UNIT-IV
TREE & GRAPH

Tree:

- A tree is a non-linear hierarchical data structure that consists of nodes connected by edges.
- A tree can be defined as a finite set of elements called nodes such that:
  i. A tree contains a special node "R" called Root node.
  ii. The remaining node of the tree contains an ordered pair of sub-trees T1, T2 & T3 called as sub-tree of the root node "R".
- in below tree there are total 12 nodes with values R,r1,a,b,r2,c,r3,d,e,f,g,h

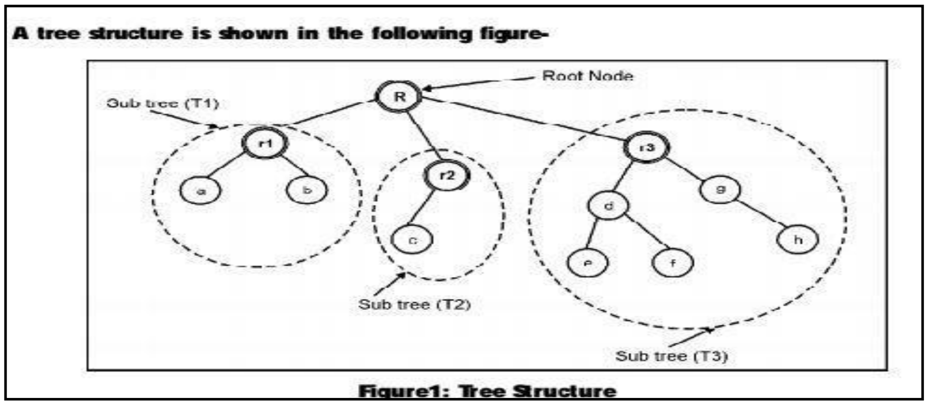
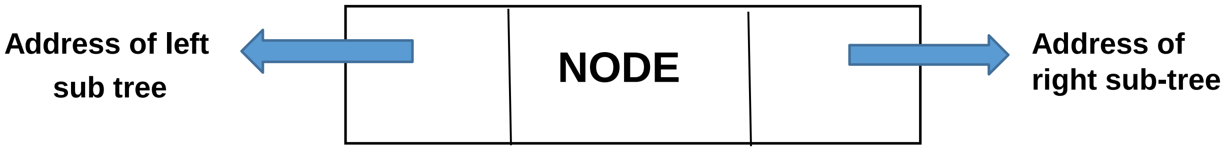


Figure1: Tree Structure

Binary Tree:

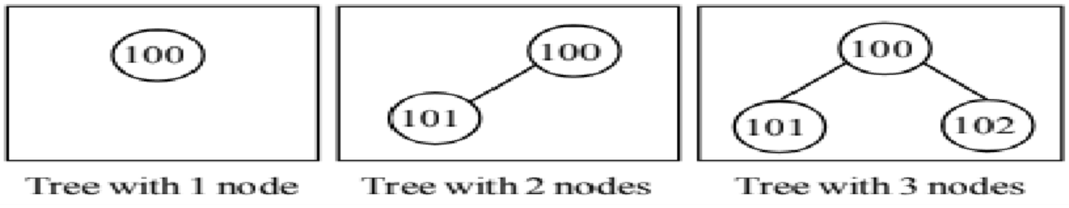
- A binary tree is a tree where each node has either zero, one or two sub-tree is called a binary tree.
- If any node has more than two sub-tree then it is not a binary tree.
- A tree can be empty or divided partitioned into 3 subgroups namely: root, left sub-tree and right sub-tree.

Pictorial representation of a node in binary tree.
Node= llink + data + rlink



Note that a node in a tree consists of three fields namely llink, data and rlink:
- llink – Basically, it contains address of left subtree.
- info – This field is used to store the actual data or information to be manipulated.
- rlink – Basically, it contains address of right subtree.

Identify Binary Tree:



Note:

- An empty tree is also a binary tree. Binary here means at most two i.e. zero,one or two sub-tree but not more than 2 subtree.
- All the tree above shown are binary tree

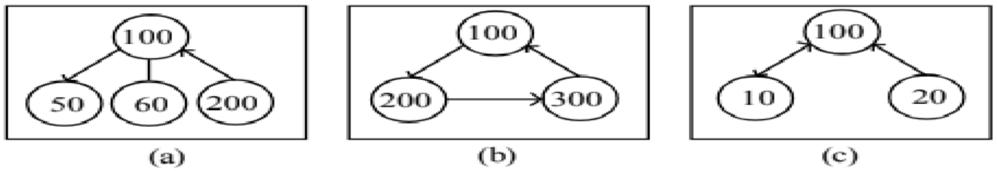


Fig 4

- The figure shown in fig 4(a) is not a binary tree because it is having 3 sub-tree which is not allowed in binary tree.
- The tree shown in Fig 4 (b) is not a binary tree. The node 100 has subtree 200. Node 200 has subtree 300, node 300 has subtree 100 and cycle repeats. There should not be any cycles in binary tree. So, it is not a binary tree.
- The tree shown in Fig 4 (c) is not a binary tree because the node 100 has subtree 10 and 10 has the subtree 100. If 10 is subtree of 100, then 100 cannot be the subtree of 10. So, it is not a binary tree.

Tree Terminologies/Technologies:

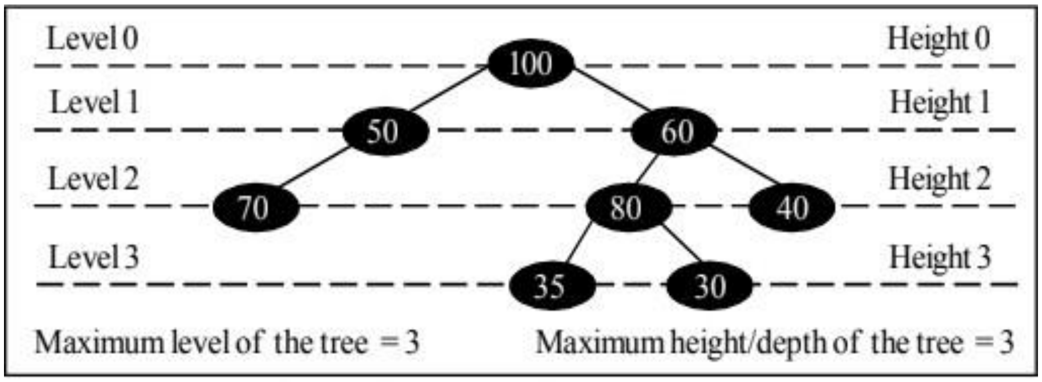


Fig 5 : Binary tree with all terms used

The various terminologies that are associated with a tree are:

1. **Root:** The node at the top of the tree is called root. There is only one root per tree and it is the first node in the tree. For example, node 100 is the root of the tree.

2. **Parent:** Any node except the root node has one edge upward to a node called parent **OR** immediate predecessor (Previous node) is called parent node. For Example:

- 50 is the parent of 70
- 60 is the parent of 80 and 40
- 80 is the parent of 35 and 30

2. **Child:** The node below a given node connected by its edge downward is called its child node **OR** immediate successor (Next node) is called Child node. For Example:
   - 50 and 60 are children of 100
   - 80 and 40 are children of 60
   - 70 is child of 50
   - 35 and 30 are children of 80

3. **Siblings :** Two or more nodes having the same parent are called siblings. For example,
   - 50 and 60 are siblings since they have same parent 100
   - 80 and 40 are siblings since they have same parent 60
   - 35 and 30 are siblings since they have same parent 80

4. **Leaf:** The node which does not have any child node is called the leaf node. For Example:
   - 70,35,30 and 40 are the leaf nodes.

5. **Internal nodes:** The nodes except leaf nodes in a tree are called internal nodes.
   - 100, 50, 60 and 80 are the internal nodes.

6. **External nodes:** The leaf nodes in a tree are called external nodes.
   - 70, 35, 30 and 40 are the external nodes.

7. **Degree of node:** No. of children of a node is called degree of the node.
   - Degree of 100 is 2
   - Degree of 70 is 0
   - Degree of 80 is 2

8. **Degree of tree:** Maximum no. of degree of a node.
   - Degree of the above tree is 2.

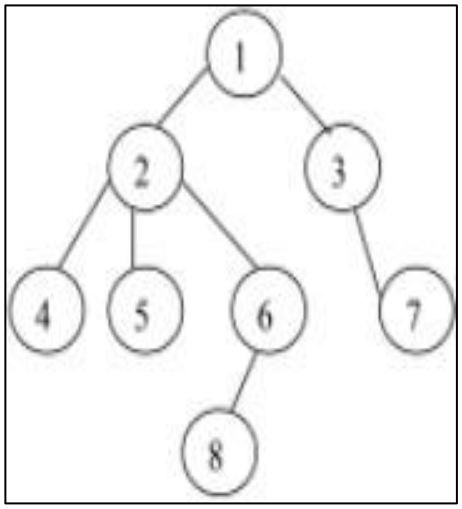
9. **Height of node:** Highest path from a node to that leaf node.
   - Height of 100 is 3
   - Height of 50 is 1
   - Height of 60 is 2

10. Height of Tree: Maximum height of node is called height of tree.

- Height of above tree is 3

Representation of tree:

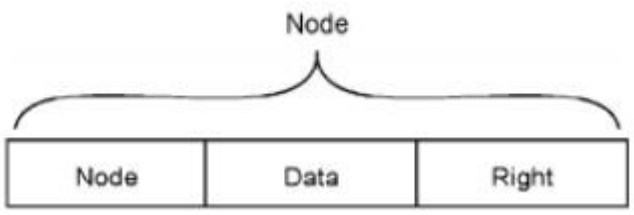
- As tree shows the hierarchical relationship among the elements or nodes, the tree is represented pictorially from top to bottom.
- The top node is of course the root of the tree and children follow the root.
- Most popular representation of tree is pictorial representation. In this representation the nodes are pictured with the circles labeled with the information of the node.
- The different level nodes are connected by edges as straight lines.
- The edges are usually undirected one, because the growth of the tree is followed from the root of the tree.



In the above representation node with information 1 is root of the tree. Nodes 2 and 3 are children of root. Nodes 4, 5 and 6 are children of 2. 6 is the root for node 8. Similarly 3 is root of node 7. Nodes 2, 3 and 6 are internal nodes of the tree. Nodes 4, 5, 8 and 7 are leaves of the tree (yield).

Representation of a binary tree in memory

- A binary tree is made of nodes, where each node contains a “left” pointer, a “right” pointer, and a data element.
- The “root” pointer points to the topmost node in the tree.
- The left and right pointers recursively point to the smaller “sub trees” on either side. A null pointer represents a binary tree with no elements - the empty tree.
- The structure of each node of a binary tree contains the data field, a pointer to the left child and a pointer to the right child. The following figure shows this structure.



This structure can be defined as follows:

```
struct tnode
{
struct tnode *left; int data;
struct tnode *right;
} ;
```

Representation of binary tree :

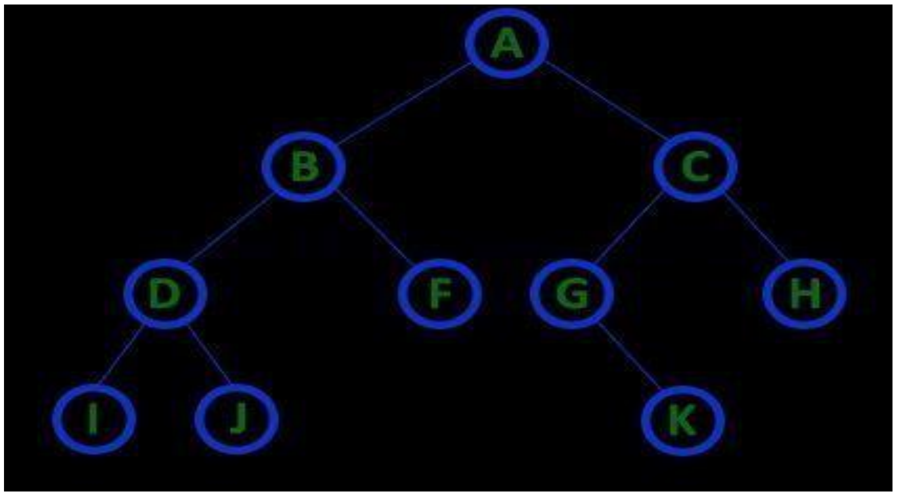
A binary tree can be represented in two ways.

1. Array representation
2. Linked List representation.

1) Array representation :

- It consumes lots of space in comparison to linked representation because the size of the array depends directly on the height of the binary tree.
- The height of the binary tree decides the maximum number of nodes. If ‘n’ is the height then the total number of nodes present in a binary tree will be maximum $2^n + 1 - 1$ which is decided as size of one-dimensional array to represent the binary tree. For eg: if height of a tree is 4 then maximum nodes/elements in a binary tree will be $2^{n+1}-1 = (2^5+1)-1=32-1=31$,so total 31 nodes can be inserted in a binary tree and 31 blocks will be allocated.

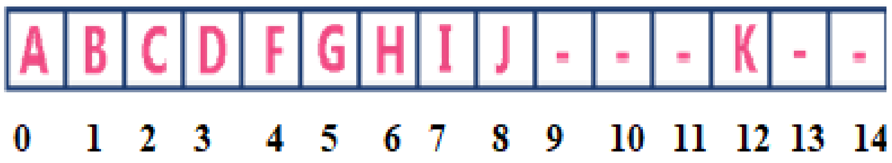
Consider the following binary tree.



- B & C are children's of A
- D & F are children's of B
- I & J are children's of D
- G , H are children's of C
- k is the child of G

Array Representation of Above Binary Tree

- The height of the tree is defined as the longest path from the root node to the leaf node.
- The tree which is shown above has a height equal to 3. Therefore, the maximum number of nodes at height 3 is equal to (2^{h+1}-1) = 15.



A	B	C	D	F	G	H	I	J	-	-	-	K	-	-
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

To represent the above binary tree an array of size 15 is required.

In above representation you can see 5 blocks are wasted because in tree only 10 elements are present but memory is allocated for 15 elements.

- 2 blocks got wasted for left and right child of F
- 1 Block got wasted for left child of G
- 2 blocks got wasted for left and right child of H

From Array representation it become very difficult to find out parent node, left child or right child of any node.

- there are few formulas to find out index no of nodes.
- Left child: 2*i+1 where i is the index of parent node.
- Right Child: 2*i+2
- Parent node: i-1/2

<u>Questions:</u>

Find out the left child of C. Index of C is 2.

Solution:

2*i+1(Formula)

2*2+1=5

at 5th index G is present that means G is the left child of C.

Find out the right child of B. Index of B is 1.

Solution:

2*i+2(Formula)

2*1+2=4

at 4th index F is present that means F is the right child of B.

Find out the right child of parent of H. Index of H is 6.

Solution:

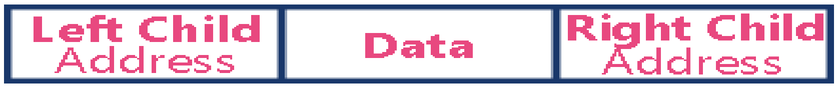
i-1/2(Formula)

6-1/2=5/2= floor(2.5)=2

at 2nd index C is present that means C is the parent of H.

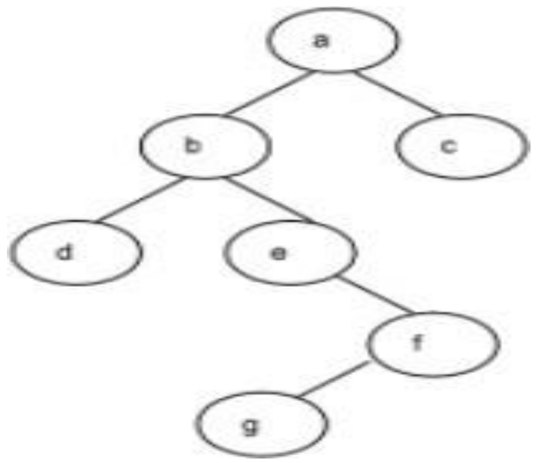
2) Linked List Representation:

- In linked representation memory for the nodes are allocated at runtime so possibility of memory wastage is less.
- We use a double linked list to represent a binary tree. In a double linked list, every node consists of three fields. First field for storing left child address, second for storing actual data and third for storing right child address.
- In this linked list representation, a node has the following structure...

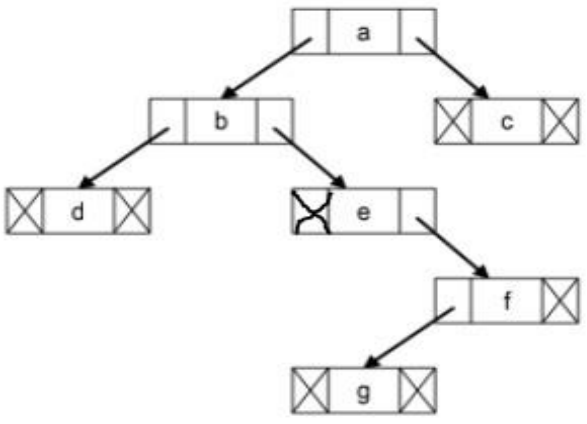


Left Child Address	Data	Right Child Address

Consider the following binary tree...



Linked List representation of above binary tree:



- The space required to store the binary tree, using linked representation is very less in Comparison to that of array representation.
- The space required is equal to the number of nodes multiplied by size of each node. But accessing the node with information 'g' is not directly possible. We should start from the root then we have to move to 'b', then 'e', then 'f' and the finally to 'g'.

Operations on Binary Tree

Following are the basic operations of a tree −

- **Search** − Searches an element in a tree.
- **Insert** − Inserts an element in a tree.
- **Pre-order Traversal** − Traverses a tree in a pre-order manner.
- **In-order Traversal** − Traverses a tree in an in-order manner.
- **Post-order Traversal** − Traverses a tree in a post-order manner.

1. Pre-order Traversal:

If the root node is visited or processed before going into the processing of left part and right part of it, then the traversal technique is called as pre-order traversal.

2. in-order Traversal:

If the root node is visited or processed after finishing the processing of left part of it, then the traversal technique is called as in-order traversal.

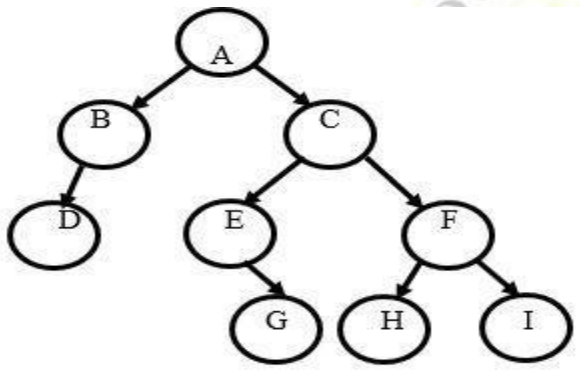
3. post-order Traversal:

If the root node is visited or processed after finishing the processing of left and right part of it, then the traversal technique is called as post-order traversal.

In a simple way all the three traversals are given as follows:

• pre-order : Process root	Process left	Process right	ROOT	LEFT	RIGHT
• in-order : Process left	Process root	Process right	LEFT	ROOT	RIGHT
• post-order : Process left	Process right	Process root	LEFT	RIGHT	ROOT

Example 1:

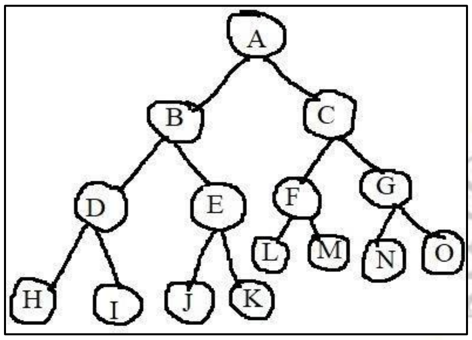


Inorder traversal (Left-Root-Right):
D B A E G C H F I

Preorder traversal (Root-Left-Right):
A B D C E G F H I

Postorder traversal (Left-Right-Root):
D B G E H I F C A

Example 2:

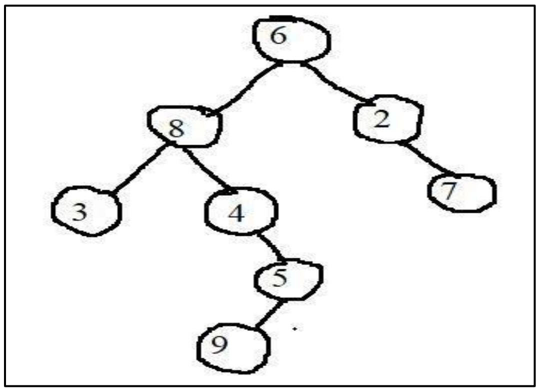


Inorder traversal (Left-Root-Right):
H D I B J E K A L F M C N G O

Preorder traversal (Root-Left-Right):
A B D H I E J K C F L M G N O

Postorder traversal (Left-Right-Root):
H I D J K E B L M F N O G C A

Example 3:



Inorder traversal (Left-Root-Right):
3 8 4 9 5 6 2 7

Preorder traversal (Root-Left-Right):
6 8 3 4 5 9 2

Postorder traversal (Left-Right-Root):
3 9 5 4 8 7 2 6

Binary Search Tree:

A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties −

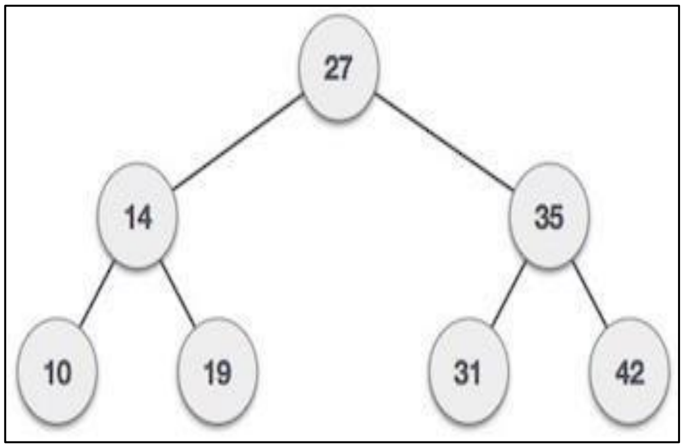
- The value of the key of the left sub-tree is less than the value of its parent (root) node's key.
- The value of the key of the right sub-tree is greater than the value of its parent (root) node's key.

Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and can be defined as −
left_subtree (keys) < root node (key) < right_subtree (keys)

Representation

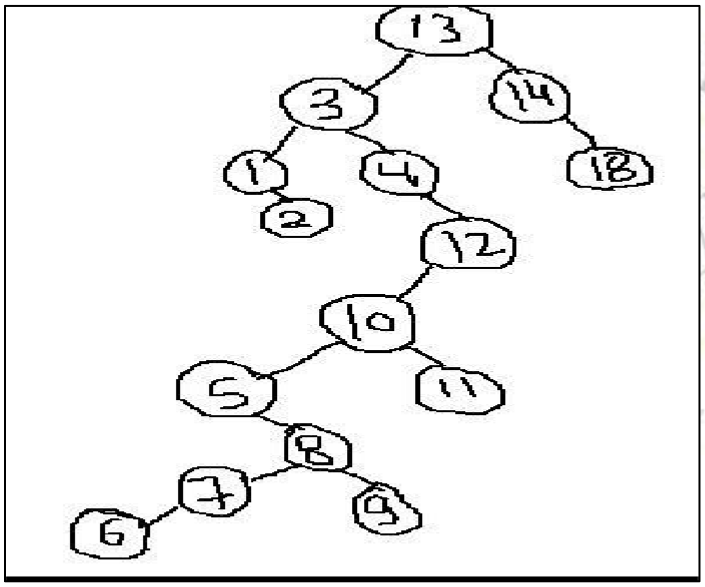
BST is a collection of nodes arranged in a way where they maintain BST properties.

Following is a pictorial representation of BST



Example 1:
Construct binary search tree for following elements

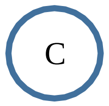
13, 3, 4, 12, 14, 10, 5, 1, 8, 2, 7, 9, 11, 6, 18



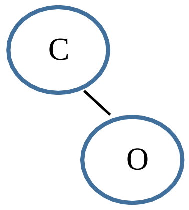
Binary search tree

Example 2:
Draw a binary search tree for the following string considering each character as information of the node in a binary search tree.
C O M P U T E R

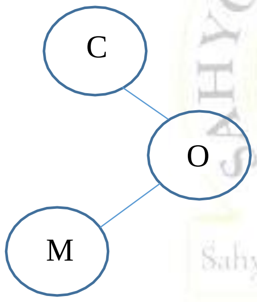
Solution
Initially the BST (Binary Search Tree) is empty. First character from the string is 'C'. It becomes the root of the tree. So, the BST is:



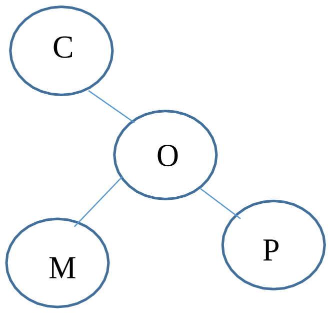
Next character from the string is ‘O’. This character is compared with the root character ‘C’, it is greater than ‘C’. As the right pointer of ‘C’ node is NULL. The node with information ‘O’ becomes the right node of ‘C’. So, the BST now is :



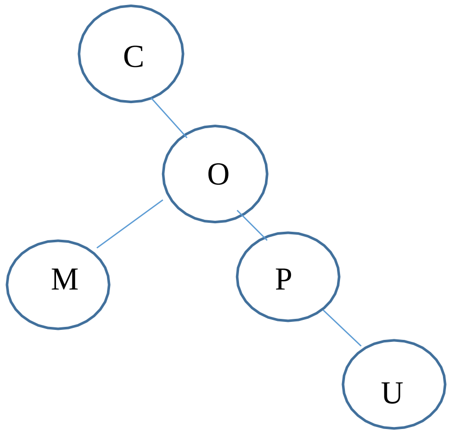
Next character is ‘M’. This character is compared with C. ‘M’ is greater than ‘C’. Right pointer of ‘C’ is not equal to NULL. So ‘M’ is compared with ‘O’. Now ‘M’ is less than ‘O’ and the left pointer ‘O’ is NULL, so the node with information ‘M’ becomes left node of ‘O’. So, the BST is :



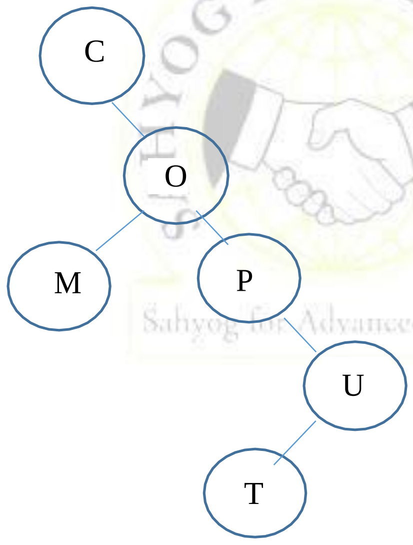
Next character is ‘P’. This character is first compared with the root information of BST. It is greater than root information that is ‘C’. It is also greater than the right node information ‘O’. As the right pointer of ‘O’ is NULL, the node that is to be inserted in BST becomes the right node of ‘O’. So, now the BST is:



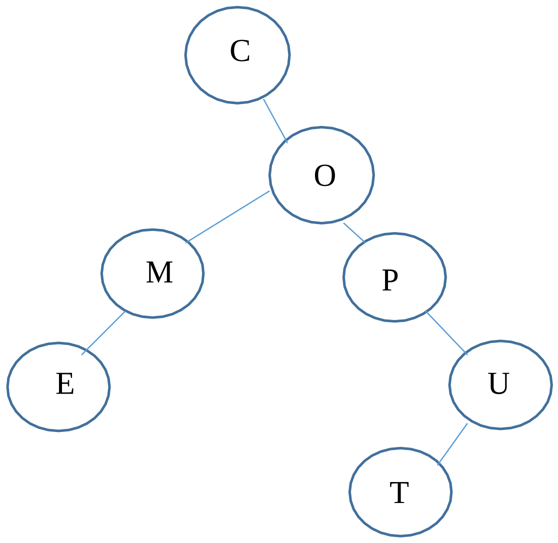
The next character is ‘U’. It is greater than ‘C’, ‘O’ and ‘P’. The right pointer of node with information ‘P’ is equal to NULL, so the new node with information ‘U’ is inserted as the right node of ‘P’. So, the BST becomes:



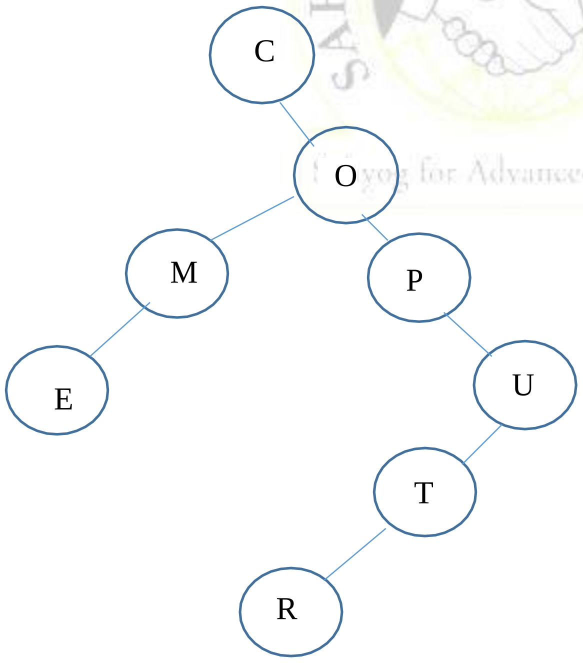
The next character is ‘T’. It is greater than ‘C’, ‘O’, and ‘P’ but less than ‘U’. The left pointer of node ‘U’ is NULL, so the new node with information ‘T’ is inserted to the left of ‘U’. So, the BST becomes:



The next character is ‘E’. It is greater than ‘C’, less than ‘O’ and less than ‘M’. The left pointer of the node ‘M’ is NULL, so the new node with information ‘E’ is inserted as left node of ‘M’. So, the BST becomes:



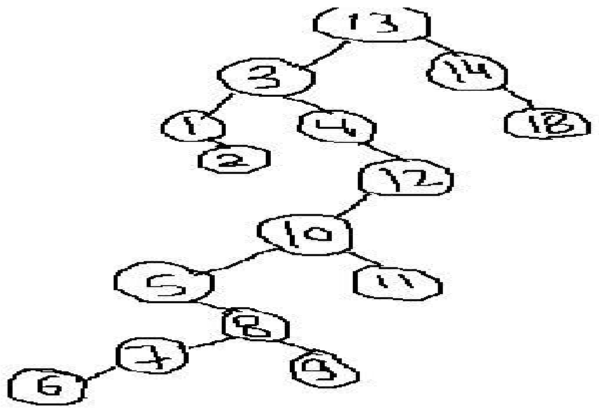
The next and last character is ‘R’. It is greater than ‘C’, ‘O’, and ‘P’ but less than ‘U’ and ‘T’. The left pointer of ‘T’ is equal to NULL, so the last new node with information ‘R’ becomes the left node of ‘T’. So, the final BST is :



Applications of Binary Search Tree:

1. **Sorting**
2. **Expression Tree-conversion of infix expression to prefix and postfix**
3. **Heap Sort**

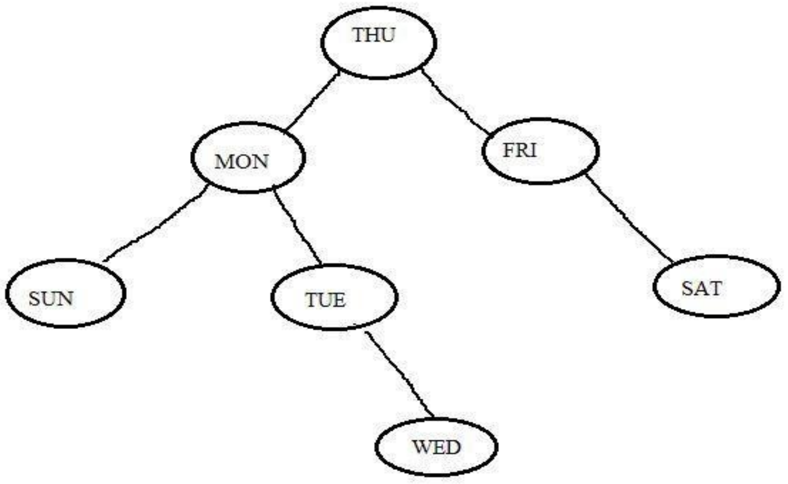
1. **Sorting:**
   When a binary tree is traversed using inorder technique the list can be sorted in ascending order,This is a good application of binary tree.



BINARY SEARCH TREE

Inorder Traversal of above BST:

1,2,3,4,5,6,7,8,9,10,11,12,13,14,18 (Sorted List)



BINARY SEARCH TREE

Inorder Traversal of above BST:

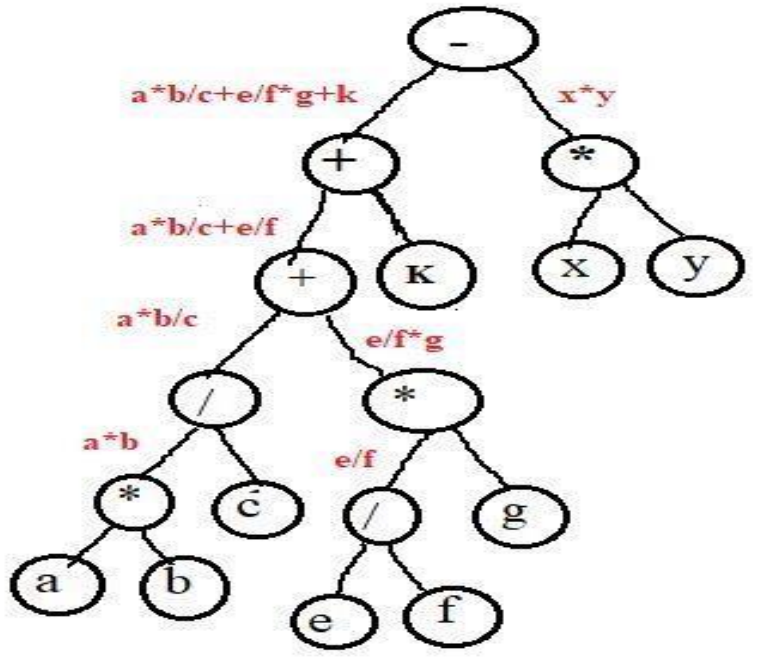
SUN, MON, TUE, WED, THU, FRI, SAT (Sorted List)

2. Expression Tree-conversion of infix expression to prefix and postfix

- The expression tree is a binary tree in which each internal node corresponds to the operator and each leaf node corresponds to the operand.
- Operators-> internal nodes
- Operands->leaf nodes

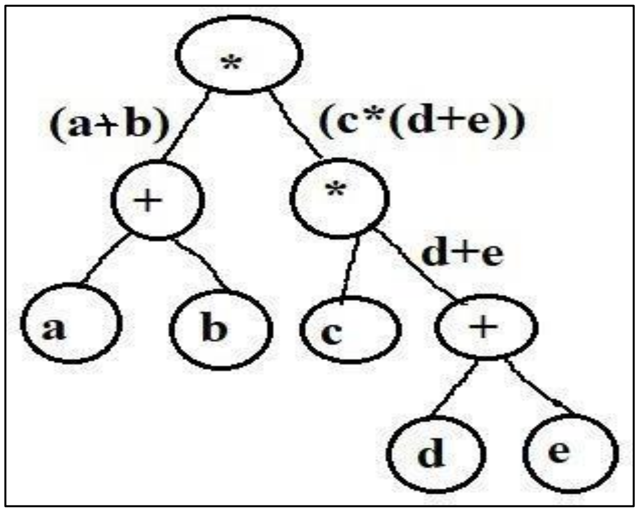
Construct expression tree for following infix expression:
i. a*b/c+e/f*g+k-x*y

- **Operator which will execute at the end will be the root node.**



Traverse the above tree using preorder technique to get prefix expression and traverse using postorder technique to get postfix expression.

ii. (a+b) * (c*(d+e)) ->infix Expression



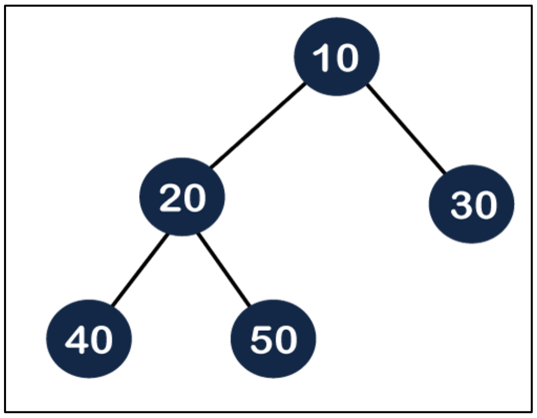
Prefix Expression: * + a b * c + d e
Postfix Expression: a b + c d e + * *

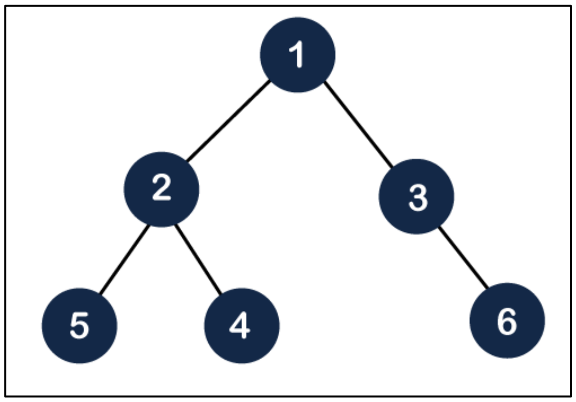
Heap Tree:

A heap tree is a complete binary tree. Complete binary tree means tree must be left justified i.e. first we have to set left child then only we can move to right side and once one level is completed then only move to next level..

What is a complete binary tree?

A complete binary tree is a binary tree in which all the levels except the last level, i.e., leaf node should be completely filled, and all the nodes should be left-justified.



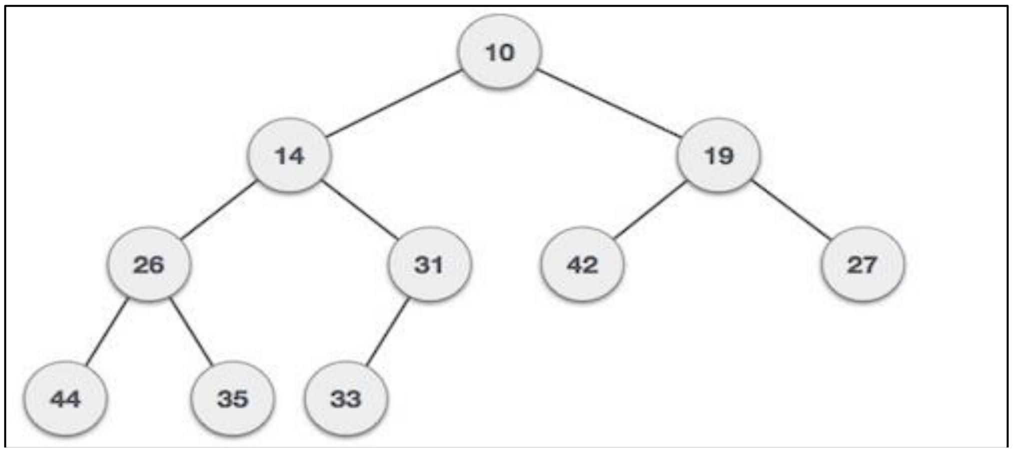


The above figure shows that all the internal nodes are completely filled except the leaf node, but the leaf nodes are added at the right part; therefore, the above tree is not a complete binary tree.

There are two types of the heap:

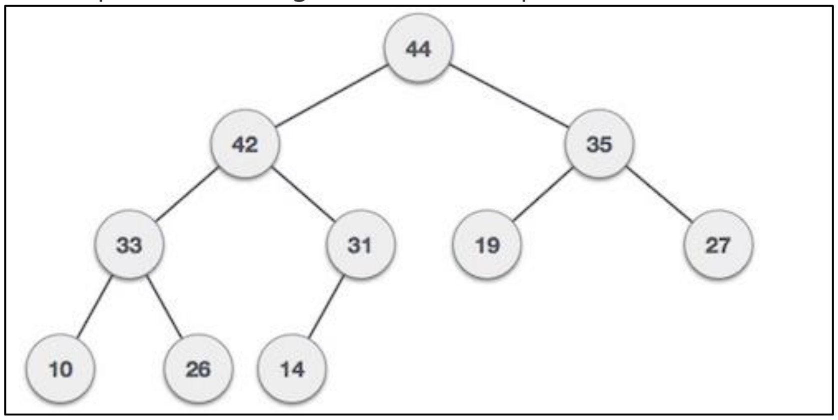
- Min Heap
- Max heap

Min Heap: The value of the parent node should be less than or equal to either of its children.



In the above figure, 11 is the root node, and the value of the root node is less than the value of all the other nodes (left child or a right child).
10<14,14<26,14<31,19<42,19<27,31<33,All they root elements are less than child element, hence it's a min heap tree.

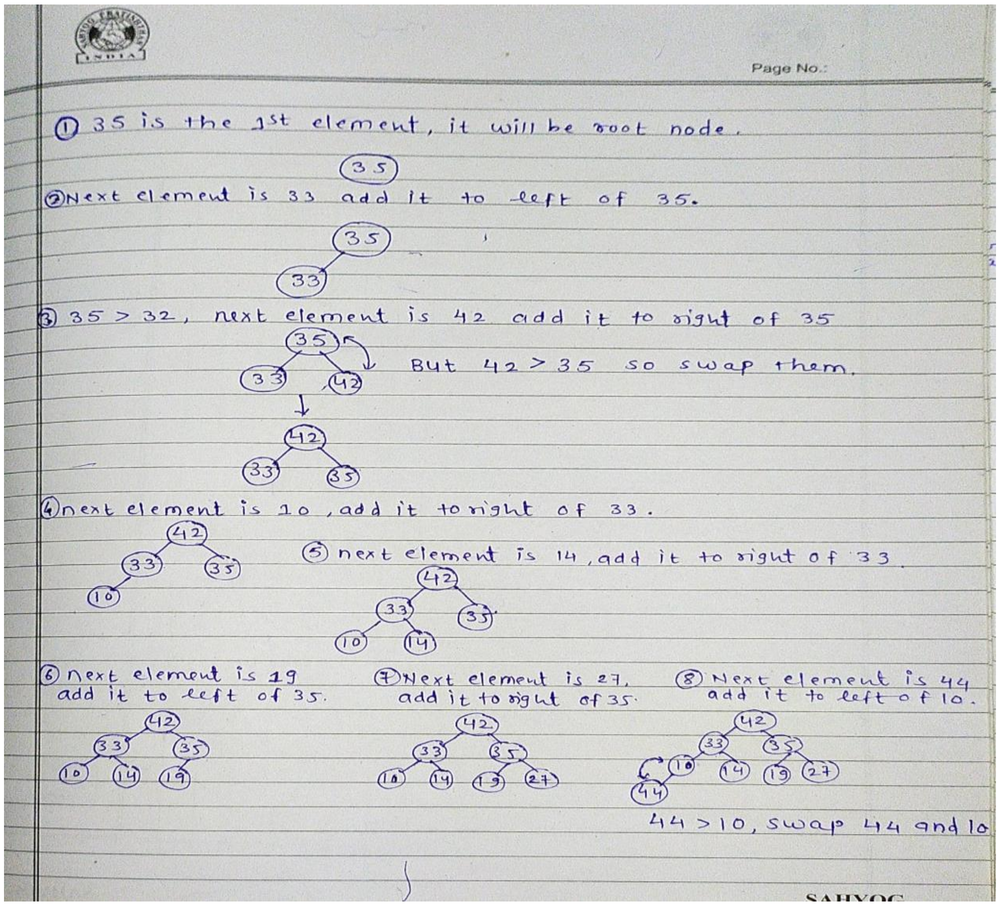
Max Heap: The value of the parent node is greater than or equal to its children.



The above tree is a max heap tree as it satisfies the property of the max heap. Now, let's see the array representation of the max heap.
44 > 42,44>35,42>33,42>31,33>10,33>26,31>16,35>19,35>27,All the parent node is greater than child element.

Construction of MAX-HEAP TREE:

Construct Max heap tree for following elements :_**35,33,42,10,14,19,27,44**



Page No.:

① 35 is the 1st element, it will be root node.

②Next element is 33 add it to left of 35.

③ 35 > 32, next element is 42 add it to right of 35

But 42 > 35 so swap them.

④next element is 10, add it to right of 33.

⑤ next element is 14, add it to right of 33

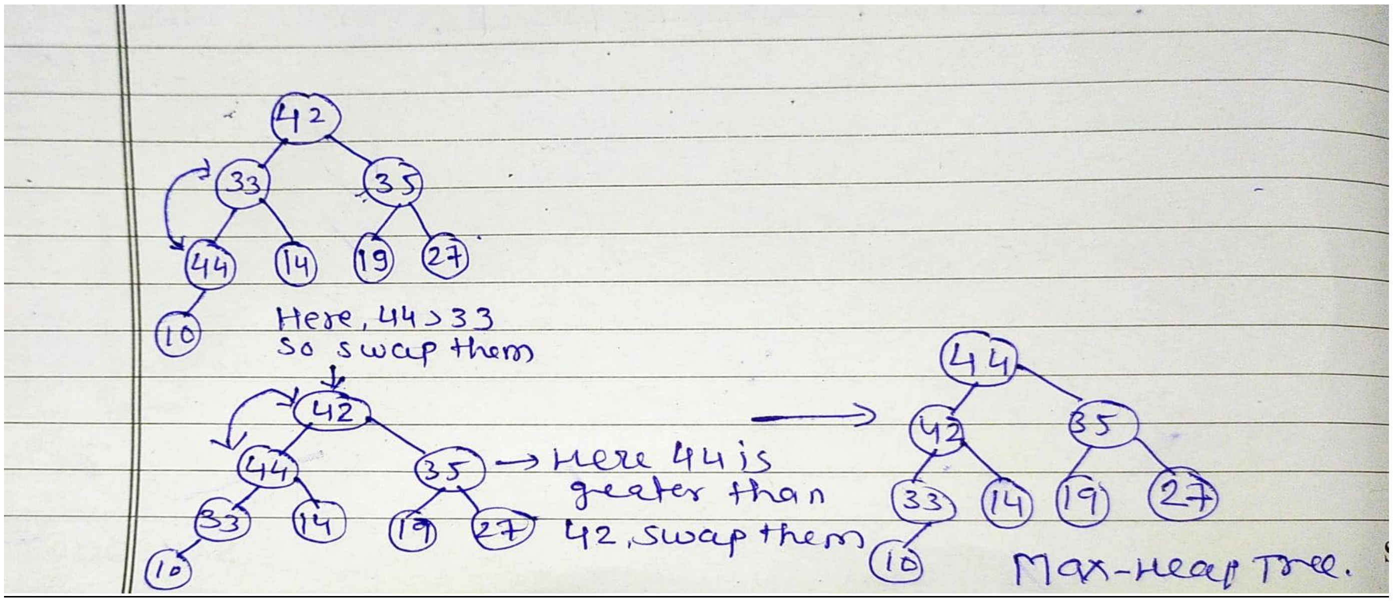
⑥ next element is 19 add it to left of 35.

⑦Next element is 27, add it to right of 35.

⑧ Next element is 44 add it to left of 10.

44 > 10, swap 44 and 10

SAHYOG



Operations of Heap Tree:

- **Heapify:** a process of creating a heap from an array.
- **Insertion:** process to insert an element
- **Deletion:** deleting the top element of the heap or the highest priority element, and then organizing the heap and returning the element with time complexity O(log N).
- **Peek:** to check or find the most prior element in the heap, (max or min element for max and min heap).

HEAP SORT:

Heapsort is a popular and efficient sorting algorithm. The concept of heap sort is to eliminate the elements one by one from the heap part of the list, and then insert them into the sorted part of the list.

Now, let's see the working of the Heapsort Algorithm.

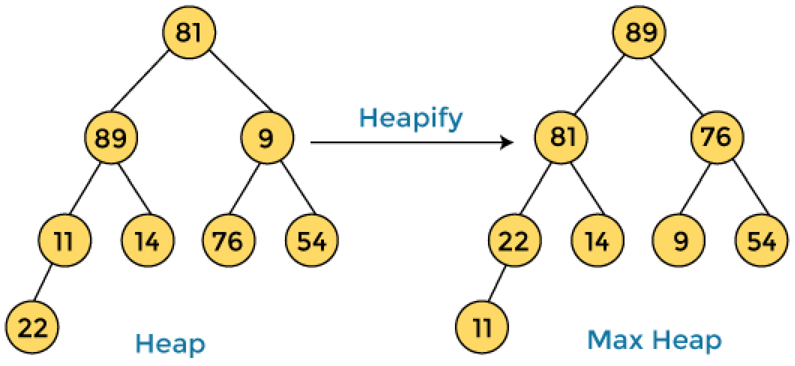
In heap sort, basically, there are two phases involved in the sorting of elements. By using the heap sort algorithm, they are as follows -

- The first step includes the creation of a heap by adjusting the elements of the array.
- After the creation of heap, now remove the root element of the heap repeatedly by shifting it to the end of the array, and then store the heap structure with the remaining elements.

Now let's see the working of heap sort in detail by using an example. To understand it more clearly, let's take an unsorted array and try to sort it using heap sort. It will make the explanation clearer and easier.

81	89	9	11	14	76	54	22

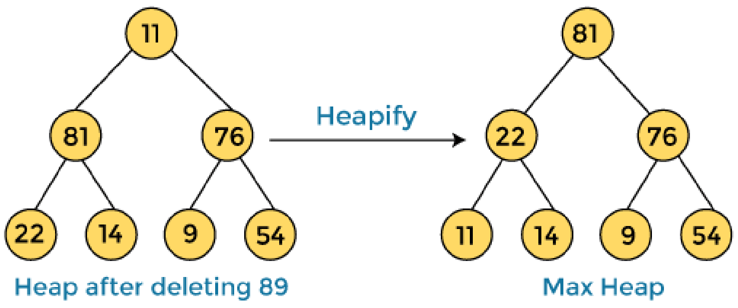
First, we have to construct a heap from the given array and convert it into max heap.



After converting the given heap into max heap, the array elements are -

89	81	76	22	14	9	54	11

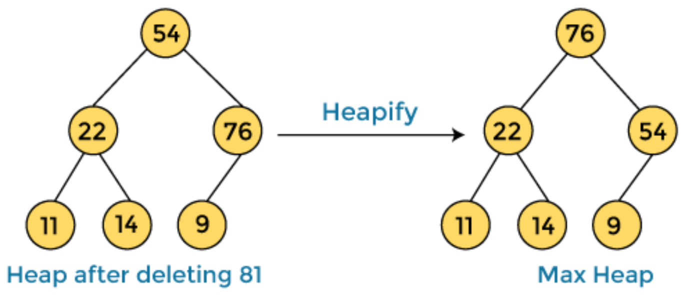
Next, we have to delete the root element **(89)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(11).** After deleting the root element, we again have to heapify it to convert it into max heap.



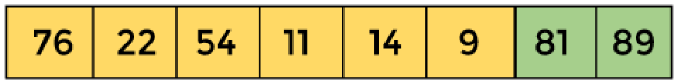
After swapping the array element **89** with **11,** and converting the heap into max-heap, the elements of array are -

81	22	76	11	14	9	54	89

In the next step, again, we have to delete the root element **(81)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(54).** After deleting the root element, we again have to heapify it to convert it into max heap.

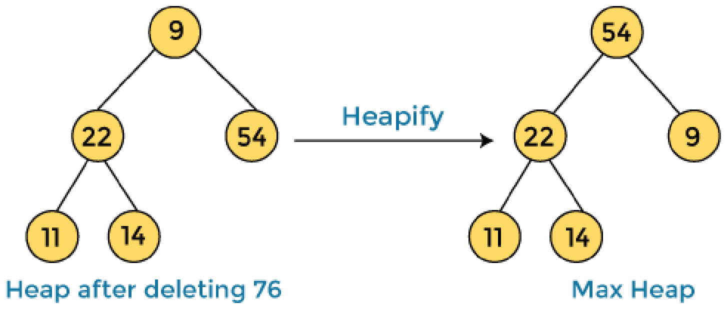


After swapping the array element **81** with **54** and converting the heap into max-heap, the elements of array are -

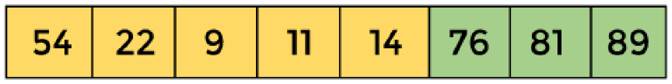


76	22	54	11	14	9	81	89

In the next step, we have to delete the root element **(76)** from the max heap again. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting the root element, we again have to heapify it to convert it into max heap.

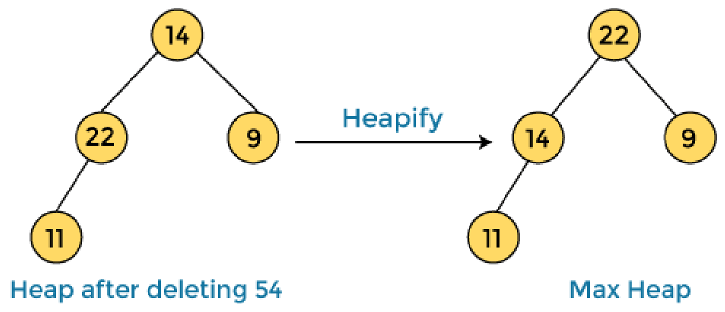


After swapping the array element **76** with **9** and converting the heap into max-heap, the elements of array are -

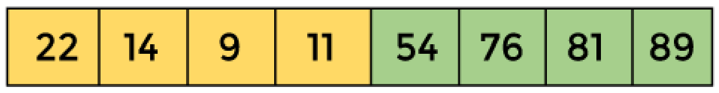


54	22	9	11	14	76	81	89

In the next step, again we have to delete the root element **(54)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(14).** After deleting the root element, we again have to heapify it to convert it into max heap.

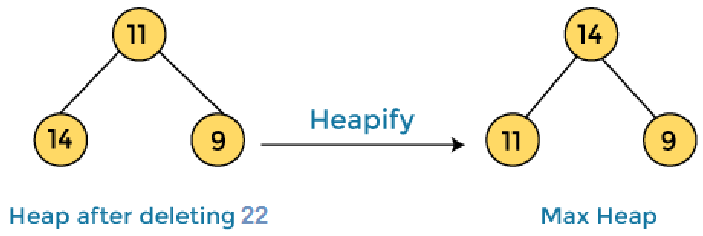


After swapping the array element **54** with **14** and converting the heap into max-heap, the elements of array are -

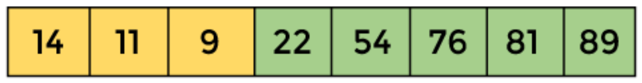


22	14	9	11	54	76	81	89

In the next step, again we have to delete the root element **(22)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(11).** After deleting the root element, we again have to heapify it to convert it into max heap.

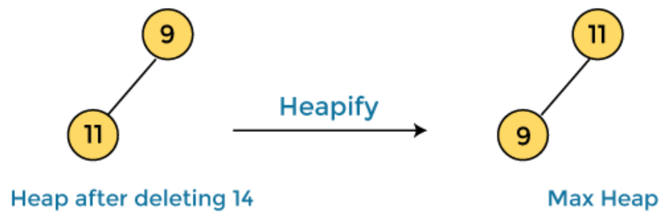


After swapping the array element **22** with **11** and converting the heap into max-heap, the elements of array are -

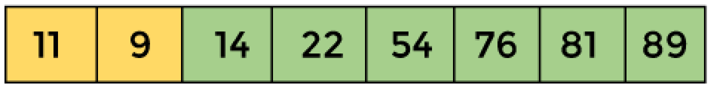


14	11	9	22	54	76	81	89

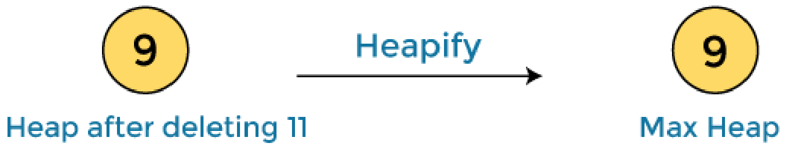
In the next step, again we have to delete the root element **(14)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting the root element, we again have to heapify it to convert it into max heap.



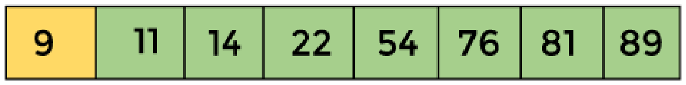
After swapping the array element **14** with **9** and converting the heap into max-heap, the elements of array are -



In the next step, again we have to delete the root element **(11)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting the root element, we again have to heapify it to convert it into max heap.



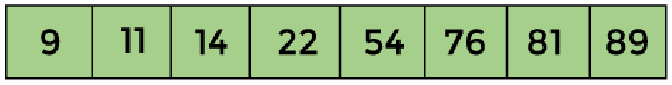
After swapping the array element **11** with **9,** the elements of array are -



Now, heap has only one element left. After deleting it, heap will be empty.

9
Remove 9
Empty

After completion of sorting, the array elements are -



Now, the array is completely sorted.

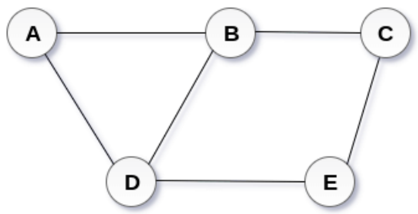
Graph

A graph can be defined as group of vertices and edges that are used to connect these vertices. A graph can be seen as a cyclic tree, where the vertices (Nodes) maintain any complex relationship among them instead of having parent child relationship.

Definition

A graph G can be defined as an ordered set G(V, E) where V(G) represents the set of vertices and E(G) represents the set of edges which are used to connect these vertices.

A Graph G(V, E) with 5 vertices (A, B, C, D, E) and six edges ((A,B), (B,C), (C,E), (E,D), (D,B), (D,A)) is shown in the following figure.



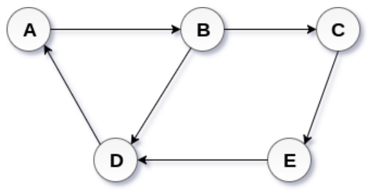
Undirected Graph

Directed and Undirected Graph

A graph can be directed or undirected. However, in an undirected graph, edges are not associated with the directions with them. An undirected graph is shown in the above figure since its edges are not attached with any of the directions. If an edge exists between vertex A and B then the vertices can be traversed from B to A as well as A to B.

In a directed graph, edges form an ordered pair. Edges represent a specific path from some vertex A to another vertex B. Node A is called initial node while node B is called terminal node.

A directed graph is shown in the following figure.



Directed Graph

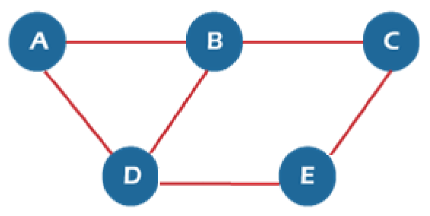
Graph representation:

The data structures 'two-dimensional array' and 'linked list' are most commonly used to represent the graphs. If the graph is represented using two-dimensional array then the representation is called as "adjacency matrix representation" and if the linked list is used to represent the graph then the representation is called as "adjacency list representation".

1. Adjacency matrix representation:

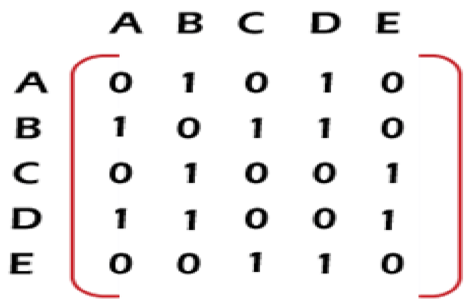
- A two-dimensional array of size NxN can be used to represent the graph where 'N' is the number of vertices of the graph.
- 'N' number of rows are used to represent the vertices and 'N' number of columns are used for each vertex.
- The matrix entries depend on the type of graph. In case of undirected or directed graph each row for the respective vertex contains the number of direct paths from it to each vertex (columns).
- If the graph is a weighted graph, then the matrix entries for each row will be the weight of the edge to other vertices (columns).

  Consider the following undirected graph:



Undirected Graph

In the given graph, the nodes 'A 'and 'D' are adjacent to node 'B' (there exist direct paths). A,B,E are adjacent to node D.D and C are adjacenet to node E and B .complete adjacency matrix is:



$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 0 & 1 & 0 \\ B & 1 & 0 & 1 & 1 & 0 \\ C & 0 & 1 & 0 & 0 & 1 \\ D & 1 & 1 & 0 & 0 & 1 \\ E & 0 & 0 & 1 & 1 & 0 \end{array}$$

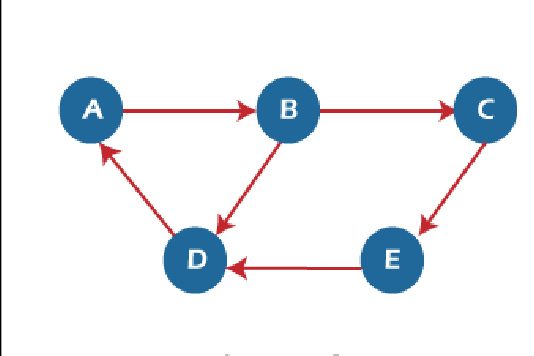
Adjacency Matrix

In this adjacency matrix each row is used for each node and each column is used for each node. Adj ij entry will be 1 if node 'i' is adjacent to 'j' otherwise it is 0 where 'Adj' is adjacency matrix, 'i' represents rows for nodes A,B,C,D and 'E' represents columns for nodes A,B,C,D, 'E'

Adjacency matrix for a directed graph

In a directed graph, edges represent a specific path from one vertex to another vertex. Suppose a path exists from vertex A to another vertex B; it means that node A is the initial node, while node B is the terminal node.

Consider the below-directed graph and try to construct the adjacency matrix of it.



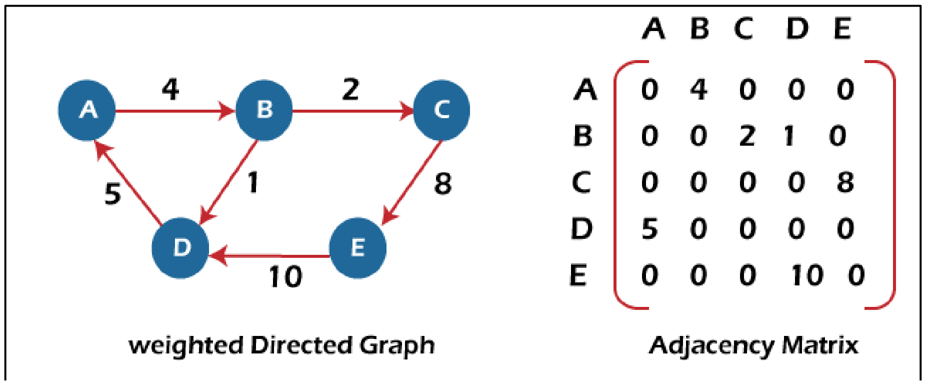
$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 0 & 0 & 0 \\ B & 0 & 0 & 1 & 1 & 0 \\ C & 0 & 0 & 0 & 0 & 1 \\ D & 1 & 0 & 0 & 0 & 0 \\ E & 0 & 0 & 0 & 1 & 0 \end{array}$$

Directed Graph Adjacency Matrix

In the above graph, we can see there is no self-loop, so the diagonal entries of the adjacent matrix are 0.

Adjacency matrix for a weighted directed graph

It is similar to an adjacency matrix representation of a directed graph except that instead of using the '1' for the existence of a path, here we have to use the weight associated with the edge. The weights on the graph edges will be represented as the entries of the adjacency matrix. We can understand it with the help of an example. Consider the below graph and its adjacency matrix representation. In the representation, we can see that the weight associated with the edges is represented as the entries in the adjacency matrix.



$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 4 & 0 & 0 & 0 \\ B & 0 & 0 & 2 & 1 & 0 \\ C & 0 & 0 & 0 & 0 & 8 \\ D & 5 & 0 & 0 & 0 & 0 \\ E & 0 & 0 & 0 & 10 & 0 \end{array}$$

In the above image, we can see that the adjacency matrix representation of the weighted directed graph is different from other representations. It is because, in this representation, the non-zero values are replaced by the actual weight assigned to the edges.

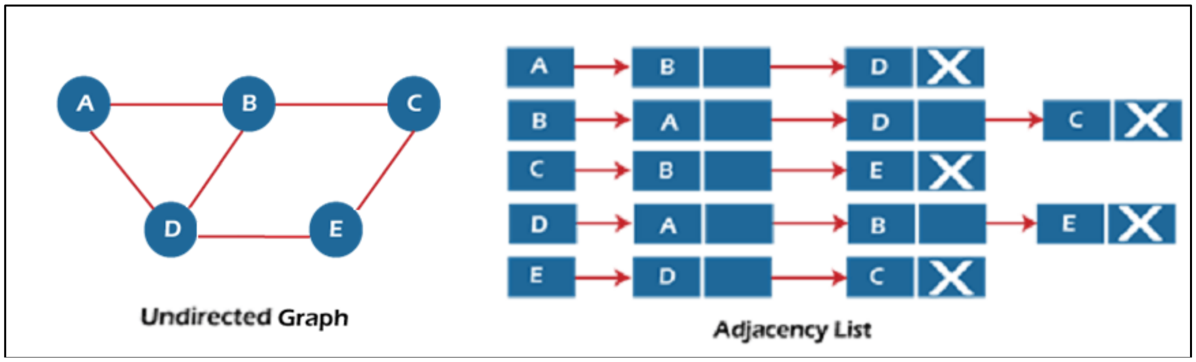
Adjacency matrix is easier to implement and follow. An adjacency matrix can be used when the graph is dense and a number of edges are large.

Though, it is advantageous to use an adjacency matrix, but it consumes more space. Even if the graph is sparse, the matrix still consumes the same space.

Linked list representation

An adjacency list is used in the linked representation to store the Graph in the computer's memory.

Let's see the adjacency list representation of an undirected graph.

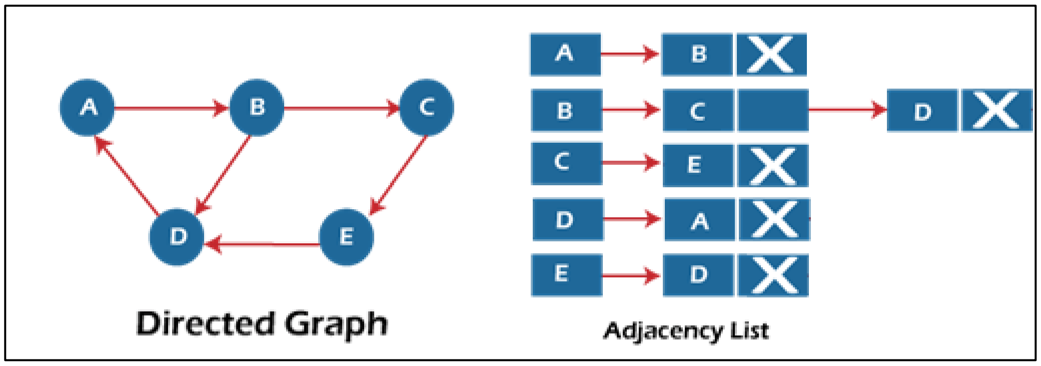


In the above figure, we can see that there is a linked list or adjacency list for every node of the graph. From vertex A, there are paths to vertex B and vertex D. These nodes are linked to nodes A in the given adjacency list.

An adjacency list is maintained for each node present in the graph, which stores the node value and a pointer to the next adjacent node to the respective node. If all the adjacent nodes are traversed, then store the NULL in the pointer field of the last node of the list.

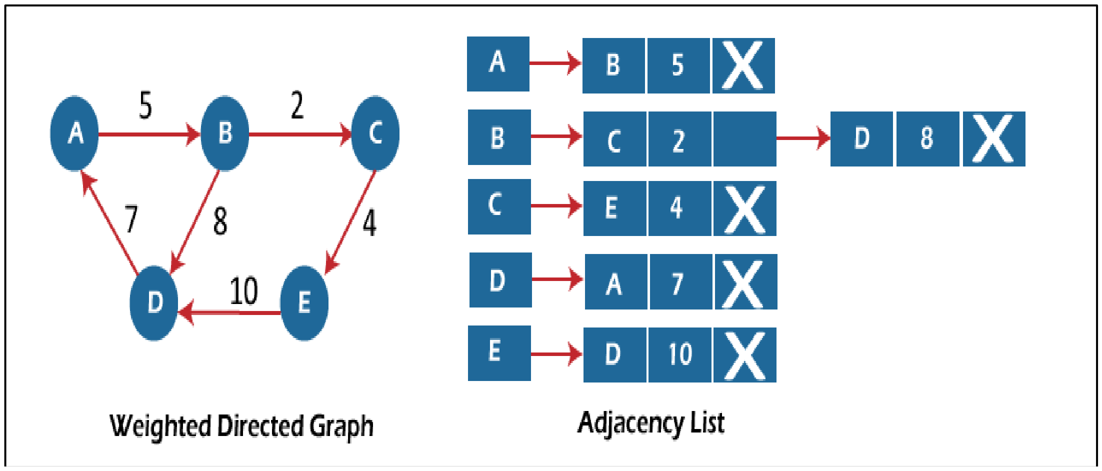
The sum of the lengths of adjacency lists is equal to twice the number of edges present in an undirected graph.

Now, consider the directed graph, and let's see the adjacency list representation of that graph.



For a directed graph, the sum of the lengths of adjacency lists is equal to the number of edges present in the graph.

Now, consider the weighted directed graph, and let's see the adjacency list representation of that graph.



In the case of a weighted directed graph, each node contains an extra field that is called the weight of the node.

Graph Traversal:

Graph traversal is a technique to visit the each nodes of a graph G.

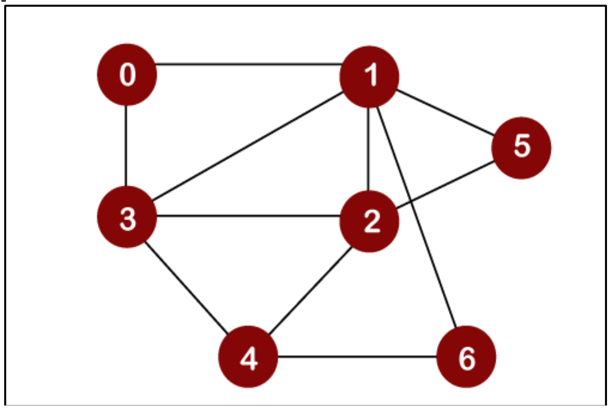
There are two graph traversal techniques:

1. Breadth-first search (BFS)
2. Depth-first search (DFS)

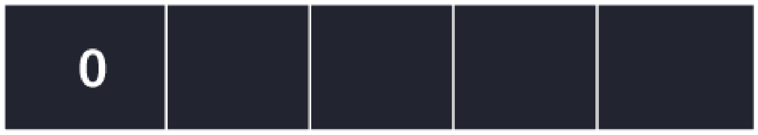
Breadth-first search

Breadth-first search graph traversal techniques use a queue data structure as an auxiliary data structure to store nodes for further processing. The size of the queue will be the maximum total number of vertices in the graph.

Let's consider the below graph for the breadth first search traversal.

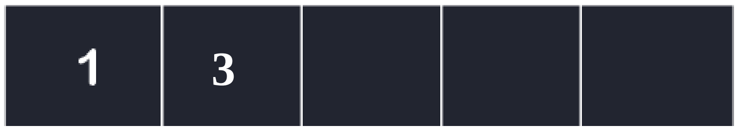


We can consider any element is root node. Suppose we consider node 0 as a root node. Therefore, the traversing would be started from node 0.



Once node 0 is removed from the Queue, it gets printed and marked as a ***visited node.***

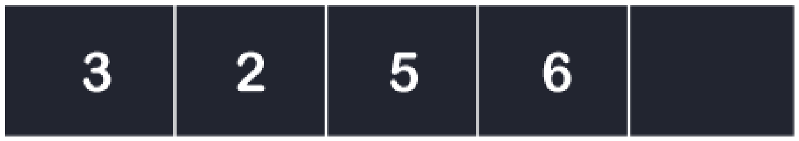
Once node 0 gets removed from the Queue, then the adjacent nodes of node 0 would be inserted in a Queue as shown below:



Result : 0

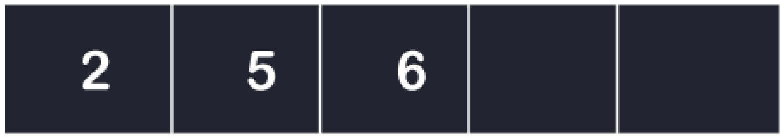
Now the node 1 will be removed from the Queue; it gets printed and marked as a visited node

Once node 1 gets removed from the Queue, then all the adjacent nodes of a node 1 will be added in a Queue. The adjacent nodes of node 1 are 0, 3, 2, 6, and 5. But we have to insert only unvisited nodes in a Queue. Since nodes 2, 6, and 5 are unvisited; therefore, these nodes will be added in a Queue as shown below:



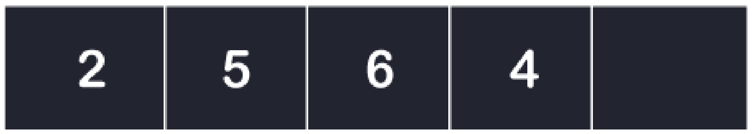
Result : 0 , 1

The next node is 3 in a Queue. So, node 3 will be removed from the Queue, it gets printed and marked as visited as shown below:



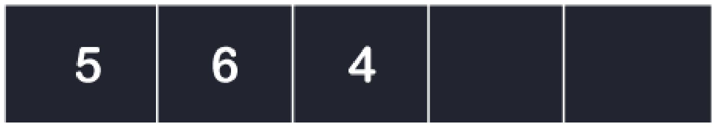
Result : 0, 1, 3

Once node 3 gets removed from the Queue, then all the adjacent nodes of node 3 except the visited nodes will be added in a Queue. The adjacent nodes of node 3 are 0, 1, 2, and 4. Since nodes 0, 1 are already visited, and node 2 is present in a Queue; therefore, we need to insert only node 4 in a Queue.



Result : 0, 1, 3

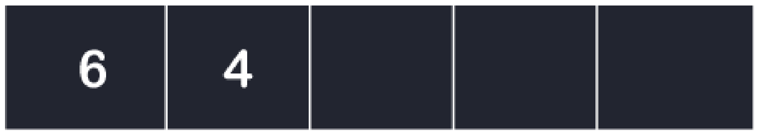
Now, the next node in the Queue is 2. So, 2 would be deleted from the Queue. It gets printed and marked as visited as shown below:



Result : 0, 1, 3, 2,

Once node 2 gets removed from the Queue, then all the adjacent nodes of node 2 except the visited nodes will be added in a Queue. The adjacent nodes of node 2 are 1, 3, 5, 6, and 4. Since the nodes 1 and 3 have already been visited, and 4, 5, 6 are already added in the Queue; therefore, we do not need to insert any node in the Queue.

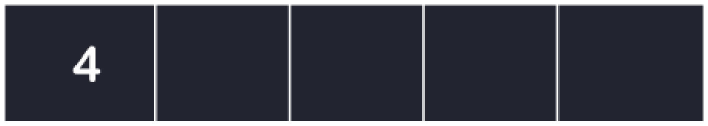
The next element is 5. So, 5 would be deleted from the Queue. It gets printed and marked as visited as shown below:



Result : 0, 1, 3, 2, 5

Once node 5 gets removed from the Queue, then all the adjacent nodes of node 5 except the visited nodes will be added in the Queue. The adjacent nodes of node 5 are 1 and 2. Since both the nodes have already been visited; therefore, there is no vertex to be inserted in a Queue.

The next node is 6. So, 6 would be deleted from the Queue. It gets printed and marked as visited as shown below:



Result : 0, 1, 3, 2, 5, 6

Once the node 6 gets removed from the Queue, then all the adjacent nodes of node 6 except the visited nodes will be added in the Queue. The adjacent nodes of node 6 are 1 and 4. Since the node 1 has already been visited and node 4 is already added in the Queue; therefore, there is not vertex to be inserted in the Queue.

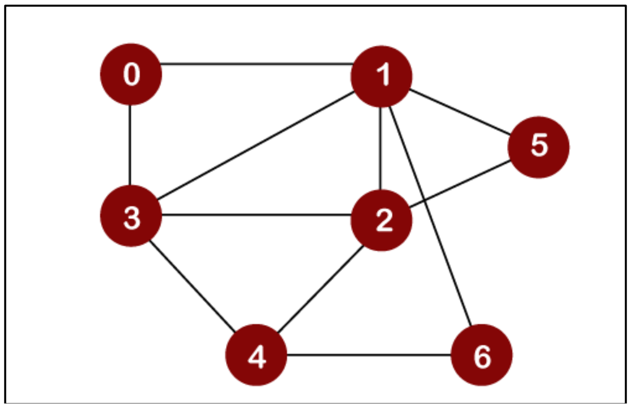
The next element in the Queue is 4. So, 4 would be deleted from the Queue. It gets printed and marked as visited.

Once the node 4 gets removed from the Queue, then all the adjacent nodes of node 4 except the visited nodes will be added in the Queue. The adjacent nodes of node 4 are 3, 2, and 6. Since all the adjacent nodes have already been visited; so, there is no vertex to be inserted in the Queue.

Depth-first search

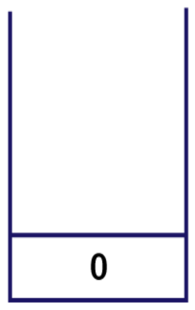
- DFS stands for Depth First Search, is one of the graph traversal algorithms that use Stack data structure. In DFS Traversal go as deep as possible of the graph and then backtrack once reached a vertex that has all its adjacent vertices already visited.
- Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack data structure to remember to get the next vertex to start a search when a dead end occurs in any iteration.
- In DFS, traversing can be started from any node, or we can say that any node can be considered as a root node until the root node is not mentioned in the problem.
- In the case of BFS, the element which is deleted from the Queue, the adjacent nodes of the deleted node are added to the Queue. In contrast, in DFS, the element which is removed from the stack, then only one adjacent node of a deleted node is added in the stack.

Let's consider the below graph for the Depth First Search traversal.

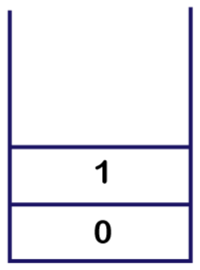


Consider node 0 as a root node.

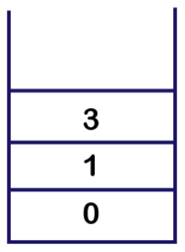
First, we insert the element 0 in the stack as shown below:



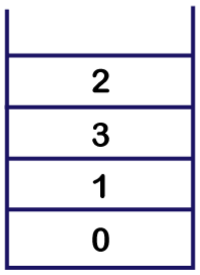
The node 0 has two adjacent nodes, i.e., 1 and 3. Now we can take only one adjacent node, either 1 or 3, for traversing. Suppose we consider node 1; therefore, 1 is inserted in a stack and gets printed as shown below:



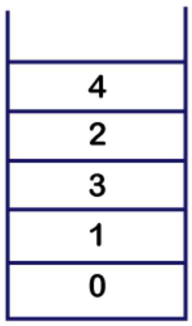
Now we will look at the adjacent vertices of node 1. The unvisited adjacent vertices of node 1 are 3, 2, 5 and 6. We can consider any of these four vertices. Suppose we take node 3 and insert it in the stack as shown below:



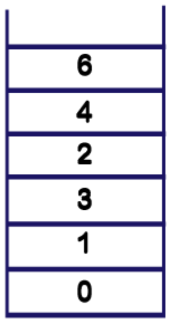
Consider the unvisited adjacent vertices of node 3. The unvisited adjacent vertices of node 3 are 2 and 4. We can take either of the vertices, i.e., 2 or 4. Suppose we take vertex 2 and insert it in the stack as shown below:



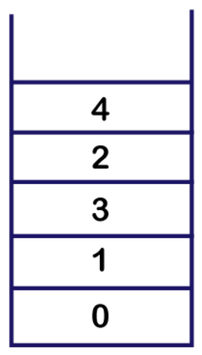
The unvisited adjacent vertices of node 2 are 5 and 4. We can choose either of the vertices, i.e., 5 or 4. Suppose we take vertex 4 and insert in the stack as shown below:

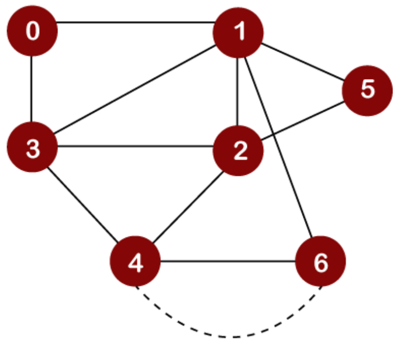


Now we will consider the unvisited adjacent vertices of node 4. The unvisited adjacent vertex of node 4 is node 6. Therefore, element 6 is inserted into the stack as shown below:

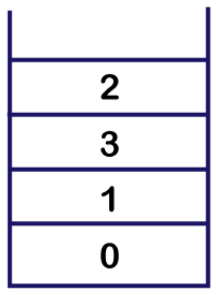


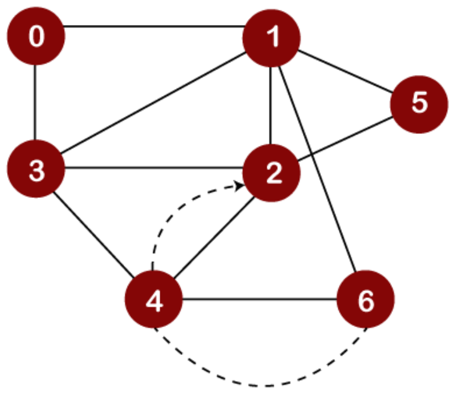
After inserting element 6 in the stack, we will look at the unvisited adjacent vertices of node 6. As there is no unvisited adjacent vertices of node 6, so we cannot move beyond node 6. In this case, we will perform **backtracking**. The topmost element, i.e., 6 would be popped out from the stack as shown below:



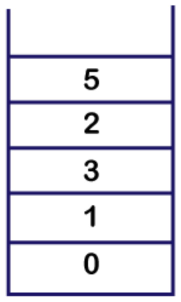


The topmost element in the stack is 4. Since there are no unvisited adjacent vertices left of node 4; therefore, node 4 is popped out from the stack as shown below:

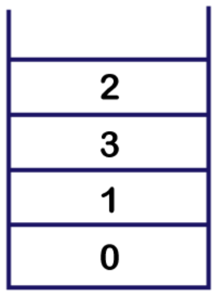




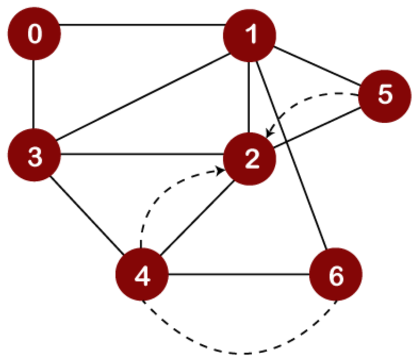
The next topmost element in the stack is 2. Now, we will look at the unvisited adjacent vertices of node 2. Since only one unvisited node, i.e., 5 is left, so node 5 would be pushed into the stack above 2 and gets printed as shown below:



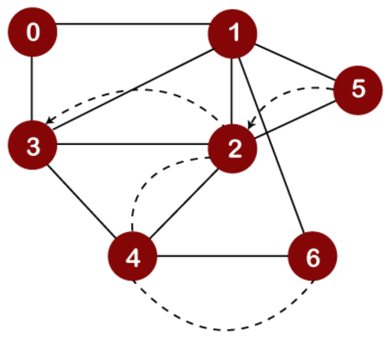
Now we will check the adjacent vertices of node 5, which are still unvisited. Since there is no vertex left to be visited, so we pop the element 5 from the stack as shown below:

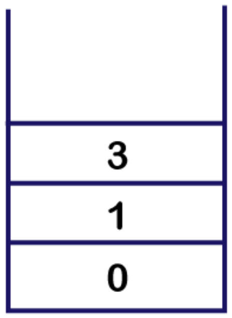


We cannot move further 5, so we need to perform backtracking. In backtracking, the topmost element would be popped out from the stack. The topmost element is 5 that would be popped out from the stack, and we move back to node 2 as shown below:

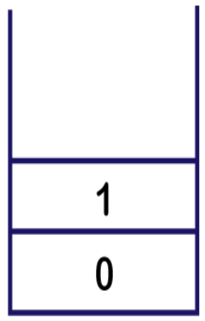


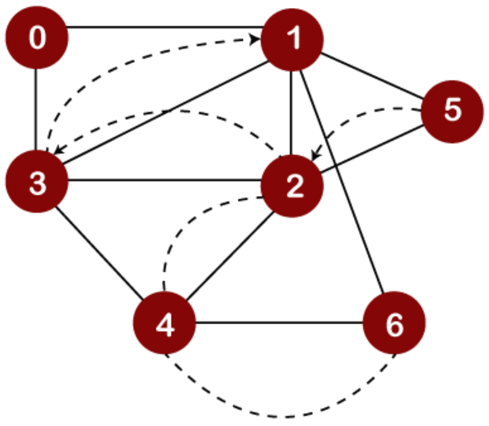
Now we will check the unvisited adjacent vertices of node 2. As there is no adjacent vertex left to be visited, so we perform backtracking. In backtracking, the topmost element, i.e., 2 would be popped out from the stack, and we move back to the node 3 as shown below:



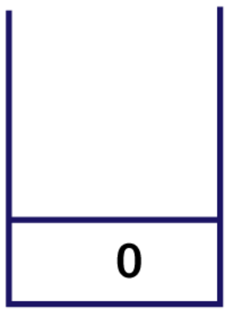


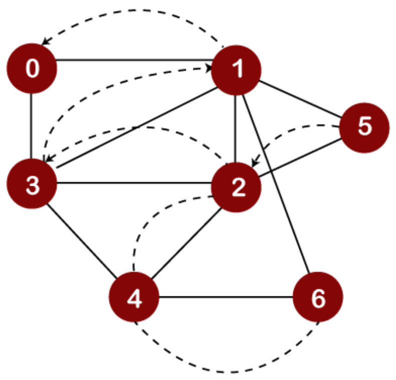
Now we will check the unvisited adjacent vertices of node 3. As there is no adjacent vertex left to be visited, so we perform backtracking. In backtracking, the topmost element, i.e., 3 would be popped out from the stack and we move back to node 1 as shown below:



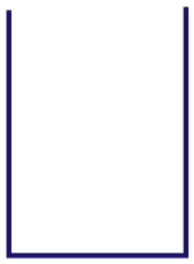


After popping out element 3, we will check the unvisited adjacent vertices of node 1. Since there is no vertex left to be visited; therefore, the backtracking will be performed. In backtracking, the topmost element, i.e., 1 would be popped out from the stack, and we move back to node 0 as shown below:





We will check the adjacent vertices of node 0, which are still unvisited. As there is no adjacent vertex left to be visited, so we perform backtracking. In this, only one element, i.e., 0 left in the stack, would be popped out from the stack as shown below:



Empty

As we can observe in the above figure that the stack is empty. So, we have to stop the DFS traversal here, and the elements which are printed is the result of the DFS traversal.

*****