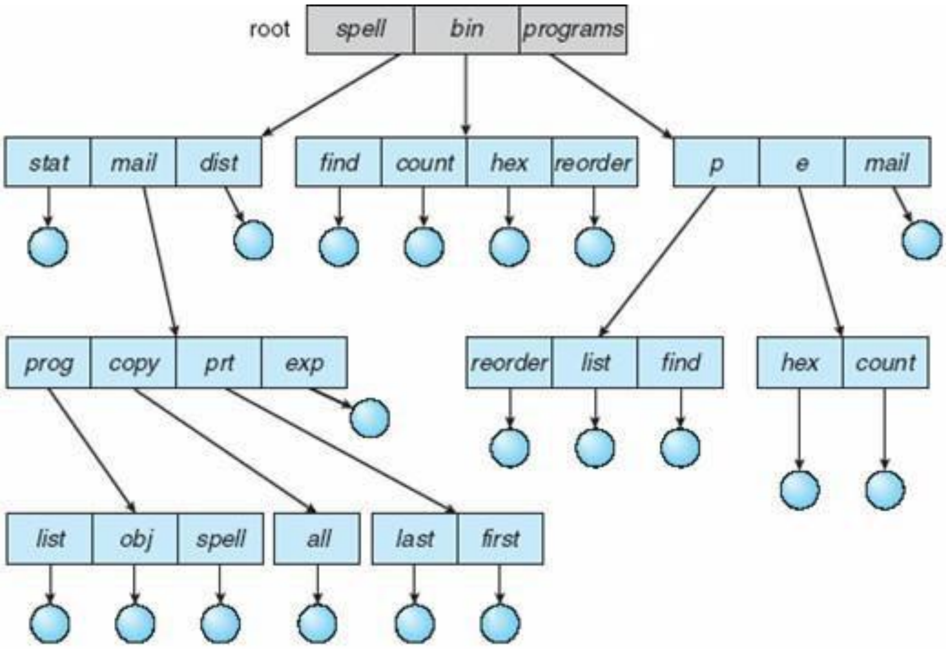
- **Advantage:**
    - No file name-collision among different users.
    - Efficient searching.
- **Disadvantage**
    - Users are isolated from one another and can't cooperate on the same task.

3. Tree Structured Directories

- A tree is the most common directory structure.
- The tree has a root directory, and every file in the system has a unique path name.
- A directory contains a set of files or subdirectories. A directory is simply another file, but it is treated in a special way.
- All directories have the same internal format. One bit in each directory entry defines the entry as a file (0) or as a subdirectory (1). Special system calls are used to create and

delete directories.

- **Two types of path-names:**
  1. Absolute path-name: begins at the root.
  2. Relative path-name: defines a path from the current directory.



How to delete directory?

1. To delete an empty directory: Just delete the directory.
2. To delete a non-empty directory:
   - First, delete all files in the directory.
   - If any subdirectories exist, this procedure must be applied recursively to them.

Advantage:

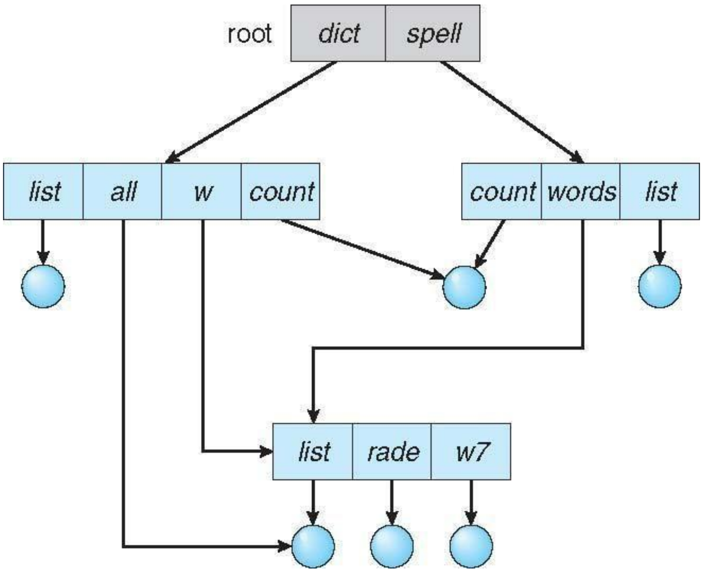
- Users can be allowed to access the files of other users.

Disadvantages:

- A path to a file can be longer than a path in a two-level directory.
- Prohibits the sharing of files (or directories).

4. Acyclic Graph Directories

- The common subdirectory should be shared. A shared directory or file will exist in the file system in two or more places at once. A tree structure prohibits the sharing of files or directories.
- An acyclic graph is a graph with no cycles. It allows directories to share subdirectories and files.



- The same file or subdirectory may be in two different directories. The acyclic graph is a natural generalization of the tree-structured directory scheme.

Two methods to implement shared-files (or subdirectories):

1. Create a new directory-entry called a link. A link is a pointer to another file (or subdirectory).
2. Duplicate all information about shared-files in both sharing directories.

Two problems:

1. A file may have multiple absolute path-names.
2. Deletion may leave dangling-pointers to the non-existent file.

Solution to deletion problem:

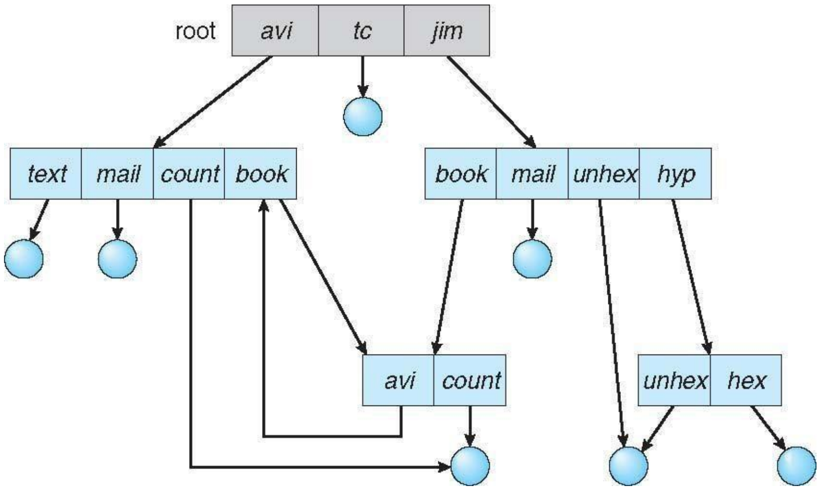
1. Use back-pointers: Preserve the file until all references to it are deleted.
2. With symbolic links, remove only the link, not the file. If the file itself is deleted, the link can be removed.

5. General Graph Directory

- **Problem:** If there are cycles, we want to avoid searching components twice.
- **Solution:** Limit the no. of directories accessed in a search.
- **Problem:** With cycles, the reference-count may be non-zero even when it is no longer possible to refer to a directory (or file). (A value of 0 in the reference count means that there are no more references to the file or directory, and the file can be deleted).
- **Solution:** Garbage-collection scheme can be used to determine when the last reference has been deleted.

Garbage collection involves

- First pass traverses the entire file-system and marks everything that can be accessed.
- A second pass collects everything that is not marked onto a list of free-space



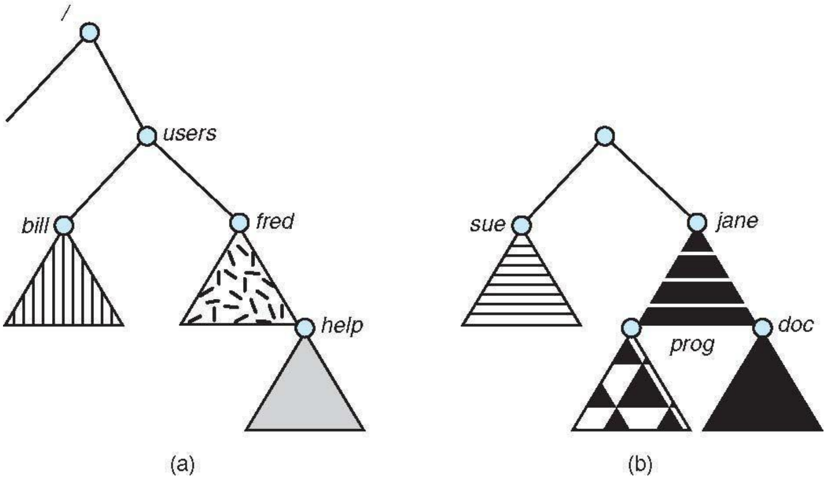
FILE SYSTEM MOUNTING

- A file must be *opened* before it is used, a file system must be *mounted* before it can be available to processes on the system
- **Mount Point:** The location within the file structure where the file system is to be attached.

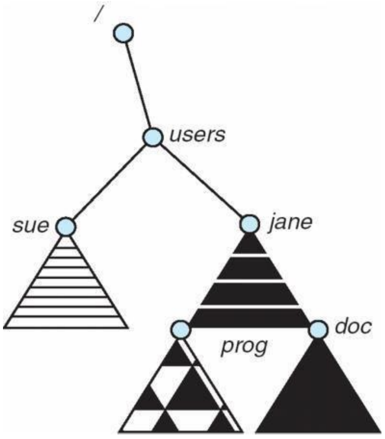
The mounting procedure:

- The operating system is given the name of the device and the mount point.
- The operating system verifies that the device contains a valid file system. It does so by asking the device driver to read the device directory and verifying that the directory has the expected format
- The operating system notes in its directory structure that a file system is mounted at the specified mount point.

To illustrate file mounting, consider the file system shown in figure. The triangles represent sub-trees of directories that are of interest



- Figure (a) shows an existing file system,
- while Figure 1(b) shows an un-mounted volume residing on */device/dsk.* At this point, only the files on the existing file system can be accessed.



- Above figure shows the effects of mounting the volume residing on */device/dsk* over */users.*
- If the volume is un-mounted, the file system is restored to the situation depicted in first Figure.

FILE SHARING

- Sharing of files on multi-user systems is desirable.
- Sharing may be done through a protection scheme.
- On distributed systems, files may be shared across a network.
- Network File-system (NFS) is a common distributed file-sharing method.

Multiple Users

File-sharing can be done in 2 ways:

1. The system can allow a user to access the files of other users by default or
2. The system may require that a user specifically grant access.

To implement file-sharing, the system must maintain more file- & directory-attributes than on a single-user system.
Most systems use concepts of file owner and group.

1. Owner

- The user who may change attributes & grant access and has the most control over the file (or directory).
- Most systems implement owner attributes by managing a list of user-names and user IDs

2. Group

- The group attribute defines a subset of users who can share access to the file.
- Group functionality can be implemented as a system-wide list of group-names and group IDs.
- Exactly which operations can be executed by group-members and other users is definable by the file's owner.
- The owner and group IDs of files are stored with the other file-attributes and can be used to allow/deny requested operations.

Remote File Systems

It allows a computer to mount one or more file-systems from one or more remote- machines.
There are three methods:

1. Manually transferring files between machines via programs like ftp.
2. Automatically DFS (Distributed file-system): remote directories are visible from a local machine.
3. Semi-automatically via www (World Wide Web): A browser is needed to gain access to the remote files, and separate operations (a wrapper for ftp) are used to transfer files.

ftp is used for both anonymous and authenticated access. **Anonymous access** allows a user to transfer files without having an account on the remote system.

Client Server Model

- Allows clients to mount remote file-systems from servers.
- The machine containing the files is called the **server**. The machine seeking access to the files is called the **client**.
- A server can serve multiple clients, and a client can use multiple servers.
- The server specifies which resources (files) are available to which clients.
- A client can be specified by a network-name such as an IP address.

Disadvantage:

- Client identification is more difficult.
- In UNIX and its NFS (network file-system), authentication takes place via the client networking information by default.
- Once the remote file-system is mounted, file-operation requests are sent to the server via the DFS protocol.

Distributed Information Systems

- Provides unified access to the information needed for remote computing.
- The DNS (domain name system) provides hostname-to-network address translations.
- Other distributed info. systems provide username/password space for a distributed facility

Failure Modes

- Local file-systems can fail for a variety of reasons such as failure of disk (containing the file-system), corruption of directory-structure & cable failure.
- Remote file-systems have more failure modes because of the complexity of network-systems.
- The network can be interrupted between 2 hosts. Such interruptions can result from hardware failure, poor hardware configuration or networking implementation issues.
- DFS protocols allow delaying of file-system operations to remote-hosts, with the hope that the remote-host will become available again.
- To implement failure-recovery, some kind of state information may be maintained on both the client and the server.

Consistency Semantics

- These represent an important criterion of evaluating file-systems that supports file- sharing. These specify how multiple users of a system are to access a shared-file simultaneously.
- In particular, they specify when modifications of data by one user will be observed by other users.
- These semantics are typically implemented as code with the file-system.

- These are directly related to the process-synchronization algorithms.
- A successful implementation of complex sharing semantics can be found in the Andrew file-system (AFS).

UNIX Semantics

- UNIX file-system (UFS) uses the following consistency semantics:
  1. Writes to an open-file by a user are visible immediately to other users who have this file opened.
  2. One mode of sharing allows users to share the pointer of current location into a file. Thus, the advancing of the pointer by one user affects all sharing users.
- A file is associated with a single physical image that is accessed as an exclusive resource.
- Contention for the single image causes delays in user processes.

Session Semantics

The AFS uses the following consistency semantics:

1. Writes to an open file by a user are not visible immediately to other users that have the same file open.
2. Once a file is closed, the changes made to it are visible only in sessions starting later. Already open instances of the file do not reflect these changes.

- A file may be associated temporarily with several (possibly different) images at the same time.
- consequently, multiple users are allowed to perform both read and write accesses concurrently on their images of the file, without delay.
- Almost no constraints are enforced on scheduling accesses.

Immutable Shared Files Semantics

- Once a file is declared as shared by its creator, it cannot be modified.
- An immutable file has 2 key properties:
  1. File-name may not be reused
  2. File-contents may not be altered.
- Thus, the name of an immutable file signifies that the contents of the file are fixed.
- The implementation of these semantics in a distributed system is simple, because the sharing is disciplined

PROTECTION

- When information is stored in a computer system, we want to keep it safe from physical damage (reliability) and improper access (protection).
- Reliability is generally provided by duplicate copies of files.
- For a small single-user system, we might provide protection by physically removing the floppy disks and locking them in a desk drawer.
- File owner/creator should be able to control what can be done and by whom.

Types of Access

- Systems that do not permit access to the files of other users do not need protection. This is too extreme, so controlled-access is needed.
- Following operations may be controlled:
  1. ***Read:*** Read from the file.
  2. ***Write:*** Write or rewrite the file.
  3. ***Execute:*** Load the file into memory and execute it.
  4. ***Append*:** Write new information at the end of the file.
  5. ***Delete:*** Delete the file and tree its space for possible reuse.
  6. ***List:*** List the name and attributes of the file.

Access Control

- Common approach to protection problem is to make access dependent on identity of user.
- Files can be associated with an ACL (access-control list) which specifies username and types of access for each user.

Problems:

1. Constructing a list can be tedious.
2. Directory-entry now needs to be of variable-size, resulting in more complicated space management.

Solution:

- These problems can be resolved by combining ACLs with an ‘owner, group, universe’ access control scheme
- To reduce the length of the ACL, many systems recognize 3 classifications of users:
  1. ***Owner:*** The user who created the file is the owner.
  2. ***Group:*** A set of users who are sharing the file and need similar access is a group.
  3. ***Universe:*** All other users in the system constitute the universe.

Other Protection Approaches

- A password can be associated with each file.
- **Disadvantages:**
    1. The no. of passwords you need to remember may become large.
    2. If only one password is used for all the files, then all files are accessible if it is discovered.
    3. Commonly, only one password is associated with all of the user's files, so protection is all-or nothing.

- In a multilevel directory-structure, we need to provide a mechanism for directory protection.
- The directory operations that must be protected are different from the File-operations:
    1. Control creation & deletion of files in a directory.
    2. Control whether a user can determine the existence of a file in a directory.

IMPLEMENTATION OF FILE SYSTEM

FILE SYSTEM STRUCTURE

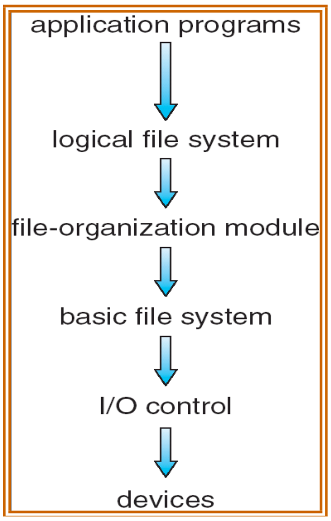
- Disks provide the bulk of secondary-storage on which a file-system is maintained.

The disk is a suitable medium for storing multiple files.

- This is because of two characteristics
  1. A disk can be rewritten in place; it is possible to read a block from the disk, modify the block, and write it back into the same place.
  2. A disk can access directly any block of information it contains. Thus, it is simple to access any file either sequentially or randomly, and switching from one file to another requires only moving the read-write heads and waiting for the disk to rotate.
- To improve I/O efficiency, I/O transfers between memory and disk are performed in units of blocks. Each block has one or more sectors. Depending on the disk drive, sector-size varies from 32 bytes to 4096 bytes. The usual size is 512 bytes.
- File-systems provide efficient and convenient access to the disk by allowing data to be stored, located, and retrieved easily
- Design problems of file-systems:
  1. Defining how the file-system should look to the user.
  2. Creating algorithms & data-structures to map the logical file-system onto the physical secondary-storage devices.

Layered File Systems:

- The file-system itself is generally composed of many different levels. Every level in design uses features of lower levels to create new features for use by higher levels.



- File system provide efficient and convenient access to the disk by allowing data to be stored, located, and retrieved easily.

A file system poses two quite different design problems.

1. The first problem is defining how the file system should look to the user. This task involves defining a file and its attributes, the operations allowed on a file, and the directory structure for organizing files.
2. The second problem is creating algorithms and data structures to map the logical file system onto the physical secondary-storage devices.

The file system itself is generally composed of many different levels. The structure shown in Figure is an example of a layered design. Each level in the design uses the features of lower levels to create new features for use by higher levels.

- The lowest level, **the *I/O control***, consists of **device drivers** and interrupts handlers to transfer information between the main memory and the disk system.

A device driver can be thought of as a translator. Its input consists of high-level commands such as "retrieve block 123."

Its output consists of low level, hardware-specific instructions that are used by the hardware controller, which interfaces the I/0 device to the rest of the system.

- The device driver usually writes specific bit patterns to special locations in the I/0 controller's memory to tell the controller which device location to act on and what actions to take.
- The **basic file system** needs only to issue generic commands to the appropriate device driver to read and write physical blocks on the disk. Each physical block is identified by its numeric disk address (for example, drive 1, cylinder 73, track 2, sector10).

This layer also manages the memory buffers and caches that hold various file-system, directory, and data blocks.

A block in the buffer is allocated before the transfer of a disk block can occur. When the buffer is full, the buffer manager must find more buffer memory or free up buffer space to allow a requested I/O to complete.

File organization

- Module knows about files and their logical blocks, as well as physical blocks. By knowing the type of file allocation used and the location of the file, the file- organization module can translate logical block addresses to physical block addresses for the basic file system to transfer.
- Each file's logical blocks are numbered from 0 (or 1) through *N*. Since the physical blocks containing the data usually do not match the logical numbers, a translation is needed to locate each block.
- The file-organization module also includes the free-space manager, which tracks unallocated blocks and provides these blocks to the file-organization module when requested.

Logical file system

- Manages metadata information. Metadata includes all of the file- system structure except the actual *data* (or contents of the files). The logical file system manages the directory structure to provide the file organization module with the information the latter needs, given a symbolic file name. It maintains file structure via **file-control blocks (FCB)**.
- FCB contains information about the file, including ownership, permissions, and location of the file contents.

File System Implementation

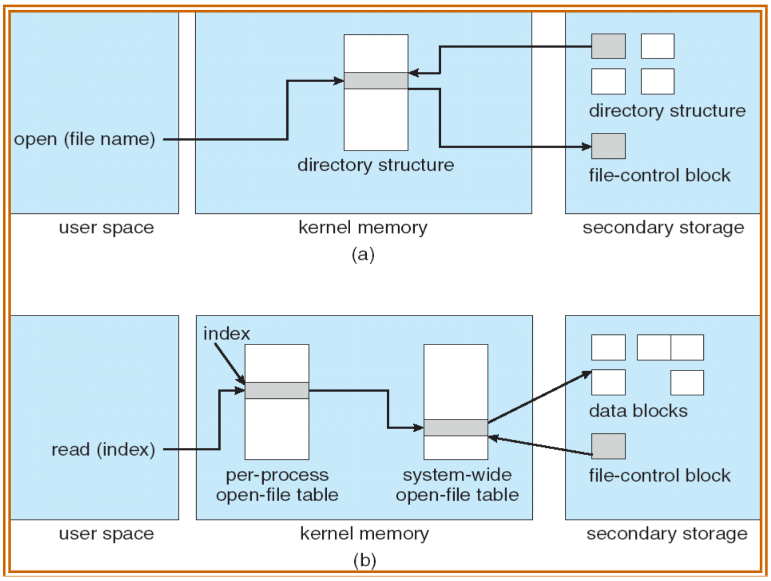
On disk, the file system may contain information about how to boot an operating system stored there, the total number of blocks, the number and location of free blocks, the directory structure, and individual files.

- **Boot Control Block**: On disk, the file system may contain information about how to boot an operating system stored there, the total number of blocks, the number and location of free blocks, the directory structure, and individual files.
- **Volume Control Block** : (per volume) contains volume (or partition) details, such as the number of blocks in the partition, the size of the blocks, a free-block count and free-block pointers and a free-FCB count and FCB pointers.
- **A directory structure** (per file system) is used to organize the files.
- **A per-file FCB** contains many details about the file. It has a unique identifier number to allow association with a directory entry.

The in-memory information is used for both file-system management and performance improvement via caching. The data are loaded at mount time, updated during file-system operations, and discarded at dismount. Several types of structures may be included.

- An in-memory mount table contains information about each mounted volume.
- An in-memory directory-structure cache holds the directory information of recently accessed directories.

- The system wide open file table contains a copy of the FCB of each open file, as well as other information.
- The per -process open file table contains a pointer to the appropriate entry in the system- wide open-file table, as well as other information.



- Buffers hold file-system blocks when they are being read from disk or written to disk.

Steps for creating a file:

1) An application program calls the logical file system, which knows the format of the directory structures
2) The logical file system allocates a new file control block(FCB)
    - If all FCBs are created at file-system creation time, an FCB is allocated from the free list
3) The logical file system then
    - Reads the appropriate directory into memory
    - Updates the directory with the new file name and FCB
    - Writes the directory back to the disk

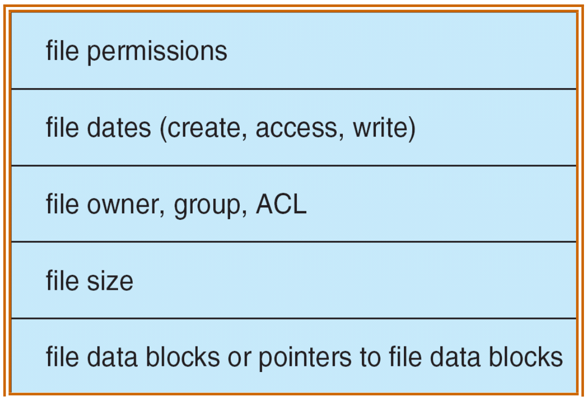
UNIX treats a directory exactly the same as a file by means of a type field in the i node Windows NT implements separate system calls for files and directories and treats directories as entities separate from files.

Steps for opening a file:

1) The function first searches the system-wide open-file table to see if the file is already in use by another process
    - If it is, a per-process open-file table entry is created pointing to the existing system-wide open-file table
    - This algorithm can have substantial overhead; consequently, parts of the directory structure are usually cached in memory to speed operations

2) Once the file is found, the FCB is copied into a system-wide open-file table in memory
    - This table also tracks the number of processes that have the file open
3) Next, an entry is made in the per-process open-file table, with a pointer to the entry in the system-wide open-file table
4) The function then returns a pointer/index to the appropriate entry in the per-process file-system table
    - All subsequent file operations are then performed via this pointer
    - UNIX refers to this pointer as the file descriptor
    - Windows refers to it as the file handle Steps for closing a file:
1) The per-process table entry is removed
2) The system-wide entry's open count is decremented
3) When all processes that have opened the file eventually close it

Any updated metadata is copied back to the disk-based directory structure. The system-wide open-file table entry is removed



file permissions

file dates (create, access, write)

file owner, group, ACL

file size

file data blocks or pointers to file data blocks

Partitions and Mounting

- Each partition can be either "raw," containing no file system, or "cooked," containing a file system.
- Raw disk is used where no file system is appropriate.
- UNIX swap space can use a raw partition, for example, as it uses its own format on disk and does not use a file system.
- Boot information can be stored in a separate partition. Again, it has its own format, because at boot time the system does not have the file-system code loaded and therefore cannot interpret the file-system format.
- Boot information is a sequential series of blocks, loaded as an image into memory.
- Execution of the image starts at a predefined location, such as the first byte. This boot loader in turn knows about the file-system structure to be able to find and load the kernel and start it executing.
- The root partition which contains the operating-system kernel and sometimes other system files, is mounted at boot time.

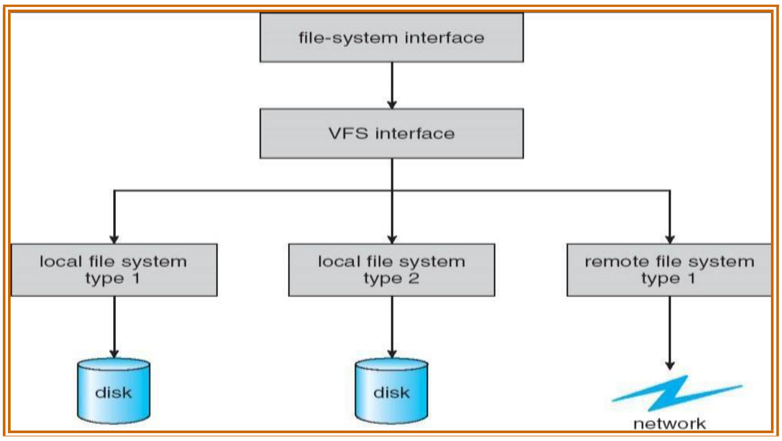
- Other volumes can be automatically mounted at boot or manually mounted later, depending on the operating system.
- After successful mount operation, the operating system verifies that the device contains a valid file system.
- It is done by asking the device driver to read the device directory and verifying that the directory has the expected format.
- If the format is invalid, the partition must have its consistency checked and possibly corrected, either with or without user intervention.
- Finally, the operating system notes in its in-memory mount table that a file system is mounted, along with the type of the file system.
- The root partition is mounted at boot time
    - It contains the operating-system kernel and possibly other system files
- Other volumes can be automatically mounted at boot time or manually mounted later
- As part of a successful mount operation, the operating system verifies that the storage device contains a valid file system
    - It asks the device driver to read the device directory and verify that the directory has the expected format
    - If the format is invalid, the partition must have its consistency checked and possibly corrected
    - Finally, the operating system notes in its in-memory mount table structure that a file system is mounted along with the type of the file system

Virtual file Systems

The file-system implementation consists of three major layers, as depicted schematically in Figure. The first layer is the file-system interface, based on the open(), read(), write(), and close() calls and on file descriptors.

The second layer is called the virtual file system (vfs) layer. The VFS layer serves two important functions:

- It separates file-system-generic operations from their implementation by defining a clean VFS interface. Several implementations for the VFS interface may coexist on the same machine, allowing transparent access to different types of file systems mounted locally.
- It provides a mechanism for uniquely representing a file throughout a network. The VFS is based on a file-representation structure, called a vnode that contains a numerical designator for a network-wide unique file. This network-wide uniqueness is required for support of network file systems.
- The kernel maintains one vnode structure for each active node.
- Thus, the VFS distinguishes local files from remote ones, and local files are further distinguished according to their file-system types.



Directory Implementation

Selection of directory allocation and directory management algorithms significantly affects the efficiency, performance, and reliability of the file system

One Approach: Direct indexing of a linear list

- Consists of a list of file names with pointers to the data blocks
- Simple to program
- Time-consuming to search because it is a linear search.
- Sorting the list allows for a binary search; however, this may complicate creating and deleting files
- To create a new file, we must first search the directory to be sure that no existing file has the same name.
- Add a new entry at the end of the directory.
- To delete a file, we search the directory for the named file and then release the space allocated to it.
- To reuse the directory entry, we can do one of several things. Mark the entry as unused.
- An alternative is to copy the last entry in the directory into the freed location and to decrease the length of the directory.
- Directory information is used frequently, and users will notice if access to it is slow.

Another Approach: List indexing via a hash function

- Takes a value computed from the file name and returns a pointer to the file name in the linear list
- Greatly reduces the directory search time
- **Can result in collisions** – situations where two file names hash to the same location
- A hash table are its generally fixed size and the dependence of the hash function on that size. (i.e., fixed number of entries).
- Each hash entry can be a linked list instead of an individual value, and we can resolve collisions by adding the new entry to the linked list.

ALLOCATION METHODS

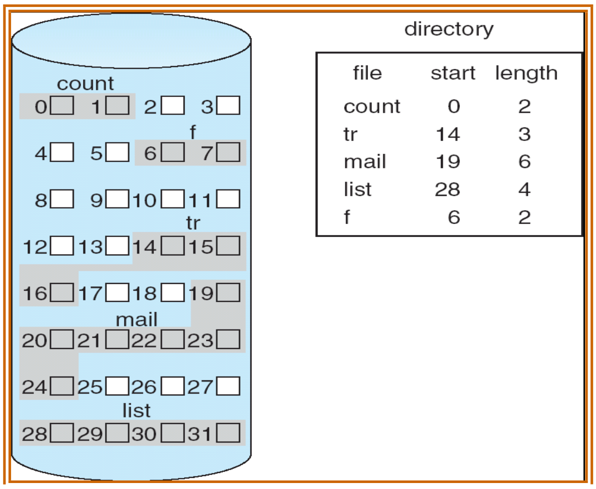
Allocation methods address the problem of allocating space to files so that disk space is utilized effectively and files can be accessed quickly.

Three methods exist for allocating disk space

- **Contiguous allocation**
- **Linked allocation**
- **Indexed allocation**

Contiguous allocation:

- Requires that each file occupy a set of contiguous blocks on the disk
- Accessing a file is easy – only need the starting location (block #) and length (number of blocks)
- Contiguous allocation of a file is defined by the disk address and length (in block units) of the first block. If the file is *n* blocks long and starts at location *b*, then it occupies blocks *b*, *b* + 1, *b* + 2, ... ,*b* + *n* - 1. The directory entry for each file indicates the address of the starting block and the length of the area allocated for this file.
- Accessing a file that has been allocated contiguously is easy. For sequential access, the file system remembers the disk address of the last block referenced and when necessary, reads the next block. For direct access to block *i* of a file that starts at block *b*, we can immediately access block *b* + *i*. Thus, both sequential and direct access can be supported by contiguous allocation.



file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Disadvantages:

1. Finding space for a new file is difficult. The system chosen to manage free space determines how this task is accomplished. Any management system can be used, but some are slower than others.
2. Satisfying a request of size *n* from a list of free holes is a problem. First fit and best fit are the most common strategies used to select a free hole from the set of available

holes.

3. The above algorithms suffer from the problem of external fragmentation.
   - As files are allocated and deleted, the free disk space is broken into pieces.
   - External fragmentation exists whenever free space is broken into chunks.
   - It becomes a problem when the largest contiguous chunk is insufficient for a request; storage is fragmented into a number of holes, none of which is large enough to store the data.
   - Depending on the total amount of disk storage and the average file size, external fragmentation may be a minor or a major problem.

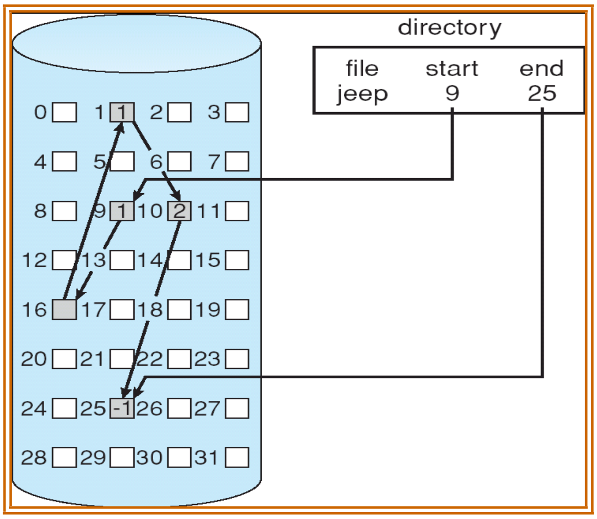
Linked Allocation:

- Solves the problems of contiguous allocation
- Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk
- The directory contains a pointer to the first and last blocks of a file
- Creating a new file requires only creation of a new entry in the directory
- Writing to a file causes the free-space management system to find a free block

- This new block is written to and is linked to the end of the file
- Reading from a file requires only reading blocks by following the pointers from block to block.

Advantages

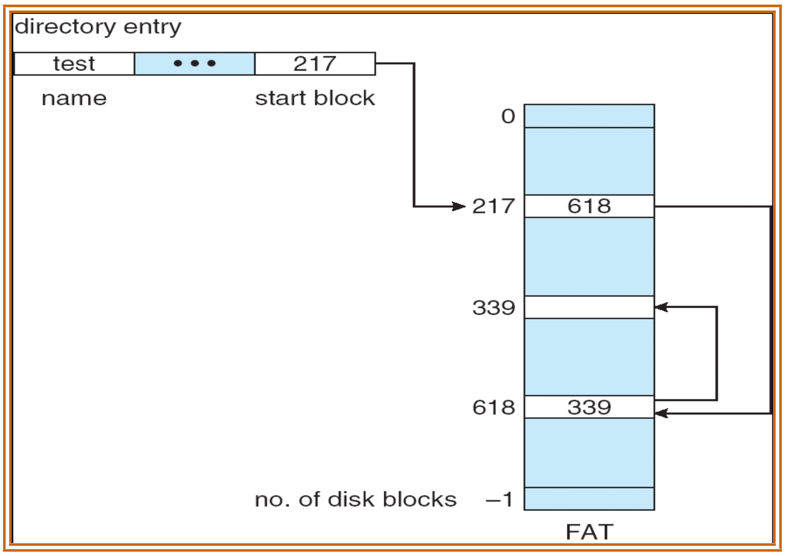
- There is no external fragmentation
- Any free blocks on the free list can be used to satisfy a request for disk space
- The size of a file need not be declared when the file is created
- A file can continue to grow as long as free blocks are available
- It is never necessary to compact disk space for the sake of linked allocation (however, file access efficiency may require it)



- Each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file.
- For example, a file of five blocks might start at block 9 and continue at block 16, then block 1, then block 10, and finally block 25. Each block contains a pointer to the next block. These pointers are not made available to the user. A disk address (the pointer) requires 4 bytes in the disk.
- To **create** a new file, we simply create a new entry ile the directory. With linked allocation, each directory entry has a pointer to the first disk block of the file. This pointer is initialized to *nil* (the end-of-list pointer value) to signify an empty file. The size field is also set to 0.
- A **write** to the file causes the free-space management system to filed a free block, and this new block is written to and is linked to the end of the file.
- To **read** a file, we simply read blocks by following the pointers from block to block. There is no external fragmentation with linked allocation, and any free block on the free- space list can be used to satisfy a request. The size of a file need not be declared when that file is created.
- A file can continue to grow as long as free blocks are available. Consequently, it is never necessary to compact disk space.
- **Disadvantages:**
  1. The major problem is that it can be used effectively only for sequential-access files. To filed the i th block of a file, we must start at the beginning of that file and follow the pointers until we get to the ith block.
  2. Space required for the pointers. Solution is clusters. Collect blocks into multiples and allocate clusters rather than blocks
  3. Reliability - the files are linked together by pointers scattered all over the disk and if a pointer were lost or damaged then all the links are lost.

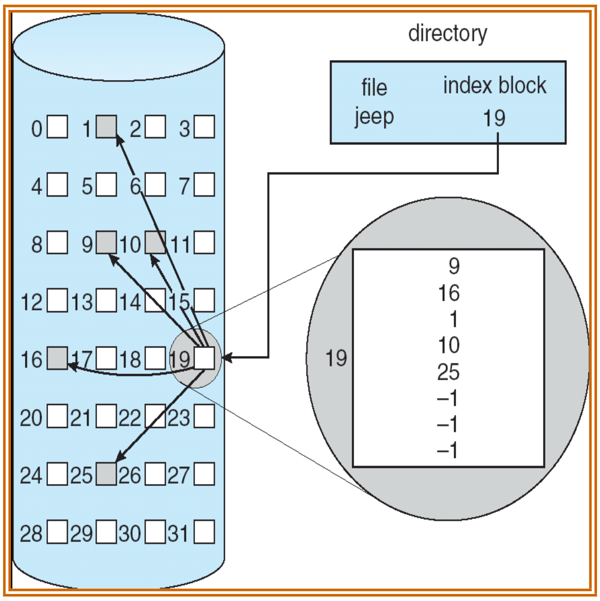
File Allocation Table:

- A section of disk at the beginning of each volume is set aside to contain the table. The table has one entry for each disk block and is indexed by block number.
- The FAT is used in much the same way as a linked list. The directory entry contains the block number of the first block of the file.
- The table entry indexed by that block number contains the block number of the next block in the file.
- The chain continues until it reaches the last block, which has a special end-of-file value as the table entry.
- An unused block is indicated by a table value of 0.
- Consider a FAT with a file consisting of disk blocks 217, 618, and 339.



Indexed allocation:

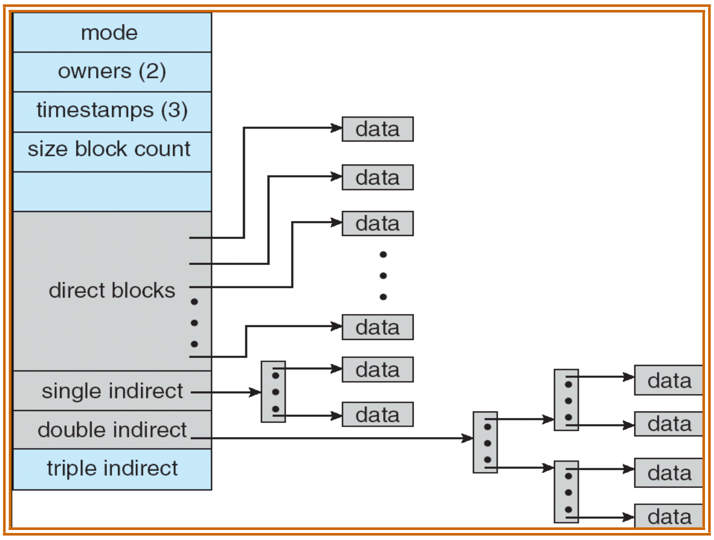
- Brings all the pointers together into one location called index block.
- Each file has its own index block, which is an array of disk-block addresses.
- The *ith* entry in the index block points to the *ith* block of the file. The directory contains the address of the index block. To find and read the *ith* block, we use the pointer in the *ith* index-block entry.
- When the file is created, all pointers in the index block are set to *nil*. When the ith block is first written, a block is obtained from the free-space manager and its address is put in the ith index- block entry.
- Indexed allocation supports direct access, without suffering from external fragmentation, because any free block on the disk can satisfy a request for more space.
- Disadvantages :
  - Suffers from some of the same performance problems as linked allocation
  - Index blocks can be cached in memory; however, data blocks may be spread all over the disk volume.
  - Indexed allocation does suffer from wasted space.
  - The pointer overhead of the index block is generally greater than the pointer overhead of linked allocation.



If the index block is too small, however, it will not be able to hold enough pointers for a large file, and a mechanism will have to be available to deal with this issue. Mechanisms for this purpose include the following:

a) **Linked scheme.** An index block is normally one disk block. Thus, it can be read and written directly by itself. To allow for large files, we can link together several index blocks. For example, an index block might contain a small header giving the name of the file and a set of the first 100 disk-block addresses. The next address (the last word in the index block) is *nil* (for a small file) or is a pointer to another index block (for a large file).

b) **Multilevel index.** A variant of linked representation uses a first-level index block to point to a set of second-level index blocks, which in turn point to the file blocks. To access a block, the operating system uses the first-level index to find a second-level index block and then uses that block to find the desired data block. This approach could be continued to a third or fourth level, depending on the desired maximum file size



c) **Combined scheme.** For eg. 15 pointers of the index block is maintained in the file's i node. The first 12 of these pointers point to **direct blocks**; that is, they contain addresses of blocks that contain data of the file. Thus, the data for small files (of no more than 12 blocks) do not need a separate index block. If the block size is 4 KB, then up to 48 KB of data can be accessed directly. The next three pointers point to indirect blocks. The first points to a **single indirect block**, which is an index block containing not data but the addresses of blocks that do contain data. The second points to a **double indirect block**, which contains the address of a block that contains the addresses of blocks that contain pointers to the actual data blocks. The last pointer contains the address of a **triple indirect block**.

Performance

- Contiguous allocation requires only one access to get a disk block. Since we can easily keep the initial address of the file in memory, we can calculate immediately the disk address of the *ith* block and read it directly.
- For linked allocation, we can also keep the address of the next block in memory and read it directly. This method is fine for sequential access. Linked allocation should not be used for an application requiring direct access.
- Indexed allocation is more complex. If the index block is already in memory, then the access can be made directly. However, keeping the index block in memory requires considerable space. If this memory space is not available, then we may have to read first the index block and then the desired data block.

Free Space Management

The space created after deleting the files can be reused. Another important aspect of disk management is keeping track of free space in memory. The list which keeps track of free space in memory is called the free-space list. To create a file, search the free-space list for the required amount of space and allocate that space to the new file. This space is then removed from the free- space list. When a file is deleted, its disk space is added to the free-space list. The free-space list, is implemented in different ways as explained below.

a) Bit Vector

- Fast algorithms exist for quickly finding contiguous blocks of a given size
- One simple approach is to use a ***bit vector***, in which each bit represents a disk block, set to 1 if free or 0 if allocated.

For example, consider a disk where blocks 2,3,4,5,8,9, 10,11, 12, 13, 17and 18 are free, and the rest of the blocks are allocated. The free-space bit map would be
0011110011111100011

- Easy to implement and also very efficient in finding the first free block or 'n' consecutive free blocks on the disk.

- The down side is that a 40GB disk requires over 5MB just to store the bitmap.

b) Linked List

a. A linked list can also be used to keep track of all free blocks.

b. Traversing the list and/or finding a contiguous block of a given size are not easy, but fortunately are not frequently needed operations. Generally the system just adds and removes single blocks from the beginning of the list.

c. The FAT table keeps track of the free list as just one more linked list on the table.

c) Grouping

a. A variation on linked list free lists. It stores the addresses of n free blocks in the first free block. The first n-1 blocks are actually free. The last block contains the addresses of another n free blocks, and so on.

b. The address of a large number of free blocks can be found quickly.

d) Counting

a. When there are multiple contiguous blocks of free space then the system can keep track of the starting address of the group and the number of contiguous free blocks.

b. Rather than keeping al list of n free disk addresses, we can keep the address of first free block and the number of free contiguous blocks that follow the first block.

c. Thus the overall space is shortened. It is similar to the extent method of allocating blocks.

e) Space Maps

a. Sun's ZFS file system was designed for huge numbers and sizes of files, directories, and even file systems.

b. The resulting data structures could be inefficient if not implemented carefully. For example, freeing up a 1 GB file on a 1 TB file system could involve updating thousands of blocks of free list bit maps if the file was spread across the disk.

c. ZFS uses a combination of techniques, starting with dividing the disk up into (hundreds of) ***Meta slabs*** of a manageable size, each having their own space map.

d. Free blocks are managed using the counting technique, but rather than write the information to a table, it is recorded in a log-structured transaction record. Adjacent free blocks are also coalesced into a larger single free block.

e. An in-memory space map is constructed using a balanced tree data structure, constructed from the log data.

f. The combination of the in-memory tree and the on-disk log provide for very fast and efficient management of these very large files and free blocks.

QUESTION BANK

1. What is a file? Distinguish between contiguous and linked allocation methods with the neat diagram.
2. Explain file allocation methods by taking an example with the neat diagram. Write the advantages and disadvantages.
3. Explain free space management. Explain typical file control block, with a neat sketch.
4. Distinguish between single level directory structure and two level directory structures. What are its advantages and disadvantages?
5. Explain the access matrix model of implementing protection in operating system.
6. For the following page reference string **1,2,3,4,1,2,5,1,2,3,4,5**. Calculate the page faults using FIFO, Optimal and LRU using 3 and 4 frames.
7. Explain Demand paging in detail.
8. For the following page reference string **7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1**. Calculate the page faults using FIFO, Optimal and LRU using 3 and 4 frames.
9. Explain copy-on-write process in virtual memory.
10. What is a page fault? with the supporting diagram explain the steps involved in handling page fault.
11. Illustrate how paging affects the system performance.
12. Explain the various types of directory structures.
13. Explain the various file attributes.
14. Explain the various file operations.
15. Explain the various mechanism of implementing file protection.

MODULE 5

SECONDARY STORAGE STRUCTURES

OVERVIEW OF MASS-STORAGE STRUCTURE

Magnetic Disks

- Magnetic disks provide the bulk of secondary storage for modern computer systems.
- Each disk platter has a flat circular shape, like a CD. Common platter diameters range from 1.8 to 5.25 inches.
- The two surfaces of a platter are covered with a magnetic material. The information stored by recording it magnetically on the platters.

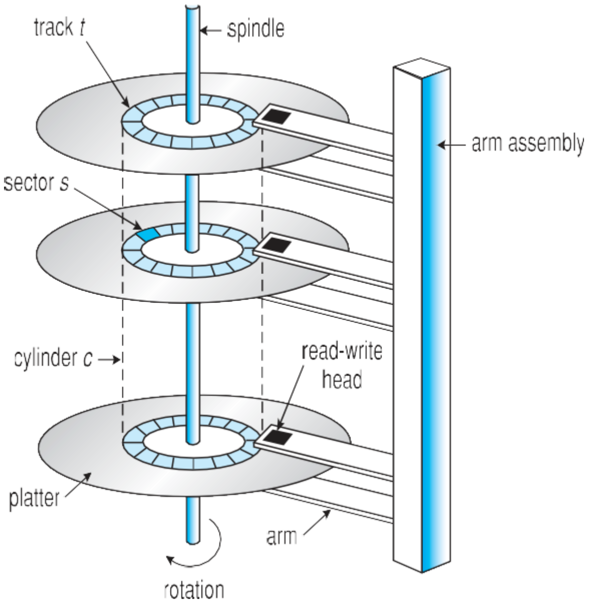


Figure: Moving-head disk mechanism

- The surface of a platter is logically divided into circular tracks, which are subdivided into sectors. Sector is the basic unit of storage. The set of tracks that are at one arm position makes up a cylinder.
- The number of cylinders in the disk drive equals the number of tracks in each platter.
- There may be thousands of concentric cylinders in a disk drive, and each track may contain hundreds of sectors.

- **Seek Time:-**Seek time is the time required to move the disk arm to the required track.
- **Rotational Latency (Rotational Delay):-** Rotational latency is the time taken for the disk to rotate so that the required sector comes under the r/w head.
- **Positioning time or random access time** is the summation of seek time and rotational delay.
- **Disk Bandwidth:-** Disk bandwidth is the total number of bytes transferred divided by total time between the first request for service and the completion of last transfer.
- **Transfer rate** is the rate at which data flow between the drive and the computer.

As the disk head flies on an extremely thin cushion of air, the head will make contact with the disk surface. Although the disk platters are coated with a thin protective layer, sometimes the head will damage the magnetic surface. This accident is called a **head crash**.

Magnetic Tapes

- Magnetic tape is a secondary-storage medium. It is a permanent memory and can hold large quantities of data.
- The time taken to access data (access time) is large compared with that of magnetic disk, because here data is accessed sequentially.
- When the nth data has to be read, the tape starts moving from first and reaches the nth position and then data is read from nth position. It is not possible to directly move to the nth position. So tapes are used mainly for backup, for storage of infrequently used information.
- A tape is kept in a spool and is wound or rewound past a read-write head. Moving to the correct spot on a tape can take minutes, but once positioned, tape drives can write data at speeds comparable to disk drives.

DISK STRUCTURE

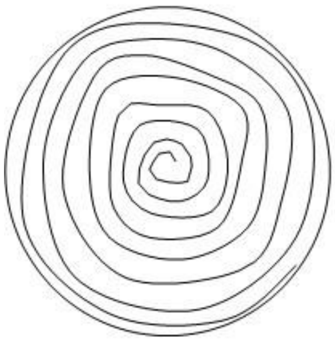
- Modern disk drives are addressed as a large one-dimensional array. The one- dimensional array of logical blocks is mapped onto the sectors of the disk sequentially.
- Sector 0 is the first sector of the first track on the outermost cylinder. The mapping proceeds in order through that track, then through the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost.

The disk structure (architecture) can be of two types –

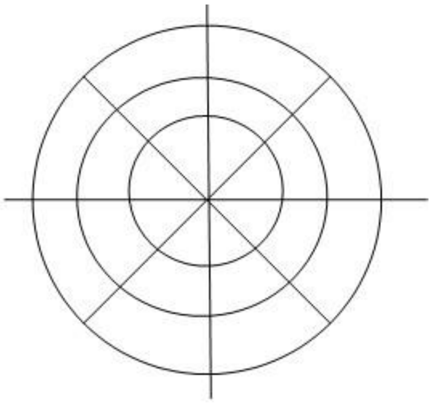
1. Constant Linear Velocity (CLV)
2. Constant Angular Velocity (CAV)

1. **CLV** – The density of bits per track is uniform. The farther a track is from the center of the disk, the greater its length, so the more sectors it can hold. As we move from outer zones to inner zones, the number of sectors per track decreases. This architecture is used in CD-ROM and DVD-ROM.

2. **CAV** – There is same number of sectors in each track. The sectors are densely packed in the inner tracks. The density of bits decreases from inner tracks to outer tracks to keep the data rate constant.



CLV



CAV

DISK ATTACHMENT

Computers can access data in two ways.

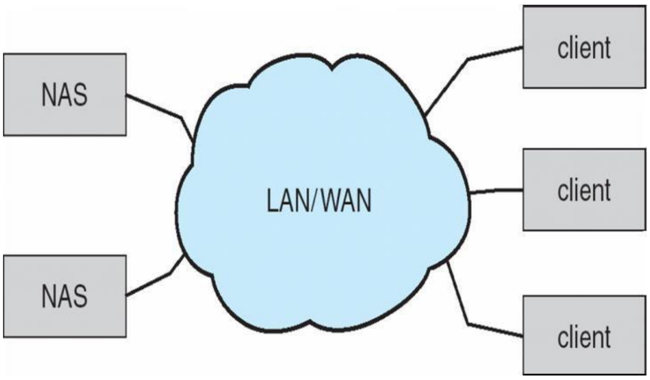
1. via I/O ports (or host-attached storage)
2. via a remote host in a distributed file system (or network-attached storage)

1. Host-Attached Storage:

- Host-attached storage is storage accessed through local I/O ports.
- Example: the typical desktop PC uses an I/O bus architecture called IDE or ATA. This architecture supports a maximum of two drives per I/O bus.
- The other cabling systems are – SATA (Serially Attached Technology Attachment), SCSI (Small Computer System Interface) and fiber channel (FC).
- SCSI is a bus architecture. Its physical medium is usually a ribbon cable. FC is a high-speed serial architecture that can operate over optical fiber or over a four- conductor copper cable. An improved version of this architecture is the basis of storage-area networks (SANs).

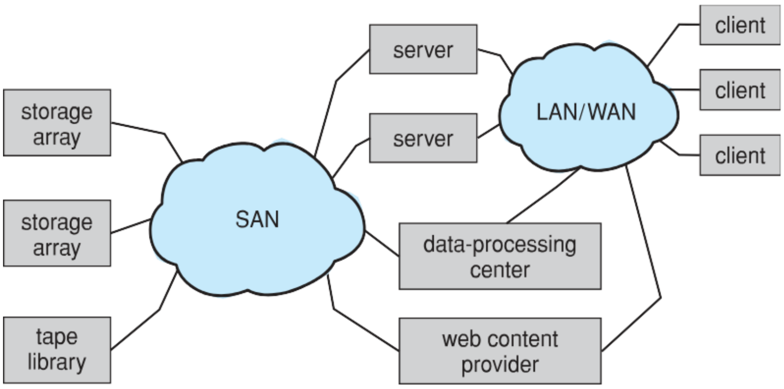
2. Network-Attached Storage

- A network-attached storage (NAS) device is a special-purpose storage system that is accessed remotely over a network as shown in the figure.
- Clients access network-attached storage via a remote-procedure-call interface. The remote procedure calls (RPCs) are carried via TCP or UDP over an IP network usually the same local-area network (LAN) carries all data traffic to the clients.
- Network- attached storage provides a convenient way for all the computers on a LAN to share a pool of storage files.



3. Storage Area Network (SAN)

- A storage-area network (SAN) is a private network connecting servers and storage units.
- The power of a SAN lies in its flexibility. Multiple hosts and multiple storage arrays can attach to the same SAN, and storage can be dynamically allocated to hosts.
- A SAN switch allows or prohibits access between the hosts and the storage. Fiber Chanel is the most common SAN interconnect.



DISK SCHEDULING

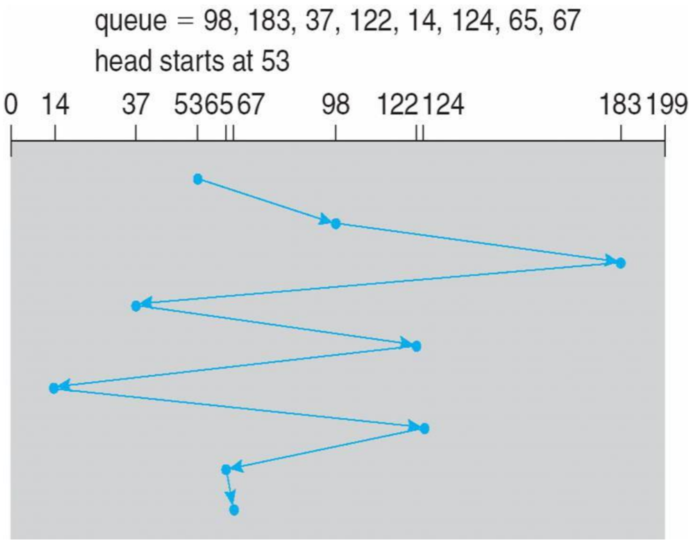
Different types of disk scheduling algorithms are as follows:

1. FCFS (First Come First Serve)
2. SSTF (Shortest Seek Time First)
3. SCAN (Elevator)
4. C-SCAN
5. LOOK
6. C-LOOK

1. FCFS scheduling algorithm:

This is the simplest form of disk scheduling algorithm. This services the request in the order they are received. This algorithm is fair but do not provide fastest service. It takes no special care to minimize the overall seek time.

Eg:- consider a disk queue with request for i/o to blocks on cylinders. 98, 183, 37, 122, 14, 124, 65, 67



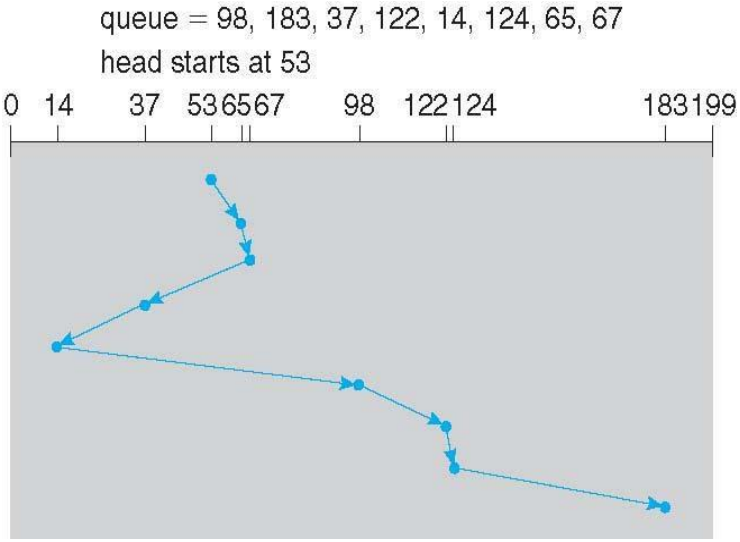
If the disk head is initially at 53, it will first move from 53 to 98 then to 183 and then to 37, 122, 14, 124, 65, 67 for a total head movement of 640 cylinders. The wild swing from 122 to 14 and then back to 124 illustrates the problem with this schedule.

Web link: https://youtu.be/hSaPhBtU_BA

2. SSTF (Shortest Seek Time First) algorithm:

This selects the request with minimum seek time from the current head position. SSTF chooses the pending request closest to the current head position.

Eg:- consider a disk queue with request for i/o to blocks on cylinders. 98, 183, 37, 122, 14, 124, 65, 67



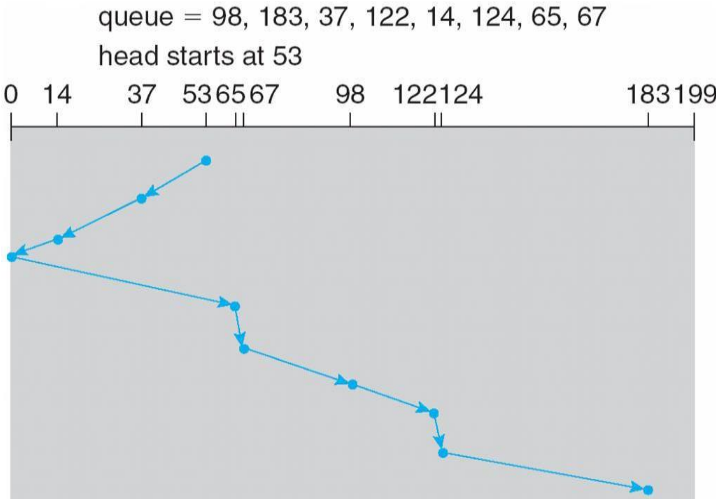
If the disk head is initially at 53, the closest is at cylinder 65, then 67, then 37 is closer than 98 to 67. So it services 37, continuing we service 14, 98, 122, 124 and finally 183. The total head movement is only 236 cylinders. SSTF is a substantial improvement over FCFS, it is not optimal.

Web link: https://youtu.be/DOnrmOGzooA

3. SCAN algorithm:

In this the disk arm starts moving towards one end, servicing the request as it reaches each cylinder until it gets to the other end of the disk. At the other end, the direction of the head movement is reversed and servicing continues. The initial direction is chosen depending upon the direction of the head.

Eg:- consider a disk queue with request for i/o to blocks on cylinders. 98, 183, 37, 122, 14, 124, 65, 67



If the disk head is initially at 53 and if the head is moving towards the outer track, it services 65, 67, 98, 122, 124 and 183. At cylinder 199 the arm will reverse and will move towards the other end of the disk servicing 37 and then 14. The SCAN is also called as elevator algorithm

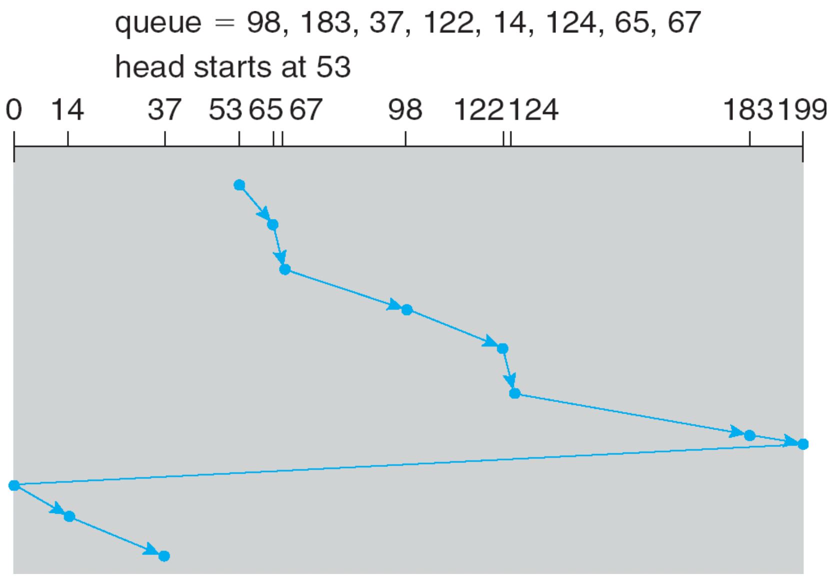
Web link: https://youtu.be/lQAYseBJPuA

4. **C-SCAN (Circular scan) algorithm**:

C-SCAN is a variant of SCAN designed to provide a more uniform wait time.

Like SCAN, C-SCAN moves the head from end of the disk to the other servicing the request along the way. When the head reaches the other end, it immediately returns to the beginning of the disk, without servicing any request on the return.

Eg:- consider a disk queue with request for i/o to blocks on cylinders. 98, 183, 37, 122, 14, 124, 65, 67.



If the disk head is initially at 53 and if the head is moving towards the outer track, it services 65, 67, 98, 122, 124 and 183. At cylinder 199 the arm will reverse and will move immediately towards the other end of the disk, then changes the direction of head and serves14 and then 37.

Note: If the disk head is initially at 53 and if the head is moving towards track 0, it services 37 and 14 first. At cylinder 0 the arm will reverse and will move immediately towards the other end of the disk servicing 65, 67, 98, 122, 124 and 183.

Web link: https://youtu.be/EgUctpbHUsQ

5. **Look Scheduling algorithm:**

Look and C-Look scheduling are different version of SCAN and C-SCAN respectively. Here the arm goes only as far as the final request in each direction. Then it reverses, withoutgoing all the way to the end of the disk. The Look and C-Look scheduling look for a requestbefore continuing to move in a given direction.

Eg:- consider a disk queue with request for i/o to blocks on cylinders. 98, 183, 37, 122, 14, 124, 65, 67

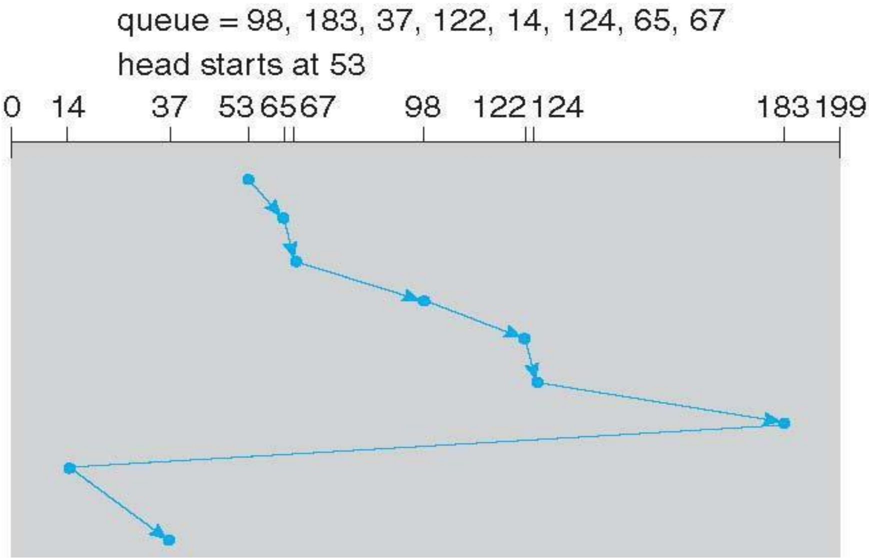


Figure: C-LOOK disk scheduling.

If the disk head is initially at 53 and if the head is moving towards the outer track, it services 65, 67, 98, 122, 124 and 183. At the final request 183, the arm will reverse and will move towards the first request 14 and then serves 37.

SELECTION OF A DISK-SCHEDULING ALGORITHM

- SSTF is commonly used and it increases performance over FCFS.
- SCAN and C-SCAN algorithm is better for a heavy load on disk. SCAN and C-SCAN have less starvation problem.
- SSTF or Look is a reasonable choice for a default algorithm.
- Selection of disk scheduling algorithm is influenced by the file allocation method, if contiguous file allocation is chosen, then FCFS is best suitable, because the files are stored in contiguous blocks and there will be limited head movements required.
- A linked or indexed file may include blocks that are widely scattered on the disk, resulting in greater head movement.
- The location of directories and index blocks is also important. Since every file must be opened to be used, and opening a file requires searching the directory structure, the directories will be accessed frequently.
- Suppose that a directory entry is on the first cylinder and a file's data are on the final cylinder. The disk head has to move the entire width of the disk. If the directory entry were on the middle cylinder, the head would have to move, at most, one-half the width. Caching the directories and index blocks in main memory can also help to reduce the disk-arm movement, particularly for read requests.

DISK MANAGEMENT

Disk Formatting

- The process of dividing the disk into sectors and filling the disk with a special data structure is called low-level formatting. Sector is the smallest unit of area that is read / written by the disk controller. The data structure for a sector typically consists of a header, a data area (usually 512 bytes in size) and a trailer. The header and trailer contain information used by the disk controller, such as a sector number and an error- correcting code (ECC).
- When the controller writes a sector of data during normal I/O, the ECC is updated with a value calculated from all the bytes in the data area. When a sector is read, the ECC is recalculated and is compared with the stored value. If the stored and calculated numbers are different, this mismatch indicates that the data area of the sector has become corrupted and that the disk sector may be bad.
- Most hard disks are low-level- formatted at the factory as a part of the manufacturing process. This formatting enables the manufacturer to test the disk and to initialize the mapping from logical block numbers to defect-free sectors on the disk.
- When the disk controller is instructed for low-level-formatting of the disk, the size of data block of all sector sit can also be told how many bytes of data space to leave between the header and trailer of all sectors. It is of sizes, such as 256, 512, and 1,024 bytes. Formatting a disk with a larger sector size means that fewer sectors can fit on eachtrack; but it also means that fewer headers and trailers are written on each track and more space is available for user data.

The operating system needs to record its own data structures on the disk. It does so in two steps i.e., Partition and logical formatting.

1. **Partition** – is to partition the disk into one or more groups of cylinders. The operating system can treat each partition as though it were a separate disk. For instance, one partition can hold a copy of the operating system's executable code, while another holds user files.
2. **Logical formatting (or creation of a file system)** - Now, the operating system stores the initial file-system data structures onto the disk. These data structures may include maps of free and allocated space (a FAT or modes) and an initial empty directory.

To increase efficiency, most file systems group blocks together into larger chunks, frequently called **clusters.**

Boot Block

When a computer is switched on or rebooted, it must have an initial program to run. This is called the bootstrap program.

The bootstrap program –

- Initializes the CPU registers, device controllers, main memory, and then starts the operating system.
- Locates and loads the operating system from the disk
- Jumps to beginning the operating-system execution.

The bootstrap is stored in read-only memory (ROM). Since ROM is read only, it cannot be infected by a computer virus. The problem is that changing this bootstrap code requires changing the ROM, hardware chips. So most systems store a tiny bootstrap loader program in the boot ROM whose only job is to bring in a full bootstrap program from disk. The full bootstrap program can be changed easily: A new version is simply written onto the disk. The full bootstrap program is stored in "the boot blocks" at a fixed location on the disk. A disk that has a boot partition is called a boot disk or system disk.

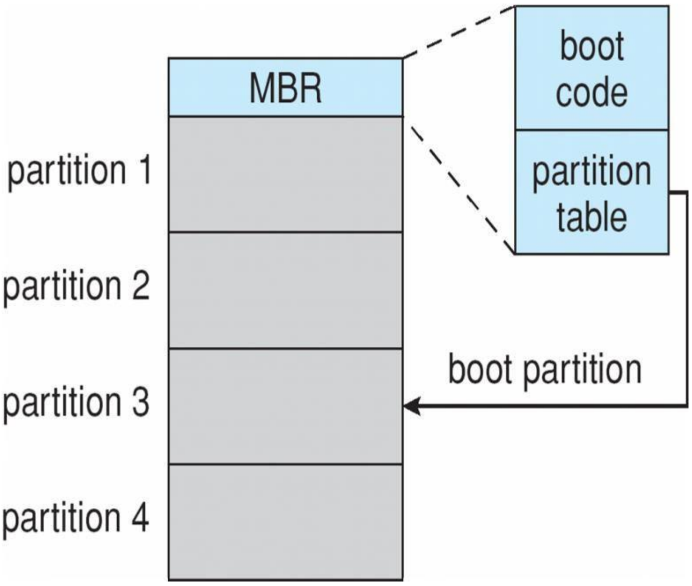


Figure: Booting from disk in Windows 2000.

The Windows 2000 system places its boot code in the first sector on the hard disk (master boot record, or MBR). The code directs the system to read the boot code from, the MBR. In addition to containing boot code, the MBR contains a table listing the partitions for the hard disk and a flag indicating which partition the system is to be booted from.

Bad Blocks

Disks are prone to failure of sectors due to the fast movement of r/w head. Sometimes the whole disk will be changed. Such group of sectors that are defective are called as **bad blocks**.

Different ways to overcome bad blocks are -

- Some bad blocks are handled manually, eg. In MS-DOS.
- Some controllers replace each bad sector logically with one of the spare sectors (extra sectors). The schemes used are sector sparing or forwarding and sector slipping.

In MS-DOS format command, scans the disk to find bad blocks. If format finds a bad block, it writes a special value into the corresponding FAT entry to tell the allocation routines not to use that block.

In SCSI disks, bad blocks are found during the low-level formatting at the factory and is updated over the life of the disk. Low-level formatting also sets aside spare sectors not visibleto the operating system. The controller can be told to replace each bad sector logically with one of the spare sectors. This scheme is known as **sector sparing or forwarding**.

A typical bad-sector transaction might be as follows:

- The operating system tries to read logical block 87.
- The controller finds that the sector is bad. It reports this finding to the operating system.
- The next time the system is rebooted, a special, command is run to tell the SCSI controller to replace the bad sector with a spare.
- After that, whenever the system requests logical block 87, the request is translated into the replacement sector's (spare) address by the controller.

Some controllers replace bad blocks by sector slipping.

Example: Suppose that logical block 17 becomes defective and the first available spare follows sector 202. Then, sector slipping remaps all the sectors from 17 to 202, moving them all down one spot. That is, sector 202 is copied into the spare, then sector 201 into 202, and then 200 into 201, and so on, until sector 18 is copied into sector 19. Slipping the sectors in this way frees up the space of sector 18, so sector 17 can be mapped to it.

PROTECTION

GOALS OF PROTECTION

- Protection is a mechanism for controlling the access of programs, processes, or users to the resources defined by a computer system. Protection ensures that only processes that have gained proper authorization from the operating system can operate on the files, memory segments, CPU, and other resources of a system.
- Protection is required to prevent mischievous, intentional violation of an access restriction by a user.

PRINCIPLES OF PROTECTION

- A key, time-tested guiding principle for protection is the ‘principle of least privilege’. It dictates that programs, users, and even systems be given just enough privileges toperform their tasks.
- An operating system provides mechanisms to enable privileges when they are needed and to disable them when they are not needed.

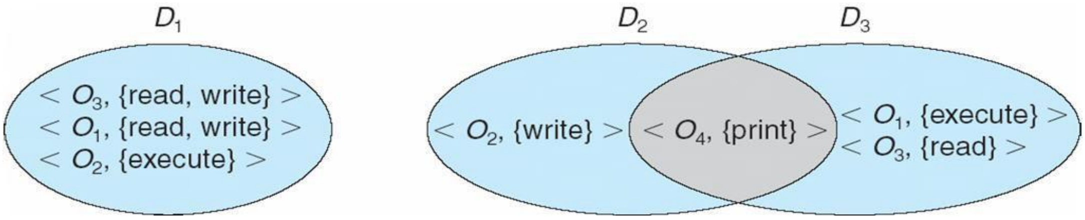
DOMAIN OF PROTECTION

- A computer system is a collection of processes and objects. Objects are both hardware objects (such as the CPU, memory segments, printers, disks, and tape drives) andsoftware objects (such as files, programs, and semaphores). Each object (resource) has a unique name that differentiates it from all other objects in the system.
- The operations that are possible may depend on the object. For example, a CPU can only be executed on. Memory segments can be read and written, whereas a CD-ROM or DVD-ROM can only be read. Tape drives can be read, written, and rewound. Data files can be created, opened, read, written, closed, and deleted; program files can be read, written, executed, and deleted.
- A process should be allowed to access only those resources for which it has authorization and currently requires to complete process

Domain Structure

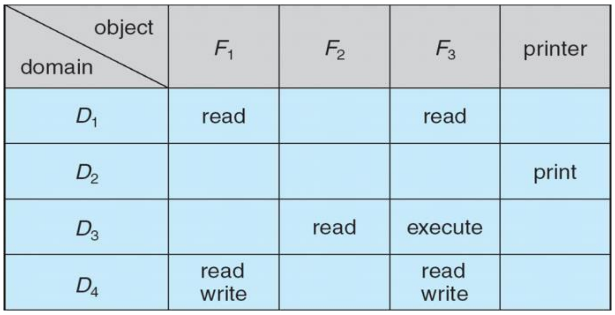
- A domain is a set of objects and types of access to these objects. Each domain is an ordered pair of <object-name, rights-set>.
- Example, if domain D has the access right <file F,{read,write}>, then all process executing in domain D can both read and write file F, and cannot perform any other operation on that object.

- Domains do not need to be disjoint; they may share access rights. For example, in below figure, we have three domains: D_1 D_2, and D_3. The access right < O_4, (print}> is shared by D_2 and D_3,it implies that a process executing in either of these two domains can print object O4.
- A domain can be realized in different ways, it can be a user, process or a procedure. ie. each user as a domain, each process as a domain or each procedure as a domain.



ACCESS MATRIX

- Our model of protection can be viewed as a matrix, called an access matrix. It is a general model of protection that provides a mechanism for protection without imposing a particular protection policy.
- The rows of the access matrix represent domains, and the columns represent objects.
- Each entry in the matrix consists of a set of access rights.
- The entry access(i,j) defines the set of operations that a process executing in domain Di can invoke on object Oj.

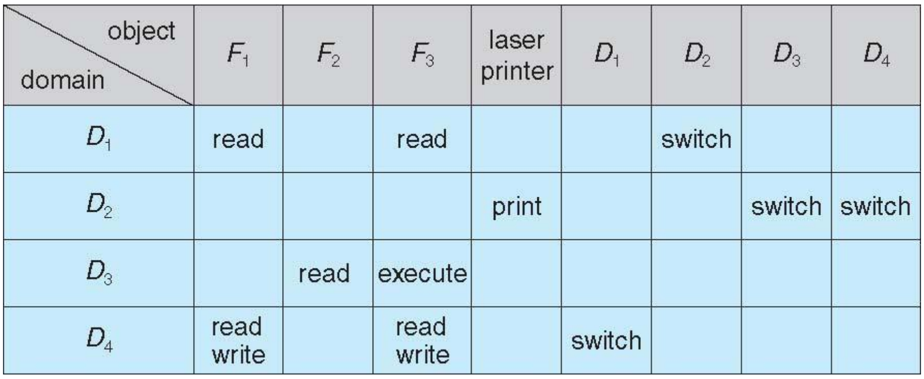


object domain	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

- In the above diagram, there are four domains and four objects—three files (F1, F2, F3) and one printer. A process executing in domain D1 can read files F1 and F3. A process executing in domain D4 has the same privileges as one executing in domain D1; but in addition, it can also write onto files F1 and F3.
- When a user creates a new object Oj, the column Oj is added to the access matrix with the appropriate initialization entries, as dictated by the creator.

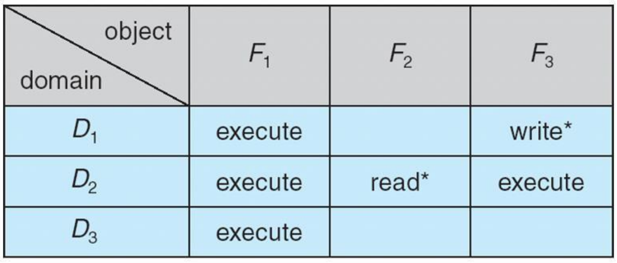
The process executing in one domain and be switched to another domain. When we switch a process from one domain to another, we are executing an operation (switch) on an object (the domain).

Domain switching from domain D_i to domain D_j is allowed if and only if the access right switch access(i,j). Thus, in the given figure, a process executing in domain D_2 can switch to domain D_3 or to domain D_4. A process in domain D_4 can switch to D_1, and one in domain D_1 can switch to domain D_2.



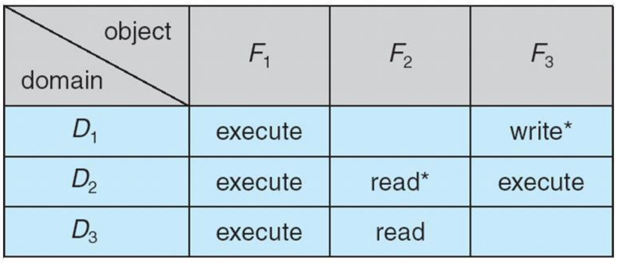
object / domain	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch
D_3		read	execute					
D_4	read write		read write		switch			

Allowing controlled change in the contents of the access-matrix entries requires three additional operations: copy, owner, and control.



object / domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute		

(a)



object / domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute	read	

(b)

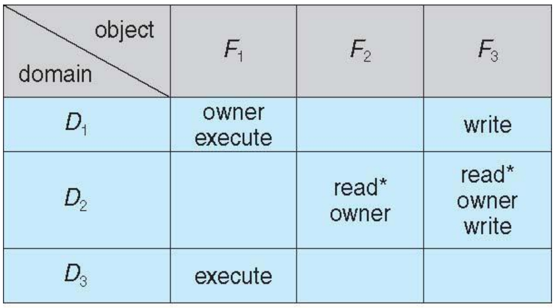
The ability to copy an access right from one domain (or row) of the access matrix to another is denoted by an asterisk (*) appended to the access right. The copy right allows the copying of the access right only within the column for which the right is defined. In the below figure, a process executing in domain D_2 can copy the read operation into any entry associated with file F_2. Hence, the access matrix of figure (a) can be modified to the access matrix shown in figure (b).

This scheme has two variants:

1. A right is copied from access(i,j) to access(k,j); it is then removed from access(i,j). This action is a transfer of a right, rather than a copy.
2. Propagation of the copy right- limited copy. Here, when the right R* is copied from access(i,j) to access(k,j), only the right R (not R*) is created. A process executing in domain D_k cannot further copy the right R.

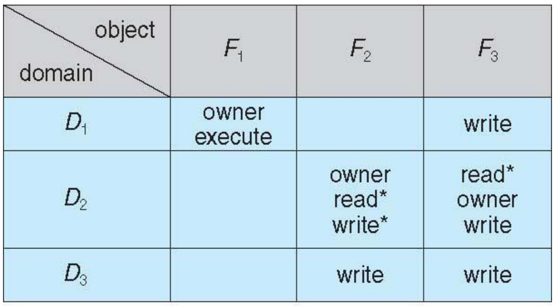
We also need a mechanism to allow addition of new rights and removal of some rights. The owner right controls these operations. If access(i,j) includes the owner right, then a process executing in domain Di, can add and remove any right in any entry in column j.

For example, in below figure (a), domain D1 is the owner of F1, and thus can add and delete any valid right in column F1. Similarly, domain D2 is the owner of F2 and F3 and thus can add and remove any valid right within these two columns. Thus, the access matrix of figure(a) can be modified to the access matrix shown in figure(b) as follows.



object domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		read* owner	read* owner write
D_3	execute		

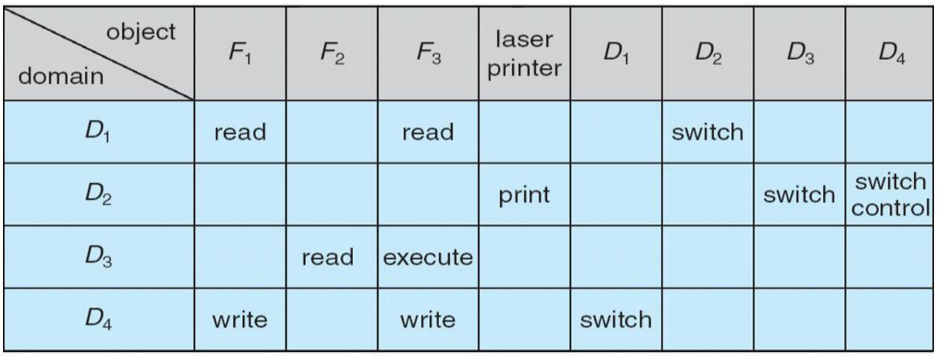
(a)



object domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		owner read* write*	read* owner write
D_3		write	write

(b)

A mechanism is also needed to change the entries in a row. If access(i,j) includes the control right, then a process executing in domain D_i, can remove any access right from row j. For example, in figure, we include the control right in access(D_3, D_4). Then, a process executing in domain D_3 can modify domain D_4.



object domain	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch control
D_3		read	execute					
D_4	write		write		switch			

QUESTION BANK

1. Explain the access matrix model of implementing protection in operating sytem.
2. Explain the following disk scheduling algorithm in brief with examples.
   i. FCFS scheduling
   ii. SSTF scheduling
   iii. SCAN scheduling
   iv. LOOK scheduling
3. Explain the components of LINUX system with a neat diagram.
4. Explain the way process is managed in LINUX platform
5. List the different disk scheduling techniques, explain any two scheduling, consider the following disk queue requests. 98,183,37,22,14,124,65,67.
6. What is an access matrix? Explain the different methods of implementing access matrix.
7. Explain bad-block recovery in disk?
8. Explain the design principle of LINUX.
9. Explain the process management in Linux platform.
10. Explain the interprocess communication mechanisms in Linux.
11. Explain the various Disk Scheduling algorithms with example.
12. Explain access matrix method of system protection
13. With a neat diagram explain in detail components of a Linux system.
14. Explain the different IPC mechanisms available in Linux.
15. Explain process scheduling in a Linux system.

BGS COLLEGE OF ENGINEERING & TECHNOLOGY

Mahalakshmipuram, West of Chord Road, Bengaluru-560086
(Approved by AICTE, New Delhi and Affiliated to VTU, Belagavi)

ACADEMIC YEAR 2023-24

DIGITAL DESIGN AND COMPUTER ORGANIZATION

For Third Semester B.E [VTU/CBCS, 2023-2024 Syllabus]

Subject Code: B C S 3 0 2

Digital Design and Computer Organization		Semester	3
Course Code	BCS302	CIE Marks	50
Teaching Hours/Week (L:T:P: S)	3:0:2:0	SEE Marks	50
Total Hours of Pedagogy	40 hours Theory + 20 Hours of Practicals	Total Marks	100
Credits	04	Exam Hours	3
Examination nature (SEE)	Theory		

Course objectives:

- To demonstrate the functionalities of binary logic system
- To explain the working of combinational and sequential logic system
- To realize the basic structure of computer system
- To illustrate the working of I/O operations and processing unit

Teaching-Learning Process (General Instructions)

These are sample Strategies; that teachers can use to accelerate the attainment of the various course outcomes.

1. Chalk and Talk
2. Live Demo with experiments
3. Power point presentation

MODULE-1 **8 Hr**

Introduction to Digital Design: Binary Logic, Basic Theorems And Properties Of Boolean Algebra, Boolean Functions, Digital Logic Gates, Introduction, The Map Method, Four-Variable Map, Don't-Care Conditions, NAND and NOR Implementation, Other Hardware Description Language – Verilog Model of a simple circuit.

Text book 1: 1.9, 2.4, 2.5, 2.8, 3.1, 3.2, 3.3, 3.5, 3.6, 3.9

MODULE-2 **8 Hr**

Combinational Logic: Introduction, Combinational Circuits, Design Procedure, Binary Adder- Subtractor, Decoders, Encoders, Multiplexers. HDL Models of Combinational Circuits – Adder, Multiplexer, Encoder. **Sequential Logic**: Introduction, Sequential Circuits, Storage Elements: Latches, Flip-Flops.

Text book 1: 4.1, 4.2, 4.4, 4.5, 4.9, 4.10, 4.11, 4.12, 5.1, 5.2, 5.3, 5.4.

MODULE-3 **8 Hr**

Basic Structure of Computers: Functional Units, Basic Operational Concepts, Bus structure, Performance – Processor Clock, Basic Performance Equation, Clock Rate, Performance Measurement.**Machine Instructions and Programs:** Memory Location and Addresses, Memory Operations, Instruction and Instruction sequencing, Addressing Modes.

Text book 2: 1.2, 1.3, 1.4, 1.6, 2.2, 2.3, 2.4, 2.5

MODULE-4 **8 Hr**

Input/output Organization: Accessing I/O Devices, Interrupts – Interrupt Hardware, Enabling and Disabling Interrupts, Handling Multiple Devices, Direct Memory Access: Bus Arbitration, Speed, size and Cost of memory systems. Cache Memories – Mapping Functions.

Text book 2: 4.1, 4.2.1, 4.2.2, 4.2.3, 4.4, 5.4, 5.5.1

MODULE-5 **8 Hr**

Basic Processing Unit: Some Fundamental Concepts: Register Transfers, Performing ALU operations, fetching a word from Memory, Storing a word in memory. Execution of a Complete Instruction. **Pipelining:** Basic concepts, Role of Cache memory, Pipeline Performance.

Text book 2: 7.1, 7.2, 8.1

PRACTICAL COMPONENT OF IPCC

Sl.NO	Experiments Simulation packages preferred: Multisim, Modelsim, PSpice or any other relevant
1	Given a 4-variable logic expression, simplify it using appropriate technique and simulate the same using basic gates.
2	Design a 4 bit full adder and subtractor and simulate the same using basic gates.
3	Design Verilog HDL to implement simple circuits using structural, Data flow and Behavioural model.
4	Design Verilog HDL to implement Binary Adder-Subtractor – Half and Full Adder, Half and Full Subtractor.
5	Design Verilog HDL to implement Decimal adder.
6	Design Verilog program to implement Different types of multiplexer like 2:1, 4:1 and 8:1.
7	Design Verilog program to implement types of De-Multiplexer.
8	Design Verilog program for implementing various types of Flip-Flops such as SR, JK and D.

Course outcomes (Course Skill Set):

At the end of the course, the student will be able to:

CO1: Apply the K–Map techniques to simplify various Boolean expressions.

CO2: Design different types of combinational and sequential circuits along with Verilog programs.

CO3: Describe the fundamentals of machine instructions, addressing modes and Processor performance.

CO4: Explain the approaches involved in achieving communication between processor and I/O devices.

CO5:Analyze internal Organization of Memory and Impact of cache/Pipelining on Processor Performance.

Assessment Details (both CIE and SEE)

The weightage of Continuous Internal Evaluation (CIE) is 50% and for Semester End Exam (SEE) is 50%. The minimum passing mark for the CIE is 40% of the maximum marks (20 marks out of 50) and for the SEE minimum passing mark is 35% of the maximum marks (18 out of 50 marks). A student shall be deemed to have satisfied the academic requirements and earned the credits allotted to each subject/ course if the student secures a minimum of 40% (40 marks out of 100) in the sum total of the CIE (Continuous Internal Evaluation) and SEE (Semester End Examination) taken together.

CIE for the theory component of the IPCC (maximum marks 50)

- IPCC means practical portion integrated with the theory of the course.
- CIE marks for the theory component are **25 marks** and that for the practical component is **25 marks.**
- 25 marks for the theory component are split into **15 marks** for two Internal Assessment Tests (Two Tests, each of 15 Marks with 01-hour duration, are to be conducted) and **10 marks** for other

assessment methods mentioned in 22OB4.2. The first test at the end of 40-50% coverage of the syllabus and the second test after covering 85-90% of the syllabus.

- Scaled-down marks of the sum of two tests and other assessment methods will be CIE marks for the theory component of IPCC (that is for **25 marks)**.
- The student has to secure 40% of 25 marks to qualify in the CIE of the theory component of IPCC.

CIE for the practical component of the IPCC

- **15 marks** for the conduction of the experiment and preparation of laboratory record, and **10 marks** for the test to be conducted after the completion of all the laboratory sessions.
- On completion of every experiment/program in the laboratory, the students shall be evaluated including viva-voce and marks shall be awarded on the same day.
- The CIE marks awarded in the case of the Practical component shall be based on the continuous evaluation of the laboratory report. Each experiment report can be evaluated for 10 marks. Marks of all experiments' write-ups are added and scaled down to **15 marks**.
- The laboratory test **(duration 02/03 hours)** after completion of all the experiments shall be conducted for 50 marks and scaled down to **10 marks.**
- Scaled-down marks of write-up evaluations and tests added will be CIE marks for the laboratory component of IPCC for **25 marks**.
- The student has to secure 40% of 25 marks to qualify in the CIE of the practical component of the IPCC.

SEE for IPCC

Theory SEE will be conducted by University as per the scheduled timetable, with common question papers for the course (**duration 03 hours**)

1. The question paper will have ten questions. Each question is set for 20 marks.
2. There will be 2 questions from each module. Each of the two questions under a module (with a maximum of 3 sub-questions), **should have a mix of topics** under that module.
3. The students have to answer 5 full questions, selecting one full question from each module.
4. Marks scored by the student shall be proportionally scaled down to 50 Marks

The theory portion of the IPCC shall be for both CIE and SEE, whereas the practical portion will have a CIE component only. Questions mentioned in the SEE paper may include questions from the practical component.

Suggested Learning Resources:

Books

1. M. Morris Mano & Michael D. Ciletti, Digital Design With an Introduction to Verilog Design, 5e, Pearson Education.

2. Carl Hamacher, ZvonkoVranesic, SafwatZaky, Computer Organization, 5th Edition, Tata McGraw Hill.

Web links and Video Lectures (e-Resources):

https://cse11-iiith.vlabs.ac.in/

Activity Based Learning (Suggested Activities in Class)/ Practical Based learning

Assign the group task to Design the various types of counters and display the output accordingly

Assessment Methods

- Lab Assessment (25 Marks)
- GATE Based Aptitude Test

MODULE 1
INTRODUCTION TO DIGITAL DESIGN

Binary Logic
Basic Theorems And Properties Of Boolean Algebra
Boolean Functions
Digital Logic Gates
The Map Method
Four-Variable Map, Don’t-Care Conditions
NAND and NOR Implementation
Other Hardware Description Language – Verilog Model of a simple circuit.

MODULE-1:

Introduction to Digital Design: Binary Logic, Basic Theorems And Properties Of Boolean Algebra, Boolean Functions, Digital Logic Gates, Introduction, The Map Method, Four-Variable Map, Don't-Care Conditions, NAND and NOR Implementation, Other Hardware Description Language – Verilog Model of a simple circuit.

Text book 1: 1.9, 2.4, 2.5, 2.8, 3.1, 3.2, 3.3, 3.5, 3.6, 3.9

1.1 Binary logic

- Binary logic deals with variables that take on two discrete values and with operationsthat assume logical meaning.
- The two values the variables assume may be called by different names (*true* and *false, yes* and *no*, etc.), but, it is convenient tothink in terms of bits and assign the values 1 and 0.
- The binary logic introduced is equivalent to an algebra called Boolean algebra.

Define binary Logic? Explain Basic Gates with Truth Table.

Definition of Binary Logic

- Binary logic consists of binary variables and a set of logical operations.
- The variables aredesignated by letters of the alphabet, such as *A*, *B*, *C*, *x*, *y*, *z*, etc., with each variable havetwo and only two distinct possible values: 1 and 0.
- There are three basic logical operations: AND, OR, and NOT. Each operation produces a binary result, denoted by *z*.
  1. **AND:** This operation is represented by a dot or by the absence of an operator.
     For example, $x \cdot y = z$ or $xy = z$ is read "*x* AND *y* is equal to *z*." The logical operation AND is interpreted to mean that $z = 1$ if and only if $x = 1$ and $y = 1$; otherwise $z = 0$. (Remember that *x*, *y*, and *z* are binary variables and can be equal either to 1 or 0, and nothing else.) The result of the operation $x \cdot y$ is *z*.
  2. **OR:** This operation is represented by a plus sign. For example, $x + y = z$ is read "*x* OR *y* is equal to *z*," meaning that $z = 1$ if $x = 1$ or if $y = 1$ or if both $x = 1$ and $y = 1$. If both $x = 0$ and $y = 0$, then $z = 0$.
  3. **NOT:** This operation is represented by a prime (sometimes by an overbar).

For example, x' = z (or x' = z) is read "not x is equal to z," meaning that z is what x is not.

In other words, if x = 1, then z = 0, but if x = 0, then z = 1. The NOT operation is also referred to as the complement operation, since it changes a 1 to 0 and a 0 to 1, i.e., the result of complementing 1 is 0, and vice versa.

Binary logic resembles binary arithmetic, and the operations AND and OR havesimilarities to multiplication and addition, respectively.

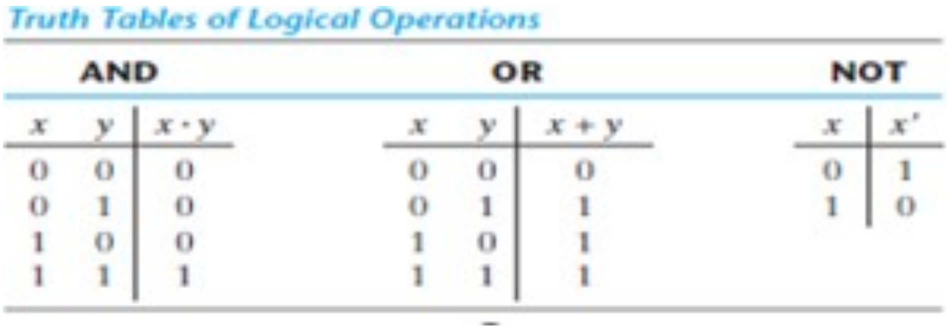
In fact, the symbols used for AND and OR are the same as those used for multiplication and addition.

However, **binary logic should not be confused with binary arithmetic.** One should realize that an arithmetic variable designates a number that may consist of many digits.

A logic vari- able is always either 1 or 0.

For example, in binary arithmetic, we have 1 + 1 = 10 (read"one plus one is equal to 2"), whereas in binary logic, we have 1 + 1 = 1 .

- For each combination of the values of x and y, there is a value of z specified by the definition of the logical operation. Definitions of logical operations may be listed in a compact form called *truth tables*. A truth table is a table of all possible combinations of the variables, showing the relation between the values that the variables may take and the result of the operation.
- The truth tables for the operations AND and OR with variables x and y are obtained by listing all possible values that the variables may have whencombined in pairs.
- For each combination, the result of the operation is then listed in a separate row. The truth tables for AND, OR, and NOT are given in Fig 1.1.



Truth Tables of Logical Operations

AND

x	y	$x \cdot y$
0	0	0
0	1	0
1	0	0
1	1	1

OR

x	y	$x + y$
0	0	0
0	1	1
1	0	1
1	1	1

NOT

x	x'
0	1
1	0

FIGURE 1.1 Truth table

Logic Gates

- Logic gates are electronic circuits that operate on one or more input signals to produce an output signal.
- Electrical signals such as voltages or currents exist as analog signals having values over a given continuous range, say, 0 to 3 V, but in a digital system these voltages are interpreted to be either of two recognizable values, 0 or 1.
- Voltage-operated logic circuits respond to two separate voltage levels that represent abinary variable equal to logic 1 or logic 0.
- For example, a particular digital system may define logic 0 as a signal equal to 0 V and logic 1 as a signal equal to 3 V. In practice, each voltage level has an acceptable range, as shown in Fig. 1.2.

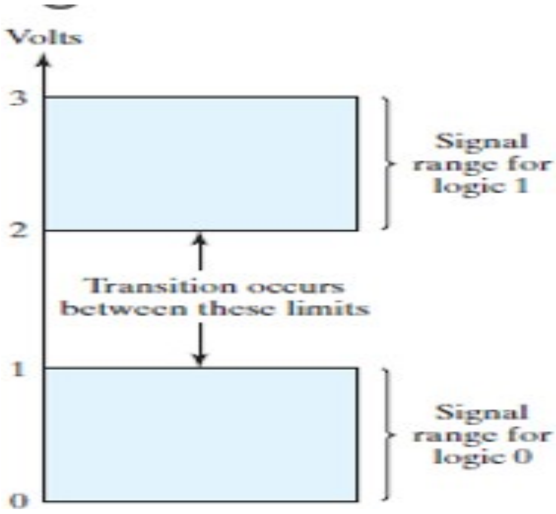
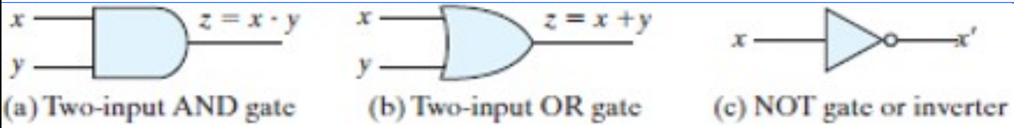


FIGURE 1.2 Signal levels for binary logic values

- The input terminals of digital circuits accept binary signals within the allowable range and respond at the output terminals with binary signals that fall within the specified range.
- The intermediate region between the allowed regions is crossed only during a state transition. Any desired information for computing or control can be operated on by passing binary signals through various combinations of logic gates, with each signal representing a particular binary variable.
- When the physical signal is in a particular range it is interpreted to be either a 0 or a 1. The graphic symbols used to designate the three types of gates are shown in Fig. 1.3.



(a) Two-input AND gate (b) Two-input OR gate (c) NOT gate or inverter

FIGURE 1.3 Symbols for digital logic circuits

The input signals *x* and *y* in the AND and OR gates may exist in one of four possible states: 00, 10, 11, or 01. These input signals are shown in Fig. 1.4 together with the corresponding output signal f each gate.

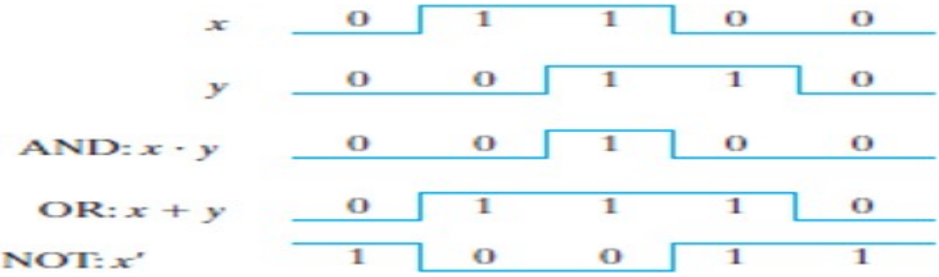
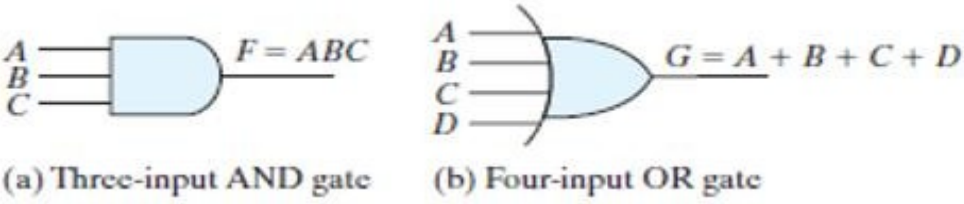


FIGURE 1.4 Input-output signals for gates

The timing diagrams illustrate the idealized response of each gate to the four input signal combinations.

- The horizontal axis of the timing diagram represents the time, and the vertical axis shows the signal as it changes between the two possible voltage levels.
- In reality, the transitions between logic values occur quickly, but not instantaneously. The low level represents logic 0, the high level logic 1.
- The AND gate responds with a logic 1 output signal when both input signals are logic 1.
- The OR gate responds with a logic 1 output signal if any input signal is logic 1.
- The NOT gate is commonly referred to as an inverter. AND and OR gates may have more than two inputs. An AND gate with three inputs and an OR gate with four inputs are shown in Fig. 1.5.



(a) Three-input AND gate (b) Four-input OR gate

FIGURE 1.5 Gates with multiple inputs

- The three-input AND gate responds with logic 1 output if all three inputs are logic 1. The output produces logic 0 if any input is logic 0.
- The four-input OR gate responds with logic 1 if any input is logic1; its output becomes logic 0 only when all inputs are logic 0.

State and Prove the Basic theorems and Postulates of Boolean Algebra

1.2 BASIC THEOREMS AND PROPERTIES OF BOOLEAN ALGEBRA

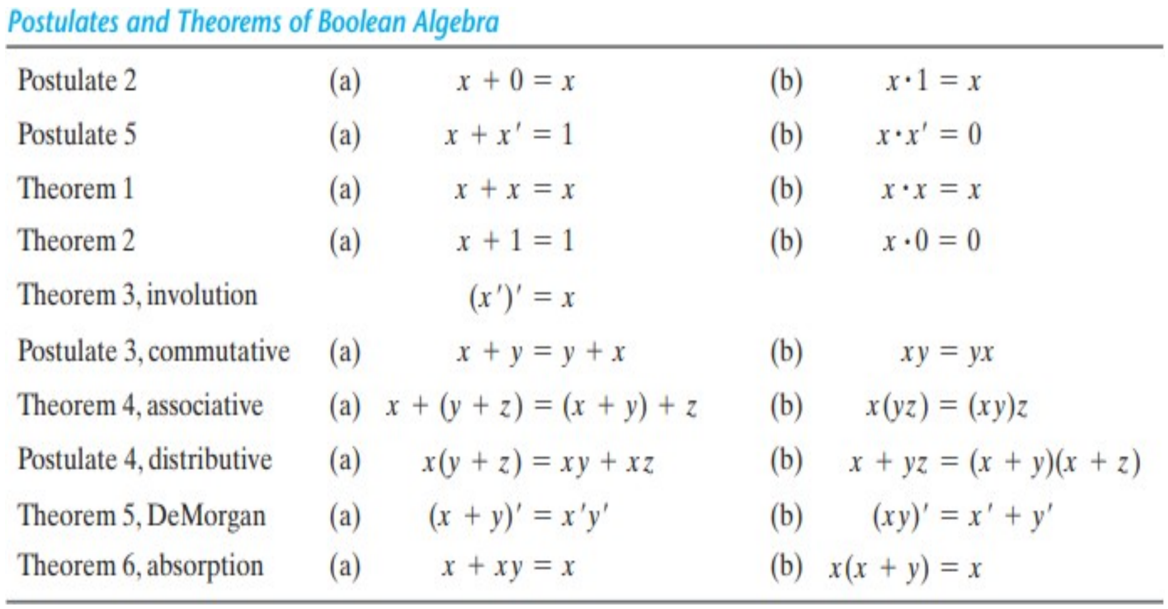
Duality

- The important property of Boolean algebra is called the *duality principle* and states that every algebraic expression deducible from the postulates of Boolean algebra remains valid if the operators and identity elements are interchanged.
- In a two-valued Boolean algebra, the identity elements and the elements of the set *B* are the same:1 and 0.
- The duality principle has many applications. If the *dual* of an algebraic expression is desired, we simply interchange OR and AND operators and replace 1's by 0's and 0's by 1's.

Basic Theorems

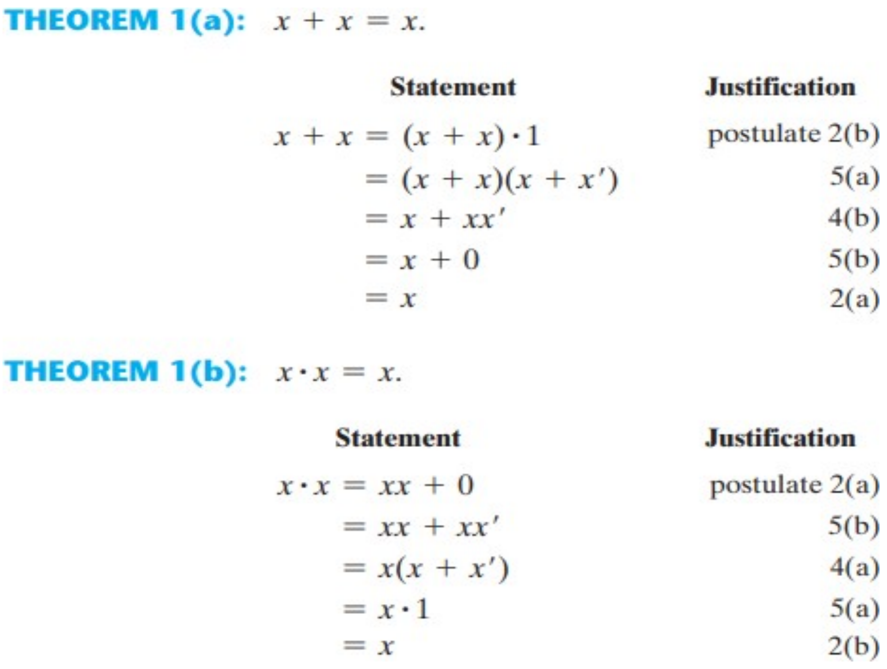
- Table 1.1 lists six theorems of Boolean algebra and four of its postulates. The notation is simplified by omitting the binary operator whenever doing so does not lead to confusion.
- The theorems and postulates listed are the most basic relationships in Boolean algebra. The theorems, like the postulates, are listed in pairs; each relation is the dual of the one paired with it.
- The postulates are basic axioms of the algebraic structure and need no proof. The theorems must be proven from the postulates. Proofs of the theorems with one variable are presented next.
    - At the right is listed the number of the postulatewhich justifies that particular step of the proof.

TABLE 1.1



Postulates and Theorems of Boolean Algebra

Postulate 2	(a)	$x + 0 = x$	(b)	$x \cdot 1 = x$
Postulate 5	(a)	$x + x' = 1$	(b)	$x \cdot x' = 0$
Theorem 1	(a)	$x + x = x$	(b)	$x \cdot x = x$
Theorem 2	(a)	$x + 1 = 1$	(b)	$x \cdot 0 = 0$
Theorem 3, involution		$(x')' = x$		
Postulate 3, commutative	(a)	$x + y = y + x$	(b)	$xy = yx$
Theorem 4, associative	(a)	$x + (y + z) = (x + y) + z$	(b)	$x(yz) = (xy)z$
Postulate 4, distributive	(a)	$x(y + z) = xy + xz$	(b)	$x + yz = (x + y)(x + z)$
Theorem 5, DeMorgan	(a)	$(x + y)' = x'y'$	(b)	$(xy)' = x' + y'$
Theorem 6, absorption	(a)	$x + xy = x$	(b)	$x(x + y) = x$



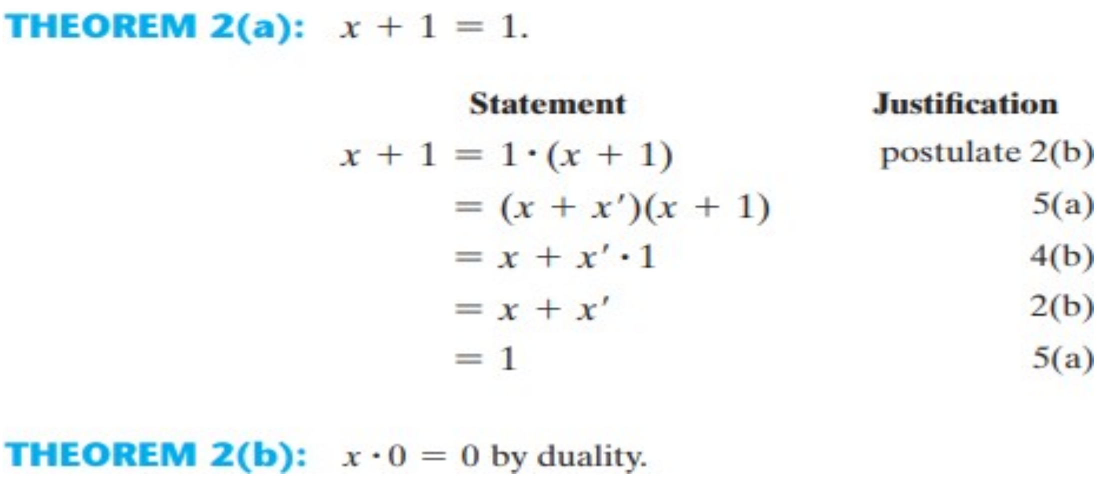
THEOREM 1(a): $x + x = x$.

Statement	Justification
$x + x = (x + x) \cdot 1$	postulate 2(b)
$= (x + x)(x + x')$	5(a)
$= x + xx'$	4(b)
$= x + 0$	5(b)
$= x$	2(a)

THEOREM 1(b): $x \cdot x = x$.

Statement	Justification
$x \cdot x = xx + 0$	postulate 2(a)
$= xx + xx'$	5(b)
$= x(x + x')$	4(a)
$= x \cdot 1$	5(a)
$= x$	2(b)

Note that theorem 1(b) is the dual of theorem 1(a) and that each step of the proof in part (b) is the dual of its counterpart in part (a). Any dual theorem can be similarlyderived from the proof of its corresponding theorem.



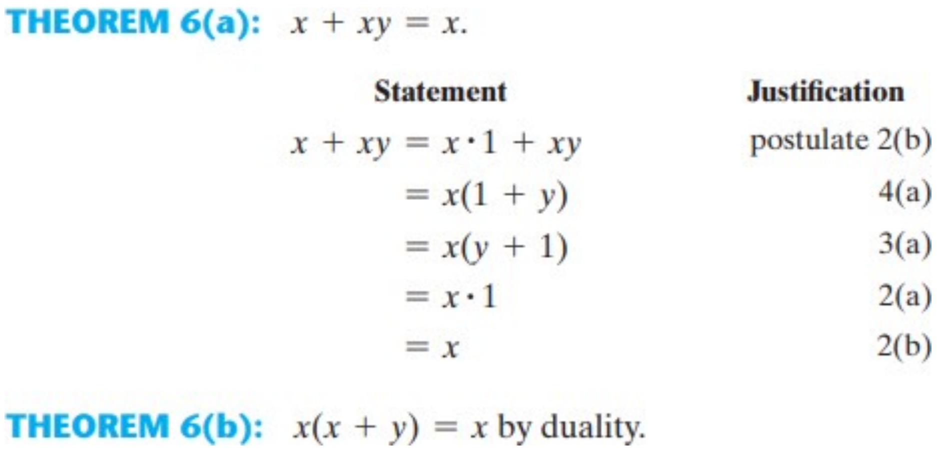
THEOREM 2(a): $x + 1 = 1$.

Statement	Justification
$x + 1 = 1 \cdot (x + 1)$	postulate 2(b)
$= (x + x')(x + 1)$	5(a)
$= x + x' \cdot 1$	4(b)
$= x + x'$	2(b)
$= 1$	5(a)

THEOREM 2(b): $x \cdot 0 = 0$ by duality.

THEOREM 3: $(x')' = x$. From postulate 5, we have $x + x' = 1$ and $x.\ x' = 0$, which together define the complement of x. The complement of x' is x and is also $(x')'$.

Therefore, since the complement is unique, we have $(x')' = x$. The theorems involving two or three variables may be proven algebraically from the postulates and thetheorems that have already been proven. Take, for example, the absorption theorem:

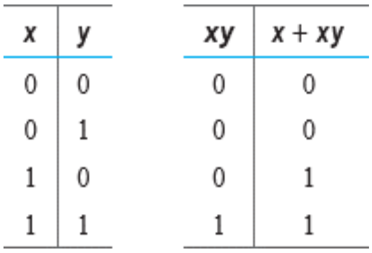


THEOREM 6(a): $x + xy = x$.

Statement	Justification
$x + xy = x \cdot 1 + xy$	postulate 2(b)
$= x(1 + y)$	4(a)
$= x(y + 1)$	3(a)
$= x \cdot 1$	2(a)
$= x$	2(b)

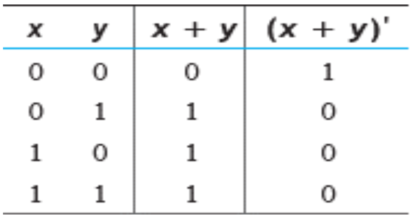
THEOREM 6(b): $x(x + y) = x$ by duality.

The theorems of Boolean algebra can be proven by means of truth tables. In truth tables, both sides of the relation are checked to see whether they yield identical results for all possible combinations of the variables involved. The following truth table verifies the first absorption theorem:

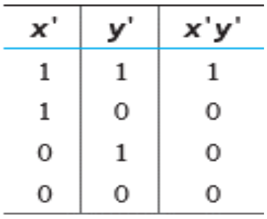


x	y	xy	x + xy
0	0	0	0
0	1	0	0
1	0	0	1
1	1	1	1

The algebraic proofs of the associative law and DeMorgan’s theorem are long and will not be shown here. However, their validity is easily shown with truth tables. For example, the truth table for the first DeMorgan's theorem, $(x + y)' = x'y'$, is as follows:



x	y	$x + y$	$(x + y)'$
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0



x'	y'	$x'y'$
1	1	1
1	0	0
0	1	0
0	0	0

Operator Precedence

- The operator precedence for evaluating Boolean expressions is **(1) parentheses,**
- **(2) NOT, (3) AND, and (4) OR.**
- In other words, expressions inside parentheses must be evaluated before all other operations.
- The next operation that holds precedence is thecomplement, and then follows the AND and, finally, the OR. As an example, consider the truth table for one of DeMorgan's theorems.
- The left side of the expression is $(x + y)'$. Therefore, the expression inside the parentheses is evaluated first and the result then complemented.
- The right side of the expression is $x'y'$, so the complement of x and the complement of y are both evaluated first and the result is then ANDed.
- Note that in ordinary arithmetic, the same precedence holds (except for the complement) when multiplication and addition are replaced by AND and OR, respectively.

1.3 BOOLEAN FUNCTIONS

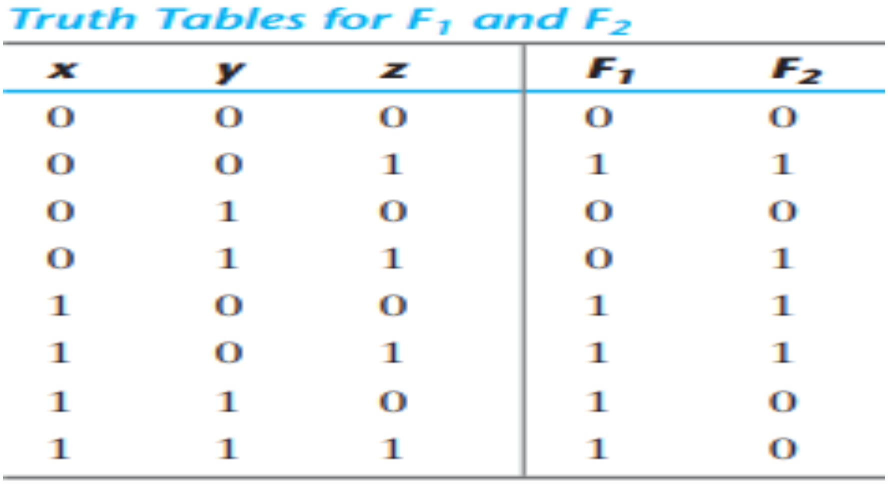
Explain the Boolean Function with Example.

- Boolean algebra is an algebra that deals with binary variables and logic operations.
- A Boolean function described by an algebraic expression consists of binary variables, the constants 0 and 1, and the logic operation symbols.
- For a given value of the binary variables, the function can be equal to either 1 or 0.
- As an example, consider the Boolean function F1 = x + y'z The function F1 is equal to 1 if x is equal to 1 or if both y' and z are equal to 1. F1 is equal to 0 otherwise.
- The complement operation dictates that when y' = 1, y = 0. Therefore, F1 = 1 if x = 1 or if y = 0 and z = 1. A Boolean function expresses the logical relationship between binary variables and is evaluated by determining the binary value of the expression for all possible values of the variables.
- A Boolean function can be represented in a truth table.

The number of rows in the truth table is 2^n , where n is the number of variables in the function.

- The binary combinations for the truth table are obtained from the binary numbers by counting from 0 through 2^n - 1.

Table 1.2



Truth Tables for F_1 and F_2

x	y	z	F_1	F_2
0	0	0	0	0
0	0	1	1	1
0	1	0	0	0
0	1	1	0	1
1	0	0	1	1
1	0	1	1	1
1	1	0	1	0
1	1	1	1	0

- Table 1.2 shows the truth table for the function F1.
- There are eight possible binary combinations for assigning bits to the three variables x, y, and z. The column labeled F1 contains either 0 or 1 for each of these combinations.
- The table shows that the function is equal to 1 when x = 1 or when yz = 01 and is equal to 0 otherwise.
- A Boolean function can be transformed from an algebraic expression into a circuit diagram composed of logic gates connected in a particular structure.
- The logic-circuit diagram (also called a schematic) for F1 is shown in Fig.1.6 . There is an inverter for input y to generate its complement. There is an AND gate for the term y'z and an OR gate that combines x with y'z.

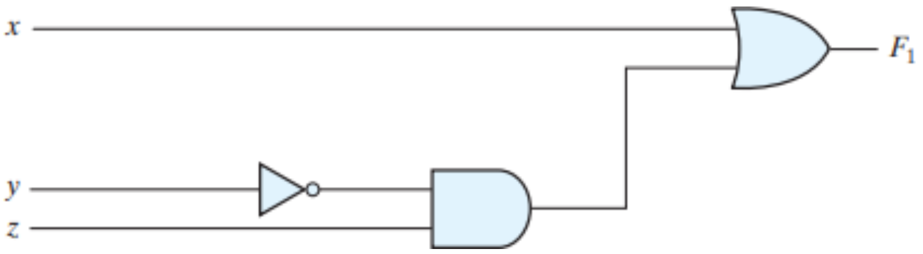


FIGURE 1.6 Gate Implementation of F_1= x+y'z

- In logic-circuit diagrams, the variables of the function are taken as the inputs of the circuit and the binary variable F_1 is taken as the output of the circuit.

- The schematic expresses the relationship between the output of the circuit and its inputs. Rather than listing each combination of inputs and outputs, it indicates how to compute the logic value of each output from the logic values of the inputs.
- The particular expression used to represent the function will dictate the interconnection of gates in the logic-circuit diagram. Conversely, the interconnection of gates will dictate the logic expression.
- Designers are motivated to reduce the complexity and number of gates because their effort can significantly reduce the cost of a circuit.
- Consider, for example, the following Boolean function:

$$F_2 = x'y'z + x'yz + xy'$$

- A schematic of an implementation of this function with logic gates is shown in Fig. 1.7 (a).
- Input variables x and y are complemented with inverters to obtain x' and y'.
- The three terms in the expression are implemented with three AND gates.
- The OR gate forms the logical OR of the three terms. The truth table for F2 is listed in Table 1.2 .
- The function is equal to 1 when xyz = 001 or 011 or when xy = 10 (irrespective of the value of z) and is equal to 0 otherwise.
- This set of conditions produces four 1's and four 0's for F2.

Now consider the possible simplification of the function by applying some of the identities of Boolean algebra:

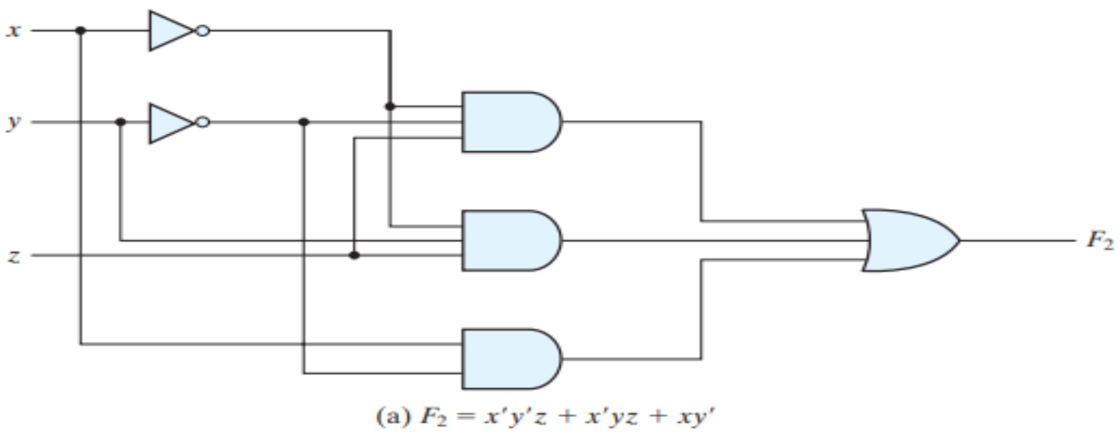
$$F2 = x'y'z + x'yz + xy' = x'z(y' + y) + xy' = x'z + xy'$$

The function is reduced to only two terms and can be implemented with gates as shown in Fig. 1.7 (b).

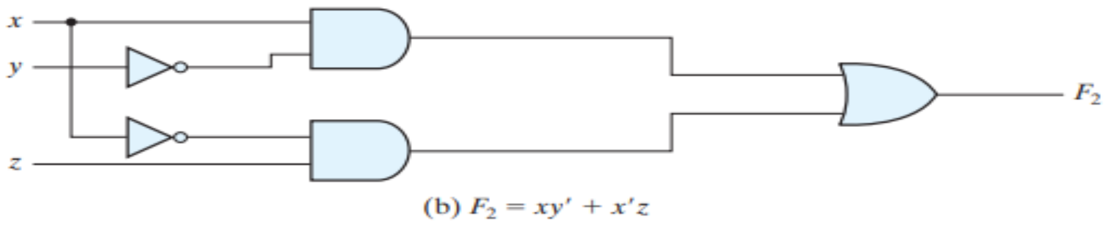
It is obvious that the circuit in (b) is simpler than the one in (a), yet both implement the same function.

By means of a truth table, it is possible to verify that the two expressions are equivalent. The simplified expression is equal to 1 when xz = 01 or when xy = 10.

This produces the same four 1's in the truth table. Since both expressions produce the same truth table, they are equivalent.



(a) $F_2 = x'y'z + x'yz + xy'$



(b) $F_2 = xy' + x'z$

FIGURE 1.7 Implementation of Boolean function F2 with gates

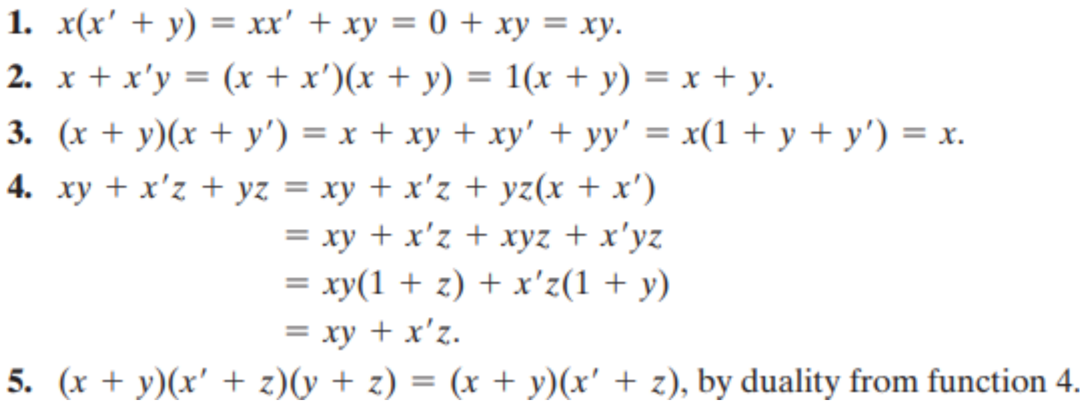
- Therefore, the two circuits have the same outputs for all possible binary combinations of inputs of the three variables.
- Each circuit implements the same identical function, but the one with fewer gates and fewer inputs to gates is preferable because it requires fewer wires and components .
- In general, there are many equivalent representations of a logic function. Finding the most economic representation of the logic is an important design task.

Algebraic Manipulation

- When a Boolean expression is implemented with logic gates, each term requires a gate and each variable within the term designates an input to the gate.
- We define a literal to be a single variable within a term, in complemented or uncomplemented form. The function of Fig. 1.9 (a) has three terms and eight literals, and the one in Fig. 1.9 (b) has two terms and four literals.
- By reducing the number of terms, the number of literals, or both in a Boolean expression, it is often possible to obtain a simpler circuit.
- The manipulation of Boolean algebra consists mostly of reducing an expression for the purpose of obtaining a simpler circuit.

- For complex Boolean functions and many different outputs, designers of digital circuits use computer minimization programs that are capable of producing optimal circuits with millions of logic gates.
- The concepts introduced in this chapter provide the framework for those tools. The only manual method available is a cut-and-try procedure employing the basic relations and other manipulation techniques that become familiar with use, but remain, nevertheless, subject to human error.
- The examples that follow illustrate the algebraic manipulation of Boolean algebra to acquaint the reader with this important design task.

Example 1.1: Simplify the following Boolean functions to a minimum number of literal



1. $x(x' + y) = xx' + xy = 0 + xy = xy.$
2. $x + x'y = (x + x')(x + y) = 1(x + y) = x + y.$
3. $(x + y)(x + y') = x + xy + xy' + yy' = x(1 + y + y') = x.$
4. $$\begin{aligned} xy + x'z + yz &= xy + x'z + yz(x + x') \\ &= xy + x'z + xyz + x'yz \\ &= xy(1 + z) + x'z(1 + y) \\ &= xy + x'z. \end{aligned}$$
5. $(x + y)(x' + z)(y + z) = (x + y)(x' + z)$, by duality from function 4.

- Functions 1 and 2 are the dual of each other and use dual expressions in corresponding steps.
- An easier way to simplify function 3 is by means of postulate 4(b) from Table 1.1 :

$$(x + y)(x + y') = x + yy' = x.$$

- The fourth function illustrates the fact that an increase in the number of literals sometimes leads to a simpler final expression.
- Function 5 is not minimized directly, but can be derived from the dual of the steps used to derive function 4. Functions 4 and 5 are together known as the consensus theorem.

Complement of a Function

The complement of a function F is F’ and is obtained from an interchange of 0’s for 1’s and 1’s for 0’s in the value of F. The complement of a function may be derived algebraically through DeMorgan’s theorems, listed in Table 1.1 for two variables.

DeMorgan’s theorems can be extended to three or more variables. The three-variable form of the first DeMorgan’s theorem is derived as follows, from postulates and theorems listed in Table 1.1 :

$$\begin{aligned}
(A + B + C)' &= (A + x)' && \text{let } B + C = x \\
&= A'x' && \text{by theorem 5(a) (DeMorgan)} \\
&= A'(B + C)' && \text{substitute } B + C = x \\
&= A'(B'C') && \text{by theorem 5(a) (DeMorgan)} \\
&= A'B'C' && \text{by theorem 4(b) (associative)}
\end{aligned}$$

DeMorgan's theorems for any number of variables resemble the two-variable case in form and can be derived by successive substitutions similar to the method used in the preceding derivation. These theorems can be generalized as follows:

$$(A + B + C + D + \cdots + F)' = A'B'C'D' \ldots F'$$

$$(ABCD \ldots F)' = A' + B' + C' + D' + \cdots + F'$$

The generalized form of DeMorgan's theorems states that the complement of a function is obtained by interchanging AND and OR operators and complementing each literal.

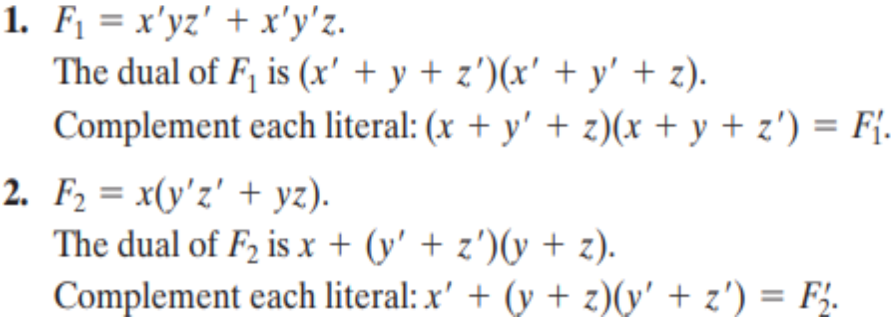
EXAMPLE1.2: Find the complement of the functions F1 = x'yz'+ x'y'z and F2 = x(y'z' + yz). By applying DeMorgan's theorems as many times as necessary, the complements are obtained as follows:

$$F_1' = (x'yz' + x'y'z)' = (x'yz')'(x'y'z)' = (x + y' + z)(x + y + z')$$

$$\begin{aligned}
F_2' = [x(y'z' + yz)]' &= x' + (y'z' + yz)' = x' + (y'z')'(yz)' \\
&= x' + (y + z)(y' + z') \\
&= x' + yz' + y'z
\end{aligned}$$

A simpler procedure for deriving the complement of a function is to take the dual of the function and complement each literal. This method follows from the generalized forms of DeMorgan's theorems. Remember that the dual of a function is obtained from the interchange of AND and OR operators and 1's and 0's.

EXAMPLE 1.3: Find the complement of the functions F1 and F2 of Example 1.2 by taking their duals and complementing each literal.



1. $F_1 = x'yz' + x'y'z$.
   The dual of F_1 is $(x' + y + z')(x' + y' + z)$.
   Complement each literal: $(x + y' + z)(x + y + z') = F_1'$.
2. $F_2 = x(y'z' + yz)$.
   The dual of F_2 is $x + (y' + z')(y + z)$.
   Complement each literal: $x' + (y + z)(y' + z') = F_2'$.

Boolean algebra has two binary operators, which we have called AND and OR, and a unary operator, NOT (complement). From the definitions, we have deduced a number of properties of these operators and now have defined other binary operators in terms of them. There is nothing unique about this procedure. We could have just as well started with the operator NOR (T), for example, and later defined AND, OR, and NOT in terms of it.

1.4 Digital logic gates

Explain All Logic Gates with symbol and Truth Table

Since Boolean functions are expressed in terms of AND, OR, and NOT operations, it is easier to implement a Boolean function with these types of gates. Factors to be weighed in considering the construction of other types of logic gates are

(1) the feasibility and economy of producing the gate with physical components,
(2) the possibility of extending the gate to more than two inputs,
(3) the basic properties of the binary operator, such as commutativity and associativity, and
(4) the ability of the gate to implement Boolean functions alone or in conjunction with other gates.

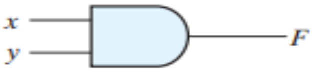
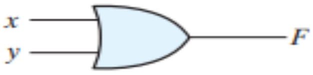
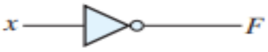
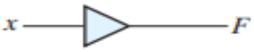
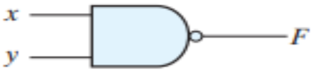
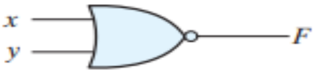
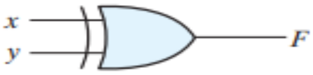
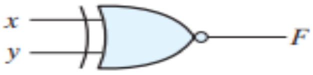
Name	Graphic symbol	Algebraic function	Truth table
AND		$F = x \cdot y$	x y | F 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1
OR		$F = x + y$	x y | F 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 1
Inverter		$F = x'$	x | F 0 | 1 1 | 0
Buffer		$F = x$	x | F 0 | 0 1 | 1
NAND		$F = (xy)'$	x y | F 0 0 | 1 0 1 | 1 1 0 | 1 1 1 | 0
NOR		$F = (x + y)'$	x y | F 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 0
Exclusive-OR (XOR)		$F = xy' + x'y$ $= x \oplus y$	x y | F 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 0
Exclusive-NOR or equivalence		$F = xy + x'y'$ $= (x \oplus y)'$	x y | F 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 1

FIGURE 1.8 Digital logic gates

- Each gate has one or two binary input variables, designated by *x* and *y*, and one binary output variable, designated by *F*. The AND, OR and The inverter circuit inverts the logic sense of a binary variable, producing the NOT, or complement, function.
- The small circle in the output of the graphic symbol of an inverter (referred to as a *bubble*) designates the logic complement. The triangle symbol by itself designates a buffer circuit. A bufferproduces the *transfer* function, but does not produce a logic operation, since the binary value of theoutput is equal to the binary value of the input.

- The NAND function is the complement of the AND function, as indicated by a graphic symbol that consists of an AND graphic symbol followed by a small circle.
- The NOR function is the complement of the OR function and uses an OR graphic symbol followed by a small circle. NAND and NOR gates are used extensively as standard logic gates and are in fact far more popular than the AND and OR gates. This is because NAND and NOR gates are easily constructed with transistor circuits and because digital circuits can be easily implemented with them.
- The exclusive-OR gate has a graphic symbol similar to that of the OR gate, except for the additional curved line on the input side.
- The exclusive-NOR, gate is the complement of the exclusive-OR, as indicated by the small circle on the output side of the graphic symbol.

Extension to Multiple Inputs

- The gates shown in Fig. 1.8 —except for the inverter and buffer—can be extended to have more than two inputs.
- A gate can be extended to have multiple inputs if the binary operation it represents is commutative and associative.

The AND and OR operations, defined in Boolean algebra, possess these two properties. For the OR function, we have

$$x + y = y + x \quad \text{(commutative)}$$

$$(x + y) + z = x + (y + z) = x + y + z \quad \text{(associative)}$$

which indicates that the gate inputs can be interchanged and that the OR function can be extended to three or more variables.

The NAND and NOR functions are commutative, and their gates can be extended to have more than two inputs, provided that the definition of the operation is modified slightly.

The difficulty is that the NAND and NOR operators are not associative (i.e., $(x \downarrow y) \downarrow z \neq x \downarrow (y \downarrow z)$), as shown in Fig. 1.9 and the following equations:

$$(x \downarrow y) \downarrow z = [(x + y)' + z]' = (x + y)z' = xz' + yz'$$
$$x \downarrow (y \downarrow z) = [x + (y + z)']' = x'(y + z) = x'y + x'z$$

To overcome this difficulty, we define the multiple NOR (or NAND) gate as a complemented OR (or AND) gate. Thus, by definition, we have

$$x \downarrow y \downarrow z = (x + y + z)'$$
$$x \uparrow y \uparrow z = (xyz)'$$

The graphic symbols for the three-input gates are shown in Fig. 1.10 . In writing cascaded NOR and NAND operations, one must use the correct parentheses to signify the proper sequence of the gates. To demonstrate this principle, consider the circuit of Fig. 1.10 (c). The Boolean function for the circuit must be written as

$$F = [(ABC)'(DE)']' = ABC + DE$$

The second expression is obtained from one of DeMorgan's theorems. It also shows that an expression in sum-of-products form can be implemented with NAND gates.

The exclusive-OR and equivalence gates are both commutative and associative and can be extended to more than two inputs. However, multiple-input exclusive-OR gates are uncommon from the hardware standpoint.

In fact, even a two-input function is usually constructed with other types of gates. Moreover, the definition of the function must be modified when extended to more than two variables. Exclusive-OR is an odd function (i.e., it is equal to 1 if the input variables have an odd number of 1's). The construction

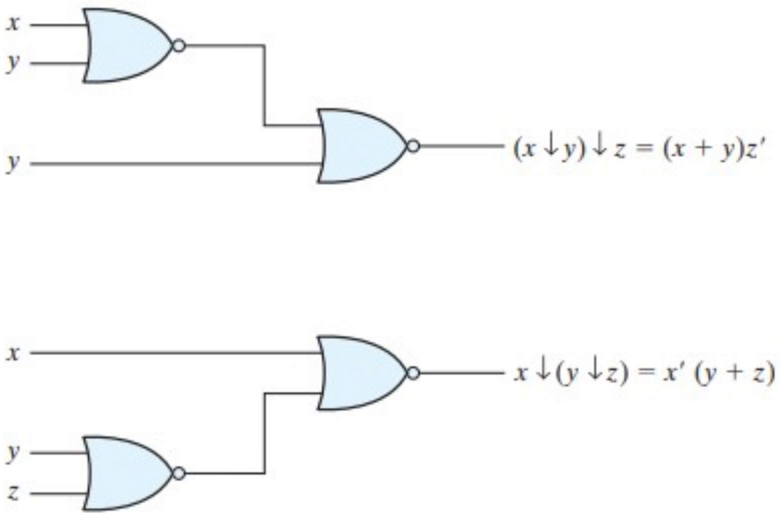
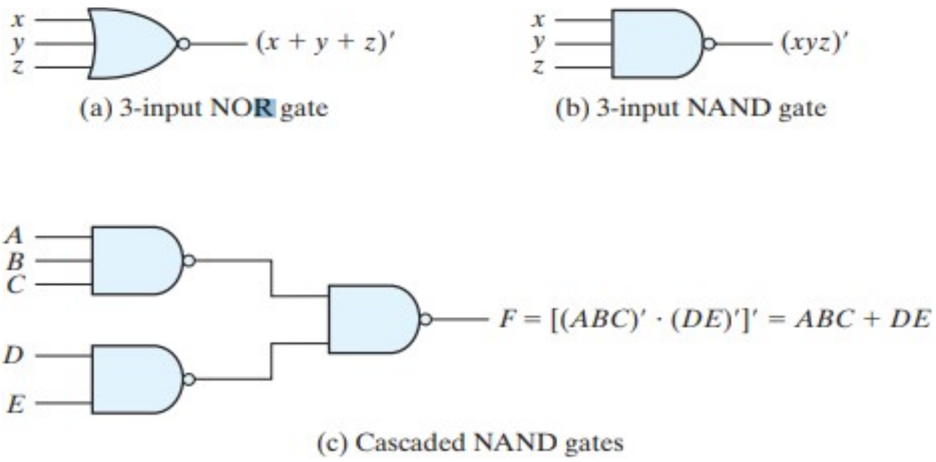

x
y
y
$(x \downarrow y) \downarrow z = (x + y)z'$
x
y
z
$x \downarrow (y \downarrow z) = x' (y + z)$

FIGURE 1.9 Demonstrating the nonassociativity of the NOR operator:

$$(x \downarrow y) \downarrow z \neq x \downarrow (y \downarrow z))$$



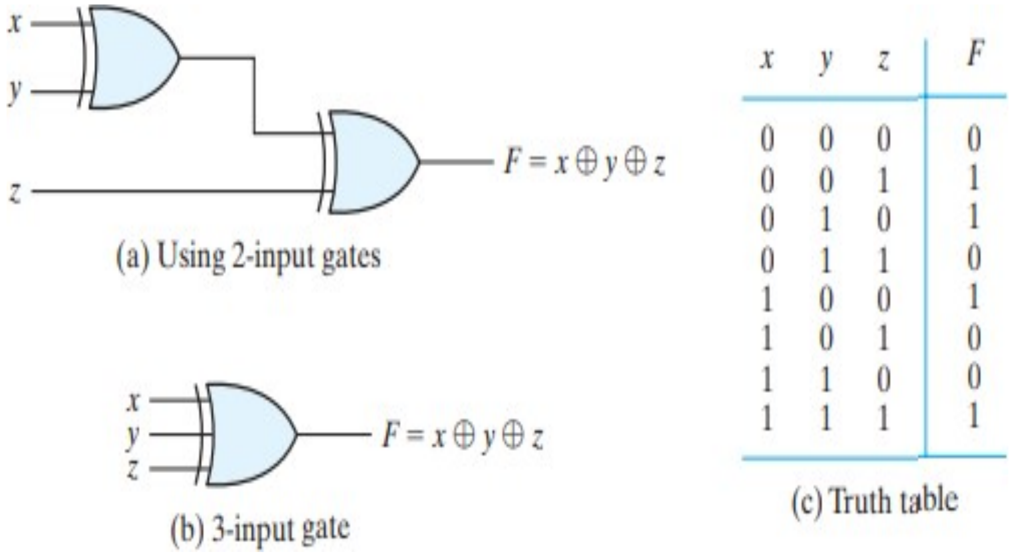
(a) 3-input NOR gate

(b) 3-input NAND gate

(c) Cascaded NAND gates

FIGURE 1.10 Multiple-input and cascaded NOR and NAND gates

of a three-input exclusive-OR function is shown in Fig. 1.11 . This function is normally implemented by cascading two-input gates, as shown in (a). Graphically, it can be represented with a single three-input gate, as shown in (b). The truth table in (c) clearly indicates that the output F is equal to 1 if only one input is equal to 1 or if all three inputs are equal to 1 (i.e., when the total number of 1's in the input variables is odd).



(a) Using 2-input gates

(b) 3-input gate

x	y	z	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

(c) Truth table

FIGURE 1.11 Three-input exclusive-OR gate

Positive and Negative Logic

Demonstrate the positive and negative logic using Basic Gates

- The binary signal at the inputs and outputs of any gate has one of two values, except during transition. One signal value represents logic 1 and the other logic 0. Since two signal values are assigned to two logic values, there exist two different assignments of signal level to logic value, as shown in Fig. 1.12 .

- The higher signal level is designated by H and the lower signal level by L. Choosing the high-level H to represent logic 1 defines a positive logic system. Choosing the low-level L to represent logic 1 defines a negative logic system.
- The terms positive and negative are somewhat misleading, since both signals may be positive or both may be negative. It is not the actual values of the signals that determine the type of logic, but rather the assignment of logic values to the relative amplitudes of the two signal levels.

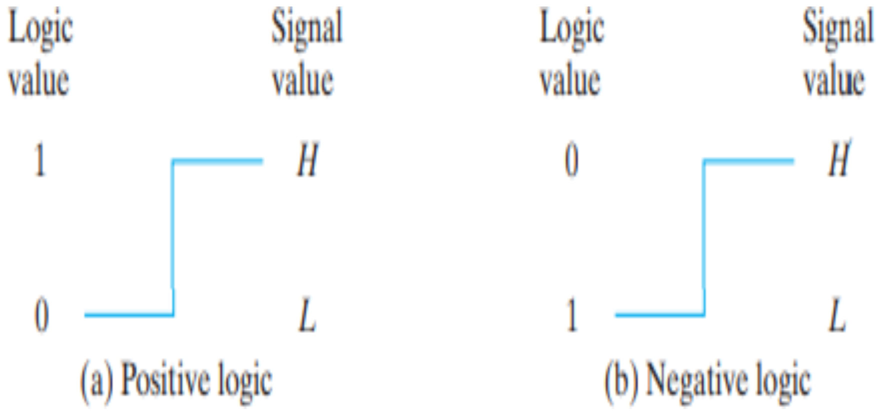
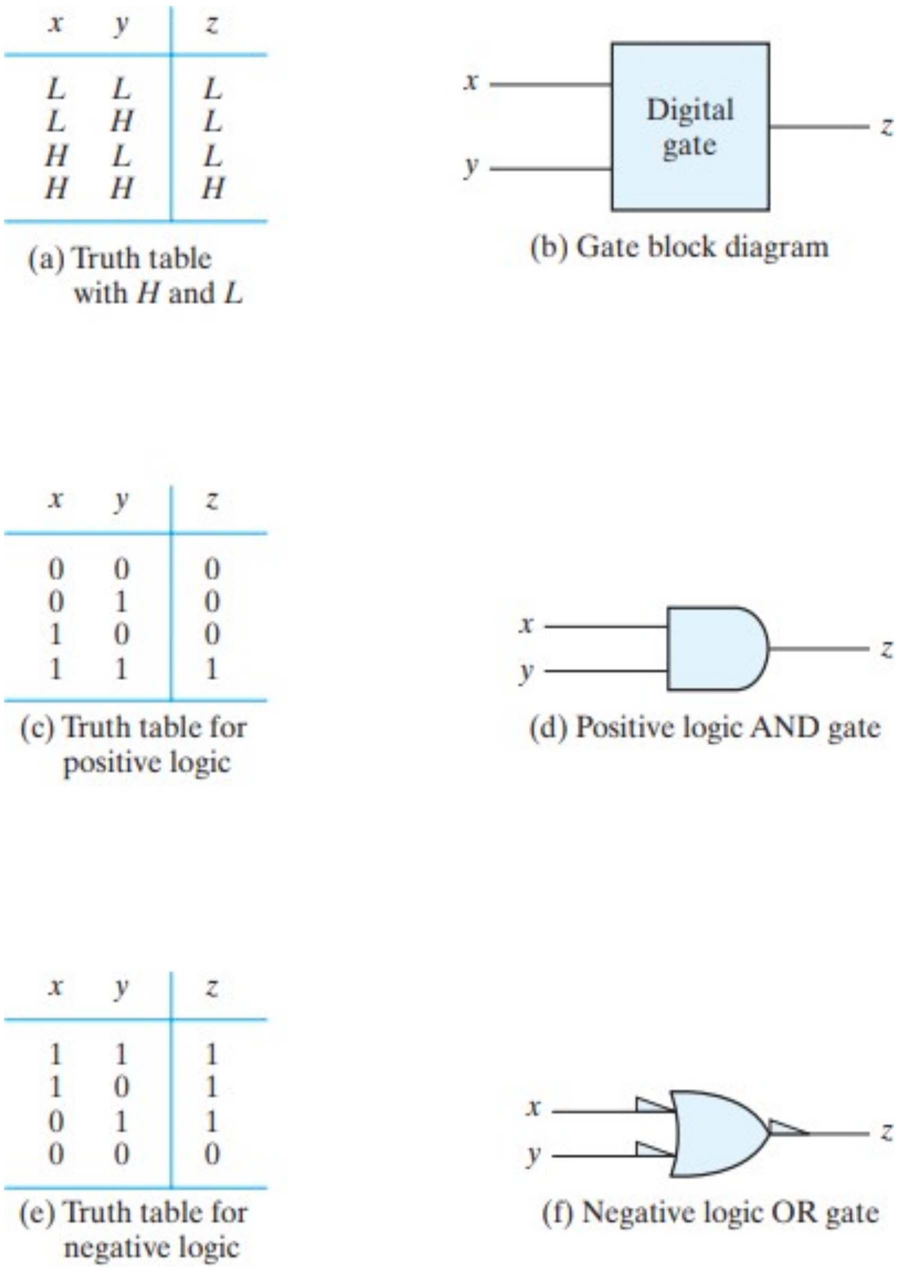


FIGURE 1.12 Signal assignment and logic polarity

Hardware digital gates are defined in terms of signal values such as H and L. It is up to the user to decide on a positive or negative logic polarity.

- Consider, for example, the electronic gate shown in Fig. 1.13 (b). The truth table for this gate is listed in Fig. 1.13 (a). It specifies the physical behavior of the gate when H is 3 V and L is 0 V. The truth table of Fig. 1.13 (c) assumes a positive logic assignment, with H = 1 and L = 0. This truth table is the same as the one for the AND operation. The graphic symbol for a positive logic AND gate is shown in Fig. 1.13 (d).
- Now consider the negative logic assignment for the same physical gate with L = 1 and H = 0. The result is the truth table of Fig. 1.13 (e). This table represents the OR operation, even though the entries are reversed. The graphic symbol for the negative- logic OR gate is shown in Fig. 1.13 (f).
- The small triangles in the inputs and output designate a polarity indicator, the presence of which along a terminal signifies that negative logic is assumed for the signal.
- Thus, the same physical gate can operate either as a positive-logic AND gate or as a negative-logic OR gate.

- The conversion from positive logic to negative logic and vice versa is essentially an operation that changes 1's to 0's and 0's to 1's in both the inputs and the output of a gate. Since this operation produces the dual of a function, the change of all terminals from one polarity to the other results in taking the dual of the function.



x	y	z
L	L	L
L	H	L
H	L	L
H	H	H

(a) Truth table with H and L

(b) Gate block diagram

x	y	z
0	0	0
0	1	0
1	0	0
1	1	1

(c) Truth table for positive logic

(d) Positive logic AND gate

x	y	z
1	1	1
1	0	1
0	1	1
0	0	0

(e) Truth table for negative logic

(f) Negative logic OR gate

FIGURE 1.13 Demonstration of positive and negative logic

1.5 INTEGRATED CIRCUITS

Define Integrated Circuit. Explain The Levels of Integration.

An integrated circuit (IC) is fabricated on a die of a silicon semiconductor crystal, called a *chip,* containing the electronic components for constructing digital gates.

The various gates are interconnected inside the chip to form the required circuit. The chip is mounted in a ceramic or plastic container, and connections are welded to external pins to form the integrated circuit. The number of pins may range from 14 on a small IC package to several thousand on a larger package. Each IC has a numeric designation printed on the surface of the package for identification.

Levels of Integration

- Digital ICs are often categorized according to the complexity of their circuits, as measured by the number of logic gates in a single package. The differentiation between those chips which have a few internal gates and those having hundreds of thousands of gates is made by customary reference to a package as being either a small-, medium-, large-, or very large-scale integration device.
- ***Small-scale integration*** **(SSI)** devices contain several independent gates in a single package. The inputs and outputs of the gates are connected directly to the pins in the package. The number of gates is usually fewer than 10 and is limited by the number of pins available in the IC.
- ***Medium-scale integration*** **(MSI)** devices have a complexity of approximately 10 to 1,000 gates in a single package. They usually perform specific elementary digital operations
- ***Large-scale integration*** **(LSI)** devices contain thousands of gates in a single package. They include digital systems such as processors, memory chips, and programmable logic devices.
- ***Very large-scale integration*** **(VLSI)** devices now contain millions of gates within a single package. Examples are large memory arrays and complex microcomputer chips. Because of their small size and low cost, VLSI devices have revolutionized the computer system design technology, giving the designer the capability to create structures that were previously uneconomical to build.

Digital Logic Families

- Digital integrated circuits are classified not only by their complexity or logical operation, but also by the specific circuit technology to which they belong. The circuit technology is referred to as a *digital logic family*.
- Each logic family has its own basic electronic circuit upon which more complex digital circuits and components are developed. The basic circuit in each technology is a NAND, NOR, or inverter gate.

1.6 Gate-Level Minimization - INTRODUCT ION

- *Gate-level minimization* is the design task of finding an optimal gate-level implementation of the Boolean functions describing a digital circuit. This task is well understood, but is difficult to execute by manual methods when the logic has more than a few inputs.
- Fortunately, computer-based logic synthesis tools can minimize a large set of Boolean equations efficiently and quickly. Nevertheless, it is important that a designer understand the underlying mathematical description and solution of the problem.

1.7 The Map Method

- The complexity of the digital logic gates that implement a Boolean function is directly related to the complexity of the algebraic expression from which the function is implemented. Although the truth table representation of a function is unique, when it is expressed algebraically it can appear in many different, but equivalent, forms.
- Boolean expressions may be simplified by algebraic means. However, this procedure of minimization is awkward because it lacks specific rules to predict each succeeding step in the manipulative process. The map method presented here provides a simple, straightforward procedure for minimizing Boolean functions. This method may be regarded as a pictorial form of a truth table. The map method is also known as the *Karnaugh map* or *K-map* .
- A K-map is a diagram made up of squares, with each square representing one minterm of the function that is to be minimized. Since any Boolean function can be expressed as a sum of minterms, it follows that a Boolean function is recognized graphically in the map from the area enclosed by those squares whose minterms are included in the function.
- In fact, the map presents a visual diagram of all possible ways a function may be expressed in standard form. By recognizing various patterns, the user can derive alternative algebraic expressions for the same function, from which the simplest can be selected.
- The simplified expressions produced by the map are always in one of the two standard forms: sum of products or product of sums. It will be assumed that the simplest algebraic expression is an algebraic expression with a minimum number of terms and with the smallest possible number of literals in each term. This expression produces a circuit diagram with a minimum number of gates and the minimum number of inputs to each gate.

Two-Variable K-Map

The two-variable map is shown in Fig. 1.14 a)

(a). There are four minterms for two variables; hence, the map consists of four squares, one for each minterm.

(b) to show the relationship between the squares and the two variables *x* and *y* . The 0 and 1 marked in each row and column designate the values of variables. Variable *x* appears primed in row 0 and unprimed in row 1. Similarly, *y* appears primed in column 0 and unprimed in column 1.

- If we mark the squares whose minterms belong to a given function, the two-variable map becomes another useful way to represent any one of the 16 Boolean functions of two variables. As an example, the function *xy* is shown in Fig. 1.15 (a).
- Since *xy* is equal to *m*3, a 1 is placed inside the square that belongs to *m*3. Similarly, the function *x* + *y* is represented in the map of Fig. 1.15 (b) by three squares marked with 1's. These squares are found from the minterms of the function:

$$m_1 + m_2 + m_3 = x'y + xy' + xy = x + y$$

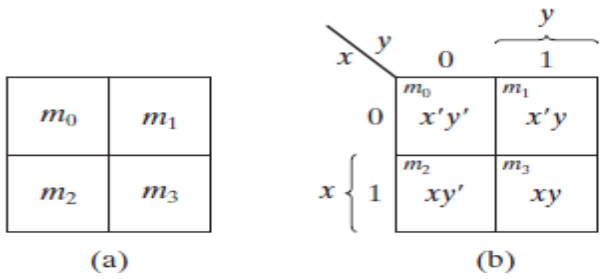


FIGURE 1.14 Two-variable K-map

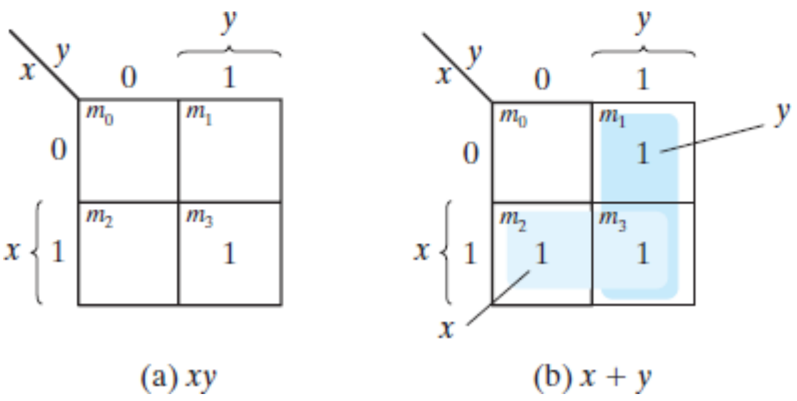
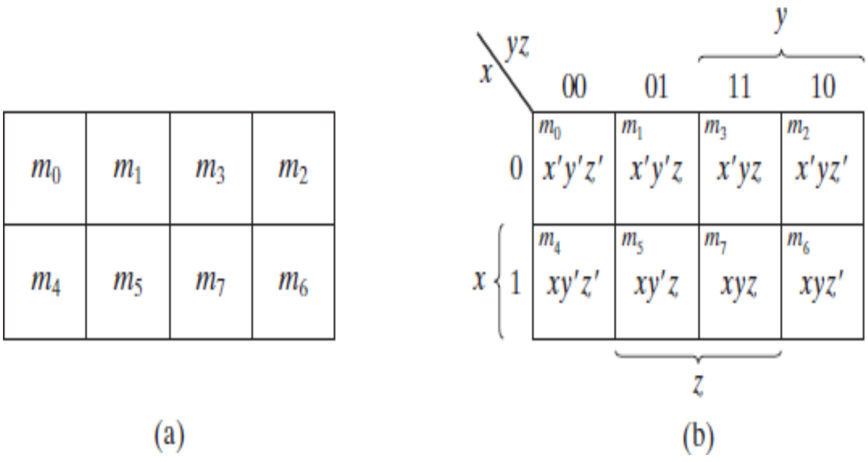


FIGURE 1.15 Representation of functions in the map

The three squares could also have been determined from the intersection of variable *x* in the second row and variable *y* in the second column, which encloses the area belonging to *x* or *y* . In each example, the minterms at which the function is asserted are marked with a 1.

Three-Variable K-Map

- A three-variable K-map is shown in Fig. 1.16 . There are eight minterms for three binary variables; therefore, the map consists of eight squares.
- Note that the minterms are arranged, not in a binary sequence, but in a sequence similar to the Gray code. The characteristic of this sequence is that **only one bit changes in value from one adjacent column to the next.**
- The map drawn in part (b) is marked with numbers in each row and each column to show the relationship between the squares and the three variables.
- For example, the square assigned to *m*5 corresponds to row 1 and column 01. When these two numbers are concatenated, they give the binary number 101, whose decimal equivalent is 5.
- Each cell of the map corresponds to a unique minterm, so another way of looking at square *m*5 = *xy'z* is to consider it to be in the row marked *x* and the column belonging to *y'z* (column 01).
- Note that there are four squares in which each variable is equal to 1 and four in which each is equal to 0. The variable appears unprimed in the former four squares and primed in the latter. For convenience, we write the variable with its letter symbol under the four squares in which it is unprimed.



m_0	m_1	m_3	m_2
m_4	m_5	m_7	m_6

(a)

$x \backslash yz$	00	01	11	10
0	m_0 $x'y'z'$	m_1 $x'y'z$	m_3 $x'yz$	m_2 $x'yz'$
1	m_4 $xy'z'$	m_5 $xy'z$	m_7 xyz	m_6 xyz'

(b)

FIGURE 1.16 Three-variable K-map

- To understand the usefulness of the map in simplifying Boolean functions, we must recognize the basic property possessed by adjacent squares: **Any two adjacent squares in the map differ by only one variable,** which is primed in one square and unprimed in the other.
- For example, *m*5 and *m*7 lie in two adjacent squares. Variable *y* is primed in *m*5 and unprimed in *m*7, whereas the other two variables are the same in both squares. From the postulates of Boolean algebra, it follows that the sum of two minterms in
- adjacent squares can be simplified to a single product term consisting of only two literals.
- To clarify this concept, consider the sum of two adjacent squares such as *m*5 and *m*7:

$$m_5 + m_7 = xy'z + xyz = xz(y' + y) = xz$$

Here, the two squares differ by the variable *y* , which can be removed when the sum of the two minterms is formed. Thus, any two minterms in adjacent squares (vertically or horizontally, but not diagonally, adjacent) that are ORed together will cause a removal of the dissimilar variable. The next four examples explain the procedure for minimizing a Boolean function with a K-map.

EXAMPLE:1.4 Simplify the Boolean function

$$F(x, y, z) = \Sigma(2, 3, 4, 5)$$

- First, a 1 is marked in each minterm square that represents the function. This is shown in Fig. 1.17 , in which the squares for minterms 010, 011, 100, and 101 are marked with 1's.
- The next step is to find possible adjacent squares. These are indicated in the map by two shaded rectangles, each enclosing two 1's.
- The upper right rectangle represents the area enclosed by *x*'*y*. This area is determined by observing that the two-square area is in row 0, corresponding to *x*', and the last two columns, corresponding to *y* .
- Similarly, the lower left rectangle represents the product term *xy*'. (The second row represents *x* and the two left column s represent *y*'.) The sum of four minterms can be replaced by a sum of only two product terms.

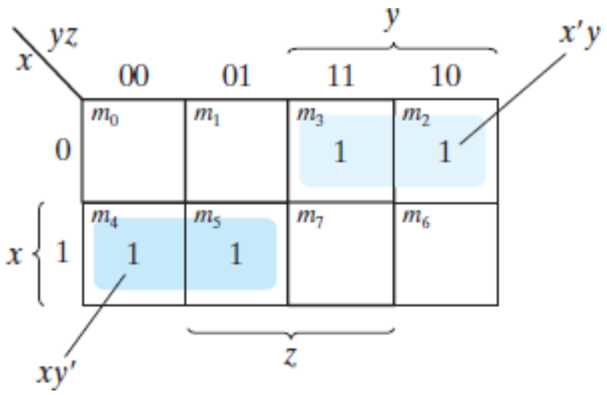


FIGURE 1.17 The logical sum of these two product terms gives the simplified Expression

$$F = x'y + xy'$$

In certain cases, two squares in the map are considered to be adjacent even though they do not touch each other. In Fig. 1.16 (b), *m*0 is adjacent to *m*2 and *m*4 is adjacent to *m*6 because their minterms differ by one variable. This difference can be readily verified algebraically:

$$m_0 + m_2 = x'y'z' + x'yz' = x'z'(y' + y) = x'z'$$
$$m_4 + m_6 = xy'z' + xyz' = xz' + (y' + y) = xz'$$

Consequently, we must modify the definition of adjacent squares to include this and other similar cases. We do so by considering the map as being drawn on a surface in which the *right and left edges* touch each other to form adjacent squares.

EXAMPL:1.5 Simplify the Boolean function

$$F(x, y, z) = \Sigma(3, 4, 6, 7)$$

The map for this function is shown in Fig. 1.18 . There are four squares marked with 1's, one for each minterm of the function. Two adjacent squares are combined in the third column to give a two-literal term *yz* . The remaining two squares with 1's are also adjacent by the new definition. These two squares, when combined, give the two-literal term *xz*'. The simplified function then becomes

$$F = yz + xz'$$

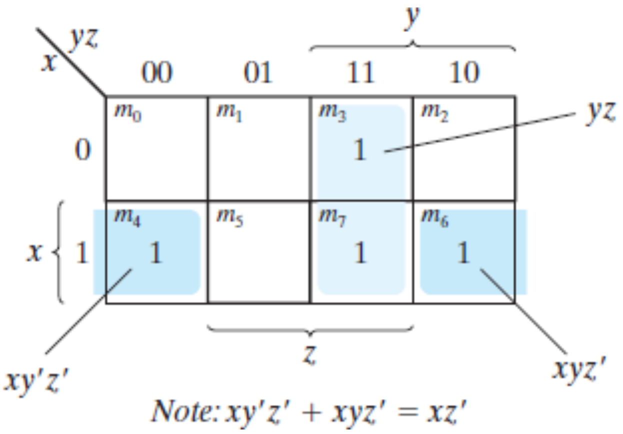


FIGURE 1.18 Map for Example $F(x, y, z) = \Sigma(3, 4, 6, 7) = yz + xz'$

Consider now any combination of four adjacent squares in the three-variable map. Any such combination represents the logical sum of four minterms and results in an expression with only one literal.

As an example, the logical sum of the four adjacent minterms 0, 2, 4, and 6 reduces to the single literal term *z*':

$$\begin{aligned} m_0 + m_2 + m_4 + m_6 &= x'y'z' + x'yz' + xy'z' + xyz' \\ &= x'z'(y' + y) + xz'(y' + y) \\ &= x'z' + xz' = z'(x' + x) = z' \end{aligned}$$

The number of adjacent squares that may be combined must always represent a number that is a power of two, such as 1, 2, 4, and 8. As more adjacent squares are combined, we obtain a product term with fewer literals.

- One square represents one minterm, giving a term with three literals.
- Two adjacent squares represent a term with two literals.
- Four adjacent squares represent a term with one literal.
- Eight adjacent squares encompass the entire map and produce a function that is always equal to 1.

Example:1.6 Simplify the Boolean function

$$F(x, y, z) = \Sigma(0, 2, 4, 5, 6)$$

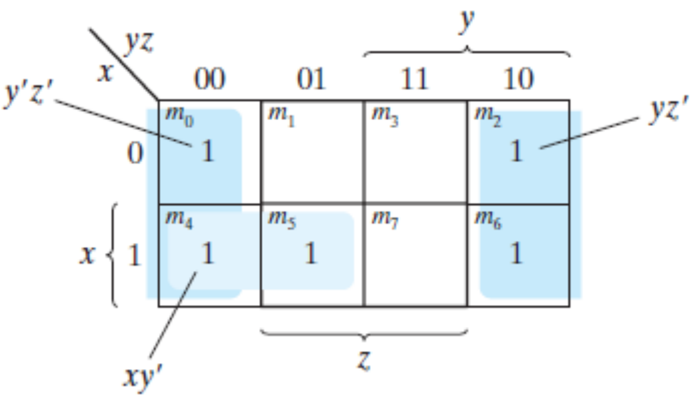
- The map for *F* is shown in Fig. 1.19 . First, we combine the four adjacent squares in the first and last columns to give the single literal term *z*'.

- The remaining single square, representing minterm 5, is combined with an adjacent square that has already been used once.

- This is not only permissible, but rather desirable, because the two adjacent squares give the two-literal term *xy*' and the single square represents the three-literal minterm *xy*'*z*.

The simplified function is

$$F = z' + xy'$$



Note: $y'z' + yz' = z'$

FIGURE 1.19 Map for Example 3 $F(x, y, z) = \Sigma(3, 4, 6, 7) = yz + xz'$

- If a function is not expressed in sum-of-minterms form, it is possible to use the map to obtain the minterms of the function and then simplify the function to an expression with a minimum number of terms.
- It is necessary, however, to make sure that the algebraic expression is in sum-of-products form. Each product term can be plotted in the map in one, two,more squares.
- The minterms of the function are then read directly from the map.

EXAMPLE :1.7 For the Boolean function

$$F = A'C + A'B + AB'C + BC$$

a) Express this function as a sum of minterms.

b) Find the minimal sum-of-products expression.

- Note that *F* is a sum of products. Three product terms in the expression have two literals and are represented in a three-variable map by two squares each. The two squares corresponding to the first term, *A'C*, are found in Fig. 1.20 from the coincidence of *A*! (first row) and *C* (two middle columns) to give squares 001 and 011.

  Note that, in marking 1's in the squares, it is possible to find a 1 already placed there from a preceding term. This happens with the second term, *A'B*, which has 1's in squares 011 and 010.
- Square 011 is common with the first term, *A'C*, though, so only one 1 is marked in it. Continuing in this fashion, we determine that the term *AB'C* belongs in square 101, corresponding to minterm 5, and the term *BC* has two 1's in squares 011 and 111.
- The function has a total of five minterms, as indicated by the five 1's in the map of Fig. 1.20 . The minterms are read directly from the map to be 1, 2, 3, 5, and 7. The function can be expressed in sum-of-minterms form as

$$F(A, B, C) = \Sigma(1, 2, 3, 5, 7)$$

The sum-of-products expression, as originally given, has too many terms. It can be simplified, as shown in the map, to an expression with only two terms:

$$F = C + A'B$$

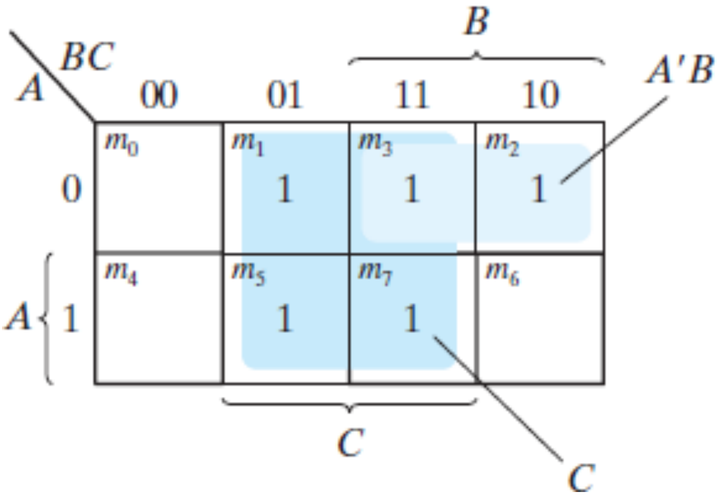


FIGURE 1.20 Map of Example

$$A'C + A'B + AB'C + BC = C + A'B$$

1.8 FOUR-VARIABLE K-MAP

- In Fig. 1.21(a) are listed the 16 minterms and the squares assigned to each. In Fig. 1.21(b), the map is redrawn to show the relationship between the squares and the four variables.
- The rows and columns are numbered in a Gray code sequence, with only one digit changing value between two adjacent rows or columns. The minterm corresponding to each square can be obtained from the concatenation of the row number with the column number.
- For example, the numbers of the third row (11) and the second column (01),when concatenated, give the binary number 1101, the binary equivalent of decimal 13. Thus, the square in the third row and second column represents minterm *m*13.
- The map minimization of four-variable Boolean functions is similar to the method used to minimize three-variable functions. Adjacent squares are defined to be squares next to each other. In addition, the map is considered to lie on a surface with the top and bottom edges, as well as the right and left edges, touching each other to form adjacent squares. For example, *m*0 and *m*2 form adjacent squares, as do *m*3 and *m*11. The combination of adjacent squares that is useful during the simplification process is easily determined from inspection of the four-variable map:
  - One square represents one minterm, giving a term with four literals.
  - Two adjacent squares represent a term with three literals.
  - Four adjacent squares represent a term with two literals.
  - Eight adjacent squares represent a term with one literal.
  - Sixteen adjacent squares produce a function that is always equal to 1.

No other combination of squares can simplify the function. The next two examples show the procedure used to simplify four-variable Boolean functions.

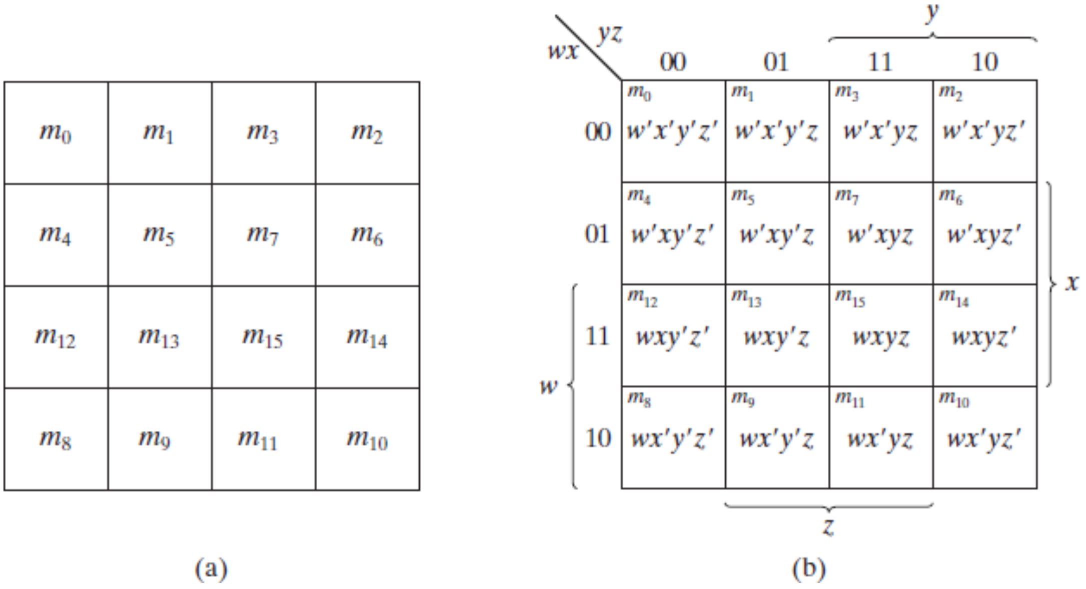

m_0 m_1 m_3 m_2
m_4 m_5 m_7 m_6
m_{12} m_{13} m_{15} m_{14}
m_8 m_9 m_{11} m_{10}
(a)
wx yz
00 01 11 10
y
x
w
z
00 $w'x'y'z'$ $w'x'y'z$ $w'x'yz$ $w'x'yz'$
01 $w'xy'z'$ $w'xy'z$ $w'xyz$ $w'xyz'$
11 $wxy'z'$ $wxy'z$ $wxyz$ $wxyz'$
10 $wx'y'z'$ $wx'y'z$ $wx'yz$ $wx'yz'$
(b)

FIGURE 1.21 Four-variable map

EXAMPLE 1.8 : Simplify the Boolean function

$$F(w, x, y, z) = \Sigma(0, 1, 2, 4, 5, 6, 8, 9, 12, 13, 14)$$

Since the function has four variables, a four-variable map must be used. The minterms listed in the sum are marked by 1’s in the map of Fig. 1.22 . Eight adjacent squares marked with 1’s can be combined to form the one literal term *y*!. The remaining three 1’s on the right cannot be combined to give a simplified term; they must be combined as two or four adjacent squares. The larger the number of squares combined, the smaller is the number of literals in the term. In this example, the top two 1’s on the right are combined with the top two 1’s on the left to give the term *w*’*z*’ . Note that it is permissible to use the same square more than once. We are now left with a square marked by 1 in the third row and fourth column (square 1110). Instead of taking this square alone (which will give a term with four literals), we combine it with squares already used to form an area of four adjacent squares. These squares make up the two middle rows and the two end columns, giving the term *xz*’. The simplified function is

$$F = y' + w'z' + xz'$$

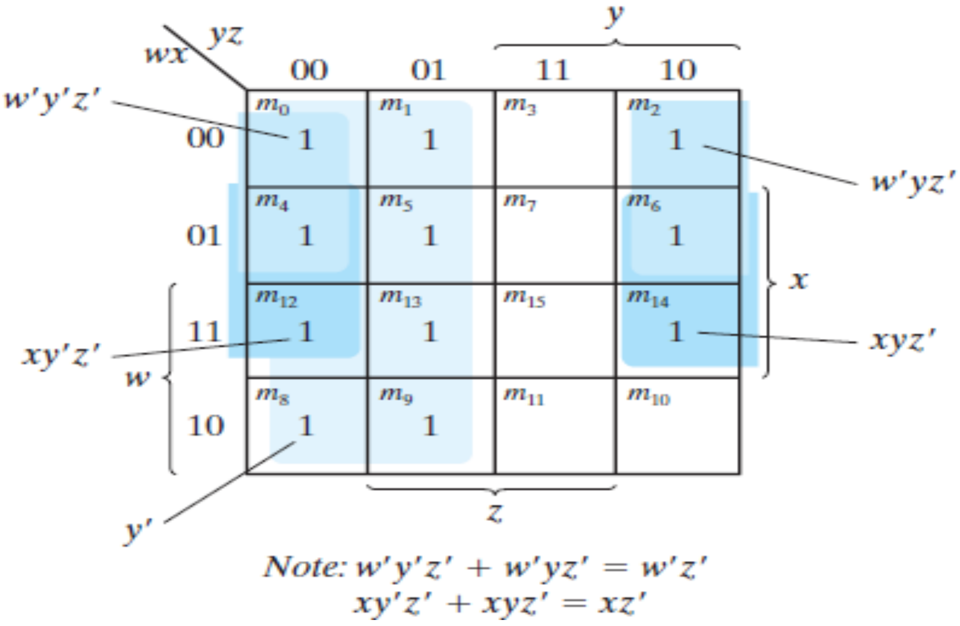


FIGURE 1.22 Map for Example

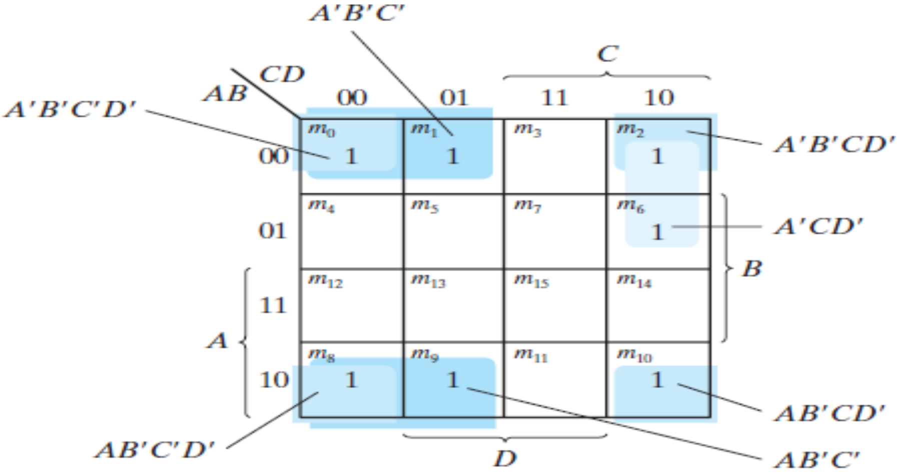
$$F(w, x, y, z) = \Sigma(0, 1, 2, 4, 5, 6, 8, 9, 12, 13, 14)$$

$$y' + w'z' + xz'$$

EXAMPLE 1.9 : Simplify the Boolean function

$$F = A'B'C' + B'CD' + A'BCD' + AB'C'$$

The area in the map covered by this function consists of the squares marked with 1's in Fig1.23 .



Note: $A'B'C'D' + A'B'CD' = A'B'D'$
$AB'C'D' + AB'CD' = AB'D'$
$A'B'D' + AB'D' = B'D'$
$A'B'C' + AB'C' = B'C'$

FIGURE1.23 Map for Example 3.6, $A'B'C' + B'CD' + A'BCD' + AB'C' = B'D' + B'C' + A'CD'$

- The function has four variables and, as expressed, consists of three terms withthree literals each and one term with four literals. Each term with three literals is represented in the map by two squares.
- For example, *A'B'C'* is represented in squares 0000 and 0001. The function can be simplified in the map by taking the 1's in the four corners to give the term *B'D'*. This is possible because these four squares are adjacent when the map is drawn in a surface with top and bottom edges, as well as left and right edges, touching one another. The two left-hand 1's in the top row are combined with the two 1's in the bottom row to give the term *B'C'*. The remaining 1 may be combined in a two square area to give the term *A'CD'*. The simplified function is

$$F = B'D' + B'C' + A'CD'$$

Prime Implicants

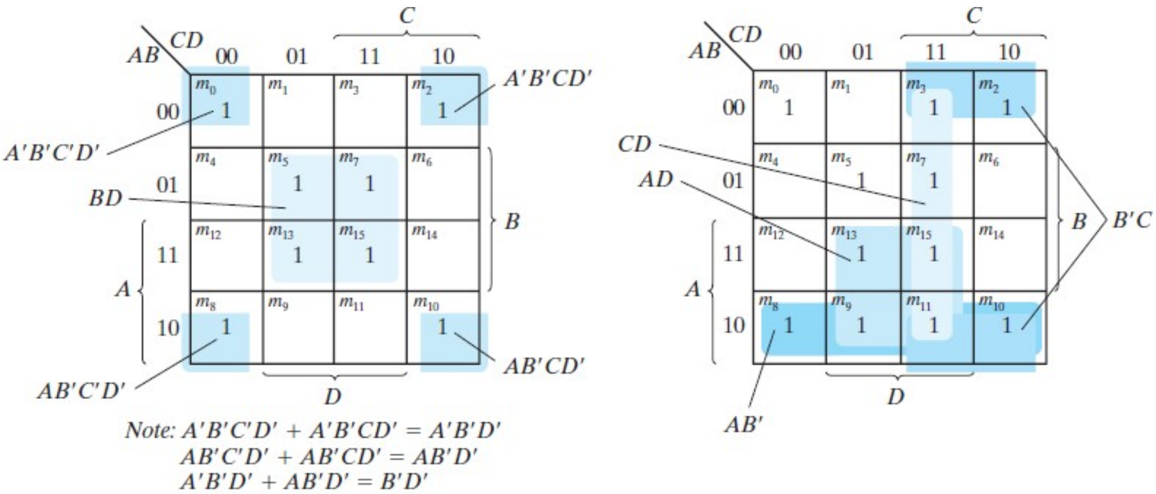
- In choosing adjacent squares in a map, we must ensure that
    - all the minterms of the function are covered when we combine the squares,
    - the number of terms in the expression is minimized, and
    - there are no redundant terms
- Sometimes there may be two or more expressions that satisfy the simplification criteria. The procedure for combining squares in the map may be made more systematic if we understand the meaning of two special types of terms.

- **A *prime implicant* is a product term obtained by combining the maximum possible number of adjacent squares in the map.** If a minterm in a square is covered by only one prime implicant, that prime implicant is said to be *essential.*
- **The prime implicants of a function can be obtained from the map by combining all possible maximum numbers of squares.** This means that a single 1 on a map represents a prime implicant if it is not adjacent to any other 1's. Two adjacent 1's form a prime implicant, provided that they are not within a group of four adjacent squares. Four adjacent 1's form a prime implicant if they are not within a group of eight adjacent
- squares, and so on. The essential prime implicants are found by looking at each square marked with a 1 and checking the number of prime implicants that cover it. The prime implicant is essential if it is the only prime implicant that covers the minterm.

Consider the following four-variable Boolean function:

$$F(A, B, C, D) = \Sigma(0, 2, 3, 5, 7, 8, 9, 10, 11, 13, 15)$$

- The minterms of the function are marked with 1's in the maps of Fig. 1.24. The partial map (Fig. 1.24(a)) shows two essential prime implicants, each formed by collapsing four cells into a term having only two literals. One term is essential because there is only one way to include minterm *m*0 within four adjacent squares. These four squares define the term *B*'*D*'.
- Similarly, there is only one way that minterm *m*5 can be combined with four adjacent squares, and this gives the second term *BD* . The two essential prime implicants cover eight minterms. The three minterms that were omitted from the partial map (*m*3, *m*9, and *m*11) must be considered next. Figure 1.24 (b) shows all possible ways that the three minterms can be covered with prime implicants. Minterm *m*3 can be covered with either prime implicant *CD* or prime implicant *B*'*C.* Minterm *m*9 can be covered with either *AD* or *AB*'. Minterm *m*11 is covered with any one of the four prime implicants.
- The simplified expression is obtained from the logical sum of the two essential prime implicants and any two prime implicants that cover minterms *m*3, *m*9, and *m*11.



Note: $A'B'C'D' + A'B'CD' = A'B'D'$
$AB'C'D' + AB'CD' = AB'D'$
$A'B'D' + AB'D' = B'D'$

(a) Essential prime implicants BD and $B'D'$

(b) Prime implicants CD, $B'C$, AD, and AB'

FIGURE1.24 Simplification using prime implicants

- There are four possible ways that the function can be expressed with four product terms of two literals each:

$$\begin{aligned} F &= BD + B'D' + CD + AD \\ &= BD + B'D' + CD + AB' \\ &= BD + B'D' + B'C + AD \\ &= BD + B'D' + B'C + AB' \end{aligned}$$

- The previous example has demonstrated that the identification of the prime implicants in the map helps in determining the alternatives that are available for obtaining a simplified expression.
- The procedure for finding the simplified expression from the map requires that we first determine all the essential prime implicants.
- The simplified expression is obtained from the logical sum of all the essential prime implicants, plus other prime implicants that may be needed to cover any remaining minterms not covered by the essential prime implicants.
- Occasionally, there may be more than one way of combining squares, and each combination may produce an equally simplified expression.

Five-Variable Map

- Maps for more than four variables are not as simple to use as maps for four or fewer variables.
- A five-variable map needs 32 squares and a six-variable map needs 64 squares.
- When the number of variables becomes large, the number of squares becomes excessive and the geometry for combining adjacent squares becomes more involved.
- Maps for more than four variables are difficult to use and will not be considered here.

1.9 DON' T-CARE CONDITIONS

- The logical sum of the minterms associated with a Boolean function specifies the conditions under which the function is equal to 1. The function is equal to 0 for the rest of the minterms. This pair of conditions assumes that all the combinations of the values for the variables of the function are valid.
- In practice, in some applications the function is not specified for certain combinations of the variables. As an example, the four-bit binary code for the decimal digits has six combinations that are not used and consequentlyv are considered to be unspecified. Functions that have unspecified outputs for some input combinations are called *incompletely specified functions* .
- In most applications, we simply don't care what value is assumed by the function for the unspecified minterms. For this reason, it is customary to call the unspecified minterms of a function *don't-care conditions* . These don't-care conditions can be used on a map to provide further simplification of the Boolean expression.
- A don't-care minterm is a combination of variables whose logical value is not specified. Such a minterm cannot be marked with a 1 in the map, because it would require that the function always be a 1 for such a combination. Likewise, putting a 0 on the square requires the function to be 0. To distinguish the don't-care condition from 1's and 0's, an X is used. Thus, an X inside a square in the map indicates that we don't care whether the value of 0 or 1 is assigned to *F* for the particular minterm.
- In choosing adjacent squares to simplify the function in a map, the don't-care minterms may be assumed to be either 0 or 1. When simplifying the function, we can choose to include each don't-care minterm with either the 1's or the 0's, depending on which combination gives the simplest expression.

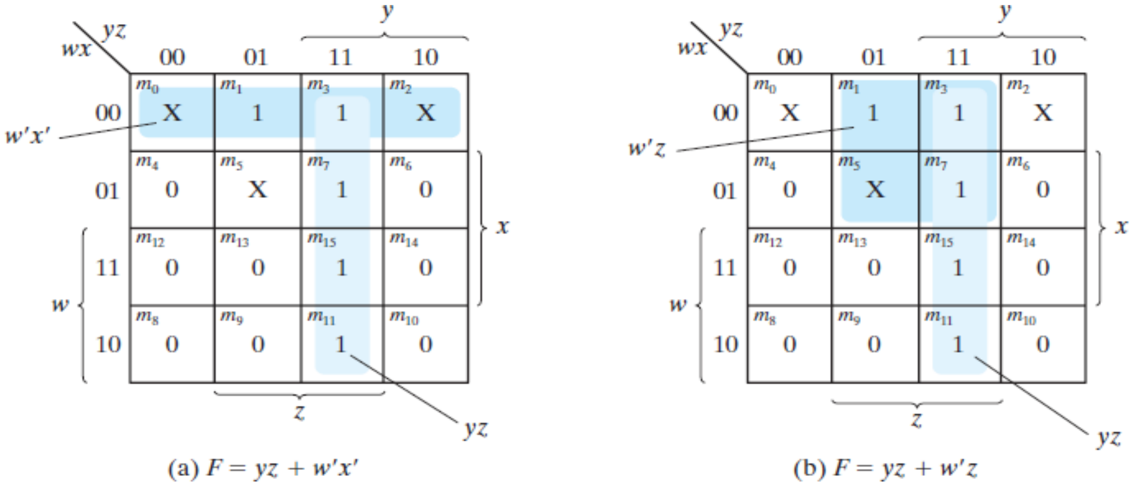
EXAMPLE 1.10 : Simplify the Boolean function

$$F(w, x, y, z) = \Sigma(1, 3, 7, 11, 15)$$

which has the don't-care conditions

$$d(w, x, y, z) = \Sigma(0, 2, 5)$$

- The minterms of *F* are the variable combinations that make the function equal to 1. The minterms of *d* are the don't-care minterms that may be assigned either 0 or 1. The map simplification is shown in Fig. 1.25 . The minterms of *F* are marked by 1's, those of *d* are marked by X's, and the remaining squares are filled with 0's. To get the simplified expression in sum-of-products form, we must include all five 1's in the map, but we may or may not include any of the X's, depending on the way the function is simplified.
- The term *yz* covers the four minterms in the third column.



(a) $F = yz + w'x'$ (b) $F = yz + w'z$

FIGURE 1.25 Example with don't-care conditions

- The remaining minterm, *m*1, can be combined with minterm *m*3 to give the three-literal term *w'x'z*. However, by including one or two adjacent X's we can combine four adjacent squares to give a two-literal term. In Fig. 1.25(a), don't-care minterms 0 and 2 are included with the 1's, resulting in the simplified Function

$F = yz + w'x'$

- In Fig. 1.25(b), don't-care minterm 5 is included with the 1's, and the simplified function is now

$F = yz + w'z$

- Either one of the preceding two expressions satisfies the conditions stated for this example all y marked with X's and are considered as being either 0 or 1. The choice between 0 and 1 is made depending on the way the incompletely specified function is simplified. Once the choice is made, the simplified function obtained will consist of a sum of minterms that includes those minterms which were initially unspecified and have been chosen to be included with the 1's. Consider the two simplified expressions obtained in Example 1.10 :

$$F(w,x,y,z) = yz + w'x' = \Sigma(0,1,2,3,7,11,15)$$
$$F(w,x,y,z) = yz + w'z = \Sigma(1,3,5,7,11,15)$$

- Both expressions include minterms 1, 3, 7, 11, and 15 that make the function *F* equal to 1. The don't-care minterms 0, 2, and 5 are treated differently in each expression. The first expression includes minterms 0 and 2 with the 1's and leaves minterm 5 with the 0's. The second expression includes minterm 5 with the 1's and leaves minterms 0 and 2 with the 0's. The two expressions represent two functions that are not algebraically equal. Both cover the specified minterms of the function, but each covers different
- don't-care minterms. As far as the incompletely specified function is concerned, either expression is acceptable because the only difference is in the value of *F* for the don't-care minterms.
- It is also possible to obtain a simplified product-of-sums expression for the function of Fig. 1.25 . In this case, the only way to combine the 0's is to include don't-care minterms 0 and 2 with the 0's to give a simplified complemented function:

$$F' = z' + wy'$$

- Taking the complement of *F*" gives the simplified expression in product-of-sums form:

$$F(w,x,y,z) = z(w' + y) = \Sigma(1,3,5,7,11,15)$$

-

- In this case, we include minterms 0 and 2 with the 0's and minterm 5 with the 1's.

1.9 NAND and NOR IMPLEMENTATION

Explain the importance of NAND and NOR logic gate implementation.

Digital circuits are frequently constructed with NAND or NOR gates rather than with AND and OR gates. NAND and NOR gates are easier to fabricate with electronic components and are the basic gates used in all IC digital logic families.

NAND Circuits

- The NAND gate is said to be a **universal gate** because any logic circuit can be implemented with it. To show that any Boolean function can be implemented with NAND gates, we need only show that the logical operations of AND, OR, and complement can be obtained with NAND gates alone.
- **A convenient way to implement a Boolean function with NAND gates is to obtain the simplified Boolean function in terms of Boolean operators and then convert the function to NAND logic.** The conversion of an algebraic expression from AND, OR, and complement to NAND can be done by simple circuit manipulation techniques that change AND–OR diagrams to NAND diagrams.
- To facilitate the conversion to NAND logic, it is convenient to define an alternative graphic symbol for the gate. Two equivalent graphic symbols for the NAND gate are shown in Fig. 1.27 .
- The AND-invert symbol has been defined previously and consists of an AND graphic symbol followed by a small circle negation indicator referred to as a bubble.
- Alternatively, it is possible to represent a NAND gate by an OR graphic symbol that is preceded by a bubble in each input.
- The invert-OR symbol for the NAND gate follows DeMorgan’s theorem and the convention that the negation indicator (bubble) denotes complementation. The two graphic symbols’ representations are useful in the analysis and design of NAND circuits.
- When both symbols are mixed in the same diagram, the circuit is said to be in mixed notation.

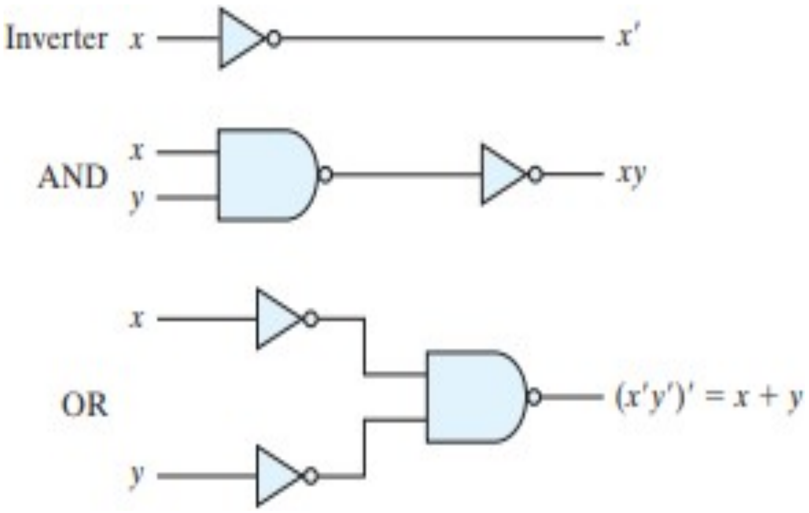


FIGURE 1.26 Logic operations with NAND gates

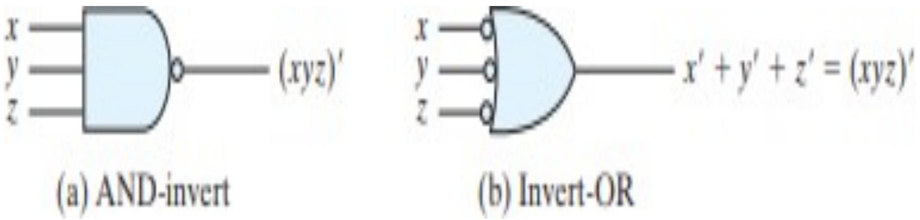


FIGURE 1.27 Two graphic symbols for a three-input NAND gate

Two-Level Implementation

- **The implementation of Boolean functions with NAND gates requires that the functions be in sum-of-products form.** To see the relationship between a sum-of-products expression and its equivalent NAND implementation, consider the logic diagrams drawn in Fig. 1.28 .
- All three diagrams are equivalent and implement the function

F = AB + CD

- The function is implemented in Fig. 1.28(a) with AND and OR gates. In Fig. 1.28(b), the AND gates are replaced by NAND gates and the OR gate is replaced by a NAND gate with an OR-invert graphic symbol.
- Remember that a bubble denotes complementation and two bubbles along the same line represent double complementation, so both can be removed. Removing the bubbles on the gates of (b) produces the circuit of (a).
- Therefore, the two diagrams implement the same function and are equivalent.

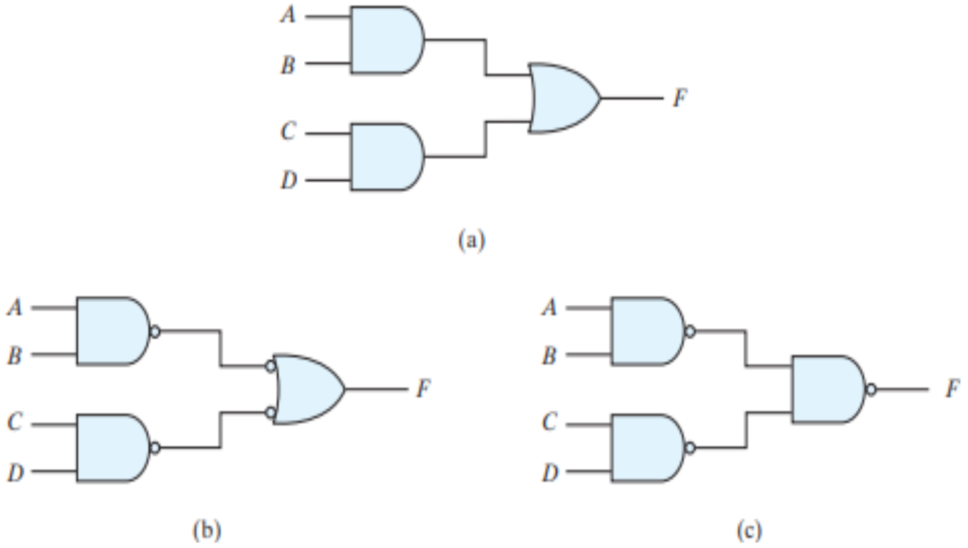


FIGURE 1.28 Three ways to implement F = AB + CD

- In Fig. 1.28 (c), the output NAND gate is redrawn with the AND-invert graphic symbol. In drawing NAND logic diagrams, the circuit shown in either Fig. 1.28(b) or (c) is acceptable. The one in Fig. 1.28(b) is in mixed notation and represents a more direct relationship to the Boolean expression it implements.
- The NAND implementation in Fig. 1.28 (c) can be verified algebraically. The function it implements can easily be converted to sum- of products form by DeMorgan's theorem:

$$F = ((AB)'(CD)')' = AB + CD$$

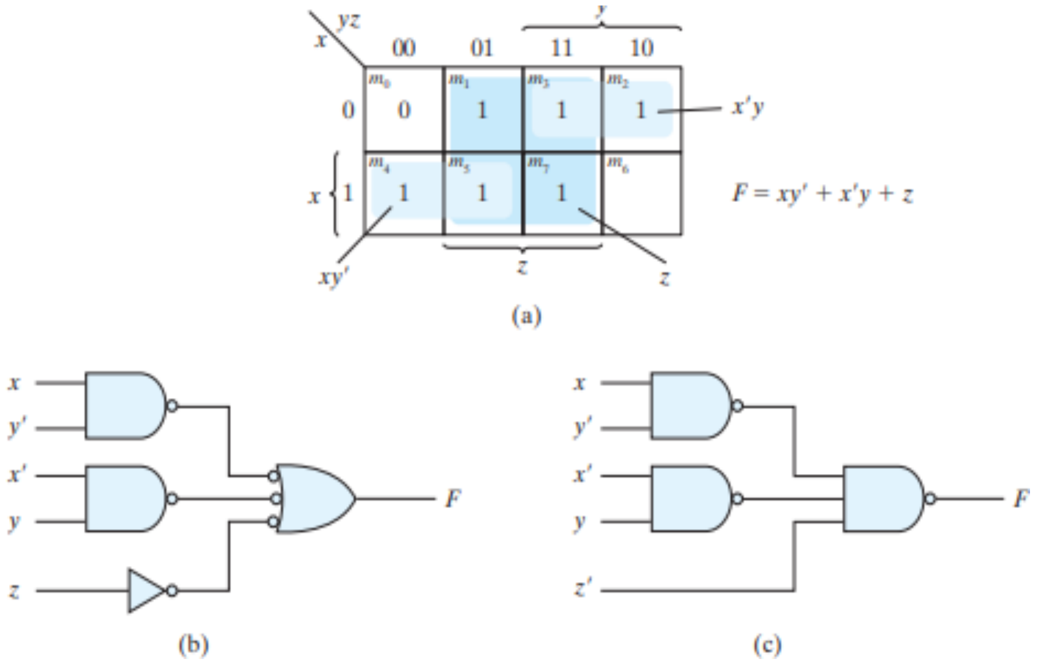
EXAMPLE 1.11 Implement the following Boolean function with NAND gates:

F (x, y, z) = (1, 2, 3, 4, 5, 7)

- The first step is to simplify the function into sum-of-products form. This is done by means of the map of Fig. 1.29 (a), from which the simplified function is obtained:

$$F = xy' + x'y + z$$

- The two-level NAND implementation is shown in Fig. 1.29 (b) in mixed notation.
- Note that input z must have a one-input NAND gate (an inverter) to compensate for the bubble in the second-level gate.
- An alternative way of drawing the logic diagram is given in Fig. 1.29 (c).
- Here, all the NAND gates are drawn with the same graphic symbol. The inverter with input z has been removed, but the input variable is complemented and denoted by z'.



(a)

(b)

(c)

FIGURE 1.29 Solution to Example 1.9

- The procedure described in the previous example indicates that a Boolean function can be implemented with two levels of NAND gates. The procedure for obtaining the logic diagram from a Boolean function is as follows:

  1. Simplify the function and express it in sum-of-products form.

  2. Draw a NAND gate for each product term of the expression that has at least two literals. The inputs to each NAND gate are the literals of the term. This procedure produces a group of first-level gates.

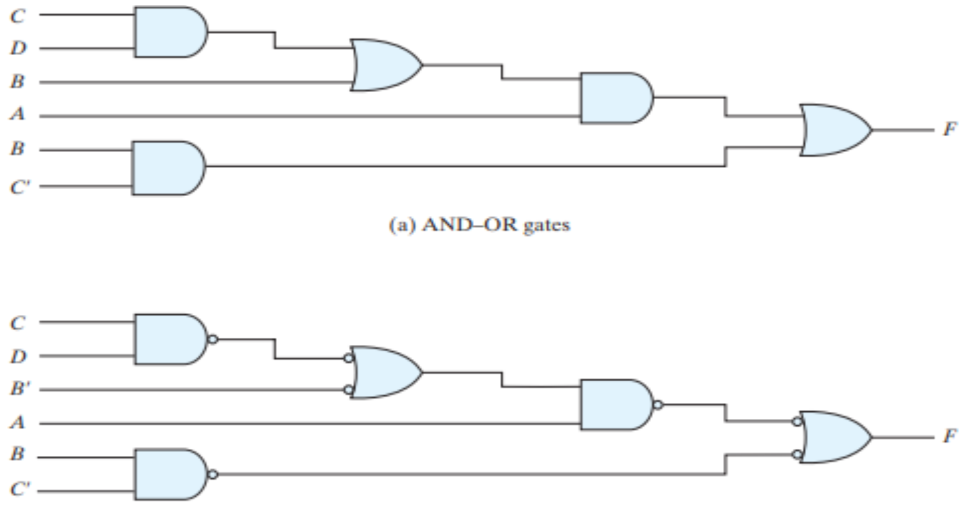
  4. Draw a single gate using the AND-invert or the invert-OR graphic symbol in the second level, with inputs coming from outputs of first-level gates.

  5. A term with a single literal requires an inverter in the first level. However, if the single literal is complemented, it can be connected directly to an input of the second level NAND gate.

Multilevel NAND Circuits

- The standard form of expressing Boolean functions results in a two-level implementation. There are occasions, however, when the design of digital systems results in gating structures with three or more levels.
- The most common procedure in the design of multilevel circuits is to express the Boolean function in terms of AND, OR, and complement operations.
- The function can then be implemented with AND and OR gates. After that, if necessary, it can be converted into an all-NAND circuit. Consider, for example, the Boolean function

$$F = A(CD + B) + BC'$$



(a) AND–OR gates

(b) NAND gates

FIGURE 1.30 Implementing F = A(CD + B) + BC'

- Although it is possible to remove the parentheses and reduce the expression into a standard sum-of-products form, we choose to implement it as a multilevel circuit for illustration.
- The AND–OR implementation is shown in Fig. 1.30 (a). There are four levels of gating in the circuit. The first level has two AND gates.
- The second level has an OR gate followed by an AND gate in the third level and an OR gate in the fourth level. A logic diagram with a pattern of alternating levels of AND and OR gates can easily be converted into a NAND circuit with the use of mixed notation, shown in Fig. 1.30 (b).
- The procedure is to change every AND gate to an AND-invert graphic symbol and every OR gate to an invert-OR graphic symbol. The NAND circuit performs the same logic as the AND–OR diagram as long as there are two bubbles along the same line.
- The bubble associated with input B causes an extra complementation, which must be compensated for by changing the input literal to B".
- The general procedure for converting a multilevel AND–OR diagram into an all-NAND diagram using mixed notation is as follows:
    1. Convert all AND gates to NAND gates with AND-invert graphic symbols.
    2. Convert all OR gates to NAND gates with invert-OR graphic symbols.
    3. Check all the bubbles in the diagram. For every bubble that is not compensated by another small circle along the same line, insert an inverter or complement the input literal.

As another example, consider the multilevel Boolean function

$$F = (AB' + A'B)(C + D')$$

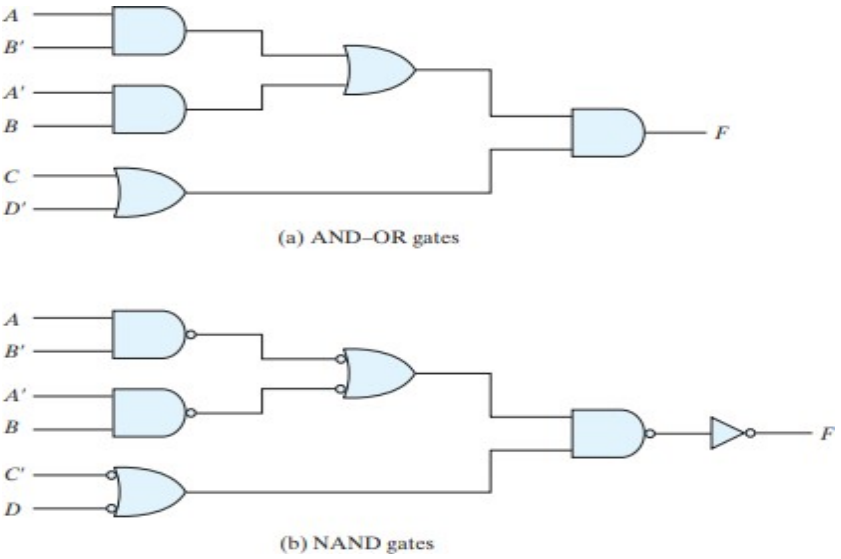


FIGURE 1.31 Implementing F = (AB’ + A’B) (C + D’)

- The AND–OR implementation of this function is shown in Fig. 1.31 (a) with three levels of gating. The conversion to NAND with mixed notation is presented in Fig. 1.31(b) of the diagram.
- The two additional bubbles associated with inputs C and D' cause these two literals to be complemented to C' and D .
- The bubble in the output NAND gate complements the output value, so we need to insert an inverter gate at the output in order to complement the signal again and get the original value back.

NOR Implementation

- The NOR operation is the dual of the NAND operation. The NOR gate is another universal gate that can be used to implement any Boolean function. The implementation of the complement, OR, and AND operations with NOR gates is shown in Fig. 1.32 .
- The complement operation is obtained from a oneinput NOR gate that behaves exactly like an inverter. The OR operation requires two NOR gates, and the AND operation is obtained with a NOR gate that has inverters in each input.
- The two graphic symbols for the mixed notation are shown in Fig. 1.33 .
- The OR-invert symbol defines the NOR operation as an OR followed by a complement. The invert-AND symbol complements each input and then performs an AND operation.
- The two symbols designate the same NOR operation and are logically identical because of DeMorgan's theorem.

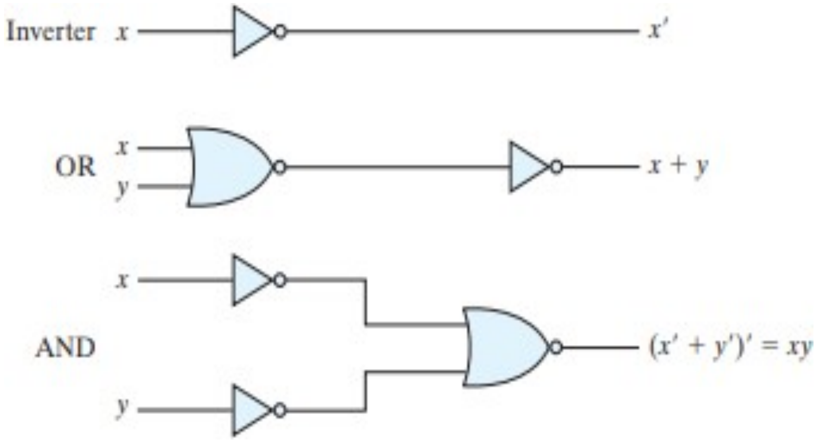


FIGURE 1.32 Logic operations with NOR gates

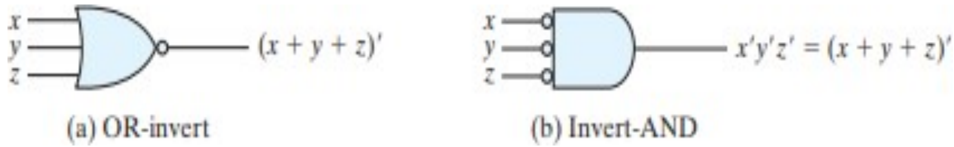


FIGURE 1.33 Two graphic symbols for the NOR gate

- A two-level implementation with NOR gates requires that the function be simplified into product-of-sums form. Remember that the simplified product-of-sums expression is obtained from the map by combining the 0's and complementing.
- A product-of-sums expression is implemented with a first level of OR gates that produce the sum terms followed by a second-level AND gate to produce the product.
- The transformation from the OR–AND diagram to a NOR diagram is achieved by changing the OR gates to NOR gates with OR-invert graphic symbols and the AND gate to a NOR gate with an invert-AND graphic symbol.
- A single literal term going into the second-level gate must be complemented. Figure 1.34, shows the NOR implementation of a function expressed as a product of sums:

$$F = (A + B)(C + D)E$$

- The OR–AND pattern can easily be detected by the removal of the bubbles along the same line. Variable E is complemented to compensate for the third bubble at the input of the second-level gate.
- The procedure for converting a multilevel AND–OR diagram to an all-NOR diagram is similar to the one presented for NAND gates. For the NOR case, we must convert each OR gate to an OR-invert symbol and each AND gate to an invert-AND symbol. Any bubble that is not compensated by another bubble along the same line needs an inverter, or the complementation of the input literal.
- The transformation of the AND–OR diagram of Fig. 1.31 (a) into a NOR diagram is shown in Fig. 1.35 .
- The Boolean function for this circuit is

$$F = (AB' + A'B)(C + D')$$

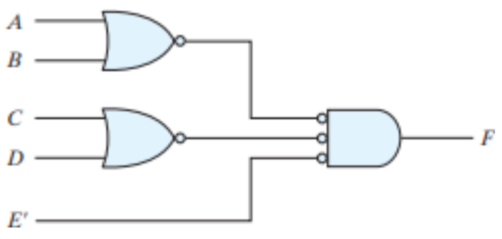


FIGURE 1.34 Implementing F = (A + B)(C + D)E

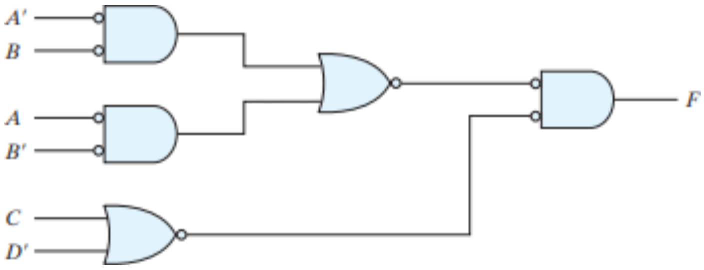


FIGURE 1.35 Implementing F = (AB’ + A,B)(C + D’)with NOR gates

- The equivalent AND–OR diagram can be recognized from the NOR diagram by removing all the bubbles.
- To compensate for the bubbles in four inputs, it is necessary to complement the corresponding input literals.

1.11 HARDWARE DESCRIPTION LANGUAGE

Write a short note on Hardware Description Language.

- **Manual vs. Computer-Based Design:** Manual methods for designing logic circuits are practical only for small circuits. For larger, practical circuits, computer-based design tools are essential. These tools reduce the risk of errors and leverage the designer's creativity.
- **Hardware Description Language (HDL):** An HDL is a computer-based language used to describe the hardware of digital systems in textual form. It's specialized for representing hardware structures and logic circuit behavior. HDLs enable the representation of logic diagrams, truth tables, Boolean expressions, and complex system behaviors.
- **Documentation and Exchange:** HDLs serve as documentation languages, allowing both humans and computers to read, edit, store, and transmit digital system descriptions efficiently. They facilitate communication between designers.
- **HDL in Design Flow:** HDLs are used in various stages of integrated circuit design, including design entry, functional simulation, logic synthesis, timing verification, and fault simulation.
- **Design Entry:** Designers use HDLs to describe the functionality they want to implement in hardware. This can take various forms, including Boolean logic equations, truth tables, netlists of interconnected gates, or abstract behavioral models.
- **Functional Simulation:** HDLs are used with simulators to predict how a digital system will behave before it's physically built. Test benches are created to test the design's functionality and detect errors.

- **Logic Synthesis:** Logic synthesis translates the HDL description into a netlist, specifying the physical components and their interconnections. It's akin to compiling a high-level program, but it produces a database for circuit fabrication.
- **Timing Verification:** Timing verification checks signal paths to ensure they are not compromised by propagation delays, confirming that the circuit will operate at the specified speed.
- **Fault Simulation:** In VLSI design, fault simulation identifies differences between ideal and flawed circuits caused by manufacturing issues. It generates test patterns to ensure only fault-free devices are shipped to customers.
- **HDL Standards:** Two widely used HDLs supported by the IEEE are VHDL and Verilog. VHDL was mandated by the Department of Defense, while Verilog is more widely used due to its ease of learning and use.
- **Choice of Verilog:** The content of the book focuses on Verilog as it's considered easier to learn and use compared to VHDL. It emphasizes computer-aided modeling of digital systems using Verilog for modeling, verification, and synthesis.
- **Evolution of Verilog:** The Verilog HDL has evolved over the years and was initially approved as a standard in 1995, with revisions and enhancements approved in 2001 and 2005. The book covers features of Verilog that support HDL-based design methodology for integrated circuits**.**

Module Declaration

- **Keywords**: Verilog uses keywords, which are predefined lowercase identifiers that define language constructs. Examples of keywords include module, endmodule, input, output, wire, and, or, and not. Keywords are displayed in boldface in code examples.
- **Comments:** Comments in Verilog are indicated by double forward slashes (//) and extend to the end of the line. Comments do not affect the simulation.
- **Multiline Comments:** Multiline comments are enclosed between /* and */.
- **Case Sensitivity:** Verilog is case-sensitive, meaning uppercase and lowercase letters are distinct. For example, not is not the same as NOT.
- **Modules**: A Verilog model is composed of one or more modules. A module is declared using the module keyword and terminated with endmodule. Modules are the fundamental descriptive units in Verilog.

- **Module Name:** A module declaration includes a name, which is an identifier. Identifiers are composed of alphanumeric characters and underscores (_). They must start with an alphabetic character or underscore but cannot start with a number.
- **Port Lists:** After the module name, a list of ports is specified. Ports define the inputs and outputs of the module.
- **Combinational Logic**: Verilog can describe combinational logic using various methods, including schematic connections of gates, Boolean equations, or truth tables.
- **Example Circuit:** The text provides an example circuit described in Verilog HDL (HDL Example p1.1) to illustrate the language's usage. Shown in figure 1.36

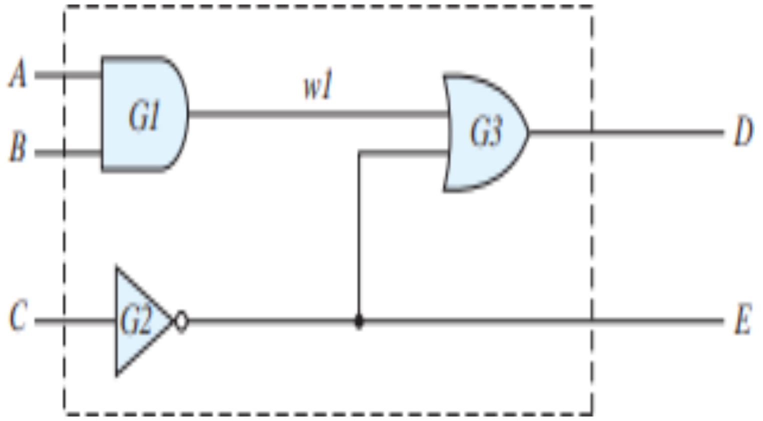
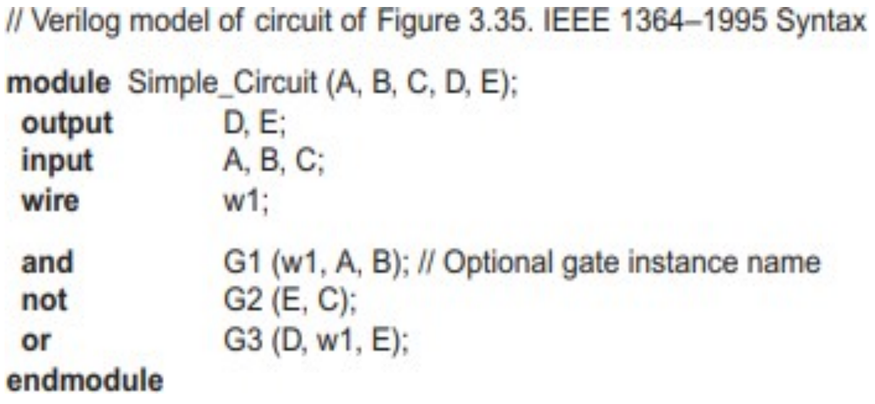


FIGURE 1.36 Circuit to demonstrate an HDL

HDL Example p1.1 (Combinational Logic Modeled with Primitives)

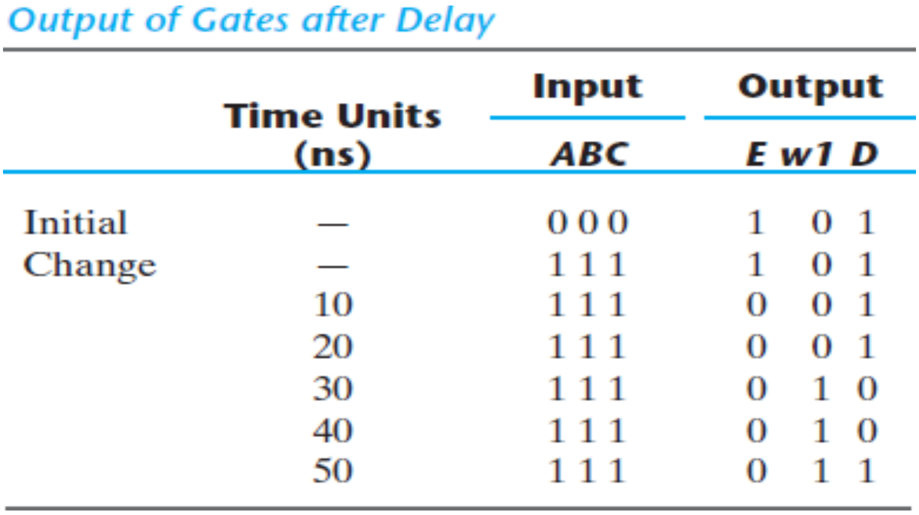


```
// Verilog model of circuit of Figure 3.35. IEEE 1364–1995 Syntax

module Simple_Circuit (A, B, C, D, E);
 output        D, E;
 input         A, B, C;
 wire          w1;

 and           G1 (w1, A, B); // Optional gate instance name
 not           G2 (E, C);
 or            G3 (D, w1, E);
endmodule
```

- **Port List:** The port list of a Verilog module defines the interface between the module and its environment, specifying inputs and outputs. It is enclosed in parentheses, separated by commas, and terminated with a semicolon.

- **Input and Output Ports**: The keywords input and output are used to specify which ports are inputs and which are outputs. Internal connections within the module are declared as wires.
- **Internal Connections:** Internal connections within the module are declared using the keyword wire.
- **Primitive Gates:** The structure of the circuit is specified using predefined primitive gates (e.g., and, or, not). Each gate is identified by a descriptive keyword, and gate instances are created with optional names, followed by the gate's output and inputs in parentheses.
- **Gate Instantiation:** Each gate instantiation consists of an optional name, the gate's output, and its inputs, separated by commas within parentheses. The output is always listed first.
- **Declaration vs. Instantiation:** Modules are declared to specify their input-output behavior, while predefined primitives (gates) are instantiated to populate the design. Primitives are not declared since their definition is predefined in the language.
- **Descriptive Model:** Verilog HDL is not a computational model like regular programming languages. The order of statements in the model does not imply a sequence of computations. It is a descriptive model that defines what primitives make up a circuit and how they are connected.
- **Behavior Specification:** The input-output behavior of the circuit is implicitly specified within the model because the behavior of each logic gate is predefined. This allows for simulating the represented circuit using the HDL-based model.

Table 1.3



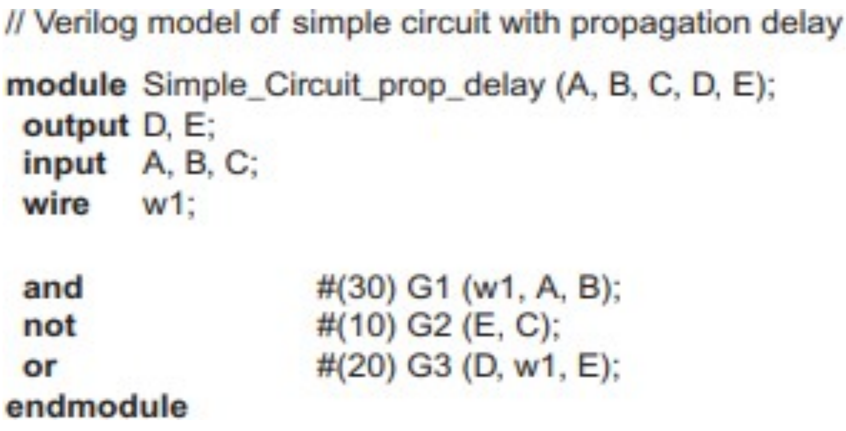
Output of Gates after Delay

	Time Units (ns)	Input	Output
		ABC	E w1 D
Initial	—	0 0 0	1 0 1
Change	—	1 1 1	1 0 1
	10	1 1 1	0 0 1
	20	1 1 1	0 0 1
	30	1 1 1	0 1 0
	40	1 1 1	0 1 0
	50	1 1 1	0 1 1

Gate Delays

- In Verilog, propagation delays in physical circuits are specified using time units and the '#' symbol. The time units are dimensionless, and the timescale is established with a compiler directive. For example, "timescale 1ns/100ps" sets the time unit to 1 nanosecond (ns) and the precision to 0.1 picoseconds (ps).
- Let's consider an example circuit described in HDL (Hardware Description Language) with specified gate delays. The circuit has AND, OR, and NOT gates with delays of 30 ns, 20 ns, and 10 ns, respectively.
- When simulating this circuit and transitioning its inputs from A, B, C = 0 to A, B, C = 1, the outputs change as follows (using the specified gate delays and assuming the default time unit is 1 ns):
- The output of the inverter at E changes from 1 to 0 after a 10-ns delay due to the NOT gate's delay.
- The output of the AND gate at w1 changes from 0 to 1 after a 30-ns delay due to the AND gate's delay.
- The output of the OR gate at D changes from 1 to 0 at t = 30 ns due to the OR gate's delay and then changes back to 1 at t = 50 ns. This change in the OR gate's output is a result of a change in its inputs 20 ns earlier.
- This behavior reveals that the gate delays in the circuit introduce a negative spike in the output waveform. Specifically, for output D, there is a 20-ns period during which the output is 0 before it returns to its final value of 1.
- This spike is a consequence of the delay in the gates and is crucial to consider when designing and analyzing digital circuits, especially in timing-critical applications.

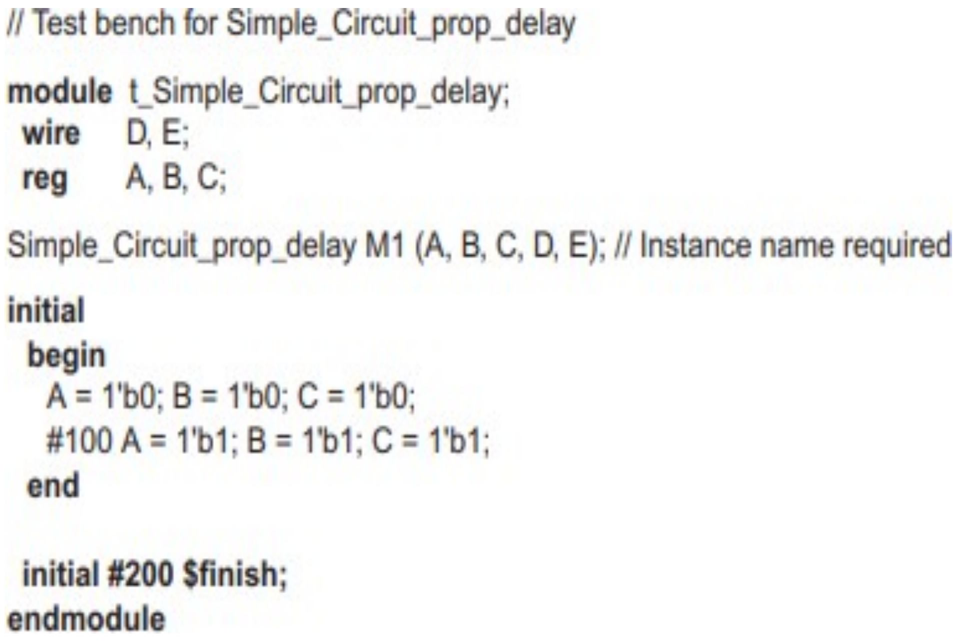
HDL Example p1. 2 (Gate-Level Model with Propagation Delays)



```
// Verilog model of simple circuit with propagation delay

module Simple_Circuit_prop_delay (A, B, C, D, E);
 output D, E;
 input  A, B, C;
 wire   w1;

 and            #(30) G1 (w1, A, B);
 not            #(10) G2 (E, C);
 or             #(20) G3 (D, w1, E);
endmodule
```

- In order to simulate a circuit with an HDL, it is necessary to apply inputs to the circuit so that the simulator will generate an output response. An HDL description that provides the stimulus to a design is called a test bench.
- HDL Example p1.3 shows a test bench for simulating the circuit with delay. (Note the distinguishing name Simple_Circuit_prop_ delay .) In its simplest form, a test bench is a module containing a signal generator and an instantiation of the model that is to be verified.
- Note that the test bench (t_Simple_ Circuit_prop_delay) has no input or output ports, because it does not interact with its environment. In general, we prefer to name the test bench with the prefix t_ concatenated with the name of the module that is to be tested by the test bench, but that choice is left to the designer. Within the test bench, the inputs to the circuit are declared with keyword reg and the outputs are declared with the keyword wire .
- The module Simple_Circuit_ prop_delay is instantiated with the instance name M1. Every instantiation of a module must include a unique instance name. Note that using a test bench is similar to testing actual hardware by attaching signal generators to the inputs of a circuit and attaching probes (wires) to the outputs of the circuit.
- The "initial" keyword is employed in the test bench to define the initial conditions. In this case, the initial statements indicate that A, B, and C are initially set to "1!b0," representing one binary digit with a value of 0. After 100 ns, these inputs transition to A, B, C = 1.

HDL Example p1.3 (Test Bench)



```
// Test bench for Simple_Circuit_prop_delay

module t_Simple_Circuit_prop_delay;
  wire    D, E;
  reg     A, B, C;

Simple_Circuit_prop_delay M1 (A, B, C, D, E); // Instance name required

initial
  begin
    A = 1'b0; B = 1'b0; C = 1'b0;
    #100 A = 1'b1; B = 1'b1; C = 1'b1;
  end

  initial #200 $finish;
endmodule
```

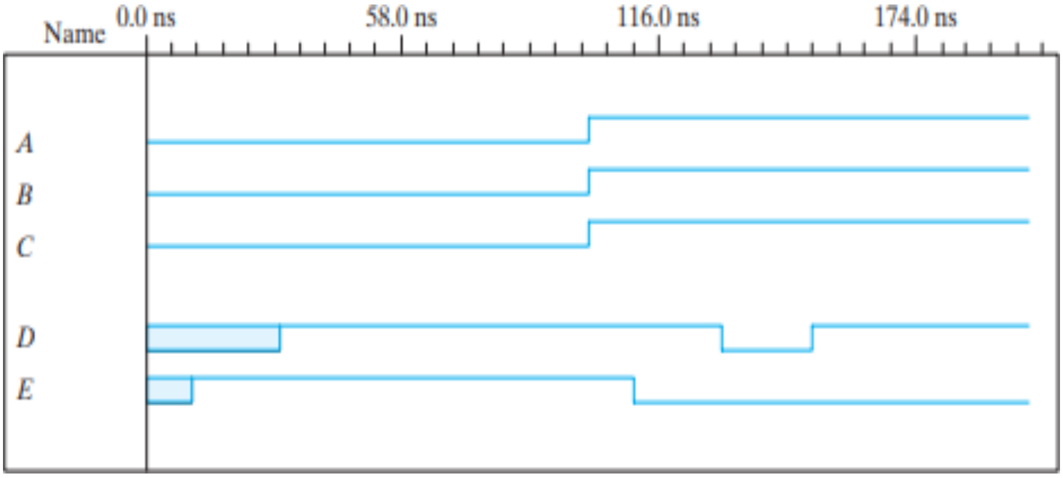
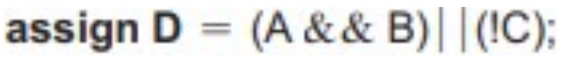


FIGURE 1.37 Simulation output of HDL Example 1.3

- The simulation runs for a total of 200 ns, and a second "initial" statement uses the "$finish" system task to specify the termination of the simulation. Delay values, such as "#100," can be used to schedule statements to execute after a specified time delay.
- The resulting timing diagram (Figure 1.37) shows the waveforms over the 200 ns interval. Initially, outputs E and D are unknown for the first 10 ns and 30 ns, respectively, as denoted by shading. Output E transitions from 1 to 0 at 110 ns, and output D transitions from 1 to 0 at 130 ns, returning to 1 at 150 ns, as predicted in Table 1.3.
- Overall, the simulation process involves abstractly modeling input signals, specifying initial conditions, and executing the simulation to verify the behavior of the HDL model over a defined time interval.

Boolean Expressions

- Boolean equations describing combinational logic are specified in Verilog with a continuous assignment statement consisting of the keyword assign followed by a Boolean expression.
- To distinguish arithmetic operators from logical operators, Verilog uses the symbols (&), (/), and (&) for AND, OR, and NOT (complement), respectively.
- Thus, to describe the simple circuit of Fig. 1.36 with a Boolean expression, we use the statement

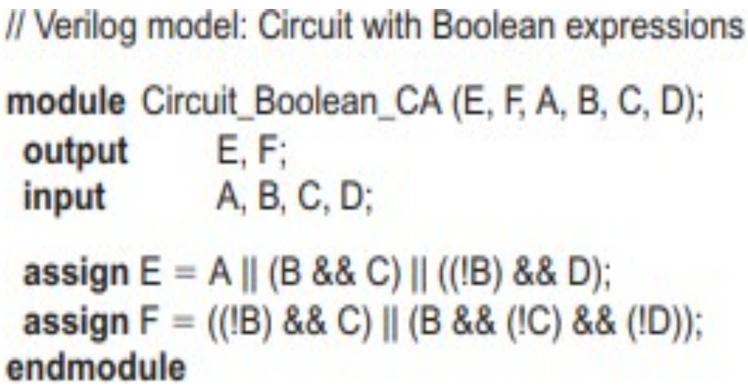


```
assign D = (A && B)||(!C);
```

- HDL Example p1.4 describes a circuit that is specified with the following two Boolean expressions:

$$E = A + BC + B'D$$
$$F = B'C + BC'D'$$

- The equations specify how the logic values E and F are determined by the values of A, B, C, and D .
  - **HDL Example p1.4 (Combinational Logic Modeled with Boolean Equations)**



```
// Verilog model: Circuit with Boolean expressions

module Circuit_Boolean_CA (E, F, A, B, C, D);
 output      E, F;
 input       A, B, C, D;

 assign E = A || (B && C) || ((!B) && D);
 assign F = ((!B) && C) || (B && (!C) && (!D));
endmodule
```

- The circuit has two outputs E and F and four inputs A, B, C, and D . The two assign statements describe the Boolean equations. The values of E and F during simulation are determined dynamically by the values of A , B , C , and D .
- The simulator detects when the test bench changes a value of one or more of the inputs. When this happens, the simulator updates the values of E and F .
- The continuous assignment mechanism is so named because the relationship between the assigned value and the variables is permanent.
- The mechanism acts just like combinational logic, has a gate-level equivalent circuit, and is referred to as implicit combinational logic .

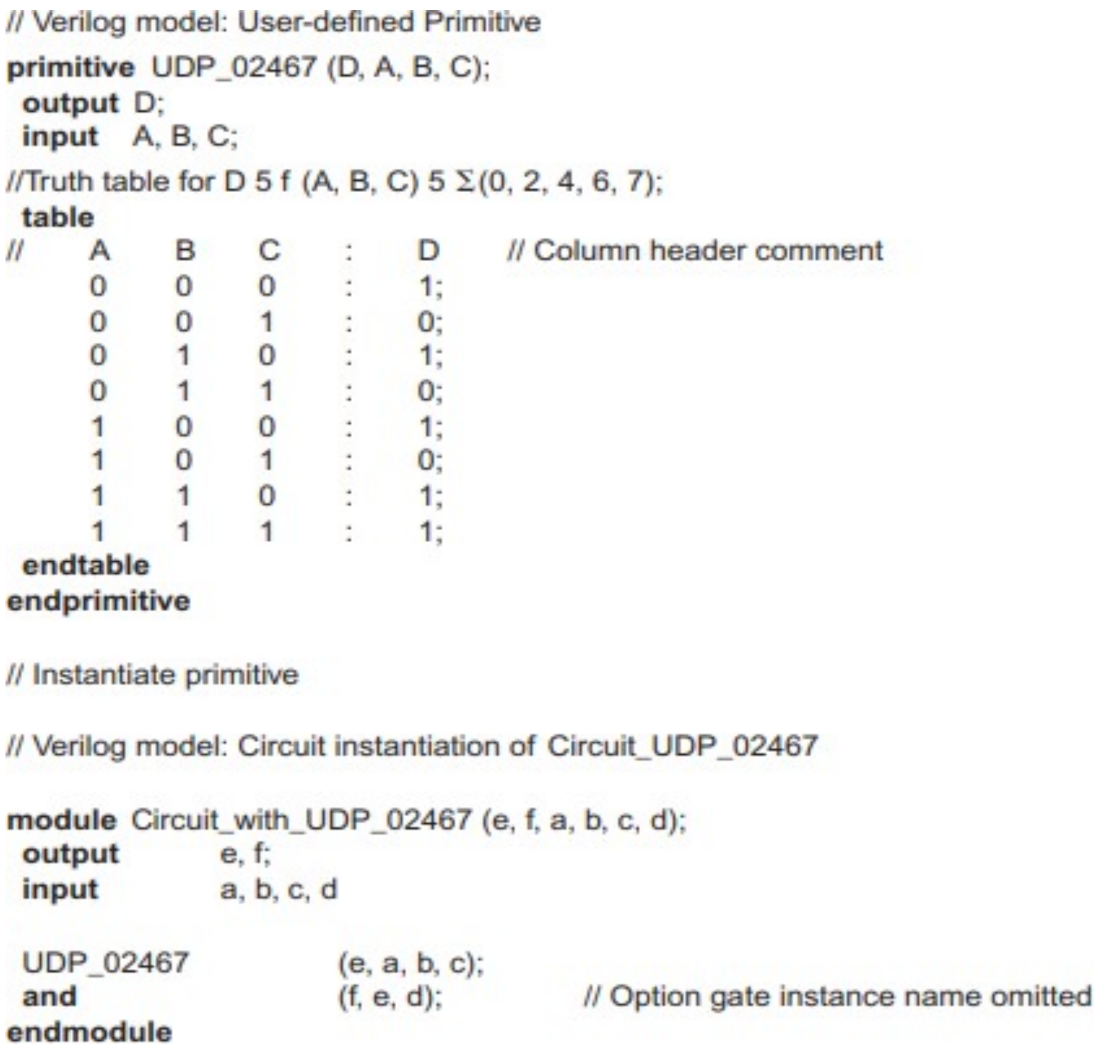
User-Defined Primitives

Explain User Defined Primitive with an HDL Example.

- The logic gates used in Verilog descriptions with keywords and, or, etc., are defined by the system and are referred to as system primitives.The user can create additional primitives by defining them in tabular form. These types of circuits are referred to as user-defined primitives (UDPs).
- One way of specifying a digital circuit in tabular form is by means of a truth table. UDP descriptions do not use the keyword pair module . . . endmodule.
- Instead, they are declared with the keyword pair primitive . . . endprimitive. The best way to demonstrate a UDP declaration is by means of an example.

- HDL Example p1.5 defines a UDP with a truth table. It proceeds according to the following general rules:
  a. It is declared with the keyword primitive , followed by a name and port list.
  b. There can be only one output, and it must be listed first in the port list and declared with keyword output .
  c. There can be any number of inputs. The order in which they are listed in the input declaration must conform to the order in which they are given values in the table that follows.
  d. The truth table is enclosed within the keywords table and endtable.
  e. The values of the inputs are listed in order, ending with a colon (:). The output is always the last entry in a row and is followed by a semicolon (;).
  f. The declaration of a UDP ends with the keyword endprimitive.

HDL Example p1.5 (User-Defined Primitive)



```
// Verilog model: User-defined Primitive
primitive  UDP_02467 (D, A, B, C);
  output  D;
  input    A, B, C;
//Truth table for D 5 f (A, B, C) 5 Σ(0, 2, 4, 6, 7);
  table
//       A      B      C      :      D      // Column header comment
         0      0      0      :      1;
         0      0      1      :      0;
         0      1      0      :      1;
         0      1      1      :      0;
         1      0      0      :      1;
         1      0      1      :      0;
         1      1      0      :      1;
         1      1      1      :      1;
  endtable
endprimitive

// Instantiate primitive

// Verilog model: Circuit instantiation of Circuit_UDP_02467

module  Circuit_with_UDP_02467 (e, f, a, b, c, d);
  output          e, f;
  input           a, b, c, d

  UDP_02467               (e, a, b, c);
  and                     (f, e, d);          // Option gate instance name omitted
endmodule
```

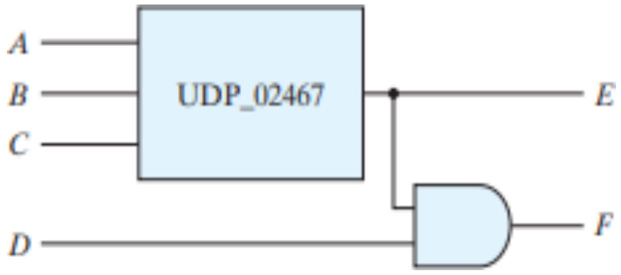


FIGURE 1.38 Schematic for Circuit with_UDP_02467

- Note that the variables listed on top of the table are part of a comment and are shown only for clarity. The system recognizes the variables by the order in which they are listed in the input declaration.
- A user-defined primitive can be instantiated in the construction of other modules (digital circuits), just as the system primitives are used. For example, the declaration Circuit _with _UDP_ 02467 (E, F, A, B, C, D); will produce a circuit that implements the hardware shown in Figure 1.38 .
- Although Verilog HDL uses this kind of description for UDPs only, other HDLs and computer-aided design (CAD) systems use other procedures to specify digital circuits in tabular form.
- The tables can be processed by CAD software to derive an efficient gate structure of the design. None of Verilog's predefined primitives describes sequential logic.
- The model of a sequential UDP requires that its output be declared as a reg data type, and that a column be added to the truth table to describe the next state. So the columns are organized as inputs : state : next state.

U	S	N										

||Jai Sri Gurudev ||

BGSKH Education Trust (R.) – A unit of Sri Adichunchanagiri Shikshana Trust(R.)

BGS College of Engineering and Technology

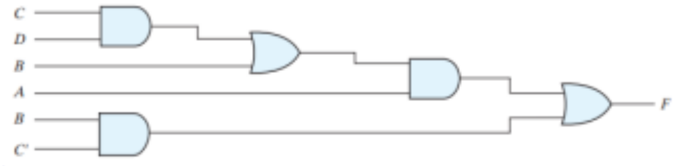
Mahalakshmipuram, West of Chord Road, Bengaluru-560086

(Approved by AICTE, New Delhi and Affiliated to VTU, Belagavi)

QUESTION BANK
MODULE 1

Subject:	**Digital Design and Computer Organization**	Subject Code	**BCS302**
Faculty Name:	MG, ASN,AR,HN		

Sl. No	**QUESTIONS**	**RBTL**
1	Demonstrate the positive and negative logic using AND gate	L3
2	What is Binary logic? List out any 4 Laws of Logic.	L3
3	State and Prove the Basic theorems and Postulates of Boolean Algebra	L1
4	Simplify the Boolean function using K maps i. F(w,x,y,z) = ∑(1,3,7,11,15) , d(w,x,y,z) = d (0,2,5) ii. F(w,x,y,z) = ∑((0,1,2,4,5,6,8,9,12,13,14)	L3
5	List the truth table of the function: *a)* $F=xy+xy^{1}+y^{1}z$ *b)* $F=bc+a^{1}c^{1}$	L2
6	Find the POS expression for F(a,b,c,d) = Π(2,3,5,8,10,13,14) + d(1,6,7,11) and realize it using NOR gates.	L3
7	Simplify the following Boolean function F, together with the don't-care conditions d, and then express the simplified function in sum-of-minterms form: $F(A,B,C,D)$ = ∑((0,6,8,13,14) $d(A,B,C,D)$ = ∑((2,4,10)	L2
8	Draw a NAND logic diagram that implements the complement of the following function: $F(A,B,C,D)$ = ∑((0,1,2,3,6,10,11,14	L2
9	Explain the importance of NAND and NOR logic gate implementation. Implement all the basic gates using NAND and NOR logic.	L2
10	Implement the Boolean function using Logic gates $F=xy + x^{1}y^{1} + y^{1}z$	L3
11	Simplify the following functions, and implement them with two-level NAND gate circuits: $F = xy^{1} + x^{1}y + z$	L3
12	Demonstrate the working of NAND & XOR gate.	L2
13	Explain the working of Test Bench in Verilog.	L2
14	Simplify the following Boolean functions, using four-variable K-maps and draw the NAND-NAND circuit for the simplified expression: F (w, x, y, z) = ∑(1, 4, 5, 6, 12, 14, 15) F (w, x, y, z) = ∑ (0, 2, 4, 5, 6, 7, 8, 10, 13, 15)	L3
15	Explain User Defined Primitive with an HDL Example	L2
16	Write a Verilog gate-level description of the circuit shown in Fig. 1(b)	L2

	 Fig. 1(b)	
17	Define Hardware Description Language (HDL). Explain an HDL identifiers and keywords with the help of circuit and Verilog code.	L2
18	Explain test bench with an HDL Example.	L2
19	Boolean function, F(w,x,y,z) = ∑(0,1,2,4,6,7,9,12,14) using kmap and Write the Verilog Program for realizing the minimized expression.	L3
20	Explain all Logic Gates with Truth Table	L2

MODULE 2
COMBINATIONAL AND SEQUENTIAL LOGIC

COMBINATIONAL LOGIC: Introduction, Combinational Circuits, Design Procedure, Binary Adder- Subtractor, Decoders, Encoders, Multiplexers. HDL Models of Combinational Circuits – Adder, Multiplexer, Encoder.

SEQUENTIAL Logic: Introduction, Sequential Circuits, Storage Elements: Latches, Flip-Flops.

Logic circuits for digital systems may be combinational or sequential. A combinational circuit consists of logic gates whose outputs at any time are determined from only the present combination of inputs. A combinational circuit performs an operation that can be specified logically by a set of Boolean functions.

COMBINATIONAL CIRCUITS:

What is a combinational circuit?

A combinational circuit consists of an interconnection of logic gates. Combinational logic gates react to the values of the signals at their inputs and produce the value of the output signal, transforming binary information from the given input data to a required output data.

A block diagram of a combinational circuit is shown in Fig 2.1

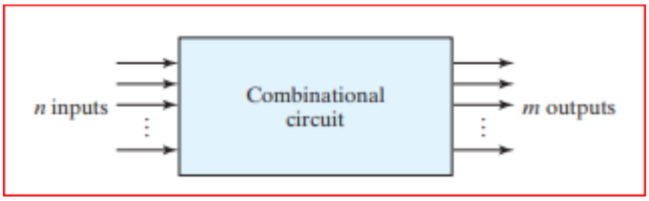


Figure 2.1: Block diagram of combinational circuit

The n input binary variables come from an external source; the m output variables are produced by the internal combinational logic circuit and go to an external destination.

.Thus, a combinational circuit can be specified with a truth table that lists the output values for each combination of input variables. A combinational circuit also can be described by m Boolean functions, one for each output variable. Each output function is expressed in terms of the n input variables.

ANALYSIS PROCEDURE

To obtain the output Boolean functions from a logic diagram, we proceed as follows:

1. Label all gate outputs that are a function of input variables with arbitrary symbols— but with meaningful names. Determine the Boolean functions for each gate output.

2. Label the gates that are a function of input variables and previously labeled gates with other arbitrary symbols. Find the Boolean functions for these gates.

3. Repeat the process outlined in step 2 until the outputs of the circuit are obtained.

4. By repeated substitution of previously defined functions, obtain the output Boolean functions in terms of input variables.

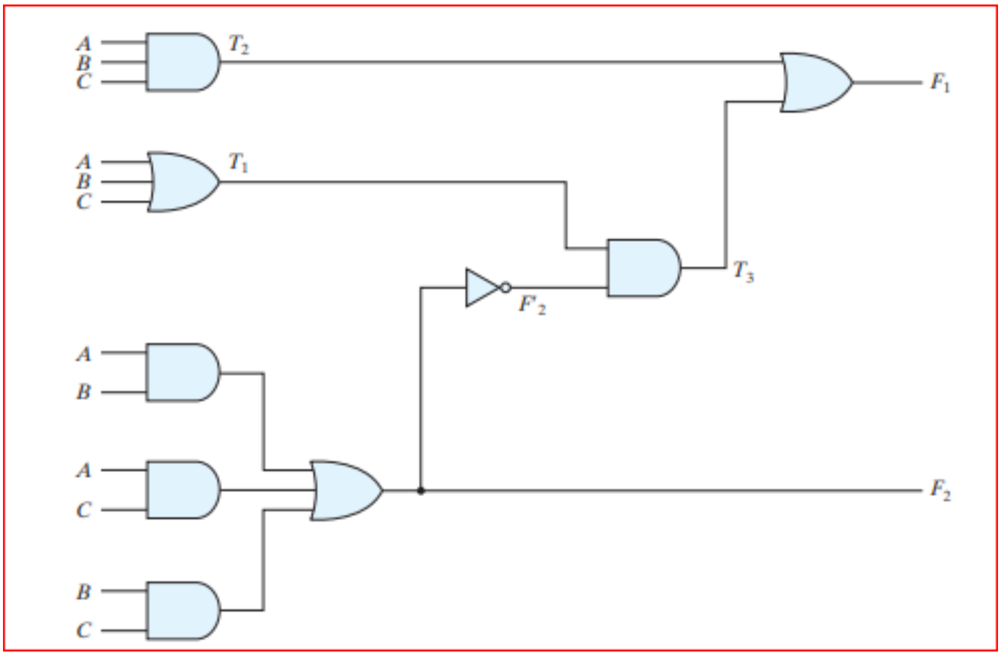


Figure 2.2: Logic diagram for analysis example

The analysis of the combinational circuit of Fig. 2.2

$$\begin{aligned} F_2 &= AB + AC + BC \\ T_1 &= A + B + C \\ T_2 &= ABC \end{aligned}$$

Next, we consider outputs of gates that are a function of already defined symbols:

$$\begin{aligned} T_3 &= F'_2T_1 \\ F_1 &= T_3 + T_2 \end{aligned}$$

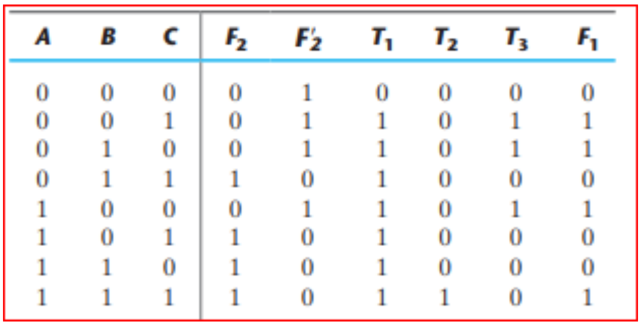
To obtain F1 as a function of A , B , and C , we form a series of substitutions as follows:

$$\begin{aligned} F_1 = T_3 + T_2 = F'_2T_1 + ABC &= (AB + AC + BC)'(A + B + C) + ABC \\ &= (A' + B')(A' + C')(B' + C')(A + B + C) + ABC \\ &= (A' + B'C')(AB' + AC' + BC' + B'C) + ABC \\ &= A'BC' + A'B'C + AB'C' + ABC \end{aligned}$$

The derivation of the truth table for a circuit is a straightforward process once the output Boolean functions are known. To obtain the truth table directly from the logic diagram without going through the derivations of the Boolean functions, we proceed as follows:

1. Determine the number of input variables in the circuit. For n inputs, form the 2^n possible input combinations and list the binary numbers from 0 to (2^n - 1) in a table.
2. Label the outputs of selected gates with arbitrary symbols.
3. Obtain the truth table for the outputs of those gates which are a function of the input variables only.
4. Proceed to obtain the truth table for the outputs of those gates which are a function of previously defined values until the columns for all outputs are determined.

Table 2.1:Truth Table for the Logic Diagram of Fig. 2.2



A	B	C	F_2	F_2'	T_1	T_2	T_3	F_1
0	0	0	0	1	0	0	0	0
0	0	1	0	1	1	0	1	1
0	1	0	0	1	1	0	1	1
0	1	1	1	0	1	0	0	0
1	0	0	0	1	1	0	1	1
1	0	1	1	0	1	0	0	0
1	1	0	1	0	1	0	0	0
1	1	1	1	0	1	1	0	1

DESIGN PROCEDURE

Design a BCD to excess 3 converters with relevant truth table and logic diagram.

The design of combinational circuits starts from the specification of the design objective and culminates in a logic circuit diagram or a set of Boolean functions from which the logic diagram can be obtained.

The procedure involves the following steps:

1. From the specifications of the circuit, determine the required number of inputs and outputs and assign a symbol to each
2. Derive the truth table that defines the required relationship between inputs and outputs.
3. Obtain the simplified Boolean functions for each output as a function of the input variables.
4. Draw the logic diagram and verify the correctness of the design truth tables and the design process of combinational circuits:

Truth Table Basics:

- A truth table for a combinational circuit contains input columns and output columns.
- Inputs are derived from the 2^n binary numbers for n input variables.
- Output values are determined based on specified conditions.

Interpreting Verbal Specifications:

- Verbal specifications are often incomplete and must be carefully interpreted to create an accurate truth table.
- Incorrect interpretations can lead to errors in the truth table.

Output Function Simplification:

- Output functions listed in the truth table are typically simplified.
- Simplification methods include algebraic manipulation, the map method, or computer-based simplification programs.
- Multiple simplified expressions may be available.

Choosing an Implementation:

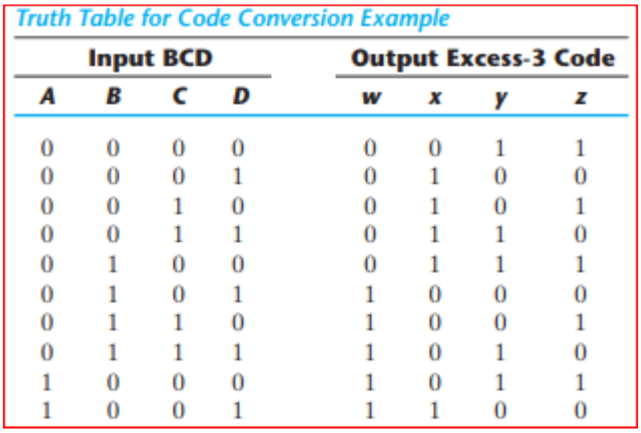
- Practical design considerations play a crucial role in choosing an implementation.
- Constraints to consider include:
  - Number of gates
  - Number of inputs to a gate
  - Propagation time of signals through gates
  - Number of interconnections
  - Limitations of the driving capability of each gate
  - Other specific performance criteria dictated by the application.

Design Process:

- The design process often starts by satisfying elementary objectives, such as achieving simplified Boolean functions in a standard form.
- Subsequent steps focus on meeting additional performance criteria.
- The importance of each constraint varies based on the specific application.

Code Conversion Example

A combinational circuit performs this transformation by means of logic gates. The design procedure will be illustrated by an example that converts binary coded decimal (BCD) to the excess-3 code for the decimal digits. The bit combinations assigned to the BCD and excess-3 codes are listed in Table 2.2



Truth Table for Code Conversion Example

Input BCD				Output Excess-3 Code			
A	*B*	*C*	*D*	*w*	*x*	*y*	*z*
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

Since each code uses four bits to represent a decimal digit, there must be four input variables and four output variables. We designate the four input binary variables by the symbols A, B, C, and D, and the four output variables by w, x, y , and z . The truth table relating the input and output variables is shown in Table 2.2

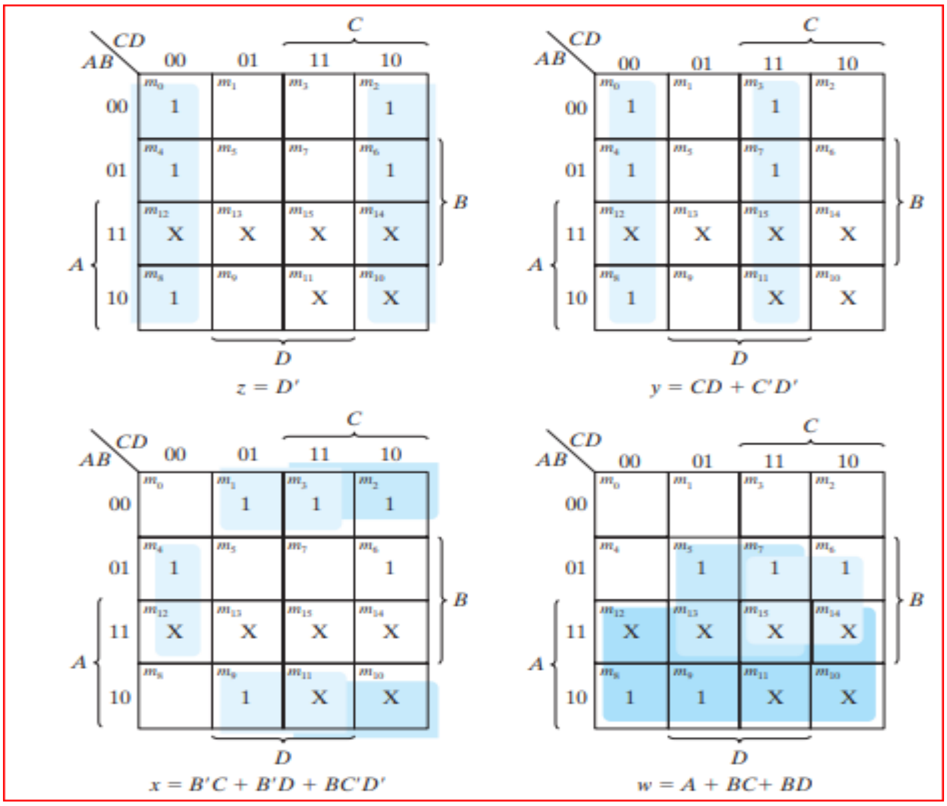


Figure 2.3: Maps for BCD-to-excess-3 code converter

implemented with three or more levels of gates:

$$
\begin{aligned}
z &= D' \\
y &= CD + C'D' = CD + (C + D)' \\
x &= B'C + B'D + BC'D' = B'(C + D) + BC'D' \\
&= B'(C + D) + B(C + D)' \\
w &= A + BC + BD = A + B(C + D)
\end{aligned}
$$

The logic diagram that implements these expressions is shown in Fig. 2.3 . Note that the OR gate whose output is C + D has been used to implement partially each of three outputs.

The implementation of Fig. 2.4 requires four AND gates, four OR gates, and one inverter. If only the normal inputs are available, the first implementation will require inverters for variables B, C , and D , and the second implementation will require inverters for variables B and D . Thus, the three-level logic circuit requires fewer gates, all of which in turn require no more than two inputs.

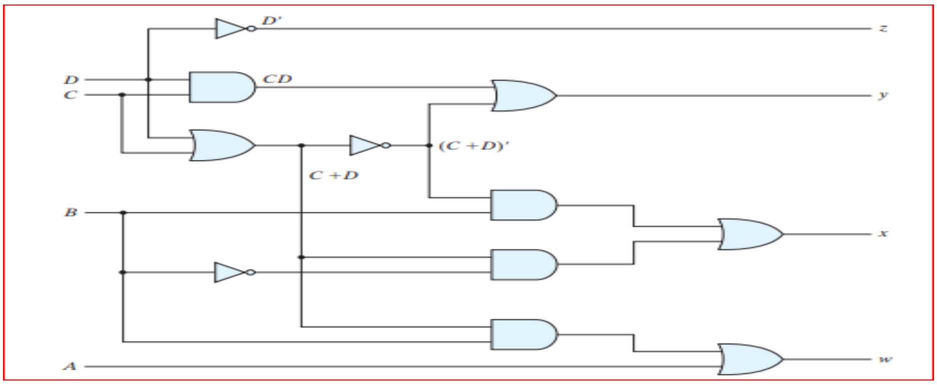


Figure 2.4: Logic diagram for BCD-to-excess-3 code converter

BINARY ADDER – SUBTRACTOR

With relevant truth table and logic diagram, explain full Adder circuit and Full Subtractor Circuit.

A combinational circuit that performs the addition of two bits is called a **half adder**. One that performs the addition of three bits (two significant bits and a previous carry) is a **full adder**.

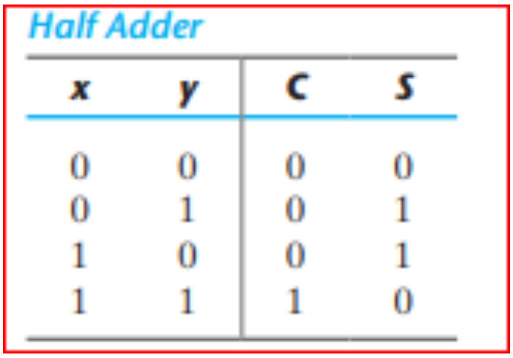
A binary adder–subtractor is a combinational circuit that performs the arithmetic operations of addition and subtraction with binary numbers.

Half Adder

$$S = x'y + xy'$$
$$C = xy$$

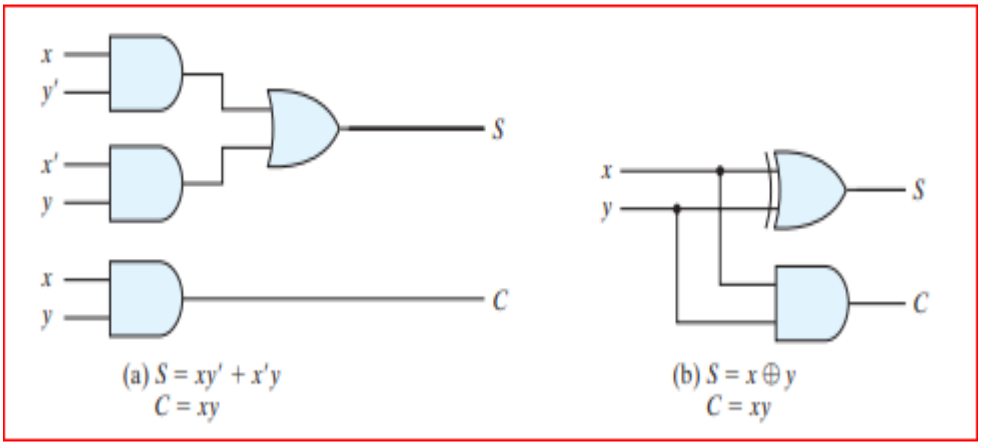
The logic diagram of the half adder implemented in sum of products is shown in Fig. 2.5(a) . It can be also implemented with an exclusive-OR and an AND gate as shown in Fig. 2.5(b) . This form is used to show that two half adders can be used to construct a full adder.

Table 2.3



Half Adder

x	y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



(a) $S = xy' + x'y$
$C = xy$

(b) $S = x \oplus y$
$C = xy$

FIGURE 2.5: Implementation of half adder

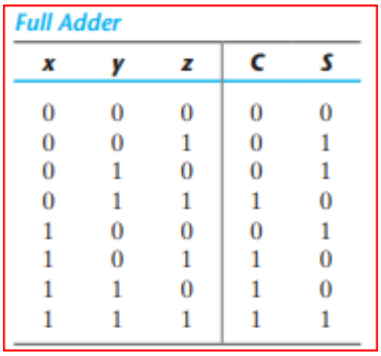
Full Adder

A full adder is a combinational circuit that forms the arithmetic sum of three bits. It consists of three inputs and two outputs. The simplified expressions are

$$S = x'y'z + x'yz' + xy'z' + xyz$$
$$C = xy + xz + yz$$

The logic diagram for the full adder implemented in sum-of-products form is shown in Fig. 2.7 . It can also be implemented with two half adders and one OR gate, as shown in Fig. 2.8

Table 2.4 Full Adder



Full Adder

x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

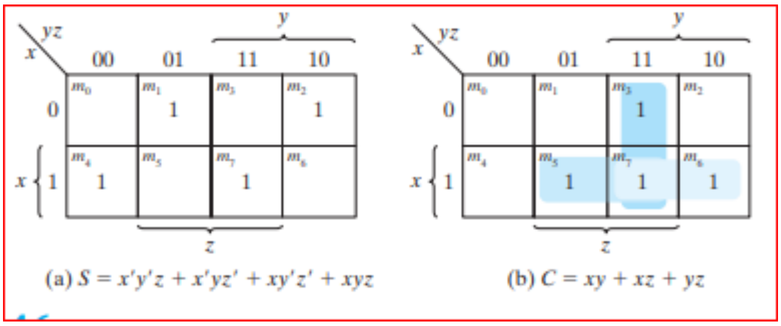


FIGURE 2.6 K-Maps for full adder

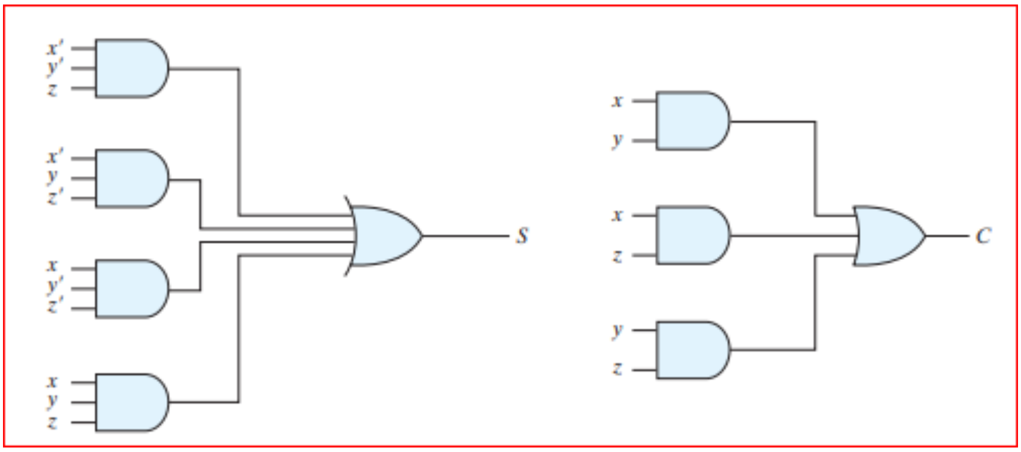
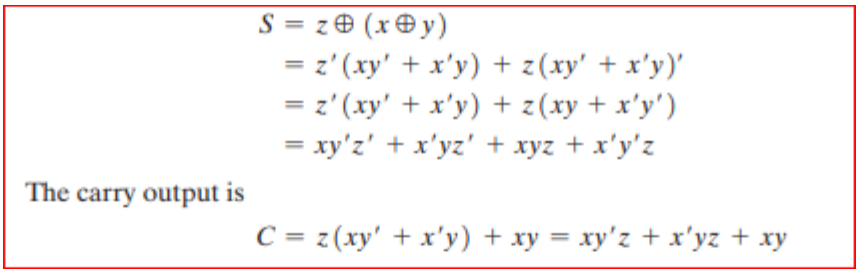


FIGURE 2.7 Implementation of full adder in sum-of-products form

The S output from the second half adder is the exclusive-OR of z and the output of the first half adder, giving



$$
\begin{aligned}
S &= z \oplus (x \oplus y) \\
&= z'(xy' + x'y) + z(xy' + x'y)' \\
&= z'(xy' + x'y) + z(xy + x'y') \\
&= xy'z' + x'yz' + xyz + x'y'z
\end{aligned}
$$

The carry output is

$$C = z(xy' + x'y) + xy = xy'z + x'yz + xy$$

Binary Adder

A binary adder is a digital circuit that produces the arithmetic sum of two binary numbers. It can be constructed with full adders connected in cascade, with the output carry from each full adder connected to the input carry of the next full adder in the chain.

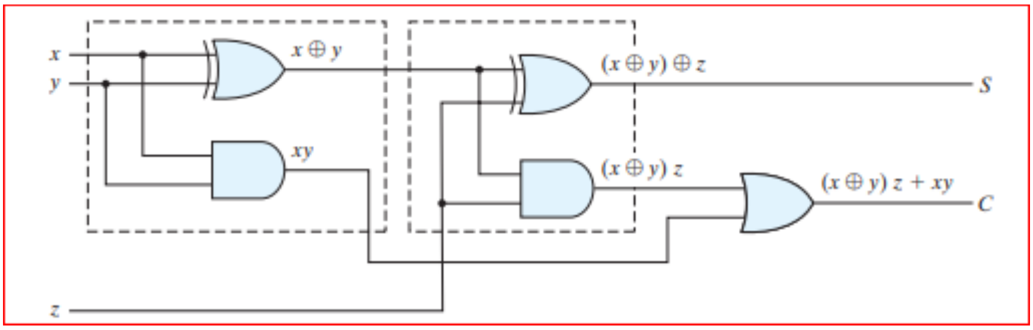
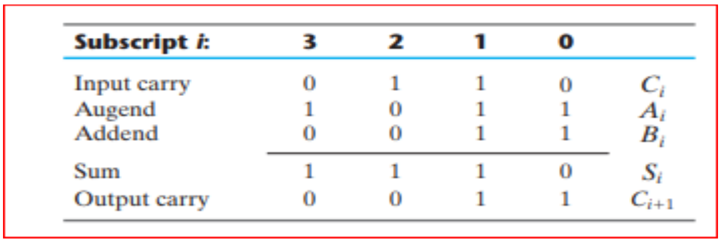


FIGURE 2.8 Implementation of full adder with two half adders and an OR gate

Addition of n-bit numbers requires a chain of n full adders or a chain of one-half adder and n-1 full adders. In the former case, the input carry to the least significant position is fixed at 0. Figure 2.9 shows the interconnection of four full-adder (FA) circuits to provide a four-bit binary ripple carry adder. The augend bits of A and the addend bits of B are designated by subscript numbers from right to left, with subscript 0 denoting the least significant bit. The carries are connected in a chain through the full adders. The input carry to the adder is C_0, and it ripples through the full adders to the output carry C_4. The S outputs generate the required sum bits. An n -bit adder requires n full adders, with each output carry connected to the input carry of the next higher order full adder. To demonstrate with a specific example, consider the two binary numbers A = 1011 and B = 0011. Their sum S = 1110 is formed with the four-bit adder as follows:



Subscript *i*:	**3**	**2**	**1**	**0**	
Input carry	0	1	1	0	C_i
Augend	1	0	1	1	A_i
Addend	0	0	1	1	B_i
Sum	1	1	1	0	S_i
Output carry	0	0	1	1	C_{i+1}

The bits are added with full adders, starting from the least significant position (subscript 0), to form the sum bit and carry bit. The input carry C_0 in the least significant position must be 0. The value of C_{i+1} in a given significant position is the output carry of the full adder. This value is transferred into the input carry of the full adder that adds the bits one higher significant position to the left. The sum bits are thus generated starting from the rightmost position and are available as soon as the corresponding previous carry bit is generated.

The four-bit adder is a typical example of a standard component. It can be used in many applications involving arithmetic operations. Observe that the design of this circuit

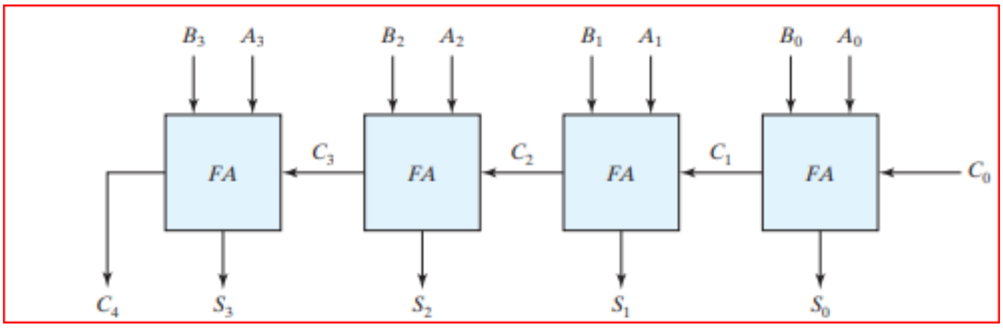


FIGURE 2.9 Four-bit adder

by the classical method would require a truth table with $2^9 = 512$ entries, since there are nine inputs to the circuit.

Carry Propagation

Design Carry Look Ahead Adder with relevant Circuit diagrams.

The addition of two binary numbers in parallel implies that all the bits of the augend and addend are available for computation at the same time. As in any combinational circuit, the signal must propagate through the gates before the correct output sum is available in the output terminals. The total propagation time is equal to the propagation delay of a typical gate, times the number of gate levels in the circuit. The longest propagation delay time in an adder is the time it takes the carry to propagate through the full adders.

Since each bit of the sum output depends on the value of the input carry, the value of S_i at any given stage in the adder will be in its steady-state final value only after the input carry to that stage has been propagated. In this regard, consider output S3 in Fig. 2.9. A_3 and B_3 are available as soon as input signals are applied to the adder. However, input carry C3 does not settle to its final value until C_2 is available from the previous stage. Similarly, C_2 has to wait for C_1 and so on down to C_0. Thus, only after the carry propagates and ripples through all stages will the last output S_3 and carry C_4 settle to their final correct value.

The number of gate levels for the carry propagation can be found from the circuit of the full adder. The circuit is redrawn with different labels in Fig. 2.10

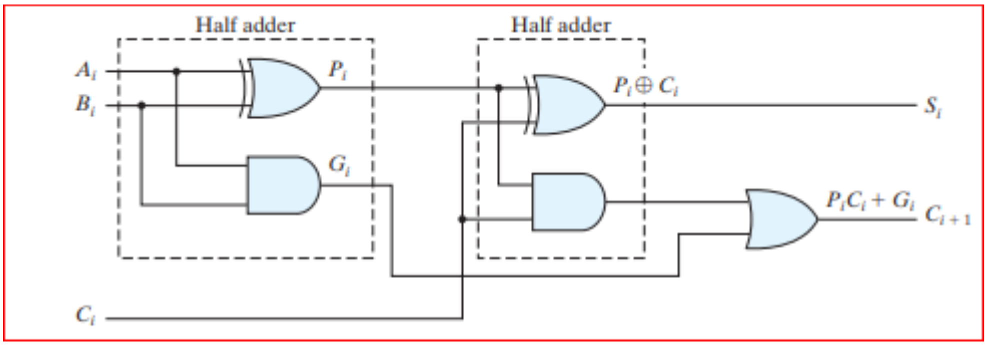
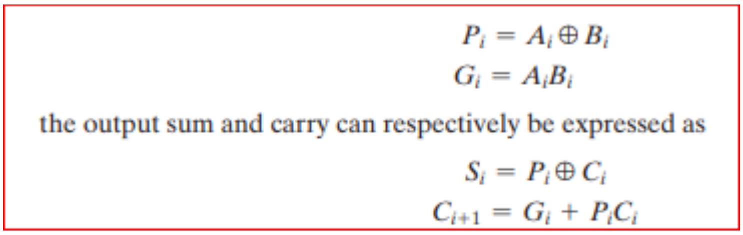


FIGURE 2.10 Full adder with P and G shown

The carry propagation time is an important attribute of the adder because it limits the speed with which two numbers are added. Although the adder—or, for that matter, any combinational circuit—will always have some value at its output terminals, the outputs will not be correct unless the signals are given enough time to propagate through the gates connected from the inputs to the outputs. Since all other arithmetic operations are implemented by successive additions, the time consumed during the addition process is critical.

Another solution is to increase the complexity of the equipment in such a way that the carry delay time is reduced. There are several techniques for reducing the carry propagation time in a parallel adder. The most widely used technique employs the principle of carry lookahead logic . Consider the circuit of the full adder shown in Fig. 2.10 . If we define two new binary variables



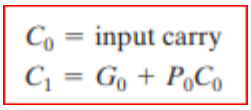
$$P_i = A_i \oplus B_i$$
$$G_i = A_i B_i$$

the output sum and carry can respectively be expressed as

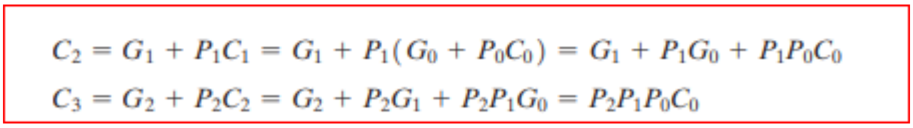
$$S_i = P_i \oplus C_i$$
$$C_{i+1} = G_i + P_i C_i$$

Gi is called a carry generate , and it produces a carry of 1 when both A_i and B_i are 1, regardless of the input carry C_i . P_i is called a carry propagate , because it determines whether a carry into stage i will propagate into stage i + 1

We now write the Boolean functions for the carry outputs of each stage and substitute the value of each Ci from the previous equations:



$$C_0 = \text{input carry}$$
$$C_1 = G_0 + P_0 C_0$$



$$C_2 = G_1 + P_1 C_1 = G_1 + P_1(G_0 + P_0 C_0) = G_1 + P_1 G_0 + P_1 P_0 C_0$$
$$C_3 = G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 = P_2 P_1 P_0 C_0$$

Since the Boolean function for each output carry is expressed in sum-of-products form, each function can be implemented with one level of AND gates followed by an OR gate.

The three Boolean functions for C_1, C_2, and C_3 are implemented in the carry lookahead generator shown in Fig. 2.11

Note that this circuit can add in less time because C_3 does not have to wait for C_2 and C_1 to propagate; in fact, C_3 is propagated at the same time as C_1 and C_2.

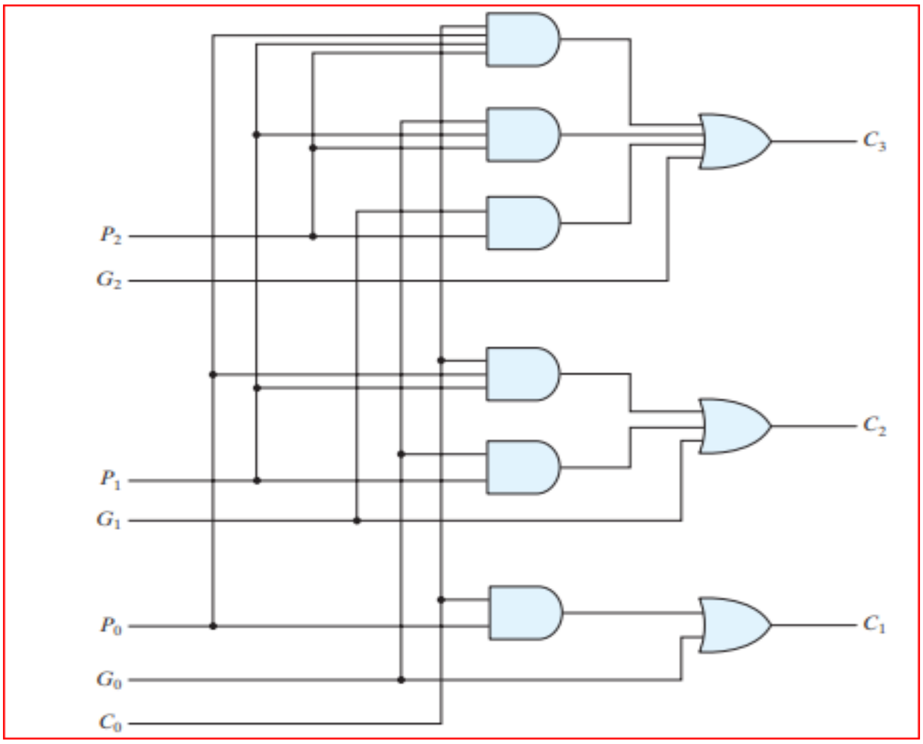


FIGURE 2.11 Logic diagram of carry lookahead generator

The construction of a four-bit adder with a carry lookahead scheme is shown in Fig. 2.12 . Each sum output requires two exclusive-OR gates. The output of the first exclusive-OR gate generates the Pi variable, and the AND gate generates the Gi variable. The carries are propagated through the carry lookahead generator (similar to that in Fig. 4.11) and applied as inputs to the second exclusive-OR gate. All output carries are generated after

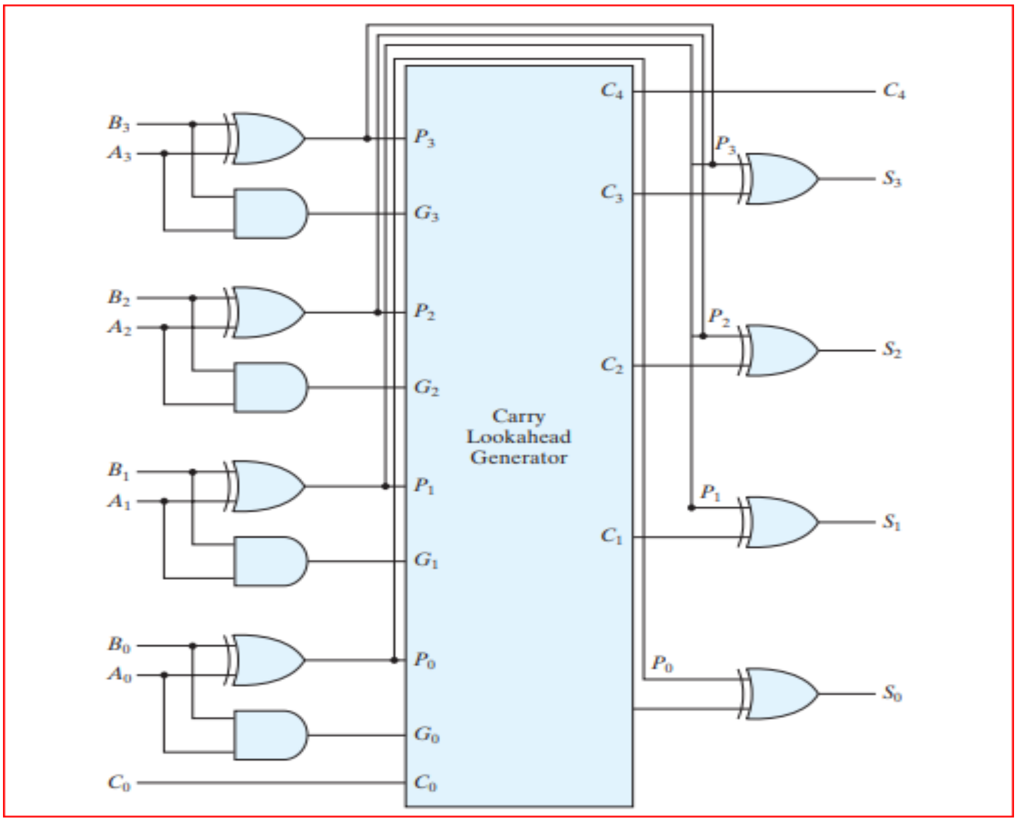


FIGURE 2.12 Four-bit adder with carry lookahead

a delay through two levels of gates. Thus, outputs S_1 through S_3 have equal propagation delay times. The two-level circuit for the output carry C_4 is not shown. This circuit can easily be derived by the equation-substitution method.

Binary Subtractor

Design Four-bit adder–subtractor Adder with relevant Circuit diagrams.

The subtraction of unsigned binary numbers can be done most conveniently by means of complements that the subtraction A - B can be done by taking the 2's complement of B and adding it to A . The 2's complement can be obtained by taking the 1's complement and adding 1 to the least significant pair of bits. The 1's complement can be implemented with inverters, and a 1 can be added to the sum through the input carry.

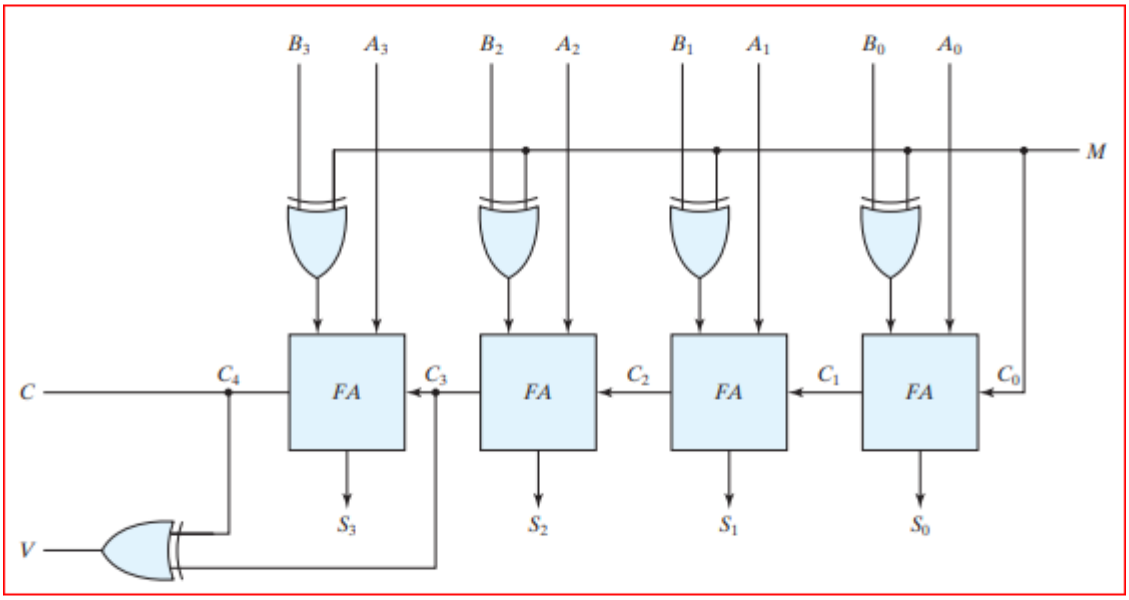


FIGURE 2.13 Four-bit adder–subtractor (with overflow detection)

The circuit for subtracting A - B consists of an adder with inverters placed between each data input B and the corresponding input of the full adder. The input carry C0 must be equal to 1 when subtraction is performed. The operation thus performed becomes A , plus the 1's complement of B , plus 1. This is equal to A plus the 2's complement of B.

For unsigned numbers, that gives A - B if A >= B or the 2's complement of 1(B – A) if A <B. For signed numbers, the result is A - B, provided that there is no overflow

The addition and subtraction operations can be combined into one circuit with one common binary adder by including an exclusive-OR gate with each full adder. A four-bit adder–subtractor circuit is shown in Fig. 2.13. The mode input M controls the operation. When M = 0, the circuit is an adder, and when M = 1, the circuit becomes a subtractor. Each exclusive-OR gate receives input M and one of the inputs of B . When M = 0, we have $B \oplus 0 = B$.

The full adders receive the value of B , the input carry is 0, and the circuit performs A plus B . When M = 1, we have B⊕ 1= B’ and C_0 = 1. The B inputs are all complemented and a 1 is added through the input carry. The circuit performs the operation A plus the 2’s complement of B.

Overflow

- Overflow occurs when two numbers with 'n' digits each are added, and the sum results in a number with 'n + 1' digits.
- This concept applies to both binary and decimal numbers, whether they are signed or unsigned.
- In manual calculations (paper and pencil), overflow is not a problem because there are no space limitations to write down the result.
- In digital computers, overflow is a concern because the number of bits used to represent a number is finite, and a result with 'n + 1' bits cannot fit into an 'n'-bit word.
- To handle overflow, many computers detect its occurrence and set a corresponding flip-flop that can be checked by the user.

Overflow Detection for Unsigned Binary Numbers

- When adding two unsigned binary numbers, overflow is detected by examining the end carry-out from the most significant position (leftmost bit).
- If there is a carry-out from the leftmost bit, it indicates an overflow.

Overflow Detection for Signed Binary Numbers

In the case of signed binary numbers:

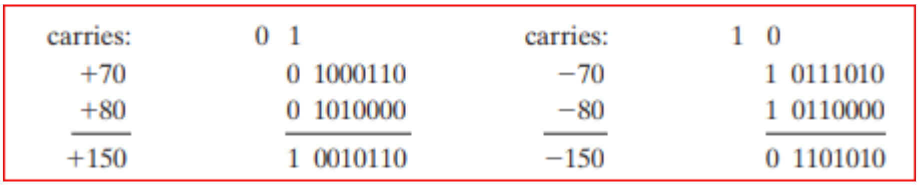
- The leftmost bit (most significant bit) represents the sign, where '0' typically denotes positive and '1' denotes negative.
- Negative numbers are typically represented in 2's complement form.

When adding two signed binary numbers:

- The sign bit is treated as part of the number and not as a sign indicator during the addition.
- The end carry does not indicate an overflow because it is part of the signed number's representation.

An overflow cannot occur after an addition if one number is positive and the other is negative, since adding a positive number to a negative number produces a result whose magnitude is smaller than the larger of the two original numbers. An overflow may occur if the two numbers added are both positive or both negative.

Example: Two signed binary numbers, +70 and +80, are stored in two eight-bit registers. The range of numbers that each register can accommodate is from binary +127 to binary -128. Since the sum of the two numbers is +150, it exceeds the capacity of an eight-bit register. This is also true for -70 and -80. The two additions in binary are shown next, together with the last two carries:



carries:	0 1	carries:	1 0
+70	0 1000110	−70	1 0111010
+80	0 1010000	−80	1 0110000
+150	1 0010110	−150	0 1101010

Note that the eight-bit result that should have been positive has a negative sign bit (i.e., the eighth bit) and the eight-bit result that should have been negative has a positive sign bit. If, however, the carry out of the sign bit position is taken as the sign bit of the result, then the nine-bit answer so obtained will be correct. But since the answer cannot be accommodated within eight bits, we say that an overflow has occurred.

An overflow condition can be detected by observing the carry into the sign bit position and the carry out of the sign bit position.

If these two carries are not equal, an overflow has occurred. This is indicated in the examples in which the two carries are explicitly shown. If the two carries are applied to an exclusive-OR gate, an overflow is detected when the output of the gate is equal to 1.

For this method to work correctly, the 2's complement of a negative number must be computed by taking the 1's complement and adding 1. This takes care of the condition when the maximum negative number is complemented.

The binary adder–subtractor circuit with outputs C and V is shown in Fig. 2.13. If the two binary numbers are considered to be unsigned, then the C bit detects a carry after addition or a borrow after subtraction. If the numbers are considered to be signed, then the V bit detects an overflow.

If V = 0 after an addition or subtraction, then no overflow occurred and the n -bit result is correct.

If V = 1, then the result of the operation contains n + 1 bits, but only the rightmost n bits of the number fit in the space available, so an overflow has occurred. The (n + 1) th bit is the actual sign and has been shifted out of position.

DECODERS

Define Decoder? Explain 3:8 Decoder.

Discrete quantities of information are represented in digital systems by binary codes. A binary code of n bits is capable of representing up to 2n distinct elements of coded information. A decoder is a combinational circuit that converts binary information from n input lines to a maximum of 2^n unique output lines. If the n -bit coded information has unused combinations, the decoder may have fewer than 2^n outputs.

The decoders presented here are called n -to- m -line decoders, where m … 2^n . Their purpose is to generate the 2^n (or fewer) minterms of n input variables. Each combination of inputs will assert a unique output. The name decoder is also used in conjunction with other code converters, such as a BCD-to-seven-segment decoder.

Example, consider the three-to-eight-line decoder circuit of Fig. 2.14 . The three inputs are decoded into eight outputs, each representing one of the minterms of the three input variables. The three inverters provide the complement of the inputs, and each one of the eight AND gates generates one of the minterms. A particular application of this decoder is binary-to-octal conversion. The input variables represent a binary number, and the outputs represent the eight digits of a number in the octal number system. However, a three-to-eight-line decoder can be used for decoding any three-bit code to provide eight outputs, one for each element of the code.

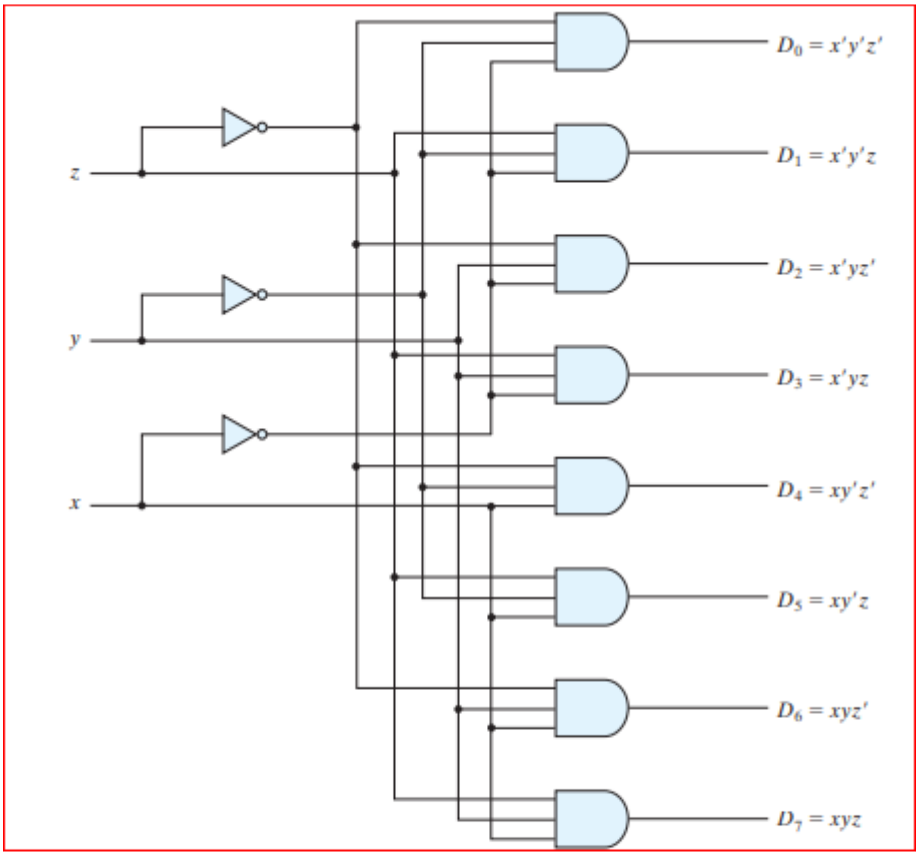
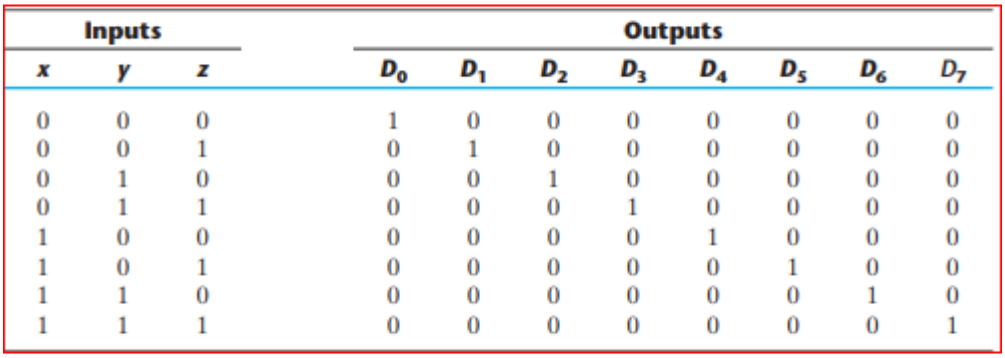


FIGURE 2.14 Three-to-eight-line decoder

Table 2.5 Truth Table of a Three-to-Eight-Line Decoder



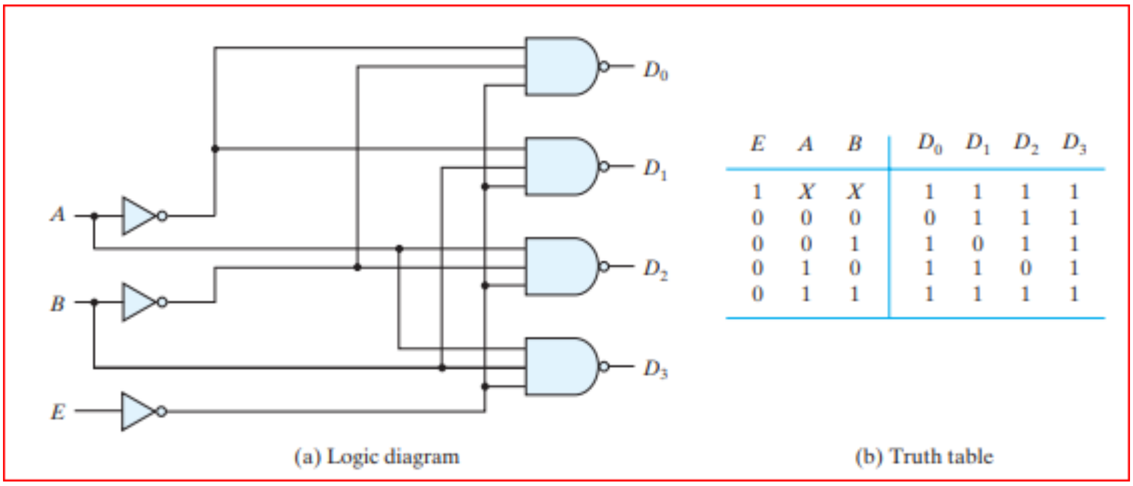
Inputs			Outputs							
x	y	z	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

The operation of the decoder may be clarified by the truth table listed in Table 2.5 . For each possible input combination, there are seven outputs that are equal to 0 and only one that is equal to 1.

The binary number now existing in the input lines is represented by the output whose value is 1, which is the minterm equivalent. NAND gates are used in the construction of some decoders. It is more cost-effective to manufacture the decoder minimum terms in their complemented form since a NAND gate generates the AND operation with an inverted output.

Furthermore, decoders include one or more enable inputs to control the circuit operation. A two-to-four-line decoder with an enable input constructed with NAND gates is shown in Fig. 2.15 .

- The circuit operates with complemented outputs and a complement enable input.
- The decoder is enabled when E is equal to 0 (i.e., active-low enable). As indicated by the truth table, only one output can be equal to 0 at any given time; all other outputs are equal to 1.
- The output whose value is equal to 0 represents the minterm selected by inputs A and B .
- The circuit is disabled when E is equal to 1, regardless of the values of the other two inputs.
- When the circuit is disabled, none of the outputs are equal to 0 and none of the minterms are selected. In general, a decoder may operate with complemented or uncomplemented outputs. The enable input may be activated with a 0 or with a 1 signal.
- Some decoders have two or more enable inputs that must satisfy a given logic condition in order to enable the circuit



(a) Logic diagram

E	A	B	D_0	D_1	D_2	D_3
1	X	X	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	1

(b) Truth table

FIGURE 2.15 Two-to-four-line decoder with enable input

A decoder with enable input can function as a demultiplexer— a circuit that receives information from a single line and directs it to one of 2^n possible output lines. The selection of a specific output is controlled by the bit combination of n selection lines.

The decoder of Fig. 2.15 can function as a one-to-four-line demultiplexer when E is taken as a data input line and A and B are taken as the selection inputs. The single input variable E has a path to all four outputs, but the input information is directed to only one of the output lines, as specified by the binary combination of the two selection lines A and B. This feature can be verified from the truth table of the circuit. For example, if the selection lines AB = 10, output D_2 will be the same as the input value E, while all other outputs are maintained at 1. Because decoder and demultiplexer operations are obtained from the same circuit, a decoder with an enable input is referred to as a decoder – demultiplexer.

Decoders with enable inputs can be connected together to form a larger decoder circuit. Figure 2.16 shows two 3-to-8-line decoders with enable inputs connected to form a 4-to-16-line decoder. When w=0, the top decoder is enabled and the other is disabled. The bottom decoder outputs are all 0's, and the top eight outputs generate minterms 0000 to 0111. When w=1, the enable conditions are reversed: The bottom decoder outputs generate minterms 1000 to 1111, while the outputs of the top decoder are all 0's. This example demonstrates the usefulness of enable inputs in decoders and other combinational logic components. In general, enable inputs are a convenient feature for interconnecting two or more standard components for the purpose of combining them into a similar function with more inputs and outputs.

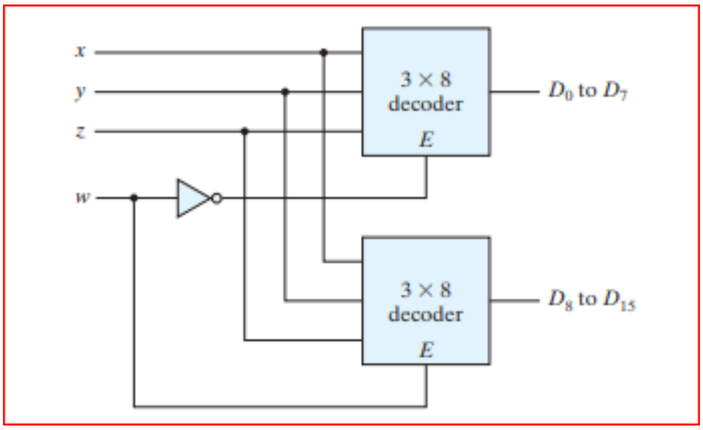


FIGURE 2.16 4x16 decoder constructed with two 3x8 decoders

Combinational Logic Implementation

A decoder provides the 2^n minterms of n input variables. Each asserted output of the decoder is associated with a unique pattern of input bits. Since any Boolean function can be expressed in sum-of-minterms form, a decoder that generates the minterms of the function, together with an external OR gate that forms their logical sum, provides a hardware implementation of the function. In this way, any combinational circuit with n inputs and m outputs can be implemented with an n -to-2^n -line decoder and m OR gates. The procedure for implementing a combinational circuit by means of a decoder and OR gates requires that the Boolean function for the circuit be expressed as a sum of minterms. A decoder is then chosen that generates all the minterms of the input variables. The inputs to each OR gate are selected from the decoder outputs according to the list of minterms of each function. This procedure will be illustrated by an example that implements a full-adder circuit. From the truth table of the full adder (see Table 2.4), we obtain the functions for the combinational circuit in sum-of-minterms form:

$$S(x, y, z) = \Sigma(1, 2, 4, 7)$$
$$C(x, y, z) = \Sigma(3, 5, 6, 7)$$

Since there are three inputs and a total of eight minterms, we need a three-to-eight-line decoder. The implementation is shown in Fig. 2.17 . The decoder generates the eight minterms for x , y , and z . The OR gate for output S forms the logical sum of minterms 1, 2, 4, and 7. The OR gate for output C forms the logical sum of minterms 3, 5, 6, and 7.

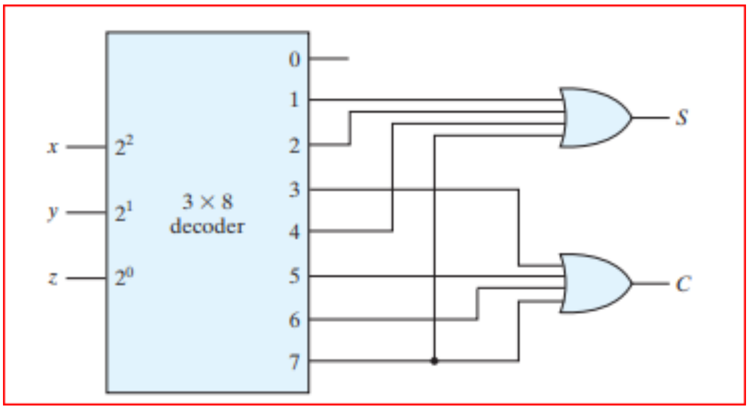


FIGURE 2.17 Implementation of a full adder with a decoder

A function with a long list of minterms requires an OR gate with a large number of inputs. A function having a list of k minterms can be expressed in its complemented form F' with 2^n - k minterms. If the number of minterms in the function is greater than $2^n/2$, then F, can be expressed with fewer minterms. The output of the NOR gate complements this sum and generates the normal output F . If NAND gates are used for the decoder, as in Fig. 2.17, then the external gates must be NAND gates instead of OR gates. This is because a two-level NAND gate circuit implements a sum-of-minterms function and is equivalent to a two-level AND–OR circuit.

ENCODERS

Define Encoder. Design a Four-input Priority Encoder.

An encoder is a digital circuit that performs the inverse operation of a decoder. An encoder has 2^n (or fewer) input lines and n output lines. The output lines, as an aggregate, generate the binary code corresponding to the input value.

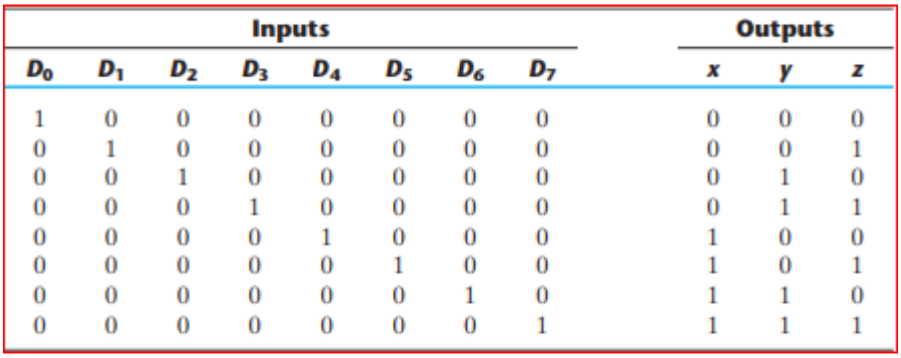
Example of an encoder is the octal-to-binary encoder whose truth table is given in Table 2.6 . It has eight inputs (one for each of the octal digits) and three outputs that generate the corresponding binary number. It is assumed that only one input has a value of 1 at any given time.

The encoder can be implemented with OR gates whose inputs are determined directly from the truth table. Output z is equal to 1 when the input octal digit is 1, 3, 5, or 7. Output y is 1 for octal digits 2, 3, 6, or 7, and output x is 1 for digits 4, 5, 6, or 7. These conditions can be expressed by the following Boolean output functions:

$$\begin{aligned} z &= D_1 + D_3 + D_5 + D_7 \\ y &= D_2 + D_3 + D_6 + D_7 \\ x &= D_4 + D_5 + D_6 + D_7 \end{aligned}$$

The encoder can be implemented with three OR gates.

Table 2.6 Truth Table of an Octal-to-Binary Encoder



Inputs								Outputs		
D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	x	y	z
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

The encoder defined in Table 2.6 has the limitation that only one input can be active at any given time. If two inputs are active simultaneously, the output produces an undefined combination. For example, if D_3 and D_6 are 1 simultaneously, the output of the encoder will be 111 because all three outputs are equal to 1. The output 111 does not represent either binary 3 or binary 6. To resolve this ambiguity, encoder circuits must establish an input priority to ensure that only one input is encoded. If we establish a higher priority for inputs with higher subscript numbers, and if both D_3 and D_6 are 1 at the same time, the output will be 110 because D6 has higher priority than D3. Another ambiguity in the octal-to-binary encoder is that an output with all 0's is generated when all the inputs are 0; but this output is the same as when D_0 is equal to 1. The discrepancy can be resolved by providing one more output to indicate whether at least one input is equal to 1.

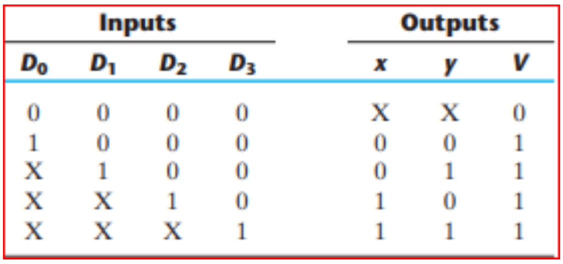
Priority Encoder

A priority encoder is an encoder circuit that includes the priority function. The operation of the priority encoder is such that if two or more inputs are equal to 1 at the same time, the input having the highest priority will take precedence.

The truth table of a four-input priority encoder is given in Table 2.7 . In addition to the two outputs x and y , the circuit has a third output designated by V ; this is a valid bit indicator that is set to 1 when one or more inputs are equal to 1. If all inputs are 0, there is no valid input and V is equal to 0. The other two outputs are not inspected when V equals 0 and are specified as don't-care conditions. Note that whereas X 's in output columns represent don't-care conditions, the X 's in the input columns are useful for representing a truth table in condensed form. Instead of listing all 16 minterms of four variables, the truth table uses an X to represent either 1 or 0. For example, X 100 represents the two minterms 0100 and 1100.

According to Table 2.7 , the higher the subscript number, the higher the priority of the input. Input D3 has the highest priority, so, regardless of the values of the other inputs, when this input is 1, the output for xy is 11 (binary 3). D_2 has the next priority level. The output is 10 if $D_2 = 1$, provided that $D_3 = 0$, regardless of the values of the other two lower priority inputs. The output for D_1 is generated only if higher priority inputs are 0, and so on down the priority levels.

Table 2.7 Truth Table of a Priority Encoder



Inputs				Outputs		
D_0	D_1	D_2	D_3	x	y	V
0	0	0	0	X	X	0
1	0	0	0	0	0	1
X	1	0	0	0	1	1
X	X	1	0	1	0	1
X	X	X	1	1	1	1

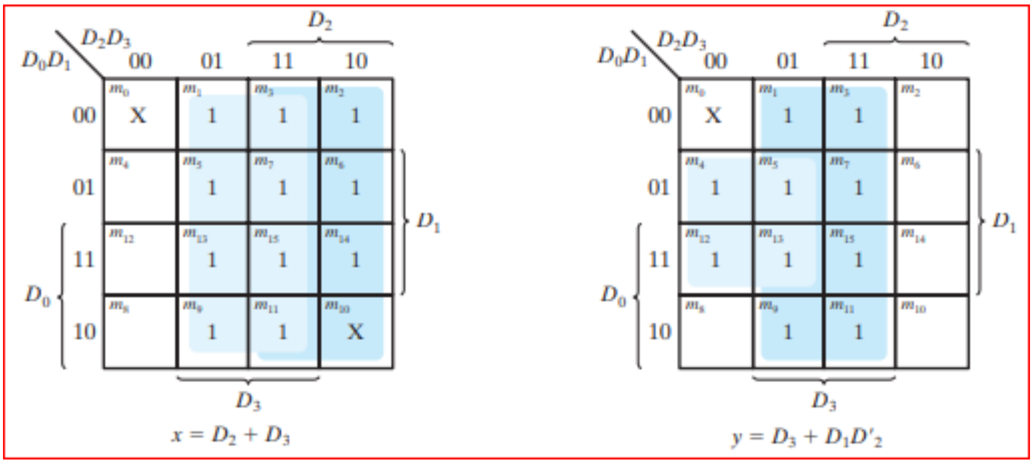


FIGURE 2.18 Maps for a priority encoder

The maps for simplifying outputs x and y are shown in Fig. 2.18 . The minterms for the two functions are derived from Table 4.8. Although the table has only five rows, when each X in a row is replaced first by 0 and then by 1, we obtain all 16 possible input combinations. For example, the fourth row in the table, with inputs XX10, represents the four minterms 0010, 0110, 1010, and 1110. The simplified Boolean expressions for the priority encoder are obtained from the maps. The condition for output V is an OR function of all the input variables. The priority encoder is implemented in Fig. 2.19 according to the following Boolean functions:

$$x = D_2 + D_3$$
$$y = D_3 + D_1 D'_2$$
$$V = D_0 + D_1 + D_2 + D_3$$

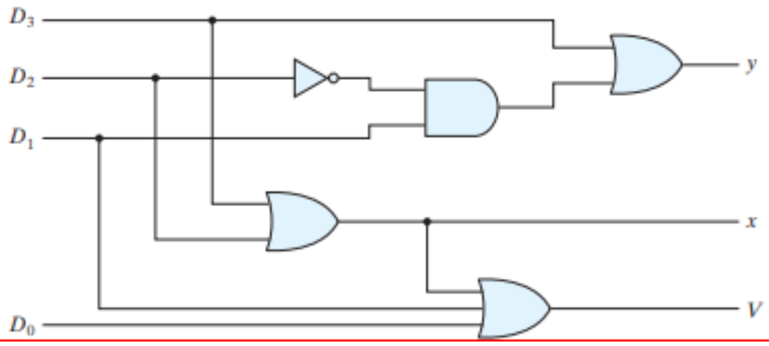


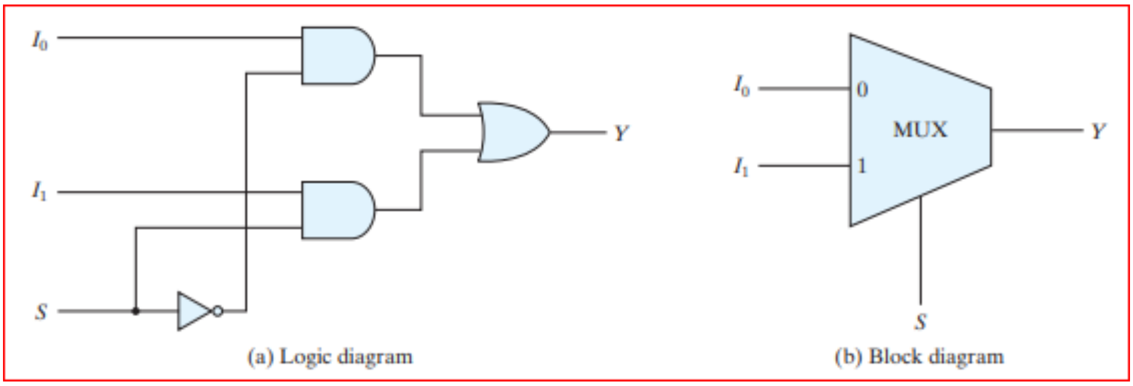
FIGURE 2.19 Four-input priority encoder

MULTIPLEXERS

What are Multiplexer? Explain 2:1 or 4:1 or 8:1 MUX.

A multiplexer is a combinational circuit that selects binary information from one of many input lines and directs it to a single output line. The selection of a particular input line is controlled by a set of selection lines. Normally, there are 2^n input lines and n selection lines whose bit combinations determine which input is selected.

A two-to-one-line multiplexer connects one of two 1-bit sources to a common destination, as shown in Fig. 2.20. The circuit has two data input lines, one output line, and one selection line S. When S = 0, the upper AND gate is enabled and I_0 has a path to the output. When S = 1, the lower AND gate is enabled and I1 has a path to the output. The multiplexer acts like an electronic switch that selects one of two sources. The block diagram of a multiplexer is sometimes depicted by a wedge-shaped symbol, as shown in Fig. 2.20(b). It suggests visually how a selected one of multiple data sources is directed into a single destination. The multiplexer is often labeled “MUX” in block diagrams.

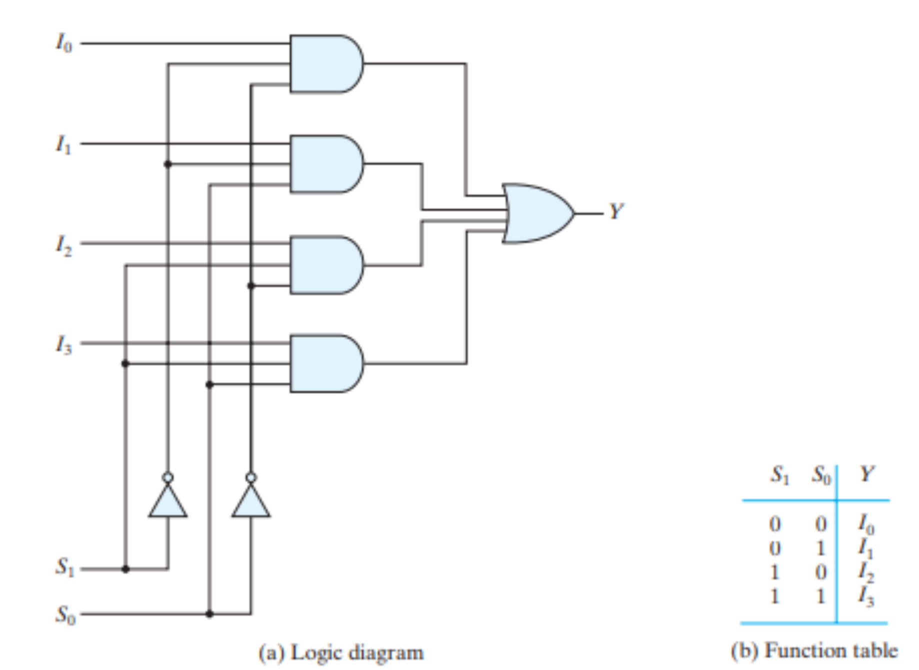


(a) Logic diagram

(b) Block diagram

FIGURE 2.20 Two-to-one-line multiplexer

A four-to-one-line multiplexer is shown in Fig. 2.21. Each of the four inputs, I_0 through I3, is applied to one input of an AND gate. Selection lines S_1 and S_0 are decoded to select a particular AND gate. The outputs of the AND gates are applied to a single OR gate that provides the one-line output. The function table lists the input that is passed to the output for each combination of the binary selection values. To demonstrate the operation of the circuit, consider the case when S_1S_0 = 10. The AND gate associated with input I_2 has two of its inputs equal to 1 and the third input connected to I_2. The other three AND gates have at least one input equal to 0, which makes their outputs equal to 0. The output of the OR gate is now equal to the value of I_2, providing a path from the selected input to the output. A multiplexer is also called a data selector, since it selects one of many inputs and steers the binary information to the output line.



(a) Logic diagram

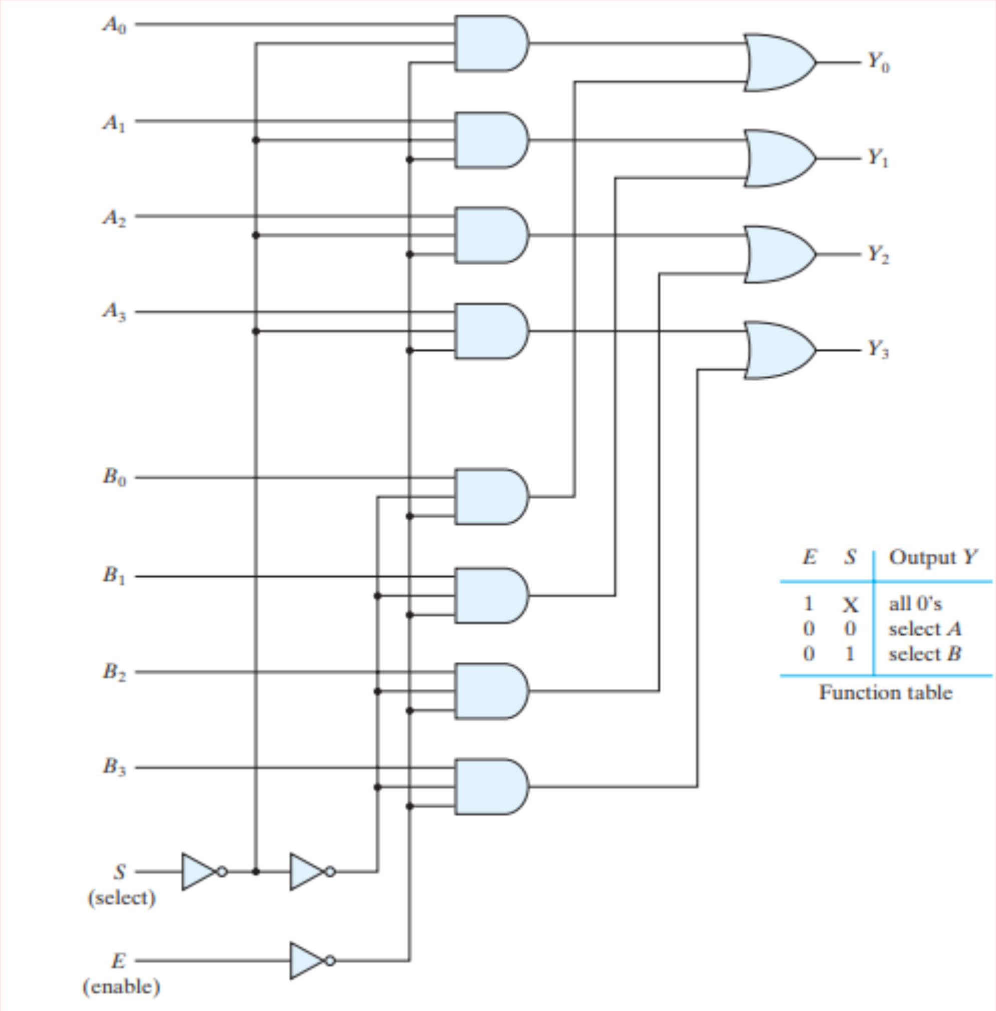
S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

(b) Function table

FIGURE 2.21 Four-to-one-line multiplexer

The AND gates and inverters in the multiplexer resemble a decoder circuit, and indeed, they decode the selection input lines. In general, a 2^n -to-1-line multiplexer is constructed from an n -to-2^n decoder by adding 2^n input lines to it, one to each AND gate. The outputs of the AND gates are applied to a single OR gate. The size of a multiplexer is specified by the number 2^n of its data input lines and the single output line. The n selection lines are implied from the 2^n data lines. As in decoders, multiplexers may have an enable input to control the operation of the unit. When the enable input is in the inactive state, the outputs are disabled, and when it is in the active state, the circuit functions as a normal multiplexer.

Multiplexer circuits can be combined with common selection inputs to provide multiple-bit selection logic. As an illustration, a quadruple 2-to-1-line multiplexer is shown in Fig. 2.22. The circuit has four multiplexers, each capable of selecting one of two input lines. Output Y_0 can be selected to come from either input A_0 or input B_0. Similarly, output Y_1 may have the value of A_1 or B_1, and so on. Input selection line S selects one of the lines in each of the four multiplexers. The enable input E must be active (i.e., asserted) for normal operation. Although the circuit contains four 2-to-1-line multiplexers, we are more likely to view it as a circuit that selects one of two 4-bit sets of data lines. As shown in the function table, the unit is enabled when E = 0. Then, if S = 0, the four A inputs have a path to the four outputs. If, by contrast, S = 1, the four B inputs are applied to the outputs. The outputs have all 0's when E = 1, regardless of the value of S.



E	S	Output Y
1	X	all 0's
0	0	select A
0	1	select B

Function table

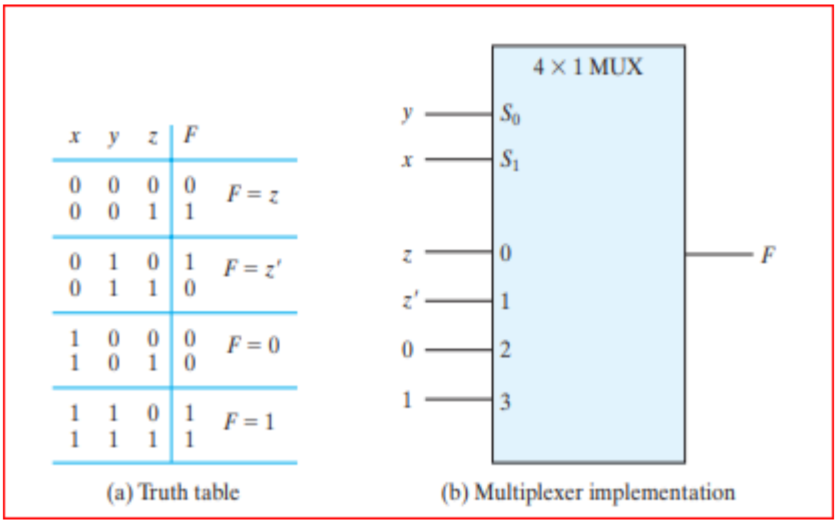
FIGURE 2.22 Quadruple two-to-one-line multiplexer

Boolean Function Implementation

An examination of the logic diagram of a multiplexer reveals that it is essentially a decoder that includes the OR gate within the unit. The minterms of a function are generated in a multiplexer by the circuit associated with the selection inputs. The individual minterms can be selected by the data inputs, thereby providing a method of implementing a Boolean function of n variables with a multiplexer that has n selection inputs and 2^n data inputs, one for each minterm.

Implementing a Boolean function of n variables with a multiplexer that has n - 1 selection inputs. The first n - 1 variables of the function are connected to the selection inputs of the multiplexer. The remaining single variable of the function is used for the data inputs. If the single variable is denoted by z , each data input of the multiplexer will be z , z', 1, or 0. To demonstrate this procedure, consider the Boolean function F (x, y, z) = ∑(1, 2, 6, 7)

This function of three variables can be implemented with a four-to-one-line multiplexer as shown in Fig. 2.23



x	y	z	F	
0	0	0	0	$F = z$
0	0	1	1	
0	1	0	1	$F = z'$
0	1	1	0	
1	0	0	0	$F = 0$
1	0	1	0	
1	1	0	1	$F = 1$
1	1	1	1	

(a) Truth table

(b) Multiplexer implementation

FIGURE 2.23 Implementing a Boolean function with a multiplexer

The two variables x and y are applied to the selection lines in that order; x is connected to the S_1 input and y to the S_0 input. The values for the data input lines are determined from the truth table of the function. When xy = 00, output F is equal to z because F = 0 when z = 0 and F = 1 when z = 1. This requires that variable z be applied to data input 0. The operation of the multiplexer is such that when xy = 00, data input 0 has a path to the output, and that makes F equal to z . In a similar fashion, we can determine the required input to data lines 1, 2, and 3 from the value of F when xy = 01, 10, and 11, respectively.

The general procedure for implementing any Boolean function of n variables with a multiplexer with n - 1 selection inputs and 2^n-1 data inputs follows from the previous example. To begin with, Boolean function is listed in a truth table. Then first n - 1 variables in the table are applied to the selection inputs of the multiplexer. For each combination of the selection variables, we evaluate the output as a function of the last variable. This function can be 0, 1, the variable, or the complement of the variable. These values are then applied to the data inputs in the proper order. As a second example, consider the implementation of the Boolean function

$$F\ (A,\ B,\ C,\ D) = \sum(1,\ 3,\ 4,\ 11,\ 12,\ 13,\ 14,\ 15)$$

This function is implemented with a multiplexer with three selection inputs as shown in Fig. 2.24 . Note that the first variable A must be connected to selection input S2 so that A , B, and C correspond to selection inputs S_2, S_1, and S_0, respectively.

A	B	C	D	F	
0	0	0	0	0	$F = D$
0	0	0	1	1	
0	0	1	0	0	$F = D$
0	0	1	1	1	
0	1	0	0	1	$F = D'$
0	1	0	1	0	
0	1	1	0	0	$F = 0$
0	1	1	1	0	
1	0	0	0	0	$F = 0$
1	0	0	1	0	
1	0	1	0	0	$F = D$
1	0	1	1	1	
1	1	0	0	1	$F = 1$
1	1	0	1	1	
1	1	1	0	1	$F = 1$
1	1	1	1	1	

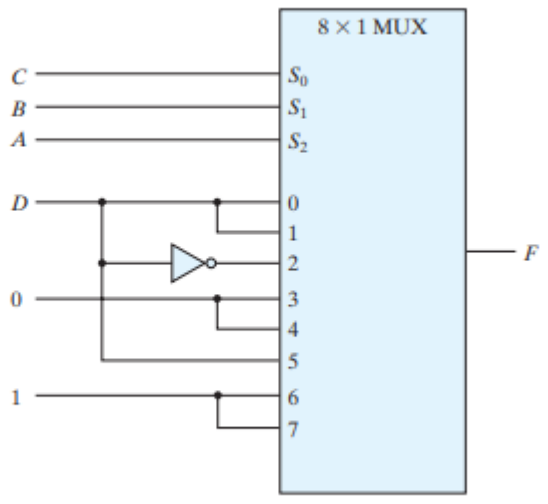


FIGURE 2.24 Implementing a four-input function with a multiplexer

data inputs are determined from the truth table listed in the figure. The corresponding data line number is determined from the binary combination of ABC. For example, the table shows that when ABC = 101, F = D, so the input variable D is applied to data input 5. The binary constants 0 and 1 correspond to two fixed signal values. When integrated circuits are used, logic 0 corresponds to signal ground and logic 1 is equivalent to the power signal, depending on the technology.

Three-State Gates

A multiplexer can be constructed with three-state gates—digital circuits that exhibit three states. Two of the states are signals equivalent to logic 1 and logic 0 as in a conventional gate. The third state is a high-impedance state in which

(1) the logic behaves like an open circuit, which means that the output appears to be disconnected,

(2) the circuit has no logic significance, and

(3) the circuit connected to the output of the three-state gate is not affected by the inputs to the gate.

Three-state gates may perform any conventional logic, such as AND or NAND. The graphic symbol for a three-state buffer gate is shown in Fig. 2.25 .

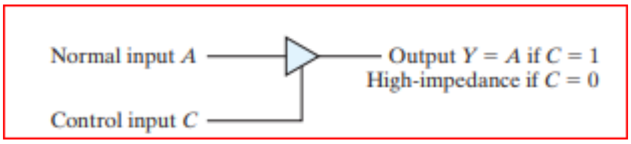


FIGURE 2.25 Graphic symbol for a three-state buffer

It is distinguished from a normal buffer by an input control line entering the bottom of the symbol. The buffer has a normal input, an output, and a control input that determines the state of the output. When the control input is equal to 1, the output is enabled and the gate behaves like a conventional buffer, with the output equal to the normal input. When the control input is 0, the output is disabled and the gate goes to a high-impedance state, regardless of the value in the normal input. The high-impedance state of a three-state gate provides a special feature not available in other gates. Because of this feature, a large number of three-state gate outputs can be connected with wires to form a common line without endangering loading effects.

The construction of multiplexers with three-state buffers is demonstrated in Fig. 2.26 . Figure 2.26(a) shows the construction of a two-to-one-line multiplexer with 2 three-state buffers and an inverter. The two outputs are connected together to form a single output line. (Note that this type of connection cannot be made with gates that do not have three-state outputs.) When the select input is 0, the upper buffer is enabled by its control input and the lower buffer is disabled. Output Y is then equal to input A . When the select input is 1, the lower buffer is enabled and Y is equal to B . The construction of a four-to-one-line multiplexer is shown in Fig. 2.26 (b) . The outputs of 4 three-state buffers are connected together to form a single output line. The control inputs to the buffers determine which one of the four normal inputs I0 through

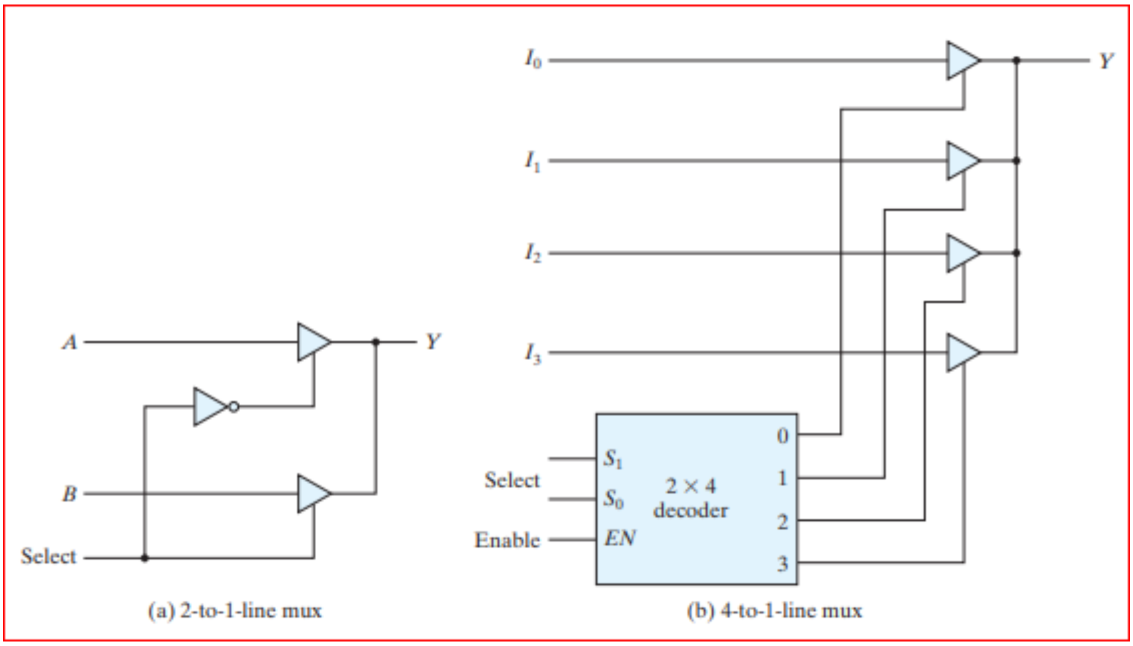


FIGURE 2.26 Multiplexers with three-state gates

I_3 will be connected to the output line. No more than one buffer may be in the active state at any given time. The connected buffers must be controlled so that only 1 three state buffer has access to the output while all other buffers are maintained in a high impedance state. One way to ensure that no more than one control input is active at any given time is to use a decoder, as shown in the diagram. When the enable input of the decoder is 0, all of its four outputs are 0 and the bus line is in a high-impedance state because all four buffers are disabled. When the enable input is active, one of the three state buffers will be active, depending on the binary value in the select inputs of the decoder. Careful investigation reveals that this circuit is another way of constructing a four-to-one-line multiplexer.

HDL MODELS OF COMBINATIONAL CIRCUITS

The logic of a module can be described in any one (or a combination) of the following modeling styles:

- Gate-level modeling using instantiations of predefined and user-defined primitive gates.
- Dataflow modeling using continuous assignment statements with the keyword assign.
- Behavioral modeling using procedural assignment statements with the keyword always.

Gate-level (structural) modeling describes a circuit by specifying its gates and how they are connected with each other.

Dataflow modeling is used mostly for describing the Boolean equations of combinational logic.

Behavioral modeling that is used to describe combinational and sequential circuits at a higher level of abstraction used to describe combinational and sequential circuits at a higher level of abstraction. Combinational logic can be designed with truth tables, Boolean equations, and schematics; Verilog has a construct corresponding to each of these "classical" approaches to design: user-defined primitives, continuous assignments, and primitives, as shown in Fig. 2.27. There is one other modeling style, called switch-level modeling. It is sometimes used in the simulation of MOS transistor circuit models, but not in logic synthesis.

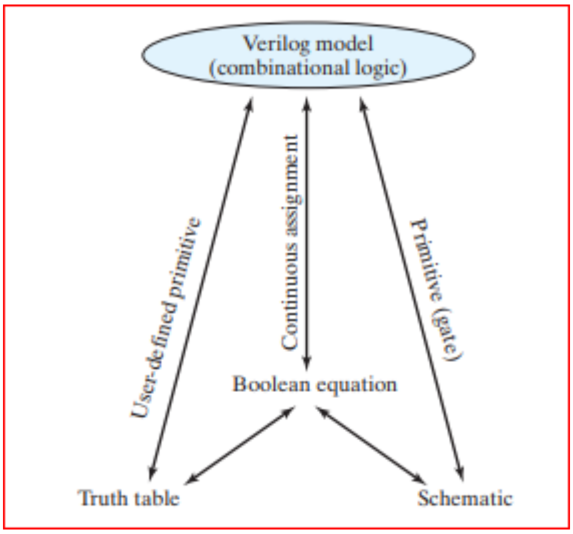
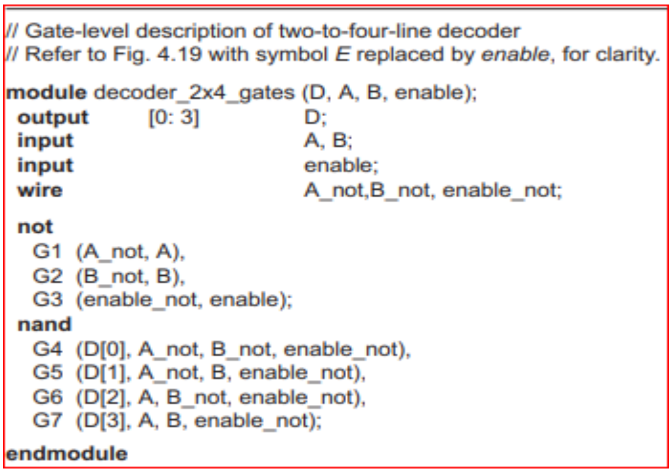


FIGURE 2.27 Relationship of Verilog constructs to truth tables, Boolean equations, and schematics

Gate-Level Modeling

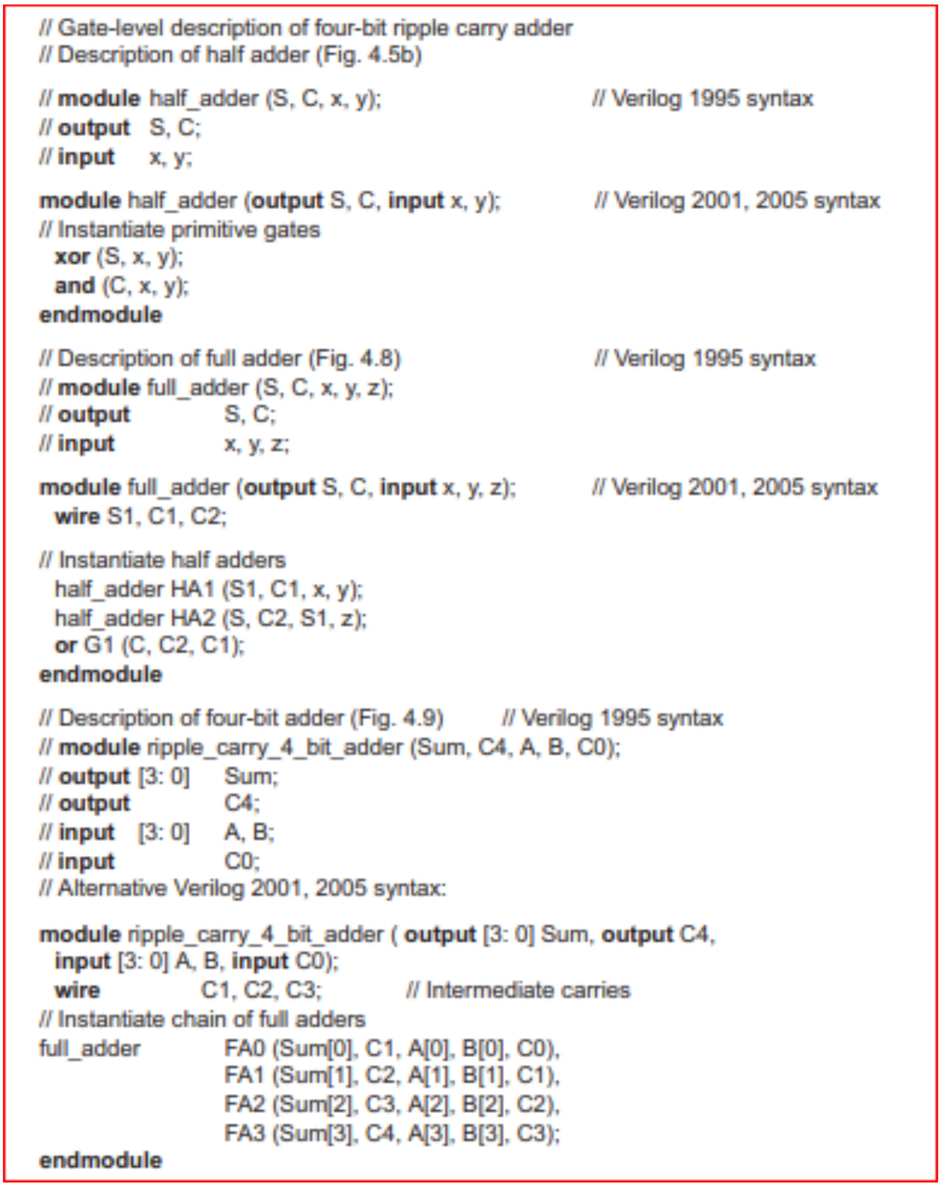
In this type of representation, a circuit is specified by its logic gates and their interconnections. Gatelevel modeling provides a textual description of a schematic diagram. The Verilog HDL includes 12 basic gates as predefined primitives. Four of these primitive gates are of the three-state type. They are all declared with the lowercase keywords and, nand, or, nor, xor, xnor, not, and buf . Primitives such as and are n -input primitives. They can have any number of scalar inputs (e.g., a three-input and primitive). The buf and not primitives are n -output primitives. A single input can drive multiple output lines distinguished by their identifiers.

HDL Example 2.1 (Two-to-Four-Line Decoder)



```
// Gate-level description of two-to-four-line decoder
// Refer to Fig. 4.19 with symbol E replaced by enable, for clarity.

module decoder_2x4_gates (D, A, B, enable);
 output      [0: 3]         D;
 input                      A, B;
 input                      enable;
 wire                       A_not,B_not, enable_not;

 not
   G1  (A_not, A),
   G2  (B_not, B),
   G3  (enable_not, enable);
 nand
   G4  (D[0], A_not, B_not, enable_not),
   G5  (D[1], A_not, B, enable_not),
   G6  (D[2], A, B_not, enable_not),
   G7  (D[3], A, B, enable_not);
endmodule
```

HDL Example 2.2 (Ripple-Carry Adder)



```
// Gate-level description of four-bit ripple carry adder
// Description of half adder (Fig. 4.5b)

// module half_adder (S, C, x, y);                    // Verilog 1995 syntax
// output  S, C;
// input    x, y;

module half_adder (output S, C, input x, y);          // Verilog 2001, 2005 syntax
// Instantiate primitive gates
  xor (S, x, y);
  and (C, x, y);
endmodule

// Description of full adder (Fig. 4.8)               // Verilog 1995 syntax
// module full_adder (S, C, x, y, z);
// output           S, C;
// input            x, y, z;

module full_adder (output S, C, input x, y, z);       // Verilog 2001, 2005 syntax
  wire S1, C1, C2;

// Instantiate half adders
  half_adder HA1 (S1, C1, x, y);
  half_adder HA2 (S, C2, S1, z);
  or G1 (C, C2, C1);
endmodule

// Description of four-bit adder (Fig. 4.9)       // Verilog 1995 syntax
// module ripple_carry_4_bit_adder (Sum, C4, A, B, C0);
// output [3: 0]    Sum;
// output           C4;
// input   [3: 0]   A, B;
// input            C0;
// Alternative Verilog 2001, 2005 syntax:

module ripple_carry_4_bit_adder ( output [3: 0] Sum, output C4,
  input [3: 0] A, B, input C0);
  wire            C1, C2, C3;          // Intermediate carries
// Instantiate chain of full adders
full_adder        FA0 (Sum[0], C1, A[0], B[0], C0),
                  FA1 (Sum[1], C2, A[1], B[1], C1),
                  FA2 (Sum[2], C3, A[2], B[2], C2),
                  FA3 (Sum[3], C4, A[3], B[3], C3);
endmodule
```

Note that modules can be instantiated (nested) within other modules, but module declarations cannot be nested; that is, a module definition (declaration) cannot be placed within another module declaration.

Three-State Gates

a three-state gate has a control input that can place the gate into a high-impedance state. The high-impedance state is symbolized by z in Verilog. There are four types of three-state gates, as shown in Fig. 2.28 . The **bufif1** gate behaves like a normal buffer if control = 1. The output goes to a high-impedance state z when control = 0. The **bufif0** gate behaves in a similar fashion, except that the high-impedance state occurs when control = 1. The two **notif** gates operate in a similar manner, except that the output is the complement of the input when the gate is not in a high-impedance state. The gates are instantiated with the statement

gate name 1output,input, control2;

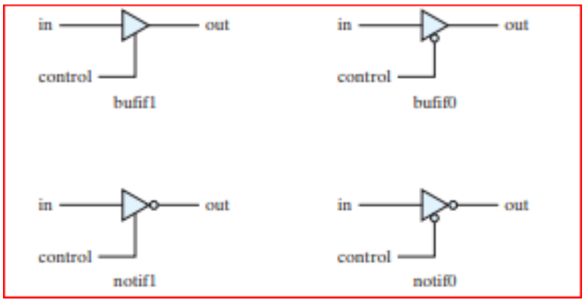


FIGURE 2.28 Three-state gates

The gate name can be that of any 1 of the 4 three-state gates. In simulation, the output can result in 0, 1, x , or z . Two examples of gate instantiation are

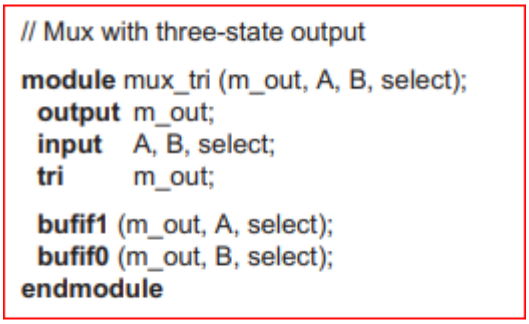
bufif1 (OUT, A, control);

notif0 (Y, B, enable);

In the first example, input A is transferred to OUT when control = 1. OUT goes to z when control = 0. In the second example, output Y = z when enable = 1 and output Y = B’ when enable = 0.

The outputs of three-state gates can be connected together to form a common output line. To identify such a connection, Verilog HDL uses the keyword **tri** (for tristate) to indicate that the output has multiple drivers. As an example, consider the two-to-one line multiplexer with three-state gates shown in Fig. 2.29 .

The HDL description must use a tri data type for the output:



```
// Mux with three-state output

module mux_tri (m_out, A, B, select);
  output  m_out;
  input    A, B, select;
  tri        m_out;

  bufif1 (m_out, A, select);
  bufif0 (m_out, B, select);
endmodule
```

The 2 three-state buffers have the same output. In order to show that they have a common connection, it is necessary to declare m_out with the keyword tri.

Keywords wire and tri are examples of a set of data types called nets , which represent connections between hardware elements. In simulation, their value is determined by a continuous assignment statement or by the device whose output they represent. The word net is not a keyword, but represents a class of data types, such as **wire , wor, wand, tri, supply1**, and **supply0**. The wire declaration is used most frequently. In fact, if an identifier is used, but not declared, the language specifies that it will be interpreted (by default) as a **wire** . The net **wor** models the hardware implementation of the wired-OR configuration (emitter-coupled logic). The nets **supply1** and **supply0** represent power supply and ground, respectively. They are used to hardwire an input of a device to either 1 or 0.

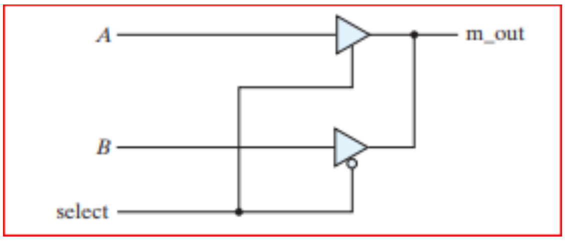


FIGURE 2.29 Two-to-one-line multiplexer with three-state buffers

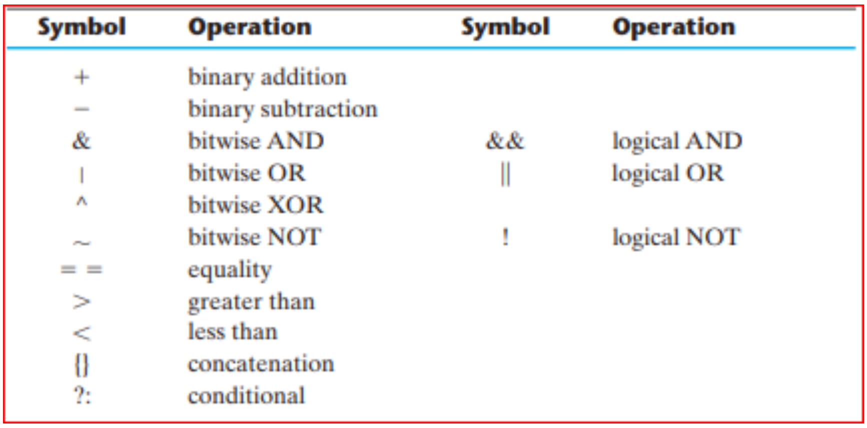
Dataflow Modeling

Dataflow modeling of combinational logic uses a number of operators that act on binary operands to produce a binary result. Verilog HDL provides about 30 different operators.

Table 2.9 lists some of these operators, their symbols, and the operation that they perform.

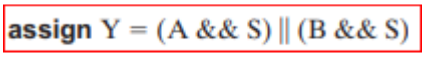
It is necessary to distinguish between arithmetic and logic operations, so different symbols are used for each. The plus symbol (+) indicates the arithmetic operation of addition; the bitwise logic AND operation (conjunction) uses the symbol &. There are special symbols for bitwise logical OR (disjunction), NOT, and XOR. The equality symbol uses two equals signs (without spaces between them) to distinguish it from the equals sign used with the assign statement. The bitwise operators operate bit by bit on a pair of vector operands to produce a vector result. The concatenation operator provides a mechanism for appending multiple operands. For example, two operands with two bits each can be concatenated to form an operand with four bits.

Table 2.9 Some Verilog HDL Operators



Symbol	Operation	Symbol	Operation
+	binary addition		
−	binary subtraction		
&	bitwise AND	&&	logical AND
|	bitwise OR	||	logical OR
^	bitwise XOR		
~	bitwise NOT	!	logical NOT
= =	equality		
>	greater than		
<	less than		
{}	concatenation		
?:	conditional		

Dataflow modeling uses continuous assignments and the keyword assign. A continuous assignment is a statement that assigns a value to a net. The data type family net is used in Verilog HDL to represent a physical connection between circuit elements. A net is declared explicitly by a net keyword (e.g., wire) or by declaring an identifier to be an input port. The logic value associated with a net is determined by what the net is connected to. If the net is connected to an output of a gate, the net is said to be driven by the gate, and the logic value of the net is determined by the logic values of the inputs to the gate and the truth table of the gate. If the identifier of a net is the left-hand side of a continuous assignment statement or a procedural assignment statement, the value assigned to the net is specified by a Boolean expression that uses operands and operators. As an example, assuming that the variables were declared, a two-to-one-line multiplexer with scalar data inputs A and B , select input S , and output Y is described with the continuous assignment

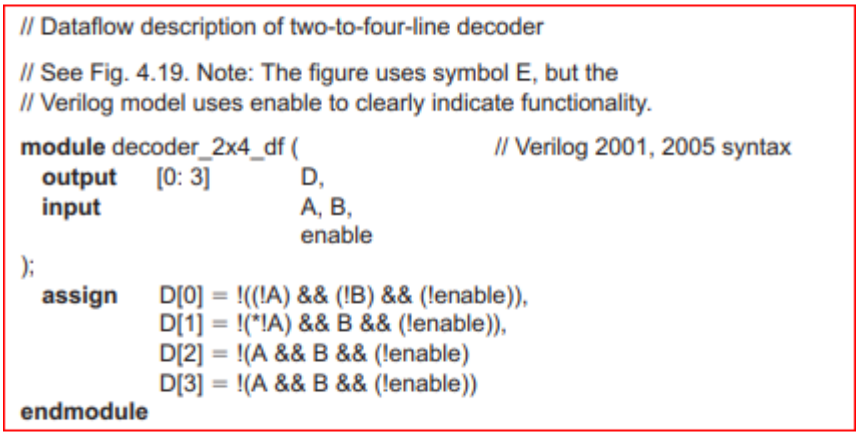


```
assign Y = (A && S) || (B && S)
```

The relationship between Y , A , B , and S is declared by the keyword assign , followed by the target output Y and an equals sign. Following the equals sign is a Boolean expression. In hardware terms, this assignment would be equivalent to connecting the output of the OR gate to wire Y.

The next two examples show the dataflow models of the two previous gate-level examples. The dataflow description of a two-to-four-line decoder with active-low output enable and inverted output is shown in HDL Example 2.3. The circuit is defined with four continuous assignment statements using Boolean expressions, one for each output.

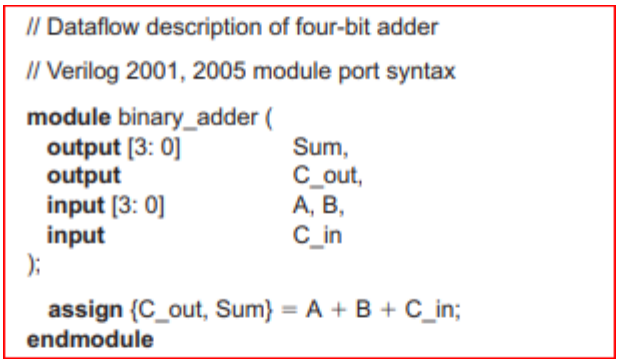
HDL Example 2.3 (Dataflow: Two-to-Four Line Decoder)



```
// Dataflow description of two-to-four-line decoder

// See Fig. 4.19. Note: The figure uses symbol E, but the
// Verilog model uses enable to clearly indicate functionality.

module decoder_2x4_df (                        // Verilog 2001, 2005 syntax
  output     [0: 3]          D,
  input                      A, B,
                             enable
);
  assign     D[0] = !((!A) && (!B) && (!enable)),
             D[1] = !(*!A) && B && (!enable)),
             D[2] = !(A && B && (!enable)
             D[3] = !(A && B && (!enable))
endmodule
```

The dataflow description of the four-bit adder is shown in HDL Example 2.4. The addition logic is described by a single statement using the operators of addition and concatenation. The plus symbol (+) specifies the binary addition of the four bits of A with the four bits of B and the one bit of C _ in . The target output is the concatenation of the output carry C _ out and the four bits of Sum . Concatenation of operands is expressed within braces and a comma separating the operands. Thus, {C_out, Sum} represents the five-bit result of the addition operation.

HDL Example 2.4 (Dataflow: Four-Bit Adder)

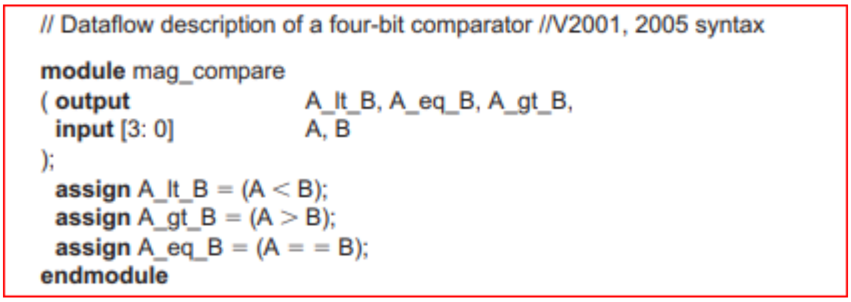


```
// Dataflow description of four-bit adder

// Verilog 2001, 2005 module port syntax

module binary_adder (
  output [3: 0]          Sum,
  output                 C_out,
  input [3: 0]           A, B,
  input                  C_in
);

  assign {C_out, Sum} = A + B + C_in;
endmodule
```

Dataflow HDL models describe combinational circuits by their function rather than by their gate structure.

To show how dataflow descriptions facilitate digital design, consider the 4-bit magnitude comparator described in HDL Example 2.5. The module specifies two 4-bit inputs A and B and three outputs. One output (A_lt_B) is logic 1 if A is less than B, a second output (A_gt_B) is logic 1 if A is greater than B, and a third output (A_eq_B) is logic 1 if A is equal to B. Note that equality (identity) is symbolized with two equals signs (= =) to distinguish the operation from that of the assignment operator (=).

HDL Example 2.5 (Dataflow: Four-Bit Comparator)

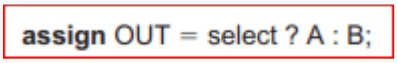


```
// Dataflow description of a four-bit comparator //V2001, 2005 syntax

module mag_compare
( output                    A_lt_B, A_eq_B, A_gt_B,
  input [3: 0]              A, B
);
  assign A_lt_B = (A < B);
  assign A_gt_B = (A > B);
  assign A_eq_B = (A = = B);
endmodule
```

The next example uses the conditional operator (?:). This operator takes three operands:

condition? true-expression : false-expression;

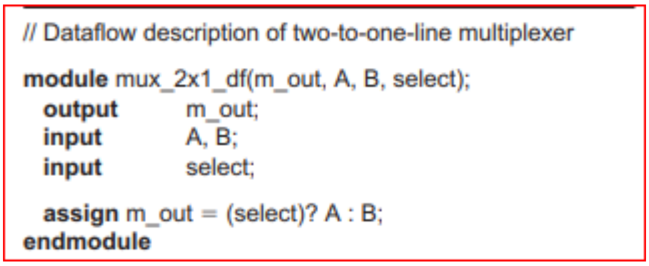
The condition is evaluated. If the result is logic 1, the true expression is evaluated and used to assign a value to the left-hand side of an assignment statement. If the result is logic 0, the false expression is evaluated. The two conditions together are equivalent to an if–else condition. HDL Example 2.6 describes a two-to-one-line multiplexer using the conditional operator. The continuous assignment.



```
assign OUT = select ? A : B;
```

specifies the condition that OUT = A if select = 1, else OUT =B if select =0.

HDL Example 2.6 (Dataflow: Two-to-One Multiplexer)



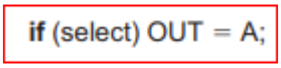
```
// Dataflow description of two-to-one-line multiplexer

module mux_2x1_df(m_out, A, B, select);
  output        m_out;
  input         A, B;
  input         select;

  assign m_out = (select)? A : B;
endmodule
```

Behavioral Modeling

Behavioral modeling represents digital circuits at a functional and algorithmic level. It is used mostly to describe sequential circuits, but can also be used to describe combinational circuits.

Behavioral descriptions use the keyword **always** , followed by an optional event control expression and a list of procedural assignment statements. The event control expression specifies when the statements will execute. The target output of a procedural assignment statement must be of the **reg** data type. Contrary to the **wire** data type, whereby the target output of an assignment may be continuously updated, a **reg** data type retains its value until a new value is assigned.

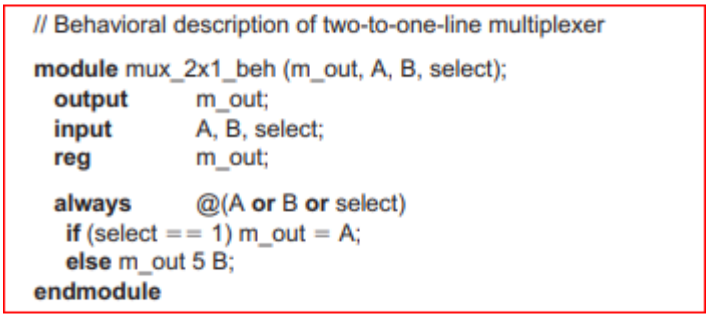
HDL Example 2.7 shows the behavioral description of a two-to-one-line multiplexer. Since variable m_out is a target output, it must be declared as reg data. The procedural assignment statements inside the always block are executed every time there is a change in any of the variables listed after the @ symbol. In this case, these variables are the input variables A, B, and select. The statements execute if A, B, or select changes value. Note that the keyword or, instead of the bitwise logical OR operator "|", is used between variables. The conditional statement if–else provides a decision based upon the value of the select input. The if statement can be written without the equality symbol:



```
if (select) OUT = A;
```

The statement implies that select is checked for logic 1.

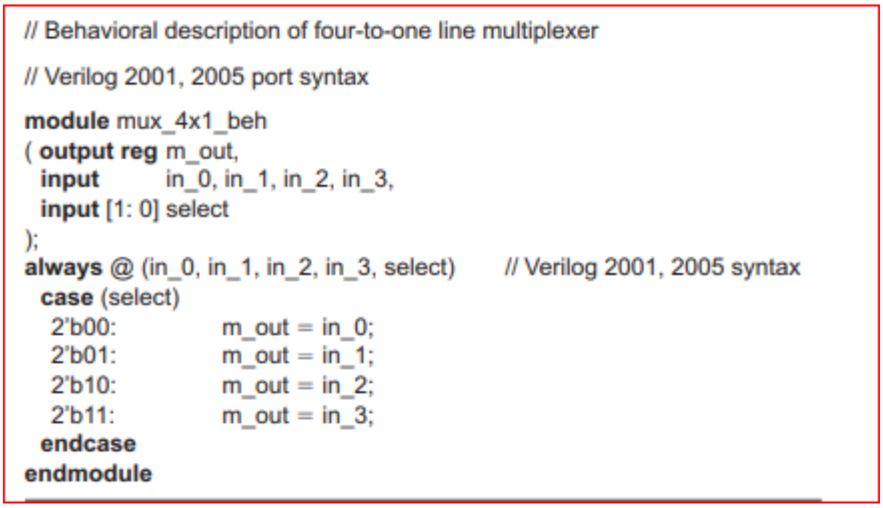
HDL Example 2.7 (Behavioral: Two-to-One Line Multiplexer)



```
// Behavioral description of two-to-one-line multiplexer

module mux_2x1_beh (m_out, A, B, select);
  output        m_out;
  input         A, B, select;
  reg           m_out;

  always        @(A or B or select)
   if (select = = 1) m_out = A;
   else m_out 5 B;
endmodule
```

HDL Example 2.8 describes the function of a four-to-one-line multiplexer. The select input is defined as a two-bit vector, and output y is declared to have type reg . The always statement, in this example, has a sequential block enclosed between the keywords case and endcase. The block is executed whenever any of the inputs listed after the @ symbol changes in value. The case statement is a multiway conditional branch construct. Whenever in_0, in_1, in_2, in_3 or select change, the case expression (select) is evaluated and its value compared, from top to bottom, with the values in the list of statements that follow, the so-called case items. The statement associated with the first case item that matches the case expression is executed. In the absence of a match, no statement is executed. Since select is a two-bit number, it can be equal to 00, 01, 10, or 11. The case items have an implied priority because the list is evaluated from top to bottom.

HDL Example 2.8 (Behavioral: Four-to-One Line Multiplexer)



```
// Behavioral description of four-to-one line multiplexer

// Verilog 2001, 2005 port syntax

module mux_4x1_beh
( output reg m_out,
  input       in_0, in_1, in_2, in_3,
  input [1: 0] select
);
always @ (in_0, in_1, in_2, in_3, select)     // Verilog 2001, 2005 syntax
  case (select)
   2'b00:        m_out = in_0;
   2'b01:        m_out = in_1;
   2'b10:        m_out = in_2;
   2'b11:        m_out = in_3;
  endcase
endmodule
```

Synchronous Sequential Logic

A block diagram of a sequential circuit is shown in Fig 2.30. It consists of a combinational circuit to which storage elements are connected to form a feedback path. The storage elements are devices capable of storing binary information. The binary information stored in these elements at any given time defines the state of the sequential circuit at that time. The sequential circuit receives binary information from external inputs that, together with the present state of the storage elements, determine the binary value of the outputs. These external inputs also determine the condition for changing the state in the storage elements. The block diagram demonstrates that the outputs in a sequential circuit are a function not only of the inputs, but also of the present state of the storage elements. The next state of the storage elements is also a function of external inputs and the present state. Thus, **a sequential circuit is specified by a time sequence of inputs, outputs, and internal states** . In contrast, the outputs of combinational logic depend only on the present values of the inputs.

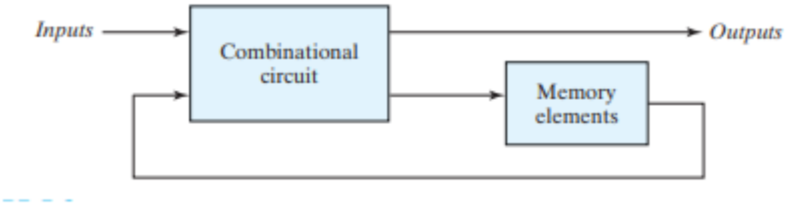
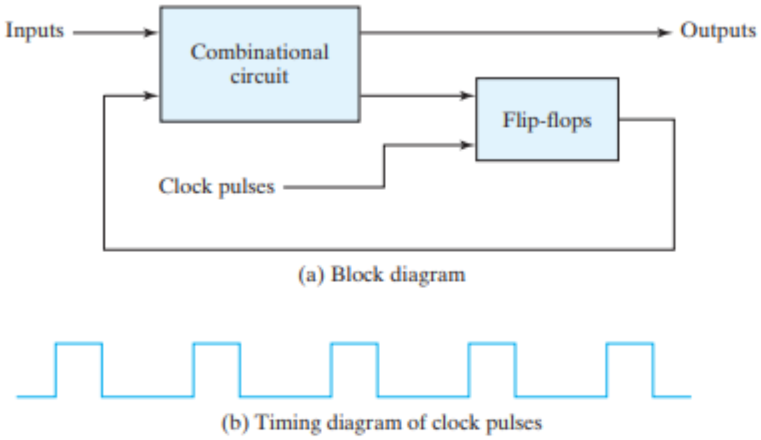


FIGURE 2.30 Block diagram of sequential circuit

The block diagram of a synchronous clocked sequential circuit is shown in Fig. 2.31 .



(a) Block diagram

(b) Timing diagram of clock pulses

FIGURE 2.31 Synchronous clocked sequential circuit

- Combinational Logic Outputs: Outputs in digital circuits are determined by combinational logic functions of inputs and flip-flop values.
- Flip-Flop State Determination: The value stored in a flip-flop during a clock pulse is influenced by inputs and current flip-flop values.
- Clock Pulse Update: Flip-flop values are updated during a clock pulse.

- Timing Criticality: Combinational logic speed is crucial to meet clock pulse intervals, ensuring correct operation.
- Propagation Delays: Delays in signal propagation impact the minimum allowable clock pulse interval.
- Clock Pulse Trigger: Flip-flop state changes occur only during clock pulse transitions (e.g., from 0 to 1).
- Feedback Loop Disconnection: During inactive clock pulses, the feedback loop between flip-flop values and inputs is disrupted.
- Deterministic State Transition: Transitions between states occur at predetermined intervals defined by clock pulses.

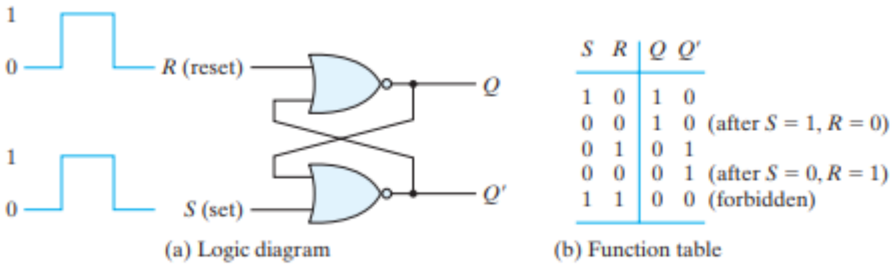
STORAGE ELEMENTS : LATCHES

- A storage element in a digital circuit can maintain a binary state indefinitely, until directed by an input signal to switch states.
- The major differences among various types of storage elements are in the number of inputs they possess and in the manner in which the inputs affect the binary state.
- Storage elements that operate with signal levels are referred to as latches; those controlled by a clock transition are flip-flops.
- Latches are said to be level sensitive devices; flip-flops are edge-sensitive devices.
- The two types of storage elements are related because latches are the basic circuits from which all flip-flops are constructed. Although latches are useful for storing binary information and for the design of asynchronous sequential circuits, they are not practical for use as storage elements in synchronous sequential circuits.

SR Latch

- The SR latch is a circuit with two cross-coupled NOR gates or two cross-coupled NAND gates, and two inputs labelled S for set and R for reset.
- The SR latch constructed with two cross-coupled NOR gates is shown in Fig. 2.32. The latch has two useful states. When output Q = 1 and Q’ = 0, the latch is said to be in the set state. When Q = 0 and Q’ = 1, it is in the reset state.
- Outputs Q and Q’ are normally the complement of each other. However, when both inputs are equal to 1 at the same time, a condition in which both outputs are equal to 0 (rather than be mutually complementary occurs. If both inputs are then switched to 0 simultaneously, the device will enter an unpredictable or undefined state or a metastable state.
- Under normal conditions, both inputs of the latch remain at 0 unless the state has to be changed. The application of a momentary 1 to the S input causes the latch to go to the set state.

- The S input must go back to 0 before any other changes take place, in order to avoid the occurrence of an undefined next state that results from the forbidden input condition. As shown in the function table of Fig. 2.32 (b) , two input conditions cause the circuit to be in the set state.



(a) Logic diagram

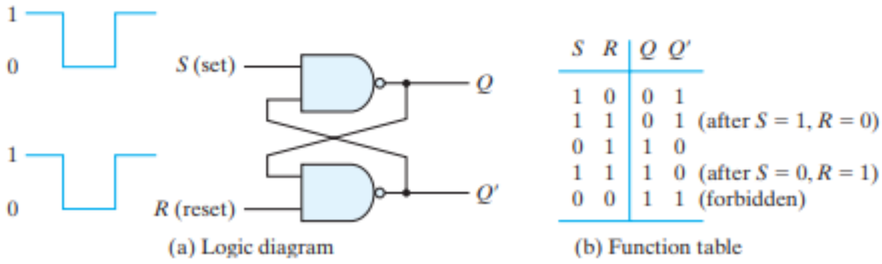
S	R	Q	Q′	
1	0	1	0	
0	0	1	0	(after $S = 1, R = 0$)
0	1	0	1	
0	0	0	1	(after $S = 0, R = 1$)
1	1	0	0	(forbidden)

(b) Function table

FIGURE 2.32 SR latch with NOR gates

- The first condition (S = 1,R = 0) is the action that must be taken by input S to bring the circuit to the set state. R
- emoving the active input from S leaves the circuit in the same state.
- After both inputs return to 0, it is then possible to shift to the reset state by momentary applying a 1 to the R input.
- The 1 can then be removed from R, where upon the circuit remains in the reset state. Thus, when both inputs S and R are equal to 0, the latch can be in either the set or the reset state, depending on which input was most recently a 1.
- If a 1 is applied to both the S and R inputs of the latch, both outputs go to 0. This action produces an undefined next state, because the state that results from the input transitions depends on the order in which they return to 0.
- It also violates the requirement that outputs be the complement of each other. In normal operation, this condition is avoided by making sure that 1's is not applied to both inputs simultaneously.

The SR latch with two cross-coupled NAND gates is shown in Fig. 2.32 . It operates with both inputs normally at 1, unless the state of the latch has to be changed. The application of 0 to the S input causes output Q to go to 1, putting the latch in the set state. When the S input goes back to 1, the circuit remains in the set state. After both inputs go back to 1, we are allowed to change the state of the latch by placing a 0 in the R input. This action causes the circuit to go to the reset state and stay there even after both inputs return to 1. The condition that is forbidden for the NAND latch is both inputs being equal to 0 at the same time, an input combination that should be avoided.



(a) Logic diagram

S	R	Q	Q′	
1	0	0	1	
1	1	0	1	(after $S = 1, R = 0$)
0	1	1	0	
1	1	1	0	(after $S = 0, R = 1$)
0	0	1	1	(forbidden)

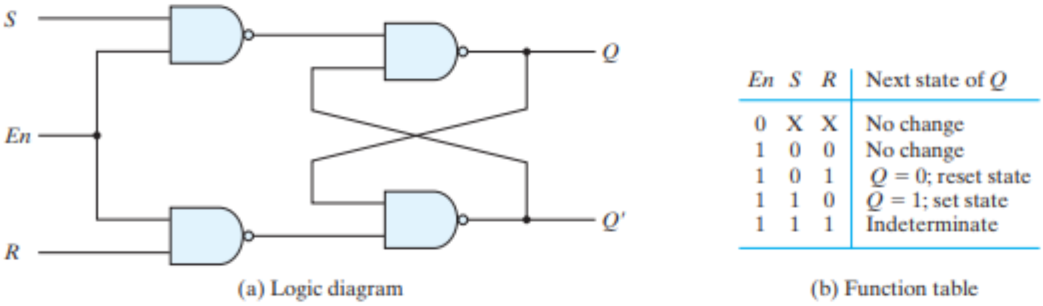
(b) Function table

FIGURE 2.32 SR latch with NAND gates

In comparing the NAND with the NOR latch, note that the input signals for the NAND require the complement of those values used for the NOR latch. Because the NAND latch requires a 0 signal to change its state, it is sometimes referred to as an S'R' latch.

The operation of the basic SR latch can be modified by providing an additional input signal that determines (controls) when the state of the latch can be changed by determining whether S and R (or S" and R") can affect the circuit. An SR latch with a control input is shown in Fig. 2.33 . It consists of the basic SR latch and two additional NAND gates. The control input En acts as an enable signal for the other two inputs. **The outputs of the NAND gates stay at the logic-1 level as long as the enable signal remains at 0.** This is the quiescent condition for the SR latch. When the enable input goes to 1, information from the S or R input is allowed to affect the latch. The set state is reached with S = 1, R = 0, and En = 1.

To change to the reset state, the inputs must be S = 0, R = 1, and En = 1. In either case, when En returns to 0, the circuit remains in its current state. The control input disables the circuit by applying 0 to En, so that the state of the output does not change regardless of the values of S and R . Moreover, when En = 1 and both the S and R inputs are equal to 0, the state of the circuit does not change. These conditions are listed in the function table accompanying the diagram.



(a) Logic diagram

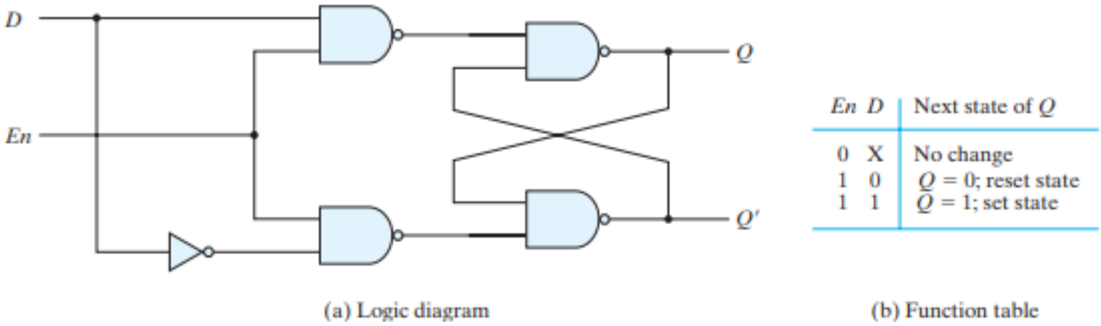
En	*S*	*R*	Next state of *Q*
0	X	X	No change
1	0	0	No change
1	0	1	*Q* = 0; reset state
1	1	0	*Q* = 1; set state
1	1	1	Indeterminate

(b) Function table

FIGURE 2.33 SR latch with control inpu

D Latch (Transparent Latch)

One way to eliminate the undesirable condition of the indeterminate state in the SR latch is to ensure that inputs S and R are never equal to 1 at the same time. This is done in the D latch, shown in Fig. 2.34 . This latch has only two inputs: D (data) and En (enable). The D input goes directly to the S input, and its complement is applied to the R input. As long as the enable input is at 0, the cross-coupled SR latch has both inputs at the 1 level and the circuit cannot change state regardless of the value of D . The D input is sampled when En = 1. If D = 1, the Q output goes to 1, placing the circuit in the set state. If D = 0, output Q goes to 0, placing the circuit in the reset state.



(a) Logic diagram

En	*D*	Next state of *Q*
0	X	No change
1	0	*Q* = 0; reset state
1	1	*Q* = 1; set state

(b) Function table

FIGURE 2.34 D latch

The graphic symbols for the various latches are shown in Fig. 2.35 . A latch is designated by a rectangular block with inputs on the left and outputs on the right. One output designates the normal output, and the other (with the bubble designation) designates the complement output. The graphic symbol for the SR latch has inputs S and R indicated inside the block.

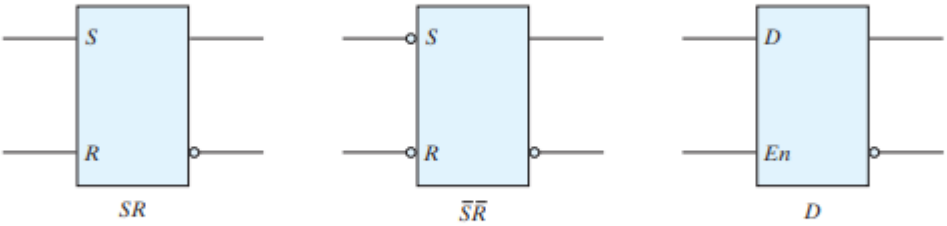


FIGURE 2.35 Graphic symbols for latches

STORAGE ELE MENTS : FLIP – FLOPS

Briefly explain

SR Flip Flop b) D Flip Flop c) T Flip Flop d) JK Flip Flop

The state of a latch or flip-flop is switched by a change in the control input. This momentary change is called a trigger, and the transition it causes is said to trigger the flip-flop. The D latch with pulses in its control input is essentially a flip-flop that is triggered every time the pulse goes to the logic-1 level. As long as the pulse input remains at this level, any changes in the data input will change the output and the state of the latch. Consequently, the inputs of the flip-flops are derived in part from the outputs of the same and other flip-flops. When latches are used for the storage elements, a serious difficulty arises. The state transitions of the latches start as soon as the clock pulse changes to the logic-1 level. The new state of a latch appears at the output while the pulse is still active. This output is connected to the inputs of the latches through the combinational circuit. If the inputs applied to the latches change while the clock pulse is still at the logic-1 level, the latches will respond to new values and a new output state may occur. The result is an unpredictable situation, since the state of the latches may keep changing for as long as the clock pulse stays at the active level.

Flip-flop Functionality: Flip-flops are essential components in sequential circuits, designed to work in synchronization with a common clock signal. Unlike latches, flip-flops are triggered only during specific signal transitions, ensuring reliable operation.

Latch Behavior: Latches respond to changes in their enable input, allowing output changes when the D input changes while the clock pulse is at logic 1, resulting in sensitivity to level changes.

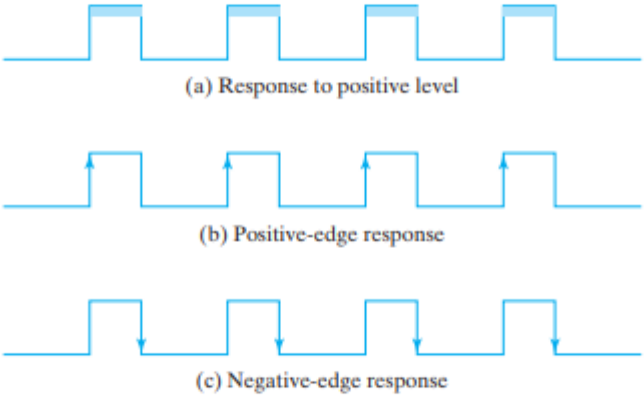
Positive and Negative Transitions: Flip-flops distinguish between two clock signal transitions: the positive edge (0 to 1) and the negative edge (1 to 0). These transitions are crucial for proper operation. As shown in figure 2.36

Feedback Elimination: To transform a latch into a flip-flop, the feedback path inherent in latches must be eliminated to prevent output interference while input changes occur.

Dual-Latch Configuration: One approach to creating a flip-flop is using two latches in a special configuration. This setup isolates the flip-flop's output and safeguards it from changes during input transitions.

Transition-Triggered Flip-Flop: Another method is to design a flip-flop that triggers exclusively during signal transitions (0 to 1 or 1 to 0) of the clock signal. It remains disabled during the remainder of the clock pulse for reliable operation in sequential circuits.

Synchronization with Clock: Flip-flops are synchronized with a clock signal to ensure changes occur precisely during clock transitions, enhancing the predictability and reliability of sequential circuit operation.


(a) Response to positive level

(b) Positive-edge response

(c) Negative-edge response

FIGURE 2.36 Clock response in latch and flip-flop

Edge-Triggered D Flip-Flop

Master and Slave Latches: The D flip-flop consists of two D latches: a master and a slave latch. These latches work together to store and transfer data.

Clock-Triggered Operation: The D flip-flop operates based on a clock signal (Clk). Changes in the output Q occur only at the negative edge (falling edge) of the clock signal. This means that data is sampled and transferred precisely at the moment when the clock transitions from 1 to 0.

Inverter and Clock Control: An inverter is used to control the enable inputs of the master and slave latches. When the clock signal is 0, the inverter outputs 1, enabling the slave latch and disabling the master latch. This means that the flip-flop is in a "latch" state when the clock is low, and no changes can occur in Q. As shown in the figure 2.37.

Data Transfer on Clock Transition: When the clock signal transitions from 0 to 1, the D input data is transferred to the master latch. However, the slave latch remains disabled because its enable input is 0 when the clock is high. Therefore, changes in the D input do not affect the flip-flop's output during this time.

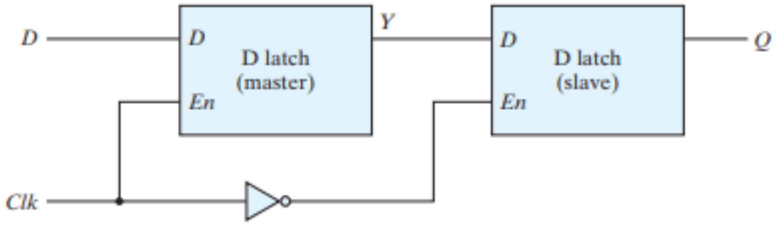


FIGURE 2.37 Master–slave D flip-flop

Output Update on Falling Clock Edge: The key operation occurs when the clock signal transitions from 1 to 0 (falling edge). At this moment, the master latch is disabled and isolated from the D input, while the slave latch is enabled. The value stored in the master latch (Y) is then transferred to the flip-flop's output Q. Therefore, any change in the output of the flip-flop can only occur during this specific clock transition.

The behavior of the master–slave flip-flop just described dictates that

(1) the output may change only once,

(2) a change in the output is triggered by the negative edge of the clock, and

(3) the change may occur only during the clock's negative level. The value that is produced at the output of the flip-flop is the value that was stored in the master stage immediately before the negative edge occurred .

It is also possible to design the circuit so that the flip-flop output changes on the positive edge of the clock. This happens in a flip-flop that has an additional inverter between the Clk terminal and the junction between the other inverter and input En of the master latch. Such a flip-flop is triggered with a negative pulse, so that the negative edge of the clock affects the master and the positive edge affects the slave and the output terminal. Another construction of an edge-triggered D flip-flop uses three SR latches as shown in Fig. 2.38 . Two latches respond to the external D (data) and Clk (clock) inputs. The third latch provides the outputs for the flip-flop. The S and R inputs of the output latch are maintained at the logic-1 level when Clk = 0. This causes the output to remain in its present state. Input D may be equal to 0 or 1. If D = 0 when Clk becomes 1, R changes to 0. This causes the flip-flop to go to the reset state, making Q = 0. If there is a change in the D input while Clk = 1, terminal R remains at 0 because Q is 0. Thus, the flip-flop is locked out and is unresponsive to further changes in the input. When the clock returns to 0, R goes to 1, placing the output latch in the quiescent condition without changing the output. Similarly, if D = 1 when Clk goes from 0 to 1, S changes to 0. This causes the circuit to go to the set state, making Q = 1. Any change in D while Clk = 1 does not affect the output.

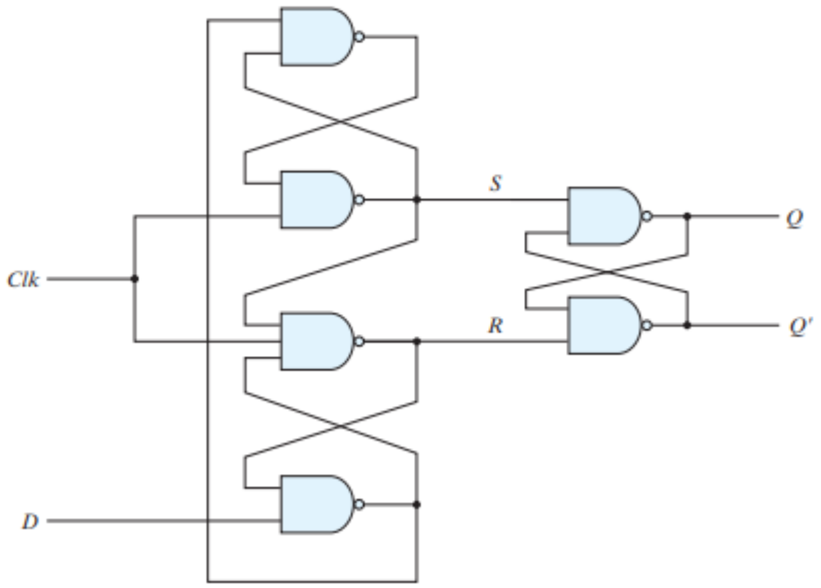
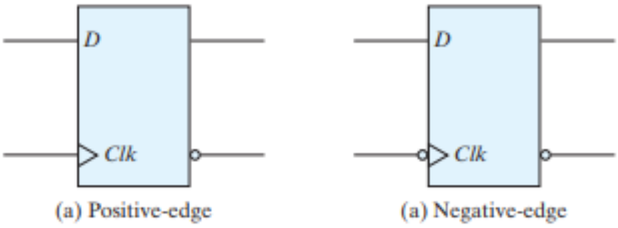


FIGURE 2.38 D-type positive-edge-triggered flip-flop

In sum, when the input clock in the positive-edge-triggered flip-flop makes a positive transition, the value of D is transferred to Q . A negative transition of the clock (i.e., from 1 to 0) does not affect the output, nor is the output affected by changes in D when Clk is in the steady logic-1 level or the logic-0 level. Hence, this type of flip-flop responds to the transition from 0 to 1 and nothing else.

The timing of the response of a flip-flop to input data and to the clock must be taken into consideration when one is using edge-triggered flip-flops. There is a minimum time called the setup time during which the D input must be maintained at a constant value prior to the occurrence of the clock transition. Similarly, there is a minimum time called the hold time during which the D input must not change after the application of the positive transition of the clock. The propagation delay time of the flip-flop is defined as the interval between the trigger edge and the stabilization of the output to a new state.

The graphic symbol for the edge-triggered D flip-flop is shown in Fig. 2.39 . It is similar to the symbol used for the D latch, except for the arrowhead-like symbol in front of the letter Clk, designating a dynamic input. The dynamic indicator (>) denotes the fact that the flip-flop responds to the edge transition of the clock. A bubble outside the block adjacent to the dynamic indicator designates a negative edge for triggering the circuit.



(a) Positive-edge

(a) Negative-edge

FIGURE 2.39 Graphic symbol for edge-triggered D flip-flop

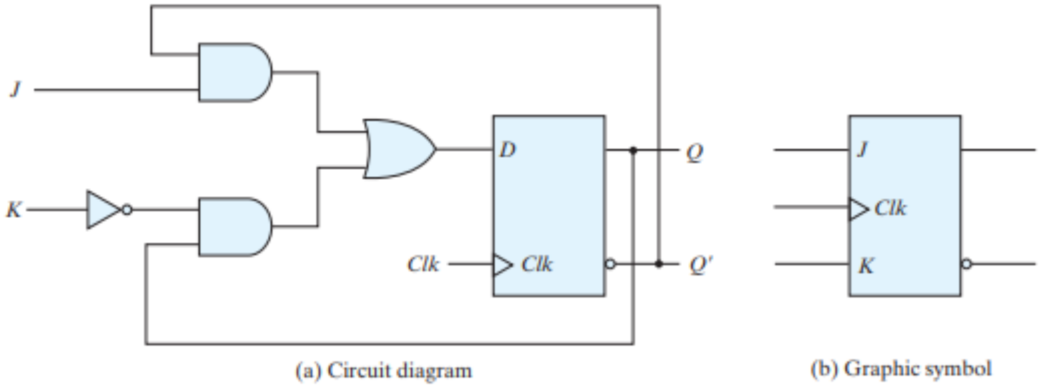
Other Flip-Flops

Other types of flip-flops can be constructed by using the D flip-flop and external logic. Two flip-flops less widely used in the design of digital systems are the JK and T flip-flops.

There are three operations that can be performed with a flip-flop: Set it to 1, reset it to 0, or complement its output. With only a single input, the D flip-flop can set or reset the output, depending on the value of the D input immediately before the clock transition. Synchronized by a clock signal, the JK flip-flop has two inputs and performs all three operations. The circuit diagram of a JK flip-flop constructed with a D flip-flop and gates is shown in Fig. 2.40 (a). The J input sets the flip-flop to 1, the K input resets it to 0, and when both inputs are enabled, the output is complemented. This can be verified by investigating the circuit applied to the D input:

$D = JQ' + K'Q$

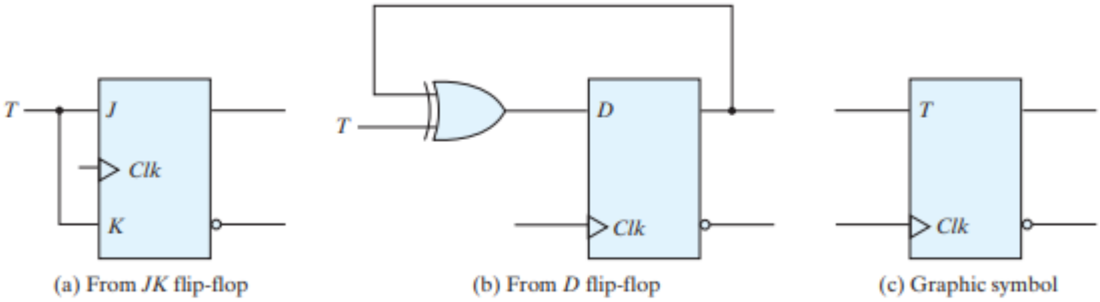
When J = 1 and K = 0, D = Q' + Q = 1, so the next clock edge sets the output to 1. When J = 0 and K = 1, D = 0, so the next clock edge resets the output to 0. When both J = K = 1 and D = Q', the next clock edge complements the output. When both J = K = 0 and D = Q, the clock edge leaves the output unchanged. The graphic symbol for the JK flip-flop is shown in Fig. 2.40 (b). It is similar to the graphic symbol of the D flip-flop, except that now the inputs are marked J and K .



(a) Circuit diagram (b) Graphic symbol

FIGURE 2.40 JK flip-flop

The T (toggle) flip-flop is a complementing flip-flop and can be obtained from a JK flip-flop when inputs J and K are tied together. This is shown in Fig. 2.41 (a).



(a) From *JK* flip-flop (b) From *D* flip-flop (c) Graphic symbol

FIGURE 2.41 T flip-flop

When T = 0 (J = K = 0), a clock edge does not change the output. When T = 1 (J = K = 1), a clock edge complements the output. The complementing flip-flop is useful for designing binary counters. The T flip-flop can be constructed with a D flip-flop and an exclusive-OR gate as shown in Fig. 2.41 (b). The expression for the D input is

$$D = T \oplus Q = TQ' + T'Q$$

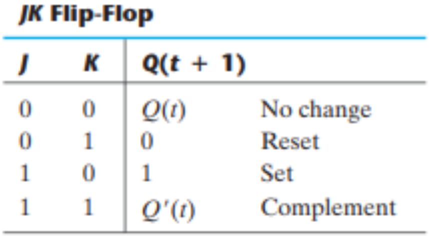
When T = 0, D = Q and there is no change in the output. When T = 1, D = Q" and the output complements.

The graphic symbol for this flip-flop has a T symbol in the input.

Characteristic Tables

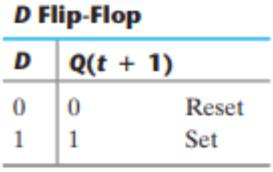
A characteristic table defines the logical properties of a flip-flop by describing its operation in tabular form. The characteristic tables of three types of flip-flops are presented in Table 2.10 . They define the next state .

Table 2.10 Flip-Flop Characteristic Tables



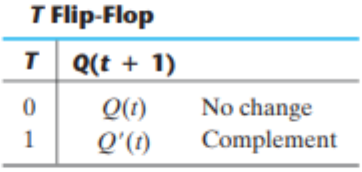
***JK* Flip-Flop**

J	*K*	Q(*t* + 1)	
0	0	$Q(t)$	No change
0	1	0	Reset
1	0	1	Set
1	1	$Q'(t)$	Complement



***D* Flip-Flop**

D	Q(*t* + 1)	
0	0	Reset
1	1	Set



***T* Flip-Flop**

T	Q(*t* + 1)	
0	$Q(t)$	No change
1	$Q'(t)$	Complement

as a function of the inputs and the present state. Q (t) refers to the present state (i.e., the state present prior to the application of a clock edge). Q(t + 1) is the next state one clock period later. Note that the clock edge input is not included in the characteristic table, but is implied to occur between times t and t + 1. Thus, Q(t) denotes the state of the flip-flop immediately before the clock edge, and Q(t + 1) denotes the state that results from the clock transition.

The characteristic table for the JK flip-flop shows that the next state is equal to the present state when inputs J and K are both equal to 0. This condition can be expressed as Q(t + 1) = Q(t), indicating that the clock produces no change of state. When K = 1 and J = 0, the clock resets the flip-flop and Q(t + 1) = 0. With J = 1 and K = 0, the flip-flop sets and Q(t + 1) = 1. When both J and K are equal to 1, the next state changes to the complement of the present state, a transition that can be expressed as Q(t + 1) = Q'(t).

The next state of a D flip-flop is dependent only on the D input and is independent of the present state. This can be expressed as Q(t + 1) = D. It means that the next-state value is equal to the value of D . Note that the D flip-flop does not have a "no-change" condition.

The characteristic table of the T flip-flop has only two conditions: When T = 0, the clock edge does not change the state; when T = 1, the clock edge complements the state of the flip-flop.

The logical properties of a flip-flop, as described in the characteristic table, can be expressed algebraically with a characteristic equation. For the D flip-flop, we have the characteristic equation

$$\mathbf{Q(t + 1) = D}$$

which states that the next state of the output will be equal to the value of input D in the present state. The characteristic equation for the JK flip-flop can be derived from the characteristic table or from the circuit of Fig. 2.40 . We obtain

$$Q(t + 1) = JQ' + K'Q$$

where Q is the value of the flip-flop output prior to the application of a clock edge. The characteristic equation for the T flip-flop is obtained from the circuit of Fig. 2.41 :

$$D = T \oplus Q = TQ' + T'Q$$

When T = 0, D = Q and there is no change in the output. When T = 1, D = Q’ and the output complements. The graphic symbol for this flip-flop has a T symbol in the input.

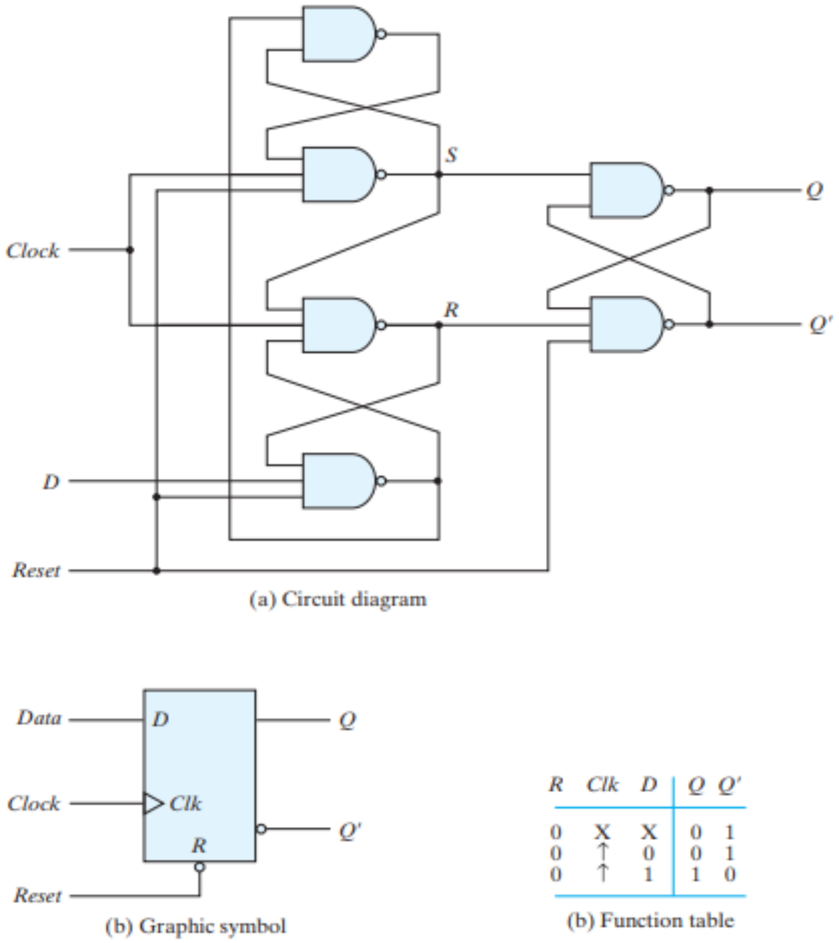
Direct Inputs

Some flip-flops have asynchronous inputs that are used to force the flip-flop to a particular state independently of the clock. The input that sets the flip-flop to 1 is called preset or direct set . The input that clears the flip-flop to 0 is called clear or direct reset . When power is turned on in a digital system, the state of the flip-flops is unknown. The direct inputs are useful for bringing all flip-flops in the system to a known starting state prior to the clocked operation.

A positive-edge-triggered D flip-flop with active-low asynchronous reset is shown in Fig. 2.42. When the reset input is 0, it forces output Q’ to stay at 1, which, in turn, clears output Q to 0, thus resetting the flip-flop. Two other connections from the reset input ensure that the S input of the third SR latch stays at logic 1 while the reset input is at 0, regardless of the values of D and Clk .

The graphic symbol for the D flip-flop with a direct reset has an additional input marked with R . The bubble along the input indicates that the reset is active at the logic-0 level. Flip-flops with a direct set use the symbol S for the asynchronous set input.

The function table specifies the operation of the circuit. When R = 0, the output is reset to 0. This state is independent of the values of D or Clk . Normal clock operation can proceed only after the reset input goes to logic 1. The clock at Clk is shown with an upward arrow to indicate that the flip-flop triggers on the positive edge of the clock. The value in D is transferred to Q with every positive-edge clock signal, provided that R = 1.



(a) Circuit diagram

(b) Graphic symbol

R	Clk	D	Q	Q'
0	X	X	0	1
0	↑	0	0	1
0	↑	1	1	0

(b) Function table

FIGURE 2.42 D flip-flop with asynchronous reset

U	S	N									

||Jai Sri Gurudev ||

BGSKH Education Trust (R.) – A unit of Sri Adichunchanagiri Shikshana Trust(R.)

BGS College of Engineering and Technology

Mahalakshmipuram, West of Chord Road, Bengaluru-560086

(Approved by AICTE, New Delhi and Affiliated to VTU, Belagavi)

QUESTION BANK
MODULE 2

Subject:	**Digital Design and Computer Organization**	Subject Code	**BCS302**
Faculty Name:	MG, ASN,AR,HN		

Sl. No	**QUESTIONS**	**RBTL**
1	What is a combinational circuit? Design a BCD to excess 3 converter with relevant truth table and logic diagram	L4
2	Explain the operation of 4:1 Mux with relevant truth table and Logic diagram	L3
3	Differentiate between Combinational and Sequential circuits	L1
4	Design Carry Look Ahead Adder with relevant Circuit diagrams.	L3
5	With relevant truth table and logic diagram, explain full Adder circuit.	L2
6	What are decoders? Implement the following Boolean functions with a decoder: F1(A,B,C) = ∑m(1, 3,4,7), F2(A,B,C) = ∑m(0,2,3,6) and F3(A,B,C) = ∑m(2,3,6,7)	L3
7	What are Multiplexers? Implement the Boolean function F(A,B,C,D) = ∑m(1,3,4,11,12,13,14,15) with a 8:1 MUX	L2
8	Define Encoder. Design a Four-input Priority Encoder.	L2
9	Write the Verilog program to Implement Full Adder and Subtractor Circuits.	L2
10	Write the Characteristic Table and Equations of SR, JK, D and T Flip Flops	L2
11	Design an Octal-to-Binary Encoder.	L3
12	Explain the working of Four-bit adders using 4-Full Adders.	L2
13	Explain Dataflow Modeling in Verilog with an example program.	L2
14	Demonstrate the working of SR Latch and Edge-Triggered D Flip-Flop.	L3
15	What are Multiplexer? Explain 8:1 MUX.	L2
16	Define: a)Encoder b) Decoder c) Multiplexer	L1
17	Explain Parallel Adder with neat diagram.	L2
18	Explain adder/ sub tractor with neat diagram	L2
19	Briefly explain a) SR Flip Flop b) D Flip Flop c) T Flip Flop d) JK Flip Flop	L2
20	Write the Verilog program to Implement 8: 1 Multiplexer Circuits.	L3

Basic Structure of Computers, Instructions and Programs

3.1 FUNCTIONAL UNITS

With a neat diagram, describe the functional units of a computer.

A computer consists of five functionally independent main parts:

- **Input**
- **Memory**
- **Arithmetic logic unit**
- **Output**
- **Control units**

- The input unit accepts the coded information from human operators, from electromechanical devices such as keyboards, or from other computers over digital communication lines
- The information received is either stored in the computer's memory for later reference or immediately used by the arithmetic and logic circuitry to perform the desired operations.
- The results are sent back to the outside world through the output unit
- All these actions are coordinated by the control unit

It is convenient to categorize the information handled by the computer as either data or instructions Instructions or machine instructions are explicit commands that

- Govern the transfer of information within a computer as well as between the computer and its I/O devices.
- Specify the arithmetic and logic operations to be performed.

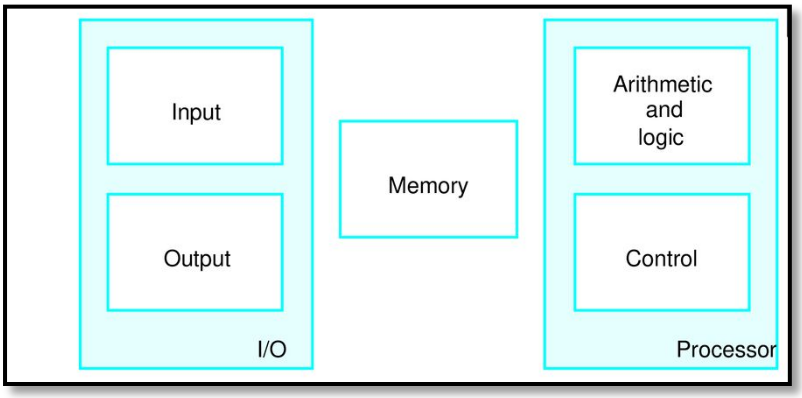


Fig 3.1 Functional Units of a Processor

3.2 BASIC OPERATIONAL CONCEPTS:

With a neat diagram, analyze the basic operational concepts of a computer. List the operating steps.

The program to be executed is stored in memory. Instructions are accessed from memory to the processor one by one and executed.

STEPS FOR INSTRUCTION EXECUTION

Consider the following instruction

Ex: 1 Add LOCA, R_0

This instruction is in the form of the following instruction format

Opcode Source, Destination

Where Add is the *operation code,* LOCA is the Memory operand and R_0 is Register operand This instruction adds the contents of memory location LOCA with the contents of Register R_0 and the result is stored in R_0 Register.

The symbolic representation of this instruction is

R_0 [LOCA] + [R_0]

The contents of memory location LOCA and Register R_0 before and after the execution of this instruction is as follows:

Before instruction execution	After instruction execution
LOCA = 23H	LOCA = 23H
R_0 = 22H	**R_0 = 45H**

Ex:2 Add R_1, R_2, R_3

This instruction is in the form of the following instruction format

Opcode, Source-1, Source-2, Destination

Where R1 is Source Operand-1, R2 is the Source Operand-2 and R3 is the Destination. This instruction adds the contents of Register R1 with the contents of R2 and the result is placed in R3 Register.

The symbolic representation of this instruction is

R3 [R1] + [R2]

The contents of Registers R1,R2,R3 before and after the execution of this instruction is as follows.

Before instruction execution.	After instruction execution
R1 = 24H	R1 = 24H
R2 = 34H	**R2 = 34H**
R3 = 38H	R3 = 58H

The steps for instruction execution is as follows

- Fetch the instruction from memory into the IR.
- Decode the instruction
- Access the First Register Operand R1
- Access the Second Register Operand R2
- Perform the operation according to the Operation Code.
- Store the result into the Destination Register R3.

3.2.1 CONNECTION BETWEEN MEMORY AND PROCESSOR

With a neat diagram, explain the bus structure of computer and its components

The connection between Memory and Processor is as shown in the figure.

The Processor consists of different types of registers.

- MAR (Memory Address Register)
- MDR (Memory Data Register)
- Control Unit
- PC (Program Counter)
- General Purpose Registers
- IR (Instruction Register)
- ALU (Arithmetic and Logic Unit)

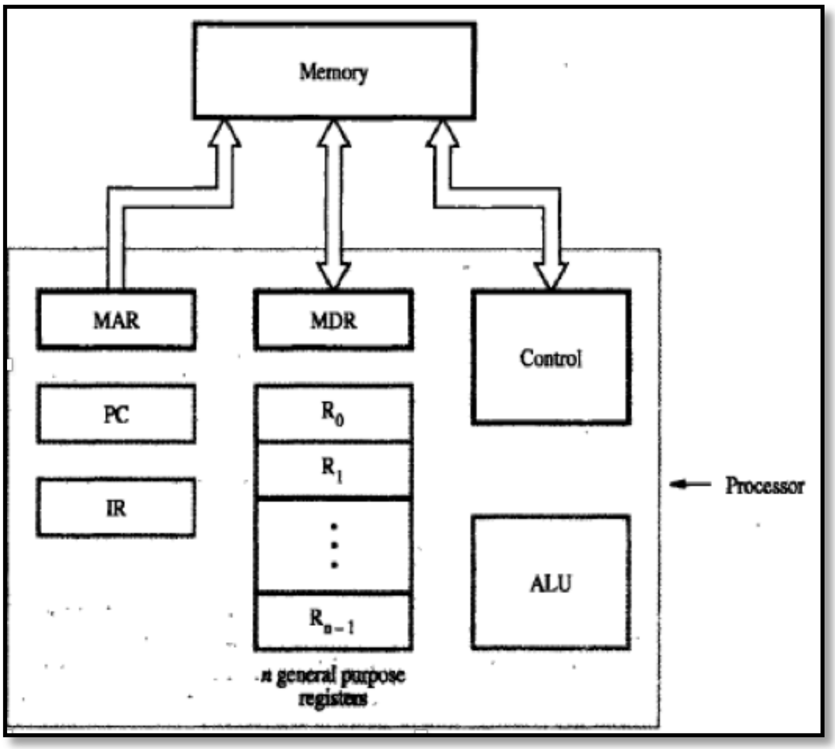
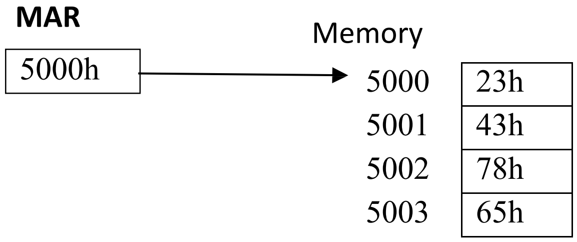


Fig 3.2 Bus Structure

The functions of these components are as follows

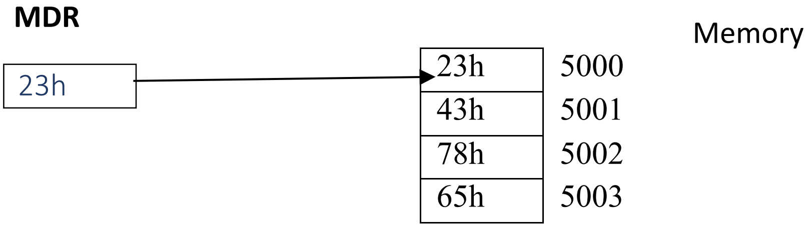
MAR – Memory Address Register

- It establishes communication between Memory and Processor
- It stores the address of the Memory Location as shown in the figure.



MDR – Memory Data Register

- It also establishes communication between Memory and the Processor.
- It stores the **contents** of the memory location (data or operand), written into or read from memory as shown in the figure.



CONTROL UNIT

- It controls the data transfer operations between memory and the processor.
- It controls the data transfer operations between I/O and processor.
- It generates control signals for Memory and I/O device

PC (PROGRAM COUNTER)

- It is a special purpose register used to hold the address of the next instruction to be executed.
- The contents of PC are incremented by 1 or 2 or 4, during the execution of current instruction.
- The contents of PC are incremented by 1 for 8 bit CPU, 2 for 16 bit CPU and for 4 for 32 bit CPU.

GENERAL PURPOSE REGISTER / REGISTER ARRAY

The structure of register file is as shown in the figure

R_0
R_1
R_2
.
R_{n-1}

- It consists of set of registers.
- A register is defined as group of flip flops. Each flip flop is designed to store 1 bit of data.
- It is a storage element.
- It is used to store the data temporarily during the execution of the program(eg: result).
- It can be used as a pointer to Memory.
- The Register size depends on the processing speed of the CPU
- EX: Register size = 8 bits for 8 bit CPU

IR (INSTRUCTION REGISTER)

It holds the instruction to be executed. It notifies the control unit, which generates timingsignals that controls various operations in the execution of that instruction.

ALU (ARITHMETIC and LOGIC UNIT)

It performs arithmetic and logical operations on given data.

Steps for reading the instruction.

PC contents are transferred to MAR and read signal is sent to memory by control unit.The data from memory location is read and sent to MDR.

The content of MDR is moved to IR.

$$[PC] \rightarrow MAR \xrightarrow[\text{CU (read signal)}]{} \text{Memory} \rightarrow MDR \rightarrow IR$$

3.3 BUS STRUCTURE

Bus is defined as set of parallel wires used for data communication between different parts of computer. Each wire carries 1 bit of data. There are 3 types of buses, namely

1. Address bus
2. Data bus and
3. Control bus.

Address bus :

- It is unidirectional.
- The processor (CPU) sends the address of an I/O device or Memory device by means of this bus.

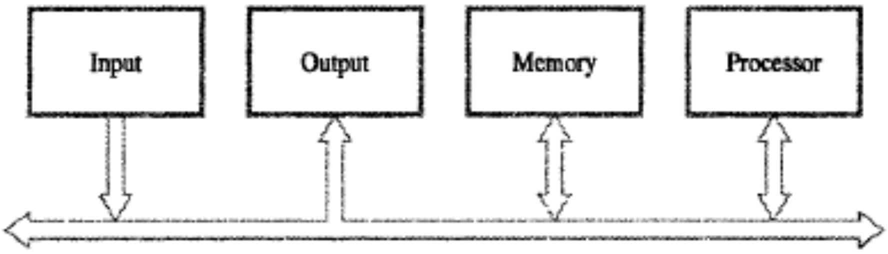
Data bus

- It is a bidirectional bus.
- The CPU sends data from Memory to CPU and vice versa as well as from I/O to CPU and vice versa by means of this bus.

Control bus:

This bus carries control signals for Memory and I/O devices. It generates control signals for Memory namely MEMRD and MEMWR and control signals for I/O devices namely IORD and IOWR.

The structure of single bus organization is as shown in the figure.



- The I/O devices, Memory and CPU are connected to this bus is as shown in the figure.
- It establishes communication between two devices, at a time.

Features of Single bus organization are

- Less Expensive
- Flexible to connect I/O devices.
- Poor performance due to single bus.

There is a variation in the devices connected to this bus in terms of speed of operation. Few devices like keyboard, are very slow. Devices like optical disk are faster. Memory and processor are faster, but all these devices uses the same bus. Hence to provide the synchronizationbetween two devices, a buffer register is attached to each device. It holds the data temporarily during the data transfer between two device.

3.4 PERFORMANCE

List the factors affecting the performance of Computer and discuss the methods to improve the performance of the processor

- The performance of a Computer System is based on **hardware design** of the processor andthe instruction set of the processors.
- To obtain high performance of computer system it is necessary to reduce the execution time of the processor.
- **Execution time:** It is defined as total time required executing one complete program.
- The processing time of a program includes time taken to read inputs, display outputs, system services, execution time etc.
- The performance of the processor is inversely proportional to execution time of the processor.

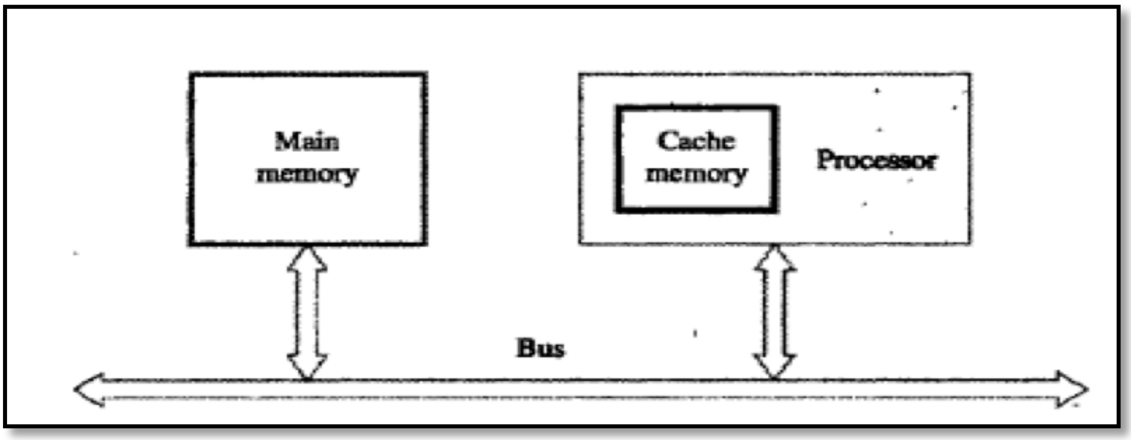
More performance = Less Execution time.

Less Performance = More Execution time.

The Performance of the Computer System is based on the following factors

1. *Cache Memory*
2. *Processor clock*
3. *Basic Performance Equation*
4. *Instructions*
5. *Compiler*

3.4.1 CACHE MEMORY: It is defined as a *fast access memory* located in between CPU and Memory. It is part of the processor as shown in the fig



The processor needs more time to read the data and instructions from main memory because main memory is away from the processor as shown in the figure. Hence it slowdown theperformance of the system.

The processor needs less time to read the data and instructions from Cache Memory because it is part of the processor. Hence it improves the performance of the system.

3.4.2 PROCESSOR CLOCK:

The processor circuits are controlled by timing signals called as Clock. It defines constant time intervals and are called as Clock Cycles. To execute one instruction thereare 3 basic steps namely

1. Fetch
2. Decode
3. Execute.

The processor uses one clock cycle to perform one operation as shown in the figure

Clock Cycle	→	T1	T2	T3
Instruction	→	Fetch	Decode	Execute

The performance of the processor depends on the length of the clock cycle. To obtain high performance reduce the length of the clock cycle. Let ‘ P ’ be the number of clock cycles generated by the Processor and ‘ R ‘ be the Clock rate .

The Clock rate is inversely proportional to the number of clock cycles.

i.e $R = 1/P$.

Cycles/second is measured in Hertz (Hz). Eg: 500MHz, 1.25GHz.

Two ways to increase the clock rate –

- Improve the IC technology by making the logical circuit work faster, so that the time taken for the basic steps reduces.
- Reduce the clock period, P.

3.5 BASIC PERFORMANCE EQUATION

Discuss the performance metrics of Computer. Explain SPEC Rating of a computer.

Let ‘ T ‘ be *total time* required to execute the program.

Let ‘N ‘ be the *number of instructions* contained in the program.

Let ‘ S ‘ be the *average number of steps* required to one instruction.

Let ‘ R’ be number of clock cycles per second generated by the processor to execute one program.

Processor Execution Time is given by

$$T = N * S / R$$

This equation is called as Basic Performance Equation.

For the programmer the value of T is important. To obtain high performance it is necessary to reduce the values of N & S and increase the value of R

Performance of a computer can also be measured by using **benchmark** programs.

SPEC (System Performance Evaluation Corporation) is an non-profitable organization, that measures performance of computer using SPEC rating. The organization publishes the application programs and also time taken to execute these programs in standard systems.

$$SPEC = \frac{Running\ time\ of\ reference\ Computer}{Running\ time\ of\ computer\ under\ test}$$

Differentiate between Multiprocessor and Multicomputer system

MULTIPROCESSOR	MULTICOMPUTER
1. It is a process of interconnection of two or more processors by means of system bus.	It is a process of interconnection of two or more computers by means of system bus.
2. It uses common memory to hold the data and instructions.	It has its own memory to store data and instructions.
3. Complexity in hardware design.	Not much complexity in hardware design.
4. Difficult to program for multiprocessor system.	Easy to program for multiprocessor system

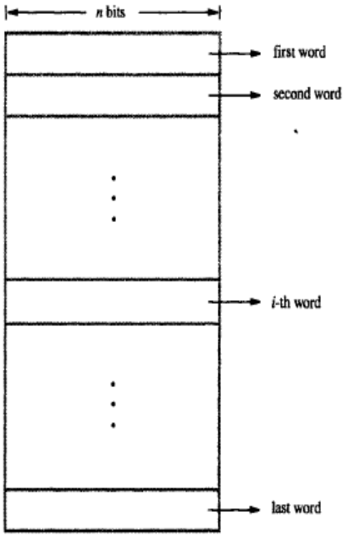
3.6 MEMORY LOCATIONS AND ADDRESSES

- Memory is a storage device. It is used to store character operands, data operands and instructions.
- It consists of number of semiconductor cells and each cell holds 1 bit of information. A group of 8 bits is called as byte and a group of 16 or 32 or 64 bits is called as word.

What is word length? Explain with neat diagram memory organization of the computer.

World length = 16 for 16 bit CPU and World length = 32 for 32 bit CPU. Word length is defined as number of bits in a word.

- Memory is organized in terms of bytes or words.
- The organization of memory for 32 bit processor is as shown in the fig.



The contents of memory location can be accessed for read and write operation. The memory is accessed either by specifying address of the memory location or by name of the memory location.

Address space : It is defined as number of bytes accessible to CPU and it depends on thenumber of address lines.

3.6.1 BYTE ADDRESSABILITY

Each byte of the memory are addressed, this addressing used in most computers are called byte addressability. Hence Byte Addressability is the process of assignment of address to successive bytes of the memory. The successive bytes have the addresses 1, 2, 3, 4…............ 2^n-1. The memory is accessed in words.

In a 32 bit machine, each word is 32 bit and the successive addresses are 0,4,8,12,… and so on.

Address	← 32 – bit word →			
0000	0th byte	1st byte	2nd byte	3rd byte
0004	4th byte	5th byte	6th byte	7th byte
0008	8th byte	9th byte	10th byte	11th byte
0012	12th byte	13th byte	14th byte	15th byte
…..	…..	…..	…..	…..
n-3	n-3th byte	n-2th byte	n-1th byte	nth byte

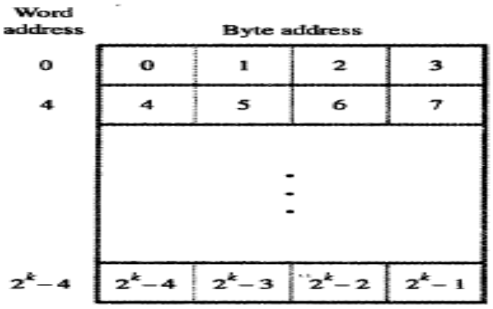
3.6.2 BIG ENDIAN and LITTLE ENDIAN ASSIGNMENT

Analyze Big Endian and Little Endian methods of byte addressing with relevant example.

Two ways in which a word is stored in memory.

1. Big endian
2. Little endian

BIG ENDIAN ASSIGNMENT



Word address	Byte address			
0	0	1	2	3
4	4	5	6	7
	⋮			
2^k-4	2^k-4	2^k-3	2^k-2	2^k-1

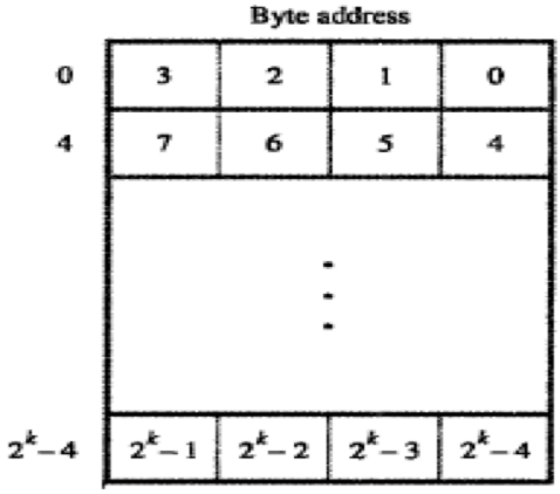
In this technique lower byte of data is assigned to higher address of the memory and higher byte of data is assigned to lower address of the memory

The structure of memory to represent 32 bit number for big endian assignment is as shown in the above figure.

LITTLE ENDIAN ASSIGNMENT

In this technique lower byte of data is assigned to lower address of the memory and higher byteof data is assigned to higher address of the memory.

The structure of memory to represent 32 bit number for little endian assignment is as shown inthe fig.



Byte address

0	3	2	1	0
4	7	6	5	4
	⋮			
2^k-4	2^k-1	2^k-2	2^k-3	2^k-4

Eg – store a word "JOHNSENA" in memory starting from word 1000, using Big Endianand Little endian.

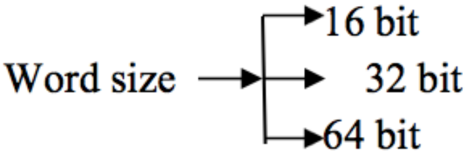
Bigendian -

1000	J 1000	O 1001	H 1002	N 1003
1004	S 1004	E 1005	N 1006	A 1007

Little endian -

1000	J 1003	O 1002	H 1001	N 1000
1004	S 1007	E 1006	N 1005	A 1004

WORD ALLIGNMENT



The structure of memory for 16 bit CPU, 32 bit CPU and 64 bit CPU are as shown in the figures1,2 and 3 respectively

For 16 bit CPU

5000	**34H**
5002	**65H**
5004	**86H**
5006	**93H**
5008	**45H**

For 32 bit CPU

5000	**34H**
5004	**65H**
5008	**86H**
5012	**93H**
5016	**45H**

For 64 bit CPU

5000	**34H**
5008	**65H**
5016	**86H**
5024	**93H**
5032	**45H**

It is process of assignment of addresses of two successive words and this address is the number of bytes in the word is called as Word alignment.

ACCESSING CHARACTERS AND NUMBERS

The character occupies 1 byte of memory and hence byte address for memory.
The numbers occupies 2 bytes of memory and hence word address for numbers.

3.6.2 MEMORY OPERATION

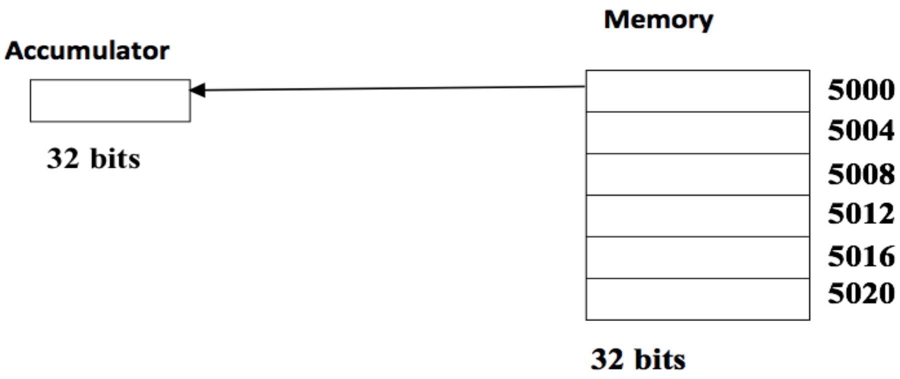
Briefly explain Memory operations.

Both program instructions and operands are in memory. To execute each instruction has to be read from memory and after execution the results must be written to memory. Thereare two types of memory operations namely 1. Memory read and 2. Memory write

Memory read operation [Load/ Read / Fetch]
Memory write operation [Store/ write]

1. MEMORY READ OPERATION:

- ✓ It is the process of transferring of 1 word of data from memory into Accumulator (GPR).
- ✓ It is also called as Memory fetch operation.
- ✓ The Memory read operation can be implemented by means of LOAD instruction.
- ✓ The LOAD instruction transfers 1 word of data (1 word = 32 bits) from Memory into the Accumulator as shown in the fig.



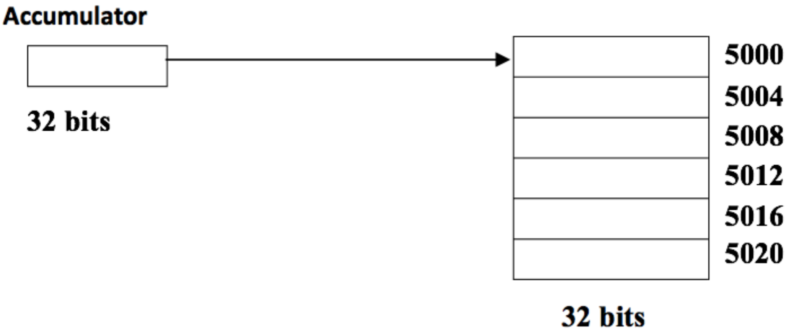
Steps for Memory Read Operation

(1) The processor loads MAR (Memory Address Register) with the address of the memory location.
(2) The Control unit of processor issues memory read control signal to enable the memory component for read operation.
(3) The processor reads the data from memory into the Accumulator by means of bi-directional data bus.

[MAR] → Memory → Accumulator

MEMORY WRITE OPERATION

- It is the process of transferring the 1 word of data from Accumulator into the Memory.
- The Memory write operation can be implemented by means of STORE instruction. The STORE instruction transfers 1 word of data from Accumulator into the Memory location as shown in the fig.



Steps for Memory Write Operation

- The processor loads MAR with the address of the Memory location.
- The Control Unit issues the Memory Write control signal.
- The processor transfers 1 word of data from the Accumulator into the Memory location by means of bi-directional data bus.

3.7 COMPUTER OPERATIONS (OR) INSTRUCTIONSAND INSTRUCTION EXECUTION

Explain with suitable example, various operations the computer is designed to perform

The Computer is designed to perform 4 types of operations, namely

- Data transfer operations
- ALU Operations
- Program sequencing and control.
- I/O Operations.

3.7.1 Data Transfer Operation

Data transfer between two registers.

Format: Opcode Source1 , Destination

The processor uses MOV instruction to perform data transfer operation between two registers The mathematical representation of this instruction is R1 → R2.

Ex : MOV R_1 , R_2 ; R1 and R2 are the registers.

Where MOV is the operation code, R1 is the source operand and R2 is the destination operand. This instruction transfers the contents of R1 to R2.

EX: Before the execution of MOV R1,R2, the contents of R1 and R2 are as follows

R1 = 34h and R2 = 65h

After the execution of MOV R1, R2, the contents of R1 and R2 are as follows

R1 = 34H and R2 = 34H

i) Data transfer from memory to register

The processor uses **LOAD** instruction to perform data transfer operation from memory to register. The mathematical representation of this instruction is

[LOCA] → ACC. Where ACC is the Accumulator.

Format : opcode operand

Ex: LOAD LOCA

For this instruction Memory Location is the source and Accumulator is the destination.

ii) Data transfer from Accumulator register to memory

The processor uses **STORE** instruction to perform data transfer operation from Accumulator register to memory location. The mathematical representation of this instruction is

[ACC] → LOCA. Where, ACC is the Accumulator.

Format: opcode operand

Ex: STORE LOCA

For this instruction accumulator is the source and memory location is the destination.

3.7.2 ALU Operations

The instructions are designed to perform arithmetic operations such as Addition, Subtraction, Multiplication and Division as well as logical operations such as AND, OR and NOT operations.

Ex1: ADD R_0, R_1

The mathematical representation of this instruction is as follows:

$R_1 \leftarrow [R_0] + [R_1]$; Adds the content of R_0 with the content of R_1 and result is placed in R_1.

Ex2: SUB R_0, R_1

The mathematical representation of this instruction is as follows:

$R_1 \leftarrow [R_0] - [R_1]$; Subtracts the content of R_0 from the content of R_1 and result is placedin R_1.

EX3: AND R_0, R_1 ;

It Logically multiplies the content of R_0 with the content of R_1 and result isstored in R_1. (R_1= R_0 AND R1)

Ex4: NOT R_0 ; It performs the function of complementation.

3.7.3 I/O Operations:

The instructions are designed to perform INPUT and OUTPUToperations. The processor uses MOV instruction to perform I/O operations.

The input Device consists of one temporary register called as DATAIN register and output register consists of one temporary register called as DATAOUT register.

i) Input Operation: It is a process of transferring one WORD of data from DATA INregister to processor register.

Ex: MOV DATAIN, R0

The mathematical representation of this instruction is as follows,

$R_0 \leftarrow$ [DATAIN]

ii) Output Operation: It is a process of transferring one WORD of data from processor register to DATAOUT register.

Ex: MOV $R_{0,}$ DATAOUT

The mathematical representation of this instruction is as follows,

$[R_0] \rightarrow$ DATAOUT

Briefly explain RTN and ALN with examples.

REGISTER TRANSFER NOTATION

- There are 3 locations to store the operands during the execution of the program namely
  1. Register
  2. Memory location
  3. I/O Port.

Location is the storage space used to store the data.

- The instructions are designed to transfer data from one location to another location. Consider the first statement to transfer data from one location to another location
- " Transfer the contents of Memory location whose symbolic name is given by AMOUNT into processor register $R_{0.}$"
- The mathematical representation of this statement is given by

  $R_0 \leftarrow$ [AMOUNT]

  Consider the second statement to add data between two registers
- "Add the contents of R_0 with the contents of R_1 and result is stored in R_2"
- The mathematical representation of this statement is given by

  $R_2 \leftarrow [R_0] + [R_1]$.

Such a notation is called as "Register Transfer Notation".

It uses two symbols

1. A pair of square brackets [] to indicate the contents of Memory location and
2. ← to indicate the data transfer operation.

ASSEMBLY LANGUAGE NOTATION

Consider the first statement to transfer data from one location to another location

- Transfer the contents of Memory location whose symbolic name is given byAMOUNT into processor register R_0."
- The assembly language notation of this statement is given by
  Opcode MOV AMOUNT, R_0
  Source Destination
- This instruction transfers 1 word of data from Memory location whose symbolic name is given byAMOUNT into the processor register R_0.
- The mathematical representation of this statement is given by

$R_0 \leftarrow$ [AMOUNT]

Consider the second statement to add data between two registers

- "Add the contents of R_0 with the contents of R_1 and result is stored in R_2"
- The assembly language notation of this statement is given by

ADD R_0 , R_1, R_2
Opcode source1, Source2, Destination

This instruction adds the contents of R_0 with the contents of R_1 and result is stored in R_2.

- The mathematical representation of this statement is given by
  $R_2 \leftarrow [R_0] + [R_1]$.

Such a notations are called as "Assembly Language Notations"

BASIC INSTRUCTION TYPES

Explain with examples various instruction types

There are 3 types basic instructions namely

1. Three address instruction format
2. Two address instruction format
3. One address instruction format

Consider the arithmetic expression Z = A + B, Where A,B,Z are the Memory locations.

Steps for evaluation

1. Access the first memory operand whose symbolic name is given by A.
2. Access the second memory operand whose symbolic name is given by B.
3. Perform the addition operation between two memory operands.
4. Store the result into the 3rd memory location Z.
5. The mathematical representation is $Z \leftarrow [A] + [B]$.
   a) Three address instruction format : Its format is as follows

opcode	Source-1	Source-2	destination

Destination ← [source-1] + [source-2]

Ex: ADD A, B, Z

$Z \leftarrow [A] + [B]$

a) Two address instruction format : Its format is as follows

opcode	**Source**	**Source/destination**

Destination ← [source] + [destination]

The sequence of two address m/c instructions to evaluate the arithmetic expression Z ← A + B are as follows

MOV A, R_0
MOV B, R_1
ADD R_0, R_1
MOV R_1, Z

b) One address instruction format : Its format is as follows

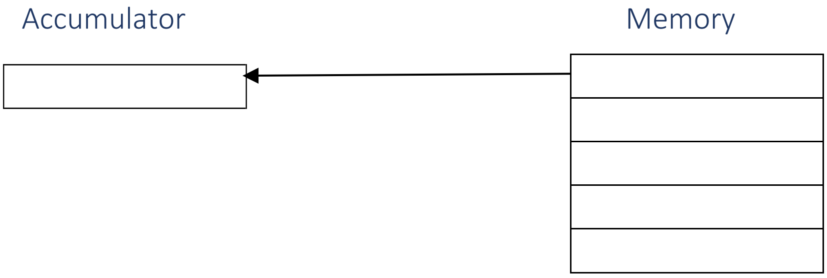
opcode	**operand**

Ex1: LOAD B

This instruction copies the contents of memory location whose symbolic name is given by 'B' into the Accumulator as shown in the figure.

The mathematical representation of this instruction is as follows

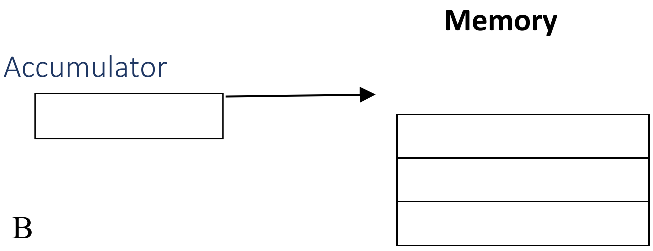
ACC ← [B]



Ex2: STORE B

This instruction copies the contents of Accumulator into memory location whose symbolic name is given by 'B' as shown in the figure. The mathematical representation is as follows

B ← [ACC].



Ex3: ADD B

- This instruction adds the contents of Accumulator with the contents of Memory location 'B' and result is stored in Accumulator.
- The mathematical representation of this instruction is as follows

ACC ←[ACC]+ [B]

STRAIGHT LINE SEQUENCING AND INSTRUCTION EXECUTION

Explain a) Straight Line Sequencing b) Condition code bits

Consider the arithmetic expression

C = A+B , Where A,B,C are the memory operands.

The mathematical representation of this instruction is

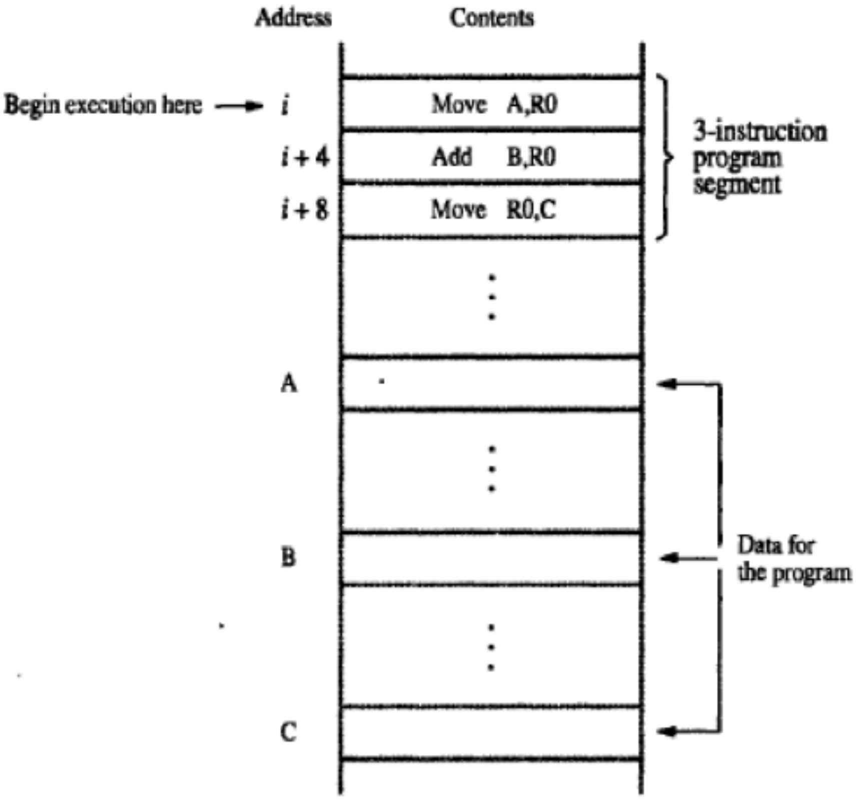
C = [A] + [B].

The sequence of instructions using two address instruction format are as follows

```
MOV    A,    R0
ADD    B,    R0
MOV    R0,   C
```

Such a program is called as 3 instruction program.

NOTE: The size of each instruction is 32 bits.



CONDITION CODE BITS

- The processor consists of series of flip-flops to store the status information after ALU operation.
- It keeps track of the results of various operations, for subsequent usage.
- The series of flip-flip-flops used to store the status and control information of the processor is called as "Condition Code Register". It defines 4 flags. The format of condition code register is as follows

C	V	Z	N

N (NEGATIVE) Flag:

- It is designed to differentiate between positive and negative result.
- It is set 1 if the result is negative, and set to 0 if result is positive.

Z (ZERO) Flag:

It is set to 1 when the result of an ALU operation is found to zero, otherwise it is cleared.

V (OVER FLOW) Flag:

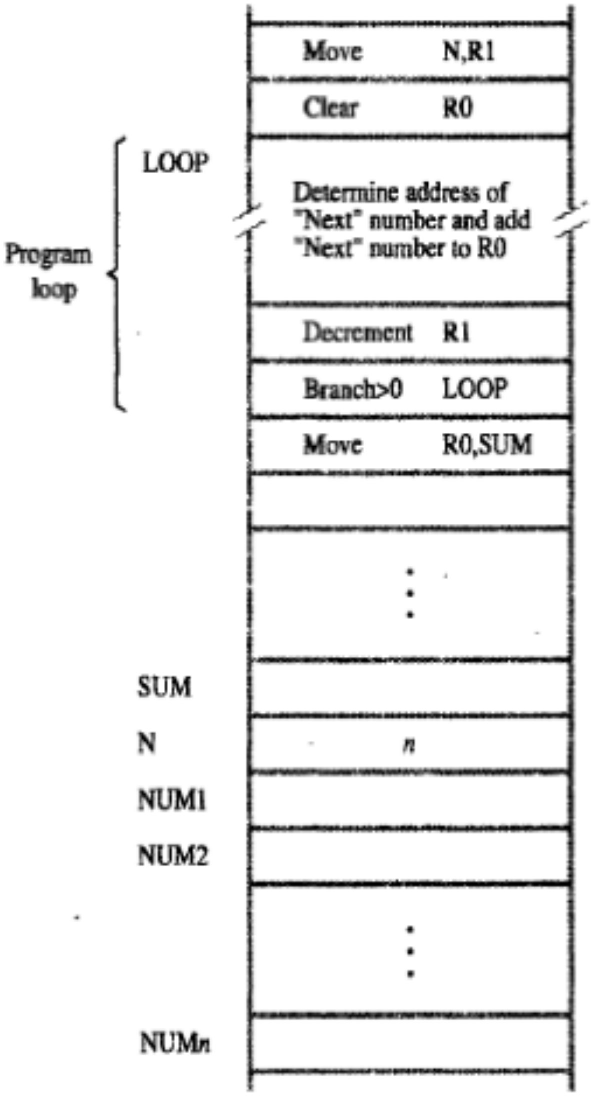
In case of 2^s Complement number system n-bit number is capable of representing a range of numbers and is given by -2^{n-1} to $+2^{n-1.}$. The Over-Flow flag is set to 1 if the resultis found to be out of this range.

C (CARRY) Flag :

This flag is set to 1 if there is a carry from addition or borrow from subtraction, otherwise it is cleared.

BRANCHING

Suppose a list of 'N' numbers have to be added. Instead of adding one after the other, the add statement can be put in a loop. The loop is a straight-line of instructions executed as many times as needed.



The 'N' value is copied to R1 and R1 is decremented by 1 each time in loop. In the loop find the value of next elemet and add it with Ro.

In conditional branch instruction, the loop continues by coming out of sequence only if the condition is true. Here the PC value is set to 'LLOP' if the condition is true.

Branch > 0 LOOP // if >0 go to LOOP

The PC value is set to LOOP, if the previous statement value is >0 ie. after decrementing R1 value is greater than 0.

If R1 value is not greater than 0, the PC value is incremented in a mormal sequential way and the next instruction is executed.

3.8 Addressing Modes

What is addressing mode? Explain all types of addressing modes with suitable examples.

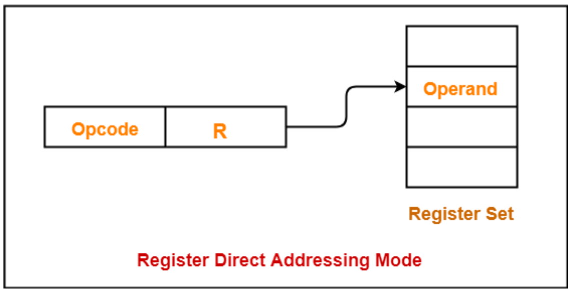
The various formats of representing operand in an instruction or location of an operand is called as "Addressing Mode". The different types of Addressing Modes are

a) Register Addressing
b) Direct Addressing
c) Immediate Addressing
d) Indirect Addressing
e) Index Addressing
f) Relative Addressing
g) Auto Increment Addressing
h) Auto Decrement Addressing

a. REGISTER ADDRESSING:

In this mode operands are stored in the registers of CPU. The name of the register is directly specified in the instruction.

Ex: MOVE R_1,R_2 Where R1 and R2 are the Source and Destination registers respectively. This instruction transfers 32 bits of data from R1 register into R2 register. This instruction does not refer memory for operands. The operands are directly available in the registers.



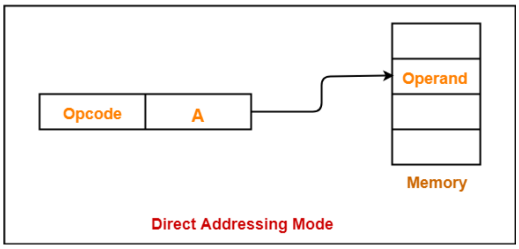
Register Direct Addressing Mode

b. DIRECT ADDRESSING

It is also called as Absolute Addressing Mode. In this addressing mode operands are stored in the memory locations. The name of the memory location is directly specified in the instruction.

Ex: MOVE LOCA, R_1 : Where LOCA is the memory location and R1 is the Register.

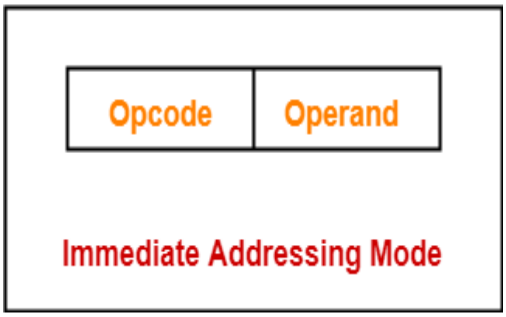
This instruction transfers 32 bits of data from memory location X into the General Purpose Register R1.



c. IMMEDIATE ADDRESSING

In this Addressing Mode operands are directly specified in the instruction. The source field is used to represent the operands. The operands are represented by # (hash) sign.

Ex: MOVE #23, R0



d. INDIRECT ADDRESSING

In this Addressing Mode effective address of an operand is stored in the memory location or General Purpose Register.

The memory locations or GPRs are used as the memory pointers.

Memory pointer: It stores the address of the memory location.

There are two types Indirect Addressing

i) Indirect through GPRs

ii) Indirect through memory location

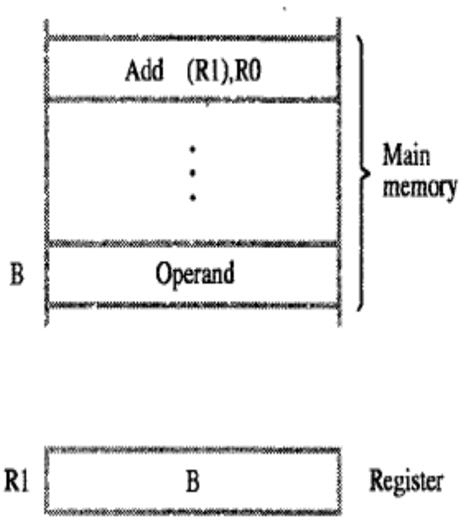
i) Indirect Addressing Mode through GPRs

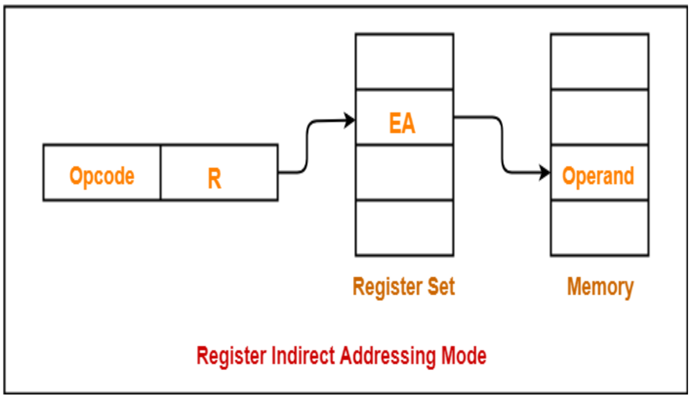
In this Addressing Mode the effective address of an operand is stored in the one of the General Purpose Register of the CPU.

Ex: ADD (R$_1$), R$_0$; Where R$_1$ and R$_0$ are GPRs

This instruction adds the data from the memory location whose address is stored in R$_1$ with the contents of R$_0$ Register and the result is stored in R$_0$ register as shown in the fig.

The diagrammatic representation of this addressing mode is as shown in the



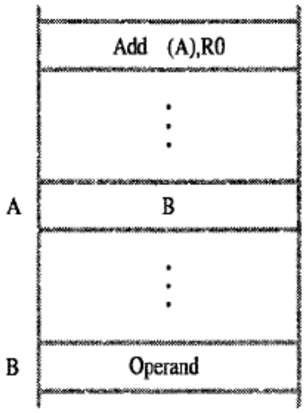


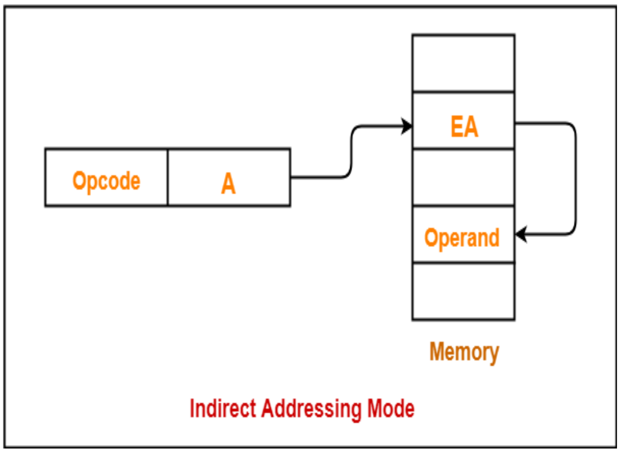
ii) Indirect Addressing Mode through Memory Location.

In this Addressing Mode, effective address of an operand is stored in the memory location.Ex: ADD (X), R$_0$

This instruction adds the data from the memory location whose address is stored in 'X' memory location with the contents of R$_0$ and result is stored in R$_0$ register.

The diagrammatic representation of this addressing mode is as shown in the fig.





e. INDEX ADDRESSING MODE

In this addressing mode, the effective address of an operand is computed by adding constant value with the contents of Index Register and any one of the General Purpose Register namely R_0 to R_{n-1} can be used as the Index Register. The constant value is directly specified in the instruction.

The symbolic representations of this mode are as follows

1. X (R_i) where X is the Constant value and R_j is the GPR.

   It can be represented as

   EA of an operand = X + (R_i)

2. (R_i , R_J) Where R_i and R_j are the General Purpose Registers used to store addresses of an operand and constant value respectively. It can be represented as

   The EA of an operand is given by

   EA = (R_i) + (R_j)

3. X (R_i , R_j) Where X is the constant value and R_I and R_J are the General Purpose Registers used to store the addresses of the operands.It can be represented as

   The EA of an operand is given by

   EA = (R_i) + (R_j) + X

   There are two types of Index Addressing Modes

i) Offset is given as constant.

ii) Offset is in Index Register.

Note : Offset : It is the difference between the starting effective address of the memory location and the effective address of the operand fetched from memory.

i) Offset is given as constant

Ex: ADD 20(R_1), R_2

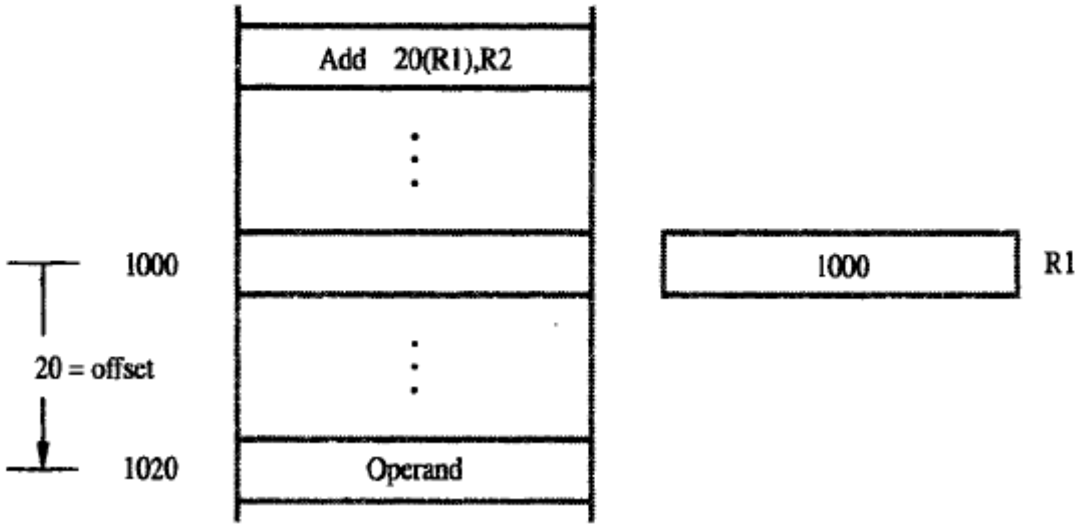
The EA of an operand is given by

$$EA = 20 + [R_1]$$

This instruction adds the data from the memory location whose address is given by [1000 + [R1] with the contents of R_2 and result is placed in R_2 register.

This instruction adds the contents of memory location whose EA is the sum of contents of R_1 with 20 and with the contents of R_2 and result is placed in R_2 register. The

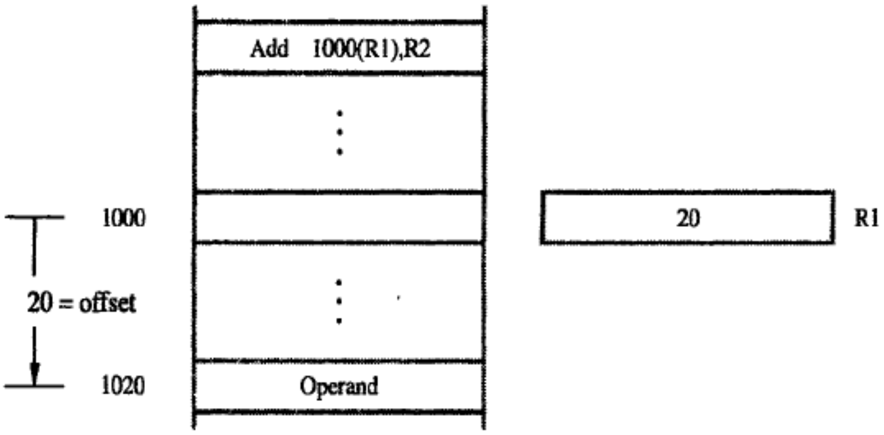
The diagrammatic representation of this mode is as shown in the fig.



ii) Offset is in Index Register

Ex: ADD 1000(R_1) , R_2 R_1 holds the offset address of an operand.The EA of an operand is given by

$EA = 1000 + [R_1]$



f. RELATIVE ADDRESSING MODE:

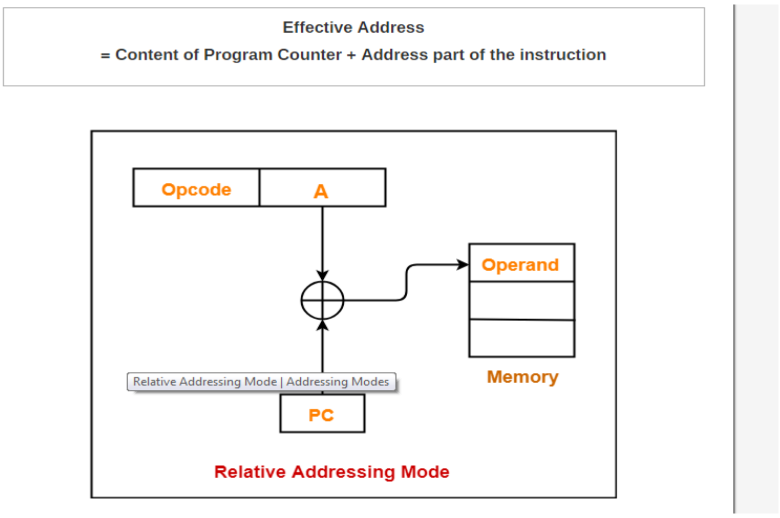
In this Addressing Mode EA of an operand is computed by the Index Addressing Mode. This Addressing Mode uses PC (Program Counter)to store the EA of the next instruction instead of GPR.

The symbolic representation of this mode is X (PC).Where X is the offset value and PC is the Program Counter to store the address of the next instruction to be executed.

It can be represented as

EA of an operand = X + (PC).

This Addressing Mode is useful to calculate the EA of the target memory location.



Effective Address
= Content of Program Counter + Address part of the instruction

Relative Addressing Mode

g. AUTO INCREMENT ADDRESSING MODE

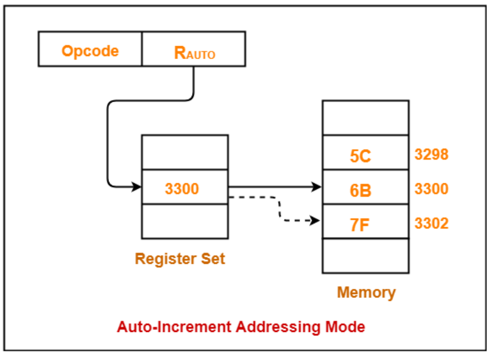
In this Addressing Mode , EA of an operand is stored in the one of the GPR^s of the CPU. This Addressing Mode increment the contents of memory register by 4 memory locations after operand access.

The symbolic representation is

$(R_I)+$ Where R_i is the one of the GPR. Ex:

MOVE $(R_1)+$, R_2

This instruction transfer's data from the memory location whose address is stored in R_1 into R_3 register and then it increments the contents of R_1 by 4 memory locations.



Auto-Increment Addressing Mode

h. AUTO DECREMENT ADDRESSING MODE

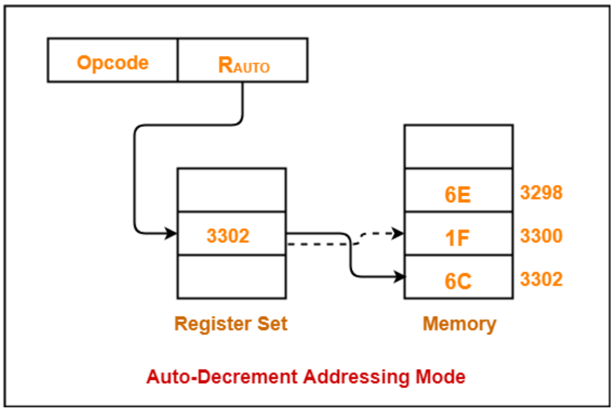
In this Addressing Mode , EA of an operand is stored in the one of the GPR^s of the CPU. This Addressing Mode decrements the contents of memory register by 4 memory locations and then transfers the data to destination.

The symbolic representation is

$-(R_I)$ Where R_i is the one of the GPR.

Ex: MOVE - (R_1) , R_2

This instruction first decrements the contents of R_1 by 4 memory locations and then transfer's data of that location to destination register.



Auto-Decrement Addressing Mode

MODULE 4
INPUT/ OUTPUT Organization

Accessing I/O Devices
Interrupts – Interrupt Hardware
Enabling and Disabling Interrupts
Handling Multiple Devices
Direct Memory Access: Bus Arbitration,
Speed, size and Cost of memory systems.
Cache Memories – Mapping Functions.

INPUT/OUTPUT ORGANIZATION

4.1 ACCESSING I/O-DEVICES

- A **single bus-structure** can be used for connecting I/O-devices to a computer as shown in Fig 4.1.
- Each I/O device is assigned a unique set of address.
- Bus consists of 3 sets of lines to carry address, data & control signals.
- When processor places an address on address-lines, the intended-device responds to the command.
- The processor requests either a read or write-operation.
- The requested-data are transferred over the data-lines.

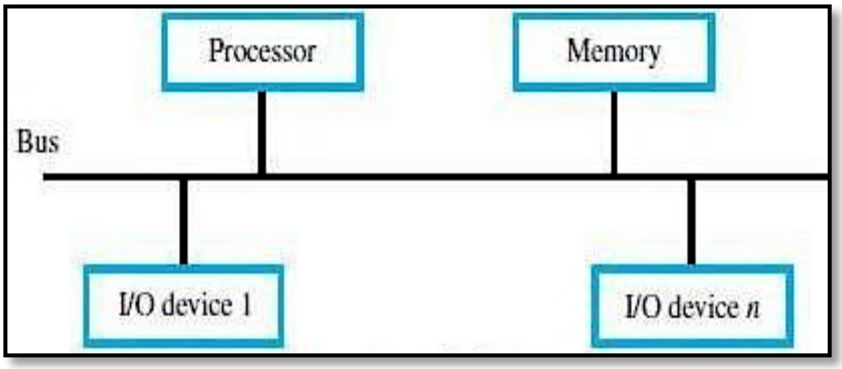


Fig 4.1 Single Bus Structure

There are 2 ways to deal with I/O devices:

- Memory-mapped I/O
- I/O-mapped I/O

4.1.1 Memory-Mapped I/O

- Memory and I/O-devices share a common address-space.
- Any data-transfer instruction (like Move, Load) can be used to exchange information.

  For example,

  Move DATAIN, R0;

  This instruction sends the contents of location DATAIN to register R0.

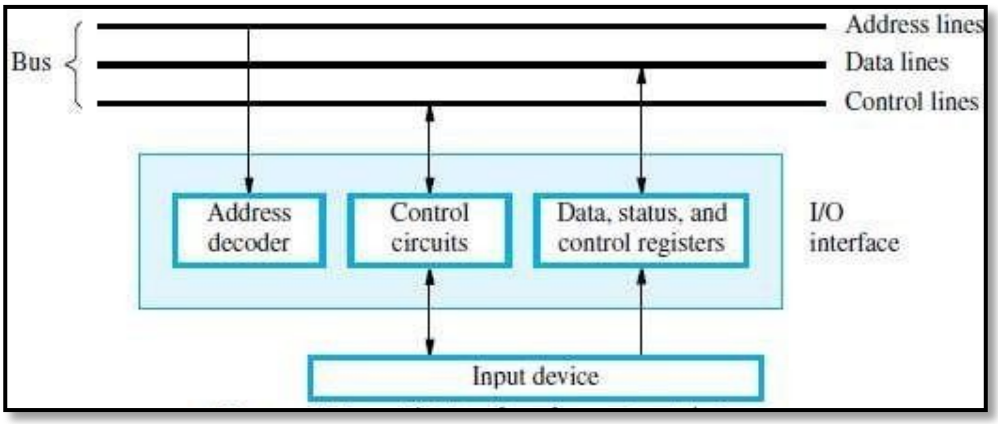
  Here, DATAIN → address of the input-buffer of the keyboard.

4.1.2 I/O-Mapped I/O

- Memory and I/0 address-spaces are different.
- Special instructions named **IN** and **OUT** are used for data-transfer.
- Advantage of separate I/O space: I/O-devices deal with fewer address-lines.

I/O Interface for an Input Device

1) **Address Decoder:** enables the device to recognize its address when this address appears on the address-lines (Figure 4.2).
2) **Status Register:** contains information relevant to operation of I/O-device.
3) **Data Register:** holds data being transferred to or from processor. There are 2 types:
   i) DATAIN - Input-buffer associated with keyboard.
   ii) DATAOUT - Output data buffer of a display/printer.



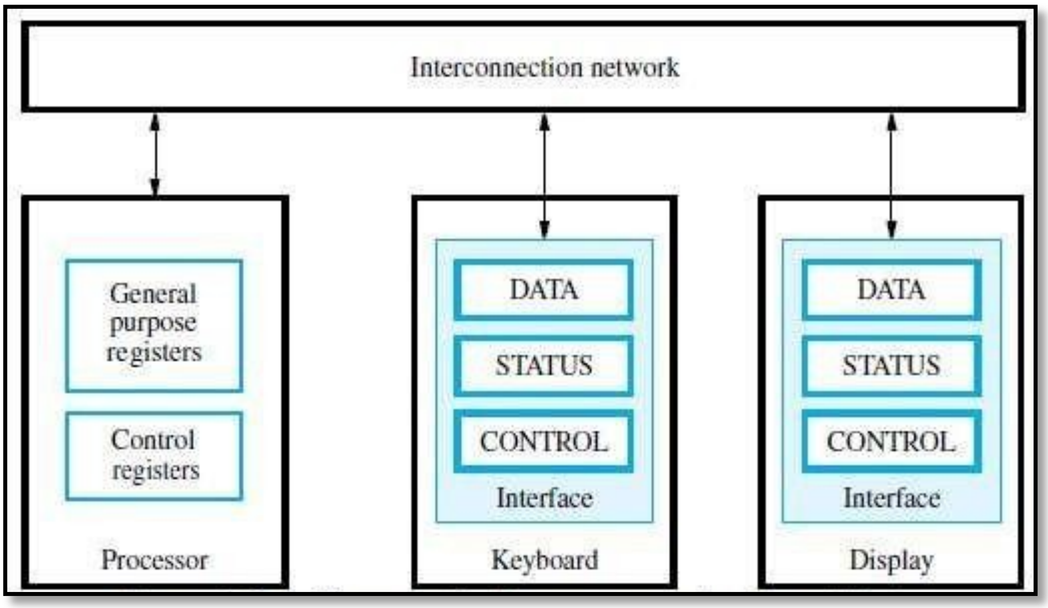


Fig 4.2 Interface for Processor, Keyboard and Display

4.2 MECHANISMS USED FOR INTERFACING I/O-DEVICES

4.2.1 Program Controlled I/O

- Processor repeatedly checks status-flag to achieve required synchronization b/w processor & I/O device. (We say that the processor polls the device).
- Main drawback:

The processor wastes time in checking status of device before actual data-transfer takes place.

4.2.2 Interrupt I/O

- I/O-device initiates the action instead of the processor.
- I/O-device sends an INTR signal over bus whenever it is ready for a data-transfer operation.
- Like this, required synchronization is done between processor & I/O device.

4.2.3 Direct Memory Access (DMA)

- Device-interface transfer data directly to/from the memory w/o continuous involvement by the processor.
- DMA is a technique used for high speed I/O-device.

4.3 INTERRUPTS

- There are many situations where other tasks can be performed while waiting for an I/O device to become ready.
- A hardware signal called an Interrupt will alert the processor when an I/O device becomes ready.
- Interrupt-signal is sent on the interrupt-request line.
- The processor can be performing its own task without the need to continuously check the I/O-device.
- The routine executed in response to an interrupt-request is called ISR.
- The processor must inform the device that its request has been recognized by sending INTA signal. (INTR → Interrupt Request, INTA → Interrupt Acknowledge, ISR → Interrupt Service Routine)

For example, consider COMPUTE and PRINT routines (Figure 4.3).

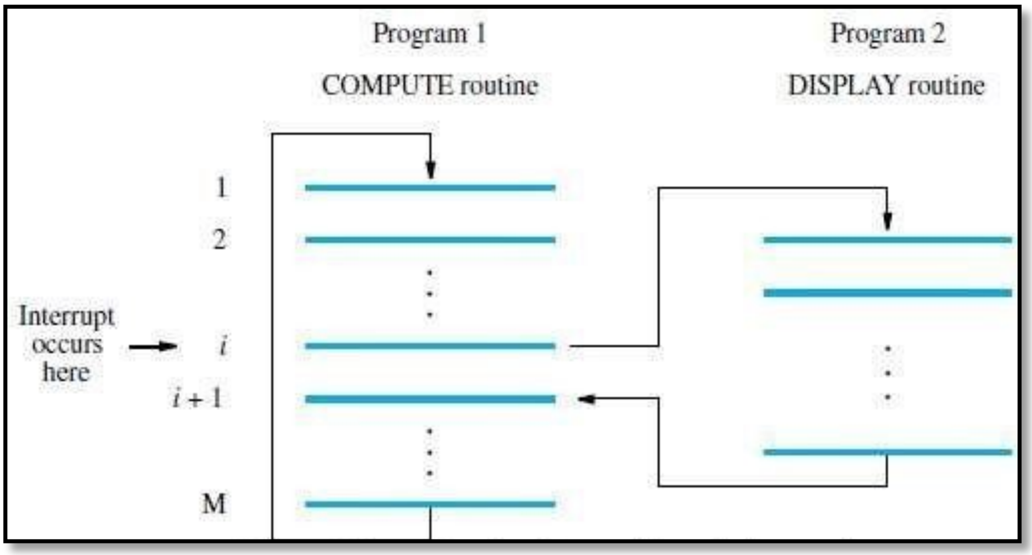


Fig 4.3 Transfer of Control through the use of Interrupts

The sequence of operations is as follows:

- The processor first completes the execution of instruction i.
- Then, processor loads the PC with the address of the first instruction of the ISR.
- After the execution of ISR, the processor has to come back to instruction i+1.
- Therefore, when an interrupt occurs, the current content of PC is put in temporary storage location.
- A return at the end of ISR reloads the PC from that temporary storage location.
- This causes the execution to resume at instruction i+1.
- When processor is handling interrupts, it must inform device that its request has been recognized.
- This may be accomplished by INTA signal.
- The task of saving and restoring the information can be done automatically by the processor.
- The processor saves only the contents of **PC & Status register.**
- Saving registers also increases the Interrupt Latency.

Interrupt Latency - is a delay between

→ time an interrupt-request is received and

→ start of the execution of the ISR.

Difference between Subroutine & ISR

Subroutine	ISR
A subroutine performs a function required by the program from which it is called.	ISR may not have anything in common with program being executed at time INTR is received
Subroutine is just a linkage of 2 or more function related to each other.	Interrupt is a mechanism for coordinating I/O transfers.

4.3.1 INTERRUPT HARDWARE

- Most computers have several I/O devices that can request an interrupt.
- A single interrupt-request (IR) line may be used to serve n devices (Figure 4.4).
- All devices are connected to IR line via switches to ground.
- To request an interrupt, a device closes its associated switch.
- Thus, if all IR signals are inactive, the voltage on the IR line will be equal to Vdd.
- When a device requests an interrupt, the voltage on the line drops to 0.
- This causes the INTR received by the processor to go to 1.
- The value of INTR is the logical OR of the requests from individual devices.

INTR=INTR1+ INTR2+.....+INTRn

- A special gates known as open-collector or open-drain are used to drive the INTR line.
- The Output of the open collector control is equal to a switch to the ground that is

  → open when gates input is in ”0‟ state and

  → closed when the gates input is in “1‟ state.
- Resistor R is called a **Pull-up Resistor** because it pulls the line voltage up to the high-voltage state when the switches are open.

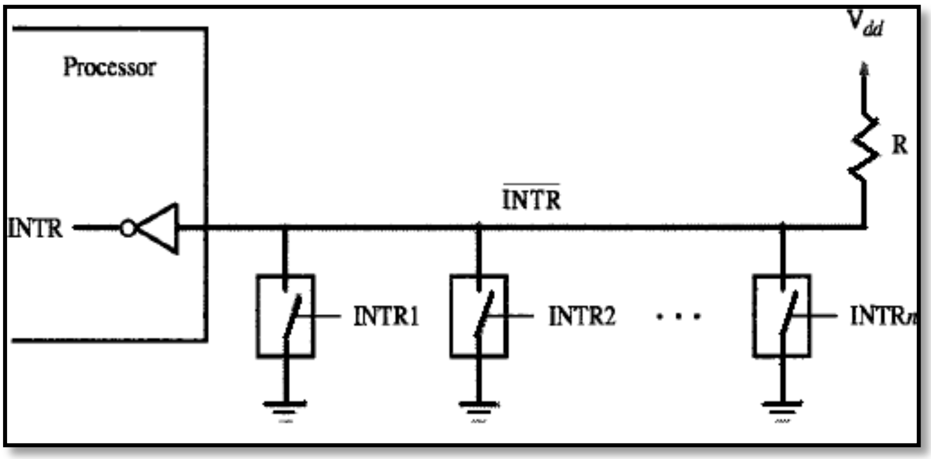


Fig 4.4 An equivalent circuit for an open-drain bus used to implement a common interrupt-request line

4.3.2 ENABLING & DISABLING INTERRUPTS

- All computers fundamentally should be able to enable and disable interruptions as desired.
- The problem of infinite loop occurs due to successive interruptions of active INTR signals.
- There are 3 mechanisms to solve problem of infinite loop:
  1) Processor should ignore the interrupts until execution of first instruction of the ISR.
  2) Processor should automatically disable interrupts before starting the execution of the ISR.
  3) Processor has a special INTR line for which the interrupt-handling circuit.

     Interrupt-circuit responds only to leading edge of signal. Such line is called edge-triggered.
- Sequence of events involved in handling an interrupt-request:
  1) The device raises an interrupt-request.
  2) The processor interrupts the program currently being executed.
  3) Interrupts are disabled by changing the control bits in the processor status register (PS).
  4) The device is informed that its request has been recognized.

     In response, the device deactivates the interrupt-request signal.
  5) The action requested by the interrupt is performed by the interrupt-service routine.
  6) Interrupts are enabled and execution of the interrupted program is resumed.

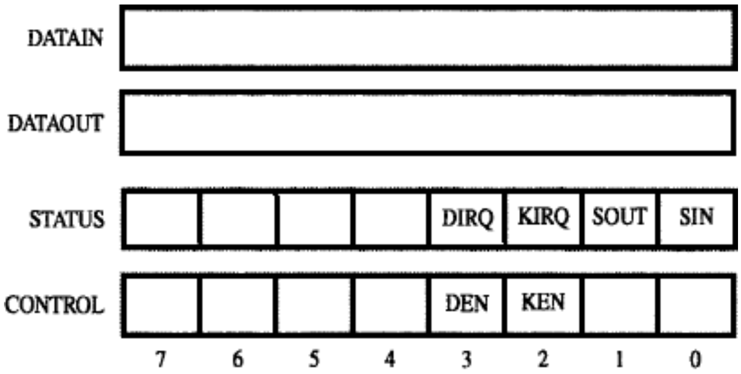
4.3.2 HANDLING MULTIPLE DEVICES

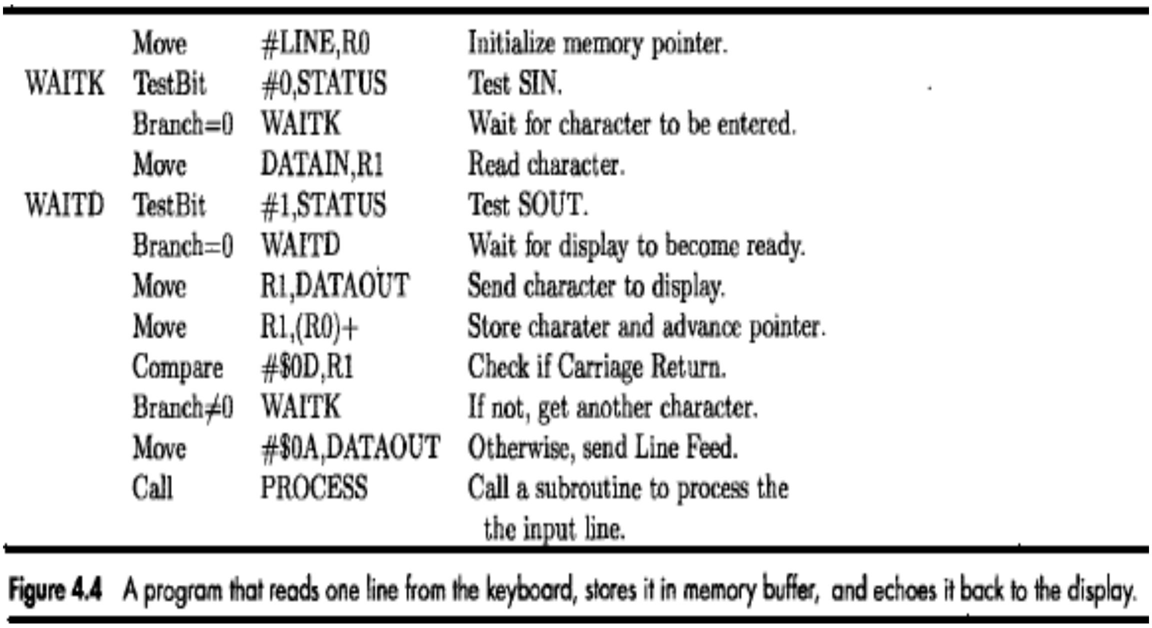
While handling multiple devices, the issues concerned are:

- How can the processor recognize the device requesting an interrupt?
- How can the processor obtain the starting address of the appropriate ISR?
- Should a device be allowed to interrupt the processor while another interrupt is being serviced?
- How should 2 or more simultaneous interrupt-requests be handled?

4.3.2 a) POLLING

- Information needed to determine whether device is requesting interrupt is available in status-register
- Following condition-codes are used:
  - ➢ DIRQ → Interrupt-request for display.
  - ➢ KIRQ → Interrupt-request for keyboard.
  - ➢ KEN → keyboard enable.
  - ➢ DEN → Display Enable.
  - ➢ SIN, SOUT → status flags.
- For an input device, SIN status flag in used.
  - ➢ SIN = 1 → when a character is entered at the keyboard.
  - ➢ SIN = 0 → when the character is read by processor.
  - ➢ IRQ=1 → when a device raises an interrupt-requests (Figure 4.5).
- Simplest way to identify interrupting-device is to have ISR poll all devices connected to bus.
- The first device encountered with its IRQ bit set is serviced.
- After servicing first device, next requests may be serviced.
- **Advantage:** Simple & easy to implement.
- **Disadvantage:** More time spent polling IRQ bits of all devices.





```
         Move       #LINE,R0        Initialize memory pointer.
WAITK    TestBit    #0,STATUS       Test SIN.
         Branch=0   WAITK           Wait for character to be entered.
         Move       DATAIN,R1       Read character.
WAITD    TestBit    #1,STATUS       Test SOUT.
         Branch=0   WAITD           Wait for display to become ready.
         Move       R1,DATAOUT      Send character to display.
         Move       R1,(R0)+        Store charater and advance pointer.
         Compare    #$0D,R1         Check if Carriage Return.
         Branch≠0   WAITK           If not, get another character.
         Move       #$0A,DATAOUT    Otherwise, send Line Feed.
         Call       PROCESS         Call a subroutine to process the
                                      the input line.
```

Figure 4.4 A program that reads one line from the keyboard, stores it in memory buffer, and echoes it back to the display.

Fig 4.5 a) Registers of Keyboard and display interfaces
b) A program that reads one line from the keyboard, stores it in memory buffer and echoes it back to the display

4.3.3 VECTORED INTERRUPTS

- A device requesting an interrupt identifies itself by sending a special-code to processor over bus.
- Then, the processor starts executing the ISR.
- The special-code indicates starting-address of ISR.
- The special-code length ranges from 4 to 8 bits.
- The location pointed to by the interrupting-device is used to store the staring address to ISR.
- The staring address to ISR is called the **interrupt vector**.
- Processor

  → loads interrupt-vector into PC &

→ executes appropriate ISR.

- When processor is ready to receive interrupt-vector code, it activates INTA line.
- Then, I/O-device responds by sending its interrupt-vector code & turning off the INTR signal.
- The interrupt vector also includes a new value for the Processor Status Register.

4.3.3 a) CONTROLLING DEVICE REQUESTS

Following condition-codes are used:

- KEN → Keyboard Interrupt Enable.
- DEN → Display Interrupt Enable.
- KIRQ/DIRQ → Keyboard/Display unit requesting an interrupt.

There are 2 independent methods for controlling interrupt-requests. (IE → interrupt-enable).

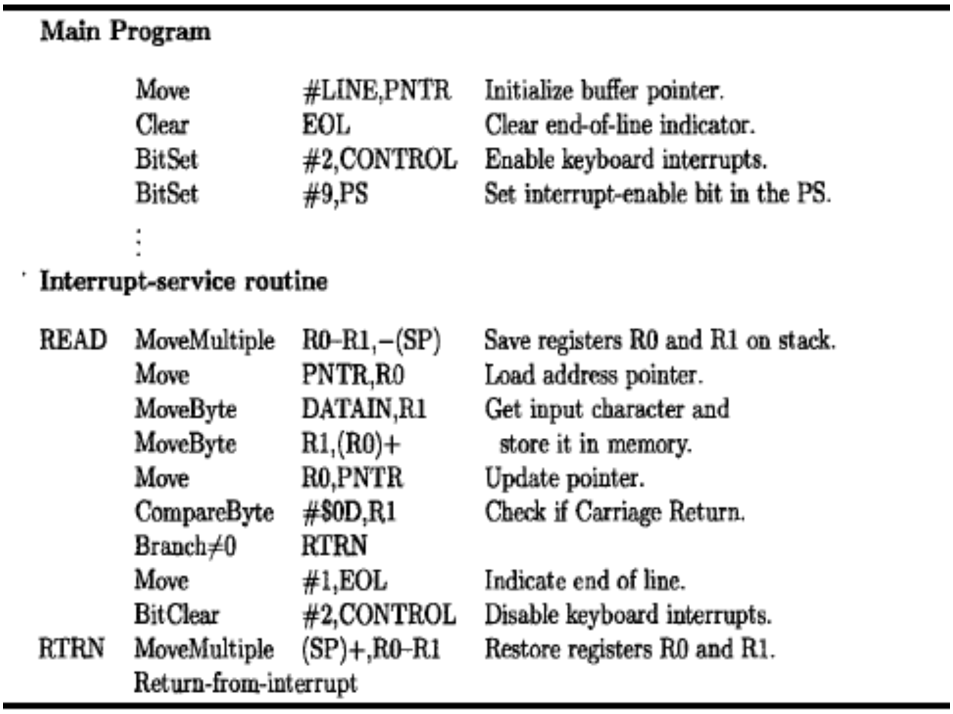
1) ***At Device-end***

IE bit in a control-register determines whether device is allowed to generate an interrupt-request.

2) **At Processor-end**, interrupt-request is determined by

→ IE bit in the PS register or

→ Priority structure



```
Main Program

        Move           #LINE,PNTR     Initialize buffer pointer.
        Clear          EOL            Clear end-of-line indicator.
        BitSet         #2,CONTROL     Enable keyboard interrupts.
        BitSet         #9,PS          Set interrupt-enable bit in the PS.
        :

Interrupt-service routine

READ    MoveMultiple   R0-R1,-(SP)    Save registers R0 and R1 on stack.
        Move           PNTR,R0        Load address pointer.
        MoveByte       DATAIN,R1      Get input character and
        MoveByte       R1,(R0)+         store it in memory.
        Move           R0,PNTR        Update pointer.
        CompareByte    #$0D,R1        Check if Carriage Return.
        Branch≠0       RTRN
        Move           #1,EOL         Indicate end of line.
        BitClear       #2,CONTROL     Disable keyboard interrupts.
RTRN    MoveMultiple   (SP)+,R0-R1    Restore registers R0 and R1.
        Return-from-interrupt
```

Figure 4.9 Using interrupts to read a line of characters from a keyboard via the registers in Figure 4.3.

4.3.4 INTERRUPT NESTING

- A multiple-priority scheme is implemented by using separate INTR & INTA lines for each device
- Each INTR line is assigned a different priority-level (Figure 4.6).
- Priority-level of processor is the priority of program that is currently being executed.
- Processor accepts interrupts only from devices that have higher-priority than its own.
- At the time of execution of ISR for some device, priority of processor is raised to that of the device.
- Thus, interrupts from devices at the same level of priority or lower are disabled.

Privileged Instruction

- Processor's priority is encoded in a few bits of PS word. (PS → Processor-Status).
- Encoded-bits can be changed by **Privileged Instructions** that write into PS.
- Privileged-instructions can be executed only while processor is running in **Supervisor Mode**.
- Processor is in supervisor-mode only when executing operating-system routines.

Privileged Exception

- User program cannot

  → accidently or intentionally change the priority of the processor &

  → disrupt the system-operation.
- An attempt to execute a privileged-instruction while in user-mode leads to a **Privileged Exception**.

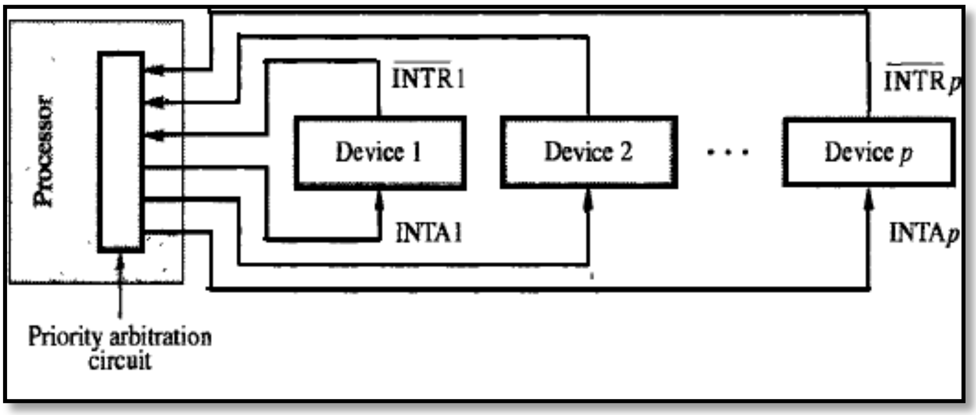

Processor
$\overline{\text{INTR}}$1
$\overline{\text{INTR}}p$
Device 1
Device 2
Device p
INTA1
INTAp
Priority arbitration circuit

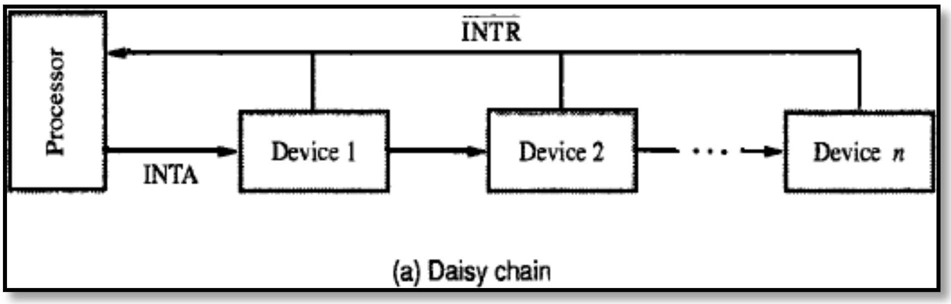
Fig 4.6 Implementation of interrupt priority using individual interrupt-request and acknowledge lines

4.3.5 SIMULTANEOUS REQUESTS

- The processor must have some mechanisms to decide which request to service when simultaneous requests arrive.
- INTR line is common to all devices (Figure 4.7a).
- INTA line is connected in a daisy-chain fashion.
- INTA signal propagates serially through devices.
- When several devices raise an interrupt-request, INTR line is activated.
- Processor responds by setting INTA line to 1. This signal is received by device 1.
- Device-1 passes signal on to device 2 only if it does not require any service.
- If device-1 has a pending-request for interrupt, the device-1

  → blocks INTA signal &

  → proceeds to put its identifying-code on data-lines.
- Device that is electrically closest to processor has highest priority
- **Advantage:** It requires fewer wires than the individual connections.
- **Arrangement of Priority Groups**
  - Here, the devices are organized in groups & each group is connected at a different priority level.
  - Within a group, devices are connected in a daisy chain.



(a) Daisy chain

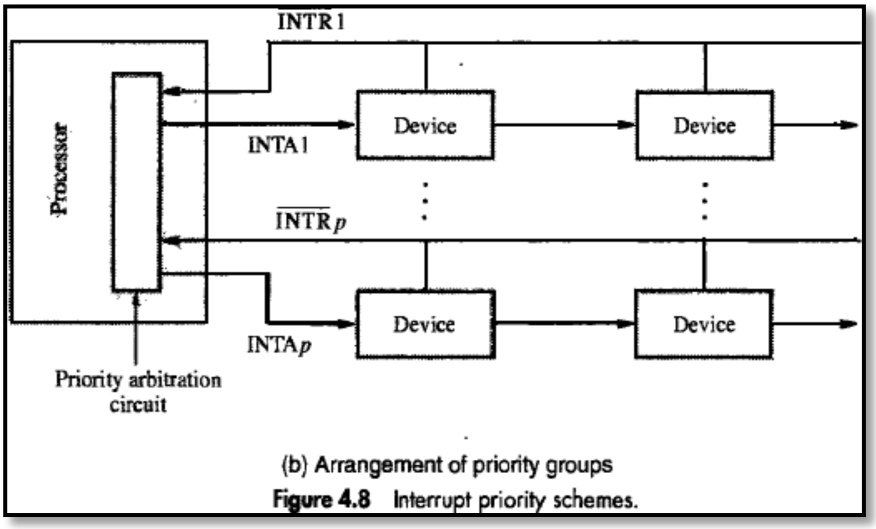

INTR 1
Processor
INTA 1
Device
Device
$\overline{\text{INTR}}p$
INTAp
Device
Device
Priority arbitration circuit
(b) Arrangement of priority groups
Figure 4.8 Interrupt priority schemes.

Fig 4.7 a and b

4.4 EXCEPTIONS

An **interrupt** is an event that causes

→ execution of one program to be suspended &

→ execution of another program to begin.

Exception refers to any event that causes an interruption. For ex: I/O interrupts.

4.4.1 Recovery from Errors

- These are techniques to ensure that all hardware components are operating properly.
- For ex: Many computers include an ECC in memory which allows detection of errors in stored-data. (ECC → Error Checking Code, ESR → Exception Service Routine).
- If an error occurs, control-hardware

  → detects the errors &

  → informs processor by raising an interrupt.
- When exception processing is initiated (as a result of errors), processor.

  → suspends program being executed &

  → starts an ESR. This routine takes appropriate action to recover from the error.

4.4.2 Debugging

Debugger

→ is used to find errors in a program and

→ uses exceptions to provide 2 important facilities: i) Trace & ii) Breakpoints

i) ***Trace***

- When a processor is operating in trace-mode, an exception occurs after execution of every instruction (using debugging-program as ESR).
- Debugging-program enables user to examine contents of registers, memory-locations and so on.
- On return from debugging-program, next instruction in program being debugged is executed, then debugging-program is activated again.
- The trace exception is disabled during the execution of the debugging-program.

ii) ***Breakpoints***

- Here, the program being debugged is interrupted only at specific points selected by user.
- An instruction called Trap (or Software interrupt) is usually provided for this purpose.
- When program is executed & reaches breakpoint, the user can examine memory & register contents.

iii) Privilege Exception

- To protect OS from being corrupted by user-programs, **Privileged Instructions** are executed only while processor is in supervisor-mode.
- For e.g. When processor runs in user-mode, it will not execute instruction that change priority of processor.
- An attempt to execute privileged-instruction will produce a **Privilege Exception**.
- As a result, processor switches to supervisor-mode & begins to execute an appropriate routine in OS.

4.5 DIRECT MEMORY ACCESS (DMA)

The transfer of a block of data directly b/w an external device & main-memory w/o continuous involvement by processor is called DMA.

DMA controller

→ is a control circuit that performs DMA transfers (Figure 4.8).

→ is a part of the I/O device interface.

→ performs the functions that would normally be carried out by processor.

While a DMA transfer is taking place, the processor can be used to execute another program.

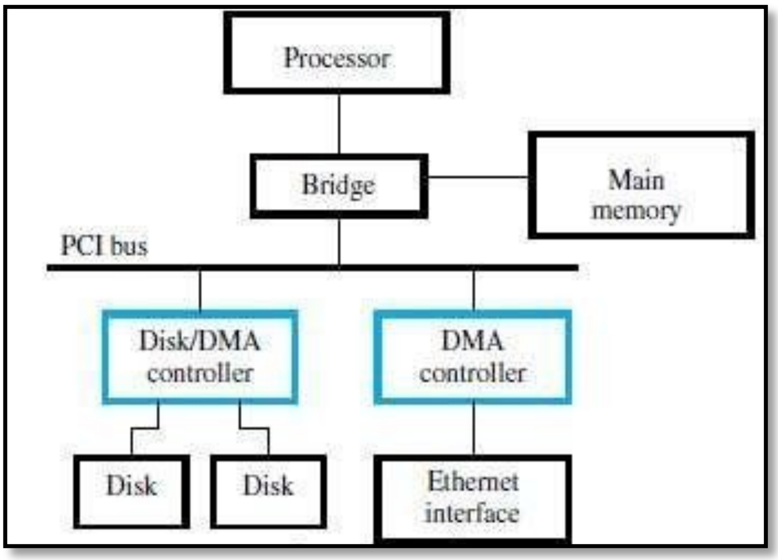


Fig 4.8 Use of DMA Controllers in a computer system

DMA interface has three registers (Figure 4.9):

- First register is used for storing starting-address.
- Second register is used for storing word-count.
- Third register contains status- & control-flags.

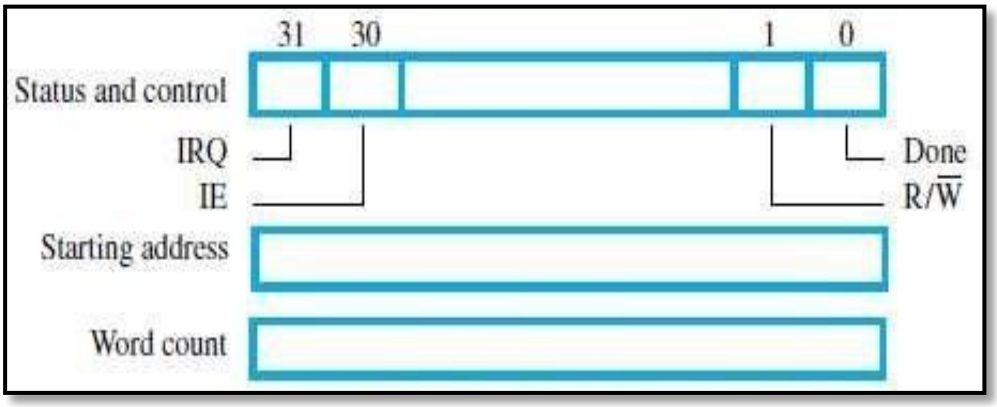


Fig 4.9 Registers in a DMA Controller

- The **R/W** bit determines direction of transfer.

  If R/W=1, controller performs a read-operation (i.e. it transfers data from memory to I/O), Otherwise, controller performs a write-operation (i.e. it transfers data from I/O to memory).

- If **Done=1**, the controller

  → has completed transferring a block of data and

  → is ready to receive another command. (IE Interrupt Enable).

- If **IE=1**, controller raises an interrupt after it has completed transferring a block of data.
- If **IRQ=1**, controller requests an interrupt.
- Requests by DMA devices for using the bus are always given higher priority than processor requests.
- There are 2 ways in which the DMA operation can be carried out:
    - Processor originates most memory-access cycles.
    - DMA controller is said to "steal" memory cycles from processor.
    - Hence, this technique is usually called **Cycle Stealing**.
    - DMA controller is given exclusive access to main-memory to transfer a block of data without any interruption. This is known as **Block Mode** (or burst mode).

MODULE 5
Basic Processing Unit

Some Fundamental Concepts: Register Transfers

Performing ALU operations

fetching a word from Memory

Storing a word in memory

Execution of a Complete Instruction

Pipelining: Basic concepts

Role of Cache memory

Pipeline Performance

5.1 SOME FUNDAMENTAL CONCEPTS

- To execute an instruction, processor has to perform following 3 steps:

  1) Fetch contents of memory-location pointed to by PC. Content of this location is an instruction to be executed. The instructions are loaded into IR, Symbolically, this operation is written as:

     IR□ [[PC]]

  2) Increment PC by 4.

     PC□ [PC] +4

  3) Carry out the actions specified by instruction (in the IR).

- The first 2 steps are referred to as Fetch Phase.

  Step 3 is referred to as Execution Phase.

- The operation specified by an instruction can be carried out by performing one or more of the following actions:

  1) Read the contents of a given memory-location and load them into a register.
  2) Read data from one or more registers.
  3) Perform an arithmetic or logic operation and place the result into a register.
  4) Store data from a register into a given memory-location.

The hardware-components needed to perform these actions are shown in Figure 5.1

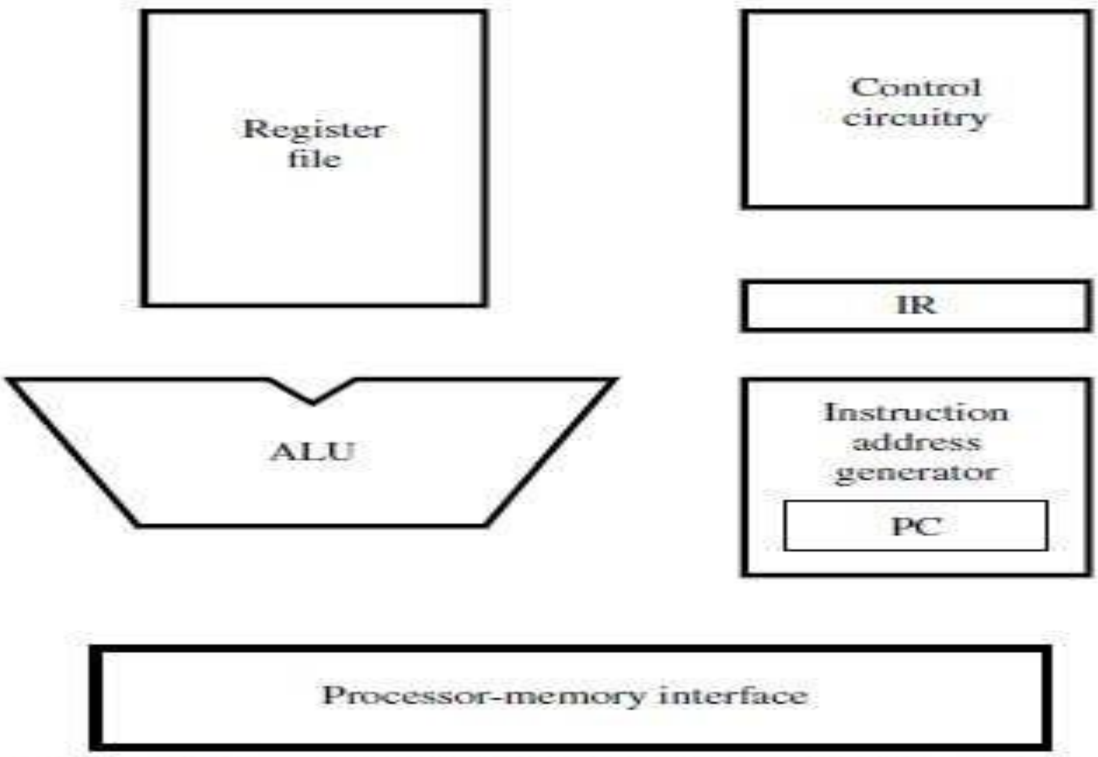


Figure 5.1 Main hardware components of a processor.

SINGLE BUS ORGANIZATION

- ALU and all the registers are interconnected via a **Single Common Bus** (Figure 7.1).
- Data & address lines of the external memory-bus is connected to the internal processor-bus via MDR & MAR respectively. (MDR□ Memory Data Register, MAR □ Memory Address Register).
- **MDR** has 2 inputs and 2 outputs. Data may be loaded

  → into MDR either from memory-bus (external) or

  → from processor-bus (internal).
- **MAR‟**s input is connected to

  internal-bus; MAR‟s

  output is connected to

  external- bus.
- **Instruction Decoder & Control Unit** is responsible for

  → issuing the control-signals to all the units inside the processor.

  → implementing the actions specified by the instruction (loaded in the IR).
- Register R0 through R(n-1) are the **Processor Registers**.

  The programmer can access these registers for general-purpose use.
- Only processor can access 3 registers **Y**, **Z** & **Temp** for temporary storage during program-execution. The programmer cannot access these 3 registers.
- In **ALU**, 1) „A‟ input gets the operand from the output of the multiplexer(MUX).

  2) „B‟ input gets the operand directly from the processor-bus.
- There are 2 options provided for „A‟ input of the ALU.
- MUX is used to select one of the 2 inputs.
- **MUX** selects either

  → output of Y or

→ constant-value 4(which is used to increment PC content).

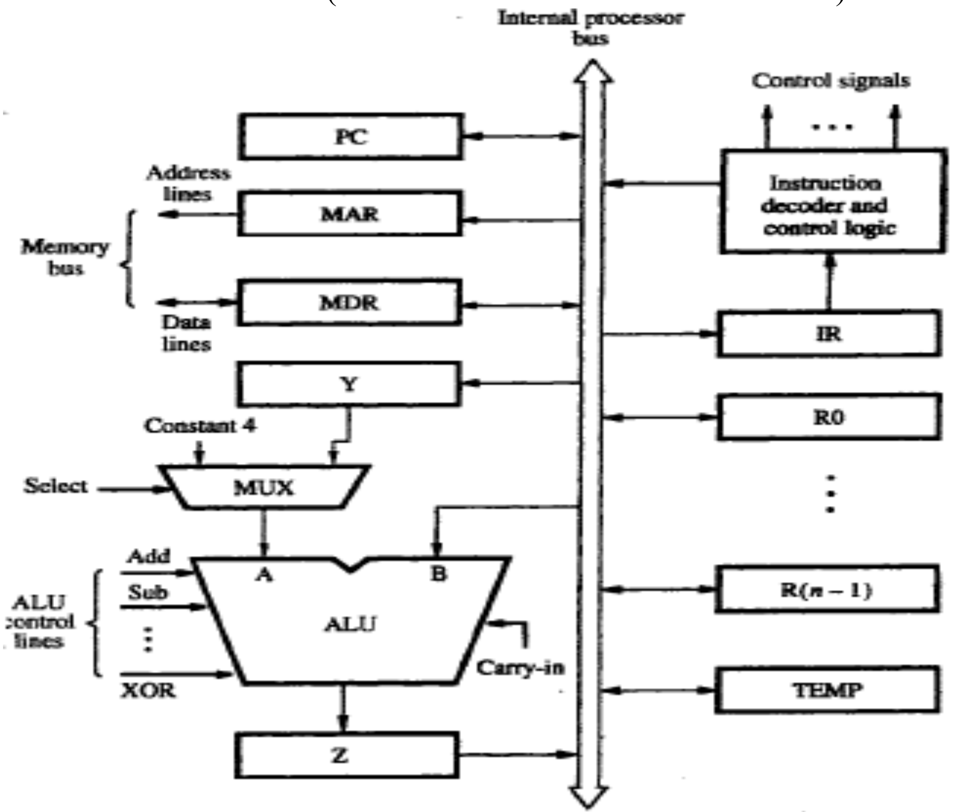


Figure 7.1 Single-bus organization of the datapath inside a processor.

- An instruction is executed by performing one or more of the following operations:
  1) Transfer a word of data from one register to another or to the ALU.
  2) Perform arithmetic or a logic operation and store the result in a register.
  3) Fetch the contents of a given memory-location and load them into a register.
  4) Store a word of data from a register into a given memory-location.
- **Disadvantage:** Only one data-word can be transferred over the bus in a clock cycle.

  Solution: Provide multiple internal-paths. Multiple paths allow several data-transfers to take place in parallel

5.2 REGISTER TRANSFERS

- Instruction execution involves a sequence of steps in which data are transferred from one register to another.
- For each register, two control-signals are used: Riin & Riout. These are called **Gating Signals.**
- Riin=1 data on bus is loaded into Ri. Riout=1

   content of Ri is placed onbus.

  Riout=0, bus can be used for transferring data from other registers.

For example, *Move R1, R2;* This transfers the contents of register R1 to register R2.

- This can be accomplished as follows:

  1) Enable the output of registers R1 by setting R1out to 1 (Figure 7.2). This places the contents of R1 on processor-bus.

  2) Enable the input of register R2 by setting R2out to 1. This loads data from processor-bus into register R4.

- All operations and data transfers within the processor take place within time-periods defined by the

Processor clock.

- The control-signals that govern a particular transfer are asserted at the start of the clock cycle.

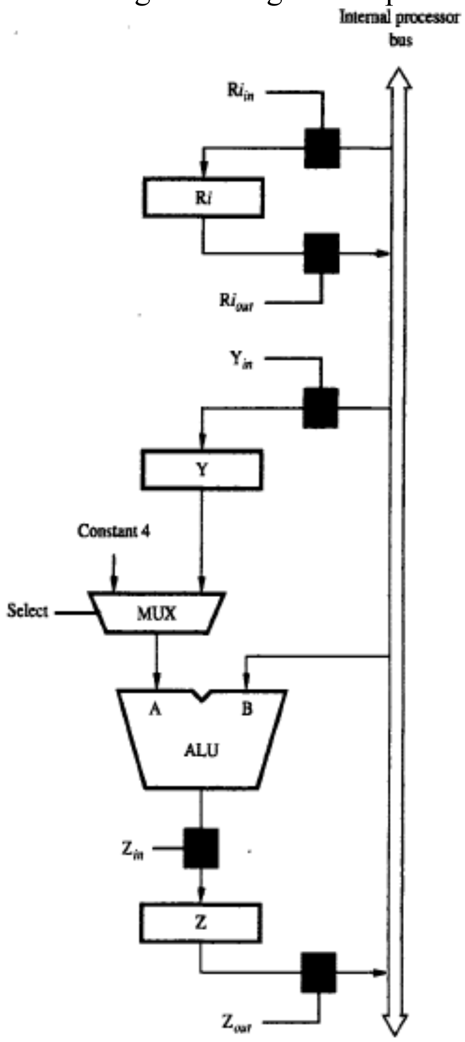


Figure 7.2 Input and output gating for the registers in Figure 7.1.

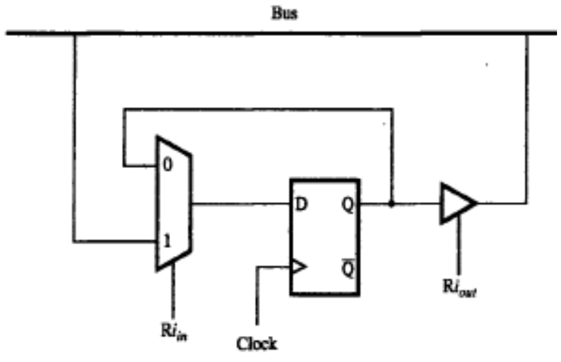


Figure 7.3 Input and output gating for one register bit.

Input & Output Gating for one Register Bit

- A 2-input multiplexer is used to select the data applied to the input of an edge-triggered D flip-flop.
- Riin=1 □ mux selects data on bus. This data will be loaded into flip-flop at rising-
  edge of clock. Riin=0 □ mux feeds back the value currently stored in flip-flop
  (Figure7.3).
- Q output of flip-flop is connected to bus via a tri-
  state gate. Riout=0 □ gate's output is in the
  high-impedancestate.
  Riout=1 □ the gate drives the bus to 0 or 1, depending on the value of Q.

5.3 PERFORMING AN ARITHMETIC OR LOGIC OPERATION

- The ALU performs arithmetic operations on the 2 operands applied to its A and B inputs.
- One of the operands is output of MUX;
  And, the other operand is obtained directly from processor-bus.
- The result (produced by the ALU) is stored temporarily in register Z.
- The sequence of operations for [R3]□[R1]+[R2] is as follows:
  1) R1out, Yin
  2) R2out, SelectY, Add, Zin
  3) Zout, R3in
- Instruction execution proceeds as follows:
  Step 1 --> Contents from register R1 are loaded into register Y.
  Step2 --> Contents from Y and from register R2 are applied to the A and B inputs of
  ALU; Addition is performed &
  Result is stored in the Z register.
  Step 3 --> The contents of Z register is stored in the R3 register.
- The signals are activated for the duration of the clock cycle corresponding to that step.

All other signals are inactive.

CONTROL-SIGNALS OF MDR

- The MDR register has 4 control-signals (Figure 7.4):
  1) MDRin & MDRout control the connection to the internal processor data bus &
  2) MDRinE & MDRoutE control the connection to the memory Data bus.
- MAR register has 2 control-signals.
  1) MARin controls the connection to the internal processor address bus &
  2) MARout controls the connection to the memory address bus.

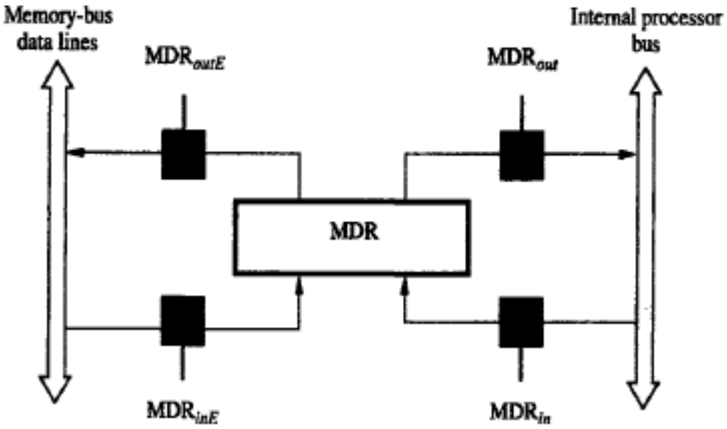


Figure 7.4 Connection and control signals for register MDR.

5.5 FETCHING A WORD FROM MEMORY

- To fetch instruction/data from memory, processor transfers required address to MAR. At the same time, processor issues Read signal on control-lines of memory-bus.
- When requested-data are received from memory, they are stored in MDR. From MDR, they are transferred to other registers.
- The response time of each memory access varies (based on cache miss, memory-mapped I/O). To accommodate this, MFC is used. (MFC Memory Function Completed).
- MFC is a signal sent from addressed-device to the processor. MFC informs the processor that the requested operation has been completed by addressed-device.
- Consider the instruction Move (R1),R2. The sequence of steps is (Figure 7.5):

1) R1out, MARin, Read ;desired address is loaded into MAR & Read command is issued.
2) MDRinE, WMFC ;load MDR from memory-bus & Wait for MFC response from memory.
3) MDRout, R2in ;load R2 from MDR.

where WMFC=control-signal that causes processor's control. circuitry to wait for arrival of MFC signal.

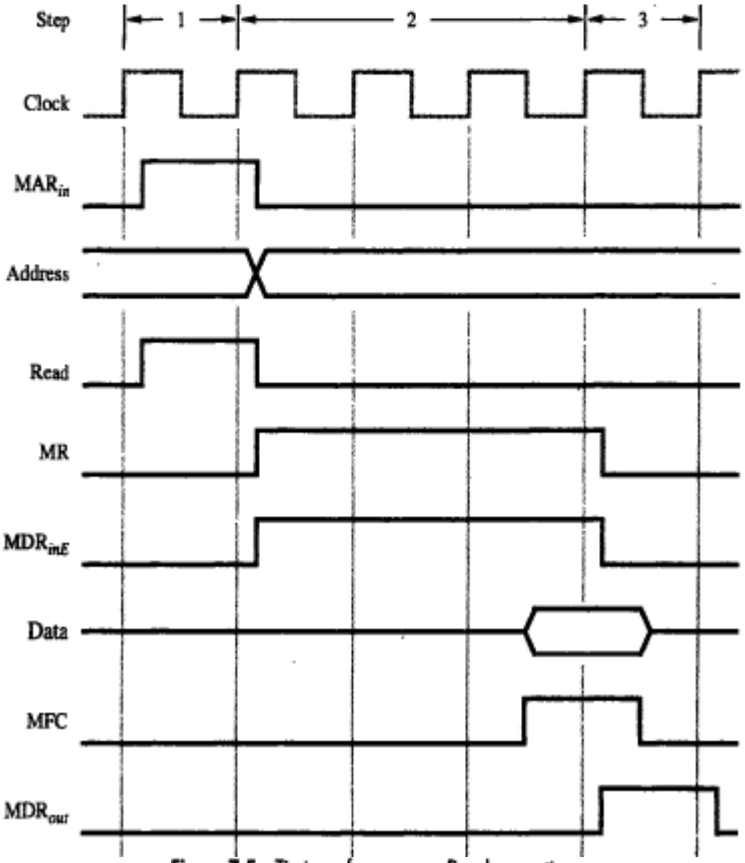


Figure 7.5 Timing of a memory Read operation.

Storing a Word in Memory

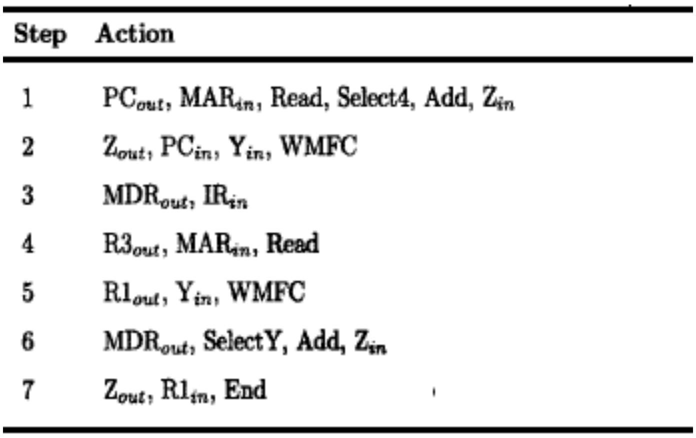
- Consider the instruction *Move R2,(R1).* This requires the following sequence:

  1) R1out, MARin ;desired address is loaded into MAR.
  2) R2out, MDRin, Write ;data to be written are loaded into MDR & Write command is issued.
  3) MDRoutE, WMFC ;load data into memory-location pointed by R1 from MDR.

EXECUTION OF A COMPLETE INSTRUCTION

- Consider the instruction *Add (R3),R1* which adds the contents of a memory-location pointed by R3 to register R1. Executing this instruction requires the following actions:

  1) Fetch the instruction.
  2) Fetch the first operand.
  3) Perform the addition &
  4) Load the result into R1.



Step	Action
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	$R3_{out}$, MAR_{in}, Read
5	$R1_{out}$, Y_{in}, WMFC
6	MDR_{out}, SelectY, Add, Z_{in}
7	Z_{out}, $R1_{in}$, End

Figure 7.6 Control sequence for execution of the instruction Add (R3),R1

- Instruction execution proceeds as follows:

Step1--> The instruction-fetch operation is initiated by

→ loading contents of PC into MAR &

→ sending a Read request to memory.

The Select signal is set to Select4, which causes the Mux to select constant 4. This value is added to operand at input B (PC‟s content), and the result is stored in Z.

Step2--> Updated value in Z is moved to PC. This completes the PC increment operation and PC will now point to next instruction.

Step3--> Fetched instruction is moved into MDR and then to IR. The step 1 through 3 constitutes the **Fetch Phase**.

At the beginning of step 4, the instruction decoder interprets the contents of the IR. This enables the control circuitry to activate the control-signals for steps 4 through 7.

The step 4 through 7 constitutes the **Execution Phase**.

Step4--> Contents of R3 are loaded into MAR & a memory read signal is issued.

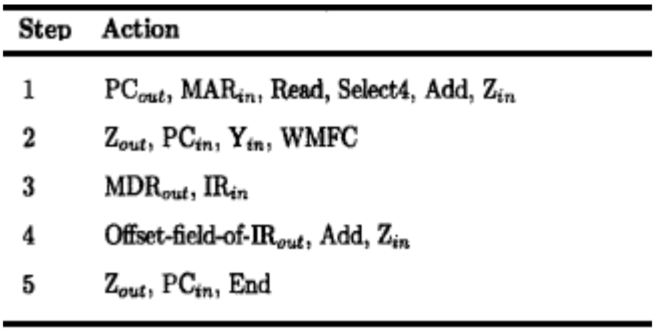
Step5--> Contents of R1 are transferred to Y to prepare for addition.

Step6--> When Read operation is completed, memory-operand is available in MDR, and the addition is performed.

Step7--> Sum is stored in Z, then transferred to R1.The End signal causes a new instruction fetch cycle to begin by returning to step1.

5.6 BRANCHING INSTRUCTIONS

- Control sequence for an **unconditional branch instruction** is as follows:



Step	Action
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	Offset-field-of-IR_{out}, Add, Z_{in}
5	Z_{out}, PC_{in}, End

Figure 7.7 Control sequence for an unconditional Branch instruction.

- Instruction execution proceeds as follows:

Step 1-3--> The processing starts & the fetch phase ends in step3.

Step 4--> The offset-value is extracted from IR by instruction-decoding circuit.

Since the updated value of PC is already available in register Y, the offset X is gated onto the bus, and an addition operation is performed.

Step 5--> the result, which is the branch-address, is loaded into the PC.

- The branch instruction loads the branch target address in PC so that PC will fetch the next instruction from the branch target address.
- The branch target address is usually obtained by adding the offset in the contents of PC.
- The offset X is usually the difference between the branch target-address and the address immediately following the branch instruction.
- In case of **conditional branch**,

we have to check the status of the condition-codes before loading a new value into the PC. e.g.: Offset-field-of-IRout, Add, Zin, If N=0 then End. If N=0, processor returns to step 1 immediately after step 4. If N=1, step 5 is performed to load a new value into PC.