DMI COLLEGE OF ENGINEERING

(An Autonomous Institution)

Palanchur – Nazarethpet P.O., Chennai – 600 123

Approved by AICTE – New Delhi, NBA Accredited Programmes,
Affiliated to Anna University – Chennai, ISO Certified Institution

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SUBJECT CODE	**: CS1202**
SUBJECT NAME	**: DATA STRUCTURES**
REGULATION	**:2024**
ACADEMIC YEAR	**:2025 – 2026(ODD SEM)**
PROGRAMME	**:B.E CSE**
YEAR/SEM	**:II/III**

CS1202 DATA STRUCTURES

UNIT I LISTS

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 ADT – Multilists

UNIT II– STACKS AND QUEUES

Stack ADT – Operations – Applications: Balancing Symbols - Evaluating arithmetic expressions- Infix to postfix conversion - Queue ADT – Operations - Circular Queue – Dequeue- Priority Queue - applications of queues.

UNIT III TREES

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

UNIT IV GRAPHS

Graph Definition - Graph Terminologies —Representation of Graphs — Types of Graph — Graph Isomorphism -Graph Traversals: Breadth-first traversal(BFS) — Depth-first traversal (DFS)–connectivity – Euler Paths and Circuits —Hamiltonian Paths and Circuits - Topological Sort – Shortest Path Algorithms :Dijkstra's algorithm – Floyd's algorithm – Minimum Spanning Tree: Prim's algorithm – Kruskal's algorithm – Applications of Graph.

UNIT V SEARCHING, SORTING AND HASHING TECHNIQUES

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.

TEXT BOOKS:

1.Reema Thareja, ―Data Structures Using C‖, Second Edition , Oxford University Press, 2014
2. Langsam, Augenstein and Tanenbaum, Data Structures Using C and C++, 2nd Edition, Pearson Education, 2015.

UNIT I LISTS

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 ADT – Multilists

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 tothose 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 relationshipwith each other.

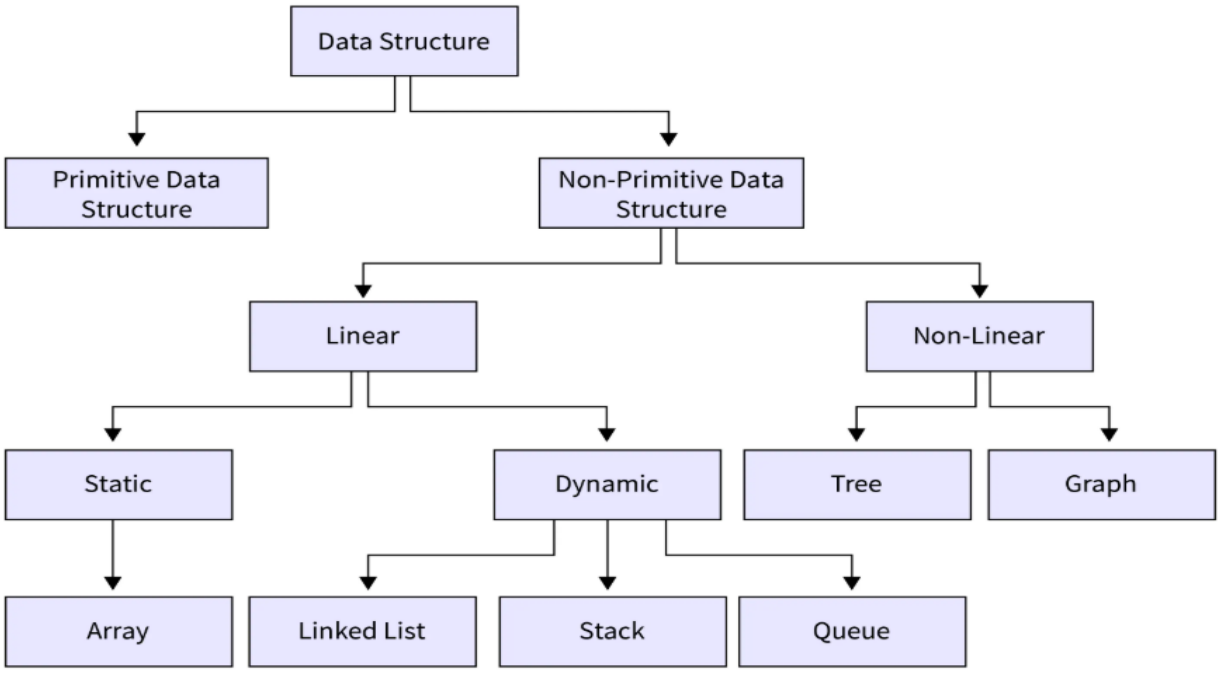
Characteristics of data structures:

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 Strucutres/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 strucutres in which, all the data elements are arranged in linear or sequential fashion. Linear Data Structures classified into 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. Examples of static data structures include arrays.

Dynamic Ds:

If the ds is created using dynamic memory allocation i.e ds formed when the number of dataitems are not known in advance is known as dynamic ds or variable size ds. Examples of static data structures include stacks, queues, linked lists, etc.

Non Linear Structures:

In non-linear data strucutres, 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.

Application of data structures:

Data structures are widely applied in the following areas:

- Compiler design
- Operating system
- Statistical analysis packageDBMS
- 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 usesthe ADTs.

THE LIST AI)T:

List is an ordered set of elements.

The general form of the list is A_1 ,A_2 , ……,A_N

A_1 - First element of the listA_2- 1st 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

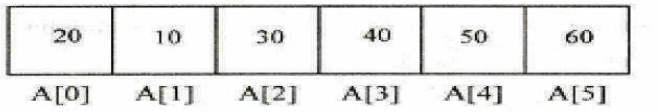
2. Linked List based implementation

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 movementsFind

and Print list operations takes constant time.



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:

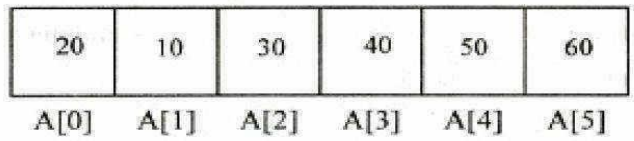
```
#define    maxsize    10    int
list[maxsize], n ;
```

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]



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 positionin 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:

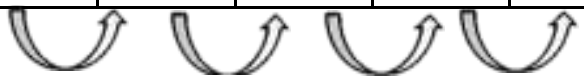
```
void Insert( )
{
        int i,data,pos;
        printf("\nEnter the data to be inserted:\t");
        scanf("%d",&data);
        printf("\nEnter the position at which element to be inserted:\t");
        scanf("%d",&pos);
        if (pos==n)
        printf (“Array overflow”); for(i =
        n-1 ; 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 2nd 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**					



10		**20**	**30**	**40**	**50**					

After this four data movement, 15 is inserted in the 2nd 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 2nd 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 storedfrom 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");
}
```

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**.**

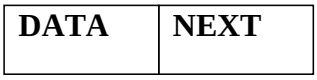
Differences between Array based and Linked based implementation

	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

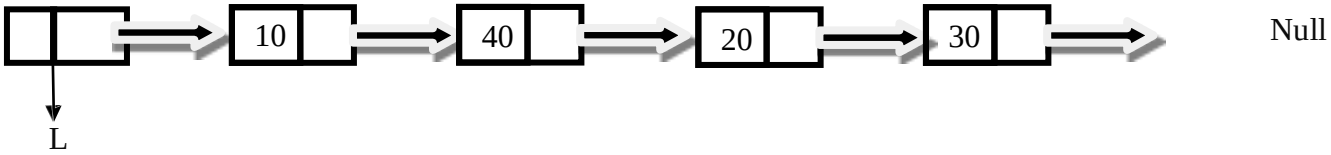
Linked list based implementation:

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



A Linked list node



Advantages of Linked list:

1. Insertion and deletion of elements can be done efficiently2.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

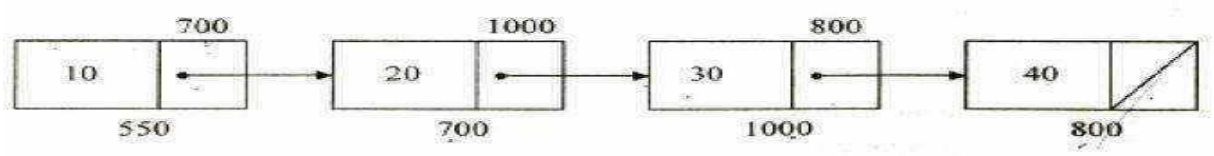
The process of allocating memory to the variables during execution of the program or at run time is known as dynamic memory allocation. C language has four library routines which allow this function.
Dynamic memory allocation gives best performance in situations in which we do not know memory requirements in advance. C provides four library routines to automatically allocate memory at the run time.

Memory allocation/de-allocation functions

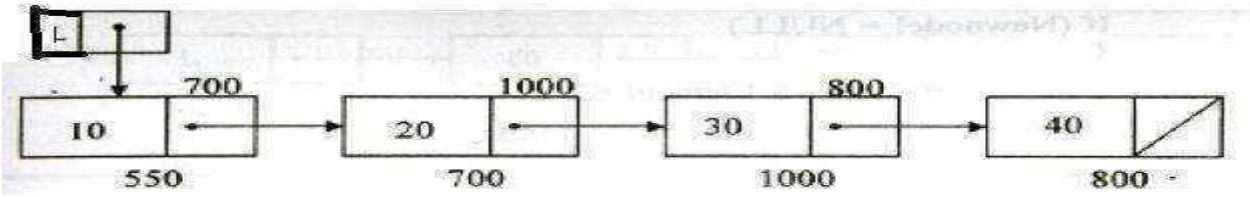
Function	Task
malloc()	Allocates memory and returns a pointer to the first byte of allocated space
calloc()	Allocates space for an array of elements, initializes them to zero and returns a pointer to the memory
free()	Frees previously allocated memory
realloc()	Alters the size of previously allocated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

Singly Linked List

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



SLL



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;
};
```

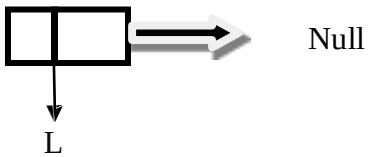
Creation of the list:

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

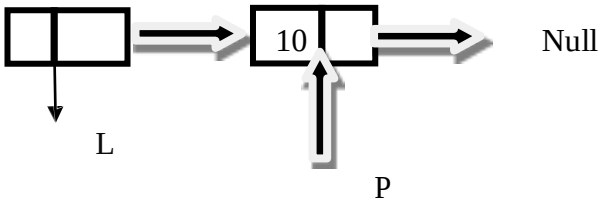
```
void create()
{
int i,n;
L=NULL;
newnode=(struct node*)malloc(sizeof(struct node));
printf("\n Enter the number of nodes to be inserted\n");
scanf("%d",&n);
printf("\n Enter the data\n");
scanf("%d",&newnode->data);
newnode->next=NULL;
L=newnode;
p=L;
for(i=2;i<=n;i++)
{
newnode=(struct node *)malloc(sizeof(struct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

Initially the list is empty

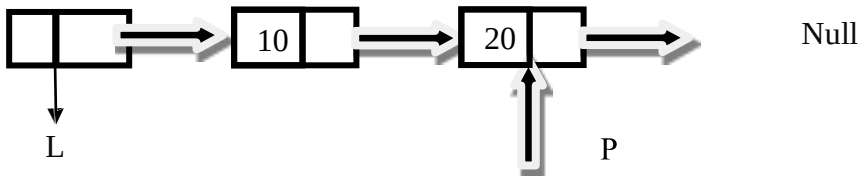
List L



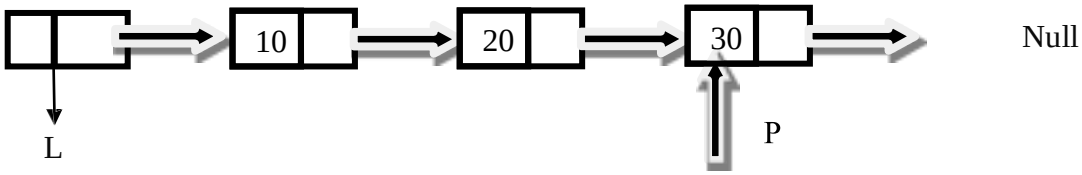
Insert(10,List L)- A new node with data 10 is inserted and the next field is updated toNULL. The next field of previous node is updated to store the address of new node.



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. Thenext 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

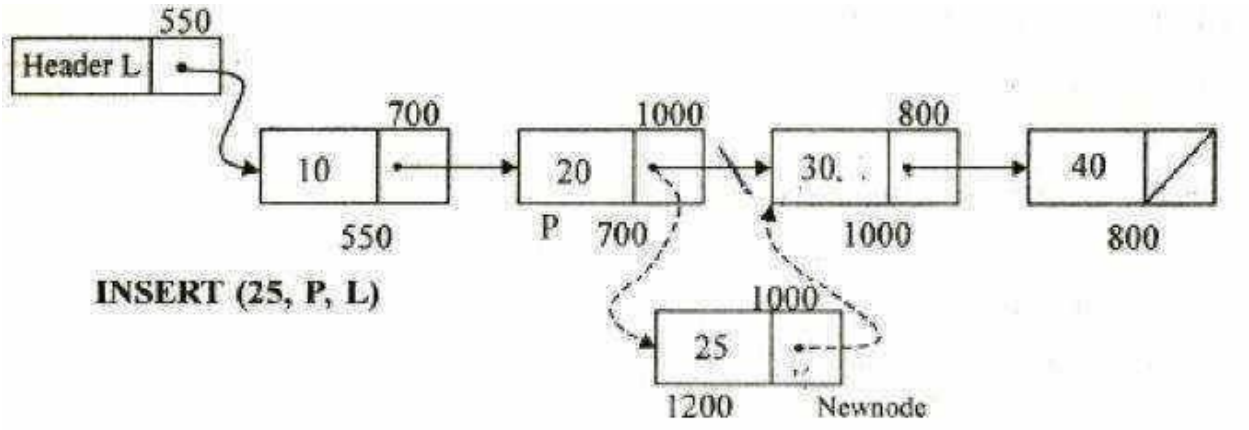
```
void insert(int X, List L, position p)
{
        p=L;
        newnode=(struct node*)malloc(sizeof(struct node));
        printf("\nEnter the data to be inserted\n");
        scanf("%d",&newnode->data);
        newnode->next=L;
        L=newnode;
}
```

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 isupdated to NULL. The next field of previous node is updated to store the address of new node.



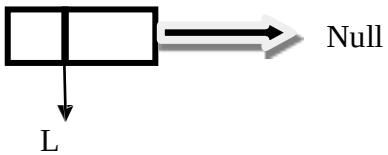
Case 3:Routine to insert an element in list at the end of the list

```
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 lis t is empty it returns 1

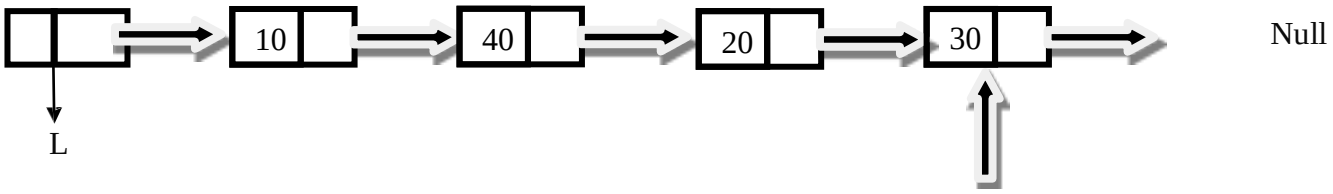
```
int IsEmpty( List L )
{
  if ( L -> next = = NULL )
        return(1);
  }
```



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);
}
```

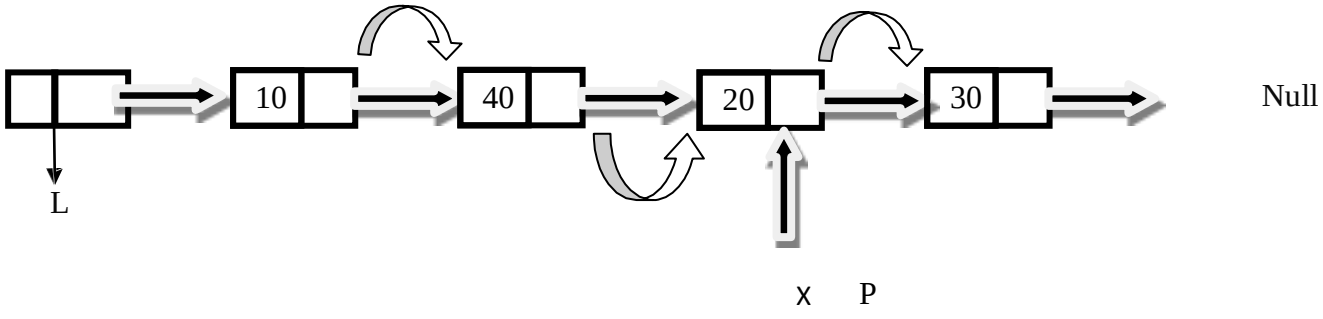


Routine to Find the Element in the List:

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);
}
```

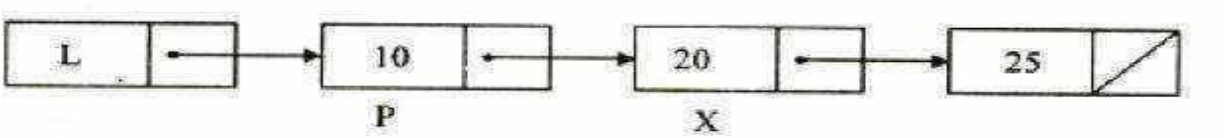
Find(List L, 20) - To find an element X traverse from the first node of the list and move to the next with the help of the address stored in the next field until data is equal to X or till the end of the list



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;
    }
```



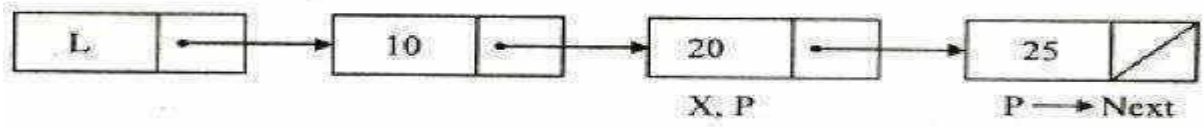
Routine to find next Element in the List

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

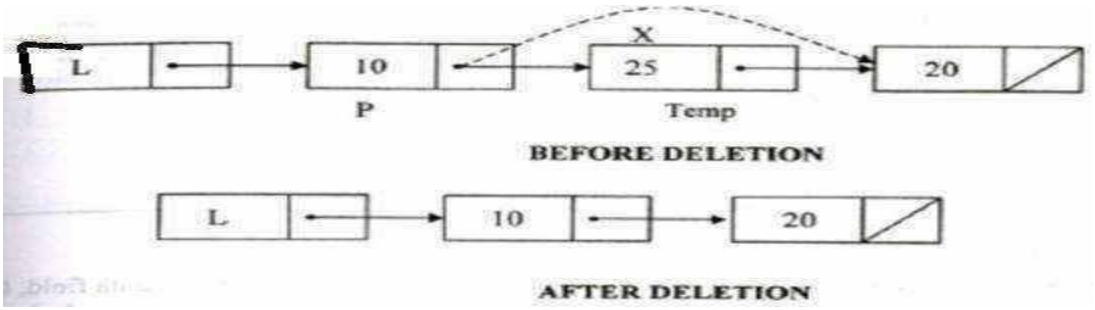
listvoid count(List L)

```
{
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

```
Lvoid 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 );
          }}
```

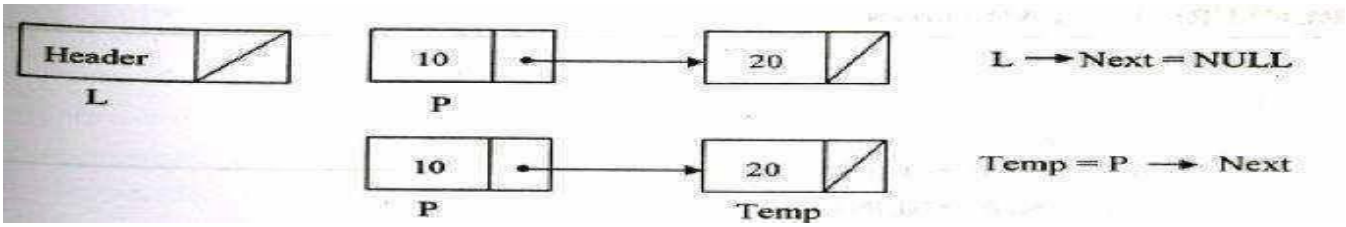


Routine to Delete the List

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;
        }
}
```



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

possible2.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 **DLL NODE**

DOUBLY LINKED LIST

Basic operations of a doubly -linked list are: Insert,Delete,Find,Prints the li

Declaration of DLL Node

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

Creation of list in DLL

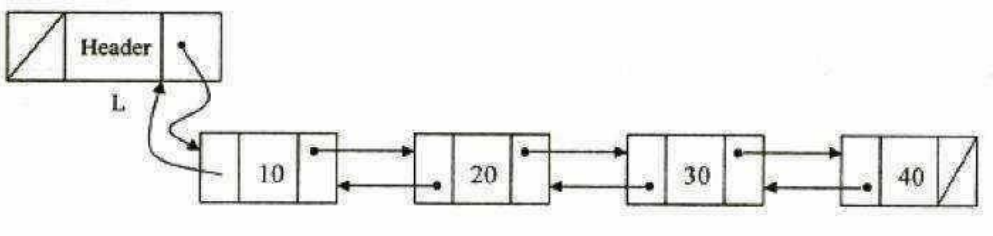
Initially the list is empty. Then assign the first node as

head.newnode->data=X;

newnode->next=NULL;newnode->prev=NULL;

L=newnode;

l L->next =newnode;
i
snewnode-prev=L
t

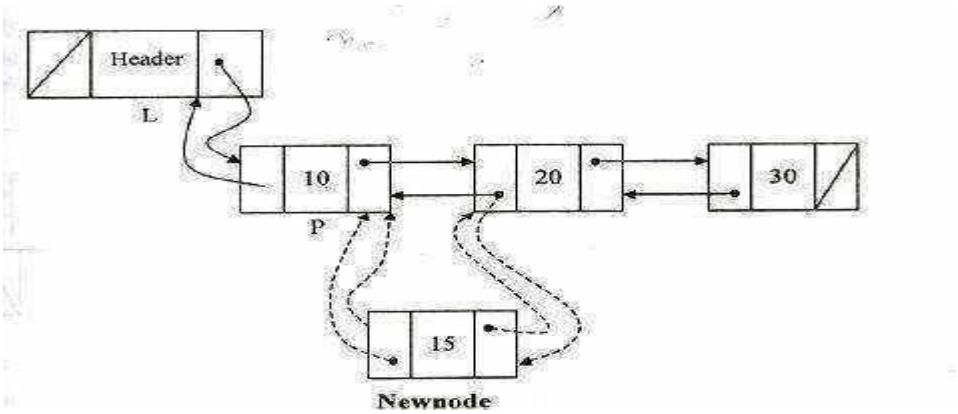


Routine to insert an element in a DLL at the beginning

```
void Insert (int x, list L, position P){
struct node  *Newnode;
if(pos==1)
P=L;
Newnode = (struc 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 = (struc 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:
}
```



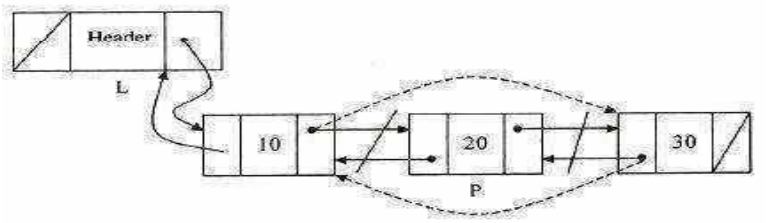
Routine to insert an element in a DLL at the end:

```
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");
}
}
```

Advantages of DLL:

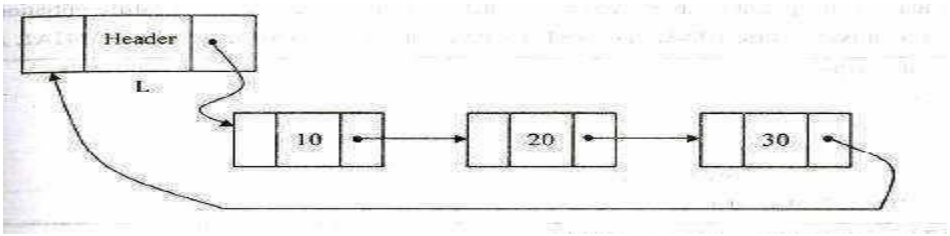
The DLL has two pointer fields. One field is prev link field and another is next link field. Because of these two pointer fields we can access any node efficiently whereas in SLL only onelink field is there which stores next node which makes accessing of any node difficult.

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.

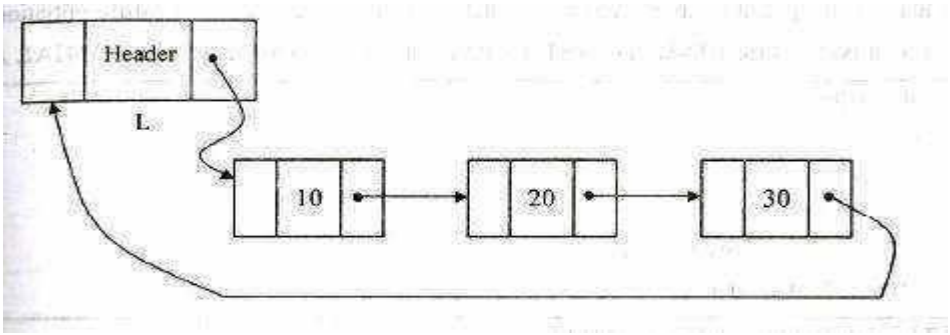


Types of CLL:

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 firstnode.

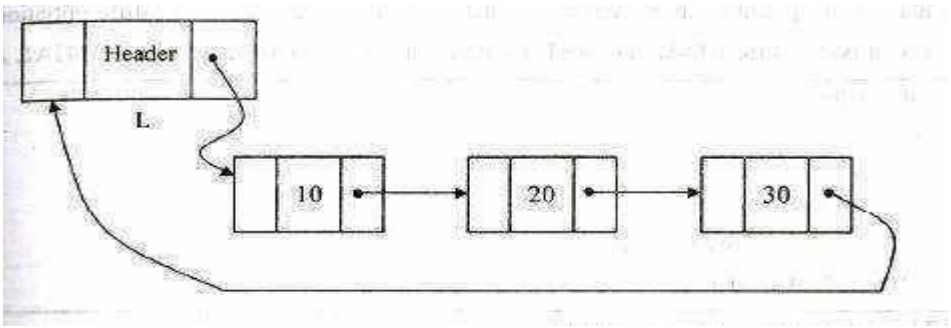


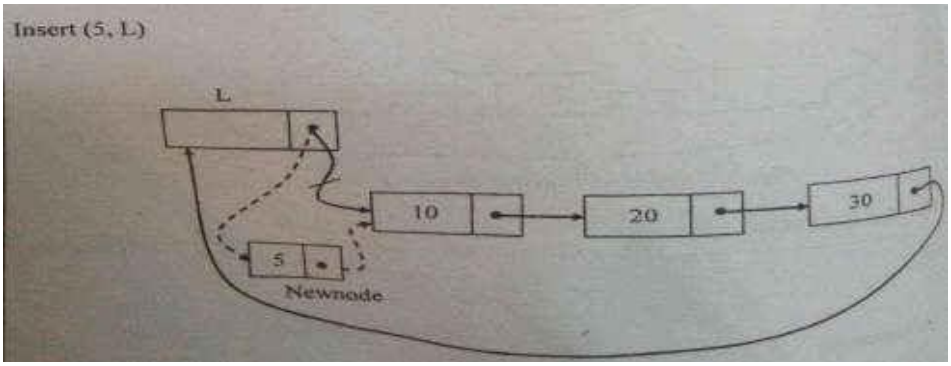
Declaration of node:

```
typedef struct node *position;

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

Routine to insert an element in the beginning

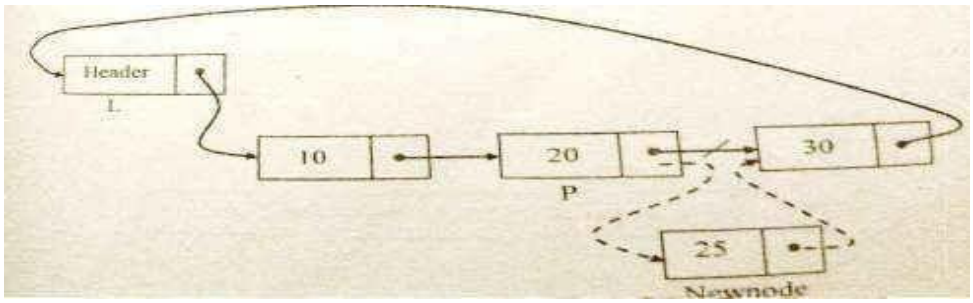
```
void insert_beg(int  X,List L)
 {
 position  Newnode;
 Newnode=(struct node*)malloc(sizeof(struct node));
 if(Newnode!=NULL)
 {
 Newnode->data=X;
 Newnode->next=L->next;
 L->next=Newnode;
 }
 }
```





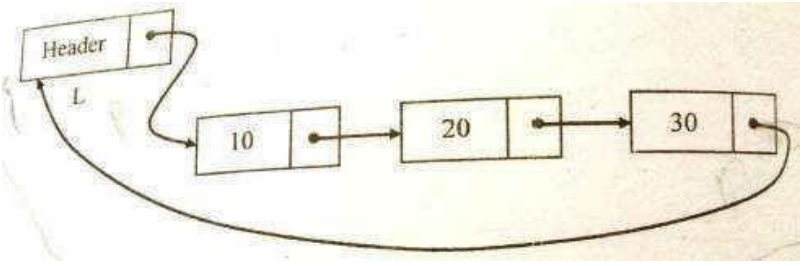
Routine to insert an element in the middle

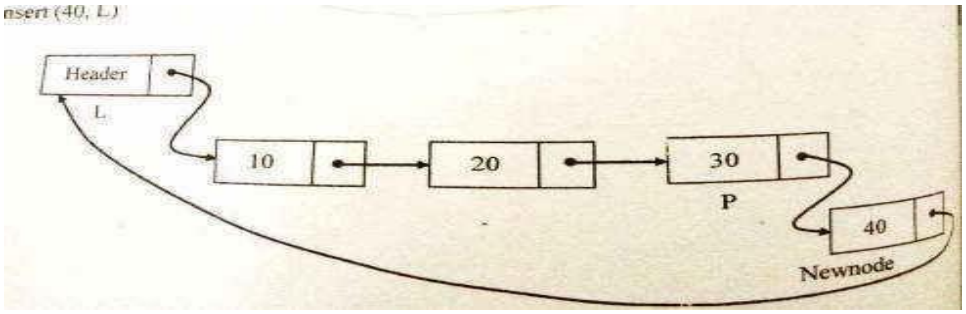
```
void insert_mid(int  X, List L, Position p)
{
position  Newnode;
Newnode=(struct node*)malloc(sizeof(struct node));
if(Newnode!=NULL)
{
Newnode->data=X;
Newnode->next=p->next;
p->next=Newnode;
}
}
```



Routine to insert an element in the last

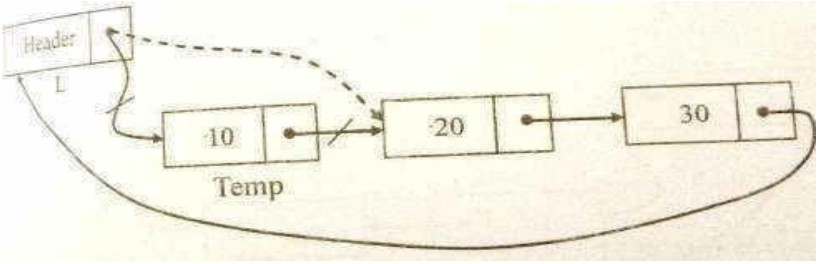
```
void insert_last(int  X, List L)
position  Newnode;
Newnode=(struct node*)malloc(sizeof(struct node));if(Newnode!=NULL)
{
P=L;
while(P->next!=L)
P=P->next;
Newnode->data=X;
P->next=Newnode;Newnode->next=L;
}
}
```

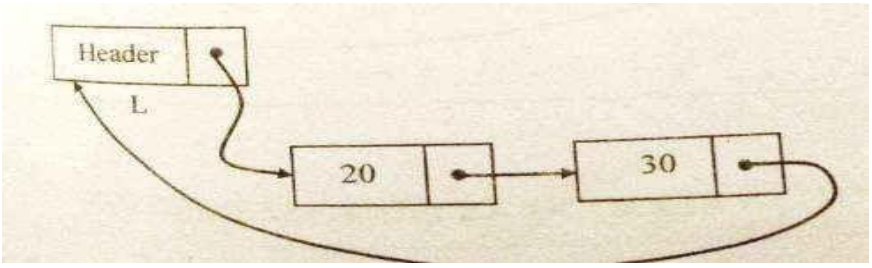




Routine to delete an element from the beginning

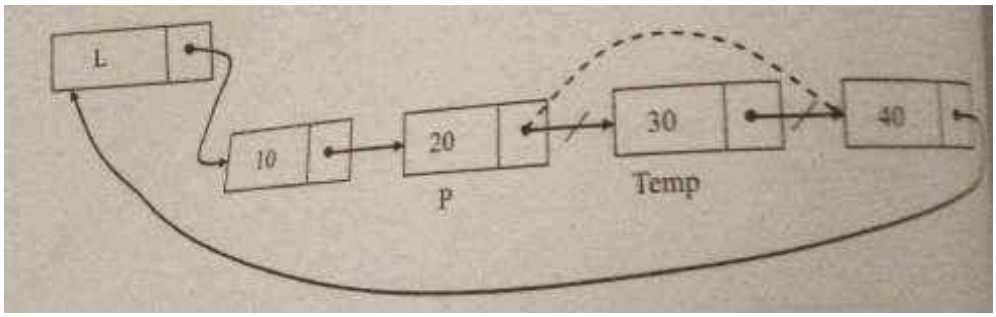
```
void del_first(List L)
{
position temp; temp=L->next;
L->next=temp->next;
free(temp);
}
```

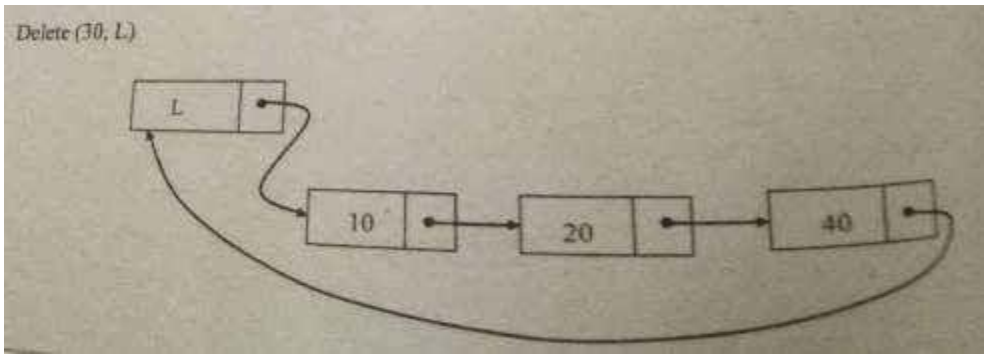




Routine to delete an element from the middle

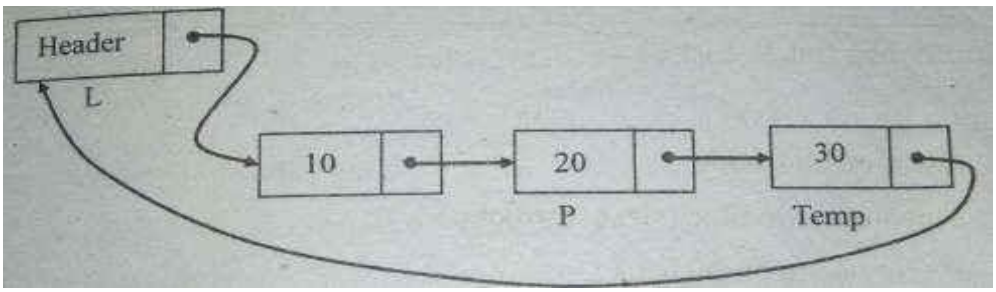
```
void del_mid(int X,List L)
{
position p, temp;
p=findprevious(X,L);
if(!Islast(P,L))
{
temp=p->next;
p->next=temp->next;
free(temp);
}
}
```

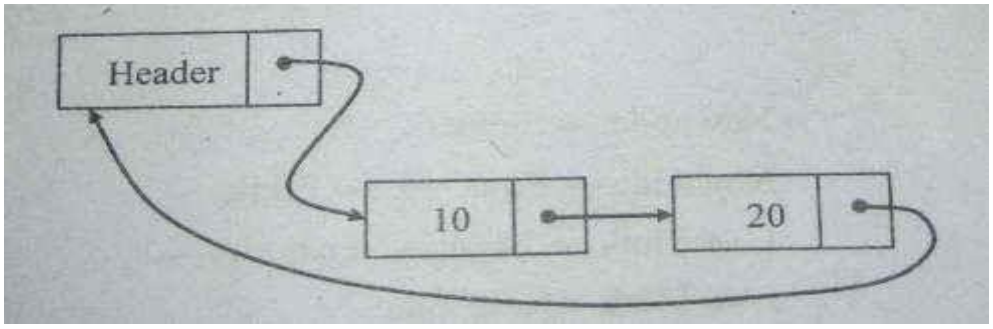




Routine to delete an element at the last position

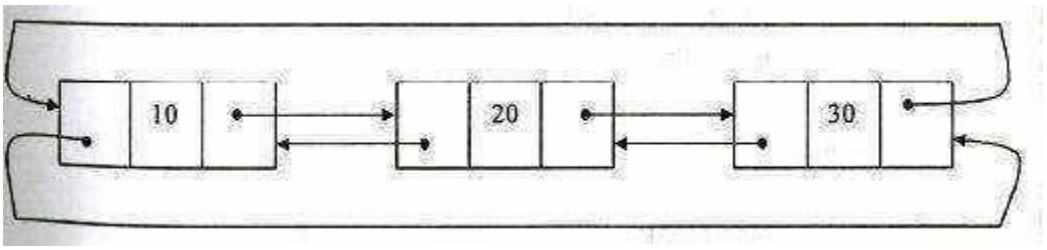
```
void del_last(List L)
{
position p, temp;p=L;
while(p->next->next!=L)
p=p->next;
temp=p->next;
p->next=L
free(temp);
}
```





Doubly Linked circular list:

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

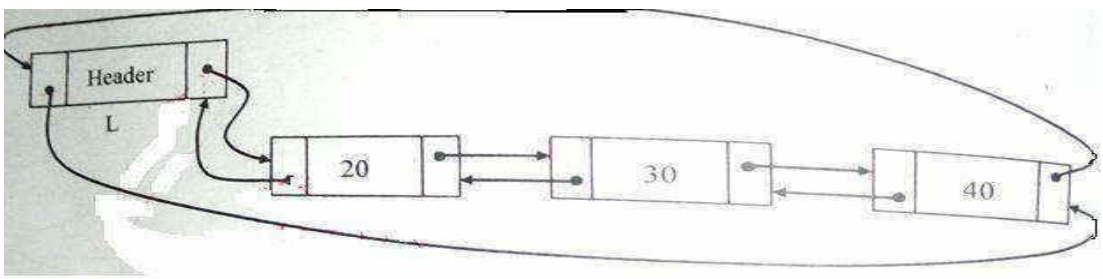


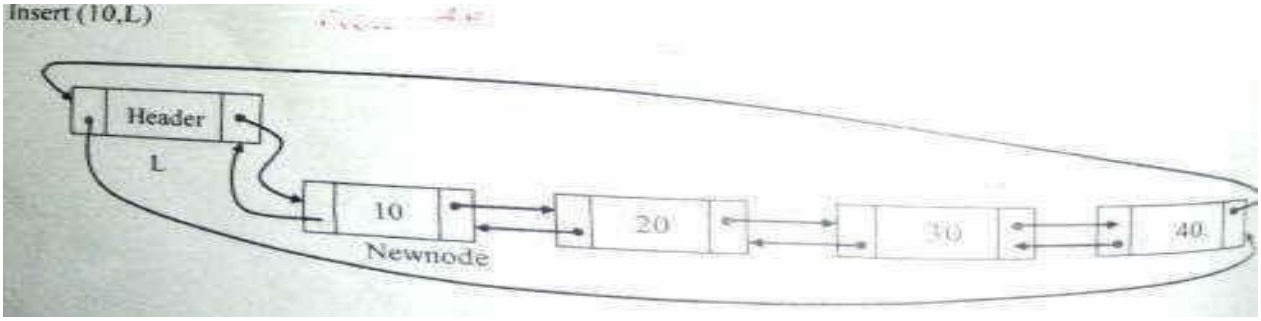
Declaration of node:

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

Routine to insert an element in the beginning

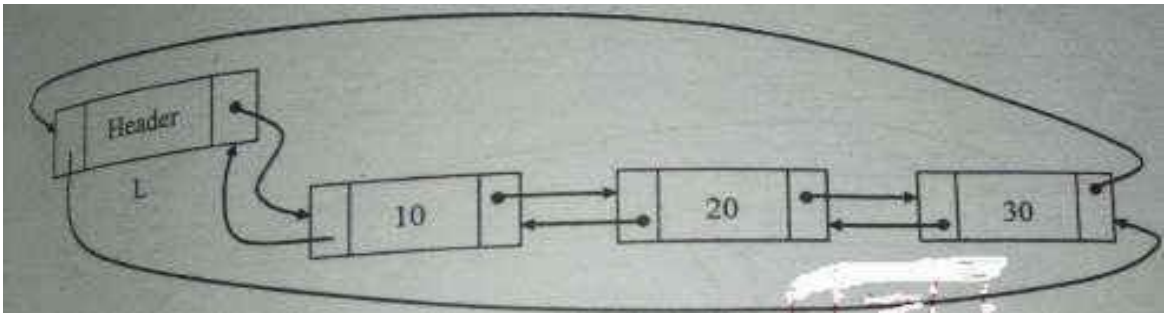
```
void insert_beg(int  X,List L)
{
position  Newnode;
Newnode=(struct node*)malloc(sizeof(struct node));
if(Newnode!=NULL)
{
Newnode->data=X;
Newnode->next=L->next;
L->next->prev=Newnode;
L->next=Newnode;
Newnode->prev=L;
}
}
```

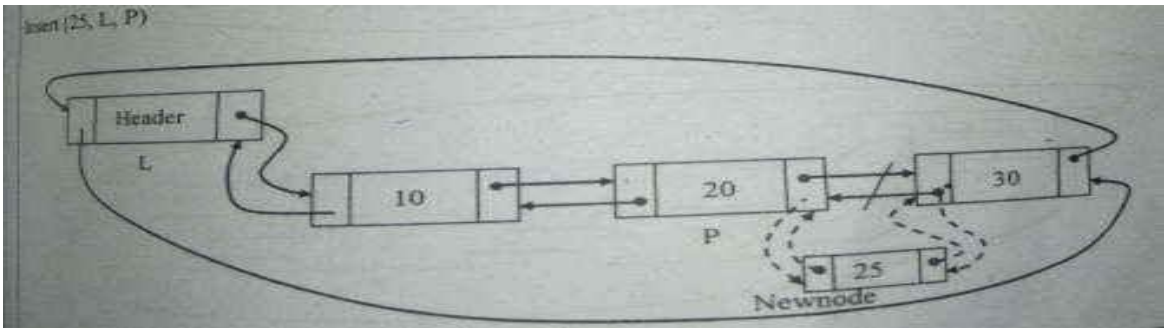




Routine to insert an element in the middle

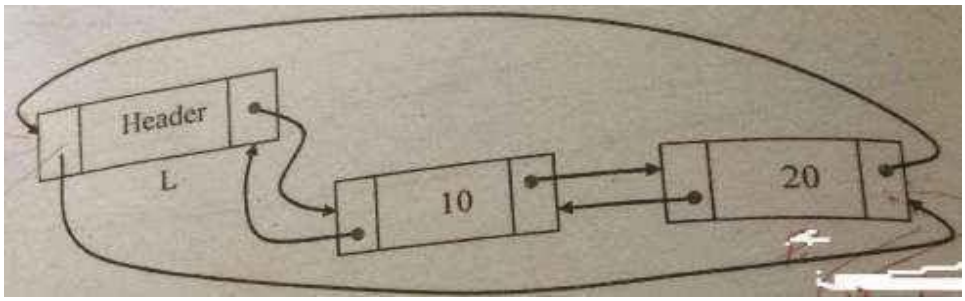
```
void insert_mid(int  X, List L, Position p)
{
position  Newnode;
Newnode=(struct node*)malloc(sizeof(struct node));
if(Newnode!=NULL)
{
Newnode->data=X;
Newnode->next=p->next;
p->next->prev=Newnode;
p->next=Newnode;
Newnode->prev=p;
}
}
```

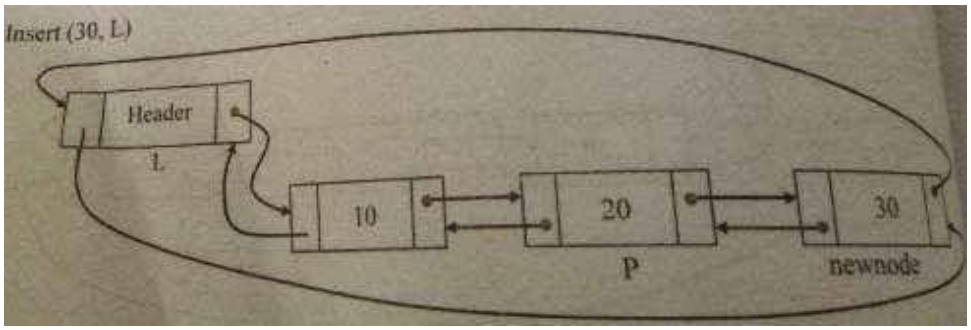




Routine to insert an element in the last

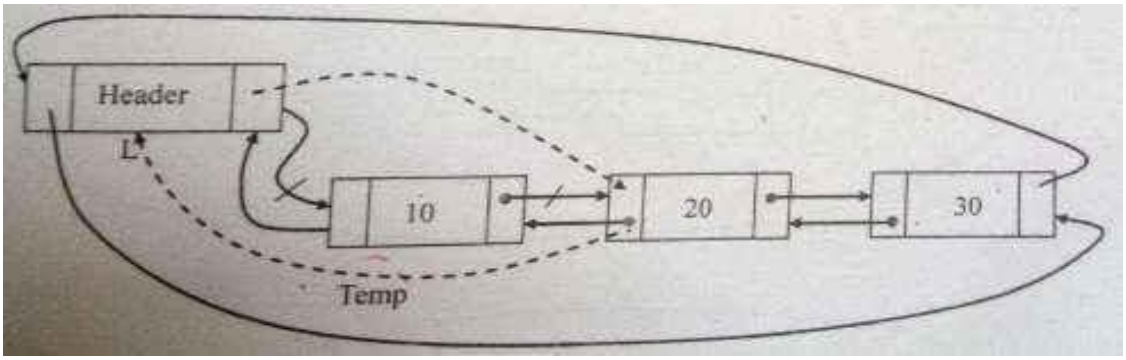
```
void insert_last(int  X, List L)
{
position  Newnode,p;
Newnode=(struct node*)malloc(sizeof(struct node));
if(Newnode!=NULL)
{ p=L;
while(p->next!=L)
p=p->next;
Newnode->data=X;
p->next =Newnode;
Newnode->next=L;
Newnode->prev=p;
L->prev=newnode;
}
}
```

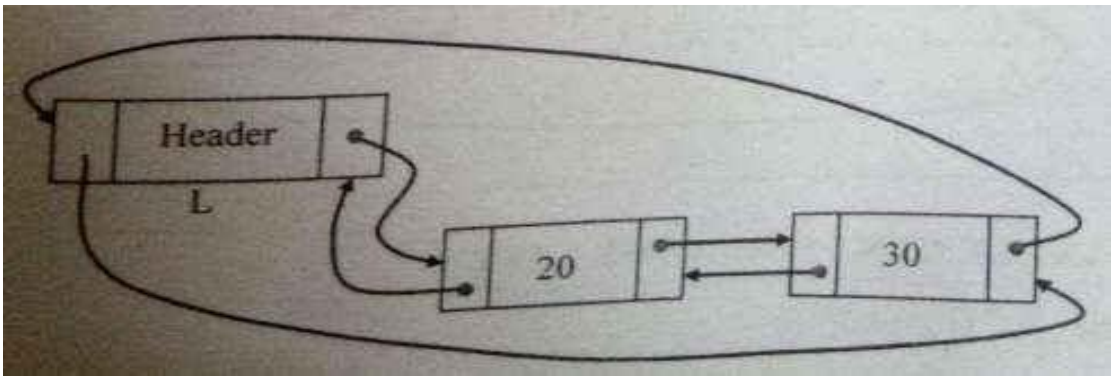




Routine to delete an element from the beginning

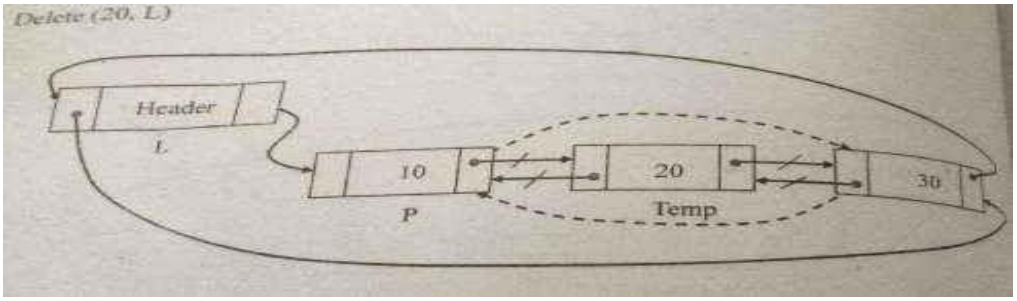
```
void del_first(List L)
{
position temp;
if(L->next!=NULL)
{
temp=L->next;
L->next=temp->next;
temp->next->prev=L;
free(temp);
}
```

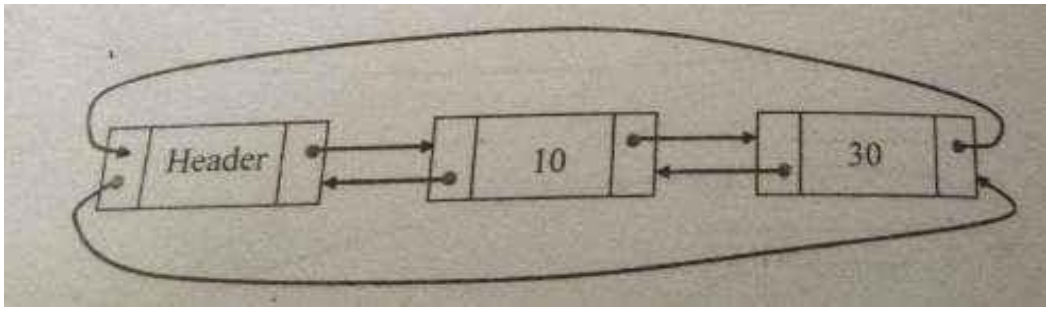




Routine to delete an element from the middle

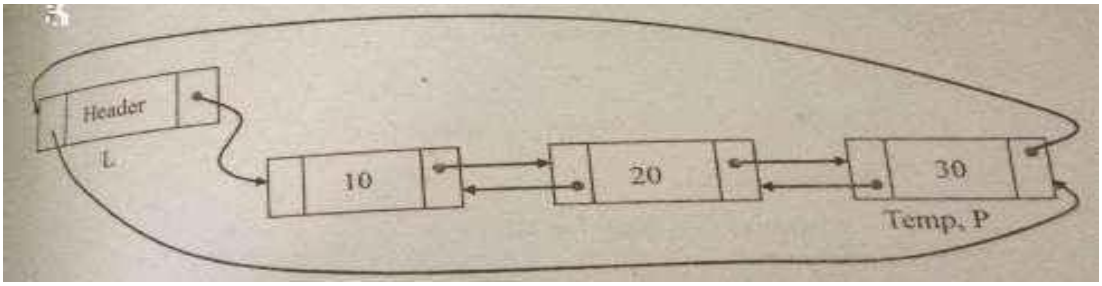
```
void del_mid(int X,List L)
{
Position p, temp;
p=FindPrevious(X);
if(!IsLast(p,L))
{
temp=p->next;
p->next=temp-next;
temp->next->prev=p;
free(temp);
}
}
```

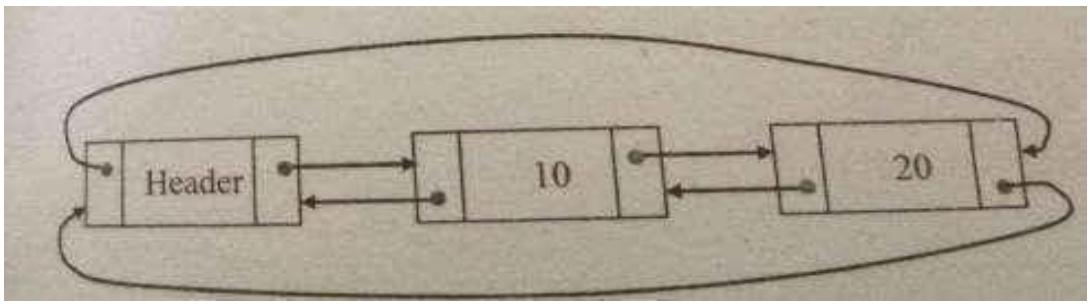




Routine to delete an element at the last position

```
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 beperformed 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)
   {

   }
   else
   {

   head1 = newnode1;
   return (head1);

   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;
         newnode->power = ptr1->power;
         newnode->next = NULL;
         list3 = create(list3, newnode);
         ptr1 = ptr1->next;
         ptr2 = ptr2->next;
      }
      else if(ptr1->power > ptr2->power)
      {
         // Intentionally left blank
      }
      else
      {
         // Intentionally left blank
         ptr2 = ptr2->next;
      }
   }

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

// Subtraction of two polynomials

```
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)
         {
            // Missing logic here
         }
         else
         {
            newnode->coeff = ptr1->coeff;
            newnode->power = ptr1->power;
            newnode->next = NULL;
            list3 = create(list3, newnode);
            ptr1 = ptr1->next;

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

Multi Lists

A "multi-list" or "multi-linked list" is a data structure where each node can have multiple pointers to other nodes, allowing for the organization of data in various ways simultaneously. This contrasts with simpler linked lists that typically have a single pointer per node. Multi-lists are useful for representing complex relationships or for sorting data based on multiple criteria.

Here's a breakdown of key aspects:

Key Features:

- **Multiple Pointers:** Each node in a multi-linked list can have multiple pointers, each potentially pointing to a different node based on a specific order or relationship.
- **Multiple Orderings:** Multi-lists are commonly used to represent the same data sorted in multiple ways (e.g., alphabetically and by age).
- **Flexibility:** They offer flexibility in how data is accessed and organized, allowing for different traversals based on the chosen pointer.

Common Use Cases:

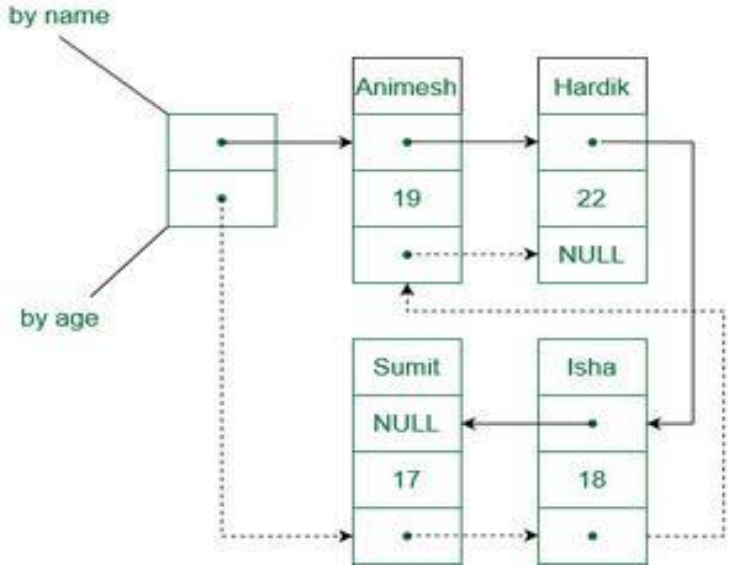
- **Representing Sparse Matrices:** Multi-lists can efficiently store sparse matrices (matrices with mostly zero values) by only storing the non-zero elements and using pointers to link them in rows and columns.
- **Managing Multiple Orders:** As mentioned, multi-lists are ideal for situations where data needs to be accessed and sorted based on multiple criteria.
- **Graph Representation:** Multi-lists can be used to represent graphs, where nodes are connected by multiple edges.

Example: Imagine storing a list of students with their names and ages. A multi-linked list could be used to:

1. **Sort alphabetically:** One pointer would link students in alphabetical order by name.
2. **Sort by age:** Another pointer would link students in ascending or descending order by age.

This allows for quick access to students either alphabetically or by age without having to re-sort the data.

In essence, multi-lists provide a powerful way to manage and access data organized in multiple ways within a single structure.



Applications of lists

□ Academic & Computational Applications

- **Polynomial Manipulation**: Lists (especially linked lists) are used to represent and operate on polynomials efficiently—ideal for addition, subtraction, and multiplication of terms.
- **Sparse Matrix Representation**: Lists help store only non-zero elements, saving memory and improving performance.
- **Symbol Tables in Compilers**: Lists store identifiers and their attributes during compilation.
- **Dynamic Memory Allocation**: Linked lists track free and allocated memory blocks in operating systems.

📊 Data Structure Implementations

- **Stacks and Queues**: Lists serve as the backbone for implementing these linear structures.
- **Graphs (Adjacency Lists)**: Lists represent graph connections efficiently, especially in sparse graphs.
- **Hash Tables (Chaining)**: Lists handle collisions by storing multiple entries at the same index.

□ Real-World Applications

- **Undo/Redo Functionality**: Doubly linked lists allow navigation between previous and next states in editors.
- **Browser History**: Pages are stored in a list to enable back and forward navigation.
- **Music & Video Playlists**: Songs or videos are linked for sequential or looped playback.
- **Image Viewers**: Navigate through images using next/previous pointers.

🛠️ System-Level Applications

- **File Systems**: Lists represent hierarchical structures of directories and files.
- **Process Scheduling**: Operating systems use lists to manage active and waiting processes.
- **Virtual Memory Management**: Lists track memory pages and their states.

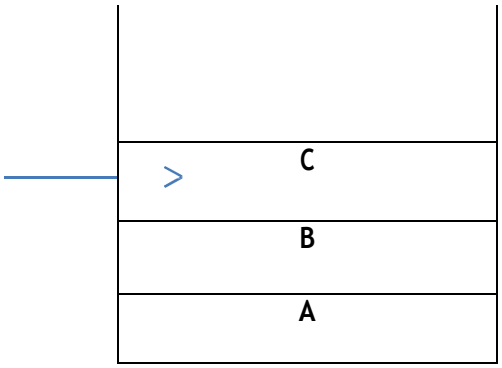
UNIT II– STACKS AND QUEUES

Stack ADT – Operations – Applications: Balancing Symbols - Evaluating arithmetic expressions- Infix to postfix conversion - Queue ADT – Operations - Circular Queue –Dequeue- Priority Queue - 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:



STACK MODEL

TOP pointer

It will always point to the last element inserted in the stack.For empty stack, top will be pointing to -1. (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. (Top = 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. (Top = 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 stackoverflow.
    - 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 stackunderflow.
    - 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, Top = -1.
- Stack is declared with its maximum size.

Array Declaration of Stack:

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

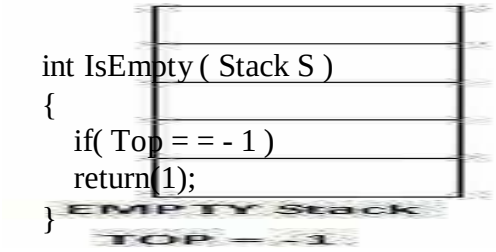
Stack Empty Operation:

- 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);
}
```

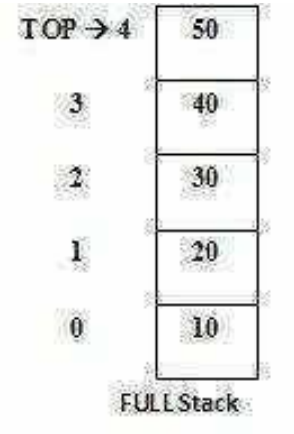


(i) Stack Full Operation:

- 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 newelement into the stack.

Routine to check whether a stack is full

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

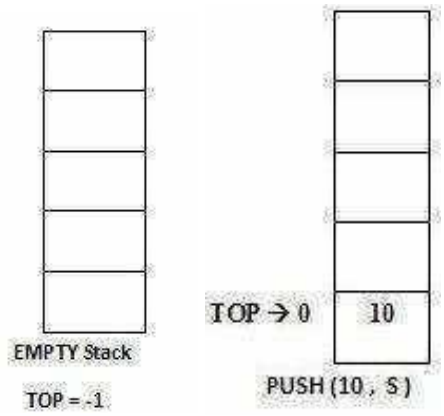


(ii) Push Operation

- 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 StackS.
- 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 theStack.

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;
    }
}
```

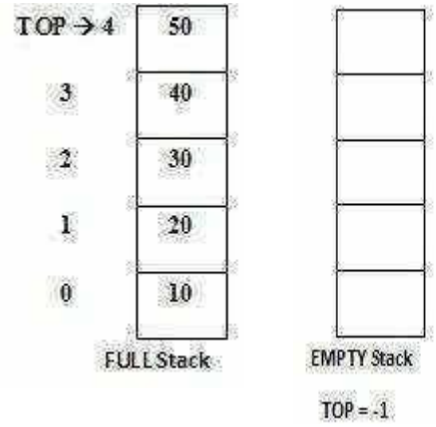


(iv) Pop Operation

- 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 theStack.
- 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

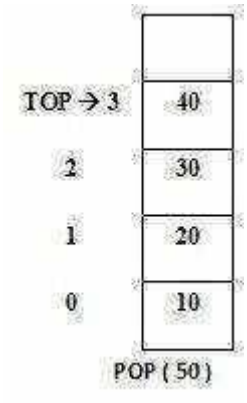
- Pop routine deletes the Top element in the stack.

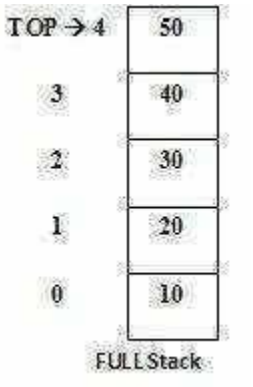
If the user needs to know the last element inserted into the stack, then the user can returnthe Top element of the stack

- 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];
}
```





(i) Stack Empty Operation:

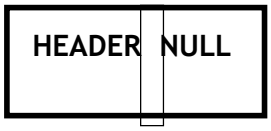
• Initially Stack is Empty.

• With Linked List implementation, Empty stack is represented as S -> next = 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

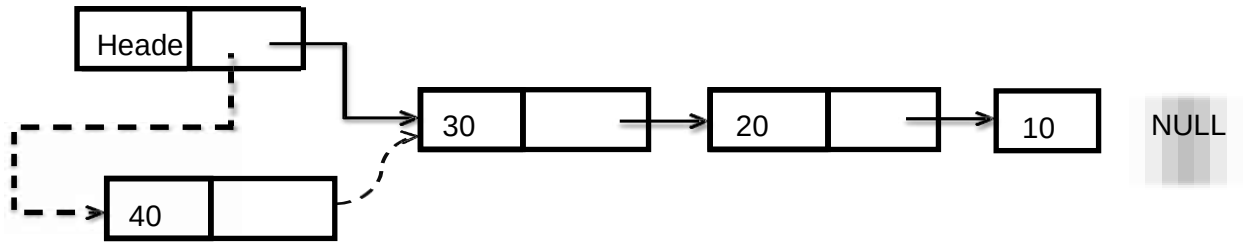
```
int IsEmpty( Stack S)
{
if ( S -> next = = NULL)
  return ( 1 );
}
```



EMPTY STACK

(i) Push Operation :

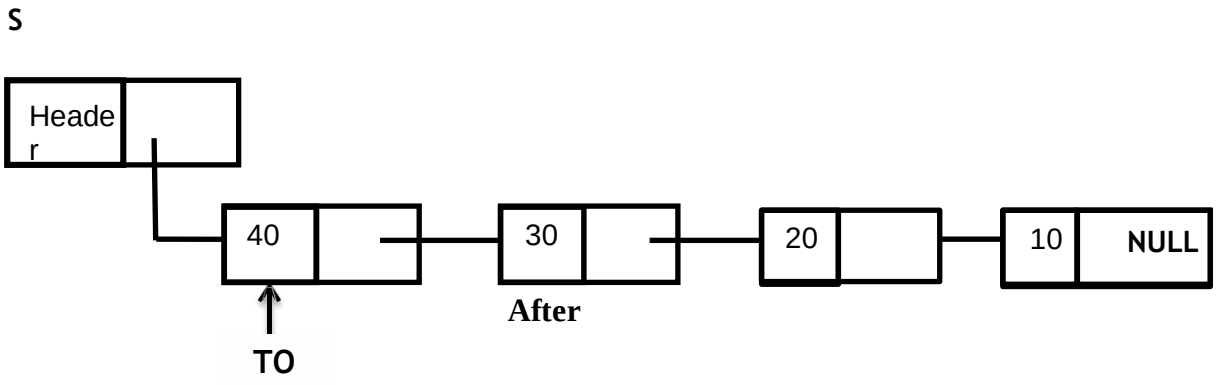
- With Linked List implementation, a new element is always inserted at the Front of theList.(i.e.) S -> next.
- It takes two parameters. Push(X, S) the element X to be inserted at the Top of the StackS.
- Allocate the memory for the newnode to be inserted.
- Insert the element in the data field of the newnode.
- Update the next field of the newnode with the address of the next node



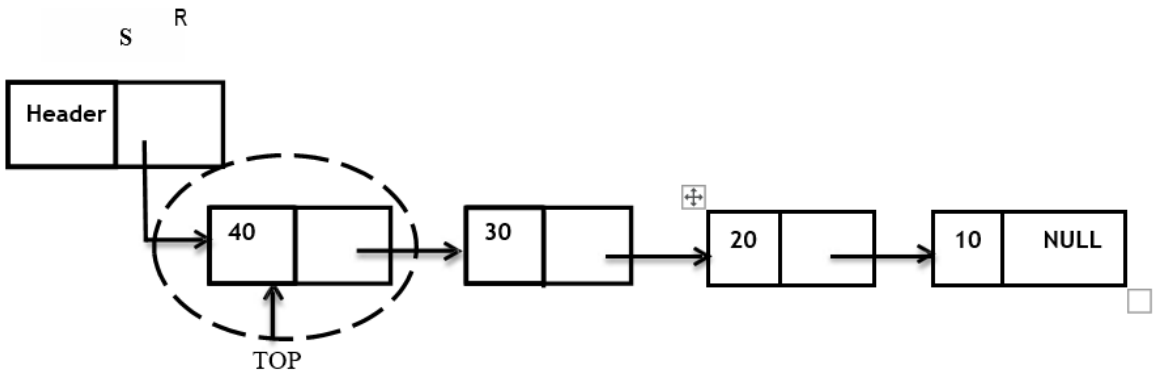
Before Insertion

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;
}
```



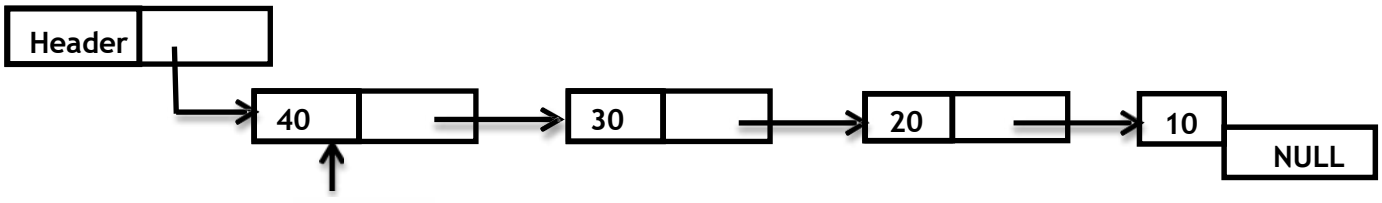
- **Pop Operation :**It is the process of deleting the Top element of the stack.
- With Linked List implementations, the element at the Front of the List (i.e.) S -> next is always deleted.
- It takes only one parameter. Pop(X).The element X to be deleted from the Front of the List.
- Before deleting 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.



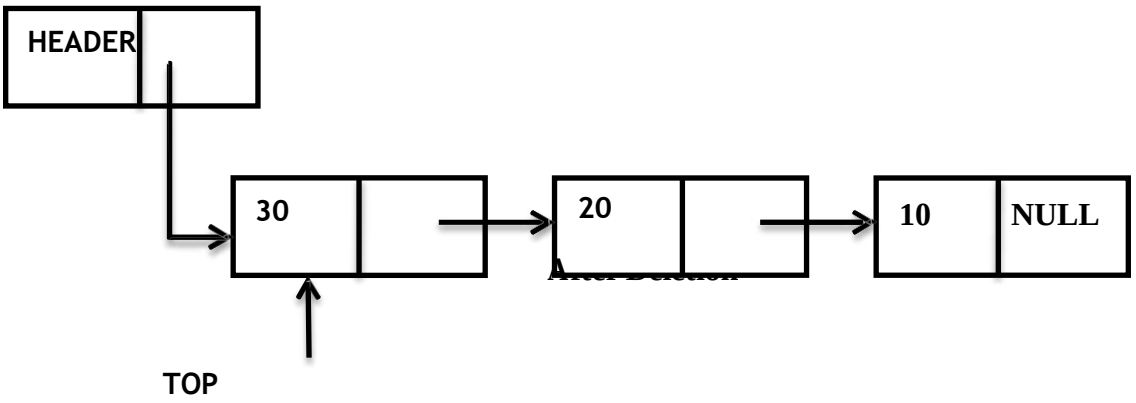
Before Deletion

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;
         } }
```



S

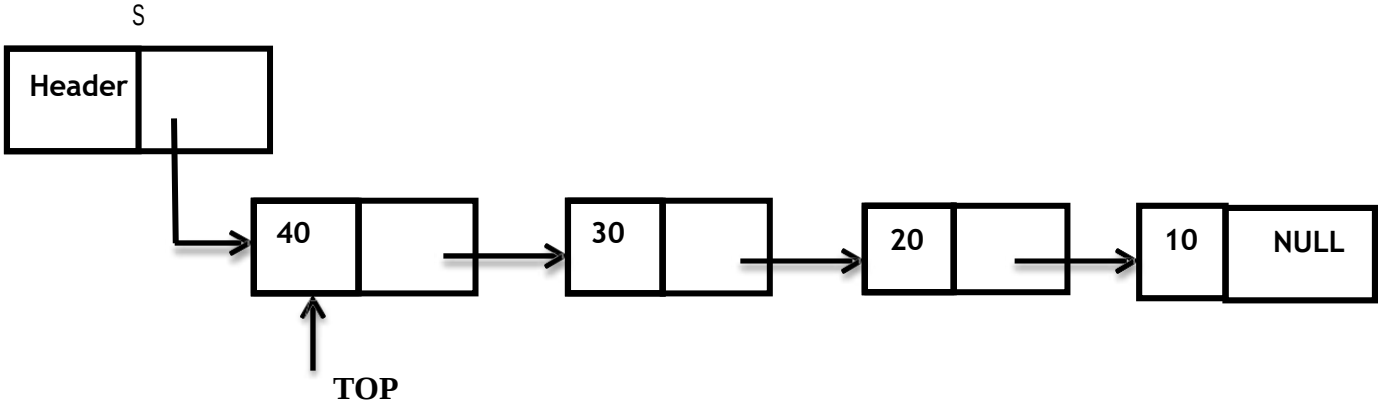


(ii) Return Top Element

- 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 -> next -> 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;
}
```



Applications of Stack

The following are some of the applications of stack:

1. Evaluating the arithmetic expressions
    - Conversion of Infix to Postfix Expression
    - 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 beingapplied.

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

PREFIX:
The arithmetic operator is placed before the two operands to which it applies. Alsocalled polish notation.

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 "#"

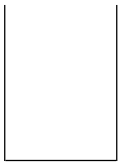
Step 1: If the character is an operand, place it on the output.

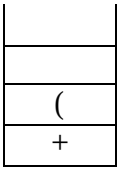
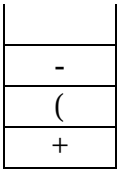
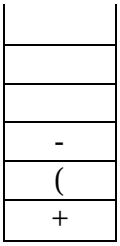
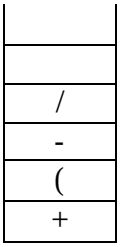
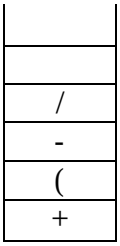
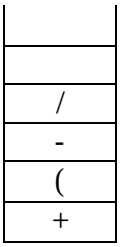
Step 2: If the character is an operator, push it onto the stack. If the stack operator has a higher or equal priority than input operator then pop that operator from the stack and place it onto the output.

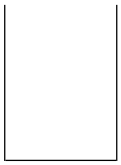
Step 3:If the character is left parenthesis, push it onto the stack

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		AB*C
-		AB*C
D		AB*CD
/		AB*CD
E		AB*CDE
)		AB*CDE/-

Read char	Stack	Out Put
#		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 tothem 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**

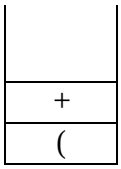
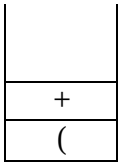
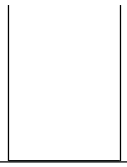
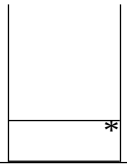
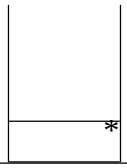
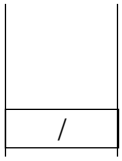
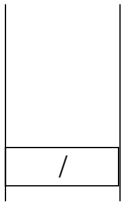
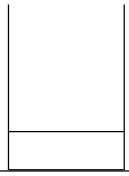
Char Read	Stack
A	2
B	3 2

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

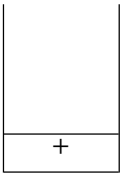
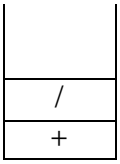
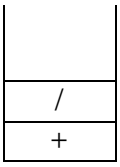
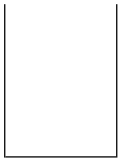
OUTPUT = 8

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

Read char	Stack	Output
(	(	
a	(	a

+		a
b		ab
)		ab+
*		ab+
c		ab+c
/		ab+c*
d		ab+c*d
+		ab+c*d/

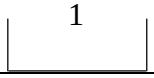
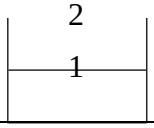
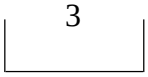
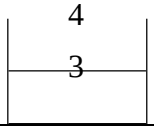
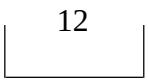
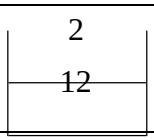
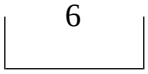
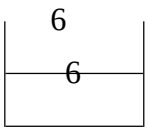
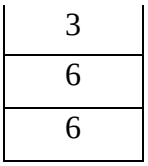
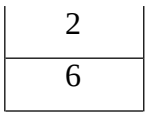
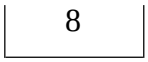
+

e		ab+c*d/e
/		ab+c*d/e
f		ab+c*d/ef
#		ab+c*d/ef/+

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	
b	
+	
c	
*****	
d	
/	
e	
F	
/	
+	

Output = 8

Recursive Solution

Towers of Hanoi

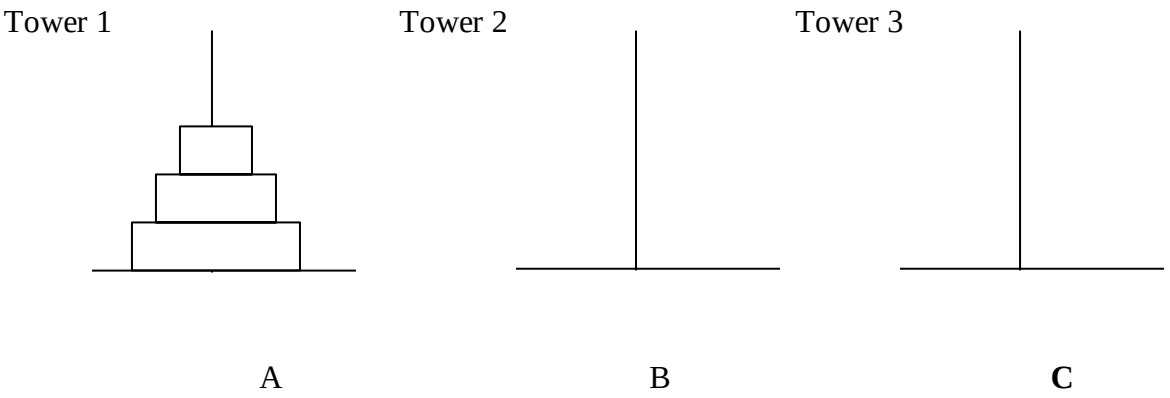
Towers of Hanoi can be easily implemented using recursion. Objective of the problem is moving a collection of N disks of decreasing size from one pillar to another pillar. The movement of the disk is restricted by the following rules.

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 theywere

being moved from source to destination.



Initial Setup of Tower of Hanoi

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 1st disk from A to B. Then move the 2nd disk from A to C, The movethe 1st disk from B to C.

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

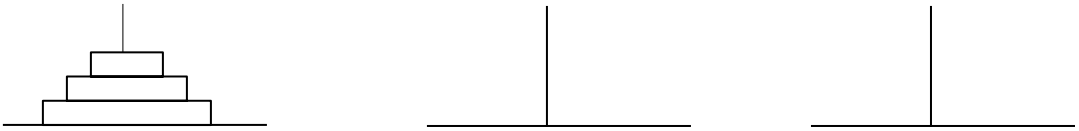
In general, to move N disks. Apply the recursive technique to move N - 1 disks from A to B using C as an intermediate. Then move the Nth disk from A to C. Then again apply the recursive technique to move N - 1 disks from B to C using A as an intermediate

Recursive Routine for Towers of Hanoi

Recursive Solution

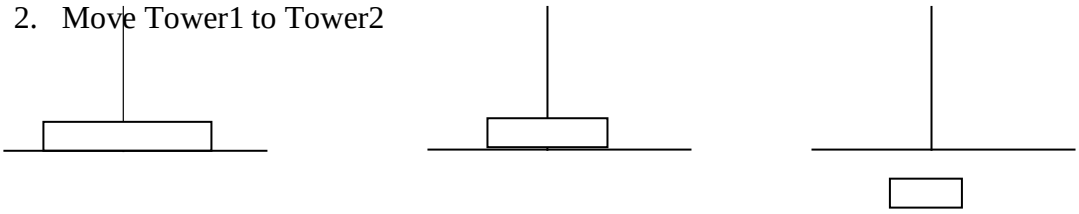
```
void hanoi (int n, char s, char d, char i)
{
/* n     no. of disks, s    source, d    destination i    intermediate
*/ if (n = = 1)
{
print (s, d);
return;
}
else
{
hanoi (n - 1, s, i, d);
print (s, d)
hanoi (n-1, i, d, s);
return;
}
}
```

Source Pillar Intermediate Pillar Destination Pillar

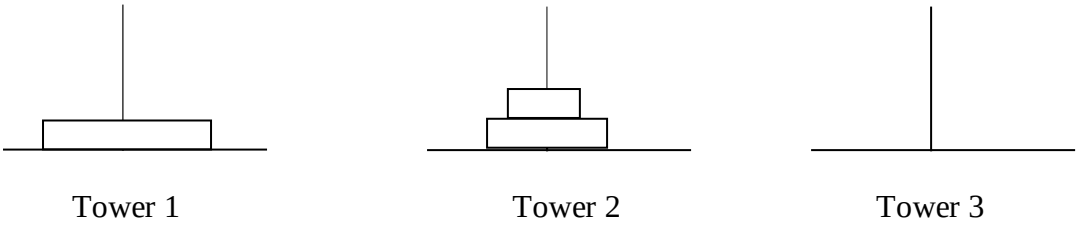


1. Move Tower1 to Tower3

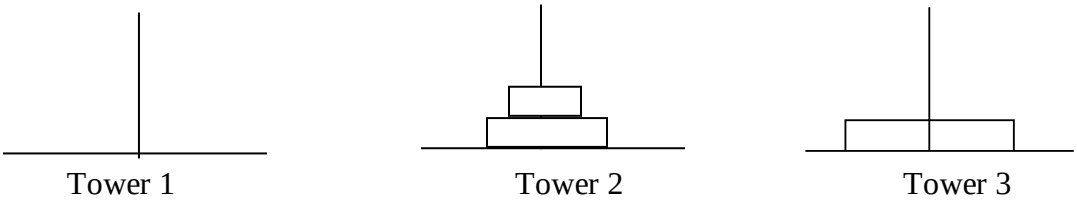
2. Move Tower1 to Tower2



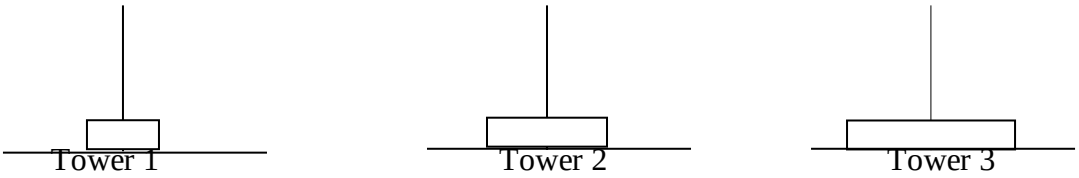
3. Move Tower 3 to Tower 2



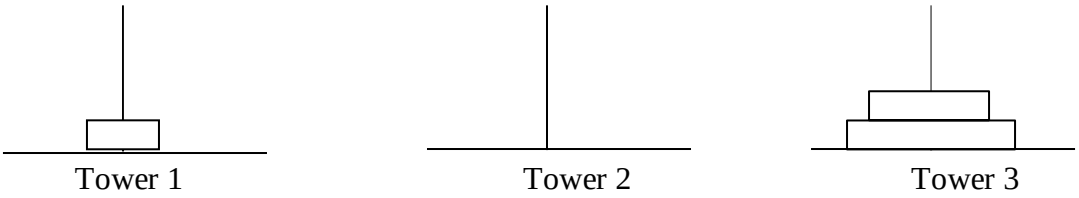
4. Move Tower 1 to Tower 3



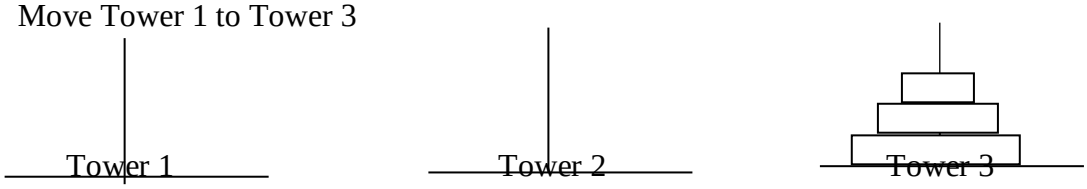
5. Move Tower 2 to Tower 1



6. Move Tower 2 to Tower 3



7. Move Tower 1 to Tower 3

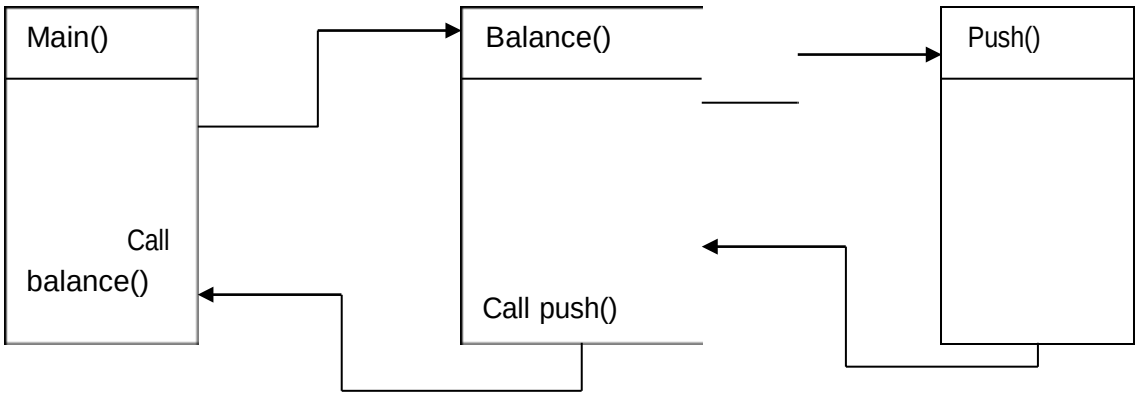


Since disks are moved from each tower in a LIFO manner, each tower may be consideredas a Stack. Least Number of moves required solving the problem according to our algorithm is given by,

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

Function Calls

When a call is made to a new function all the variables local to the calling routine need tobe 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 goafter 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 asmissing opening symbol.

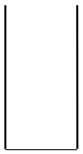
Step 3 : - If it is a closing symbol and if it has corresponding opening symbol in the stack, POPit from the stack. Otherwise, report an error as mismatched symbols.

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:

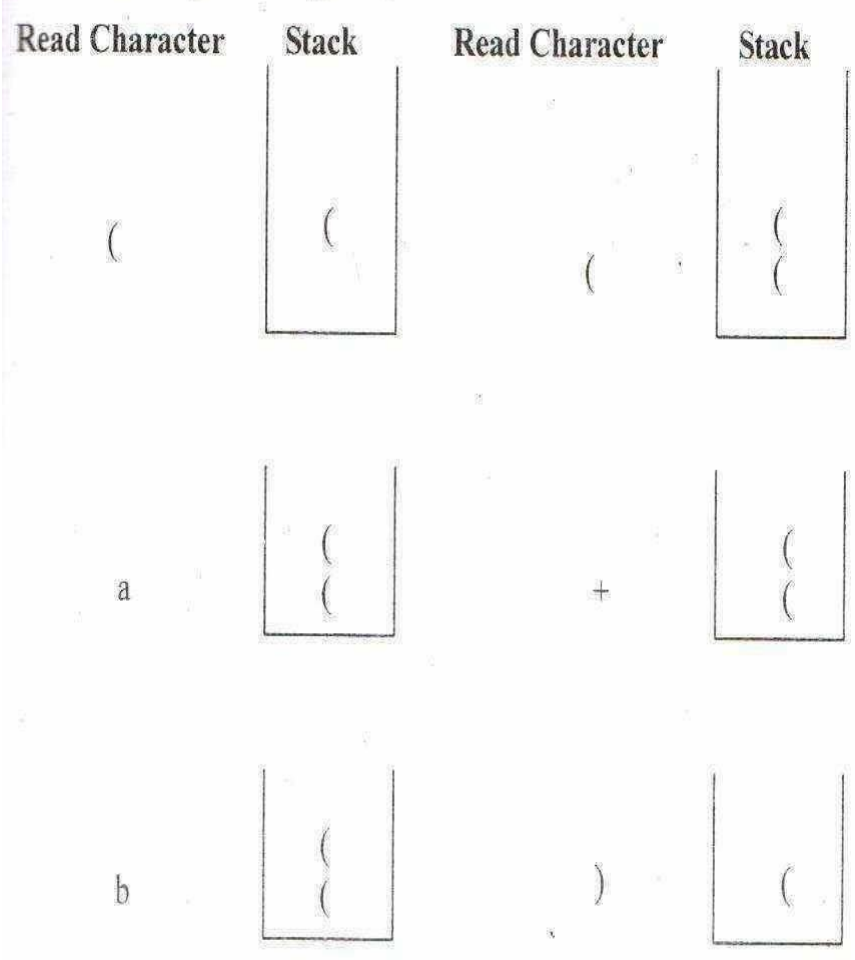
E.g. Let us consider the expression ((B*B)-{4*A*C}/[2*A]) #

((B*B)-{4*A*C}/[2*A]) #	
Read Character	**Stack**
(	(
(	((
)	 (
{	{ (
}	 (

[	[(
]	 (
)	

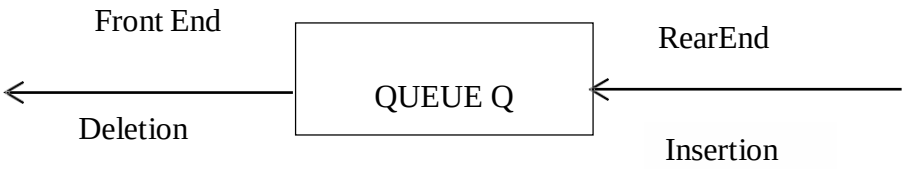
Empty stack, hence the symbols the balanced in the given expression.

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



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.



Queue Model

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

1. 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
  o Check for Full Queue
  o If the Queue is full, Insertion is not possible.
  o 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 thefront by 1.

Exceptional Conditions of Queue

(i) Queue Overflow:

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

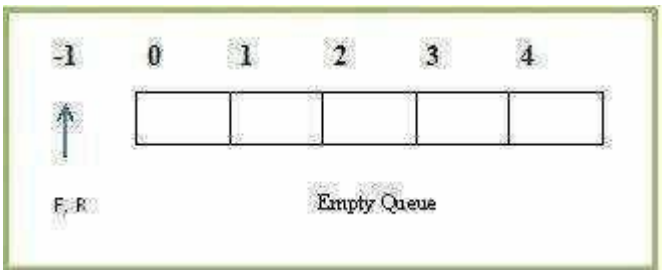
Queue can be implemented in two ways.

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:

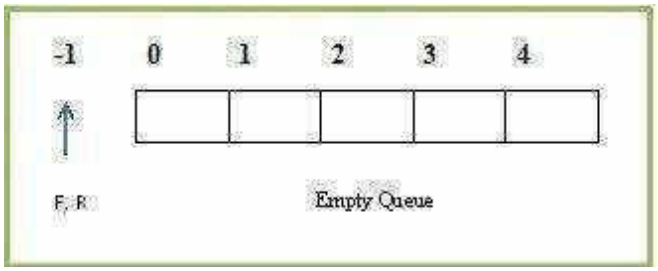


(i) Queue Empty Operation:

- 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).

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

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

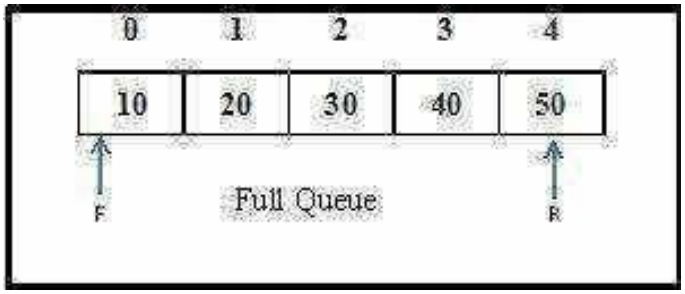


(ii) Queue Full Operation

- 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 );
}
```

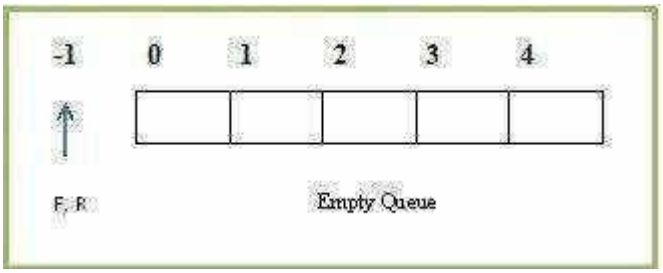


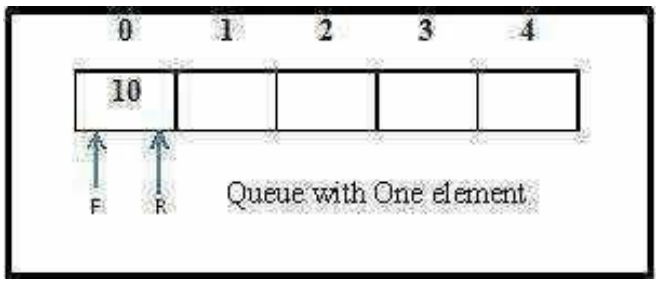
(iii) Enqueue Operation

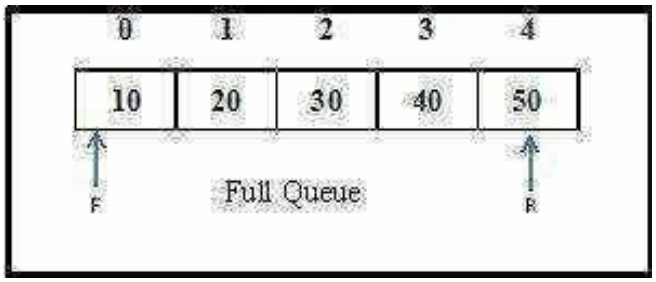
- It is the process of inserting a new element at the Rear end of the Queue.
- It takes two parameters, Enqueue(X, Q). The elements X to be inserted at the Rear end ofthe Queue Q.
- Before inserting a new Element into the Queue, check for Full Queue.If the
- Queue is already Full, Insertion is not possible.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

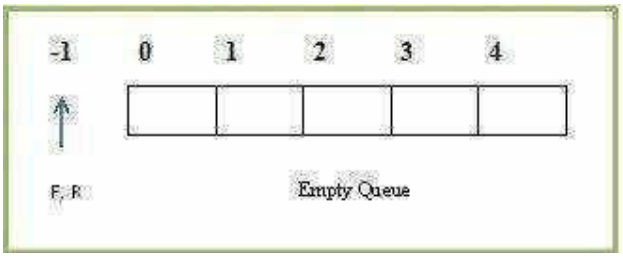






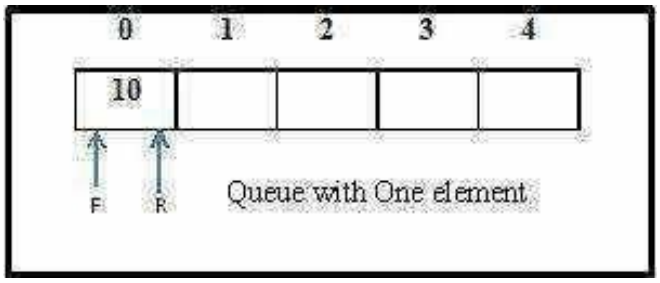
(iv) DeQueue Operation

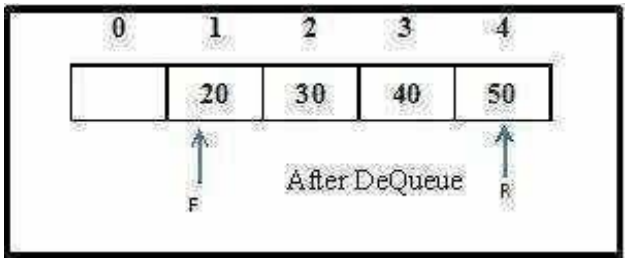
- It is the process of deleting a element from the Front end of the Queue.
- It takes one parameter, DeQueue (Q). Always front element in the Queue will be deleted.Before
- deleting an Element from the Queue, check for Empty Queue.
- If the Queue is empty, deletion is not possible.
- If the Queue has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





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 DeQueueoperation 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

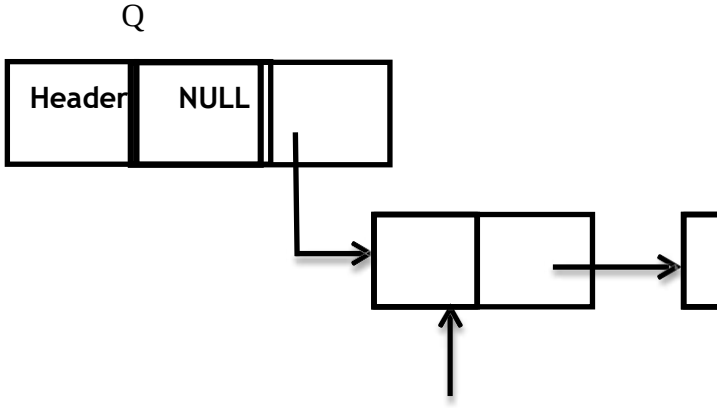
Declaration for Linked List Implementation of Queue ADT

```
struct node
{
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 -> next = 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);
}
```

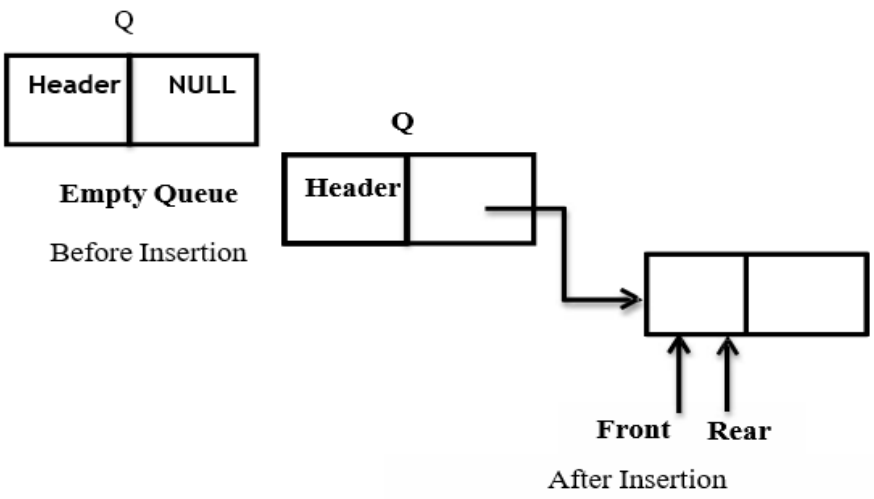


(i) EnQueue Operation

- 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 intothe 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 thelist. Hence Front and Rear points to the newnode.
- Otherwise insert the newnode in the Rear -> next and update the Rear pointer.

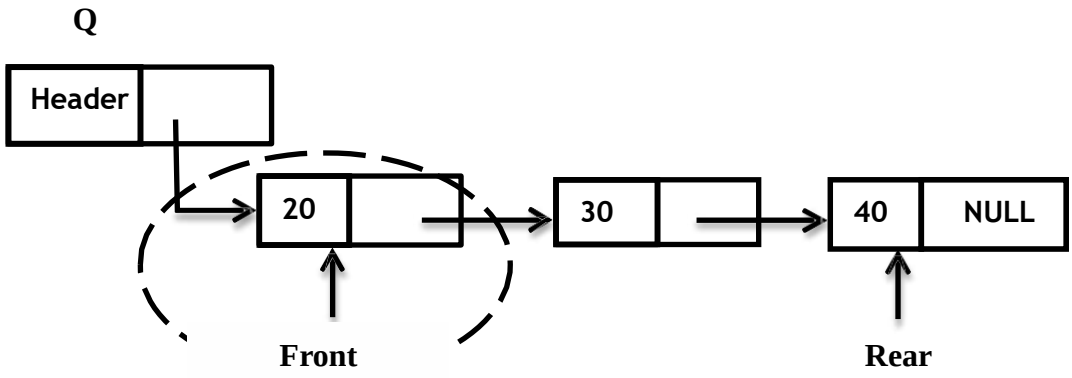
Routine to EnQueue an Element in Queue

```
void EnQueue(int X, Queue Q)
{
   struct node *newnode;
   newnode = malloc(sizeof(struct node));

   newnode->data = X;        // Assuming struct node has a 'data' field
   newnode->next = NULL;

   if (Rear == NULL)
   {
      Q->next = newnode;
      Front = newnode;
      Rear = newnode;
   }
   else
   {
      Rear->next = newnode;
      Rear = newnode;
   }
}
```



(ii) 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 pointedby Q -> 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 ismade 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);
}

}

   }
```

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 anorder, until a service representative is free.
3. Handling of interrupts in real-time systems. The interrupts are handled in the same orderas 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 bymoving Front = 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.

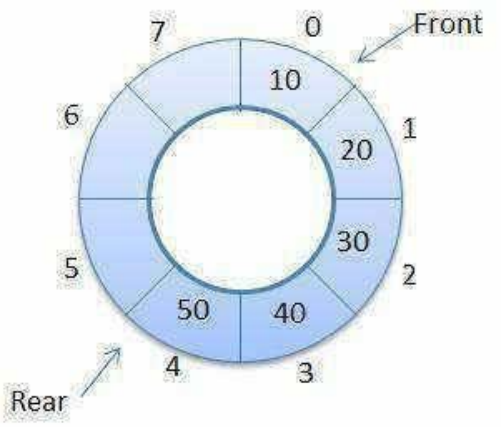
• A circular queue is an abstract data type that contains a collection of data which allows addition of data at the end of the queue and removal of data at the beginning of the queue.

• 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 Rear = (Rear + 1) % ArraySize

For Dequeue Front = (Front + 1) % 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;
    }
}
```

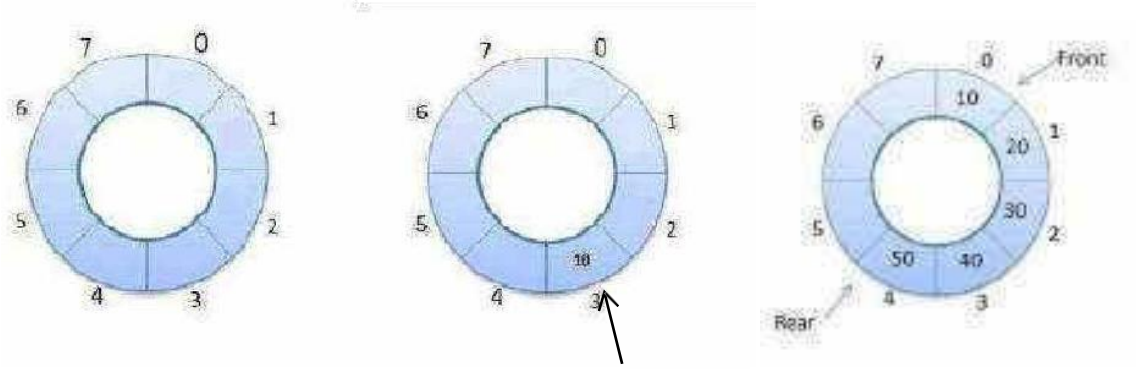
Circular Queue DeQueue Operation

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 Rearpointer 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 theCircular Queue.



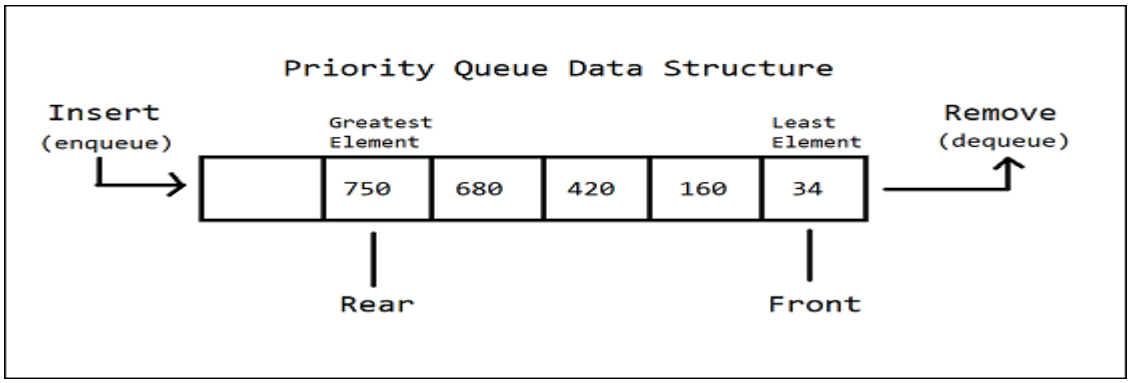
F= -1,R= --1

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;
   }
}
```

PRIORITY QUEUE

A priority queue is a data structure where each element has an associated priority, and elements are processed based on their priority rather than their order of insertion. Higher priority elements are processed before lower priority elements. If elements have the same priority, they are typically processed in a first-in-first-out (FIFO) manner.



Key Concepts:

Priority: Each element in a priority queue has a priority value. This value determines the order in which elements are processed.

Ordering: Elements with higher priority are processed before elements with lower priority.

- **FIFO (First-In, First-Out) for Equal Priority:**

If two or more elements have the same priority, the one that was added to the queue first is processed first.

Common Operations:

- enqueue(element, priority): Adds an element to the queue with a specified priority.
- dequeue(): Removes and returns the element with the highest priority.
- peek(): Returns the element with the highest priority without removing it.
- isEmpty(): Checks if the priority queue is empty.
- size(): Returns the number of elements in the queue.

Implementations:

Priority queues can be implemented using different data structures, with the most common being:

- **Heaps:** Binary heaps (min-heap or max-heap) are a popular choice due to their efficient insertion and deletion operations (O(log n)).
- **Sorted Arrays/Lists:** While simpler to implement, these can be less efficient for large datasets, especially for insertion and deletion (O(n)).
- **Self-Balancing Binary Search Trees:** Can also be used, providing good performance for various operations.

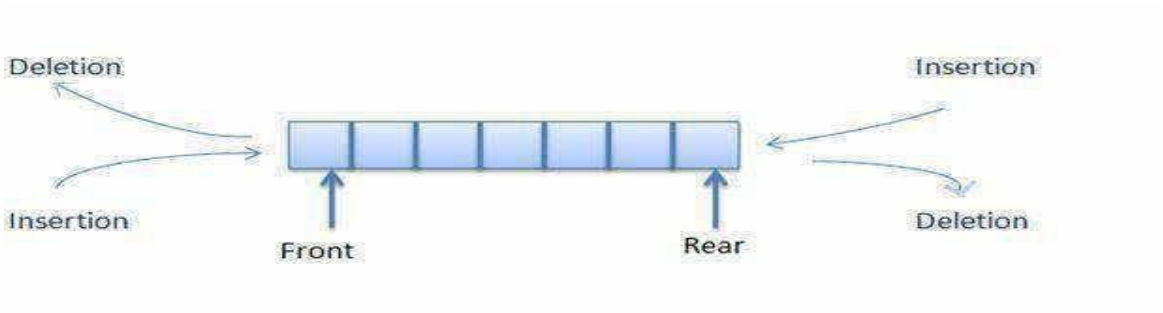
Use Cases:

- **Operating Systems:** Scheduling tasks based on priority (e.g., real-time tasks).
- **Graph Algorithms:** Dijkstra's algorithm, for example, uses a priority queue to find the shortest path.
- **Event Simulation:** Handling events based on their scheduled time (priority).
- **Data Compression:** Huffman coding uses a priority queue to build the coding tree.

Example (Conceptual):Imagine a hospital emergency room. Patients arrive and are assigned a priority (e.g., based on the severity of their condition). The priority queue would ensure that the most critical patients are treated first, regardless of when they arrived.

DOUBLE-ENDED QUEUE (DEQUE)

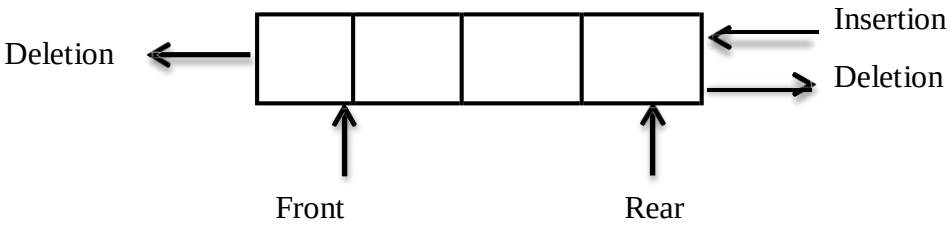
In DEQUE, insertion and deletion operations are performed at both ends of the Queue.



Exceptional Condition of DEQUE

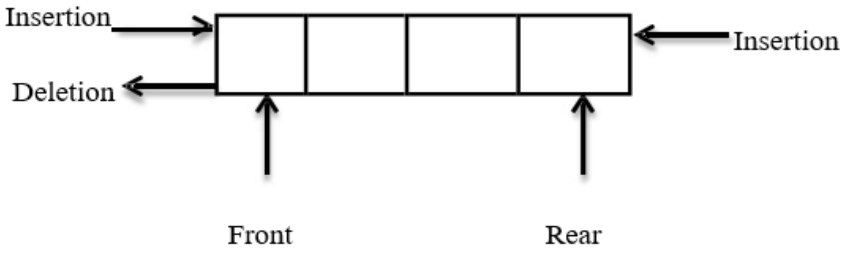
(i) Input Restricted DEQUE

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.

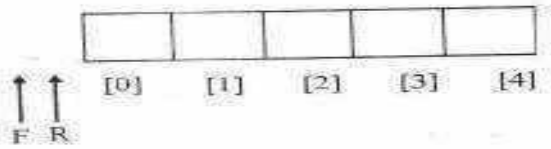
Step 4 : Increment the rear pointer

Step 5 : Assign the value to Q[rear]

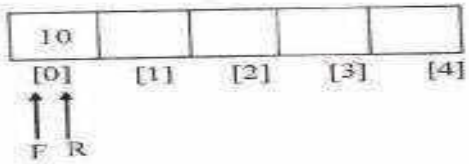
Case 1: Routine to insert an element at Rear end

```
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;
   }
}
```

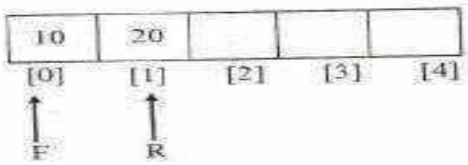
Insertion at the rear end



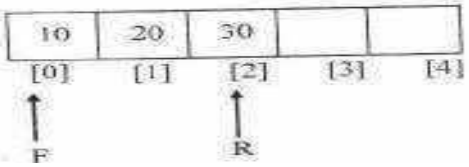
Insert_rear (10)



Insert_rear (20)



Insert_rear (30)



Insertion at front end

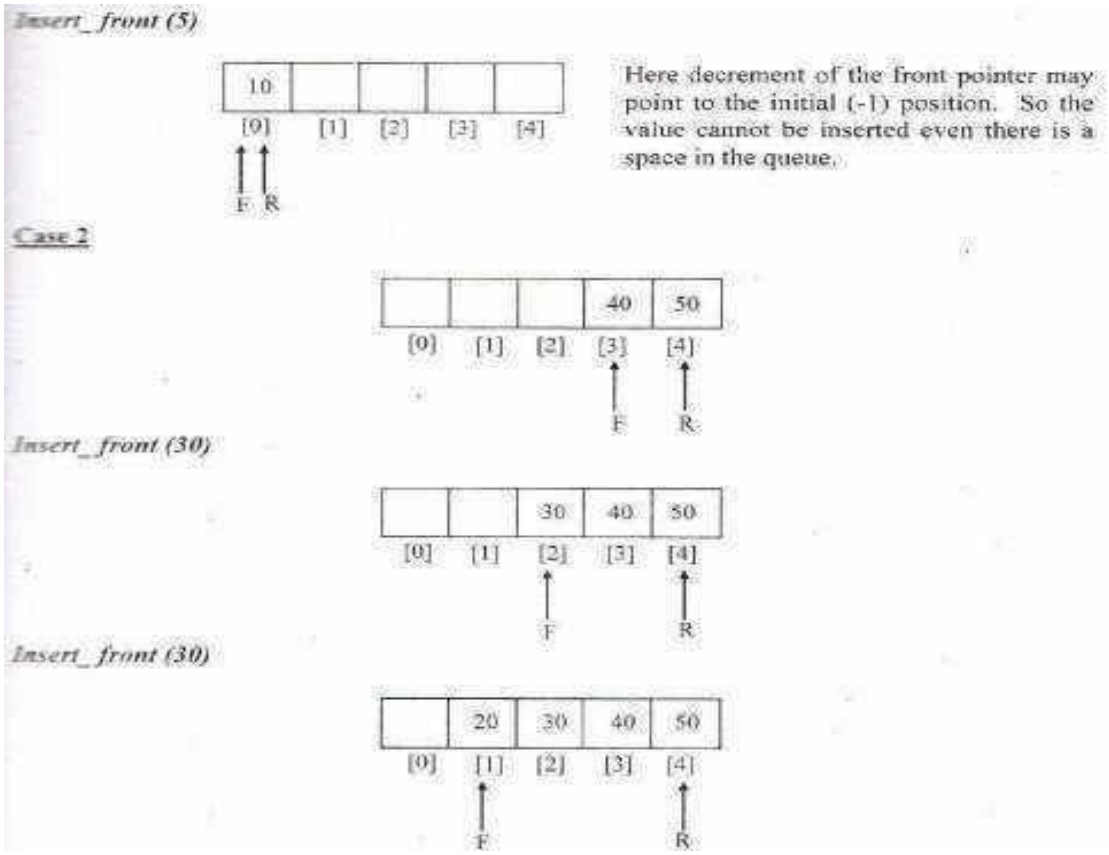
Step 1 : Check the front pointer, if it is in the first position (0) then display an error message that the value cannot be inserted at the front end.

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 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.

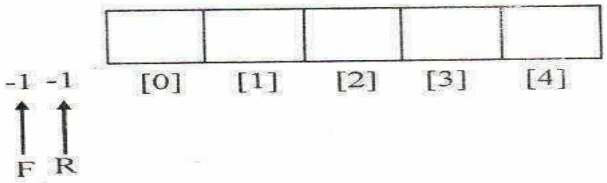
Step 2 : Otherwise, delete the element at the front position, by asigning X as Q[front]

Step 3 : If the rear and front pointer points to the same position (ie) only one value is present, then reinitialize both the pointers.

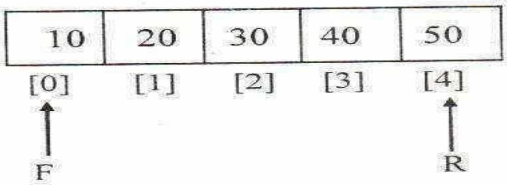
Step 4 : Otherwise, Increment the front pointer

Deletion from front end

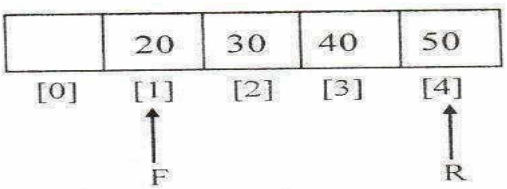
Dequeue_front ()



Q - is empty

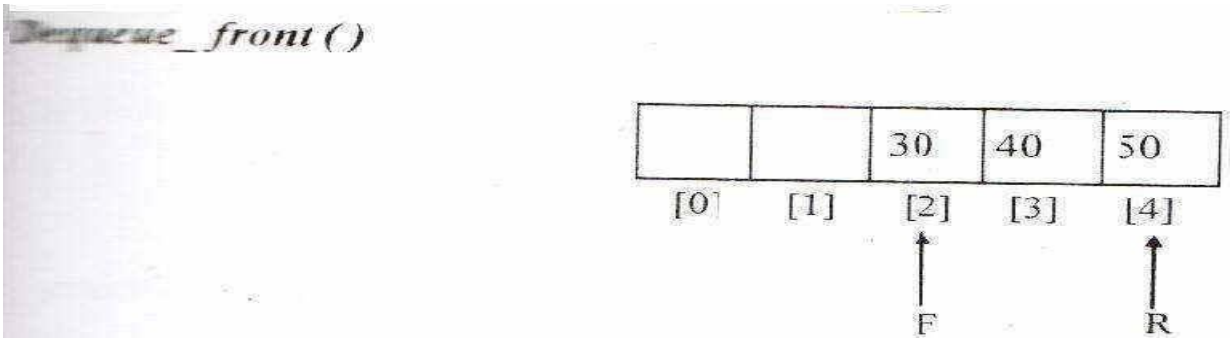


Dequeue_front ()



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;
  }

}
```



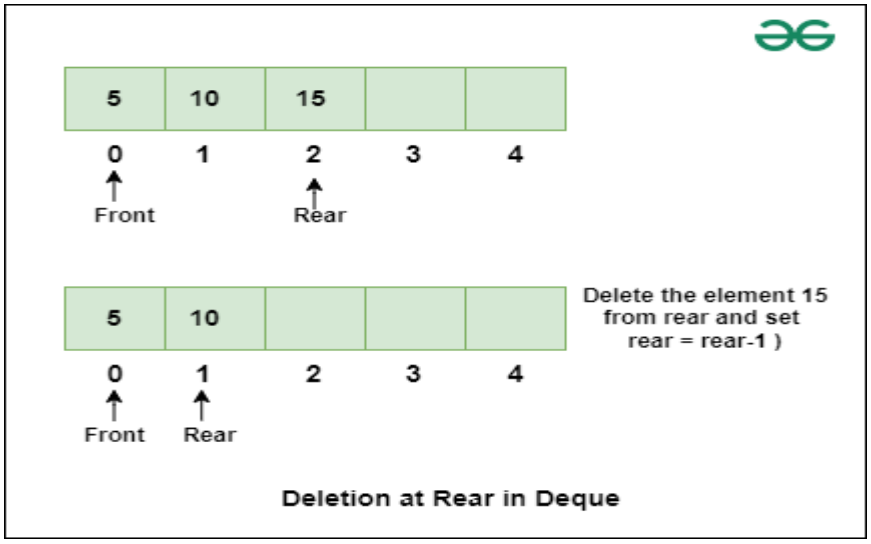
Deletion from Rear End

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.



Applications of queues

💻 Computer Science & Data Structures

- **CPU Scheduling**: Queues manage processes in operating systems using algorithms like First-Come-First-Serve and Round Robin.
- **Breadth-First Search (BFS)**: Graph traversal algorithms use queues to explore nodes level by level.
- **Memory Management**: Queues help allocate and deallocate memory blocks in OS environments.
- **I/O Buffers**: Used to store data temporarily during input/output operations.
- **Print Spooling**: Print jobs are queued and processed sequentially.
- **Task Scheduling**: Queues organize tasks based on arrival or priority.

🌐 Networking & Communication

- **Packet Scheduling**: Routers and switches use queues to manage data packets efficiently.
- **Message Queues**: In distributed systems (e.g., RabbitMQ, Kafka), queues handle asynchronous communication.
- **Load Balancing**: Web servers queue incoming requests before distributing them to backend services.

□ Real-World Scenarios

- **Customer Service Systems**: Banks, call centers, and ticket counters use queues to serve people in order.
- **Traffic Management**: Vehicles line up at signals or toll booths, following FIFO logic.
- **Food Delivery & Ordering**: Orders are queued and processed sequentially.
- **Ticket Booking Systems**: Online platforms queue users during high demand.

□ Advanced Applications

- **Simulation Systems**: Queues model real-world systems like manufacturing or service centers.
- **Event Handling**: GUI and real-time systems use queues to manage user or system events.
- **Cloud Computing**: Queues help in job scheduling and resource allocation across servers.

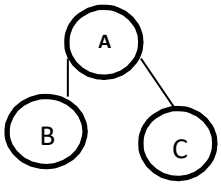
UNIT III TREES

Tree ADT – Tree Traversals - Binary Tree ADT – Expression trees – Binary Search Tree ADT – AVL Trees – Heaps Tree – Binary Heap – B-Tree – B+ Tree – Applications of Tree.

Tree ADT

A tree abstract data type (ADT) represents hierarchical data using nodes and edges, forming a tree-like structure. It's non-linear, unlike lists or arrays, and is characterized by a root node with child nodes, which can further have their own children, forming subtrees. Key concepts include root, parent, child, leaf, and subtree, which are fundamental to understanding tree structures

BINARY TREE TRAVERSALS



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)**

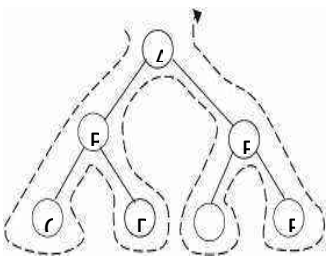
Inorder--- B A C Preorder --- A B C Postorder --- B C A

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

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



The Output is : C □ B □ D □A □E □ F

Algorithm

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

Routines

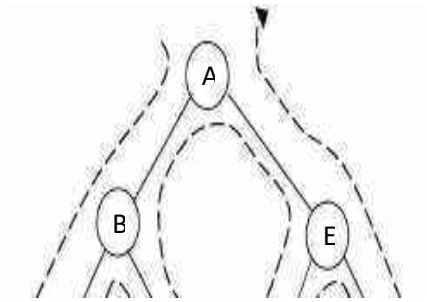
```
void inorder_traversal ( NODE * T)
{
if( T ! = NULL)
{
inorder_traversal(T->lchild);printf("%d \t
", T->info); inorder_traversal(T->rchild);
}
}
```

Preorder Traversal

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

Algorithm

Algorithm inoder 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

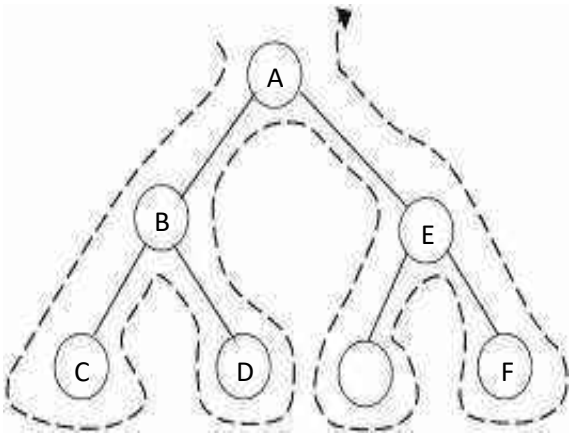
Postorder Traversal

Steps :
Traverse left subtree in postorder Traverse right subtree in postorderprocess root node

Algorithm
Algorithm postorder traversal (BinTree T)Begin
If (not empty (T)) thenBegin
Postorder_traversal (left subtree (T))
Postorder_traversal (right subtree(T))Print (Info (T))
/ * process node */ End
End

Routines

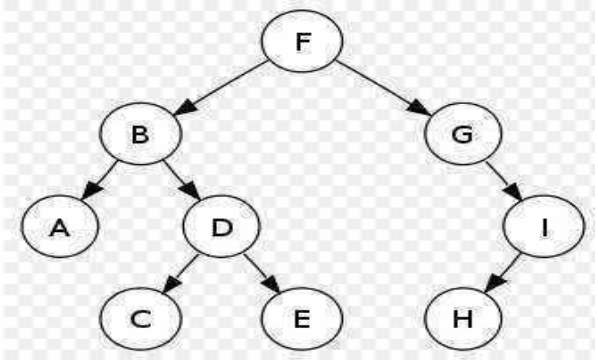
```
void postorder_traversal ( NODE * T)
 {
if( T ! = NULL)
{
postorder_traversal(T->lchild);
postorder_traversal(T->rchild); printf("%d  \t",
T->info);
}
```



}

Output is : C □ D □ B □ F □ E □ A

Example



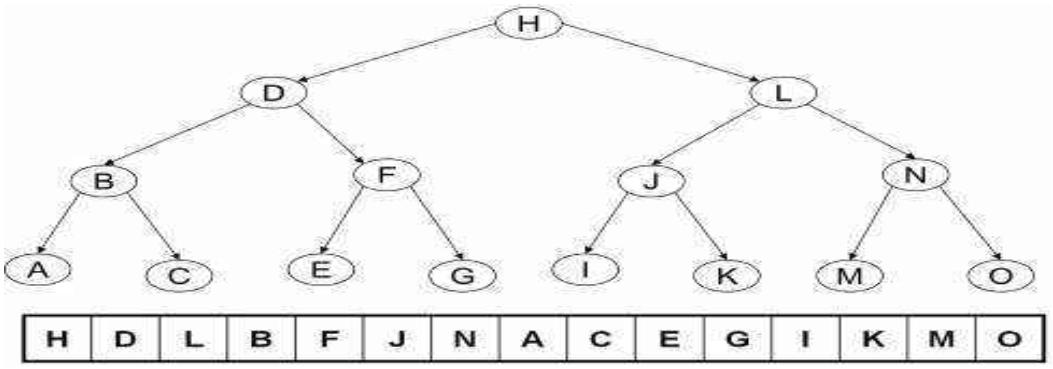
Preorder traversal sequence: F, B, A, D, C, E, G, I, H (root, left, right) Inorder traversal

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)

Binary Tree ADT

A Binary Tree Abstract Data Type (ADT) defines the logical structure and behavior of a binary tree, independent of its specific implementation. It specifies the data elements stored within the tree and the operations that can be performed on it.

Binary Tree using Array Representation

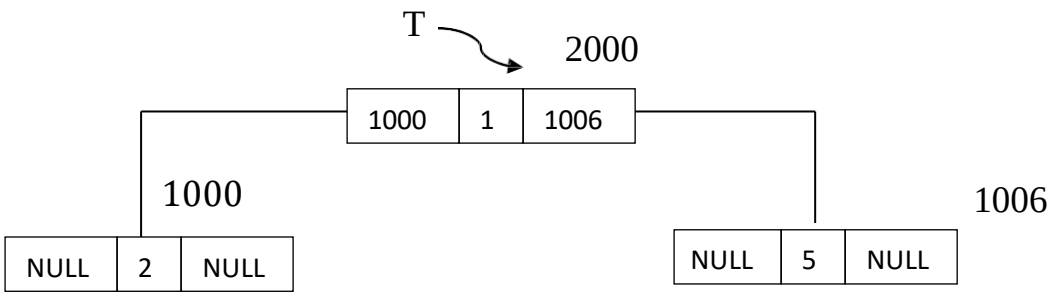


Binary Tree using Link Representation

The problems of sequential representation can be easily overcome throughthe use of a linked representation.
Each node will have three fields LCHILD, DATA and RCHILD as representedbelow

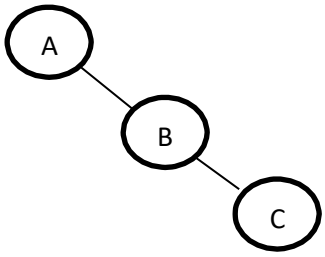
Using the linked implementation we may declare,

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



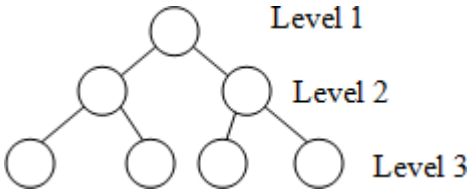
TYPES OF BINARY TREES

Skewed Binary tree :A Binary tree which has only right child or left child is called skewed binary tree.

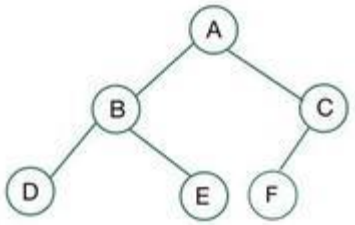


Full Binary Tree :

It is the one which has exactly two children for each node at each level andall 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 befilled from the left to the right.All full binary trees are complete binary tree. But all complete binary treesneed not be full binary tree.



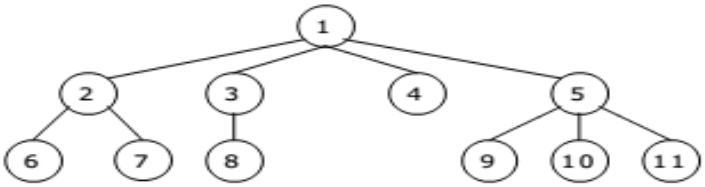
CONVERSION OF A GENERAL TREE TO BINARY TREE

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 binarytree format.

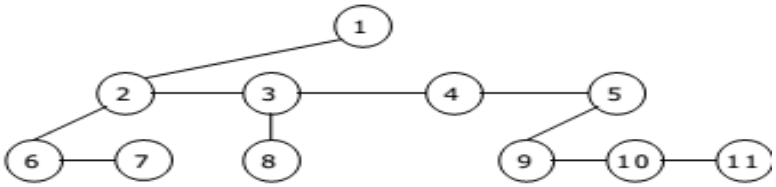
Changing general tree to Binary tree:

The binary tree format can be adopted by changing the meaning of the left andright 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.

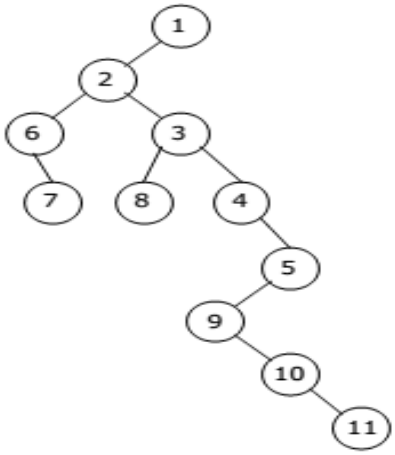


Solution:

Stage 1 tree by using the above mentioned procedure is as follows:

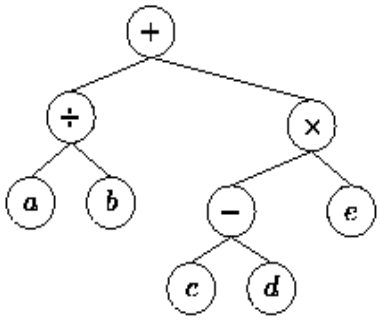


Stage 2 tree by using the above mentioned procedure is as follows:



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

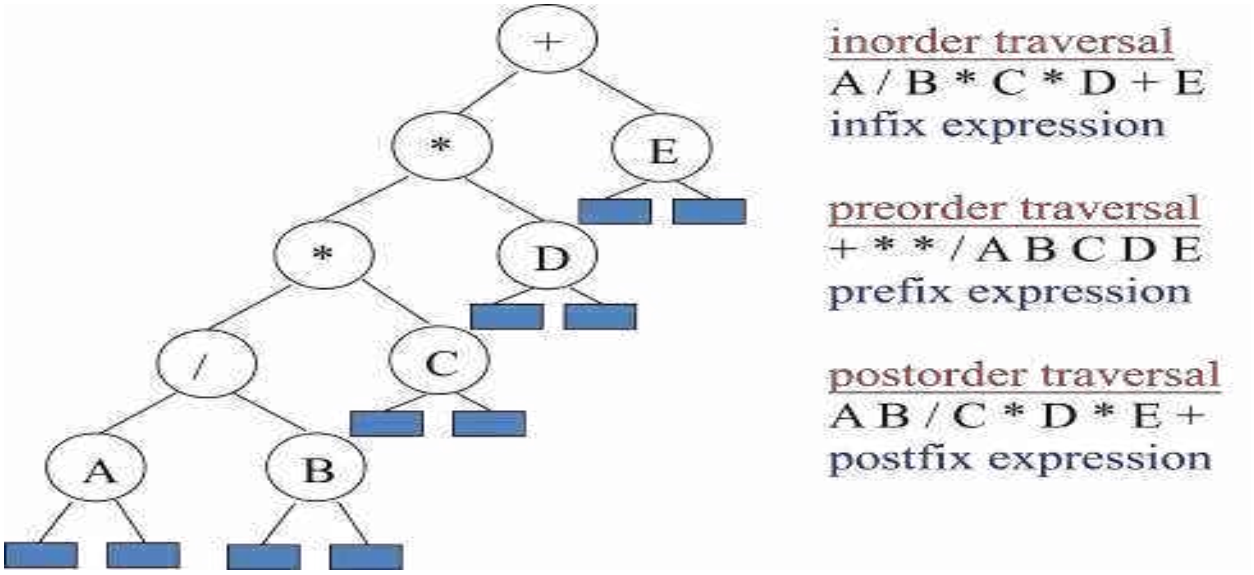
1) Examine the next element in the input.
2) If it is an operand, output it.
3) If it is opening parenthesis, push it on stack.
4) If it is an operator, then
i) If stack is empty, push operator on stack.
ii) If the top of the stack is opening parenthesis, push operator on stack.
iii) If it has higher 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 themuntil 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 Postfix:

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-*+

EVALUATION OF EXPRESSIONS



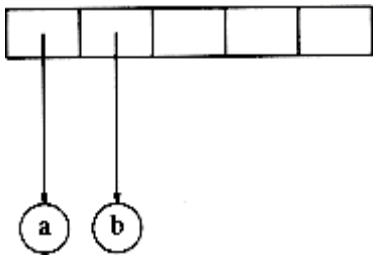
CONSTRUCTING AN EXPRESSION TREE

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

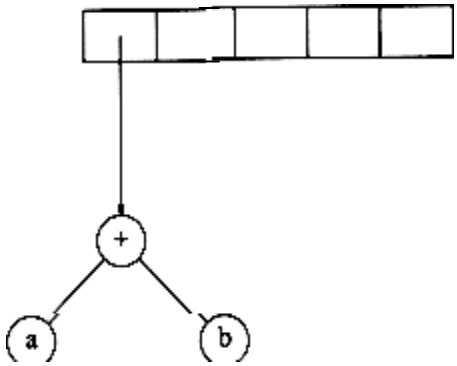
1. Read one symbol at a time from the postfix expression.
2. Check whether the symbol is an operand or operator.
   i. If the symbol is an operand, create a one node tree and push apointer on to the stack.
   ii.If the symbol is an operator, pop two pointers from the stacknamely, T1 and T2 and form a new tree with root as the operator, and T2 as the left child and T1 as the right child.
   iii.A pointer to this new tree is then pushed on to the stack.

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 thesymbol is an operator, we pop pointers to two trees $T1$ and $T2$ from the stack ($T1$ is popped first) and form a new tree whose root is the operator and whoseleft and right children point to $T2$ and $T1$ respectively. A pointer to this new tree is then pushed onto the stack.

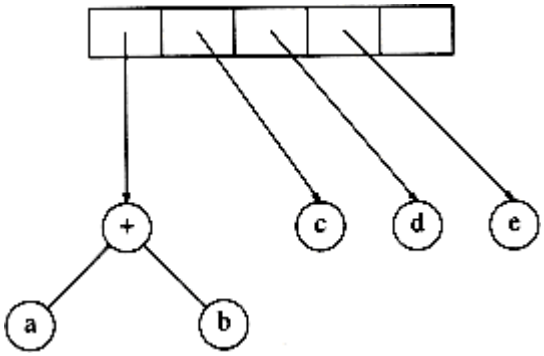
As an example, suppose the input is a b + c d e + * *



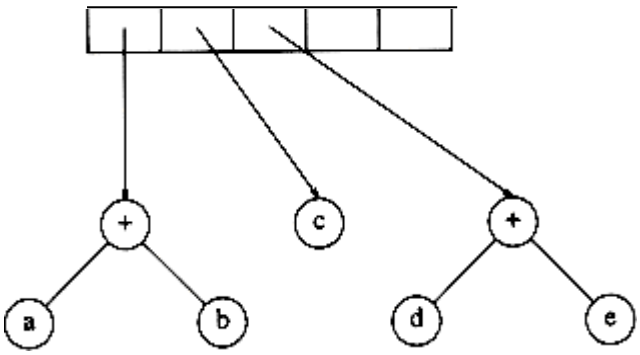
Next, a '+' is read, so two pointers to trees are popped, a new tree is formed,and a pointer to it is pushed onto the stack.*



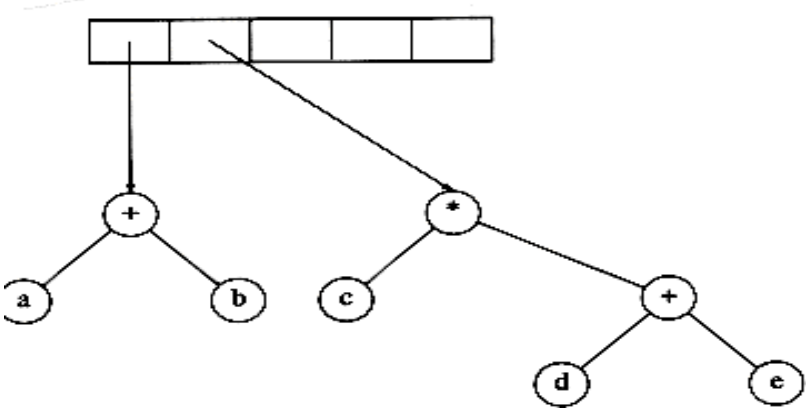
Next, *c*, *d*, and *e* are read, and for each a one-node tree is created and a pointerto the corresponding tree is pushed onto the stack.



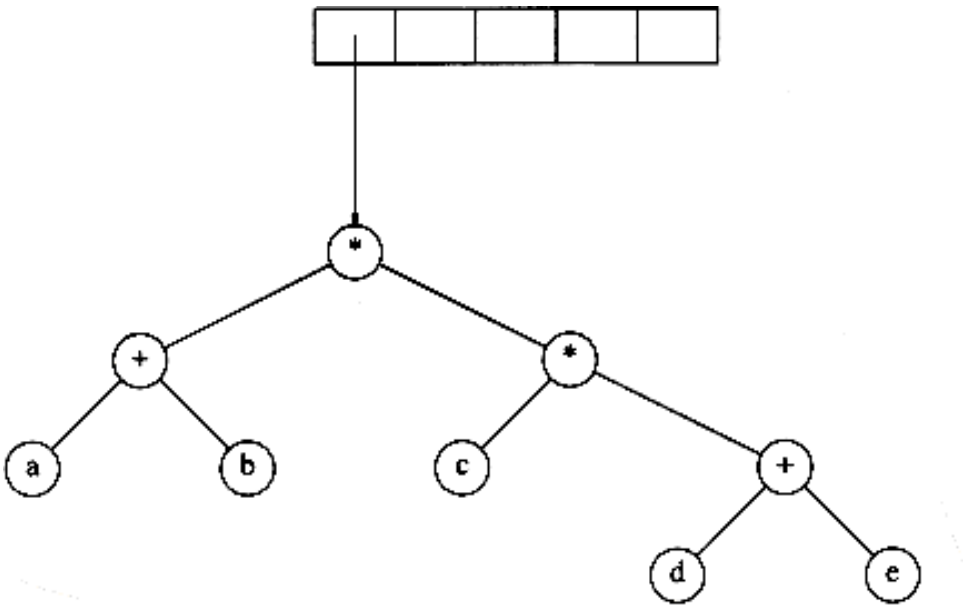
Now a '+' is read, so two trees are merged.



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



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



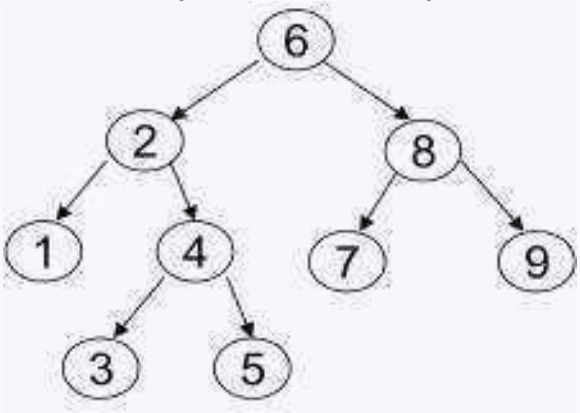
BINARY SEARCH TREE

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

- The left sub-tree of a node contains only nodes with keys less than thenode's key.
  The right sub-tree of a node contains only nodes with keys greater thanthe 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.



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 leftsubtree 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;
};
```

Binary Search Tree

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

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 *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 storedat *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 searchtrees

```
// 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 oranother search tree.

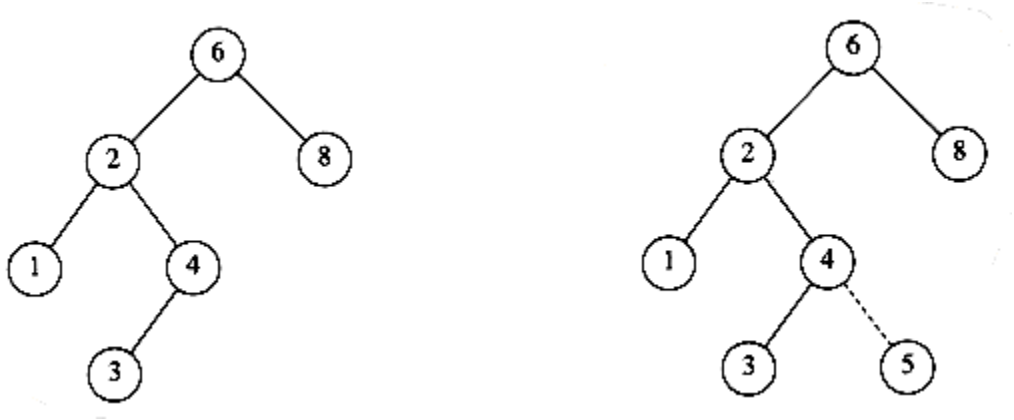


Figure shows the code for the insertion routine. Since *T* points to the root ofthe tree, and the root changes on the first insertion, *insert* is written as a function that returns a pointer to the root of the new tree. Lines 8 and 10 recursively insert and attach *x* into the appropriate subtree.

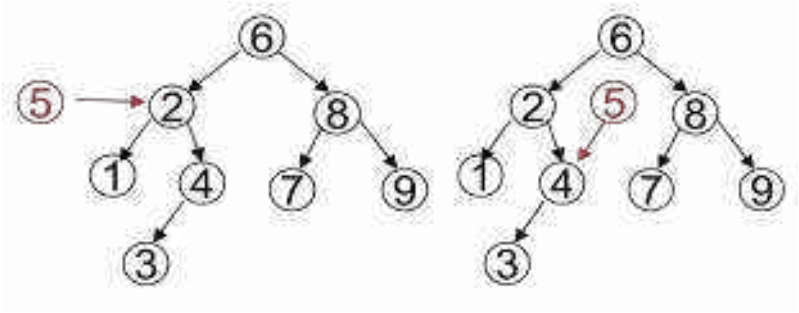
Insertion into a binary search tree

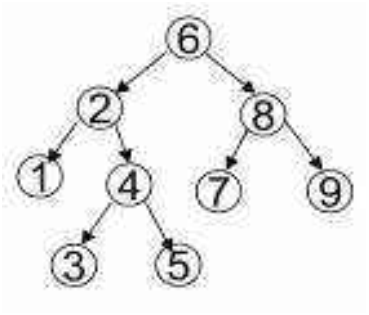
```
Searchtree insert(elementtype X, Searchtree T)
{
If(T== NULL)
{
/* create and return a one node tree*/
T=malloc(sizeof(structtreenode)); If(T==NULL)
Fatalerror("Out of Space");Else
{
T-->element=X;
T-->left=T-->right=NULL;
}
}
```

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 ofthe 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 with no children | Leaf node :

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

Case 2: Node with only one child :

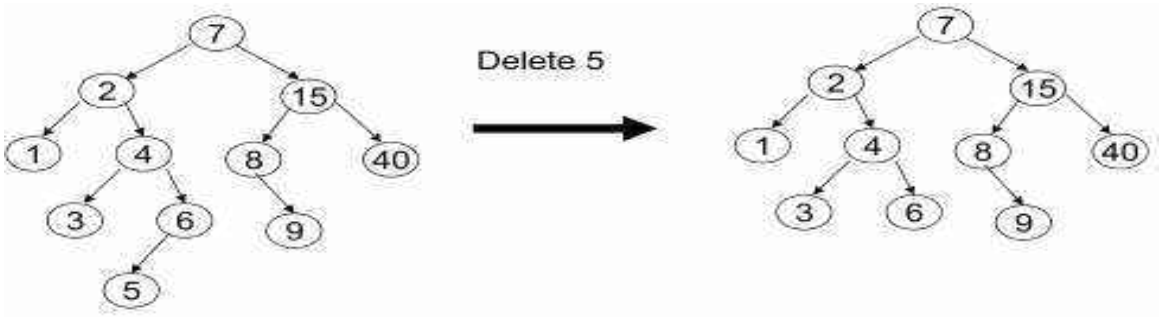
1. Search the parent of the node to be deleted.
2. Assign the link of the parent node to the child of the node to be deleted.
3. Release the memory for the deleted node.

Case 3: Node with two children :

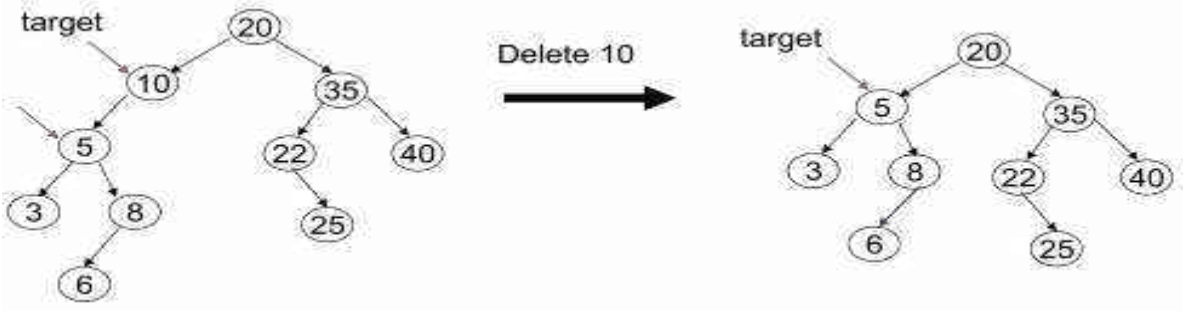
It is difficult to delete a node which has two children.So, a general strategy has to be followed.

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

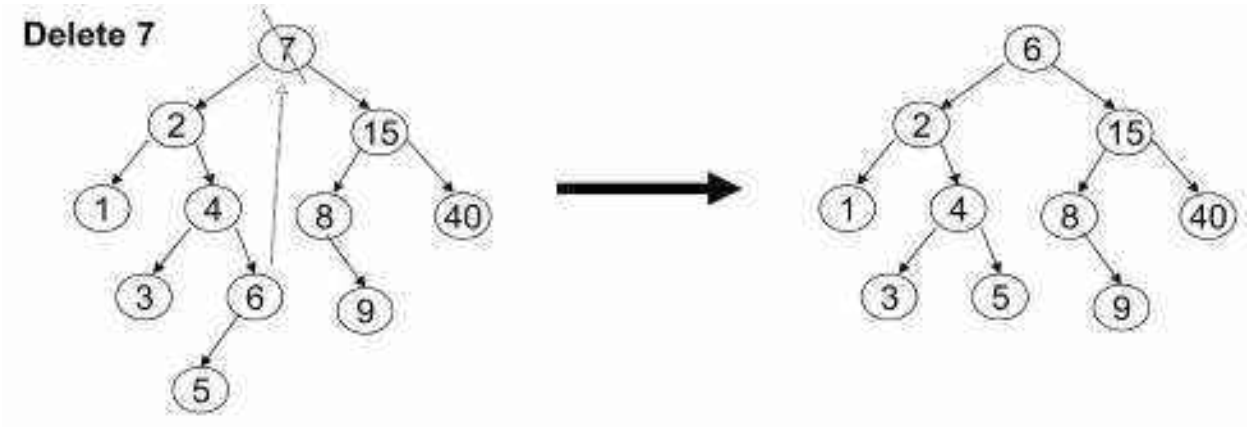
Case 1:



Case 2 :



Case 3 :



The code in 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.

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;
}
```

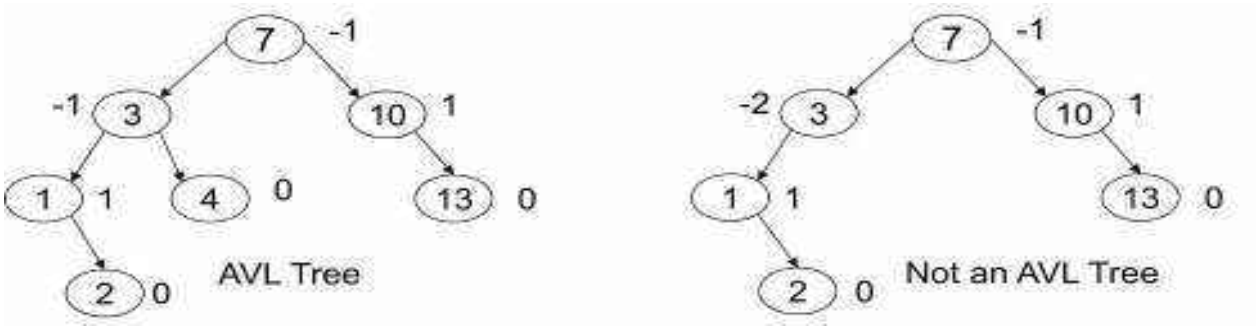
AVL TREE

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 ofthe two child subtrees of any node differ by at most one; therefore, it is alsosaid to be height-balanced.

Need for AVL tree

- The disadvantage of a binary search tree is that its height can be as largeas N-1
- This means that the time needed to perform insertion and deletion andmany 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 Q(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 AVLtree, red-black tree.

Thus we go for AVL tree. The balance factor of a node is the height of its right subtree minus theheight 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



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:

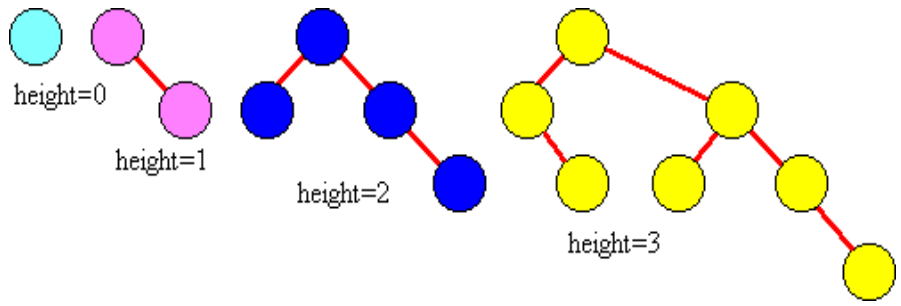
1. The height of a tree with no elements is 0
2. The height of a tree with 1 element is 1
3. The height of a tree with > 1 element is equal to 1 + the height of itstallest subtree.
4. The height of a leaf is 1. The height of a null pointer is zero.

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:
|Height (Tl)-Height(Tr)|<=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.

Balance Factor(BF) = Hl --- Hr. Hl

=> Height of the left subtree. Hr

=> 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 bedone.

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.

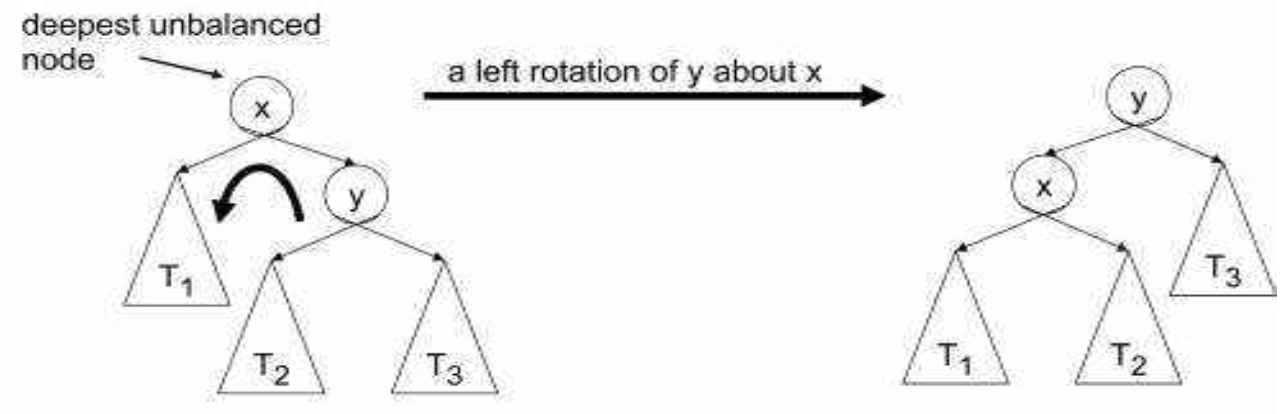
2. Double Rotation – (If insertion occurs on the inside, i.e., LR or RL)

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

1.General Representation of Single Rotation

LL Rotation:

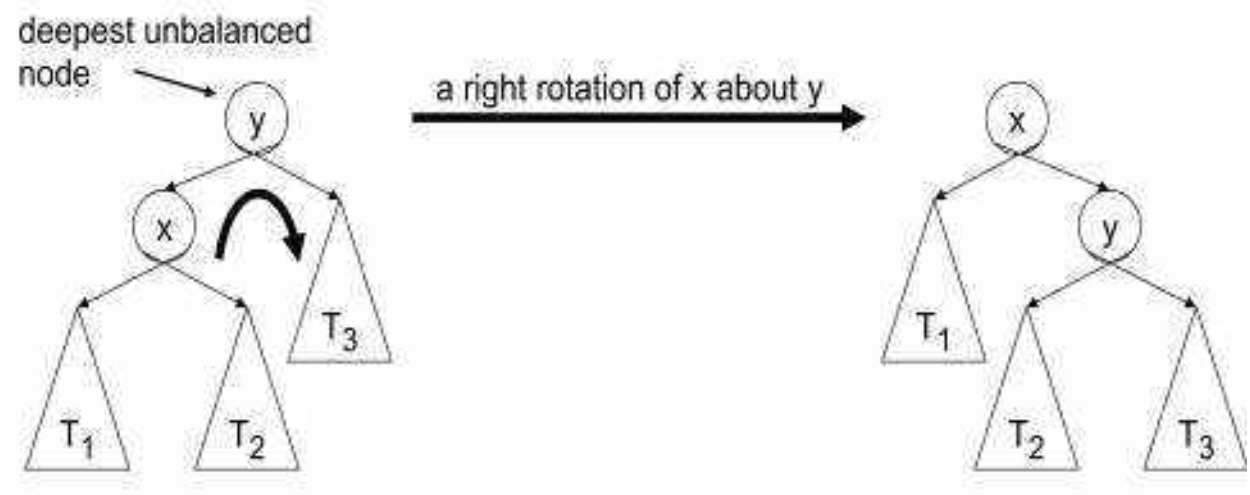
- The right child **y** of a node **x** becomes **x**'s parent.
- **x** becomes the **left child** of **y**.
- The left child **T2** of **y**, if any, becomes the **right child** of **x**.



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

RR Rotation :

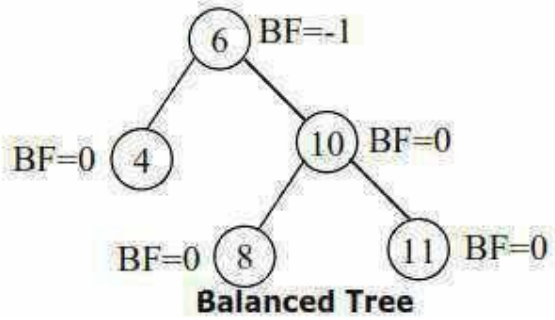
- The left child x of a node y becomes y's parent.
- y becomes the right child of x.
- The right child T_2 of x, if any, becomes the left child of y.



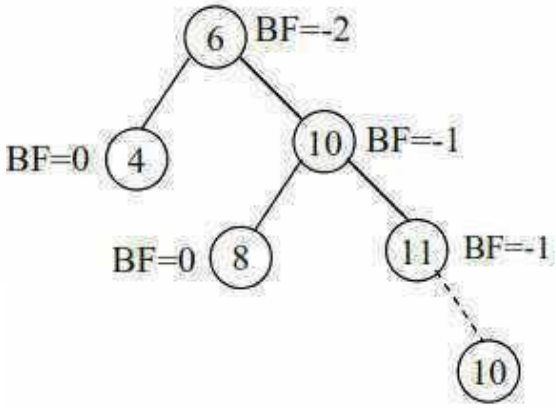
Note: The pivot of the rotation is the deepest unbalanced node

Example:

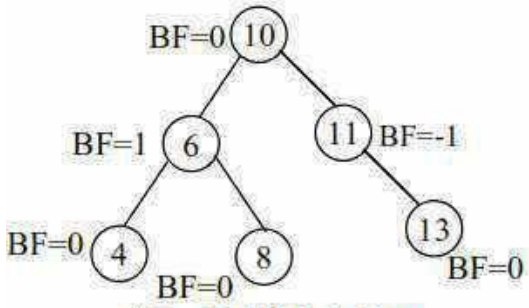
Consider the following tree which is balanced.



Balanced Tree



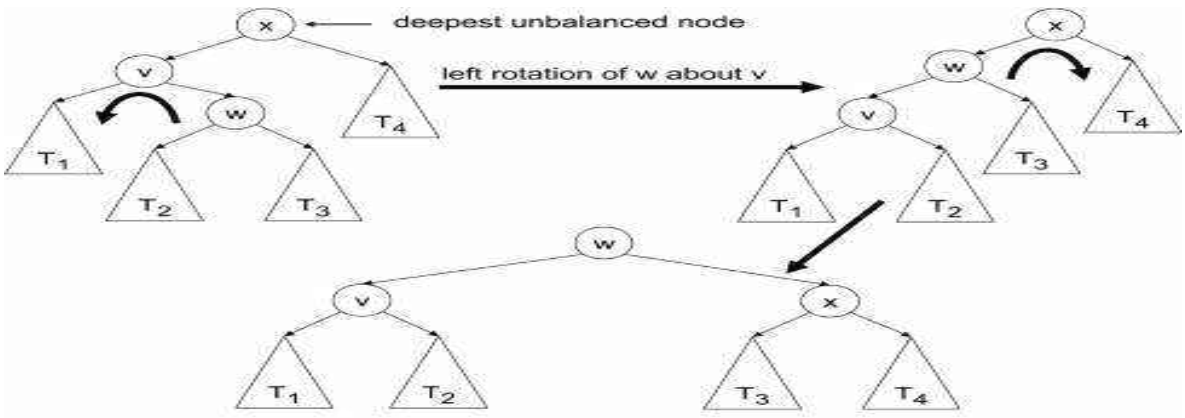
Now insert the value '13' it becomes unbalanced due to the insertion of new node in the Right Subtree of the Right Subtree. So we have to make single Right rotation in the root node.



After Right Rotation

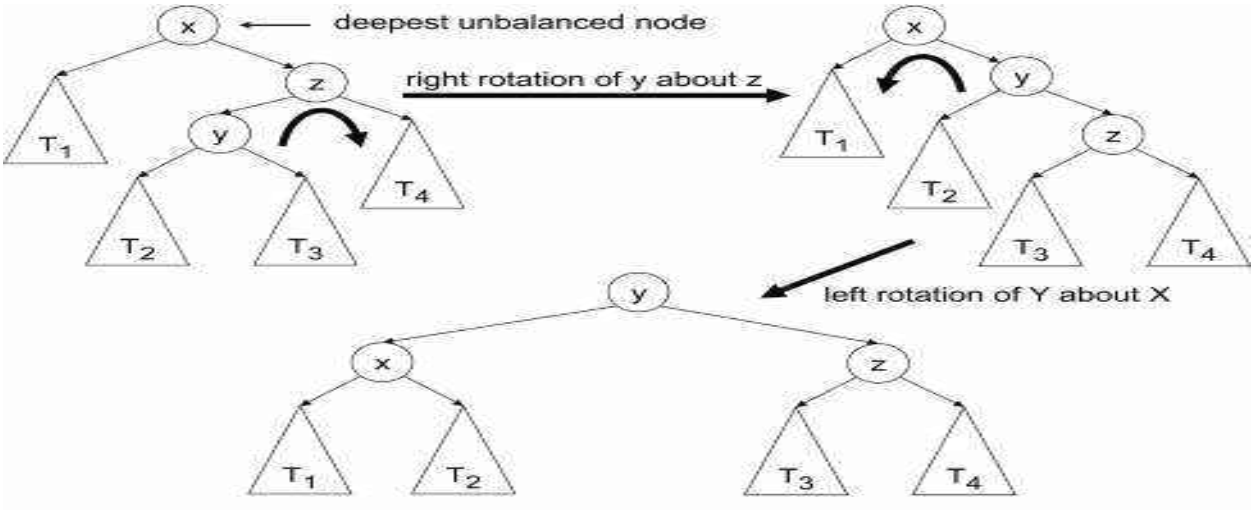
2. General Representation of Double Rotation :

LR(Left -- Right rotation):



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

RL(Right -- Left rotation) :



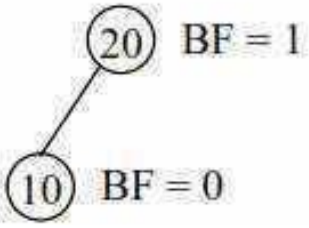
Note: First pivot is the left child of the deepest unbalanced node; second pivot is the deepest unbalanced node

EXAMPLE:

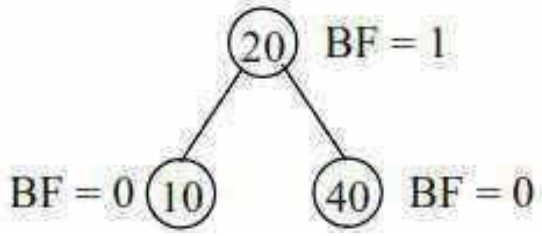
LET US CONSIDER INSERTING OF NODES 20,10,40,50,90,30,60,70 in an AVLTREE

Step 1:(Insert the value 20)

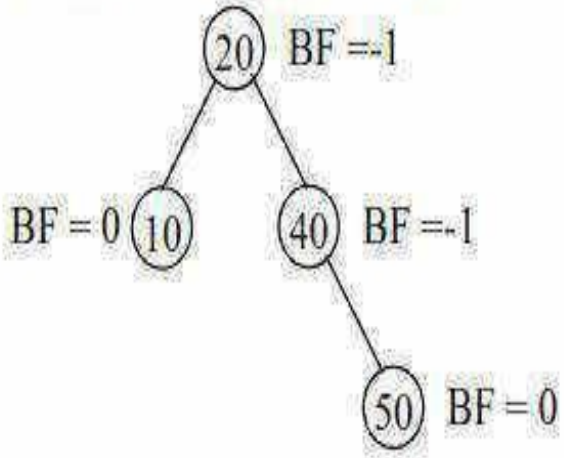
Step 2: (Insert the value 10)



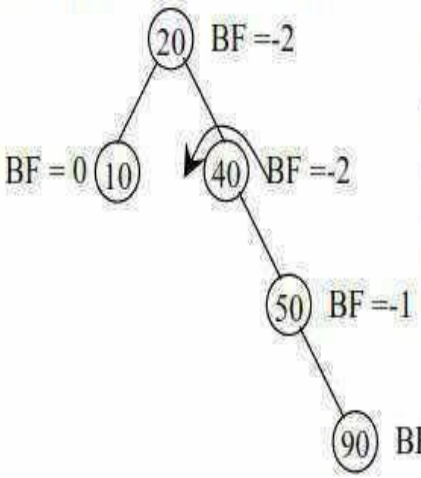
Step 3: (Insert the value 40)



Step 4:(Insert the value 50)

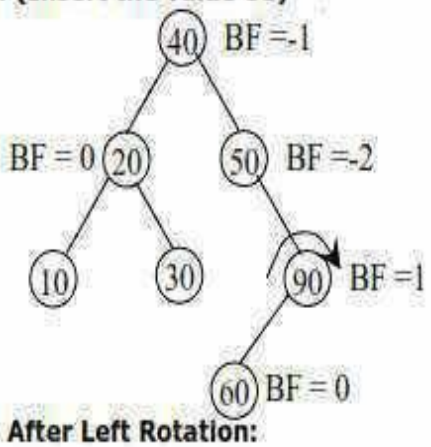


Step 5: (Insert the value 90)

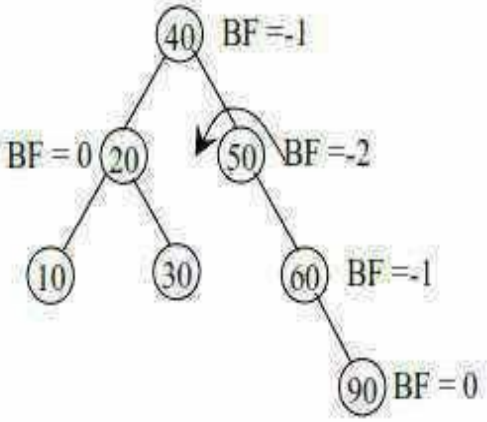


Now the tree is unbalanced due to the insertion of node in the Right subtree of the Right subtree. So we have to make single Right rotation in the node '40'.

Step 7: (Insert the value 60)

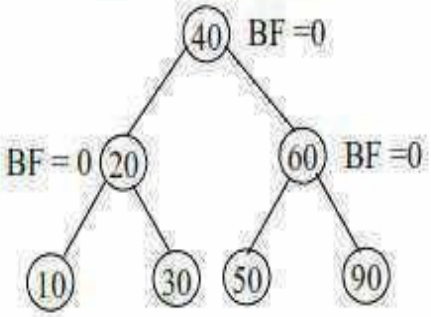


After Left Rotation:



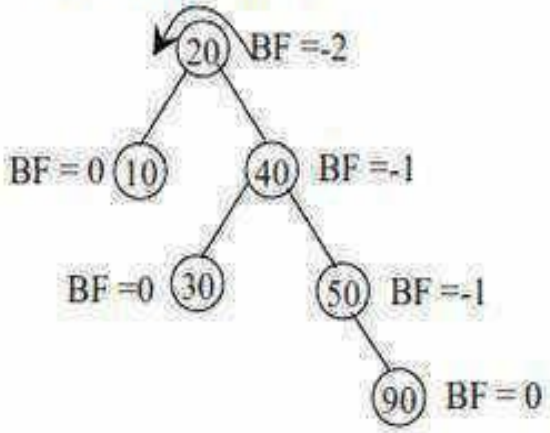
Now the tree at the node '50'is unbalanced due to the insertion of node '60' in the Left subtree of the Right subtree. So we have to make Double rotation first with Left on the node '90' and then with Right on the node '50'.

After Right Rotation:

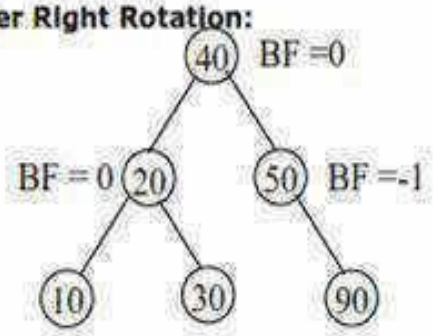


Now the tree is Balanced

After Left Rotation:



After Right Rotation:



Now 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);
}
```

APPLICATIONS

AVL trees play an important role in most computer related applications. Theneed 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 languageSpell 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 astandard bst.
- Height of an AVL tree with n nodes is always very close to thetheoretical minimum.
- Since the avl tree is height balabced the operation like insertion anddeletion have low time complexity.
- Since tree is always height balanced.Recursive implementation ispossible.
- The height of left and the right sub-trees should differ by atmost1.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 everylevel 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 useother structures

BINARY HEAPS

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

- it is empty *or*
- the key in the root is larger than that in either child and both subtreeshave the heap property.

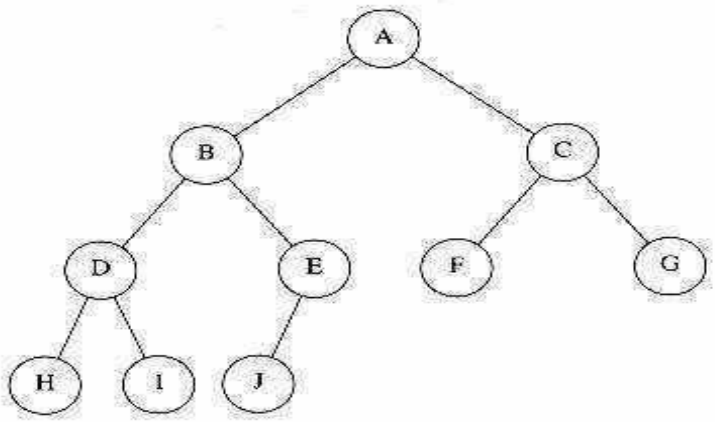
- In eneral heap is a group of things placed or thrown, one on top of theother.
- In data structures a heap is a binary tree storing keys at its nodes.Heaps are based on the concepts of a complete tree

Structure Property :

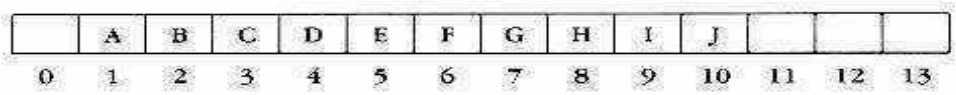
COMPLETE TREE

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 iscompletely full of height *h*-2 *or*
- its left subtree is completely full of height *h*-1 and its right subtree iscomplete of height *h*-1.



A complete binary tree



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 issatisfied.
- 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 obtainthe 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 propertyis 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 morecollection 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) timeto 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 thebottom 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 irrelated value, inorder to terminate theprogram 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 propertiesMax-heap & Min-heap

Min-heap: The smallest element is always in the root node.Each node must have akey 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 itschildren.

Examples

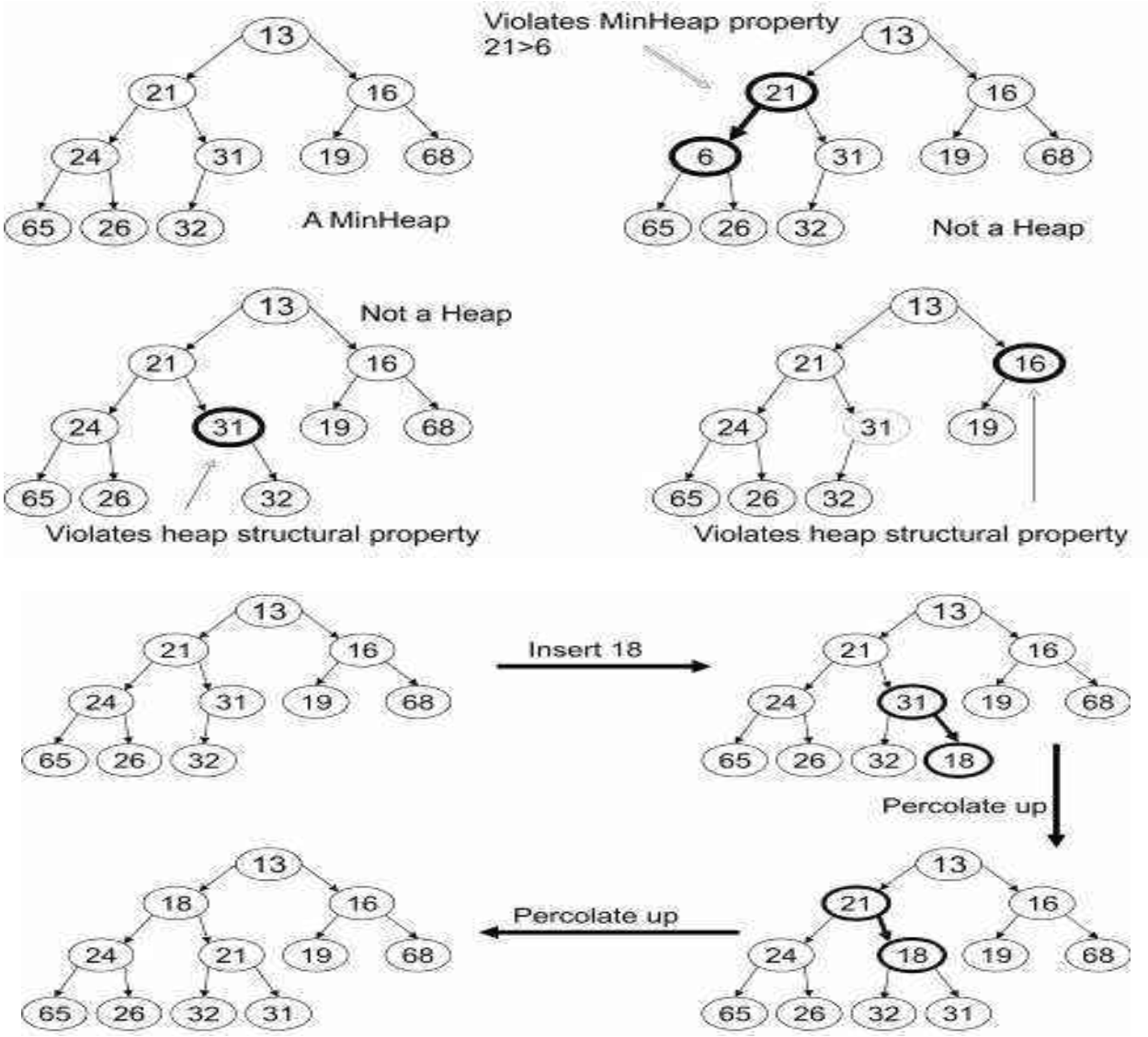
HEAP OPERATIONS: : There are 2 operations of heap Insertion Deletion

2.12.1 Insert: Adding a new key to the heap To insert an element X into the heap, do the following:

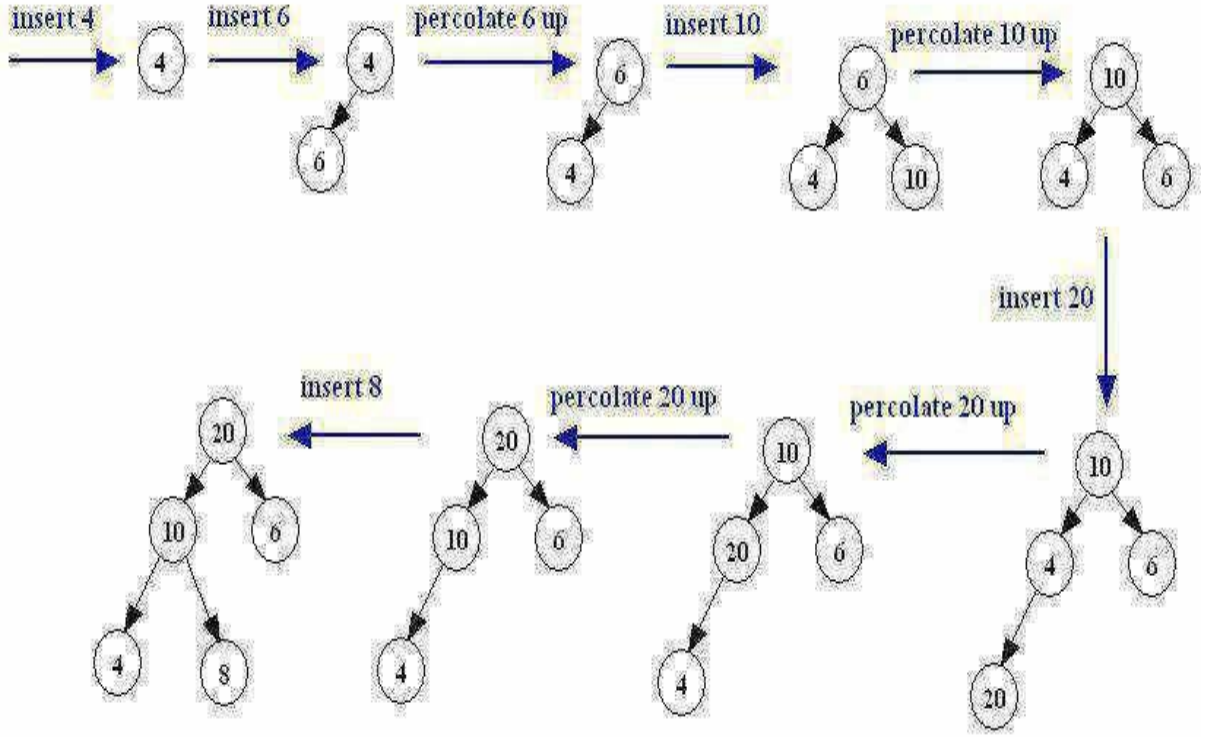
Step1: Create a hole in the next available location, since otherwise the tree will not be complete.

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.

Example Problem :



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



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

Procedure for Deletemin :

* Deletemin operation is deleting the minimum element from the loop.

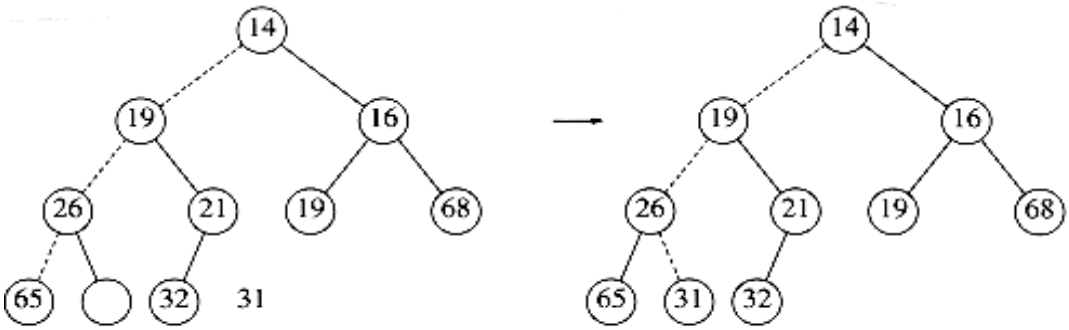
* In Binary heap | min heap the minimum element is found in the root.

* 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 tomove 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.

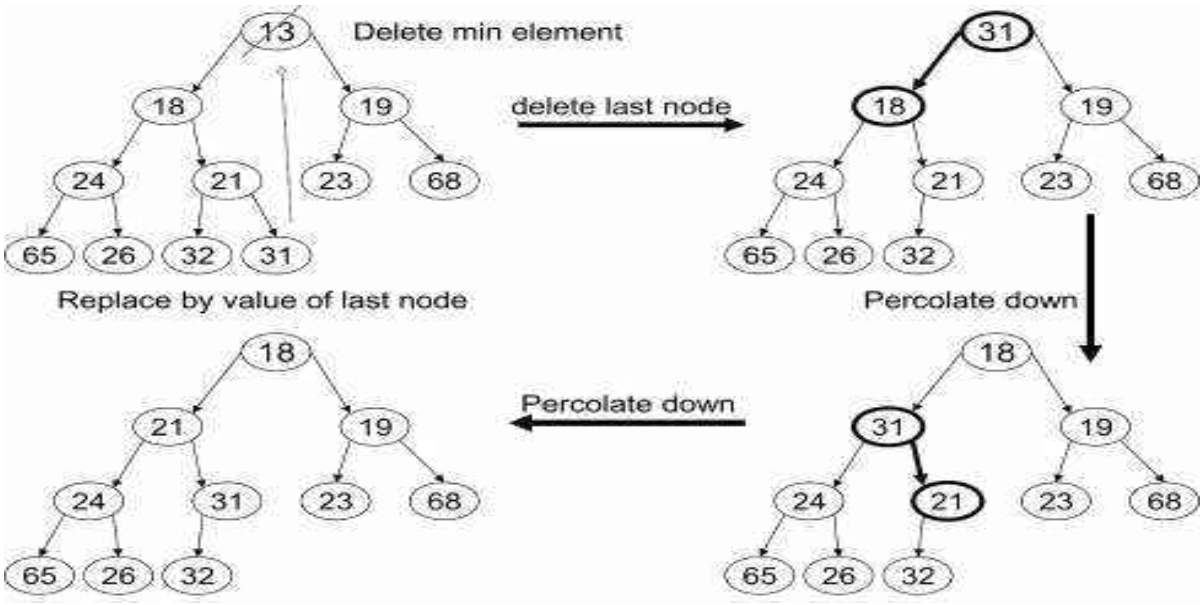


Last two steps in delete_min

EXAMPLE PROBLEMS :

1.DELETE MIN
BINARY HEAP ROUTINES [Priority Queue]

```
Typedef struct heapstruct *priorityqueue;
Typedef int elementype;
Struct heapstruct
{
int capacity;
int size;
elementtype *element;
};
```



Insert Routine

```
Void insert(elementtype X, priorityqueue H)
{
int i;
if(isfull(H))
{
Error("Priority queue is full");
Return;
}
For(i=++H-->size;H-->elements[i/2]>X;i=i/2)
H-->elements[i]=H-->elements[i/2];
H-->elements[i]=X;
}
```

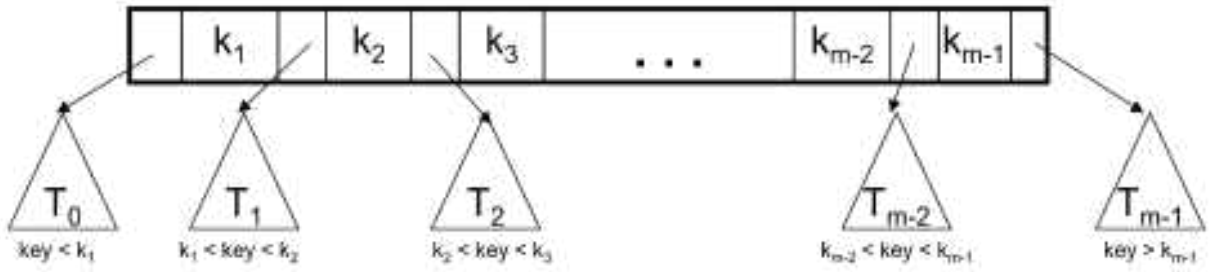
Delete Routine

```
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-->elements[child++]<H-->elements[child])
{
Child++;
}
/ * Percolate one level * / If(lastelement>H-
->elements[child])H-->element[i]=H--
>elements[child];Else
Break;
}
H-->element[i]=lastelement;

Return minelement;
}
```

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

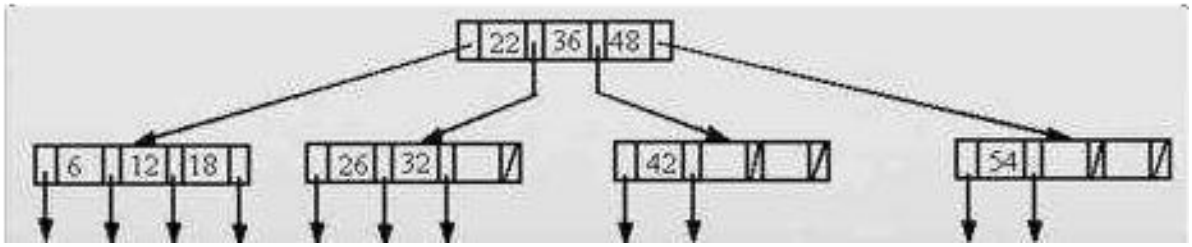
- Each node has at-most m subtrees, where the subtrees may beempty.
- Each node consists of at least 1 and at most m-1 distinct keysThe keys in
- each node are sorted.



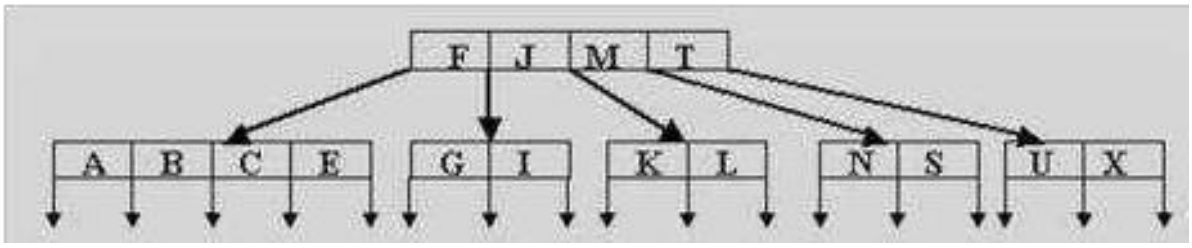
- The keys and subtrees of a non-leaf node are ordered as:
  - $T_0, k_1, T_1, k_2, T_2, \ldots, k_{m-1}, T_{m-1}$ such that:
- All keys in subtree T0 are less than k1.
- All keys in subtree T_i , $1 <= i <= m - 2$, are greater than k_i but less thank_{i+1}.
- All keys in subtree T_{m-1} are greater than k_{m-1}

B-Tree Examples

Example: A B-tree of order 4



Example: A B-tree of order 5

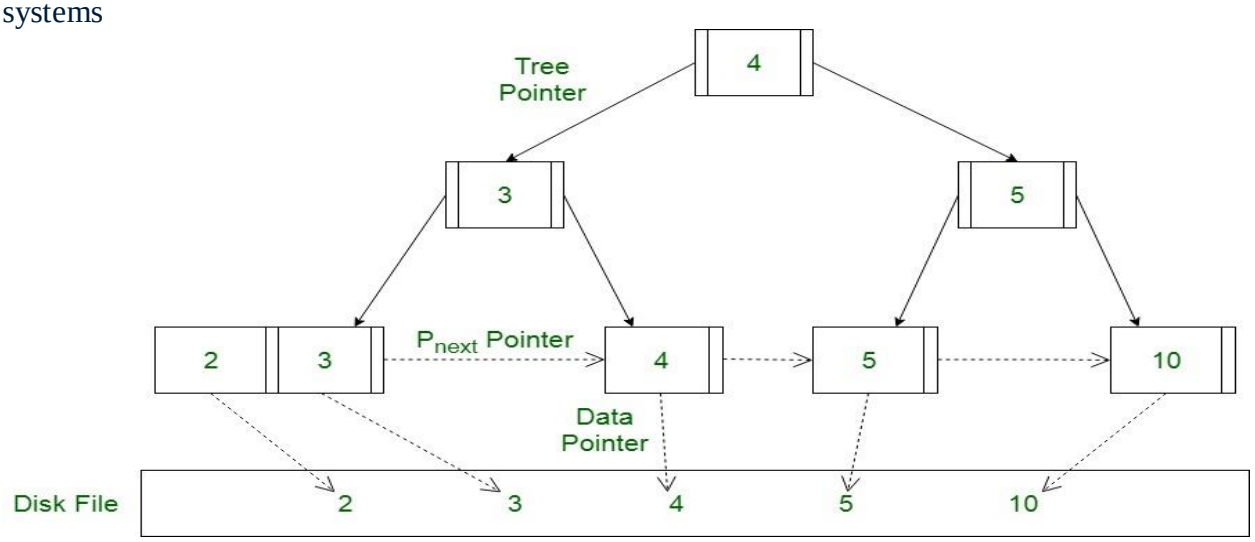


B+ Tree : A **B+ Tree** is a balanced tree data structure that maintains sorted data and allows efficient insertion, deletion, and search operations. It's widely used in **database indexing** and **file systems**.

Key Characteristics

- All **data records** are stored **only in leaf nodes**.
- **Internal nodes** store **keys** for indexing and routing, not actual data.
- **Leaf nodes** are linked together in a **linked list** for fast range queries.
- All leaf nodes are at the **same depth**, ensuring balanced access paths.
- Supports **high fan-out**, reducing tree height and disk I/O.

B+ trees are a specialized tree data structure designed for efficient storage and retrieval of data, particularly on disk-based systems. They are a variation of B-trees, optimizing for faster data access by storing all data (or pointers to data) only in leaf nodes, while internal nodes primarily serve as indexes. This structure helps minimize disk I/O operations, making them suitable for databases and file systems



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.

UNIT IV GRAPHS

Graph Definition - Graph Terminologies —Representation of Graphs — Types of Graph — Graph Isomorphism -Graph Traversals: Breadth-first traversal(BFS) — Depth-first traversal (DFS)–connectivity – Euler Paths and Circuits —Hamiltonian Paths and Circuits - Topological Sort – Shortest Path Algorithms :Dijkstra's algorithm – Floyd's algorithm – Minimum Spanning Tree: Prim's algorithm – Kruskal's algorithm – Applications of Graph.

Graph Definition

Graph is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph is composed of a set of vertices(V) and a set of edges(E). The graph is denoted by G(V, E).

Graph Terminologies

1.A **Graph** is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph is composed of a set of vertices(**V**) and a set of edges(**E**). The graph is denoted by **G(V, E)**.

2. Vertex (Node)

A Vertex, often referred to as a **Node**, is a fundamental unit of a graph. It represents an **entity** within the graph. In applications like social networks, vertices can represent individuals, while in road networks, they can represent intersections or locations.

3. Edge

An Edge is a **connection between two vertices** in a graph. It can be either directed or undirected. In a directed graph, edges have a specific direction, indicating a one-way connection between vertices. In contrast, undirected graphs have edges that do not have a direction and represent bidirectional connections.

4. Degree of a Vertex

The Degree of a Vertex in a graph is the **number of edges incident** to that vertex. In a directed graph, the degree is further categorized into the in-degree (number of incoming edges) and out-degree (number of outgoing edges) of the vertex.

5. Path

A Path in a graph is a **sequence of vertices** where each adjacent pair is connected by an edge. Paths can be of varying lengths and may or may not visit the same vertex more than once. The shortest path between two vertices is of particular interest in algorithms such as Dijkstra's algorithm for finding the shortest path in weighted graphs.

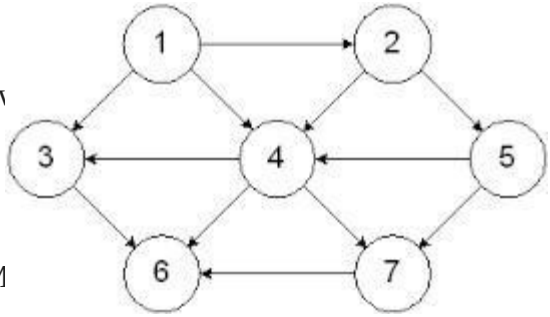
6. Cycle

A Cycle in a graph is a **path that starts and ends at the same vertex**, with no repetitions of vertices (except the starting and ending vertex, which are the same). Cycles are essential in understanding the connectivity and structure of a graph and play a significant role in cycle detection algorithms.

Types of Graphs:

Ø Directed graph Ø Undirected Graph

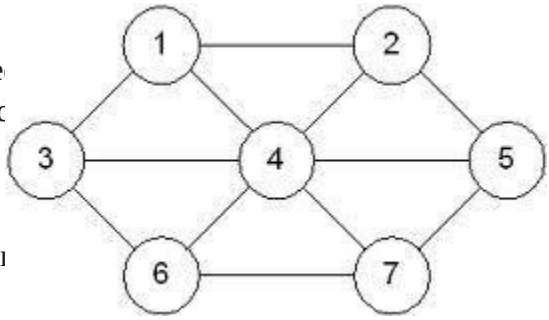
Directed Graph:



In representing of graph there is a directions are shov edges then that graph is calledDirected graph.
That is,
A graph G=(V, E) is a directed graph ,Edge is a

pair (v, w), where v, w ∈ V, and the pair is ordered.M vertex 'w' is adjacent to v.
Directed graph is also called digraph.

Undirected Graph:



In graph vertices are not ordered is called undirecte Means in which (graph) there is no direction (arrow heac line (edge).
A graph G=(V, E) is a directed graph ,Edge is a pair

(v, w), where v, w ∈ V, and the pair is not ordered. Mear 'w' is adjacent to 'v', and vertex 'v' is adjacent to 'w'

Difference between Graph and Tree

Feature	Graph	Tree
1. Definition	A structure with nodes connected by edges (may be cyclic)	A hierarchical structure with nodes connected acyclically
2. Root Node	Not mandatory	Always has a single root node
3. Direction	Can be directed or undirected	Typically directed from root to leaves
4. Connectivity	May or may not be fully connected	Always connected from root to all nodes
5. Cycles	Cycles are allowed	Cycles are strictly not allowed
6. Edge Count	No fixed edge count	Exactly n−1n - 1 edges for nn nodes
7. Node Relationship	No strict parent-child relationships	Each node has one parent (except root); clear hierarchy
8. Traversal Techniques	DFS, BFS, Dijkstra, Floyd-Warshall, A*	DFS, BFS, Preorder, Inorder, Postorder
9. Complexity	Can be complex depending on type and connectivity	Generally simpler due to strict acyclic structure
10. Common Applications	Networking, road maps, AI search algorithms	File systems, decision-making, XML data representation

Graph Isomorphism

Graph isomorphism assesses if two graphs possess the same structure, despite potential differences in vertex or edge labeling. Isomorphic graphs share the same number of vertices, edges, and connectivity. The graph isomorphism problem focuses on determining if such structural equivalence exists and examining the computational complexity involved in this determination.

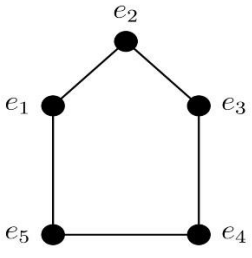
To check if two graphs are isomorphic, one needs to determine if a bijection exists between their vertex sets that preserves the adjacency relationship. Essentially, you're looking for a way to rename the vertices of one graph so that it becomes structurally identical to the other.

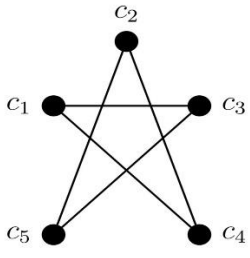
Here's a breakdown of the process:

1. Check for Basic Properties:
   - **Same Number of Vertices:** The two graphs must have the same number of vertices.
   - Same Number of Edges: The two graphs must have the same number of edges.
   - Same Degree Sequence: The number of vertices with each degree must be the same in both graphs.
2. Look for Invariants:
   - Cycle Structure:

     If the graphs have different numbers of cycles of a particular length (e.g., triangles, quadrilaterals), they cannot be isomorphic.
   - Other Invariants:
     There are other graph properties that can be used as invariants, like the chromatic number, clique number, etc. If any of these differ, the graphs are not isomorphic.
3. If Basic Properties and Invariants Match, Search for a Bijection:
   - If the basic checks and invariants hold, you'll need to try to find a specific mapping (bijection) between the vertices of the two graphs.
   - This mapping should preserve adjacency, meaning if two vertices are connected by an edge in the first graph, their corresponding vertices in the second graph must also be connected by an edge.
   - If you can find such a mapping, the graphs are isomorphic.
   - If you can't find such a mapping after a thorough search, the graphs are not isomorphic.

Example 1:





Let's analyze whether the two graphs shown in the image are **isomorphic**.

Observations:

- Both graphs have **5 vertices**.
- The vertices in the first graph are labeled e1,e2,e3,e4,e5$e1,e2,e3,e4,e5$.
- The vertices in the second graph are labeled c1,c2,c3,c4,c5$c1,c2,c3,c4,c5$.

Step 1: Degree of each vertex

- **First graph:**
  - e1$e1$: connected to e2,e5$e2,e5$,$e5$ ⇒ degree 2
  - e2$e2$: connected to e1,e3$e1,e3$,$e3$ ⇒ degree 2
  - e3$e3$: connected to e2,e4$e2,e4$,$e4$ ⇒ degree 2
  - e4$e4$: connected to e3,e5$e3,e5$,$e5$ ⇒ degree 2
  - e5$e5$: connected to e1,e4$e1,e4$,$e4$ ⇒ degree 2
- **Second graph:**
  - c1$c1$: connected to c3,c5$c3,c5$,$c5$ ⇒ degree 2
  - c2$c2$: connected to c3$c3$ ⇒ degree 1
  - c3$c3$: connected to c1,c2,c4$c1,c2,c4$ ⇒ degree 3
  - c4$c4$: connected to c3,c5$c3,c5$,$c5$ ⇒ degree 2
  - c5$c5$: connected to c1,c4$c1,c4$,$c4$ ⇒ degree 2

Step 2: Comparing degrees

- In the first graph, **all vertices have degree 2**.
- In the second graph, vertices c2$c2$ has degree 1, and c3$c3$ has degree 3, while others have degree 2.

Conclusion:

Since the degree sequences differ, the graphs **cannot be isomorphic**.

Representation of Graphs

Here are the two most common ways to represent a graph : For simplicity, we are going to consider only unweighted graphs in this post.

1. Adjacency Matrix
2. Adjacency List

Adjacency Matrix Representation

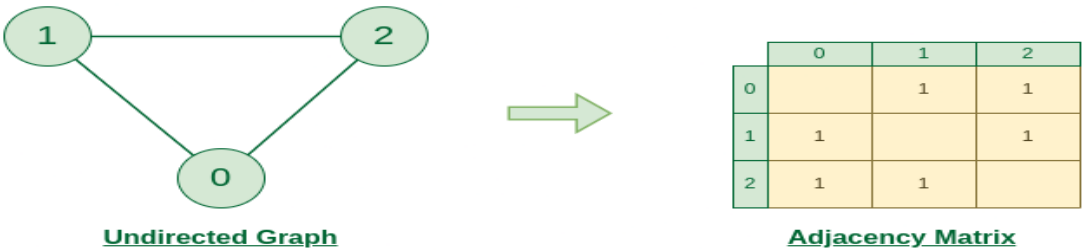
An adjacency matrix is a way of representing a graph as a matrix of boolean (0's and 1's)

Let's assume there are **n** vertices in the graph So, create a 2D matrix **adjMat[n][n]** having dimension n x n.

- If there is an edge from vertex **i** to **j**, mark **adjMat[i][j]** as **1**.
- If there is no edge from vertex **i** to **j**, mark **adjMat[i][j]** as **0**.

Representation of Undirected Graph as Adjacency Matrix:

The below figure shows an undirected graph. Initially, the entire Matrix is initialized to **0**.
If there is an edge from source to destination, we insert **1** to both cases
(**adjMat[source][destination]** and **adjMat**[**destination][source])** because we can go either way.



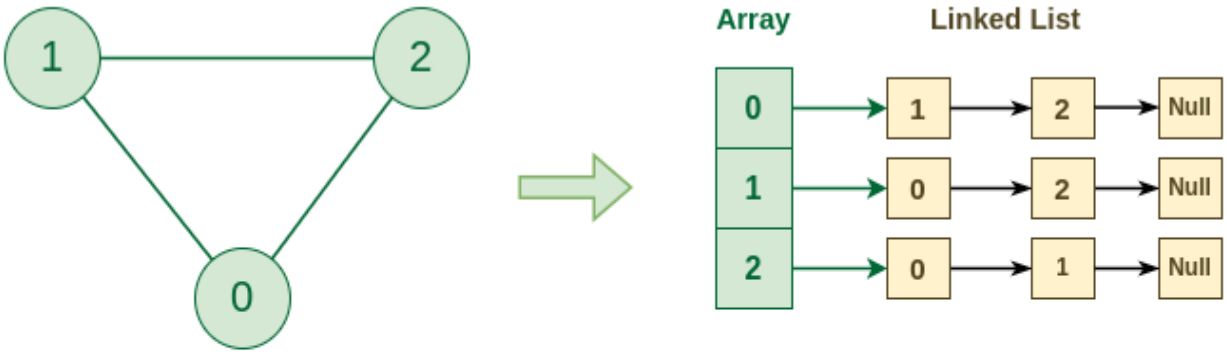
Undirected Graph Adjacency Matrix

Graph Representation of Undirected graph to Adjacency Matrix

Adjacency List Representation
An array of Lists is used to store edges between two vertices. The size of array is equal to the number of **vertices (i.e, n)**. Each index in this array represents a specific vertex in the graph. The entry at the index i of the array contains a linked list containing the vertices that are adjacent to vertex **i**. Let's assume there are **n** vertices in the graph So, create an **array of list** of size **n** as **adjList[n].**

- *adjList[0] will have all the nodes which are connected (neighbour) to vertex **0**.*
- *adjList[1] will have all the nodes which are connected (neighbour) to vertex **1** and so on.*

Representation of Undirected Graph as Adjacency list:
The below undirected graph has 3 vertices. So, an array of list will be created of size 3, where each indices represent the vertices. Now, vertex 0 has two neighbours (i.e, 1 and 2). So, insert vertex 1 and 2 at indices 0 of array. Similarly, For vertex 1, it has two neighbour (i.e, 2 and 0) So, insert vertices 2 and 0 at indices 1 of array. Similarly, for vertex 2, insert its neighbours in array of list.



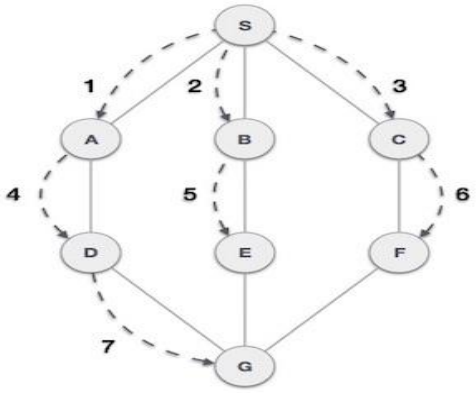
Undirected Graph Adjacency List

Graph Representation of Undirected graph to Adjacency List

Breadth-first traversal

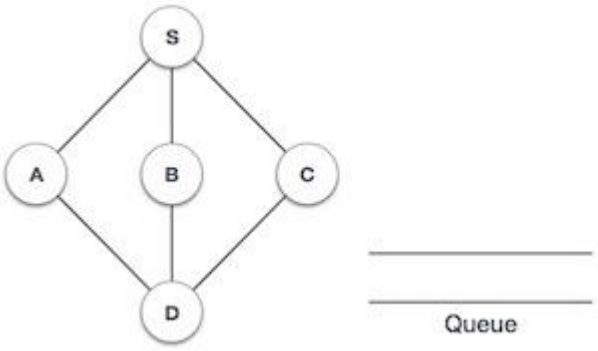
Breadth First Search (BFS) algorithm traverses a graph in a breadthward motion to search a graph data structure for a node that meets a set of criteria. It uses a queue to remember the next vertex to start a search, when a dead end occurs in any iteration.

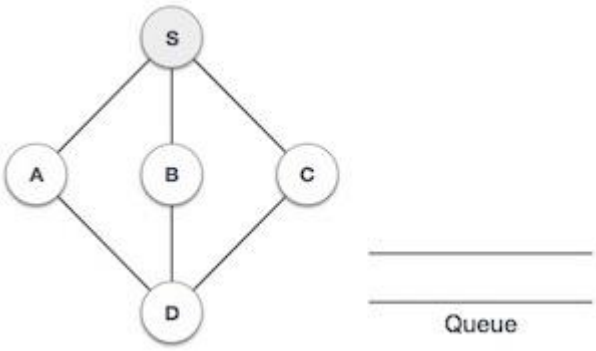
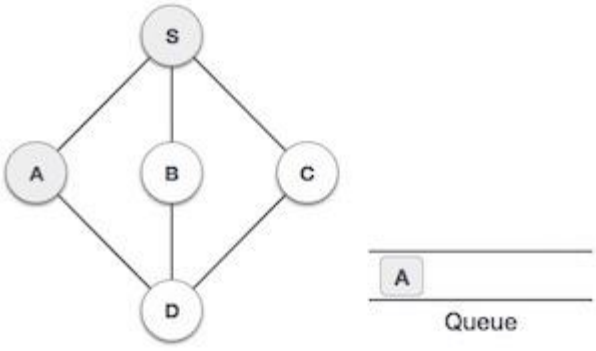
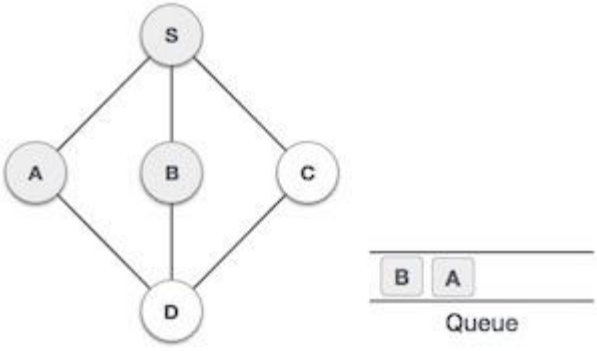
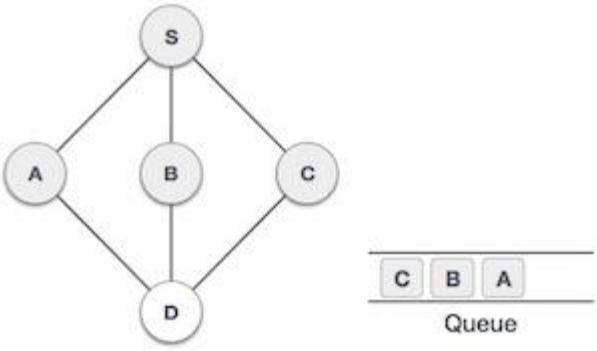
Breadth First Search (BFS) algorithm starts at the tree root and explores all nodes at the present depth prior to moving on to the nodes at the next depth level.

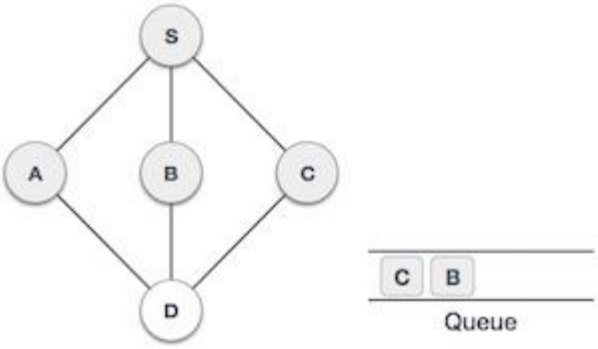
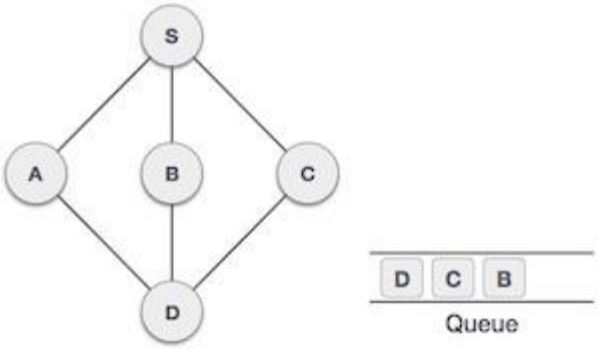


As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs the following rules.

- **Rule 1** − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
- **Rule 2** − If no adjacent vertex is found, remove the first vertex from the queue.
- **Rule 3** − Repeat Rule 1 and Rule 2 until the queue is empty.

Step	Traversal	Description
1		Initialize the queue.

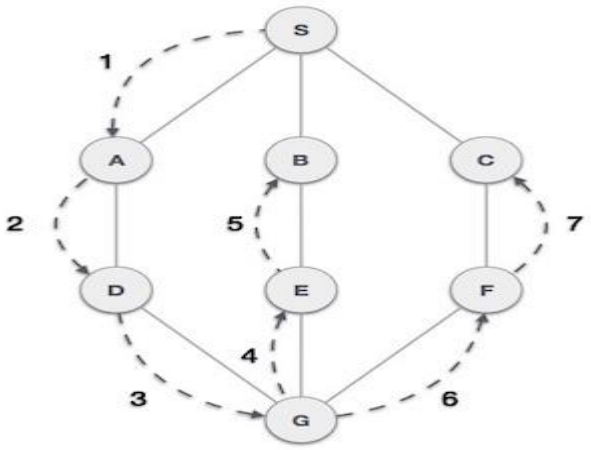
2		We start from visiting **S** (starting node), and mark it as visited.
3		We then see an unvisited adjacent node from **S**. In this example, we have three nodes but alphabetically we choose **A**, mark it as visited and enqueue it.
4		Next, the unvisited adjacent node from **S** is **B**. We mark it as visited and enqueue it.
5		Next, the unvisited adjacent node from **S** is **C**. We mark it as visited and enqueue it.

6		Now, **S** is left with no unvisited adjacent nodes. So, we dequeue and find **A**.
7		From **A** we have **D** as unvisited adjacent node. We mark it as visited and enqueue it.

At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the program is over.

Depth First Search (DFS) Algorithm

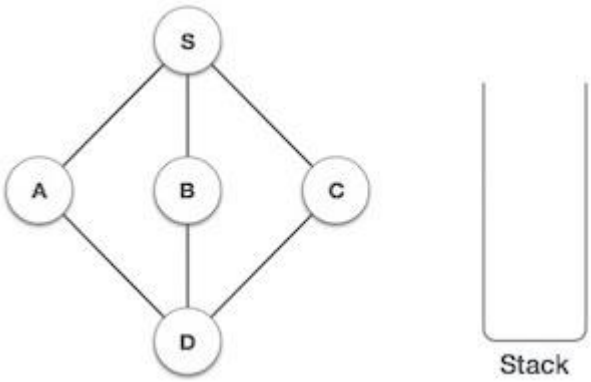
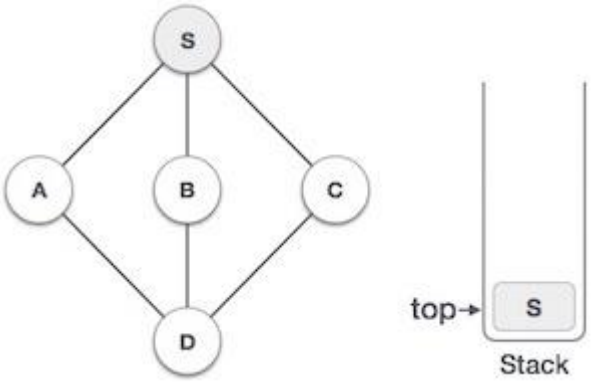
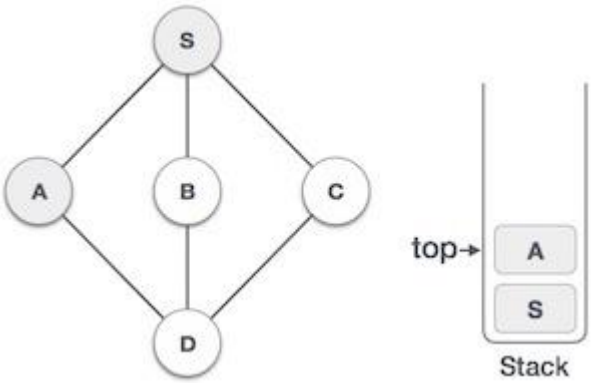
Depth First Search (DFS) algorithm is a recursive algorithm for searching all the vertices of a graph or tree data structure. This algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

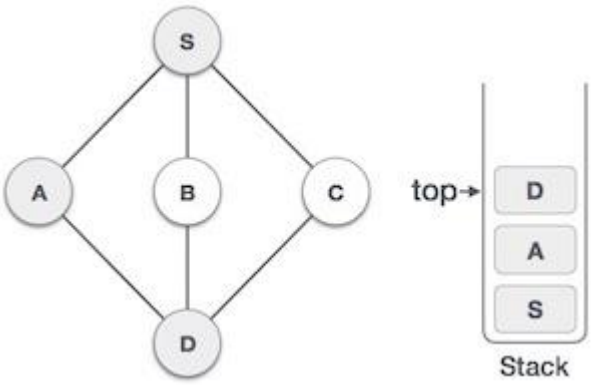
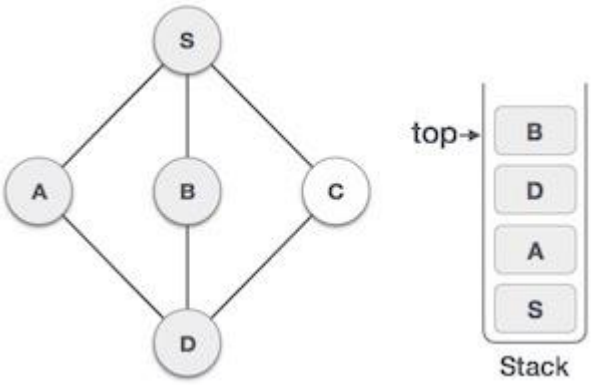
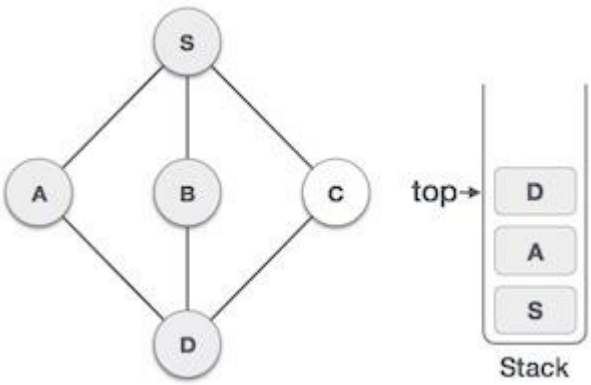
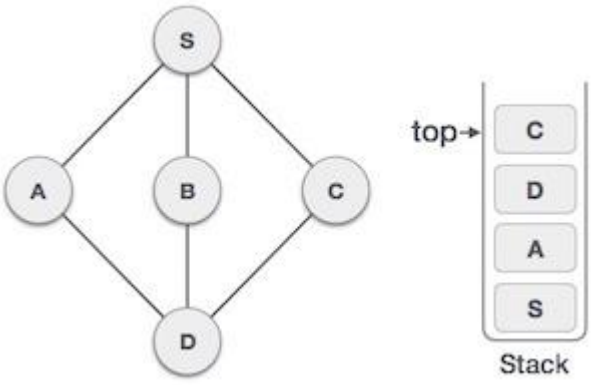


As in the example given above, DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to C. It employs the following rules.

- **Rule 1** − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.

- **Rule 2** − If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
- **Rule 3** − Repeat Rule 1 and Rule 2 until the stack is empty.

Step	Traversal	Description
1		Initialize the stack.
2		Mark **S** as visited and put it onto the stack. Explore any unvisited adjacent node from **S**. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.
3		Mark **A** as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both **S** and **D** are adjacent to **A** but we are concerned for unvisited nodes only.

4		Visit **D** and mark it as visited and put onto the stack. Here, we have **B** and **C** nodes, which are adjacent to **D** and both are unvisited. However, we shall again choose in an alphabetical order.
5		We choose **B**, mark it as visited and put onto the stack. Here **B** does not have any unvisited adjacent node. So, we pop **B** from the stack.
6		We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find **D** to be on the top of the stack.
7		Only unvisited adjacent node is from **D** is **C** now. So we visit **C**, mark it as visited and put it onto the stack.

As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited adjacent node. In this case, there's none and we keep popping until the stack is empty.

Bi-connectivity

Biconnectivity in graph theory refers to a graph property where the removal of any single vertex (and its incident edges) does not disconnect the graph. In simpler terms, a graph is biconnected if it remains connected even after removing any single vertex. This concept is crucial for understanding the robustness and fault tolerance of networks.

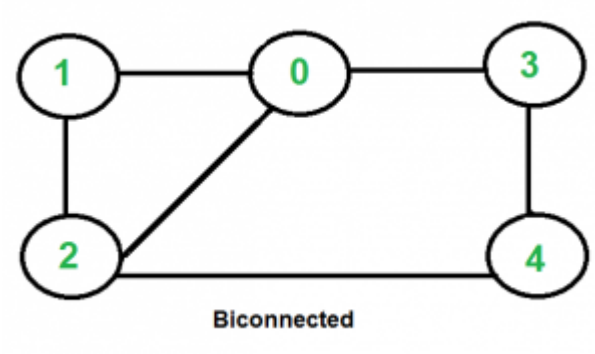
Formal Definition:

An undirected graph is biconnected if for every pair of vertices, there exist at least two vertex-disjoint paths between them. This means that even if one path is broken, there is always another independent path available to connect the two vertices.

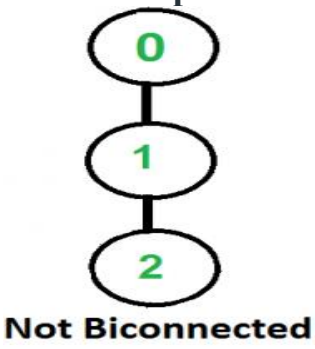
Example:

Consider a social network where users are represented as vertices and friendships as edges. If the network is biconnected, losing one user doesn't isolate any other user from the rest of the network. Even if a user leaves, there are still alternative paths for communication and interaction between any two remaining users.

Example-1



Example-2



Euler circuits

An Euler circuit is a path within a graph that traverses every edge exactly once and returns to the starting vertex. A graph has an Euler circuit if and only if all its vertices have even degrees (number of edges connected to the vertex) and the graph is connected.

Example:

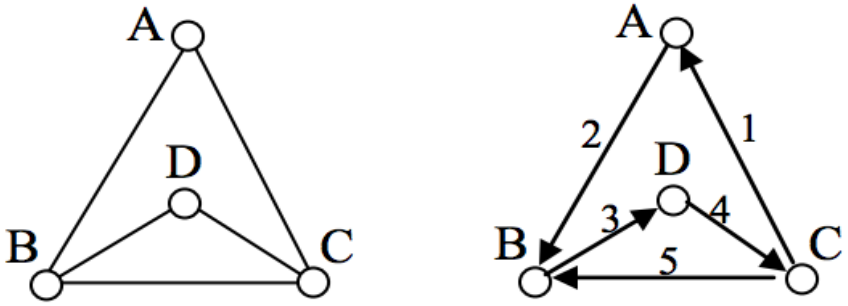
Consider a graph in the shape of a square (quadrilateral). Each corner of the square is a vertex, and each side is an edge. This graph has four vertices, and each vertex has a degree of 2 (even). A possible Euler circuit could be: starting at any vertex, traverse to an adjacent vertex, then to the next, and so on, until you return to

the initial vertex. For instance, starting at vertex A, you could go A -> B -> C -> D -> A. This path uses each edge (side of the square) exactly once and returns to the starting point.

Key Characteristics of Euler Circuits:

- **Traverses every edge:** The circuit must use every edge of the graph once and only once.
- **Starts and ends at the same vertex:** This distinguishes an Euler circuit from an Euler path, which doesn't necessarily need to end at the starting point.
- **All vertices have even degrees:** This is a necessary and sufficient condition for the existence of an Euler circuit in a connected graph.

In the graph shown below, there are several Euler paths. One such path is CABDCB. The path is shown in arrows to the right, with the order of edges numbered.



Hamiltonian circuits

A Hamiltonian Cycle or Circuit is a path in a graph that visits every vertex exactly once and returns to the starting vertex, forming a closed loop. A graph is said to be a Hamiltonian graph only when it contains a hamiltonian cycle, otherwise, it is called non-Hamiltonian graph.

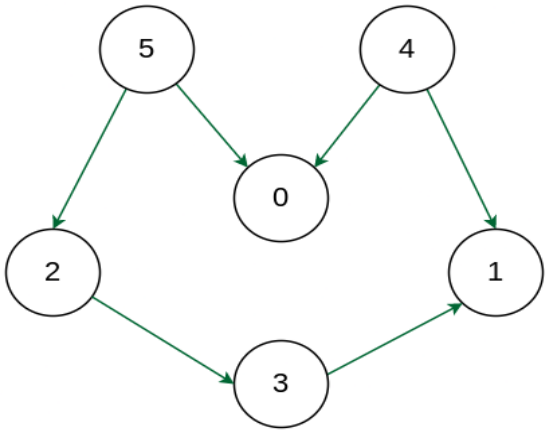
A graph is an abstract data type (ADT) consisting of a set of objects that are connected via links.

The practical applications of hamiltonian cycle problem can be seen in the fields like network design, delivery systems and many more. However, the solution to this problem can be found only for small types of graphs, and not for larger ones.

Input Output Scenario :Suppose the given undirected graph G(V, E) and its adjacency matrix are as follows −

The backtracking algorithm can be used to find a Hamiltonian path in the above graph. If found, the algorithm returns the path. If not, it returns false. For this case, the output should be (0, 1, 2, 4, 3, 0).

Topological Sort
Topological sorting for **Directed Acyclic Graph (DAG)** is a linear ordering of vertices such that for every directed edge u-v, vertex **u** comes before **v** in the ordering.
Note: Topological Sorting for a graph is not possible if the graph is not a **DAG**.
Example:
Input: *V = 6, edges = [[2, 3], [3, 1], [4, 0], [4, 1], [5, 0], [5, 2]]*



Example

Output: 5 4 2 3 1 0
Explanation: The first vertex in topological sorting is always a vertex with an in-degree of 0 (a vertex with no incoming edges). A topological sorting of the following graph is "5 4 2 3 1 0". There can be more than one topological sorting for a graph. Another topological sorting of the following graph is "4 5 2 3 1 0".

Shortest Path Algorithms

Shortest path algorithms are fundamental in graph theory and computer science, designed to find the most efficient route between nodes in a graph. They are widely used in various fields, including network routing, transportation planning, logistics, and GPS navigation systems.

Types of shortest path algorithms

Shortest path algorithms can be categorized into two main types based on their purpose:

- Single-Source Shortest Path Algorithms: These algorithms find the shortest path from a given source node to all other nodes in the graph. Examples include Dijkstra's algorithm and Bellman-Ford algorithm.
- All-Pairs Shortest Path Algorithms: These algorithms find the shortest path between every pair of nodes in the graph. Examples include Floyd-Warshall algorithm and Johnson's algorithm

Dijkstra’s Algorithm

Dijkstra’s algorithm is a popular algorithm for solving single-source shortest path problems having non-negative edge weight in the graphs i.e., it is to find the shortest distance between two vertices on a graph. It was conceived by Dutch computer scientist **Edsger W. Dijkstra** in 1956.

The algorithm maintains a set of visited vertices and a set of unvisited vertices. It starts at the source vertex and iteratively selects the unvisited vertex with the smallest tentative distance from the source. It then visits the neighbors of this vertex and updates their tentative distances if a shorter path is found. This process continues until the destination vertex is reached, or all reachable vertices have been visited.

Can Dijkstra's Algorithm work on both Directed and Undirected graphs?

Yes, Dijkstra's algorithm can work on both directed graphs and undirected graphs as this algorithm is designed to work on any type of graph as long as it meets the requirements of having non-negative edge weights and being connected.

- **In a directed graph,** each edge has a direction, indicating the direction of travel between the vertices connected by the edge. In this case, the algorithm follows the direction of the edges when searching for the shortest path.
- **In an undirected graph,** the edges have no direction, and the algorithm can traverse both forward and backward along the edges when searching for the shortest path.

Algorithm for Dijkstra's Algorithm

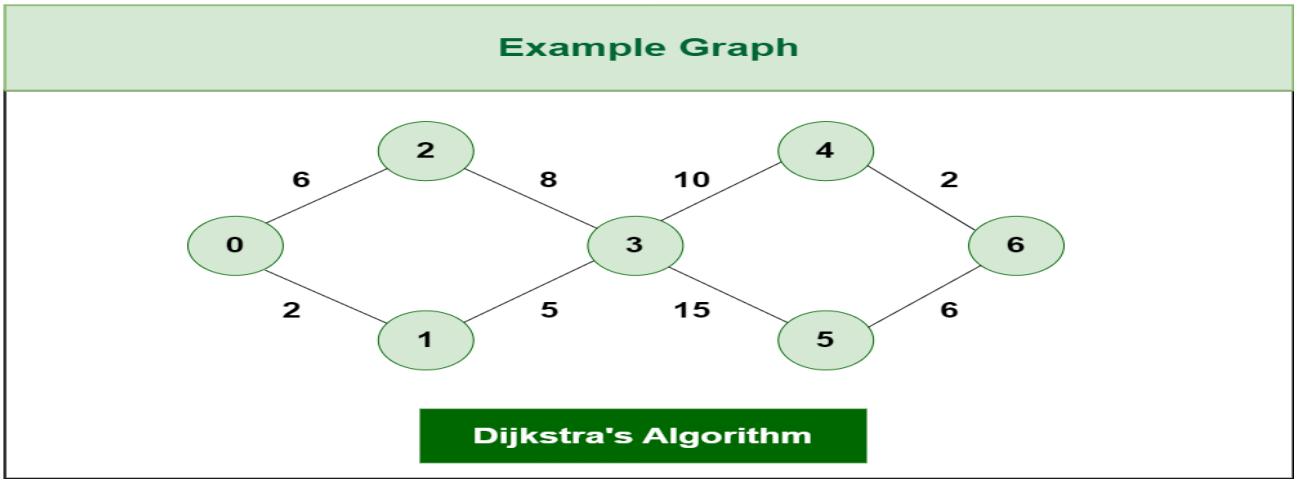
1. Mark the source node with a current distance of 0 and the rest with infinity.
2. Set the non-visited node with the smallest current distance as the current node.
3. For each neighbor, N of the current node adds the current distance of the adjacent node with the weight of the edge connecting 0->1. If it is smaller than the current distance of Node, set it as the new current distance of N.
4. Mark the current node 1 as visited.
5. Go to step 2 if there are any nodes are unvisited.

How does Dijkstra's Algorithm works?

Let's see how Dijkstra's Algorithm works with an example given below:

Dijkstra's Algorithm will generate the shortest path from Node 0 to all other Nodes in the graph.

Consider the below graph:



The algorithm will generate the shortest path from node 0 to all the other nodes in the graph.

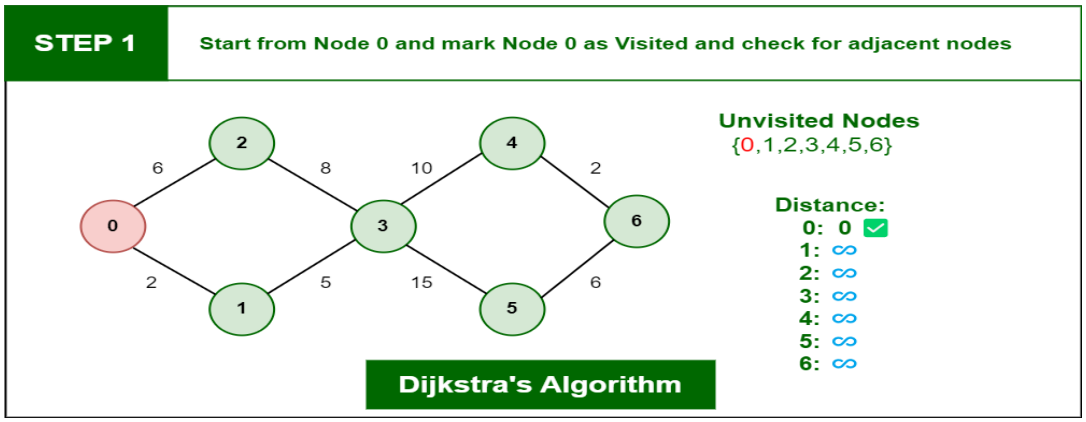
For this graph, we will assume that the weight of the edges represents the distance between two nodes.

Initially we have:

- The Distance from the source node to itself is 0. In this example the source node is 0.
- The distance from the source node to all other node is unknown so we mark all of them as infinity.

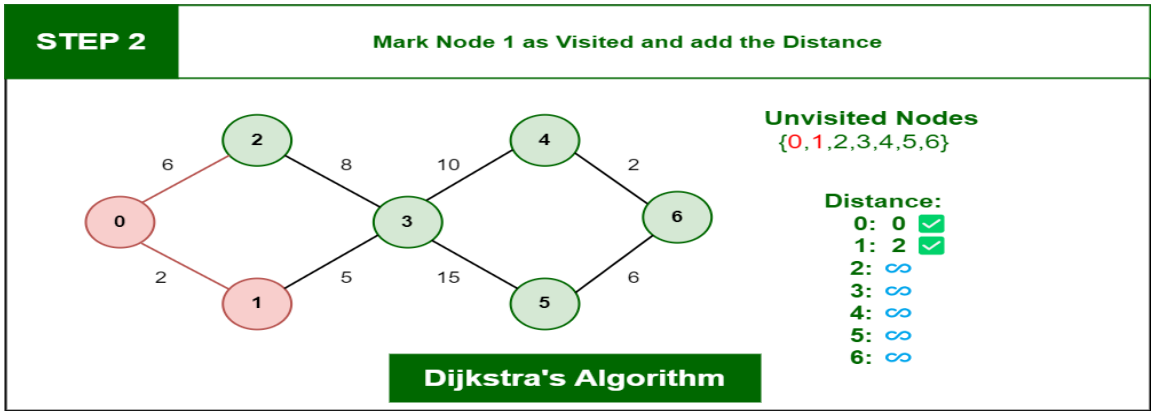
  Example: 0 -> 0, 1-> ∞,2-> ∞,3-> ∞,4-> ∞,5-> ∞,6-> ∞.
- we'll also have an array of unvisited elements that will keep track of unvisited or unmarked Nodes.
- Algorithm will complete when all the nodes marked as visited and the distance between them added to the path. **Unvisited Nodes:- 0 1 2 3 4 5 6.**

Step 1: Start from Node 0 and mark Node as visited as you can check in below image visited Node is marked red.



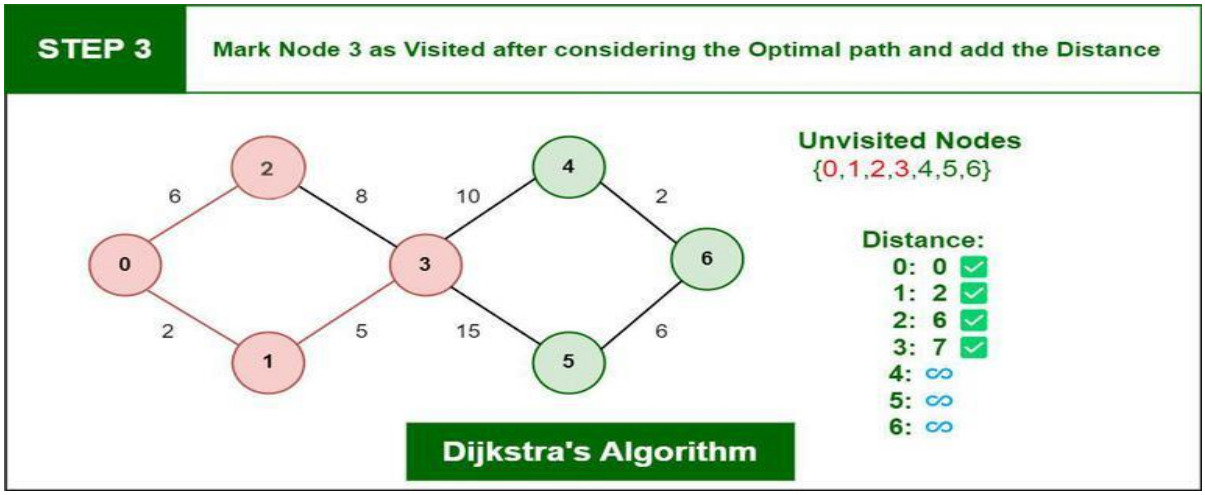
Step 2: Check for adjacent Nodes, Now we have to choices (Either choose Node1 with distance 2 or either choose Node 2 with distance 6) and choose Node with minimum distance. In this step **Node 1** is Minimum distance adjacent Node, so marked it as visited and add up the distance.

Distance: Node 0 -> Node 1 = 2



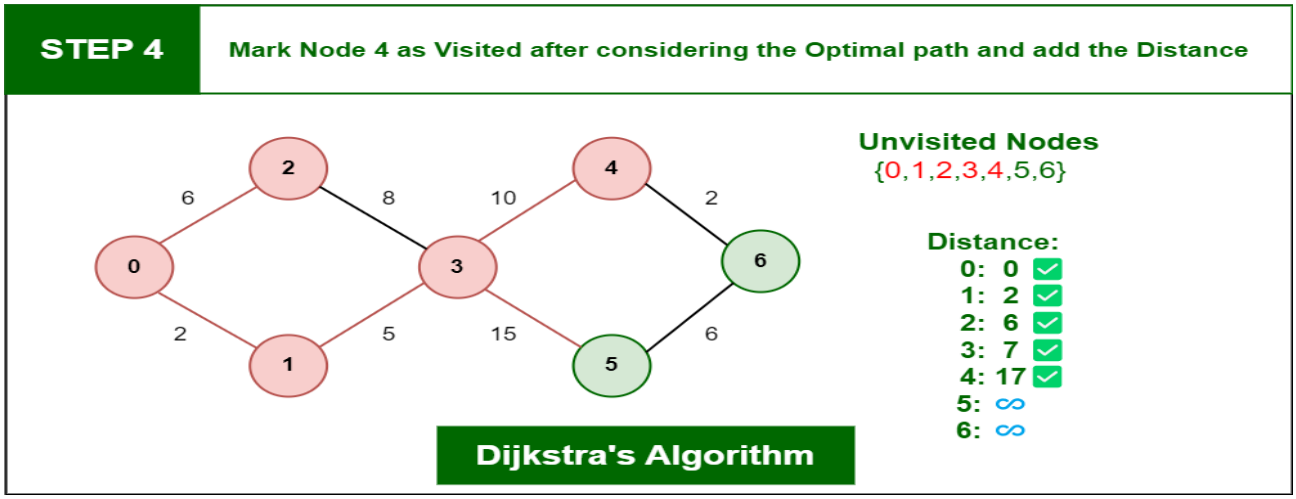
Step 3: Then Move Forward and check for adjacent Node which is Node 3, so marked it as visited and add up the distance, Now the distance will be:

Distance: Node 0 -> Node 1 -> Node 3 = 2 + 5 = 7



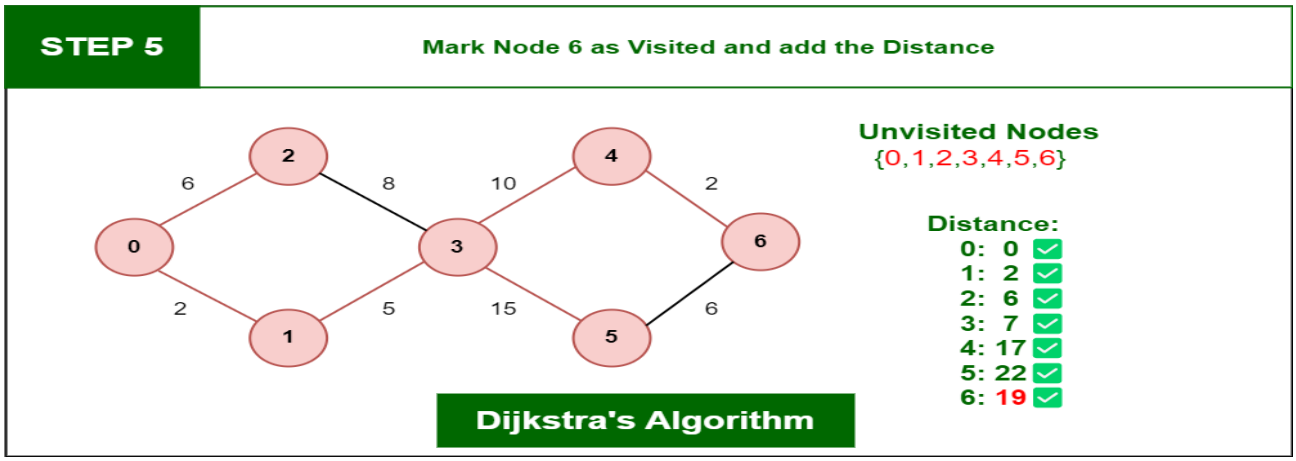
Step 4: Again we have two choices for adjacent Nodes (Either we can choose Node 4 with distance 10 or either we can choose Node 5 with distance 15) so choose Node with minimum distance. In this step **Node 4** is Minimum distance adjacent Node, so marked it as visited and add up the distance.

Distance: Node 0 -> Node 1 -> Node 3 -> Node 4 = 2 + 5 + 10 = 17



Step 5: Again, Move Forward and check for adjacent Node which is **Node 6**, so marked it as visited and add up the distance, Now the distance will be:

Distance: Node 0 -> Node 1 -> Node 3 -> Node 4 -> Node 6 = 2 + 5 + 10 + 2 = 19



So, the Shortest Distance from the Source Vertex is 19 which is optimal one

Floyd's algorithm

The Floyd-Warshall algorithm is a graph algorithm that is deployed to find the shortest path between all the vertices present in a weighted graph. This algorithm is different from other shortest path algorithms; to describe it simply, this algorithm uses each vertex in the graph as a pivot to check if it provides the shortest way to travel from one point to another.

Floyd-Warshall algorithm works on both directed and undirected weighted graphs unless these graphs do not contain any negative cycles in them. By negative cycles, it is meant that the sum of all the edges in the graph must not lead to a negative number.

Since, the algorithm deals with overlapping sub-problems the path found by the vertices acting as pivot are stored for solving the next steps it uses the dynamic programming approach.

Floyd-Warshall algorithm is one of the methods in All-pairs shortest path algorithms and it is solved using the Adjacency Matrix representation of graphs.

Floyd-Warshall Algorithm

Consider a graph, **G = {V, E}** where **V** is the set of all vertices present in the graph and E is the set of all the edges in the graph. The graph, **G**, is represented in the form of an adjacency matrix, **A**, that contains all the weights of every edge connecting two vertices.

Algorithm

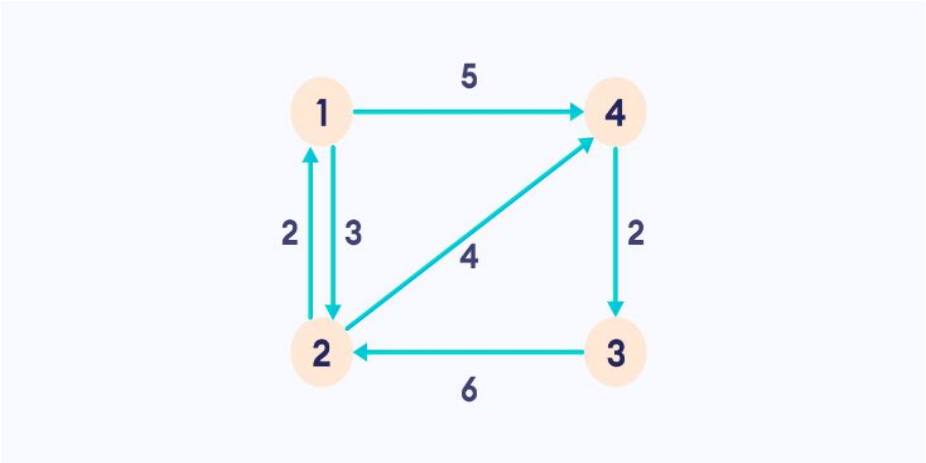
Step 1 − Construct an adjacency matrix **A** with all the costs of edges present in the graph. If there is no path between two vertices, mark the value as ∞.

Step 2 − Derive another adjacency matrix $\mathbf{A_1}$ from **A** keeping the first row and first column of the original adjacency matrix intact in $\mathbf{A_1}$. And for the remaining values, say $\mathbf{A_1[i,j]}$, if **A[i,j]>A[i,k]+A[k,j]** then replace $\mathbf{A_1[i,j]}$ with **A[i,k]+A[k,j]**. Otherwise, do not change the values. Here, in this step, **k = 1** (first vertex acting as pivot).

Step 3 − Repeat **Step 2** for all the vertices in the graph by changing the **k** value for every pivot vertex until the final matrix is achieved.

Step 4 − The final adjacency matrix obtained is the final solution with all the shortest paths.

Example Follow the steps below to find the shortest path between all the pairs of vertices.



1.Create a matrix A^0 of dimension n*n where n is the number of vertices. The row and the column are indexed as i and j respectively. i and j are the vertices of the graph.
Each cell A[i][j] is filled with the distance from the i^{th} vertex to the j^{th} vertex. If there is no path from i^{th} vertex to j^{th} vertex, the cell is left as infinity.

$$A^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & \infty & 4 \\ \infty & 1 & 0 & \infty \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{array}$$

2.Now, create a matrix A^1 using matrix A^0. The elements in the first column and the first row are left as they are. The remaining cells are filled in the following way.

Let k be the intermediate vertex in the shortest path from source to destination. In this step, k is the first vertex. A[i][j] is filled with (A[i][k] + A[k][j]) if (A[i][j] > A[i][k] + A[k][j]).

That is, if the direct distance from the source to the destination is greater than the path through the vertex k, then the cell is filled with A[i][k] + A[k][j].

In this step, k is vertex 1. We calculate the distance from source vertex to destination vertex through this vertex k

$$A^1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & & \\ \infty & & 0 & \\ \infty & & & 0 \end{bmatrix} \end{array} \rightarrow \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & \infty & 4 \\ \infty & 1 & 0 & \infty \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{array}$$

Calculate the distance from the source vertex to destination vertex through this vertex k

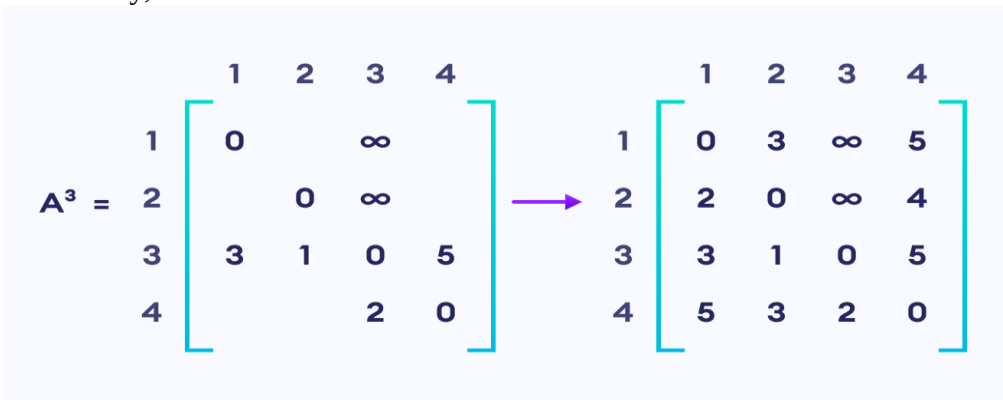
For example: For $A^1[2, 4]$, the direct distance from vertex 2 to 4 is 4 and the sum of the distance from vertex 2 to 4 through vertex (ie. from vertex 2 to 1 and from vertex 1 to 4) is 7. Since 4 < 7, $A^0[2, 4]$ is filled with 4.

3.Similarly, A^2 is created using A^1. The elements in the second column and the second row are left as they are.In this step, k is the second vertex (i.e. vertex 2). The remaining steps are the same as in **step** .

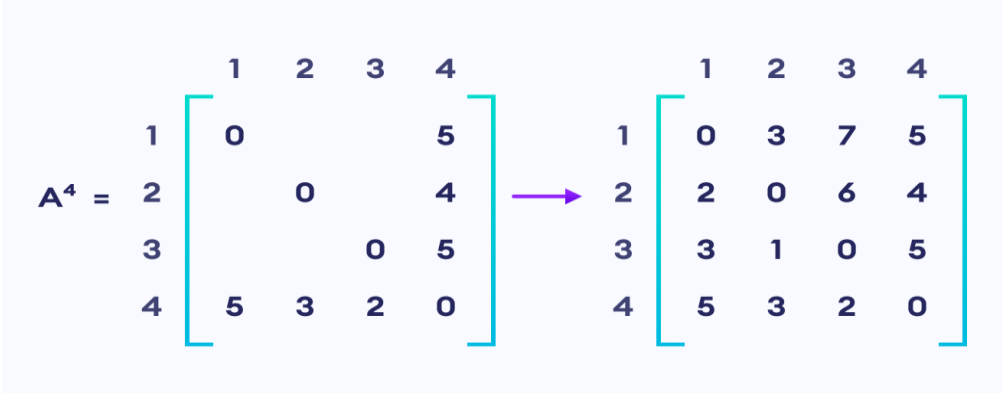
$$A^2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & & \\ 2 & 0 & \infty & 4 \\ & 1 & 0 & \\ & \infty & & 0 \end{bmatrix} \end{array} \rightarrow \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & \infty & 4 \\ 3 & 1 & 0 & 5 \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{array}$$

Calculate the distance from the source vertex to destination vertex through this vertex 2

4.Similarly, A^3 and A^4 is also created.



Calculate the distance from the source vertex to destination vertex through this vertex 3



Calculate the distance from the source vertex to destination vertex through this vertex 4

5.A^4 gives the shortest path between each pair of vertices.

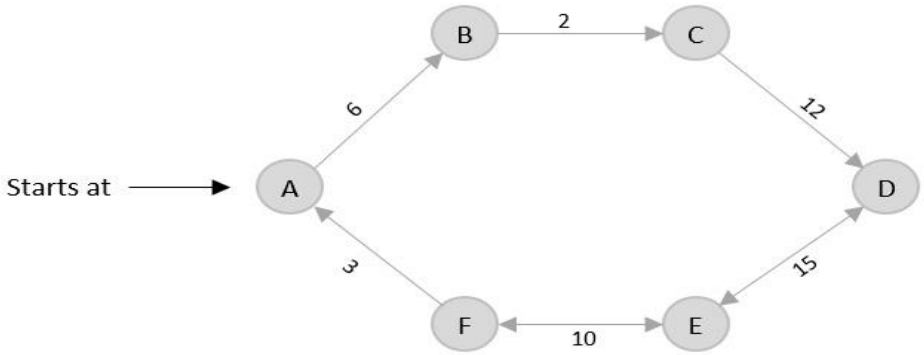
Minimum Spanning Tree

Prim's and Kruskal's algorithms are both used to find the Minimum Spanning Tree (MST) of a graph, but they differ in their approach. Prim's algorithm builds the MST by starting from a single vertex and iteratively adding the nearest vertex until all vertices are included. Kruskal's algorithm, on the other hand, adds edges in increasing order of weight, connecting different trees until all vertices are part of a single MST, while ensuring no cycles are formed.

Prim's algorithm

Prim's minimal spanning tree algorithm is one of the efficient methods to find the minimum spanning tree of a graph. A minimum spanning tree is a sub graph that connects all the vertices present in the main graph with the least possible edges and minimum cost (sum of the weights assigned to each edge).

The algorithm, similar to any shortest path algorithm, begins from a vertex that is set as a root and walks through all the vertices in the graph by determining the least cost adjacent edges.



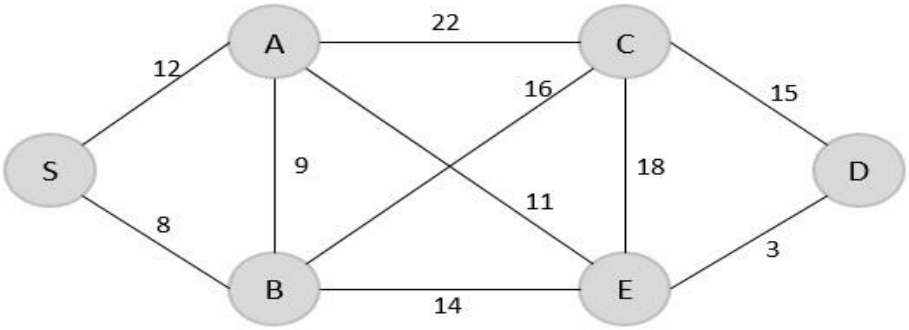
To execute the prim's algorithm, the inputs taken by the algorithm are the graph G {V, E}, where V is the set of vertices and E is the set of edges, and the source vertex S. A minimum spanning tree of graph G is obtained as an output.

Algorithm

- Declare an array *visited*[] to store the visited vertices and firstly, add the arbitrary root, say S, to the visited array.
- Check whether the adjacent vertices of the last visited vertex are present in the *visited*[] array or not.
- If the vertices are not in the *visited*[] array, compare the cost of edges and add the least cost edge to the output spanning tree.
- The adjacent unvisited vertex with the least cost edge is added into the *visited*[] array and the least cost edge is added to the minimum spanning tree output.
- Steps 2 and 4 are repeated for all the unvisited vertices in the graph to obtain the full minimum spanning tree output for the given graph.
- Calculate the cost of the minimum spanning tree obtained.

Examples

- Find the minimum spanning tree using prims method (greedy approach) for the graph given below with S as the arbitrary root.



Solution

Step 1

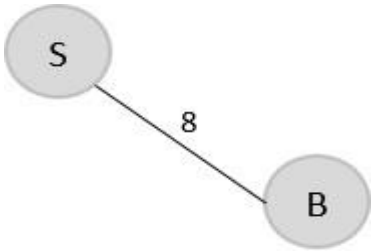
Create a visited array to store all the visited vertices into it.

V = { }

The arbitrary root is mentioned to be S, so among all the edges that are connected to S we need to find the least cost edge.

S → B = 8

V = {S, B}



Step 2

Since B is the last visited, check for the least cost edge that is connected to the vertex B.

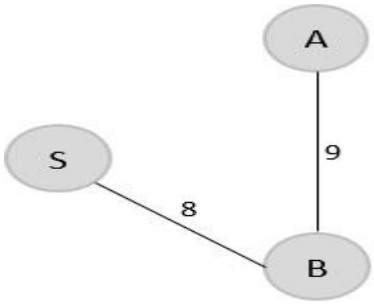
B → A = 9

B → C = 16

B → E = 14

Hence, B → A is the edge added to the spanning tree.

V = {S, B, A}



Step 3

Since A is the last visited, check for the least cost edge that is connected to the vertex A.

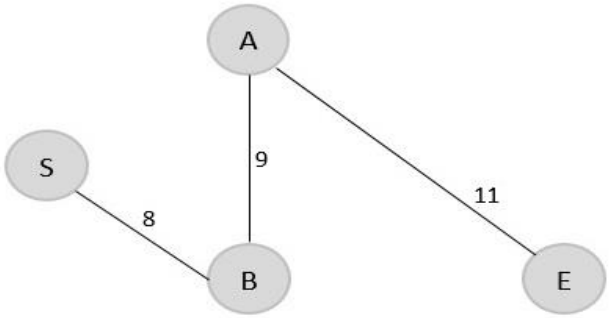
A → C = 22

A → B = 9

A → E = 11

But A → B is already in the spanning tree, check for the next least cost edge. Hence, A → E is added to the spanning tree.

V = {S, B, A, E}



Step 4

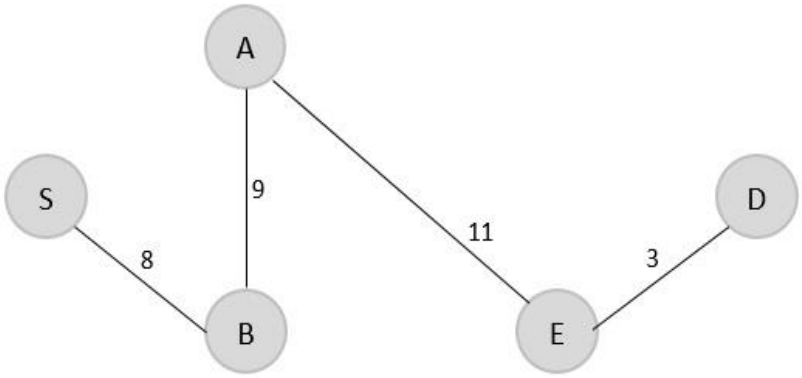
Since E is the last visited, check for the least cost edge that is connected to the vertex E.

E → C = 18

E → D = 3

Therefore, E → D is added to the spanning tree.

V = {S, B, A, E, D}



Step 5

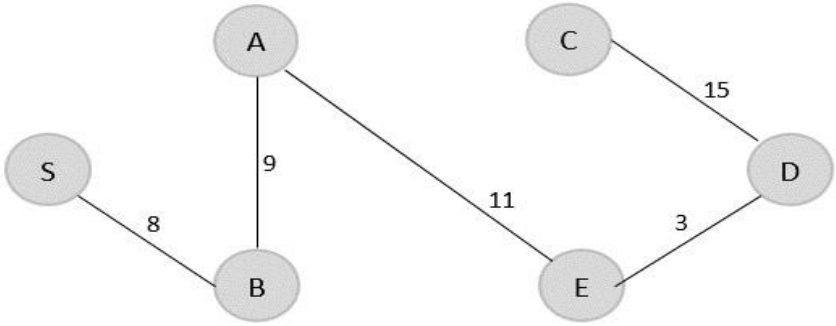
Since D is the last visited, check for the least cost edge that is connected to the vertex D.

D → C = 15

E → D = 3

Therefore, D → C is added to the spanning tree.

V = {S, B, A, E, D, C}



The minimum spanning tree is obtained with the minimum cost = 46

Kruskal's algorithm

Kruskal's minimal spanning tree algorithm is one of the efficient methods to find the minimum spanning tree of a graph. A minimum spanning tree is a subgraph that connects all the vertices present in the main graph with the least possible edges and minimum cost (sum of the weights assigned to each edge).

The algorithm first starts from the forest which is defined as a subgraph containing only vertices of the main graph of the graph, adding the least cost edges later until the minimum spanning tree is created without forming cycles in the graph.

Kruskal's algorithm has easier implementation than prims algorithm, but has higher complexity.

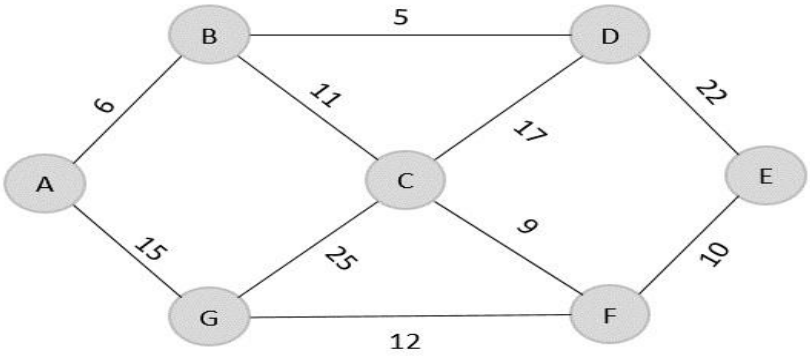
The inputs taken by the kruskals algorithm are the graph G {V, E}, where V is the set of vertices and E is the set of edges, and the source vertex S and the minimum spanning tree of graph G is obtained as an output.

Algorithm

- Sort all the edges in the graph in an ascending order and store it in an array *edge*[].
- Construct the forest of the graph on a plane with all the vertices in it.
- Select the least cost edge from the edge[] array and add it into the forest of the graph. Mark the vertices visited by adding them into the visited[] array.
- Repeat the steps 2 and 3 until all the vertices are visited without having any cycles forming in the graph
- When all the vertices are visited, the minimum spanning tree is formed.
- Calculate the minimum cost of the output spanning tree formed.

Examples

Construct a minimum spanning tree using kruskals algorithm for the graph given below −

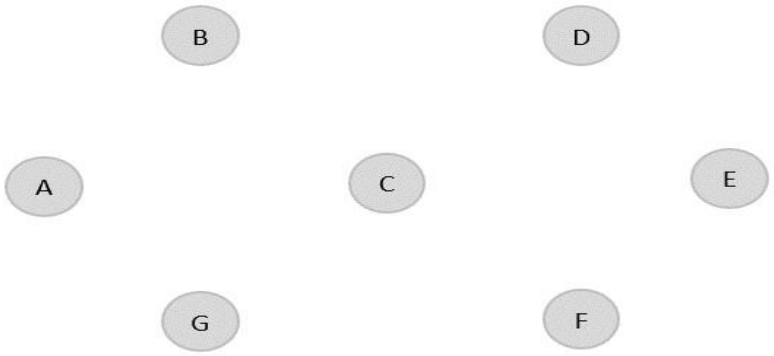


Solution

As the first step, sort all the edges in the given graph in an ascending order and store the values in an array.

Edge	B→D	A→B	C→F	F→E	B→C	G→F	A→G	C→D	D→E	C→G
Cost	5	6	9	10	11	12	15	17	22	25

Then, construct a forest of the given graph on a single plane.

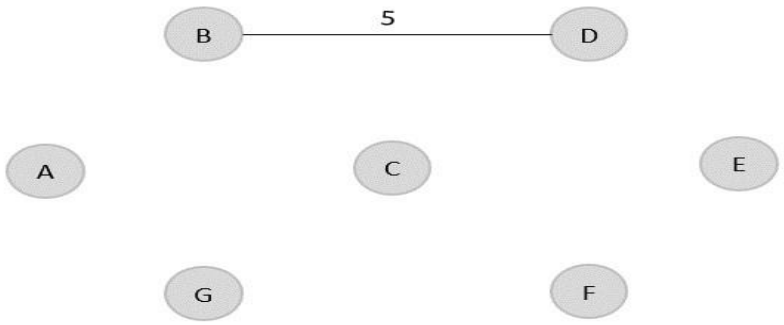


From the list of sorted edge costs, select the least cost edge and add it onto the forest in output graph.

B → D = 5

Minimum cost = 5

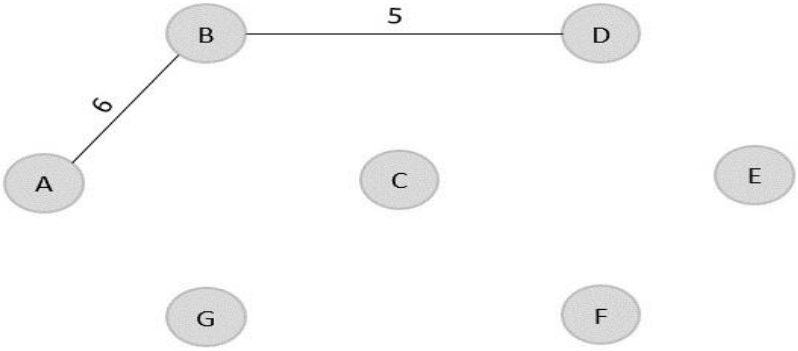
Visited array, v = {B, D}



Similarly, the next least cost edge is B → A = 6; so we add it onto the output graph.

Minimum cost = 5 + 6 = 11

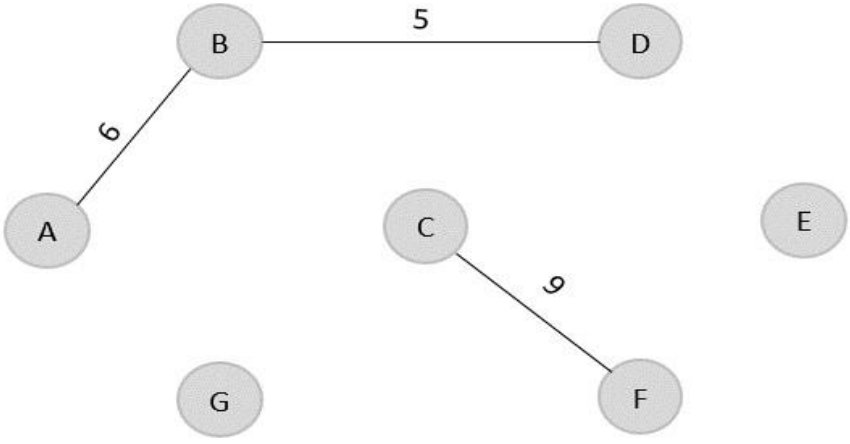
Visited array, v = {B, D, A}



The next least cost edge is C → F = 9; add it onto the output graph.

Minimum Cost = 5 + 6 + 9 = 20

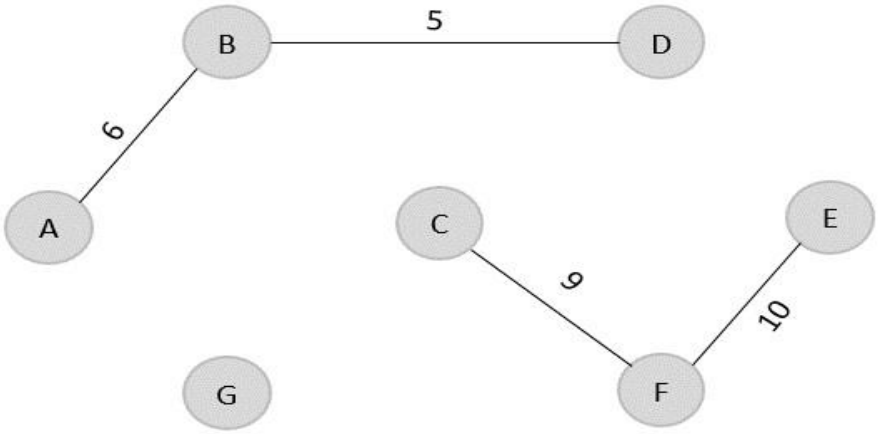
Visited array, v = {B, D, A, C, F}



The next edge to be added onto the output graph is F → E = 10.

Minimum Cost = 5 + 6 + 9 + 10 = 30

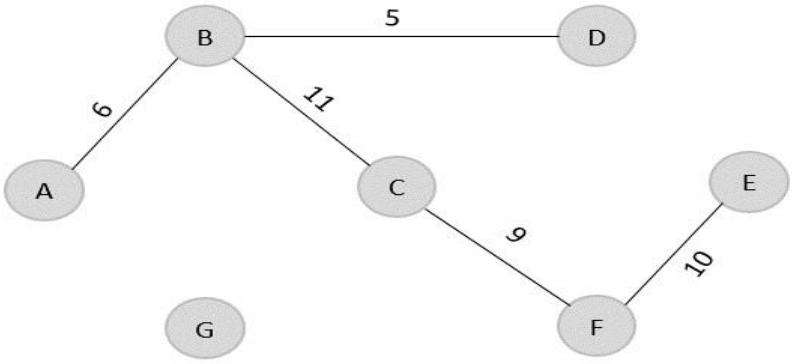
Visited array, v = {B, D, A, C, F, E}



The next edge from the least cost array is B → C = 11, hence we add it in the output graph.

Minimum cost = 5 + 6 + 9 + 10 + 11 = 41

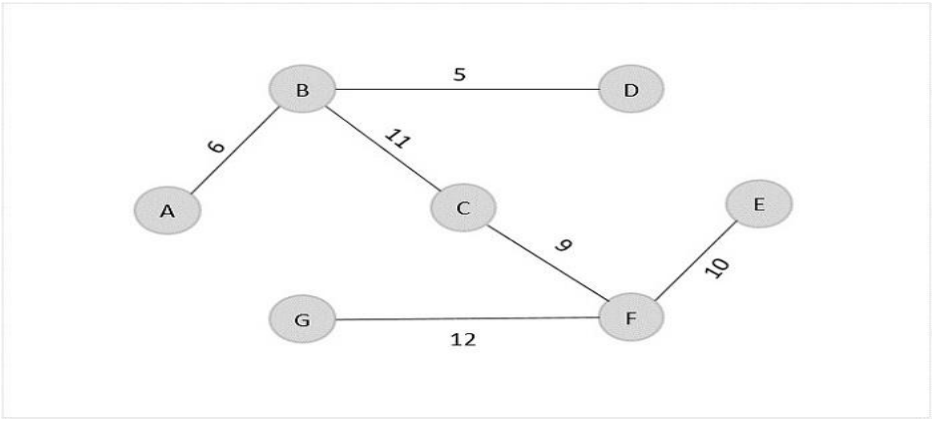
Visited array, v = {B, D, A, C, F, E}



The last edge from the least cost array to be added in the output graph is F → G = 12.

Minimum cost = 5 + 6 + 9 + 10 + 11 + 12 = 53

Visited array, v = {B, D, A, C, F, E, G}



The obtained result is the minimum spanning tree of the given graph with cost = 53.

Applications of Graph

- In Computer science graphs are used to represent the flow of computation.
- Google maps uses graphs for building transportation systems, where intersection of two(or more) roads are considered to be a vertex and the road connecting two vertices is considered to be an edge, thus their navigation system is based on the algorithm to calculate the shortest path between two vertices.
- In Facebook, users are considered to be the vertices and if they are friends then there is an edge running between them. Facebook's Friend suggestion algorithm uses graph theory. Facebook is an example of undirected graph.
- In World Wide Web, web pages are considered to be the vertices. There is an edge from a page u to other page v if there is a link of page v on page u. This is an example of Directed graph. It was the basic idea behind Google Page Ranking Algorithm.
- In Operating System, we come across the Resource Allocation Graph where each process and resources are considered to be vertices. Edges are drawn from resources to the allocated process, or from requesting process to the requested resource. If this leads to any formation of a cycle then a deadlock will occur.
- In mapping system we use graph. It is useful to find out which is an excellent place from the location as well as your nearby location. In GPS we also use graphs.
- Facebook uses graphs. Using graphs suggests mutual friends. it shows a list of the f following pages, friends, and contact list.
- Microsoft Excel uses DAG means Directed Acyclic Graphs.
- In the Dijkstra algorithm, we use a graph. we find the smallest path between two or many nodes.
- On social media sites, we use graphs to track the data of the users. liked showing preferred post suggestions, recommendations, etc.
- Graphs are used in biochemical applications such as structuring of protein, DNA etc.

UNIT V SEARCHING, SORTING AND HASHING TECHNIQUES

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.

Searching: Linear Search and Binary Search are two fundamental algorithms used to find a specific element within a collection of data, typically an array or list. They differ significantly in their approach and efficiency.

Linear search

Linear search is a type of sequential searching algorithm. In this method, every element within the input array is traversed and compared with the key element to be found. If a match is found in the array the search is said to be successful; if there is no match found the search is said to be unsuccessful and gives the worst-case time complexity.

For instance, in the given animated diagram, we are searching for an element 33. Therefore, the linear search method searches for it sequentially from the very first element until it finds a match. This returns a successful search.

In the same diagram, if we have to search for an element 46, then it returns an unsuccessful search since 46 is not present in the input.

Linear Search Algorithm

The algorithm for linear search is relatively simple. The procedure starts at the very first index of the input array to be searched.

Step 1 − Start from the 0th index of the input array, compare the key value with the value present in the 0th index.

Step 2 − If the value matches with the key, return the position at which the value was found.

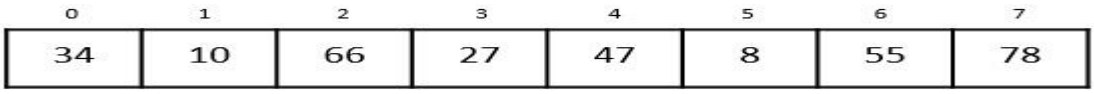
Step 3 − If the value does not match with the key, compare the next element in the array.

Step 4 − Repeat Step 3 until there is a match found. Return the position at which the match was found.

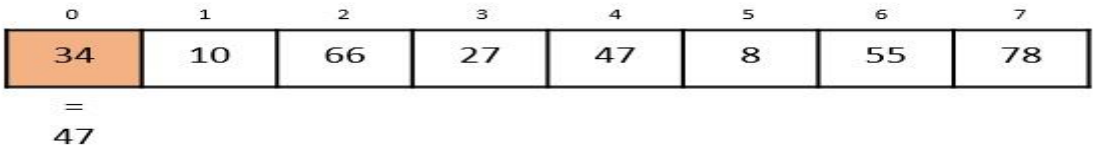
Step 5 − If it is an unsuccessful search, print that the element is not present in the array and exit the program.

Example

Let us look at the step-by-step searching of the key element (say 47) in an array using the linear search method.

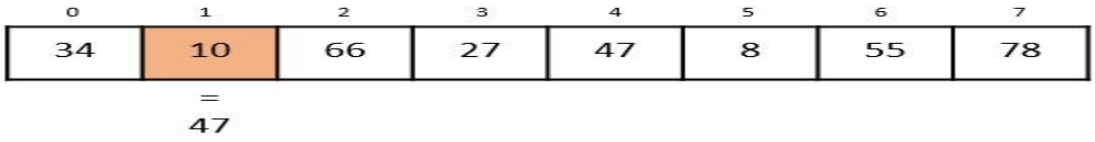


Step 1 : The linear search starts from the 0^{th} index. Compare the key element with the value in the 0^{th} index, 34.



However, 47 34. So it moves to the next element.

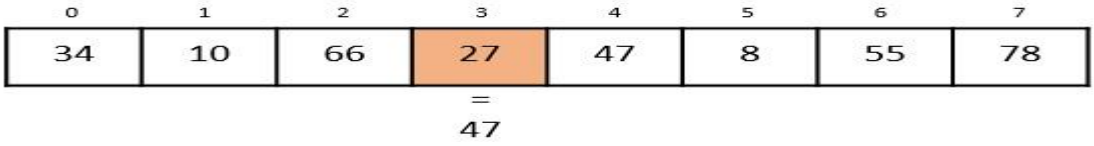
Step 2 : Now, the key is compared with value in the 1st index of the array.



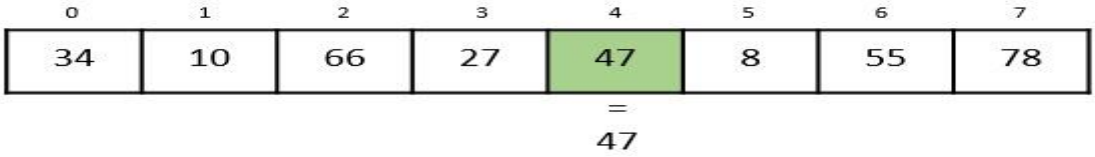
Still, 47 10, making the algorithm move for another iteration.

Step 3 : The next element 66 is compared with 47. They are both not a match so the algorithm compares the further elements.

Step 4 : Now the element in 3rd index, 27, is compared with the key value, 47. They are not equal so the algorithm is pushed forward to check the next element.



Step 5: Comparing the element in the 4^{th} index of the array, 47, to the key 47. It is figured that both the elements match. Now, the position in which 47 is present, i.e., 4 is returned.

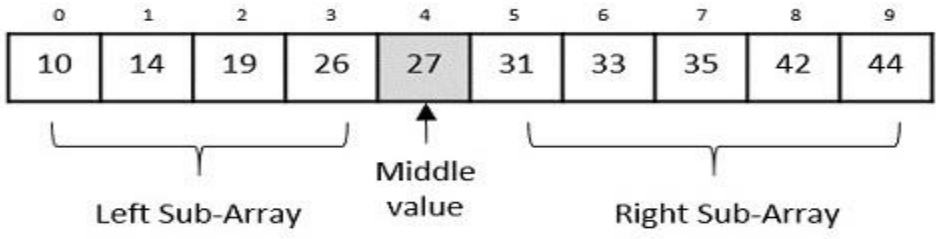


The output achieved is Element found at 4th index.

Binary Search:

Binary search is a fast search algorithm with run-time complexity of (log n). This search algorithm works on the principle of divide and conquer, since it divides the array into half before searching. For this algorithm to work properly, the data collection should be in the sorted form.

Binary search looks for a particular key value by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. But if the middle item has a value greater than the key value, the right sub-array of the middle item is searched. Otherwise, the left sub-array is searched. This process continues recursively until the size of a subarray reduces to zero.



Binary Search Algorithm

Binary Search algorithm is an interval searching method that performs the searching in intervals only. The input taken by the binary search algorithm must always be in a sorted array since it divides the array into subarrays based on the greater or lower values. The algorithm follows the procedure below −

Step 1 − Select the middle item in the array and compare it with the key value to be searched. If it is matched, return the position of the median.

Step 2 − If it does not match the key value, check if the key value is either greater than or less than the median value.

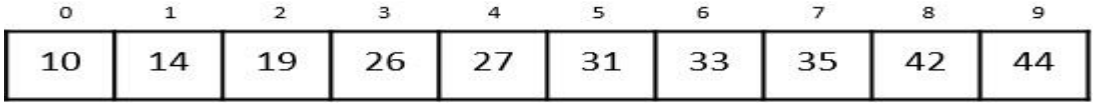
Step 3 − If the key is greater, perform the search in the right sub-array; but if the key is lower than the median value, perform the search in the left sub-array.

Step 4 − Repeat Steps 1, 2 and 3 iteratively, until the size of sub-array becomes 1.

Step 5 − If the key value does not exist in the array, then the algorithm returns an unsuccessful search.

Example

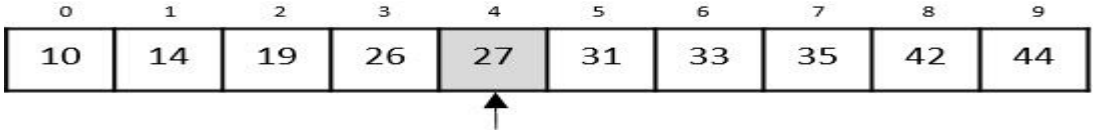
For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with a pictorial example. The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.



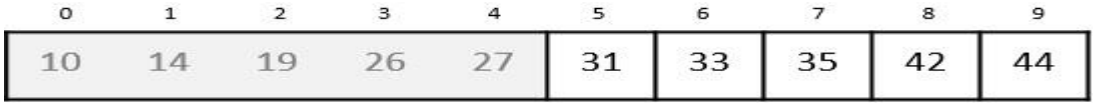
First, we shall determine half of the array by using this formula −

mid = low + (high - low) / 2

Here it is, 0 + (9 - 0) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.

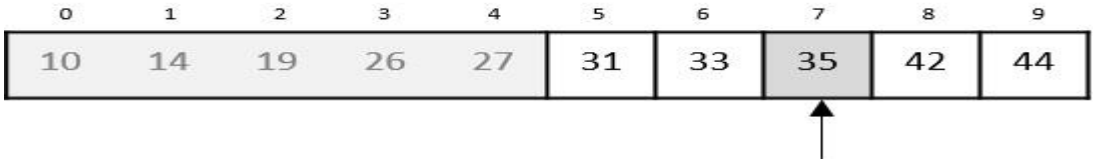


We change our low to mid + 1 and find the new mid value again.

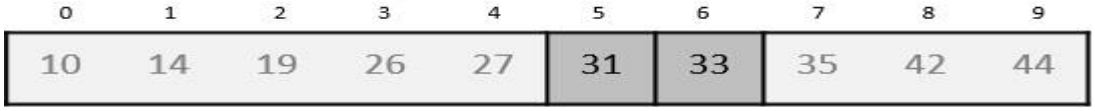
low = mid + 1

mid = low + (high - low) / 2

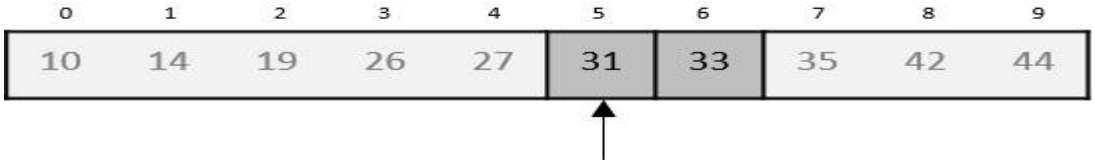
Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.



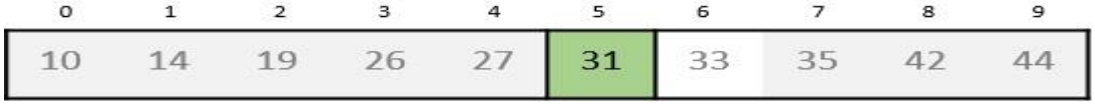
The value stored at location 7 is not a match, rather it is less than what we are looking for. So, the value must be in the lower part from this location.



Hence, we calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.



We conclude that the target value 31 is stored at location 5.

Sorting

Sorting is the process of arranging items in a specific order. Several algorithms can achieve this, including bubble sort, selection sort, insertion sort, shell sort, and radix sort, each with its own approach to arranging data.

Bubble sort

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$ where n is the number of items.

Bubble Sort Algorithm

Bubble Sort is an elementary sorting algorithm, which works by repeatedly exchanging adjacent elements, if necessary. When no exchanges are required, the file is sorted.

We assume list is an array of n elements. We further assume that swap function swaps the values of the given array elements.

Step 1 − Check if the first element in the input array is greater than the next element in the array.

Step 2 − If it is greater, swap the two elements; otherwise move the pointer forward in the array.

Step 3 − Repeat Step 2 until we reach the end of the array.

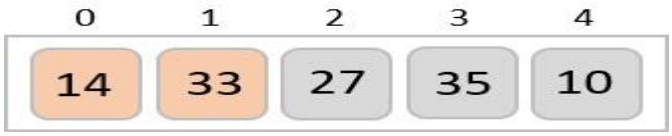
Step 4 − Check if the elements are sorted; if not, repeat the same process (Step 1 to Step 3) from the last element of the array to the first.

Step 5 − The final output achieved is the sorted array.

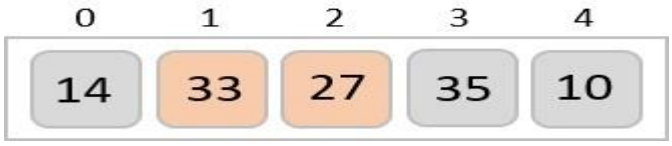
Example

We take an unsorted array for our example. Bubble sort takes (n2) time so we're keeping it short and precise.

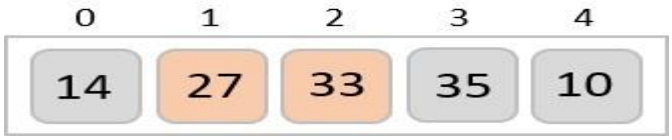
Bubble sort starts with very first two elements, comparing them to check which one is greater.



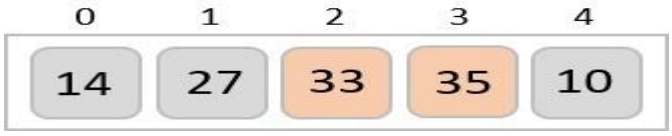
In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.



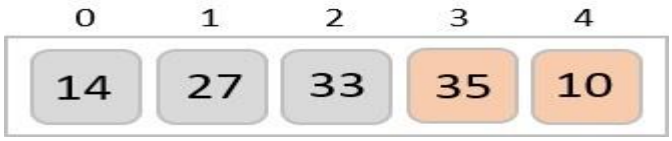
We find that 27 is smaller than 33 and these two values must be swapped.



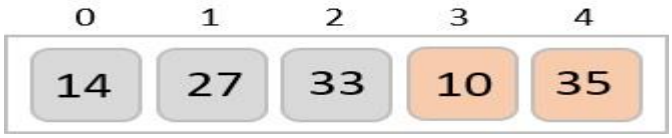
Next we compare 33 and 35. We find that both are in already sorted positions.



Then we move to the next two values, 35 and 10.

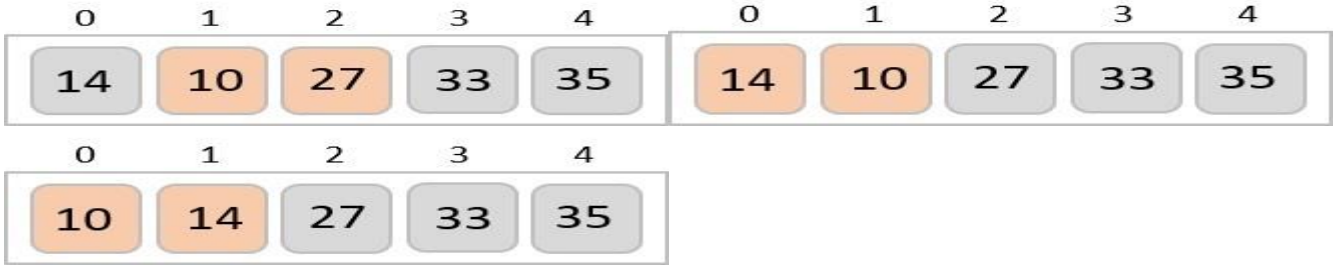


We know then that 10 is smaller 35. Hence they are not sorted. We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −



To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −

Notice that after each iteration, at least one value moves at the end.



And when there's no swap required, bubble sort learns that an array is completely sorted.

Selection sort :

Selection sort is a simple sorting algorithm. This sorting algorithm, like insertion sort, is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundaries by one element to the right.

This algorithm is not suitable for large data sets as its average and worst case complexities are of $O(n^2)$, where n is the number of items.

Selection Sort Algorithm

This type of sorting is called Selection Sort as it works by repeatedly sorting elements. That is: we first find the smallest value in the array and exchange it with the element in the first position, then find the second smallest element and exchange it with the element in the second position, and we continue the process in this way until the entire array is sorted.

1. Set MIN to location 0.

2. Search the minimum element in the list.

3. Swap with value at location MIN.

4. Increment MIN to point to next element.

5. Repeat until the list is sorted.

Example

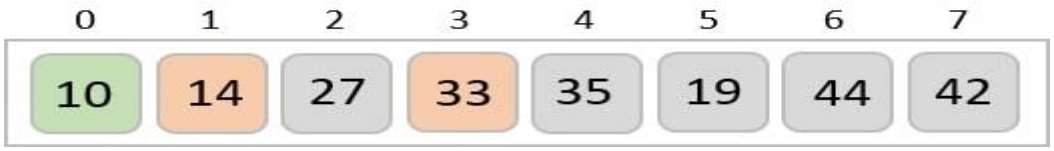
Consider the following depicted array as an example.

For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.

So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.

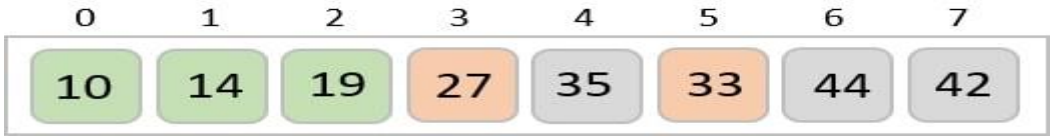
For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.

We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.



After two iterations, two least values are positioned at the beginning in a sorted manner.

The same process is applied to the rest of the items in the array −



Insertion sort

Insertion sort is a very simple method to sort numbers in an ascending or descending order. This method follows the incremental method. It can be compared with the technique how cards are sorted at the time of playing a game.

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'inserted' in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, insertion sort.

The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of (n^2), where n is the number of items.

Insertion Sort Algorithm

Now we have a bigger picture of how this sorting technique works, so we can derive simple steps by which we can achieve insertion sort.

Step 1 − If it is the first element, it is already sorted. return 1;

Step 2 − Pick next element

Step 3 − Compare with all elements in the sorted sub-list

Step 4 − Shift all the elements in the sorted sub-list that is greater than the value to be sorted

Step 5 − Insert the value

Step 6 − Repeat until list is sorted

Example

We take an unsorted array for our example.

Insertion sort compares the first two elements.

It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.

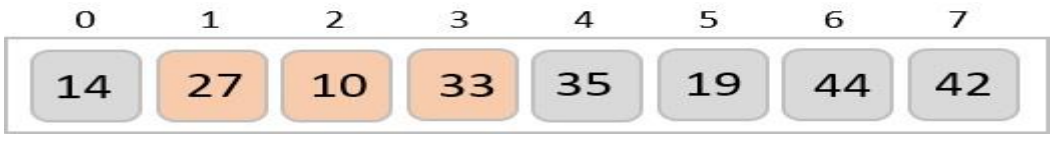
Insertion sort moves ahead and compares 33 with 27.

And finds that 33 is not in the correct position. It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.

By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10. These values are not in a sorted order.

So they are swapped.

However, swapping makes 27 and 10 unsorted.

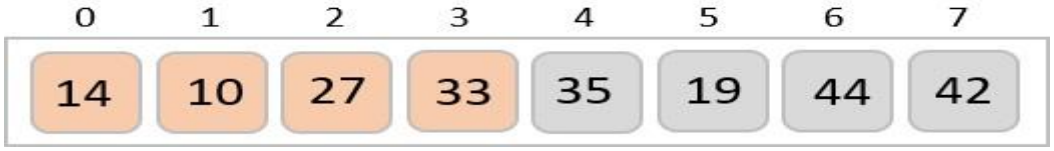


Hence, we swap them too.

Again we find 14 and 10 in an unsorted order.

We swap them again.

By the end of third iteration, we have a sorted sub-list of 4 items.



This process goes on until all the unsorted values are covered in a sorted sub-list.

Shell sort

Shell sort is a highly efficient sorting algorithm and is based on insertion sort algorithm. This algorithm avoids large shifts as in case of insertion sort, if the smaller value is to the far right and has to be moved to the far left.

This algorithm uses insertion sort on a widely spread elements, first to sort them and then sorts the less widely spaced elements. This spacing is termed as interval. This interval is calculated based on Knuth's formula as −

h = h * 3 + 1

where h is interval with initial value 1

This algorithm is quite efficient for medium-sized data sets as its average and worst case complexity are of O(n), where n is the number of items.

Shell Sort Algorithm

Following is the algorithm for shell sort.

1. Initialize the value of h.

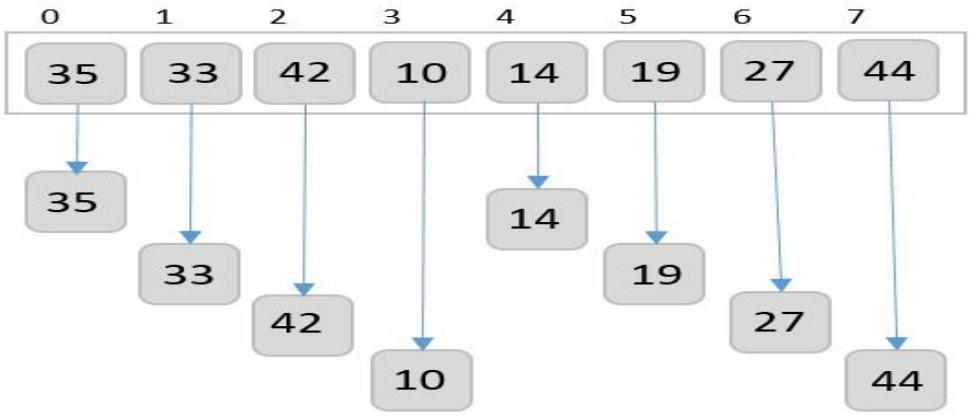
2. Divide the list into smaller sub-list of equal interval h.

3. Sort these sub-lists using insertion sort.

4. Repeat until complete list is sorted.

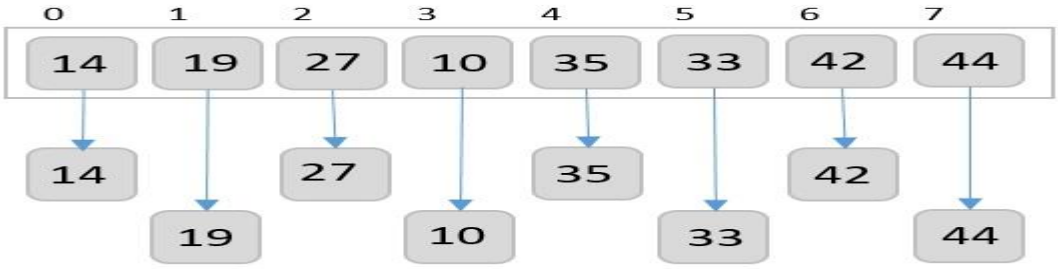
Example

Let us consider the following example to have an idea of how shell sort works. We take the same array we have used in our previous examples. For our example and ease of understanding, we take the interval of 4. Make a virtual sub-list of all values located at the interval of 4 positions. Here these values are {35, 14}, {33, 19}, {42, 27} and {10, 14}



We compare values in each sub-list and swap them (if necessary) in the original array. After this step, the new array should look like this −

Then, we take interval of 2 and this gap generates two sub-lists - {14, 27, 35, 42}, {19, 10, 33, 44}



We compare and swap the values, if required, in the original array. After this step, the array should look like this −

Finally, we sort the rest of the array using interval of value 1. Shell sort uses insertion sort to sort the array.

Following is the step-by-step depiction −

We see that it required only four swaps to sort the rest of the array.

Radix Sort

Radix sort is a step-wise sorting algorithm that starts the sorting from the least significant digit of the input elements. Like Counting Sort and Bucket Sort, Radix sort also assumes something about the input elements, that they are all k-digit numbers.

The sorting starts with the least significant digit of each element. These least significant digits are all considered individual elements and sorted first; followed by the second least significant digits. This process is continued until all the digits of the input elements are sorted.

Note − If the elements do not have same number of digits, find the maximum number of digits in an input element and add leading zeroes to the elements having less digits. It does not change the values of the elements but still makes them k-digit numbers.

Radix Sort Algorithm

The radix sort algorithm makes use of the counting sort algorithm while sorting in every phase. The detailed steps are as follows −

Step 1 − Check whether all the input elements have same number of digits. If not, check for numbers that have maximum number of digits in the list and add leading zeroes to the ones that do not.

Step 2 − Take the least significant digit of each element.

Step 3 − Sort these digits using counting sort logic and change the order of elements based on the output achieved. For example, if the input elements are decimal numbers, the possible values each digit can take would be 0-9, so index the digits based on these values.

Step 4 − Repeat the Step 2 for the next least significant digits until all the digits in the elements are sorted.

Step 5 − The final list of elements achieved after kth loop is the sorted output.

Radix sort

Radix sort is a step-wise sorting algorithm that starts the sorting from the least significant digit of the input elements. Like Counting Sort and Bucket Sort, Radix sort also assumes something about the input elements, that they are all k-digit numbers.

The sorting starts with the least significant digit of each element. These least significant digits are all considered individual elements and sorted first; followed by the second least significant digits. This process is continued until all the digits of the input elements are sorted.

Note − If the elements do not have same number of digits, find the maximum number of digits in an input element and add leading zeroes to the elements having less digits. It does not change the values of the elements but still makes them k-digit numbers.

Radix Sort Algorithm

The radix sort algorithm makes use of the counting sort algorithm while sorting in every phase. The detailed steps are as follows −

Step 1 − Check whether all the input elements have same number of digits. If not, check for numbers that have maximum number of digits in the list and add leading zeroes to the ones that do not.

Step 2 − Take the least significant digit of each element.

Step 3 − Sort these digits using counting sort logic and change the order of elements based on the output achieved. For example, if the input elements are decimal numbers, the possible values each digit can take would be 0-9, so index the digits based on these values.

Step 4 − Repeat the Step 2 for the next least significant digits until all the digits in the elements are sorted.

Step 5 − The final list of elements achieved after kth loop is the sorted output.

Example

For the given unsorted list of elements, 236, 143, 26, 42, 1, 99, 765, 482, 3, 56, we need to perform the radix sort and obtain the sorted output list −

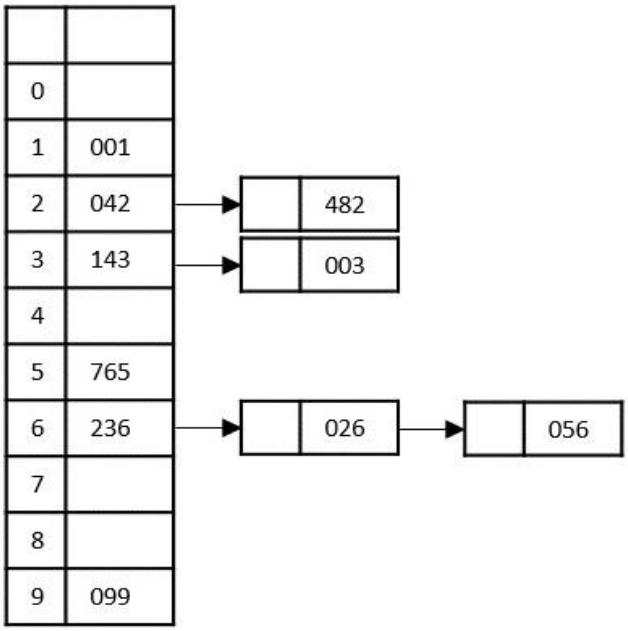
Step 1: Check for elements with maximum number of digits, which is 3. So we add leading zeroes to the numbers that do not have 3 digits. The list we achieved would be −

236, 143, 026, 042, 001, 099, 765, 482, 003, 056

Step 2: Construct a table to store the values based on their indexing. Since the inputs given are decimal numbers, the indexing is done based on the possible values of these digits, i.e., 0-9.

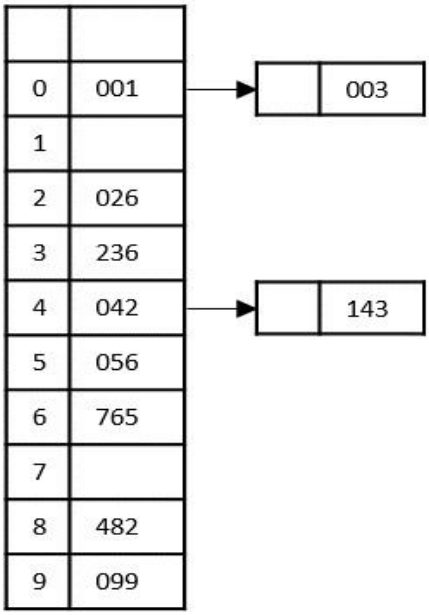
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Step 3: Based on the least significant digit of all the numbers, place the numbers on their respective indices.



The elements sorted after this step would be 001, 042, 482, 143, 003, 765, 236, 026, 056, 099.

Step 4: The order of input for this step would be the order of the output in the previous step. Now, we perform sorting using the second least significant digit.

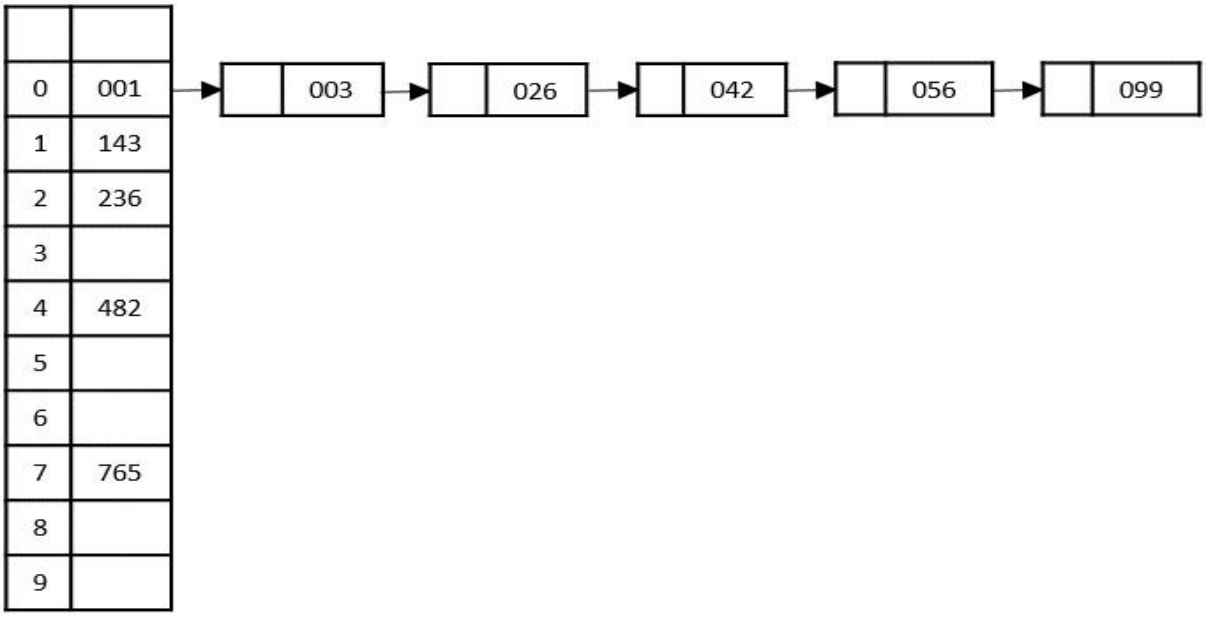


The order of the output achieved is 001, 003, 026, 236, 042, 143, 056, 765, 482, 099.

Step 5 : The input list after the previous step is rearranged as −

001, 003, 026, 236, 042, 143, 056, 765, 482, 099

Now, we need to sort the last digits of the input elements.



Since there are no further digits in the input elements, the output achieved in this step is considered as the final output.

The final sorted output is –

1, 3, 26, 42, 56, 99, 143, 236, 482, 765

Hashing

Hashing is a technique that maps data of arbitrary size to a fixed-size table using a hash function. It enables efficient storage and retrieval of data, allowing for fast search, insertion, and deletion operations, often in O(1) average time.

Hash Function: A hash function takes an input (key) and produces an index (hash value) that corresponds to a location in a hash table.
Separate Chaining: A collision resolution technique where each slot in the hash table holds a linked list. If multiple keys hash to the same slot, they are added to the linked list at that slot.

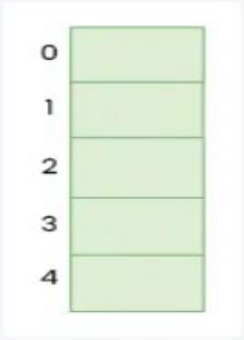
Separate Chaining

Separate Chaining is a collision handling technique. Separate chaining is one of the most popular and commonly used techniques in order to handle collisions. In this article, we will discuss about what is Separate Chain collision handling technique, its advantages, disadvantages, etc.

There are mainly two methods to handle collision: Separate Chaining Open Addressing

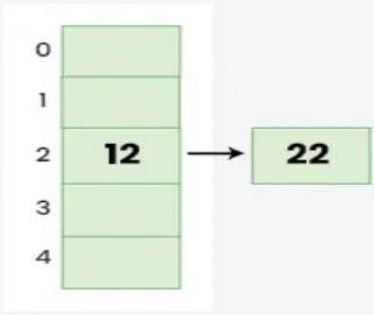
Separate Chaining: The idea behind separate chaining is to implement the array as a linked list called a chain.Linked List (or a Dynamic Sized Array) is used to implement this technique. So what happens is, when multiple elements are hashed into the same slot index, then these elements are inserted into a singly-linked list which is known as a chain.

Let us consider a simple hash function as "**key mod 5**" and a sequence of keys as 12, 22, 15, 25



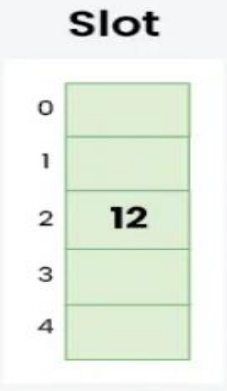
Step 01

Empty hash table with range of hash values from **0 to 4** according to the hash function provided.



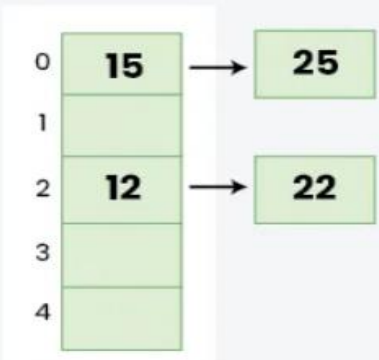
Step 03

The next key is **22** which is mapped to **slot 2 (22%5=2)** but **slot 2** is already occupied by **key 12**. Separate chaining will handle collision by creating a linked list to **slot 2.**



Step 02

The first key to be inserted is **12** which is mapped to **slot 2 (12%5=2)**.



Step 05

The next key is **25** which is mapped to **slot 0 (25%5=0)**. But slot 0 is already occupied by **key 25**. Again, Separate chaining will handle collision by creating a linked list to **slot 2.**

Open Addressing

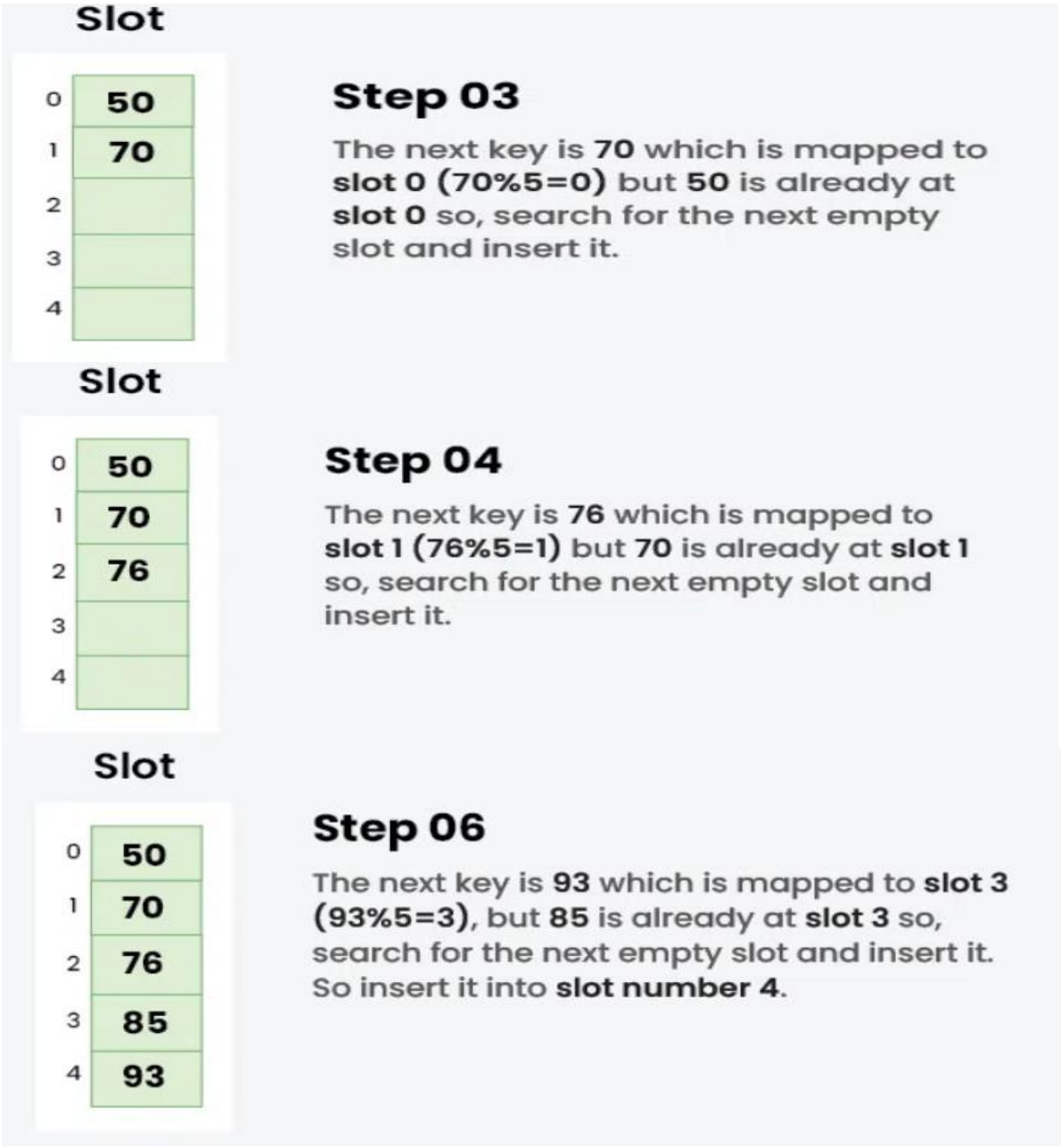
Open Addressing is a method for handling collisions. In Open Addressing, all elements are stored in the **hash table** itself. So at any point, the size of the table must be greater than or equal to the total number of keys (Note that we can increase table size by copying old data if needed). This approach is also known as closed hashing. This entire procedure is based upon probing. We will understand the types of probing ahead:

- **Insert(k):** Keep probing until an empty slot is found. Once an empty slot is found, insert k.
- **Search(k):** Keep probing until the slot's key doesn't become equal to k or an empty slot is reached.
- **Delete(k)**: **Delete operation is interesting**. If we simply delete a key, then the search may fail. So slots of deleted keys are marked specially as "deleted".
  The insert can insert an item in a deleted slot, but the search doesn't stop at a deleted slot.

Different ways of Open Addressing:

1. Linear Probing:

In linear probing, the hash table is searched sequentially that starts from the original location of the hash. If in case the location that we get is already occupied, then we check for the next location.
The function used for rehashing is as follows: rehash(key) = (n+1)%table-size.
For example, The typical gap between two probes is 1 as seen in the example below:
Let **hash(x)** be the slot index computed using a hash function and **S** be the table size
If slot hash(x) % S is full, then we try (hash(x) + 1) % S
If (hash(x) + 1) % S is also full, then we try (hash(x) + 2) % S
If (hash(x) + 2) % S is also full, then we try (hash(x) + 3) % S
Example: Let us consider a simple hash function as “key mod 5” and a sequence of keys that are to be inserted are 50, 70, 76, 85, 93.



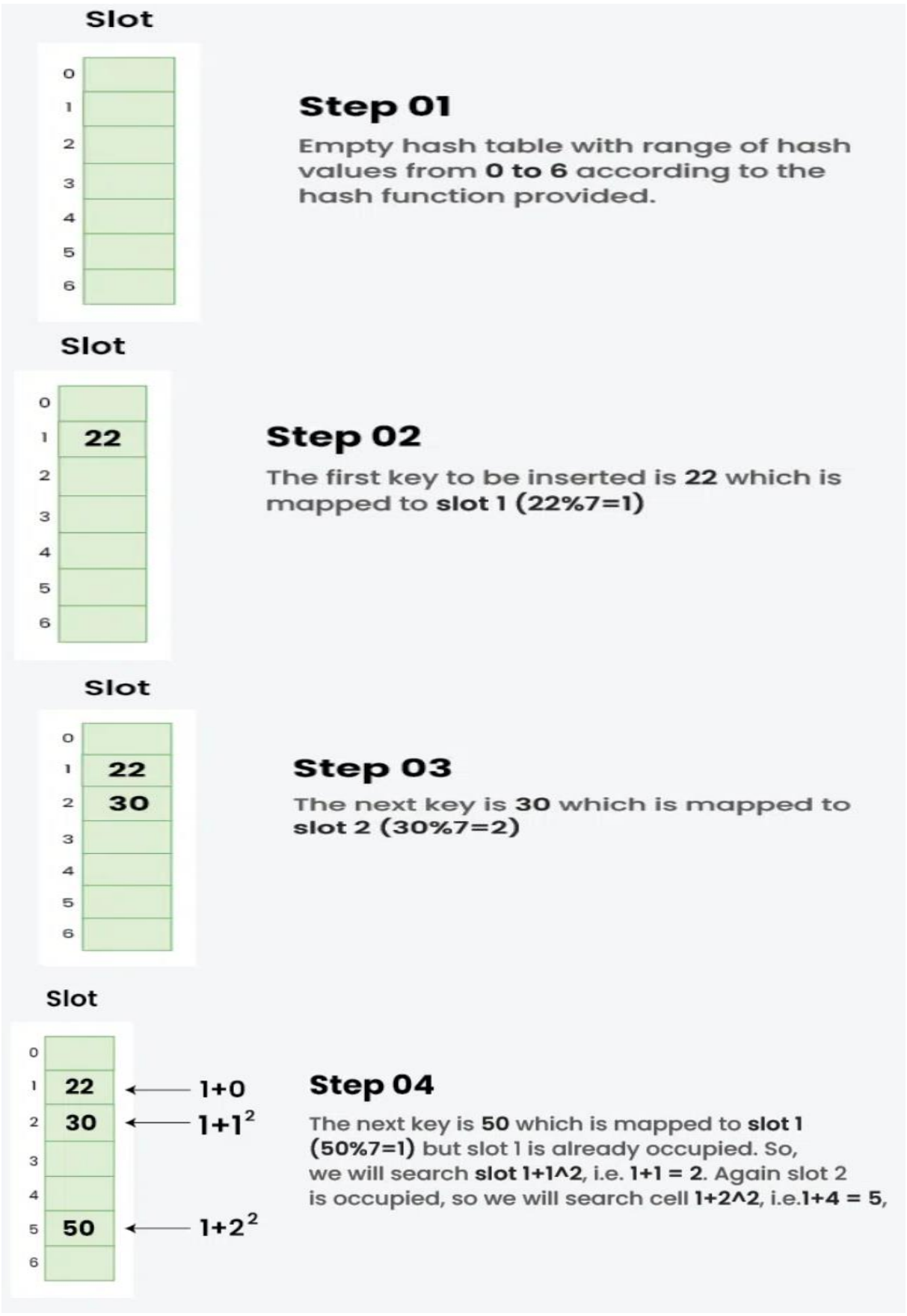
2. <u>Quadratic Probing</u>

If you observe carefully, then you will understand that the interval between probes will increase proportionally to the hash value. Quadratic probing is a method with the help of which we can solve the problem of clustering that was discussed above. This method is also known as the **mid-square** method. In this method, we look for the **i2'th** slot in the **ith** iteration. We always start from the original hash location. If only the location is occupied then we check the other slots.

let hash(x) be the slot index computed using hash function.

If slot hash(x) % S is full, then we try (hash(x) + 1*1) % S
If (hash(x) + 1*1) % S is also full, then we try (hash(x) + 2*2) % S
If (hash(x) + 2*2) % S is also full, then we try (hash(x) + 3*3) % S

Example: Let us consider table Size = 7, hash function as Hash(x) = x % 7 and collision resolution strategy to be f(i) = i2 . Insert = 22, 30, and 50.



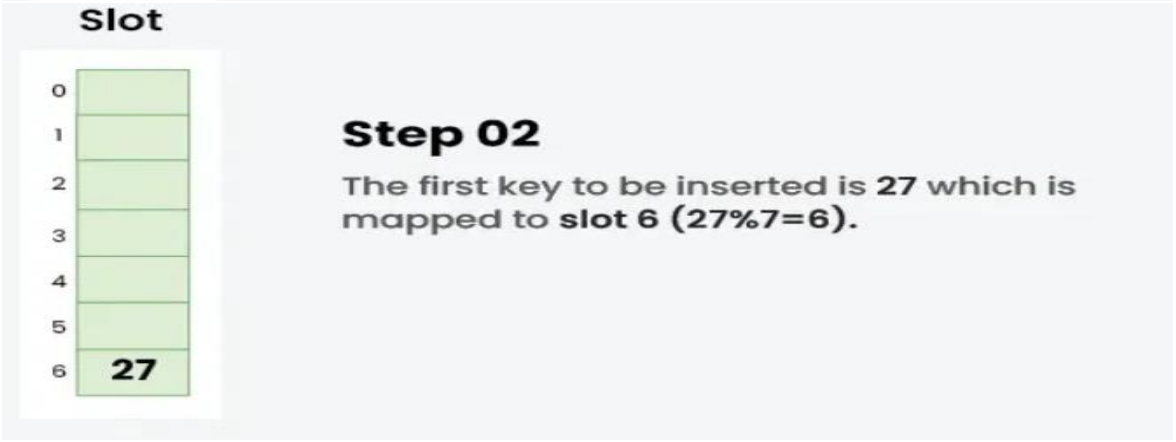
3. Double Hashing

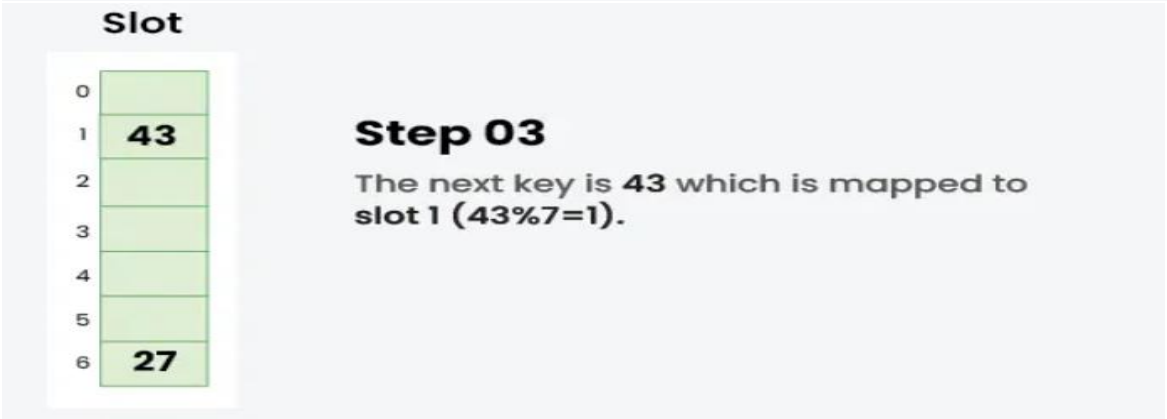
The intervals that lie between probes are computed by another hash function. Double hashing is a technique that reduces clustering in an optimized way. In this technique, the increments for the probing sequence are computed by using another hash function. We use another hash function hash2(x) and look for the i*hash2(x) slot in the **ith** rotation.

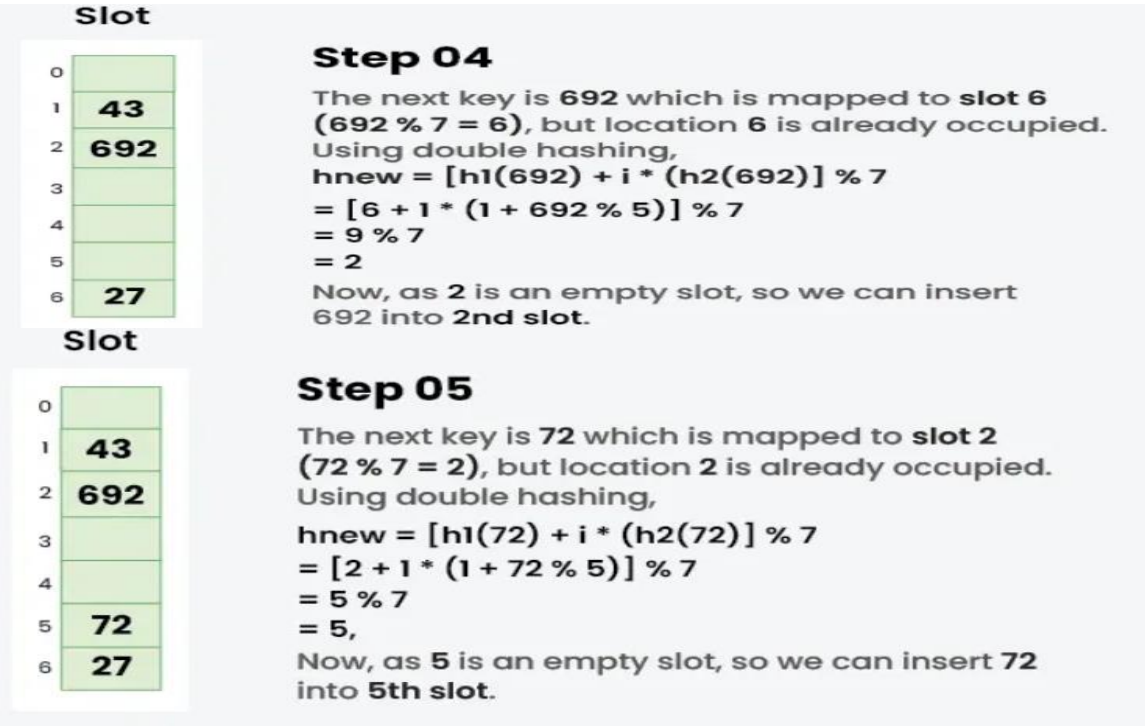
let hash(x) be the slot index computed using hash function.

If slot hash(x) % S is full, then we try (hash(x) + 1*hash2(x)) % S
If (hash(x) + 1*hash2(x)) % S is also full, then we try (hash(x) + 2*hash2(x)) % S
If (hash(x) + 2*hash2(x)) % S is also full, then we try (hash(x) + 3*hash2(x)) % S

Example: Insert the keys 27, 43, 692, 72 into the Hash Table of size 7. where first hash-function is **h1(k) = k mod 7** and second hash-function is **h2(k) = 1 + (k mod 5)**







Comparison of the above three

Open addressing is a collision handling technique used in hashing where, when a collision occurs (i.e., when two or more keys map to the same slot), the algorithm looks for another empty slot in the hash table to store the collided key.

- In **linear probing**, the algorithm simply looks for the next available slot in the hash table and places the collided key there. If that slot is also occupied, the algorithm continues searching for the next available slot until an empty slot is found. This process is repeated until all collided keys have been stored. Linear probing has the best cache performance but suffers from clustering. One more advantage of Linear probing is easy to compute.
- In **quadratic probing**, the algorithm searches for slots in a more spaced-out manner. When a collision occurs, the algorithm looks for the next slot using an equation that involves the original hash value and a quadratic function. If that slot is also occupied, the algorithm increments the value of the quadratic function and tries again. This process is repeated until an empty slot is found. Quadratic probing lies between the two in terms of cache performance and clustering.
- In **double hashing**, the algorithm uses a second hash function to determine the next slot to check when a collision occurs. The algorithm calculates a hash value using the original hash function, then uses the second hash function to calculate an offset. The algorithm then checks the slot that is the sum of the original hash value and the offset. If that slot is occupied, the algorithm increments the offset and tries again. This process is repeated until an empty slot is found. Double hashing has poor cache performance but no clustering. Double hashing requires more computation time as two hash functions need to be computed.

No.	Separate Chaining	Open Addressing
1.	Chaining is Simpler to implement.	Open Addressing requires more computation.
2.	In chaining, Hash table never fills up, we can always add more elements to chain.	In open addressing, table may become full.
3.	Chaining is Less sensitive to the hash function or load factors.	Open addressing requires extra care to avoid clustering and load factor.
4.	Chaining is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.	Open addressing is used when the frequency and number of keys is known.
5.	Cache performance of chaining is not good as keys are stored using linked list.	Open addressing provides better cache performance as everything is stored in the same table.
6.	Wastage of Space (Some Parts of hash table in chaining are never used).	In Open addressing, a slot can be used even if an input doesn't map to it.
7.	Chaining uses extra space for links.	No links in Open addressing

Rehashing

Rehashing is the process of increasing the size of a hashmap and redistributing the elements to new buckets based on their new hash values. It is done to improve the performance of the hashmap and to prevent collisions caused by a high load factor.

When a hashmap becomes full, the load factor (i.e., the ratio of the number of elements to the number of buckets) increases. As the load factor increases, the number of collisions also increases, which can lead to poor performance. To avoid this, the hashmap can be resized and the elements can be rehashed to new buckets, which decreases the load factor and reduces the number of collisions.

During rehashing, all elements of the hashmap are iterated and their new bucket positions are calculated using the new hash function that corresponds to the new size of the hashmap. This process can be time-consuming but it is necessary to maintain the efficiency of the hashmap.

Why rehashing?
Rehashing is needed in a hashmap to prevent collision and to maintain the efficiency of the data structure.

As elements are inserted into a hashmap, the load factor (i.e., the ratio of the number of elements to the number of buckets) increases. If the load factor exceeds a certain threshold (often set to 0.75), the hashmap becomes inefficient as the number of collisions increases. To avoid this, the hashmap can be resized and the elements can be rehashed to new buckets, which decreases the load factor and reduces the number of collisions. This process is known as rehashing.

Rehashing can be costly in terms of time and space, but it is necessary to maintain the efficiency of the hashmap.

How Rehashing is done?
Rehashing can be done as follows:

- For each addition of a new entry to the map, check the load factor.
- If it's greater than its pre-defined value (or default value of 0.75 if not given), then Rehash.
- For Rehash, make a new array of double the previous size and make it the new bucketarray.
- Then traverse to each element in the old bucketArray and call the insert() for each so as to insert it into the new larger bucket array.

Extendible Hashing

Extendible Hashing is a dynamic hashing method wherein directories, and buckets are used to hash data. It is an aggressively flexible method in which the hash function also experiences dynamic changes.

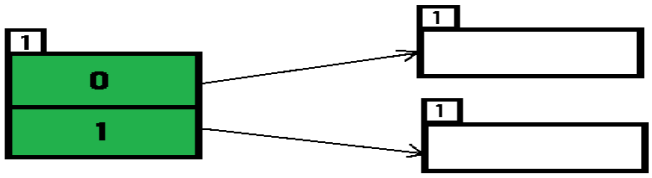
Frequently used terms in Extendible Hashing:

- **Directories:** These containers store pointers to buckets. Each directory is given a unique id which may change each time when expansion takes place. The hash function returns this directory id which is used to navigate to the appropriate bucket. Number of Directories = 2^Global Depth.
- **Buckets:** They store the hashed keys. Directories point to buckets. A bucket may contain more than one pointers to it if its local depth is less than the global depth.
- **Global Depth:** It is associated with the Directories. They denote the number of bits which are used by the hash function to categorize the keys. Global Depth = Number of bits in directory id.
- **Local Depth:** It is the same as that of Global Depth except for the fact that Local Depth is associated with the buckets and not the directories. Local depth in accordance with the global depth is used to decide the action that to be performed in case an overflow occurs. Local Depth is always less than or equal to the Global Depth.
- **Bucket Splitting:** When the number of elements in a bucket exceeds a particular size, then the bucket is split into two parts.
- **Directory Expansion:** Directory Expansion Takes place when a bucket overflows. Directory Expansion is performed when the local depth of the overflowing bucket is equal to the global depth.

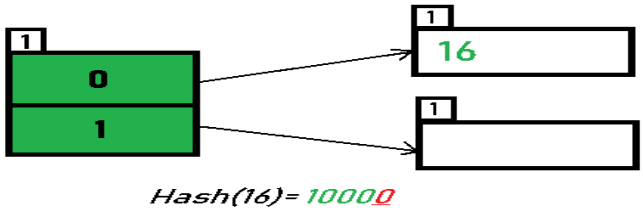
- **Step 1 - Analyze Data Elements:** Data elements may exist in various forms eg. Integer, String, Float, etc.. Currently, let us consider data elements of type integer. eg: 49.
- **Step 2 - Convert into binary format:** Convert the data element in Binary form. For string elements, consider the ASCII equivalent integer of the starting character and then convert the integer into binary form. Since we have 49 as our data element, its binary form is 110001.
- **Step 3 - Check Global Depth of the directory.** Suppose the global depth of the Hash-directory is 3.
- **Step 4 - Identify the Directory:** Consider the 'Global-Depth' number of LSBs in the binary number and match it to the directory id.
  Eg. The binary obtained is: 110001 and the global-depth is 3. So, the hash function will return 3 LSBs of 110**001** viz. 001.
- **Step 5 - Navigation:** Now, navigate to the bucket pointed by the directory with directory-id 001.
- **Step 6 - Insertion and Overflow Check:** Insert the element and check if the bucket overflows. If an overflow is encountered, go to **step 7** followed by **Step 8**, otherwise, go to **step 9**.
- **Step 7 - Tackling Over Flow Condition during Data Insertion:** Many times, while inserting data in the buckets, it might happen that the Bucket overflows. In such cases, we need to follow an appropriate procedure to avoid mishandling of data.
  First, Check if the local depth is less than or equal to the global depth. Then choose one of the cases below.
    - **Case1:** If the local depth of the overflowing Bucket is equal to the global depth, then Directory Expansion, as well as Bucket Split, needs to be performed. Then increment the global depth and the local depth value by 1. And, assign appropriate pointers.
      Directory expansion will double the number of directories present in the hash structure.
    - **Case2:** In case the local depth is less than the global depth, then only Bucket Split takes place. Then increment only the local depth value by 1. And, assign appropriate pointers.
- **Step 8 - Rehashing of Split Bucket Elements:** The Elements present in the overflowing bucket that is split are rehashed w.r.t the new global depth of the directory.
- **Step 9 -** The element is successfully hashed.

Example based on Extendible Hashing: Now, let us consider a prominent example of hashing the following elements: **16,4,6,22,24,10,31,7,9,20,26.**
Bucket Size: 3 (Assume)
Hash Function: Suppose the global depth is X. Then the Hash Function returns X LSBs.

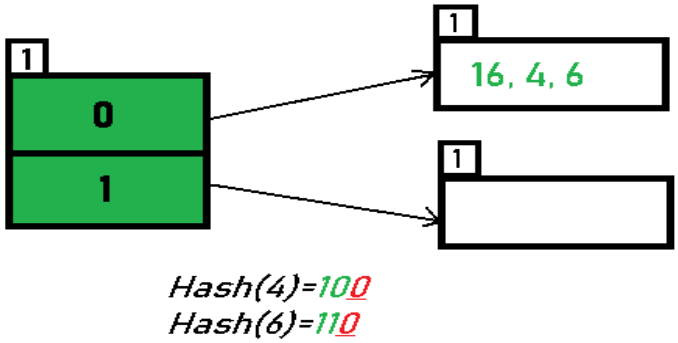
- **Solution:** First, calculate the binary forms of each of the given numbers.
  16- 10000 4- 00100 6- 00110 22- 10110 24- 11000 10- 01010 31- 11111
  7- 00111 9- 01001 20- 10100 26- 11010
- Initially, the global-depth and local-depth is always 1. Thus, the hashing frame looks like this:



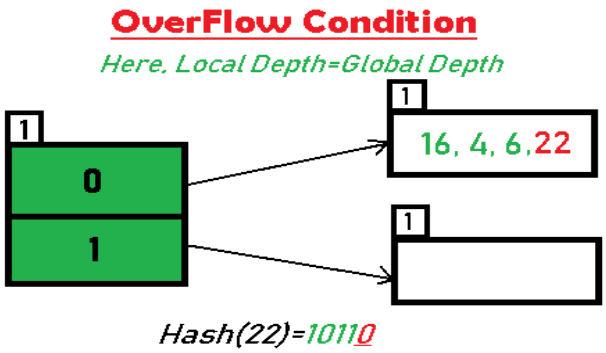
- **Inserting 16:**
  The binary format of 16 is 10000 and global-depth is 1. The hash function returns 1 LSB of 1000**0** which is 0. Hence, 16 is mapped to the directory with id=0.



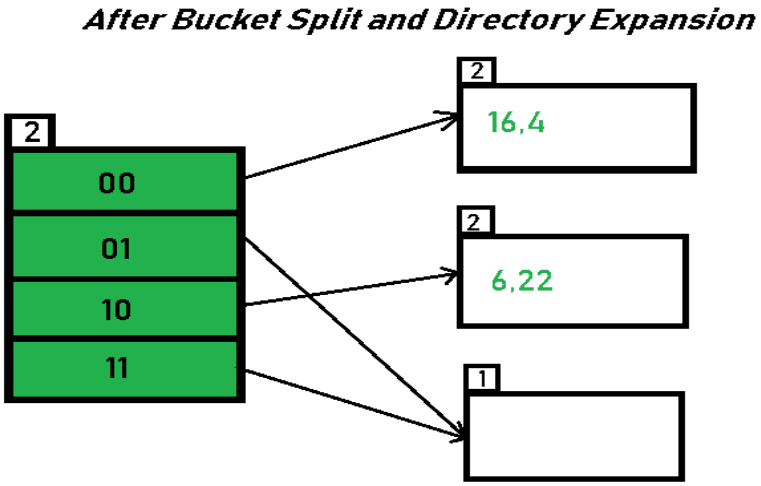
- **Inserting 4 and 6:**
  Both 4(10**0**) and 6(11**0**)have 0 in their LSB. Hence, they are hashed as follows:



- **Inserting 22:** The binary form of 22 is 1011**0**. Its LSB is 0. The bucket pointed by directory 0 is already full. Hence, Over Flow occurs.

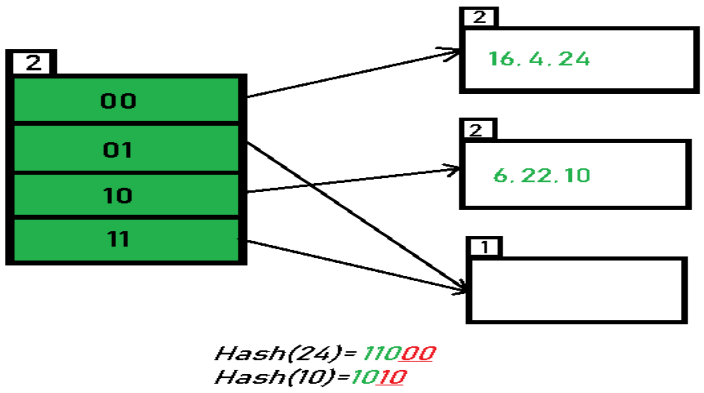


- As directed by **Step 7-Case 1**, Since Local Depth = Global Depth, the bucket splits and directory expansion takes place. Also, rehashing of numbers present in the overflowing bucket takes place after the split. And, since the global depth is incremented by 1, now,the global depth is 2. Hence, 16,4,6,22 are now rehashed w.r.t 2 LSBs.[16(100**00**),4(1**00**),6(1**10**),22(101**10**)]

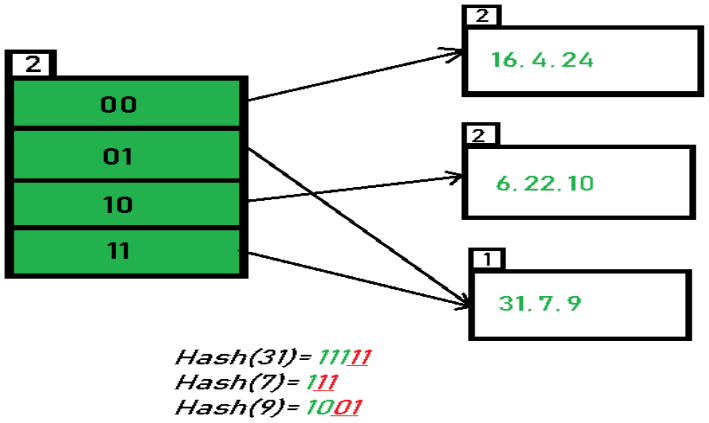


**Notice that the bucket which was underflow has remained untouched. But, since the number of directories has doubled, we now have 2 directories 01 and 11 pointing to the same bucket. This is because the local-depth of the bucket has remained 1. And, any bucket having a local depth less than the global depth is pointed-to by more than one directories.*

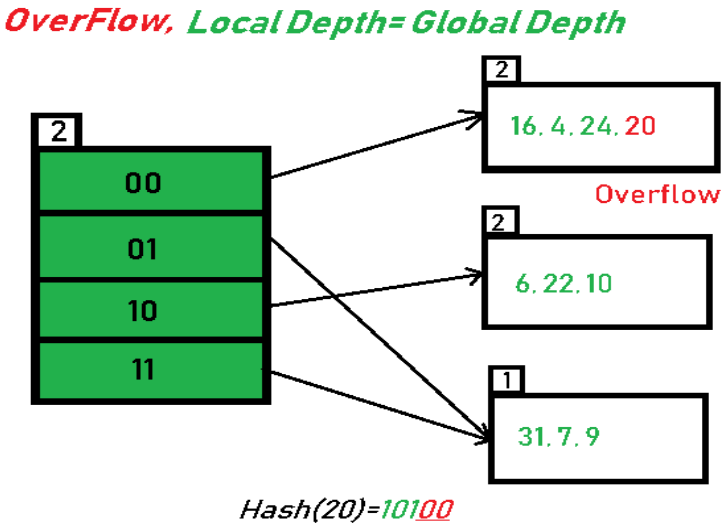
- **Inserting 24 and 10:** 24(11**00**) and 10 (10**10**) can be hashed based on directories with id 00 and 10. Here, we encounter no overflow condition.



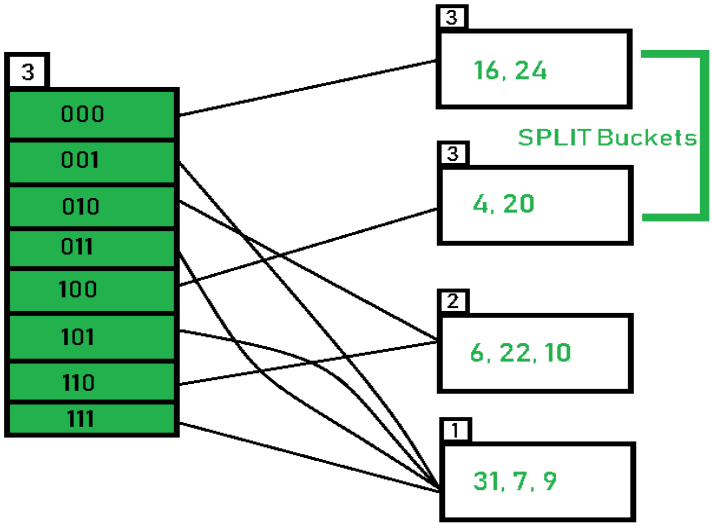
- **Inserting 31,7,9:** All of these elements[31(111**11**), 7(1**11**), 9(10**01**)] have either 01 or 11 in their LSBs. Hence, they are mapped on the bucket pointed out by 01 and 11. We do not encounter any overflow condition here.



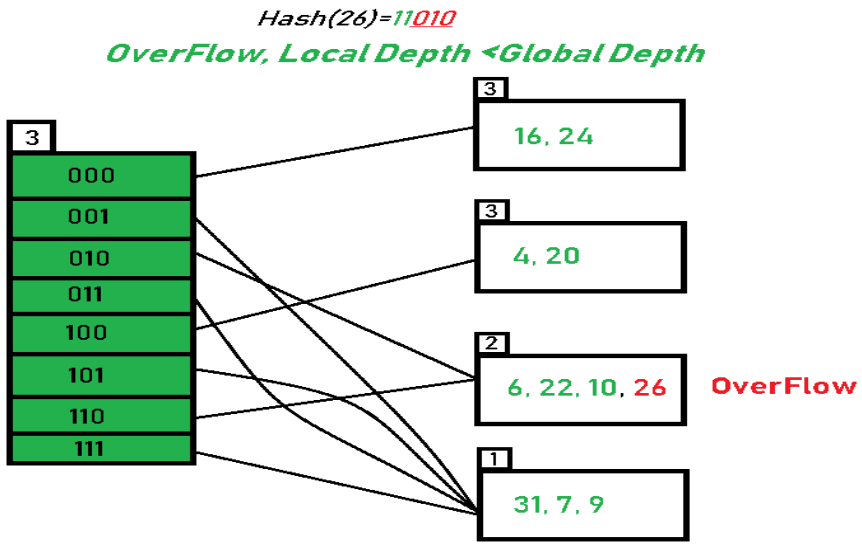
- **Inserting 20:** Insertion of data element 20 (101**00**) will again cause the overflow problem.



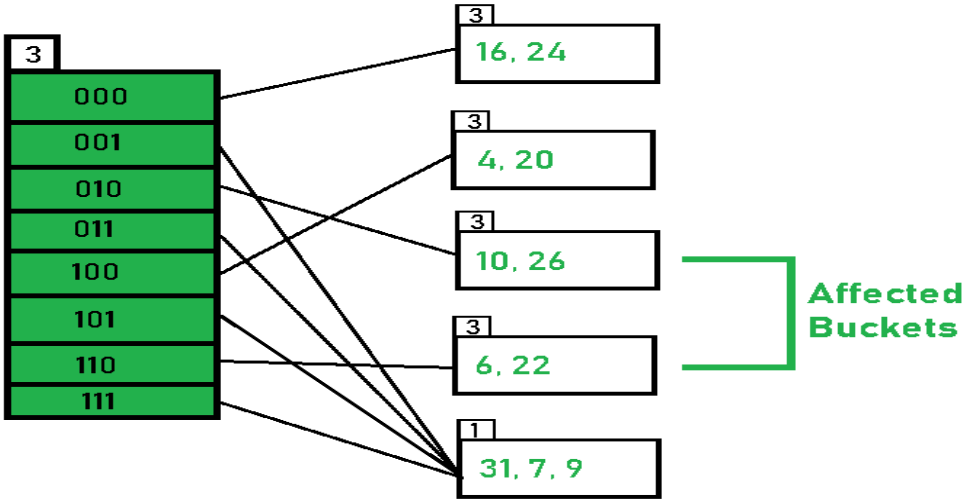
- 20 is inserted in bucket pointed out by 00. As directed by **Step 7-Case 1**, since the **local depth of the bucket = global-depth**, directory expansion (doubling) takes place along with bucket splitting. Elements present in overflowing bucket are rehashed with the new global depth. Now, the new Hash table looks like this:



- **Inserting 26:** Global depth is 3. Hence, 3 LSBs of 26(11**010**) are considered. Therefore 26 best fits in the bucket pointed out by directory 010.



- The bucket overflows, and, as directed by **Step 7-Case 2,** since the **local depth of bucket < Global depth (2<3)**, directories are not doubled but, only the bucket is split and elements are rehashed. Finally, the output of hashing the given list of numbers is obtained.



- **Hashing of 11 Numbers is Thus Completed.**

Key Observations:

1. A Bucket will have more than one pointers pointing to it if its local depth is less than the global depth.
2. When overflow condition occurs in a bucket, all the entries in the bucket are rehashed with a new local depth.
3. If Local Depth of the overflowing bucket
4. The size of a bucket cannot be changed after the data insertion process begins.

Advantages:

1. Data retrieval is less expensive (in terms of computing).
2. No problem of Data-loss since the storage capacity increases dynamically.
3. With dynamic changes in hashing function, associated old values are rehashed w.r.t the new hash function.

Limitations Of Extendible Hashing:

1. The directory size may increase significantly if several records are hashed on the same directory while keeping the record distribution non-uniform.
2. Size of every bucket is fixed.
3. Memory is wasted in pointers when the global depth and local depth difference becomes drastic.
4. This method is complicated to code.