

ANNA UNIVERSITY: CHENNAIREGULATION:2017SYLLABUS  
**CS8391 DATA STRUCTURES****OBJECTIVES:**

1. To understand the concepts of ADTs
2. To Learn linear data structures – lists, stacks, and queues
3. To understand sorting, searching and hashing algorithms
4. To apply Tree and Graph structures

**UNIT I            LINEAR DATA STRUCTURES – LIST**

9

Abstract Data Types (ADTs) – List ADT – array-based implementation – linked list implementation — singly linked lists- circularly linked lists- doubly-linked lists – applications of lists –Polynomial Manipulation – All operations (Insertion, Deletion, Merge, Traversal).

**UNIT II            LINEAR DATA STRUCTURES – STACKS, QUEUES**

9

Stack ADT – Operations - Applications - Evaluating arithmetic expressions- Conversion of Infix to postfix expression - Queue ADT – Operations - Circular Queue – Priority Queue - deQueue – applications of queues.

**UNIT III            NON LINEAR DATA STRUCTURES – TREES**

9

Tree ADT – tree traversals - Binary Tree ADT – expression trees – applications of trees – binary search tree ADT –Threaded Binary Trees- AVL Trees – B-Tree - B+ Tree - Heap – Applications of heap

**UNIT IV            NON LINEAR DATA STRUCTURES - GRAPHS**

9

Definition – Representation of Graph – Types of graph - Breadth-first traversal - Depth-first traversal – Topological Sort – Bi-connectivity – Cut vertex – Euler circuits – Applications of graphs

**UNIT V            SEARCHING, SORTING AND HASHING TECHNIQUES**

9

Searching- Linear Search - Binary Search. Sorting - Bubble sort - Selection sort - Insertion sort - Shell sort – Radix sort. Hashing- Hash Functions – Separate Chaining – Open Addressing – Rehashing – Extendible Hashing.

**TOTAL: 45****PERIODS****OUTCOMES:**

At the end of the course, the student should be able to:

1. Implement abstract data types for linear data structures.
2. Apply the different linear and non-linear data structures to problem solutions.
3. Critically analyze the various sorting algorithms.

**TEXT BOOKS:**

1. Mark Allen Weiss, "Data Structures and Algorithm Analysis in C", 2nd Edition, Pearson Education, 1997.
2. Reema Thareja, "Data Structures Using C", Second Edition, Oxford University Press, 2011

**REFERENCES:**

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, "Introduction to Algorithms", Second Edition, McGraw Hill, 2002.
2. Aho, Hopcroft and Ullman, "Data Structures and Algorithms", Pearson Education, 1983.
3. Stephen G. Kochan, "Programming in C", 3rd edition, Pearson Education.
4. Ellis Horowitz, Sartaj Sahni, Susan Anderson-Freed, "Fundamentals of Data Structures in C", Second Edition, University Press, 2008

**UNIT I LINEAR DATA STRUCTURES – LIST**

Abstract Data Types (ADTs) – List ADT – array-based implementation – linked list implementation — singly linked lists- circularly linked lists- doubly-linked lists– applications of lists –Polynomial Manipulation – All operation (Insertion, Deletion, Merge, Traversal)

**Data:**

A collection of facts, concepts, figures, observations, occurrences or instructions in a formalized manner.

**Information:**

The meaning that is currently assigned to data by means of the conventions applied to those data(i.e. processed data)

**Record:**

Collection of related fields.

**Data type:**

Set of elements that share common set of properties used to solve a program.

**Data Structures:**

Data Structure is the way of organizing, storing, and retrieving data and their relationship with each other.

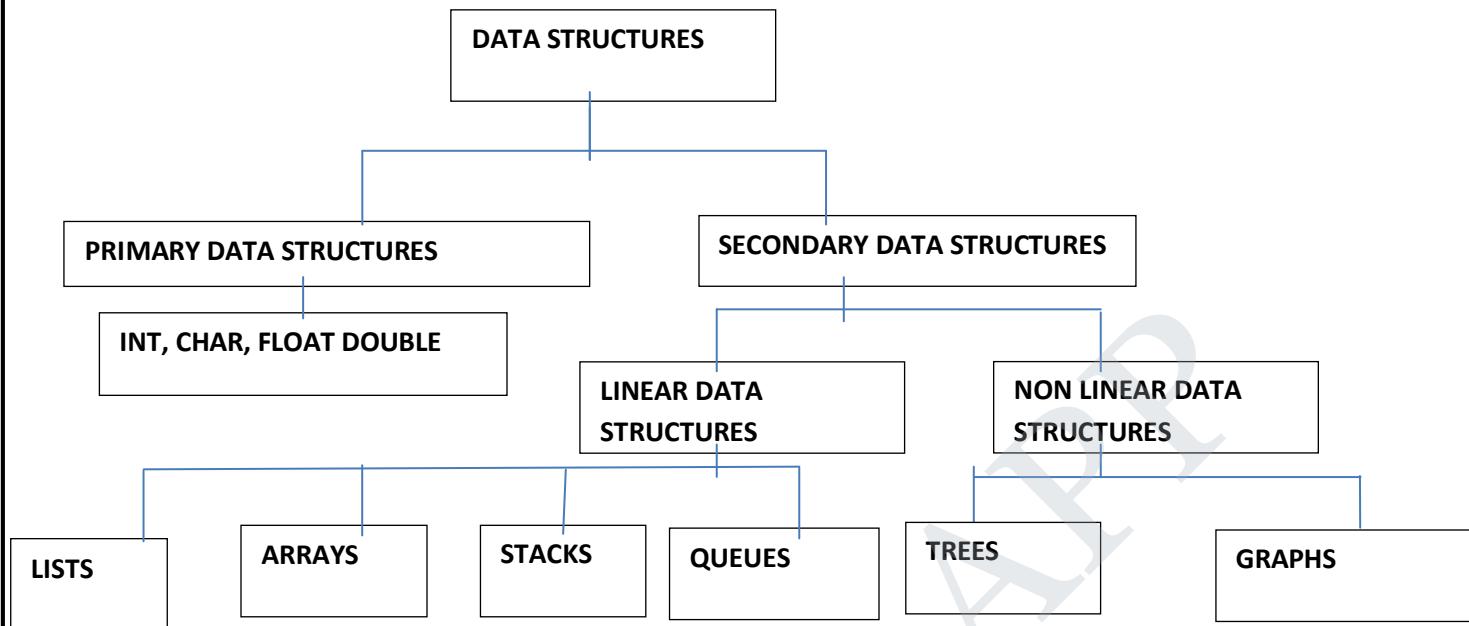
**Characteristics of data structures:**

1. It depicts the logical representation of data in computer memory.
2. It represents the logical relationship between the various data elements.
3. It helps in efficient manipulation of stored data elements.
4. It allows the programs to process the data in an efficient manner.

**Operations on Data Structures:**

- 1.Traversal
- 2.Search
- 3.Insertion
- 4.Deletion

## CLASSIFICATION OF DATA STRUCTURES



### Primary Data Structures/Primitive Data Structures:

Primitive data structures include all the fundamental data structures that can be directly manipulated by machine-level instructions. Some of the common primitive data structures include integer, character, real, boolean etc

### Secondary Data Structures/Non Primitive Data Structures:

Non primitive data structures, refer to all those data structures that are derived from one or more primitive data structures. The objective of creating non-primitive data structures is to form sets of homogeneous or heterogeneous data elements.

#### Linear Data Structures:

Linear data structures are data structures in which, all the data elements are arranged in a linear or sequential fashion. Examples of data structures include arrays, stacks, queues, linked lists, etc.

#### Non Linear Structures:

In non-linear data structures, there is definite order or sequence in which data elements are arranged. For instance, a non-linear data structures could arrange data elements in a hierarchical fashion. Examples of non-linear data structures are trees and graphs.

#### Static and dynamic data structure:

##### Static Ds:

If a ds is created using static memory allocation, ie. ds formed when the number of data items are known in advance ,it is known as static data static ds or fixed size ds.

Dynamic Ds:

If the ds is created using dynamic memory allocation i.e ds formed when the number of data items are not known in advance is known as dynamic ds or variable size ds.

### Application of data structures:

Data structures are widely applied in the following areas:

- Compiler design
- Operating system
- Statistical analysis package
- DBMS
- Numerical analysis
- Simulation
- Artificial intelligence
- Graphics

---

### ABSTRACT DATA TYPES (ADTS):

An abstract Data type (ADT) is defined as a mathematical model with a collection of operations defined on that model. Set of integers, together with the operations of union, intersection and set difference form a example of an ADT. An ADT consists of data together with functions that operate on that data.

### Advantages/Benefits of ADT:

- 1.Modularity
  - 2.Reuse
  - 3.code is easier to understand
  - 4.Implementation of ADTs can be changed without requiring changes to the program that uses the ADTs.
- 

### THE LIST ADT:

List is an ordered set of elements.

The general form of the list is  $A_1, A_2, \dots, A_N$

$A_1$  - First element of the list

$A_2$ - 1<sup>st</sup> element of the list

N –Size of the list

If the element at position  $i$  is  $A_i$ , then its successor is  $A_{i+1}$  and its predecessor is  $A_{i-1}$

### Various operations performed on List

1. Insert (X, 5)- Insert the element X after the position 5.
2. Delete (X) - The element X is deleted
3. Find (X) - Returns the position of X.
4. Next (i) - Returns the position of its successor element i+1.
5. Previous (i) Returns the position of its predecessor i-1.
6. Print list - Contents of the list is displayed.
7. Makeempty- Makes the list empty.

### Implementation of list ADT:

1. Array based Implementation

2. Linked List based implementation

#### Array Implementation of list:

Array is a collection of specific number of same type of data stored in consecutive memory locations. Array is a static data structure i.e., the memory should be allocated in advance and the size is fixed. This will waste the memory space when used space is less than the allocated space.

Insertion and Deletion operation are expensive as it requires more data movements

Find and Print list operations takes constant time.

20	10	30	40	50	60
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]

The basic operations performed on a list of elements are

- a. Creation of List.
- b. Insertion of data in the List
- c. Deletion of data from the List
- d. Display all data's in the List
- e. Searching for a data in the list

**Declaration of Array:**

```
#define maxsize 10  
int list[maxsize], n ;
```

**Create Operation:**

Create operation is used to create the list with „ n „ number of elements .If „ n „ exceeds the array's maxsize, then elements cannot be inserted into the list. Otherwise the array elements are stored in the consecutive array locations (i.e.) list [0], list [1] and so on.

```
void Create ()  
{  
    int i;  
    printf("\nEnter the number of elements to be added in the list:\t");  
    scanf("%d",&n);  
    printf("\nEnter the array elements:\t");  
    for(i=0;i<n;i++)  
        scanf("%d",&list[i]);  
}
```

If n=6, the output of creation is as follows:

list[6]

20	10	30	40	50	60
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]

**Insert Operation:**

Insert operation is used to insert an element at particular position in the existing list. Inserting the element in the last position of an array is easy. But inserting the element at a particular position in an array is quite difficult since it involves all the subsequent elements to be shifted one position to the right.

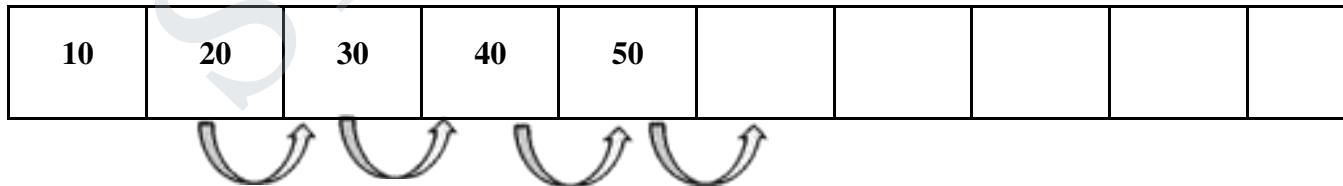
**Routine to insert an element in the array:**

```
void Insert( )  
{  
    int i,data,pos;  
    printf("\nEnter the data to be inserted:\t");  
    scanf("%d",&data);  
    printf("\nEnter the position at which element to be inserted:\t");  
    scanf("%d",&pos);  
    if (pos==n)  
        printf ("Array overflow");  
    for(i = n-1 ; i >= pos-1 ; i--)  
        list[i+1] = list[i];  
    list[pos-1] = data;  
    n=n+1;  
    Display();}
```

Consider an array with 5 elements [ max elements = 10 ]

10	20	30	40	50					
----	----	----	----	----	--	--	--	--	--

If data 15 is to be inserted in the 2<sup>nd</sup> position then 50 has to be moved to next index position, 40 has to be moved to 50 position, 30 has to be moved to 40 position and 20 has to be moved to 30 position.



10		20	30	40	50				
----	--	----	----	----	----	--	--	--	--

After this four data movement, 15 is inserted in the 2<sup>nd</sup> position of the array.

10	15	20	30	40	50				
----	----	----	----	----	----	--	--	--	--

**Deletion Operation:**

Deletion is the process of removing an element from the array at any position.

Deleting an element from the end is easy. If an element is to be deleted from any particular position ,it requires all subsequent element from that position is shifted one position towards left.

**Routine to delete an element in the array:**

```
void Delete( )
{
    int i, pos ;
    printf("\nEnter the position of the data to be deleted:\t");
    scanf("%d",&pos);
    printf("\nThe data deleted is:\t %d", list[pos-1]);
    for(i=pos-1;i<n-1;i++)
        list[i]=list[i+1];
    n=n-1;
    Display();
}
```

Consider an array with 5 elements [ max elements = 10 ]

10	20	30	40	50					
----	----	----	----	----	--	--	--	--	--

If data 20 is to be deleted from the array, then 30 has to be moved to data 20 position, 40 has to be moved to data 30 position and 50 has to be moved to data 40 position.

10	20	30	40	50					
----	----	----	----	----	--	--	--	--	--

After this 3 data movements, data 20 is deleted from the 2<sup>nd</sup> position of the array.

10	30	40	50						
----	----	----	----	--	--	--	--	--	--

### Display Operation/Traversing a list

Traversal is the process of visiting the elements in a array.

Display( ) operation is used to display all the elements stored in the list. The elements are stored from the index 0 to n - 1. Using a for loop, the elements in the list are viewed

#### Routine to traverse/display elements of the array:

```
void display( )
{
    int i;
    printf("\n*****Elements in the array*****\n");
    for(i=0;i<n;i++)
        printf("%d\t",list[i]);
}
```

### Search Operation:

Search( ) operation is used to determine whether a particular element is present in the list or not. Input the search element to be checked in the list.

#### Routine to search an element in the array:

```
void Search()
{
    int search,i,count = 0;
    printf("\nEnter the element to be searched:\t");
    scanf("%d",&search);
    for(i=0;i<n;i++)
    {
        if(search == list[i])
            count++;
    }
}
```

```
if(count==0)
    printf("\nElement not present in the list");
else
    printf("\nElement present in the list");
}
```

### Program for array implementation of List

```
#include<stdio.h>
#include<conio.h>
#define maxsize 10
int list[maxsize],n;
void Create();
void Insert();
void Delete();
void Display();
void Search();
void main()
{
    int choice;
    clrscr();
    do
    {
        printf("\n Array Implementation of List\n");
        printf("\t1.Create\n");
        printf("\t2.Insert\n");
        printf("\t3.Delete\n");
        printf("\t4.Display\n");
        printf("\t5.Search\n");
        printf("\t6.Exit\n");
        printf("\nEnter your choice:\t");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1: Create();
                      break;
            case 2: Insert();
                      break;
            case 3: Delete();
                      break;
            case 4: Display();
                      break;
            case 5: Search();
                      break;
            case 6: exit(1);
        }
    }
}
```

```
        default: printf("\nEnter option between 1 - 6\n");
                  break;
    }
}while(choice<7);
}
void Create()
{
    int i;
    printf("\nEnter the number of elements to be added in the list:\t");
    scanf("%d",&n);
    printf("\nEnter the array elements:\t");
    for(i=0;i<n;i++)
        scanf("%d",&list[i]);
    Display();
}
void Insert()
{
    int i,data,pos;
    printf("\nEnter the data to be inserted:\t");
    scanf("%d",&data);
    printf("\nEnter the position at which element to be inserted:\t");
    scanf("%d",&pos);
    for(i = n-1 ; i >= pos-1 ; i--)
        list[i+1] = list[i];
    list[pos-1] = data;
    n+=1;
    Display();
}
void Delete( )
{
    int i,pos;
    printf("\nEnter the position of the data to be deleted:\t");
    scanf("%d",&pos);
    printf("\nThe data deleted is:\t %d", list[pos-1]);
    for(i=pos-1;i<n-1;i++)
        list[i]=list[i+1];
    n=n-1;
    Display();
}
void Display()
{
    int i;
    printf("\n*****Elements in the array*****\n");
    for(i=0;i<n;i++)
        printf("%d\t",list[i]);
}
```

```
void Search()
{
    int search,i,count = 0;
    printf("\nEnter the element to be searched:\t");
    scanf("%d",&search);
    for(i=0;i<n;i++)
    {
        if(search == list[i])
        {
            count++;
        }
    }
    if(count==0)
        printf("\nElement not present in the list");
    else
        printf("\nElement present in the list");
}
```

**Output**

Array Implementation of List

- 1.create
- 2.Insert
- 3.Delete
- 4.Display
- 5.Search
- 6.Exit

Enter your choice: 1

Enter the number of elements to be added in the list: 5

Enter the array elements: 1 2 3 4 5

\*\*\*\*\*Elements in the array\*\*\*\*\*

1 2 3 4 5

Array Implementation of List

- 1.create
- 2.Insert
- 3.Delete
- 4.Display
- 5.Search
- 6.Exit

Enter your choice: 2

Enter the data to be inserted: 3

Enter the position at which element to be inserted: 1

\*\*\*\*\*Elements in the array\*\*\*\*\*

3 1 2 3 4 5

Array Implementation of List

- 1.create
- 2.Insert
- 3.Delete
- 4.Display
- 5.Search
- 6.Exit

Enter your choice: 3

Enter the position of the data to be deleted: 4

The data deleted is: 3

\*\*\*\*\*Elements in the array\*\*\*\*\*

3 1 2 4 5

Array Implementation of List

- 1.create
- 2.Insert
- 3.Delete
- 4.Display
- 5.Search
- 6.Exit

Enter your choice: 5

Enter the element to be searched: 1

Element present in the list

Array Implementation of List

- 1.create
- 2.Insert
- 3.Delete
- 4.Display
- 5.Search
- 6.Exit

Enter your choice:6

### **Advantages of array implementation:**

- 1.The elements are faster to access using random access
- 2.Searching an element is easier

### **Limitation of array implementation**

- An array store its nodes in consecutive memory locations.
- The number of elements in the array is fixed and it is not possible to change the number of elements .
- Insertion and deletion operation in array are expensive. Since insertion is performed by pushing the entire array one position down and deletion is performed by shifting the entire array one position up.

### **Applications of arrays:**

Arrays are particularly used in programs that require storing large collection of similar type data elements.

### **Differences between Array based and Linked based implementation**

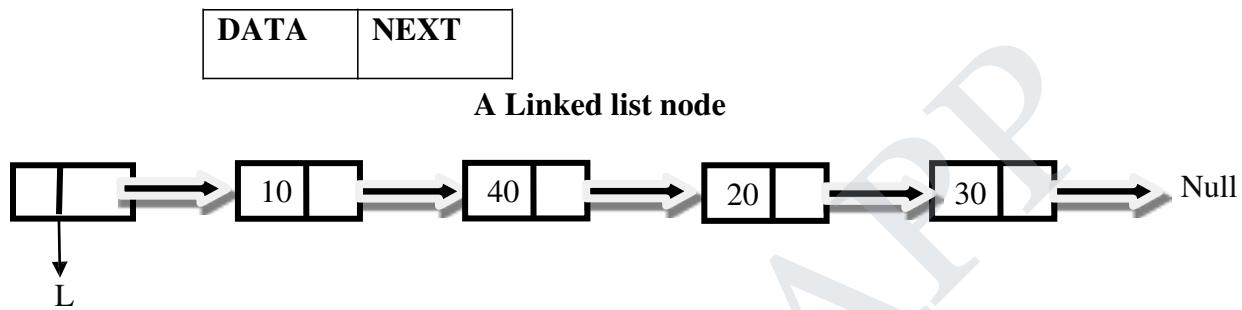
	Array	Linked List
Definition	Array is a collection of elements having same data type with common name	Linked list is an ordered collection of elements which are connected by links/pointers
Access	Elements can be accessed using index/subscript, random access	Sequential access
Memory structure	Elements are stored in contiguous memory locations	Elements are stored at available memory space
Insertion & Deletion	Insertion and deletion takes more time in array	Insertion and deletion are fast and easy
Memory Allocation	Memory is allocated at compile time i.e static memory allocation	Memory is allocated at run time i.e dynamic memory allocation
Types	1D,2D,multi-dimensional	SLL, DLL circular linked list
Dependency	Each elements is independent	Each node is dependent on each other as address part contains address of next node in the list

**Linked list based implementation:****Linked Lists:**

A Linked list is an ordered collection of elements. Each element in the list is referred as a node. Each node contains two fields namely,

Data field-The data field contains the actual data of the elements to be stored in the list

Next field- The next field contains the address of the next node in the list

**Advantages of Linked list:**

1. Insertion and deletion of elements can be done efficiently
2. It uses dynamic memory allocation
3. Memory utilization is efficient compared to arrays

**Disadvantages of linked list:**

1. Linked list does not support random access
2. Memory is required to store next field
3. Searching takes time compared to arrays

**Types of Linked List**

1. Singly Linked List or One Way List
2. Doubly Linked List or Two-Way Linked List
3. Circular Linked List

**Dynamic allocation**

The process of allocating memory to the variables during execution of the program or at run time is known as dynamic memory allocation. C language has four library routines which allow this function.

Dynamic memory allocation gives best performance in situations in which we do not know memory requirements in advance. C provides four library routines to automatically allocate memory at the run time.

**Memory allocation/de-allocation functions**

Function	Task
<code>malloc()</code>	Allocates memory and returns a pointer to the first byte of allocated space
<code>calloc()</code>	Allocates space for an array of elements, initializes them to zero and returns a pointer to the memory
<code>free()</code>	Frees previously allocated memory
<code>realloc()</code>	Alters the size of previously allocated memory

---

To use dynamic memory allocation functions, you must include the header file stdlib.h.

**malloc()**

The malloc function reserves a block of memory of specified size and returns a pointer of type void. This means that we can assign it to any type of pointer.

The general syntax of malloc() is

`ptr = (cast-type*)malloc(byte-size);`

where `ptr` is a pointer of type cast-type. malloc() returns a pointer (of cast type) to an area of memory with size byte-size.

**calloc():**

calloc() function is another function that reserves memory at the run time. It is normally used to request multiple blocks of storage each of the same size and then sets all bytes to zero. calloc() stands for contiguous memory allocation and is primarily used to allocate memory for arrays.

The syntax of calloc() can be given as:

`ptr = (cast-type*)calloc(n, elem-size);`

The above statement allocates contiguous space for n blocks each of size elem-size bytes. The only difference between malloc() and calloc() is that when we use calloc(), all bytes are initialized to zero. calloc() returns a pointer to the first byte of the allocated region.

**free():**

The free() is used to release the block of memory.

**Syntax:**

The general syntax of the free() function is,

`free(ptr);`

where `ptr` is a pointer that has been created by using malloc() or calloc(). When memory is deallocated using free(), it is returned back to the free list within the heap.

**realloc():**

At times the memory allocated by using calloc() or malloc() might be insufficient or in excess.

In both the situations we can always use realloc() to change the memory size already allocated by calloc() and malloc(). This process is called *reallocation of memory*. The general syntax for realloc() can be given as,

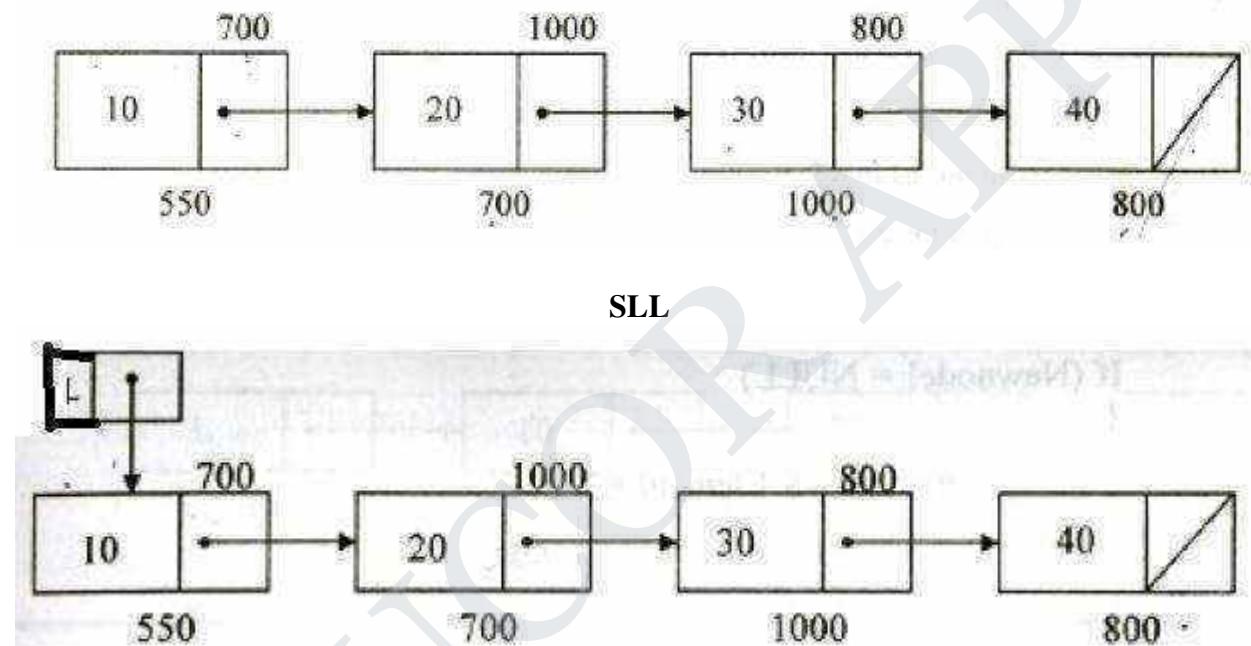
`ptr = realloc(ptr, newsize);`

The function realloc() allocates new memory space of size specified by newsize to the pointer

variable ptr. It returns a pointer to the first byte of the memory block. The allocated new block may be or may not be at the same region. Thus, we see that realloc() takes two arguments. The first is the pointer referencing the memory and the second is the total number of bytes you want to reallocate.

### Singly Linked List

A singly linked list is a linked list in which each node contains only one link field pointing to the next node in the list



### **SLL with a Header**

Basic operations on a singly-linked list are:

1. Insert – Inserts a new node in the list.
2. Delete – Deletes any node from the list.
3. Find – Finds the position( address ) of any node in the list.
4. FindPrevious - Finds the position( address ) of the previous node in the list.
5. FindNext- Finds the position( address ) of the next node in the list.
6. Display - display the date in the list
7. Search-find whether a element is present in the list or not

**Declaration of Linked List**

```
void insert(int X,List L,position P);
void find(List L,int X); void
delete(int x , List L); typedef
struct node *position;
position L,p,newnode;
struct node
{
    int data;
    position next;
};
```

**Creation of the list:**

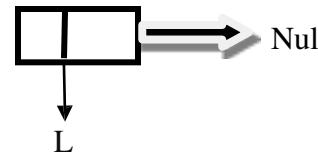
This routine creates a list by getting the number of nodes from the user. Assume n=4 for this example.

```
void create()
{
int i,n;
L=NULL;
newnode=(struct node*)malloc(sizeof(struct node));
printf("\n Enter the number of nodes to be inserted\n");
scanf("%d",&n);
printf("\n Enter the data\n");
scanf("%d",&newnode->data);
newnode->next=NULL;
L=newnode;
p=L;
for(i=2;i<=n;i++)
{
    newnode=(struct node *)malloc(sizeof(struct node));
    scanf("%d",&newnode->data);
    newnode->next=NULL;
```

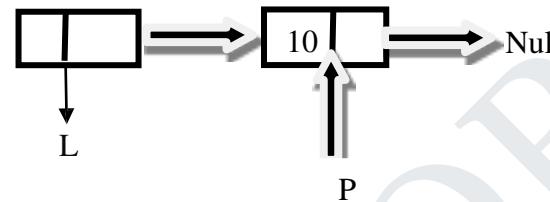
```
p->next=newnode;  
p=newnode;  
}  
}
```

**Initially the list is empty**

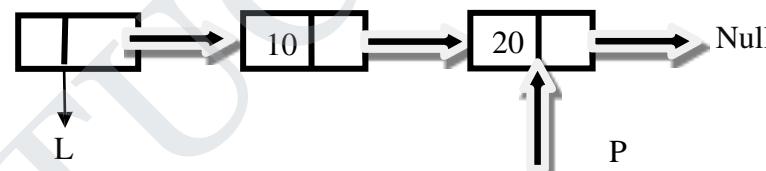
**List L**



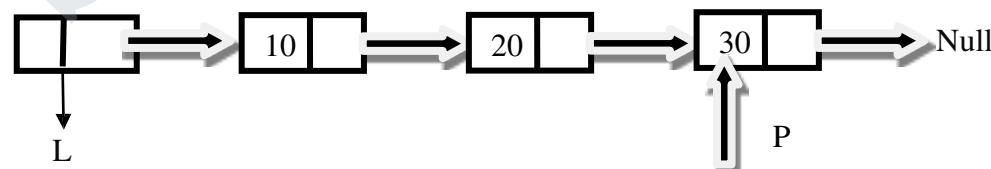
**Insert(10,List L)-** A new node with data 10 is inserted and the next field is updated to NULL. The next field of previous node is updated to store the address of new node.



**Insert(20,L) -** A new node with data 20 is inserted and the next field is updated to NULL. The next field of previous node is updated to store the address of new node.



**Insert(30,L) -** A new node with data 30 is inserted and the next field is updated to NULL. The next field of previous node is updated to store the address of new node.



**Case 1:Routine to insert an element in list at the beginning**

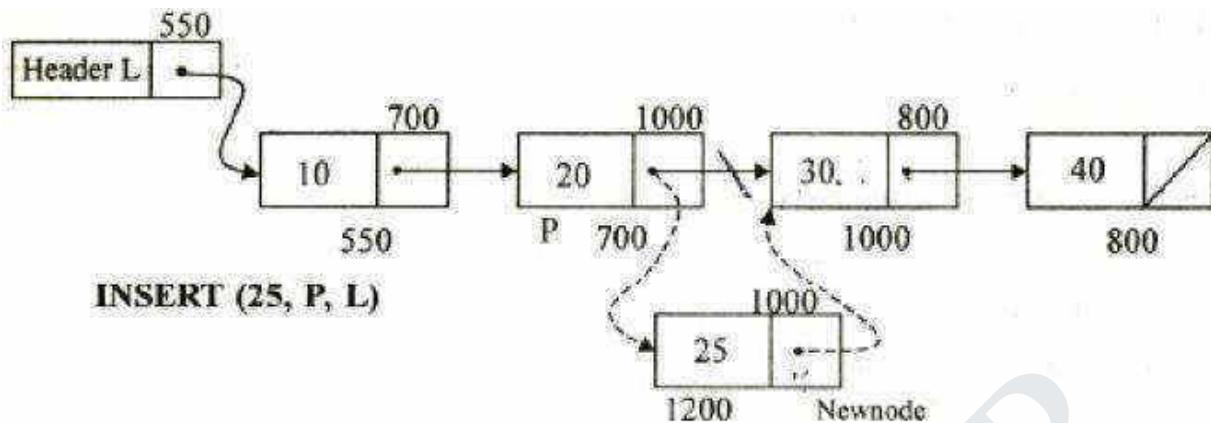
```
void insert(int X, List L, position p)
{
    p=L;
    newnode=(struct node*)malloc(sizeof(struct node));
    printf("\nEnter the data to be inserted\n");
    scanf("%d",&newnode->data);
    newnode->next=L;
    L=newnode;
}
```

**Case 2:Routine to insert an element in list at Position**

This routine inserts an element X after the position P.

```
Void Insert(int X, List L, position p)
{
    position newnode;
    newnode =(struct node*) malloc( sizeof( struct node ) );
    if( newnode == NULL )
        Fatal error( " Out of Space " );
    else
    {
        Newnode -> data = x ;
        Newnode -> next = p ->next ;
        P -> next = newnode ;
    }
}
```

Insert(25,L, P) - A new node with data 25 is inserted after the position P and the next field is updated to NULL. The next field of previous node is updated to store the address of new node.

**Case 3: Routine to insert an element in list at the end of the list**

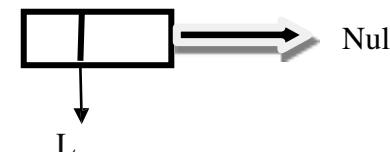
```
void insert(int X, List L, position p)
```

```
{
p=L;
newnode=(struct node*)malloc(sizeof(struct node));
printf("\nEnter the data to be inserted\n");
scanf("%d",&newnode->data);
while(p->next!=NULL)
p=p->next;
newnode->next=NULL;
p->next=newnode;
p=newnode;
}
```

**Routine to check whether a list is Empty**

This routine checks whether the list is empty .If the list is empty it returns 1

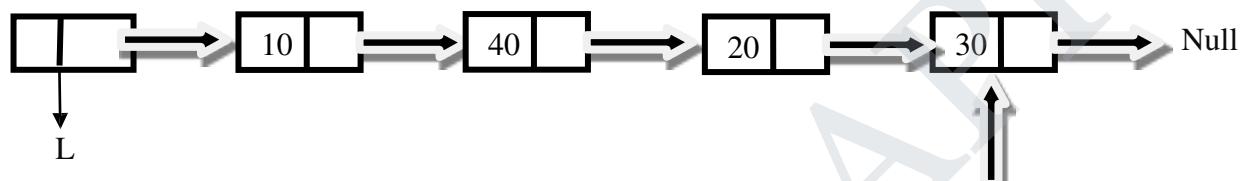
```
int IsEmpty( List L )
{
    if ( L -> next == NULL )
        return(1);
}
```



**Routine to check whether the current position is last in the List**

This routine checks whether the current position p is the last position in the list. It returns 1 if position p is the last position

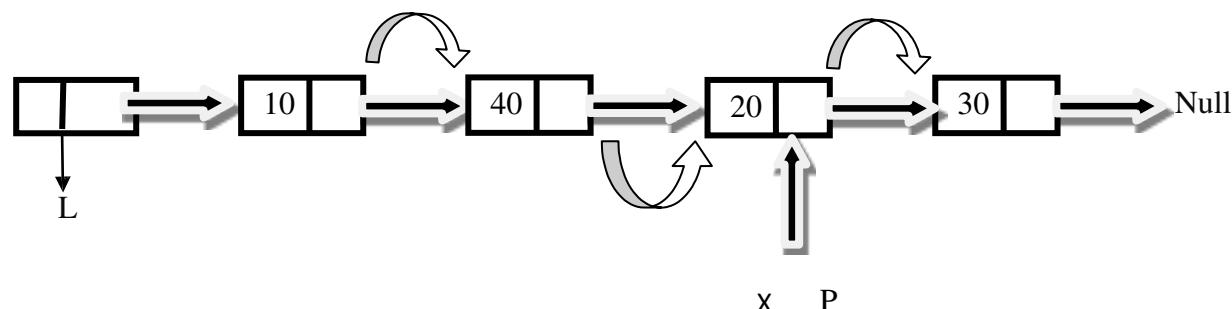
```
int IsLast(List L , position p)
{
    if( p -> next==NULL)
        return(1);
}
```

**Routine to Find the Element in the List:**

This routine returns the position of X in the list L

```
position find(List L, int X)
{
    position p;
    p=L->next;
    while(p!=NULL && p->data!=X)
        p=p->next;
    return(p);
}
```

Find(List L, 20) - To find an element X traverse from the first node of the list and move to the next with the help of the address stored in the next field until data is equal to X or till the end of the list



### Find Previous

It returns the position of its predecessor.

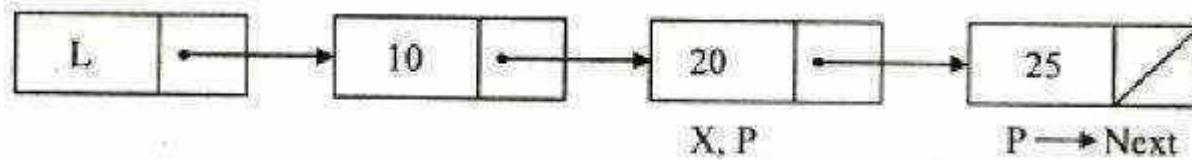
```
position FindPrevious (int X, List L)
{
    position p;
    p = L;
    while( p -> next != NULL && p -> next -> data! = X )
        p = p -> next;
    return P;
}
```



### Routine to find next Element in the List

It returns the position of successor.

```
void FindNext(int X, List L)
{
    position P;
    P=L->next;
    while(P!=NULL && P->data!=X)
        P = P->next;
    return P->next;
}
```



### Routine to Count the Element in the List:

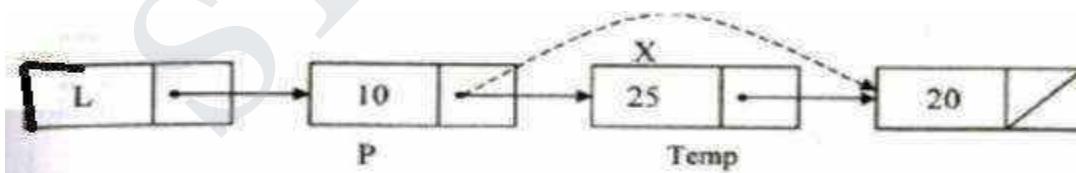
This routine counts the number of elements in the list

```
void count(List L)
{
    P = L -> next;
    while( p != NULL )
    {
        count++;
        p = p -> next;
    }
    print count;
}
```

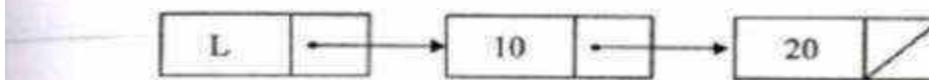
### Routine to Delete an Element in the List:

It delete the first occurrence of element X from the list L

```
void Delete( int x , List L){
    position p, Temp;
    p = FindPrevious( X, L);
    if( ! IsLast (p, L)){
        temp = p -> next;
        P -> next = temp -> next;
        free ( temp );
    }
}
```



**BEFORE DELETION**

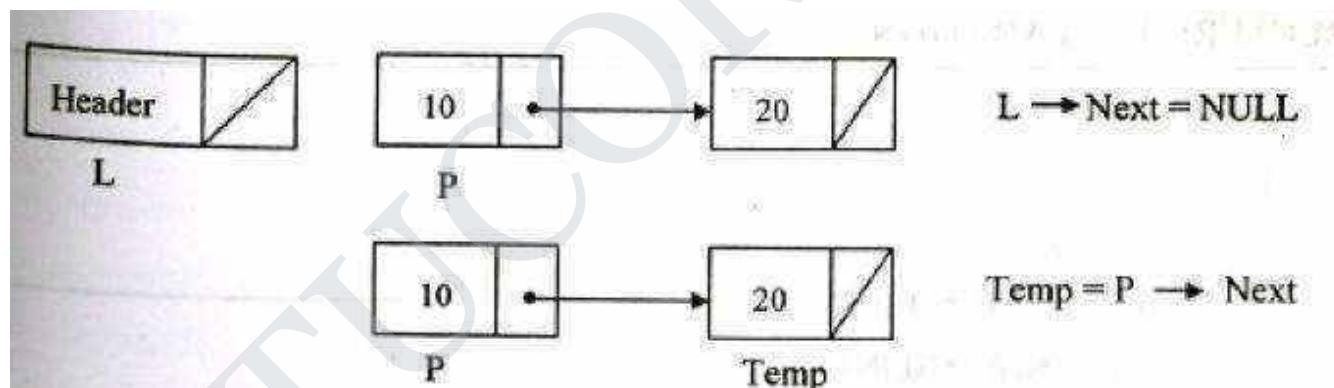


**AFTER DELETION**

**Routine to Delete the List**

This routine deleted the entire list.

```
void Delete_list(List L)
{
    position P,temp;
    P=L->next;
    L->next=NULL;
    while(P!=NULL)
    {
        temp=P->next;
        free(P);
        P=temp;
    }
}
```



Program 1: Implementation of Singly linked List	Output
#include<stdio.h> #include<conio.h> #include<stdlib.h> void create(); void display(); void insert(); void find(); void delete(); typedef struct node *position; position L,p,newnode; struct node {	1.create 2.display 3.insert 4.find 5.delete  Enter your choice

<pre>int data; position next; }; void main() { int choice; clrscr(); do { printf("1.create\n2.display\n3.insert\n4.find\n5.delete\n\n\n"); printf("Enter your choice\n\n"); scanf("%d",&amp;choice); switch(choice) { case 1:     create();     break; case 2:     display();     break; case 3:     insert();     break; case 4:     find();     break; case 5:     delete();     break; case 6:     exit(0); } } while(choice&lt;7); getch(); } void create() { int i,n; L=NULL; newnode=(struct node*)malloc(sizeof(struct node)); printf("\n Enter the number of nodes to be inserted\n"); scanf("%d",&amp;n); printf("\n Enter the data\n"); scanf("%d",&amp;newnode-&gt;data); newnode-&gt;next=NULL;</pre>	<p>1 Enter the number of nodes to be inserted 5 Enter the data 1 2 3 4 5 1.create 2.display 3.insert 4.find 5.delete Enter your choice 2 1 -&gt; 2 -&gt; 3 -&gt; 4 -&gt; 5 -&gt; Null 1.create 2.display 3.insert 4.find 5.delete Enter your choice 3</p>
---	---

```

L=newnode; p=L;
for(i=2;i<=n;i++)
{
newnode=(struct node *)malloc(sizeof(struct node));
scanf("%d",&newnode->data);
newnode->next=NULL;
p->next=newnode;
p=newnode;
}
}
void display()
{ p=L;
while(p!=NULL)
{
printf("%d -> ",p->data);
p=p->next;
}
printf("Null\n");
}
void insert()
{
int ch;

printf("\nEnter ur choice\n");
printf("\n1.first\n2.middle\n3.end\n");
scanf("%d",&ch);
switch(ch)
{
case 1:
{
int pos,i=1;
p=L;
newnode=(struct node*)malloc(sizeof(struct node));
printf("\nEnter the data to be inserted\n");
scanf("%d",&newnode->data);
printf("\nEnter the position to be inserted\n");
scanf("%d",&pos);
newnode->next=NULL;
while(i<pos-1)
{
p=p->next;
i++;
}
newnode->next=p->next;
p->next=newnode;
}
}
}

```

Enter ur choice

1.first

2.middle

3.end

1

Enter the data to be inserted

7

7 -&gt; 1 -&gt; 2 -&gt; 3 -&gt; 4 -&gt; 5 -&gt; Null

1.create

2.display

3.insert

4.find

5.delete

Enter your choice

```
p=newnode;
display();
break;
}
case 1:
{
p=L;
    newnode=(struct node*)malloc(sizeof(struct node));
    printf("\nEnter the data to be inserted\n");
    scanf("%d",&newnode->data);
    newnode->next=L;
    L=newnode;
    display();
    break;
}
case 3:
{
p=L;
    newnode=(struct node*)malloc(sizeof(struct node));
    printf("\nEnter the data to be inserted\n");
    scanf("%d",&newnode->data);
    while(p->next!=NULL)
        p=p->next;
    newnode->next=NULL;
    p->next=newnode;
    p=newnode;
    display();
    break;
}
}
void find()
{
int search,count=0;
printf("\n Enter the element to be found:\n");
scanf("%d",&search);
p=L;
while(p!=NULL)
{
if(p->data==search)
{
    count++;
    break;
}
p=p->next;
}
}
```

```
if(count==0)
printf("\n Element Not present\n");
else
printf("\n Element present in the list \n\n");
}
void delete()
{
position p,temp;
int x; p=L;
if(p==NULL)
{
printf("empty list\n");
}
else
{
printf("\nEnter the data to be deleted\n");
scanf("%d",&x);
if(x==p->data)
{ temp=p;
L=p->next;
free(temp);
display();
}
else
{
while(p->next!=NULL && p->next->data!=x)
{
p=p->next;
}
temp=p->next;
p->next=p->next->next;
free(temp);
display();
}
}
}
```

**Advantages of SLL**

- 1.The elements can be accessed using the next link
- 2.Occupies less memory than DLL as it has only one next field.

**Disadvantages of SLL**

- 1.Traversal in the backwards is not possible
- 2.Less efficient to for insertion and deletion.

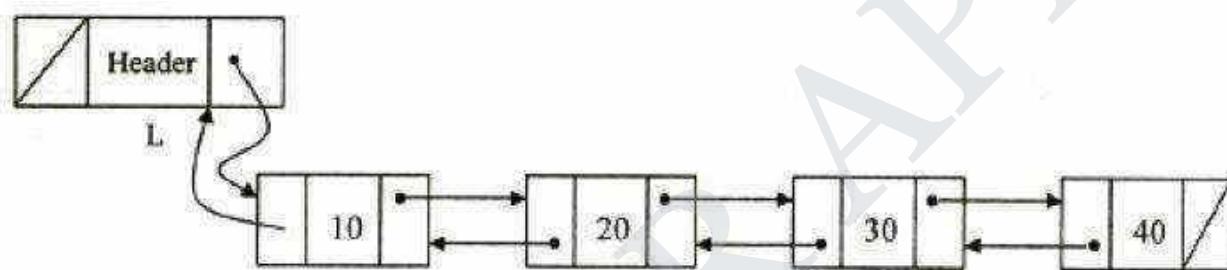
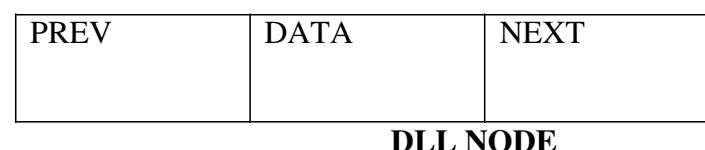
### Doubly-Linked List

A doubly linked list is a linked list in which each node has three fields namely Data, Next, Prev.

Data-This field stores the value of the element

Next-This field points to the successor node in the list

Prev-This field points to the predecessor node in the list



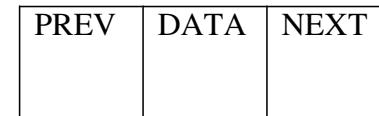
**DOUBLY LINKED LIST**

Basic operations of a doubly -linked list are:

1. Insert – Inserts a new element at the end of the list.
2. Delete – Deletes any node from the list.
3. Find – Finds any node in the list.
4. Print – Prints the list

### Declaration of DLL Node

```
typedef struct node *position ;  
struct node  
{  
    int data;  
    position prev;  
    position next;  
};
```



**Creation of list in DLL**

Initially the list is empty. Then assign the first node as head.

```
newnode->data=X;  
newnode->next=NULL;  
newnode->prev=NULL;  
L=newnode;
```

list. If we add one more node in the list, then create a newnode and attach that node to the end of the list.

```
L->next=newnode;
```

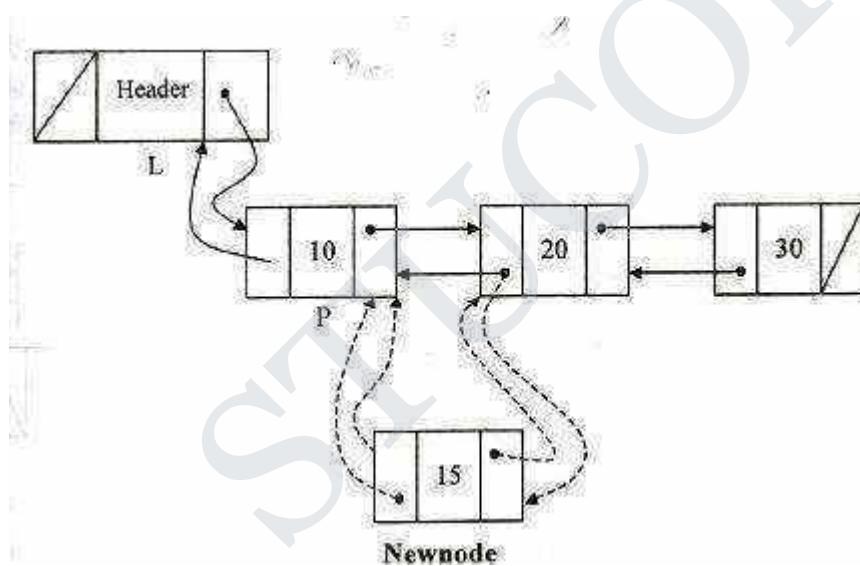
```
newnode->prev=L;
```

**Routine to insert an element in a DLL at the beginning**

```
void Insert (int x, list L, position P){  
    struct node *Newnode;  
    if(pos==1)  
        P=L;  
  
    Newnode = (struct node*)malloc (sizeof(struct node));  
    if (Newnode!=NULL)  
        Newnode->data= X;  
        Newnode ->next= L->next;  
        L->next ->prev=Newnode  
        L->next = Newnode;  
        Newnode ->prev = L;  
}
```

**Routine to insert an element in a DLL any position :**

```
void Insert (int x, list L, position P)
{
    struct node *Newnode;
    Newnode = (struct node*)malloc (sizeof(struct node));
    if (Newnode!=NULL)
        Newnode->data= X;
    Newnode ->next= P ->next;
    P->next ->prev=Newnode
    P ->next = Newnode;
    Newnode ->prev = P;
}
```

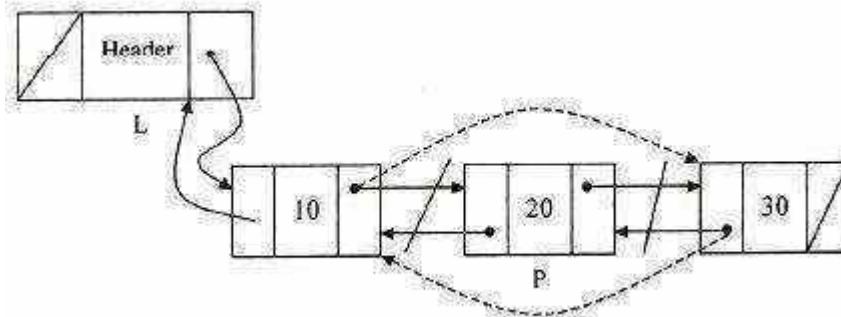
**Routine to insert an element in a DLL at the end:**

```
void insert(int X, List L, position p)
{
    p=L;
    newnode=(struct node*)malloc(sizeof(struct node));
```

```
printf("\nEnter the data to be inserted\n");
scanf("%d",&newnode->data);
while(p->next!=NULL)
p=p->next;
newnode->next=NULL;
p->next=newnode;
newnode->prev=p;
}
```

**Routine for deleting an element:**

```
void Delete (int x ,List L)
{
    Position p , temp;
    P = Find( x, L );
    if(P==L->next)
        temp=L;
    L->next=temp->next;
    temp->next->prev=L;
    free(temp);
    elseif( IsLast( p, L ) )
    {
        temp = p;
        p -> prev -> next = NULL;
        free(temp);
    }
    else
    {
        temp = p;
        p -> prev -> next = p -> next;
        p -> next -> prev = p -> prev;
        free(temp);
    }
}
```

**Routine to display the elements in the list:**

```
void Display( List L )  
{  
    P = L -> next ;  
    while ( p != NULL)  
    {  
        printf("%d", p -> data ;  
        p = p -> next ;  
    }  
    printf(" NULL");  
}
```

**Routine to search whether an element is present in the list**

```
void find()  
{  
int a,flag=0,count=0;  
if(L==NULL)  
printf("\nThe list is empty");  
else  
{  
printf("\nEnter the elements to be searched");  
scanf("%d",&a);  
for(P=L;P!=NULL;P=P->next)  
{  
count++;  
if(P->data==a)  
{  
flag=1;  
printf("\nThe element is found");  
printf("\nThe position is %d",count);  
}
```

```
break;
}
}
}
if(flag==0)
printf("\nThe element is not found");
}
}
```

Program 2 Implementation of Doubly linked list	Output
#include<conio.h> void insert(); void deletion(); void display(); void find(); typedef struct node *position; position newnode,temp,L=NULL,P; struct node { int data; position next; position prev; }; void main() { int choice; clrscr(); do { printf("\n1.INSERT"); printf("\n2.DELETE"); printf("\n3.DISPLAY"); printf("\n4.FIND"); printf("\n5.EXIT"); printf("\nEnter ur option"); scanf("%d",&choice); switch(choice) { case 1: insert(); break; case 2: deletion(); break; case 3: display(); } <td>1.INSERT 2.DELETE 3.DISPLAY 4.FIND 5.EXIT Enter ur option1</td>	1.INSERT 2.DELETE 3.DISPLAY 4.FIND 5.EXIT Enter ur option1
	Enter the data to be inserted10
	1.INSERT 2.DELETE 3.DISPLAY 4.FIND 5.EXIT Enter ur option1
	Enter the data to be inserted 20
	Enter the position where the data is to be inserted 2

```

break;
case 4:
find();
break;
case 5:
exit(1);
}
}while(choice!=5);
getch();
}
void insert()
{
int pos,I;
newnode=(struct node*)malloc(sizeof(struct node));
printf("\nEnter the data to be inserted");
scanf("%d",&newnode->data);
if(L==NULL)
{
L=newnode;
L->next=NULL;
L->prev=NULL;
}
else
{
printf("\nEnter the position where the data is to be inserted");
scanf("%d",&pos);
if(pos==1)
{
newnode->next=L;
newnode->prev=NULL;
L->prev=newnode;
L=newnode;
}
else
{
P=L;
for(i=1;i<pos-1&&P->next!=NULL;i++)
{
P=P->next;
}
newnode->next=P->next;
P->next=newnode;
newnode->prev=P;
P->next->prev=newnode;
}
}
}

```

- |  |  |
|--|--|
| 1.INSERT   |  |
| 2.DELETE   |  |
| 3.DISPLAY  |  |
| 4.FIND   |  |
| 5.EXIT   |  |
| Enter ur option1                                     |  |
| Enter the data to be inserted 30                     |  |
| Enter the position where the data is to be inserted3 |  |
| 1.INSERT   |  |
| 2.DELETE   |  |
| 3.DISPLAY  |  |
| 4.FIND   |  |
| 5.EXIT   |  |
| Enter ur option 3                                    |  |
| The elements in the list are                         |  |
| 10 20 30   |  |
| 1.INSERT   |  |
| 2.DELETE   |  |
| 3.DISPLAY  |  |
| 4.FIND   |  |
| 5.EXIT   |  |
| Enter ur option 2                                    |  |

{ void deletion() { int pos,I; if(L==NULL) printf("\nThe list is empty"); else { printf("\nEnter the position of the data to be deleted"); scanf("%d",&pos);  if(pos==1) { temp=L; L=temp->next; L->prev=NULL; printf("\nThe deleted element is %d",temp->data); free(temp); } else { P=L; for(i=1;i<pos-1;i++) P=P->next; temp=P->next; printf("\nThe deleted element is %d",temp->data); P->next=temp->next; temp->next->prev=P; free(temp); } } } } } } void display() { if(L==NULL) printf("\nNo of elements in the list"); else { printf("\nThe elements in the list are\n"); for(P=L;P!=NULL;P=P->next) printf("%d",P->data); } } } void find() { int a,flag=0,count=0;	Enter the position of the data to be deleted 2  The deleted element is 20 1.INSERT 2.DELETE 3.DISPLAY 4.FIND 5.EXIT Enter ur option 3  The elements in the list are 10 30 1.INSERT 2.DELETE 3.DISPLAY 4.FIND 5.EXIT Enter ur option4  Enter the elements to be searched 20  The element is not found 1.INSERT 2.DELETE 3.DISPLAY 4.FIND 5.EXIT Enter ur option 4
---	---

```
if(L==NULL)
printf("\nThe list is empty");
else
{
printf("\nEnter the elements to be searched");
scanf("%d",&a);
for(P=L;P!=NULL;P=P->next)
{
count++;
if(P->data==a)
{
flag=1;
printf("\nThe element is found");
printf("\nThe position is %d",count);
break;
}
}
if(flag==0)
printf("\nThe element is not found");
}
```

Enter the elements to be searched 30  
The element is found  
The position is 2  
1.INSERT  
2.DELETE  
3.DISPLAY  
4.FIND  
5.EXIT  
Enter ur option5  
Press any key to continue .  
..

### Advantages of DLL:

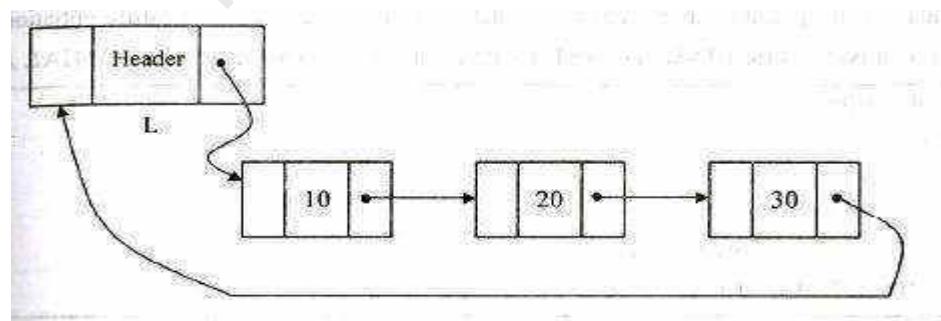
The DLL has two pointer fields. One field is prev link field and another is next link field. Because of these two pointer fields we can access any node efficiently whereas in SLL only one link field is there which makes accessing of any node difficult.

### Disadvantages of DLL:

The DLL has two pointer fields. One field is prev link field and another is next link field. Because of these two pointer fields, more memory space is used by DLL compared to SLL

### CIRCULAR LINKED LIST:

Circular Linked list is a linked list in which the pointer of the last node points to the first node.

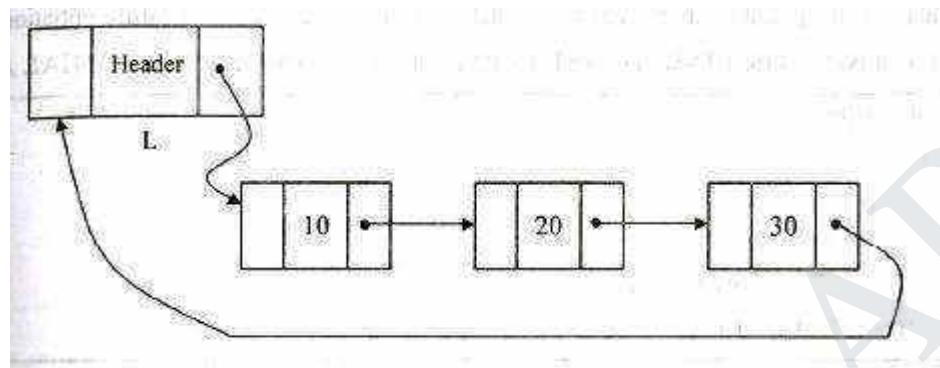


**Types of CLL:**

CLL can be implemented as circular singly linked list and circular doubly linked list.

**Singly linked circular list:**

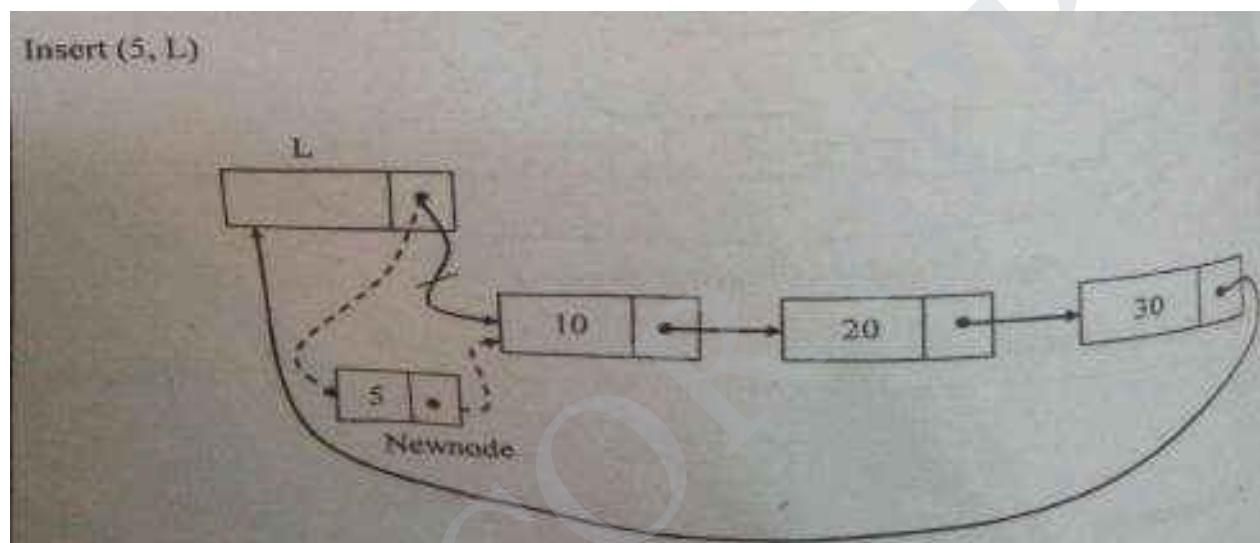
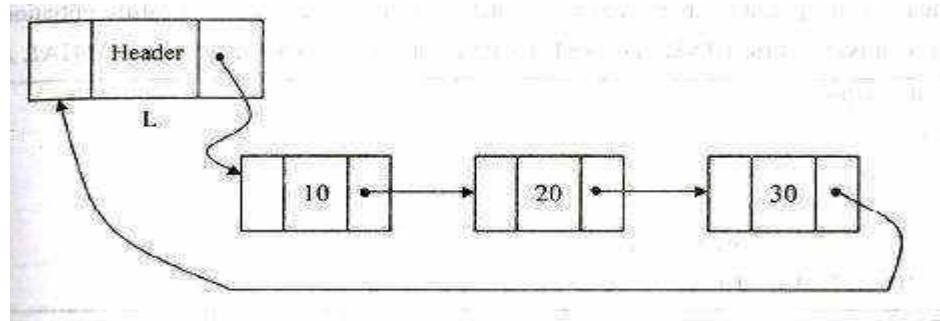
A Singly linked circular list is a linked list in which the last node of the list points to the first node.

**Declaration of node:**

```
typedef struct node *position;  
  
struct node  
{  
    int data;  
    position next;  
};
```

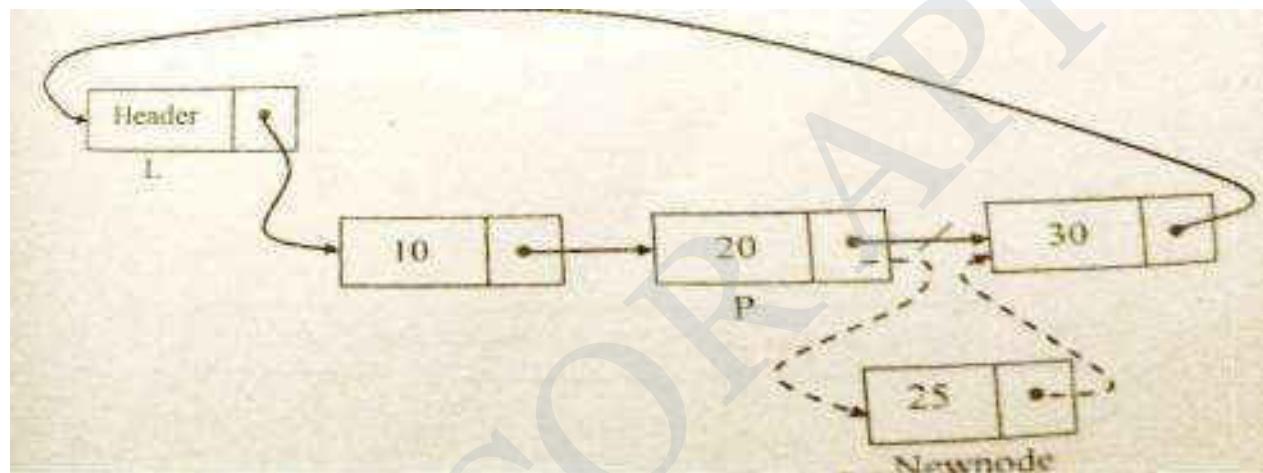
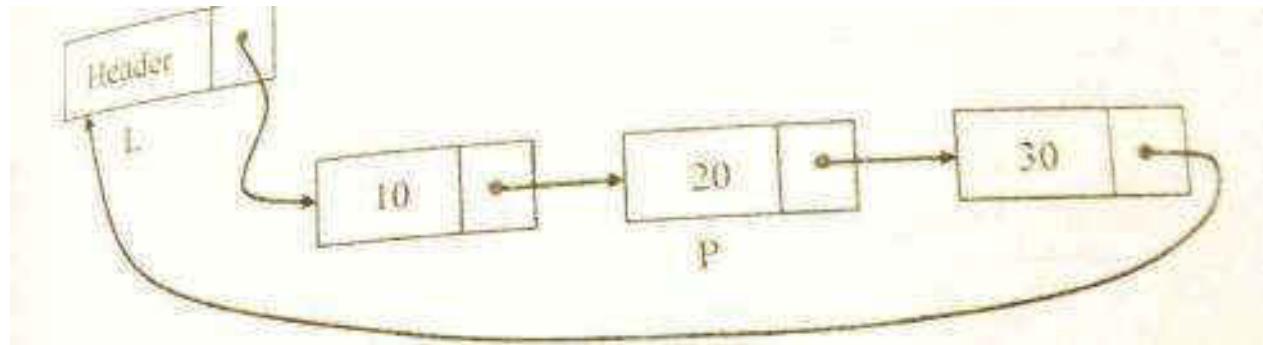
**Routine to insert an element in the beginning**

```
void insert_beg(int X,List L)  
{  
    position Newnode;  
    Newnode=(struct node*)malloc(sizeof(struct node));  
    if(Newnode!=NULL)  
    {  
        Newnode->data=X;  
        Newnode->next=L->next;  
        L->next=Newnode;  
    }  
}
```

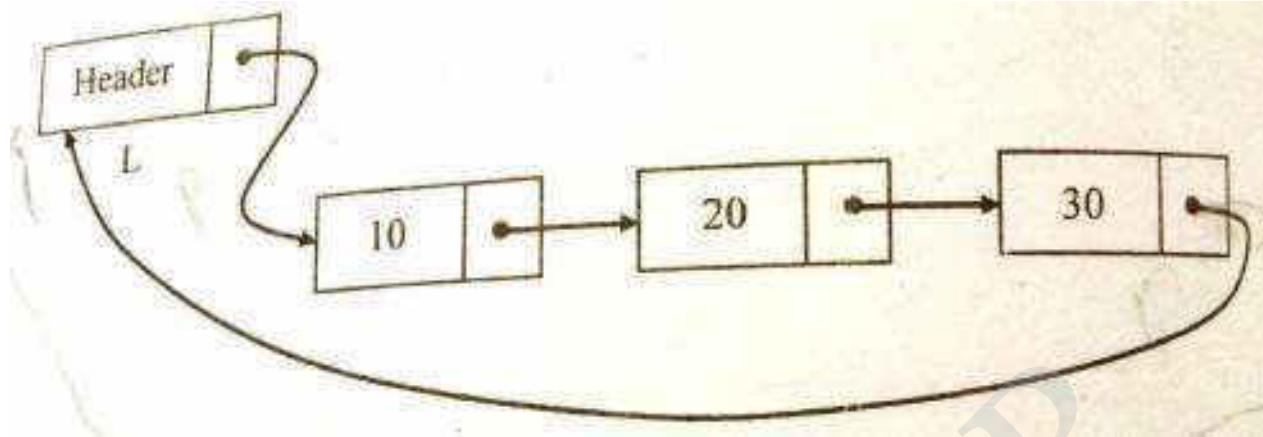


### Routine to insert an element in the middle

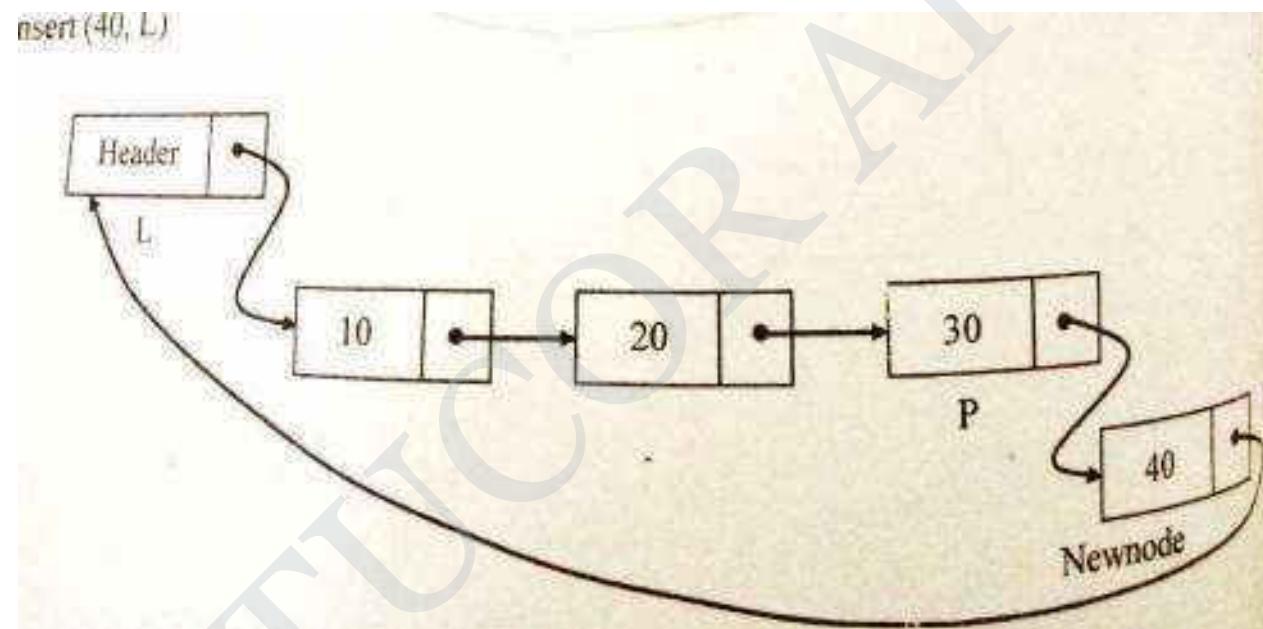
```
void insert_mid(int X, List L, Position p)
{
    position Newnode;
    Newnode=(struct node*)malloc(sizeof(struct node));
    if(Newnode!=NULL)
    {
        Newnode->data=X;
        Newnode->next=p->next;
        p->next=Newnode;
    }
}
```

**Routine to insert an element in the last**

```
void insert_last(int X, List L)
position Newnode;
Newnode=(struct node*)malloc(sizeof(struct node));
if(Newnode!=NULL)
{
P=L;
while(P->next!=L)
P=P->next;
Newnode->data=X;
P->next=Newnode;
Newnode->next=L;
}
```

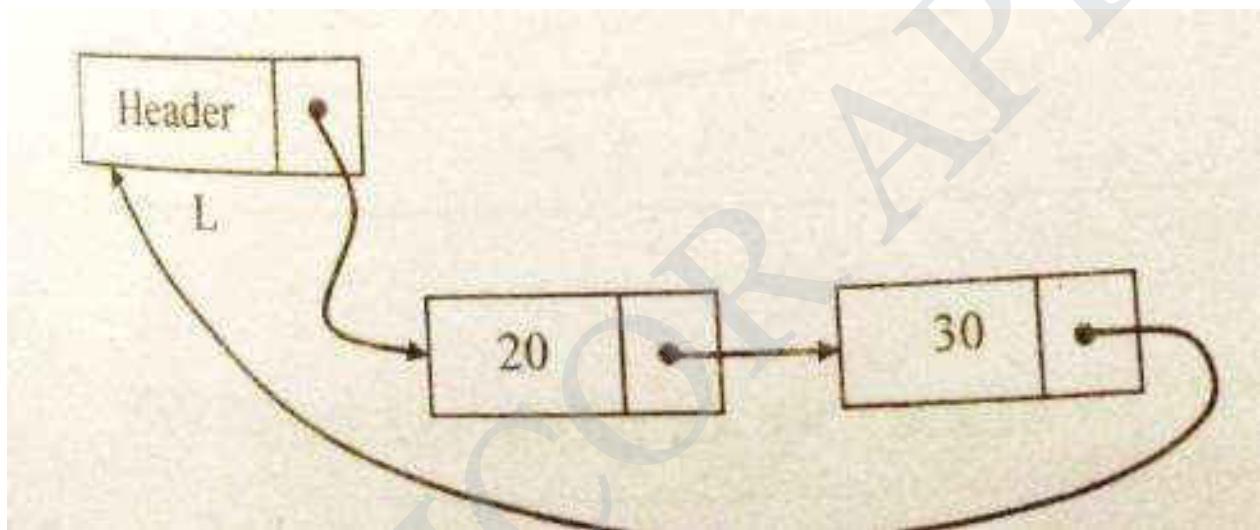
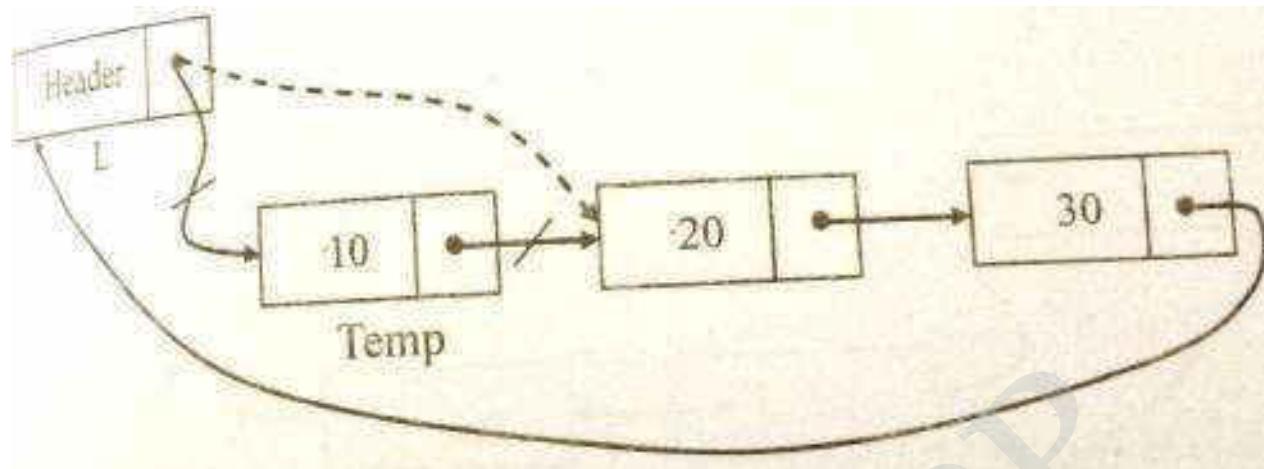


insert(40, L)



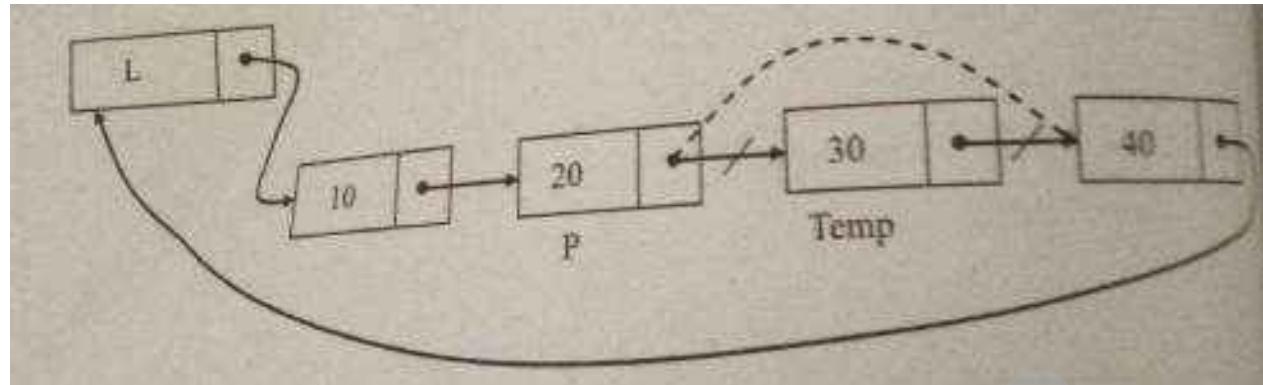
#### Routine to delete an element from the beginning

```
void del_first(List L)
{
    position temp;
    temp=L->next;
    L->next=temp->next;
    free(temp);
}
```

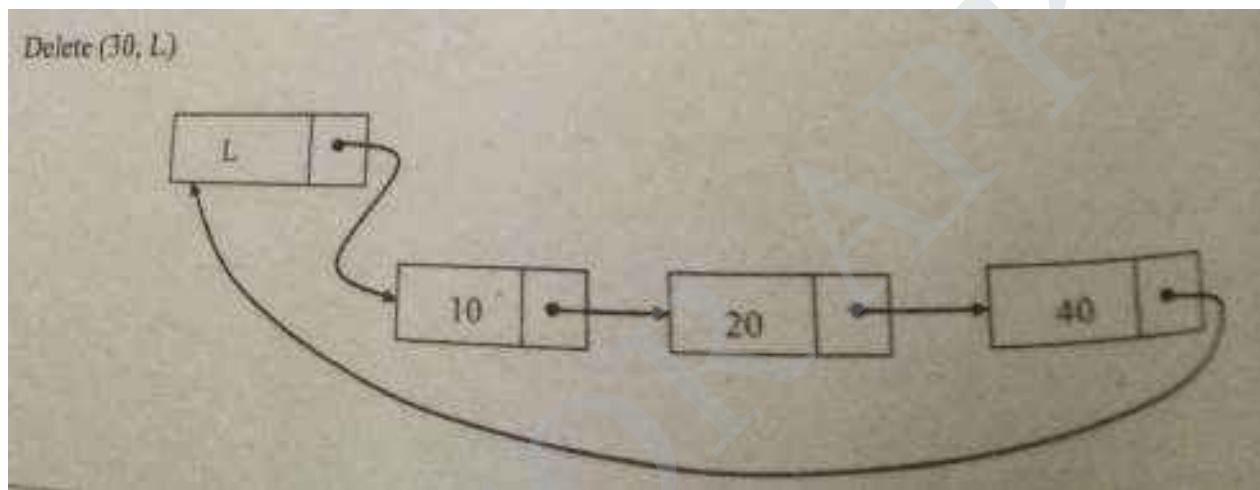


### Routine to delete an element from the middle

```
void del_mid(int X,List L)
{
position p, temp;
p=findprevious(X,L);
if(!Islast(P,L))
{
temp=p->next;
p->next=temp->next;
free(temp);
}
}
```

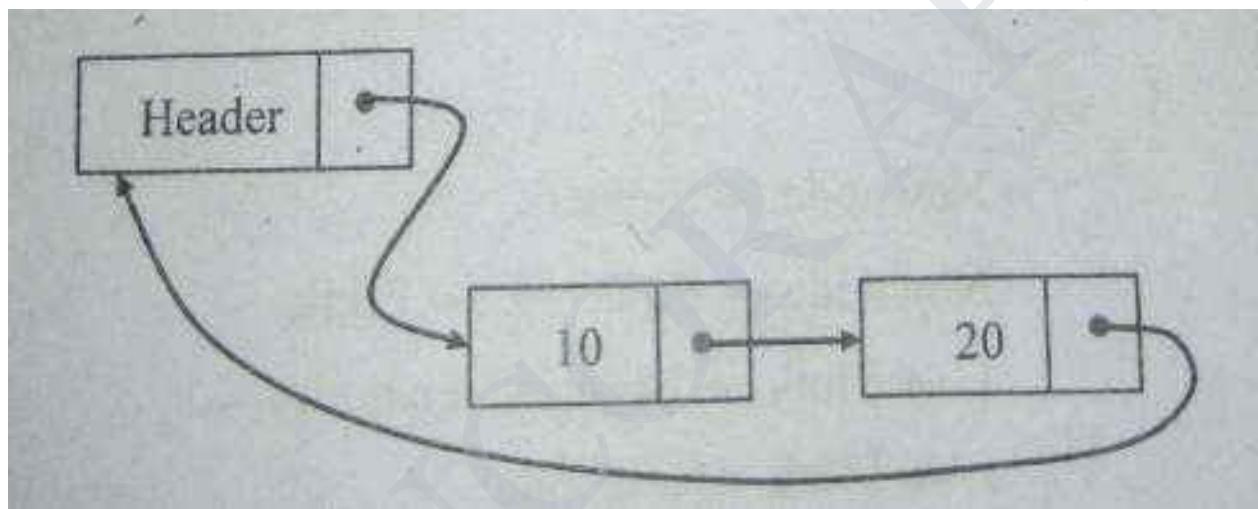
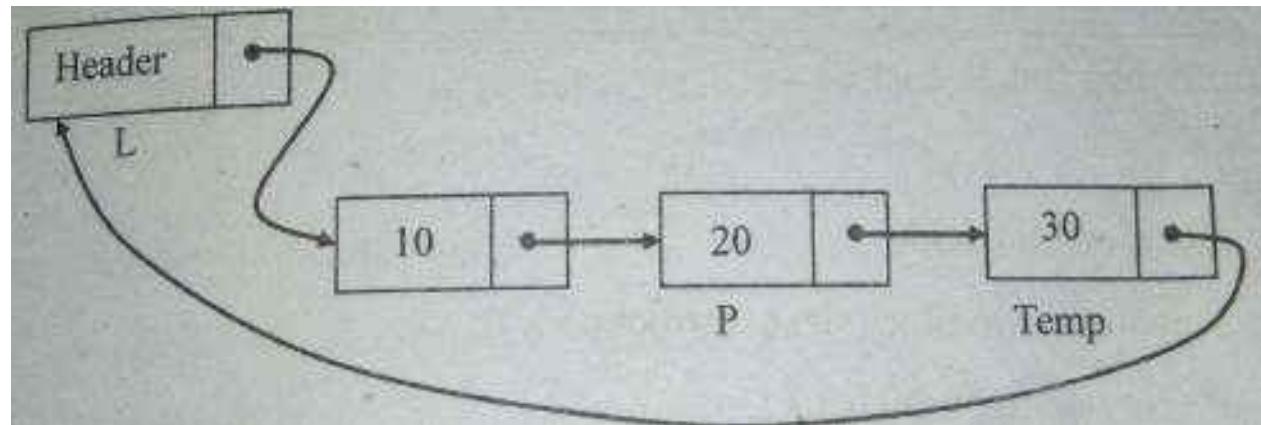


*Delete(30, L)*



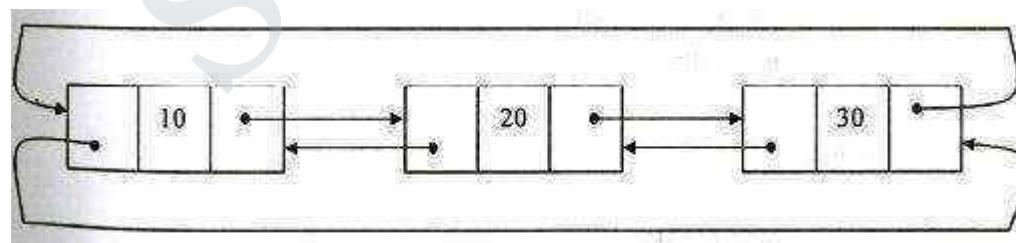
### Routine to delete an element at the last position

```
void del_last(List L)
{
    position p, temp;
    p=L;
    while(p->next->next!=L)
        p=p->next;
    temp=p->next;
    p->next=L
    free(temp);}
```



### Doubly Linked circular list:

A doubly linked circular list is a doubly linked list in which the next link of the last node points to the first node and prev link of the first node points to the last node of the list.

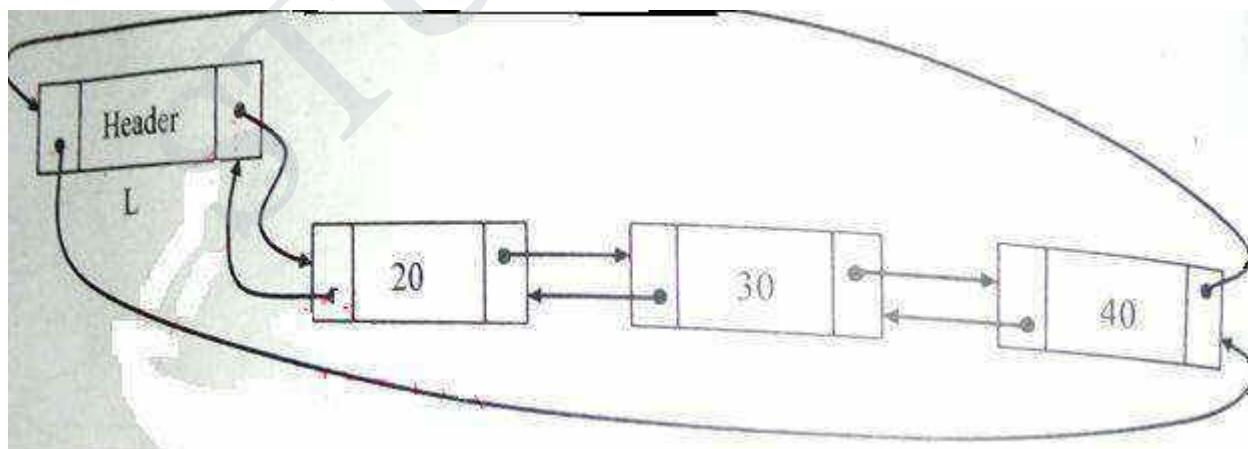


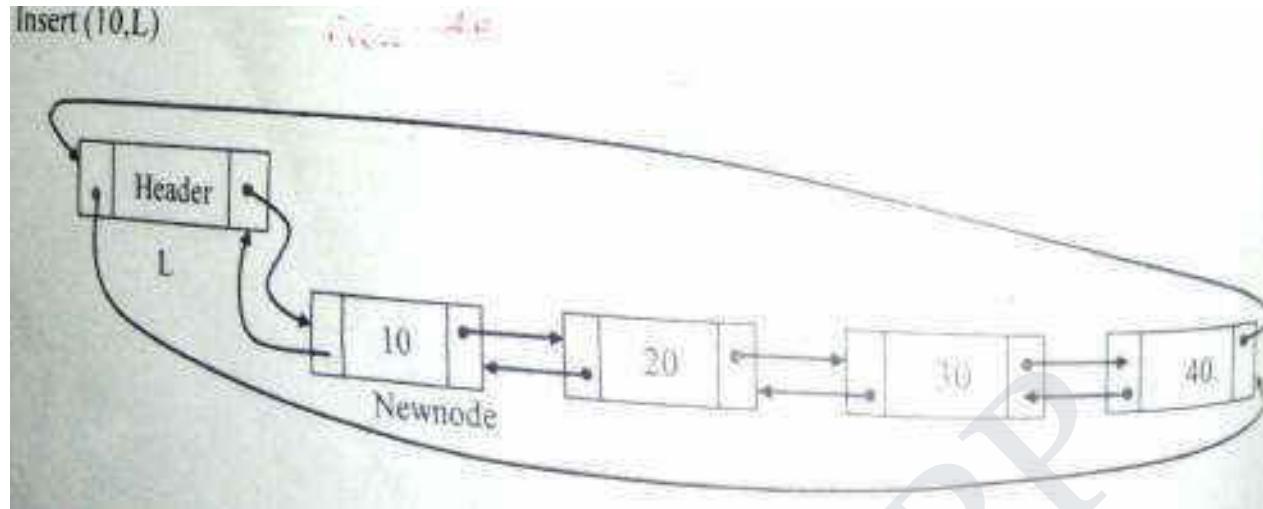
**Declaration of node:**

```
typedef struct node *position;
struct node
{
    int data;
    position next;
    position prev;
};
```

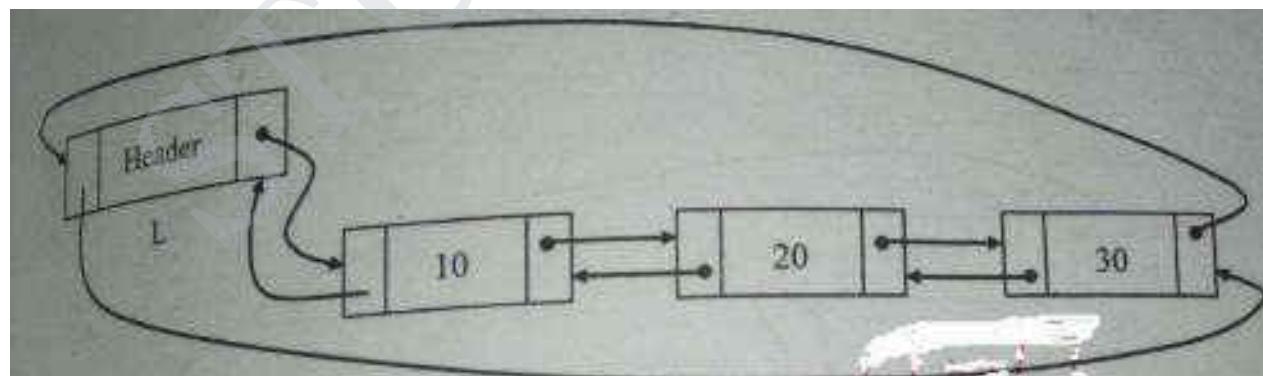
**Routine to insert an element in the beginning**

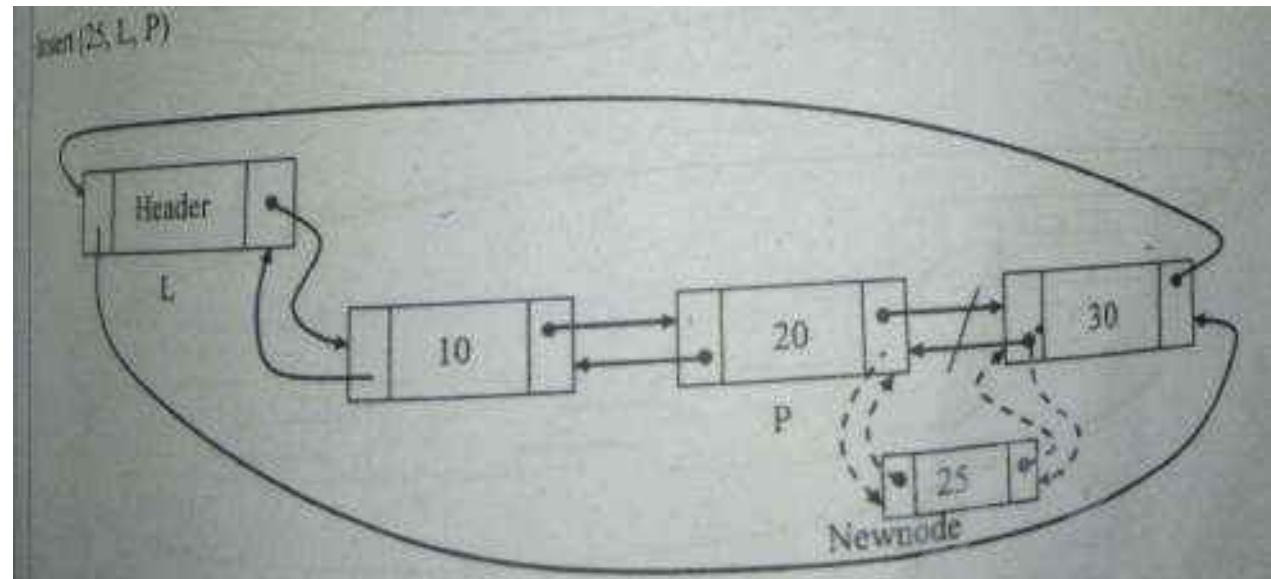
```
void insert_beg(int X,List L)
{
    position Newnode;
    Newnode=(struct node*)malloc(sizeof(struct node));
    if(Newnode!=NULL)
    {
        Newnode->data=X;
        Newnode->next=L->next;
        L->next->prev=Newnode;
        L->next=Newnode;
        Newnode->prev=L;
    }
}
```



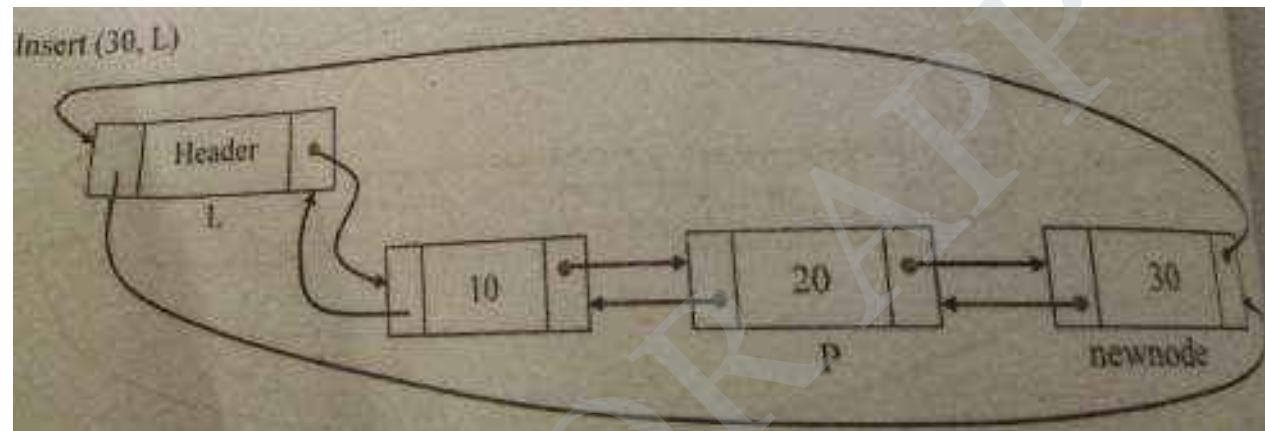
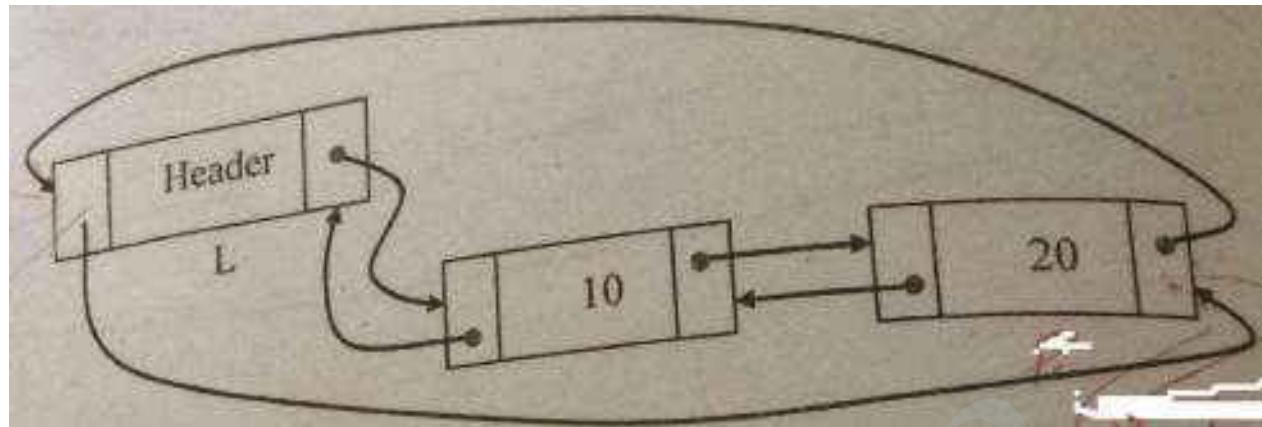
**Routine to insert an element in the middle**

```
void insert_mid(int X, List L, Position p)
{
    position Newnode;
    Newnode=(struct node*)malloc(sizeof(struct node));
    if(Newnode!=NULL)
    {
        Newnode->data=X;
        Newnode->next=p->next;
        p->next->prev=Newnode;
        p->next=Newnode;
        Newnode->prev=p;
    }
}
```



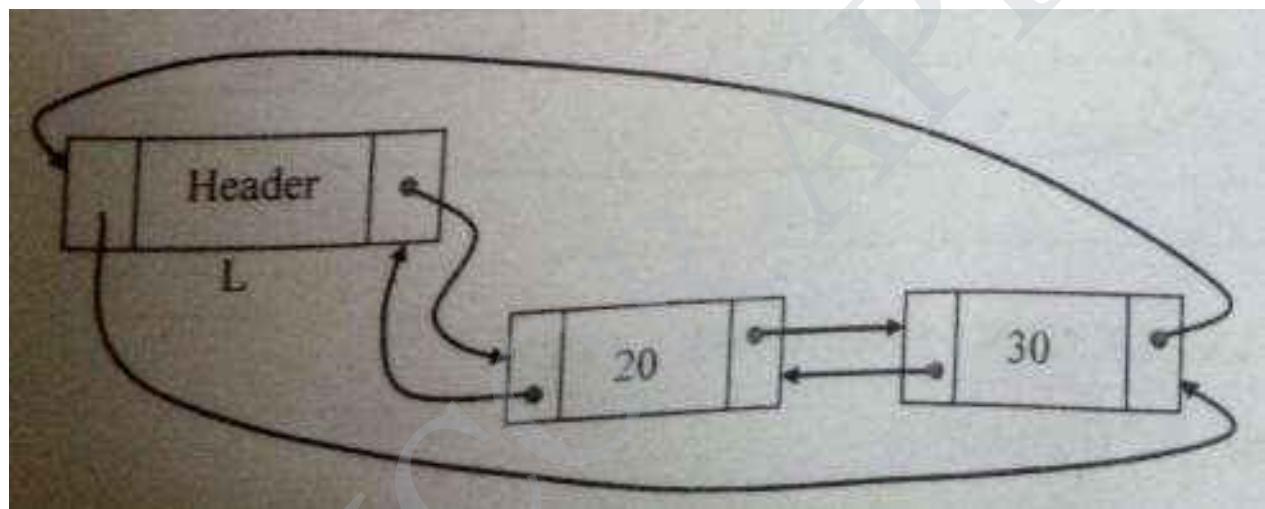
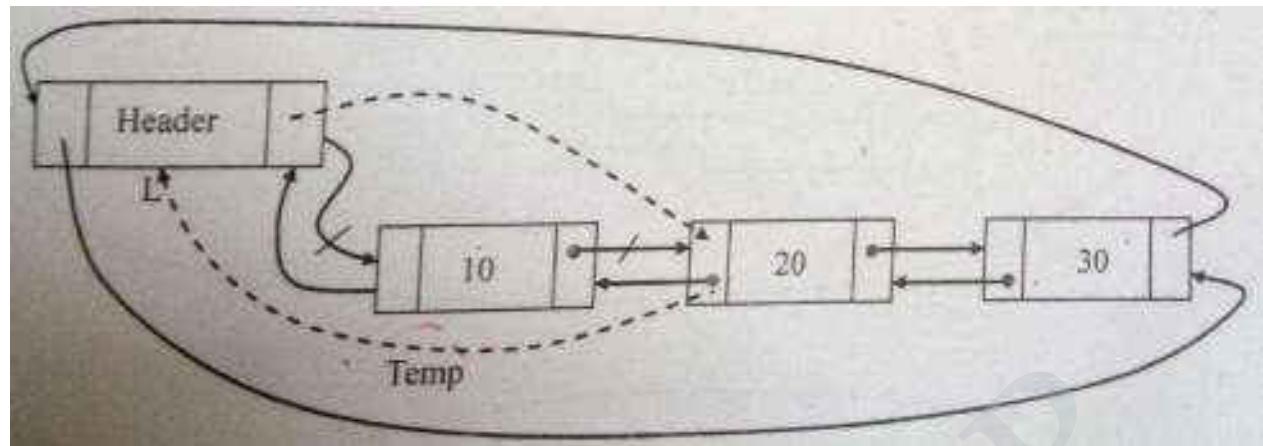
**Routine to insert an element in the last**

```
void insert_last(int X, List L)
{
    position Newnode,p;
    Newnode=(struct node*)malloc(sizeof(struct node));
    if(Newnode!=NULL)
    {
        p=L;
        while(p->next!=L)
            p=p->next;
        Newnode->data=X;
        p->next =Newnode;
        Newnode->next=L;
        Newnode->prev=p;
        L->prev=newnode;
    }
}
```



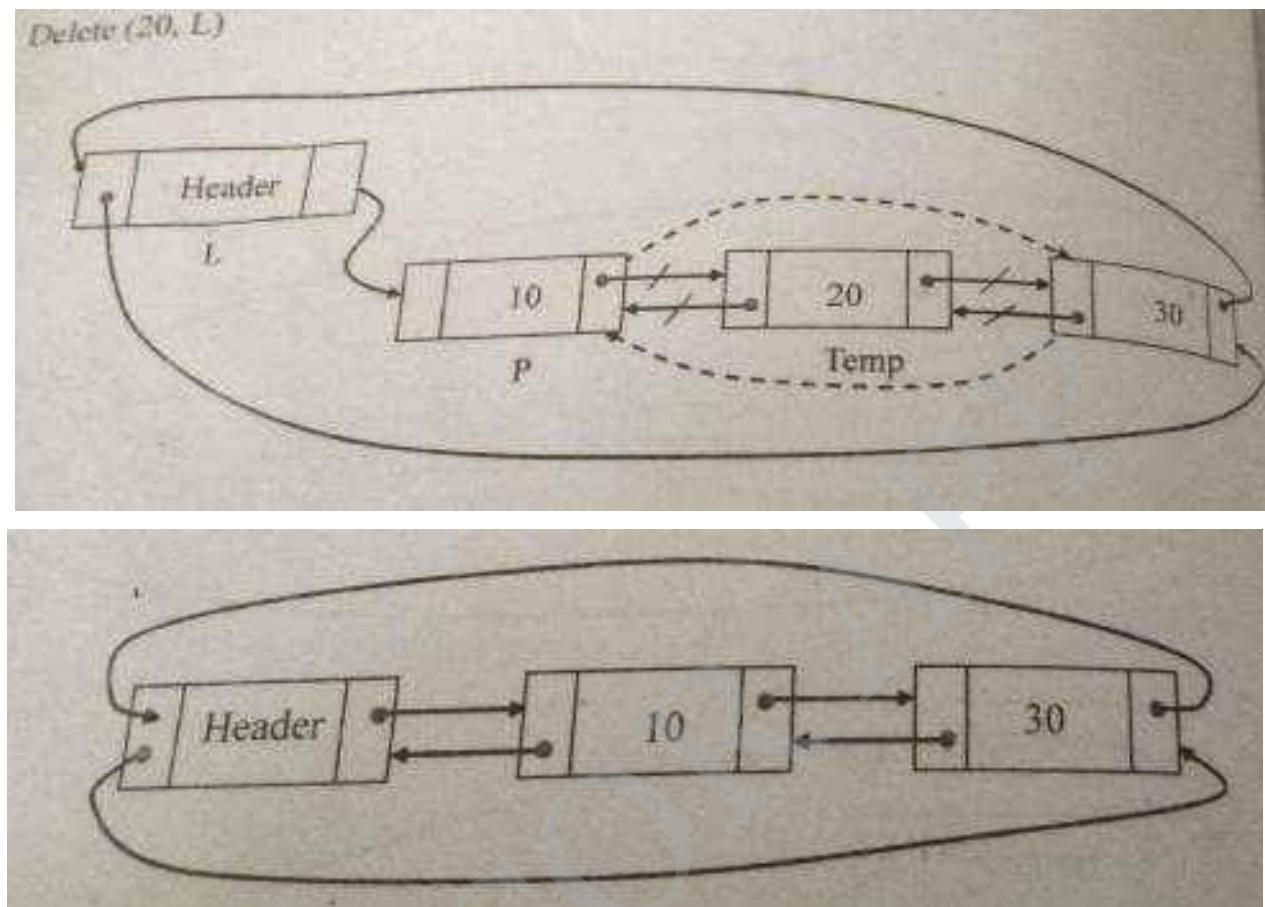
#### Routine to delete an element from the beginning

```
void del_first(List L)
{
    position temp;
    if(L->next!=NULL)
    {
        temp=L->next;
        L->next=temp->next;
        temp->next->prev=L;
        free(temp);
    }
}
```

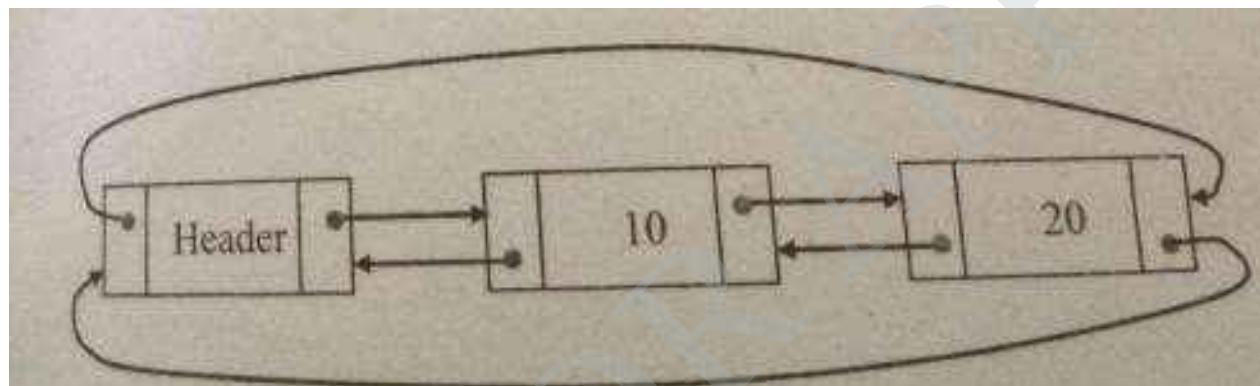
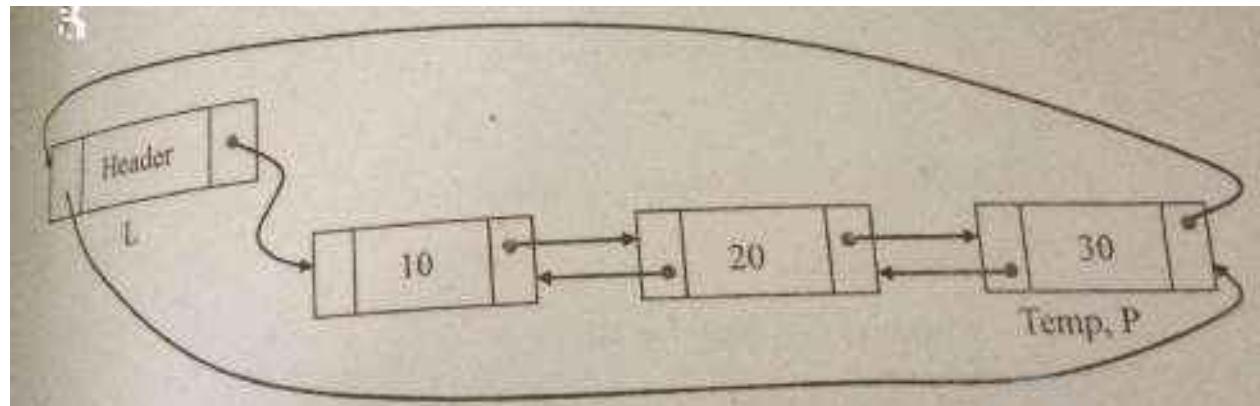


### Routine to delete an element from the middle

```
void del_mid(int X,List L)
{
Position p,temp;
p=FindPrevious(X);
if(!IsLast(p,L))
{
temp=p->next;
p->next=temp->next;
temp->next->prev=p;
free(temp);
}
}
```

**Routine to delete an element at the last position**

```
void del_last(List L)
{
    position p, temp;
    p=L;
    while(p->next!=L)
        p=p->next;
    temp=p;
    p->next->prev=L;
    L->prev=p->prev;
    free(temp);
}
```



### Advantages of Circular linked List

- It allows to traverse the list starting at any point.
- It allows quick access to the first and last records.
- Circularly doubly linked list allows to traverse the list in either direction.

### Applications of List:

- 1.Polynomial ADT
- 2.Radix sort
- 3.Multilist

### Polynomial Manipulation

Polynomial manipulations such as addition, subtraction & differentiation etc.. can be performed using linked list

**Declaration for Linked list implementation of Polynomial ADT**

```
struct poly
{
    int coeff;
    int power;
    struct poly *next;
}*list1,*list2,*list3;
```

**Creation of the Polynomial**

```
poly create(poly*head1,poly*newnode1)
{
    poly *ptr;
    if(head1==NULL)
    {
        head1=newnode1;
        return (head1);
    }
    else
    {
        ptr=head1;
        while(ptr->next!=NULL)
            ptr=ptr->next;
        ptr->next=newnode1;
    }
    return(head1);
}
```

**Addition of two polynomials**

```
void add()
{
    poly *ptr1, *ptr2, *newnode ;
    ptr1= list1;
    ptr2 = list2;
    while( ptr1 != NULL && ptr2 != NULL )
    {
        newnode = (struct poly*)malloc( sizeof( struct poly ) );
        if( ptr1 -> power == ptr2 -> power )
        {
            newnode -> coeff = ptr1 -> coeff + ptr2 -> coeff;
        }
        else if( ptr1 -> power < ptr2 -> power )
        {
            newnode -> coeff = ptr1 -> coeff;
            newnode -> power = ptr2 -> power;
        }
        else
        {
            newnode -> coeff = ptr2 -> coeff;
            newnode -> power = ptr1 -> power;
        }
        ptr1 = newnode;
    }
}
```

```
        newnode -> power = ptr1 -> power ;  
newnode -> next = NULL;  
list3 = create( list3, newnode );  
ptr1 = ptr1 -> next;  
ptr2 = ptr2 -> next;  
}  
else if(ptr1 -> power > ptr2 -> power )  
{  
    newnode -> coeff = ptr1 -> coeff;  
    newnode -> power = ptr1 -> power;  
    newnode -> next = NULL;  
    list3 = create( list3, newnode );  
    ptr1 = ptr1 -> next;  
}  
else  
{  
    newnode -> coeff = ptr2 -> coeff;  
    newnode -> power = ptr2 -> power;  
    newnode -> next = NULL;  
    list3 = create( list3, newnode );  
ptr2 = ptr2 -> next;  
}}}
```

### Subtraction of two polynomial

```
void sub() poly *ptr1, *ptr2, *newnode;  
ptr1 = list1;  
ptr2 = list2;  
while( ptr1 != NULL && ptr2 != NULL )  
{  
    newnode = (struct poly*)malloc( sizeof( struct poly) ) ;  
    if(ptr1->power==ptr2->power)  
    {
```

```
    newnode->coeff=(ptr1-coeff)-(ptr2->coeff);
    newnode->power=ptr1->power;
newnode->next=NULL;
list3=create(list3,newnode);
ptr1=ptr1->next;

ptr2=ptr2->next;
}
else
{
    if(ptr1-power>ptr2-power)
    {
        newnode->coeff=ptr1->coeff;
        newnode->power=ptr1->power;
        newnode->next=NULL;
        list3=create(list3,newnode);
        ptr1=ptr1->next;
    }
else
{
    newnode->coeff=-(ptr2->coeff);
    newnode->power=ptr2->power;
    newnode->next=NULL;
    list3=create(list3,newnode);
    ptr2=ptr2->next;
}
}
}
```

### Polynomial Differentiation:

```
void diff()
{
    poly *ptr1, *newnode;
    ptr1 = list1;
    while( ptr1 != NULL)
    {
        newnode = (struct poly*)malloc( sizeof (struct poly));
        newnode->coeff=(ptr1-coeff)*(ptr1->power);
        newnode->power=ptr1->power-1;
```

```
newnode->next=NULL;  
list3=create(list3,newnode);  
ptr1=ptr1->next;  
}  
}
```

### Polynomial Multiplication

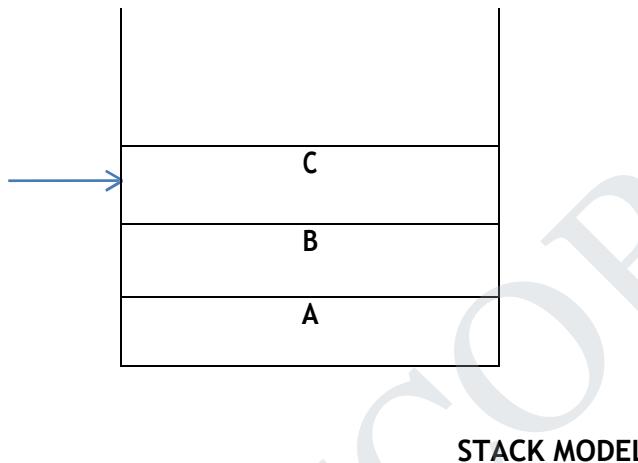
```
void mul()  
{  
    poly *ptr1, *ptr2, *newnode ;  
    ptr1= list1;  
    ptr2 = list2;  
    while( ptr1 != NULL && ptr2 != NULL )  
    {  
        newnode = (struct poly*)malloc( sizeof( struct poly ));  
        if( ptr1 -> power == ptr2 -> power )  
        {  
            newnode -> coeff = ptr1 -> coeff * ptr2 -> coeff;  
            newnode -> power = ptr1 -> power+ptr2->power; ;  
            newnode -> next = NULL;  
            list3 = create( list3, newnode );  
            ptr1 = ptr1 -> next;  
            ptr2 = ptr2 -> next;  
        }  
    }  
}
```

**UNIT II LINEAR DATA STRUCTURES – STACKS, QUEUES****9**

Stack ADT – Evaluating arithmetic expressions- other applications- Queue ADT – circular queue implementation – Double ended Queues – applications of queues

**STACK**

- Stack is a Linear Data Structure that follows Last In First Out(LIFO) principle.
- Insertion and deletion can be done at only one end of the stack called TOP of the stack.
- Example: - Pile of coins, stack of trays

**STACK ADT:****TOP pointer**

It will always point to the last element inserted in the stack.

For empty stack, top will be pointing to -1. ( $\text{TOP} = -1$ )

**Operations on Stack (Stack ADT)**

Two fundamental operations performed on the stack are PUSH and POP.

**(a) PUSH:**

It is the process of inserting a new element at the Top of the stack.

For every push operation:

1. Check for Full stack ( overflow ).
2. Increment Top by 1. ( $\text{Top} = \text{Top} + 1$ )

3. Insert the element X in the Top of the stack.

(b) POP:

It is the process of deleting the Top element of the stack.

For every pop operation:

1. Check for Empty stack ( underflow ).
2. Delete (pop) the Top element X from the stack
3. Decrement the Top by 1. ( $\text{Top} = \text{Top} - 1$  )

### Exceptional Conditions of stack

#### 1. Stack Overflow

- An Attempt to insert an element X when the stack is Full, is said to be stack overflow.
- For every Push operation, we need to check this condition.

#### 2. Stack Underflow:

- An Attempt to delete an element when the stack is empty, is said to be stack underflow.
- For every Pop operation, we need to check this condition.

### 12.4 Implementation of Stack

Stack can be implemented in 2 ways.

1. Static Implementation (Array implementation of Stack)
2. Dynamic Implementation (Linked List Implementation of Stack)

#### 12.4.1 Array Implementation of Stack

- Each stack is associated with a Top pointer.
- For Empty stack,  $\text{Top} = -1$ .
- Stack is declared with its maximum size.

#### Array Declaration of Stack:

```
#define ArraySize 5
int S [ Array Size];
or
int S [ 5 ];
```

**(i) Stack Empty Operation:**

- Initially Stack is Empty.
- With Empty stack Top pointer points to - 1.
- It is necessary to check for Empty Stack before deleting (pop) an element from the stack.

**Routine to check whether stack is empty**

```
int IsEmpty ( Stack S )  
{  
    if( Top == - 1 )  
        return(1);  
}
```

**(ii) Stack Full Operation:**

- As we keep inserting the elements, the Stack gets filled with the elements.
- Hence it is necessary to check whether the stack is full or not before inserting a new element into the stack.

**Routine to check whether a stack is full**

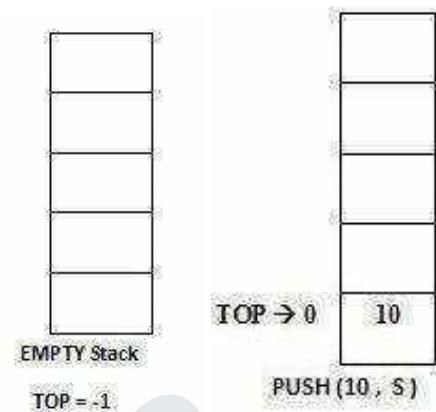
```
int IsFull ( Stack S )  
{  
    if( Top == Arraysize - 1 )  
        return(1);  
}
```

**(ii) Push Operation**

- It is the process of inserting a new element at the Top of the stack.
- It takes two parameters. Push(X, S) the element X to be inserted at the Top of the Stack S.
- Before inserting an Element into the stack, check for Full Stack.
- If the Stack is already Full, Insertion is not possible.
- Otherwise, Increment the Top pointer by 1 and then insert the element X at the Top of the Stack.

**Routine to push an element into the stack**

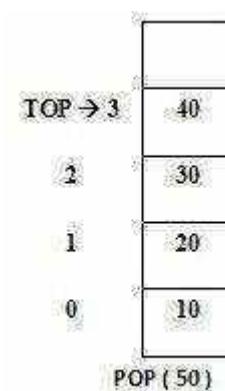
```
void Push ( int X , Stack S )
{
    if ( Top == Arraysize - 1 )
        Error("Stack is full!!Insertion is not possible");
    else
        {
            Top = Top + 1;
            S [ Top ] =X;
        }
}
```

**(iv) Pop Operation**

- It is the process of deleting the Top element of the stack.
- It takes only one parameter. Pop(X).The element X to be deleted from the Top of the Stack.
- Before deleting the Top element of the stack, check for Empty Stack.
- If the Stack is Empty, deletion is not possible.
- Otherwise, delete the Top element from the Stack and then decrement the Top pointer by 1.

**Routine to Pop the Top element of the stack**

```
void Pop ( Stack S )
{
    if ( Top == - 1 )
        Error ( "Empty stack! Deletion not possible");
    else
        {
            X = S [ Top ];
            Top = Top - 1 ;
        }
}
```



**(v) Return Top Element**

- Pop routine deletes the Top element in the stack.
- If the user needs to know the last element inserted into the stack, then the user can return the Top element of the stack.
- To do this, first check for Empty Stack.
- If the stack is empty, then there is no element in the stack.
- Otherwise, return the element which is pointed by the Top pointer in the Stack.

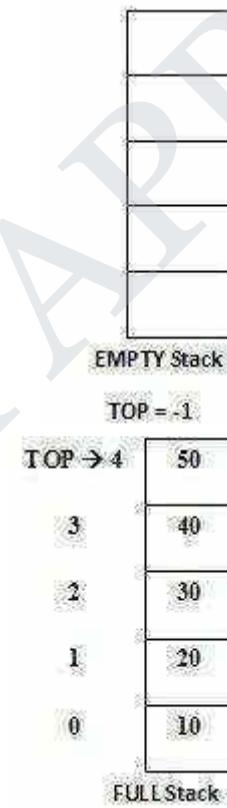
**Routine to return top Element of the stack**

```
int TopElement(Stack S)
{
    if(Top== -1)
    {
        Error("Empty stack!!No elements");
        return 0;
    }
    else
        return S[Top];
}
```

**Implementation of stack using Array**

```
/* static implementation of stack */

#include<stdio.h>
#include<conio.h>
#define size 5
int stack [ size ];
int top;
void push( )
{
    int n ;
    printf( "\n Enter item in stack" );
    scanf( " %d " , &n ) ;
    if( top == size - 1)
    {
        printf( "\nStack is Full" );
    }
    else
    {
        top = top + 1 ;
```



```
        stack [ top ] = n ;
    }
}
void pop( )
{
    int item;
    if( top == - 1)
    {
        printf( "\n Stack is empty" );
    }
    else
    {
        item = stack[ top ] ;
        printf( "\n item popped is = %d" , item );
        top - ;
    }
}
void display( )
{
    int i;
    printf("\n item in stack are");
    for(i = top; i > = 0; i --)
        printf("\n %d", stack[ i ] );
}
void main( )
{
    char ch,ch1;
    ch = 'y';
    ch1 = 'y';
    top = -1;
    clrscr();
    while(ch !='n')
    {
        push( );
        printf("\n Do you want to push any item in stack y/n");
        ch=getch();
    }
    display( );
    while( ch1!= 'n' )
    {
        printf("\n Do you want to delete any item in stack y/n");
        ch1=getch();
        pop( );
    }
    display( );
    getch();
}
```

**OUTPUT:**

Enter item in stack20  
Do you want to push any item in stack y/n  
Enter item in stack25  
Do you want to push any item in stack y/n  
Enter item in stack30  
Stack is Full  
Do you want to push any item in stack y/n  
item in stack are  
25  
20  
15  
10  
5  
Do you want to delete any item in stack y/n  
item popped is = 25  
Do you want to delete any item in stack y/n  
item popped is = 20  
Do you want to delete any item in stack y/n  
item popped is = 15  
item in stack are  
10  
5

**Linked list implementation of Stack**

- Stack elements are implemented using SLL (Singly Linked List) concept.
- Dynamically, memory is allocated to each element of the stack as a node.

**Type Declarations for Stack using SLL**

```
struct node;  
  
typedef struct node *stack;  
  
typedef struct node *position;  
  
stack S;  
  
struct node{ int  
    data; position  
    next;};  
  
int IsEmpty(Stack S);  
  
void Push(int x, Stack S);  
  
void Pop(Stack S);  
  
int TopElement(Stack S);
```

**(i) Stack Empty Operation:**

- Initially Stack is Empty.
- With Linked List implementation, Empty stack is represented as  $S \rightarrow \text{next} = \text{NULL}$ .
- It is necessary to check for Empty Stack before deleting ( pop) an element from the stack.

**Routine to check whether the stack is empty**

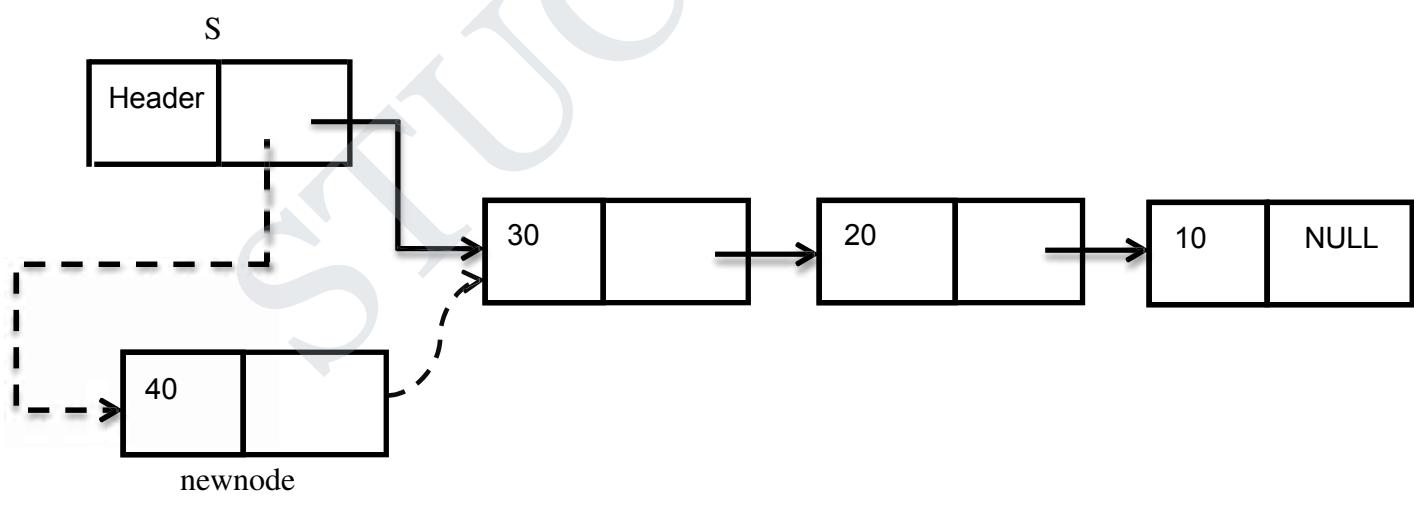
```
int IsEmpty( Stack S )
{
    if ( S -> next == NULL)
        return ( 1 );
}
```



EMPTY STACK

**(ii) Push Operation**

- It is the process of inserting a new element at the Top of the stack.
- With Linked List implementation, a new element is always inserted at the Front of the List.(i.e.)  $S \rightarrow \text{next}$ .
- It takes two parameters. Push(X, S) the element X to be inserted at the Top of the StackS.
- Allocate the memory for the newnode to be inserted.
- Insert the element in the data field of the newnode.
- Update the next field of the newnode with the address of the next node which is stored in the  $S \rightarrow \text{next}$ .

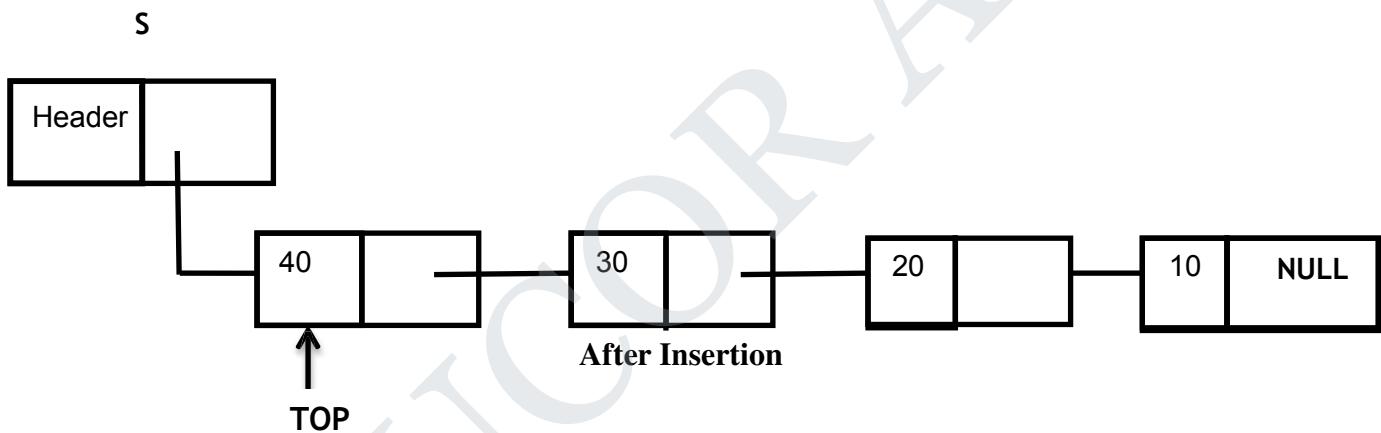


Before Insertion

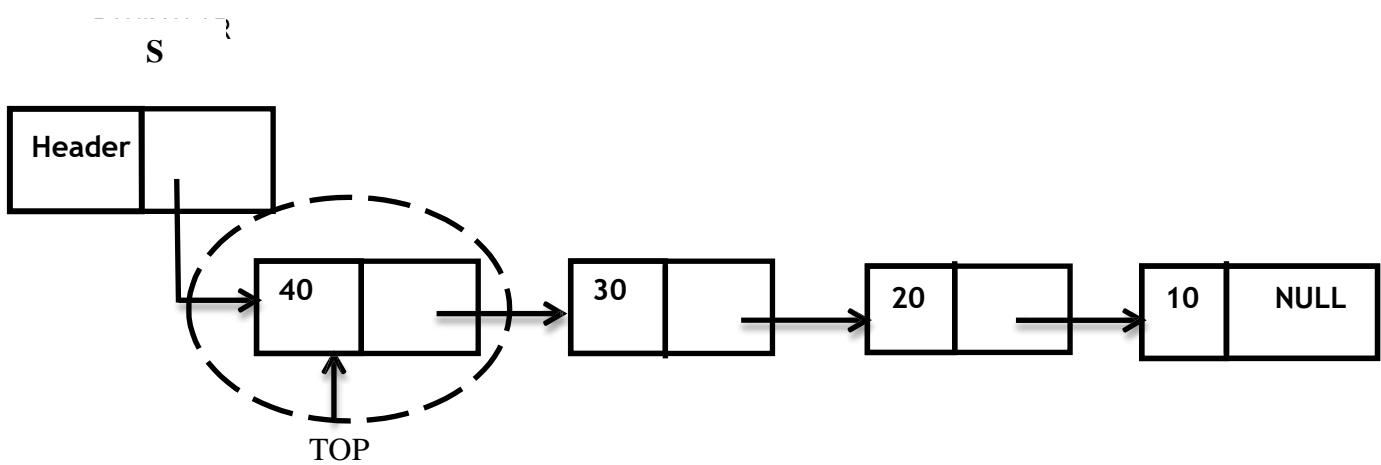
**Push routine**

/\*Inserts element at front of the list

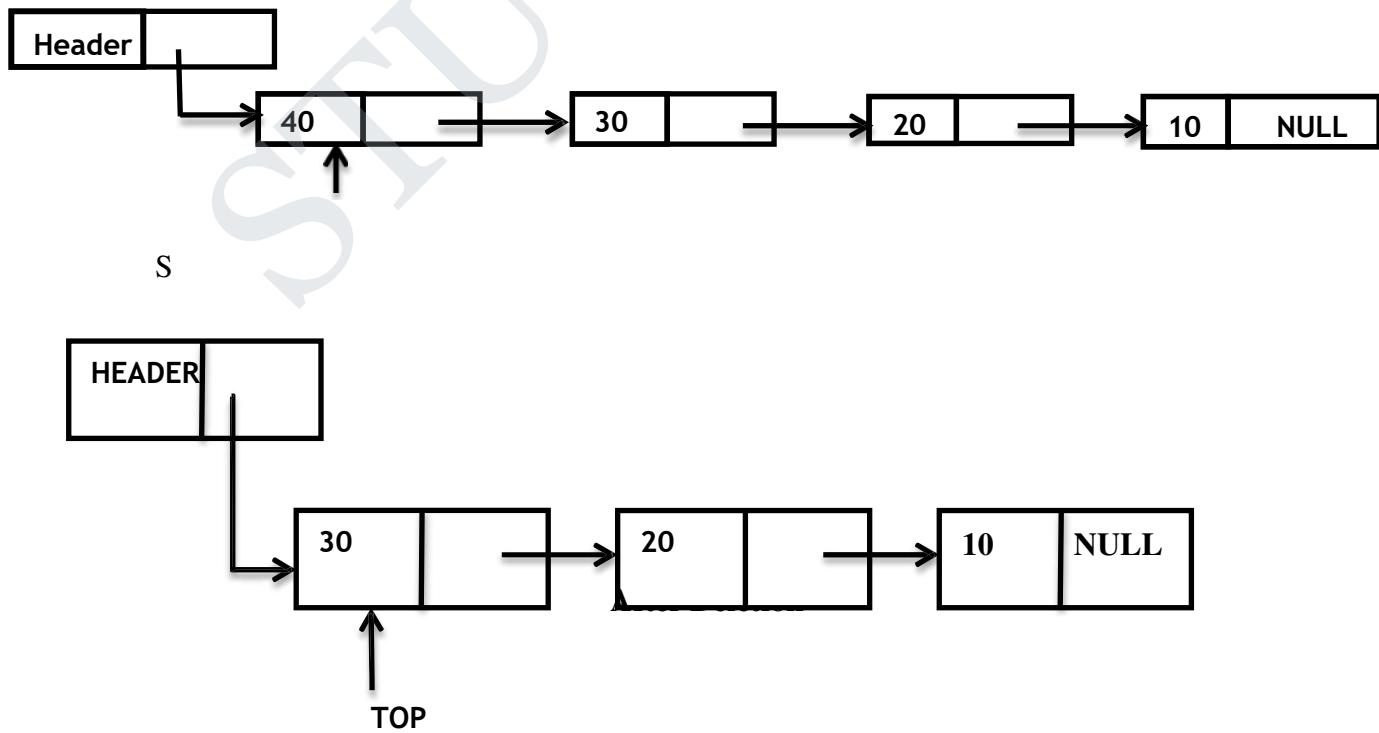
```
void push(int X, Stack S)
{
    Position newnode, Top;
    newnode = malloc (sizeof( struct node ) );
    newnode -> data = X;
    newnode -> next = S -> next;
    S -> next = newnode;
    Top = newnode;
}
```

**(iii) Pop Operation**

- It is the process of deleting the Top element of the stack.
- With Linked List implementations, the element at the Front of the List (i.e.)  $S \rightarrow \text{next}$  is always deleted.
- It takes only one parameter.  $\text{Pop}(X)$ . The element  $X$  to be deleted from the Front of the List.
- Before deleting the front element in the list, check for Empty Stack.
- If the Stack is Empty, deletion is not possible.
- Otherwise, make the front element in the list as “temp”.
- Update the next field of header.
- Using free ( ) function, Deallocate the memory allocated for temp node.

**Before Deletion****Pop routine****/\*Deletes the element at front of list**

```
void Pop( Stack S )
{
    Position temp, Top;
    Top = S -> next;
    if( S -> next == NULL)
        Error("empty stack! Pop not possible");
    else
    {
        Temp = S -> next;
        S -> next = temp -> next;
        free(temp);
        Top = S -> next;
    }
}
```



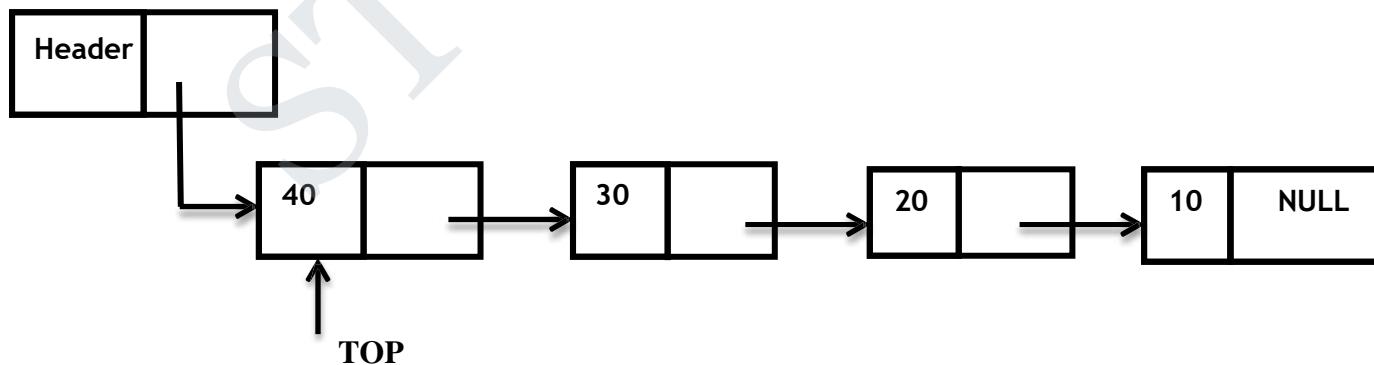
**(iv) Return Top Element**

- Pop routine deletes the Front element in the List.
- If the user needs to know the last element inserted into the stack, then the user can return the Top element of the stack.
- To do this, first check for Empty Stack.
- If the stack is empty, then there is no element in the stack.
- Otherwise, return the element present in the  $S \rightarrow \text{next} \rightarrow \text{data}$  in the List.

**Routine to Return Top Element**

```
int TopElement(Stack S)
{
    if(S->next==NULL)
    {
        error("Stack is empty");
        return 0;
    }
    else
        return S->next->data;
}
```

S



**Implementation of stack using 'Linked List'**

```
/* Dynamic implementation of stack*/
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
typedef struct node * position ;
struct node
{
    int data ;
    position next ;
} ;
void create( ) ;
void push( ) ;
void pop( ) ;
void display( ) ;
position s, newnode, temp, top ; /* Global Declarations */
void main() {
    /* Main Program */
    int op ;
    clrscr( ) ;
    do {
        printf( "\n ### Linked List Implementation of STACK Operations ### \n\n" ) ;
        printf( "\n Press 1-create\n 2-Push\n 3-Pop\n 4-Display\n 5-Exit\n" ) ;
        printf( "\n Your option ? " ) ;
        scanf( " % d ", & op ) ;
        switch (op) {
            case 1:
                create( ) ;
                break ;
            case 2:
                push();
                break;
            case 3:
                pop();
                break;
            case 4:
                display();
                break;
            case 5:
                exit(0);
        }
    }while(op<5);
```

```
getch();
}
void create()
{
    int n,i;
    s=NULL;
    printf("Enter the no of nodes to be created\n");
    scanf("%d",&n);
    newnode=(struct node*)malloc(sizeof(struct node));
    printf("Enter the data\t");
    scanf("%d",&newnode->data);
    newnode->next=NULL;
    top=newnode;
    s=newnode;
    for(i=2;i<=n;i++)
    {
        newnode=(struct node*)malloc(sizeof(struct node));
        printf("Enter the data\t");
        scanf("%d",&newnode->data);
        newnode->next=top;
        s=newnode;
        top=newnode;
    }
}
void display()
{
    top=s;
    while(top!=NULL)
    {
        printf("%d->",top->data);
        top=top->next;
    }
    printf("NULL\n");
}
void push()
{
    top=s;
    newnode=(struct node*)malloc(sizeof(struct node));
    printf("Enter the data\t");
    scanf("%d",&newnode->data);
    newnode->next=top;
    top=newnode;
    s=newnode;
    display();
}
void pop()
{
    top=s;
```

```
if(top==NULL)
printf("Empty stack\n\n");
else
{
temp=top;
printf("Deleted element is %d\n\n",top->data);
s=top->next;
free(temp);
display();
}
```

### Output

### Linked List Implementation of STACK Operations ###

Press 1-create

2-Push

3-Pop

4-Display

5-Exit

Your option ? 1

Enter the no of nodes to be created5

Enter the data 10

Enter the data20

Enter the data30

Enter the data40

Enter the data50

### Linked List Implementation of STACK Operations ###

Press 1-create

2-Push

3-Pop

4-Display

5-Exit

Your option ? 4

50->40->30->20->10->NULL

### Linked List Implementation of STACK Operations ###

Press 1-create

2-Push

3-Pop

4-Display

5-Exit

Your option ?2

Enter the data60

Your option ? 4

60->50->40->30->20->10->NULL

Your option ?2

Enter the data70

Your option ? 4

70->60->50->40->30->20->10->NULL

Your option ?3

Deleted element is70

Your option ? 4

50->40->30->20->10->NULL

## Applications of Stack

The following are some of the applications of stack:

1. Evaluating the arithmetic expressions
  - o Conversion of Infix to Postfix Expression
  - o Evaluating the Postfix Expression
2. Balancing the Symbols
3. Function Call
4. Tower of Hanoi
5. 8 Queen Problem

## Evaluating the Arithmetic Expression

There are 3 types of Expressions

- Infix Expression
- Postfix Expression
- Prefix Expression

### INFIX:

The arithmetic operator appears between the two operands to which it is being applied.

A / B + C

### POSTFIX:

The arithmetic operator appears directly after the two operands to which it applies.  
Also called reverse polish notation.

$((A / B) + C)$  $AB/C +$ **PREFIX:**

The arithmetic operator is placed before the two operands to which it applies. Also called polish notation.

 $((A/B) + C)$   
+/ABC**Evaluating Arithmetic Expressions**

1. Convert the given infix expression to Postfix expression
2. Evaluate the postfix expression using stack.

**Algorithm to convert Infix Expression to Postfix Expression:**

Read the infix expression one character at a time until it encounters the delimiter “#”

Step 1: If the character is an operand, place it on the output.

Step 2: If the character is an operator, push it onto the stack. If the stack operator has a higher or equal priority than input operator then pop that operator from the stack and place it onto the output.

Step 3: If the character is left parenthesis, push it onto the stack

Step 4: If the character is a right parenthesis, pop all the operators from the stack till it encounters left parenthesis, discard both the parenthesis in the output.

E.g. Consider the following Infix expression: - A\*B+(C-D/E)#

Read char	Stack	Output
A		A
*	*	A
B	+	AB
+	+	AB*
(	(	AB*

Read char	Stack	Output
C		<div style="border: 1px solid black; padding: 5px;">AB*C</div>
-		<div style="border: 1px solid black; padding: 5px;">AB*C</div>
D		<div style="border: 1px solid black; padding: 5px;">AB*CD</div>
/		<div style="border: 1px solid black; padding: 5px;">AB*CD</div>
E		<div style="border: 1px solid black; padding: 5px;">AB*CDE</div>
)		<div style="border: 1px solid black; padding: 5px;">AB*CDE/-</div>

Read char	Stack	Output
#		AB*CDE/-+

**Output: Postfix expression:- AB\*CDE/-+**

### Evaluating the Postfix Expression

#### Algorithm to evaluate the obtained Postfix Expression

Read the postfix expression one character at a time until it encounters the delimiter „#“

Step 1: If the character is an operand, push its associated value onto the stack.

Step 2: If the character is an operator, POP two values from the stack, apply the operator to them and push the result onto the stack.

E.g consider the obtained Postfix expression:- AB\*CDE/-+

Operand	Value
A	2
B	3
C	4
D	4
E	2

Char Read	Stack
A	2
B	3 2

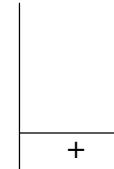
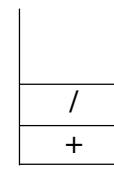
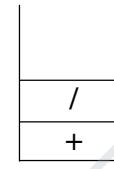
Char Read	Stack
*	6
C	4 6
D	4 4 6
E	
/	2 4 6
-	2 6
+	8

OUTPUT = 8

**Example 2: Infix expression:- (a+b)\*c/d+e/f#**

Read char	Stack	Output
(	(	
a	(	a

+	<table border="1"><tr><td></td><td>+</td></tr><tr><td>(</td><td></td></tr></table>		+	(		a
	+					
(						
b	<table border="1"><tr><td></td><td>+</td></tr><tr><td>(</td><td></td></tr></table>		+	(		ab
	+					
(						
)	<table border="1"><tr><td></td><td></td></tr><tr><td></td><td></td></tr></table>					ab+
*	<table border="1"><tr><td></td><td>*</td></tr><tr><td></td><td></td></tr></table>		*			ab+
	*					
c	<table border="1"><tr><td></td><td>*</td></tr><tr><td></td><td></td></tr></table>		*			ab+c
	*					
/	<table border="1"><tr><td></td><td>/</td></tr><tr><td></td><td></td></tr></table>		/			ab+c*
	/					
d	<table border="1"><tr><td></td><td>/</td></tr><tr><td></td><td></td></tr></table>		/			ab+c*d
	/					
+	<table border="1"><tr><td></td><td>+</td></tr><tr><td></td><td></td></tr></table>		+			ab+c*d/
	+					

e		<div style="border: 1px solid black; padding: 5px;">ab+c*d/e</div>
/		<div style="border: 1px solid black; padding: 5px;">ab+c*d/e</div>
f		<div style="border: 1px solid black; padding: 5px;">ab+c*d/ef</div>
#		<div style="border: 1px solid black; padding: 5px;">ab+c*d/ef/+</div>

Postfix expression:- ab+c\*d/ef/+

### Evaluating the Postfix Expression

Operand	Value
a	1
b	2
c	4
d	2
e	6
f	3

Char Read	Stack
a	1
b	2 1
+	3
c	4 3
*	12
d	2 12
/	6
e	6 6
F	3 6 6
/	2 6
+	8

Output = 8

Infix to Postfix Conversion	Output
<pre>#define SIZE 50      /* Size of Stack */ #include &lt;ctype.h&gt; char s[SIZE]; int top=-1;      /* Global declarations */  void push(char elem) {     s[++top]=elem; }  char pop() {     return(s[top--]); }  int pr(char elem) {           /* Function for precedence */ switch(elem) { case '#': return 0; case '(': return 1; case '+': case '-': return 2; case '*': case '/': return 3; } return 0; }  Void main() {           /* Main Program */ char infix[50],pofx[50],ch,elem; int i=0,k=0; printf("\nRead the Infix Expression ? "); scanf("%s",infx); push('#'); while( (ch=infx[i++]) != '\0') {     if( ch == '(') push(ch);     else         if(isalnum(ch)) pofx[k++]=ch;         else             if( ch == ')')             {                 while( s[top] != '(') </pre>	<p><b>Read the Infix Expression ?</b></p> <p><b>(a+b)-(c-d)</b></p> <p><b>Given Infix Expn: (a+b)*(c-d)</b></p> <p><b>Postfix Expn: ab+cd-*</b></p>

```
        pofx[k++]=pop();
        elem=pop(); /* Remove */
    }
else
{
    /* Operator */
    while( pr(s[top]) >= pr(ch) )
        pofx[k++]=pop();
    push(ch);
}
while( s[top] != '#' ) /* Pop from stack
till empty */
    pofx[k++]=pop();
pofx[k]='\0'; /* Make pofx as valid
string */
printf("\n\nGiven Infix Expn: %s Postfix
Expn: %s\n",infx,pofx);
}
```

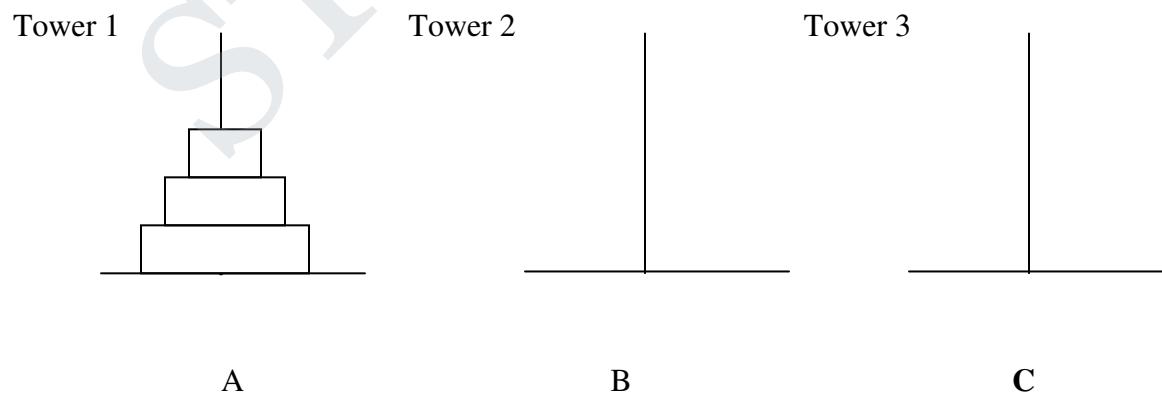
## Towers of Hanoi

Towers of Hanoi can be easily implemented using recursion. Objective of the problem is moving a collection of N disks of decreasing size from one pillar to another pillar. The movement of the disk is restricted by the following rules.

Rule 1 : Only one disk could be moved at a time.

Rule 2 : No larger disk could ever reside on a pillar on top of a smaller disk.

Rule 3 : A 3rd pillar could be used as an intermediate to store one or more disks, while they were being moved from source to destination.



**Initial Setup of Tower of Hanoi**

## Recursive Solution

**N - represents the number of disks.**

Step 1. If  $N = 1$ , move the disk from A to C.

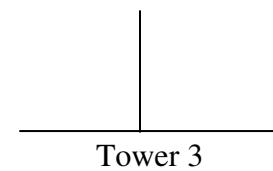
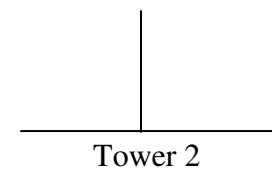
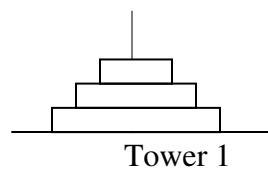
Step 2. If  $N = 2$ , move the 1<sup>st</sup> disk from A to B. Then move the 2<sup>nd</sup> disk from A to C, Then move the 1<sup>st</sup> disk from B to C.

Step 3. If  $N = 3$ , Repeat the step (2) to move the first 2 disks from A to B using C as intermediate. Then the 3<sup>rd</sup> disk is moved from A to C. Then repeat the step (2) to move 2 disks from B to C using A as intermediate.

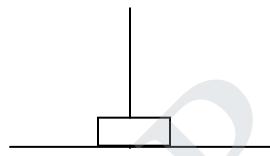
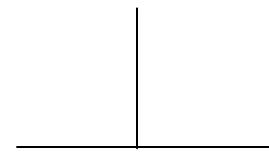
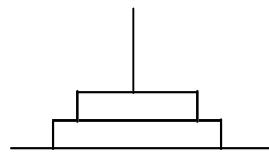
In general, to move N disks. Apply the recursive technique to move  $N - 1$  disks from A to B using C as an intermediate. Then move the  $N^{\text{th}}$  disk from A to C. Then again apply the recursive technique to move  $N - 1$  disks from B to C using A as an intermediate

## Recursive Routine for Towers of Hanoi

```
void hanoi (int n, char s, char d, char i)
{
    /* n   no. of disks, s   source, d   destination i   intermediate
     */
    if (n == 1)
    {
        print (s, d);
        return;
    }
    else
    {
        hanoi (n - 1, s, i, d);
        print (s, d)
        hanoi (n-1, i, d, s);
        return;
    }
}
```

**Source Pillar**   **Intermediate Pillar**   **Destination Pillar**

1. Move Tower1 to Tower3

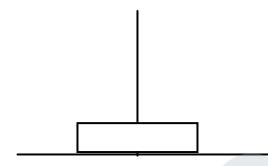
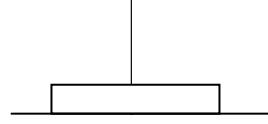


Tower 1

Tower 2

Tower 3

2. Move Tower1 to Tower2

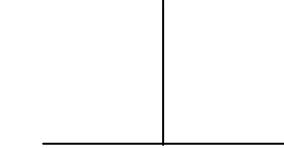
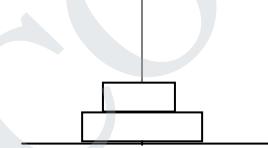
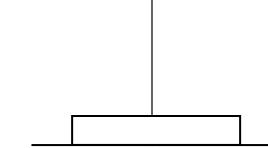


Tower 1

Tower 2

Tower 3

3. Move Tower 3 to Tower 2

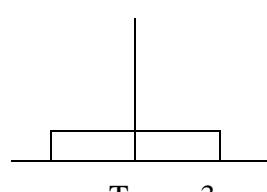
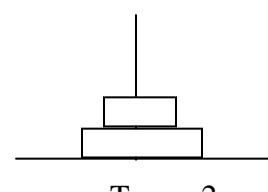
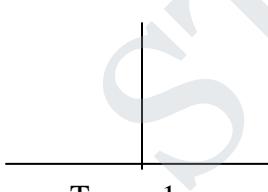


Tower 1

Tower 2

Tower 3

4. Move Tower 1 to Tower 3

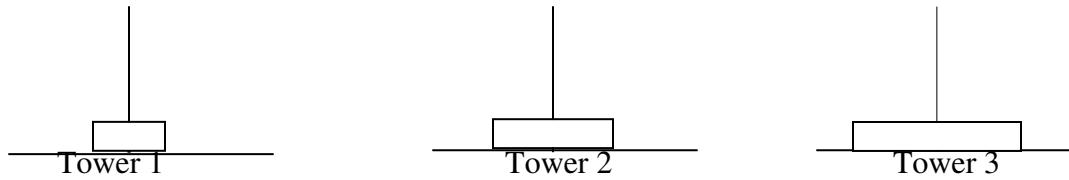


Tower 1

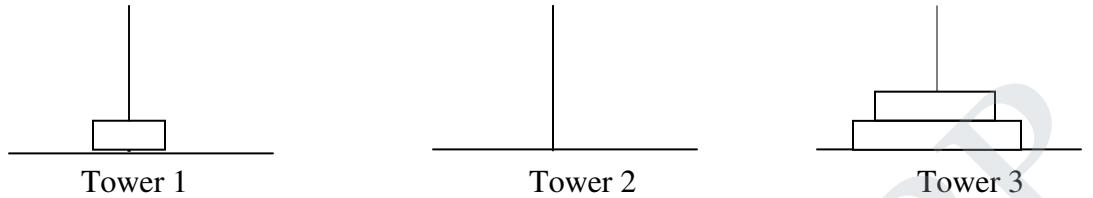
Tower 2

Tower 3

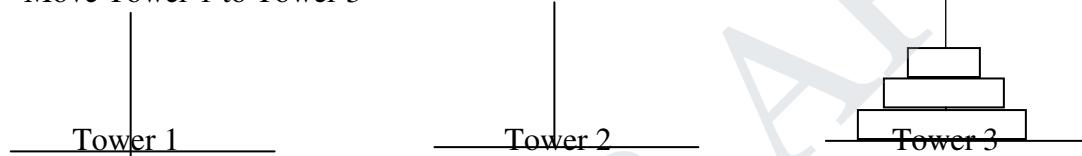
5. Move Tower 2 to Tower 1



6. Move Tower 2 to Tower 3



7. Move Tower 1 to Tower 3

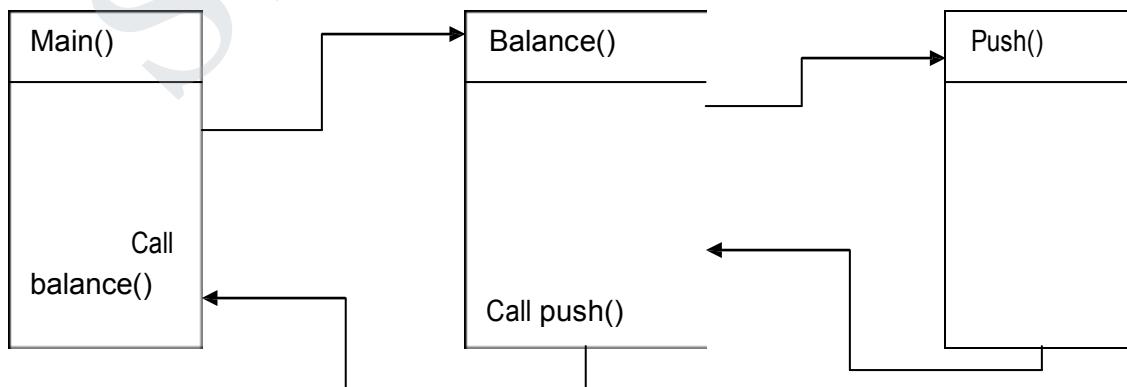


Since disks are moved from each tower in a LIFO manner, each tower may be considered as a Stack. Least Number of moves required solving the problem according to our algorithm is given by,

$$O(N) = O(N-1) + 1 + O(N-1) = 2^N - 1$$

### Function Calls

When a call is made to a new function all the variables local to the calling routine need to be saved, otherwise the new function will overwrite the calling routine variables. Similarly the current location address in the routine must be saved so that the new function knows where to go after it is completed.



### Recursive Function to Find Factorial

```
int fact(int n)
{
    int S;
    if(n==1)
        return(1);
    else
        S = n * fact( n - 1 );
        return(S)
}
```

### Balancing the Symbols

- Compilers check the programs for errors, a lack of one symbol will cause an error.
- A Program that checks whether everything is balanced.
- Every right parenthesis should have its left parenthesis.
- Check for balancing the parenthesis brackets braces and ignore any other character.

### Algorithm for balancing the symbols

Read one character at a time until it encounters the delimiter `#'.

**Step 1 :** - If the character is an opening symbol, push it onto the stack.

**Step 2 :** - If the character is a closing symbol, and if the stack is empty report an error as missing opening symbol.

**Step 3 :** - If it is a closing symbol and if it has corresponding opening symbol in the stack, POP it from the stack. Otherwise, report an error as mismatched symbols.

**Step 4 :** - At the end of file, if the stack is not empty, report an error as Missing closing symbol. Otherwise, report as balanced symbols.

**Example for Balanced symbols:**

E.g. Let us consider the expression  $((B*B)-\{4*A*C\}/[2*A]) \#$

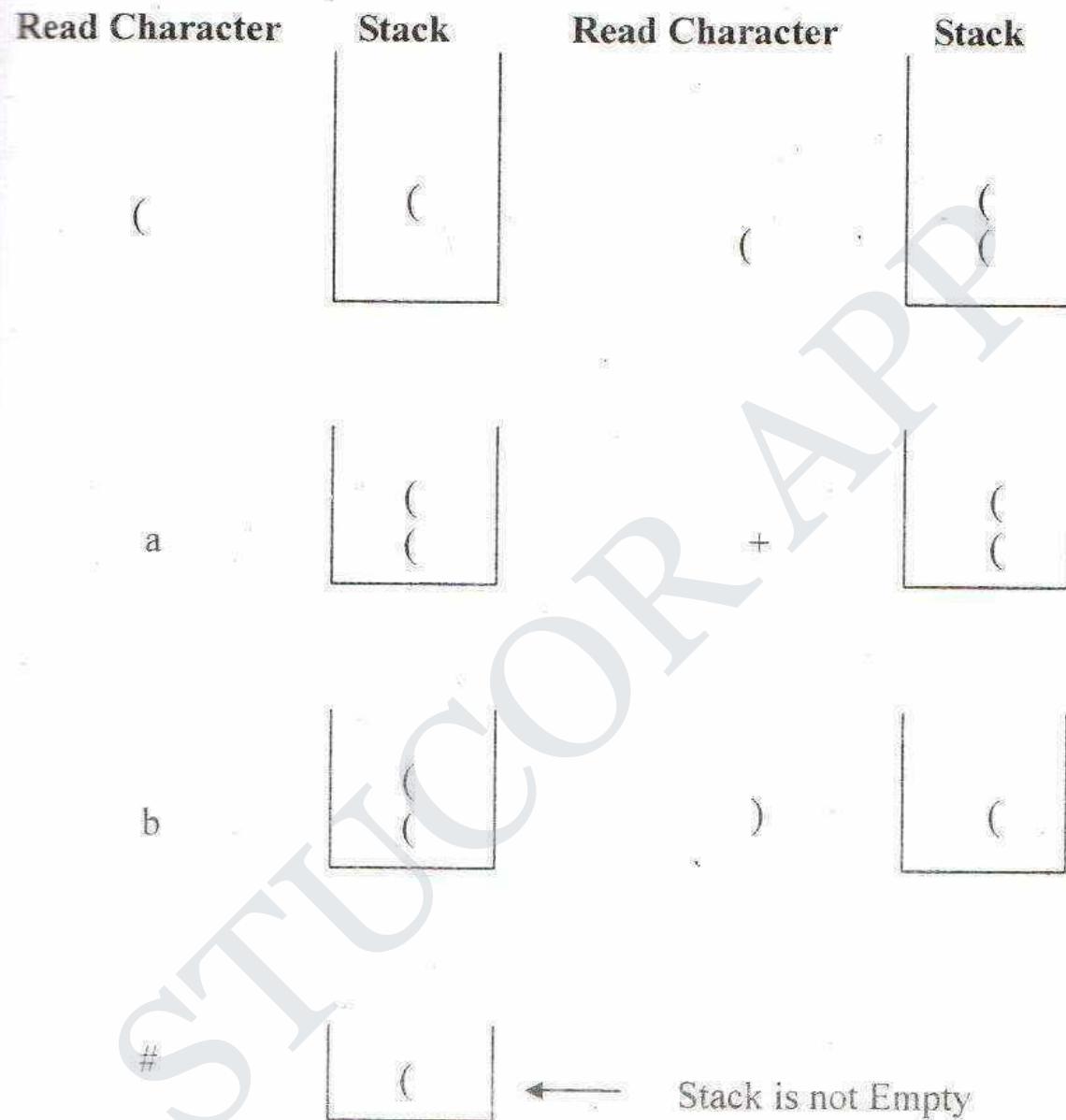
Read Character	Stack
(	(
(	(
)	□ (
{	{ (
}	□ (

[	[ ()
]	( □
)	U

Empty stack, hence the symbols are balanced in the given expression.

Example for unbalanced symbols:

Consider the expression ((a + b) # : -

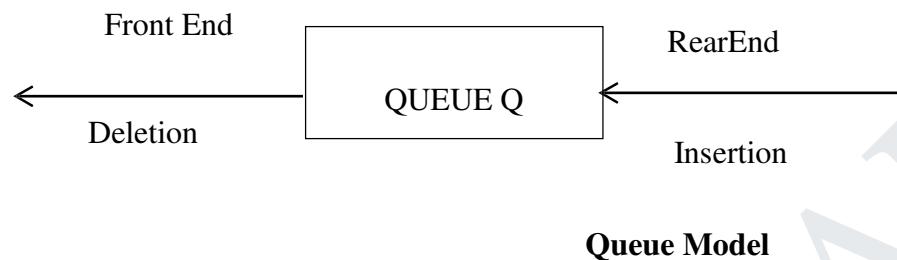


Report an error message

Illustration For Unbalanced Symbols

**QUEUES:**

- Queue is a Linear Data Structure that follows First in First out (FIFO) principle.
- Insertion of element is done at one end of the Queue called “**Rear** “end of the Queue.
- Deletion of element is done at other end of the Queue called “**Front** “end of the Queue.
- Example: - Waiting line in the ticket counter.

**Front Pointer:-**

It always points to the first element inserted in the Queue.

**Rear Pointer:-**

It always points to the last element inserted in the Queue.

**For Empty Queue:-**

Front (F) = - 1
Rear (R) = - 1

**Operations on Queue**

Fundamental operations performed on the queue are

1. EnQueue
2. DeQueue

**(i) EnQueue operation:-**

- It is the process of inserting a new element at the rear end of the Queue.
- For every EnQueue operation
  - Check for Full Queue
  - If the Queue is full, Insertion is not possible.
  - Otherwise, increment the rear end by 1 and then insert the element in the rear end of the Queue.

**(ii) DeQueue Operation:-**

- It is the process of deleting the element from the front end of the queue.
- For every DeQueue operation
  - Check for Empty queue
  - If the Queue is Empty, Deletion is not possible.
  - Otherwise, delete the first element inserted into the queue and then increment the front by 1.

**Exceptional Conditions of Queue**

- Queue Overflow
- Queue Underflow

**(i) Queue Overflow:**

- An Attempt to insert an element X at the Rear end of the Queue when the Queue is full is said to be Queue overflow.
- For every Enqueue operation, we need to check this condition.

**(ii) Queue Underflow:**

- An Attempt to delete an element from the Front end of the Queue when the Queue is empty is said to be Queue underflow.
- For every DeQueue operation, we need to check this condition.

## Implementation of Queue

Queue can be implemented in two ways.

1. Implementation using Array (**Static Queue**)
2. Implementation using Linked List (**Dynamic Queue**)

### Array Declaration of Queue:

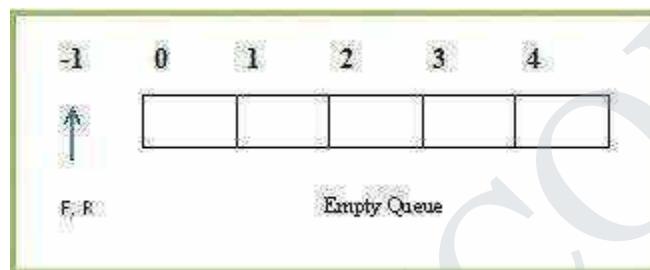
```
#define ArraySize 5
```

```
int Q [ ArraySize];
```

or

```
int Q [ 5 ];
```

### Initial Configuration of Queue:



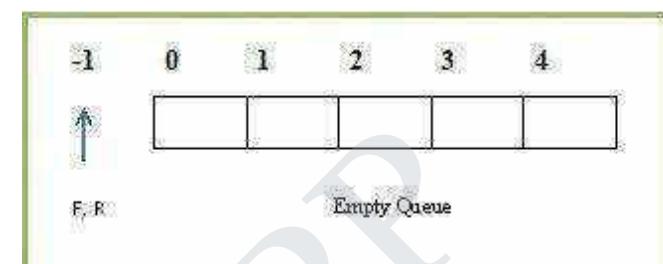
#### (i) Queue Empty Operation:

- Initially Queue is Empty.
- With Empty Queue, Front ( F ) and Rear ( R ) points to – 1.
- It is necessary to check for Empty Queue before deleting (DeQueue) an element from the Queue (Q).

**Routine to check for Empty Queue**

```
int IsEmpty ( Queue Q )  
{  
if( ( Front == - 1 ) && ( Rear == - 1 ) )  
return ( 1 );  
}
```

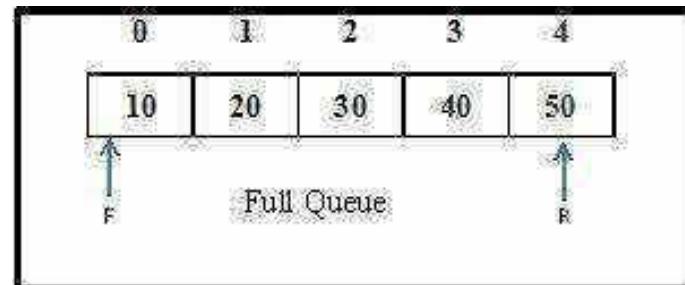
```
int IsEmpty ( Queue Q )  
{  
if( ( Front == - 1 ) && ( Rear == - 1 ) )  
return ( 1 );  
}
```

**(ii) Queue Full Operation**

- As we keep inserting the new elements at the Rear end of the Queue, the Queue becomes full.
- When the Queue is Full, Rear reaches its maximum Arraysize.
- For every Enqueue Operation, we need to check for full Queue condition.

**Routine to check for Full Queue**

```
int IsFull( Queue Q )  
{  
if ( Rear == ArraySize - 1 )  
return ( 1 );  
}
```

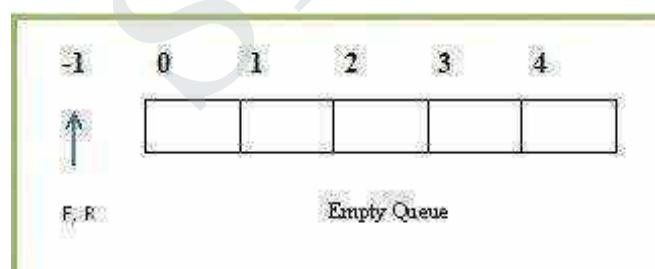
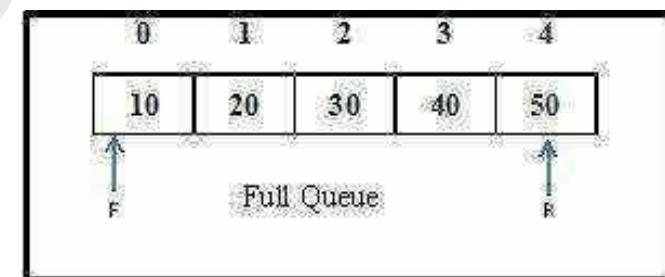


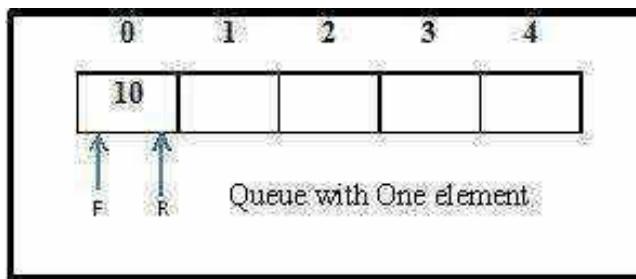
**(iii) Enqueue Operation**

- It is the process of inserting a new element at the Rear end of the Queue.
- It takes two parameters, Enqueue(X, Q). The elements X to be inserted at the Rear end of the Queue Q.
- Before inserting a new Element into the Queue, check for Full Queue.
- If the Queue is already Full, Insertion is not possible.
- Otherwise, Increment the Rear pointer by 1 and then insert the element X at the Rear end of the Queue.
- If the Queue is Empty, Increment both Front and Rear pointer by 1 and then insert the element X at the Rear end of the Queue.

**Routine to Insert an Element in a Queue**

```
void EnQueue (int X , Queue Q)
{
    if ( Rear == Arraysize - 1)
        print (" Full Queue !!!!. Insertion not
               possible");
    else if (Rear == - 1)
    {
        Front = Front + 1;
        Rear = Rear + 1;
        Q [Rear] = X;
    }
    else
    {
        Rear = Rear + 1;
        Q [Rear] = X;
    }
}
```

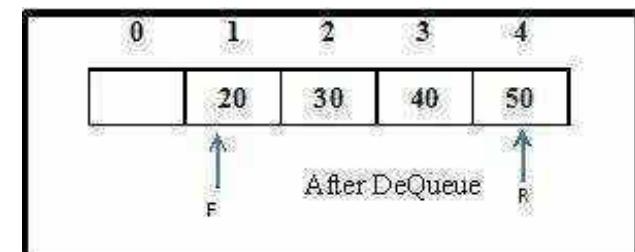
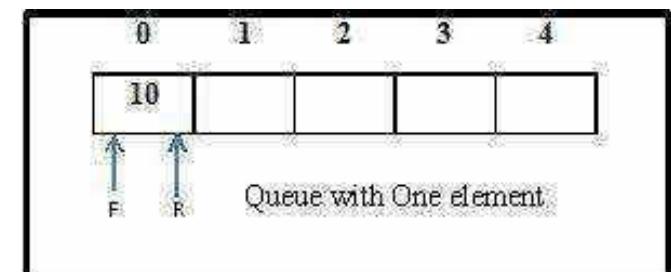
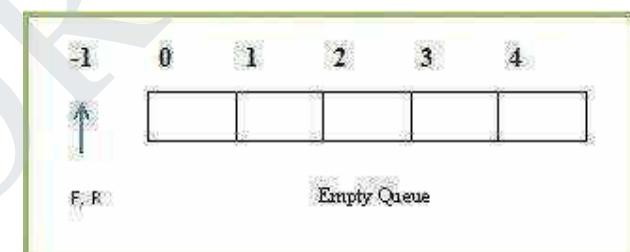


**(iv) DeQueue Operation**

- It is the process of deleting a element from the Front end of the Queue.
- It takes one parameter, DeQueue (Q). Always front element in the Queue will be deleted.
- Before deleting an Element from the Queue, check for Empty Queue.
- If the Queue is empty, deletion is not possible.
- If the Queue has only one element, then delete the element and represent the empty queue by updating Front = - 1 and Rear = - 1.
- If the Queue has many Elements, then delete the element in the Front and move the Front pointer to next element in the queue by incrementing Front pointer by 1.

**ROUTINE FOR DEQUEUE**

```
void DeQueue ( Queue Q )
{
    if ( Front == - 1 )
        print (" Empty Queue !. Deletion not possible ");
    else if( Front == Rear )
    {
        X = Q [ Front ];
        Front = - 1;
        Rear = - 1;
    }
    else
    {
        X = Q [ Front ];
        Front = Front + 1 ;
    }
}
```



## Array implementation of Queue

```
#include<stdio.h>
#include<conio.h>
#define SIZE 5
int front = - 1;
int rear = - 1;
int q[SIZE];
void insert( );
void del();
void display();
void main()
{
    int choice;
    clrscr();
    do
    {
        printf("\t Menu");
        printf("\n 1. Insert");
        printf("\n 2. Delete");
        printf("\n 3. Display ");
        printf("\n 4. Exit");

        printf("\n Enter Your Choice:");
        scanf("%d", &choice);
        switch(choice)
        {
            case 1:
                insert();
                display();
                break;
            case 2:
                del();
                display();
                break;
            case 3:
                display();
                break;
            case 4:
                printf("End of Program....!!!!");
                exit(0);
        }
    }while(choice != 4);
}

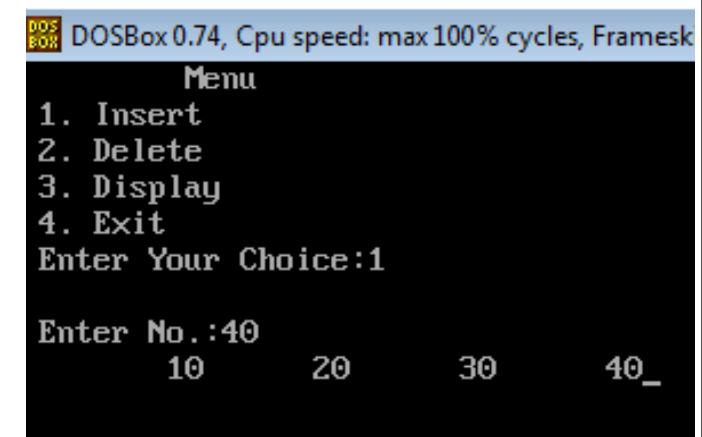
void insert()
{
    int no;
```

```
printf("\n Enter No.:");
scanf("%d", &no);

if(rear < SIZE - 1)
{
    q[++rear]=no;
    if(front == - 1)
        front=0;// front=front+1;
}
else
{
    printf("\n Queue overflow");
}

void del()
{
    if(front == - 1)
    {
        printf("\n Queue Underflow");
        return;
    }
    else
    {
        printf("\n Deleted Item:-->%d\n", q[front]);
    }
    if(front == rear)
    {
        Front = - 1;
        Rear = - 1;
    }
    else
    {
        Front = front + 1;
    }
}

void display()
{
    int i;
    if( front == - 1)
    {
        printf("\nQueue is empty....");
        return;
    }
    for(i = front; i<=rear; i++)
        printf("\t%d",q[i]);}
```

**output**

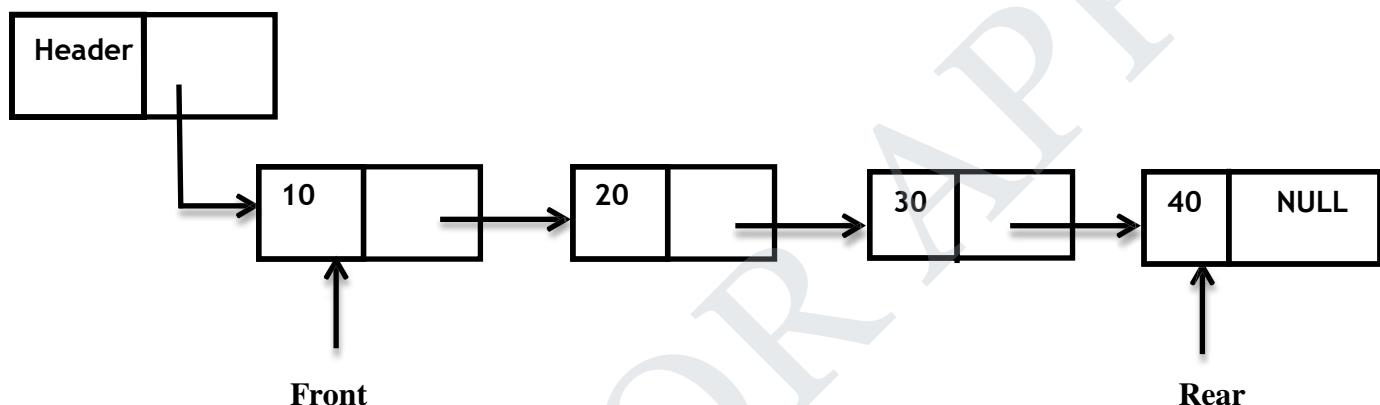
### Linked List Implementation of Queue

- Queue is implemented using SLL (Singly Linked List ) node.
- Enqueue operation is performed at the end of the Linked list and DeQueue operation is performed at the front of the Linked list.
- With Linked List implementation, for Empty queue

**Front = NULL & Rear = NULL**

Linked List representation of Queue with 4 elements

**Q**



### Declaration for Linked List Implementation of Queue ADT

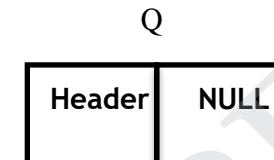
```
struct node;
typedef struct node * Queue;
typedef struct node * position;
int IsEmpty (Queue Q);
Queue CreateQueue (void);
void MakeEmpty (Queue Q);
void Enqueue (int X, Queue Q);
void Dequeue (Queue Q);
struct node
{
    int data ;
    position next;
}* Front = NULL, *Rear = NULL;
```

**(i) Queue Empty Operation:**

- Initially Queue is Empty.
- With Linked List implementation, Empty Queue is represented as  $S \rightarrow \text{next} = \text{NULL}$ .
- It is necessary to check for Empty Queue before deleting the front element in the Queue.

**ROUTINE TO CHECK WHETHER THE QUEUE IS EMPTY**

```
int IsEmpty (Queue Q
{
    if (Q->Next == NULL)
        return (1);
}
```

**(ii) EnQueue Operation**

- It is the process of inserting a new element at the Rear end of the Queue.
- It takes two parameters, EnQueue ( int X , Queue Q ). The elements X to be inserted into the Queue Q.
- Using malloc ( ) function allocate memory for the newnode to be inserted into the Queue.
- If the Queue is Empty, the newnode to be inserted will become first and last node in the list. Hence Front and Rear points to the newnode.
- Otherwise insert the newnode in the Rear  $\rightarrow$  next and update the Rear pointer.

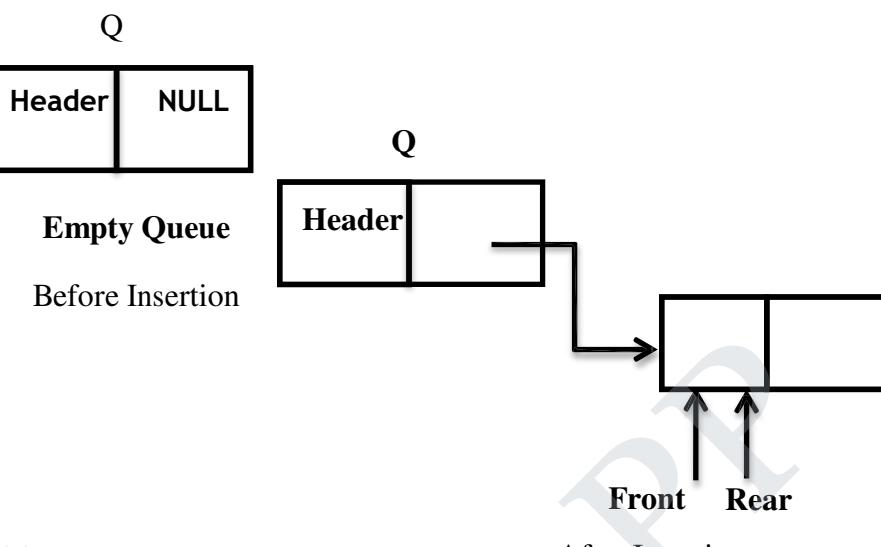
**Routine to EnQueue an Element in Queue**

```
void EnQueue ( int X, Queue Q )
{
    struct node *newnode;
    newnode = malloc (sizeof (struct node));
    if (Rear == NULL)
    {
        newnode->data = X;
        newnode->next = NULL;
        Q->next = newnode;
        Front = newnode;
        Rear = newnode;
    }
    else
    {
        newnode->data = X;
        newnode->next = NULL;
        Rear->next = newnode;
    }
}
```

```

        Rear = newnode;
    }
}

```



### (iii) DeQueue Operation

It is the process of deleting the front element from the Queue.

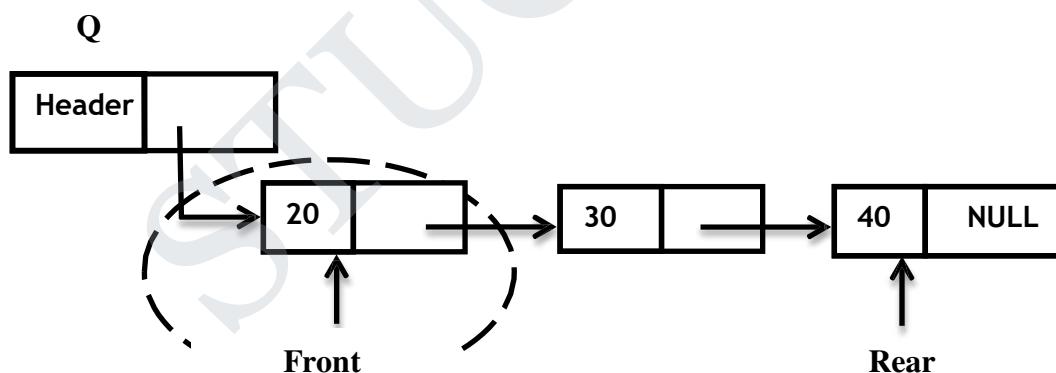
It takes one parameter, Dequeue ( Queue Q ). Always element in the front (i.e) element pointed by  $Q \rightarrow \text{next}$  is deleted always.

Element to be deleted is made “temp”.

If the Queue is Empty, then deletion is not possible.

If the Queue has only one element, then the element is deleted and Front and Rear pointer is made NULL to represent Empty Queue.

Otherwise, Front element is deleted and the Front pointer is made to point to next node in the list. The free ( ) function informs the compiler that the address that temp is pointing to, is unchanged but the data present in that address is now undefined.



**Routine to DeQueue an Element from the Queue**

```
void DeQueue ( Queue Q )
{
    struct node *temp;
    if ( Front == NULL )
        Error ("Empty Queue!!! Deletion not possible.");
    else if (Front == Rear)
    {
        temp = Front;
        Q -> next = NULL;
        Front = NULL;
        Rear = NULL;
        free ( temp );
    }
    else
    {
        temp = Front;
        Q -> next = temp -> next;
        Front = Front -> Next;
        free (temp);
    }
}
```

**Linked list implementation of Queue**

```
#include<stdio.h>
#include<conio.h>
void enqueue();
void dequeue();
void display();
typedef struct node *position;
position front=NULL,rear=NULL,newnode,temp,p;

struct node
{
    int data;
    position next;
};
void main()
{
    int choice;
    clrscr();
    do
    {
        printf("1.Enqueue\n2.Dequeue\n3.display\n4.exit\n");
        printf("Enter your choice\n\n");
    }
```

```
scanf("%d",&choice);
switch(choice)
{
    case 1:
        enqueue();
        break;
    case 2:
        dequeue();
        break;
    case 3:
        display();
        break;
    case 4:
        exit(0);
}
}
while(choice<5);
}

void enqueue()
{
    newnode=(struct node*)malloc(sizeof(struct node));
    printf("\n Enter the data to be enqueued\n");
    scanf("%d",&newnode->data);
    newnode->next=NULL;
    if(rear==NULL)
        front=rear=newnode;
    else {
        rear->next=newnode;
        rear=newnode;
    }
    display();
}
void dequeue()
{
    if(front==NULL)
        printf("\nEmpty queue!!!! Deletion not possible\n");
    else if(front==rear)
    {
        printf("\nFront element %d is deleted from queue!!!! now queue is
empty!!!! no more deletion possible!!!!\n",front->data);
        front=rear=NULL;
    }
    else
    {
        temp=front;
        front=front->next;
        printf("\nFront element %d is deleted from queue!!!!\n",temp->data);
        free(temp);
    }
    display();
}
```

```
    }
void display()
{
    p=front;
    while(p!=NULL)
    {
        printf("%d -> ",p->data);
        p=p->next;
    }
    printf("Null\n");
}
```

### Output

```
DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: TC
1.Enqueue
2.Dequeue
3.display
4.exit
Enter your choice

1

Enter the data to be enqueue
10
10 -> Null
1.Enqueue
2.Dequeue
3.display
4.exit
Enter your choice
```

### Applications of Queue

1. Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
2. In real life, Call Center phone systems will use Queues, to hold people calling them in an order, until a service representative is free.
3. Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive, First come first served.
4. Batch processing in operating system.
5. Job scheduling Algorithms like Round Robin Algorithm uses Queue.

### Drawbacks of Queue (Linear Queue)

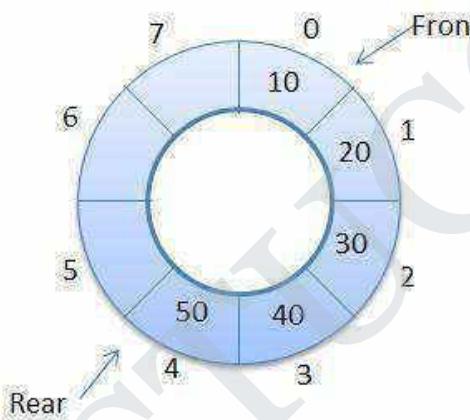
- With the array implementation of Queue, the element can be deleted logically only by moving  $\text{Front} = \text{Front} + 1$ .
- Here the Queue space is not utilized fully.

To overcome the drawback of this linear Queue, we use Circular Queue.

## CIRCULAR QUEUE

In Circular Queue, the insertion of a new element is performed at the very first location of the queue if the last location of the queue is full, in which the first element comes just after the last element.

- A circular queue is an abstract data type that contains a collection of data which allows addition of data at the end of the queue and removal of data at the beginning of the queue.
- Circular queues have a fixed size.
- Circular queue follows FIFO principle.
- Queue items are added at the rear end and the items are deleted at front end of the circular queue
- Here the Queue space is utilized fully by inserting the element at the Front end if the rear end is full.



## Operations on Circular Queue

Fundamental operations performed on the Circular Queue are

- Circular Queue Enqueue
- Circular Queue Dequeue

### Formula to be used in Circular Queue

**For Enqueue**       $\text{Rear} = (\text{Rear} + 1) \% \text{ArraySize}$

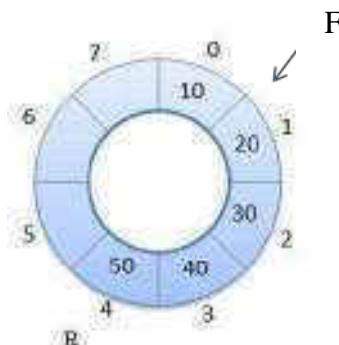
**For Dequeue**       $\text{Front} = (\text{Front} + 1) \% \text{ArraySize}$

#### (i) Circular Queue Enqueue Operation

- It is same as Linear Queue EnQueue Operation (i.e) Inserting the element at the Rear end.
- First check for full Queue.
- If the circular queue is full, then insertion is not possible.
- Otherwise check for the rear end.
- If the Rear end is full, the elements start getting inserted from the Front end.

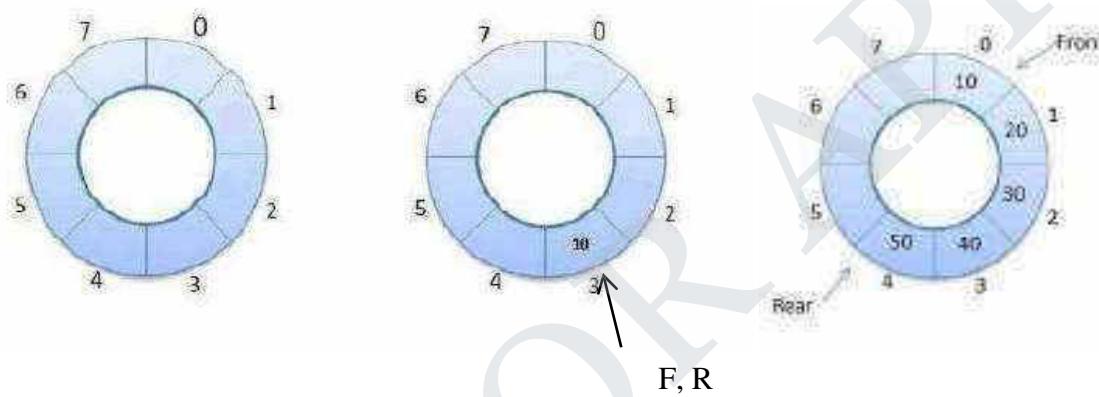
#### Routine to Enqueue an element in circular queue

```
void Enqueue ( int X, CircularQueue CQ )
{
    if( Front == ( Rear + 1 ) % ArraySize)
        Error( "Queue is full!!Insertion not possible" );
    else if( Rear == -1 )
    {
        Front = Front + 1;
        Rear = Rear + 1;
        CQ[ Rear ] = X;
    }
    else
    {
        Rear = ( Rear + 1 ) % Arraysize;
        CQ[ Rear ] = X;
    }
}
```



## Circular Queue DeQueue Operation

- It is same as Linear Queue DeQueue operation (i.e) deleting the front element.
- First check for Empty Queue.
- If the Circular Queue is empty, then deletion is not possible.
- If the Circular Queue has only one element, then the element is deleted and Front and Rear pointer is initialized to - 1 to represent Empty Queue.
- Otherwise, Front element is deleted and the Front pointer is made to point to next element in the Circular Queue.



F= -1,R= --1

## Routine To DeQueue An Element In Circular Queue

```
void DeQueue (CircularQueue CQ)
{
    if(Front== - 1)
        Empty("Empty Queue!");
    else if(Front==rear)
    {
        X=CQ[Front];
        Front=-1;
        Rear=-1;
    }
    else
    {
        X=CQ[Front];
        Front=(Front+1)%Arraysize;
    }
}
```

**Implementation of Circular Queue**

```
#include<stdio.h>
#include<conio.h>
#define max 3
void insert(); void delet(); void display();
int q[10],front=0,rear=-1;
void main()
{ int ch;
clrscr();
printf("\nCircular Queue operations\n"); printf("1.insert\n2.delete\n3.display\n4.exit\n");
while(1)
{
printf("Enter your choice:");
scanf("%d",&ch);
switch(ch)
{
case 1:
    insert(); break;
case 2:
    delet(); break;
case 3:
    display(); break;
case 4:
    exit();
default:
    printf("Invalid option\n");
}}}

void insert()
{
int x;
if((front==0&&rear==max-1)||(front>0&&rear==front-1))
printf("Queue is overflow\n");
else
{
printf("Enter element to be insert:");
scanf("%d",&x);
if(rear==max-1&&front>0)
{ rear=0;
q[rear]=x;
}
else
{
if((front==0&&rear==-1)||(rear!=front-1))
q[++rear]=x;
}
}
```

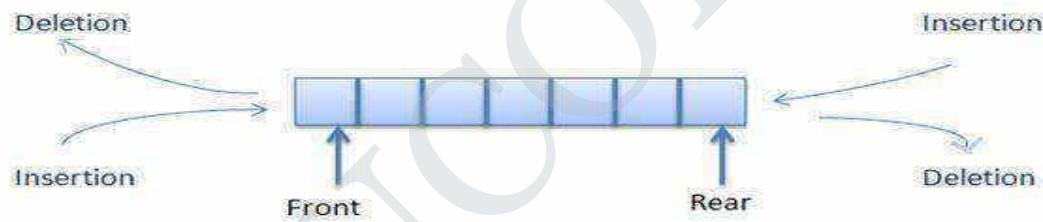
```
    } } }
void delet()
{
    int a;
    if((front==0)&&(rear==1))
        printf("Queue is underflow\n");
    if(front==rear)
    {
        a=q[front];
        rear=-1;
        front=0;
    }
    else if(front==max-1)
    {
        a=q[front];
        front=0;
    }
    else
        a=q[front++];
    printf("Deleted element is:%d\n",a);
}
void display()
{
    int i,j; if(front==0&&rear==1)
    printf("Queue is underflow\n");
    if(front>rear) {
        for(i=0;i<=rear;i++)
            printf("\t%d",q[i]);
        for(j=front;j<=max-1;j++)
            printf("\t%d",q[j]);
        printf("\nrear is at %d\n",q[rear]);
        printf("\nfront is at %d\n",q[front]); }
    else
    {
        for(i=front;i<=rear;i++)
            printf("\t%d",q[i]); printf("\nrear
is at %d\n",q[rear]); printf("\nfront
is at %d\n",q[front]);
    }
    printf("\n");
}
```

## OUTPUT

```
DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: TC
1.insert ( max 3 element)
2.delete
3.display
4.exit
Enter your choice:1
Enter element to be insert:10
Enter your choice:1
Enter element to be insert:20
Enter your choice:1
Enter element to be insert:30
Enter your choice:1
Queue is overflow
Enter your choice:2
Deleted element is:10
Enter your choice:1
Enter element to be insert:40
Enter your choice:1
Queue is overflow
Enter your choice:3
      40      20      30
rear is at 40
front is at 20
Enter your choice:
```

## DOUBLE-ENDED QUEUE (DEQUE)

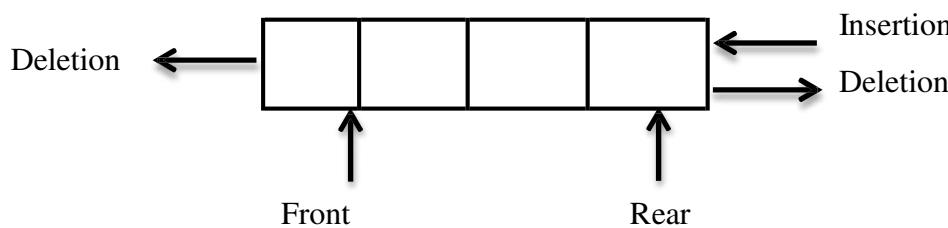
In DEQUE, insertion and deletion operations are performed at both ends of the Queue.



## Exceptional Condition of DEQUE

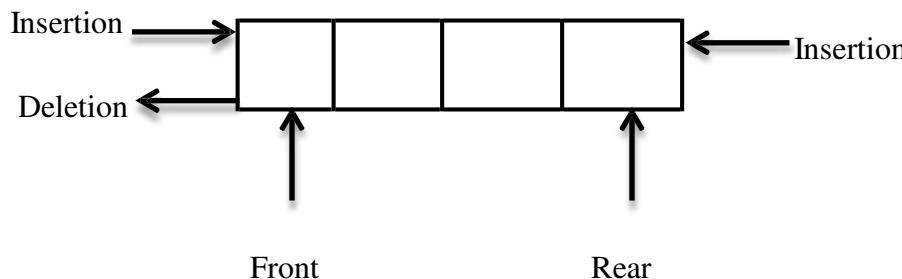
### (i) Input Restricted DEQUE

Here insertion is allowed at **one end** and deletion is allowed at **both ends**.



## (ii) Output Restricted DEQUE

Here insertion is allowed at **both ends** and deletion is allowed at **one end**.



## Operations on DEQUE

**Four cases for inserting and deleting the elements in DEQUE are**

1. Insertion At Rear End [ same as Linear Queue ]
2. Insertion At Front End
3. Deletion At Front End [ same as Linear Queue ]
4. Deletion At Rear End

### Insertion at the rear end

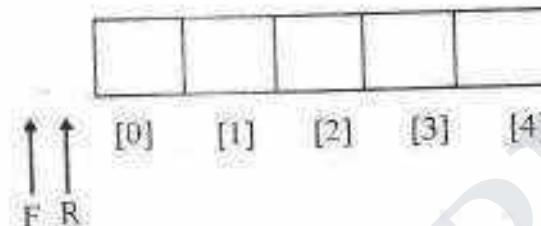
- Step 1 : Check for the overflow condition.
- Step 2 : If it is true, display that the queue is full
- Step 3 : Otherwise, If the rear and front pointers are at the initial values (-1). Increment both the pointers. Goto step 5.
- Step 4 : Increment the rear pointer
- Step 5 : Assign the value to Q[rear]

### Case 1: Routine to insert an element at Rear end

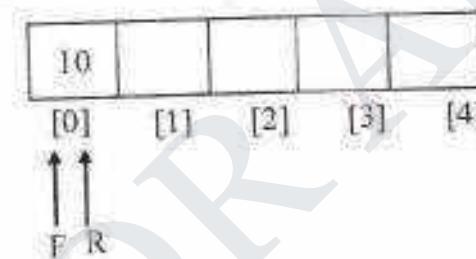
```
void Insert_Rear (int X, DEQUE DQ)
{
    if( Rear == Arraysize - 1)
        Error("Full Queue!!!! Insertion not possible");
    else if( Rear == -1)
    {
        Front = Front + 1;
        Rear = Rear + 1;
        DQ[ Rear ] = X;
    }
}
```

```
else
{
    Rear = Rear + 1;
    DQ[ Rear ] = X;
}
}
```

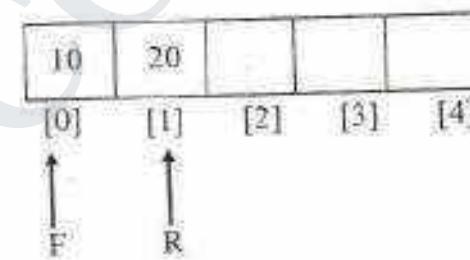
### insertion at the rear end



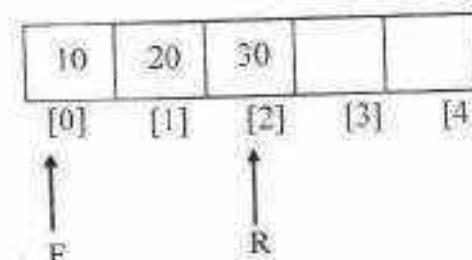
insert\_rear (10)



insert\_rear (20)



Insert\_rear (30)



**Insertion at front end**

- Step 1 : Check the front pointer, if it is in the first position (0) then display an error message that the value cannot be inserted at the front end.
- Step 2 : Otherwise, decrement the front pointer
- Step 3 : Assign the value to Q[front]

**Case 2: Routine to insert an element at Front end**

```
void Insert_Front ( int X, DEQUE DQ )
{
    if( Front == 0 )
        Error("Element present in Front!!!! Insertion not possible");
    else if(Front == -1)
    {
        Front = Front + 1;
        Rear = Rear + 1;
        DQ[Front] = X;
    }
    else
    {
        Front = Front - 1;
        DQ[Front] = X;
    }
}
```

Case 1: When the front pointer is -1.

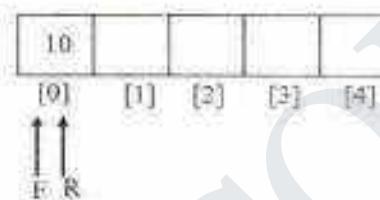
Initial:



Insert\_front(5)

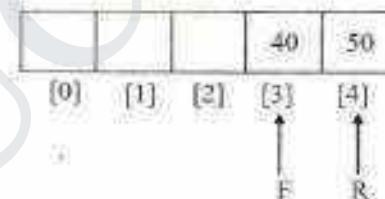


Insert\_front(5)

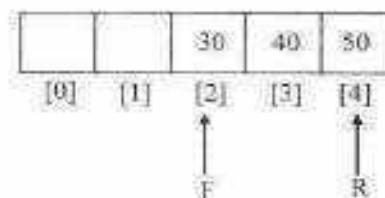


Here decrement of the front pointer may point to the initial (-1) position. So the value cannot be inserted even there is a space in the queue.

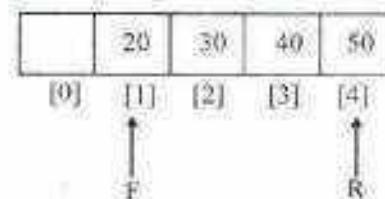
Case 2



Insert\_front(30)



Insert\_front(30)

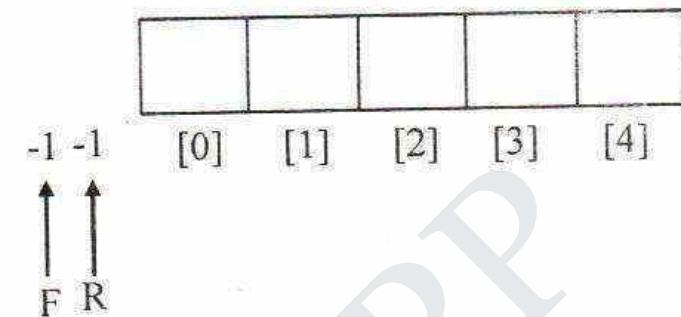
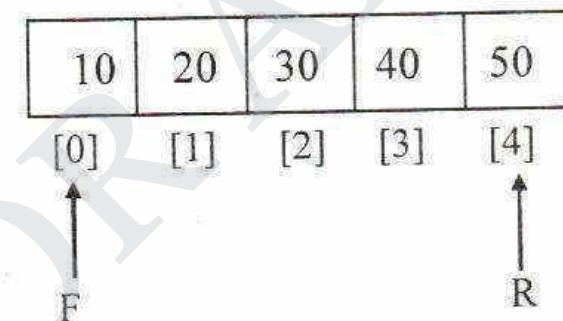
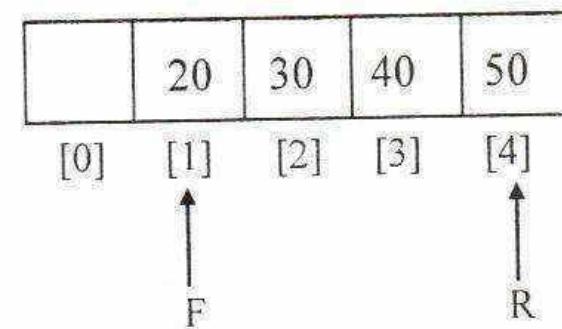


**Case 3: Routine to delete an element from Front end**

```
void Delete_Front(DEQUE DQ)
{
    if(Front == - 1)
        Error("Empty queue!!!! Deletion not possible");
    else if( Front == Rear )
    {
        X = DQ[ Front ];
        Front = - 1;
        Rear = - 1;
    }
    else
    {
        X = DQ [ Front ];
        Front = Front + 1;
    }
}
```

**Deletion from Front End :**

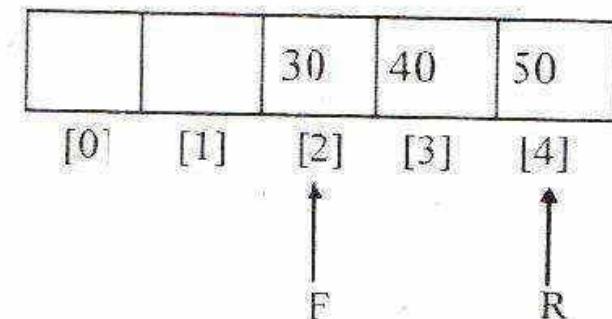
- Step 1 : Check for the underflow condition. If it is true display that the queue is empty.
- Step 2 : Otherwise, delete the element at the front position, by assigning X as Q[front]
- Step 3 : If the rear and front pointer points to the same position (ie) only one value is present, then reinitialize both the pointers.
- Step 4 : Otherwise, Increment the front pointer

**Deletion from front end***Dequeue\_front()**Q - is empty**Dequeue\_front()*

**Case 4: Routine to delete an element from Rear end**

```
void Delete_Rear(DEQUE DQ)
{
    if( Rear == - 1)
        Error("Empty queue!!!! Deletion not possible");
    else if( Front == Rear )
    {
        X = DQ[ Rear ];
        Front = - 1;
        Rear = - 1;
    }
    else
    {
        X = DQ[ Rear ];
        Rear = Rear - 1;
    }
}
```

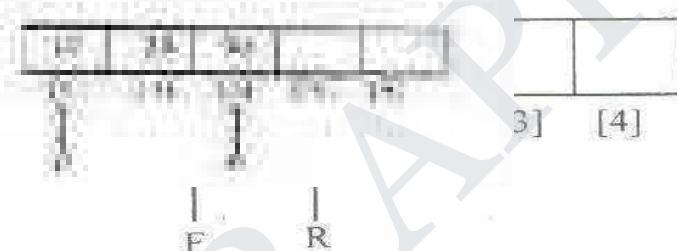
*Delete\_front()*



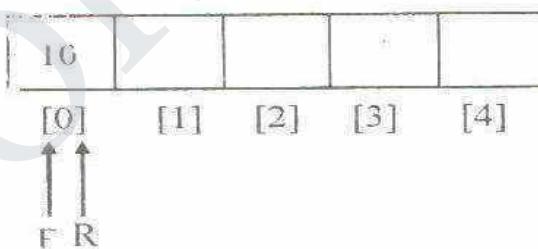
### Deletion from Rear End

- Step 1 : Check the rear pointer. If it is in the initial value then display that the value cannot be deleted.
- Step 2 : Otherwise, delete element at the rear position.
- Step 3 : If the rear and front pointers are at the same position, reinitialize both the pointers.
- Step 4 : Otherwise, decrement the rear pointer.

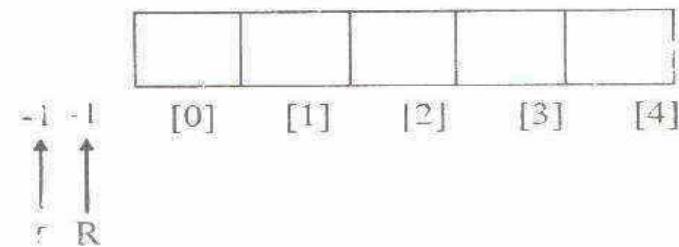
Deletion from Rear End



*Dequeue\_rear ()*



*Dequeue\_rear ()*



## Binary Tree using Array Representation

Each node contains **info**, **left**, **right** and **father** fields. The left, right and father fields of a node point to the node's left son, right son and father respectively.

Using the array implementation, we may declare,

```
#define NUMNODES 100
struct nodetype
{
    int info;
    int left;
    int right;
    int father;
};
struct nodetype node[NUMNODES];
```

This representation is called linked array representation.

**Example:** -

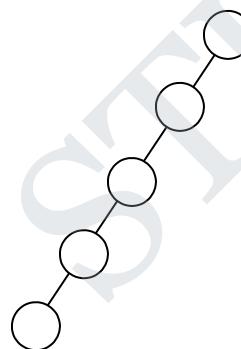


Fig (a)

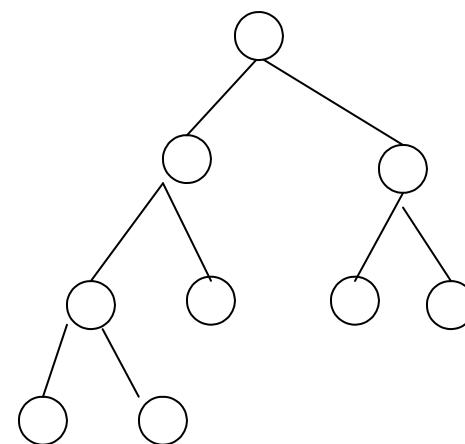
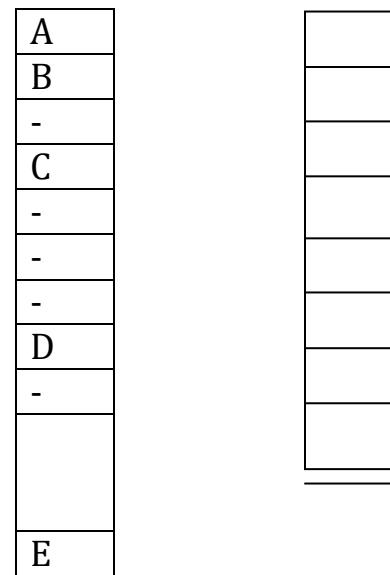


Fig (b)

The above trees can be represented in memory sequentially as follows



**The above representation appears to be good for complete binary trees and wasteful for many other binary trees.** In addition, the insertion or deletion of nodes from the middle of a tree requires the insertion of many nodes to reflect the change in level number of these nodes.

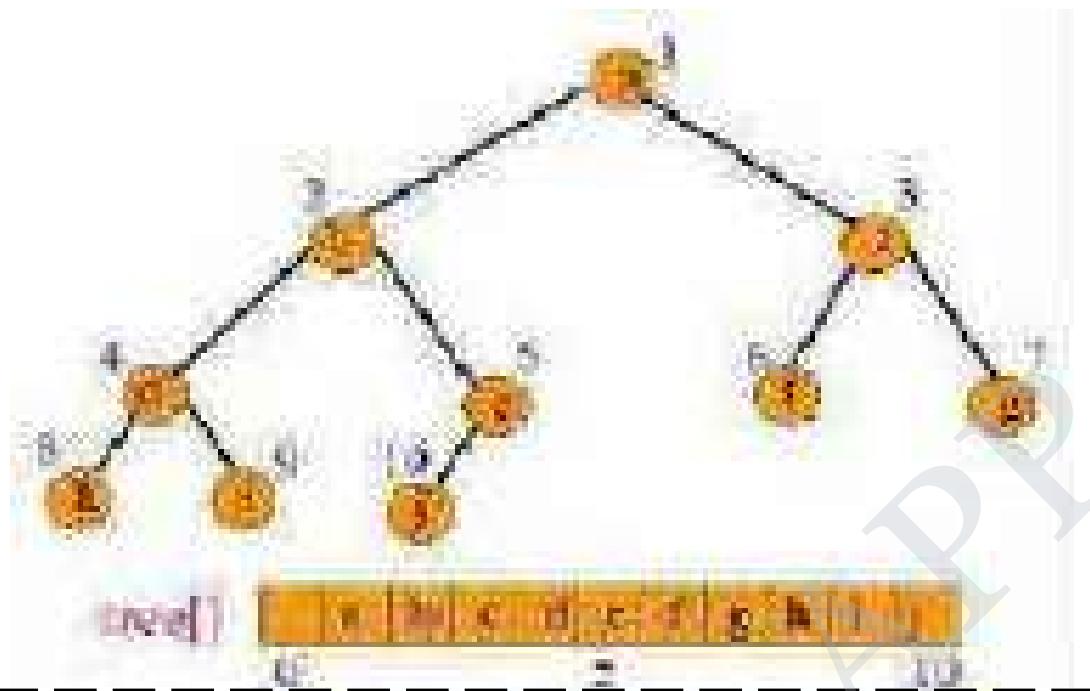
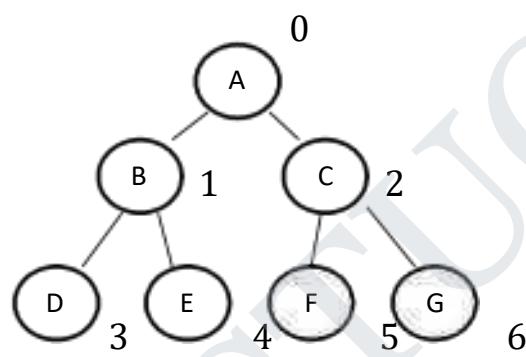


Figure 2.5



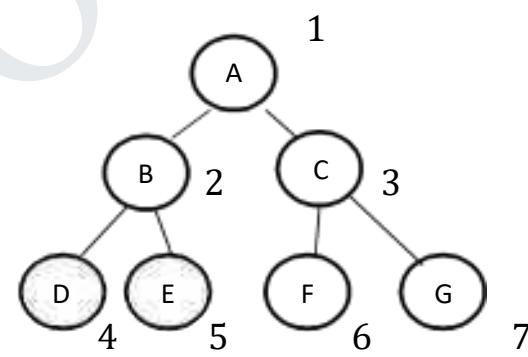
0    1    2    3    4    5    6

For Figure 2.5

Root = **i**

**leftchild=2i+1**

Figure 2.6



1    2    3    4    5    6    7

For Figure 2.6

Root = **i**

**leftchild=2i**

**rightchild=2i+2****leftchild's parent position = i/2** **$2^{n+1} - 1 \Rightarrow$  array size****n => no of levels of a tree****rightchild's position= i-1/2****rightchild=2i+1****parent position= i/2** **$2^{n+1} - 1 \Rightarrow$  size of array****n => number of levels of a tree**

## Binary Tree using Link Representation

The problems of sequential representation can be easily overcome through the use of a linked representation.

Each node will have three fields LCHILD, DATA and RCHILD as represented below

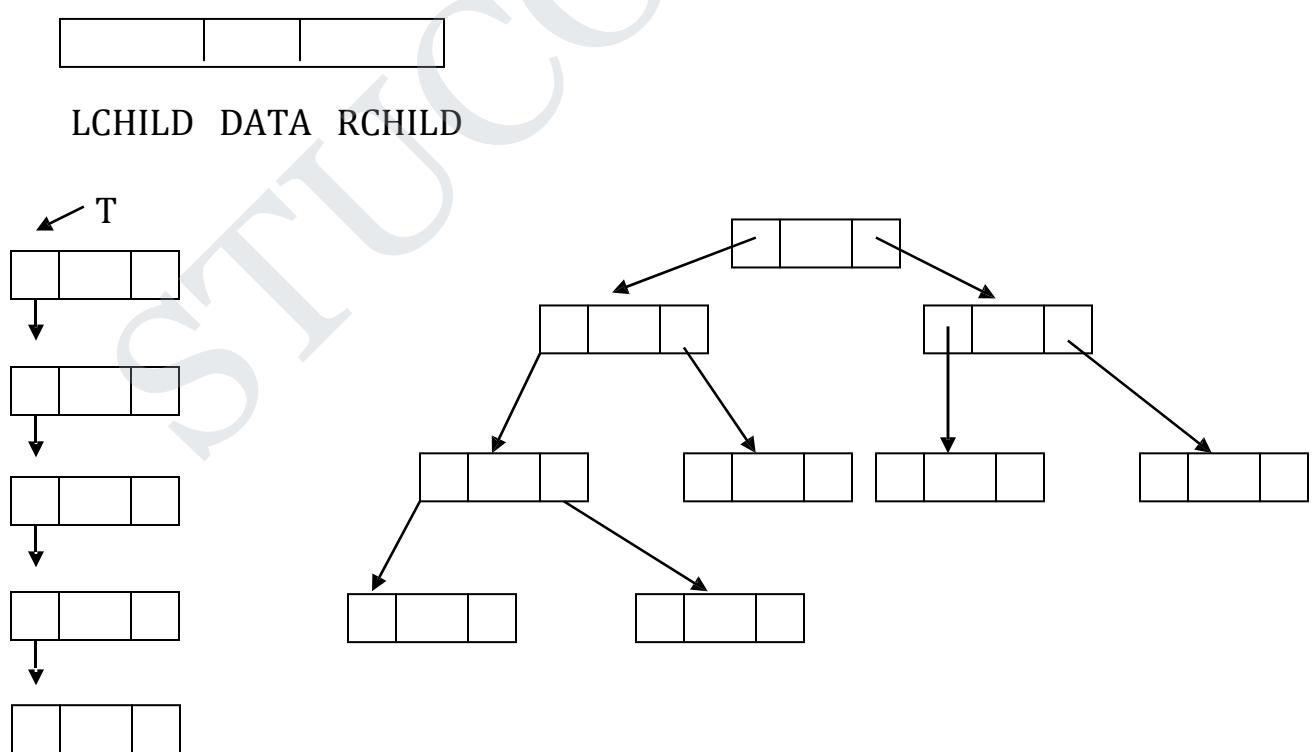


Fig (a)

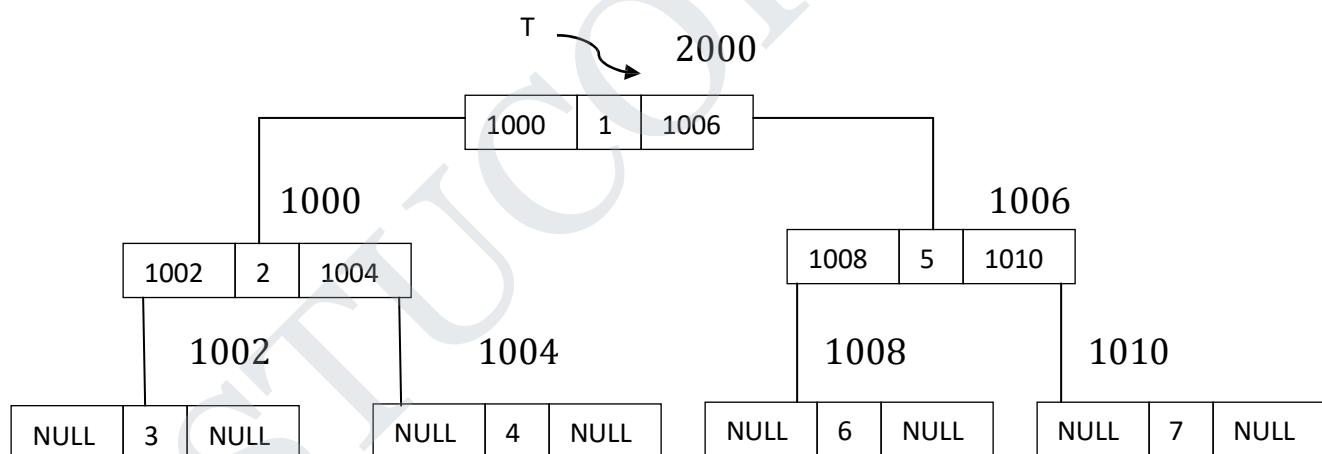
Fig (b)

In most applications it is adequate. But this structure make it difficult to determine the parent of a node since this leads only to the forward movement of the links.

Using the linked implementation we may declare,

### Struct treenode

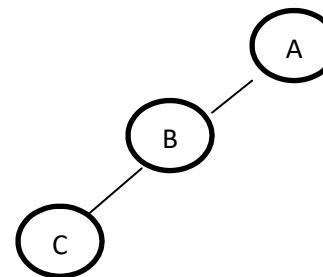
```
{  
int data;  
structtreenode *leftchild;  
structtreenode *rightchild;  
}*T;
```



## TYPES OF BINARY TREES

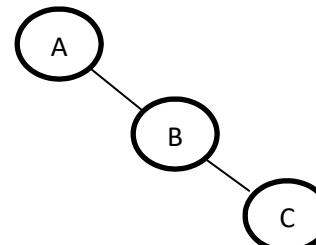
### Left Skewed Binary tree :

A binary tree which has only left child is called left skewed binary tree.



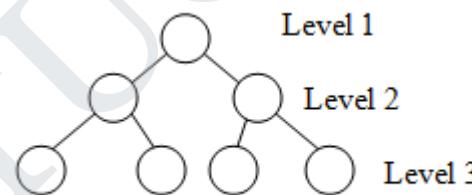
### Right Skewed Binary tree :

A Binary tree which has only right child is called right skewed binary tree.



### Full Binary Tree :

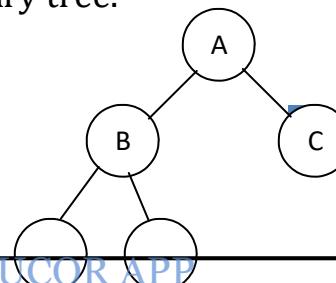
It is the one which has exactly two children for each node at each level and all the leaf nodes should be at the same level.



### Complete Binary Tree :

It is the one tree where all the leaf nodes need not be at the same level and at the bottom level of the complete binary tree, the nodes should be filled from the left to the right.

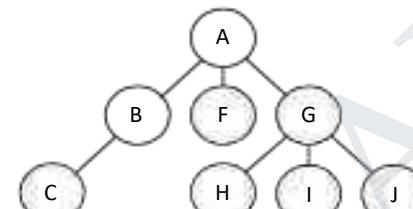
All full binary trees are complete binary tree. But all complete binary trees need not be full binary tree.



## CONVERSION OF A GENERAL TREE TO BINARY TREE

### General Tree:

A General Tree is a tree in which each node can have an unlimited out degree. Each node may have as many children as is necessary to satisfy its requirements. *Example: Directory Structure*



It is considered easy to represent binary trees in programs than it is to represent general trees. So, the general trees can be represented in binary tree format.

### Changing general tree to Binary tree:

The binary tree format can be adopted by changing the meaning of the left and right pointers. There are two relationships in binary tree,

Parent to child

Sibling to sibling

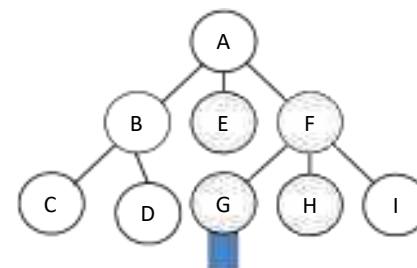
Using these relationships, the general tree can be implemented as binary tree.

### Algorithm

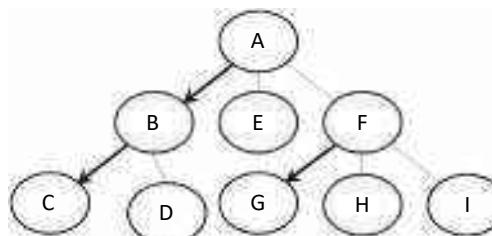
Identify the branch from the parent to its first or leftmost child. These branches from each parent become left pointers in the binary tree

Connect siblings, starting with the leftmost child, using a branch for each sibling to its right sibling.

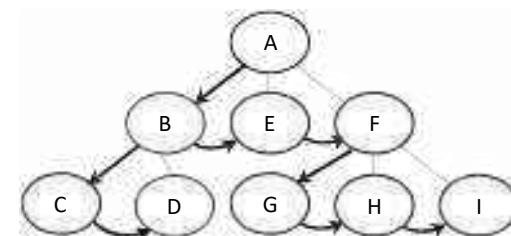
Remove all unconnected branches from the parent to its children



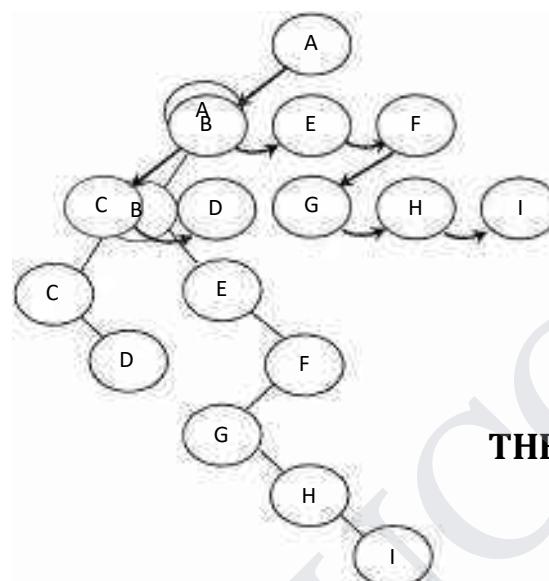
(a) General Tree



Step 1: Identify all leftmost children



Step 2: Connect Siblings



Step 3: Delete unneeded branches

### THE RESULTING BINARY TREE

## BINARY TREE TRAVERSALS

Compared to linear data structures like linked lists and one dimensional array, which have only one logical means of traversal, tree structures can be traversed in many different ways. Starting at the root of a binary tree, there are three main steps that can be performed and the order in which they are performed defines the traversal type. These steps (in no particular order) are: performing an action on the current node (referred to as "visiting" the node),

traversing to the left child node, and traversing to the right child node. Thus the process is most easily described through recursion.

A binary tree traversal requires that each node of the tree be processed once and only once in a predetermined sequence.

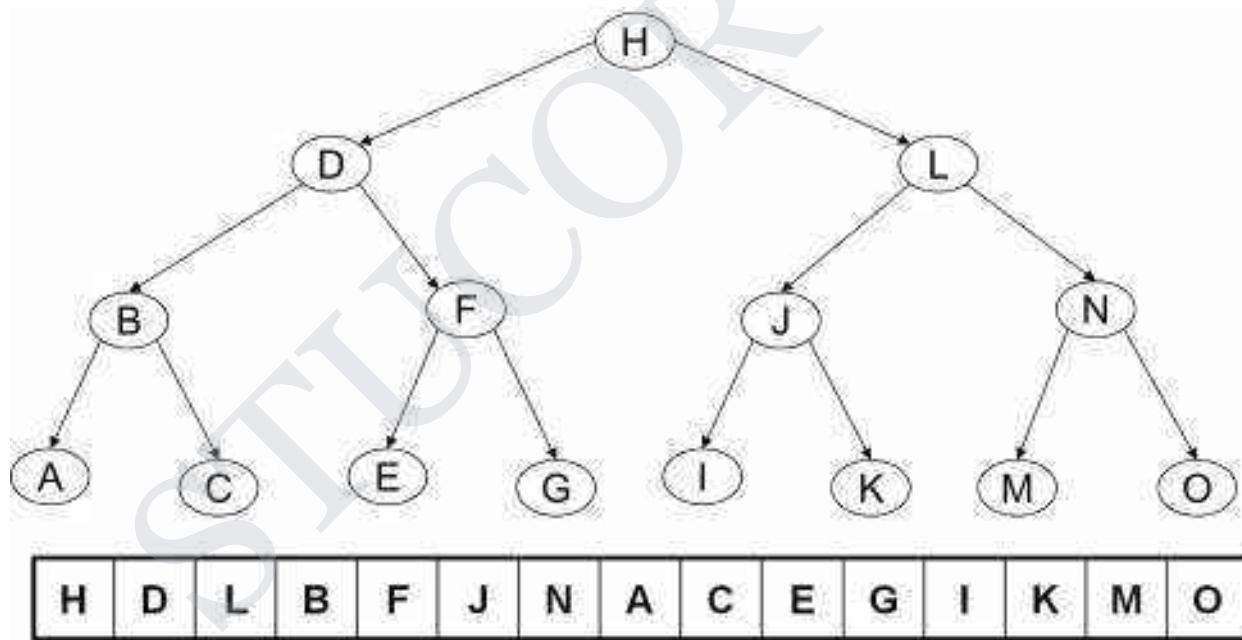
The two general approaches to the traversal sequence are,

***Depth first traversal***

***Breadth first traversal***

## Breadth-First Traversal

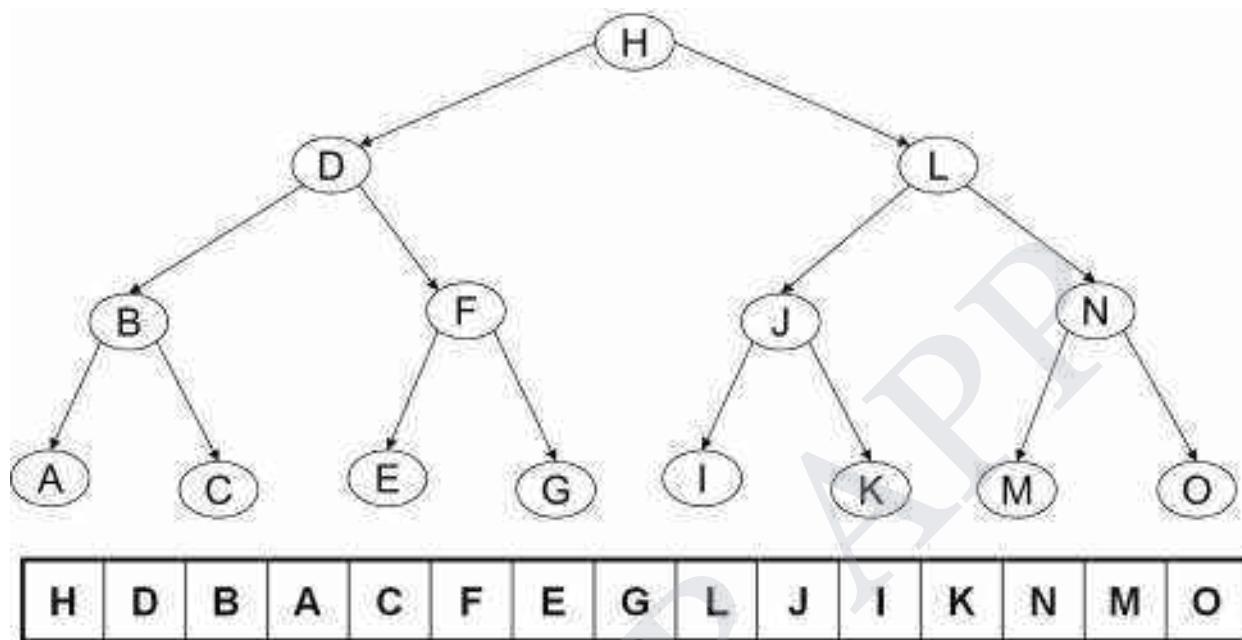
In a breadth-first traversal, the processing proceeds horizontally from the root to all its children, then to its children's children, and so forth until all nodes have been processed. In other words, in breadth traversal, each level is completely processed before the next level is started.



## Depth-First Traversal

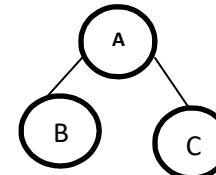
In depth first traversal, the processing proceeds along a path from the root through one child to the most distant descendent of that first child before

processing a second child. In other words, in the depth first traversal, all the descendants of a child are processed before going to the next child.



There are basically three ways of binary tree traversals.

1. **Inorder --- (left child,root,right child)**
2. **Preorder --- (root,left child,right child)**
3. **Postorder --- (left child,right child,root)**



Inorder--- B A C

Preorder --- A B C

Postorder --- B C A

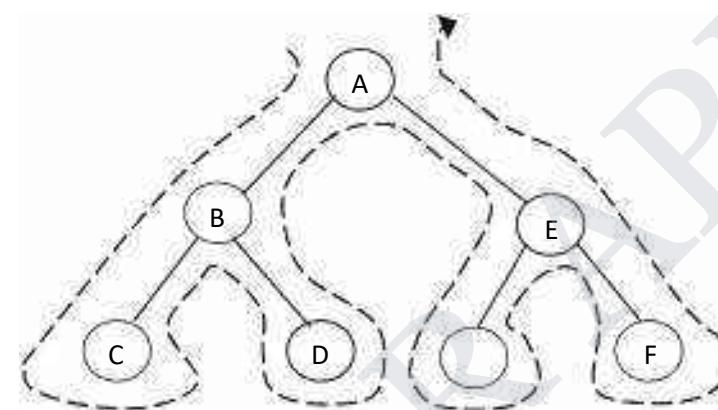
In C, each node is defined as a structure of the following form:

```
struct node
{
    int info;
    struct node *lchild;
    struct node *rchild;
}
typedef struct node NODE;
```

## Inorder Traversal

### Steps :

Traverse left subtree in inorder  
Process root node  
Traverse right subtree in inorder



The Output is : C → B → D → A → E → F

### Algorithm

```
Algorithm inorder traversal (BinTree T)
Begin
If ( not empty (T) ) then
Begin
Inorder_traversal ( left subtree ( T ) )
Print ( info ( T ) ) /* process node */
Inorder_traversal ( right subtree ( T ) )
End
End
```

### Routines

```
void inorder_traversal ( NODE * T )
{
if( T != NULL)
{
```

```
inorder_traversal(T->lchild);
printf("%d \t ", T->info);
inorder_traversal(T->rchild);
}
}
```

## Preorder Traversal

### Steps :

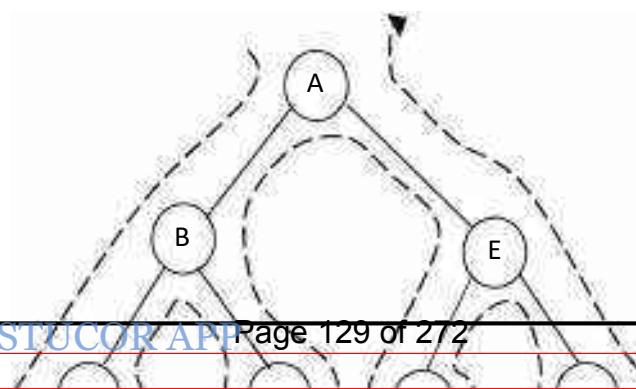
- Process root node
- Traverse left subtree in preorder
- Traverse right subtree in preorder

### Algorithm

```
Algorithm inorder traversal (BinTree T)
Begin
If ( not empty (T) ) then
Begin
Print ( info ( T ) ) / * process node */
Preorder_traversal ( left subtree ( T ) )
Preorder_traversal ( right subtree ( T ) )
End
End
```

### Routines

```
void inorder_traversal ( NODE * T )
{
if( T != NULL)
{
printf("%d \t ", T->info);
preorder_traversal(T->lchild);
preorder_traversal(T->rchild);
}
}
```



Output is : A → B → C → D → E → F

## Postorder Traversal

### Steps :

Traverse left subtree in postorder  
Traverse right subtree in postorder  
process root node

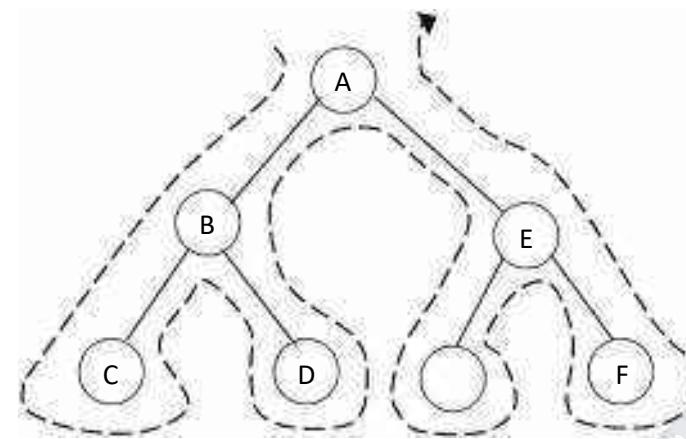
### Algorithm

Algorithm postorder traversal (BinTree T)

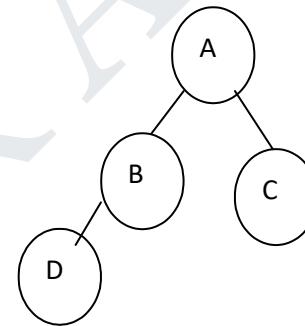
```
Begin
If ( not empty (T) ) then
Begin
Postorder_traversal ( left subtree ( T ) )
Postorder_traversal ( right subtree( T) )
Print ( Info ( T ) ) /* process node */
End
End
```

### Routines

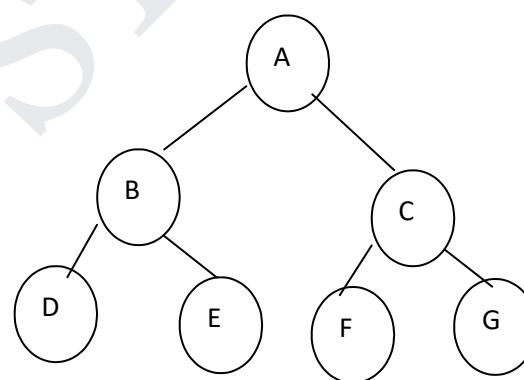
```
void postorder_traversal ( NODE * T )
{
if( T != NULL)
{
postorder_traversal(T->lchild);
postorder_traversal(T->rchild);
printf("%d \t", T->info);
```

{  
}

Output is : C → D → B → F → E → A

**Examples :****1.FIND THE TRAVERSAL OF THE FOLLOWING TREE**

ANSWER : POSTORDER: DBCA INORDER: DBAC PREORDER:ABCD

**2.FIND THE TRAVERSAL OF THE FOLLOWING TREE**

ANSWER : POSTORDER: DEBFGCA INORDER: DBEAFCG

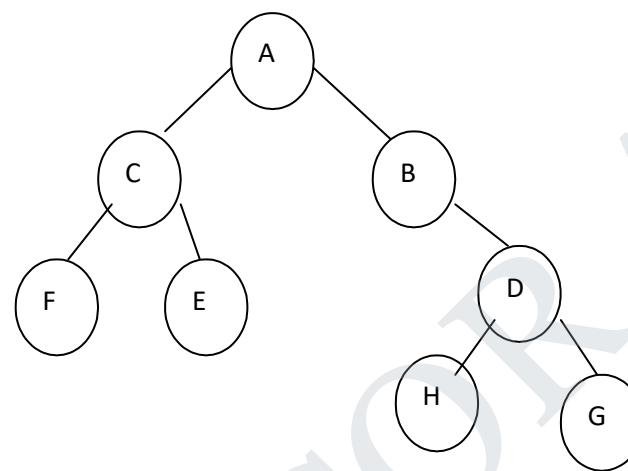
PREORDER: ABDECFCG

3.A BINARY TREE HAS 8 NODES. THE INORDER AND POSTORDER TRAVERSAL OF THE TREE ARE GIVEN BELOW. DRAW THE TREE AND FIND PREORDER.

POSTORDER: F E C H G D B A

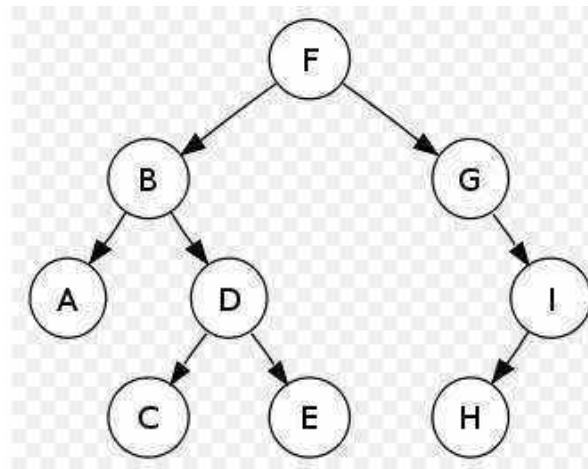
INORDER: F C E A B H D G

Answer:



PREORDER: ACFEBDHG

**Example 4**



Preorder traversal sequence: F, B, A, D, C, E, G, I, H (root, left, right)

Inorder traversal sequence: A, B, C, D, E, F, G, H, I (left, root, right)

Postorder traversal sequence: A, C, E, D, B, H, I, G, F (left, right, root)

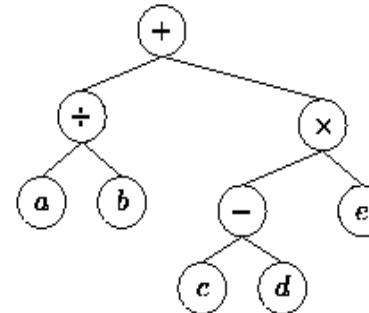
## APPLICATIONS

1. Some applications of preorder traversal are the evaluation of expressions in prefix notation and the processing of abstract syntax trees by compilers.
2. Binary search trees (a special type of BT) use inorder traversal to print all of their data in alphanumeric order.
3. A popular application for the use of postorder traversal is the evaluating of expressions in postfix notation.

## EXPRESSION TREES

Algebraic expressions such as

$$a/b + (c-d)e$$



Tree representing the expression  $a/b+(c-d)e$ .

### Converting Expression from Infix to Postfix using STACK

To convert an expression from infix to postfix, we are going to use a stack.

#### Algorithm

- 1) Examine the next element in the input.
- 2) If it is an operand, output it.
- 3) If it is opening parenthesis, push it on stack.
- 4) If it is an operator, then
  - i) If stack is empty, push operator on stack.
  - ii) If the top of the stack is opening parenthesis, push operator on stack.
  - iii) If it has higher priority than the top of stack, push operator on stack.
  - iv) Else pop the operator from the stack and output it, repeat step 4.
- 5) If it is a closing parenthesis, pop operators from the stack and output them until an opening parenthesis is encountered. pop and discard the opening parenthesis.
- 6) If there is more input go to step 1
- 7) If there is no more input, unstack the remaining operators to output.

#### Example

Suppose we want to convert  $2*3/(2-1)+5*(4-1)$  into Prefix form: Reversed Expression: )1-4(\*5+)1-2(/3\*2

Char Scanned	Stack Contents(Top on right)	Postfix Expression
2	Empty	2
*	*	2

3	*	23
/	/	23*
(	/(	23*
2	/(	23*2
-	/(-	23*2
1	/(-	23*21
)	/	23*21-
+	+	23*21-/
5	+	23*21-/5
*	+*	23*21-/5
(	+*(	23*21-/5
4	+*(	23*21-/54
-	+*(-	23*21-/54
1	+*(-	23*21-/541
)	+*	23*21-/541-
	Empty	23*21-/541-*+

So, the Postfix Expression is  $23*21-/541-*+$

### Converting Expression from Infix to Prefix using STACK

It is a bit trickier algorithm, in this algorithm we first reverse the input expression so that  $a+b*c$  will become  $c*b+a$  and then we do the conversion and then again the output string is reversed. Doing this has an advantage that except for some minor modifications the algorithm for Infix->Prefix remains almost same as the one for Infix->Postfix.

#### Algorithm

- 1) Reverse the input string.
- 2) Examine the next element in the input.
- 3) If it is operand, add it to output string.
- 4) If it is Closing parenthesis, push it on stack.
- 5) If it is an operator, then

- i) If stack is empty, push operator on stack.
- ii) If the top of stack is closing parenthesis, push operator on stack.
- iii) If it has same or higher priority than the top of stack, push operator on stack.
- iv) Else pop the operator from the stack and add it to output string, repeat step 5.
- 6) If it is a opening parenthesis, pop operators from stack and add them to output string until a closing parenthesis is encountered. Pop and discard the closing parenthesis.
- 7) If there is more input go to step 2
- 8) If there is no more input, unstack the remaining operators and add them to output string.
- 9) Reverse the output string.

### Example

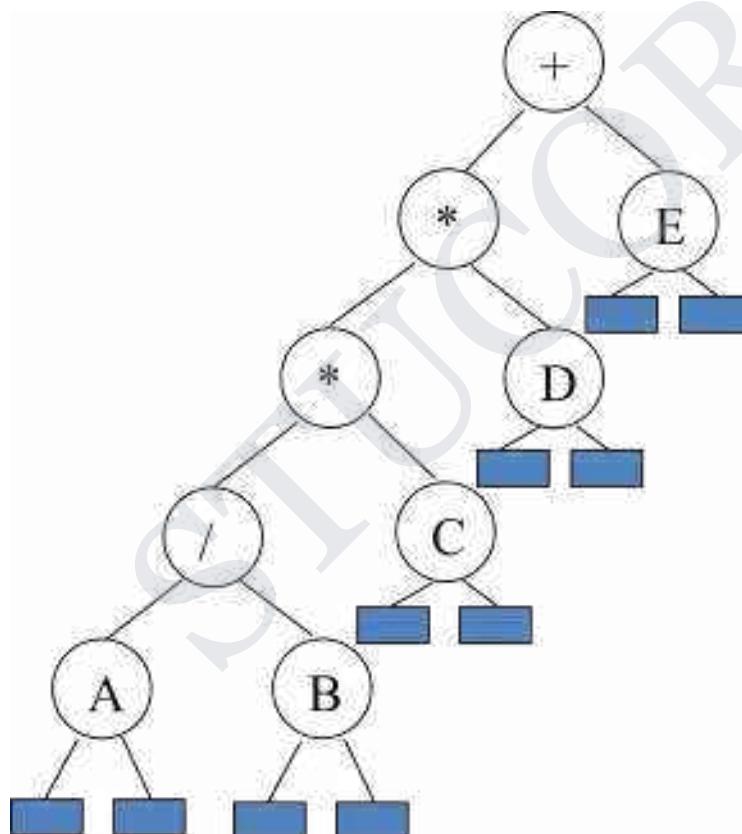
Suppose we want to convert  $2*3/(2-1)+5*(4-1)$  into Prefix form: Reversed Expression: )1-4(\*5+)1-2(/3\*2

Char Scanned	Stack Contents(Top on right)	Prefix Expression(right to left)
)	)	
1	)	1
-	)-	1
4	)-	14
(	Empty	14-
*	*	14-
5	*	14-5
+	+	14-5*
)	+)	14-5*
1	+)	14-5*1
-	+)-	14-5*1
2	+)-	14-5*12
(	+	14-5*12-

/	+ /	14-5*12-
3	+ /	14-5*12-3
*	+ /*	14-5*12-3
2	+ /*	14-5*12-32
	Empty	14-5*12-32*/+

Reverse the output string : +/\*23-21\*5-41 So, the final Prefix Expression is  
+/\*23-21\*5-41

## EVALUATION OF EXPRESSIONS



inorder traversal  
A / B \* C \* D + E  
infix expression

preorder traversal  
+ \* \* / A B C D E  
prefix expression

postorder traversal  
A B / C \* D \* E +  
postfix expression

## CONSTRUCTING AN EXPRESSION TREE

Let us consider the postfix expression given as the input, for constructing an expression tree by performing the following steps :

1. Read one symbol at a time from the postfix expression.
2. Check whether the symbol is an operand or operator.
  - i. If the symbol is an operand, create a one node tree and push a pointer on to the stack.
  - ii. If the symbol is an operator, pop two pointers from the stack namely,  $T_1$  and  $T_2$  and form a new tree with root as the operator, and  $T_2$  as the left child and  $T_1$  as the right child.
  - iii. A pointer to this new tree is then pushed on to the stack.

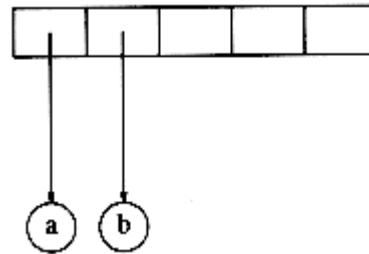
We now give an algorithm to convert a postfix expression into an expression tree. Since we already have an algorithm to convert infix to postfix, we can generate expression trees from the two common types of input. The method we describe strongly resembles the postfix evaluation algorithm of Section 3.2.3. We read our expression one symbol at a time. If the symbol is an operand, we create a one-node tree and push a pointer to it onto a stack. If the symbol is an operator, we pop pointers to two trees  $T_1$  and  $T_2$  from the stack ( $T_1$  is popped first) and form a new tree whose root is the operator and whose left and right children point to  $T_2$  and  $T_1$  respectively. A pointer to this new tree is then pushed onto the stack.

As an example, suppose the input is

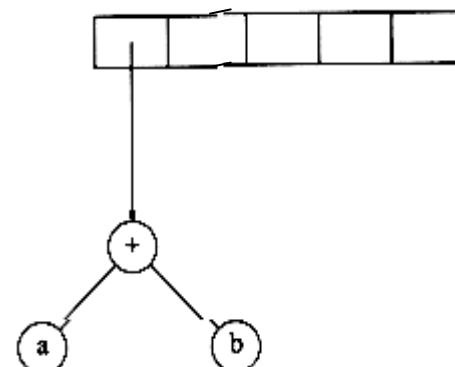
a b + c d e + \*\*

The first two symbols are operands, so we create one-node trees and push pointers to them onto a stack.\*

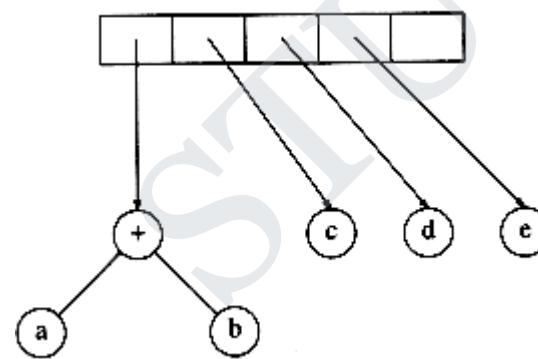
\*For convenience, we will have the stack grow from left to right in the diagrams.



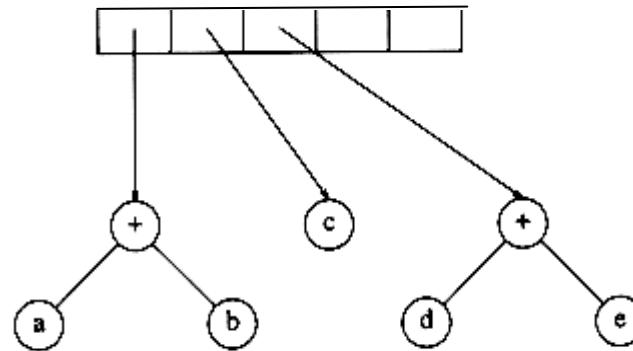
Next, a '+' is read, so two pointers to trees are popped, a new tree is formed, and a pointer to it is pushed onto the stack.\*



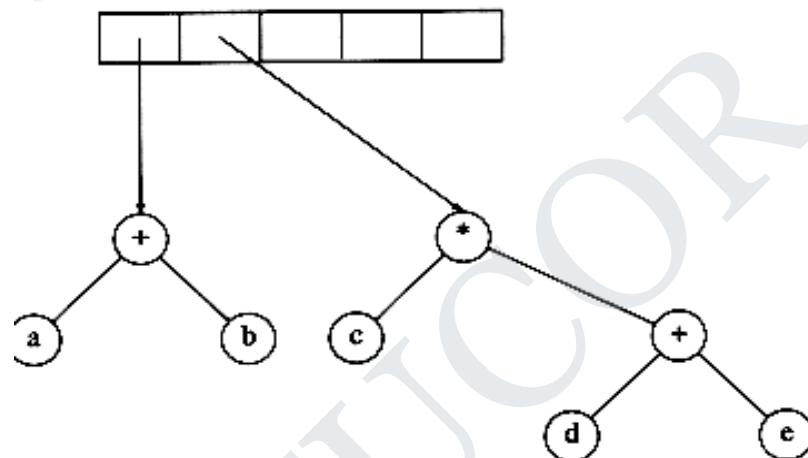
Next, *c*, *d*, and *e* are read, and for each a one-node tree is created and a pointer to the corresponding tree is pushed onto the stack.



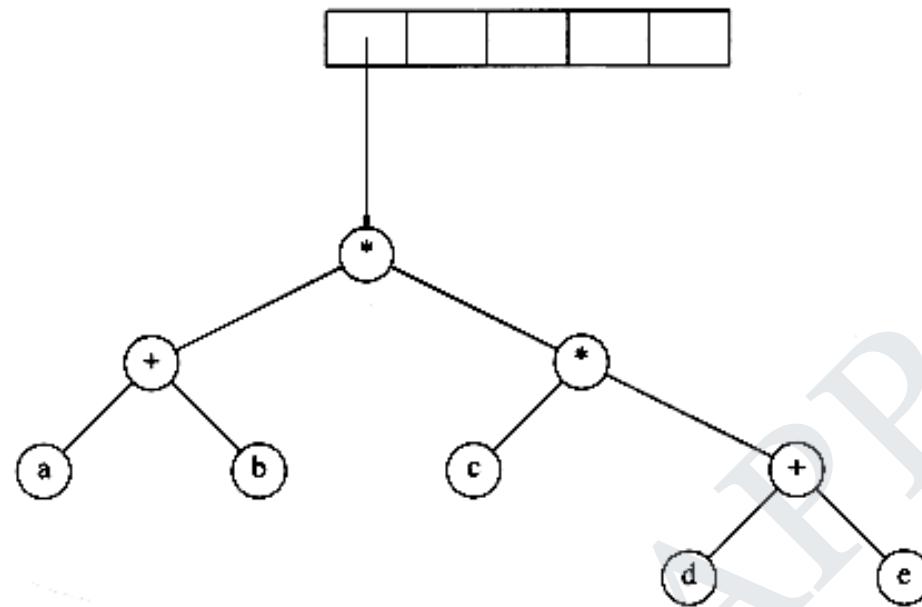
Now a '+' is read, so two trees are merged.



Continuing, a '\*' is read, so we pop two tree pointers and form a new tree with a '\*' as root.



Finally, the last symbol is read, two trees are merged, and a pointer to the final tree is left on the stack.



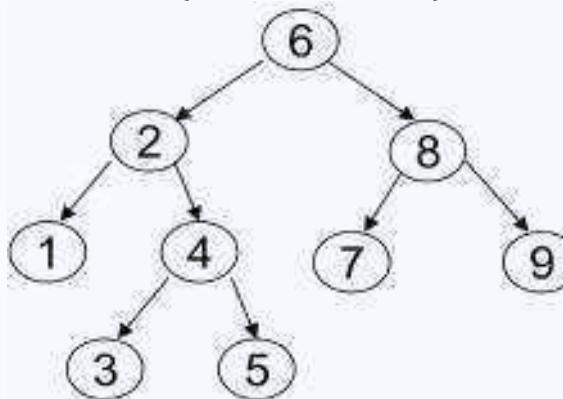
## BINARY SEARCH TREE

**Binary search tree (BST)** is a node-based binary tree data structure which has the following properties:

- The left sub-tree of a node contains only nodes with keys less than the node's key.
- The right sub-tree of a node contains only nodes with keys greater than the node's key.
- Both the left and right sub-trees must also be binary search trees.

From the above properties it naturally follows that:

- Each node (item in the tree) has a distinct key.



### Program: Creating a Binary Search Tree

We assume that every node of a binary search tree is capable of holding an integer data item and that the links can be made to point to the root of the left subtree and the right subtree, respectively. Therefore, the structure of the node can be defined using the following declaration:

```
struct tnode
{
    int data;
    struct tnode *lchild,*rchild;
};
```

A complete C program to create a binary search tree follows:

```
#include <stdio.h>
#include <stdlib.h>
struct tnode
{
    int data;
    struct tnode *lchild, *rchild;
};

struct tnode *insert(struct tnode *p,int val)
{
    struct tnode *temp1,*temp2;
    if(p == NULL)
    {
```

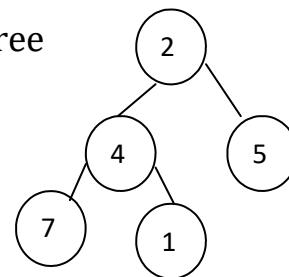
```
p = (struct tnode *) malloc(sizeof(struct tnode)); /* insert the new node as
root node*/
if(p == NULL)
{
printf("Cannot allocate\n");
exit(0);
}
p->data = val;
p->lchild=p->rchild=NULL;
}
else
{
temp1 = p;
/* traverse the tree to get a pointer to that node whose child will be the newly
created node*/
while(temp1 != NULL)
{
temp2 = temp1;
if( temp1 ->data > val)
temp1 = temp1->lchild;
else
temp1 = temp1->rchild;
}
if( temp2->data > val)
{
temp2->lchild = (struct tnode*)malloc(sizeof(struct tnode));/*inserts the
newly created node as left child*/
temp2 = temp2->lchild;
if(temp2 == NULL)
{
printf("Cannot allocate\n");
exit(0);
}
temp2->data = val;
temp2->lchild=temp2->rchild = NULL;
}
else
{
```

```
temp2->rchild = (struct tnode*)malloc(sizeof(struct tnode)); /*inserts the
newly created node
as left child*/
temp2 = temp2->rchild;
if(temp2 == NULL)
{
printf("Cannot allocate\n");
exit(0);
}
temp2->data = val;
temp2->lchild=temp2->rchild = NULL;
}
}
return(p);
}
/* a function to binary tree in inorder */
void inorder(struct tnode *p)
{
if(p != NULL)
{
inorder(p->lchild);
printf("%d\t",p->data);
inorder(p->rchild);
}
}
void main()
{
struct tnode *root = NULL;
int n,x;
printf("Enter the number of nodes\n");
scanf("%d",&n);
while( n -> 0)
{
printf("Enter the data value\n");
scanf("%d",&x);
root = insert(root,x);
}
inorder(root);
```

}

**EXAMPLE** Construct a BST with nodes 2,4,5,7,1

Normal Tree

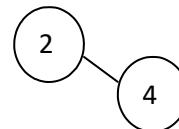


**Binary Search Tree**

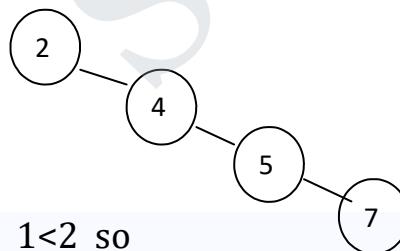
- The Values in the left subtree must be smaller than the keyvalue to be inserted.
- The Values in the right subtree must be larger than the keyvalue to be inserted.

Take the 1<sup>st</sup> element 2 and compare with 4.  $2 < 4$

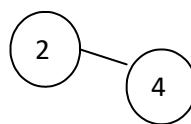
So



Similarly  $2 < 5, 5 > 4$  and  $7 > 2, 7 > 4, 7 > 5$



and  $1 < 2$  so





is the final BST.

## OPERATIONS

- Operations on a binary tree require comparisons between nodes. These
- comparisons are made with calls to a comparator, which is a subroutine that computes the total order (linear order) on any two values. This
- comparator can be explicitly or implicitly defined, depending on the language in which the BST is implemented.
- The following are the operations that are being done in Binary Tree
  - Searching.
  - Sorting.
  - Deletion.
  - Insertion.

### Binary search tree declaration routine

```
Struct treenode;  
Typedef struct treenode *position;  
Typedef struct treenode *searchtree;  
Typedef int elementtype;  
Structtreenode  
{  
Elementtype element;  
Searchtree left;  
Searchtree right;  
};  
Struct treenode  
{  
int element;  
struct treenode *left;  
struct treenode *right;
```

```
};
```

### **Make\_null**

This operation is mainly for initialization. Some programmers prefer to initialize the first element as a one-node tree, but our implementation follows the recursive definition of trees more closely. It is also a simple routine.

#### **Routine to make an empty tree**

```
SEARCH_TREE  
make_null ( void )  
{  
    return NULL;  
}
```

### **Find**

This operation generally requires returning a pointer to the node in tree  $T$  that has key  $x$ , or  $\text{NULL}$  if there is no such node. The structure of the tree makes this simple. If  $T$  is, then we can just return . Otherwise, if the key stored at  $T$  is  $x$ , we can return  $T$ . Otherwise, we make a recursive call on a subtree of  $T$ , either left or right, depending on the relationship of  $x$  to the key stored in  $T$ . The code in Figure 4.18 is an implementation of this strategy.

### **Find operation for binary search trees**

```
Position find(structtreenode T, intnum)  
{  
    While(T!=NULL)  
    {  
        if(num>T-->data)  
        {  
            T=T-->right;  
        if(num<T-->data)
```

```
T=T-->left;
}
else if(num< T-->data)
{
T=T-->left;
if(num>T-->data)
T=T-->right;
}
if(T-->data==num)
break;
}
return T;
}
// To find a Number
Position find(elementtype X, searchtree T)
{
If(T==NULL)
return NULL;
if(x< T-->element)
return find(x,T-->left);
else if(X> T-->element)
return find(X,T-->right);
else
return T;
}
```

### Find\_min and Find\_max

#### Recursive implementation of find\_min & find\_max for binary search trees

```
// Finding Minimum
Position findmin(searchtree T)
{
if(T==NULL)
```

```
return NULL;
else if(T-->left==NULL)
return T;
else return findmin(T-->left);
}
// Finding Maximum
Position findmax(searchtree T)
{
if(T==NULL)
return NULL;
else if(T-->right==NULL)
return T;
else return findmin(T-->right);
}
```

### **Nonrecursive implementation of find\_min & find\_max for binary search trees**

```
// Finding Maximum
Position findmax(searchtree T)
{
if( T!=NULL)
while(T-->Right!=NULL)
T=T-->right;
Return T;
}
// Finding Minimum
Position findmin(searchtree T)
{
if( T!=NULL)
while(T-->left!=NULL)
T=T-->left;
Return T;
}
```

## Insert

The insertion routine is conceptually simple. To insert  $x$  into tree  $T$ , proceed down the tree as you would with a *find*. If  $x$  is found, do nothing (or "update" something). Otherwise, insert  $x$  at the last spot on the path traversed. Figure below shows what happens. To insert 5, we traverse the tree as though a *find* were occurring. At the node with key 4, we need to go right, but there is no subtree, so 5 is not in the tree, and this is the correct spot.

Duplicates can be handled by keeping an extra field in the node record indicating the frequency of occurrence. This adds some extra space to the entire tree, but is better than putting duplicates in the tree (which tends to make the tree very deep). Of course this strategy does not work if the key is only part of a larger record. If that is the case, then we can keep all of the records that have the same key in an auxiliary data structure, such as a list or another search tree.

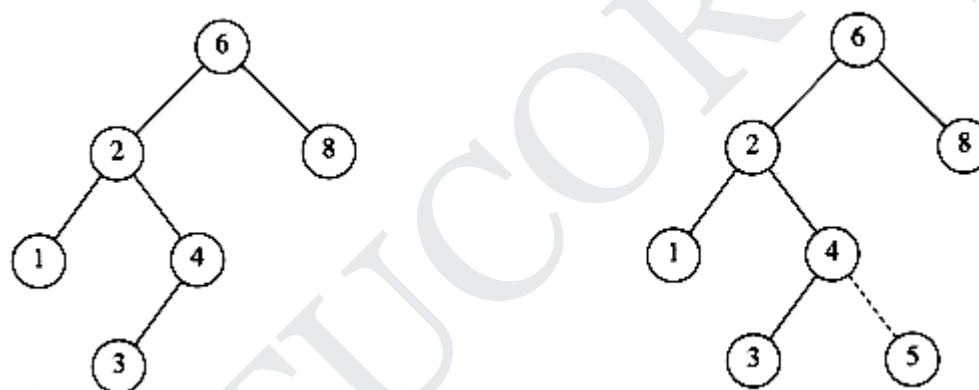
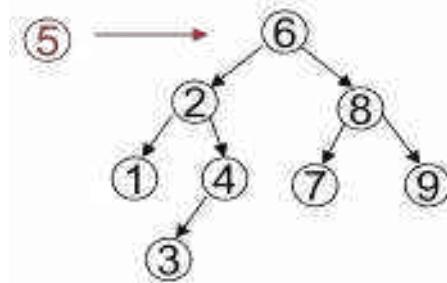


Figure shows the code for the insertion routine. Since  $T$  points to the root of the tree, and the root changes on the first insertion, *insert* is written as a function that returns a pointer to the root of the new tree. Lines 8 and 10 recursively insert and attach  $x$  into the appropriate subtree.

### Insertion into a binary search tree

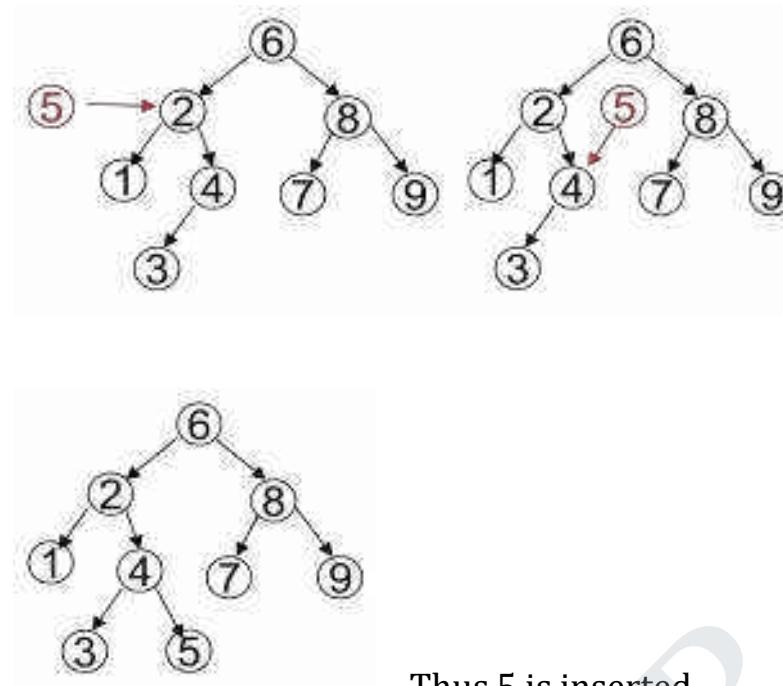
```
Searchtree insert(elementtype X, Searchtree T)
```

```
{  
If(T== NULL)  
{  
/* create and return a one node tree*/  
T=malloc(sizeof(structtreenode));  
If(T==NULL)  
Fatalerror("Out of Space");  
Else  
{  
T->element=X;  
T->left=T->right=NULL;  
}  
}  
Else if(x<T->element)  
T->left=insert(X,T->left);  
Else if(X>=T->left)  
T->right=insert(X,T->right);  
Return T;  
}
```



**EXAMPLE** Insert node 5 in given tree

**STEP 1:** Now  $5 < 6$  and  $5 > 2$  and  $5 < 4$  so



Thus 5 is inserted.

## Delete

As is common with many data structures, the hardest operation is deletion. Once we have found the node to be deleted, we need to consider several possibilities.

If the node is a leaf, it can be deleted immediately. If the node has one child, the node can be deleted after its parent adjusts a pointer to bypass the node (we will draw the pointer directions explicitly for clarity).. Notice that the deleted node is now unreferenced and can be disposed of only if a pointer to it has been saved. The complicated case deals with a node with two children. The general strategy is to replace the key of this node with the smallest key of the right subtree (which is easily found) and recursively delete that node (which is now empty). Because the smallest node in the right subtree cannot have a left child, the second *delete* is an easy one.

To delete an element, consider the following three possibilities :

Case 1: Node to be deleted is a leaf node.

Case 2: Node with only one child.

Case 3: Node with two children.

Case 1: Node with no children | Leaf node :

1. Search the parent of the leaf node and make the link to the leaf node as NULL.
2. Release the memory of the deleted node.

Case 2: Node with only one child :

1. Search the parent of the node to be deleted.
2. Assign the link of the parent node to the child of the node to be deleted.
3. Release the memory for the deleted node.

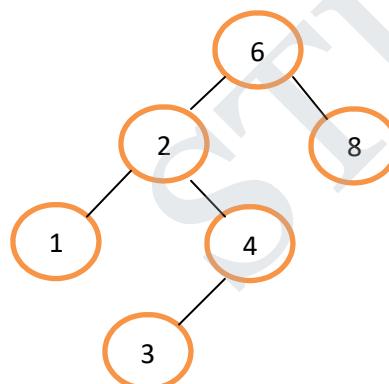
Case 3: Node with two children :

It is difficult to delete a node which has two children.

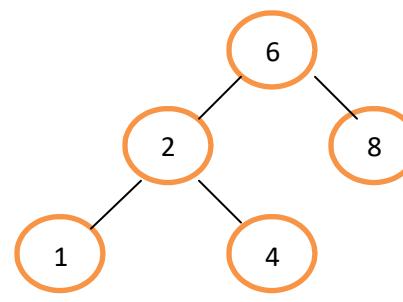
So, a general strategy has to be followed.

1. Replace the data of the node to be deleted with either the largest element from the left subtree or the smallest element from the right subtree.

**Case 1:**

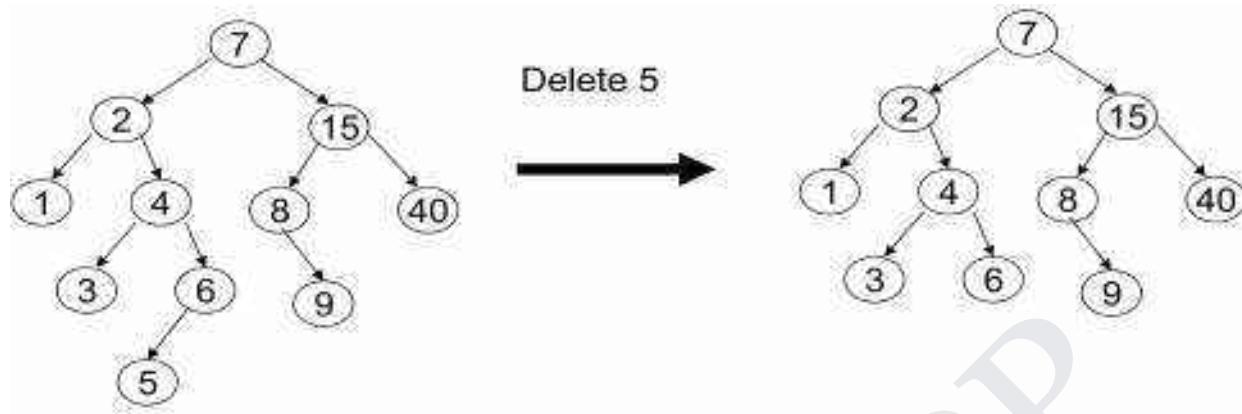


Before Deletion

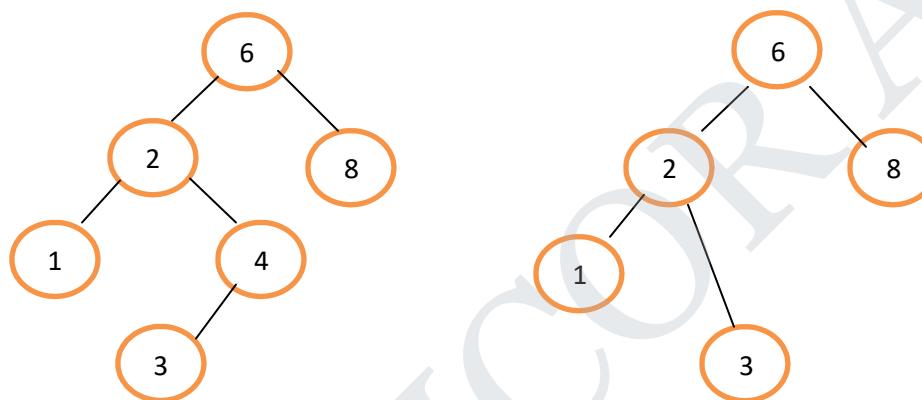


After Deletion of 3

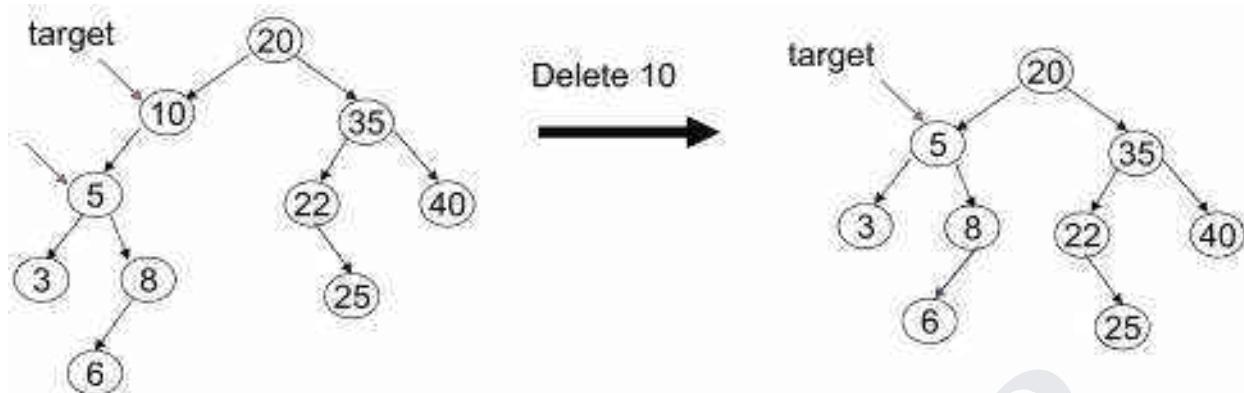
**EXAMPLE :**



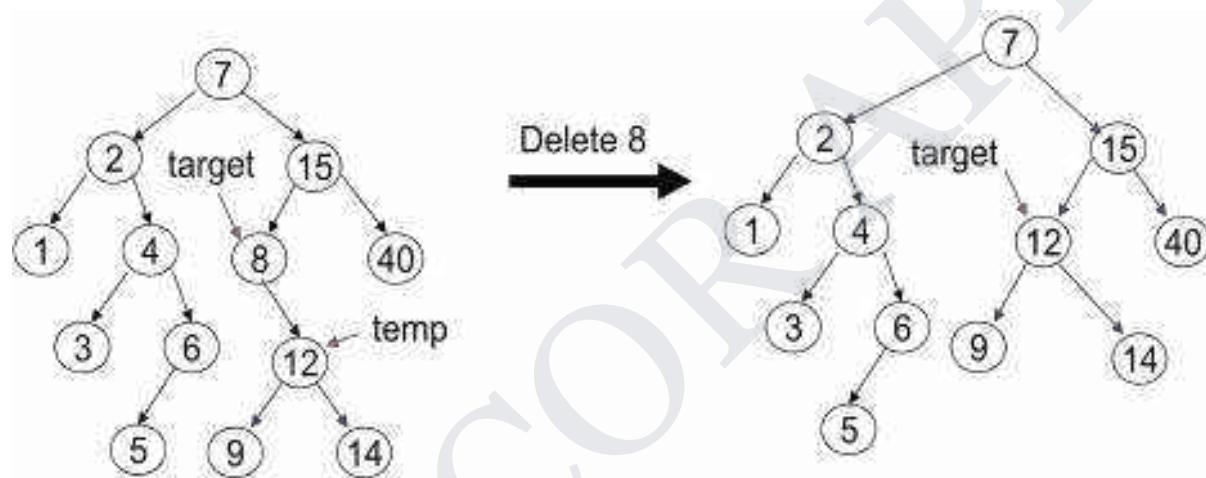
**Case 2 :**



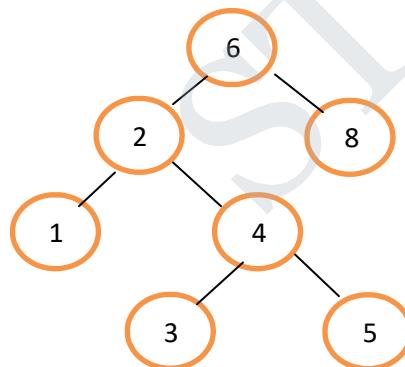
**EXAMPLE** The right subtree of the node x to be deleted is empty.



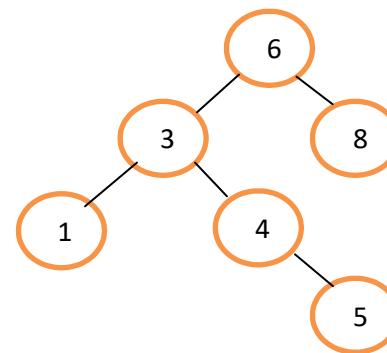
**EXAMPLE** The left subtree of the node x to be deleted is empty.



**Case 3 :**



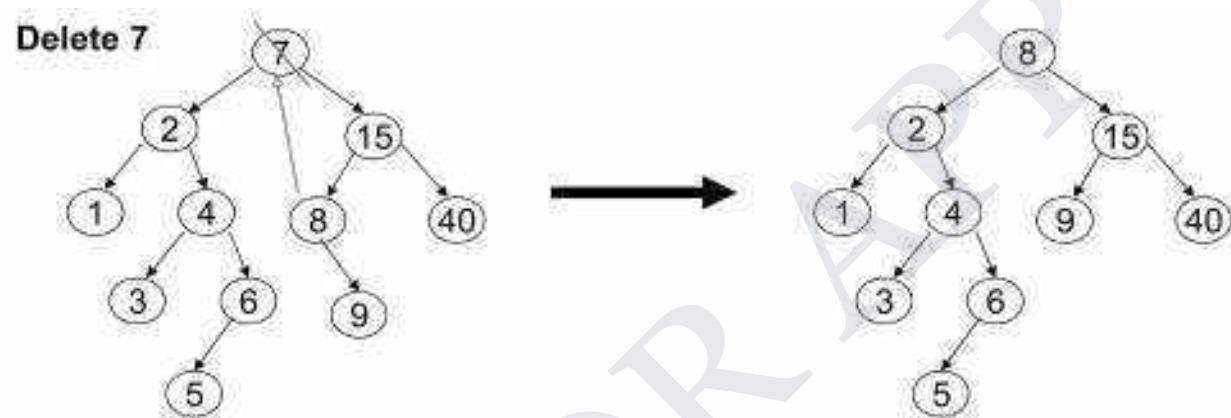
Before deletion of 2



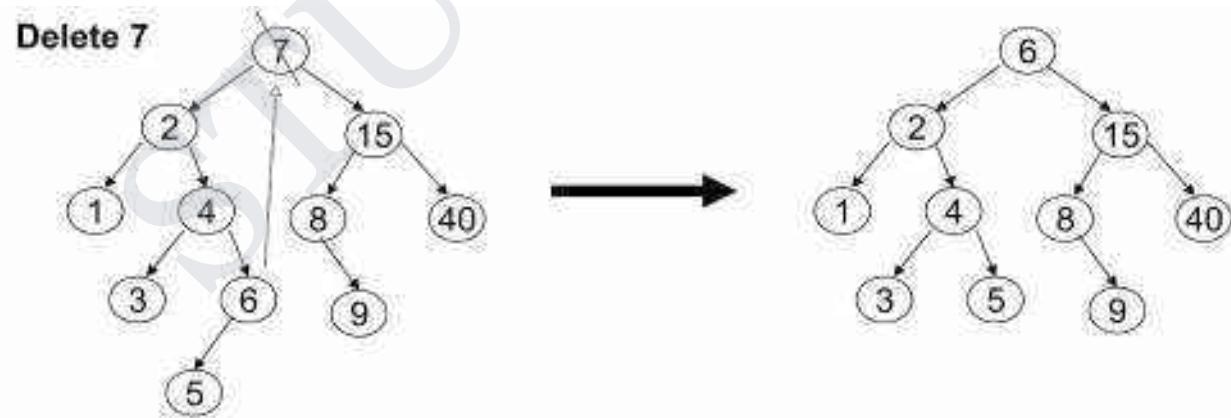
After deletion

**EXAMPLE****DELETION BY COPYING: METHOD#1**

Copy the **minimum key** in the **right subtree** of x to the node x, then delete the **one-child or leaf-node** with this minimum key.

**DELETION BY COPYING: METHOD#2**

Copy the **maximum key** in the **left subtree** of x to the node x, then delete the **one-child or leaf-node** with this **maximum key**.



The code performs deletion. It is inefficient, because it makes two passes down the tree to find and delete the smallest node in the right subtree when

this is appropriate. It is easy to remove this inefficiency, by writing a special *delete\_min* function, and we have left it in only for simplicity.

If the number of deletions is expected to be small, then a popular strategy to use is *lazy deletion*: When an element is to be deleted, it is left in the tree and merely *marked* as being deleted. This is especially popular if duplicate keys are present, because then the field that keeps count of the frequency of appearance can be decremented. If the number of real nodes in the tree is the same as the number of "deleted" nodes, then the depth of the tree is only expected to go up by a small constant (why?), so there is a very small time penalty associated with lazy deletion. Also, if a deleted key is reinserted, the overhead of allocating a new cell is avoided.

### **Deletion routine for binary search trees**

```
Searchtree delete(elementtype X, searchtree T)
{
    positiontmpcell;
    if(T==NULL)
        error("element not found");
    else if(X<T-->element)
        T-->left=delete(X,T-->left);
    Else if(X>T-->element)
        T-->right=delete(X,T-->right);
    Else if(T-->left != NULL && T-->right!=NULL)
    {
        /* Replace with smallest in right subtree*/
        Tmpcell=findmin(T-->right);
        T-->element=tmpcell-->element;
        T-->right=delete(T-->element,T-->right);
    }
    Else
    {
        /* One or Zero children*/
        tmpcell=T;
        if(T-->left==NULL)
```

```
T=T-->right;  
Else if(T-->right==NULL)  
T=T-->left;  
Free(tmpcell);  
}  
Return T;  
}
```

## COUNTING THE NUMBER OF NODES IN A BINARY SEARCH TREE

### Introduction

To count the number of nodes in a given binary tree, the tree is required to be traversed recursively until a leaf node is encountered. When a leaf node is encountered, a count of 1 is returned to its previous activation (which is an activation for its parent), which takes the count returned from both the children's activation, adds 1 to it, and returns this value to the activation of its parent. This way, when the activation for the root of the tree returns, it returns the count of the total number of the nodes in the tree.

### Program

A complete C program to count the number of nodes is as follows:

```
#include <stdio.h>  
#include <stdlib.h>  
struct tnode  
{  
    int data;  
    struct tnode *lchild, *rchild;  
};  
int count(struct tnode *p)  
{  
    if( p == NULL)  
        return(0);  
    else
```

```
if( p->lchild == NULL && p->rchild == NULL)
return(1);
else
return(1 + (count(p->lchild) + count(p->rchild)));


}

struct tnode *insert(struct tnode *p,int val)
{
struct tnode *temp1,*temp2;
if(p == NULL)
{
p = (struct tnode *) malloc(sizeof(struct tnode)); /* insert the new node as root
node*/
if(p == NULL)
{
printf("Cannot allocate\n");
exit(0);
}
p->data = val;
p->lchild=p->rchild=NULL;
}
else
{
temp1 = p;
/* traverse the tree to get a pointer to that node whose child will be the newly
created node*/
while(temp1 != NULL)
{
temp2 = temp1;
if( temp1 ->data > val)
temp1 = temp1->lchild;
else
temp1 = temp1->rchild;
}
if( temp2->data > val)
{
temp2->lchild = (struct tnode*)malloc(sizeof(struct tnode)); /
```

```
*inserts the newly created node
as left child*/
temp2 = temp2->lchild;
if(temp2 == NULL)
{
printf("Cannot allocate\n");
exit(0);
}
temp2->data = val;
temp2->lchild=temp2->rchild = NULL;
}
else
{
temp2->rchild = (struct tnode*)malloc(sizeof(struct tnode));/*inserts the
newly created node
as left child*/
temp2 = temp2->rchild;
if(temp2 == NULL)
{
printf("Cannot allocate\n");
exit(0);
}
temp2->data = val;
temp2->lchild=temp2->rchild = NULL;
}
}
return(p);
}
/* a function to binary tree in inorder */
void inorder(struct tnode *p)
{
if(p != NULL)
{
inorder(p->lchild);
printf("%d\t",p->data);
inorder(p->rchild);
}
}
```

```
void main()
{
    struct tnode *root = NULL;
    int n,x;
    printf("Enter the number of nodes\n");
    scanf("%d",&n);
    while( n --- > 0)
    {
        printf("Enter the data value\n");
        scanf("%d",&x);
        root = insert(root,x);
    }
    inorder(root);
    printf("\nThe number of nodes in tree are :%d\n",count(root));
}
```

### Explanation

- Input:
  - 1.The number of nodes that the tree to be created should have
  - 2. The data values of each node in the tree to be created
- Output:
  - The data value of the nodes of the tree in inorder
  - 2. The count of number of node in a tree.

### Example

- Input:
  - 1.The number of nodes the created tree should have = 5
  - 2. The data values of nodes in the tree to be created are: 10, 20, 5, 9, 8
- Output: 1. 5 8 9 10 20
  - 2. The number of nodes in the tree is 5

## SWAPPING OF LEFT & RIGHT SUBTREES OF A GIVEN BINARY TREE

### Introduction

An elegant method of swapping the left and right subtrees of a given binary tree makes use of a recursive algorithm, which recursively swaps the left and right subtrees, starting from the root.

### Program

```
#include <stdio.h>
#include <stdlib.h>
struct tnode
{
    int data;
    struct tnode *lchild, *rchild;
};

struct tnode *insert(struct tnode *p,int val)
{
    struct tnode *temp1,*temp2;
    if(p == NULL)
    {
        p = (struct tnode *) malloc(sizeof(struct tnode)); /* insert the new node as
root node*/
        if(p == NULL)
        {
            printf("Cannot allocate\n");
            exit(0);
        }
        p->data = val;
        p->lchild=p->rchild=NULL;
    }
    else
    {
        temp1 = p;
        /* traverse the tree to get a pointer to that node whose child will be the newly
created node*/
        while(temp1 != NULL)
        {
            temp2 = temp1;
            if( temp1 ->data > val)
                temp1 = temp1->lchild;
            else
                temp1 = temp1->rchild;
        }
        if(temp2->data > val)
            temp2->lchild = insert(temp2->lchild, val);
        else
            temp2->rchild = insert(temp2->rchild, val);
    }
}
```

```
else
temp1 = temp1->rchild;
}
if( temp2->data > val)
{
temp2->lchild = (struct tnode*)malloc(sizeof(struct tnode));/*inserts the
newly created node
as left child*/
temp2 = temp2->lchild;
if(temp2 == NULL)
{
printf("Cannot allocate\n");
exit(0);
}
temp2->data = val;
temp2->lchild=temp2->rchild = NULL;
}
else
{
temp2->rchild = (struct tnode*)malloc(sizeof(struct tnode));/*inserts the
newly created node
as left child*/
temp2 = temp2->rchild;
if(temp2 == NULL)
{
printf("Cannot allocate\n");
exit(0);
}
temp2->data = val;
temp2->lchild=temp2->rchild = NULL;
}
}
return(p);
}
/* a function to binary tree in inorder */
void inorder(struct tnode *p)
{
if(p != NULL)
```

```
{  
inorder(p->lchild);  
printf("%d\t",p->data);  
inorder(p->rchild);  
}  
}  
struct tnode *swaptree(struct tnode *p)  
{  
struct tnode *temp1=NULL, *temp2=NULL;  
if( p != NULL)  
{ temp1= swaptree(p->lchild);  
temp2 = swaptree(p->rchild);  
p->rchild = temp1;  
p->lchild = temp2;  
}  
return(p);  
}  
void main()  
{  
struct tnode *root = NULL;  
int n,x;  
printf("Enter the number of nodes\n");  
scanf("%d",&n);  
while( n - > 0)  
{  
printf("Enter the data value\n");  
scanf("%d",&x);  
root = insert(root,x);  
}  
printf("The created tree is :\n");  
inorder(root);  
printf("The tree after swapping is :\n");  
root = swaptree(root);  
inorder(root);  
printf("\nThe original tree is \n");  
root = swaptree(root);  
inorder(root);  
}
```

## Explanation

- Input:
  - 1.The number of nodes that the tree to be created should have
  - 2. The data values of each node in the tree to be created
- Output:
  - 1.The data value of the nodes of the tree in inorder before interchanging the left and right subtrees
  - 2. The data value of the nodes of the tree in inorder after interchanging the left and right subtrees

## Example

- Input:
  - 1.The number of nodes that the created tree should have = 5
  - 2. The data values of the nodes in tree to be created are: 10, 20, 5, 9, 8
- Output:
  - 1. 5 8 9 10 20
  - 2. 20 10 9 8 5

## Applications of Binary Search Trees

One of the applications of a binary search tree is the implementation of a dynamic dictionary. This application is appropriate because a dictionary is an ordered list that is required to be searched frequently, and is also required to be updated (insertion and deletion mode) frequently. So it can be implemented by making the entries in a dictionary into the nodes of a binary search tree. A more efficient implementation of a dynamic dictionary involves considering a key to be a sequence of characters, and instead of searching by comparison of entire keys, we use these characters to determine a multi-way branch at each step. This will allow us to make a 26-way branch according to the first letter, followed by another branch according to the second letter and so on.

## Applications of Trees

1. Compiler Design.
2. Unix / Linux.
3. Database Management.
4. Trees are very important data structures in computing.
5. They are suitable for:
  - a. Hierarchical structure representation, e.g.,
    - i. File directory.
    - ii. Organizational structure of an institution.
    - iii. Class inheritance tree.
  - b. Problem representation, e.g.,
    - i. Expression tree.
    - ii. Decision tree.
  - c. Efficient algorithmic solutions, e.g.,
    - i. Search trees.
    - ii. Efficient priority queues via heaps.

## AVL TREE

The AVL tree is named after its two inventors, G.M. Adelson-Velsky and E.M. Landis, who published it in their 1962 paper "An algorithm for the organization of information."

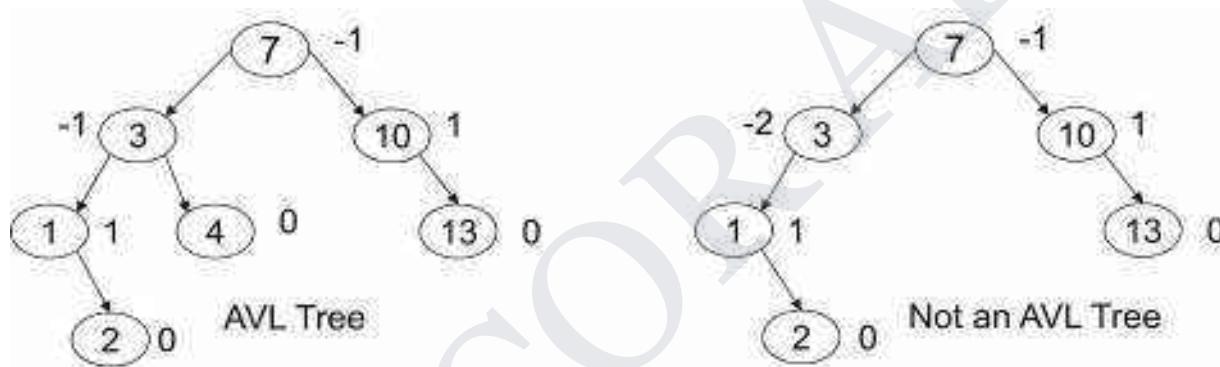
Avl tree is a self-balancing binary search tree. In an AVL tree, the heights of the two child subtrees of any node differ by at most one; therefore, it is also said to be height-balanced.

The balance factor of a node is the height of its right subtree minus the height of its left subtree and a node with balance factor 1, 0, or -1 is considered balanced. A node with any other balance factor is considered unbalanced and requires rebalancing the tree. This can be done by avl tree rotations

## Need for AVL tree

- The disadvantage of a binary search tree is that its height can be as large as  $N-1$
- This means that the time needed to perform insertion and deletion and many other operations can be  $O(N)$  in the worst case
- We want a tree with small height
- A binary tree with  $N$  node has height at least  $\Omega(\log N)$
- Thus, our goal is to keep the height of a binary search tree  $O(\log N)$
- Such trees are called balanced binary search trees. Examples are AVL tree, red-black tree.

Thus we go for AVL tree.



## HEIGHTS OF AVL TREE

An AVL tree is a special type of binary tree that is always "partially" balanced. The criteria that is used to determine the "level" of "balanced-ness" which is the difference between the heights of subtrees of a root in the tree. The "height" of tree is the "number of levels" in the tree. The height of a tree is defined as follows:

1. The height of a tree with no elements is 0
2. The height of a tree with 1 element is 1
3. The height of a tree with  $> 1$  element is equal to  $1 + \text{the height of its tallest subtree}$ .
4. The height of a leaf is 1. The height of a null pointer is zero.

The height of an internal node is the maximum height of its children plus 1.

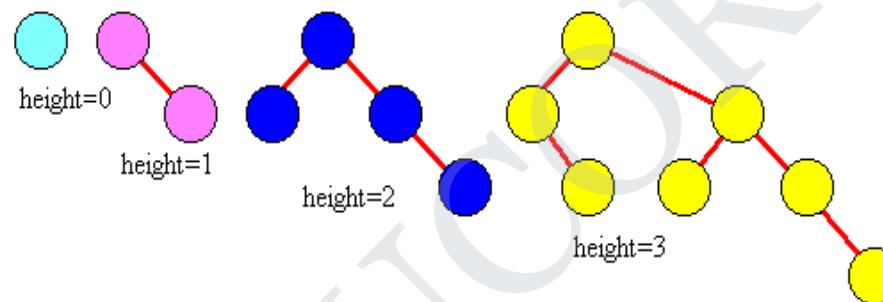
### FINDING THE HEIGHT OF AVL TREE

AVL trees are identical to standard binary search trees except that for every node in an AVL tree, the height of the left and right subtrees can differ by at most 1 . AVL trees are HB-k trees (height balanced trees of order k) of order HB-1. The following is the height differential formula:

$$| \text{Height (Tl)} - \text{Height(Tr)} | \leq k$$

When storing an AVL tree, a field must be added to each node with one of three values: 1, 0, or -1. A value of 1 in this field means that the left subtree has a height one more than the right subtree. A value of -1 denotes the opposite. A value of 0 indicates that the heights of both subtrees are the same.

#### EXAMPLE FOR HEIGHT OF AVL TREE



**An AVL tree is a binary search tree with a balanced condition.**

$$\text{Balance Factor(BF)} = H_l - H_r$$

$\Rightarrow$  Height of the left subtree.  $H_l$

$\Rightarrow$  Height of the right subtree.

If  $BF = \{-1, 0, 1\}$  is satisfied, only then the tree is balanced.

AVL tree is a Height Balanced Tree.

If the calculated value of BF goes out of the range, then balancing has to be done.

### Rotation :

Modification to the tree. i.e., If the AVL tree is Imbalanced, proper rotations has to be done.

A rotation is a process of switching children and parents among two or three adjacent nodes to restore balance to a tree.

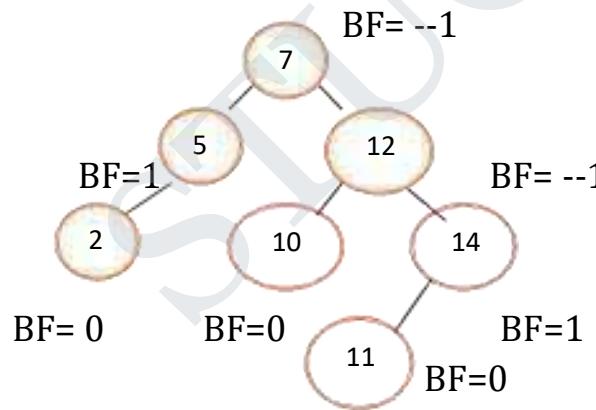
- There are two kinds of single rotation:



**An insertion or deletion may cause an imbalance in an AVL tree.**

The deepest node, which is an ancestor of a deleted or an inserted node, and whose balance factor has changed to -2 or +2 requires rotation to rebalance the tree.

### Balance Factor :



This Tree is an AVL Tree and a height balanced tree.

**An AVL tree causes imbalance when any of following condition occurs:**

- i. An insertion into Right child's right subtree.
- ii. An insertion into Left child's left subtree.
- iii. An insertion into Right child's left subtree.

iv. An insertion into Left child's right subtree.

These imbalances can be overcome by,

**1. Single Rotation** - ( If insertion occurs on the outside,i.e.,LL or RR)

-> LL (Left -- Left rotation) --- Do single Right.

-> RR (Right -- Right rotation) – Do single Left.

**2. Double Rotation** - ( If insertion occurs on the inside,i.e.,LR or RL)

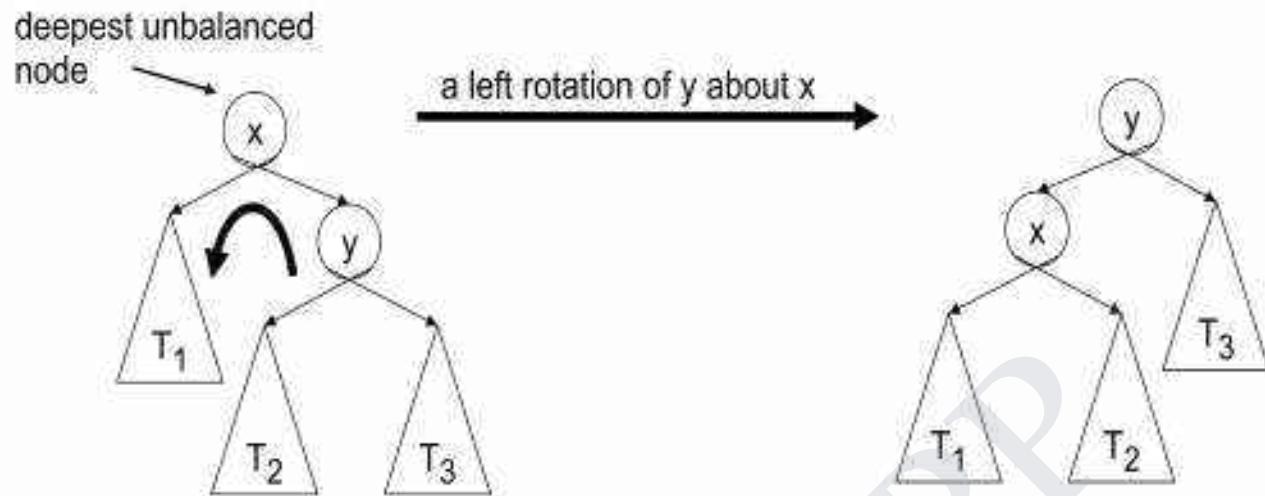
-> RL ( Right -- Left rotation) --- Do single Right, then single Left.

-> LR ( Left -- Right rotation) --- Do single Left, then single Right.

## General Representation of Single Rotation

### 1. LL Rotation :

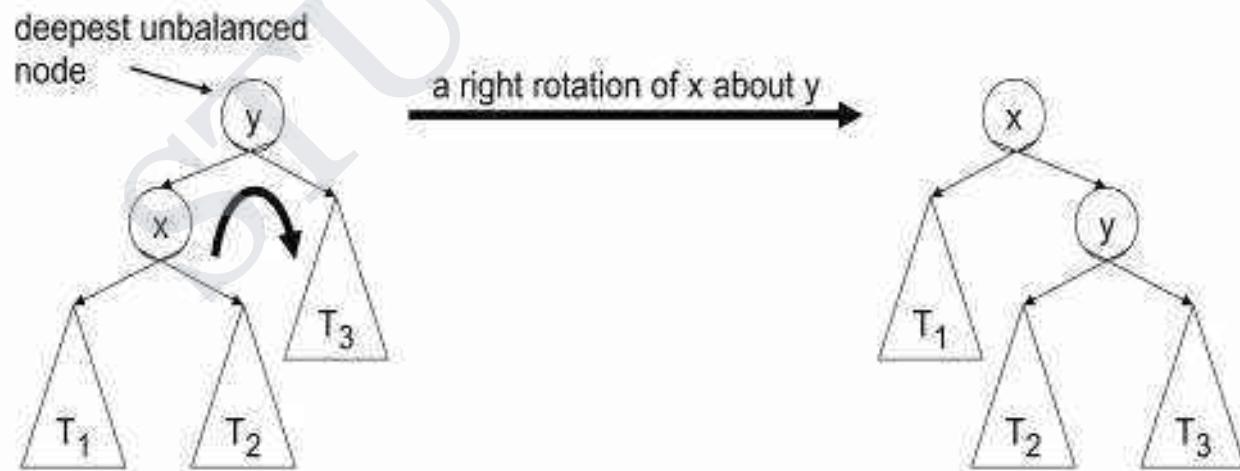
- The right child y of a node x becomes x's parent.
- x becomes the left child of y.
- The left child  $T_2$  of y, if any, becomes the right child of x.



Note: The pivot of the rotation is the deepest unbalanced node

## 2. RR Rotation :

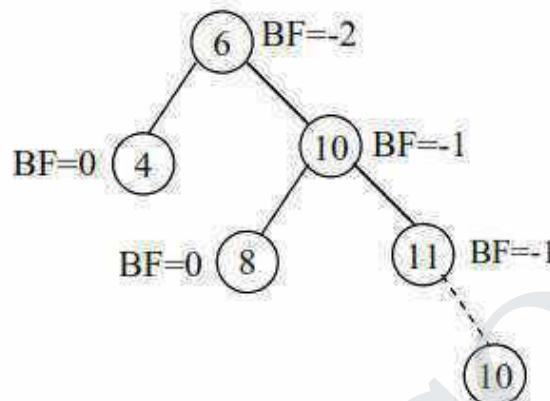
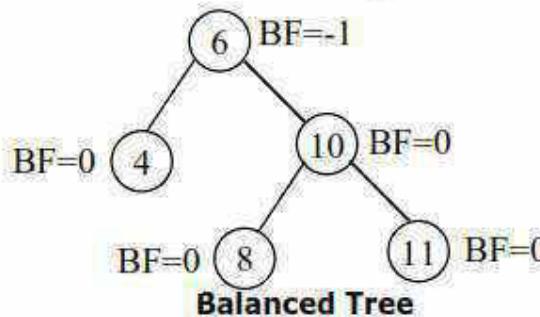
- The left child  $x$  of a node  $y$  becomes  $y$ 's parent.
- $y$  becomes the right child of  $x$ .
- The right child  $T_2$  of  $x$ , if any, becomes the left child of  $y$ .



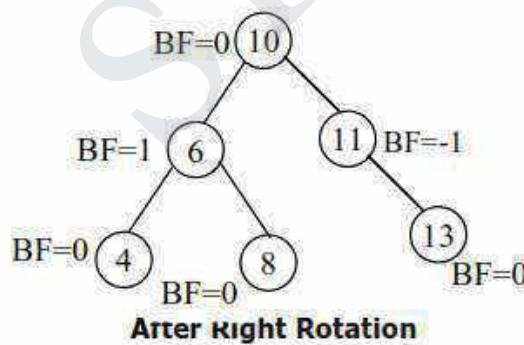
Note: The pivot of the rotation is the deepest unbalanced node

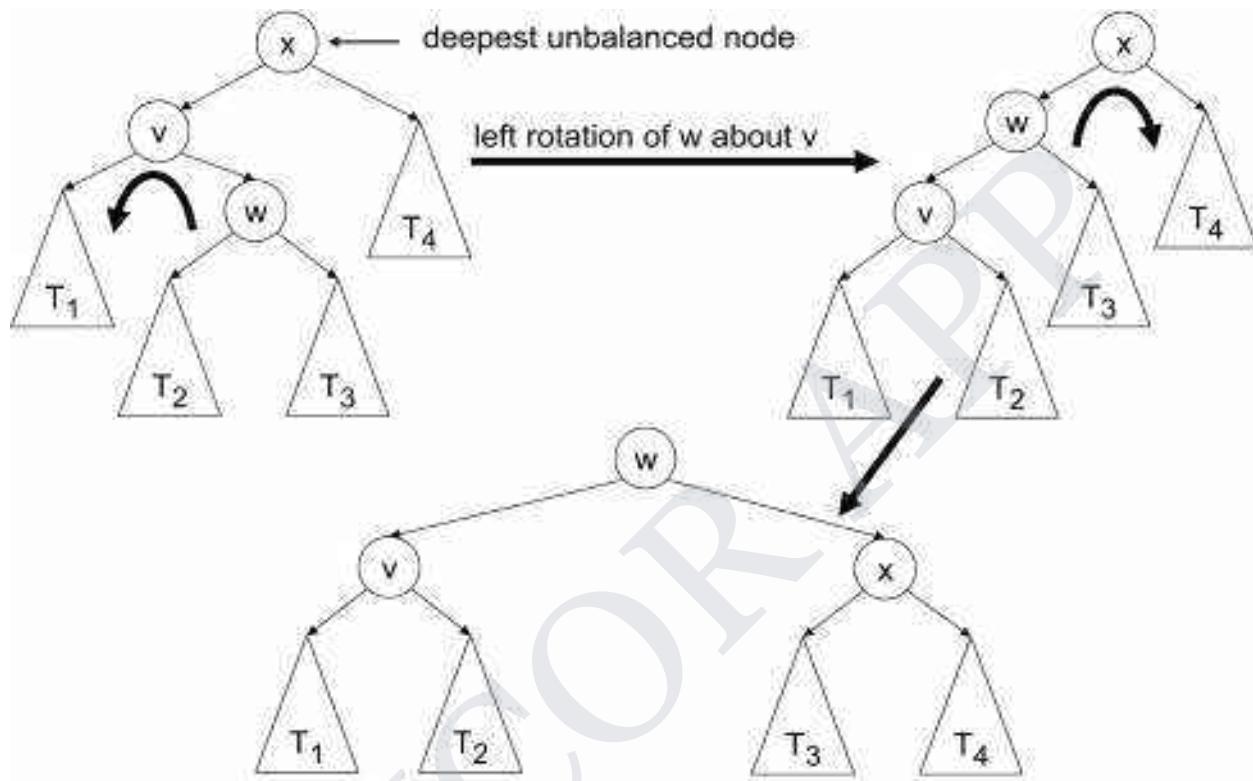
**Example:**

Consider the following tree which is balanced.



Now insert the value '13' it becomes unbalanced due to the insertion of new node in the Right Subtree of the Right Subtree. So we have to make single Right rotation in the root node.

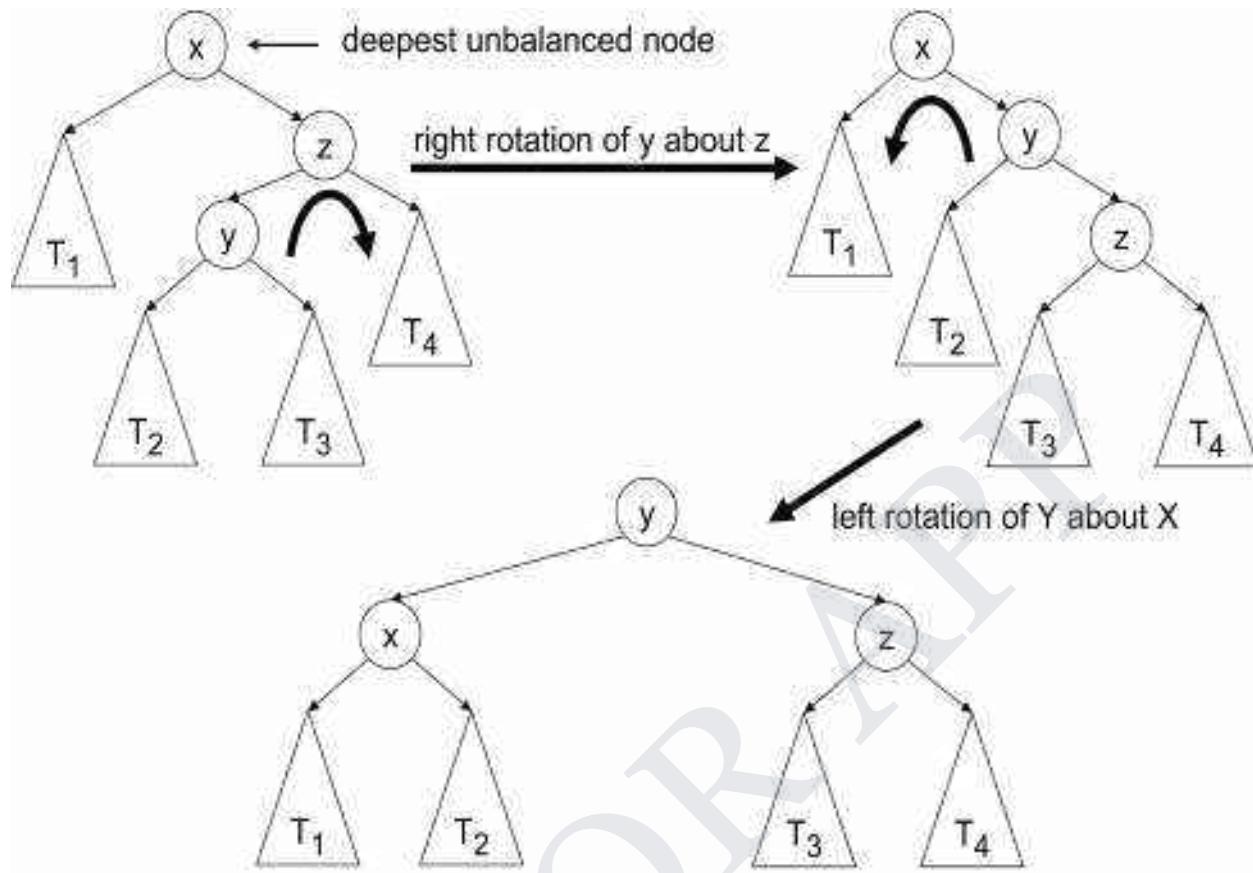
**General Representation of Double Rotation**

**1. LR( Left -- Right rotation):**

**Note:** First pivot is the left child of the deepest unbalanced node; second pivot is the deepest unbalanced node

**2. RL( Right -- Left rotation) :**

**Note:** First pivot is the right child of the deepest unbalanced node; second pivot is the deepest unbalanced node

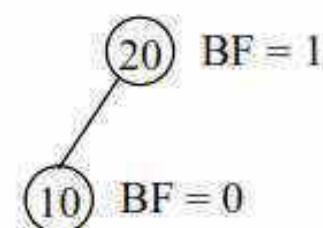
**EXAMPLE:**

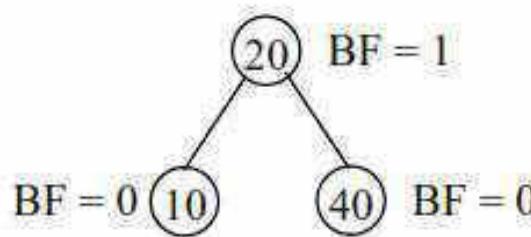
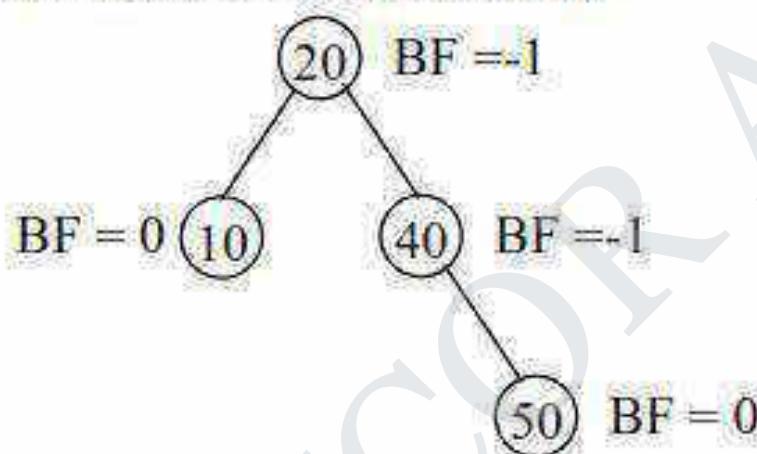
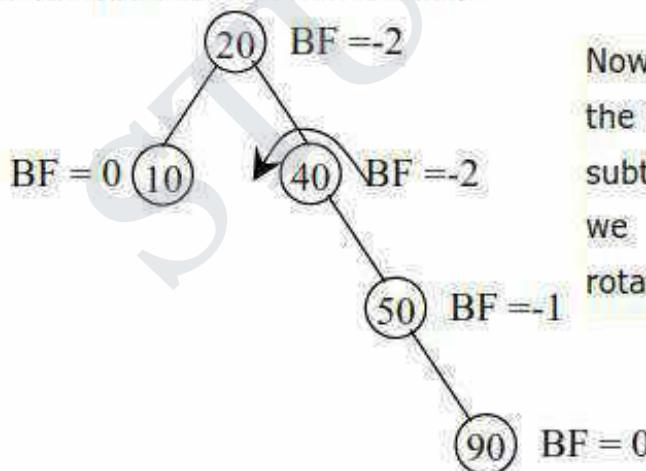
LET US CONSIDER INSERTING OF NODES 20,10,40,50,90,30,60,70 in an AVL TREE

**Step 1:(Insert the value 20)**

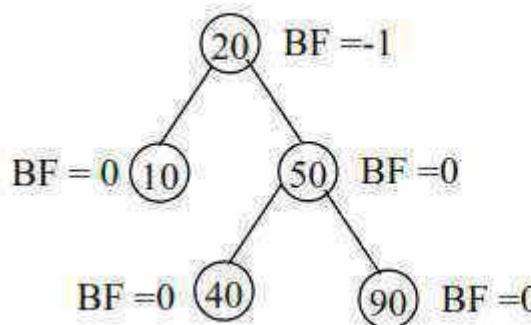


**Step 2: (Insert the value 10)**

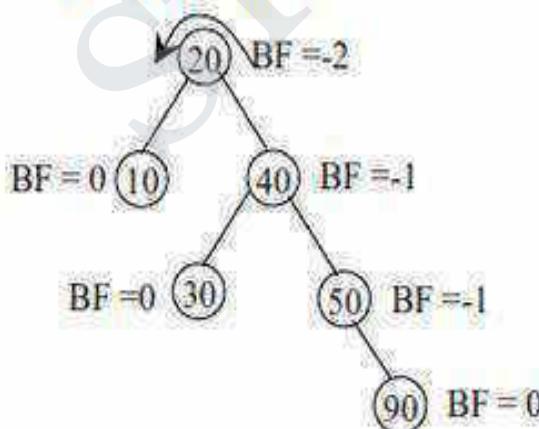
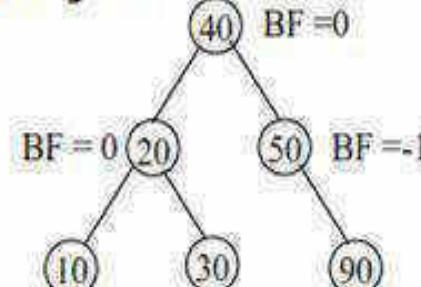


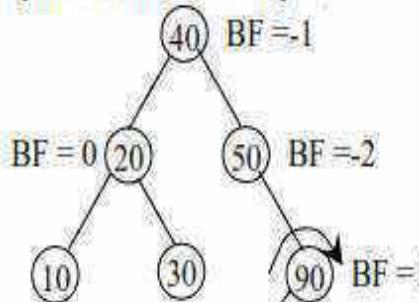
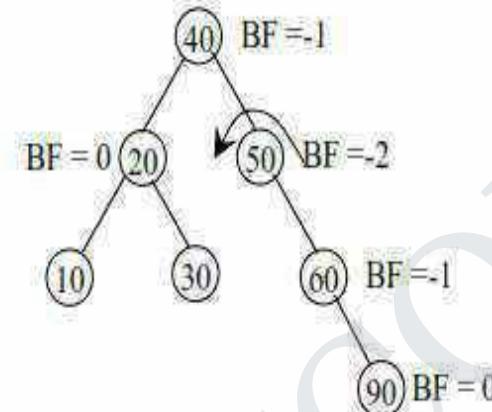
**Step 3: (Insert the value 40)****Step 4:(Insert the value 50)****Step 5: (Insert the value 90)**

Now the tree is unbalanced due to the insertion of node in the Right subtree of the Right subtree. So we have to make single Right rotation in the node '40'.

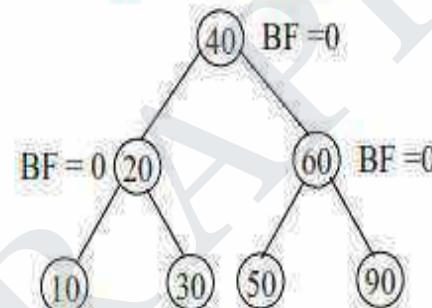
**After the Right Rotation:****Now the tree is Balanced****Step 6: Insert the value 30:**

Now the tree is unbalanced due to the insertion of value 30 in the left subtree of the right child 40. So we have to make Double rotation first with left on the node '30' and then with Right on the node '20'.

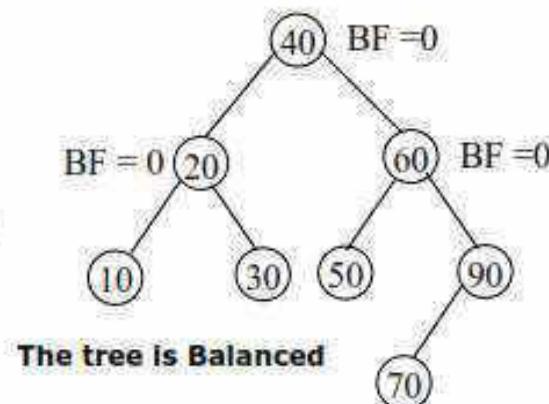
**After Left Rotation:****After Right Rotation:****Now the tree is Balanced**

**Step 7: (Insert the value 60)****After Left Rotation:**

Now the tree at the node '50' is unbalanced due to the insertion of node '60' in the Left subtree of the Right subtree. So we have to make Double rotation first with Left on the node '90' and then with Right on the node '50'.

**After Right Rotation:**

**Now the tree is Balanced**

**Step 8: (Insert the value 70)**

**The tree is Balanced**

## **AVL TREE ROUTINES**

### **Creation of AVL Tree and Insertion**

Struct avlnode

Typedef struct avlnode \*position;

Typedef structavlnode \*avltree;

Typedef int elementtype;

Struct avlnode

{

Elementtype element;

Avltree left;

Avltree right;

Int height;

};

Static int height(position P)

{

If(P==NULL)

return -1;

else

return P-->height;

}

Avltree insert(elementtype X, avltree T)

{

If(T==NULL)

{ /\* Create and return a one node tree\*/

T= malloc(sizeof(structavlnode));

If(T==NULL)

Fatalerror("Out of Space");

Else

{

T-->element=X;

```
T-->height=0;
T-->left=T-->right=NULL;
}
}
Else if(X<T-->element)
{
T-->left=Insert(X,T-->left);
If(height(T-->left) - height(T-->right)==2)
If(X<T-->left-->element)
T=singlerotatewithleft(T);
Else
T=doublerotatewithleft(T);
}
Else if(X>T-->element)
{
T-->right=insert(X,T-->right);
If(height(T-->left) - height(T-->right)==2)
If(X>T-->right-->element)
T= singlerotatewithright(T);
Else
T= doublerotatewithright(T);
}
T-->height=max(height(T-->left),height(T-->right)) + 1;
Return T;
}
```

### Routine to perform Single Left :

- . This function can be called only if k2 has a left child.
- . Perform a rotate between a node k2 and its left child.
- . Update height, then return the new root.

Static position singlerotatewithleft(position k2)

```
{  
Position k1;  
k1=k2-->left;  
k2-->left=k1-->right;  
k1-->right=k2;  
k2-->height= max(height(k2-->left),height(k2-->right)) + 1;  
k1-->height= max(height(k1-->left),height(k1-->right)) + 1;  
return k1; /* New Root */  
}
```

### Routine to perform Single Right :

```
Static position singlerotationwithright(position k1)  
{  
position k2;  
k2=k1-->right;  
k1-->right=k2-->left;  
k2-->left=k1;  
k2-->height=max(height(k2-->left),height(k2-->right)) + 1;  
k1-->height=max(height(k1-->left),height(k1-->right)) + 1;  
return k1; /* New Root */  
}
```

### Double rotation with Left :

```
Static position doublerotationwithleft(position k3)  
{  
/* Rotate between k1 & k2 */  
k3-->left=singlerotatewithright(k3-->left);  
/* Rotate between k3 & k2 */  
returnsinglerotatewithleft(k3);  
}
```

**Double rotation with Right :**

Static position doublerotatewithright(position k1)

```
{  
/* Rotation between k2& k3 */  
k1-->right=singlerotatewithleft(k1-->right);  
/* Rotation between k1 &k2 */  
returnsinglerotatewithright(k1);  
}
```

**PROBLEMS****APPLICATIONS**

AVL trees play an important role in most computer related applications. The need and use of avl trees are increasing day by day. their efficiency and less complexity add value to their reputation. Some of the applications are

- Contour extraction algorithm
- Parallel dictionaries
- Compression of computer files
- Translation from source language to target language
- Spell checker

**ADVANTAGES OF AVL TREE**

- AVL trees guarantee that the difference in height of any two subtrees rooted at the same node will be at most one. This guarantees an asymptotic running time of  $O(\log(n))$  as opposed to  $O(n)$  in the case of a standard bst.
- Height of an AVL tree with  $n$  nodes is always very close to the theoretical minimum.
- Since the avl tree is height balanced the operation like insertion and deletion have low time complexity.

- Since tree is always height balanced. Recursive implementation is possible.
- The height of left and the right sub-trees should differ by atmost 1. Rotations are possible.

## DISADVANTAGES OF AVL TREE

- one limitation is that the tree might be spread across memory
- as you need to travel down the tree, you take a performance hit at every level down
- one solution: store more information on the path
- Difficult to program & debug ; more space for balance factor.
- asymptotically faster but rebalancing costs time.
- most larger searches are done in database systems on disk and use other structures

## BINARY HEAPS

A **heap** is a specialized complete tree structure that satisfies the *heap property*:

- it is empty or
- the key in the root is larger than that in either child and both subtrees have the heap property.
- In general heap is a group of things placed or thrown, one on top of the other.
- In data structures a heap is a binary tree storing keys at its nodes.
- Heaps are based on the concepts of a complete tree

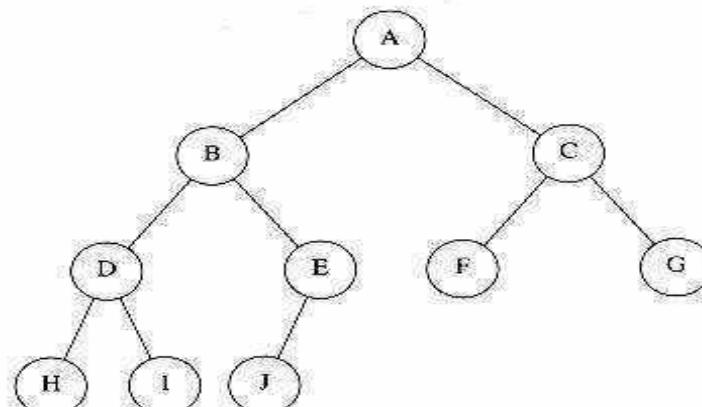
### Structure Property :

#### COMPLETE TREE

A binary tree is completely full if it is of height,  $h$ , and has  $2^h + 1 - 1$  nodes.

- it is empty or
- its left subtree is complete of height  $h-1$  and its right subtree is completely full of height  $h-2$  or

- its left subtree is completely full of height  $h-1$  and its right subtree is complete of height  $h-1$ .



	A	B	C	D	E	F	G	H	I	J		
0	1	2	3	4	5	6	7	8	9	10	11	12

A complete tree is filled from the left:

- all the leaves are on
  - the same level *or*
  - two adjacent ones *and*
- all nodes at the lowest level are as far to the left as possible.

## PROCEDURE

### INSERTION:

Let us consider the element X is to be inserted.

- First the element X is added as the last node.
- It is verified with its parent and adjacent node for its heap property.
- The verification process is carried upwards until the heap property is satisfied.
- If any verification is not satisfied then swapping takes place.
- Then finally we have the heap.

### DELETION:

- The deletion takes place by removing the root node.
- The root node is then replaced by the last leaf node in the tree to obtain the complete binary tree.
- It is verified with its children and adjacent node for its heap property.
- The verification process is carried downwards until the heap property is satisfied.
- If any verification is not satisfied then swapping takes place.
- Then finally we have the heap.

## PRIORITY QUEUE

It is a data structure which determines the priority of jobs.

The Minimum the value of Priority, Higher is the priority of the job.

The best way to implement Priority Queue is **Binary Heap**.

A Priority Queue is a special kind of queue datastructure. It has zero or more collection of elements, each element has a priority value.

- Priority queues are often used in resource management, simulations, and in the implementation of some algorithms (e.g., some graph algorithms, some backtracking algorithms).
- Several data structures can be used to implement priority queues.  
Below is a comparison of some:

### Basic Model of a Priority Queue



### Implementation of Priority Queue

1. Linked List.
2. Binary Search Tree.
3. Binary Heap.

**Linked List :**

A simple linked list implementation of priority queue requires  $O(1)$  time to perform the insertion at the front and  $O(n)$  to delete at minimum element.

**Binary Search tree :**

This gives an average running time of  $O(\log n)$  for both insertion and deletion.(deletemin).

**The efficient way of implementing priority queue is Binary Heap (or) Heap.**

Heap has two properties :

1. Structure Property.
2. Heap Order Preoperty.

**1.Structure Property :**

The Heap should be a complete binary tree, which is a completely filled tree, which is a completely filled binary tree with the possible exception of the bottom level, which is filled from left to right.

A Complete Binary tree of height  $H$ , has between  $2^h$  and  $(2^{h+1} - 1)$  nodes.

**Sentinel Value :**

The zeroth element is called the sentinel value. It is not a node of the tree. This value is required because while addition of new node, certain operations are performed in a loop and to terminate the loop, sentinel value is used.

Index 0 is the sentinel value. It stores irrelrated value, inorder to terminate the program in case of complex codings.

Structure Property : Always index 1 should be starting position.

**2.Heap Order Property :**

The property that allows operations to be performed quickly is a heap order property.

**Mintree:**

Parent should have lesser value than children.

**Maxtree:**

Parent should have greater value than children.

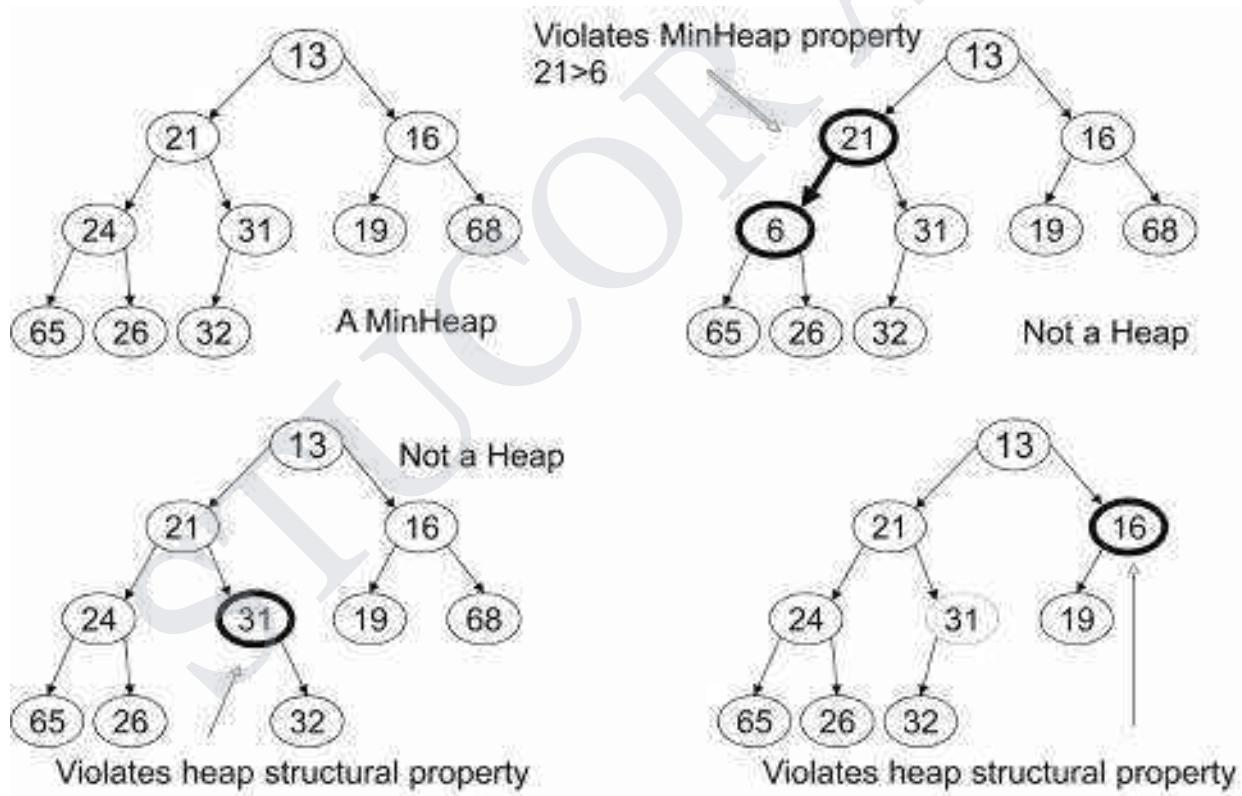
These two properties are known as heap properties

- Max-heap
- Min-heap

**Min-heap:**

The smallest element is always in the root node. Each node must have a key that is less or equal to the key of each of its children.

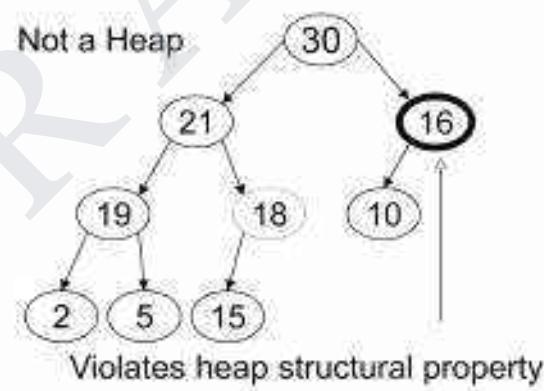
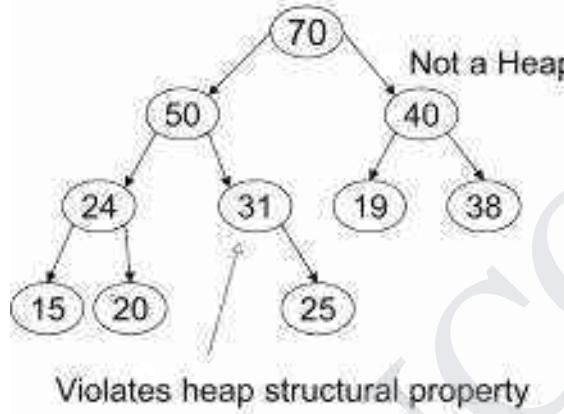
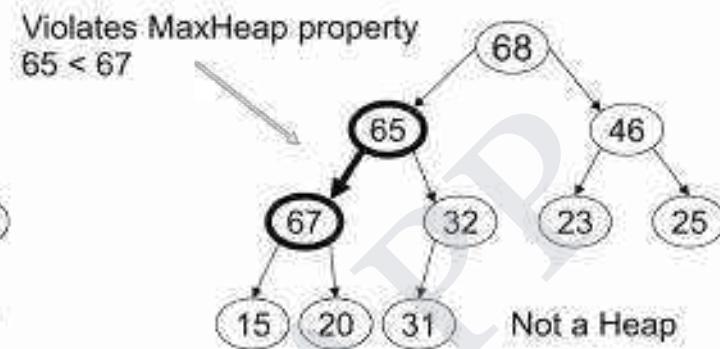
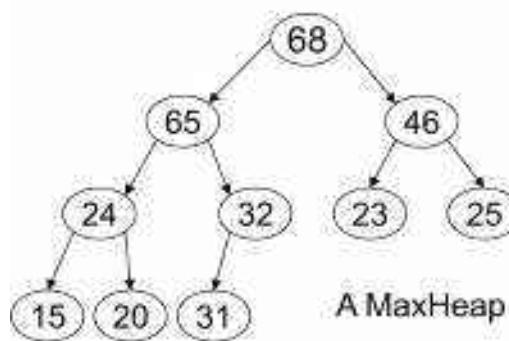
Examples

**Max-Heap:**

The largest Element is always in the root node.

Each node must have a key that is greater or equal to the key of each of its children.

Examples



## HEAP OPERATIONS:

There are 2 operations of heap

- Insertion
- Deletion

### 2.12.1 Insert:

Adding a new key to the heap

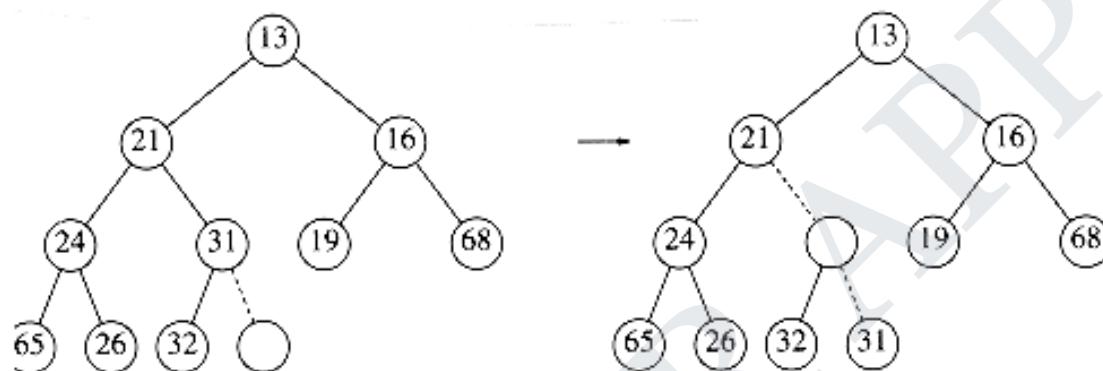
**Rules for the insertion:**

To insert an element X, into the heap, do the following:

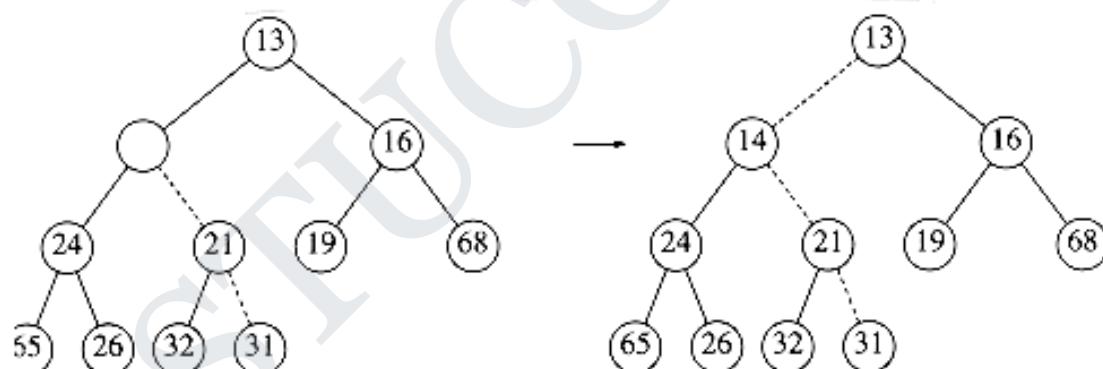
Step1: Create a hole in the next available location , since otherwise the tree will not be complete.

Step2: If X can be placed in the hole, without violating heap order, then do insertion, otherwise slide the element that is in the hole's parent node, into the hole, thus, bubbling the hole up towards the root.

Step3: Continue this process until X can be placed in the hole.



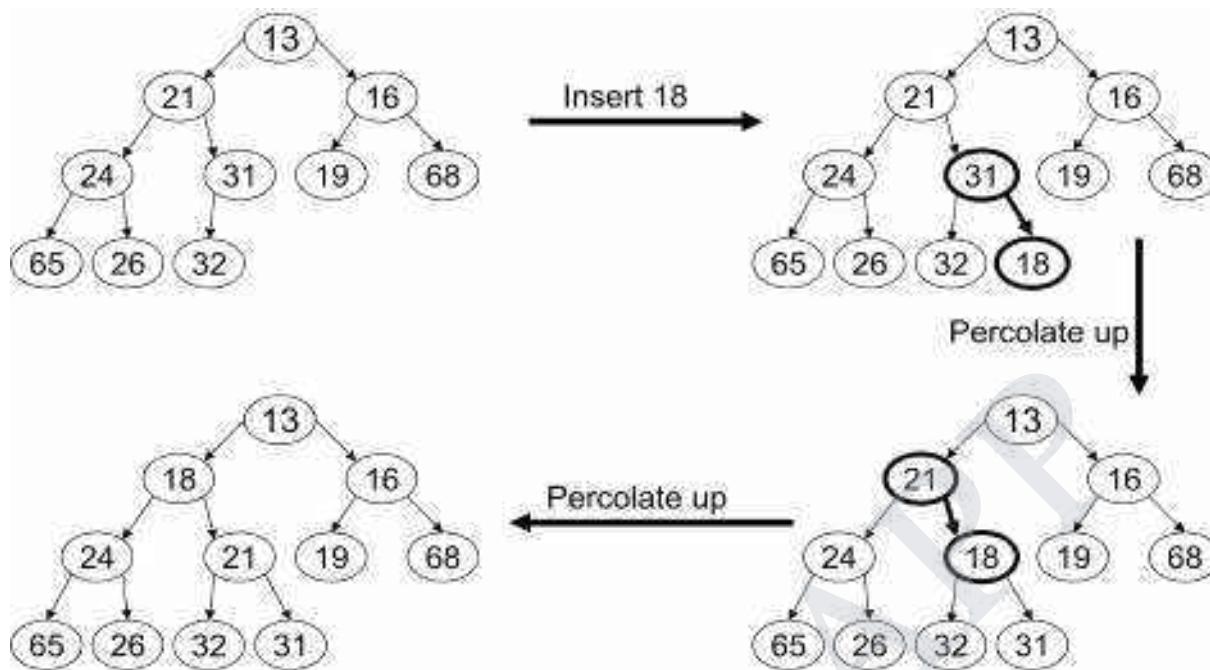
**Attempt to insert 14: creating the hole, and bubbling the hole up**



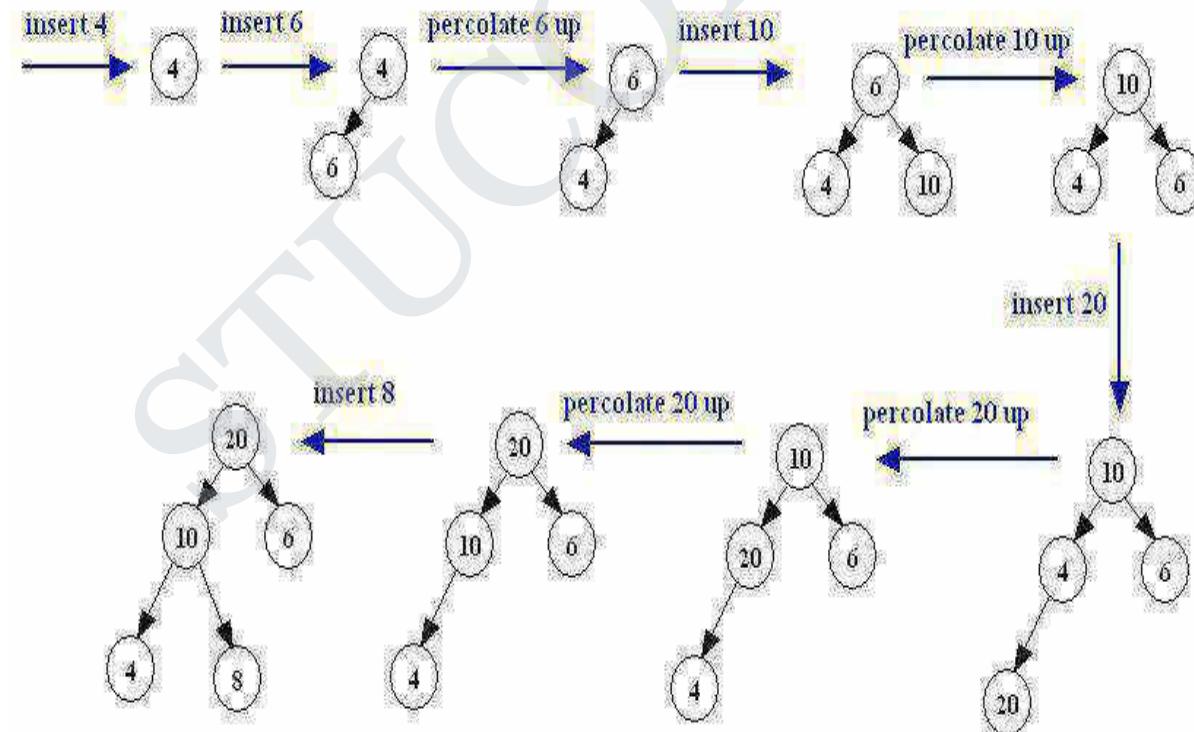
**The remaining two steps to insert 14 in previous heap**

### Example Problem :

1.Insert- 18 in a Min Heap



2. Insert the keys 4, 6, 10, 20, and 8 in this order in an originally empty max-heap



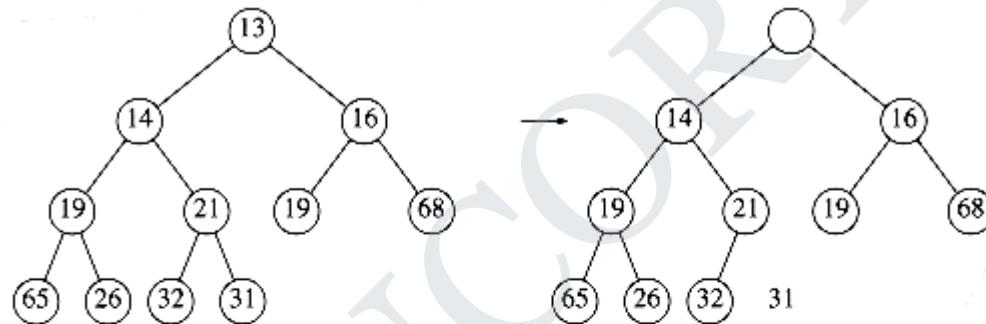
### 2.12.2 Delete-max or Delete-min:

Removing the root node of a max- or min-heap, respectively

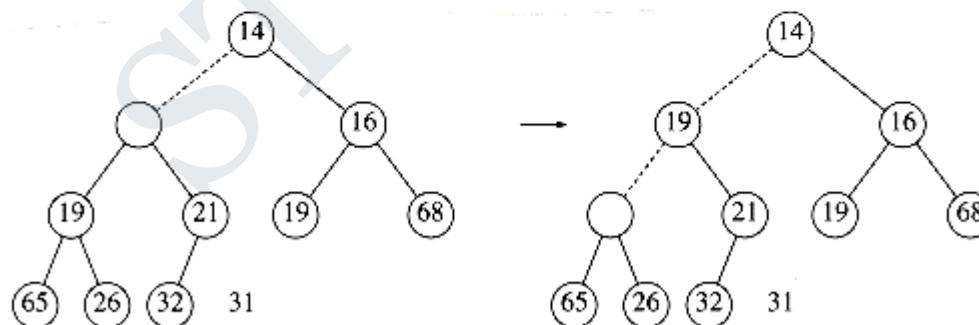
### Procedure for DeleteMin :

- \* DeleteMin operation is deleting the minimum element from the loop.
- \* In Binary heap | min heap the minimum element is found in the root.
- \* When this minimum element is removed, a hole is created at the root.
- \* Since the heap becomes one smaller, make the last element X in the heap to move somewhere in the heap.
- \* If X can be placed in the hole, without violating heap order property, place it , otherwise slide the smaller of the hole's children into the hole, thus , pushing the hole down one level.
- \* Repeat this process until X can be placed in the hole.

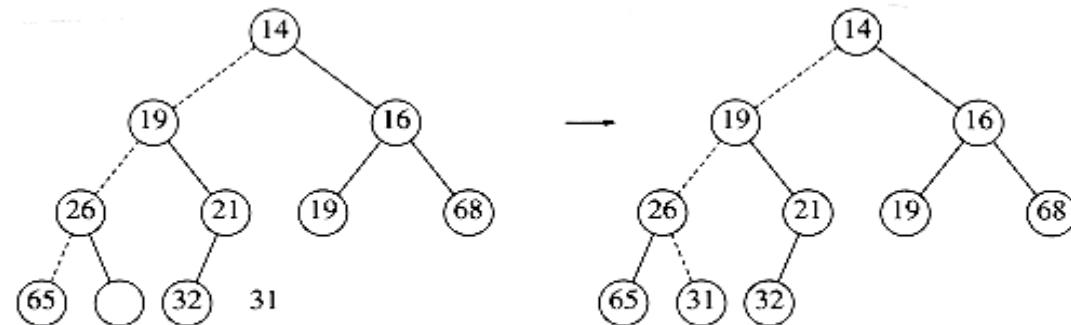
This general strategy is known as Percolate Down.



Creation of the hole at the root



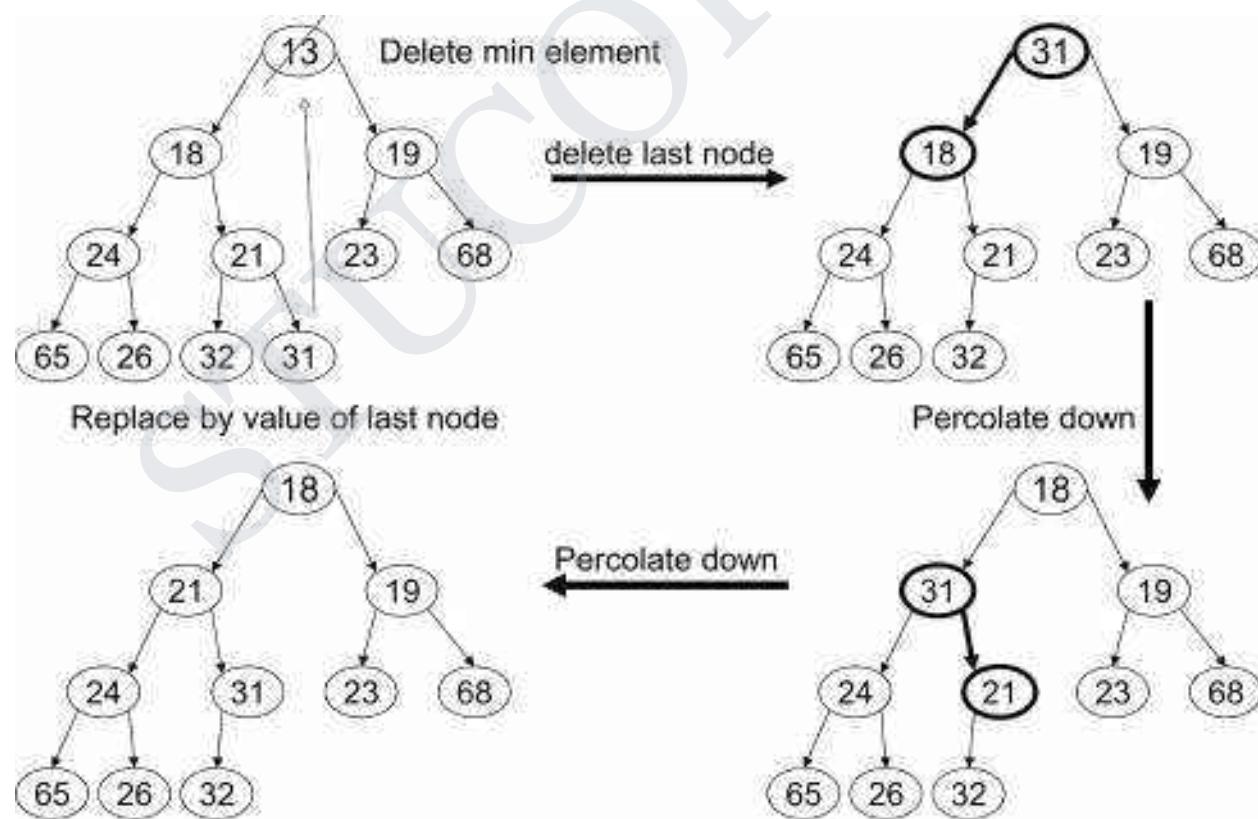
Next two steps in delete\_min

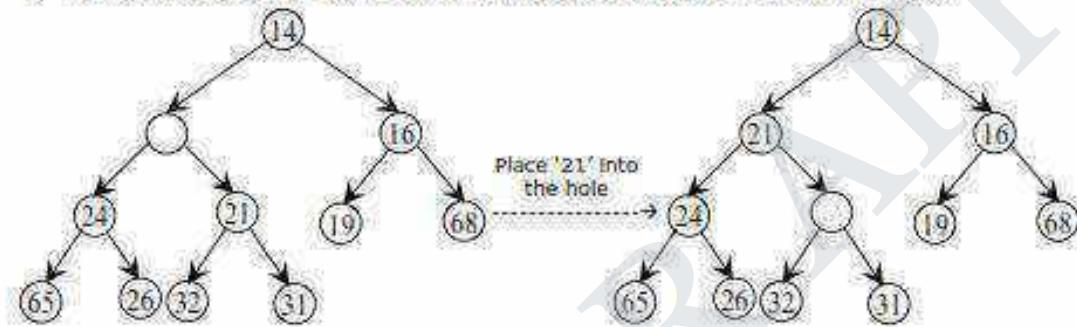
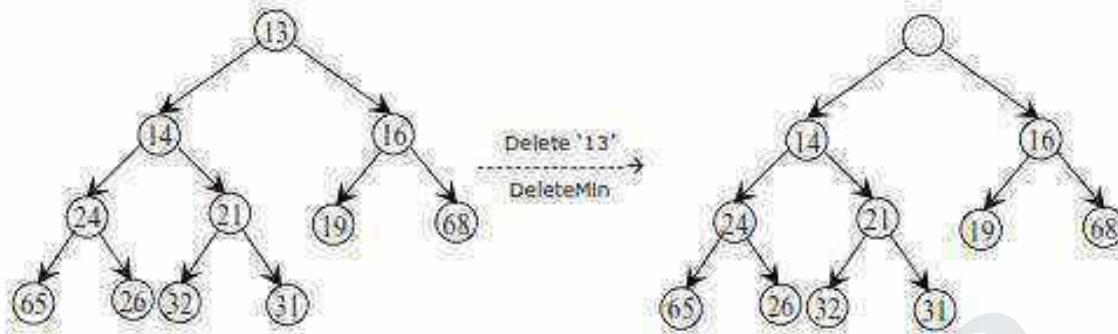


Last two steps in delete\_min

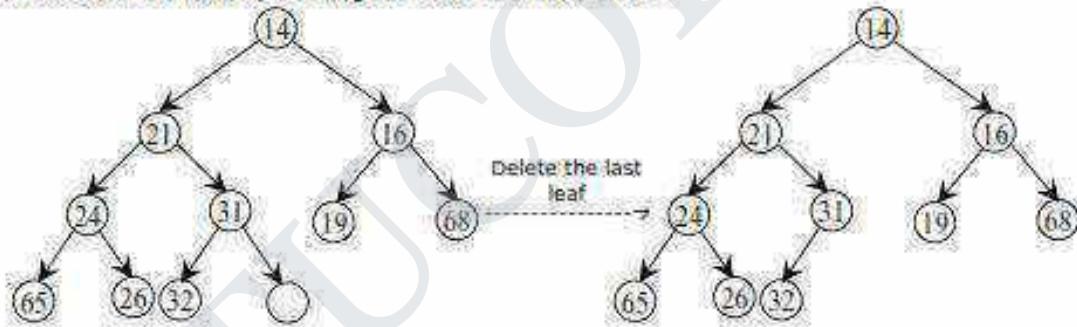
EXAMPLE PROBLEMS :

### 1.DELETE MIN



**2. Delete Min -- 13**

Now slide the hole down by inserting the value '31' in to the hole.

**BINARY HEAP ROUTINES [Priority Queue]**

```
Typedef struct heapstruct *priorityqueue;
```

```
Typedef int elementype;
```

```
Struct heapstruct
```

```
{
```

```
int capacity;
```

```
int size;
```

```
elementtype *element;  
};
```

### **Declaration of Priority Queue**

```
Priorityqueue initialize(int maxelement)  
{  
Priorityqueue H;  
If(minpsize<maxelements)  
Error("Priority queue size is too small");  
H=malloc(sizeof(struct heapstruct));  
If(H=NULL)  
Fatalerror("Out of space");  
/* Allocate the array plus one extra for sentinel */  
H-->elements=malloc((maxelements+1)*sizeof(elementtype));  
If(H-->elements==NULL)  
Fatalerror("out of space");  
H-->capacity=maxelements;  
H-->size=0;  
H-->elements[0]=mindata;  
Return H;  
}  
/* H-->elements[0]=sentinelvalue */
```

### **Insert Routine**

```
Void insert(elementtype X, priorityqueue H)  
{  
int i;  
if(isfull(H))  
{  
Error("Priority queue is full");  
Return;  
}
```

```
For(i=++H-->size;H-->element[i/2]>X;i=i/2)
H-->element[i]=H-->element[i/2];
H-->element[i]=X;
}
```

## Delete Routine

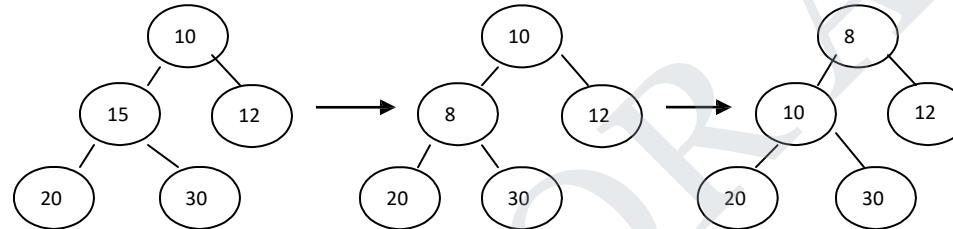
```
Elementtype deletemin(priorityqueue H)
{
int i,child;
elementtype minelement,lastelement;
if(isempty(H))
{
Error("Priority queue is empty");
Return H-->element[0];
}
Minelement=H-->element[1];
Lastelement=H-->element[H-->size--];
For(i=1;i*2<=H-->size;i=child)
{
/*Find smaller child */
Child=i*2;
If(child!=H-->size && H-->element[child++]<H-->element[child])
{
Child++;
}
/* Percolate one level */
If(lastelement>H-->element[child])
H-->element[i]=H-->element[child];
Else
Break;
}
H-->element[i]=lastelement;
```

```
    Return minelement;  
}
```

## Other Heap Operations

1. Decrease Key.
2. Increase Key.
3. Delete.
4. Build Heap.

### 1. Decrease Key :

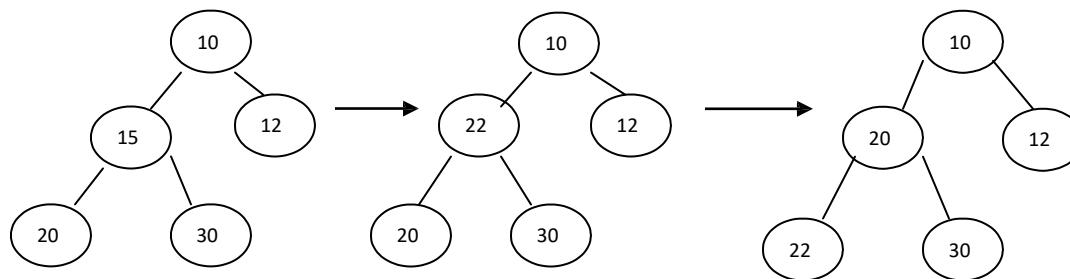


The Decrease key( $P, \Delta, H$ ) operation decreases the value of the key at position  $P$ , by a positive amount  $\Delta$ . This may violate the heap order property, which can be fixed by percolate up Ex : decreasekey(2,7,H)

### 2. Increase Key :

The Increase Key( $P, \Delta, H$ ) operation increases the value of the key at position  $P$ , by a positive amount  $\Delta$ . This may violate heap order property, which can be fixed by percolate down.

Ex : increase key(2,7,H)



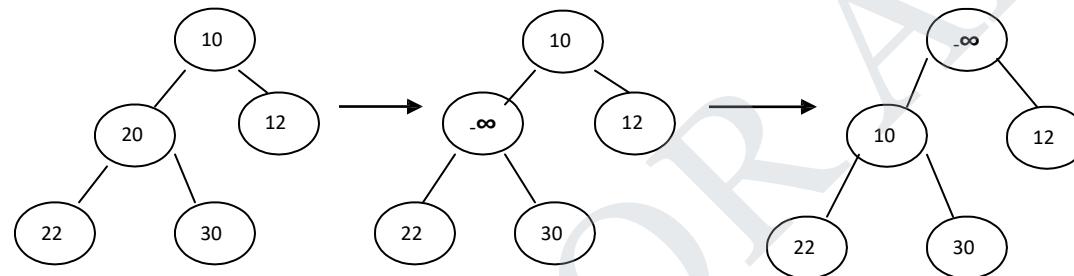
### 3. Delete :

The delete( $P, H$ ) operation removes the node at the position  $P$ , from the heap  $H$ . This can be done by,

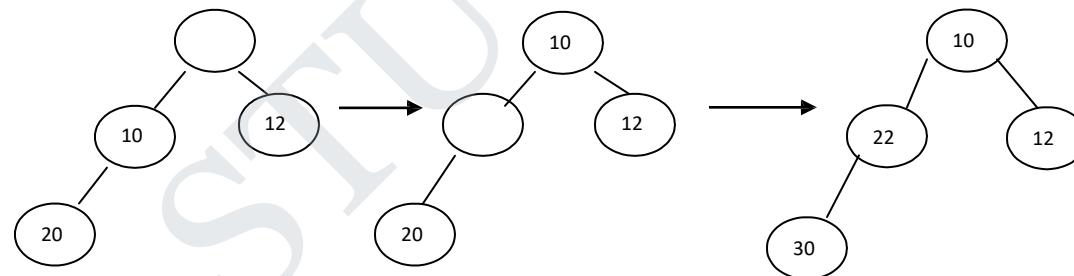
Step 1: Perform the decrease key operation, decrease key( $P, \infty, H$ ).

Step 2: Perform deletemin( $H$ ) operation.

Step 1: Decreasekey(2,  $\infty$ , H)



Step 2 : Deletemin(H)



## APPLICATIONS

The heap data structure has many applications

- Heap sort
- Selection algorithms

- Graph algorithms

Heap sort :

One of the best sorting methods being in-place and with no quadratic worst-case scenarios.

Selection algorithms:

Finding the min, max, both the min and max, median, or even the  $k$ -th largest element can be done in linear time using heaps.

Graph algorithms:

By using heaps as internal traversal data structures, run time will be reduced by an order of polynomial. Examples of such problems are Prim's minimal spanning tree algorithm and Dijkstra's shortest path problem.

## ADVANTAGE

The biggest advantage of heaps over trees in some applications is that construction of heaps can be done in linear time.

- It is used in
  - Heap sort
  - Selection algorithms
  - Graph algorithms

## DISADVANTAGE

Heap is expensive in terms of

- safety
- maintenance
- performance

Performance :

Allocating heap memory usually involves a long negotiation with the OS.

### Maintenance:

Dynamic allocation may fail; extra code to handle such exception is required.

### Safety :

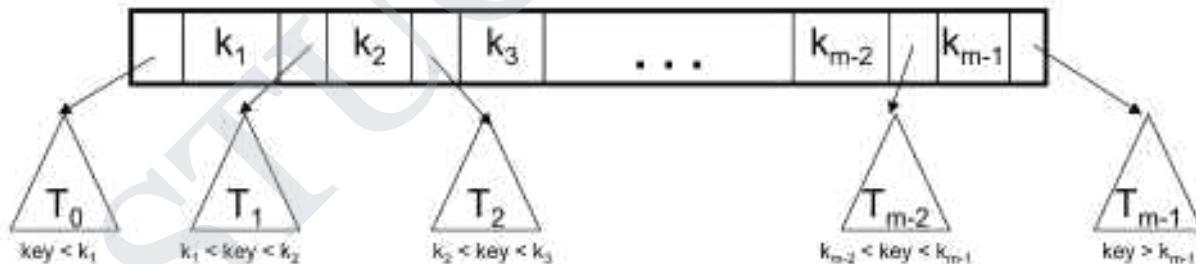
Object may be deleted more than once or not deleted at all .

## B-TREES

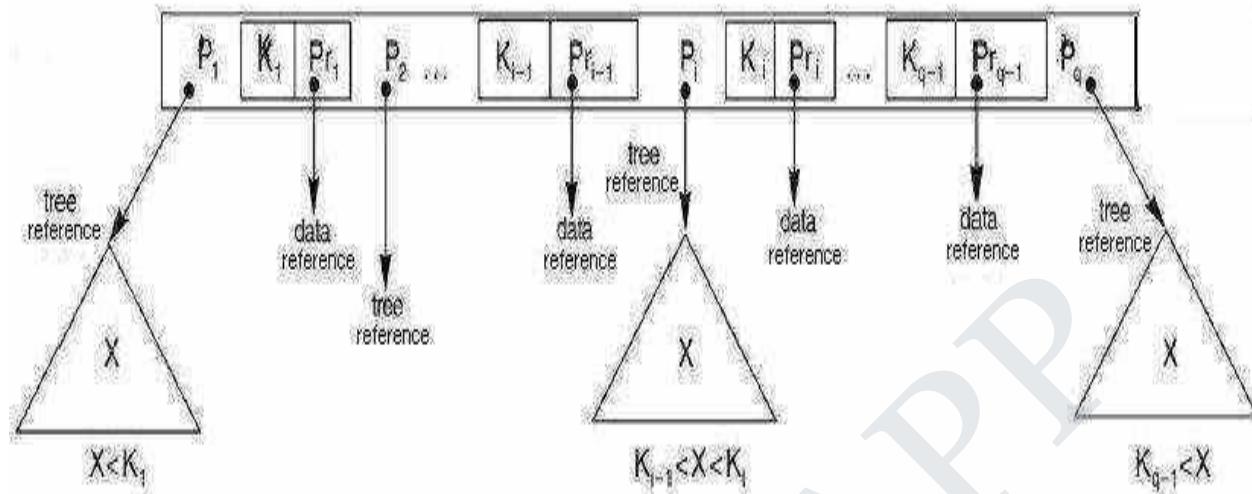
### Multi-way Tree

A multi-way (or m-way) search tree of order m is a tree in which

- Each node has at-most m subtrees, where the subtrees may be empty.
- Each node consists of at least 1 and at most m-1 distinct keys
- The keys in each node are sorted.



- The keys and subtrees of a non-leaf node are ordered as:
  - $T_0, k_1, T_1, k_2, T_2, \dots, k_{m-1}, T_{m-1}$  such that:
- All keys in subtree  $T_0$  are less than  $k_1$ .
- All keys in subtree  $T_i$ ,  $1 \leq i \leq m - 2$ , are greater than  $k_i$  but less than  $k_{i+1}$ .
- All keys in subtree  $T_{m-1}$  are greater than  $k_{m-1}$

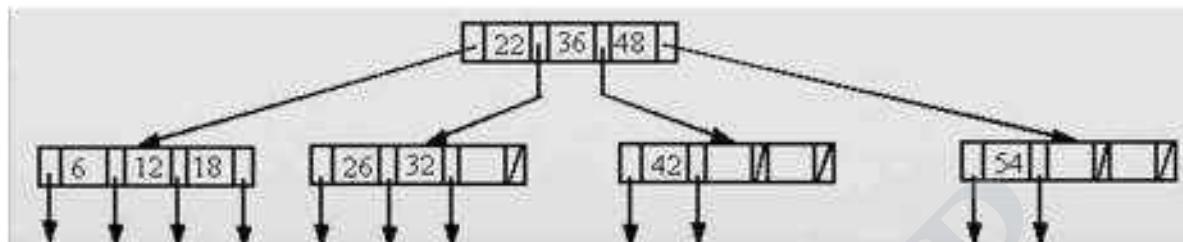


A **B-tree** of order  $m$  (or branching factor  $m$ ), where  $m > 2$ , is either an empty tree or a multiway search tree with the following properties:

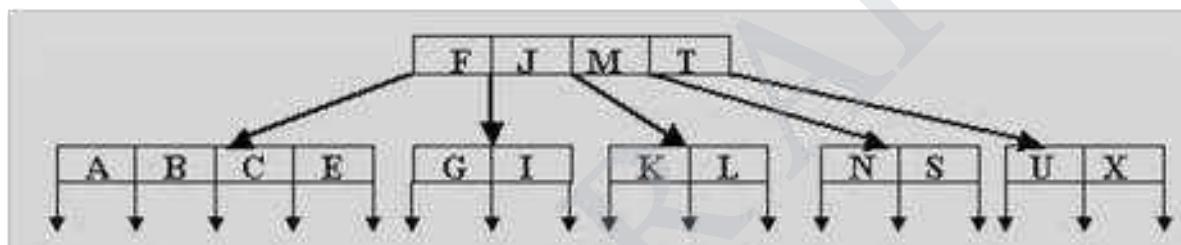
- The root is either a leaf or it has at least two non-empty subtrees and at most  $m$  non-empty subtrees.
- Each non-leaf node, other than the root, has at least  $\lceil m/2 \rceil$  non-empty subtrees and at most  $m$  non-empty subtrees. (Note:  $\lceil x \rceil$  is the lowest integer  $> x$  ).
- The number of keys in each non-leaf node is one less than the number of non-empty subtrees for that node.
- All leaf nodes are at the same level; that is the tree is perfectly balanced.

## B-Tree Examples

### Example: A B-tree of order 4



### Example: A B-tree of order 5



## Insertion in B-Trees

### OVERFLOW CONDITION:

A root-node or a non-root node of a B-tree of order  $m$  overflows if, after a key insertion, it contains  $m$  keys.

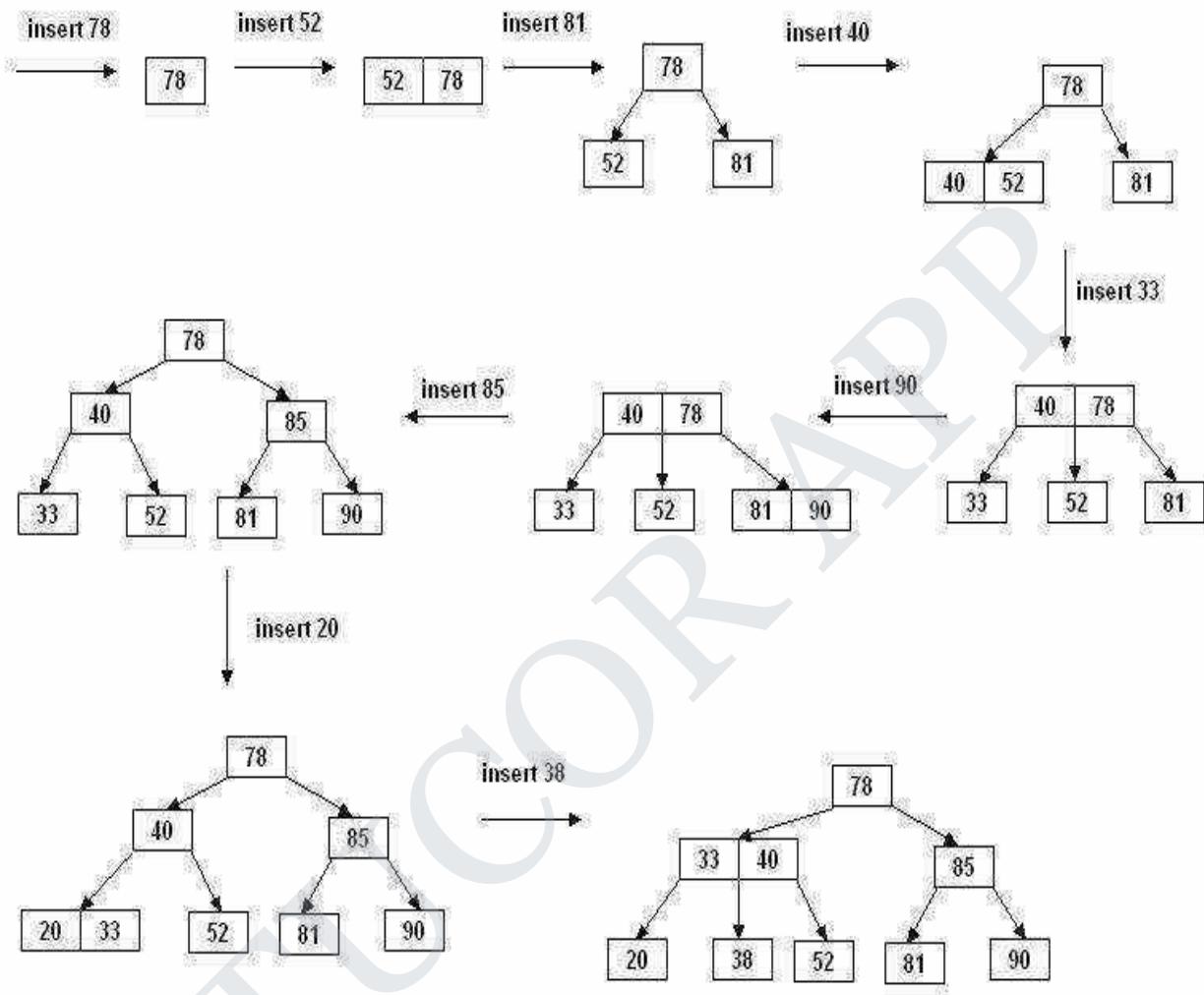
### Insertion algorithm:

If a node overflows, split it into two, propagate the "middle" key to the parent of the node. If the parent overflows the process propagates upward. If the node has no parent, create a new root node.

- Note: Insertion of a key always starts at a leaf node.

### Insertion in a B-tree of odd order

Example: Insert the keys 78, 52, 81, 40, 33, 90, 85, 20, and 38 in this order in an initially empty B-tree of order 3

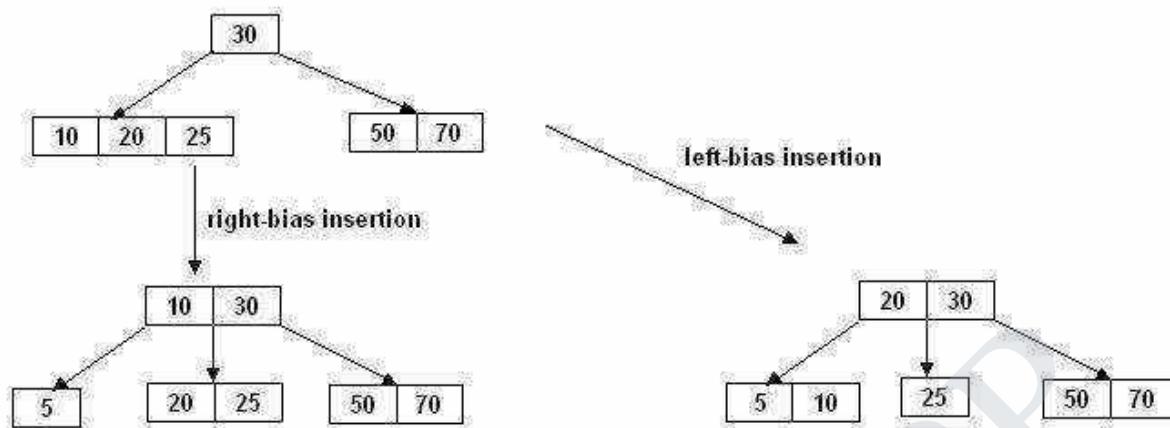


### Insertion in a B-tree of even order

At each node the insertion can be done in two different ways:

- **right-bias:** The node is split such that its right subtree has more keys than the left subtree.
- **left-bias:** The node is split such that its left subtree has more keys than the right subtree.

Example: Insert the key 5 in the following B-tree of order 4:



### ADVANTAGES:

- B-trees are suitable for representing huge tables residing in secondary memory because:
  1. With a large branching factor  $m$ , the height of a B-tree is low resulting in fewer disk accesses.
  2. The branching factor can be chosen such that a node corresponds to a block of secondary memory.
  3. The most common data structure used for database indices is the B-tree. An **index** is any data structure that takes as input a property (e.g. a value for a specific field), called the search key, and **quickly** finds all records with that property.

**Note: As  $m$  increases the amount of computation at each node increases; however this cost is negligible compared to hard-drive accesses.**

**Unit IV: Graphs: DEFINITION-REPRESENTATION OF GRAPH-TYPES OF GRAPH-BREADTH FIRST TRACERSAL-EPTH FIRST TRAVERSAL -TOPOLOGICAL SORT-BI-CONNECTIVITY-CUT VERTEX-EUCLER CIRCUITS-APPLICATIONS OF HEAP.**

**Graph:** - A graph is data structure that consists of following two components.

- A finite set of vertices also called as nodes.
- A finite set of ordered pair of the form  $(u, v)$  called as edge.

(or)

A graph  $G=(V, E)$  is a collection of two sets  $V$  and  $E$ , where

$V \rightarrow$  Finite number of vertices

$E \rightarrow$  Finite number of Edges,

Edge is a pair  $(v, w)$ , where  $v, w \in V$ .

**Application of graphs:**

- Coloring of MAPS
- Representing network
  - o Paths in a city
  - o Telephone network o
  - Electrical circuits etc.
- It is also using in social network
  - including o LinkedIn
  - o Facebook

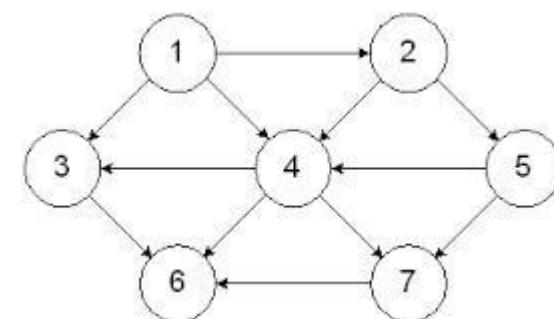
**Types of Graphs:**

- Directed graph
- Undirected Graph

**Directed Graph:**

In representing of graph there is a directions are shown on the edges then that graph is called Directed graph.  
That is,

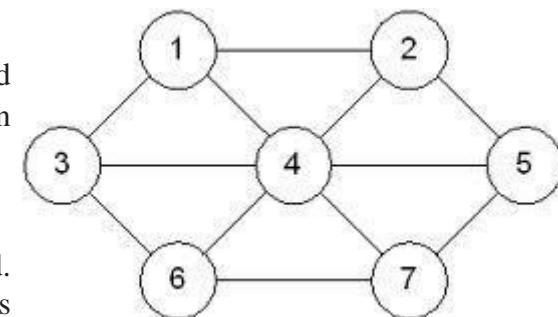
A graph  $G=(V, E)$  is a directed graph ,Edge is a pair  $(v, w)$ , where  $v, w \in V$ , and the pair is ordered.  
Means vertex ‘w’ is adjacent to v.  
Directed graph is also called digraph.



**Undirected Graph:**

In graph vertices are not ordered is called undirected graph. Means in which (graph) there is no direction (arrow head) on any line (edge).

A graph  $G=(V, E)$  is a directed graph ,Edge is a pair  $(v, w)$ , where  $v, w \in V$ , and the pair is not ordered.  
Means vertex ‘w’ is adjacent to ‘v’, and vertex ‘v’ is adjacent to ‘w’

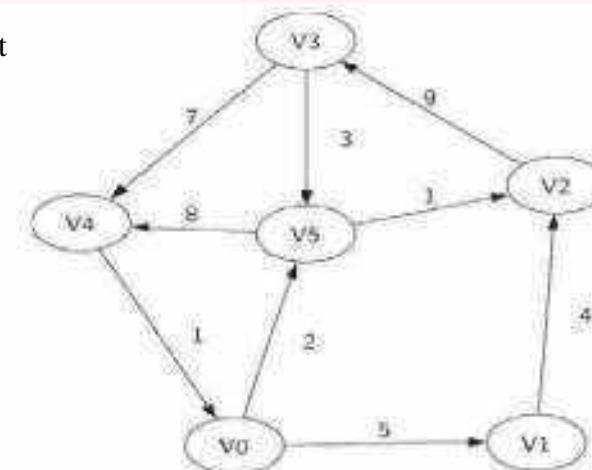


Note: in graph there is another component called weight/ cost.

**Weight graph:**

Edge may be weight to show that there is a cost to go from one vertex to another.

**Example:** In graph of roads (edges) that connect one city to another (vertices), the weight on the edge might represent the distance between the two cities (vertices).



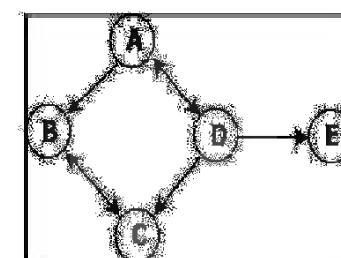
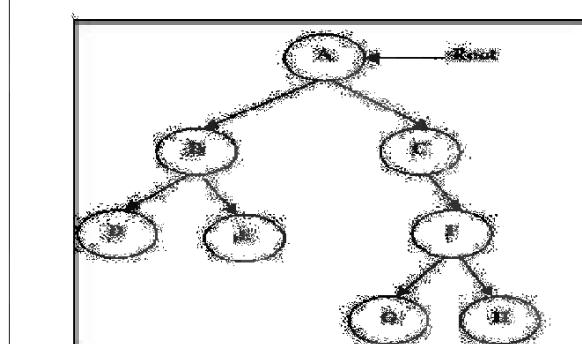
$$V = \{V_0, V_1, V_2, V_3, V_4, V_5\}$$

$$E = \left\{ \begin{array}{l} (v_0, v_1, 5), (v_1, v_2, 4), (v_2, v_3, 9), (v_3, v_4, 7), (v_4, v_0, 1), \\ (v_0, v_5, 2), (v_5, v_4, 8), (v_3, v_5, 3), (v_5, v_2, 1) \end{array} \right\}$$

## Difference between Trees and Graphs

	Trees	Graphs
<b>Path</b>	Tree is special form of graph i.e. <b>minimally connected graph</b> and having only one path between any two vertices.	In graph there can be more than one path i.e. graph can have uni-directional or bi-directional paths (edges) between nodes
<b>Loops</b>	Tree is a special case of graph having <b>no loops</b> , <b>no circuits</b> and no self-loops.	Graph can have loops, circuits as well as can have <b>self-loops</b> .
<b>Root Node</b>	In tree there is exactly one root node and every <b>child</b> have only one <b>parent</b> .	In graph there is no such concept of <b>root</b> node.
<b>Parent Child relationship</b>	In trees, there is parent child relationship so flow can be there with direction top to bottom or vice versa.	In Graph there is no such parent child relationship.
<b>Complexity</b>	Trees are less complex than graphs as having no cycles, no self-loops and still connected.	Graphs are more complex in compare to trees as it can have cycles, loops etc
<b>Types of Traversal</b>	Tree traversal is a kind of special case of traversal of graph. Tree is traversed in <b>Pre-Order, In-Order and Post-Order</b> (all three in DFS or in BFS algorithm)	Graph is traversed by <b>DFS:</b> Depth First Search <b>BFS :</b> Breadth First Search algorithm
<b>Connection Rules</b>	In trees, there are many rules / restrictions for making connections between nodes through edges.	In graphs no such rules/ restrictions are there for connecting the nodes through edges.
<b>DAG</b>	Trees come in the category of <b>DAG : Directed Acyclic Graphs</b> is a kind of directed graph that have no cycles.	Graph can be <b>Cyclic or Acyclic</b> .
<b>Different Types</b>	Different types of trees are : <b>Binary Tree , Binary Search Tree, AVL tree, Heaps.</b>	There are mainly two types of Graphs : <b>Directed and Undirected graphs.</b>
<b>Applications</b>	Tree applications: sorting and searching like Tree Traversal & Binary Search.	Graph applications : Coloring of maps, in OR ( <b>PERT &amp; CPM</b> ), algorithms, Graph coloring, job scheduling, etc.
<b>No. of edges</b>	Tree always has <b>n-1</b> edges.	In Graph, no. of edges depends on the graph.
<b>Model</b>	Tree is a <b>hierarchical model</b> .	Graph is a <b>network model</b> .

Figure



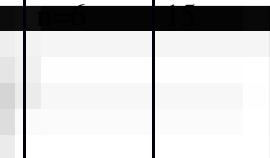
**Other types of graphs:**Complete Graph:

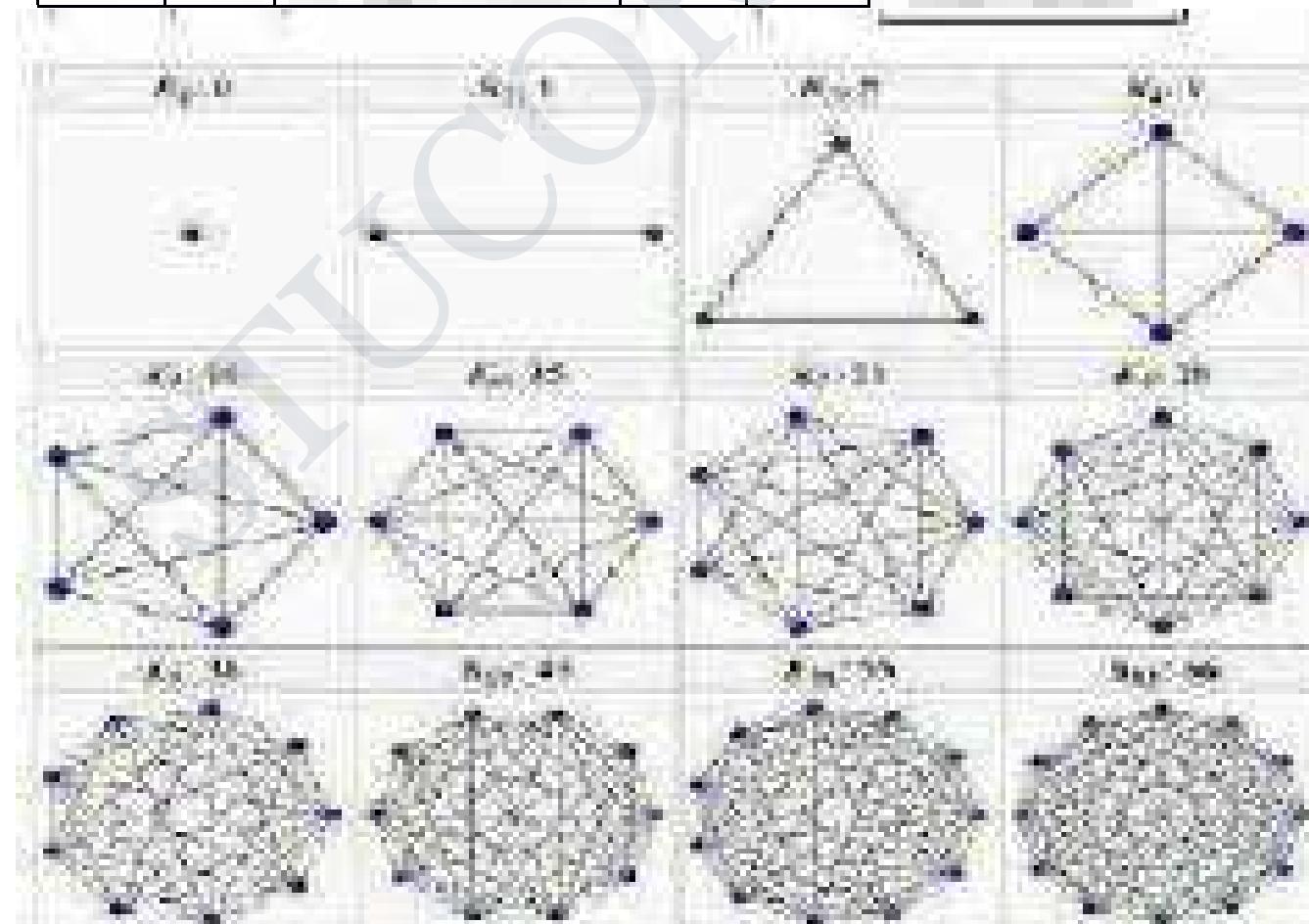
A **complete graph** is a simple undirected graph in which every pair of distinct vertices is connected by a unique edge.

OR

If an undirected graph of  $n$  vertices consists of  $n(n-1)/2$  number of edges then the graph is called complete graph.

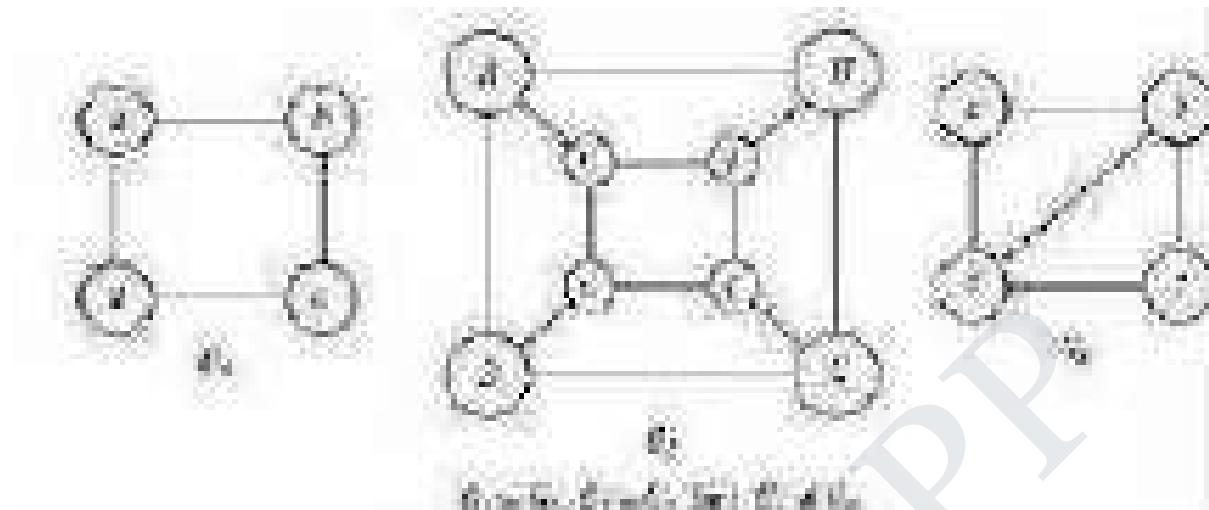
**Example:**

vertices	Edges	Complete graph	vertices	Edges	Complete graph
$n=2$	1		$n=3$	3	
$n=4$	6		$n=5$	10	

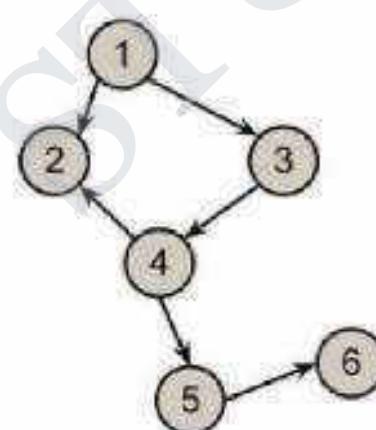
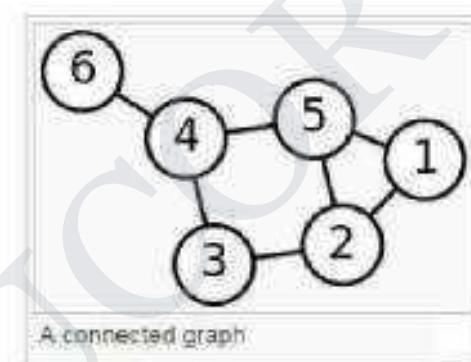


**Sub graph:**

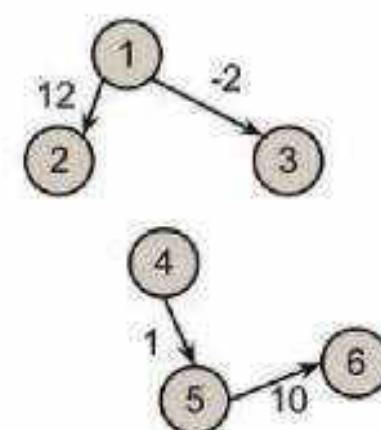
A sub-graph  $G'$  of graph G is a graph, such that the set of vertices and set of edges of  $G'$  are proper subset of the set of vertices and set of edges of graph G respectively.

**Connected Graph:**

A graph which is connected in the sense of a topological space (study of shapes), i.e., there is a path from any point to any other point in the graph. A graph that is not connected is said to be disconnected.



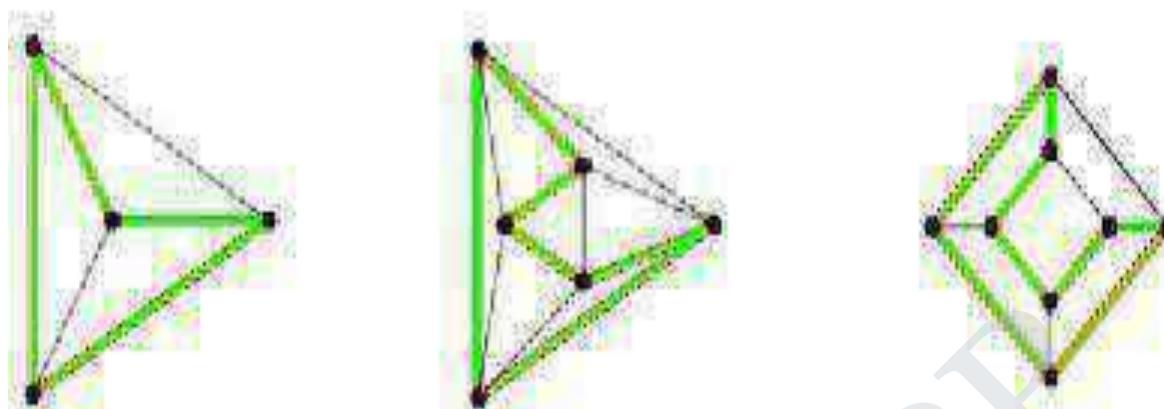
Connected



Disconnected

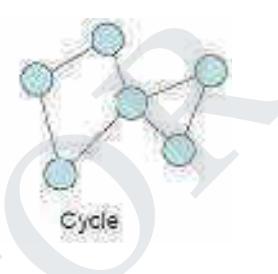
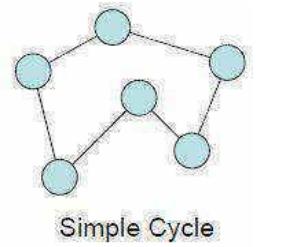
**Path:**

A path in a graph is a finite or infinite sequence of edges which connect a sequence of vertices. Means a path from one vertex to another vertex in a graph is represented by collection of all vertices (including source and destination) between those two vertices.



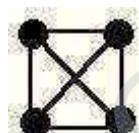
**Cycle:** A path that begins and ends at the same vertex.

**Simple Cycle:** a cycle that does not pass through other vertices more than once

**Degree:**

The degree of a graph vertex  $v$  of a graph  $G$  is the number of graph edges which touch  $v$ . The vertex degree is also called the local degree or valency. Or

The degree (or valence) of a vertex is the number of edge ends at that vertex.



For example, in this graph all of the vertices have degree three.

In a digraph (directed graph) the degree is usually divided into the **in-degree** and the **out-degree**

- | In-degree: The in-degree of a vertex  $v$  is the number of edges with  $v$  as their terminal vertex.
- | Out-degree: The out-degree of a vertex  $v$  is the number of edges with  $v$  as their initial vertex.

## TOPOLOGICAL SORT

A topological sort is a linear ordering of vertices in a **Directed Acyclic Graph** such that if there is a path from  $V_i$  to  $V_p$ , then  $V_j$  appears after  $V_i$  in the linear ordering. Topological sort is not possible if the graph has a cycle.

### INTRODUCTION

- In graph theory, a topological sort or topological ordering of a directed acyclic graph (DAG) is a linear ordering of its nodes in which each node comes before all nodes to which it has outbound edges.
- Every DAG has one or more topological sorts.
- More formally, define the partial order relation  $R$  over the nodes of the DAG such that  $xRy$  if and only if there is a directed path from  $x$  to  $y$ . Then, a topological sort is a linear extension of this partial order, that is, a total order compatible with the partial order.

### PROCEDURE

Step - 1 : Find the indegree for every vertex.

Step - 2 : Place the vertex whose indegree is 0, on the empty queue.

Step - 3 : Dequeue the vertex  $V$  and decrement the indegrees of all its adjacent vertices.

Step - 4 : Enqueue the vertex on the queue if its indegree falls to zero.

Step - 5 : Repeat from Step -3 until the queue becomes empty.

The topological ordering is the order in which the vertices dequeue.

Vertices	Indegree
1	0
2	0


## GRAPH TRAVERSAL

Graph traversal is the Visiting all the nodes of a graph.

The traversals are :

1. DFS (Depth First Search)
2. BFS (Breadth First Search)

## BREADTH FIRST SEARCH

- . The Breadth first search was one of the systematic approaches for exploring and searching the vertices/nodes in a given graph. The approach is called "breadth-first" because from each vertex 'v' that we visit, we search as broadly as possible by next visiting all the vertices adjacent to v.
- . It can also be used to find out whether a node is reachable from a given node or not.
- . It is applicable to both directed and undirected graphs.
- . Queue is used in the implementation of the breadth first search.

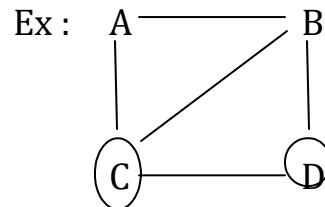
## ALGORITHM

```
Procedure bfs()
{
    //BFS uses Queue data structure
    Queue q=new LinkedList();
    q.add(this.rootNode);
    printNode(this.rootNode);
    rootNode.visited=true;
    while(!q.isEmpty())
    {
        Node n=(Node)q.remove(); Node child=null;
        while((child=getUnvisitedChildNode(n))!=null)
        {
            child.visited=true;
            printNode(child);
            q.add(child);
        }
    }
    //Clear visited property of nodes
    clearNodes();
}
```

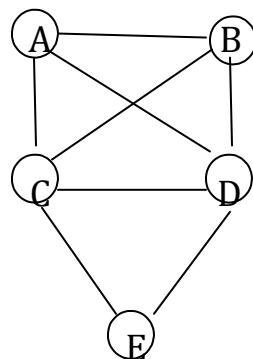
## Procedure

- Step -1 Select the start vertex/source vertex. Visit the vertex and mark it as one (1) (1 represents visited vertex).
- Step -2 Enqueue the vertex.
- Step -3 Dequeue the vertex.
- Step -4 Find the Adjacent vertices.
- Step -5 Visit the unvisited adjacent vertices and mark the distance as 1.
- Step -6 Enqueue the adjacent vertices.
- Step -7 Repeat from Step – 3 to Step – 5 until the queue becomes empty.





Vertices	Visited Vertices
A	1
B	0 1
C	0 1
D	0 1
Enqueue	A B C → D
Dequeue	A B C D



Vertices	Visited Vertices
C	1
A	0 1
B	0 1
D	0 1
E	0 1
Enqueue	C → A → B → D E
Dequeue	C A B D E

### Pseudo Code :

```
void BFS(Vertex S)
{
    Vertex v,w;
    Queue Q;
    visited[s] = 1; enqueue(S,Q);
    while(!IsEmpty(a))
    {
        v = dequeue(Q);
```

```
v = Dequeue(Q);  
print(v);  
for each adjacent vertices w to v  
if(visited[w]==0)  
{  
    visited[w] = 1;  
    Enqueue(w,Q);  
}  
}  
}
```

## APPLICATIONS

Breadth-first search can be used to solve many problems in graph theory, for example.

- Finding all nodes within one connected component
- Copying Collection, Cheney's algorithm
- Finding the shortest path between two nodes u and v (in an unweighted graph)
- Finding the shortest path between two nodes u and v (in a weighted graph: see talk page)
- Testing a graph for bipartiteness
- (Reverse) Cuthill–McKee mesh numbering
- Testing whether graph is connected.
- Computing a spanning forest of graph.
- Computing, for every vertex in graph, a path with the minimum number of edges between start vertex and current vertex or reporting that no such path exists.
- Computing a cycle in graph or reporting that no such cycle exists.

## Uses

- Testing whether graph is connected.
- Computing a spanning forest of graph.
- Computing, for every vertex in graph, a path with the minimum number of edges between start vertex and current vertex or reporting that no such path exists.
- Computing a cycle in graph or reporting that no such cycle exists.

### Depth first search

1. Backtracking is possible from a dead end
2. Vertices from which exploration is incomplete are processed in a LIFO order.
3. Search is done in one particular direction at the time.

### Breadth first search

1. Backtracking is not possible.
2. The vertices to be explored are organized as a FIFO queue.
3. Search is done parallelly in all possible direction.

## DEPTH FIRST SEARCH

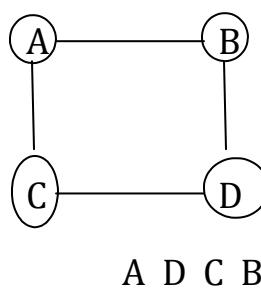
- Depth first search works by taking a node, checking its neighbors, expanding the first node it finds among the neighbors, checking if that expanded node is our destination, and if not, continue exploring more nodes.
- In depth-first search, edges are explored out of the most recently discovered vertex  $v$  that still has unexplored edges leaving it.
- When all of  $v$ 's edges have been explored, the search "backtracks" to explore edges leaving the vertex from which  $v$  was discovered.
- This process continues until we have discovered all the vertices that are reachable from the original source vertex.
- If any undiscovered vertices remain, then one of them is selected as a new source and the search is repeated from that source.

- This entire process is repeated until all vertices are discovered.
- Stack is used in the implementation of the depth first search.

In DFS, the basic data structures for storing the adjacent nodes is stack.

### Procedure:

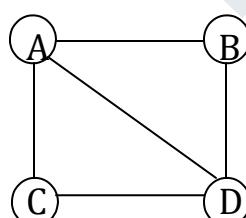
- Step -1 Select the start vertex.
- Step -2 Visit the vertex.
- Step -3 Push the vertex on to the stack.
- Step -4 Pop the vertex.
- Step -5 Find the adjacent vertices, and select any 1 of them.
- Step -6 Repeat from Step – 4 to Step – 5 until the stack becomes empty.



Vertices	Visited Vertices
A	1
B	0 1
C	0 1
D	0 1

Stack      O/P  
              A  
              D  
              C

B



Vertices	Visited Vertices
A	1
B	0 1
C	0 1
D	0 1

Stack o/p    A B D C

**Pseudo Code:**

```
void DFS(Vertex S)
{
    int top = 0;
    Stack C; vertex v,w;
    visited [S] = 1
    C[top] = S;
    while(!IsEmpty(C))
    {
        v = pop(C);
        print(V);
        for each unknown adjacent vertices of V,
        {
            if(visited[w] == 0)
            {
                visited[w] = 1;
                push(w,C);
            }
        }
    }
}
(OR)
```

```
void DFS(vertex V)
{
    visited[v] = 1;
    push(v);
    pop(v);
    for each w adjacent to V
        if(!visited[w])
            Dfs(w);
}
```

## ALGORITHM

```
Procedure dfs()
{
    //DFS uses Stack data structure
    Stack s=new Stack();
    s.push(this.rootNode);
    rootNode.visited=true;
    printNode(rootNode);
    while(!s.isEmpty())
    {
        Node n=(Node)s.peek();
        Node child=getUnvisitedChildNode(n);
        if(child!=null)
        {
            child.visited=true;
            printNode(child);
            s.push(child);
        }
        else
        {
            s.pop();
        }
    }
}
```

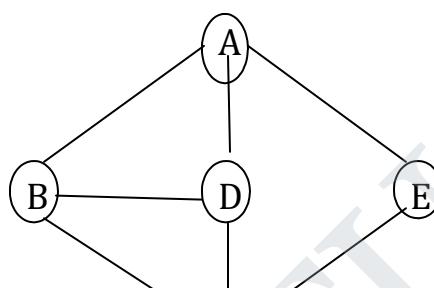
```
        }  
    }  
    //Clear visited property of nodes  
    clearNodes();  
}
```

### Applications of DFS :

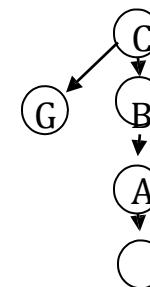
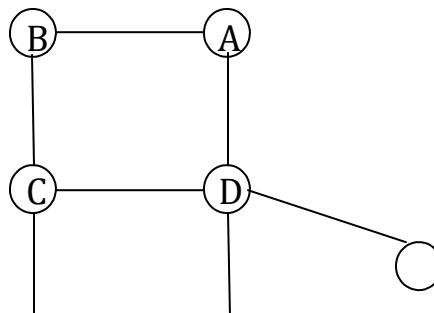
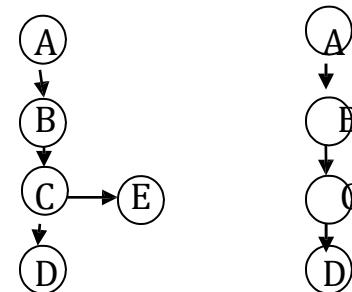
1. To check whether the undirected graph is connected or not.
2. To check if the connected undirected graph is bi – connected or not.
3. To check whether the directed graph is acyclic or not.

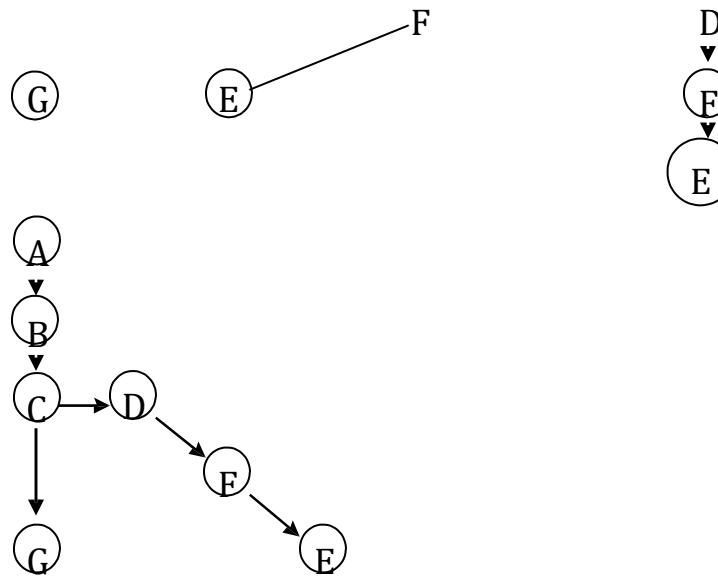
#### I. Undirected Graph :

An undirected graph is connected if and only if a depth first search starting from any node visits every node.



1. Tree Edge →  
2. Back Edge ----->





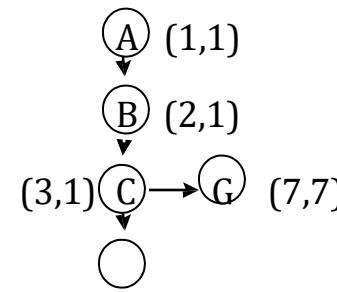
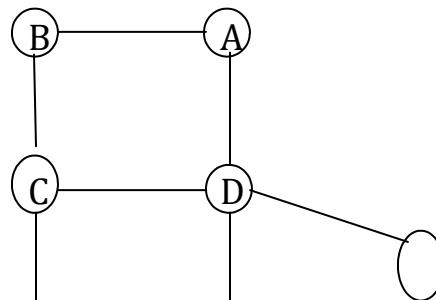
## II. Bi-Connectivity :

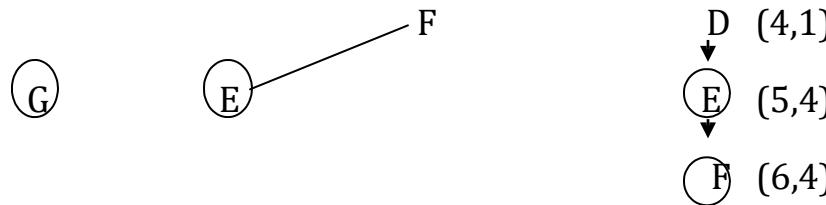
The vertices which are responsible for disconnection is called as Articulation points.

If in a connected undirected graph, the removal of any node does not affect the connectivities, then the graph is said to be biconnected graph.

1. Num       $\text{Low}(w) \geq \text{Num}(v)$
2. Low

Calculation of Num and Low gives the articulation point.





## BICONNECTIVITY:

A connected undirected graph is biconnected if there are no vertices whose removal disconnects the rest of the graph.

- A biconnected undirected graph is a connected graph that is not broken into disconnected pieces by deleting any single vertex (and its incident edges).
- A biconnected directed graph is one such that for any two vertices  $v$  and  $w$  there are two directed paths from  $v$  to  $w$  which have no vertices in common other than  $v$  and  $w$ .
- If a graph is not biconnected, the vertices whose removal would disconnect the graph are called articulation points.

## DEFINITION

Equivalent definitions of a biconnected graph  $G$ :

- Graph  $G$  has no separation edges and no separation vertices
- For any two vertices  $u$  and  $v$  of  $G$ , there are two disjoint simple paths between  $u$  and  $v$  (i.e., two simple paths between  $u$  and  $v$  that share no other vertices or edges)
- For any two vertices  $u$  and  $v$  of  $G$ , there is a simple cycle containing  $u$  and  $v$ .

## Biconnected Components

**Biconnected component of a graph  $G$  are:**

- A maximal biconnected subgraph of  $G$ , or
- A subgraph consisting of a separation edge of  $G$  and its end vertices

## Interaction of biconnected components

- An edge belongs to exactly one biconnected component
- A nonseparation vertex belongs to exactly one biconnected component
- A separation vertex belongs to two or more biconnected components

## Articulation Point :

The vertices whose removal disconnects the graph are known as Articulation Points.

Steps to find Articulation Points :

- (i) Perform DFS, starting at any vertex.
- (ii) Number the vertex as they are visited as  $\text{Num}(V)$ .
- (iii) Compute the lowest numbered vertex for every vertex  $V$  in the DFS tree, which we call as  $\text{low}(W)$ , that is reachable from  $V$  by taking one or more tree edges and then possible one back edge by definition.

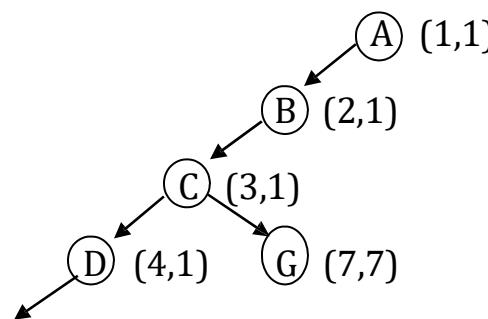
$$\text{Low}(V) = \min(\text{Num}(V), \text{Num}(W), \text{Low}(W))$$

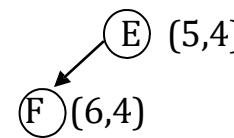
The Lowest  $\text{Num}(W)$  among all back edges  $V, W$ .

The Lowest  $\text{Low}(W)$  among all the tree edges  $V, W$ .

The root is an articulation if and only if (iff) it has more than two children.

Any vertex  $V$  other than the root is an Articulation point iff  $V$  has some child  $W$  such that  $\text{Low}(W) \geq \text{Num}(V)$





$$\text{Low}(F) = \min(\text{Num}(V), \text{Num}(W), \text{Low}(W))$$

$$= \min(6, 4, -1)$$

$$= 4$$

$$\text{Low}(E) = \min(5, 6, 4)$$

$$= 4$$

$$\text{Low}(D) = \min(4, 1, 4)$$

$$= 1$$

$$\text{Low}(G) = \min(7, -, -)$$

$$= 7$$

$$\text{Low}(C) = \min(3, (4,7), (1,7))$$

$$= 1$$

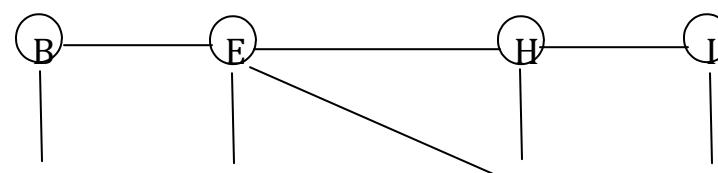
$$\text{Low}(B) = \min(2, 3, 1)$$

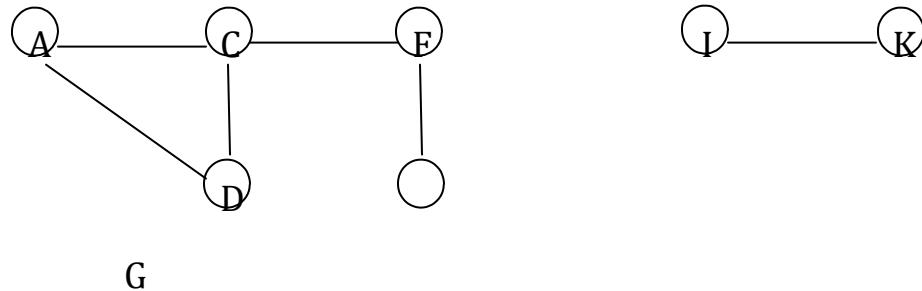
$$= 1$$

$$\text{Low}(A) = \min(1, 2, 1)$$

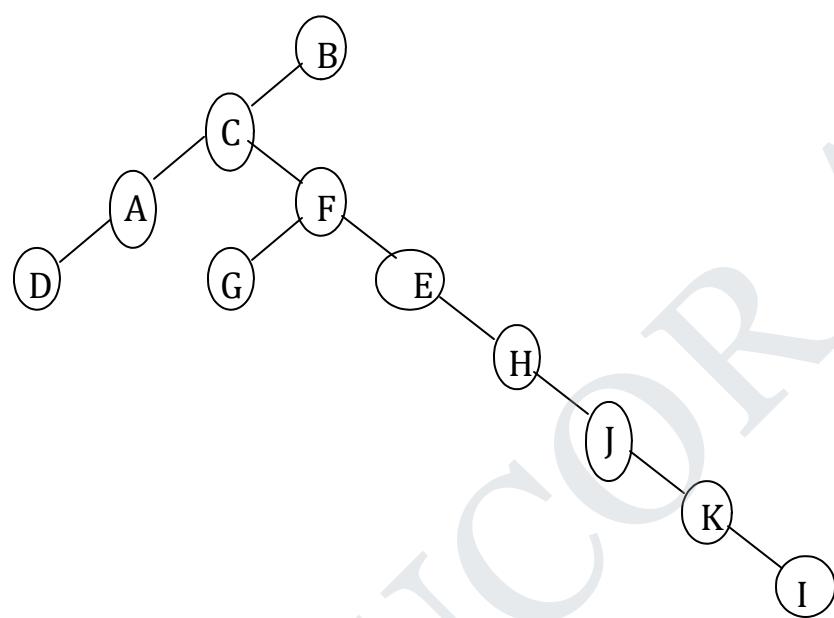
$$= 1$$

Example 2





G



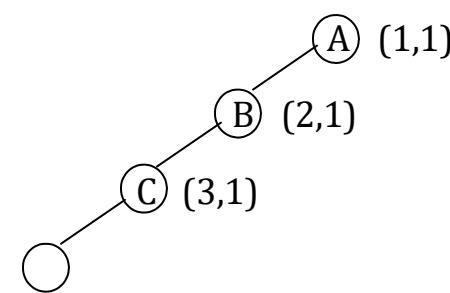
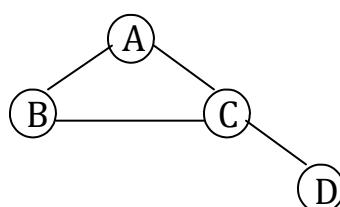
Visited	1	2	3	4
	A	B	C	D

Num[]

1	0	1	2	3
---	---	---	---	---

Parent[]

0	1	2	3
A	B	C	D



D (4,4)

**ALGORITHM****Routine to assign Num to Vertices**

```
void AssignNum(Vertex V)
{
    Vertex W;
    int counter = 0;
    Num[V] = ++counter;
    visited[V] = True;
    for each W adjacent to V
        if(!visited[W])
    {
        parent[W] = V;
        AssignNum(W);
    }
}
```

**Routine to compute low and to test for points**

```
void AssignLow(Vertex V)
{
    Vertex W;
    Low[V] = Num[V]; /* Rule 1 */
    for each W adjacent to V
    {
        if(Num[W] > Num[V]) /* forward edge or free edge */
        { AssignLow(W)
            if(low[W] >= Num[V])
                printf("%v Articulation point is", V);
            Low[V] = min(Low[V], Low[W]); /* Rule 3 */
        }
        else
        {
            if(parent[V] != W) /* Back edge */
                Low[V] = min(Low[V], Num[W]); /* Rule 2 */
        }
    }
}
```

```
}
```

```
}
```

```
}
```

## APPLICATION

- Bio-Connectivity is a application of depth first search.
- Used mainly in network concepts.

## BICONNECTIVITY ADVANTAGES

- Total time to perform traversal is *minimum*.
- Adjacency lists are used
- Traversal is given by  $O(E+V)$ .

## DISADVANTAGES

- Have to be careful to avoid cycles
- Vertices should be carefully removed as it affects the rest of the graph.

## EULER CIRCUIT

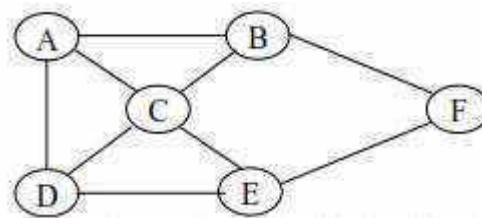
### EULERIAN PATH

An Eulerian path in an undirected graph is a path that uses each edge exactly once. If such a path exists, the graph is called traversable or semi-eulerian.

### EULERIAN CIRCUIT

An Eulerian circuit or Euler tour in an undirected graph is a cycle that uses each edge exactly once. If such a cycle exists, the graph is called

Unicursal. While such graphs are Eulerian graphs, not every Eulerian graph possesses an Eulerian cycle.



## EULER'S THEOREM

### Euler's theorem 1

- If a graph has any vertex of odd degree then it cannot have an Euler circuit.
- If a graph is connected and every vertex is of even degree, then it at least has one Euler circuit.

### Euler's theorem 2

- If a graph has more than two vertices of odd degree then it cannot have an Euler path.
- If a graph is connected and has just two vertices of odd degree, then it at least has one Euler path. Any such path must start at one of the odd-vertices and end at the other odd vertex.

## ALGORITHM

### Fleury's Algorithm for finding an Euler Circuit

1. Check to make sure that the graph is **connected and all vertices are of even degree**
2. Start at any vertex
3. Travel through an edge:
  - If it is **not a bridge for the untraveled part**, or
    - there is no other alternative
4. Label the edges in the order in which you travel them.
5. When you cannot travel any more, stop.

## Fleury's Algorithm

1. pick any vertex to start .
2. from that vertex pick an edge to traverse .
3. darken that edge, as a reminder that you can't traverse it again .
4. travel that edge, coming to the next vertex .
5. repeat 2-4 until all edges have been traversed, and you are back at the starting vertex .

At each stage of the algorithm:

- the original graph minus the darkened (already used) edges = **reduced graph**
- **important rule:** *never cross a bridge* of the reduced graph *unless there is no other choice*

Note:

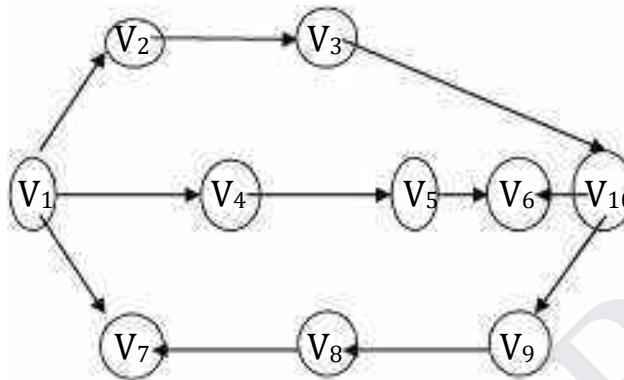
- the same algorithm works for *Euler paths*
- before starting, use Euler's theorems to check that the graph has an Euler path and/or circuit to find.

## APPLICATION

Eulerian paths are being used in bioinformatics to reconstruct the DNA SEQUENCE from its fragments.

## IMPORTANT SUMS

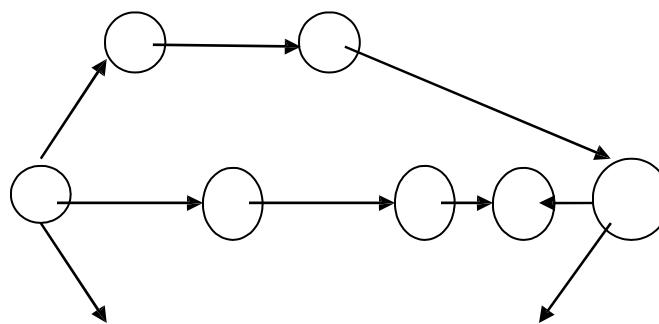
### 1. Topological Sort

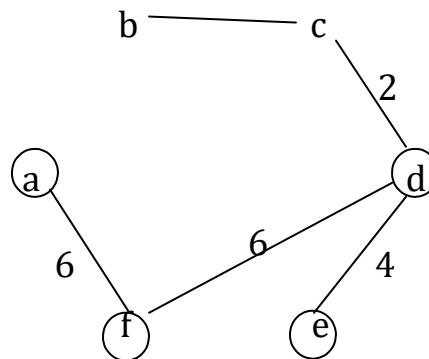


Sorted Vertices :

V<sub>1</sub> V<sub>2</sub> V<sub>4</sub> V<sub>3</sub> V<sub>5</sub> V<sub>10</sub> V<sub>6</sub> V<sub>9</sub> V<sub>8</sub> V<sub>7</sub>

Vertices	Indegree
V <sub>1</sub>	0
V <sub>2</sub>	1 0
V <sub>3</sub>	1 0 0
V <sub>4</sub>	1 0
V <sub>5</sub>	1 0
V <sub>6</sub>	2 1 0
V <sub>7</sub>	2 1 0
V <sub>8</sub>	1 0
V <sub>9</sub>	1 0
V <sub>10</sub>	1 0
Enqueue	V <sub>1</sub> V <sub>2</sub> V <sub>4</sub> V <sub>3</sub> V <sub>5</sub> V <sub>10</sub> V <sub>6</sub> V <sub>9</sub> V <sub>8</sub> V <sub>7</sub>
Dequeue	V <sub>1</sub> V <sub>2</sub> V <sub>4</sub> V <sub>3</sub> V <sub>5</sub> V <sub>10</sub> V <sub>6</sub> V <sub>9</sub> V <sub>8</sub> V <sub>7</sub>



**5. Write the pseudo code for Sorting algorithms****Pseudo Code for Topological Sort :**

```
void Topsort(Graph G)
{
    Queue Q;
    vertex v,w; int counter = 0;
    Q = Create Queue (Num Vertex);
    Make Empty(Q);
    for each vertex V
        if (Indegree[V]==0)
            enqueue(V,Q);
    while(!Isempty(Q))
    {
        V = Dequeue(Q);
        Topnum[V]=++Counter;
        for each w adjacent to V
```

```
if(--Indegree[w]==0)
Enqueue(w,0);
}
if(counter!=Num Vertex)
Error("Graph has a cycle");
Dispose Queue(Q);
}
```

**Pseudo Code to Perform Unweighted Sort :**

```
void unweighted(Table T)
{
Queue Q;
Vertex v,w;
Q = Create Queue(Num Vertex);
Make Empty(Q);
while(!IsEmpty(Q))
{
V = Dequeue(o);
T[V].Known = True;
for each w adjacent to v
if(!T[w].Known)
{
```

```
for each w adjacent to v
if(!T[w].Known)
{
if(!T[v].Dist + Cv,w < T[w].Dist)
{
/* update of w */
Decrease(T[w].Dist to T[v].Dist + Cv,w);
T[w].path = v;
}
}
}
Dispose Queue(Q);
}
```

**UNIT V SORTING, SEARCHING AND HASH TECHNIQUES**

Sorting algorithms: Insertion sort - Selection sort - Shell sort - Bubble sort - Quick sort - Merge sort - Radix sort – Searching: Linear search –Binary Search Hashing: Hash Functions – Separate Chaining – Open Addressing – Rehashing – Extendible Hashing.

**SORTING:****Definition:**

Sorting is a technique for arranging data in a particular order.

**Order of sorting:**

Order means the arrangement of data. The sorting order can be ascending or descending. The ascending order means arranging the data in increasing order and descending order means arranging the data in decreasing order.

**Types of Sorting**

- Internal Sorting
- External Sorting

**Internal Sorting**

Internal Sorting is a type of sorting technique in which data resides on main memory of computer. It is applicable when the number of elements in the list is small.

**E.g.** Bubble Sort, Insertion Sort, Shell Sort, Quick Sort., Selection sort, Radix sort

**External Sorting**

External Sorting is a type of sorting technique in which there is a huge amount of data and it resides on secondary devise(for eg hard disk,Magnetic tape and so no) while sorting.

**E.g.** Merge Sort, Multiway Merge Sort,Polyphase merge sort

**Sorting can be classified based on**

- 1.Computational complexity
- 2.Memory utilization
- 3.Stability
- 4.Number of comparisons.

**ANALYSIS OF ALGORITHMS:**

Efficiency of an algorithm can be measured in terms of:

**Space Complexity:** Refers to the space required to execute the algorithm

**Time Complexity:** Refers to the time required to run the program.

**Sorting algorithms:**

- Insertion sort
- Selection sort
- Shell sort
- Bubble sort
- Quick sort
- Merge sort
- Radix sort

**INSERTION SORTING:**

- The insertion sort works by taking elements from the list one by one and inserting them in the correct position into a new sorted list.
- Insertion sort consists of  $N-1$  passes, where  $N$  is the number of elements to be sorted.
- The  $i^{\text{th}}$  pass will insert the  $i^{\text{th}}$  element  $A[i]$  into its rightful place among  $A[1], A[2], \dots, A[i-1]$ .
- After doing this insertion the elements occupying  $A[1], \dots, A[i]$  are in sorted order.

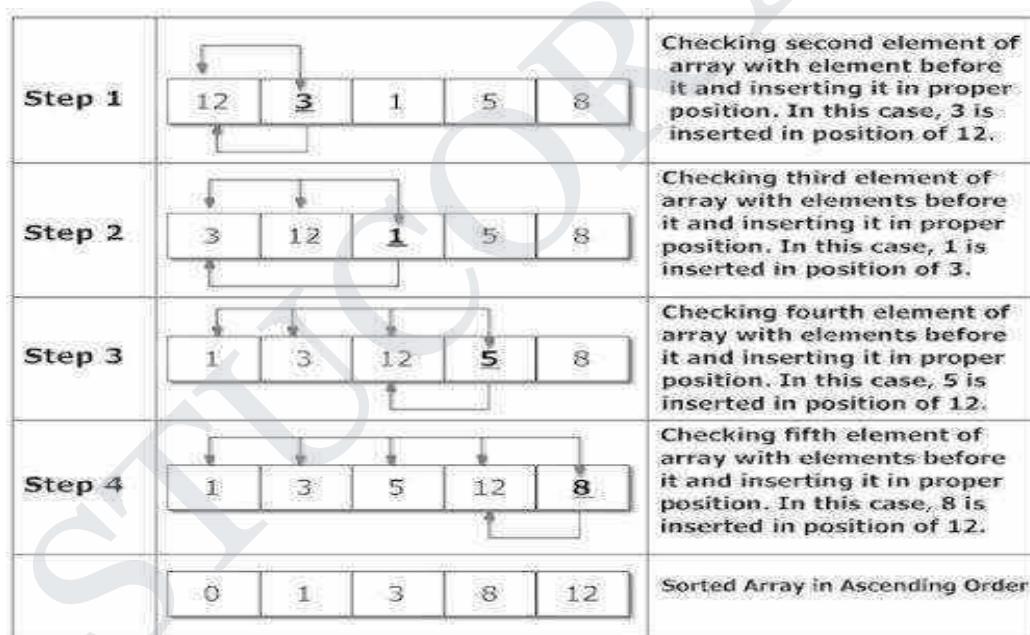
**How Insertion sort algorithm works?**

Figure: Sorting Array in Ascending Order Using Insertion Sort Algorithm

**Insertion Sort routine:**

```
void Insertion_sort(int a[ ], int n)
{
    int i, j, temp;
    for ( i = 0 ; i < n - 1 ; i ++ )
    {
        temp = a [ j ] ;
    }
}
```

```

for ( j = i ; j > 0 && a [ j -1 ] > temp ; j -- )
{
    a[ j ] = a [ j - 1 ] ;

}

a[j]=temp;
}

```

**Program for Insertion sort**

```

#include<stdio.h>
void main( ){
int n, a[ 25 ], i, j, temp;
printf( "Enter number of elements \n" );
scanf( "%d", &n );
printf( "Enter %d integers \n", n );
for ( i = 0; i < n; i++ )
    scanf( "%d", &a[i] );
for ( i = 0 ; i < n; i++ ){
temp=a[i];
for (j=i;j > 0 && a[ j -1]>temp;j--)
{
    a[ j ] = a [ j - 1 ];
}
a[j]=temp;}
printf( "Sorted list in ascending order: \n " );
for ( i = 0 ; i < n ; i++)
    printf ( "%d \n ", a[ i ] );
}

```

**OUTPUT:**

Enter number of elements

6

Enter 6 integers

20 10 60 40 30 15

Sorted list in ascending order:

10

15

20

30

40

60

**Advantage of Insertion sort**

- Simple implementation.
- Efficient for (quite) small data sets.
- Efficient for data sets that are already substantially sorted.

### Disadvantages of Insertion sort

- It is less efficient on list containing more number of elements.
- As the number of elements increases the performance of the program would be slow.
- Insertion sort needs a large number of element shifts.

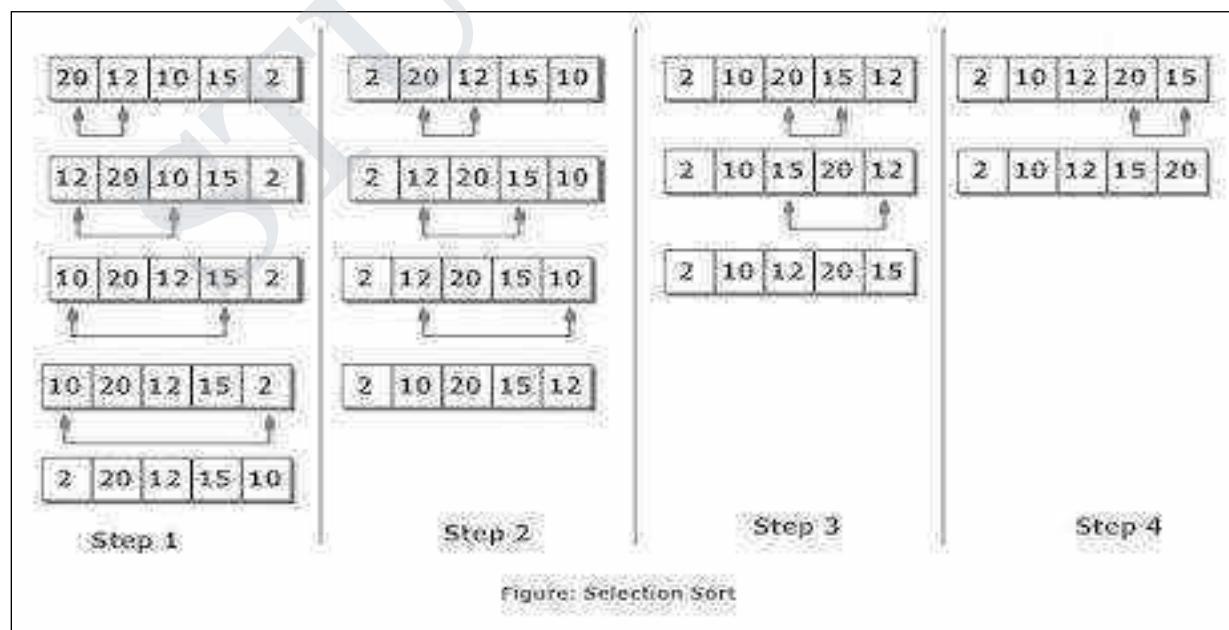
### Selection Sort

Selection sort selects the smallest element in the list and place it in the first position then selects the second smallest element and place it in the second position and it proceeds in the similar way until the entire list is sorted. For “n” elements,  $(n-1)$  passes are required. At the end of the  $i^{\text{th}}$  iteration, the  $i^{\text{th}}$  smallest element will be placed in its correct position.

#### Selection Sort routine:

```
void Selection_sort( int a[ ], int n )
{
    int i , j , temp , position ;
    for ( i = 0 ; i < n - 1 ; i ++ )
    {
        position = i ;
        for ( j = i + 1 ; j < n ; j ++ )
        {
            if ( a[ position ] > a[ j ] )
                position = j;
        }
        temp = a[ i ];
        a[ i ] = a[ position ];
        a[ position ] = temp;
    }
}
```

#### How Selection sort algorithm works?



**Program for Selection sort**

```
#include <stdio.h>
void main( )
{
    int a [ 100 ] , n , i , j , position , temp ;
    printf ( "Enter number of elements \n" );
    scanf ( "%d" , &n ) ;
    printf ( " Enter %d integers \n " , n ) ;
    for ( i = 0 ; i < n ; i ++ )
        scanf ( "%d" , &a[ i ] ) ;
    for ( i = 0 ; i < ( n - 1 ) ; i ++ )
    {
        position = i ;
        for ( j = i + 1 ; j < n ; j ++ )
        {
            if ( a [ position ] > a [ j ] )
                position = j ;
        }
        if ( position != i )
        {
            temp = a [ i ] ;
            a [ i ] = a [ position ] ;
            a [ position ] = temp ;
        }
    }
    printf ( "Sorted list in ascending order: \n" );
    for ( i = 0 ; i < n ; i ++ )
        printf ( " %d \n " , a[ i ] ) ;
}
```

**OUTPUT:**

Enter number of elements

5

Enter 5 integers

8 3 9 5 1

Sorted list in ascending order:

1

3

5

8

9

**Advantages of selection sort**

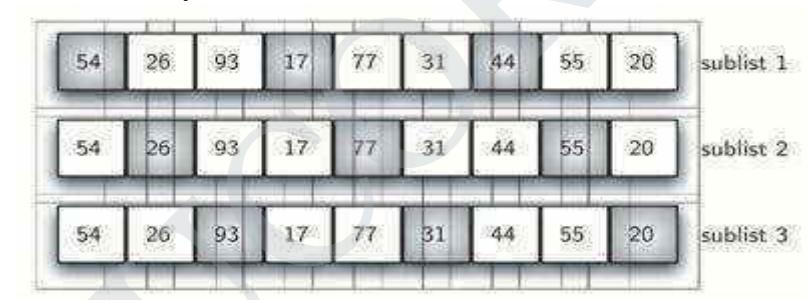
- Memory required is small.
- Selection sort is useful when you have limited memory available.
- Relatively efficient for small arrays.

### Disadvantage of selection sort

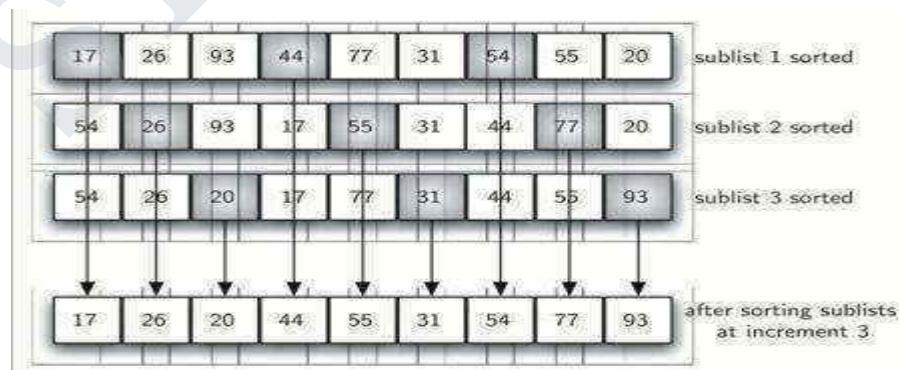
- Poor efficiency when dealing with a huge list of items.
- The selection sort requires  $n^2$  number of steps for sorting  $n$  elements.
- The selection sort is only suitable for a list of few elements that are in random order.

### Shell Sort

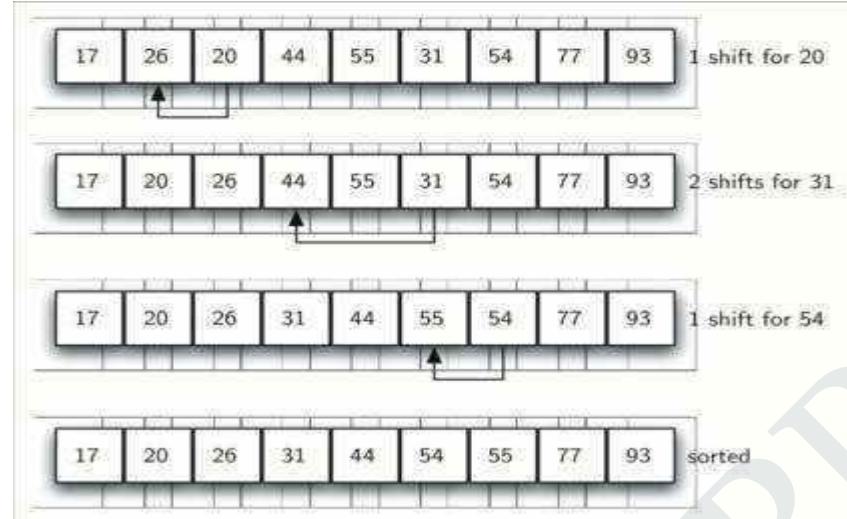
- Invented by Donald shell.
- It improves upon bubble sort and insertion sort by moving out of order elements more than one position at a time.
- In shell sort the whole array is first fragmented into  $K$  segments, where  $K$  is preferably a prime number.
- After the first pass the whole array is partially sorted.
- In the next pass, the value of  $K$  is reduced which increases the size of each segment and reduces the number of segments.
- The next value of  $K$  is chosen so that it is relatively prime to its previous value.
- The process is repeated until  $K=1$  at which the array is sorted.
- The insertion sort is applied to each segment so each successive segment is partially sorted.
- The shell sort is also called the Diminishing Increment sort, because the value of  $k$  decreases continuously



A Shell Sort with Increments of Three



A Shell Sort after Sorting Each Sublist



Shell Sort: A Final Insertion Sort with Increment of 1

### Shell Sort routine:

```
void Shell_sort ( int a[ ], int n )
{
    int i, j, k, temp;
    for ( k = n / 2 ; k > 0 ; k = k / 2 )
        for ( i = k ; i < n ; i ++ )
        {
            temp = a [ i ];
            for ( j = i ; j >= k && a [ j - k ] > temp ; j = j - k )
                {
                    a [ j ] = a [ j - k ];
                }
            a [ j ] = temp ;
        }
}
```

### Program for Shell sort

```
#include<stdio.h>
void main( )
{
    int n, a[ 25 ], i, j,k,temp;
    printf( "Enter number of elements \n" );
    scanf( "%d", &n );
    printf( "Enter %d integers \n", n );
    for ( i = 0; i < n; i++ )
        scanf( "%d", &a[i] );
        for (k = n / 2 ; k>0 ; k=k/2){
```

```
for ( i = k ; i < n ; i ++ )
{
    temp = a [ i ] ;
    for (j = i ; j >= k && a [ j - k ] > temp ; j = j - k )
    {
        a [ j ] = a [ j - k ] ;
    }
    a [ j ] = temp ;
}
printf( "Sorted list in ascending order using shell sort: \n ");
for ( i = 0 ; i < n ; i ++ )
    printf ( "%d\t ", a[ i ] );
```

**OUTPUT:**

Enter number of elements

10

Enter 10 integers

81 94 11 96 12 35 17 95 28 58

Sorted list in ascending order using shell sort:

11    12    17    28    35    58    81    94    95    96

**//PROGRAM FOR SHELL USING FUNCTION**

```
#include < stdio.h >
void main( )
{
    int a [ 5 ] = { 4, 5, 2, 3, 6 } , i = 0 ;
    ShellSort ( a, 5 ) ;
printf("Example using function");
    printf( " After Sorting :" );
    for ( i = 0 ; i < 5 ; i ++ )
        printf ( " %d ", a[ i ] );
}
void ShellSort (int a [ 5 ] , int n )
{
    int i , j , k , temp ;
    for ( k = n / 2 ; k > 0 ; k /= 2 )
    {
        for ( i = k ; i < n ; i ++ )
        {
            temp = a [ i ] ;
            for ( j = i ; j >= k && a [ j - k ] > temp ; j = j - k ){
                a [ j ] = a [ j - k ] ;
            }
            a [ j ] = temp ; } }}
```

**OUTPUT:**

After Sorting : 2 3 4 5 6

**Advantages of Shell sort**

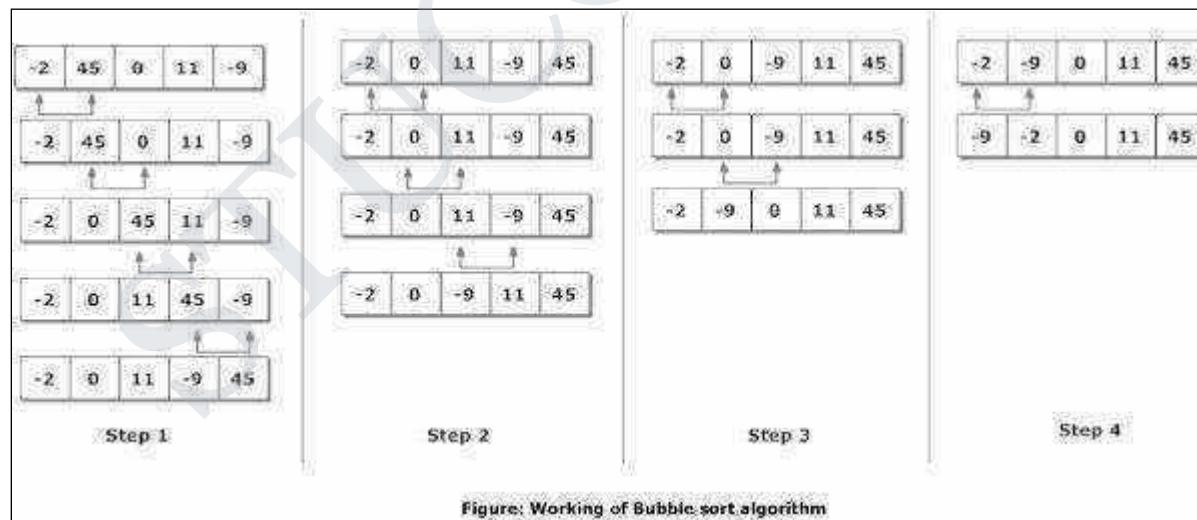
- Efficient for medium-size lists.

**Disadvantages of Shell sort**

- Complex algorithm, not nearly as efficient as the merge, heap and quick sorts

**Bubble Sort**

- Bubble sort is one of the simplest internal sorting algorithms.
- Bubble sort works by comparing two consecutive elements and the largest element among these two bubbles towards right at the end of the first pass the largest element gets sorted and placed at the end of the sorted list.
- This process is repeated for all pairs of elements until it moves the largest element to the end of the list in that iteration.
- Bubble sort consists of  $(n-1)$  passes, where  $n$  is the number of elements to be sorted.
- In 1<sup>st</sup> pass the largest element will be placed in the  $n^{\text{th}}$  position.
- In 2<sup>nd</sup> pass the second largest element will be placed in the  $(n-1)^{\text{th}}$  position.
- In  $(n-1)^{\text{th}}$  pass only the first two elements are compared.

**Bubble sort routine:**

```
void Bubble_sort (int a [ ] , int n )  
{  
    int i, j, temp;  
    for( i = 0; i < n - 1; i++ ){
```

```
for( j = 0; j < n - i - 1; j++ )  
{  
    if( a[ j ] > a[ j + 1 ] )  
    {  
        temp = a[ j ];  
        a[ j ] = a[ j + 1 ];  
        a[ j + 1 ] = temp;  
    }  
} } }
```

**Program for Bubble sort**

```
#include<stdio.h>  
#include<conio.h>  
void main()  
{  
int a [ 20 ], i, j, temp, n ;  
printf ("Enter the number of elements");  
    scanf ("%d",&n);  
printf("Enter the numbers");  
for(i=0;i < n ;i++)  
scanf("%d",&a[i]);  
for(i=0;i<n-1;i++)  
{  
for(j=0;j<n-i-1;j++)  
{  
if(a[j]>a[j + 1])  
    {  
        temp = a[ j ] ;  
        a[ j ] = a[ j + 1 ] ;  
        a[ j + 1 ] = temp;  
    }  
}  
}  
printf("\nSorted array\t");  
for(i=0;i<n;i++)  
    printf("%d\t",a[i]);  
}
```

**OUTPUT:**

Enter the number of elements5

Enter the numbers8 3 9 5 1

Sorted array 1 3 5 8 9

**Advantage of Bubble sort**

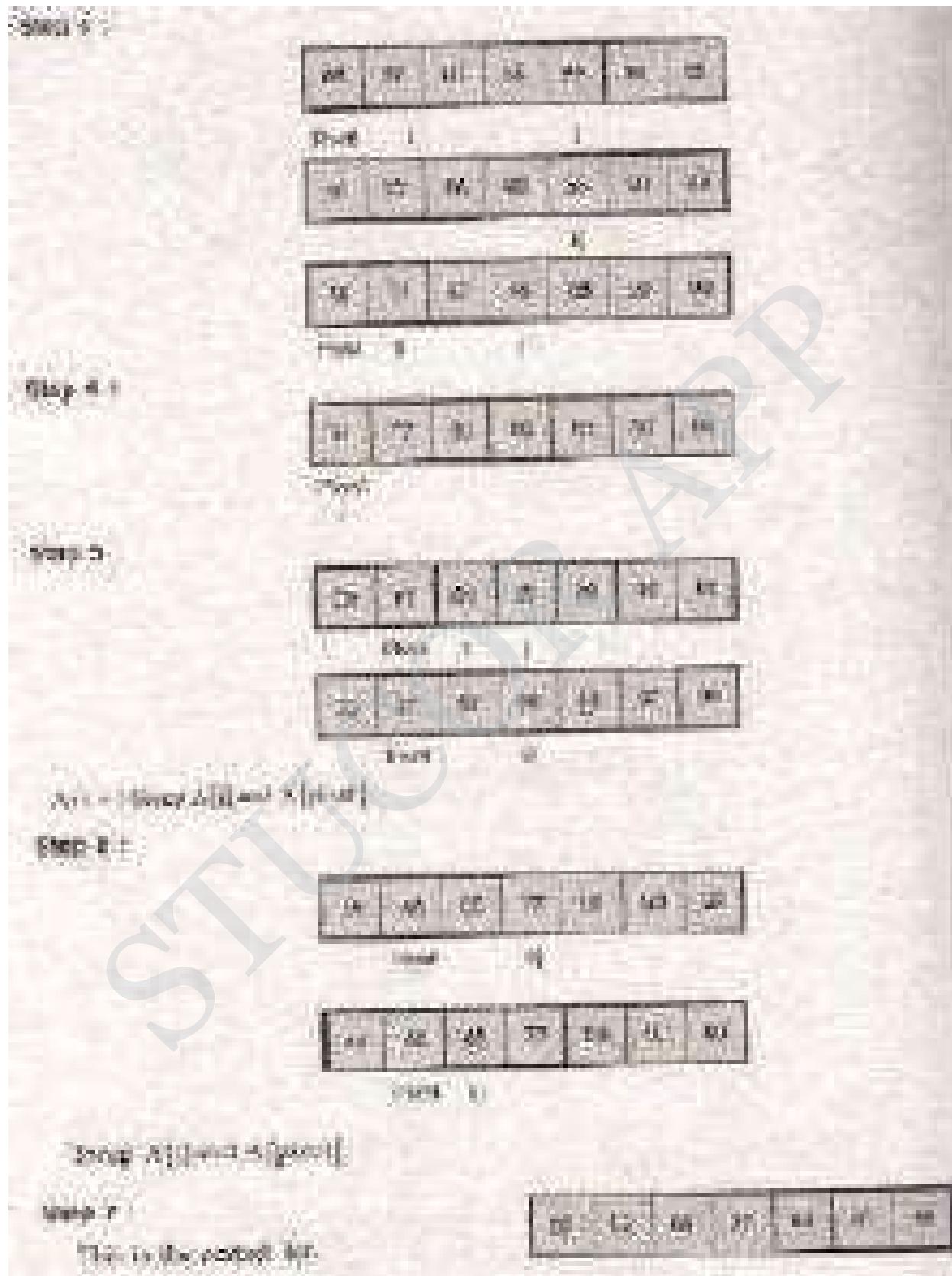
- It is simple to write
- Easy to understand
- It only takes a few lines of code.

**Disadvantage of Bubble sort**

- The major drawback is the amount of time it takes to sort.
- The average time increases almost exponentially as the number of table elements increase.

**Quick Sort**

- Quicksort is a divide and conquer algorithm.
  - The basic idea is to find a “pivot” item in the array and compare all other items with pivot element.
  - Shift items such that all of the items before the pivot are less than the pivot value and all the items after the pivot are greater than the pivot value.
  - After that, recursively perform the same operation on the items before and after the pivot.
  - Find a “pivot” item in the array. This item is the basis for comparison for a single round.
  - Start a pointer (the left pointer) at the first item in the array.
  - Start a pointer (the right pointer) at the last item in the array.
1. Assume  $A[0]=\text{pivot}$  which is the left. i.e  $\text{pivot}=\text{left}$ .
  2. Set  $i=\text{left}+1$ ; i.e  $A[1]$ ;
  3. Set  $j=\text{right}$ . ie.  $A[6]$  if there are 7 elements in the array
  4. If  $A[\text{pivot}]>A[i]$ ,increment  $i$  and if  $A[j]>A[\text{pivot}]$ ,then decrement  $j$ , Otherwise swap  $A[i]$  and  $A[j]$  element.
  5. If  $i=j$ ,then swap  $A[\text{pivot}]$  and  $A[j]$ .



**Quick Sort routine:**

```

void Quicksort ( int a [ ] , int left, int right )
{
    int i, j, p, temp;
    if ( left < right )
    {
        p = left;
        i = left + 1; j
        = right;
        while ( i < j )
        {
            while ( a [ i ] <= a [ p ] )
                i = i + 1;
            while ( a [ j ] > a [ p ] )
                j = j - 1;
            if ( i < j )
            {
                temp = a [ i ];
                a [ i ] = a [ j ];
                a [ j ] = temp;
            }
        }
        temp = a [ p ];
        a [ p ] = a [ j ];
        a [ j ] = temp;
        quicksort ( a, left, j - 1 );
        quicksort ( a, j + 1, right );
    }
}

```

**Program for Quick sort**

```

#include<stdio.h>
void quicksort (int [10], int, int ) ;
void main( )
{
    int a[20], n, i ;
    printf("Enter size of the array: " );
    scanf("%d",&n);
    printf( " Enter the numbers :");
    for ( i = 0 ; i < n ; i ++ )
        scanf ("%d",&a[i]);
    quicksort ( a , 0 , n - 1 );
    printf ( " Sorted elements: " );
    for ( i = 0 ; i < n ; i ++ )
        printf ("%d\t",a[ i]);
}

```

```
void quicksort ( int a[10], int left, int right )
{
    int p, j, temp, i ;
    if ( left < right )
    {
        p = left ;
        i = left ;
        j = right ;
        while ( i < j )
        {
            while(a[i]<= a[p] && i<right )
                i++ ;
            while ( a [ j ] > a [ p ] )
                j-- ;
            if ( i < j )
            {
                temp = a [ i ] ;
                a [ i ] = a [ j ] ;
                a[ j ] = temp ;
            }
        }
        temp = a [ p ] ;
        a [ p ] = a [ j ] ;
        a [ j ] =temp ;
        quicksort ( a , left , j - 1 ) ;
        quicksort ( a , j + 1 , right ) ;
    }
}
```

**OUTPUT:**

Enter size of the array:8

Enter the numbers :40 20 70 14 60 61 97 30

Sorted elements: 14    20    30    40    60    61    70    97

**Advantages of Quick sort**

- Fast and efficient as it deals well with a huge list of items.
- No additional storage is required.

**Disadvantages of Quick sort**

- The difficulty of implementing the partitioning algorithm.

## Merge Sort

Merge sort is a sorting algorithm that uses the divide, conquer, and combine algorithmic paradigm.

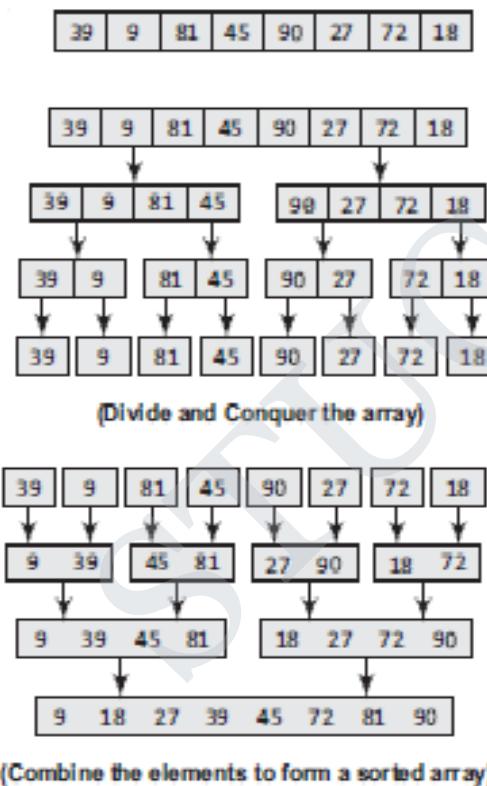
Divide means partitioning the n-element array to be sorted into two sub-arrays of  $n/2$  elements. If there are more elements in the array, divide A into two sub-arrays, A<sub>1</sub> and A<sub>2</sub>, each containing about half of the elements of A.

Conquer means sorting the two sub-arrays recursively using merge sort.

Combine means merging the two sorted sub-arrays of size  $n/2$  to produce the sorted array of n elements.

The basic steps of a merge sort algorithm are as follows:

- If the array is of length 0 or 1, then it is already sorted.
- Otherwise, divide the unsorted array into two sub-arrays of about half the size.
- Use merge sort algorithm recursively to sort each sub-array.
- Merge the two sub-arrays to form a single sorted list.



## Merge Sort routine:

```
void Merge_sort (int a [ ] , int temp [ ] , int n )  
{  
    msort ( a , temp , 0 , n - 1 ) ;  
}
```

```
void msort ( int a[ ] , int temp [ ] , int left , int right ){
    int center ;
    if( left < right ){
        center = ( left + right ) / 2 ;
        msort ( a , left , center ) ;
        msort ( a , temp , center + 1 , right ) ;
        merge ( a , temp , n , left , center , right ) ;
    }
}

void merge ( int a[ ] , int temp [ ] , int n , int left , int center , int right )
{
    int i = 0 , j , left_end = center , center = center + 1 ;
    while( ( left <= left_end ) && ( center <= right ) )
    {
        if( a [ left ] <= a [ center ] )
        {
            temp [ i ] = a [ left ] ;
            i ++ ;
            left ++ ;
        }
        else
        {
            temp [ i ] = a [ center ] ;
            i ++ ;
            center ++ ;
        }
    }
    while( left <= left_end )
    {
        temp [ I ] = a [ left ] ;
        left ++ ;
        i ++ ;
    }
    while( center <= right )
    {
        temp [ i ] = a [ center ] ;
        center ++ ;
        i ++ ;
    }
    for ( i = 0 ; i < n ; i ++ )
        print temp [ i ] ;
}
```

### Program for merge sort

```
#include<stdio.h>
void mergesort(int a[],int i,int j);
void merge(int a[],int i1,int j1,int i2,int j2);
```

```
int main()
{
    int a[30],n,i;
    printf("Enter no of elements:");
    scanf("%d",&n);
    printf("Enter array elements:");
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);
    mergesort(a,0,n-1);
    printf("\nSorted array is :");
    for(i=0;i<n;i++)
        printf("%d ",a[i]);
    return 0;
}
void mergesort(int a[],int i,int j)
{
    int mid;
    if(i<j)
    {
        mid=(i+j)/2;
        mergesort(a,i,mid);      //left recursion mergesort(a,mid+1,j);
        //right recursion merge(a,i,mid,mid+1,j); //merging of two
        sorted sub-arrays
    }
}
void merge(int a[],int i1,int j1,int i2,int j2)
{
    int temp[50]; //array used for merging
    int i,j,k;
    i=i1; //beginning of the first list
    j=j1; //beginning of the second list
    k=0;
    while(i<=j1 && j<=j2) //while elements in both lists
    { if(a[i]<a[j])
        temp[k++]=a[i++];
    else
        temp[k++]=a[j++];
    }
    while(i<=j1) //copy remaining elements of the first list
    temp[k++]=a[i++];
    while(j<=j2) //copy remaining elements of the second list
    temp[k++]=a[j++];
    //Transfer elements from temp[] back to a[]
    for(i=i1,j=0;i<=j2;i++,j++)
        a[i]=temp[j];
}
```

**OUTPUT:**

Enter no of elements:8  
Enter array elements:24 13 26 1 2 27 38 15  
Sorted array is :1 2 13 15 24 26 27 38

**Advantages of Merge sort**

- Mergesort is well-suited for sorting really huge amounts of data that does not fit into memory.
- It is fast and stable algorithm

**Disadvantages of Merge sort**

- Merge sort uses a lot of memory.
- It uses extra space proportional to number of element n.
- This can slow it down when attempting to sort very large data.

**Radix Sort**

- Radix sort is one of the linear sorting algorithms. It is generalized form of bucket sort. It can be performed using buckets from 0 to 9.
- It is also called binsort, card sort.
- It works by sorting the input based on each digit. In first pass all the elements are stored according to the least significant digit.
- In second pass the elements are arranged according to the next least significant digit and so on till the most significant digit.
- The number of passes in a Radix sort depends upon the number of digits in the given numbers.

**Algorithm for Radix sort**

**Steps1:** Consider 10 buckets (1 for each digit 0 to 9)

**Step2:** Consider the LSB (Least Significant Bit) of each number (numbers in the one's place.... E.g., in 43 LSB = 3)

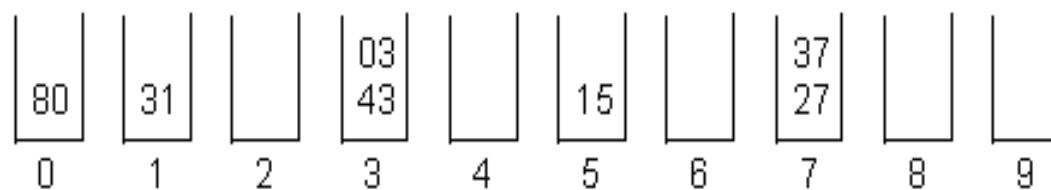
**Step3:** Place the elements in their respective buckets according to the LSB of each number

**Step4:** Write the numbers from the bucket (0 to 9) bottom to top.

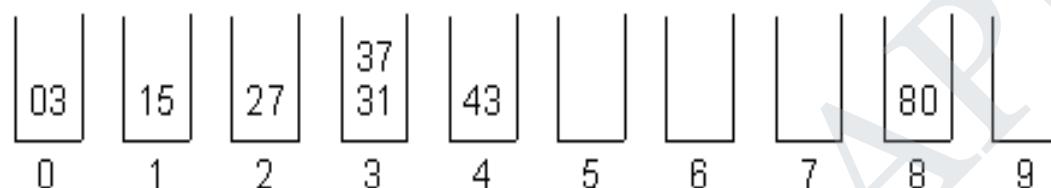
**Step5:** repeat the same process with the digits in the  $10^{\text{th}}$  place (e.g. In 43 MSB =4)

**Step6:** repeat the same step till all the digits of the given number are consider.

43 27 31 15 37 80 03



: 80 31 43 03 15 27 37



03 15 27 31 37 43 80

Sorted list of array : 3 15 27 31 37 43 80

### Routine for Radix sort

```
void Radix_sort ( int a [ ] , int n )
{
    int bucket [ 10 ] [ 5 ] , buck [ 10 ] , b [ 10 ] ;
    int i , j , k , l , num , div , large , passes ;
    div = 1 ;
    num = 0 ;
    large = a [ 0 ] ;
    for ( i = 0 ; i < n ; i ++ )
    {
        if ( a[ I ] > large )
        {
            large = a [ i ] ;
        }
        while ( large > 0 )
        {
            num ++ ;
```

```
        large = large / 10 ;
    }
    for ( passes = 0 ; passes < num ; passes ++ )
    {
        for ( k = 0 ; k < 10 ; k ++ )
        {
            buck [ k ] = 0 ;
        }
        for ( i = 0 ; i < n ; i ++ )
        {
            l = ( ( a [ i ] / div ) % 10 ) ;
            bucket [ 1 ] [ buck [ 1 ] ++ ] = a [ i ] ;
        }
        i = 0 ;
        for ( k = 0 ; k < 10 ; k ++ )
        {
            for ( j = 0 ; j < buck [ k ] ; j ++ )
            {
                a [ i ++ ] = bucket [ k ] [ j ] ;
            }
        }
        div * = 10 ;
    }
}
```

### Program for Radix sort

```
#include<stdio.h >
void main( )
{
    int a [ 5 ] = { 4, 5, 2, 3, 6 } , i = 0 ;
    void Radix_sort ( int a [ ] , int n );
    Radix_sort(a,5);
    printf( " After Sorting :" );
    for ( i = 0 ; i < 5 ; i ++ )
        printf ( " %d ", a[ i ] );
}
void Radix_sort ( int a [ ] , int n )
{
    int bucket [ 10 ] [ 5 ] , buck [ 10 ] , b [ 10 ] ;
    int i , j , k , l , num , div , large , passes ;
    div = 1 ;
    num = 0 ;
    large = a [ 0 ] ;
    for ( i = 0 ; i < n ; i ++ ){
```

```
if ( a[ i ] > large )
{
    large = a [ i ] ;
}
while ( large > 0 )
{
    num ++ ;
    large = large / 10 ;
}
for ( passes = 0 ; passes < num ; passes ++ )
{
    for ( k = 0 ; k < 10 ; k ++ )
    {
        buck [ k ] = 0 ;
    }
    for ( i = 0 ; i < n ; i ++ )
    {
        l = ( ( a [ i ] / div ) % 10 ) ;
        bucket [ 1 ] [ buck [ 1 ] ++ ] = a [ i ] ;
    }
    i = 0 ;
    for ( k = 0 ; k < 10 ; k ++ )
    {
        for(j=0 ; j<buck[k];j++)
        {
            a[i++]=bucket[ k ][ j ] ;
        }
    }
    div *= 10 ;
}
```

**Advantages of Radix sort:**

- Fast and complexity does not depend on the number of data.
- Radix Sort is very simple.

**Disadvantages of Radix sort:**

- Radix Sort takes more space than other sorting algorithms, since in addition to the array that will be sorted, you need to have a sub list for each of the possible digits or letters.
- Since Radix Sort depends on the digits or letters, Radix Sort is also much less flexible than other sorts.

**Analysis of Sorting algorithms:**

S.No	Algorithm	Best Case Analysis	Average Case Analysis	Worst Case Analysis
1	Insertion sort	$O(N)$	$O(N^2)$	$O(N^2)$
2	Selection sort	$O(N^2)$	$O(N^2)$	$O(N^2)$
3	Shell sort	$O(N \log N)$	$O(N^{1.5})$	$O(N^2)$
4	Bubble sort	$O(N^2)$	$O(N^2)$	$O(N^2)$
5	Quick sort	$O(N \log N)$	$O(N \log N)$	$O(N^2)$
6	Merge sort	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$
7	Radix or bucket or binsort sort or card sort	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$

**SEARCHING**

Searching is an algorithm, to check whether a particular element is present in the list.

**Types of searching:-**

- Linear search
- Binary Search

**Linear Search**

Linear search is used to search a data item in the given set in the sequential manner, starting from the first element. It is also called as sequential search

**Linear Search routine:**

```
void Linear_search ( int a[ ] , int n )
{
int search , count = 0 ;
for ( i = 0 ; i < n ; I ++ )
{
if ( a [ i ] == search )
{
count ++ ;
}
}
if ( count == 0 )
print "Element not Present" ;
else
print "Element is Present in list" ;
}
```

**Program for Linear search**

```
#include < stdio.h >
void main( )
{
int a [ 10 ] , n , i , search, count = 0 ;
printf ( " Enter the number of elements \t " );
scanf ( "%d" , & n );
printf ( "\n Enter %d numbers \n " , n );
for ( i = 0 ; i < n ; i ++ )
    scanf ( "%d" , & a [ i ] );
printf ( "\n Array Elements \n " );
for ( i = 0 ; i < n ; i ++ )
    printf ( "%d \t " , a [ i ] );
printf ( "\n \n Enter the Element to be searched: \t " );
scanf ( "%d" , & search );
for ( i = 0 ; i < n ; i ++ )
{
    if ( search == a [ i ] )

        count ++ ;
}
if ( count == 0 )
    printf( "\n Element %d is not present in the array " , search );
else
    printf ( "\n Element %d is present %d times in the array \n " , search , count );
}
```

**OUTPUT:**

```
Enter the number of elements 5
Enter the numbers
20 10 5 25 100
Array Elements
20 10 5 25 100
Enter the Element to be searched: 25
Element 25 is present 1 times in the array
```

**Advantages of Linear search:**

- The linear search is simple - It is very easy to understand and implement;
- It does not require the data in the array to be stored in any particular order.

**Disadvantages of Linear search:**

- Slower than many other search algorithms.
- It has a very poor efficiency.

**Binary Search**

- Binary search is used to search an item in a sorted list. In this method , initialize the lower limit and upper limit.
- The middle position is computed as  $(\text{first}+\text{last})/2$  and check the element in the middle position with the data item to be searched.
- If the data item is greater than the middle value then the lower limit is adjusted to one greater than the middle value.Otherwise the upper limit is adjusted to one less than the middle value.

**Working principle:**

Algorithm is quite simple. It can be done either recursively or iteratively:

1. Get the middle element;
2. If the middle element equals to the searched value, the algorithm stops;
3. Otherwise, two cases are possible:
  - o Search value is less than the middle element. In this case, go to the step 1 for the part of the array, before middle element.
  - o Searched value is greater, than the middle element. In this case, go to the step 1 for the part of the array, after middle element.

**Example 1.**

Find 6 in {-1, 5, 6, 18, 19, 25, 46, 78, 102, 114}.

Step 1 (middle element is 19 > 6): -1 5 6 18 19 25 46 78 102 114

Step 2 (middle element is 5 < 6): -1 5 6 18 19 25 46 78 102 114

Step 3 (middle element is 6 == 6): -1 5 6 18 19 25 46 78 102 114

**Binary Search routine:**

```
void Binary_search ( int a[ ] , int n , int search )
{
int first, last, mid ;
first = 0 ;
last = n-1 ;
mid = ( first + last ) / 2 ;
while ( first <= last )
{
if ( Search > a [ mid ] )
first = mid + 1 ;
else if ( Search == a [ mid ] )
{
print "Element is present in the list" ;
break ;
}
else {
```

```
last = mid - 1 ;
mid = ( first + last ) / 2 ;
}
if( first > last )
print "Element Not Found" ;
}
```

**Program for Binary Search:**

```
#include<stdio.h>
void main( )
{
int a [ 10 ] , n , i , search, count = 0 ;
void Binary_search ( int a[ ] , int n , int search );
printf ("Enter the number of elements \t");
scanf ("%d",&n);
printf("\nEnter the numbers\n");
for (i = 0; i<n;i++)
    scanf("%d",&a[i]);
printf("\nArray Elements\n");
for (i = 0 ; i < n ; i++)
printf("%d\t",a[i]);
printf ("\n\nEnter the Element to be searched:\t");
scanf("%d",&search );
Binary_search(a,n,search);
}
void Binary_search ( int a[ ] , int n , int search )
{
int first, last, mid ;
first = 0 ;
last = n-1 ;
mid = (first + last ) / 2 ;
while (first<=last )
{
if(search>a[mid])
first = mid + 1 ;
else if (search==a[mid])
{
printf("Element is present in the list");
break ;
}
else
last = mid - 1 ;
mid = ( first + last ) / 2 ;
}
if( first > last )
printf("Element Not Found");
}
```

**OUTPUT:**

Enter the number of elements 5

Enter the numbers

20 25 50 75 100

Array Elements

20 25 50 75 100

Enter the Element to be searched: 75

Element is present in the listPress any key to continue . . .

**Advantages of Binary search:**

- In Linear search, the search element is compared with all the elements in the array. Whereas in Binary search, the search element is compared based on the middle element present in the array.
- A technique for searching an ordered list in which we first check the middle item and - based on that comparison - "discard" half the data. The same procedure is then applied to the remaining half until a match is found or there are no more items left.

**Disadvantages of Binary search:**

- Binary search algorithm employs recursive approach and this approach requires more stack space.
- It requires the data in the array to be stored in sorted order.
- It involves additional complexity in computing the middle element of the array.

**Analysis of Searching algorithms:**

S.No	Algorithm	Best Case Analysis	Average Case Analysis	Worst Case Analysis
1	Linear search	O(1)	O(N)	O(N)
2	Binary search	O(1)	O(log N)	O(log N)

## HASHING :

Hashing is a technique that is used to store, retrieve and find data in the data structure called Hash Table. It is used to overcome the drawback of Linear Search (Comparison) & Binary Search (Sorted order list). It involves two important concepts-

- Hash Table
- Hash Function

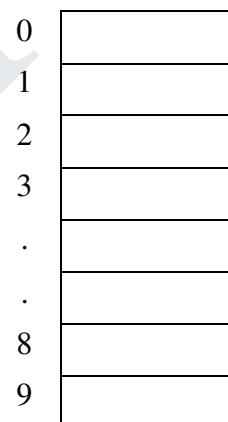
### **Hash table**

- A **hash table** is a data structure that is used to store and retrieve data (keys) very quickly.
- It is an array of some fixed size, containing the keys.
- Hash table run from 0 to Tablesize – 1.
- Each key is mapped into some number in the range 0 to Tablesize – 1.
- This mapping is called Hash function.
- Insertion of the data in the hash table is based on the key value obtained from the hash function.
- Using same hash key value, the data can be retrieved from the hash table by few or more Hash key comparison.
- The **load factor** of a hash table is calculated using the formula:

$$\text{(Number of data elements in the hash table) / (Size of the hash table)}$$

### **Factors affecting Hash Table Design**

- Hash function
- Table size.
- Collision handling scheme



**Simple Hash table with table size = 10**

**Hash function:**

- It is a function, which distributes the keys evenly among the cells in the Hash Table.
- Using the same hash function we can retrieve data from the hash table.
- Hash function is used to implement hash table.
- **The integer value returned by the hash function is called hash key.**
- If the input keys are integer, the commonly used hash function is

$$H(\text{key}) = \text{key \% Tablesize}$$

```
typedef unsigned int index;  
index Hash ( const char *key , int Tablesize )  
{  
    unsigned int Hashval = 0 ;  
    while ( *key != '\0' )  
        Hashval += *key ++ ;  
    return ( Hashval % Tablesize ) ;  
}
```

**A simple hash function****Types of Hash Functions**

1. Division Method
2. Mid Square Method
3. Multiplicative Hash Function
4. Digit Folding

**1. Division Method:**

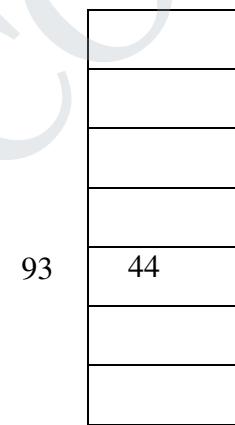
- It depends on remainder of division.
- Divisor is Table Size.
- Formula is ( $H(\text{key}) = \text{key \% table size}$ )

E.g. consider the following data or record or key (36, 18, 72, 43, 6) table size = 8

	[0]	72
Assume a table with 8 slots:	[1]	
Hash key = key % table size	[2]	18
4 = 36 % 8	[3]	43
2 = 18 % 8	[4]	36
0 = 72 % 8	[5]	
3 = 43 % 8	[6]	6
6 = 6 % 8	[7]	

## 2. Mid Square Method:

We first square the item, and then extract some portion of the resulting digits. For example, if the item were 44, we would first compute  $44^2=1,936$ . Extract the middle two digit 93 from the answer. Store the key 44 in the index 93.



## 3. Multiplicative Hash Function:

Key is multiplied by some constant value.

Hash function is given by,

$$H(\text{key})=\text{Floor} (\text{P} * (\text{key} * \text{A}))$$

P = Integer constant [e.g. P=50]

A = Constant real number [A=0.61803398987], suggested by Donald Knuth to use this constant

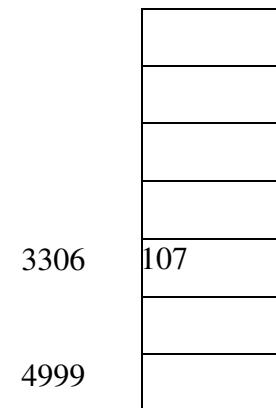
E.g. Key 107

$$H(107)=\text{Floor}(50*(107*0.61803398987))$$

$$=\text{Floor}(3306.481845)$$

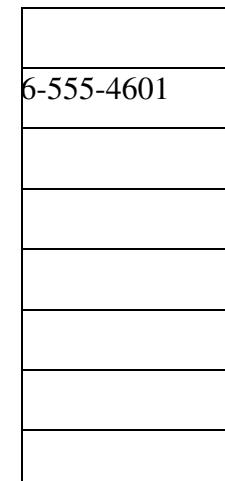
$$H(107)=3306$$

Consider table size is 5000



#### 4. Digit Folding Method:

The folding method for constructing hash functions begins by dividing the item into equal-size pieces (the last piece may not be of equal size). These pieces are then added together to give the resulting hash key value. For example, if our item was the phone number 436-555-4601, we would take the digits and divide them into groups of 2 (43, 65, 55, 46, 01). After the addition,  $43+65+55+46+01$ , we get 210. If we assume our hash table has 11 slots, then we need to perform the extra step of dividing by 11 and keeping the remainder. In this case  $210 \% 11$  is 1, so the phone number 436-555-4601 hashes to slot 1.



**Collision:**

If two or more keys hash to the same index, the corresponding records cannot be stored in the same location. This condition is known as collision.

**Characteristics of Good Hashing Function:**

- It should be Simple to compute.
- Number of Collision should be less while placing record in Hash Table.
- **Hash function with no collision → Perfect hash function.**
- Hash Function should produce keys which are distributed uniformly in hash table.
- The hash function should depend upon every bit of the key. Thus the hash function that simply extracts the portion of a key is not suitable.

**Collision Resolution Strategies / Techniques (CRT):**

If collision occurs, it should be handled or overcome by applying some technique. Such technique is called CRT.

There are a number of collision resolution techniques, but the most popular are:

- **Separate chaining** (Open Hashing)
- **Open addressing**. (Closed Hashing)
  - **Linear Probing**
  - **Quadratic Probing**
  - **Double Hashing**

**Separate chaining (Open Hashing)**

- Open hashing technique.
- Implemented using singly linked list concept.
- Pointer (ptr) field is added to each record.
- When collision occurs, a separate chaining is maintained for colliding data.
- Element inserted in front of the list.

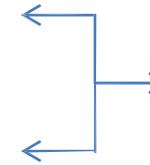
$$H(\text{key}) = \text{key \% table size}$$

Two operations are there:-

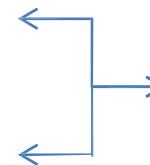
- Insert
- Find

**Structure Definition for Node**

```
typedef Struct node *Position;  
Struct node  
{  
    int data;  
    Position next;  
};
```

**Structure Definition for Hash Table**

```
typedef Position List;  
struct Hashtbl  
{  
    int Tablesiz;  
    List * theLists;  
};
```

**Initialization for Hash Table for Separate Chaining**

```
Hashtable initialize(int Tablesiz)  
{  
    HashTable H;  
    int i;  
    H = malloc (sizeof(struct HashTbl)); → Allocates table  
    H → Tablesiz = NextPrime(Tablesiz);  
    H → the Lists=malloc(sizeof(List) * H → Tablesiz); → Allocates array of list  
    for( i = 0; i < H → Tablesiz; i++ )  
    {  
        H → TheLists[i] = malloc(Sizeof(Struct node)); → Allocates list headers  
        H → TheLists[i] → next = NULL;  
    }  
    return H;  
}
```

**Insert Routine for Separate Chaining**

```
void insert (int Key, Hashtable H)  
{  
    Position P, newnode; *[Inserts element in the Front of the list always]*  
    List L;
```

```
P = find ( key, H );
if(P == NULL)
{
    newnode = malloc(sizeof(Struct node));
    L = H → TheLists[Hash(key,Tablesize)];
    newnode → nex t= L → next;
    newnode → data = key;
    L → next = newnode;
}
Position find( int key, Hashtable H){
    Position P, List L;
    L = H → TheLists[Hash(key,Tablesize)];
    P = L → next;
    while(P != NULL && P → data != key)
        P = P → next;
    return P;
}
```

If two keys map to same value, the elements are chained together.

Initial configuration of the hash table with separate chaining. Here we use SLL(Singly Linked List) concept to chain the elements.

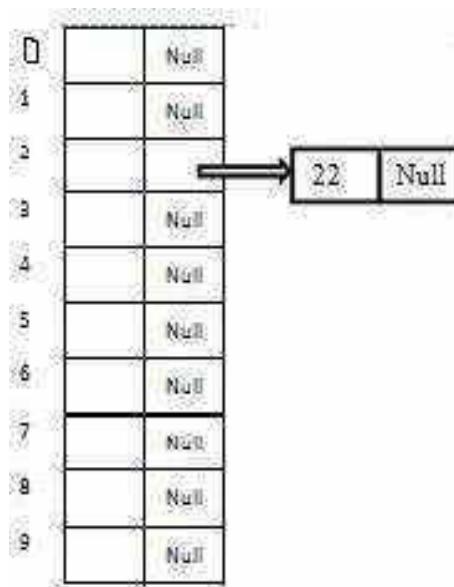
0	NULL
1	NULL
2	NULL
3	NULL
4	NULL
5	NULL
6	NULL
7	NULL
8	NULL
9	NULL

Insert the following four keys 22 84 35 62 into hash table of size 10 using separate chaining.

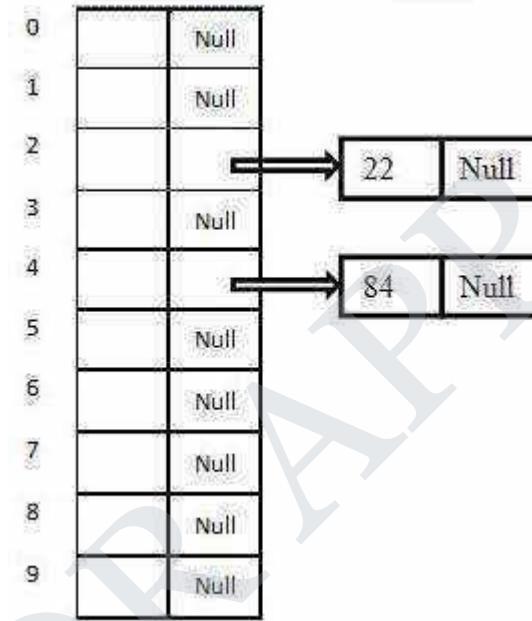
The hash function is

$$H(key) = key \% 10$$

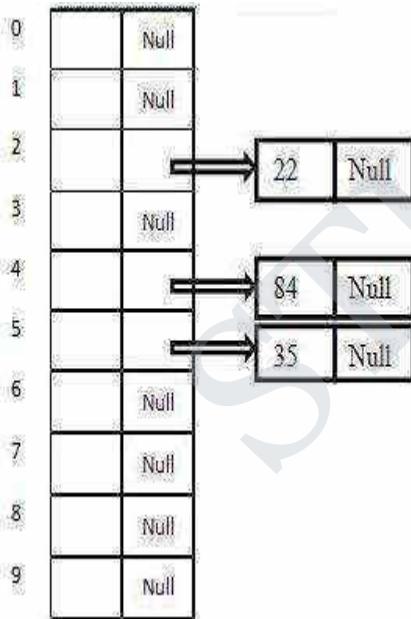
$$1. H(22) = 22 \% 10 = 2$$



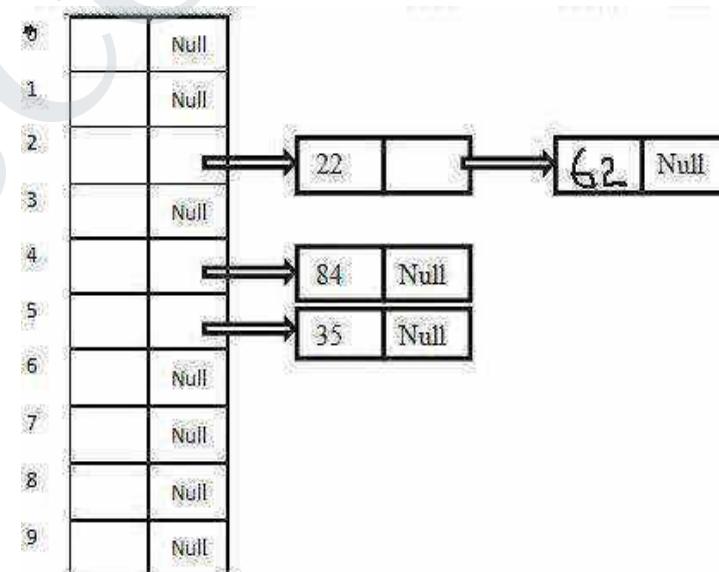
$$2. H(84) = 84 \% 10 = 4$$



$$3. H(35) = 35 \% 10 = 5$$



$$4. H(62) = 62 \% 10 = 2$$



### **Advantages**

1. More number of elements can be inserted using array of Link List

### **Disadvantages**

1. It requires more pointers, which occupies more memory space.
2. Search takes time. Since it takes time to evaluate Hash Function and also to traverse the List

### **Open Addressing**

- Closed Hashing
- Collision resolution technique
- Uses  $H_i(X) = (\text{Hash}(X) + F(i)) \bmod \text{TableSize}$
- When collision occurs, alternative cells are tried until empty cells are found.
- Types:-
  - Linear Probing
  - Quadratic Probing
  - Double Hashing
- Hash function
  - $H(\text{key}) = \text{key \% table size.}$
- Insert Operation
  - To insert a key; Use the hash function to identify the list to which the element should be inserted.
  - Then traverse the list to check whether the element is already present.
  - If exists, increment the count.
  - Else the new element is placed at the front of the list.

#### **Linear Probing:**

Easiest method to handle collision.

Apply the hash function  $H(\text{key}) = \text{key \% table size}$

$H_i(X) = (\text{Hash}(X) + F(i)) \bmod \text{TableSize}$ , where  $F(i)=i$ .

#### **How to Probing:**

- first probe – given a key k, hash to  $H(\text{key})$
- second probe – if  $H(\text{key})+f(1)$  is occupied, try  $H(\text{key})+f(2)$
- And so forth.

#### **Probing Properties:**

- We force  $f(0)=0$
- The  $i^{\text{th}}$  probe is to  $(H(\text{key}) + f(i)) \% \text{table size.}$
- If i reach size-1, the probe has failed.
- Depending on  $f(i)$ , the probe may fail sooner.
- Long sequences of probe are costly.

#### **Probe Sequence is:**

- $H(\text{key}) \% \text{table size}$
- $H(\text{key})+1 \% \text{Table size}$
- $H(\text{Key})+2 \% \text{Table size}$

**1. H(Key)=Key mod Tablesiz**

This is the common formula that you should apply for any hashing

If collocation occurs use Formula 2

**2. H(Key)=(H(key)+i) Tablesiz**

Where i=1, 2, 3, ..... etc

Example: - 89 18 49 58 69; Tablesiz=10

$$1. \quad H(89) = 89 \% 10$$

$$= 9$$

$$2. \quad H(18) = 18 \% 10$$

$$= 8$$

$$3. \quad H(49) = 49 \% 10$$

= 9 ((colloids with 89. So try for next free cell using formula 2))

$$\boxed{i=1} \quad h1(49) = (H(49)+1) \% 10$$

$$= (9+1) \% 10$$

$$= 10 \% 10$$

$$= 0$$

$$4. \quad H(58) = 58 \% 10$$

= 8 ((colloids with 18))

$$\boxed{i=1} \quad h1(58) = (H(58) + 1) \% 10$$

$$= (8+1) \% 10$$

$$= 9 \% 10$$

= 9 => Again collision

$$i=2 \quad h2(58) = (H(58)+2) \% 10$$

$$= (8+2) \% 10$$

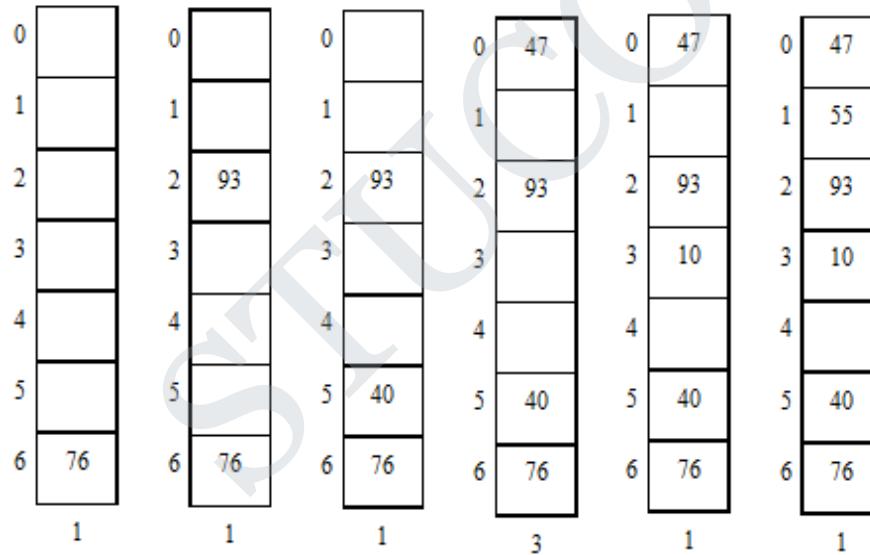
$$= 10 \% 10$$

= 0 => Again collision

	EMPTY	89	18	49	58	69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	
9		89	89	89	89	

### Linear probing

insert(76)      Insert(93)      Insert(40)      Insert(47)      Insert(10)      Insert(55)  
 $76 \% 7 = 6$        $93 \% 7 = 2$        $40 \% 7 = 5$        $47 \% 7 = 5$        $10 \% 7 = 3$        $55 \% 7 = 6$



### Quadratic Probing

To resolve the primary clustering problem, quadratic probing can be used. With quadratic probing, rather than always moving one spot, move  $i^2$  spots from the point of collision, where  $i$  is the number of attempts to resolve the collision.

- Another collision resolution method which distributes items more evenly.

- From the original index H, if the slot is filled, try cells H+12, H+22, H+32,..., H + i2 with wrap-around.
- $H_i(X) = (\text{Hash}(X) + F(i)) \bmod \text{Tablesize}$ ,  $F(i) = i^2$
- $H_i(X) = (\text{Hash}(X) + i^2) \bmod \text{Tablesize}$

Insert  
18, 89, 21      Insert  
58

0	
1	21
2	
3	
4	
5	
6	
7	
8	18
9	89

Insert  
68

21
58
68
18
89

For 58:

- $H = \text{hash}(58, 10) = 8$
- Probe sequence:  
 $i = 0, (8+0) \% 10 = 8$   
 $i = 1, (8+1) \% 10 = 9$   
 $i = 2, (8+4) \% 10 = 2$

For 68:

- $H = \text{hash}(68, 10) = 8$
- Probe sequence:  
 $i = 0, (8+0) \% 10 = 8$   
 $i = 1, (8+1) \% 10 = 9$   
 $i = 2, (8+4) \% 10 = 2$   
 $i = 3, (8+9) \% 10 = 7$

**Limitation:** at most half of the table can be used as alternative locations to resolve collisions.

This means that once the table is more than half full, it's difficult to find an empty spot. This new problem is known as secondary clustering because elements that hash to the same hash key will always probe the same alternative cells.

### Double Hashing

Double hashing uses the idea of applying a second hash function to the key when a collision occurs. The result of the second hash function will be the number of positions forms the point of collision to insert.

There are a couple of requirements for the second function:

It must never evaluate to 0 must make sure that all cells can be probed.

$$H_i(X) = (\text{Hash}(X) + i * \text{Hash}_2(X)) \bmod \text{Tablesize}$$

A popular second hash function is:

$\text{Hash}_2(\text{key}) = R - (\text{key} \% R)$  where R is a prime number that is smaller than the size of the table.

**Table Size = 10 elements**

**Hash<sub>1</sub>(key) = key % 10**

**Hash<sub>2</sub>(key) = 7 – (k % 7)**

**Insert keys : 89, 18, 49, 58, 69**

$$\text{Hash}(89) = 89 \% 10 = 9$$

[0]	49
[1]	
[2]	
[3]	69
[4]	
[5]	
[6]	
[7]	58
[8]	18
[9]	89

$$\text{Hash}(18) = 18 \% 10 = 8$$

$$\text{Hash}(49) = 49 \% 10 = 9 \text{ a collision !}$$

$$= 7 - (49 \% 7)$$

= 7 positions from [9]

$$\text{Hash}(58) = 58 \% 10 = 8$$

$$= 7 - (58 \% 7)$$

= 5 positions from [8]

$$\text{Hash}(69) = 69 \% 10 = 9$$

$$= 7 - (69 \% 7)$$

= 1 position from [9]

### Rehashing

Once the hash table gets too full, the running time for operations will start to take too long and may fail. To solve this problem, a table at least twice the size of the original will be built and the elements will be transferred to the new table.

### Advantage:

- A programmer doesn't worry about table system.
- Simple to implement
- Can be used in other data structure as well

### The new size of the hash table:

- should also be prime
- will be used to calculate the new insertion spot (hence the name rehashing)
- This is a very expensive operation! O(N) since there are N elements to rehash and the table size is roughly 2N. This is ok though since it doesn't happen that often.

### The question becomes when should the rehashing be applied?

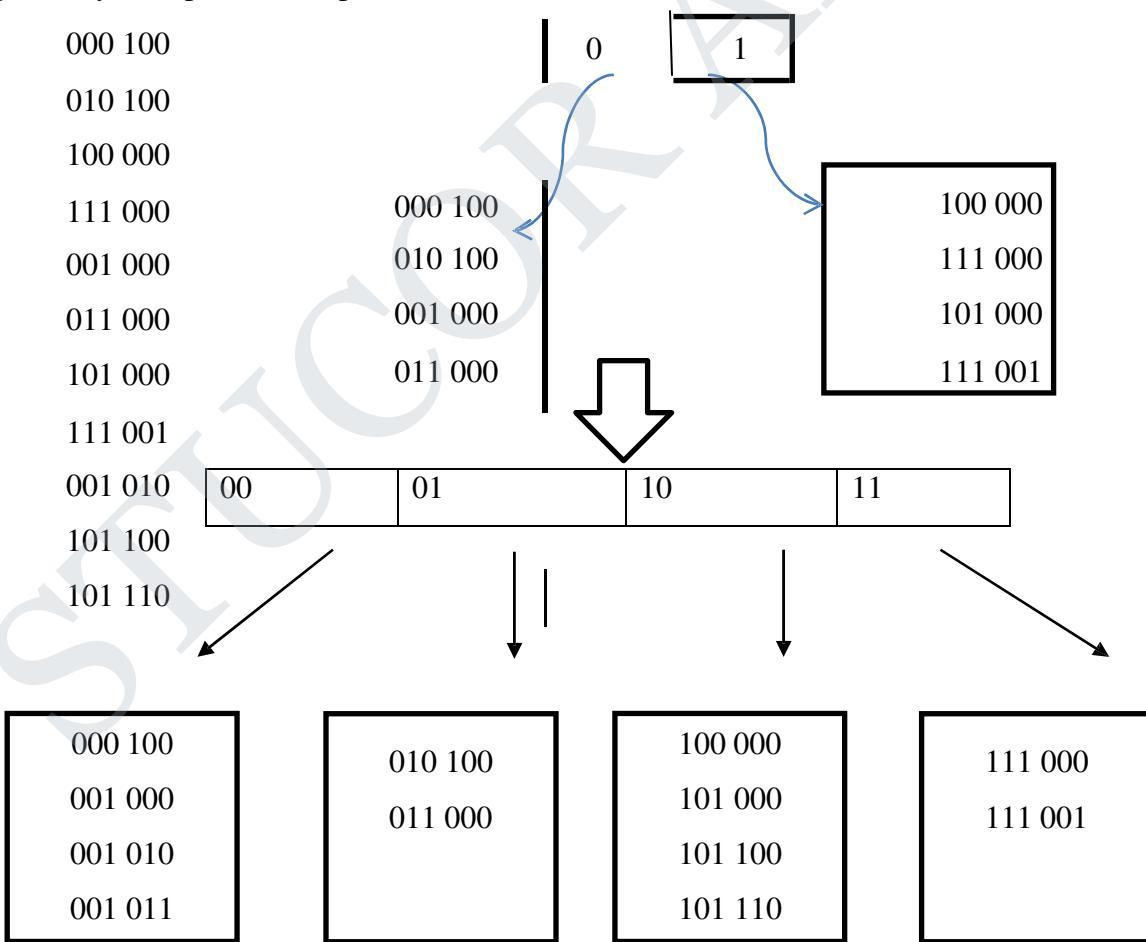
Some possible answers:

- once the table becomes half full
- once an insertion fails

- once a specific load factor has been reached, where load factor is the ratio of the number of elements in the hash table to the table size

### Extendible Hashing

- Extendible Hashing is a mechanism for altering the size of the hash table to accommodate new entries when buckets overflow.
- Common strategy in internal hashing is to double the hash table and rehash each entry. However, this technique is slow, because writing all pages to disk is too expensive.
- Therefore, instead of doubling the whole hash table, we use a directory of pointers to buckets, and double the number of buckets by doubling the directory, splitting just the bucket that overflows.
- Since the directory is much smaller than the file, doubling it is much cheaper. Only one page of keys and pointers is split.



## UNIT - 1

### PART – A

1. Should arrays or linked lists be used for the following types of applications: - Justify your answer.

- Many search operations in sorted list

In this case, using arrays will save time because there will be fewer comparisons. Since the list is sorted the key value can be compared with the middle element of the array and if the key < array element, then left part of array should be searched; if key > array, right part of array should be searched. Best way is to implement a binary search algorithm.

- Many search operation in unsorted list.

In this case array should be used because the elements of an array will be stored in consecutive memory locations whereas in the linked list the elements can be stored in any location and each node has to hold the address of the next element.

2. What is the advantage of an ADT?

- It can be reused in future programs.
- It reduces coding efforts.
- It ensured a robust data structure.
- Debugging is easier.
- Implementing of ADTs can be changed without requiring changes to the program that uses the ADTs.

3. What are Abstract Data Types?

The abstract data type is a triple of D-set of Domains, F- set of Functions, A- Axioms in which only what is to be done is mentioned but how is to be done is not mentioned.

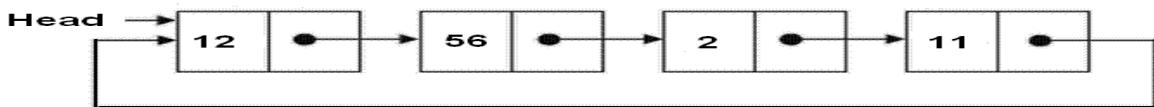
In short, all the implementation details are hidden.

ADT = Type + Function name + Behavior of each function.

4. What is circular linked list?

The circular linked list (CLL) is similar to singly linked list except that the last node's next pointer points to first node.

The list will be accessed like a chain. Circular linked list can be used to help the traverse the same list again and again if needed.



5. What is linked list? Give its types.

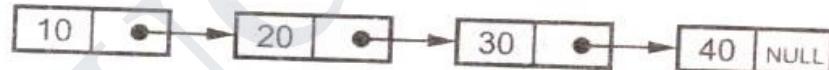
A linked list is a set of nodes where each node has two fields 'data' and a 'link'. Where 'data' filed stores the actual piece of information and 'link' field is used to point to next node. Basically link filed is nothing but the address node.

Linked list is a kind of series of data structures, which are not necessarily adjacent in memory. Each structure contains the element and a pointer to a record containing its successor.

Structure :

Data	Link
------	------

Example:

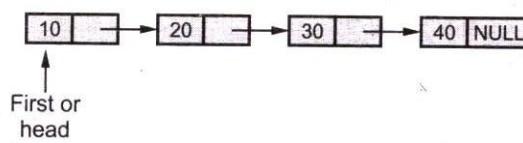


Types:

1. Singly linked list
2. Doubly linked list
3. Singly circularly linked list
4. Doubly circularly linked list

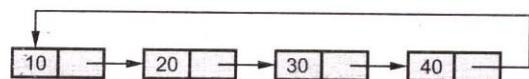
6. Define singly linear linked list.

This list consists of only one link, to point to next node or element. This is also called linear list because the last element points to nothing it is linear in nature. The last field of last node is NULL which means that there is no further list. The very first node is called head or first.



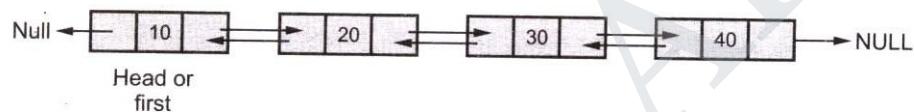
7. Define singly circular linked list.

In the singly circular linked list only one link is used to point to next element. The last node's link filed points to the first or head node.



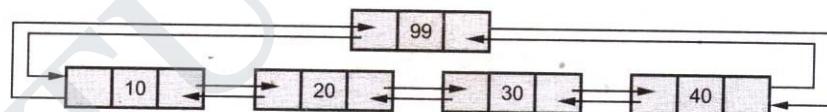
8. Define doubly linear linked list.

In this linked list each node has two pointers previous and next pointers. The previous pointer points to previous node and next pointer points to next node. Only in case of head node the previous pointer is obviously NULL and last node's next pointers points to NULL. This list is a linear one.



9. Define doubly circular linked list.

In circular doubly linked list the previous pointer of first node and the next pointer of last nodes are pointed to head node. Head node is a special node which may have any dummy data or it may have some useful information. Such as total number of nodes in the list which may be used to simplify the algorithms carrying various operations on the list.



10. List the operations of linked list.

- Creation
- Display
- Insertion
- Deletion
- Searching

11. What are the applications of linked list?

- It is used for performing polynomial operations such as addition, multiplication evaluation and so on.
- It is used for handling the set operations.
- The stack data structure can be implemented using linked list.
- The queue data structure can be implemented using linked list.

12. Define data structure.

The data structure can be defined as the collection of elements and all the possible operations which are required for those set of elements.

A data structure is a set of domains D, a set of Function F and set of axioms A. This triple (D, F, A) denotes the data structure d.

13. List the ADT operations.

- a. Create - This operation creates the database.
- b. Display - This operation is for displaying all the elements of the data structure.
- c. Insertion - By this operation the element can be inserted at any desired position.
- d. Deletion - By this operation any desired element can be deleted from the data structure.
- e. Modification - this operation modifies the desired element's value by any other desired new value.

14. Write the ADT for Set.

AbstractDataType SET

{

Instance: Set is a collection of integer type of elements.

Preconditions: None.

Operations:

1. Store ( ): This operation is for storing the integer element in a set.
2. Retrieve ( ): This operation is for retrieving the desired element from the given set.
3. Display ( ): This operation is for displaying the contents of set.

}

15. Define list. How it is implemented?

- List is a collection of elements in sequential order.
- In memory we can store the list in two ways; one way is we can store the elements in sequential memory locations. This is known as arrays and the other way is we can use pointer or links to associate the elements sequentially.
- Implementation ways
  - Array based implementation
  - Linked list based implementation

16. What is static and dynamic memory management?

The static memory management means allocating or de-allocating of memory at compilation time.

The dynamic memory management means allocating or de-allocating of memory at running time (after compilation).

17. Define polynomial. How it is represented using linked list?

A polynomial is homogeneous ordered list of pairs <exponent, coefficient>, where each coefficient is unique.

Example:

$$3x^2 + 5x + 7$$

#### Linked list representation

The main fields of polynomial are coefficient and exponent, in linked list it will have one more field called ‘link’ field to point to next term in the polynomial. If there are ‘n’ terms in the polynomial then ‘n’ such nodes have to be created.

Nodes of polynomial:

Example:

18. What is the advantage of linked list over arrays?

- The linked list makes use of the dynamic memory allocation. Hence the user can allocate or de-allocate the memory as per his requirements.
- On the other hand, the array makes use of the static memory location. Hence there are chances of wastage of the memory or shortage of memory.

19. Compare linked list over arrays.

S.No	Linked list	Arrays
1.	The linked list is a collection of nodes and each node is having one data field and next link field.	The array is a collection of similar types of data elements. In array the data is always stored at some index of the array.
2.	Any element can be accessed by sequential access only.	Any element can be accessed randomly i.e. with the help of index of the array.
3.	Physically the data can be deleted.	Only logical deletion of the data is possible.
4.	Insertions and deletion of data is easy.	Insertions and deletion of data is difficult.
5.	Memory allocation is dynamic. Hence developer can allocate as well as deallocate the memory and so no wastage of memory is there.	The memory allocation is static. Hence once the fixed amount of size is declared then that much memory is allocated. Therefore there is a chance of either memory wastage or memory shortage,

20. What are the advantages of linked list implementation of list? List the limitation in array based implementation of list ADT.

- Linked list facilities dynamic memory management by allowing elements to be added or deleted at any time during program execution.
- It ensures the efficient utilization of memory space as only that much amount of memory space is reserved as is required for storing the list of elements.
- It is easy to insert or delete elements in a linked list, unlike arrays, which require shuffling of other elements with each insert and delete operation.

#### Limitation of array implementation

- Insertion and deletion operation are expensive as it requires more data movement.
- Find and print list operation takes constant time.
- Even if the array is dynamically allocated, an estimate of the maximum size of the list is required which considerably wastes the memory space.

**QUESTION**  
**UNIT - 4**

1. Write an algorithm to perform insertion and deletion on a doubly linked list. Give the relevant coding in C. (8)
2. Consider an array A [1, 2....n]. Given a position, write an algorithm to insert an element in the array. If the position is empty, the element is inserted easily. If the position is already occupied, the element should be inserted with minimum number of shifts. (Note: The elements can be shifted to left or right to make minimum number of moves). (8)
3. Explain the following:
  - a. Application of lists.
  - b. Polynomial manipulation(8)
4. Define singly linked list. Write the routine for its operations.
5. Define doubly linked list. Write the routine for its operations.
6. Define circular linked list. Write the routine for its operations.
7. Define circularly doubly linked list. Write the routine for its operations.

**UNIT -****PART – A**

1. Define double ended queue (dqueue).

In the double ended queue we can make use of both the ends for insertion of the elements as well as we can use both the ends for deletion of the elements. That means it is possible to insert the elements by rear as well as by front. Similarly it is possible to delete the elements from front as well as from rear.

**Types:**

- i. Input restricted queue

In this type insertion is allowed at one end and deletion is allowed at both ends.

- ii. Output restricted queue

In this type deletion is allowed at one end and insertion is allowed at both ends.

2. List the applications of queue.

- Job scheduling
- Categorizing data
- Simulation and modeling
- Time sharing systems
- Mathematics user queuing theory
- Computer networks
- In implanting depth first search and breadth first search.

3. Give the applications of stack.

- Expression conversion
- Expression evaluation
- Parsing well formed parenthesis
- Reversing a string
- Storing function calls
- Page visited history in a web browser.
- Undo sequence in a text editor.

4. Define stack.

A stack is an ordered list in which all insertions and deletions are made at one end called the top.

The stack is also called as LIFO i.e. Last In First Out data structure.

If we have to make stack of elements 10, 20, 30, 40, 50, 60 then 10 will be the bottom most element and 60 will be the top most element in the stack.

5. Define stack full and stack empty.

Stack full

Stfull () – This condition indicates whether the stack is full or not. if the stack is full then we cannot insert the elements in the stack. Before performing push we must check stfull () condition.

```
int stfull ()
{
    if (st.top==size-1)
        return 1;
    else
        return 0;
}
```

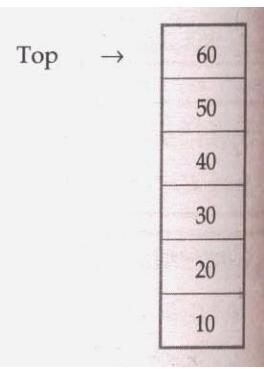
Thus stfull is a Boolean function. If stack is full it returns 1 otherwise it returns 0.

Stack empty

This condition indicates whether the stack is empty or not. if the stack is empty then we cannot pop or remove an y element from the stack.

Before popping the elements from the stack we should check stempty() condition.

```
int stempty ()
{
    if (st.top===-1)
        return 1;
    else
        return 0;
}
```



6. List the operations of stack.

a. Push

By this operation one can push elements onto the stack. Before performing push we must check stfull( ) condition.

```
void push(int item)
{
    st.top++;
    st.s[st.top]=item;
}
```

b. Pop

By this operation one can remove the elements from stack. Before popping the elements from the stack we should check stempty( ) condition.

```
void pop()
{
    int item;
    item = st.s[st.top];
    st.top --;
    return(item);
}
```

7. Define expression. Give its types.

Expression is a string of operands and operators. Operands are some numeric values and operators are of two types: Unary operator and Binary operator. Unary operators are ‘+’ and ‘-’ and binary operators are ‘+’, ‘-’, ‘\*’, ‘/’ and exponential.

Types:

- Infix expression
- Postfix expression
- Prefix expression

8. Define infix, postfix, and prefix expression with example.

• Infix expression

In this type of expression the arrangement of operands and operator is as follows.

Infix expression = operand 1 operator operand 2

Example: A + B

- Postfix expression

In this type of expression the arrangement of operands and operator is as follows.

postfix expression = operand 1 operand 2 operator

Example: AB+

- Prefix expression

In this type of expression the arrangement of operands and operator is as follows.

prefix expression = operator operand 1 operand 2

Example: +AB

9. Define queue. Give example.

The queue can be formally defined as ordered collection of elements that has two ends named as front and rear. From the front end one can deleted the elements and from rear end one can insert the elements.

The queue is also called as FIFO i.e. a First In First Out data structure.

Example:

A queue of people who can wait for a city bus at the bus stop. Any new person is joining at one end of the queue, you can call it as the rear end. When the bus arrives the person at the other end first enters in the bus, you can call it as the front end.

Representation of queue:

```
struct queue
{
    int que[size], front, rear;
} Q;
```

10. List the operation on queue.

- Queue overflow
- Insertion
- Queue underflow
- Deletion
- Display

11. Define queue overflow and queue underflow.

Queue overflow:

The condition resulting from trying to add an element on to a full queue.

Queue underflow:

The condition resulting from trying to remove an element from an empty queue.

12. Write about push and pop operation on queues.

Push:

The insertion of the element in the queue is done by the end called rear. Before the insertion of the element in the queue it is checked whether or not the queue is full.

Pop:

The deletion of the element from the queue is done by the end called front. Before performing the delete operation it is checked whether the queue is empty or not.

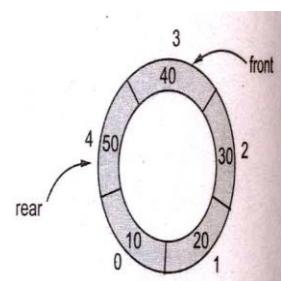
13. What is circular queue? What is its advantage?

Circular queue is a queue in which the front end is just adjacent to the rear end. The elements got arranged in circular manner.

The formula which has to be applied for setting the front and rear pointers:

$$\text{rear} = (\text{rear} + 1) \% \text{ size}$$

$$\text{front} = (\text{front} + 1) \% \text{ size}$$



Advantages

The main advantage of circular queue is we can utilize the space of the queue fully.

14. What are the rules of towers of Honai?

- Only one disk could be moved at a time.
- No longer could disk over reside on a pillar on top of a smaller disk.
- A 3<sup>rd</sup> pillar could be used as an intermediate to store one or more disks, while they were being moved from source to destination.

15. What is priority queue?

It is a type of queue in which each element is assigned certain priority such that the order of deletion of elements is decided by their associated priorities.

16. What are the postfix and prefix forms of the expression?  $A+B*(C-D)/(P-R)$

POSTFIF FORM : ABCD-\*PR-/+

PREFIX FORM : +A/\*B-CD-PR

17. What are the types of queues?

- Simple queues.
- Circular queue.
- Double ended queue.
- Priority queue.

18. Mention the advantages of representing stacks using linked list than array.

- It is easier to store data of different sizes in a linked list. An array assumes every element is exactly the same size.
- It is easier for a linked list to grow organically. An array's size needs to be known ahead of time, or re-created when it needs to grow.
- Shuffling a linked list is just a matter of changing what points to what. Shuffling an array is more complicated and/or takes more memory.
- As long as your iterations all happen in a “foreach” context, you don't lose any performance in iteration.
- Adding a new element somewhere in the middle, anywhere the item can be inserted.
- In linked list implementation of list memory are not fixed and it can be allocated by dynamic memory allocation scheme during the runtime of the program. So there is no wastage of memory.

19. What is the data structure used to perform recursion? How?

Stack. Because of its LIFO property it remembers its ‘caller’ so knows whom to return when the function has to return. Recursion makes use of system stack for storing the return addresses of the function calls.

Every recursive function has its equivalent iterative (non-recursive) function.

Even when such equivalent iterative procedures are written, explicit stack is to be used.

20. What do you understand by polish notation? Explain.

This is also called as prefix notation. In this type of notation the operator followed by two operands. For example if  $(a+b)*c$  is a given expression then its polish notation will be  $*+abc$ .

**UNIT - 5**

1. Write an algorithm to convert infix to postfix. Trace the algorithm to convert  $(a+b)*c/d+e/f$  to postfix. Explain the need for infix and postfix expressions.
2. Write an algorithm to perform the four operations in double ended queue that is implemented as array.
3. Discuss about stack ADT in detail. Explain any one application of stack.
4. Explain queue ADT in detail. Explain any one application of queue with a suitable example.
5. What is the stack ADT? Give array implementation of stack.
6. Explain about circular queue with example.
7. What is queue ADT? Give linked implementation of queue.

**UNIT - 3****PART – A**

1. What is the time complexity of binary search?

Worst case :  $O(\log N)$

Best case :  $O(1)$

Average case :  $O(1 \log N)$

2. List the sorting algorithm which uses logarithmic time complexity.

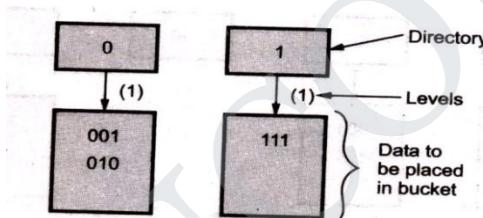
Quick sort, merge sort and heap sort has logarithmic time complexity

3. Define extendible hashing.

Extendible hashing is a technique which handles a large amount of data. The data to be placed in the hash table is by extracting certain number of bits.

In extendible hashing referring the size of directory the elements are to be placed in buckets. The levels are indicated in parenthesis.

Example:



4. Give the types of sorting. Differentiate it.

Types:

1. Internal sorting
2. External sorting

<b>Internal Sorting</b>	<b>External Sorting</b>
<p>This is a type of sorting technique in which data resided on main memory of computer</p> <p><u>Example:</u></p> <p>Insertion Sort, Selection sort, shell sort, bubble sort, heap sort, quick sort, etc.</p>	<p>This is a sorting technique in which there is a huge amount of data and it resides on secondary devices while sorting.</p> <p><u>Example:</u></p> <p>Merge Sort, Multiway merge, Polyphase merge.</p>

5. What is meant by sorting and searching?

**Sorting:**

Sorting is a technique for arranging data in particular order. Order means the arrangement of data. The sorting order can be ascending or descending. The ascending order means arranging the data in increasing order whereas descending order means arranging the data in decreasing order.

**Searching:**

Searching is the technique for finding particular element from the set.

6. What are the types of searching? Give its difference.

**Types:**

- a. Linear Search
- b. Binary Search

Sr.No.	Linear Search	Binary Search
1.	For searching the element by using linear search method it is not required to arrange the elements in some specific order	The elements need to be arranged either in ascending or descending order.
2.	Each and every element is compared with the key element from the beginning of the list.	The list is subdivided into two sublists. The key element is searched in the sublist.
3.	Less efficient method.	Efficient method.
4.	Simple to implement.	Additional computation is required for computing mid element.

7. What are the types of sorting available in C?

- Insertion sort
- Merge sort
- Quick sort
- Radix sort
- Heap sort
- Selection sort
- Bubble sort.

8. Define bubble sort.

Bubble sort is the one of the easiest sorting method. In this method each fate item is compared with its neighbor and if it is a descending order sorting, then the bigger element is moved to the top of all.

The smaller numbers are slowly moved to the bottom position. Hence it is also called as the exchange sort.

9. Define merge sort.

Merge sort is a sorting algorithm in which array is divided repeatedly. The sub arrays are sorted independently and then these sub-arrays are combined together to form a final sorted list.

Merge sort on an input array with 'n' elements consists of three steps:

- **Divide:** Partition array into two sub lists s1 and s2 with  $n/2$  elements each.
- **Conquer:** Then sort sub list s1 and sub list s2.
- **Combine:** Merge s1 and s2 into a unique sorted group.

10. Define insertion sort. Give its advantages.

Successive element in the array to be sorted and inserted into its proper place with respect to the other already sorted element.

We start with the second element and put it in its correct place, so that the first and second elements of the array are in order.

Advantages:

- Simple to implement
- This method is efficient when we want to sort small number of elements and this method has excellent performance on almost sorted list of elements.
- More efficient than most other simple  $O(n^2)$  algorithms such as selection sort or bubble sort.
- This is a stable.
- It is called in-place sorting algorithm. The in-place sorting algorithm is an algorithm in which the input is overwritten by output and to execute the sorting method it does not require any more additional space.

11. Define selection sort.

Scan the array to find its smallest element and swap it with the first element. Then, starting with the second element scan the entire list to find the smallest element and swap it with the second element. Then starting from the third element the entire list is scanned in order to find the next smallest element. Continuing in this fashion we can sort the entire list.

12. Define shell sort.

In this method the elements at fixed distance are compared. The distance will then be decremented by some fixed amount and again the comparison will be made. Finally, individual elements will be compared.

It uses an increment sequence. The increment size is reduced after each pass until increment size is 1.

13. Define quick sort. Give its advantages.

In this method, the array is split into two sub arrays. This splitting of the array is based on pivot element. All the elements that are less than pivot should be in the left sub-array and all the elements that are greater than the pivot should be in the right sub-array.

Advantages:

It reduces unnecessary swaps and moves an item to a greater distance, in one move.

14. Define radix sort. Give its algorithm.

In radix sort the elements are processed by its individual digits. It processes the digits either by least significant digit (LSD) method or by most significant digit (MSD) method.

Radix sort is a clever and intuitive little sorting algorithm; radix sort puts the elements in order by comparing the digits of the numbers.

15. Explain why binary search cannot be performed using linked list.

In binary search algorithm, the mid element needs to be searched. If binary search is implemented using arrays then by simply saying  $a[mid]$  we can access the middle element of an array in constant time. But for finding the mid element of a linked list we have to execute separate algorithm and it cannot be done in constant time. Thus implementing binary search using linked list is very inefficient way. Hence it is not preferred to implement a binary search using linked list.

16. Define hash table.

Hash table is a data structure used for storing and retrieving data very quickly. Insertion of data in the hash table is based on the key value. Hence every entry in the hash table is associated with some key. For example for storing an employee record in the hash table the employee ID will work as a key.

Using the hash key the required piece of data can be searched in the hash table by few or more key comparisons. The searching time is then dependent upon the size of the hash table.

The effective representation of dictionary can be done using hash table. We can place the dictionary entries in the hash table using hash function.

17. What is hashing? List its techniques.

Hashing is a technique of storing the elements directly at the specific location in the hash table. The hashing makes use of hash function to place the record at its position. Using the same hash function the data can be retrieved directly from the hash table.

Techniques

- Division method
- Mid square method
- Multiplicative hash function.
- Digit folding
- Digit analysis.

18. What is rehashing? Give its advantages.

Rehashing is a technique in which the table is resized. That means the size of the table is doubled by creating a new table. The total size of the table is usually a prime number. Following are the situations in which the rehashing is required-

- When table is completely full.
- With quadratic probing the table is filled half.
- When insertions fail due to overflow.

Advantages:

- This technique provides the programmer a flexibility to enlarge the table size if required.
- Only the space gets doubled with simple hash function which avoids occurrence of collisions.

19. What is collision in hashing?

The situation in which the hash function returns the same hash key for more than one record is called collision.

20. What is overflow in hashing?

The situation in which there is no room for a new pair in the hash table is called overflow.

PART - B

1. Write short note on hashing and various collision resolution techniques.
2. Write an algorithm to sort n numbers using quick sort. Show how the following numbers are sorted using quick sort: 45, 28, 90, 1, 46, 9, 33 and 87
3. What are the different types of hashing techniques? Explain them in detail with examples.
4. Describe equivalence relations. Write down the algorithm for dynamic equivalence problem.
5. Explain open addressing with its probing in detail. Also explain the collision resolution techniques in detail.
6. Explain separate chaining and extendible hashing.
7. Write about different types of hashing techniques in detail.
8. Write down the shell sort algorithm and trace the following numbers. 10, 9, 8, 7, 6, 5, 4, 3
9. Write down the merger sort algorithm and give its analysis.
10. Show how the insertion sort and selection sort processes the input 24, 12, 46, 5, 20, 18, 38, 10.
11. Given input (4371, 1323, 6173, 4199, 4344, 9679, 1989) and a hash function  $H(X) = X \bmod 10$  show the results of following.
  - a. Separate chaining table
  - b. Open addressing hash table using linear probing
  - c. Open addressing hash table using quadratic probing
  - d. Open addressing hash table using double hashing with second hash function  $H_2(X) = 7 - (X \bmod 7)$ .
12. Explain Rehashing and Extendible hashing techniques.

## UNIT -I

### LINEAR STRUCTURES

#### **1. What are the main objectives of Data structure?**

- To identify and create useful mathematical entities and operations to determine what classes of problems can be solved by using these entities and operations.
- To determine the representation of these abstract entities and to implement the abstract operations on these concrete representation.

#### **2. Need of a Data structure?**

- To understand the relationship of one data elements with the other and organize it within the memory.
- A data structures helps to analyze the data, store it and organize it in a logical or mathematical manner.

#### **3. Define data structure with example.**

A data structure is a way of organizing data that considers not only the items stored but also their relationship to each other.

e.g.: Arrays, Records etc.

e.g of complex data structure

Stacks, Queues, Linked list, Trees, Graphs.

#### **4. Define abstract data type and list its advantages.**

ADT is a set of operations.

An abstract data type is a data declaration packaged together with the operations that are meaningful on the data type.

#### Abstract Data Type

1. Declaration of data
2. Declaration of operations.

Advantages:

1. Easy to debug small routines than large ones.
2. Easy for several people to work on a modular program simultaneously.
3. A modular program places certain dependencies in only one routine, making changes easier.

#### 5. What are the two basic types of Data structures?

1. Primitive Data structure

Eg., int,char,float

2. Non Primitive Data Structure

- i. Linear Data structure

Eg., Lists Stacks Queues

- ii. Non linear Data structure

Eg., Trees Graphs

#### 6. What are the four basic Operations of Data structures?

1. Traversing
2. Searching
3. Inserting
4. Deleting

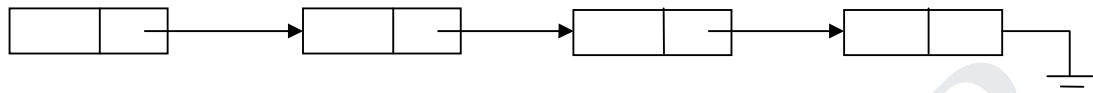
#### 7. List out the different ways to implement the list?

1. Array Based Implementation
2. Linked list Implementation
  - i. Singly linked list
  - ii. Doubly linked list
  - iii. Cursor based linked list

**8. Define singly linked list with neat diagram.**

A singly linked list is a collection of nodes each node is a structure it consisting of an element and a pointer to a structure containing its successor, which is called a next pointer.

The last cell's next pointer points to NULL specified by zero.

**9. List the operations of single linked list.**

MakeEmpty

IsEmpty

IsLast

Find

Delete

FindPrevious

Insert

DeleteList

Header

First

Advance

Retrieve

**10. Write the routine for insertion operation of singly linked list.**

```

/* Insert (after legal position P)*/

/* Header implementation assumed*/

/* Parameter L is unused in this implementation*/
  
```

Void

Insert (ElementType X, List L, Position P)

{

Position TmpCell;

```
TmpCell=malloc(sizeof(struct Node));  
if(TmpCell==NULL)  
    FatalError("Out of space!!!");  
TmpCell->Element =X;  
TmpCell->Next=P->Next;  
P->Next=TmpCell;  
}
```

**11. Write the routine for deletion operation of singly linked list.**

```
/* Delete first occurrence of x from a list*/  
/* Assume use of a header node*/  
Void  
Delete(ElementType X, List L)  
{  
    Position P, TmpCell;  
    P=FindPrevious(X, L);  
    if (! IsLast(P,L))  
    {  
        TmpCell=P->Next;  
        P->Next=TmpCell->Next;  
        Free(TmpCell);  
    }  
}
```

**12. List out the advantages and disadvantages of singly linked list.**

Advantages:

1. Easy insertion and deletion.
2. Less time consumption.

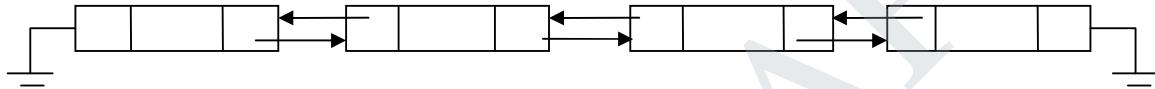
Disadvantages:

1. Data not present in a linear manner.
2. Insertion and deletion from the front of the list is difficult without the use of header node.

### 13. Define doubly linked list with neat diagram.

Doubly linked list is a collection of nodes where each node is a structure containing the following fields

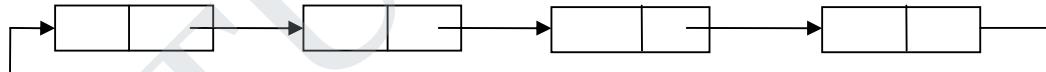
1. Pointer to the previous node.
2. Data.
3. Pointer to the next node.



### 14. Define circularly linked list with neat diagram.

Circularly linked list is a collection of nodes , where each node is a structure containing the element and a pointer to a structure containing its successor.

The pointer field of the last node points to the address of the first node. Thus the linked list becomes circular.



### 15. Write the difference between Singly and doubly linked list.

#### Singly

It is a collection of nodes and each node is having one data field and next link field

#### Doubly

It is a collection of nodes and each node is having one data field one previous link field and one next link field

The elements can be accessed

The elements can be accessed using

using next link	both previous link as well as next link
No extra field is required hence,	One field is required to store previous
Node takes less memory in SLL.	Link Hence, node takes memory in DLL.
Less efficient access to elements	More efficient access to elements.

**16. Write the difference between doubly and circularly linked list.**

Doubly	Circularly
If the pointer to next node is Null, it specifies the last node.	There is no first and last node.
Last node's next field is always Null	Last nodes next field points to the address of the first node.
Every node has three fields one is Pointer to the pervious node, next Is data and the third is pointer to the next node.	it can be a singly linked list and doubly linked list.

**17. Write the difference between cursor and pointer implementation of singly linked list.**

CURSOR IMPLEMENTATION	POINTER IMPLEMENTATION
Global array is maintained for storing the data elements .The corresponding index value for each data item represents its address.	The data are stored in a collection of structures. Each structure contains data and a pointer to the next structure.
Collection of empty cells is maintained in the form of a array from which cells	A new structure can be obtained from the systems global memory by a call to

can be allocated and also cells can be returned after use.

a *malloc()* and released by a call to *free()*.

### **18. Define stack ADT with example.**

A stack is a list with a restriction that insertions and deletions can be performed in only one position namely the end of the list called the top.

e.g.: undo statement in text editor.

Pile of bills in a hotel.

### **19. State the operations on stack. Define them and give the diagrammatic representation.**

Push

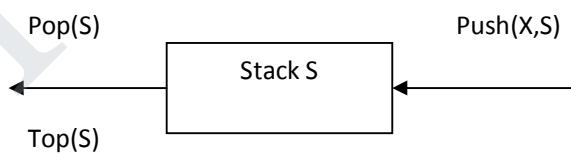
Pop

Top

Push: Push is performed by inserting at the top of stack.

Pop: pop deletes the most recently inserted element.

Top: top operation examines the element at the top of the stack and returns its value.



### **20. Write the routine for push and pop of linked list.**

/\*routine for push\*/

Void

Push(ElementType X,Stack S)

```

{

    PtrToNode TmpCell;

    TmpCell=malloc(sizeof(struct Node));

    If(TmpCell==NULL)

        FatalError("out of space!!!");

    else

    {

        TmpCell->Element=x;

        TmpCell->Next=s->Next;

        S->Next=TmpCell;

    }

}

```

/\*Routine for pop\*/

```

Void

Pop(stack S)

{

    PtrToNode FirstCell;

    If(IsEmpty(S))

        Error("Empty stack");

    Else

    {

        FirstCell=S->Next;

        S->Next=S->Next->Next;

        free(firstCell);

    }

}

```

**21. What is the purpose of top and pop?**

Top operation examines the element in the top of the list and returns its value. Pop operation deletes the element at the top of the stack and decrements the top of the stack pointer by one.

**22. State the disadvantages of linked list implementation of stack.**

1. Calls to malloc and free functions are expensive.
2. Using pointers is expensive.

**23. State the applications of stack.**

1. Balancing parentheses.
2. Postfix Expression.
  - i. Infix to postfix conversion
3. Function calls.

**24. Write the algorithm for balancing symbols.**

1. Make an empty stack.
2. Read characters until end of file.
3. If the character is an opening symbol, then push it onto the stack.
4. If it is a closing symbol

Then

If the stack is empty

Report an error

Otherwise pop the stack

5. If the symbol popped is not the corresponding opening symbol

Then

Report an error

6. If the stack is not empty at the end of file

Then

Report an error

**25. Give the Features of balancing symbols.**

1. It is clearly linear.
2. Makes only one pass through the input.
3. It is online and quite fast.
4. It must be decided what to do when an error is reported.

**26. Convert the given infix to postfix.**

$(j*k)+(x+y)$

Ans: $j k^* x y + +$

**27. Convert into postfix and evaluate the following expression.**

$(a+b*c)/d$

a=2 b=4 c=6 d=2

Ans:

Post fix:

$a b c ^ * + d /$

Evaluation:

$2 \ 4 \ 6 \ ^ \ * \ + \ 2 \ /$

=13

**28. Write the features of representing calls in a stack.**

1. When a function is called the register values and return address are saved.
2. After a function has been executed the register values are resumed on returning to the calling statement.
3. The stack is used for resuming the register values and for returning to the calling statement.

4. The stack overflow leads to fatal error causing loss of program and data.
5. Recursive call at the last line of the program is called tail recursion and also leads to error.

**29. Define queue with examples.**

Queue is a list in which insertion is done at one end called the rear and deletion is performed at another called front.

e.g: Ticket counter.

Phone calls waiting in a queue for the operator to receive.

**30. List the operations of queue.**

Two operations

1. Enqueue-inserts an element at the end of the list called the rear.
2. Dequeue-delets and returns the element at the start of the list called as the front.

**31. What are the prerequisites for implementing the queue ADT using array?**

For each queue data structure the following prerequisites must be satisfied

1. Keep an array queue[ ].
2. The positions front and rear represents the ends of the queue.
3. The number of elements that are actually in the queue is kept track of using ‘size’.

**32. How the enqueue and dequeue operations are performed in queue.**

To enqueue an element X:

increment size and rear

set queue[rear]=x

To dequeue an element

set the return value to queue[front].

Decrement size

Increment front

**33. Write the routines for enqueue operation in queue.**

Void

Enqueue(Element type X,Queue Q)

{

If(IsFull(Q))

Error("Full queue");

Else

{

Q->Size++;

Q->Rear=Succ(Q->Rear,Q);

Q->Array[Q->Rear]=X;

}

}

**34. Write the routines for dequeue operation in queue.**

Void Dequeue(Queue Q)

{

If(IsEmpty(Q))

Error("Empty Queue");

Else

Q->front++;

}

**35. List the Applications of queue?**

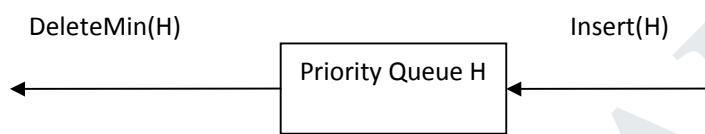
- Graph Algorithm
- Priority Algorithm

- Job scheduling
- Categorizing Data

**36. Define priority queue with diagram and give the operations.**

Priority queue is a data structure that allows at least the following two operations.

1. Insert-inserts an element at the end of the list called the rear.
2. DeleteMin-Finds, returns and removes the minimum element in the priority Queue.



Operations:

Insert  
DeleteMin

**37. Give the applications of priority queues.**

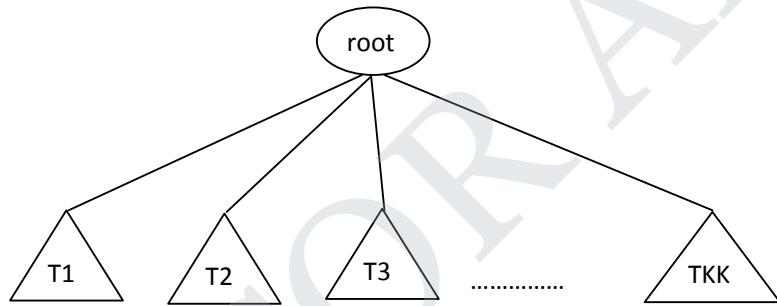
There are three applications of priority queues

1. External sorting.
2. Greedy algorithm implementation.
3. Discrete even simulation.
4. Operating systems.

**UNIT II**  
**TREE STRUCTURES**

1. Define Tree .Give an example.

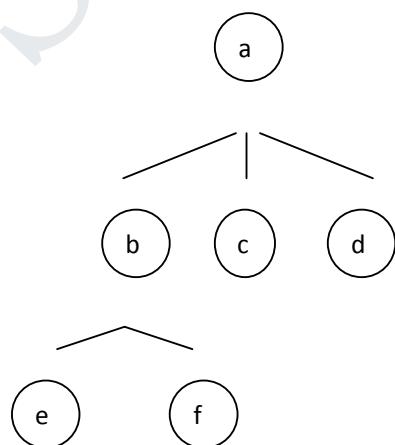
A tree is a collection of nodes .The collection can be empty .Otherwise a tree consists of a distinguished node r called the root and 0 or more non empty sub-trees  $T_1, T_2, T_3, \dots, T_k$  each of whose roots are connected by a directed edge from r.



Eg: directory structure hierarchy

2. Define depth of a node in a tree. Give example.

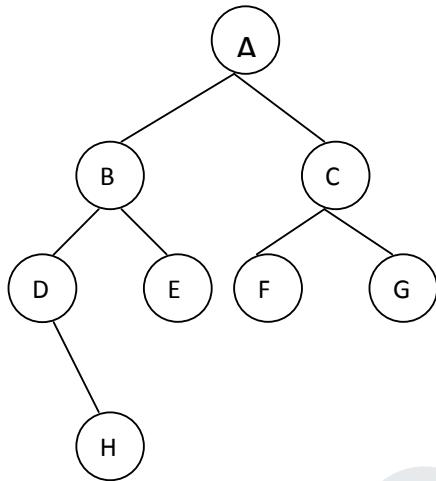
For any node  $n_i$  the depth of  $n_i$  is the length of the unique path from the root to  $n_i$   
eg:



The depth of e is 2.

3. Define length of the path in a tree with an example.

Length of a path is the number of edges on the

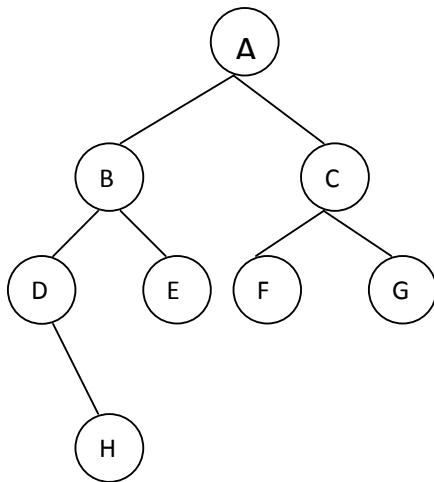


path.

The length of the path from A-H is 3.

4. Define a path in a tree. Give example.

A path from a node  $n_1$  to  $n_k$  is defined as the sequence of nodes  $n_1, n_2, \dots, n_k$  such that  $n_i$  is the parent of  $n_{i+1}$  for  $1 \leq i < k$ .



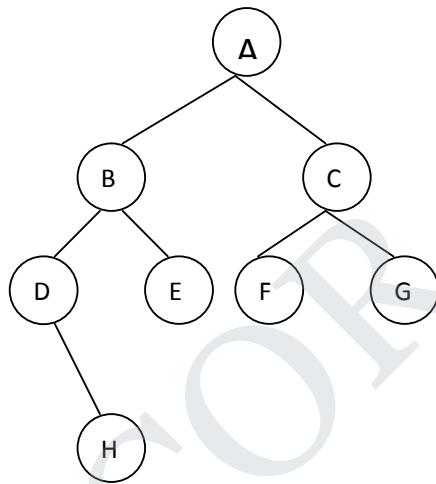
Eg:

The path from A-H is A-B-D-H

5. Define height of the node in a tree. Give example.

The height of node  $n_i$  the length of the longest path from  $n_i$  to a leaf

Eg:



The height of node B is 2.

6. Write the routine for node declaration in trees.

```
typedef struct TreeNode *PtrToNode;  
  
struct TreeNode  
{  
    ElementType Element;  
    PtrToNode FirstChild;  
    PtrToNode NextSibling;  
};
```

7. List the applications of trees.

- Binary search trees
- Expression trees
- Threaded binary trees

8. Define Binary tree.

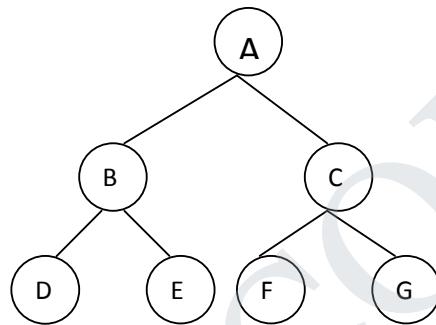
A Binary tree is a tree in which no node can have more than two children.

9. List the tree traversal applications.

1. Listing a directory in an hierachal file system (preorder)
2. Calculating the size of a directory (post order)

10. Define binary tree ADT with an example.

A binary tree is a tree in which no node can have more than two children.



11. Define binary search tree?

Binary Search tree is a binary tree in which each internal node  $x$  stores an element such that the element stored in the left sub tree of  $x$  are less than or equal to  $x$  and elements stored in the right sub tree of  $x$  are greater than or equal to  $x$ . This is called binary-search-tree

12. List the Types of binary search trees

- i) Performance comparisons
- ii) Optimal binary search trees

13. List the Operations of binary search tree?

- Make Empty
- Find
- Insert
- Delete
- Search
- Display

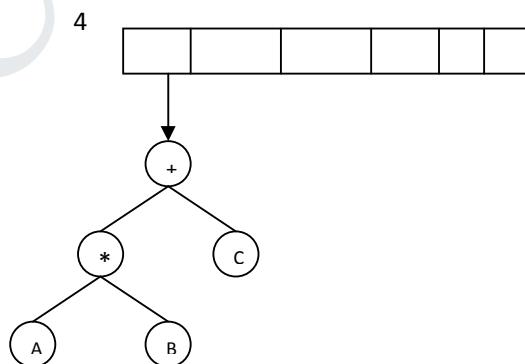
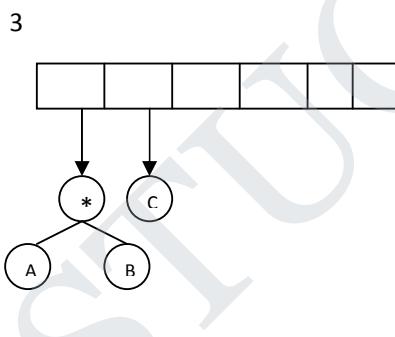
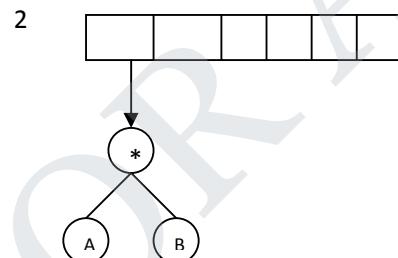
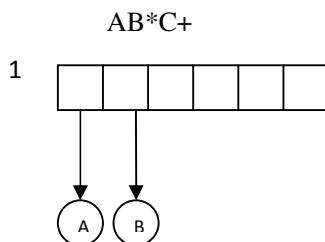
14. Define Threaded Binary tree.

A Threaded Binary Tree is a binary tree in which every node that does not have a right child has a THREAD (in actual sense, a link) to its INORDER successor. By doing this threading we avoid the recursive method of traversing a Tree, which makes use of stacks and consumes a lot of memory and time.

15. List the uses of binary tree.

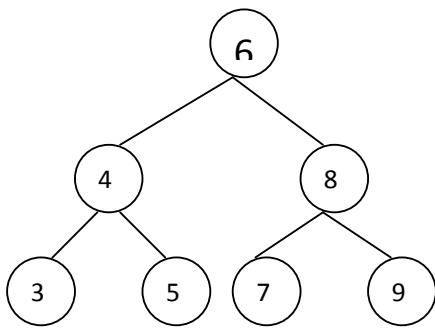
1. Searching.
2. Compiler design.

16. Draw the expression tree for the given postfix expression using stack.



17. Define binary search tree ADT with an example.

A binary search tree is a tree in which for every node X, the values of all the keys in its left sub tree are smaller than the key value in X and the values of all the keys in its right sub tree are larger than the key value in X.



18. How deletion is performed in a binary search tree.

Once the node to be deleted is found there are three possibilities

1. If the node is a leaf, it can be deleted immediately.
2. If the node has one child the node can be deleted after its parent adjusts a pointer to bypass the node.
3. If the node has two children the general strategy is to replace the data of this node with the smallest data of the right sub tree and recursively delete the node which is empty.

19. Define internal path length.

It is the sum of the depths of all nodes in a tree.

20. What is the average depth of all nodes in an equally likely tree?

The average depth of all nodes in an equally likely tree is  $O(\log N)$ .

21. List out the disadvantages of Binary search tree.

1. Deletions in a binary search tree leads to trees which are not equally likely.
2. Absence of balanced search tree.
3. The average depth is not  $O(\log N)$  in trees which are not equally likely.

22. Define tree traversal.

Traveling through all the nodes of the tree in such a way that each node is visited exactly once.

23. List out the types of Tree traversal?

There are three types of tree traversal

1. Preorder traversal
2. Inorder traversal
3. Postorder traversal

24. Write the steps and routine for the postorder traversal.

Steps:

1. Traverse the left sub tree.
2. Traverse the right sub tree.
3. Visit the root.

Routine:

```
void postorder(node *temp)
{
    if(temp!=NULL)
    {
        postorder(temp->left);
        postorder(temp->right);
        printf("%d",temp->data);
    }
}
```

25. Write the steps and routine for the preorder traversal.

Steps:

1. Visit the root
2. Traverse the left sub tree.
3. Traverse the right sub tree.

Routine:

```
void preorder(node *temp)
{
}
```

```
if(temp!=NULL)
{
    printf("%d",temp->data);
    preorder(temp->left);
    preorder(temp->right);

}
}
```

26. Write the steps and routine for the inorder traversal.

Steps:

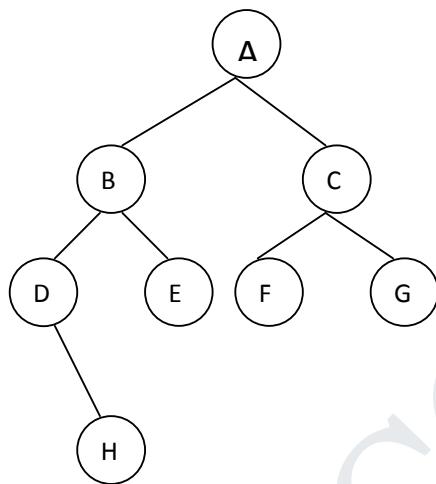
1. Traverse the left sub tree
2. Visit the root
3. Traverse the right sub tree.

Routine:

```
void inorder(node *temp)
{
    if (temp!=NULL)
    {
        inorder(temp->left);
        printf("%d",temp->data);
        inorder(temp->right);

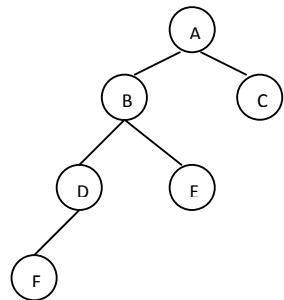
    }
}
```

27. Perform preorder traversal for the given tree.



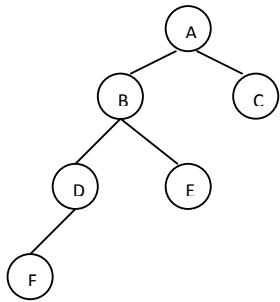
ABDHECFG

28. Perform inorder traversal for the given tree.



FDBEAC

29. Perform postorder traversal for the given tree.



FDEBCA

30. Define the following.

i) Leaf

Nodes at the bottommost level of the tree are called **leaf nodes**

ii) Sibling

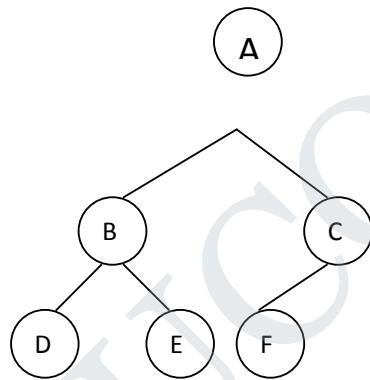
The nodes with common parent are called Sibling

**UNIT III****BALANCED TREE**

1. Define AVL Tree. Give Example.

An AVL Tree is a binary search tree with a balance condition, which is easy to maintain and ensure that the depth of the tree is  $O(\log N)$ . Balance condition require that the left and the right sub trees have the same height.

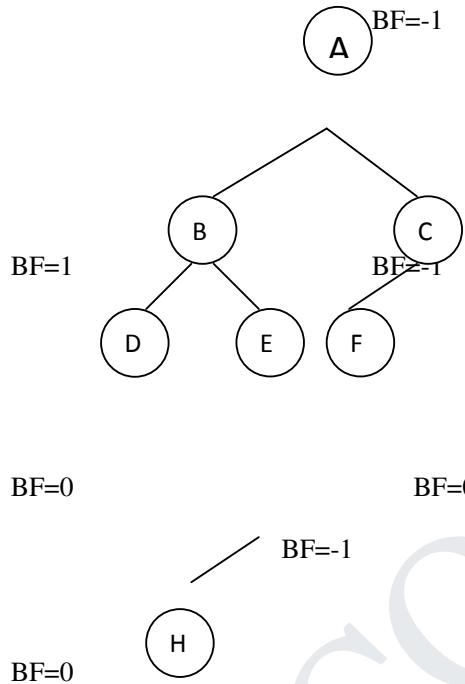
Example:



2. Define Balance factor.

The balance factor of a node in binary tree is defined to be  $h_R - h_L$  where  $h_L$  and  $h_R$  are heights of left and right subtrees of T. For any node in AVL tree the balance factor should be 1,0 or -1.

3. Give the balance factor of each node for the following tree.



4. When AVL tree property is violated and how to solve it?

After insertion of any node in an AVL tree if the balance factor of any node becomes other than -1,0, or 1 then it is said that AVL property is violated. So the node on the path from the inserted node to the root needs to be readjusted. Check the balance factor for each node in the path from inserted node to the root node and adjust the affected subtree such that the entire subtree should satisfy the AVL property.

5. Mention the four cases to rebalance the AVL tree.

- An insertion of new node into Left subtree of Left child(LL).
  - An insertion of new node into Right subtree of Left child(LR).
  - An insertion of new node into Left subtree of Right child(RL).
  - An insertion of new node into Right subtree of Right child(RR).

6. Define Rotation in AVL tree. Mention the two types of rotations.

Some modifications done on AVL tree in order to rebalance it is called Rotation of AVL tree.

The two types of rotations are

- Single Rotation
- Left-Left Rotation
- Right-Right Rotation
- Double Rotation
- Left-Right Rotation
- Right-Left Rotation

7. Define Splay Tree.

A **splay tree** is a self-balancing binary search tree with the additional property that recently accessed elements are quick to access again. It performs basic operations such as insertion, look-up and removal in  $O(\log(n))$  amortized time. For many non-uniform sequences of operations, splay trees perform better than other search trees, even when the specific pattern of the sequence is unknown.

8. List the Operations of Splay tree.

- Splaying
- Insertion
- Deleting

9. List the Operations on B-Trees.

- Search
- Create
- Insert

10. List the B-Trees Applications.

- Databases
- Concurrent Access to B-Trees

11. Define B-Tree.

A search tree that is not a binary tree is called B-Tree. That satisfies the following structural properties

- Root is either a leaf or has between 2 and M children
- All non leaf nodes except the root have between  $[M/2]$  and M children.
- All leaves are at the same depth.

12. Define binary heaps.

A **binary heap** is a heap data structure created using a binary tree. It can be seen as a binary tree with two additional constraints:

- The *shape property*: the tree is an *almost complete binary tree*; that is, all levels of the tree, except possibly the last one (deepest) are fully filled, and, if the last level of the tree is not complete, the nodes of that level are filled from left to right.
- The *heap property*: each node is greater than or equal to each of its children according to some comparison predicate which is fixed for the entire data structure.

13. List the Operations on Binary heap.

- Adding to the heap
- Deleting the root from the heap

14. List the Applications of Binary heap

Heap sort

Selection Algorithm

Graph Algorithm

## UNIT -IV

### HASHING AND SET

1. Define hashing.

It is the implementation of hash tables. Hashing is a technique used for performing insertions, deletions and finds in constant average time.

2. Define a Key in hashing.

a. A key is a string with an associated value.  
e.g: Salary information.

3. Define Hash table. Give an example.

The hash table data structure is an array of fixed size containing the keys.

0	50
1	
2	
3	
4	49

4. Define table size of a hash table.

Table size is the size of the table and it is part of the hash data structure. The table runs from 0 to Tablesize-1.

5. List the types of hash function.

1. Division method
2. Mid square

3. Multiplicative hash function
4. Digit folding
5. Digit analysis
  
6. Define a Hash function with an example.  
Hash function maps each key into some number in the range 0 to Tableaize-1 and places the key in the appropriate cell.
  

e.g:Key mod Tablesiz is a basic hash function

  
7. Properties of a hash function.
  1. Hash function should be simple to compute.
  2. Should ensure that any two distinct keys get different cells.
  3. It should distribute the keys evenly among the cells.
  
8. What is collision in hashing?  
If an element is inserted and if it hashes to the same value as an already inserted element collision takes place.
  
9. What are the methods of resolving collision in hashing?
  1. Separate chaining hashing.
  2. Open addressing hashing.
  
10. Define separate chaining hashing  
Separate chaining means to keep a list of all elements that hash to the same value.
  
11. What are the operations of separate chaining?
  1. Find
  2. Insert
  
12. List out the applications of hashing.
  1. DBMS.
  2. Computer networks
  3. Storage of secret data
  4. Cryptography.
  5. Securing the database
  6. Database applications
  7. Storing data in a database
  
13. What are the advantages of separate chaining?
  1. Avoids collision by maintaining linked lists.
  2. The link lists will be short if the table is large and the hash function is good.
  
14. What are the disadvantages of separate chaining?
  1. Requires pointers.
  2. Slows down the algorithm
  3. Time is required to allocate new cells.

4. Requires the implementation of a second data structure.
  
15. What is open addressing?  
In open addressing if a collision occurs alternative cells are tried until an empty cell is found.  
It is an alternative to resolving collisions with linked list.  
(i.e.)  $h_0(X), h_1(X), \dots$  Are tried in succession.  
Where  $h_i(X) = (\text{Hash}(X) + F(i)) \bmod \text{TableSize}$  with  $F(0) = 0$  where  $F$  is the collision resolution strategy.
  
16. What are the types of open addressing hashing?
  - Linear probing.
  - Quadratic probing
  - Double hashing

## 17. Differentiate between linear probing and quadratic probing

Linear probing	Quadratic probing
<ul style="list-style-type: none"> <li>• Definition:</li> <li>• It amounts to trying cells sequentially with wraparound in search of an empty cell.</li> </ul>	<ul style="list-style-type: none"> <li>• Definition:</li> <li>• It is a collision resolution method that eliminates the primary clustering problem of linear probing.</li> </ul>
<ul style="list-style-type: none"> <li>• Function:</li> <li>• It is linear ( i.e.)<math>F(i)=i</math></li> </ul>	<ul style="list-style-type: none"> <li>• Function:</li> <li>• It is quadratic (i.e.) <math>F(i)=i^2</math></li> </ul>
<ul style="list-style-type: none"> <li>• Advantages:</li> <li>• 1.Does not use pointers</li> <li>• No second data structure</li> <li>• 3. Time is not required for allocating new cells.</li> </ul>	<ul style="list-style-type: none"> <li>• Advantages:</li> <li>• 1.Does not use pointers</li> <li>• No second data structure</li> <li>• Time is not required for allocating new cells.</li> <li>• If the table is half empty and the table size is prime then inserting a new element is successful and guaranteed.</li> <li>• No primary clustering.</li> </ul>
<ul style="list-style-type: none"> <li>• Disadvantages:</li> <li>• Time for inserting is quite large.</li> <li>• Primary clustering.</li> <li>• Linear probing is a bad idea if the table is more than half full.</li> <li>• 4. Bigger table is needed than separate chaining.</li> </ul>	<ul style="list-style-type: none"> <li>• Disadvantages:</li> <li>• There is no guarantee of finding an empty cell if the table gets more than half full or if the table size is not prime.</li> <li>• Bigger table is needed than separate chaining.</li> <li>• The number of alternative locations is severely reduced if the table size is not prime.</li> </ul>

	<ul style="list-style-type: none"> <li>• 4. Standard deletion cannot be performed.</li> </ul>
--	---

18. Define primary clustering.

It is the forming of blocks of occupied cells. It means that any key that hashes into the cluster will require several attempts to resolve the collision and then it will add to the cluster.

19. Define double hashing.

It is a collision resolution method .For double hashing  $F(i)=i \cdot hash_2(X)$ .

(i.e) a second hash function is applied to X and probing is done at a distance  $hash_2(X)$ ,  $2hash_2(x)$ ..... and so on.

20. Define rehashing.

Rehashing is the building of another table with an associated new hash function that is about twice as big as the original table and scanning down the entire original hash table computing the new hash value for each non deleted element and inserting it in the new table.

21. List out the advantages and disadvantages of rehashing.

Advantage:

- Table size is not a problem.
- Hash tables cannot be made arbitrarily large.
- Rehashing can be used in other data structures.

Disadvantage:

- It is a very expensive operation.
- The running time is  $O(N)$ .
- Slowing down of rehashing method.

22. What are the ways in which rehashing can be implemented.

1. Rehash as soon as the table is half full.
2. Rehash only when an insertion fails.
3. Middle of the road strategy is to rehash when the table reaches a certain load factor.

23. Define Extendible hashing.

Extendible hashing allows a find to be performed in two disk access and insertions also require few disk accesses.

24. What is heap order property?

The smallest element should be at the root .Any node should be smaller than all of its descendants.

25. What is meant by Expression Tree?

An expression tree is a binary tree in which the operands are attached as leaf nodes and operators become the internal nodes.

26. List the applications of set.

i. Disjoint-set data structures model the partitioning of a set, for example to keep track of the connected components of an undirected graph. This model can then be used to determine whether two vertices belong to the same component, or whether adding an edge between them would result in a cycle.

ii. This data structure is used by the Boost Graph Library to implement its Incremental Connected Components functionality. It is also used for implementing Kruskal's algorithm to find the minimum spanning tree of a graph.

27. Define Disjoint set.

A **disjoint-set data structure** is a data structure that keeps track of such a partitioning. A **union-find algorithm** is an algorithm that performs two useful operations on such a data structure:

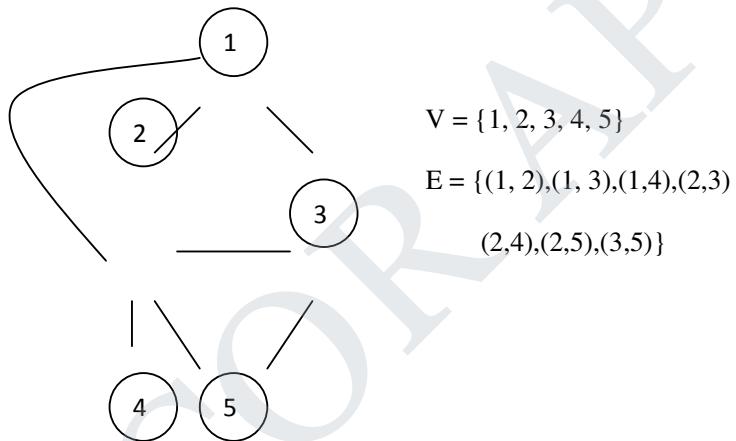
- *Find*: Determine which set a particular element is in. Also useful for determining if two elements are in the same set.
- *Union*: Combine or merge two sets into a single set.

## UNIT V

## GRAPHS

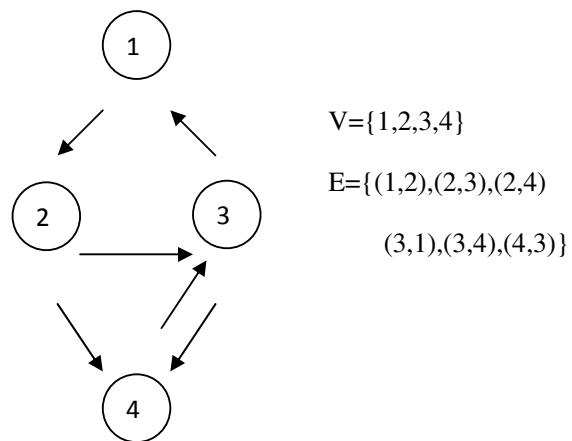
## 1. Define Graph.

A Graph is defined as  $G = (V, E)$  where  $V$  is the set of vertices (nodes) and  $E$  is the set of edges(arcs) connecting the vertices. An edge is represented as a pair of vertices  $(u,v)$ .



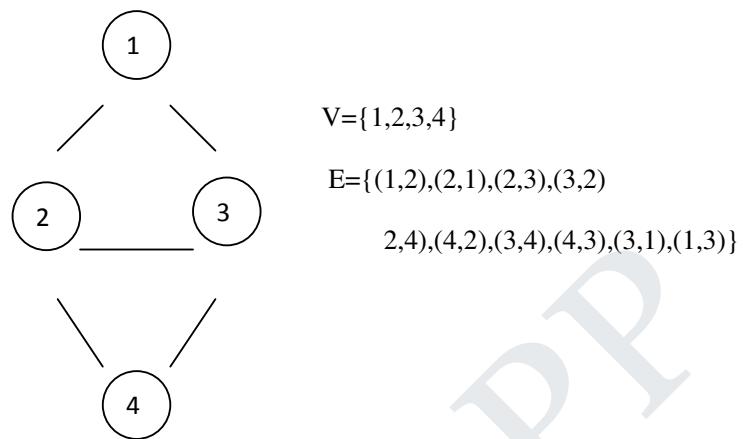
## 2. Define Directed Graph.

A Directed graph is a graph in which the edges are directed. It is also called Digraph.



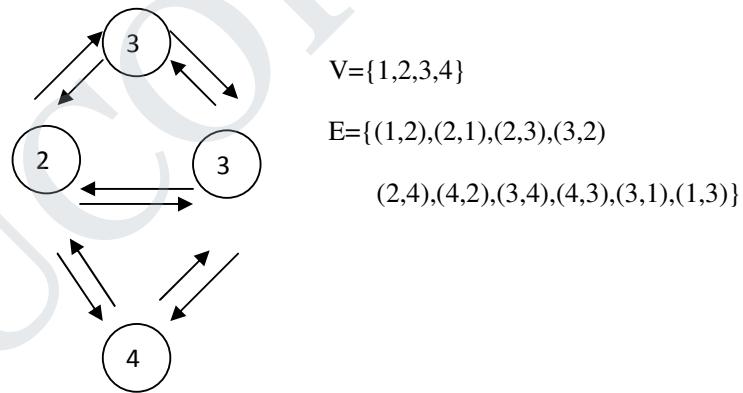
## 3. What do you mean by Undirected Graph?

Edges in the graph do not have directions marked. Such graphs are referred to as undirected graphs.



Define Symmetric Digraph.

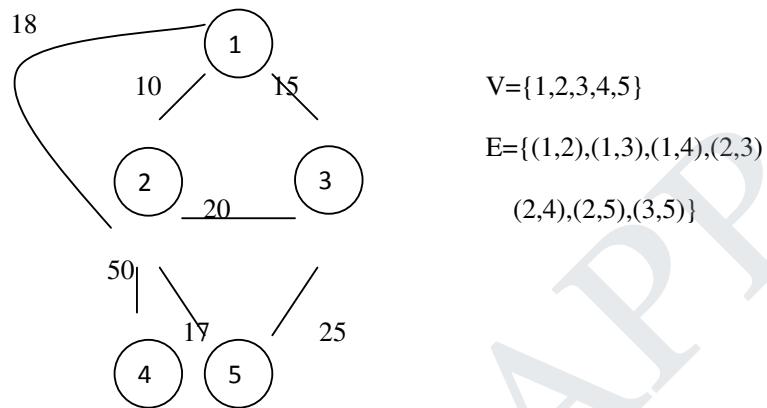
Every edge has an edge in the reverse direction i.e for every edge  $(U,V)$  there is an edge  $(V,U)$ . Such type of graph is called Symmetric Digraph.



4. What do you mean by weighted graph?

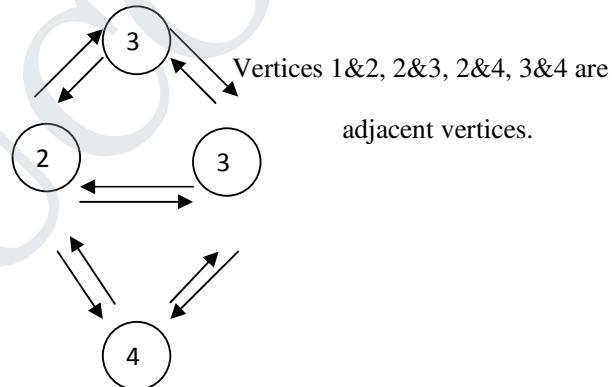
Weighted graphs are such graphs where the edges are associated with weights. These weights are used to mark the importance of edges in representing a problem.

Ex. Road map represented as graph where the weight is the distance between two places.



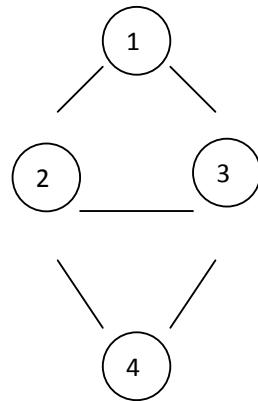
#### 5. Define adjacent vertices.

Two vertices are said to be adjacent vertices if there is an edge between them.



## 6. Define path.

A Path between vertices  $(u,v)$  is a sequence of edges which connects vertices  $u$  &  $v$ .

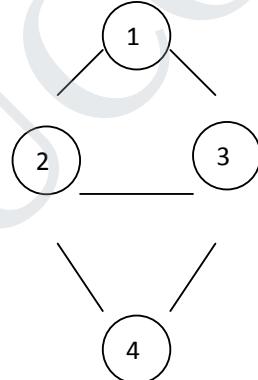


The path between the vertices 1 and 4 is 1-2-4 or 1-3-4

## 7. Define Length of the Path.

Length of the path is the number of edges in a path which is equal to  $N-1$ .

$N$  represents the number of vertices.

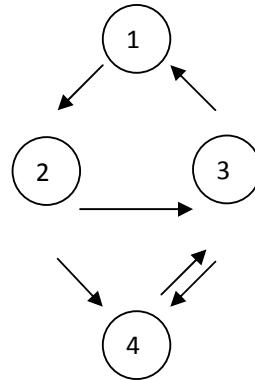


The Length of the path 1 -4  $\{(1,2),(2,3),(3,4)\}$  is 3

$\{(1,3),(3,4)\}$  is 2 and also  $\{(1,2),(2,4)\}$  is 2.

9. Define cycle.

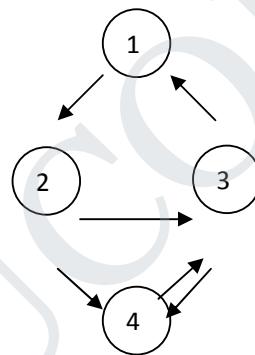
A Cycle in a graph is a path that starts and ends at the same vertex. i.e path in which the first and last vertices are same.



Ex:  $\{(3, 1), (1, 2), (2, 4), (4, 3)\}$  is a Cycle.

10. Define simple cycle.

Simple cycle means the vertex should not be repeated and the first and last vertex should be the same.

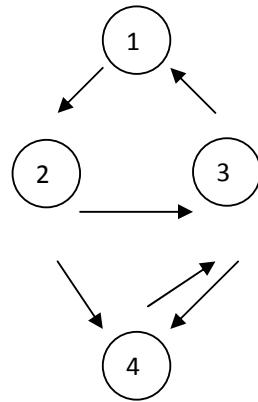


Example: 1-2-4-3-1 is a simple cycle

1-2-3-4-3-1 is not a simple Cycle

11. Define Simple Path.

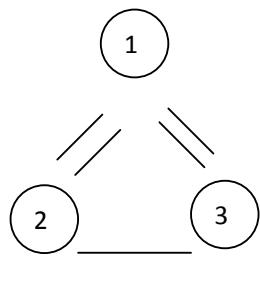
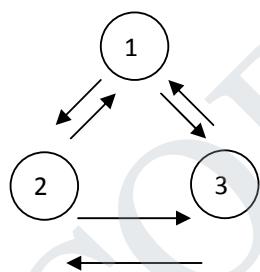
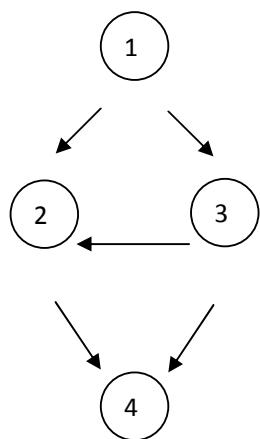
All vertices are distinct, except that the first and last may or may not be the same.



Example:    1-2-4-3-1 is a simple path

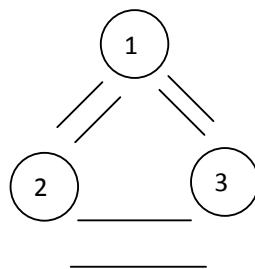
1-2-3-4-3-1 is not a simple path

1-2-4-3 is a simple path



15. Define complete graph.

A graph in which there is an edge between every pair of vertices.



16. Mention the ways of representing a graph?

- a) Adjacency Matrix representation
- b) Adjacency List representation

17. What do you mean by Adjacency Matrix representation?

The adjacency matrix  $M$  of a graph  $G = (V, E)$  is a matrix of order  $V_i \times V_j$  and the elements of the unweighted graph  $M$  are defined as

$$M[i][j] = 1, \text{ if } (V_i, V_j) \in E$$

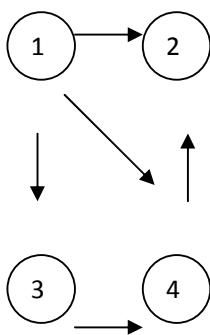
$$= 0 \text{ Otherwise}$$

For a weighted graph the elements of  $M$  are defined as

$$M[i][j] = W_{ij}, \text{ if } (V_i, V_j) \in E \text{ and } W_{ij} \text{ is the weight of edge } (V_i, V_j)$$

$$= 0 \text{ Otherwise}$$

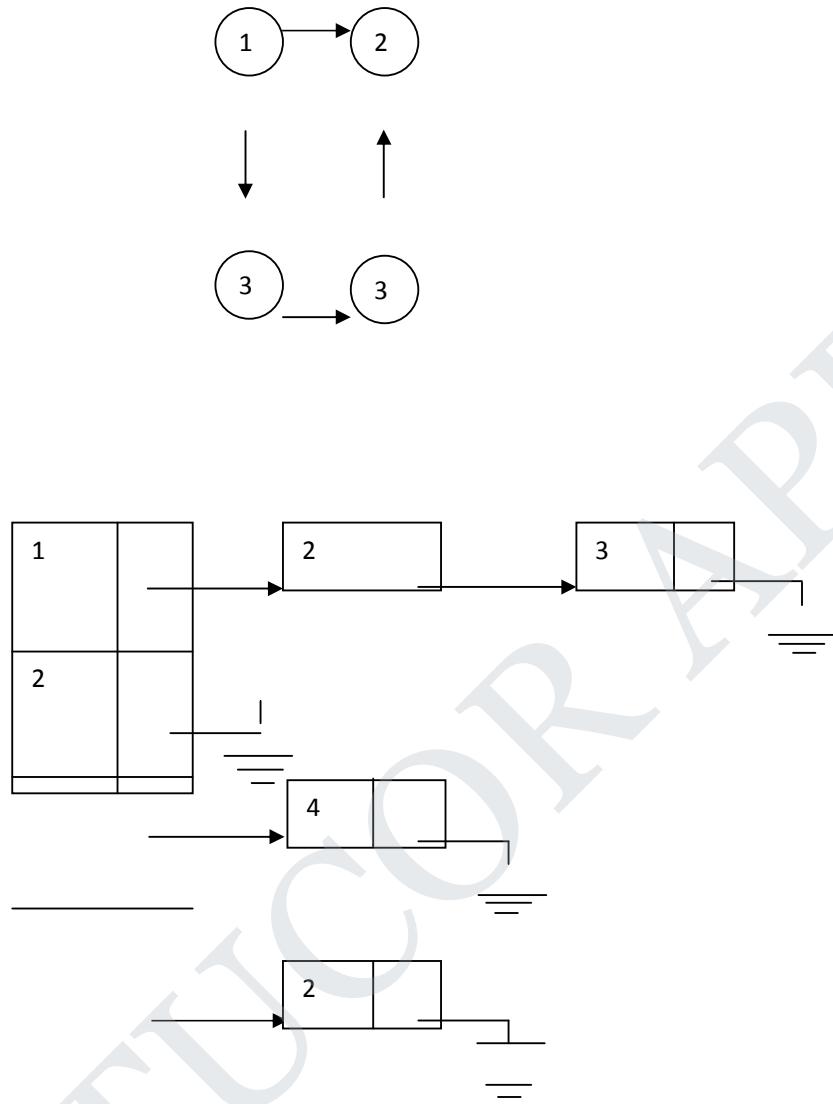
Example:



$$\begin{matrix} & & 2 & 3 & 4 \\ 1 & 0 & 1 & 1 & 1 \\ 2 & 0 & 0 & 0 & 1 \\ 3 & 0 & 0 & 0 & 1 \\ 4 & 0 & 1 & 0 & 0 \end{matrix}$$

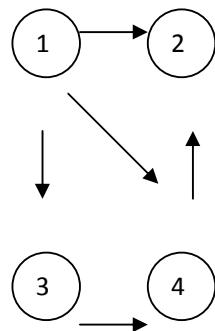
18. What do you mean by adjacency List representation?

It is an array of linked list ,for each vertex a linked list of all adjacent vertices is maintained.



19. What do you mean by indegree and outdegree of a graph?

- Indegree of a vertex in a graph is the number of incoming edges.
  - Outdegree of a vertex is the number of edges that leaves the vertex.
- Example:



Indegree for the vertex 1 is 0, vertex 2 is 2, vertex 3 is 1, and vertex 4 is 2

Outdegree for the vertex 1 is 3, vertex 2 is 0, vertex 3 is 1, and vertex 4 is 1.

20. Define Topological sort.

Topological sort is defined as an ordering of vertices in a directed acyclic graph. such that if there is a path from  $V_i$  to  $V_j$ , then  $V_j$  appears after  $V_i$  in the ordering.

21. Explain the principle of topological sort.

- Find the vertex with no incoming edge.
- Print the vertex and remove it along with its edges from the graph.
- Apply the same strategy to the rest of the graph.
- Finally all recorded vertices give topological sorted list.

22. What is the disadvantage of topological sort?

Ordering of vertices is not possible if the graph is a cyclic graph. Because if there are two vertices v and w on the cycle, v proceeds w and w proceeds v, so ordering not unique.

23. What is the running time for topological sort?

The running time of the algorithm for topological sort is  $O(|V| + |E|)$ . If adjacency list are used, the running time is  $O(|E| + |V|)$ .

24. State the shortest path problem OR Single source shortest path problem.

Given as input a weighted graph or an unweighted graph  $G = (V, E)$  and a distinguished vertex  $s$ , the shortest path problem is to find the shortest weighted or unweighted path from  $s$  to every other vertex.

25. Explain weighted path length.

$N-1$

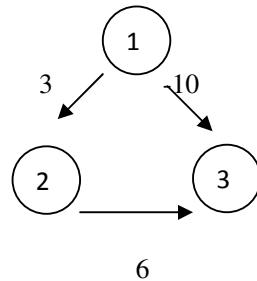
The cost of the path  $V_1, V_2, \dots, V_N$  is  $\sum_{i=1}^{N-1} C_{i, i+1}$ . This is referred to as the weighted path length.

26. Explain unweighted path length

Unweighted path length is the number of edges on the path namely,  $N-1$ (where  $N$  is the number of vertices).

27. What do you mean by Negative edge?

Negative edge means a graph having atleast one edge with a negative weight.



28. What do you mean by negative cost cycle?

A graph having the shortest path with negative weight is known Negative cost cycle.

29. Give examples for problems solved by shortest path algorithm.

- Cheapest way of sending electronic news from one computer to another.
- To compute the best route

30. What is the running time for the weighted and unweighted shortest path?

- The running time for the weighted shortest path is  $O(|E| + |V|)$
- The running time for the Unweighted shortest path is  $O(|E| \log |V|)$

31. What is the advantage of unweighted shortest path algorithm?

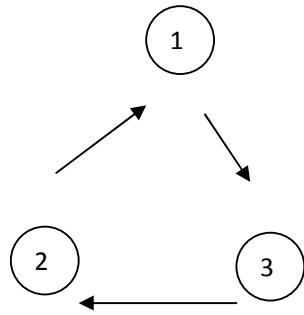
There is no calculation with weights. Only the number of edges on the shortest path is found.

35. What are the applications of graphs?

1. Airport system
2. Street traffic

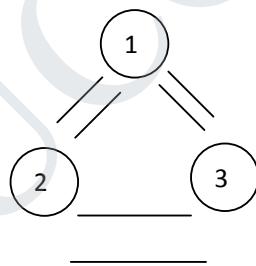
36. What is a cyclic graph?

A graph which has atleast one cycle is called a cyclic graph.



37. What is a connected graph?

An undirected graph is connected if there is a path from every vertex to every other vertex.



**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**UNIT I**

**2 Marks Questions**

1. Define Abstract Data Type.
2. What are the operations of ADT?
3. List out the areas in which Data Structures are applied extensively.
4. Define Non-linear Data Structure.
5. Distinguish between Linear and Non-Linear Data Structures.
6. What are the various operations done under the List ADT?
7. Mention the applications of List.
8. What is a Linked List?
9. What are the types of Linked List?
10. What is the need for the header?
11. Define Single Linked list.
12. Define Doubly Linked list.
13. Define Circular doubly Linked List?
14. Write down the steps to modify a Node in a Linked List.
15. Write the ways in which List ADT can be implemented.
16. List out the disadvantages of using a Linked List.
17. State the difference between Arrays and Linked Lists.
18. What is the advantage of Double Linked list over Single Linked list implementation?
19. List out the advantage of Circular linked list.
20. Write the find routine in array implementation of list.

**16/10/8 Marks Questions**

1. Explain the array implementation of list ADT with routine and example. (16)
2. What is a Linked List? Explain with suitable routine segments for any four Operations (16)
3. Explain the application of Linked list in detail. (16)
4. Explain Polynomial manipulation using Linked Lists with an example.(16)
5. Explain the different types of Linked Lists and its Implementation.(16)
6. Explain Doubly Linked List with an example.(16)
7.
  - a. With a pseudo code explain how a node can be inserted at a user specified position in a Doubly Linked List (8)
  - b. Sort the Following Elements Using Radix Sort. (8)  
10, 8, 21, 125, 54, 174, 187, 250, 1, 65
8. Write a routine to merge two arrays and traverse the array (16)
9. Explain the insertion and deletion operations in a circular single linked list with a routine and an example. (16)
10. Explain the insertion and deletion operations in a circular double linked list with a routine and an example. (16)

**UNIT II**

**2 Marks Questions**

1. Define a stack
2. List out the basic operations that can be performed on a stack
3. State the different ways of representing expressions
4. State the advantages of using infix notations.
5. State the advantages of using postfix notations.
6. State the rules to be followed during infix to postfix conversions
7. State the rules to be followed during infix to prefix conversions.
8. State the difference between stacks and linked lists
9. List the applications of stack
10. Write a routine to check whether the stack is full or empty.
11. Mention the advantages of representing stacks using linked lists than arrays
12. Write the equivalent prefix notation for infix expression  $p/q*r+s$ .
13. Define a queue
14. What are the types of queues?
15. Define circular queue
16. Define a priority queue
17. State the difference between queues and linked lists
18. Define a Deque
19. List the applications of queues
20. Write a routine to display the contents of queue.

**16/10/8 Marks Questions**

1. Explain the operations and the implementation of Stack ADT using Array. (16)
2. Explain the different operation that can be performed on Stacks? Write the algorithm for each operation. (16)
3. Explain the various application of stack? (Balanced parenthesis, Conversion, Evaluation, Recursion, etc). (16)
4. Explain the operations and the implementation of Stack ADT using Linked list. (16)
5. Convert the infix expression  $a+b^c+(d*e/f)*g$  to postfix expression and evaluate the same using stack.  $a=3, b=5, c=2, d=7, e=4, f=1, g=8$  (16)
6. What is a queue? Write an algorithm to implement queue with example. (16)
7. Explain the operations and the implementation of Queue ADT using Array (16)
8. Explain the different types of queue. (16)
9. Explain the operations and the implementation of Queue ADT using Linked list. (16)
10. Explain Prefix, Infix and postfix expressions with an example.

## UNIT 1&2

1. Define ADT.
2. What are the advantages of Linked List over arrays?
3. What are the advantages of doubly Linked List over singly linked list?
4. List the applications of List ADT.
5. List the applications of Stack and Queue.
6. What is Deque?
7. Convert the infix expression  $(a+b*c-d)/(e*f-g)$
8. What is circular queue?
9. Write a routine to return the top element of stack.
10. What is the working principle of Radix sort?
11. What is priority Queue?
  
12. Define ADT.
13. What are the advantages of Linked List over arrays?
14. What are the advantages of doubly Linked List over singly linked list?
15. List the applications of List ADT.
16. List the applications of Stack and Queue.
17. What is Deque?
18. Convert the infix expression  $(a+b*c-d)/(e*f-g)$
19. What is circular queue?
20. Write a routine to return the top element of stack.
21. What is the working principle of Radix sort?
22. What is priority Queue?

## PART B

1. Explain the array and linked list implementation of Stack.
2. Explain the array and linked list implementation of Queue.
3. What are the various linked list operations? Explain
4. Write routines to implement addition, subtraction & differentiation of two polynomials.
5. Explain how stack is applied for evaluating an arithmetic expression.
6. Explain cursor implementation of list.

**Unit 3****PART A**

1. Compare General tree with binary tree.
2. Define the following terminologies.  
Sibling,Parent,Depth,height,Level,leaf,Degree.
3. What is Full and Complete binary tree?
4. Define binary search tree.
5. Give the array and linked list representation of tree with example.
6. Define tree traversal.
7. Give the preorder, inorder and post order traversal for the following graph.
8. Draw the binary search tree for the following inputs. 70,15,29,33,44,12,79
9. Differentiate binary tree and binary search tree.
10. What is threaded binary tree?
11. Show the maximum number of nodes in a binary tree of height H is  $2^{H+1}-1$
12. List the steps involved in deleting a node from a binary search tree.
13. A full node is a node with two children. Prove that the number of full nodes plus one is equal to the number of leaves in a non empty binary tree.
14. List the applications of tree.

**PART B**

1. Write an algorithm to find an element from binary search tree.
2. Write an algorithm to insert , delete, Find minimum and maximum element from a binary search tree.
3. What are the tree traversal techniques? Explain with an example.
4. Explain the operations performed on threaded binary tree in detail.

**Unit 4****PART A**

1. Define AVL tree.
2. What is binary heap?
3. What are the 2 properties of a binary heap?
4. Define B-tree.
5. What is percolating Up and percolating down?
6. What do you mean by self adjusting tree?
7. What is splay tree?
8. What is a balance factor?
9. List the applications of Binary Heap.

10. Difference between B tree and B+ tree.
11. List the different ways of implementing priority queue.
12. What is Min heap and Max heap?

#### **PART B**

1. Explain the AVL rotations.
2. Construct splay tree for the following values.  
1,2,3,4,5,6,7,8,9
3. Explain the Basic operations performed in a Binary heap.
4. Construct a Min and MAX heap for the following values.  
23,67,1,45,7,89,56,35
5. Write a routine to perform insertion in a B-tree

#### **Unit 5**

#### **PART A**

6. Define hashing.
7. What is collision? What are the different collision resolving techniques?
8. What is open addressing?
9. What is extendible hashing?
10. Write a routine to perform the hash function.
11. What is hash table?
12. Define a graph.
13. Define path,degree,cycle,loop,directed graph, undirected graph,bigraph, weighted graph.
14. Define topological sort.
15. Define minimum spanning tree.
16. Write the routine for Depth first and breadth first traversal.
17. List the graph traversal techniques.
18. List the applications of depth first traversal.
19. List the applications of graph.
20. What is an adjacency matrix? What are the different ways for implementing it?
21. Compare directed and undirected graph.

**PART B**

1. Define Hash function. Write routines to find and insert an element in separate chaining.
2. Explain extendible hashing to resolve collision.
3. Explain open addressing with example.
4. Write a note on Dynamic equivalence problem.
5. Write short notes on Smart Union algorithm.

STUCOR APP

**B.E./B.Tech. DEGREE EXAMINATION, NOVEMBER/DECEMBER 2016.**

**Second Semester**

**Computer Science and Engineering**

**CS 6202 – PROGRAMMING AND DATA STRUCTURES – I**

**(Common to Information Technology)**

**(Regulation 2013)**

Time : Three hours

Maximum : 100 marks

Answer ALL questions.

**PART A —(10 x 2 = 20 marks)**

1. Differentiate between call by value and call by reference.
2. What are preprocessor directive? Give examples.
3. Write the different file manipulators.
4. Mention the functions for opening the file in read mode.
5. What is an ADT?
6. What is data structure? How it is classified?
7. Define Queue.
8. Give any two applications of stack.
9. Name the sorting techniques which use the divide and conquer strategy.
10. What is the difference between linear search and binary search?

**PART B — (5 x 16 = 80 marks)**

11. (a) Explain functions with variable number of arguments in detail. Write a

C program to find the sum of a numbers using functions with variable number of

arguments. (16)

**Or**

- (b) Explain the following:

(i) Function pointer in C (8)

(ii) Control Statements in C. (8)

12. (a) (i) What is a structure? Write a C program to add and subtract the two

complex numbers using structures. (8)

(ii) Explain the concept of random access files with an example. (8)

**Or**

(b) Store ten names of students in a file called ‘data .txt’. Perform operations to sort their names alphabetically. Write the sorted names into another file called ‘names .txt’. Display the names from ‘names .txt’ by opening the file in read mode. Close the files after performing all operations. (16)

13. (a) What is singly linked list? Write a C program that uses functions to perform the following operation on singly list with suitable diagrammatic representations. Consider all cases:

(i) Insertion (ii) Deletion (iii) Traversal in both ways. (16)

**Or**

(b) Illustrate the necessary algorithms to implement doubly linked list and perform all the operations on the created list. (16)

14. (a) Develop an algorithm to implement Stack ADT. Give relevant examples with diagrammatic representations. (16)

**Or**

(b) (i) Write an algorithm to implement circular queue using arrays. (10)

(ii) Show the simulation using stack for converting the expression

$p*q+(r-s/t)$  from infix to prefix. (6)

15. (a) (i) Sort the given integers and show the intermediate results using selection sort. 45, 25, 10, 2, 9, 85, 102, 1. (8)

(ii) Write a C code to sort an integer array using selection sort. (8)

**Or**

(b) What is hashing? Explain open addressing and separate chaining methods of collision resolution techniques with examples. (16)

**B.E./B.Tech. DEGREE EXAMINATION, MAY/JUNE 2016.**  
**Second Semester**  
**Computer Science and Engineering**  
**CS 6202 – PROGRAMMING AND DATA STRUCTURES – I**  
**(Common to Information Technology)**  
**(Regulation 2013)**

Time : Three hours

Maximum : 100 marks

Answer ALL questions.

PART A —(10 x 2 = 20 marks)

1. What is pointer to pointer? Give Example
2. Write the difference between while and do while loop
3. Write a C program to read a single character from a file
4. Differentiate between random access and sequential access file
5. Define ADT
6. What is circular linked list ?
7. Given the infix for an expression. write its prefix  $a^*b/c+d$
8. How do you define double ended queue?
9. What is the time complexity of the insertion sort?
10. What is hashing ?

PART B — (5 x 16 = 80 marks)

11(a) Explain in detail about variable number of arguments with an example. (16)

**OR**

(b) What is function pointer? Write a C program to add two matrices using function pointer.(16)

12. (a) (i) Write a C program to read the contents of a file “input.txt” and write the contents to “output.txt” (8)

(ii) Explain in detail about random access file concept with an example (8)

**OR**

(b) Explain in detail about structure with suitable example (16)

13. (a) Write C code for singly linked list with insert,delete,display operations using structure pointer (16)

**OR**

(b) Illustrate the algorithms to implement the doubly linked list and perform all the operations on the created list (16)

14 (a) (i) Develop an algorithm to implement Queue ADT.Give relevant examples and diagrammatic representations (12)  
(ii) Differentiate between double ended queue and Circular queue. (4)

**OR**

(b)(i) Write an algorithm to convert the infix expression to postfix expression.(10)  
(ii) Show the simulation using stack for the following expression to convert infix to postfix  $p*q+(r-s/t)$  (6)

15. (a) Explain the following  
(i) Binary searching

(8)

(ii) Rehashing (8)

**OR**

(b) Sort the following integer elements using Quick Sort 40, 20,70,14,60,61,97,30 (16)

**B.E./B.Tech. DEGREE EXAMINATION, DECEMBER 2015/JANUARY 2016**  
**Second Semester**  
**Computer Science and Engineering**  
**CS 6202 – PROGRAMMING AND DATA STRUCTURES – I**  
**(Common to Information Technology)**  
**(Regulation 2013)**

Time : Three hours

Maximum : 100 marks

Answer ALL questions.

**PART A —(10 x 2 = 20 marks)**

1. List any four advantages of pointers.
2. Give the significance of function declaration
3. Compare Structures and Unions
4. List the file opening modes in C
5. Define an abstract type. List out few
6. What are the advantages of linked lists over arrays?
7. Define Queues
8. What is double ended queue?
9. Sort the following numbers using bubble sort 105, 7, 11,4,1
10. Define hashing.

**PART B—(5x 16=80 marks)**

11. (a) (i) Illustrate the various control structures used in C with suitable example. (10)  
(ii) Write a C Program to implement the following string handling functions using functions and pointers. (6)  
(1) Strlen ()  
(2) Strcat ()  
(3) Strcpy ()

**(OR)**

- (b)(i)Write a C program to print the Fibonacci series using recursion (6)  
(ii) Write a C Program with functions and pointers to multiply two matrices and return the resultant matrix to the calling function (10)

12.(a) Define a structure data type named date containing three integer members day, month and year. Develop an interactive modular program to perform the following tasks :

(i) To read data into structure members by a function. (8)

(ii) To print the date in the following format : April 15, 2015 by a second function. (8)

**(OR)**

(b) Write a C program to create a file that could store details about five products. Details include product code, product names cost and number of items available which are provided through keyboard. Get the input as product code and display the details of the product from the file.(16)

13 (a) Illustrate the algorithms to create the singly linked list and perform all the operations on the created list. (16)

**(OR)**

(b) Write a program to add two polynomials using linked list. (16)

14. (a) (i) Write an algorithm to convert the infix expression to postfix expression. (10)

(ii) Show the simulation using stack for the following expression:

$12 + 3 * 14 - (5 * 16) + 7$  (6)

**(OR)**

b) (i) Write an algorithm to implement the circular queue using arrays. (10)

(ii) List the applications of queues. (6)

15. (a) (i) Sort the following sequence using quick sort.

2, 13, 45, 56, 27, 18, 24, 30, 87, 9 (8)

(ii) Write an algorithm to search a number in a given set of numbers using binary search (8)

**(OR)**

(b) Explain the open addressing and chaining methods of collision resolution techniques in hashing. (16)

**B.E./B.Tech. DEGREE EXAMINATION, APRIL/MAY 2015.****Second Semester****Computer Science and Engineering****CS 6202 – PROGRAMMING AND DATA STRUCTURES – I****(Common to Information Technology)****(Regulation 2013)**

Time : Three hours

Maximum : 100 marks

Answer ALL Questions

PART A-( $10 \times 2 = 20$  marks)

1. Give two examples of C preprocessor with syntax.
2. What are function pointers in C ? Explain with example.
3. What is the difference between `getc()` and `getchar()` ? Explain.
4. Explain the syntax as given below:

`Fread(&my_record,sizeof(struct rec),ptr_myfile);`

5. Define ADT.
6. What is static linked list ? State any two applications of it.
7. Write the syntax of `calloc()` and `realloc()` and mention its application in linked list.
8. Given the prefix for an expression, write its postfix

- \* - + a b c / e f - g / h i

9. What is meant by internal and external sorting ? Give four example of each type.
10. State the applications of linear and binary search techniques.

PART B (  $5 \times 16 = 80$  marks )

11. (a) (i) Write a function that returns a pointer to the maximum value of an array of double's. If the array is empty, return NULL.

`Double * maximum(double *a, int size);` (8)

(ii) Write a C program to find all the roots of a quadratic equation. (8)

(or)

- (b) (i) Write a C program using function to check if the given input number is palindrome or not. (8)

(ii) Explain the C preprocessor operations, each with a neat example that is used to create macros. (8)

12. (a) (i) Write a C program that uses functions to perform the following operations using structure:

- 1) Reading a complex number
- 2) Writing a complex number
- 3) Addition of two complex numbers
- 4) Multiplication of two complex numbers (12)

- (ii) State the advantage and disadvantage of structures and unions in C Programming (4)

(or)

- (b) (i) Perform the following to manipulate file handling using C.
- 1) Define an input file handle called *input\_file* , which is a pointer to a type FILE.
  - 2) Using *input\_file*, open the file *results.dat* for read mode.
  - 3) Write C statements which tests to see if *input\_file* has opened the data file successfully. If not, print an error message and exit the program.
  - 4) Write C code which will read a line of characters (terminated by a \n) from *input\_file* into a character array called buffer. NULL terminate the buffer upon reading a \n.
  - 5) Close the file associated with *input\_file*.
- (12)
- (ii) Using C programming, display the contents of a file on screen (4)
13. (a) Write a C program to perform addition, subtraction and multiplication operations on polynomial using linked list. (16)
- (or)
- (b) Write C code for Circular link list with create, insert, delete, display operations using structure pointer.
- (16)
14. (a) (i) Write C program that checks if expression is correctly parenthesized using stack. (12)
- (ii) Write the function to check for state status as Full ( ) or Empty ( ). (4)
- (or)
- (b) Write C program to implement Queue functions using Arrays and Macros. (16)
15. (a) (i) Sort the given integers and show the intermediate results using shell sort  
 35 , 12 , 14 , 9 , 15 , 45 , 32 , 95 , 40 , 5 (8)
- (ii) Write C code to sort an integer array using shell sort. (8)
- (or)
- (b) (i) Explain a C code to perform binary search. (10)
- (ii) Explain the Rehashing techniques. (6)

**B.E./B.Tech. DEGREE EXAMINATION, NOVEMBER/DECEMBER 2014.**

**Second Semester**

**Computer Science and Engineering**

**CS 6202 – PROGRAMMING AND DATA STRUCTURES – I**

**(Common to Information Technology)**

**(Regulation 2013)**

Time : Three hours

Maximum : 100 marks

**Answer ALL Questions**

**PART A-( $10 \times 2 = 20$  marks)**

1. Define an array. Give an example.
2. Give example on call by reference.
3. What are the statements used for reading a file.
4. Define the need for union in C.
5. What are abstract data type?
6. What is circular linked list?
7. Give the applications of stack.
8. What is double ended queue?
9. Define extendible hashing.
10. Differentiate internal and external sorting.

**PART B – ( $5 \times 16 = 80$  marks)**

1. (a) Explain the various control statements in C language with example in detail.(16)

**Or**

- (b) Briefly discuss about:

- (i) Functions with number of arguments.
  - (ii) Function Pointers.

**(8 + 8)**

12. (a) Explain the difference between structure and union with examples.

**(16)**

**Or**

- (b) Explain about file manipulations in detail with suitable program.

**(16)**

- 13 (a) Describe the creation of a doubly linked list and appending the list. Give relevant coding in C.

**(16)**

**Or**

- (b) Explain the following:

- (i) Applications of lists.
  - (ii) Polynomial manipulation.

**(8 + 8)**

- 14 (a) Discuss about Stack ADT in detail. Explain any one application of stack.

**(16)**

**Or**

- (b) Explain about Queue ADT in detail. Explain any one application of queue with suitable example.

**(16)**

15 (a) What are the different types of hashing techniques? Explain them in detail with example.  
(8 + 8)

**Or**

(b) Write an algorithm to sort a set of 'N' numbers using quick sort. Trace the algorithm for  
the following set of numbers: 88, 11, 22, 44, 66, 99, 32, 67, 54, 10. (16)

STUCOR APP

**B.E./B.Tech. DEGREE EXAMINATION, MAY/JUNE 2014.**

**Second Semester**

**Computer Science and Engineering**

**CS 6202 – PROGRAMMING AND DATA STRUCTURES – I**

**(Common to Computer and Communication Engineering and Information Technology)**

**(Regulation 2013)**

Time : Three hours

Maximum : 100 marks

**Answer ALL Questions**

**PART A - (10 X 2 = 20 marks)**

1. With the help of the printf function show how C handles functions with variable number of arguments.
2. Define macro with an example.
3. Give applications in which union rather than structures can be used.
4. Will the following declaration work. Justify your answer.

Struct Student

```
{  
    int rollno = 12;  
    float marks[] = { 55, 60, 56 );  
    char gender;  
};
```

5. Should arrays or linked lists be used for the following types of applications. Justify your answer.
  - (a) Many search operations in sorted list
  - (b) Many search operations in unsorted list
6. What is advantage of an ADT?
7. Define double ended queue.
8. List the applications of a Queue.
9. What is the time complexity of binary search?
10. List sorting algorithm which uses logarithmic time complexity.

PART B – ( 5 X 16 = 80 MARKS)

11 (a) (i) Write a c program to find the unique elements in an array using a function ‘Unique’.

The function takes the array as a parameter and prints the unique elements . (10)

(ii) Write a C program to print Fibonacci numbers. (6)

**Or**

(b) Write a C program to multiply two matrices that are represented as pointers. Use a function pointer to the function ‘Multiply’ which takes the two matrices as parameter and prints the result of the multiplication. (16)

12 (a) (i) Write a C program to read the contents of a file “in.txt” from last to first and write the contents to “out.txt”. (8)

(ii) Write the function prototype and explain how files are manipulated in C. (8)

**Or**

(b) (i) Create a structure to store a complex number and write function (for addition) that handles this new structure. ( 8 )

(ii) Write a program to perform the following operations for the customers of a bank using the concept of structures. ( 8 )

(1) Input the customer details like name, account number and balance.

( 2) When a withdrawal transaction is made the balance must change to reflect it.

( 3) When a deposit transaction is made the balance must change to reflect it..

13 (a) Write an algorithm to perform insertion and deletion on a doubly linked list. (16)

**Or**

(b) Consider an array A[1:n]. Given a position, write an algorithm to insert an element in the array. If the position is empty, the element is inserted easily. If the position is already occupied the element should be inserted with the minimum number of shifts. (Note: The elements can shift to the left or right to make the minimum number of moves) (16)

14 (a) Write an algorithm to convert an infix expression to a postfix expression. Trace the algorithm to convert the infix expression “(a+b)\*c/d+e/f” to a postfix expression. Explain the need for infix and postfix expressions. (16)

**Or**

- (b) Write an algorithm to perform the four operations in a double ended queue that is implemented as an array. (16)
- 15 (a) Write short notes on hashing and the various collision resolution techniques. (16)

B.E./B.Tech. DEGREE EXAMINATION, NOVEMBER/DECEMBER 2009

Third Semester

Information Technology

IT 2201 — DATA STRUCTURES AND ALGORITHMS

(Regulation 2008)

Time : Three hours Maximum : 100 Marks

Answer ALL Questions

PART A — (10 × 2 = 20 Marks)

1. What is meant by abstract data type (ADT)?
2. What are the postfix and prefix forms of the expression  
 $A + B^* (C - D) / (P - R)$
3. What is a Binary tree?
4. Define expression tree.
5. What are the applications of hash table?
6. What is an equivalence relation?
7. Define indegree and out degree of a graph.
8. What is a minimum spanning tree?
9. Compare backtracking and branch-and-bound.
10. List the various decision problems which are NP-Complete.

PART B — (5 × 16 = 80 Marks)

11. (a) (i) Write the insertion and deletion procedures for cursor based linked lists. (8)  
(ii) Write the algorithm for the deletion and reverse operations on doubly linked list. (8)

Or

- (b) (i) Write an algorithm for Push and Pop operations on Stack using Linked List. (8)  
(ii) Explain the addition and deletion operations performed on a circular queue with necessary algorithms. (8)

12. (a) (i) Write the algorithm for pre-order and post-order traversals of a binary tree. (8)  
(ii) Explain the algorithm to convert a postfix expression into an expression tree with an example. (8)

Or

- (b) (i) Write an algorithm to insert an item into a binary search tree and trace the algorithm with the items 6, 2, 8, 1, 4, 3, 5. (8)  
(ii) Describe the algorithms used to perform single and double rotation

on AVL tree. (8)

13. (a) Discuss the common collision resolution strategies used in closed hashing system. (16)

Or

(b) (i) What is union-by-height? Write the algorithm to implement it. (8)

(ii) Explain the path compression with an example. (8)

14. (a) (i) What is topological sort? Write an algorithm to perform topological sort. (8)

(ii) Write the Dijkstra's algorithm to find the shortest path. (8)

Or

(b) Write the Kruskal's algorithm and construct a minimum spanning tree for the following weighted graph. (16)

15. (a) (i) Formulate an algorithm to multiply n-digit integers using divide and conquer approach. (8)

(ii) Briefly discuss the applications of greedy algorithm. (8)

Or

(b) Find the optimal tour in the following traveling salesperson problem using dynamic programming : (16)

B.E./B.Tech. DEGREE EXAMINATION, NOVEMBER/DECEMBER 2010

Third Semester

Information Technology

IT 2201 — DATA STRUCTURES AND ALGORITHMS

(Regulation 2008)

Time : Three hours Maximum : 100 Marks

Answer ALL questions

PART A — (10 × 2 = 20 Marks)

1. Define ADT.
2. Clearly distinguish between linked lists and arrays. Mention their relative advantages and disadvantages.
3. What is meant by depth and height of a tree?
4. Discuss the application of trees.
5. What are the important factors to be considered in designing the hash function?
6. What is a disjoint set? Define the ADT for a disjoint set.
7. What is Euler circuit?
8. What are the two ways of representing a graph? Give examples.
9. Define NP-complete problems.
10. What is meant by backtracking?

PART B — (5 × 16 = 80 Marks)

11. (a) (i) Derive an ADT to perform insertion and deletion in a singly linked list. (8)

(ii) Explain cursor implementation of linked lists. Write the essential operations. (8)

Or

(b) (i) Write an ADT to implement stack of size N using an array. The elements in the stack are to be integers. The operations to be supported are PUSH, POP and DISPLAY. Take into account the exceptions of stack overflow and stack underflow. (8)

(ii) A circular queue has a size of 5 and has 3 elements 10, 20 and 40 where F = 2 and R = 4. After inserting 50 and 60, what is the value of F and R. Trying to insert 30 at this stage what happens? Delete 2 elements from the queue and insert 70, 80 & 90. Show the sequence of steps with necessary diagrams with the value of F & R. (8)

12. (a) (i) Write an ADT to construct an AVL tree. (8)

(ii) Suppose the following sequences list nodes of a binary tree T in preorder and inorder, respectively :

Preorder : A, B, D, C, E, G, F, H, J

Inorder : D, B, A, E, G, C, H, F, J

Draw the diagram of the tree. (8)

Or

(b) (i) Write an ADT for performing insert and delete operations in a Binary Search Tree. (8)

(ii) Describe in detail the binary heaps. Construct a min heap tree for the following :

5, 2, 6, 7, 1, 3, 8, 9, 4 (8)

13. (a) (i) Formulate an ADT to implement separate chaining hashing scheme. (8)

(ii) Show the result of inserting the keys 2, 3, 5, 7, 11, 13, 15, 6, 4 into an initially empty extendible hashing data structure with  $M = 3$ . (8)

Or

(b) (i) Formulate an ADT to perform for the Union and find operations of disjoint sets. (8)

(ii) Describe about Union-by-rank and Find with path compression with code. (8)

14. (a) (i) Write routines to find shortest path using Dijkstra's algorithm. (8)

(ii) Find all articulation points in the below graph. Show the depth first spanning tree and the values of DFN and Low for each vertex. (8)

Or

(b) (i) Write the pseudo code to find a minimum spanning tree using Kruskal's algorithm. (8)

(ii) Find the topological ordering of the below graph. (8)

15. (a) (i) Discuss the running time of Divide-and-Conquer merge sort algorithm. (8)

(ii) Explain the concept of greedy algorithm with Huffman codes example. (8)

Or

(b) Explain the Dynamic Programming with an example. (16)

Reg. No. : 

--	--	--	--	--	--	--	--	--	--	--	--

**Question Paper Code : 21507**

B.E./B.Tech. DEGREE EXAMINATION, MAY/JUNE 2013.

Third Semester

Information Technology

IT 2201/ IT 33/10144 IT 304/080250005 – DATA STRUCTURES AND ALGORITHMS

(Regulation 2008/2010)

Time : Three hours

Maximum : 100 marks

Answer ALL questions.

PART A — (10 × 2 = 20 marks)

1. What is abstract data types? Give example.
2. What are the applications of stack and queue?
3. Show that in a binary tree of N nodes, there are  $N + 1$  NULL pointer.
4. Show the result of inserting 2; 1; 4; 5; 9; 3; 6; 7 into an initially empty AVL-tree.
5. What is rehashing?
6. Write code for disjoint set find.
7. Does either prim's or Kruskal's algorithm work if there are negative edge weights?
8. List out the applications of graph.
9. Compare and contrast greedy algorithm and dynamic programming.
10. Draw the solution for the 4-queen problem.

## PART B — (5 × 16 = 80 marks)

11. (a) (i) Explain how stack is used to convert the following infix expression into postfix form  $a + b^* c + (d^* e + f)^* g$ . (8)

- (ii) Give the linked list implementation of stack. (8)

Or

- (b) Explain and write the routine for insertion, deletion and finding element in the cursor based linked list. (16)

12. (a) (i) Construct an expression tree for the expression  $ab + cde +^{**}$  (10)

- (ii) Give a precise expression for the minimum number of nodes in an AVL tree of height  $h$  and what is the minimum number of nodes in an AVL tree of height 15? (6)

Or

- (b) (i) Write function to perform delete min in a binary heap. (8)

- (ii) Show the result of inserting 3; 1; 4; 6; 9; 2; 5; 7 into an initially empty binary search tree. (8)

13. (a) Given input {4371, 1323, 6173, 4199, 4344, 9679, 1989} and a hash function  $h(x) = x \bmod 10$ , show the resulting:

- (i) Separate chaining hash table.

- (ii) Open addressing hash table using linear probing.

- (iii) Open addressing hash table using quadratic probing.

- (iv) Open addressing hash table with second hash function  $h_2(x) = 7 - (x \bmod 7)$ . (16)

Or

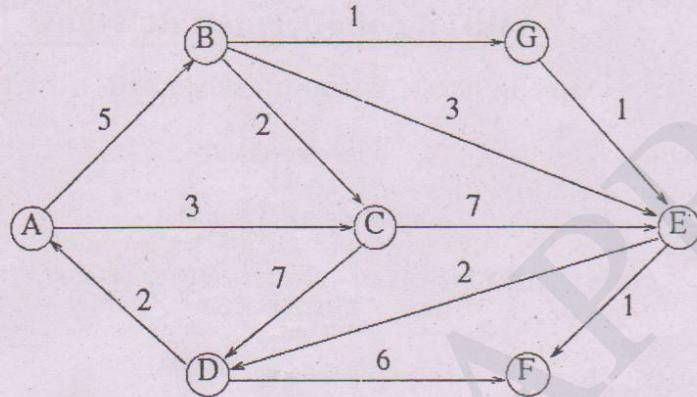
- (b) Give short note on:

- (i) Dynamic equivalence problem. (8)

- (ii) Smart union algorithm. (8)

14. (a) (i) Find the shortest weighted path from A to all other vertices for the graph in given below figure.

(ii) Find the shortest unweighted path from B to all other vertices for the graph in given below figure. (16)



Or

(b) (i) Write a routine to implement Kruskal's algorithm. (8)

(ii) Discuss in detail about bi connectivity. (8)

15. (a) (i) Explain in detail about branch and bound algorithm design technique with an example. (10)

(ii) Write a routine for random number generator algorithm. (6)

Or

(b) (i) Explain in detail about asymptotic notation. (6)

(ii) Discuss in detail about NP-complete problem with example. (10)

Reg. No. : 

--	--	--	--	--	--	--	--	--	--	--	--

**Question Paper Code : 11380**

B.E./B.TECH DEGREE EXAMINATION, APRIL/MAY 2011

Third Semester

Information Technology

IT 2201 — DATA STRUCTURES AND ALGORITHMS

(Regulation 2008)

Time : Three Hours

Maximum : 100 marks

Answer ALL questions

**PART A — (10 × 2 = 20 marks)**

1. Which operations are supported by the list ADT?
2. Write a routine to make an Empty Queue using array implementation.
3. When does a binary tree become a binary search tree?
4. Draw an expression tree for the expression  $(a + b/c) + ((d * e - f)/g)$ .
5. Give some applications of hashing.
6. Why do we need extensible hashing and say how this could be achieved?
7. Define topological sort.
8. Define height and forest with respect to the following tree :



9. Give the running time equation for divide and conquer algorithms.
10. Define NP-hard and NP-complete Problems.

**PART B — (5 × 16 = 80 marks)**

11. (a) Give cursor implementation of linked list operations and explain the same. (16)

Or

- (b) (i) Explain in detail any three applications of stack. (8)
- (ii) Illustrate with a neat example the implementation of a circular queue. (8)

12. (a) (i) With an example explain the algorithms of inorder and postorder traversals on a binary search tree. (8)  
(ii) Explain the binary heap in detail. (8)

Or

- (b) (i) Explain the tree traversals in detail. (8)  
(ii) Explain in detail the AVL tree rotations. (8)

13. (a) (i) Write a brief note on equivalence relations. (4)  
(ii) Explain in detail the collision resolution methods. (12)

Or

- (b) (i) Write brief note on the abstract data types based on sets. (6)  
(ii) Discuss in detail open addressing and rehashing. (10)

14. (a) (i) Explain the method of constructing a minimum cost spanning tree using Kruskal's algorithm. (10)  
(ii) Explain briefly articulation points and biconnected components. (6)

Or

- (b) (i) Explain the traversals of directed graphs also give its analysis. (8)  
(ii) Explain Prim's algorithm to construct a minimum spanning tree from an undirected graph. (8)

15. (a) (i) Solve the problem of 8-queens using backtracking approach. Explain every step of the solving process. (10)  
(ii) List and brief the asymptotic notations. (6)

Or

- (b) (i) Explain how the knapsack problem is handled by greedy algorithm to arrive at a solution. (1)  
(ii) Write short notes on :  
(1) Divide and Conquer algorithms (3)  
(2) Randomized algorithms. (3)
-