Algorithms, performance analysis- time complexity and space complexity,
Searching: Linear and binary search methods.
Sorting: Bubble sort, selection sort, Insertion sort, Quick sort, Merge sort, Heap sort. Time complexities.

ALGORITHMS

Definition: An Algorithm is a method of representing the step-by-step procedure for solving a problem. It is a method of finding the right answer to a problem or to a different problem by breaking the problem into simple cases.

It must possess the following properties:

1. **Finiteness:** An algorithm should terminate in a finite number of steps.
2. **Definiteness:** Each step of the algorithm must be precisely (clearly) stated.
3. **Effectiveness:** Each step must be effective. i.e.; it should be easily convertible into program statement and can be performed exactly in a finite amount of time.
4. **Generality:** Algorithm should be complete in itself, so that it can be used to solve all problems of given type for any input data.
5. **Input/output:** Each algorithm must take zero, one or more quantities as input data and gives one of more output values.

An algorithm can be written in English like sentences or in any standard representations. The algorithm written in English language is called Pseudo code.

Example: To find the average of 3 numbers, the algorithm is as shown below.

Step1: Read the numbers a, b, c, and d.
Step2: Compute the sum of a, b, and c.
Step3: Divide the sum by 3.
Step4: Store the result in variable of d.
Step5: End the program.

Development Of An Algorithm

The steps involved in the development of an algorithm are as follows:

- Specifying the problem statement.
- Designing an algorithm.
- Coding.
- Debugging
- Testing and Validating
- Documentation and Maintenance.

Specifying the problem statement: The problem which has to be implemented in to a program must be thoroughly understood before the program is written. Problem must be analyzed to determine the input and output requirements of the program.

Designing an Algorithm: Once the problem is cleared then a solution method for solving the problem has to be analyzed. There may be several methods available for obtaining the required solution. The best suitable method is designing an Algorithm. To improve the **clarity** and **understandability** of the program flowcharts are drawn using algorithms.

Coding: The actual program is written in the required programming language with the help of information depicted in flowcharts and algorithms.

Debugging: There is a possibility of occurrence of errors in program. These errors must be removed for proper working of programs. The process of checking the errors in the program is known as _Debugging'.

There are three types of errors in the program.

Syntactic Errors: They occur due to wrong usage of syntax for the statements.

Ex: x=a*%b

Here two operators are used in between two operands. Runtime Errors : They are determined at the execution time of the program

Ex: Divide by zero

Range out of bounds.

Logical Errors : They occur due to incorrect usage of instructions in the program. They are neither displayed during compilation or execution nor cause any obstruction to the program execution. They only cause incorrect outputs.

Testing and Validating: Once the program is written , it must be tested and then validated. i.e., to check whether the program is producing correct results or not for different values of input.

Documentation and Maintenance: Documentation is the process of collecting, organizing and maintaining, in written the complete information of the program for future references. Maintenance is the process of upgrading the program, according to the changing requirements.

PERFORMANCE ANALYSIS

When several algorithms can be designed for the solution of a problem, there arises the need to determine which among them is the best. The efficiency of a program or an algorithm is measured by computing its time and/or space complexities.

- The **time complexity** of an algorithm is a function of the running time of the algorithm.
- The **space complexity** is a function of the space required by it to run to completion.
- The time complexity is therefore given in terms of **frequency count**.
- Frequency count is basically a count denoting number of times a statement execution

Asymptotic Notations:

- To choose the best algorithm, we need to check efficiency of each algorithm. The efficiency can be measured by computing time complexity of each algorithm. Asymptotic notation is a shorthand way to represent the time complexity.
- Using asymptotic notations we can give time complexity as –fastest possible‖, –slowest possible‖ or –average time‖.
- Various notations such as Ω, θ, O used are called asymptotic notions.

Big Oh Notation

Big Oh notation denoted by ‗O' is a method of representing the upper bound of algorithm's running time. Using big oh notation we can give longest amount of time taken by the algorithm to complete.

Definition:

Let, f(n) and g(n) are two non-negative functions. And if there exists an integer n_0 and constant C such that C > 0 and for all integers $n > n_0$, $f(n) \leq c*g(n)$, then

f(n) = Og(n).

Various meanings associated with big-oh are

O(1)	constant computing time
O(n)	linear
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(2^n)$	exponential
O(logn)	logarithmic

The relationship among these computing time is
$O(1) < O(\log n) < O(n) < O(n\log n) < O(n^2) < O(2^n)$

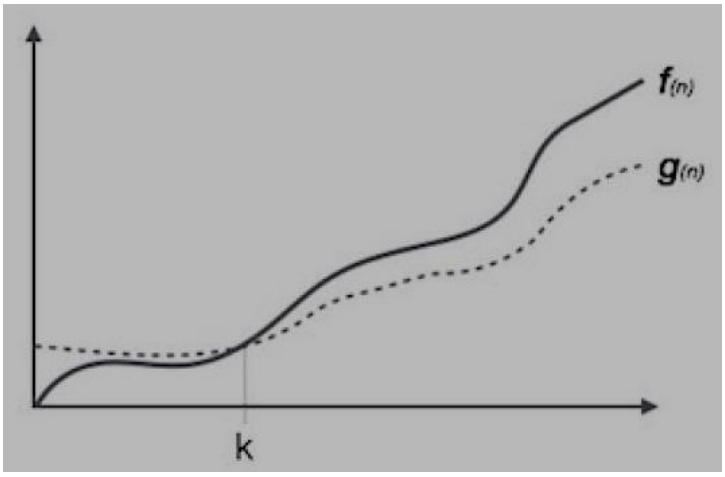
Omega Notation:-

Omega notation denoted ‗Ω' is a method of representing the lower bound of algorithm's running time. Using omega notation we can denote shortest amount of time taken by algorithm to complete.

Definition:

Let, f(n) and g(n) are two non-negative functions. And if there exists an integer n_0 and constant C such that C > 0 and for all integers $n > n_0$, $f(n) > c*g(n)$, then

$f(n) = \Omega\, g(n)$.

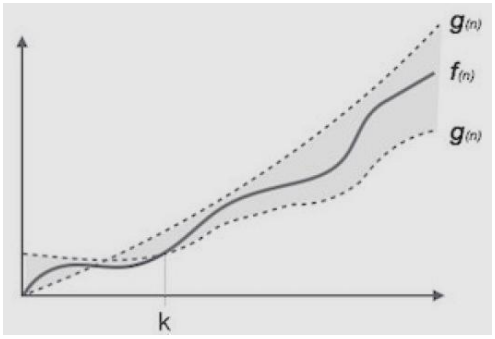


Theta Notation:-

Theta notation denoted as _θ‘ is a method of representing running time between upper bound and lower bound.

Definition:

Let, f(n) and g(n) are two non-negative functions. There exists positive constants C_1 and C_2 such that $C_1\ g(n) \le f(n) \le C_2\ g(n)$ and $f(n) = \theta\ g(n)$



How to compute time complexity

1	Algorithm Message(n)	0
2	{	0
3	for i=1 to n do	n+1
4	{	0
5	write(–Hello‖);	n
6	}	0
7	}	0
	total frequency count	2n+1

While computing the time complexity we will neglect all the constants, hence ignoring 2 and 1 we will get n. Hence the time complexity becomes O(n).

f(n) = Og(n).
i.e f(n)=O(2n+1)
=O(n) // ignore constants

1	Algorithm add(A,B,m,n)	0
2	{	0
3	for i=1 to m do	m+1
4	for j=1 to n do	m(n+1)
5	C[i,j] = A[i,j]+B[i,j]	mn

6	}	0
	total frequency count	2mn+2m+1

f(n) = Og(n).

=> O(2mn+2m+1)// when m=n;

= $O(2n^2+2n+1)$; By neglecting the constants,

we get the time complexity as $O(n^2)$.

The maximum degree of the polynomial has to be considered.

Best Case, Worst Case and Average Case Analysis

- If an algorithm takes minimum amount of time to run to completion for a specific set of input then it is called **best case** complexity.
- If an algorithm takes maximum amount of time to run to completion for a specific set of input then it is called **worst case time** complexity.
- The time complexity that we get for certain set of inputs is as a average same. Then for corresponding input such a time complexity is called average case time complexity.

Space Complexity: The space complexity can be defined as amount of memory required by an algorithm to run.

Let p be an algorithm,To compute the space complexity we use two factors: constant and instance characteristics. The space requirement S(p) can be given as

S(p) = C + Sp

where C is a constant i.e.. fixed part and it denotes the space of inputs and outputs. This space is an amount of space taken by instruction, variables and identifiers.

- Sp is a space dependent upon instance characteristics. This is a variable part whose space
  requirement depend on particular problem instance.

Eg:1

```
Algorithm add(a,b,c)
{return a+b+c;
}
```

If we assume a, b, c occupy one word size then total size comes to be 3

S(p) = C

Eg:2

```
Algorithm add(x,n)
{
        sum=0.0;
        for i= 1 to n do
        sum:=sum+x[i];
        return sum;
}
```

S(p) ≥ (n+3)

The n space required for x[], one space for n, one for i, and one for sum

Searching: Searching is the technique of finding desired data items that has been stored

within some data structure. Data structures can include linked lists, arrays, search trees, hash tables, or various other storage methods. The appropriate search algorithm often depends on the data structure being searched.

Search algorithms can be classified based on their mechanism of searching. They are

- Linear searching
- Binary searching

Linear or Sequential searching: Linear Search is the most natural searching method and It is very simple but very poor in performance at times .In this method, the searching begins with searching every element of the list till the required record is found. The elements in the list may be in any order. i.e. sorted or unsorted.

We begin search by comparing the first element of the list with the target element. If it matches, the search ends and position of the element is returned. Otherwise, we will move to next element and compare. In this way, the target element is compared with all the elements until a match occurs. If the match do not occur and there are no more elements to be compared, we conclude that target element is absent in the list by returning position as -1.

For example consider the following list of elements.

55 95 75 85 11 25 65 45

Suppose we want to search for element 11(i.e. Target element = 11). We first compare the target element with first element in list i.e. 55. Since both are not matching we move on the next elements in the list and compare. Finally we will find the match after 5 comparisons at position 4 starting from position 0.

Linear search can be implemented in two ways.i)Non recursive ii)recursive

Algorithm for Linear search

```
Linear_Search (A[ ], N, val , pos )
Step 1 : Set pos = -1 and k = 0
Step 2 : Repeat while k < N
          Begin
Step 3 : if A[ k ] = val
                Set pos = k
                print pos
                Goto step 5
          End while
Step 4 : print –Value is not present‖
Step 5 : Exit
```

BINARY SEARCHING

Binary search is a fast search algorithm with run-time complexity of O(log n). This search algorithm works on the principle of divide and conquer. Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the subarray reduces to zero.

Before applying binary searching, the list of items should be sorted in ascending or descending order.

Best case time complexity is O(1)

Worst case time complexity is O(log n)

Algorithm:

```
Binary_Search (A [ ], U_bound, VAL)
Step 1 : set BEG = 0 , END = U_bound , POS = -1
Step 2 : Repeat while (BEG <= END )
Step 3 :     set MID = ( BEG + END ) / 2
Step 4 :     if A [ MID ] == VAL then
                POS = MID
                print VAL — is available at —, POS
                GoTo Step 6
             End if
             if A [ MID ] > VAL then
               set END = MID – 1
             Else
               set BEG = MID + 1
             End if
          End while
Step 5 : if POS = -1 then
             print VAL — is not present —
          End if
Step 6 : EXIT
```

SORTING

Arranging the elements in a list either in ascending or descending order. various sorting algorithms are

- Bubble sort
- selection sort
- Insertion sort
- Quick sort
- Merge sort
- Heap sort

BUBBLE SORT

The bubble sort is an example of exchange sort. In this method, repetitive comparison is performed among elements and essential swapping of elements is done. Bubble sort is commonly used in sorting algorithms. It is easy to understand but time consuming i.e. takes more number of comparisons to sort a list . In this type, two successive elements are compared and swapping is done. Thus, step-by-step entire array elements are checked. It is different from the selection sort. Instead of searching the minimum element and then applying swapping, two records are swapped instantly upon noticing that they are not in order.

ALGORITHM:

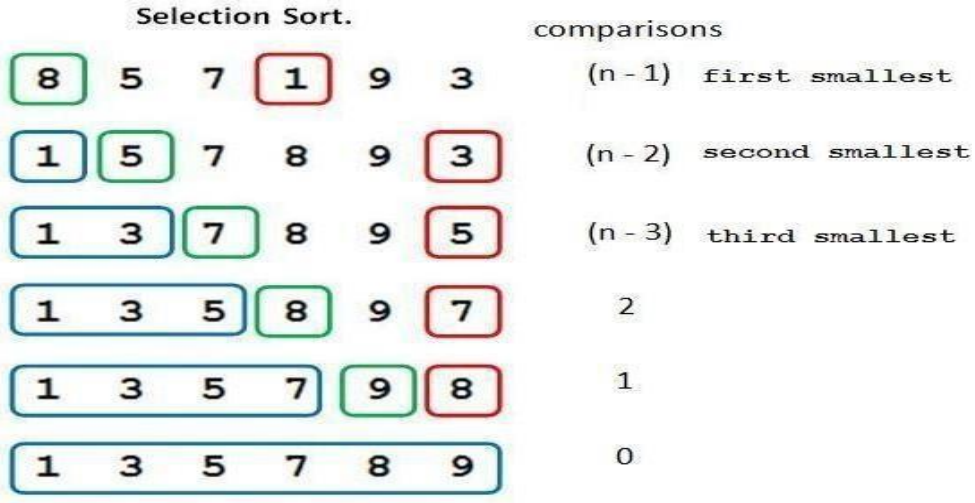
Bubble_Sort (A [ ] , N)

Step 1: Start
Step 2: Take an array of n elements
Step 3: for i=0,….....n-2
Step 4: for j=i+1,…….n-1
Step 5: if arr[j]>arr[j+1] then
Interchange arr[j] and arr[j+1]
End of if
Step 6: Print the sorted array arr
Step 7:Stop

SELECTION SORT

selection sort:- Selection sort (Select the smallest and Exchange):

The first item is compared with the remaining n-1 items, and whichever of all is lowest, is put in the first position.Then the second item from the list is taken and compared with the remaining (n-2) items, if an item with a value less than that of the second item is found on the (n-2) items, it is swapped (Interchanged) with the second item of the list and so on.



INSERTION SORT

Insertion sort: It iterates, consuming one input element each repetition, and growing a sorted output list. Each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.

ALGORITHM:
Step 1: start
Step 2: for i ← 1 to length(A)
Step 3: j ← i
Step 4: while j > 0 and A[j-1] > A[j]
Step 5: swap A[j] and A[j-1]
Step 6: j ← j - 1
Step 7: end while
Step 8: end for
Step9: stop

QUICK SORT

Quick sort: It is a divide and conquer algorithm. Developed by Tony Hoare in 1959. Quick sort

first divides a large array into two smaller sub-arrays: the low elements and the high elements.
Quick sort can then recursively sort the sub-arrays.
ALGORITHM:

Step 1: Pick an element, called a pivot, from the array.

Step 2: Partitioning: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.

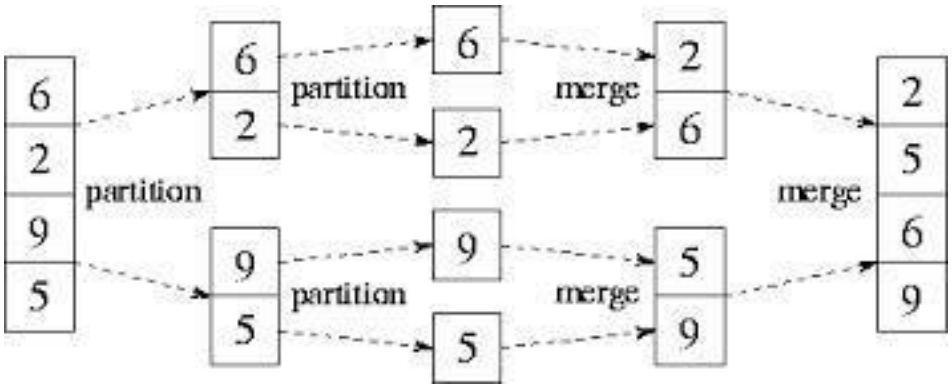
Step 3: Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

MERGE SORT

Merge sort is a sorting technique based on divide and conquer technique. In merge sort the unsorted list is divided into N sublists, each having one element, because a list of one element is considered sorted. Then, it repeatedly merge these sublists, to produce new sorted sublists, and at lasts one sorted list is produced. Merge Sort is quite fast, and has a time complexity of O(n log n).

C**onceptually, merge sort works as follows:**

1. Divide the unsorted list into two sub lists of about half the size.
2. Divide each of the two sub lists recursively until we have list sizes of length 1, in which case the list itself is returned.
3. Merge the two sub lists back into one sorted list.




```
int main()
{
int n,i;
int list[30];
        cout<<"enter no of elements\n";
        cin>>n;
        cout<<"enter "<<n<<" numbers ";
        for(i=0;i<n;i++)
        cin>>list[i];
        mergesort (list,0,n-1);
        cout<<" after sorting\n";
        for(i=0;i<n;i++)
        cout<<list[i]<<‖\t‖;
return 0;
}
```

RUN 1:

```
enter no of elements 5
enter 5 numbers 44 33 55 11 -1
  after sorting -1 11 33 44 55
```

HEAP SORT

It is a completely binary tree with the property that a parent is always greater than or equal to either of its children (if they exist). first the heap (max or min) is created using binary tree and then heap is sorted using priority queue.

Steps Followed:

a) Start with just one element. One element will always satisfy heap property.
b) Insert next elements and make this heap.
c) Repeat step b, until all elements are included in the heap.

a) Exchange the root and last element in the heap.
b) Make this heap again, but this time do not include the last node.
c) Repeat steps a and b until there is no element left.

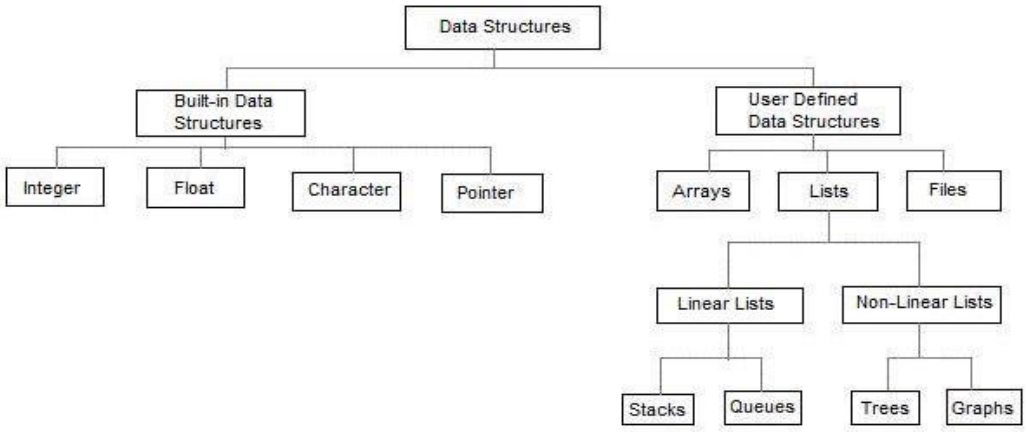
Algorithm	Worst case	Average case	Best case
Bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Quick sort	O(n log n)	O(n log n)	$O(n^2)$
Merge sort	O(n log n)	O(n log n)	O(n log n)
Heap sort	O(n log n)	O(n log n)	O(n log n)
Linear search	O(n)	O(n)	O(1)
Binary search	O(log n)	O(log n)	O(1)

Basic data structures- The list ADT, Stack ADT, Queue ADT,array and linked list Implementation using template classes in C++.**Trees**-Basic terminology Binary Tree ADT, array and linked list Implementation, Binary tree traversals, threaded binary tree.

Data structure A data structure is a specialized format for organizing and storing data. General data structure types include the array, the file, the record, the table, the tree, and so on. Any data structure is designed to organize data to suit a specific purpose so that it can be accessed and worked with in appropriate ways

Abstract Data Type

In computer science, an abstract data type (ADT) is a mathematical model for data types where a data type is defined by its behavior (semantics) from the point of view of a user of the data, specifically in terms of possible values, possible operations on data of this type, and the behavior of these operations. When a class is used as a type, it is an abstract type that refers to a hidden representation. In this model an ADT is typically implemented as a class, and each instance of the ADT is usually a n object of that class. In ADT all the implementation details are hidden



- Linear data structures are the data structures in which data is arranged in a list or in a sequence.
- Non linear data structures are the data structures in which data may be arranged in a hierarchic al manner

LIST ADT

List is basically the collection of elements arrange d in a sequential manner. In memory we can store the list in two ways: one way is we can store the elements in sequential memory locations. That means we can store the list in arrays.
The other way is we can use pointers or links to associate elements sequentially. This is known as linked list.

LINKED LISTS

The linked list is very different type of collection from an array. Using such lists, we can store collections of information limited only by the total amount of memory that the OS will allow us to use.Further more, there is no need to specify our needs in advance. The linked list is very flexible dynamic data structure : items may be added to it or deleted from it at will. A programmer need not worry about how many items a program will have to accommodate in advance. This allows us to write robust programs which require much less maintenance.

The linked allocation has the following draw backs:

1. No direct access to a particular element.
2. Additional memory required for pointers.

Linked list are of 3 types:

1. Singly Linked List
2. Doubly Linked List
3. Circularly Linked List

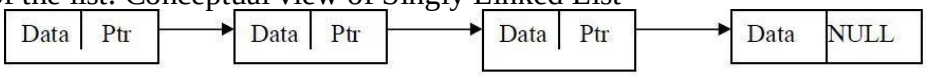
SINGLY LINKED LIST

A singly linked list, or simply a linked list, is a linear collection of data items. The linear order is given by means of POINTERS. These types of lists are often referred to as **linear linked list**.

* Each item in the list is called a node.
* Each node of the list has two fields:

1. Information- contains the item being stored in the list.
2. Next address- contains the address of the next item in the list.

* The last node in the list contains NULL pointer to indicate that it is the end of the list. Conceptual view of Singly Linked List



Operations on Singly linked list:

- Insertion of a node
- Deletions of a node
- Traversing the list

Structure of a node:

Method -1:

```
struct node
{
        int data;
        struct node *link;
};
```

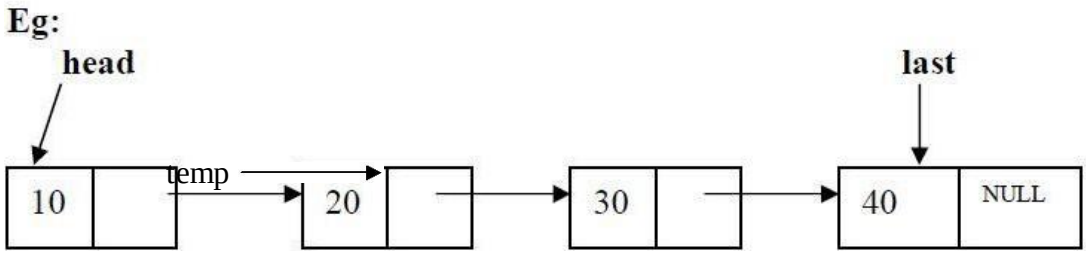
Method -2:

```
class node
{
public:
        int data;
        node *link;
};
```

Insertions: To place an elements in the list there are 3 cases :

1. At the beginning
2. End of the list
3. At a given position

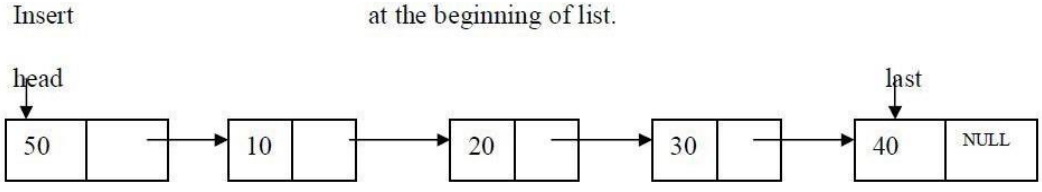
case 1:Insert at the beginning



head is the pointer variable which contains address of the first node and **temp** contains address of new node to be inserted then sample code is

```
temp->link=head;
head=temp;
```

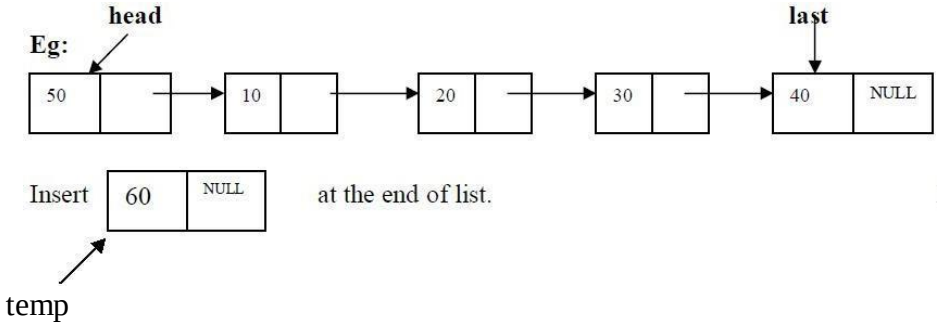
After insertion:



Code for insert front:-

```
template <class T>
void list<T>::insert_front()
{
struct node <T>*t,*temp;
        cout<<"Enter data into node:";
        cin>>item;
        temp=create_node(item);
        if(head==NULL)
                head=temp;
        else
        {temp->link=head;
                head=temp;
        }
}
```

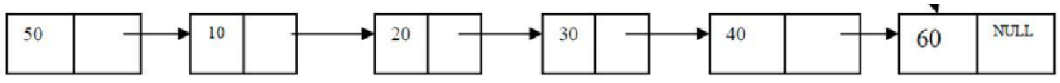
case 2: Inserting end of the list



head is the pointer variable which contains address of the first node and **temp** contains address of new node to be inserted then sample code is

```
t=head;
while(t->link!=NULL)
{
        t=t->link;
}
t->link=temp;
```

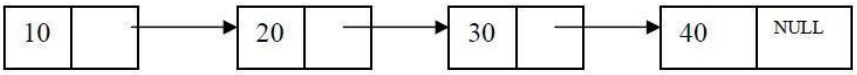
After insertion the linked list is

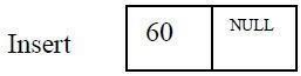


Code for insert End:-

```
template <class T>
void list<T>::insert_end()
{
struct node<T> *t,*temp;
int n;
        cout<<"Enter data into node:";
        cin>>n;
        temp=create_node(n);
        if(head==NULL)
                head=temp;
        else
        {t=head; while(t-
                >link!=NULL)
                        t=t->link;
                t->link=temp;
        }
}
```

case 3: Insert at a position

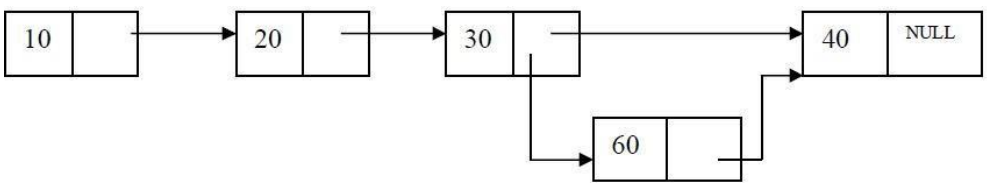




insert node at position 3

head is the pointer variable which contains address of the first node and **temp** contains address of new node to be inserted then sample code is

```
c=1;
while(c<pos)
{
        prev=cur;
        cur=cur->link;
        c++;
}
prev->link=temp;
temp->link=cur;
```



Code for inserting a node at a given position:-

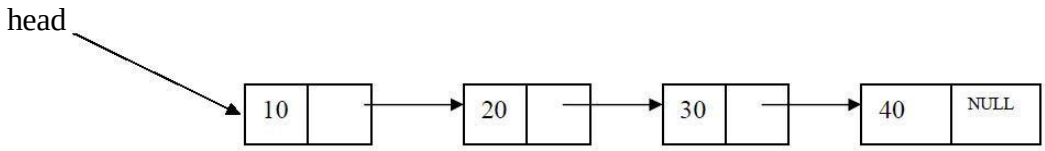
```
template <class T>
void list<T>::Insert_at_pos(int pos)
{struct node<T>*cur,*prev,*temp;
int c=1;
        cout<<"Enter data into node:";
        cin>>item
        temp=create_node(item);
        if(head==NULL)
                head=temp;
         else
         {
                 prev=cur=head;
                if(pos==1)
                {
                        temp->link=head;
                        head=temp;
                }
                else
                {
                while(c<pos)
                {c++;
                        prev=cu
                        r;
                        cur=cur->link;
                }
                prev->link=temp;
                temp->link=cur;
                }
        }
}
```

Deletions: Removing an element from the list, without destroying the integrity of the list itself.

To place an element from the list there are 3 cases :

1. Delete a node at beginning of the list
2. Delete a node at end of the list
3. Delete a node at a given position

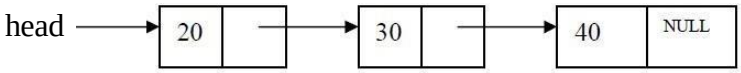
Case 1: Delete a node at beginning of the list



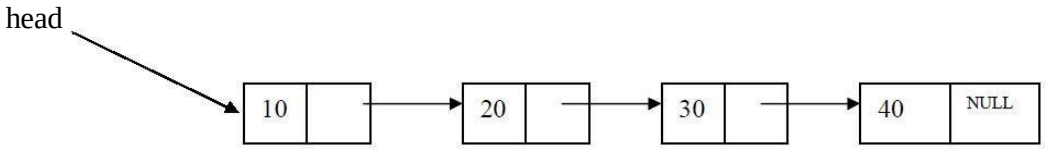
head is the pointer variable which contains address of the first node

sample code is

```
t=head;
head=head->link;
cout<<"node "<<t->data<<" Deletion is sucess";
delete(t);
```

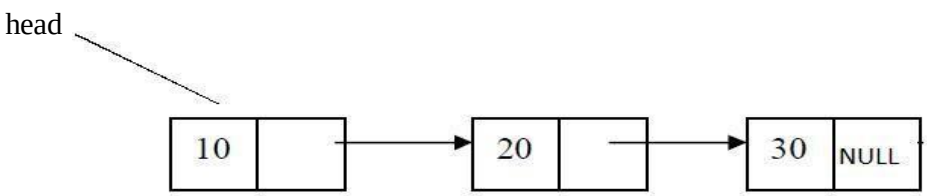


Case 2. Delete a node at end of the list



To delete last node , find the node using following code

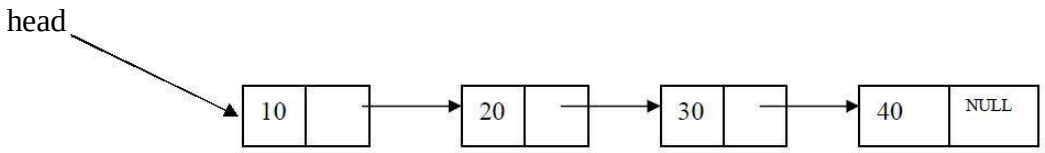
```
struct node<T>*cur,*prev;
cur=prev=head;
while(cur->link!=NULL)
{prev=cur; cur=cur-
        >link;
}
prev->link=NULL;
cout<<"node "<<cur->data<<" Deletion is sucess";
free(cur);
```



Code for deleting a node at end of the list

```
template <class T>
void list<T>::delete_end()
{
struct node<T>*cur,*prev;
        cur=prev=head;
        if(head==NULL)
                cout<<"List is Empty\n";
        else
        {cur=prev=head; if(head-
                >link==NULL)
                {
                        cout<<"node "<<cur->data<<" Deletion is
                        sucess"; free(cur);
                        head=NULL;
                }
else
                {while(cur->link!=NULL)
                        {prev=cur; cur=cur-
                                >link;
                        }
                        prev->link=NULL;
                        cout<<"node "<<cur->data<<" Deletion is sucess";
                        free(cur);
                }
        }
}
```

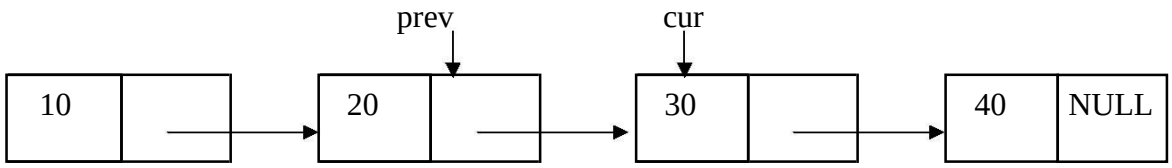
CASE 3. Delete a node at a given position



Delete node at position 3

head is the pointer variable which contains address of the first node. Node to be deleted is node
containing value 30.
Finding node at position 3

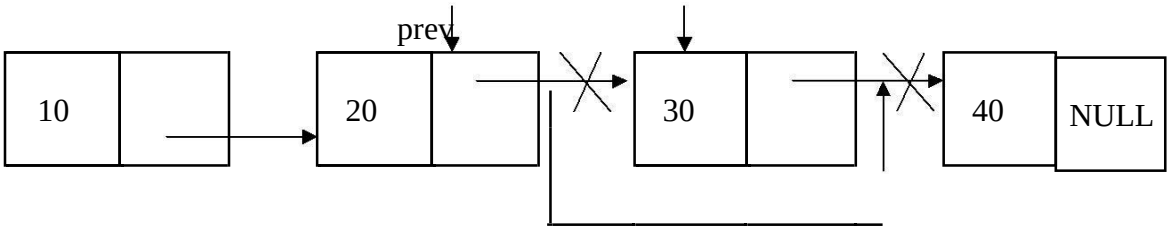
```
c=1;
while(c<pos)
{c++;
        prev=cu
        r;
        cur=cur->link;
}
```



cur is the node to be deleted . before deleting update links

code to update links

```
prev->link=cur->link;
cout<<cur->data <<"is deleted successfully";
delete cur;
```



Traversing the list: Assuming we are given the pointer to the head of the list, how do we get the end of the list.

```
template <class T>
void list<T>:: display()
{
        struct node<T>*t;

        if(head==NULL)
         {
                        cout<<"List is Empty\n";
         }
         else
         {t=head;
                while(t!=NUL
                L)
                {cout<<t->data<<"->";
                        t=t->link;
                }
         }
}
```

DOUBLY LINKED LIST

A singly linked list has the disadvantage that we can only traverse it in one direction. Many applications require searching backwards and forwards through sections of a list. A useful refinement that can be made to the singly linked list is to create a doubly linked list. The distinction made between the two list types is that while singly linked list have pointers going in one direction, doubly linked list have pointer both to the next and to the previous element in the list. The main advantage of a doubly linked list is that, they permit traversing or searching of the list in both directions.

In this linked list each node contains three fields.

a) One to store data
b) Remaining are self referential pointers which points to previous and next nodes in the list

prev	data	next

Implementation of node using structure

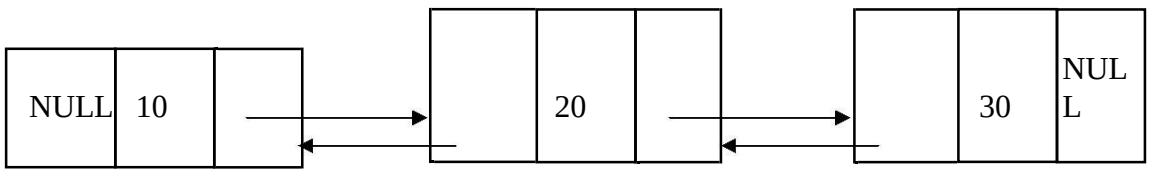
Method -1:

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

Implementation of node using class

Method -2:

```
class node
{
public:
        int data;
        node *prev;
        node * next;
};
```



Operations on Doubly linked list:

- Insertion of a node
- Deletions of a node
- Traversing the list

Doubly linked list ADT:

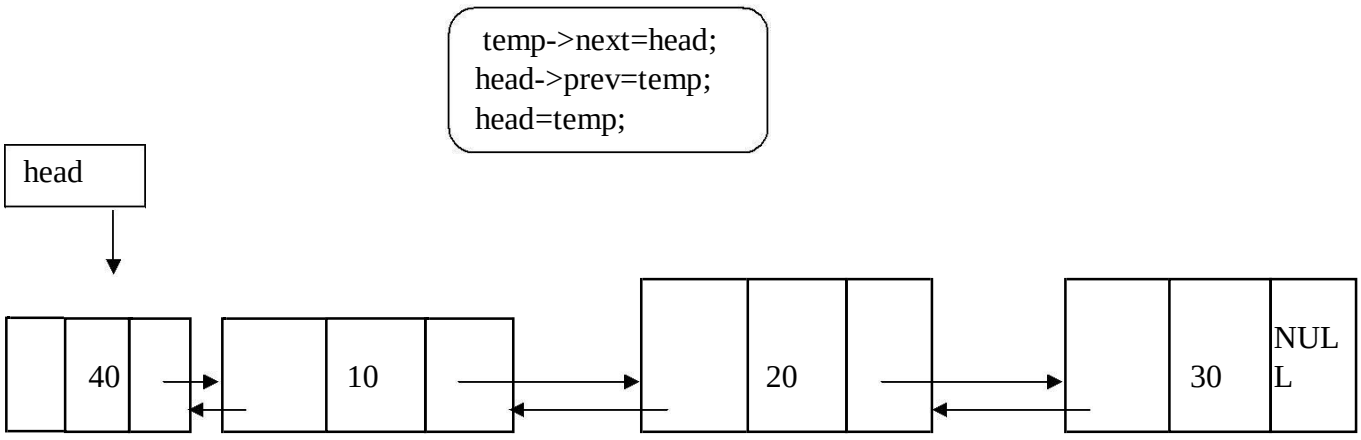
```
template <class T>
class dlist
{
        int data;
        struct dnode<T>*head;
public:
        dlist()
        {
                head=NULL;
        }
        void display();
        struct dnode<T>*create_dnode(int n);
        void insert_end();
        void insert_front();
        void delete_end();
        void delete_front();
        void dnode_count();

          void Insert_at_pos(int pos);
          void Delete_at_pos(int pos);
  };
```

Insertions: To place an elements in the list there are 3 cases

- 1. At the beginning
- 2. End of the list
- 3. At a given position

case 1:Insert at the beginning

head is the pointer variable which contains address of the first node and **temp** contains address of new node to be inserted then sample code is



Code for insert front:-

```
template <class T>
void DLL<T>::insert_front()
{
struct dnode <T>*t,*temp;
        cout<<"Enter data into node:";
        cin>>data;
        temp=create_dnode(data);
        if(head==NULL)
                head=temp;
        else
        {temp-
                >next=head
                ; head-
                >prev=temp
                ;
                head=temp;
        }
 }
```

Code to insert a node at End:-

```
template <class T>

void DLL<T>::insert_end()
{
struct dnode<T> *t,*temp;
int n;
        cout<<"Enter data into dnode:";
        cin>>n;
        temp=create_dnode(n);
        if(head==NULL)
                head=temp;
        else

        {t=head; while(t-
                >next!=NULL)
                        t=t->next;
                 t->next=temp;

                 temp->prev=t;
          }
  }
```

Code to insert a node at a position

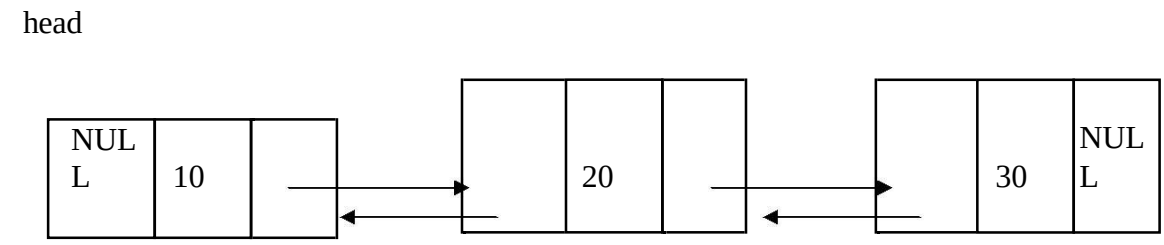
```
template <class T>
void dlist<T>::Insert_at_pos(int pos)
{
struct dnode<T>*cr,*pr,*temp;
int count=1;
        cout<<"Enter data into dnode:";
        cin>>data;
        temp=create_dnode(data);
        display();
        if(head==NULL)
        {//when list is empty
                head=temp;
        }
         else
         {pr=cr=head;
                if(pos==1)
                {//inserting at pos=1
                        temp-
                        >next=head;
                        head=temp;
                }
                else
                {
                        while(count<pos)
                        {count++;
                                pr=cr;
                                cr=cr->next;
                        }
                        pr->next=temp;
                        temp->prev=pr;
                        temp->next=cr;
                        cr->prev=temp;
                }
        }
}
```

Deletions: Removing an element from the list, without destroying the integrity of the list itself.

To place an element from the list there are 3 cases :

1. Delete a node at beginning of the list
2. Delete a node at end of the list
3. Delete a node at a given position

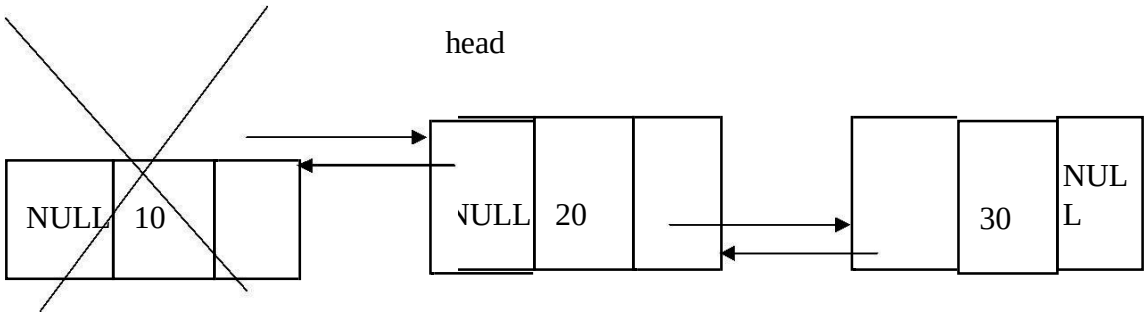
Case 1: Delete a node at beginning of the list



head is the pointer variable which contains address of the first node

sample code is

```
t=head;
head=head->next;
head->prev=NULL;
cout<<"dnode "<<t->data<<" Deletion is sucess";
delete(t);
```



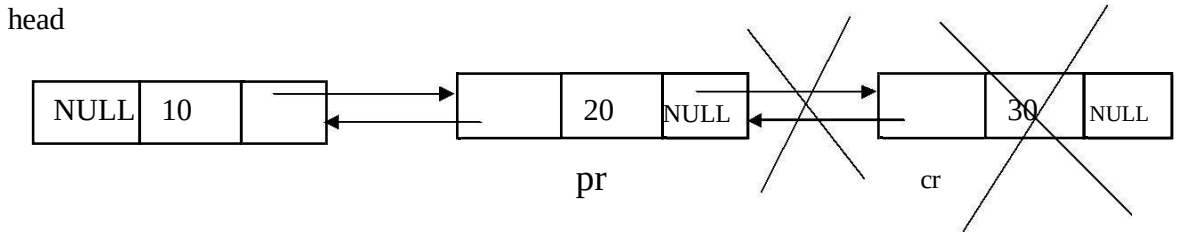
code for deleting a node at front

```
template <class T>
void dlist<T>:: delete_front()
{struct dnode<T>*t;
        if(head==NULL)
                cout<<"List is Empty\n";
        else
        {t=head;
                head=head->next;
                head->prev=NULL;
                cout<<"dnode "<<t->data<<" Deletion is sucess";
                delete(t);
        }
}
```

Case 2. Delete a node at end of the list

To deleted the last node find the last node. find the node using following code

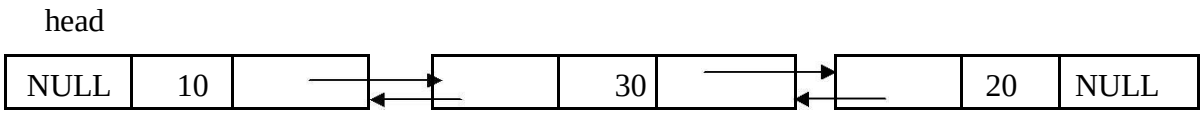
```
struct dnode<T>*pr,*cr;
        pr=cr=head;
while(cr->next!=NULL)
{pr=cr; cr=cr-
        >next;
}
pr->next=NULL;
cout<<"dnode "<<cr->data<<" Deletion is sucess";
delete(cr);
```



code for deleting a node at end of the list

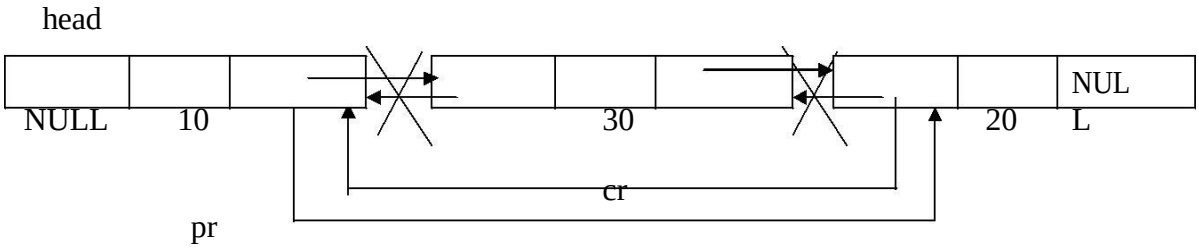
```
template <class T>
void dlist<T>::delete_end()
{
struct dnode<T>*pr,*cr;
        pr=cr=head;
        if(head==NULL)
                cout<<"List is Empty\n";
        else
        {cr=pr=head; if(head-
                >next==NULL)
                {
                        cout<<"dnode "<<cr->data<<" Deletion is
                        sucess"; delete(cr);
                        head=NULL;
                }
                else
                {while(cr->next!=NULL)
                        {pr=cr;
                                cr=cr->next;
                        }
                        pr->next=NULL;
                        cout<<"dnode "<<cr->data<<" Deletion is sucess";
                        delete(cr);
                }
        }
}
```

CASE 3. Delete a node at a given position



Delete node at position 2
head is the pointer variable which contains address of the first node. Node to be deleted is node
containing value 30.
Finding node at position 2.

```
while(count<pos)
{pr=cr; cr=cr-
        >next;
        count++;
}
pr->next=cr->next;
cr->next->prev=pr;
```

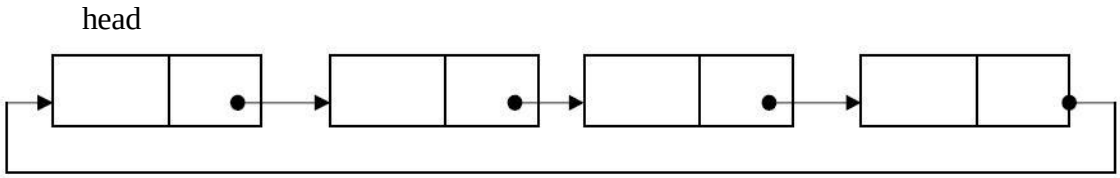


CIRCULARLY LINKED LIST

A circularly linked list, or simply circular list, is a linked list in which the last node is always points to the first node. This type of list can be build just by replacing the NULL pointer at the end of the list with a pointer which points to the first node. There is no first or last node in the circular list.

Advantages:

- Any node can be traversed starting from any other node in the list.
- There is no need of NULL pointer to signal the end of the list and hence, all pointers contain valid addresses.
- In contrast to singly linked list, deletion operation in circular list is simplified as the search for the previous node of an element to be deleted can be started from that item itself.



STACK ADT:- A Stack is a linear data structure where insertion and deletion of items takes place at one end called top of the stack. A Stack is defined as a data structure which operates on a last-in first-out basis. So it is also is referred as Last-in First-out(LIFO).

Stack uses a single index or pointer to keep track of the information in the stack. The basic operations associated with the stack are:

a) push(insert) an item onto the stack.
b) pop(remove) an item from the stack.

The general terminology associated with the stack is as follows:

A stack pointer keeps track of the current position on the stack. When an element is placed on the stack, it is said to be **pushed** on the stack. When an object is removed from the stack, it is said to be **popped** off the stack. Two additional terms almost always used with stacks are **overflow**, which occurs when we try to push more information on a stack that it can hold, and **underflow**, which occurs when we try to pop an item off a stack which is empty.

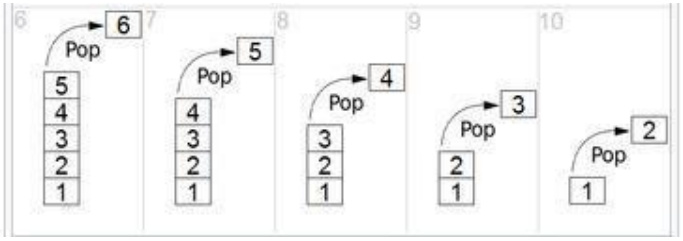
Pushing items onto the stack:

Assume that the array elements begin at 0 (because the array subscript starts from 0) and the maximum elements that can be placed in stack is max. The stack pointer, **top**, is considered to be pointing to the top element of the stack. A push operation thus involves adjusting the stack pointer to point to next free slot and then copying data into that slot of the stack. Initially the top is initialized to -1.

```
//code to push an element on to stack;
template<class T>
void stack<T>::push()
{
        if(top==max-1)
                cout<<"Stack Overflow...\n";
        else
        {
                cout<<"Enter an element to be pushed:";
                top++;
                cin>>data;
                stk[top]=data;
                cout<<"Pushed Sucesfully ...\n";
        }
}
```

Popping an element from stack:

To remove an item, first extract the data from top position in the stack and then decrement the stack pointer, top.



Applications of Stack:

1. Stacks are used in conversion of infix to postfix expression.
2. Stacks are also used in evaluation of postfix expression.
3. Stacks are used to implement recursive procedures.
4. Stacks are used in compilers.
5. Reverse String

An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression. These notations are −

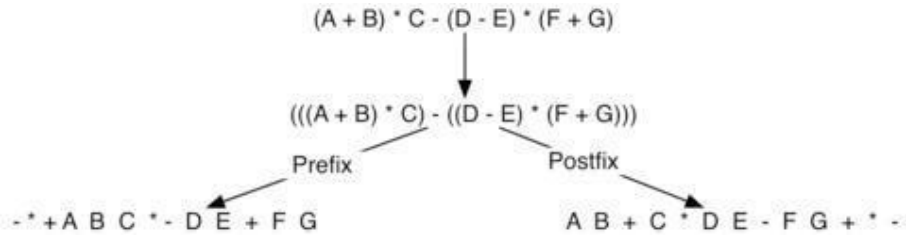
1. Infix Notation
2. Prefix (Polish) Notation
3. Postfix (Reverse-Polish) Notation

Expression	Example	Note
Infix	a + b	Operator Between Operands
Prefix	+ a b	Operator before Operands
Postfix	a b +	Operator after Operands

Conversion of Infix Expressions to Prefix and Postfix

Infix Expression	Prefix Expression	Postfix Expression
A + B * C + D	+ + A * B C D	A B C * + D +
(A + B) * (C + D)	* + A B + C D	A B + C D + *
A * B + C * D	+ * A B * C D	A B * C D * +
A + B + C + D	+ + + A B C D	A B + C + D +

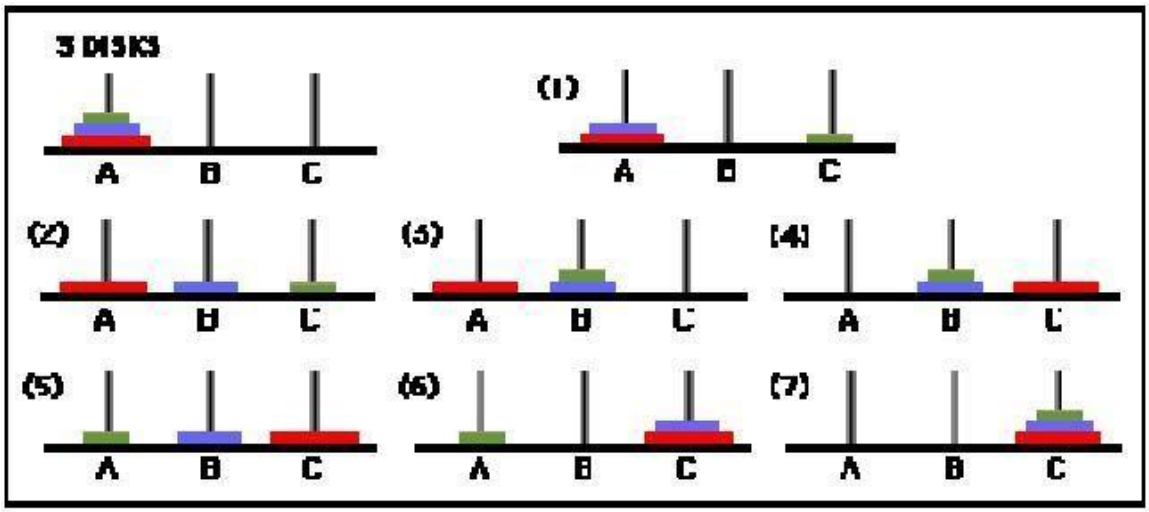
Convert following infix expression to prefix and postfix
(A + B) * C - (D - E) * (F + G)



The Tower of Hanoi (also called the Tower of Brahma or Lucas' Tower,[1] and sometimes pluralized) is a mathematical game or puzzle. It consists of three rods, and a number of disks of different sizes which can slide onto any rod. The puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top, thus making a conical shape.

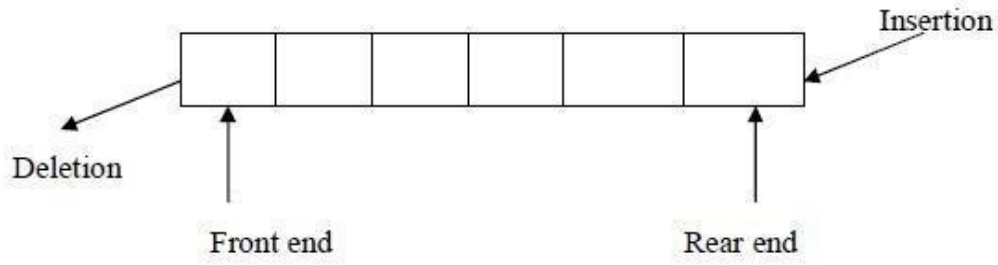
The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
3. No disk may be placed on top of a smaller disk.



QUEUE ADT

A queue is an ordered collection of data such that the data is inserted at one end and deleted from another end. The key difference when compared stacks is that in a queue the information stored is processed first-in first-out or FIFO. In other words the information receive from a queue comes in the same order that it was placed on the queue.



Representing a Queue:

One of the most common way to implement a queue is using array. An easy way to do so is to
define an array Queue, and two additional variables front and rear. The rules for manipulating these
variables are
simple:

- Each time information is added to the queue, increment rear.
- Each time information is taken from the queue, increment front.
- Whenever **front >rear or front=rear=-1** the queue is empty.

Array implementation of a Queue do have drawbacks. The maximum queue size has to be set at compile time, rather than at run time. Space can be wasted, if we do not use the full capacity of the array.

Operations on Queue:

A queue have two basic operations:

a) adding new item to the queue

b) removing items from queue.

The operation of adding new item on the queue occurs only at one end of the queue called the **rea**r or back.

The operation of removing items of the queue occurs at the other end called the **front.**

For insertion and deletion of an element from a queue, the array elements begin at 0 and the maximum elements of the array is **maxSize.** The variable front will hold the index of the item that is considered the front of the queue, while the rear variable will hold the index of the last item in the queue.

Assume that initially the front and rear variables are initialized to -1. Like stacks, underflow and overflow conditions are to be checked before operations in a queue.

Queue empty or underflow condition is

```
if((front>rear)||front= =-1)
    cout<‖Queue is empty‖;
```

Queue Full or overflow condition is

```
if((rear==max)
    cout<‖Queue is full‖;
```

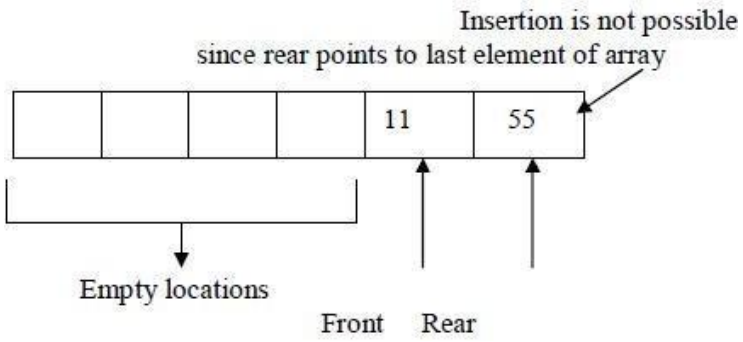
Application of Queue:

Queue, as the name suggests is used whenever we need to have any group of objects in an order in which the first one coming in, also gets out first while the others wait for there turn, like in the following scenarios :

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

CIRCULAR QUEUE

Once the queue gets filled up, no more elements can be added to it even if any element is removed from it consequently. This is because during deletion, rear pointer is not adjusted.



When the queue contains very few items and the rear pointer points to last element. i.e. rear=maxSize-1, we cannot insert any more items into queue because the overflow condition satisfies. That means a lot of space is wasted

.Frequent reshuffling of elements is time consuming. One solution to this is arranging all elements in a circular fashion. Such structures are often referred to as **Circular Queues.**

A circular queue is a queue in which all locations are treated as circular such that the first location CQ[0] follows the last location CQ[max-1].

Circular Queue empty or underflow condition is

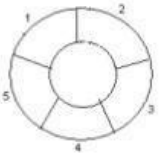
```
if(front==-1)
                cout<<"Queue is empty";
```

Circular Queue Full or overflow condition is

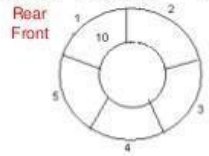
```
if(front==(rear+1)%max)
{
        cout<<"Circular Queue is full\n";
}
```

Example: Consider the following circular queue with N = 5.

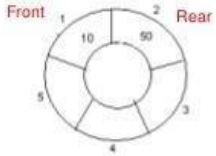
1. Initially, Rear = 0, Front = 0.



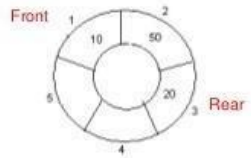
2. Insert 10, Rear = 1, Front = 1.



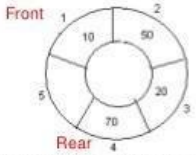
3. Insert 50, Rear = 2, Front = 1.



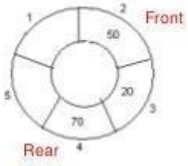
4. Insert 20, Rear = 3, Front = 0.



5. Insert 70, Rear = 4, Front = 1.



6. Delete front, Rear = 4, Front = 2.



Insertion into a Circular Queue:

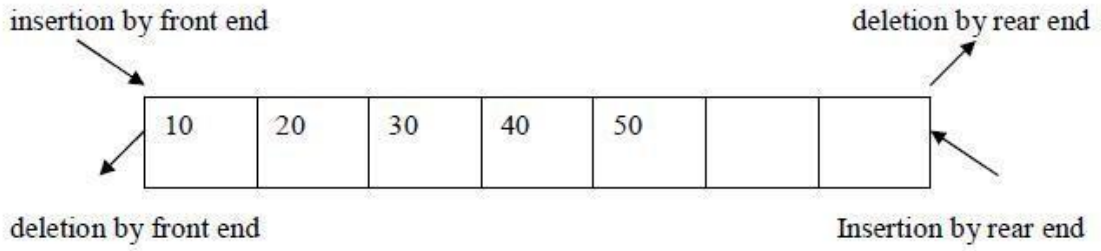
```
Algorithm CQueueInsertion(Q,maxSize,Front,Rear,item)
Step 1: If Rear = maxSize-1 then
                Rear = 0
        else
                Rear=Rear+1
Step 2: If Front = Rear then
        print –Queue Overflow‖
        Return
Step 3: Q[Rear] = item

Step 4: If Front = 0 then
        Front = 1
Step 5: Return
```

Deletion from Circular Queue:

```
Algorithm CQueueDeletion(Q,maxSize,Front,Rear,item)
Step 1: If Front = 0 then
        print –Queue Underflow‖
        Return
Step 2: K=Q[Front]
Step 3: If Front = Rear then
        begin
                Front = -1
                Rear = -1
        end
        else
        If Front = maxSize-1 then
                Front = 0
        else
                Front = Front + 1
Step 4: Return K
```

DEQUEUE

In a linear queue, the usual practice is for insertion of elements we use one end called rear for deletion of elements we use another end called as front. But in the doubly ended queue we can make use of both the ends for insertion of the elements as well as we can use both the ends for deletion of the elements. That means it is possible to insert the elements by rear as well as by front. Similarly it is possible to delete the elements from rear.

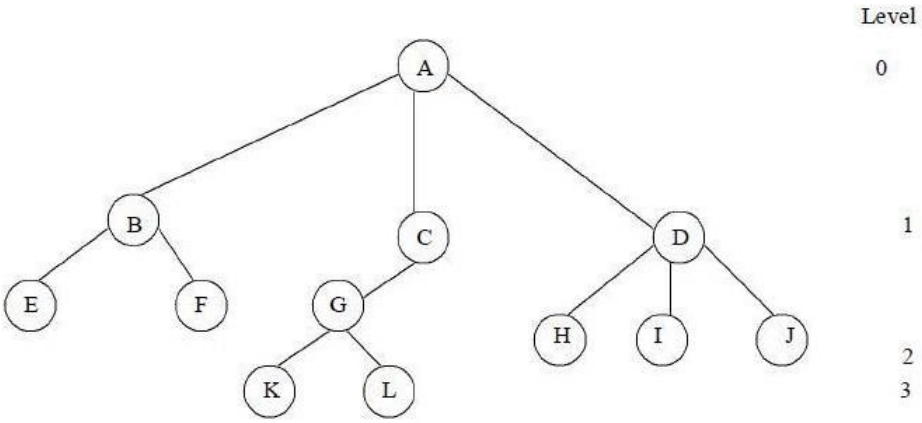


Normally insertion of elements is done at rear end and delete the elements from front end. For example elements 10,20,30 are inserted at rear end.

To insert any element from front end then first shift all the elements to the right. It s

TREES:

Definition : A Tree is a data structure in which each element is attached to one or more elements directly beneath it.



Terminology

- The connections between elements are called branches.
- A tree has a single root, called root node, which is shown at the top of the tree. i.e. root is always at the highest level 0.
- Each node has exactly one node above it, called parent. Eg: A is the parent of B,C and D.
- The nodes just below a node are called its children. ie. child nodes are one level lower than the
- parent node.
- A node which does not have any child called **leaf or terminal node**.
- Nodes with at least one child are called non terminal or internal nodes.
- The child nodes of same parent are said to be **siblings.**
- A path in a tree is a list of distinct nodes in which successive nodes are connected by branches in the tree.
- The length of a particular path is the number of branches in that path.
- The degree of a node of a tree is the number of children of that node. The total number of sub-trees attached to the node is called the degree of the node.Eg: For node A degree is 3. For node K degree is 0
- The maximum number of children a node can have is often referred to as **the order of a tree.** The height or depth of a tree is the length of the longest path from root to any leaf.

BINARY TREES

Binary tree is a tree in which each node has at most two children, a left child and a right child. Thus the order of binary tree is 2.

A binary tree is either empty or consists of

a) a node called the root

b)left and right sub trees are themselves binary trees.

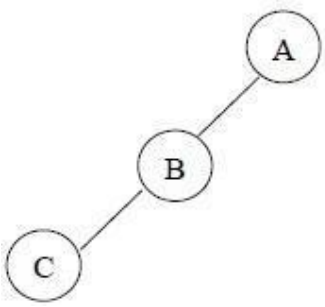
A binary tree is a finite set of nodes which is either empty or consists of a root and two disjoint trees called left sub-tree and right sub-tree.

In binary tree each node will have one data field and two pointer fields for representing the sub branches.

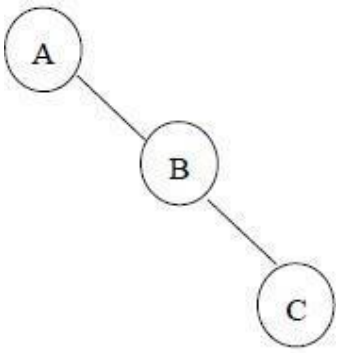
The degree of each node in the binary tree will be at the most two.

Types Of Binary Trees: There are 3 types of binary trees:

1. **Left skewed binary tree:** If the right sub-tree is missing in every node of a tree we call it as left skewed tree.



2. **Right skewed binary tree:** If the left sub-tree is missing in every node of a tree we call it is right subtree.



3. **Complete binary tree:**

The tree in which degree of each node is at the most two is called a complete binary tree. In a

complete binary tree there is exactly one node at level 0, two nodes at level 1 and four nodes at level 2 and so on. So we can say that a complete binary tree depth d will contain exactly 2*l* nodes at each level *l*, where *l* is from 0 to d.

Note:

1. A binary tree of depth n will have maximum 2n -1 nodes.
2. A complete binary tree of level *l* will have maximum 2*l* nodes at each level, where *l* starts from 0.
3. Any binary tree with n nodes will have at the most n+1 null branches.
4. The total number of edges in a complete binary tree with n terminal nodes are 2(n-1).

Binary Tree Representation

A binary tree can be represented mainly in 2 ways:

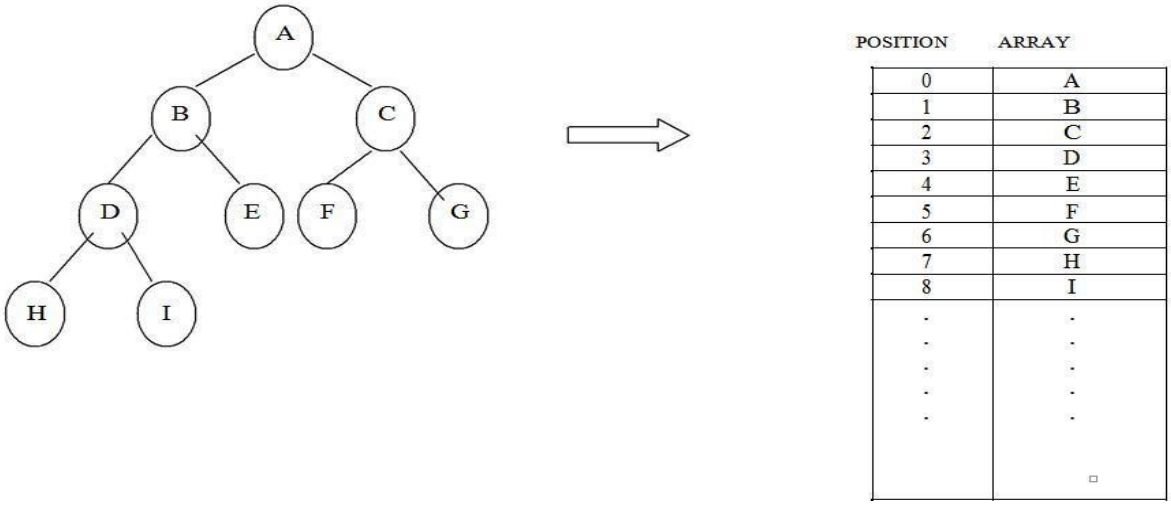
a) Sequential Representation

b) Linked Representation

a) Sequential Representation

The simplest way to represent binary trees in memory is the sequential representation that uses one-dimensional array.

1) The root of binary tree is stored in the 1 st location of array
2) If a node is in the i th location of array, then its left child is in the location 2i+1 and its right child in the location 2i+2
3) The maximum size that is required for an array to store a tree is 2^{d+1}-1, where d is the depth of the tree.



Advantages of sequential representation:

The only advantage with this type of representation is that the direct access to any node can be possible and finding the parent or left children of any particular node is fast because of the random access.

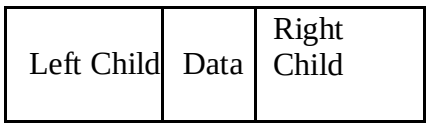
Disadvantages of sequential representation:

1. The major disadvantage with this type of representation is wastage of memory. For example in the skewed tree half of the array is unutilized.
2. In this type of representation the maximum depth of the tree has to be fixed. Because we have decide the array size. If we choose the array size quite larger than the depth of the tree, then it will be wastage of the memory. And if we choose array size lesser than the depth of the tree then we will be unable to represent some part of the tree.
3. The insertions and deletion of any node in the tree will be costlier as other nodes has to be adjusted at appropriate positions so that the meaning of binary tree can be preserved.

As these drawbacks are there with this sequential type of representation, we will search for more flexible representation. So instead of array we will make use of linked list to represent the tree.

b) Linked Representation

Linked representation of trees in memory is implemented using pointers. Since each node in a binary tree can have maximum two children, a node in a linked representation has two pointers for both left and right child, and one information field. If a node does not have any child, the corresponding pointer field is made NULL pointer. In linked list each node will look like this:



Advantages of linked representation:

1. This representation is superior to our array representation as there is no wastage of memory. And so there is no need to have prior knowledge of depth of the tree. Using dynamic memory concept one can create as much memory(nodes) as required. By chance if some nodes are unutilized one can delete the nodes by making the address free.

2. Insertions and deletions which are the most common operations can be done without moving the nodes.

Disadvantages of linked representation:

1. This representation does not provide direct access to a node and special algorithms are required.
2. This representation needs additional space in each node for storing the left and right sub-trees.

TRAVERSING A BINARY TREE

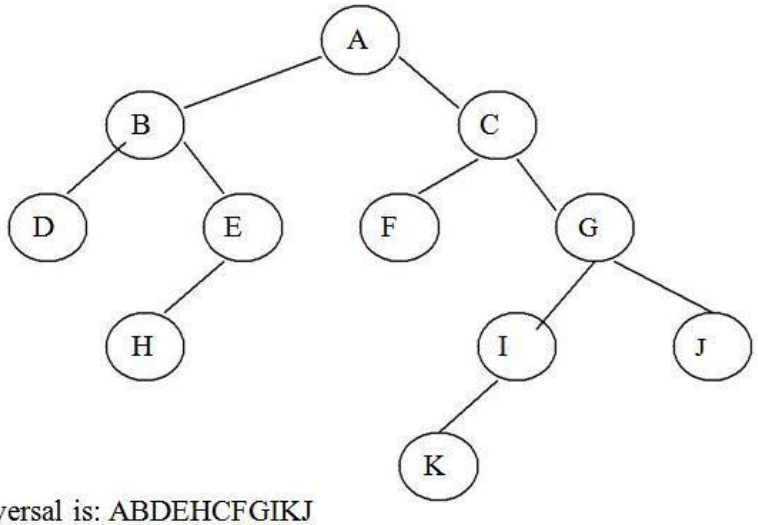
Various Tree Traversals are

a. In-order

b. pre-order

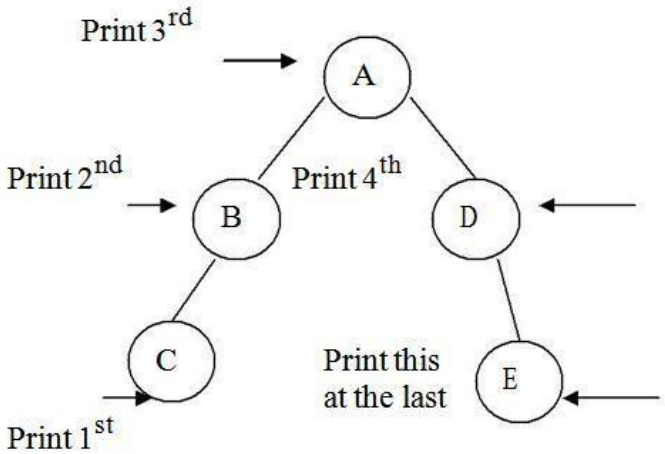
c. post-order

Pre-order	1.Visit the root 2.Traverse the left sub tree in pre-order 3.Traverse the right sub tree in pre-order.	Root | Left | Right
In-order	1.Traverse the left sub tree in in-order 2.Visit the root 3.Traverse the right sub tree in in-order.	Left | Root | Right
Post-order	1.Traverse the left sub tree in post-order 2.Traverse the right sub tree in post-order. 3.Visit the root	Left | Right | Root



The pre-order traversal is: ABDEHCFGIKJ
The in-order traversal is : DBHEAFCKIGJ
The post-order traversal is:DHEBFKIJGCA

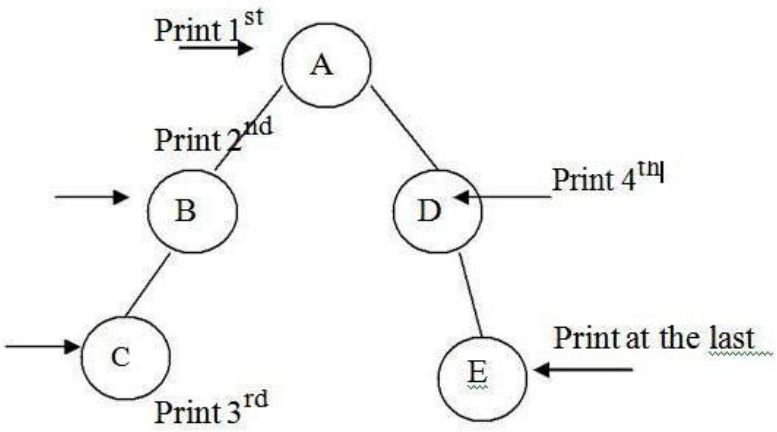
Inorder Traversal:



C-B-A-D-E is the inorder traversal i.e. first we go towards the leftmost node. i.e. C so print that node C. Then go back to the node B and print B. Then root node A then move towards the right sub-tree print D and finally E. Thus we are following the tracing sequence of **Left|Root|Right**. This type of traversal is called inorder traversal. The basic principle is to traverse left sub-tree then root and then the right sub-tree.

```
template <class T>
void inorder(bintree<T> *root)
{
        if(temp!=NULL)
        {
                inorder(root->left);
                cout<<‖root->data‖;
                inorder(root->right);
        }
}
```

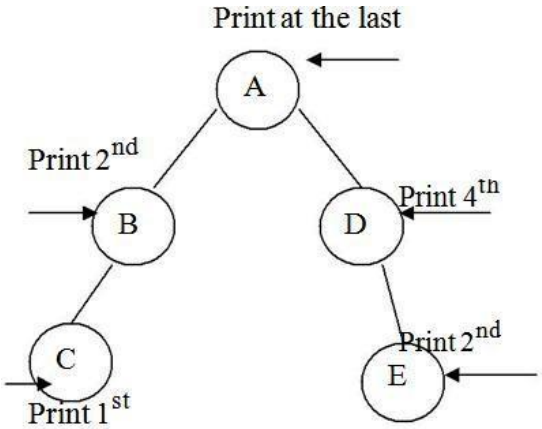
Preorder Traversal



A-B-C-D-E is the preorder traversal of the above fig. We are following **Root|Left|Right** path i.e. data at the root node will be printed first then we move on the left sub-tree and go on printing the data till we reach to the left most node. Print the data at that node and then move to the right sub-tree. Follow the same principle at each sub-tree and go on printing the data accordingly.

```
template <class T>
void inorder(bintree<T> *root)
{
        if(temp!=NULL)
        {
                cout<<‖root->data‖;
                preorder(root->left);
                preorder(root->right);
        }
}
```

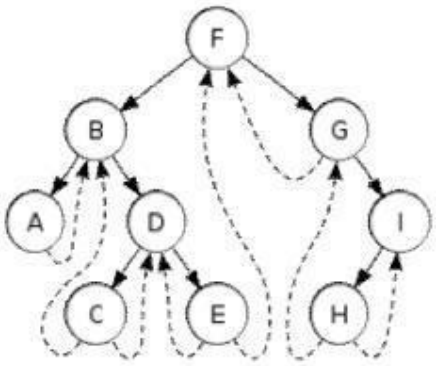
Postorder Traversal:



From figure the postorder traversal is C-D-B-E-A. In the postorder traversal we are following the **Left|Right|Root** principle i.e. move to the leftmost node, if right sub-tree is there or not if not then print the leftmost node, if right sub-tree is there move towards the right most node. The key idea here is that at each subtree we are following the Left|Right|Root principle and print the data accordingly.

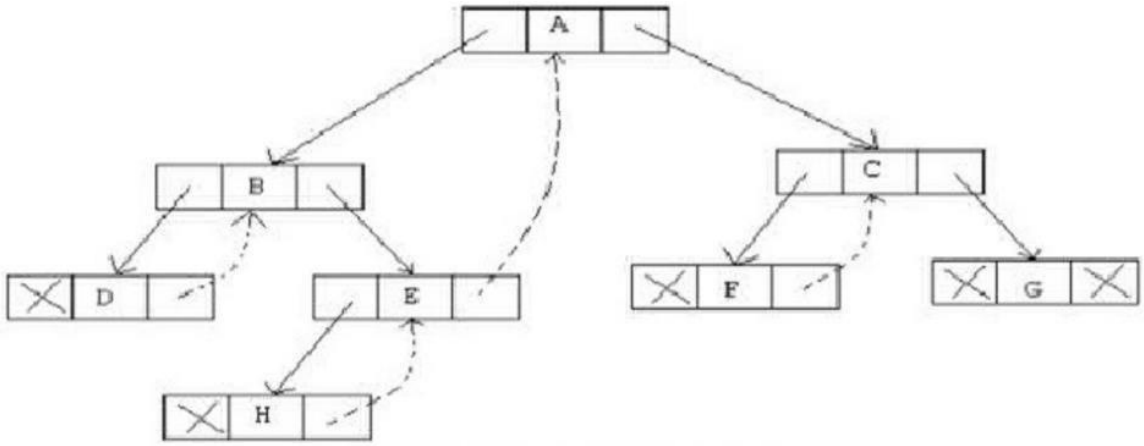
```
template <class T>
void inorder(bintree<T> *root)
{
        if(temp!=NULL)
        {
                postorder(root->left);
                postorder(root->right);
                cout<<‖root->data‖;
        }
}.
```

Threaded binary tree:- "A binary tree is threaded by making all right child pointers that would normally be null point to the inorder successor of the node (if it exists), and all left child pointers that would normally be null point to the inorder predecessor of the node."



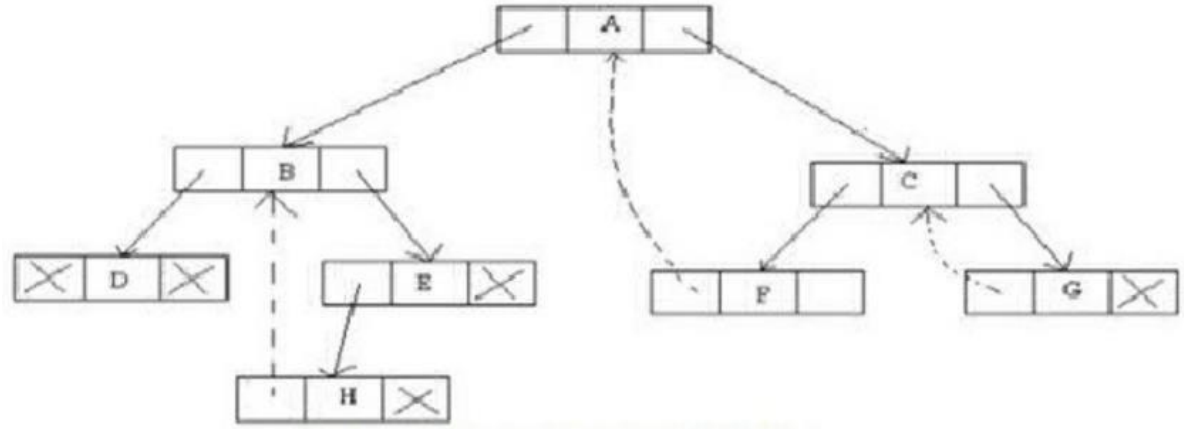
There are many ways to thread a binary tree these are—

1. The right NULL pointer of each leaf node can be replaced by a thread to the successor of that node under in order traversal called a right thread, and the tree will called a **right in- threaded tree or right threaded binary tree.**



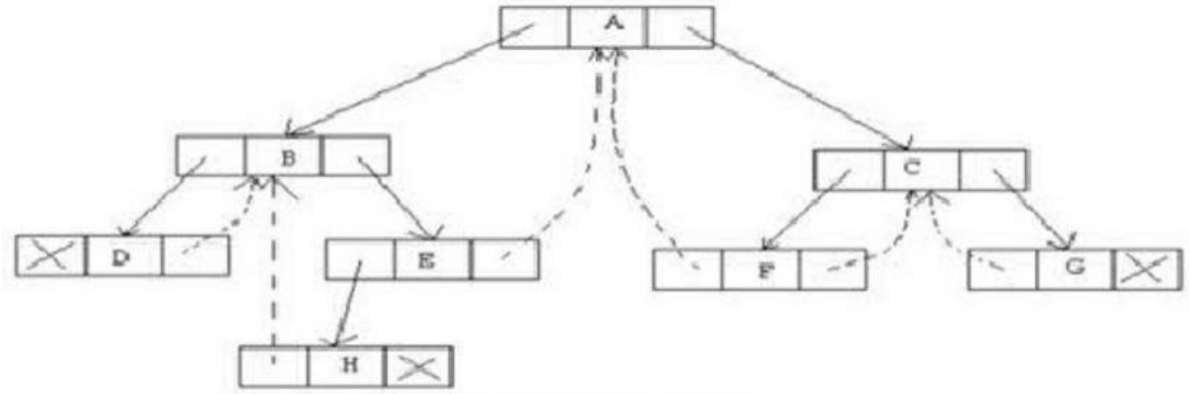
RIGHT THREADED BINARY TREE

2. The left NULL pointer of each node can be replaced by a thread to the predecessor of that node under in order traversal called left thread, and the tree will called **a left in-threaded tree.**



LEFT THREADED BINARY TREE

3. Both left and right NULL pointers can be used to point to predecessor and successor of that node respectively, under in order traversal. Such a tree is called **a fully threaded tree**. A threaded binary tree where only one thread is used is also known as one way threaded tree and where both threads are used is also known as two way threaded tree



Fully Threaded Binary Tree

UNIT-3

Priority Queues – Definition, ADT, Realizing a Priority Queue using Heaps, Definition, insertion, Deletion, External Sorting- Model for external sorting, Multiway merge, Polyphase merge.

Priority Queue

DEFINITION:

A priority queue is a collection of zero or more elements. Each element has a priority or value.

Unlike the queues, which are FIFO structures, the order of deleting from a priority queue is determined by the element priority.

Elements are removed/deleted either in increasing or decreasing order of priority rather than in the order in which they arrived in the queue.

There are two types of priority queues:

- Min priority queue
- Max priority queue

Min priority queue: Collection of elements in which the items can be inserted arbitrarily, but onlysmallest element can be removed.

Max priority queue: Collection of elements in which insertion of items can be in any order but only largest element can be removed.

In priority queue, the elements are arranged in any order and out of which only the smallest or largest element allowed to delete each time.

The implementation of priority queue can be done using arrays or linked list. The data structure **heap** is used to implement the priority queue effectively.

APPLICATIONS:

1. The typical example of priority queue is scheduling the jobs in operating system. Typically OS allocates priority to jobs. The jobs are placed in the queue and position of the job in priority queue determines their priority. In OS there are 3 jobs- real time jobs, foreground jobs and background jobs. The OS always schedules the real time jobs first. If there is no real time jobs pending then it schedules foreground jobs. Lastly if no real time and foreground jobs are pending then OS schedules the background jobs.
2. In network communication, the manage limited bandwidth for transmission the priority queue is used.
3. In simulation modeling to manage the discrete events the priority queue is used.

Various operations that can be performed on priority queue are-

1. Find an element
2. Insert a new element
3. Remove or delete an element

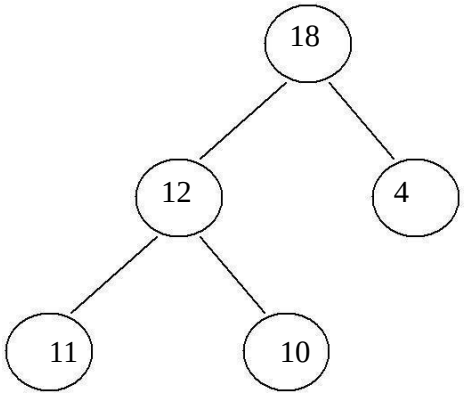
The abstract data type specification for a max priority queue is given below. The specification for a min priority queue is the same as ordinary queue except while deletion, find and remove the element with minimum priority

ABSTRACT DATA TYPE(ADT):

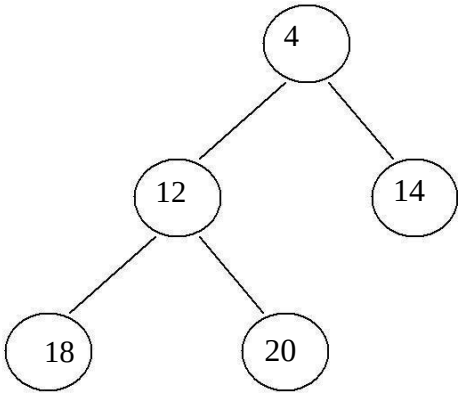
Abstract data type maxPriorityQueue
{
Instances
Finite collection of elements, each has a priority
Operations empty():return true iff the queue is empty
size() :return number of elements in the queue
top() :return element with maximum priority
del() :remove the element with largest priority from the queue
insert(x): insert the element x into the queue
}

HEAPS

Heap is a tree data structure denoted by either a max heap or a min heap.
A max heap is a tree in which value of each node is greater than or equal to value of its children nodes. A min heap is a tree in which value of each node is less than or equal to value of its children nodes.



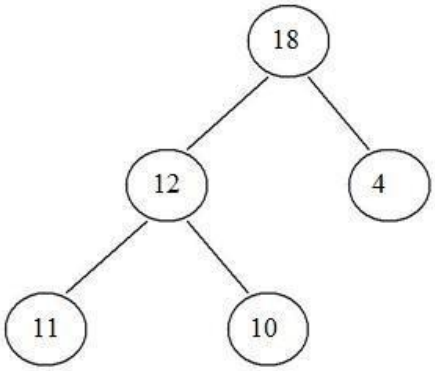
Max heap



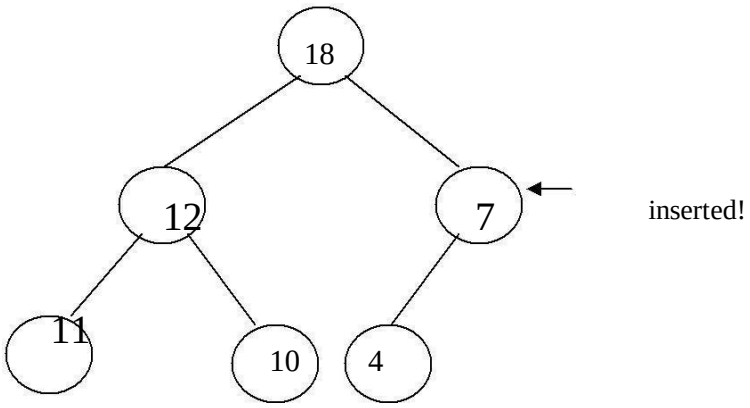
Min heap

Insertion of element in the Heap:

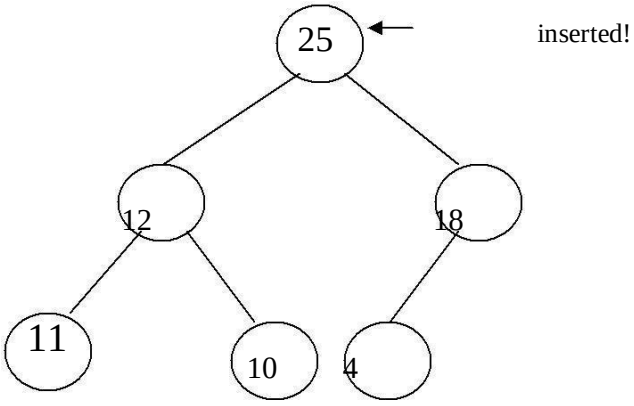
Consider a max heap as given below:



Now if we want to insert 7. We cannot insert 7 as left child of 4. This is because the max heap has a property that value of any node is always greater than the parent nodes. Hence 7 will bubble up 4 will be left child of 7.
Note: When a new node is to be inserted in complete binary tree we start from bottom and from left child on the current level. The heap is always a complete binary tree.



If we want to insert node 25, then as 25 is greatest element it should be the root. Hence 25 will bubble up and 18 will move down.



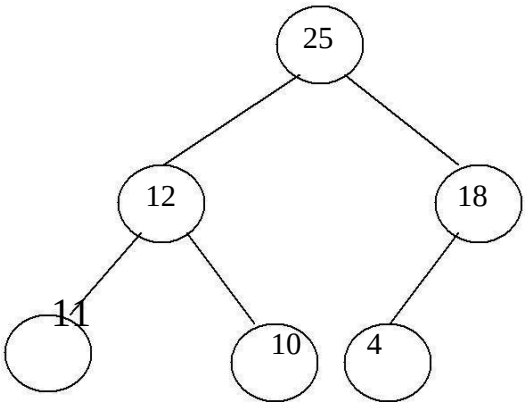
The insertion strategy just outlined makes a single bubbling pass from a leaf toward the root. At each level we do (1) work, so we should be able to implement the strategy to have complexity O(height) = O(log n).

```
void Heap::insert(int item)
{
        int temp;       //temp node starts at leaf and moves up.
        temp=++size;
        while(temp!=1 && heap[temp/2]<item)     //moving element down
        {
                H[temp] = H[temp/2]; temp=temp/2;
                //finding the parent
        }
        H[temp]=item;
}
```

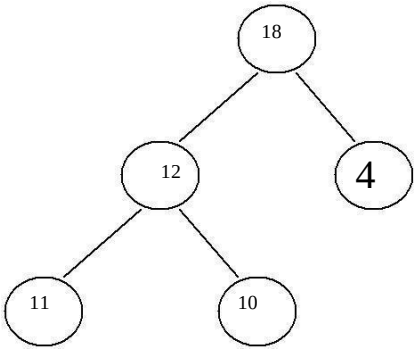
Deletion of element from the heap:

For deletion operation always the maximum element is deleted from heap. In Max heap the maximum element is always present at root. And if root element is deleted then we need to reheapify the tree.

Consider a Max heap



Delete root element:25, Now we cannot put either 12 or 18 as root node and that should be greater than all its children elements.



Now we cannot put 4 at the root as it will not satisfy the heap property. Hence we will bubble up 18 and place 18 at root, and 4 at position of 18.

If 18 gets deleted then 12 becomes root and 11 becomes parent node of 10.

Thus deletion operation can be performed. The time complexity of deletion operation is O(log n).

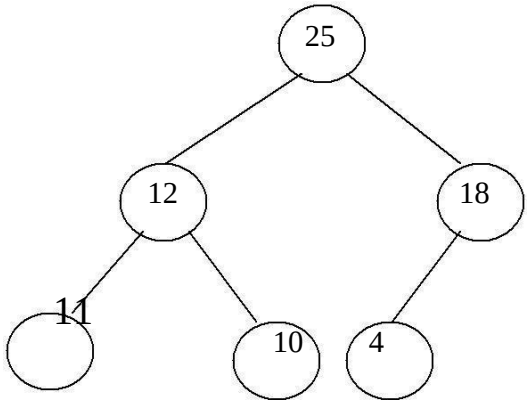
1. Remove the maximum element which is present at the root. Then a hole is created at the root.
2. Now reheapify the tree. Start moving from root to children nodes. If any maximum element is found then place it at root. Ensure that the tree is satisfying the heap property or not.
3. Repeat the step 1 and 2 if any more elements are to be deleted.

```
void heap::delet(int item)
{
int item, temp;
if(size==0)
cout<<"Heap is empty\n"; else
{
//remove the last elemnt and reheapify
item=H[size--];
```

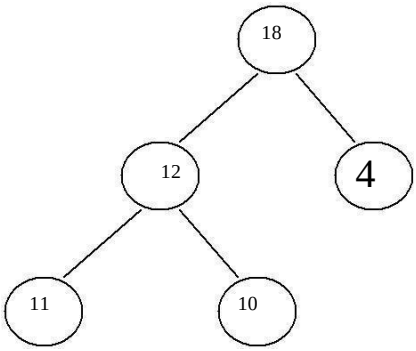
```
//item is placed at root temp=1;
child=2;
while(child<=size)
{
```

For deletion operation always the maximum element is deleted from heap. In Max heap the maximum element is always present at root. And if root element is deleted then we need to reheapify the tree.

Consider a Max heap



Delete root element:25, Now we cannot put either 12 or 18 as root node and that should be greater than all its children elements.



Now we cannot put 4 at the root as it will not satisfy the heap property. Hence we will bubble up 18 and place 18 at root, and 4 at position of 18.

If 18 gets deleted then 12 becomes root and 11 becomes parent node of 10.

Thus deletion operation can be performed. The time complexity of deletion operation is O(log n).

4. Remove the maximum element which is present at the root. Then a hole is created at the root.
5. Now reheapify the tree. Start moving from root to children nodes. If any maximum element is found then place it at root. Ensure that the tree is satisfying the heap property or not.
6. Repeat the step 1 and 2 if any more elements are to be deleted.

```
void heap::delet(int item)
{
int item, temp;
if(size==0)
cout<<"Heap is empty\n"; else
{
//remove the last elemnt and reheapify
item=H[size--];
//item is placed at root temp=1;
```

```
child=2;
while(child<=size)
{
if(child<size && H[child]<H[child+1]) child++;
if(item>=H[child])
break;
H[temp]=H[child];
temp=child;
child=child*2;
}
//pl;ace the largest item at root
H[temp]=item;
}
```

Applications Of Heap:

1. Heap is used in sorting algorithms. One such algorithm using heap is known as heap sort.
2. In priority queue implementation the heap is used.

HEAP SORT

Heap sort is a method in which a binary tree is used. In this method first the heap is created using binary tree and then heap is sorted using priority queue.

Eg:

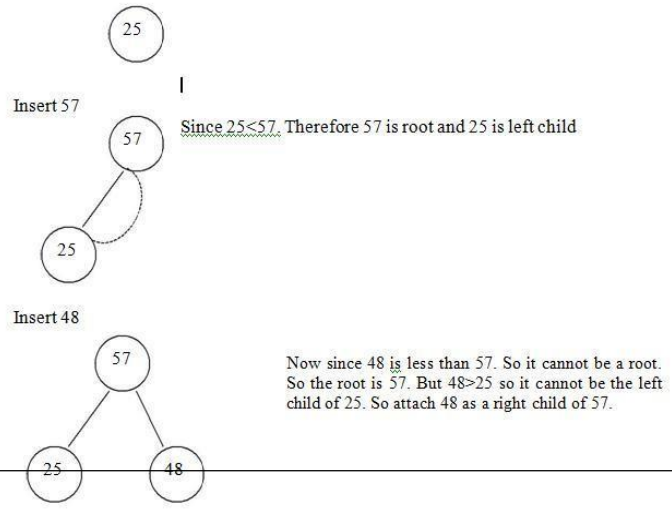
25 57 48 38 10 91 84 33

In the heap sort method we first take all these elements in the array –A‖

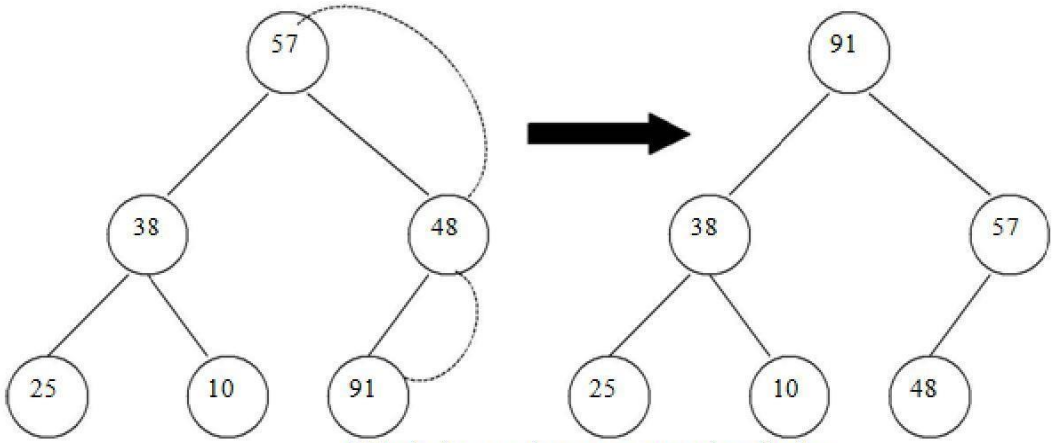
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
25	57	48	38	10	91	84	33

Now start building the heap structure. In forming the heap the key point is build heap in such a way that the highest value in the array will always be a root.

Insert 25

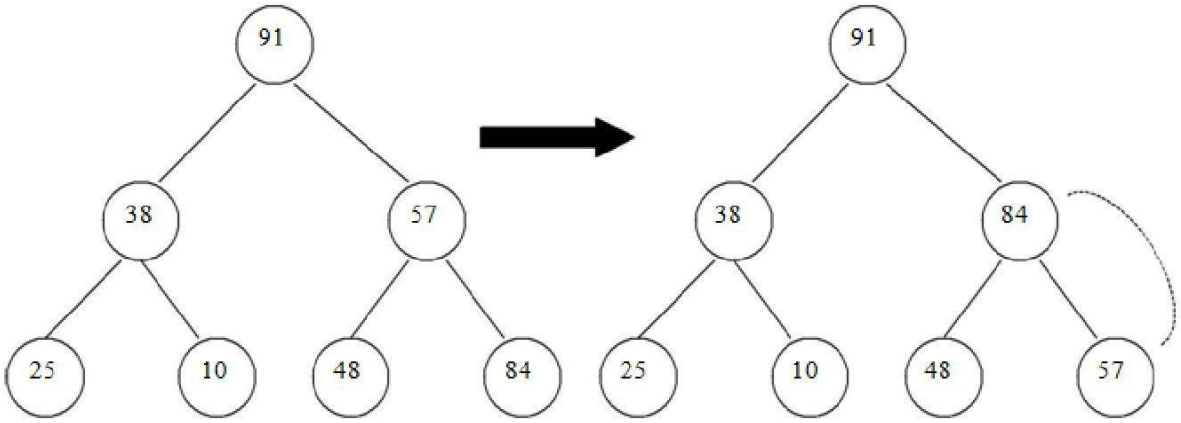


Insert 91



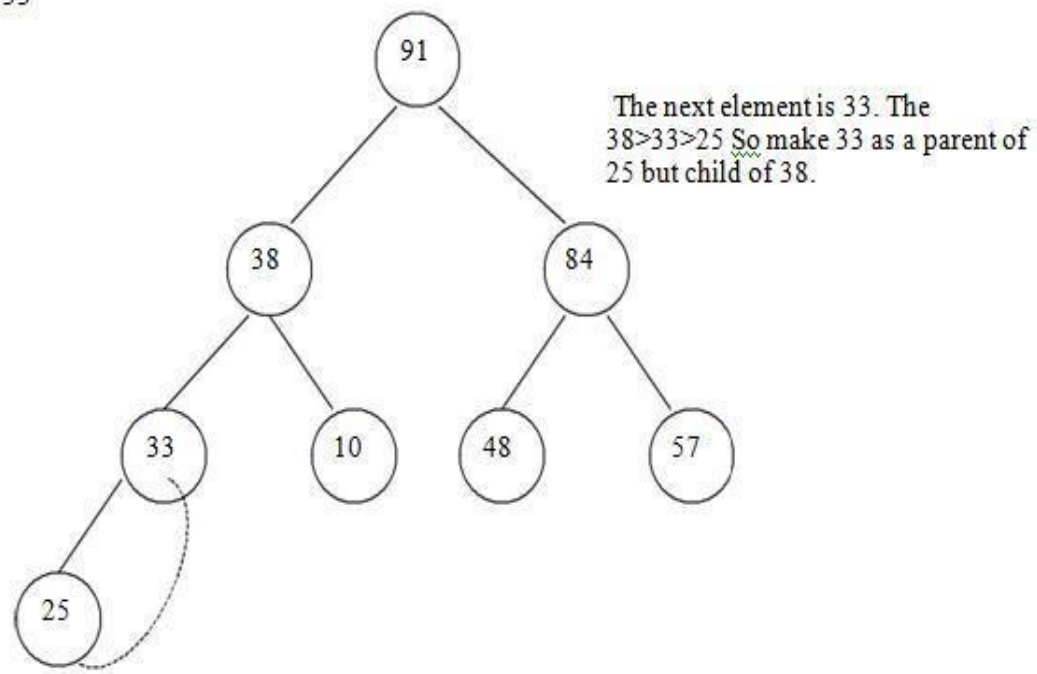
91 is the largest element compared to all other elements , naturally it will be the root node.

Insert 84



The next element is 84, which 91>84>57 the middle element. So 84 will be the parent of 57. For making the complete binary tree 57 will be attached as right of 84.

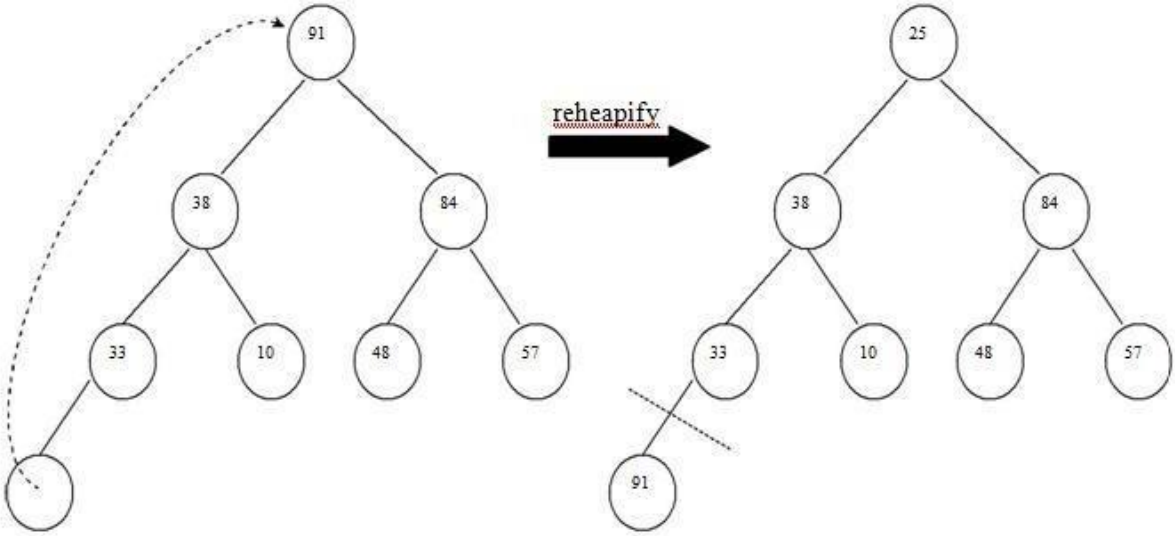
Insert 33



The next element is 33. The 38>33>25 So make 33 as a parent of 25 but child of 38.

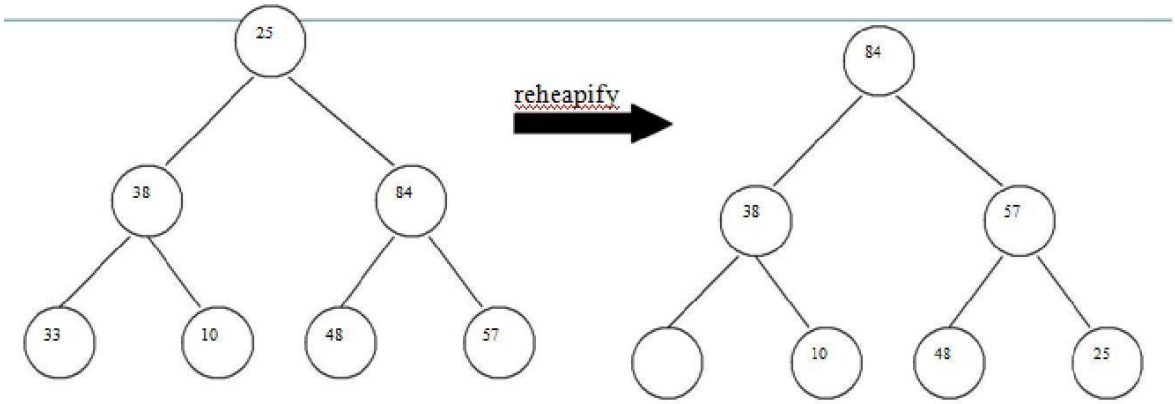
Now the heap is formed. Let us sort it. For sorting the heap remember two main things the first thing is that the binary tree form of the heap should not be distributed at all. For the complete sorting binary tree should be remained. And the second thing is that we will start sorting the higher elements at the end of array in sorted manner i.e.. A[7]=91, A[6]=84 and so on..

Step 1:- Exchange A[0] with A[7]

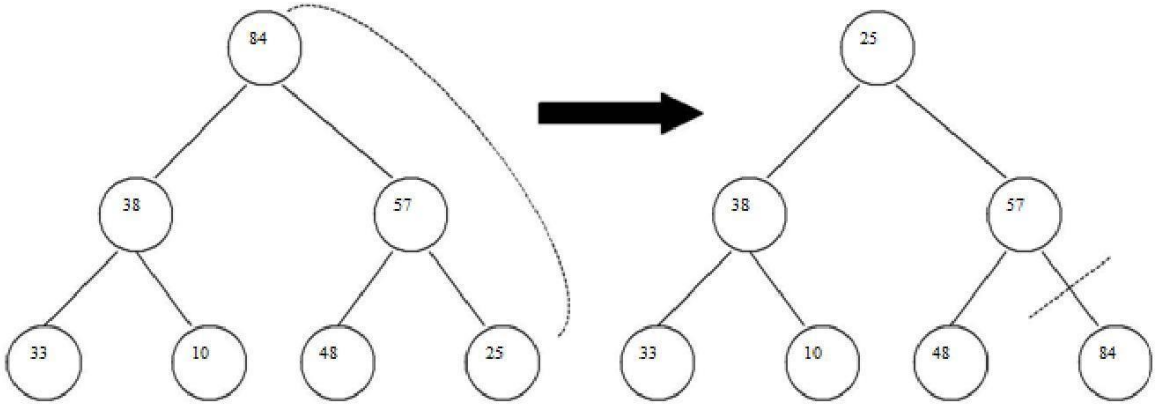


Queue

							91



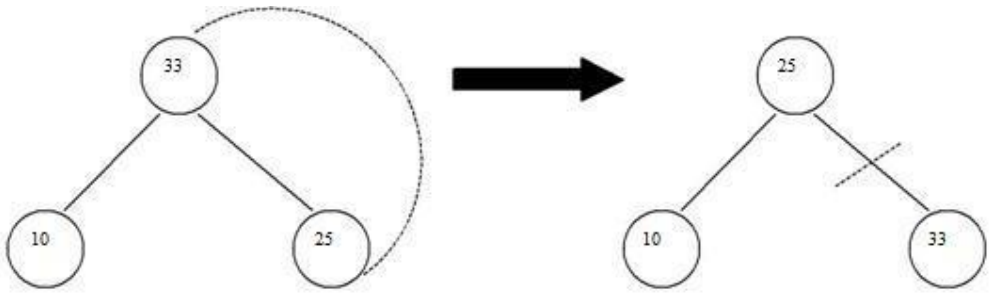
Step 2:-Exchane A[0] with A[6]



Queue

						84	91

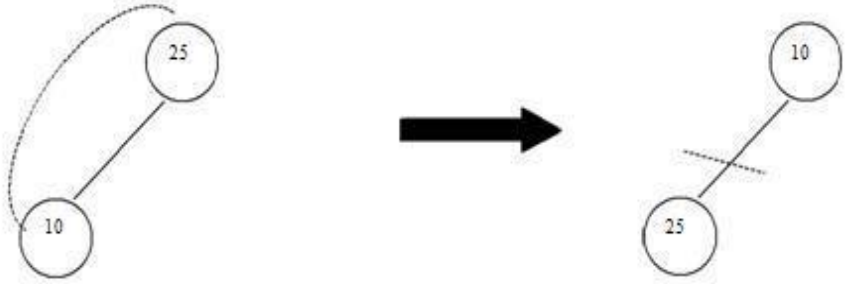
ep 5:-Exchane A[0] with A[2]



Queue

		33	38	48	57	84	91

Step 6:-Exchane A[0] with A[1]



Queue

	25	33	38	48	57	84	91

Step 6: The remaining element 10 has already occupied its proper position because only one position is empty so insert 10 also in the queue.

10	25	33	38	48	57	84	91

```
    // If largest is not
    root if (largest != i)
    {
            swap(&arr[i], &arr[largest]);

            // Recursively heapify the affected sub-
            tree heapify(arr, n, largest);
    }
}

// function to do heap sort
void heapSort(int arr[], int n)
{ int i;
    // Build heap (rearrange array) for
    ( i = n / 2 - 1; i >= 0; i--)
            heapify(arr, n, i);

    // One by one extract an element from
    heap for ( i=n-1; i>=0; i--)
    {
            // Move current root to end
            swap(&arr[0], &arr[i]);

            // call max heapify on the reduced heap
            heapify(arr, i, 0);
    }
}
/* A utility function to print array of size n */
void printArray(int arr[], int n)
{
    for (int i=0; i<n; ++i)
            cout << arr[i] << " ";
    cout << "\n";
}
int main()
{
int n,i;
int list[30];
            cout<<"enter no of elements\n";
            cin>>n;
            cout<<"enter "<<n<<" numbers ";
            for(i=0;i<n;i++)
            cin>>list[i];
            heapSort(list, n);
            cout << "Sorted array is \n";
    printArray(list, n);
return 0;
}
```

EXTERNAL SORTING

- All the algorithms require that the input fit into main memory. There are, some applications where the input is much too large to fit into memory.
- To do so, external sorting algorithms are designed to handle very large inputs.
- Internal sorting deals with the ordering of records of a file in the ascending or descending order when the whole file or list is compact enough to be accommodate in the internal memory of the computer.
- In many applications and problems it is quite common to encounter huge files comprising millions of records which need to be sorted for their effective use in the application concerned.
- The application domains of e-governance, digital library, search engines, on-line telephone directory and electoral system, to list a few, deal with voluminous files of records.

Majority of the internal sorting techniques are virtually incapable of sorting large files since they require the whole file in the internal memory of the computer, which is impossible. Hence the need for external sorting methods which are exclusive strategies to sort huge files.

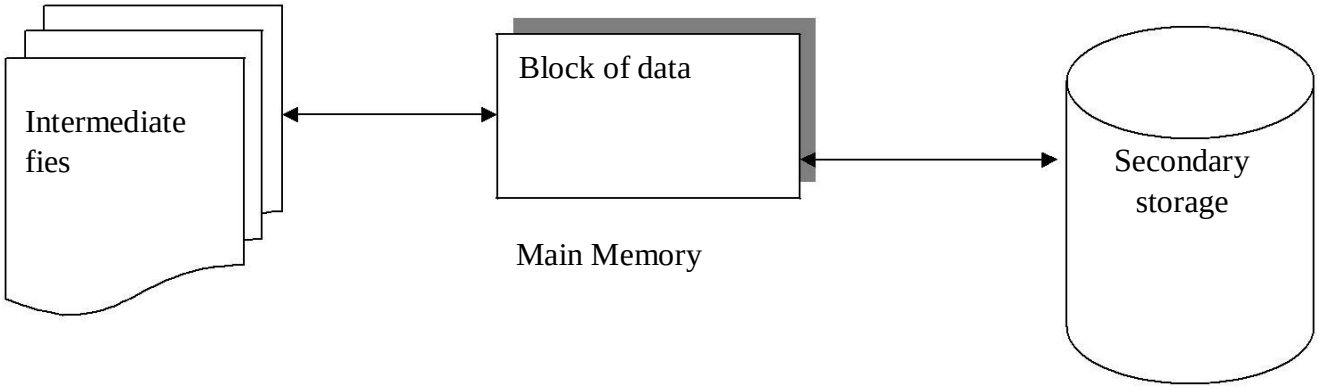
External sorting is a term for a class of sorting algorithms that can handle massive amounts of data. External sorting is required when the data being sorted do not fit into the main memory of a computing device (usually RAM) and instead they must reside in the slower external memory (usually a hard drive). External sorting typically uses a hybrid sort-merge strategy. In the sorting phase, chunks of data small enough to fit in main memory are read, sorted, and written out to a temporary file. In the merge phase, the sorted sub-files are combined into a single larger file.

One example of external sorting is the external merge sort algorithm, which sorts chunks that each fit in RAM, then merges the sorted chunks together. We first divide the file into **runs** such that the size of a run is small enough to fit into main memory. Then sort each run in main memory using merge sort sorting algorithm. Finally merge the resulting runs together into successively bigger runs, until the file is sorted.

- The principle behind external sorting

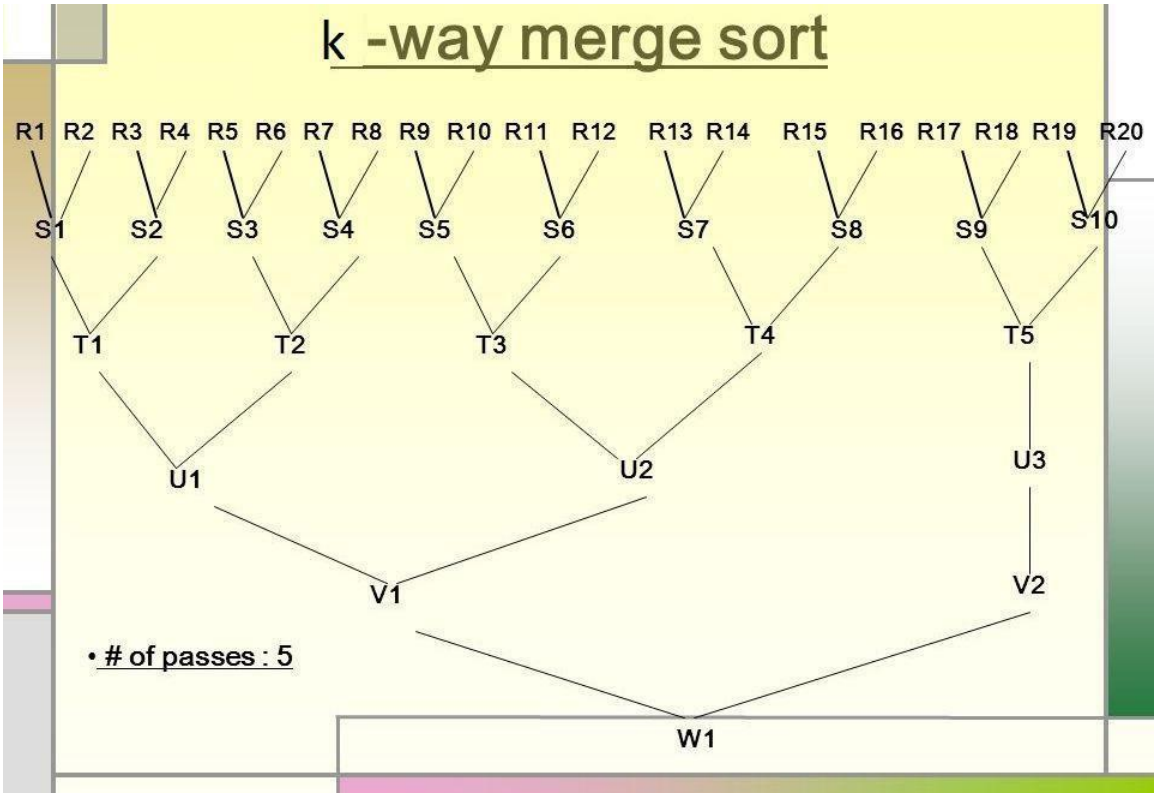
- Due to their large volume, the files are stored in external storage devices such as tapes, disks or drums.
- The external sorting strategies therefore need to take into consideration the kind of medium on which the files reside, since these influence their work strategy.
- A common principal behind most popular external sorting methods is outlined below:

1. Internally sort batches of records from the source file to generate runs. Write out the runs as and when they are generated on to the external storage devices.
2. Merge the runs generated in the earlier phase, to obtain larger but fewer runs, and write them out onto the external storage devices.
3. Repeat the run generated and merge, until in the final phase only one run gets generated, on which



MULTIWAY MERGE:

K-Way Merge Algorithms or Multiway Merges are a specific type of Sequence Merge Algorithms that specialize in taking in multiple sorted lists and merging them into a single sorted list.



Example 1:

External Sorting: Example of multiway external sorting

Ta1: 17, 3, 29, 56, 24, 18, 4, 9, 10, 6, 45, 36, 11, 43

Assume that we have three tapes (k = 3) and the memory can hold three records.

Main memory sort

The first three records are read into memory, sorted and written on Tb1, the second three records are read into memory, sorted and stored on Tb2, finally the third three records are read into memory, sorted and stored on Tb3. Now we have one run on each of the three tapes:

Tb1: 3, 17, 29
Tb2: 18, 24, 56
Tb3: 4, 9, 10

The next portion of three records is sorted into main memory
and stored as the second run on Tb1: Tb1: 3, 17, 29, 6, 36,
45

The next portion, which is also the last one, is sorted and stored onto Tb2:
Tb2: 18, 24, 56, 11, 43

Nothing is stored on Tb3.

Thus, after the main memory sort, our tapes look like this:

Tb1: 3, 17, 29, | 6, 36, 45,
Tb2: 18, 24, 56, | 11, 43
Tb3: 4, 9, 10

Merging

Merging runs of length M to obtain runs of length

k*M In our example we merge runs of length 3
and the resulting runs would be of length 9.

We build a heap tree in main memory out of the first records in each tape.
These records are: 3, 18, and 4.
We take the smallest of them - 3, using the deleteMin
operation, and store it on tape Ta1.
The record '3' belonged to Tb1, so we read the next record from Tb1 -
17, and insert it into the heap. Now the heap contains 18, 4, and 17.

The next deleteMin operation will output 4, and it will be stored on Ta1.
The record '4' comes from Tb3, so we read the next record '9' from Tb3
and insert it into the heap.
Now the heap contains 18, 17 and 9.

Proceeding in this way, the first three runs will be stored in sorted order on
Ta1. Ta1: 3, 4, 9, 10, 17, 18, 24, 29, 56

Now it is time to build a heap of the second three runs.
(In fact they are only two, and the run on Tb2 is not complete.)
The resulting sorted run on Ta2 will be:
Ta2: 6, 11, 36, 43, 45

This finishes the first pass.

Building runs of length k*k*M

We have now only two tapes: Ta1 and Ta2.
We build a heap of the first elements of the two tapes - 3 and 6,
and output the smallest element '3' to tape Tb1.
Then we read the next record from the tape where the record '3' belonged - Ta1,
and insert it into the heap.
Now the heap contains 6 and 4, and using the deleteMin operation
the smallest record - 4 is output to tape Tb1.
Proceeding in this way, the entire file will be sorted on tape Tb1.

Tb1: 3, 4, 6, 9, 10, 11, 17, 18, 24, 29, 36, 43, 45, 56

The number of passes for the multiway merging is logk(N/M).
In the example this is [log3(14/3)] + 1 = 2.

Example 2:

If we have extra tapes, then we can expect to reduce the number of passes required to sort the input. This is done by extending the basic (two- way) merge to a k-way merge.
Merging two runs is done by winding each input tape to the beginning of each run. Then the smaller element is found, placed on an output tape, and the appropriate input tape is advanced. If there are k input tapes, this strategy works the same way, the only difference being that it is slightly more complicated to find the smallest of the k elements. We can find the smallest of these elements by using a priority queue. To obtain the next element to write on the output tape, perform a deleteMin operation. The appropriate input tape is advanced, and if the run on the input tape is not yet completed, we insert the new element into the priority queue. For example distribute the input onto three tapes.

T_{a1}						
T_{a2}						
T_{a3}						
T_{b1}	11	81	94	41	58	75
T_{b2}	12	35	96	15		
T_{b3}	17	28	99			

We need two more passes of three-way merging to complete the sort.

Dictionaries:- linear list representation, skip list representation, operations insertion, deletion and searching, hash table representation, hash functions, collision resolution-separate chaining, open addressing-linear probing, quadratic probing, double hashing, rehashing, extendible hashing, comparison of hashing and skip lists.

DICTIONARIES:

Dictionary is a collection of pairs of key and value where every value is associated with the corresponding key.
Basic operations that can be performed on dictionary are:
1. Insertion of value in the dictionary
2. Deletion of particular value from dictionary
3. Searching of a specific value with the help of key

Linear List Representation

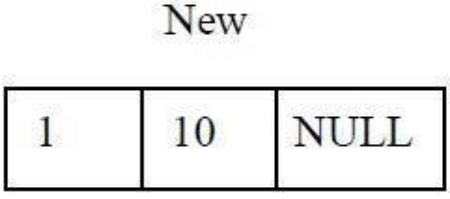
The dictionary can be represented as a linear list. The linear list is a collection of pair and value.
There are two method of representing linear list.
1. Sorted Array- An array data structure is used to implement the dictionary.
2. Sorted Chain- A linked list data structure is used to implement the dictionary

Structure of linear list for dictionary:

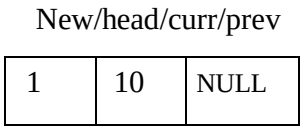
```
class dictionary
{

private:
        int k,data;
        struct node
        {
        public: int key;
        int value;
        struct node *next;
        } *head;

public:
        dictionary();
        void insert_d( );
        void delete_d( );
        void display_d( );
        void length();
};
```

Insertion of new node in the dictionary:

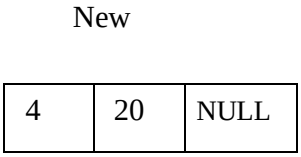
Consider that initially dictionary is empty then
head = NULL
We will create a new node with some key and value contained in it.



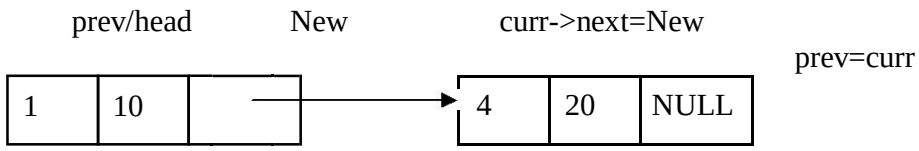
Now as head is NULL, this new node becomes head. Hence the dictionary contains only one record. this node will be _curr‘ and _prev‘ as well. The _cuur‘ node will always point to current visiting node and _prev‘ will always point to the node previous to _curr‘ node. As now there is only one node in the list mark as _curr‘ node as _prev‘ node.



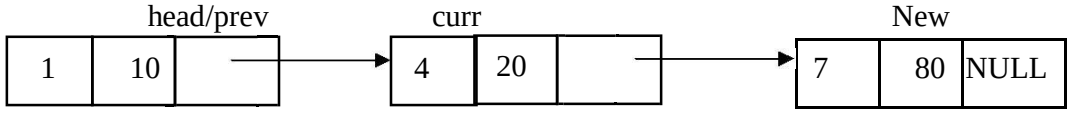
Insert a record, key=4 and value=20,



Compare the key value of _curr‘ and _New‘ node. If New->key > Curr->key then attach New node to _curr‘ node.

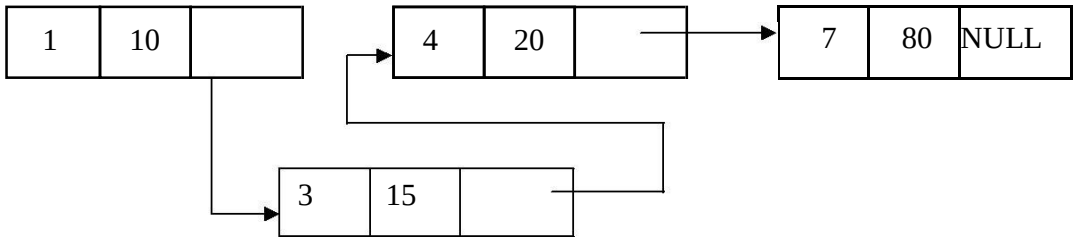


Add a new node <7,80> then



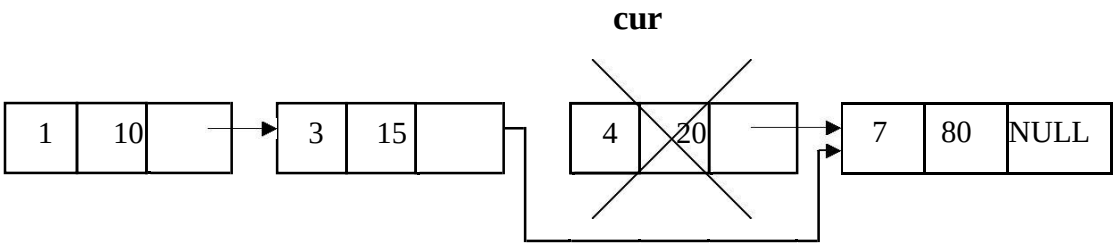
If we insert <3,15> then we have to search for it proper position by comparing key value.

(curr->key < New->key) is false. Hence else part will get executed.



The delete operation:

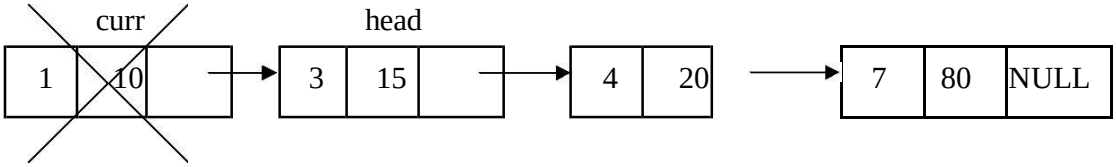
Case 1: Initially assign _head‘ node as _curr‘ node.Then ask for a key value of the node which is to be deleted. Then starting from head node key value of each jode is cked and compared with the desired node‘s key value. We will get node which is to be deleted in variable _curr‘. The node given by variable _prev‘ keeps track of previous node of _cuu‘ node. For eg, delete node with key value 4 then



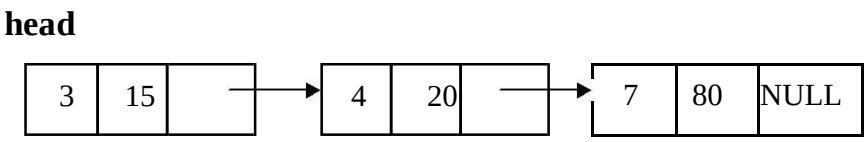
se 2:

If the node to be deleted is head node
i.e.. if(curr==head)

Then, simply make _head‘ node as next node and delete _curr‘

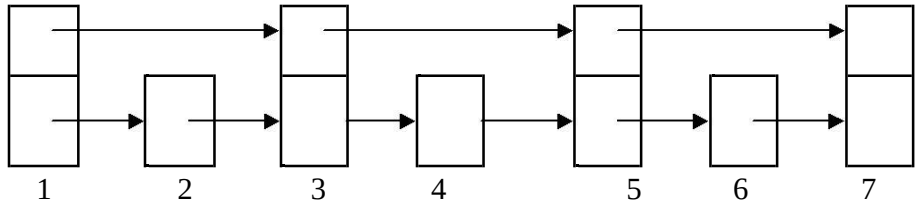


Hence the list becomes



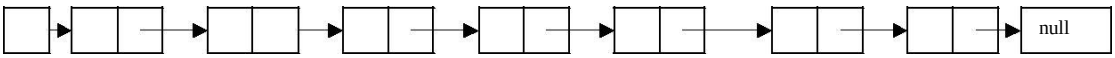
SKIP LIST REPRESENTATION

Skip list is a variant list for the linked list. Skip lists are made up of a series of nodes connected one after the other. Each node contains a key and value pair as well as one or more references, or pointers, to nodes further along in the list. The number of references each node contains is determined randomly. This gives skip lists their probabilistic nature, and the number of references a node contains is called its node level.
There are two special nodes in the skip list one is head node which is the starting node of the list and tail node is the last node of the list

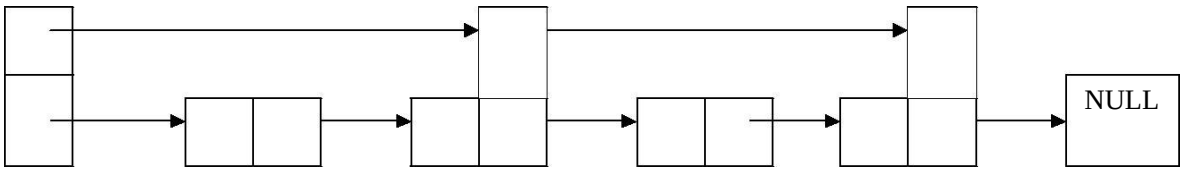


The skip list is an efficient implementation of dictionary using sorted chain. This is because in skip list each node consists of forward references of more than one node at a time.

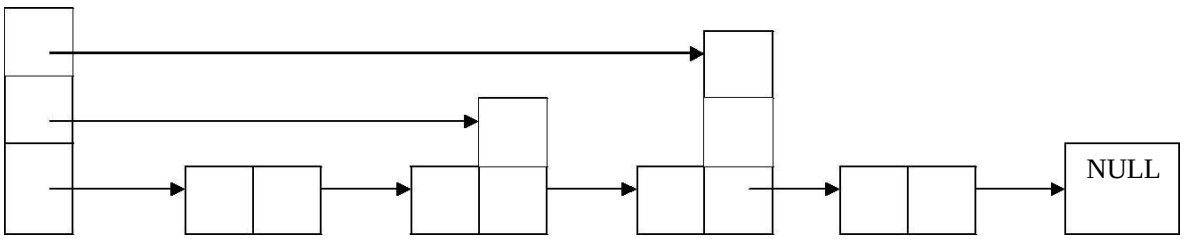
Eg:



Now to search any node from above given sorted chain we have to search the sorted chain from head node by visiting each node. But this searching time can be reduced if we add one level in every alternate node. This extra level contains the forward pointer of some node. That means in sorted chain come nodes can holds pointers to more than one node.



If we want to search node 40 from above chain there we will require comparatively less time. This search again can be made efficient if we add few more pointers forward references.

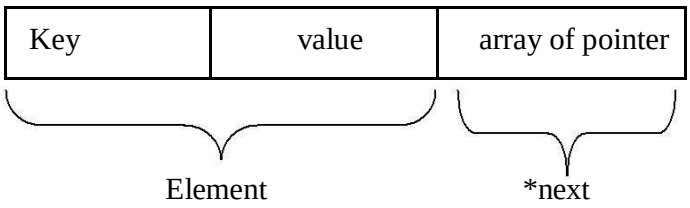


Node structure of skip list:

```
template <class K, class E>
struct skipnode
{
        typedef pair<const K,E> pair_type;
        pair_type element;
        skipnode<K,E> **next;
        skipnode(const pair_type &New_pair, int MAX):element(New_pair)
        {
                next=new skipnode<K,E>*[MAX];
```

```
        }
};
```

The individual node looks like this:



Searching:
The desired node is searched with the help of a key value.

```
template<class K, class E>
skipnode<K,E>* skipLst<K,E>::search(K& Key_val)
{
skipnode<K,E>* Forward_Node = header;
for(int i=level;i>=0;i--)
{
while (Forward_Node->next[i]->element.key < key_val)
Forward_Node = Forward_Node->next[i];
last[i] = Forward_Node;
}
return Forward_Node->next[0];
}
```

Searching for a key within a skip list begins with starting at header at the overall list level and moving forward in the list comparing node keys to the key_val. If the node key is less than the key_val, the search continues moving forward at the same level. If o the other hand, the node key is equal to or greater than the key_val, the search drops one level and continues forward. This process continues until the desired key_val has been found if it is present in the skip list. If it is not, the search will either continue at the end of the list or until the first key with a value greater than the search key is found.

Insertion:

There are two tasks that should be done before insertion operation:

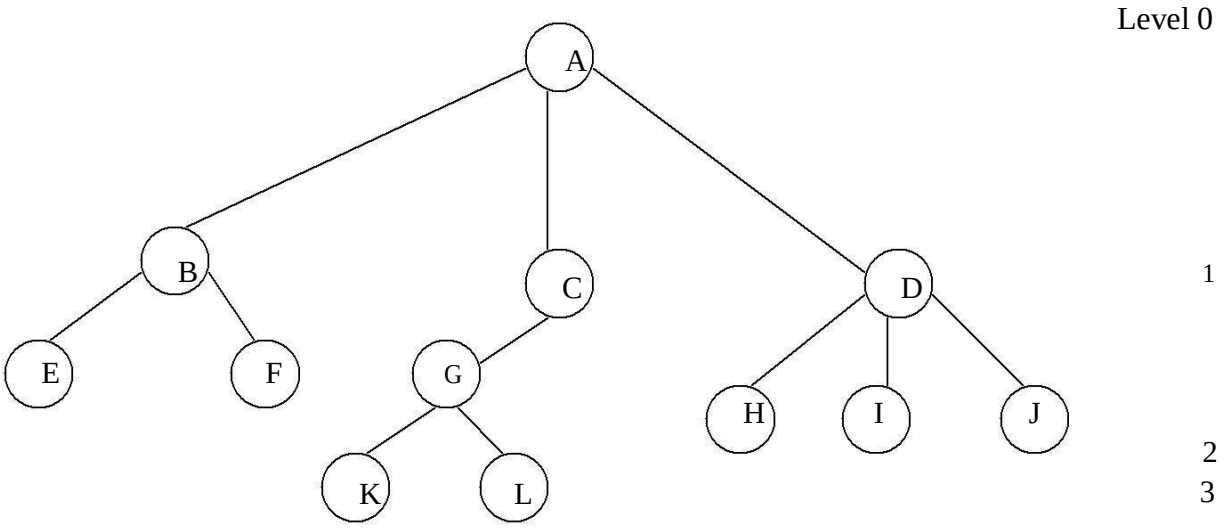
1. Before insertion of any node the place for this new node in the skip list is searched. Hence before any insertion to take place the search routine executes. The last[] array in the search routine is used to keep track of the references to the nodes where the search, drops down one level.

2. The level for the new node is retrieved by the routine randomelevel()

Search Trees:-Binary Search Trees, Definition, ADT, Implementation, Operations- Searching, Insertion and Deletion, AVL Trees, Definition, Height of an AVL Tree, Operations – Insertion, Deletion and Searching, B-Trees, B-Tree of order m, height of a B-Tree, insertion, deletion and searching. **Graphs:** Basic terminology representation of graphs, graph search methods DFS,BFS.

TREES

A Tree is a data structure in which each element is attached to one or more elements directly beneath it.



Terminology

- The connections between elements are called **branches**.
- A tree has a single root, called **root** node, which is shown at the top of the tree. i.e. root is always at the highest level 0.
- Each node has exactly one node above it, called **parent.** Eg: A is the parent of B,C and D.
- The nodes just below a node are called its **children**. ie. child nodes are one level lower than the parent node.
- A node which does not have any child called **leaf or terminal node**. Eg: E, F, K, L, H, I and M are leaves.
- Nodes with at least one child are called **non terminal or internal nodes**.
- The child nodes of same parent are said to be **siblings**.
- A **path** in a tree is a list of distinct nodes in which successive nodes are connected by branches in the tree.
- The **length** of a particular path is the number of branches in that path. The **degree** of a node of a tree is the number of children of that node.
- The maximum number of children a node can have is often referred to as the **order** of a tree. The **height or depth** of a tree is the length of the longest path from root to any leaf.

1. **Root:** This is the unique node in the tree to which further sub trees are attached. Eg: A

Degree of the node: The total number of sub-trees attached to the node is called the degree of the node.Eg: For node A degree is 3. For node K degree is 0

3. **Leaves:** These are the terminal nodes of the tree. The nodes with degree 0 are always the leaf nodes.

Eg: E, F, K, L,H, I, J

4. **Internal nodes:** The nodes other than the root node and the leaves are called the internal nodes. Eg: B, C, D, G

5. **Parent nodes:** The node which is having further sub-trees(branches) is called the parent node of those sub-trees. Eg: B is the parent node of E and F.

6. **Predecessor:** While displaying the tree, if some particular node occurs previous to some other node then that node is called the predecessor of the other node. Eg: E is the predecessor of the node B.

7. **Successor:** The node which occurs next to some other node is a successor node. Eg: B is the successor of E and F.

8. **Level of the tree:** The root node is always considered at level 0, then its adjacent children are supposed to be at level 1 and so on. Eg: A is at level 0, B,C,D are at level 1, E,F,G,H,I,J are at level 2, K,L are at level 3.

9. **Height of the tree:** The maximum level is the height of the tree. Here height of the tree is 3. The height if the tree is also called depth of the tree.

10. **Degree of tree:** The maximum degree of the node is called the degree of the tree.

BINARY TREES

Binary tree is a tree in which each node has at most two children, a left child and a right child. Thus the order of binary tree is 2.

A binary tree is either empty or consists
of a) a node called the root
b) left and right sub trees are themselves binary trees.

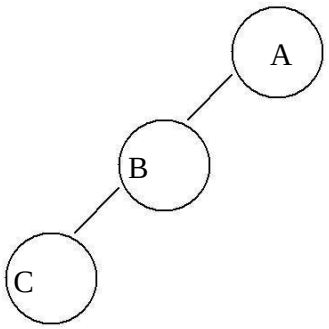
A binary tree is a finite set of nodes which is either empty or consists of a root and two disjoint trees called left sub-tree and right sub- tree.

In binary tree each node will have one data field and two pointer fields for representing the sub-branches. The degree of each node in the binary tree will be at the most two.

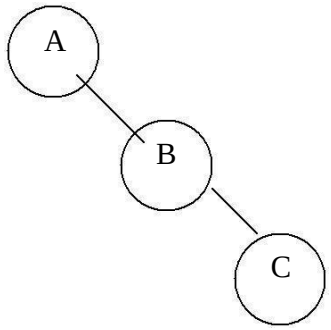
Types Of Binary Trees:

There are 3 types of binary trees:

1. **Left skewed binary tree:** If the right sub-tree is missing in every node of a tree we call it as left skewed tree.

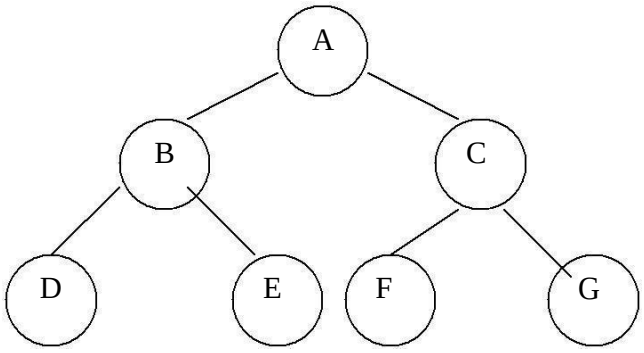


2. **Right skewed binary tree:** If the left sub-tree is missing in every node of a tree we call it is right sub-tree.



3. Complete binary tree:

The tree in which degree of each node is at the most two is called a complete binary tree. In a complete binary tree there is exactly one node at level 0, two nodes at level 1 and four nodes at level 2 and so on. So we can say that a complete binary tree depth d will contain exactly 2^l nodes at each level *l*, where *l* is from 0 to d.



Note:

1. A binary tree of depth n will have maximum $2^n - 1$ nodes.
2. A complete binary tree of level *l* will have maximum 2^l nodes at each level, where *l* starts from 0.
3. Any binary tree with n nodes will have at the most n+1 null branches.
4. The total number of edges in a complete binary tree with n terminal nodes are 2(n-1).

Binary Tree Representation

A binary tree can be represented mainly in 2 ways:

a) Sequential Representation

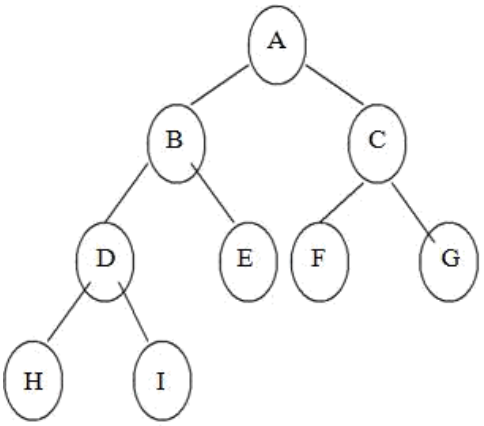
b) Linked Representation

a) Sequential Representation

The simplest way to represent binary trees in memory is the sequential representation that uses one-dimensional array.

1) The root of binary tree is stored in the 1 st location of array
2) If a node is in the j^{th} location of array, then its left child is in the location 2J+1 and its right child in the location 2J+2

The maximum size that is required for an array to store a tree is $2^{d+1} - 1$, where d is the depth of the tree.



POSITION	ARRAY
0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H
8	I
.	.
.	.
.	.
.	.
.	.

Advantages of sequential representation:

The only advantage with this type of representation is that the direct access to any node can be possible and finding the parent or left children of any particular node is fast because of the random access.

Disadvantages of sequential representation:

1. The major disadvantage with this type of representation is wastage of memory. For example in the skewed tree half of the array is unutilized.
2. In this type of representation the maximum depth of the tree has to be fixed. Because we have decide the array size. If we choose the array size quite larger than the depth of the tree, then it will be wastage of the memory. And if we coose array size lesser than the depth of the tree then we will be unable to represent some part of the tree.
3. The insertions and deletion of any node in the tree will be costlier as other nodes has to be adjusted at appropriate positions so that the meaning of binary tree can be preserved.

As these drawbacks are there with this sequential type of representation, we will search for more flexible representation. So instead of array we will make use of linked list to represent the tree.

b) Linked Representation

Linked representation of trees in memory is implemented using pointers. Since each node in a binary tree can have maximum two children, a node in a linked representation has two pointers for both left and right child, and one information field. If a node does not have any child, the corresponding pointer field is made NULL pointer.

In linked list each node will look like this:

Left Child	Data	Right Child

Advantages of linked representation:

1. This representation is superior to our array representation as there is no wastage of memory. And so there is no need to have prior knowledge of depth of the tree. Using dynamic memory concept one can create as much memory(nodes) as required. By chance if some nodes are unutilized one can delete the nodes by making the address free.

2. Insertions and deletions which are the most common operations can be done without moving the nodes.

Disadvantages of linked representation:

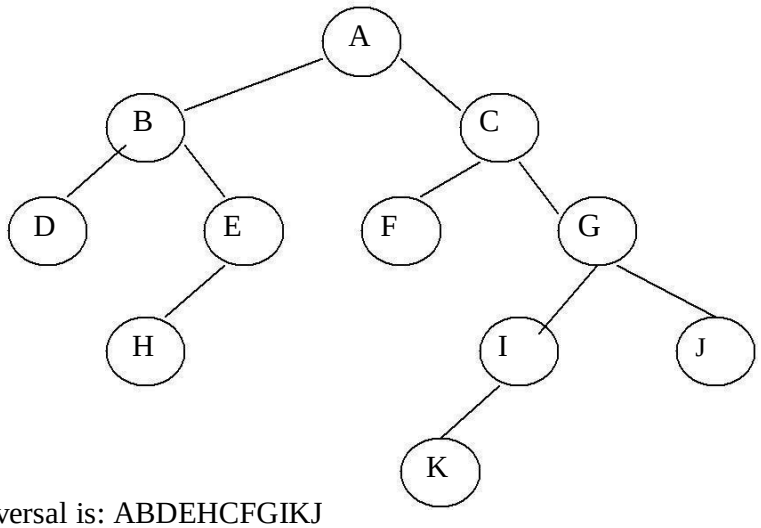
1. This representation does not provide direct access to a node and special algorithms are required.
2. This representation needs additional space in each node for storing the left and right sub-trees.

TRAVERSING A BINARY TREE

Traversing a tree means that processing it so that each node is visited exactly once. A binary tree can be
traversed a number of ways.The most common tree traversals are

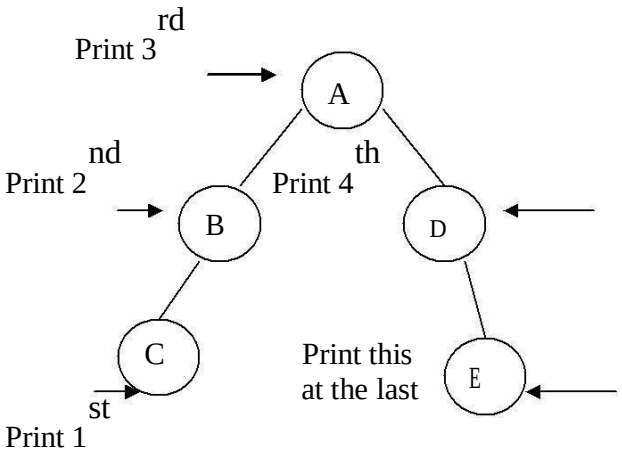
- In-order
- Pre-order and
- Post-order

Pre-order	1.Visit the root 2.Traverse the left sub tree in pre-order 3.Traverse the right sub tree in pre-order.	Root | Left | Right
In-order	1.Traverse the left sub tree in in-order 2.Visit the root 3.Traverse the right sub tree in in-order.	Left | Root | Right
Post-order	1.Traverse the left sub tree in post-order 2.Traverse the right sub tree in post-order. 3.Visit the root	Left | Right | Root



The pre-order traversal is: ABDEHCFGIKJ
The in-order traversal is : DBHEAFCKIGJ
The post-order traversal is:DHEBFKIJGCA

Inorder Traversal:

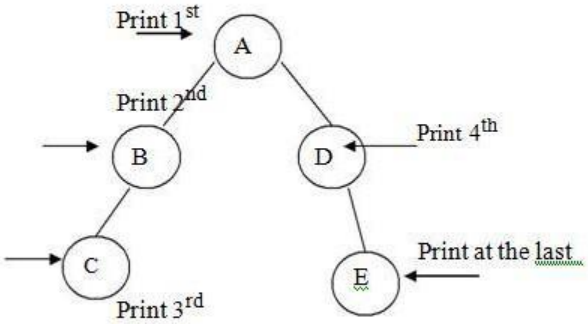


C-B-A-D-E is the inorder traversal i.e. first we go towards the leftmost node. i.e. C so print that node C. Then go back to the node B and print B. Then root node A then move towards the right sub-tree print D and finally E. Thus we are following the tracing sequence of Left|Root|Right. This type of traversal is called inorder traversal. The basic principle is to traverse left sub-tree then root and then the right sub-tree.

Pseudo Code:

```
template <class T>
void inorder(bintree<T> *temp)
{
        if(temp!=NULL)
        {
                inorder(temp->left);
                cout<<‖temp->data‖;
                inorder(temp->right);
        }
}
```

Preorder Traversal:

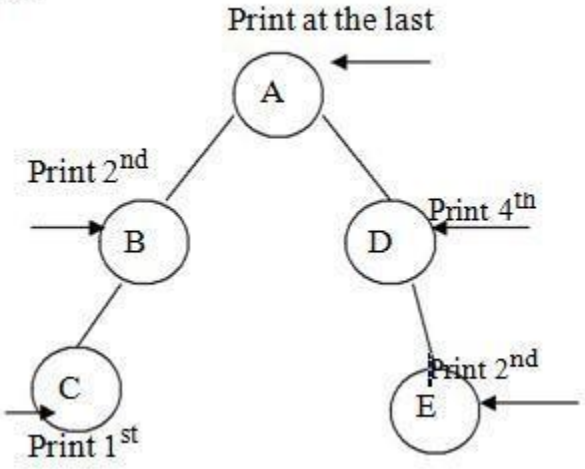


is the preorder traversal of the above fig. We are following Root|Left|Right path i.e. data at the root node will be printed first then we move on the left sub-tree and go on printing the data till we reach to the left most node. Print the data at that node and then move to the right sub- tree. Follow the same principle at each sub-tree and go on printing the data accordingly.

```
template <class T>
void preorder(bintree<T> *temp)
```

```
{
        if(temp!=NULL)
        {
                cout<<‖temp->data‖; preorder(temp->left);
                preorder(temp->right);
        }
}
```

Postorder Traversal:

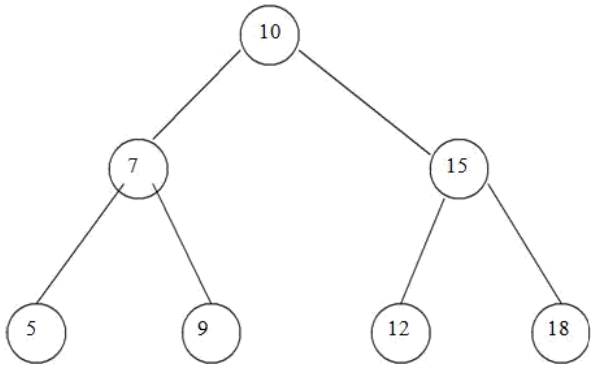


From figure the postorder traversal is C-D-B-E-A. In the postorder traversal we are following the Left|Right|Root principle i.e. move to the leftmost node, if right sub-tree is there or not if not then print the leftmost node, if right sub-tree is there move towards the right most node. The key idea here is that at each sub-tree we are following the Left|Right|Root principle and print the data accordingly.
Pseudo Code:

```
template <class T>
void postorder(bintree<T> *temp)
{
        if(temp!=NULL)
        {
                postorder(temp->left);
                postorder(temp->right);
                cout<<‖temp->data‖;
        }
}
```

BINARY SEARCH TREE

In the simple binary tree the nodes are arranged in any fashion. Depending on user‘s desire the new nodes can be attached as a left or right child of any desired node. In such a case finding for any node is a long cut procedure, because in that case we have to search the entire tree. And thus the searching time complexity will get increased unnecessarily. So to make the searching algorithm faster in a binary tree we will go for building the binary search tree. The binary search tree is based on the binary search algorithm. While creating the binary search tree the data is systematically arranged. That means values at **left sub-tree < root node value < right sub-tree values.**



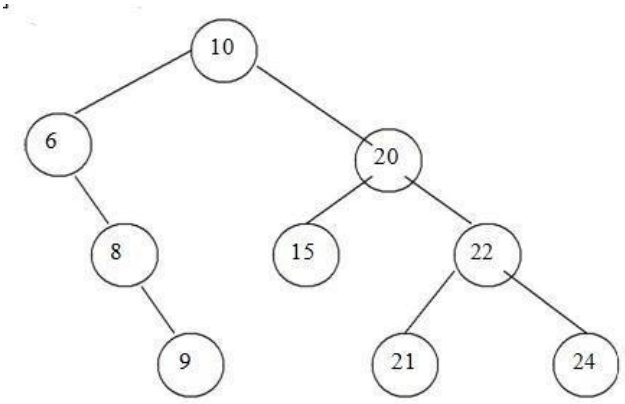
Operations On Binary Search Tree:

The basic operations which can be performed on binary search tree are.

1. Insertion of a node in binary search tree.
2. Deletion of a node from binary search tree.
3. Searching for a particular node in binary search tree.

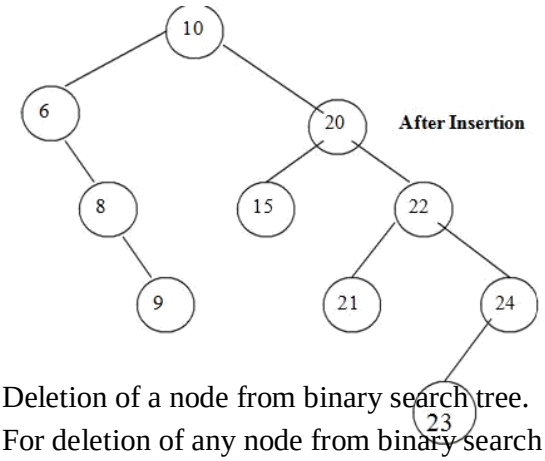
Insertion of a node in binary search tree.

While inserting any node in binary search tree, look for its appropriate position in the binary search tree. We start comparing this new node with each node of the tree. If the value of the node which is to be inserted is greater than the value of the current node we move on to the right sub-branch otherwise we move on to the left sub-branch. As soon as the appropriate position is found we attach this new node as left or right child appropriately.



Before Insertion

In the above fig, if we wan to insert 23. Then we will start comparing 23 with value of root node i.e. 10. As 23 is greater than 10, we will move on right sub-tree. Now we will compare 23 with 20 and move right, compare 23 with 22 and move right. Now compare 23 with 24 but it is less than 24. We will move on left branch of 24. But as there is node as left child of 24, we can attach 23 as left child of 24.



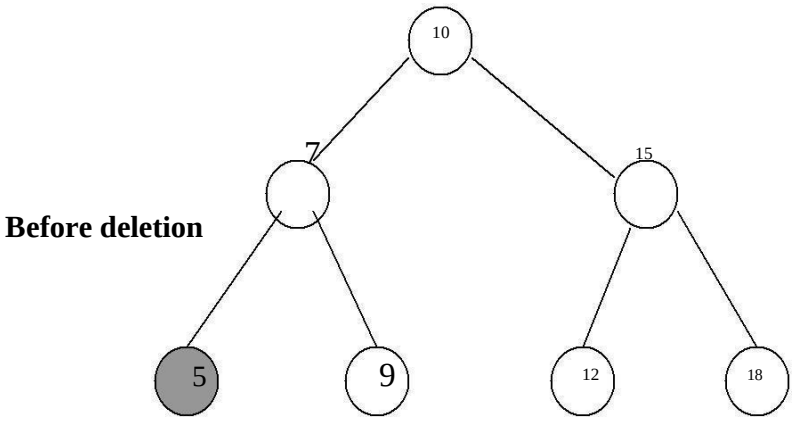
Deletion of a node from binary search tree.

For deletion of any node from binary search tree there are three which are possible.

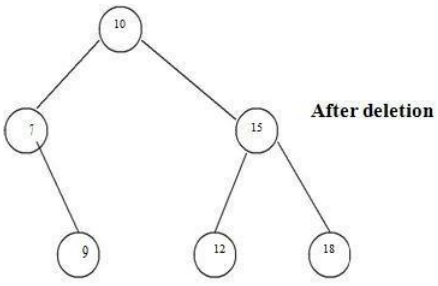
i. Deletion of leaf node.
ii. Deletion of a node having one child.
iii. Deletion of a node having two children.

Deletion of leaf node.

This is the simplest deletion, in which we set the left or right pointer of parent node as NULL.

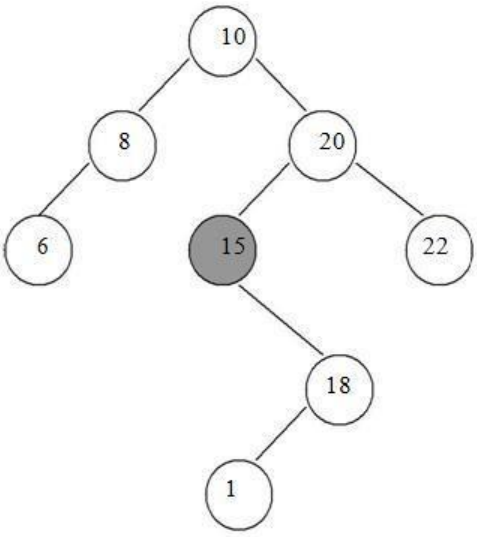


From the above fig, we want to delete the node having value 5 then we will set left pointer of its parent node as NULL. That is left pointer of node having value 7 is set to NULL.

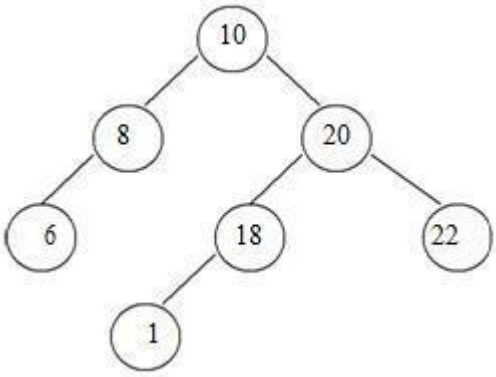


Deletion of a node having one child.

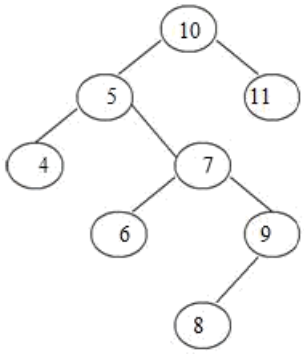
To explain this kind of deletion, consider a tree as given below.



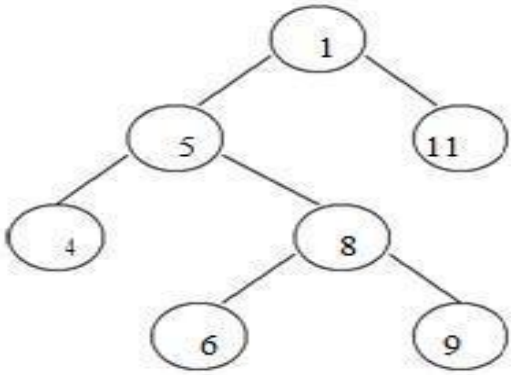
If we want to delete the node 15, then we will simply copy node 18 at place of 16 and then set the node free



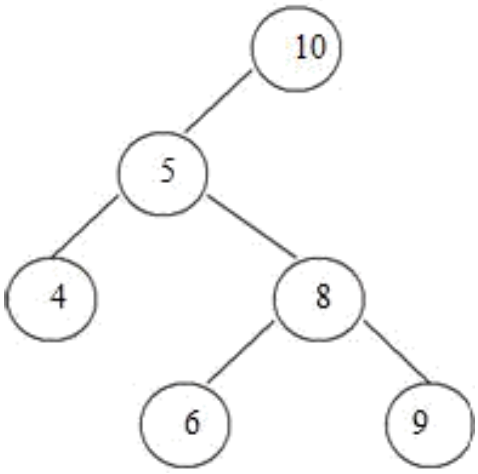
Deletion of a node having two children.
Consider a tree as given below.



Let us consider that we want to delete node having value 7. We will then find out the inorder successor of node 7. We will then find out the inorder successor of node 7. The inorder successor will be simply copied at location of node 7.
That means copy 8 at the position where value of node is 7. Set left pointer of 9 as NULL. This completes the deletion procedure.



Searching for a node in binary search tree.
In searching, the node which we want to search is called a key node. The key node will be compared with each node starting from root node if value of key node is greater than current node then we search for it on right sub branch otherwise on left sub branch. If we reach to leaf node and still we do not get the value of key node then we declare –node is not present in the tree‖.



In the above tree, if we want to search for value 9. Then we will compare 9 with root node 10. As 9 is less than 10 we will search on left sub branch. Now compare 9 with 5, but 9 is greater than 5. So we will move on right sub tree. Now compare 9 with 8 but 9 is greater than 8 we will move on right sub branch. As the node we will get holds the value 9. Thus the desired node can be searched.

AVL TREES

Adelsion Velski and Lendis in 1962 introduced binary tree structure that is balanced with respect to height of sub trees. The tree can be made balanced and because of this retrieval of any node can be done in O(log n) times, where n is total number of nodes. From the name of these scientists the tree is called AVL tree.

Definition:

An empty tree is height balanced if T is a non empty binary tree with T_L and T_R as its left and right sub trees. The T is height balanced if and only if

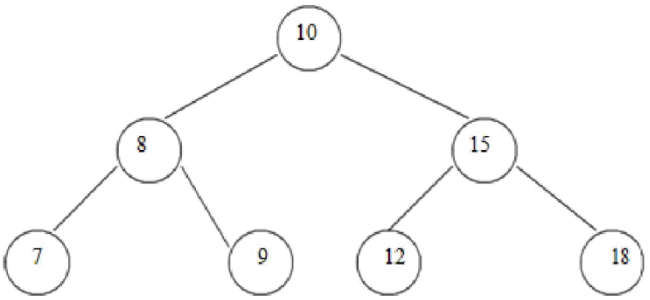
i. T_L and T_R are height balanced.
ii. h_L-h_R <= 1 where h_L and h_R are heights of T_L and T_R.

The idea of balancing a tree is obtained by calculating the balance factor of a tree.

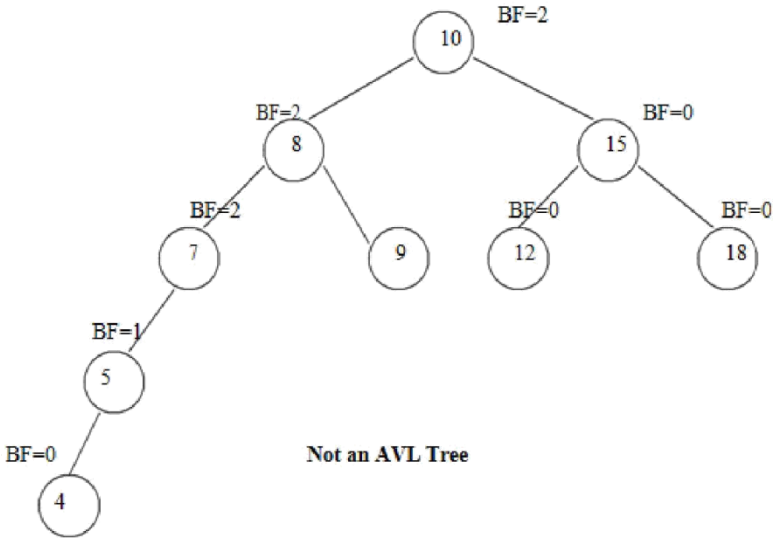
Definition of Balance Factor:

The balance factor BF(T) of a node in binary tree is defined to be h_L-h_R where h_L and h_R are heights of left and right sub trees of T.

For any node in AVL tree the balance factor i.e. BF(T) **is -1, 0 or +1**.



AVL Tree



Not an AVL Tree

Height of AVL Tree:

Theorem: The height of AVL tree with n elements (nodes) is O(log n).

Proof: Let an AVL tree with n nodes in it. Nh be the minimum number of nodes in an AVL tree of height h.

In worst case, one sub tree may have height h-1 and other sub tree may have height h-2. And both these sub trees are AVL trees. Since for every node in AVL tree the height of left and right sub trees differ by at most 1.

Hence

$$N_h = N_{h-1} + N_{h-2} + 1$$

Where N_h denotes the minimum number of nodes in an AVL tree of height h.

$N_0=0$ $N_1=2$

We can also write it as

$N > N_h = N_{h-1}+N_{h-2}+1$

$> 2N_{h-2}$

$> 4N_{h-4}$

.
.

$> 2iNh-2i$

If value of h is even, let i = h/2-1

Then equation becomes

N > 2h/2-1N2

= N > 2(h-1)/2x4 (N2 = 4)

= O(log N)

If value of h is odd, let I = (h-1)/2 then equation becomes

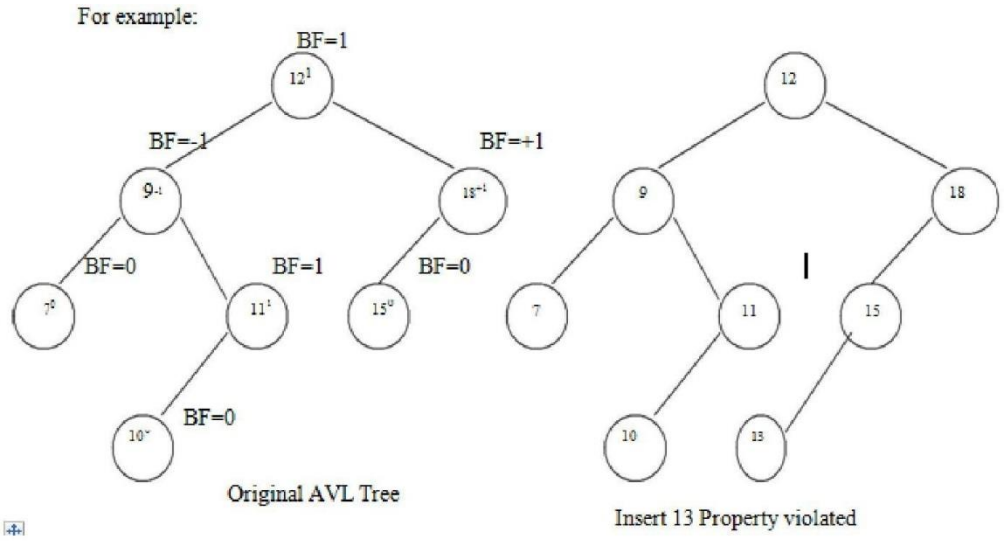
N > 2(h-1)/2 N1
N > 2(h-1)/2 x 1
H = O(log N)

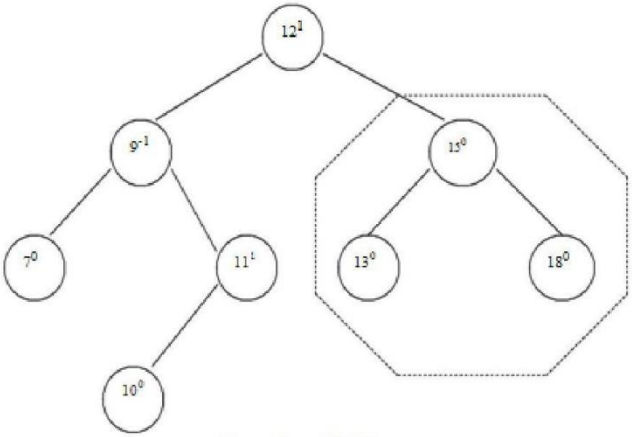
This proves that height of AVL tree is always O(log N). Hence search, insertion and deletion can be carried out in logarithmic time.

Representation of AVL Tree

The AVL tree follows the property of binary search tree. In fact AVL trees are basically binary search trees with balance factors as -1, 0, or +1.

After insertion of any node in an AVL tree if the balance factor of any node becomes other than -1, 0, or +1 then it is said that AVL property is violated. Then we have to restore the destroyed balance condition. The balance factor is denoted at right top corner inside the node.

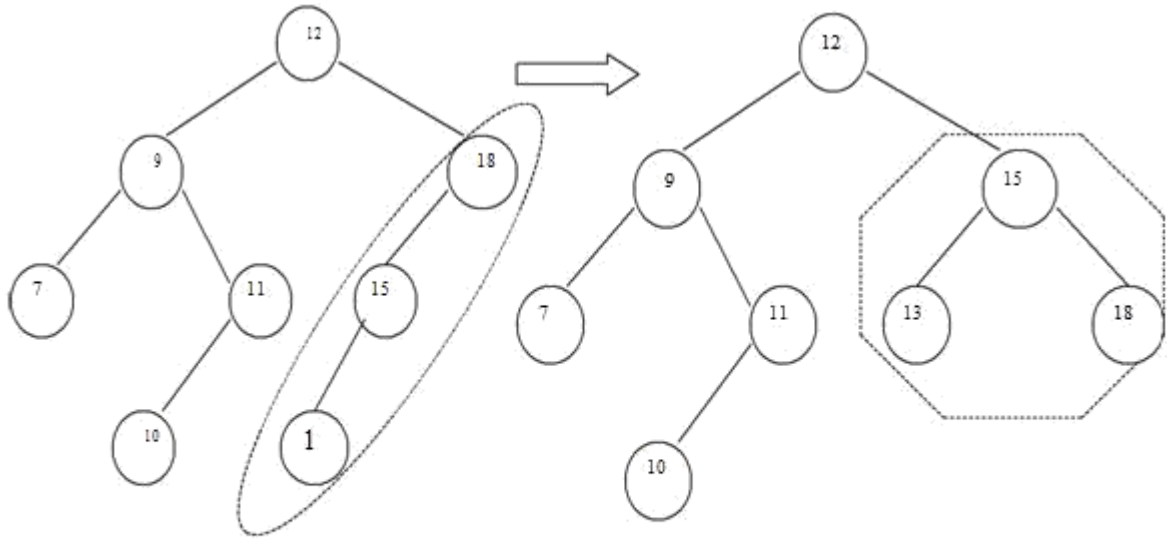




After insertion of a new node if balance condition gets destroyed, then the nodes on that path(new node insertion point to root) needs to be readjusted. That means only the affected sub tree is to be rebalanced.

The rebalancing should be such that entire tree should satisfy AVL property.

In above given example-



Nodes 18, 15, 13 are to be adjusted

By adjusting 15 the entire Tree satisfies AVL property

Insertion of a node.

There are four different cases when rebalancing is required after insertion of new node.

1. An insertion of new node into left sub tree of left child. (LL).
2. An insertion of new node into right sub tree of left child. (LR).
3. An insertion of new node into left sub tree of right child. (RL).
4. An insertion of new node into right sub tree of right child.(RR).

Some modifications done on AVL tree in order to rebalance it is called rotations of AVL tree

There are two types of rotations:

Single rotation
- Left-Left(LL rotation)
- Right-Right(RR rotation)

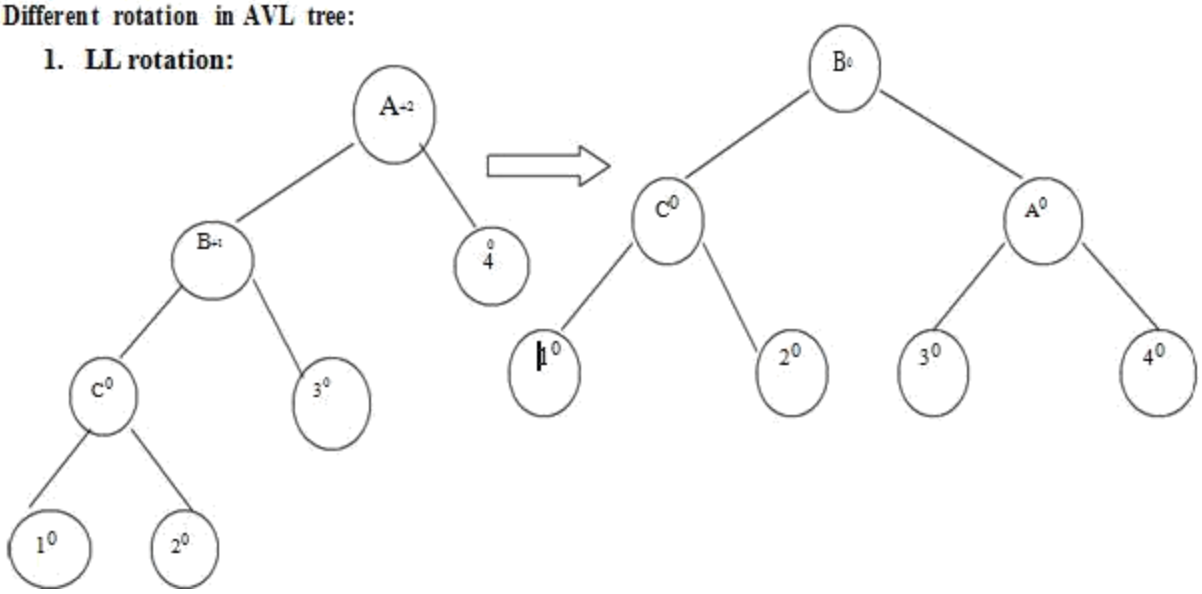
Double rotation
- Left-Right(LR rotation)
- Right-Left(RL rotation)

Insertion Algorithm:

1.Insert a new node as new leaf just as an ordinary binary search tree.
2.Now trace the path from insertion point(new node inserted as leaf) towards root. For each node _n' encountered, check if heights of left (n) and right (n) differ by at most 1. a)If yes, move towards parent (n).
b)Otherwise restructure by doing either a single rotation or a double rotation.
Thus once we perform a rotation at node _n' we do not require to perform any rotation at any ancestor on _n'.

Different rotation in AVL tree:

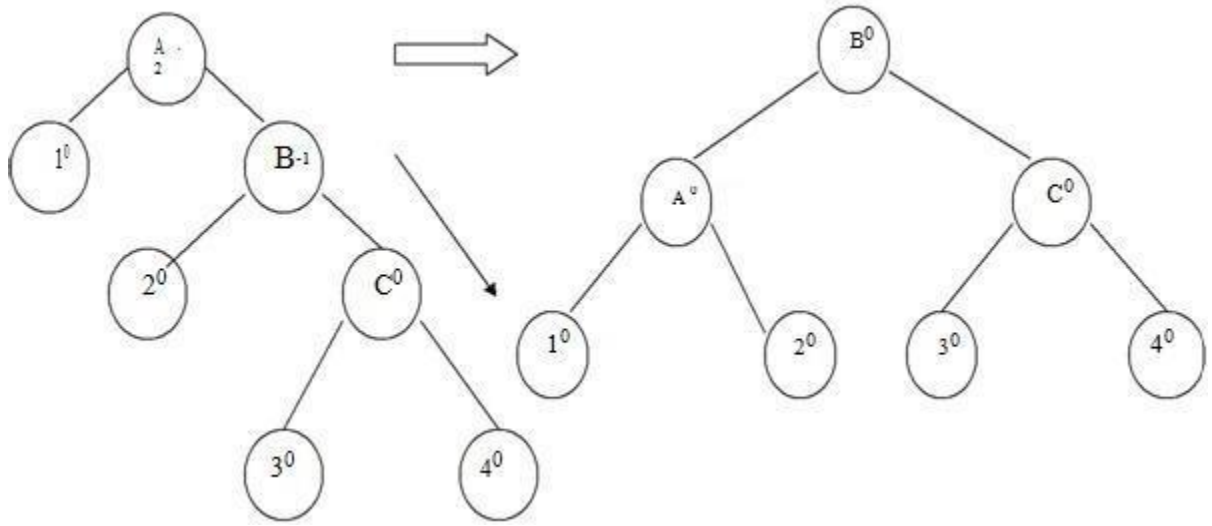
1. LL rotation:



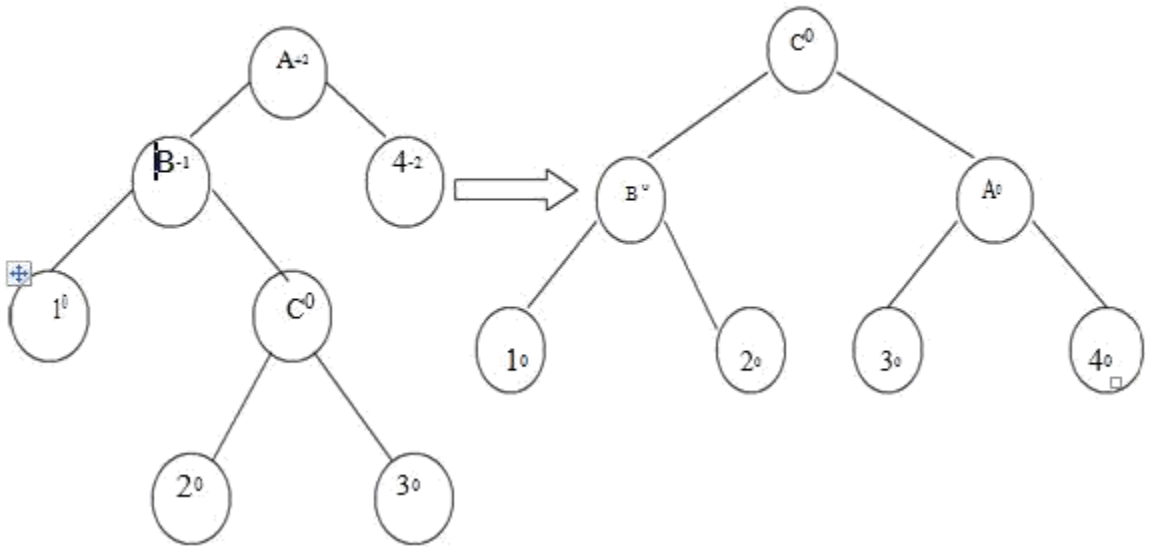
When node _1' gets inserted as a left child of node _C' then AVL property gets destroyed i.e. node A has balance factor +2.
The LL rotation has to be applied to rebalance the nodes.

2. RR rotation:

When node _4' gets attached as right child of node _C' then node _A' gets unbalanced. The rotation which needs to be applied is RR rotation as shown in fig.

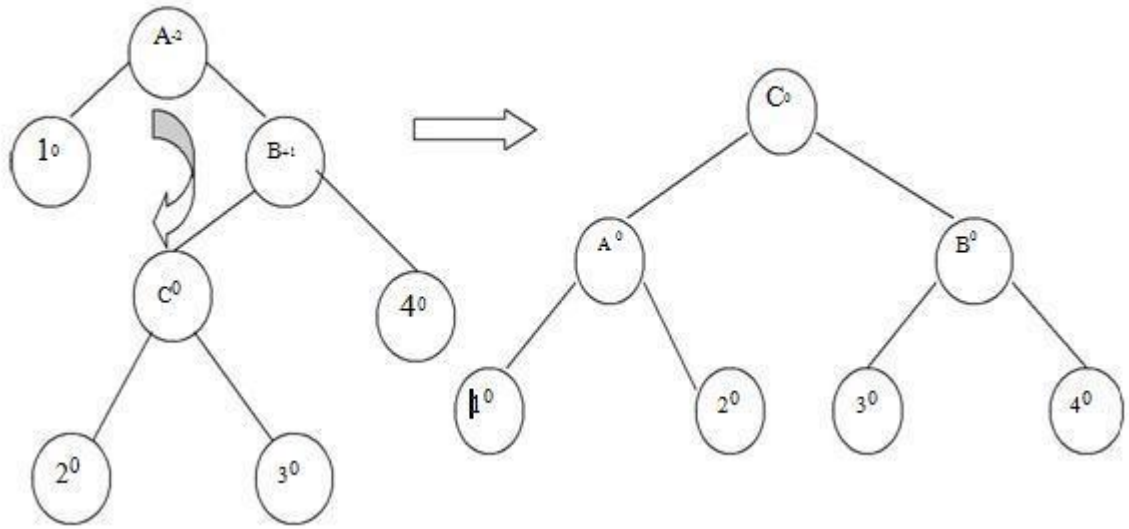


3. LR rotation:



When node _3' is attached as a right child of node _C' then unbalancing occurs because of LR. Hence LR rotation needs to be applied.

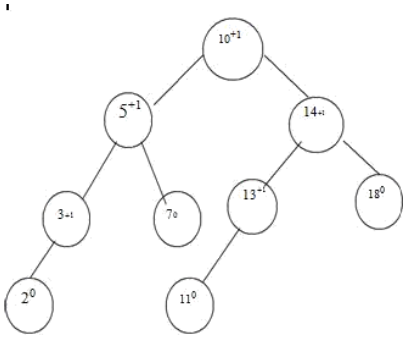
4. RL rotation



When node _2' is attached as a left child of node _C' then node _A' gets unbalanced as its balance factor becomes -2. Then RL rotation needs to be applied to rebalance the AVL tree.

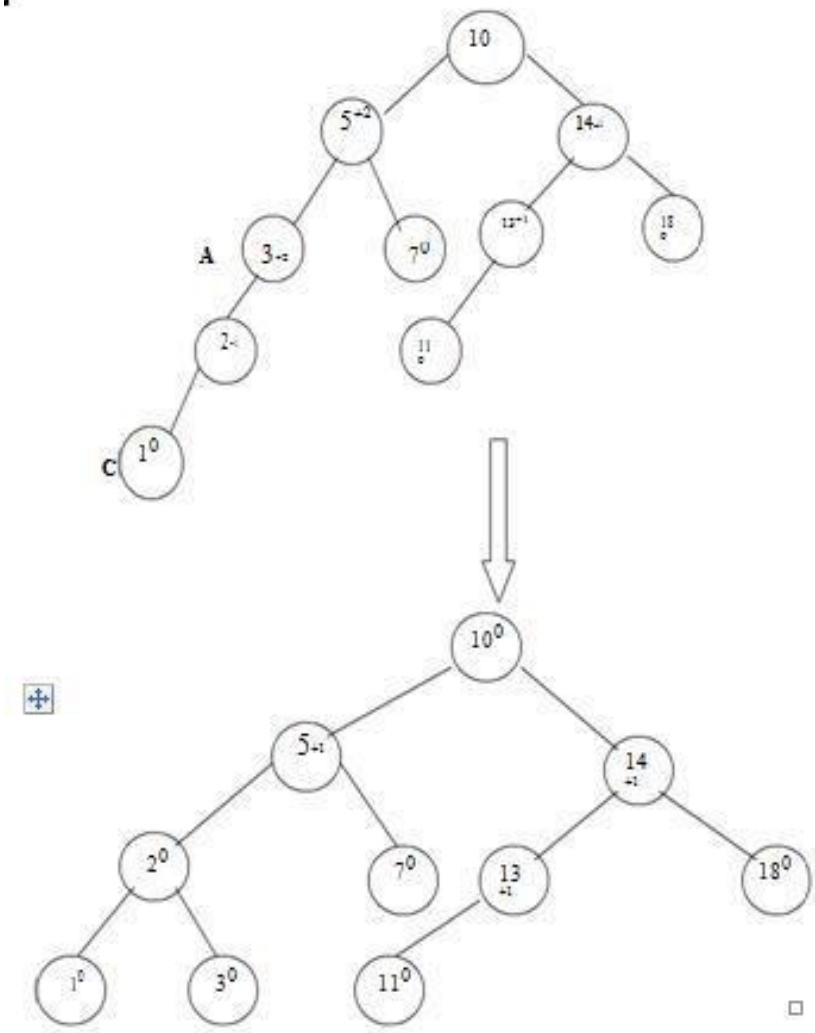
Example:

Insert 1, 25, 28, 12 in the following AVL tree.



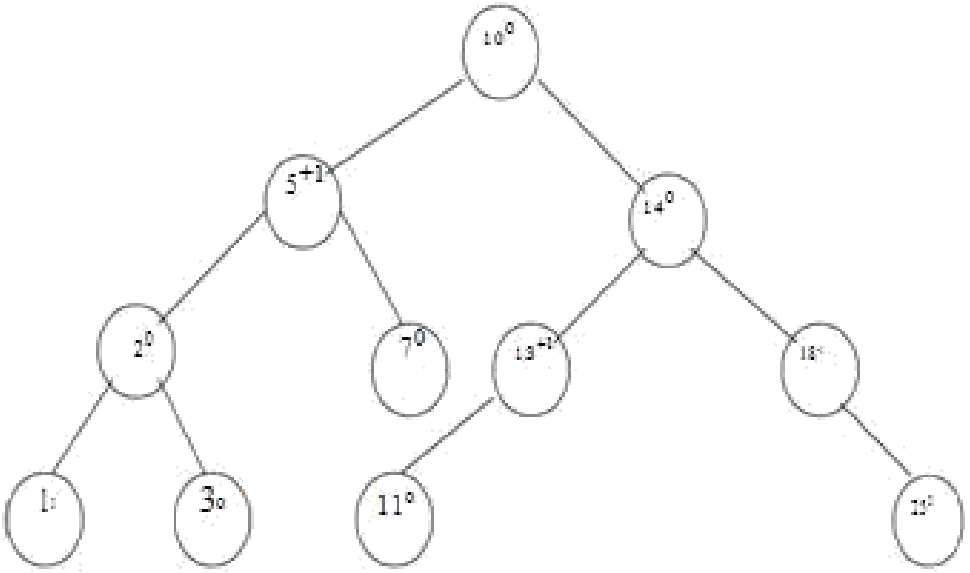
Insert 1

To insert node _1‘ we have to attach it as a left child of _2‘. This will unbalance the tree as follows. We will apply LL rotation to preserve AVL property of it.



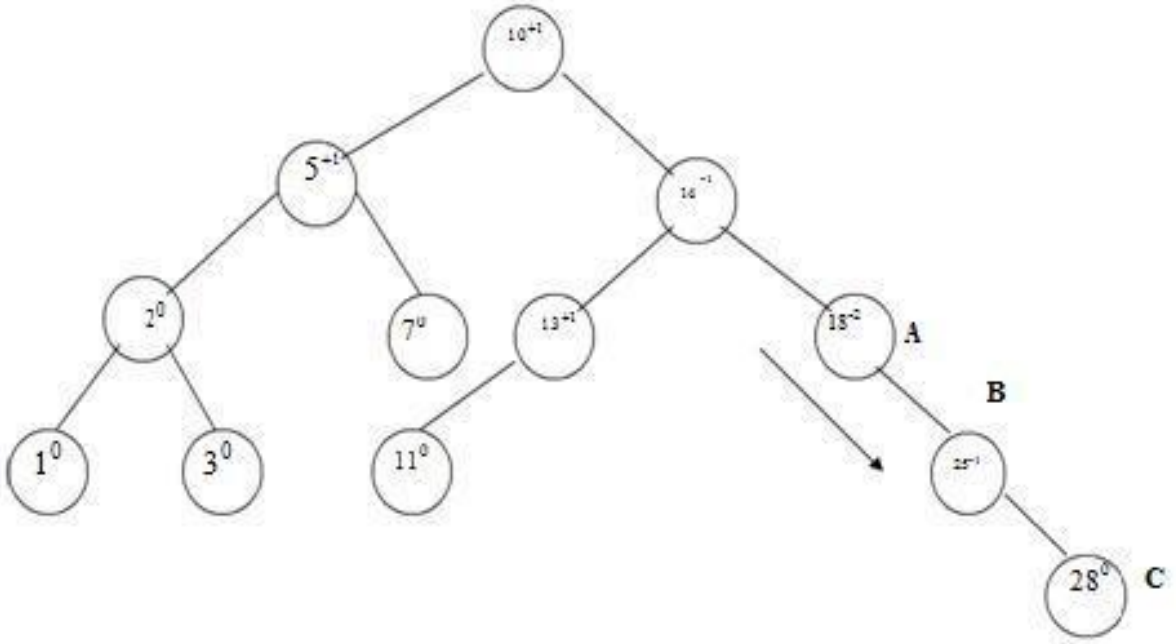
Insert 25

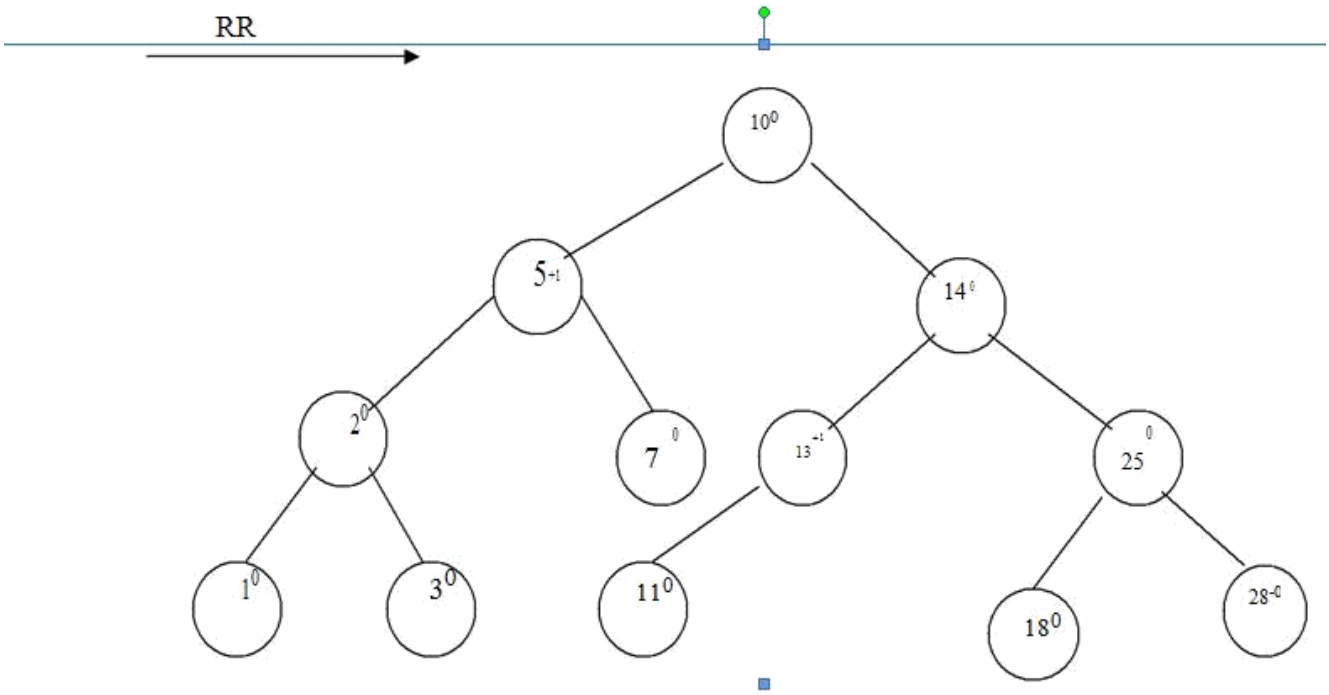
We will attach 25 as a right child of 18. No balancing is required as entire tree preserves the AVL property



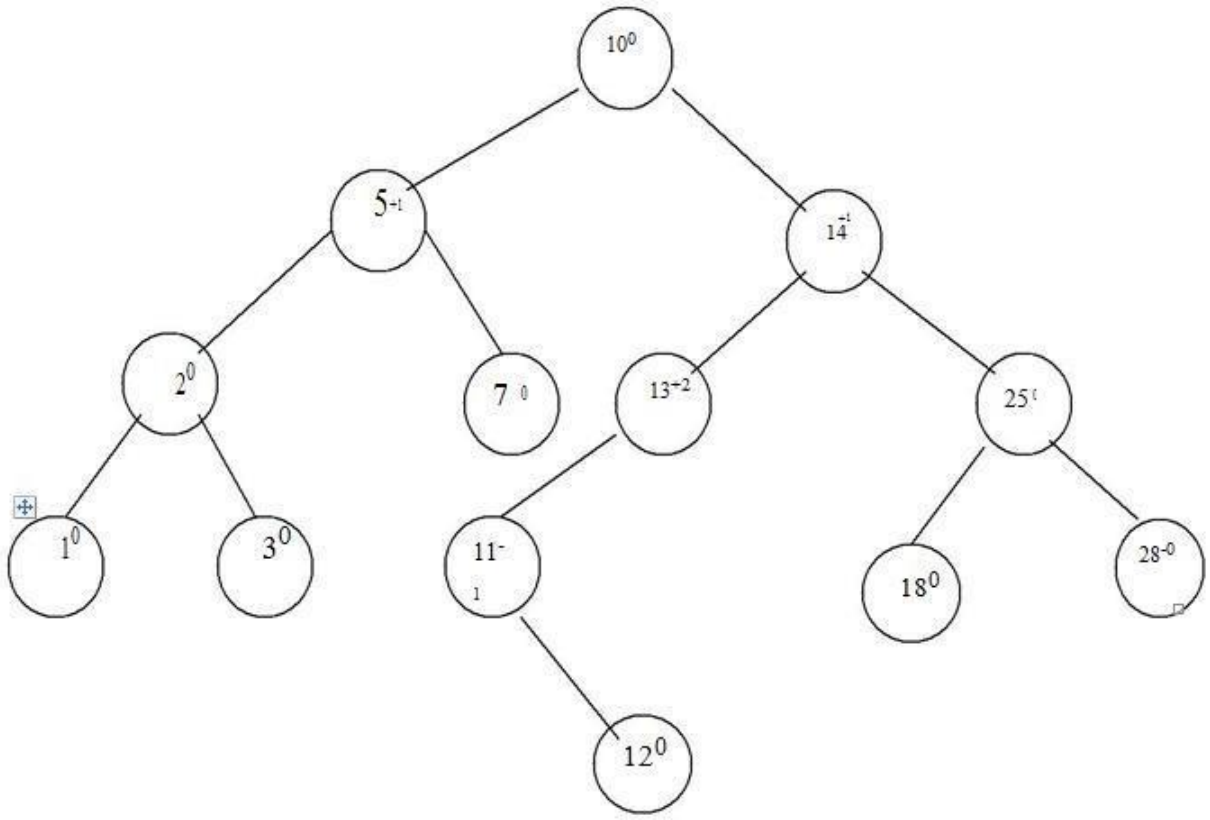
Insert 28

The node ‗28' is attached as a right child of 25. RR rotation is required to rebalance.

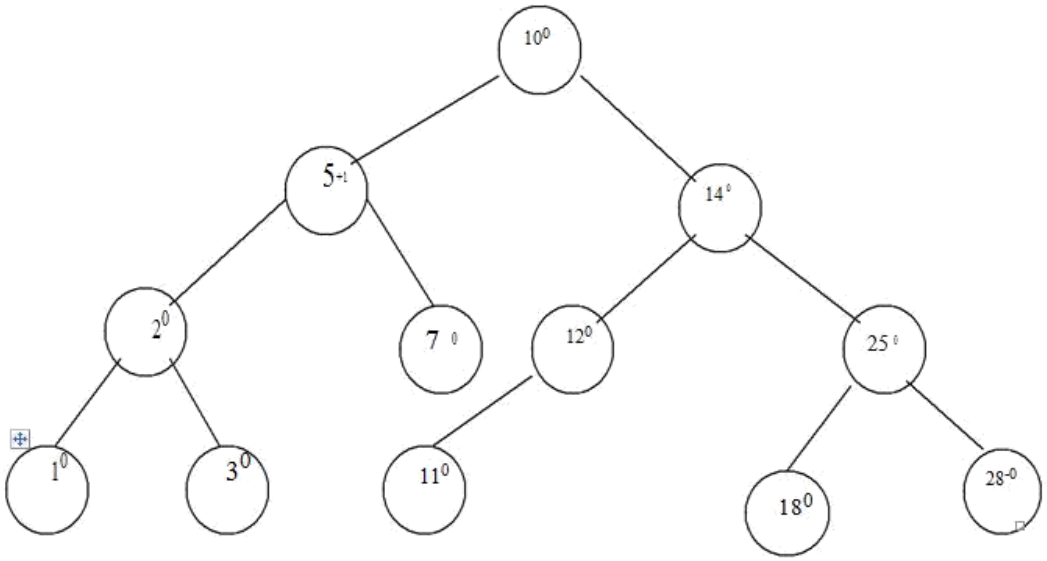




Insert 12



To rebalance the tree we have to apply LR rotation.



Thus by applying various rotations depending upon direction of insertion of new node the AVL tree can be restructured.

Deletion:

Even after deletion of any particular node from AVL tree, the tree has to be restructured in order to preserve AVL property. And thereby various rotations need to be applied.

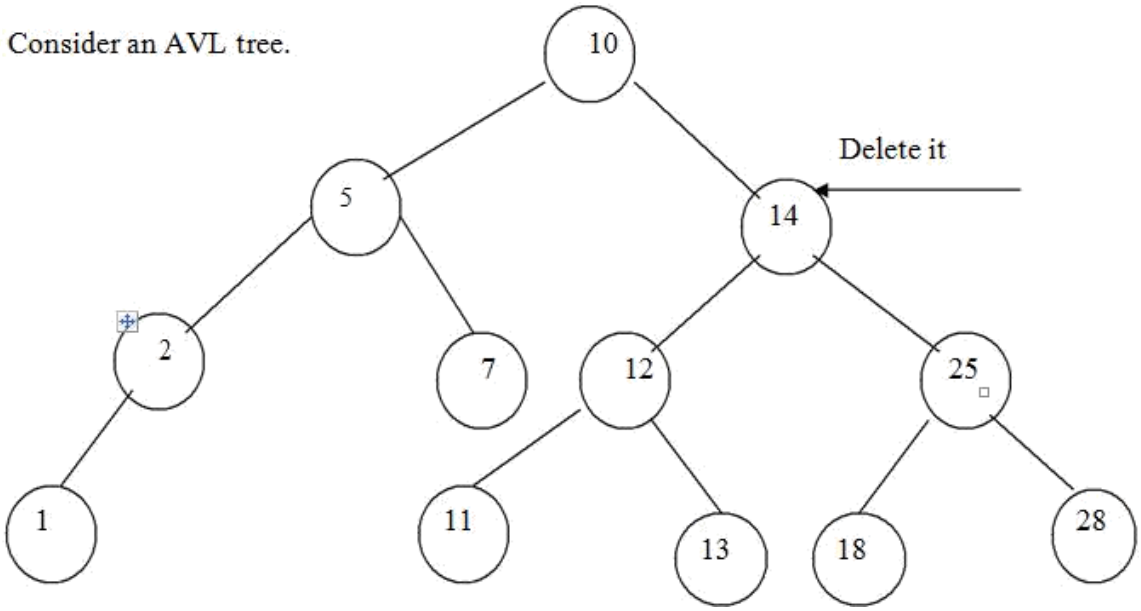
Algorithm for deletion:

The deletion algorithm is more complex than insertion algorithm.

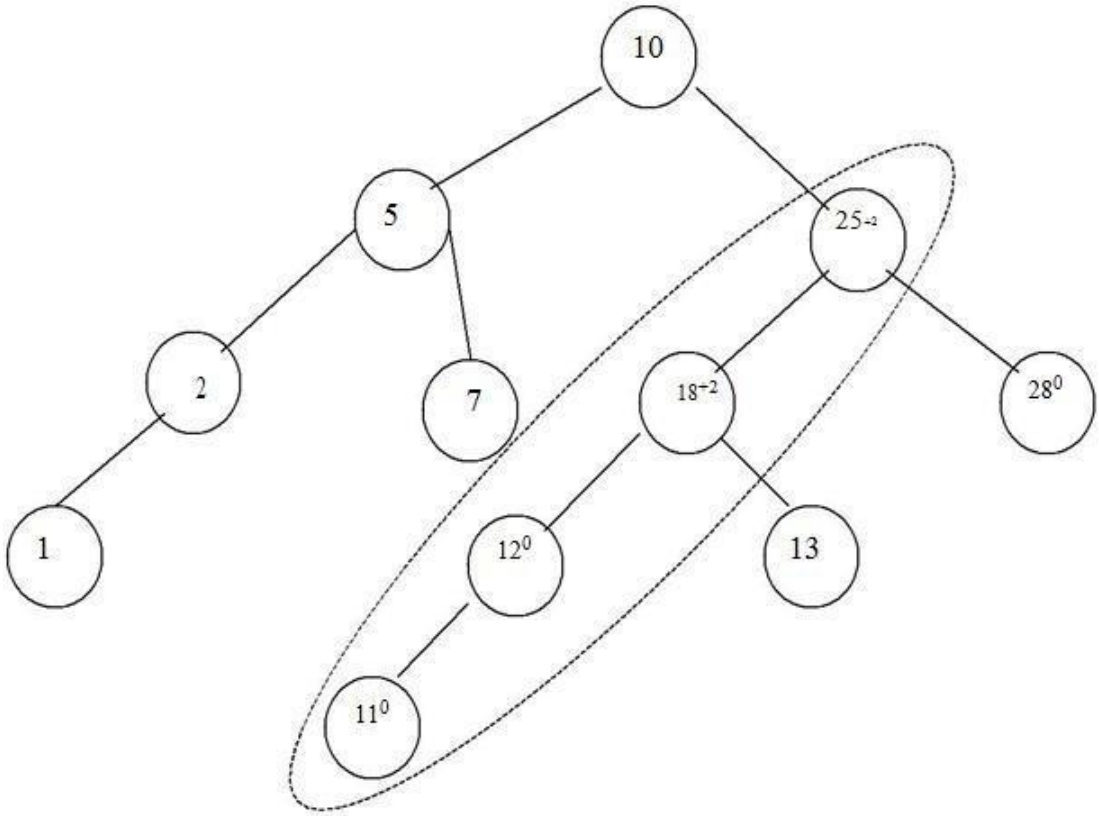
1. Search the node which is to be deleted.

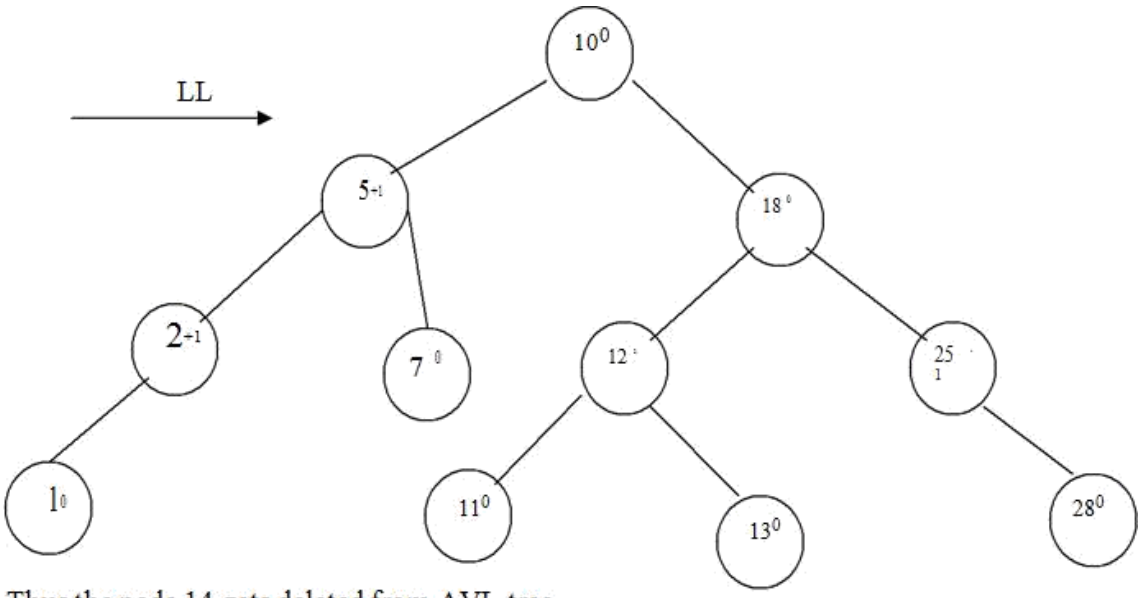
2. a) If the node to be deleted is a leaf node then simply make it NULL to remove.
   b) If the node to be deleted is not a leaf node i.e. node may have one or two children, then the node must be swapped with its in order successor. Once the node is swapped, we can remove this node.

3. Now we have to traverse back up the path towards root, checking the balance factor of every node along the path. If we encounter unbalancing in some sub tree
   then balance that sub tree using appropriate single or double rotations. The deletion algorithm takes O(log n) time to delete any node.



The tree becomes





Thus the node 14 gets deleted from AVL tree.

Searching:

The searching of a node in an AVL tree is very simple. As AVL tree is basically binary search tree, the algorithm used for searching a node from binary search tree is the same one is used to search a node from AVL tree.

The searching of a node from AVL tree takes O(log n) time.

BTREES

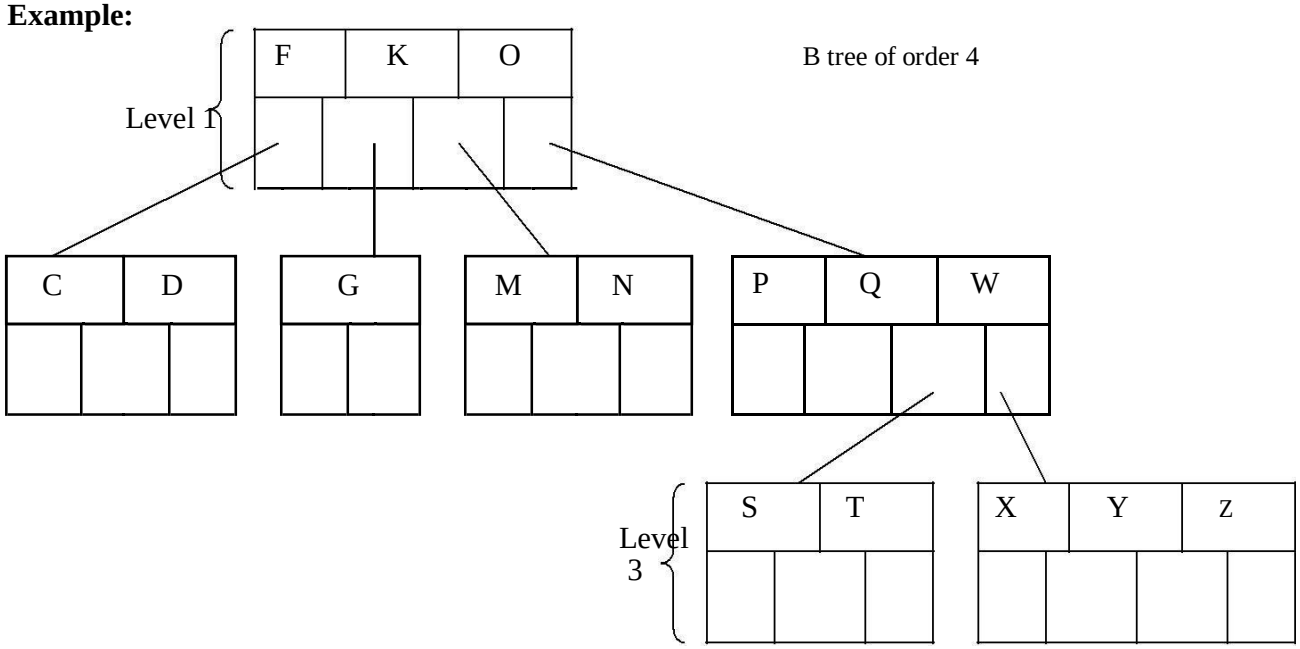
- Multi-way trees are tree data structures with more than two branches at a node. The data structures of m-way search trees, B trees and Tries belong to this category of tree structures.
- AVL search trees are height balanced versions of binary search trees, provide efficient retrievals and storage operations. The complexity of insert, delete and search operations on AVL search trees id O(log n).
- Applications such as File indexing where the entries in an index may be very large, maintaining the index as m-way search trees provides a better option than AVL search trees which are but only balanced binary search trees.
- While binary search trees are two-way search trees, m-way search trees are extended binary search trees and hence provide efficient retrievals.
- B trees are height balanced versions of m-way search trees and they do not recommend representation of keys with varying sizes.
- Tries are tree based data structures that support keys with varying sizes.

Definition:

A B tree of order m is an m-way search tree and hence may be empty. If non empty, then the following properties are satisfied on its extended tree representation:

i. The root node must have at least two child nodes and at most m child nodes.
ii. All internal nodes other than the root node must have at least |m/2 | non empty child nodes and at most m non empty child nodes.
iii. The number of keys in each internal node is one less than its number of child nodes and these keys partition the keys of the tree into sub trees.
iv. All external nodes are at the same level.
v.

Example:

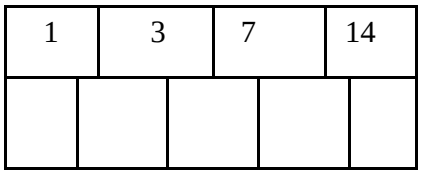


Insertion

For example construct a B-tree of order 5 using following numbers. 3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16, 18, 24, 25, 19

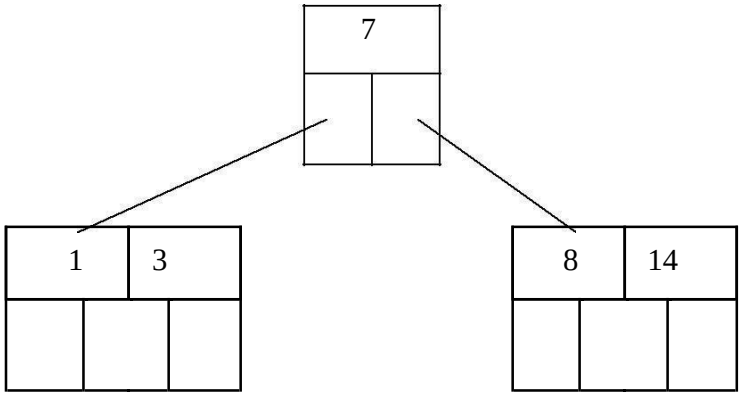
The order 5 means at the most 4 keys are allowed. The internal node should have at least 3 non empty children and each leaf node must contain at least 2 keys.

Step 1: Insert 3, 14, 7, 1

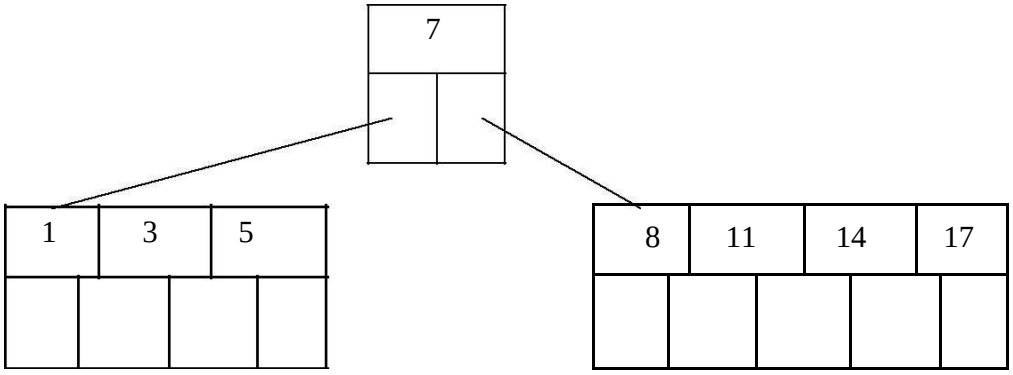


.

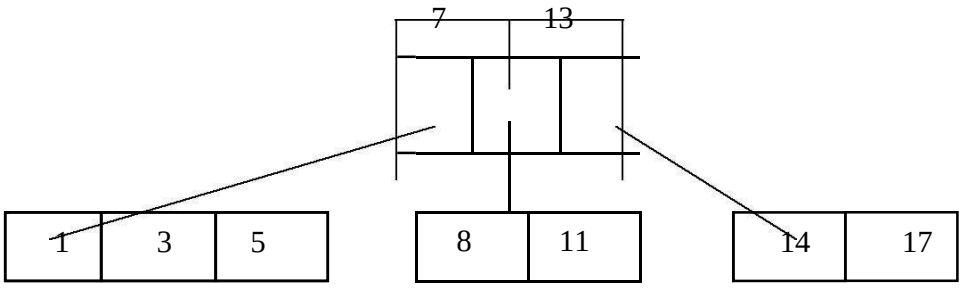
Step 2: Insert 8, Since the node is full split the node at medium 1, 3, 7, 8, 14



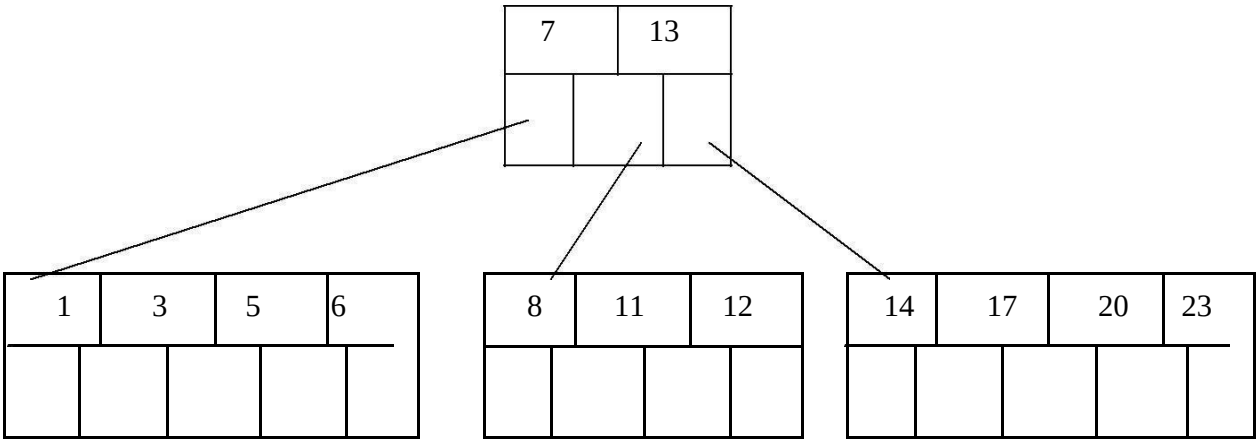
Step 3: Insert 5, 11, 17 which can be easily inserted in a B-tree.



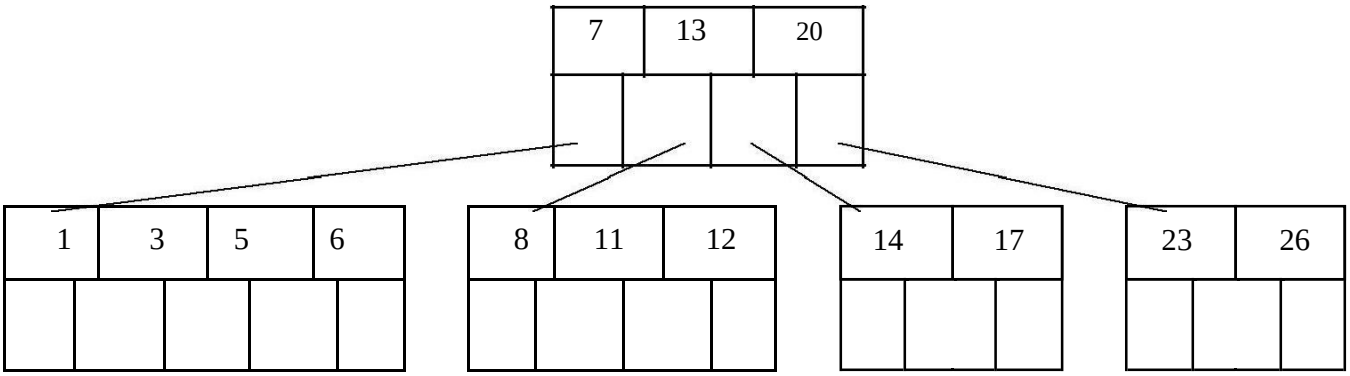
Step 4: Now insert 13. But if we insert 13 then the leaf node will have 5 keys which is not allowed. Hence 8,
11, 13, 14, 17 is split and medium node 13 is moved up.



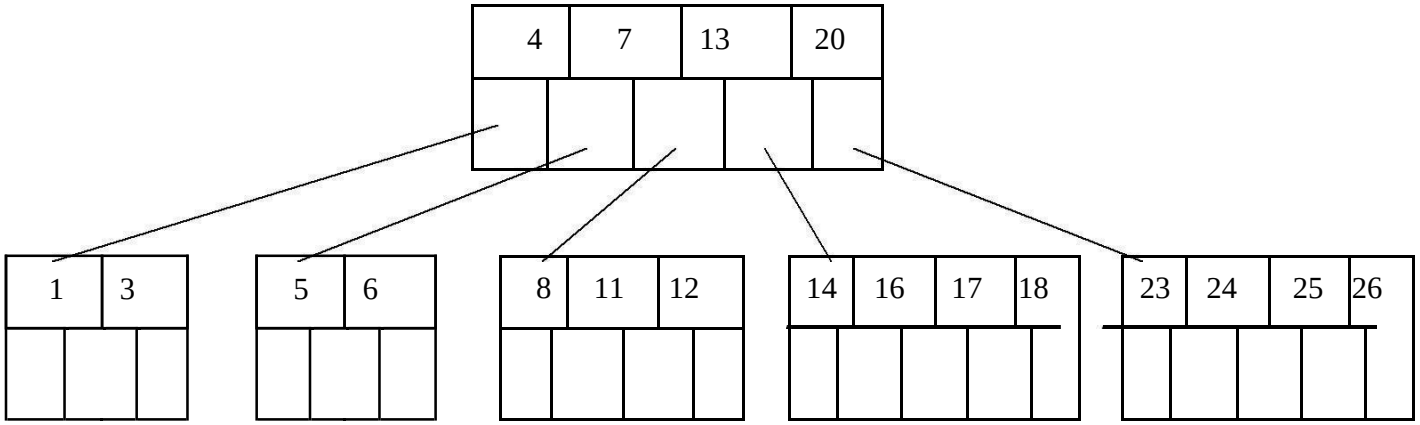
Step 5: Now insert 6, 23, 12, 20 without any split.



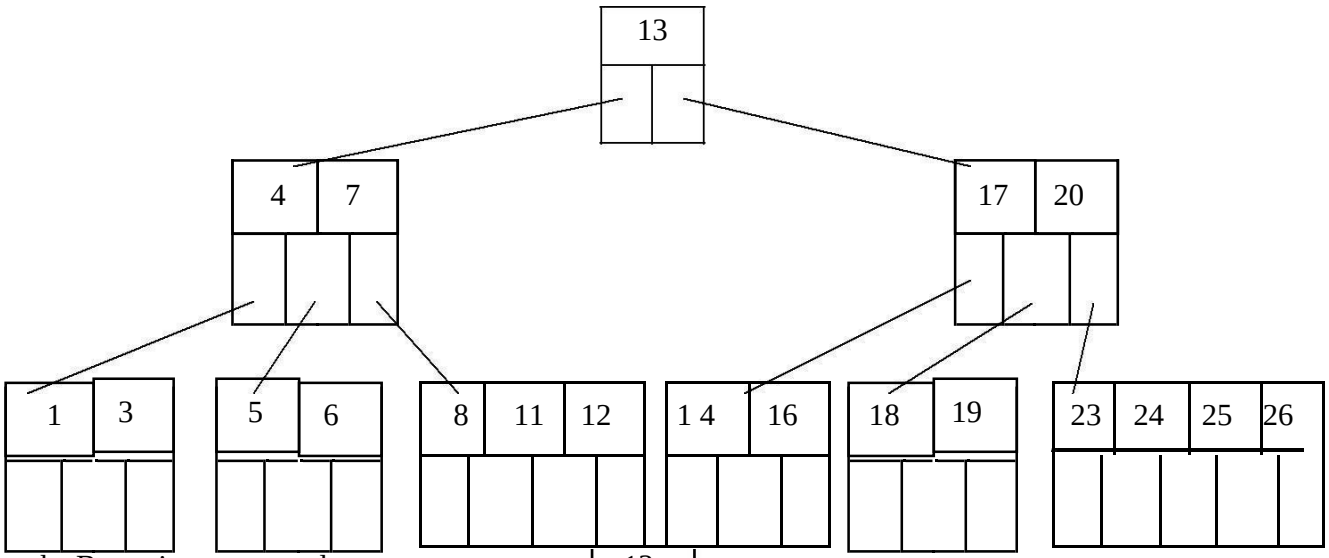
Step 6: The 26 is inserted to the right most leaf node. Hence 14, 17, 20, 23, 26 the node is split and 20 will be moved up.



Step 7: Insertion of node 4 causes left most node to split. The 1, 3, 4, 5, 6 causes key 4 to move up. Then insert 16, 18, 24, 25.



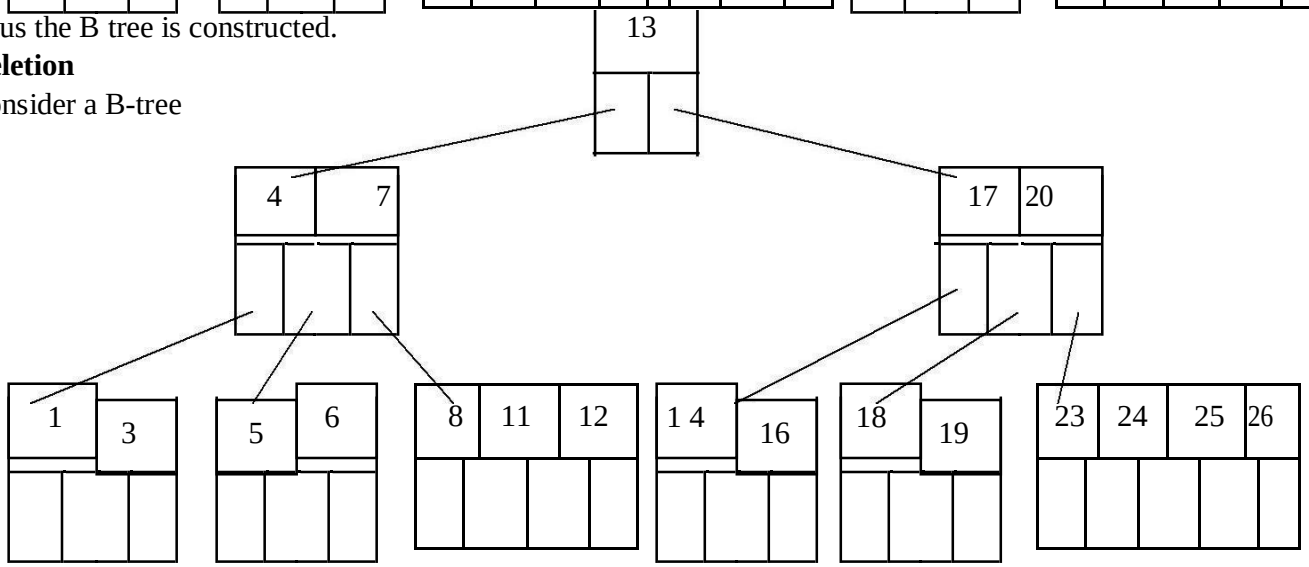
Step 8: Finally insert 19. Then 4, 7, 13, 19, 20 needs to be split. The median 13 will be moved up to from a root node.
The tree then will be -



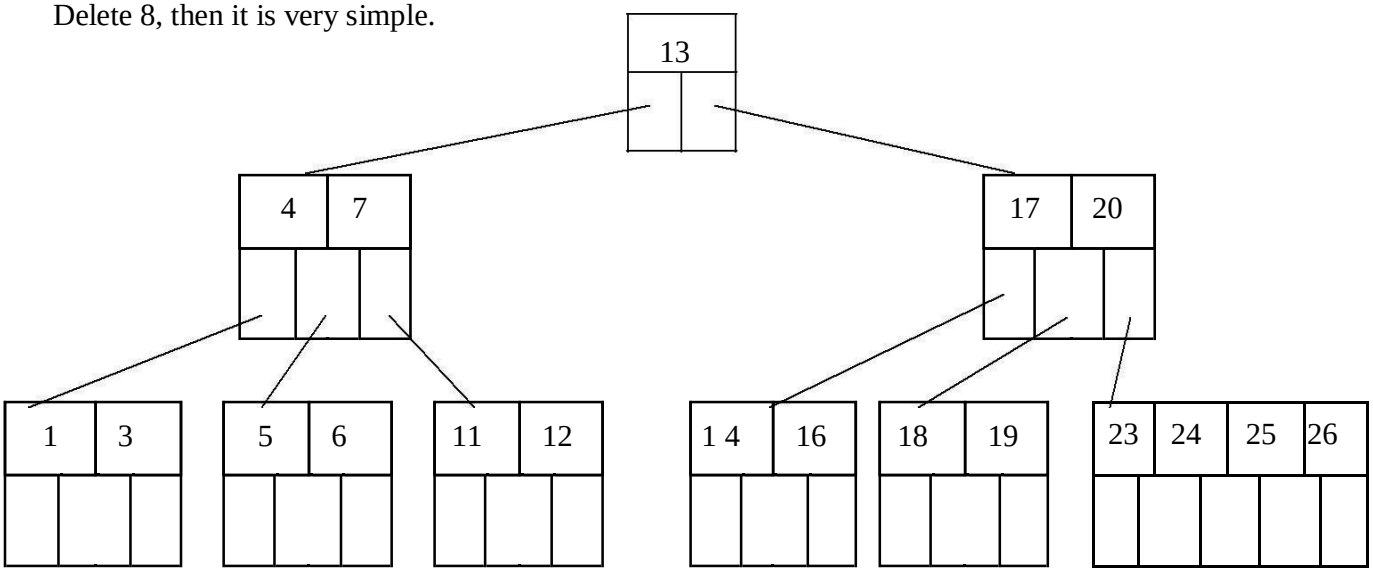
Thus the B tree is constructed.

Deletion

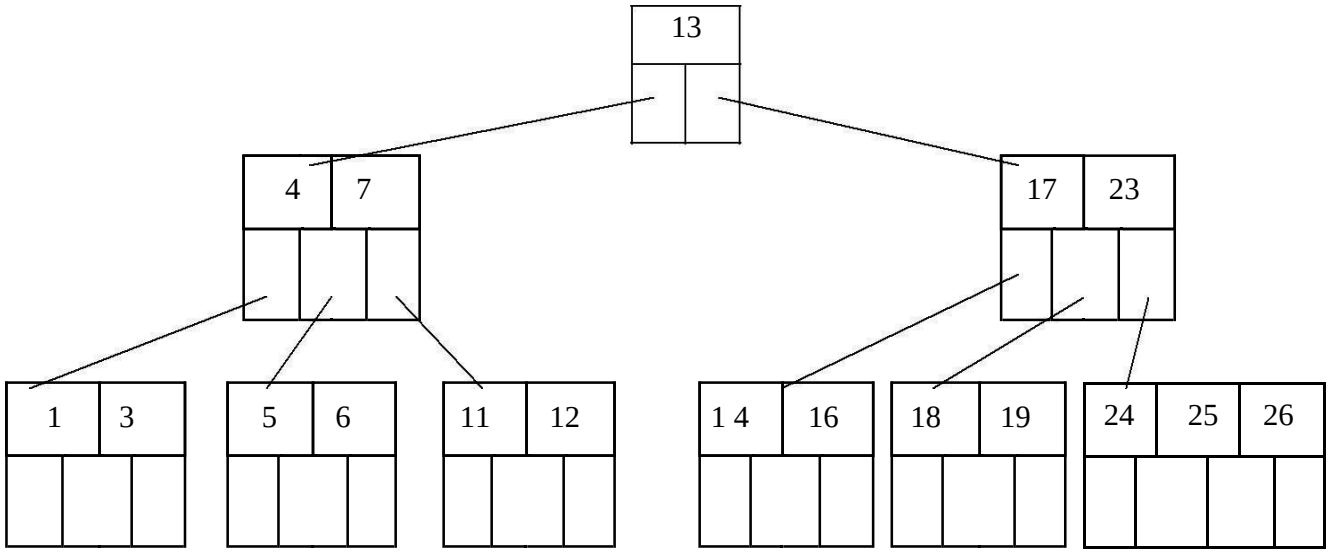
Consider a B-tree



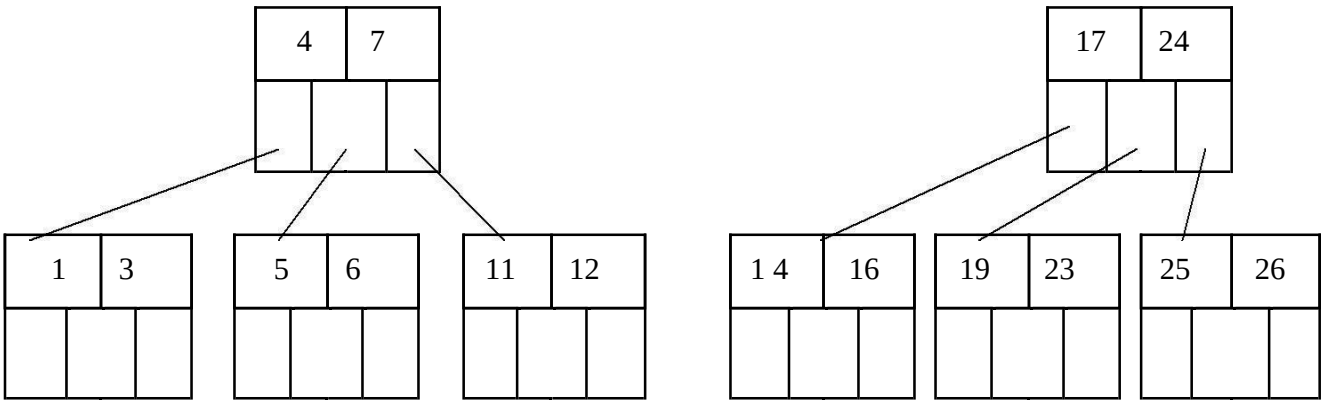
Delete 8, then it is very simple.



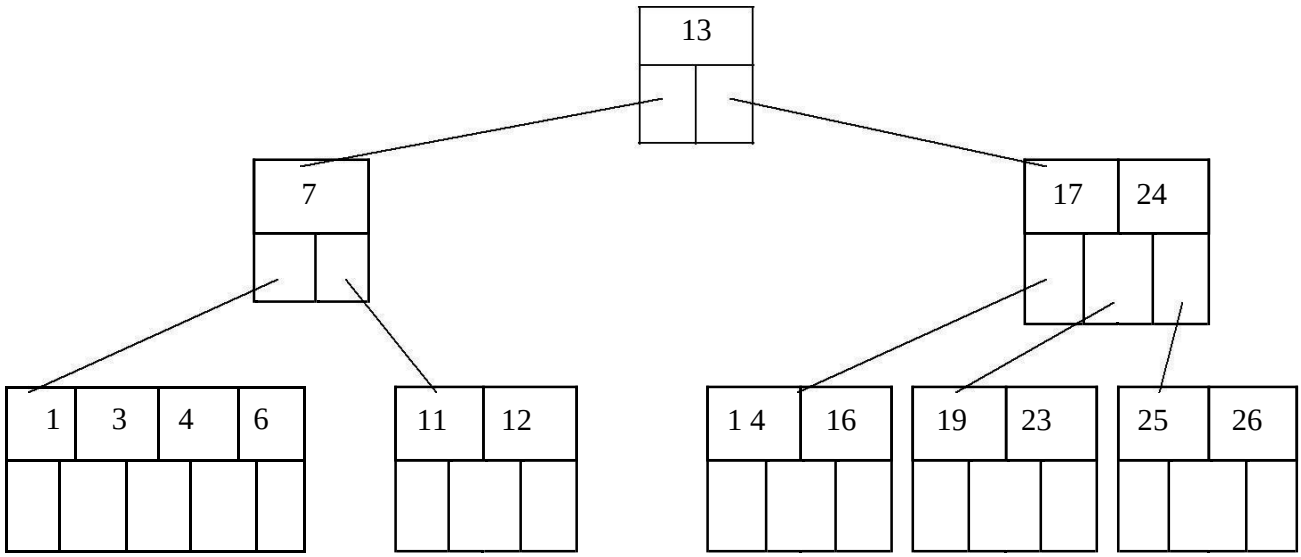
Now we will delete 20, the 20 is not in a leaf node so we will find its successor which is 23, Hence 23 will be moved up to replace 20.



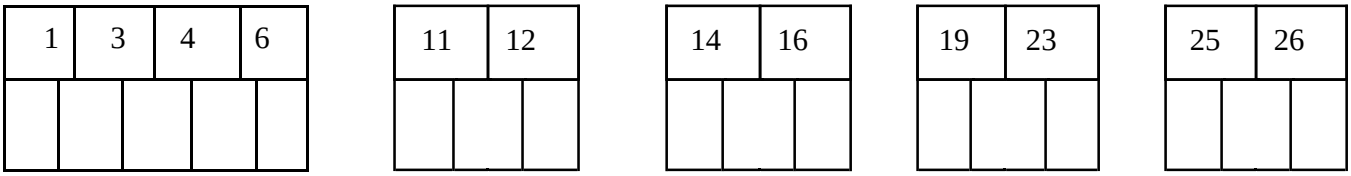
Next we will delete 18. Deletion of 18 from the corresponding node causes the node with only one key, which is not desired (as per rule 4) in B-tree of order 5. The sibling node to immediate right has an extra key. In such a case we can borrow a key from parent and move spare key of sibling up.



Now delete 5. But deletion of 5 is not easy. The first thing is 5 is from leaf node. Secondly this leaf node has no extra keys nor siblings to immediate left or right. In such a situation we can combine this node with one of the siblings. That means remove 5 and combine 6 with the node 1, 3. To make the tree balanced we have to move parent‘s key down. Hence we will move 4 down as 4 is between 1, 3, and 6. The tree will be-

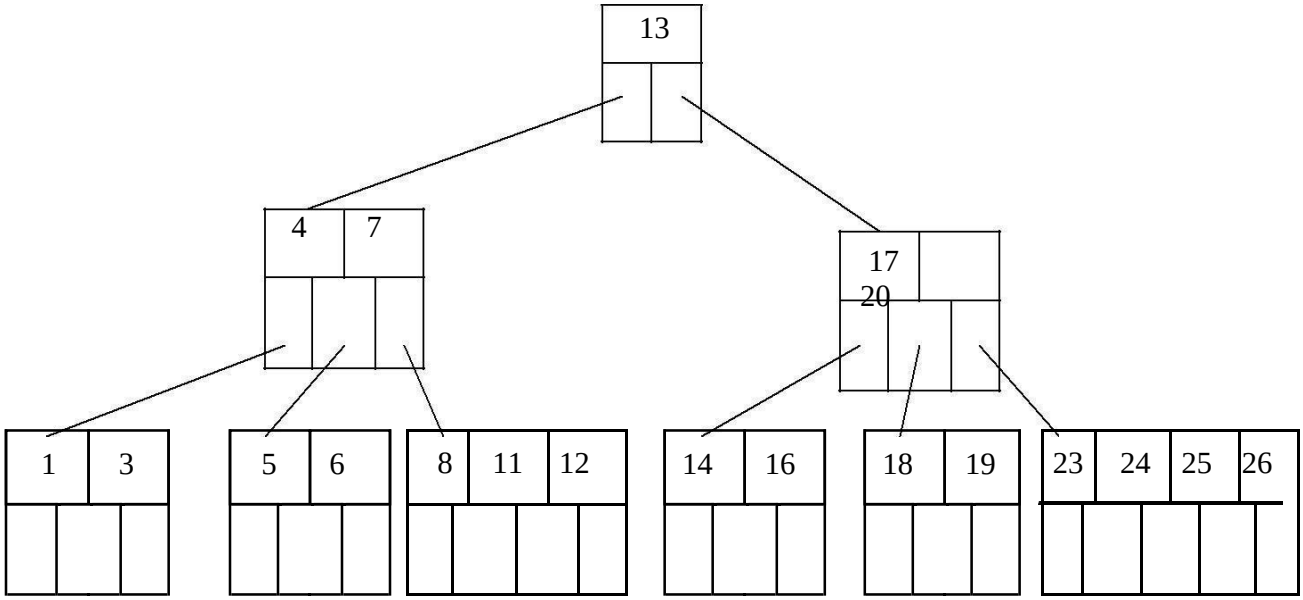


But again internal node of 7 contains only one key which not allowed in B-tree. We then will try to borrow a key from sibling. But sibling 17, 24 has no spare key. Hence we can do is that, combine 7 with 13 and 17, 24. Hence the B-tree will be



Searching

The search operation on B-tree is similar to a search to a search on binary search tree. Instead of choosing between a left and right child as in binary tree, B-tree makes an m-way choice. Consider a B-tree as given below.



If we want to search 11 then

i. 11 < 13 ; Hence search left node

ii. 11 > 7 ; Hence right most node

iii. 11 > 8 ; move in second block

iv. node 11 is found

The running time of search operation depends upon the height of the tree. It is O(log n).

Height of B-tree

The maximum height of B-tree gives an upper bound on number of disk access. The maximum number of keys in a B-tree of order 2m and depth h is

$$1 + 2m + 2m(m+1) + 2m(m+1)^2 + \ldots + 2m(m+1)^{h-1}$$

$$= 1 + \sum_{i=1}^{h} 2m(m+1)^{i-1}$$

The maximum height of B-tree with n keys

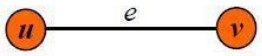
$$\log_{m+1}\frac{n}{2m} = O(\log n)$$

Terminology of Graph

Graphs:-

A graph G is a discrete structure consisting of nodes (called vertices) and lines joining the nodes (called edges). Two vertices are adjacent to each other if they are joint by an edge. The edge joining the two vertices is said to be an edge incident with them. We use V (G) and E(G) to denote the set of vertices and edges of G respectively.

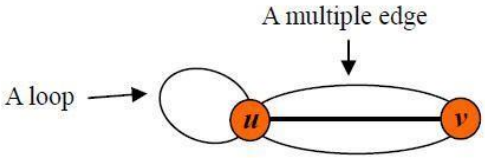
Example



u and *v* are adjacent vertices; *e* is an edge incident with *u* and *v*. *e* can also be denoted by *uv* or *vu*.

Loops and Multiple Edges

An edge joining only one vertex is called a *loop*. If there are more than one edge joining *u* and *v* of *G*, then all edges joining *u* and *v* form a *multiple edge* of *G*.



Euler Circuit and Euler Path

An *Euler circuit* in a graph *G* is a simple circuit containing every edge of *G*. An *Euler path* in *G* is a simple path containing every edge of *G*.

Graph Representations

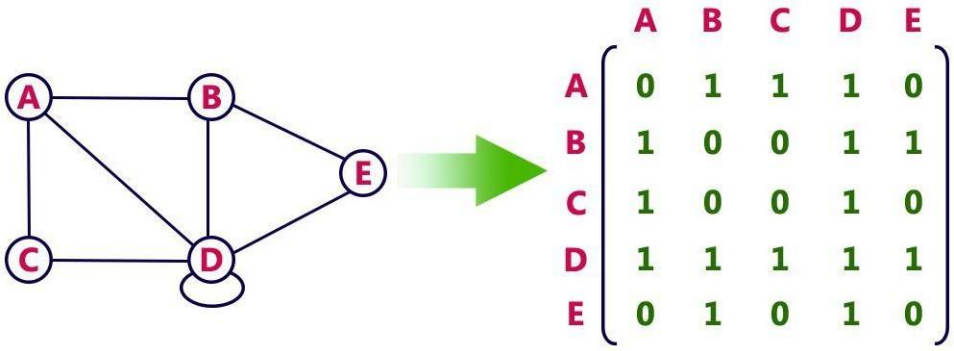
Graph data structure is represented using following representations...

1. **Adjacency Matrix**
2. **Incidence Matrix**
3. **Adj**

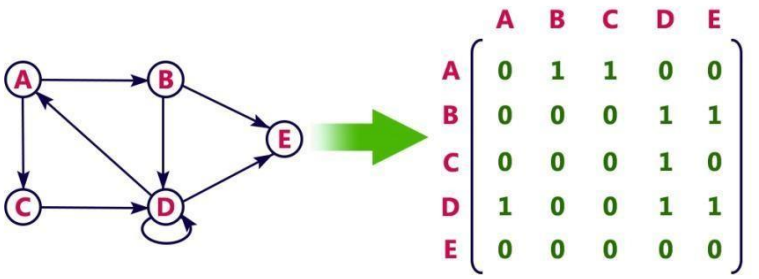
acency
List
Adjacency
Matrix

In this representation, graph can be represented using a matrix of size total number of vertices by total number of vertices. That means if a graph with 4 vertices can be represented using a matrix of 4X4 class. In this matrix, rows and columns both represents vertices. This matrix is filled with either 1 or 0. Here, 1 represents there is an edge from row vertex to column vertex and 0 represents there is no edge from row vertex to column vertex.

For example, consider the following undirected graph representation...



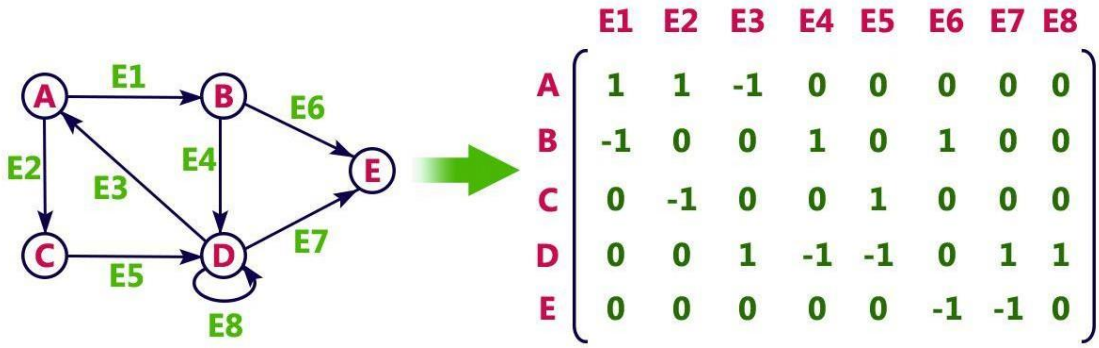
Directed graph representation...



Incidence Matrix

In this representation, graph can be represented using a matrix of size total number of vertices by total number of edges. That means if a graph with 4 vertices and 6 edges can be represented using a matrix of 4X6 class. In this matrix, rows represents vertices and columns represents edges. This matrix is filled with either 0 or 1 or -1. Here, 0 represents row edge is not connected to column vertex, 1 represents row edge is connected as outgoing edge to column vertex and -1 represents row edge is connected as incoming edge to column vertex.

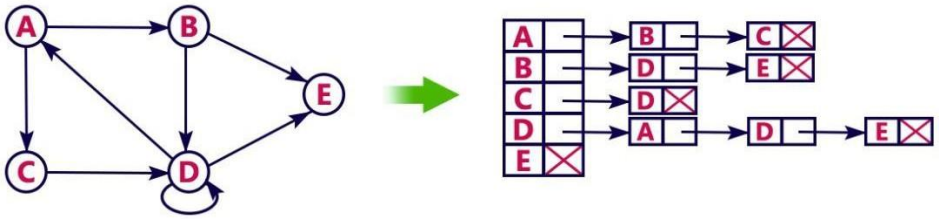
For example, consider the following directed graph representation...



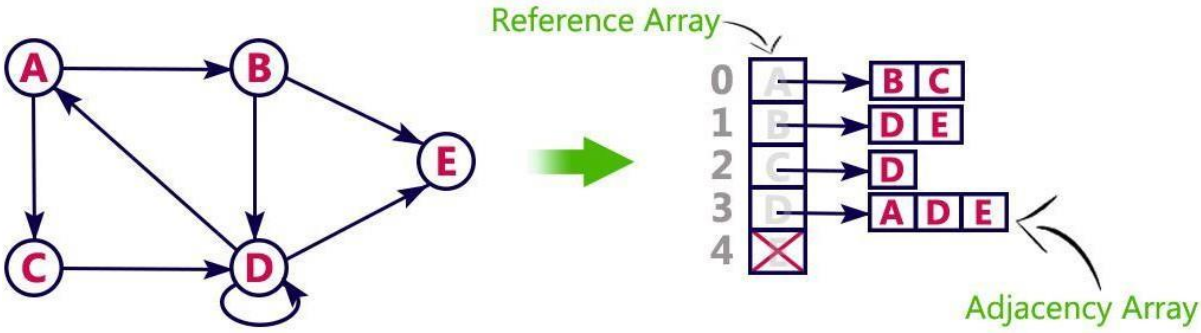
Adjacency List

In this representation, every vertex of graph contains list of its adjacent vertices.

For example, consider the following directed graph representation implemented using linked list...



This representation can also be implemented using array as follows..



Graph traversals

Graph traversal means visiting every vertex and edge exactly once in a well-defined order. While using certain graph algorithms, you must ensure that each vertex of the graph is visited

exactly once. The order in which the vertices are visited are important and may depend upon the algorithm or question that you are solving.

During a traversal, it is important that you track which vertices have been visited. The most common way of tracking vertices is to mark them.

Depth First Search (DFS)

The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.

This recursive nature of DFS can be implemented using stacks. The basic idea is as follows:
Pick a starting node and push all its adjacent nodes into a stack.
Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack.
Repeat this process until the stack is empty. However, ensure that the nodes that are visited are marked. This will prevent you from visiting the same node more than once. If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop.

```
DFS-iterative (G, s):                    //Where G is graph and s is source vertex
    let S be stack
    S.push( s )          //Inserting s in stack
    mark s as  visited.
    while ( S is not empty):
       //Pop a vertex from stack to visit next
      v = S.top( ) S.pop( )
      //Push all the neighbours of v in stack that
     are not visited for all neighbours w of v in
     Graph G:
        if w is not visited : S.push( w ) mark w as visited
  DFS-recursive(G, s):
     mark s as visited
     for all neighbours w of s in Graph G:
        if w is not visited:
           DFS-recursive(G, w)
```

Breadth First Search (BFS);

There are many ways to traverse graphs. BFS is the most commonly used approach.BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layerwise thus exploring the neighbour nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbour nodes.As the name BFS suggests, you are required to traverse the graph breadthwise as follows:
1.First move horizontally and visit all the nodes of the current layer
2.Move to the next layer.