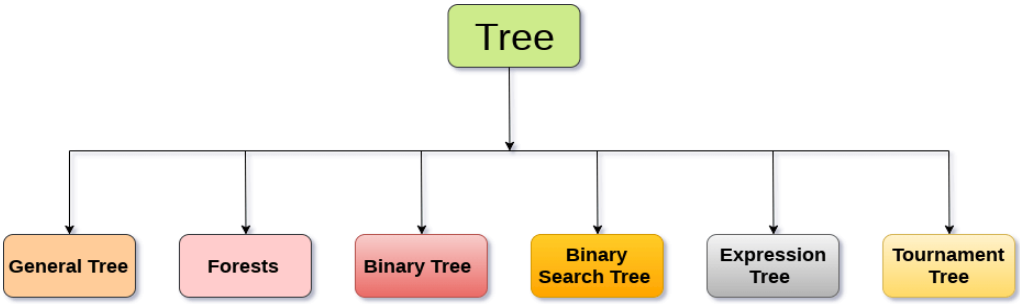
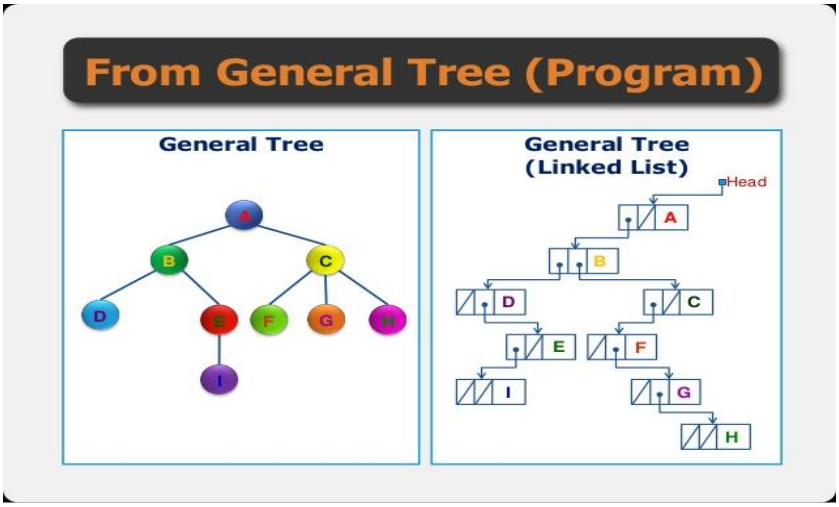
CLASSIFICATION OF TREES



1. **General Tree**
2. **Forest**
3. **Binary Tree**
4. **Binary-search Tree(BST)**
5. **Expression Tree**
6. **Tournament Tree**

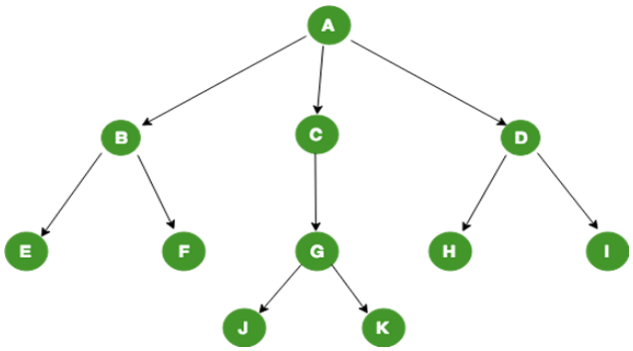
1. General Tree

➔ **A tree where each node can have zero or many children is called a general tree.**



➔ **In a general tree the degree of a node can be 0, 1 , 2, 3….**

➔ **In a general tree there is no limitation on the degree of any node i.e. on the number children of a node.**



➔ **The topmost node in general tree is its root node.**

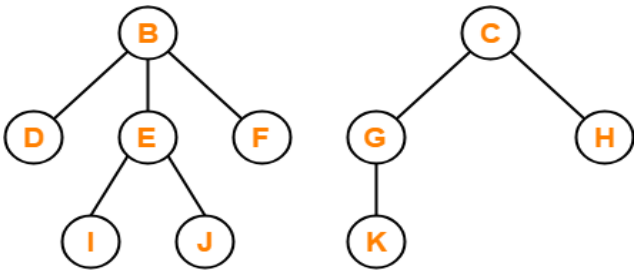
➔ **There are many sub trees in a general tree.**

➔ **The sub-trees of a general tree are un-ordered because, nodes of a general tree can't be ordered according a specific criteria.**

2. Forest:

A set of disjoint sub-trees is called a forest.

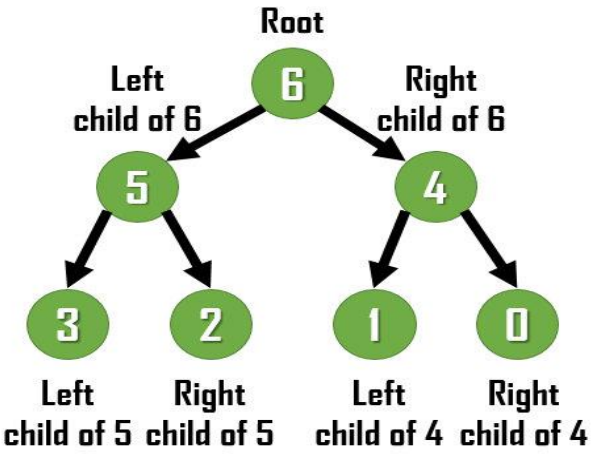
➔ **In a tree if you remove the root node then it becomes a Forest.**



Forest

3. Binary-Tree:

Binary-tree is a special type of tree consisting of a collection of nodes where each node can have only 0 or 1 or two children.



➔ **In binary tree any node can have maximum two children.**

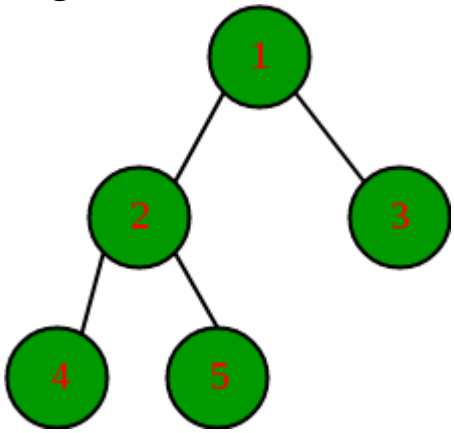
➔ **In a binary tree the node to the left of a node N is termed as left child of N and the node to the right of N is termed as right child of N.**

Types of Binary trees

1. Full binary tree/2-Tree/Strictly binary tree:

A binary tree where each node has either zero or exactly two children is called full binary tree.

➔ **The following figure is showing the structure of a full binary tree.**

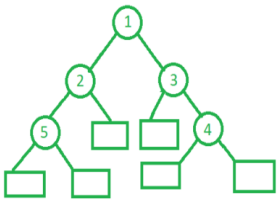


(Full binary / 2-tree)

➔ **In a full binary tree number of leaf nodes = no. of non-leaf nodes + 1.**

2. **Extended binary tree:**

   A binary tree where all null sub-trees of the original tree are replaced with special nodes called external nodes and other nodes are called internal nodes is called an extended binary tree.



Extended Binary Tree

3. **Skewed binary tree:**

   A binary tree where all nodes have either one child or no child is called a skewed binary tree.

   ➔ **A skewed binary tree where all the nodes are having a left child or no child at all is called left-skewed binary tree. It is left side dominated binary tree and all the right children remain as null.**

   ➔ **A skewed binary tree where all the nodes are having a right child or no child at all is called right-skewed binary tree. It is a right side dominated binary tree and all the left children remain as null.**

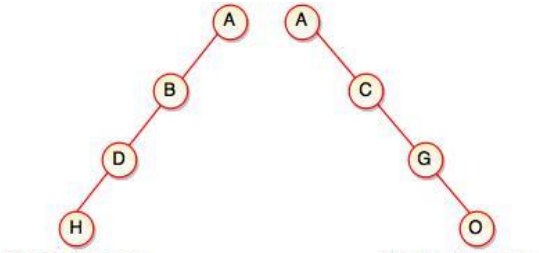


Fig. Left Skewed Binary Tree

Fig. Right Skewed Binary Tree

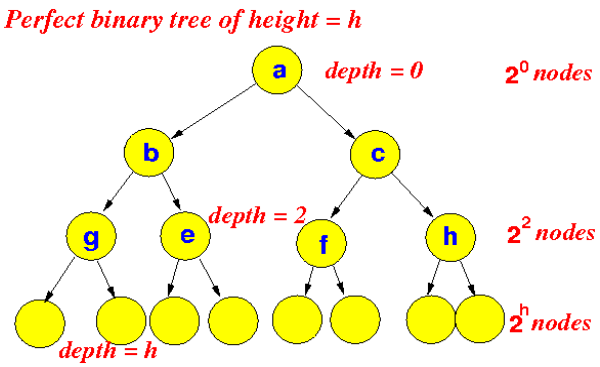
4. **Complete binary tree / Perfect binary tree:**

   A binary tree of depth d, where all the non-leaf nodes has exactly two children and all the leaf nodes are at the last level or maximum level i.e at level d is called a complete or perfect binary tree.

   ➔ **In a complete binary tree all levels up-to the last level d or h are completely filled.**

   ➔ **At a particular level L number of nodes = 2^L.**

Counting the number of nodes in binary tree:



In a complete binary tree total number of nodes =

Sum of total number of nodes in each level from 0 to h or d

$= 2^0 + 2^1 + 2^2 + 2^3 + \ldots\ldots + 2^{h-1} + 2^h$

$= (2^{h+1} - 1) / (2-1) = (2^{h+1} - 1)/1$

Total no. of nodes in a full binary tree: $N = (2^{h+1} - 1)$

L is the maximum level of a tree

d is depth of a tree

h is height of a tree

We know that maximum level = depth =height in a tree

Therefore, we can also write:

Total number of nodes in a full binary tree is: $N = (2^{h+1} - 1)$

Or $N = (2^{d+1} - 1)$

Or $N = (2^{L+1} - 1)$

If total number of nodes in a full binary tree is given then to calculate height or depth of the tree, we can use the above equation.

$N = 2^{h+1} - 1$

$\Rightarrow 2^{h+1} = N + 1$

Applying log on both sides we got

$\Rightarrow \text{Log } 2^{h+1} = \log (N + 1)$

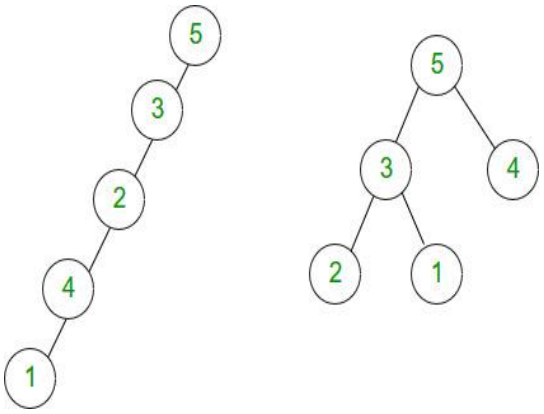
$\Rightarrow (h+1) \log 2 = \log (N+1)$

$\Rightarrow h + 1 = \log_2 (N+1)$

So, height or depth of a perfect binary tree $h = \log_2 (N+1) - 1$

➔ **The maximum height or depth h of a binary tree consisting of N nodes is equal to $h \text{ or } d = n-1$**

Minimum height or depth $h = \text{floor of } [\log_2 (N)]$



For example: in the above figure number of nodes in binary tree n is 4

So maximum height or depth of the tree = n-1 = 3

If number of nodes n is 5 then minimum height = floor of $\log_2 N$

= floor of $\log_2 5$

= floor of 2.321

Height h = 2

- ➔ **Number of leaf nodes in a complete binary tree = 2^h, because the leaf nodes can appear only at last level i.e. at level h.**
- ➔ **Number of non-leaf nodes = total number of nodes – number of leaf nodes.**

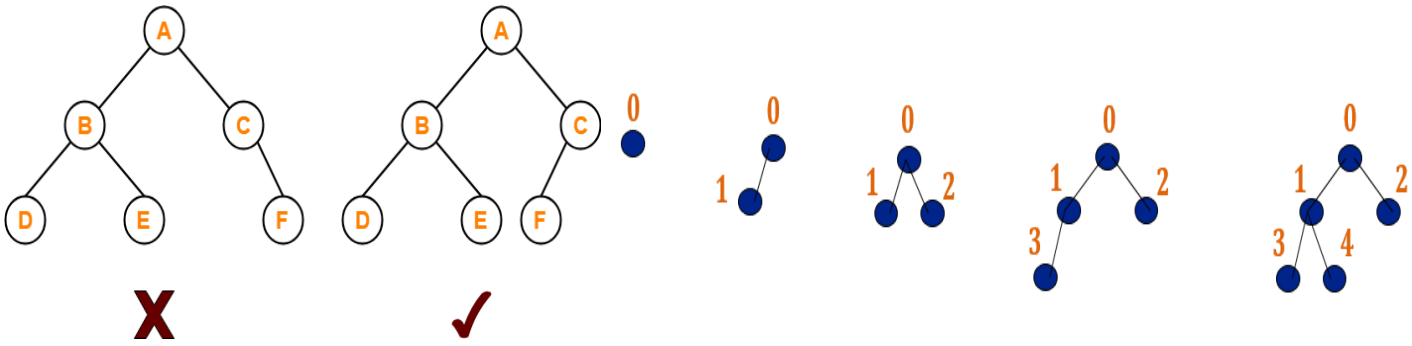
  $(2^{h+1} – 1) – 2^h = 2^h – 1$

  So number of non-leaf nodes = $2^h – 1$

5. **Almost-complete binary tree:**

   A binary tree of depth d where all levels except possibly the last level d are completely full i.e contains maximum number of nodes and leaf node nodes at the level d or last level appear to the left side as far as possible is called almost complete binary tree.

   - ➔ **Height or depth of an almost complete binary tree consisting of N nodes is d or h = log N.**
   - ➔ **Number of leaf nodes = ceiling of N/2.**



✗ ✓

APPLICATION OF BINARY TREES

➔Used in searching algorithms to make the search operation faster.

➔To represent and evaluate arithmetic expressions.

➔Used in game programming and 3-D graphics applications

➔Use to represent the players and winners in tournament.

➔Used in data compression algorithms and file system of operating systems.

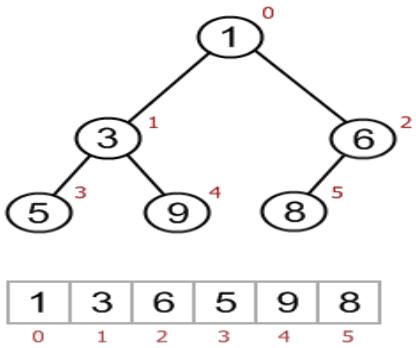
Implementation of binary trees

Binary trees can be implemented in program using two methods:

1. **Array implementation(Static implantation)**
2. **Linked list implementation(Dynamic implementation)**

Array implementation of binary trees (Static implementation):

→In array implantation the nodes in a binary tree are numbered starting from 0 to n-1 or from 1 to n, where n is number of nodes in the tree as shown in the following figure:

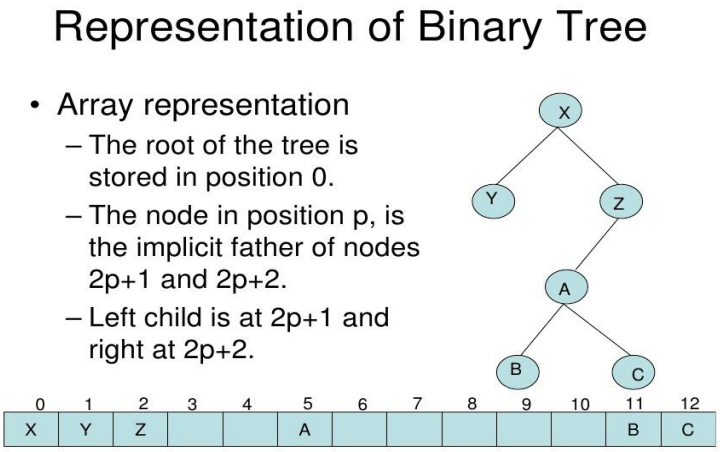


→Numbering the nodes can be started from 0 to n-1 or from 1 to n, where n is total number of nodes in the binary tree.

Here assuming the nodes are numbered from 0 to n-1 in the tree:

Steps to store tree elements in an array:

→The array that will hold the information of nodes of the tree should be declared with maximum size MAX= $2^{h+1} - 1$, where h→ height or depth of the binary tree.



→The root node always lies at index 0 i.e. at 0^{th} position of the array.

→So, the 'ROOT' pointer always refers to 0^{th} index of the array i.e. the value of 'ROOT' pointer should be 0.

→If a parent node N lies at i^{th} position in the array then, its left child will lie at $(2*i+1)^{th}$ position in the array.

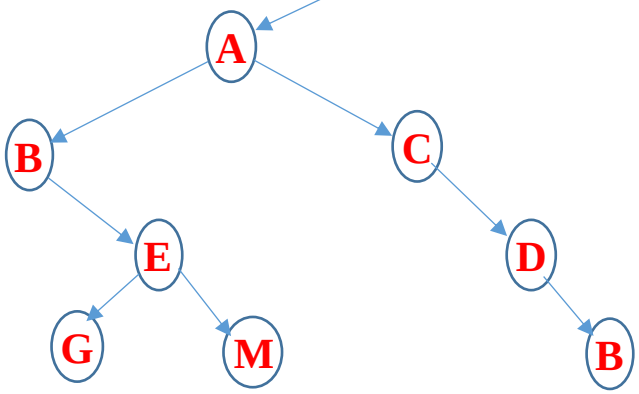
→The right child of parent node N will lie at $(2*i+2)^{th}$ position in the array.

→If a node lies at i^{th} position in the array then its parent node lies at $((i-1)/2)^{th}$ position .

→If a particular node is index i and left or right child or both child are empty then its left or right child position in the array remains empty i.e remains unused.

Assuming array index start from 1, and nodes are numbered from 1 to n, where n is the number of nodes in the tree: then the root will lie at 1^{st} index of the array.

(Fig: 1) Root of the tree



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	C		E		D			G	M				B

(Array representation of the above tree in fig: 1)

→The root node always lies at index 1 i.e. at 1^{st} position of the array.

→So, the 'ROOT' pointer always refers to 1st index of the array i.e. the value of 'ROOT' pointer should be 1.

→If a parent node N lies at i^{th} position in the array then, its left child will lie at $(2*i)^{th}$ position in the array.

→The right child of parent node N will lie at $(2*i+1)^{th}$ position in the array.

→If a node lies at i^{th} position in the array then its parent node lies at $(i/2)^{th}$ position .

→If a particular node is index i and left or right child or both child are empty then its left or right child position in the array remains empty i.e. remains unused.

The above set of steps should be followed to store the elements of a binary tree in array representation.

Linked-list implementation (Dynamic implementation):

In the implementation a binary tree consists of a collection of nodes 1where each node has 3 parts:

→Information part (data to be stored in the node)

→Left child link (A reference to left child)

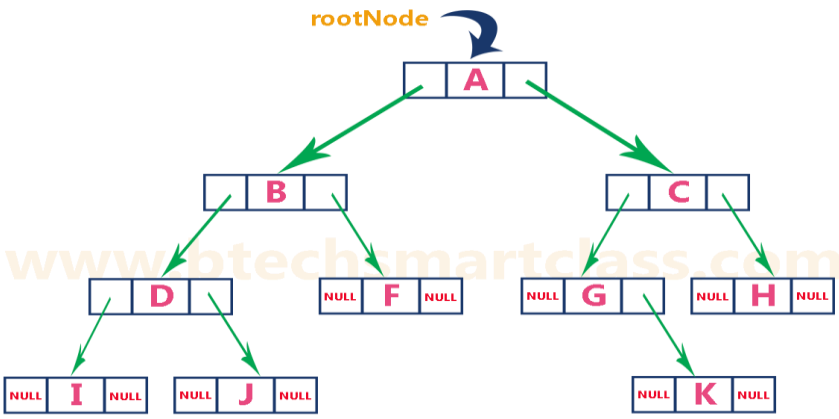
→Right child link (A reference to right child)

(Fig: 2 Structure of a node in linked list representation of binary trees)

→A node class type reference variable called as 'ROOT' always refers to root node of the tree.

→If the tree is empty, then 'ROOT' refers to null.

→If left child of a node is empty, then left link part of that node contains a null value, if right child is empty, right link part contains a null value.

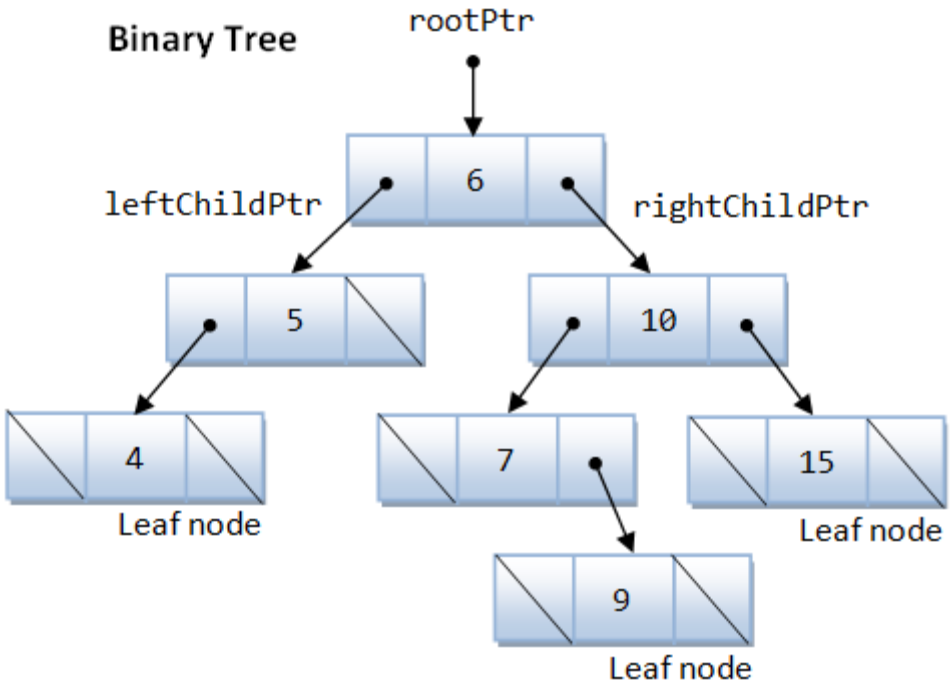


→To create a binary tree using linked list representation, we should declare a node type class that will represent the nodes of the tree as follow:

```
class Tree_node      /*A node of this class type can store an integer value only in the info part*/
{
      char info;
      Tree_node Lchild_link;
      Tree_node Rchild_link;
}

Public class BINARY_TREE_LL_DEMO
{
       Static Tree_node ROOT=null;  /* creates an empty binary tree*/
```

```
    Public static void main (String [] args)
    {
        /*body of the main method*/
    }

    /*here define all other methods on tree*/

} /*End of  BINARY_TREE_LL_DEMO CLASS*/
```

/*Fig: 3 A binary tree which uses the above class declaration containing a collection of integers in the nodes*/

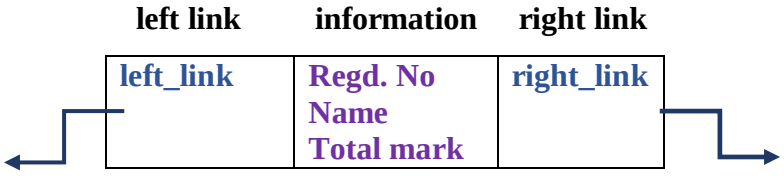


→You can also declared a node type class for a binary tree as follows:

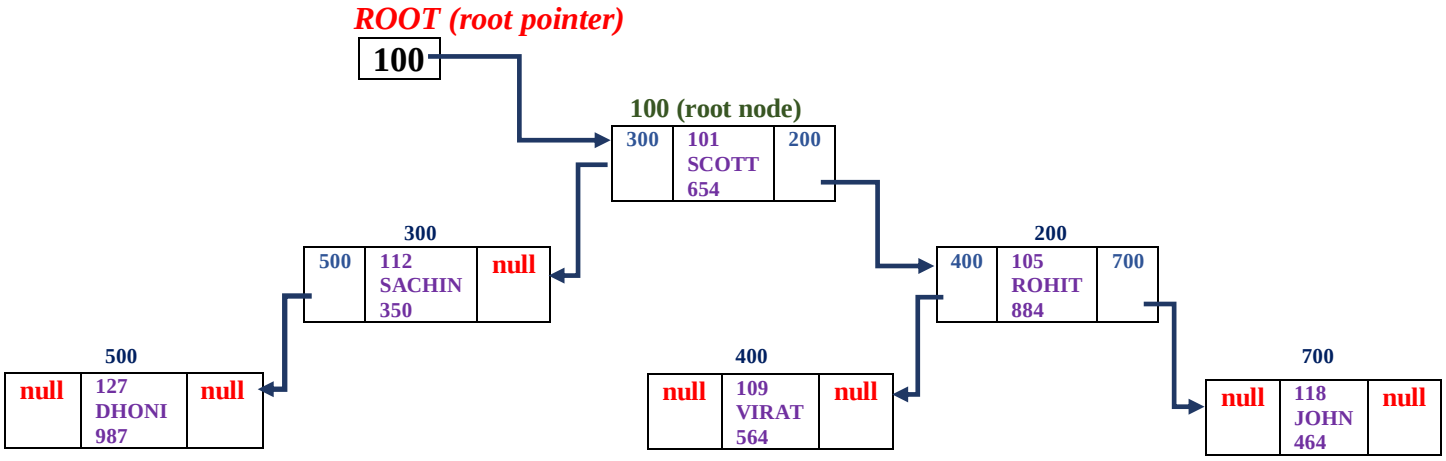
```
class TNODE        /*A node of this class type can store only a character value in the info part*/
{
    char info;
    TNODE Lchild_link;
    TNODE Rchild_link;
}
```

→To store student information you can declare a class as follows:

```
class TNODE      /*A node of this class type can store information of students(regd. no, name)*/
{
    int  regd_no;
    String  name;
    float  total_mark;
    TNODE left_link;
    TNODE right_link;
}
```



(Fig: 4 Student node structure in a binary tree)



(Fig: 5 Structure of a binary tree containing student information in linked list representation)

Binary tree traversal algorithms:

Visiting each and every node only once starting from the root node of binary tree is called tree traversal.

→There are three traversal algorithms to traverse a binary tree which are differing in terms of order in which nodes are visited:

1. **In-order traversal**
2. **Pre-order traversal**
3. **Post-order traversal**

→All the above algorithms visit a binary tree recursively.

In-order traversal algorithm:

Algorithm IN-ORDER (TREE) /* 'TREE' is address of root node of a binary tree*/

Step 1: Repeat Steps 2 to 4 while TREE != NULL

Step 2: IN-ORDER(TREE -> LEFT) /*Traverse the left sub-tree recursively*/

Step 3: Print TREE. INFO /*print information of root node*/

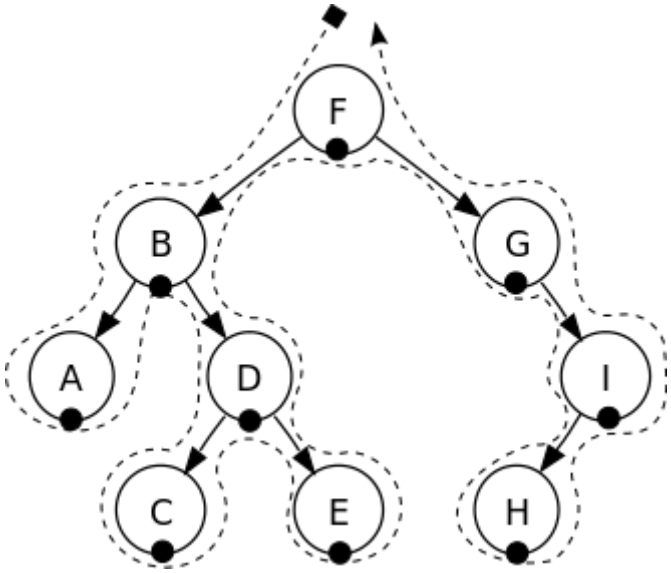
Step 4: IN-ORDER(TREE -> RIGHT) /*Traverse the right sub-tree recursively*/
[END OF LOOP]

Step 5: END

→According to in-order algorithm visit the left sub-tree of root recursively, then visit the root then visit the right sub-tree of the root recursively i.e. in the following sequence a binary tree is visited using in-order traversal algorithm:

In-order traversal sequence: *Left sub-tree →visit root →right sub-tree*

Example→1: consider the following tree in fig: 1

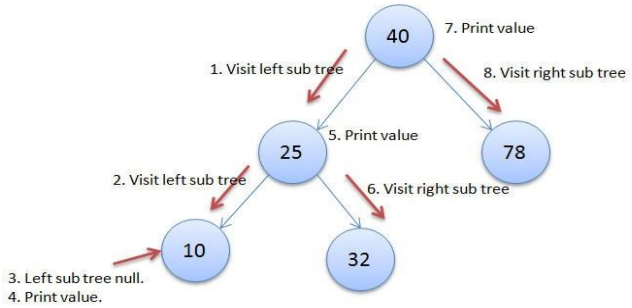


/*In the above Fig: 1 follow the dotted line and visit the node when you find black dot*/

According to the above sequence at first the left most leaf node of the tree in Fig: A is visited.
Output: A
→Then root of A i.e. node B
Output: A B
→Then right sub-tree of B whose root is D: so visit left sub-tree of D i.e. C
Output: A B C
→Then visit D
Output: A B C D
→Then visit right sub-tree of D i.e. E
Output: A B C D E
→Now left-sub tree of root node F completed, so visit root node F
Output: A B C D E F
→Then go to right sub-tree of Root F
→The left sub-tree of G is empty so visit G
Output: A B C D E F G
→Then visit right sub-tree of G
→Visit the left sub-tree of I i.e. H
Output: A B C D E F G H
→Then visit I
Output: A B C D E F G H I
→There is no right sub-tree of I
→Now there is no more nodes to visit, so stop here
Therefore In-order traversal sequence of the binary tree in Fig: A is:

A B C D E F G H I

Example→2: consider the following binary tree in Fig: 2



The above INORDER traversal gives: **10, 25, 32, 40, 78**

The in-order traversal of the above binary tree in Fig: 2 is→ 10 25 32 40 78

Pre-order traversal algorithm:

Algorithm PRE-ORDER (TREE) /* 'TREE' is address of root node of a binary tree*/

Step 1: Repeat Steps 2 to 4 while TREE != NULL

Step 2: Print TREE. INFO /*print information of root node*/

Step 3: PRE-ORDER(TREE.LEFT) /*Traverse the left sub-tree recursively*/

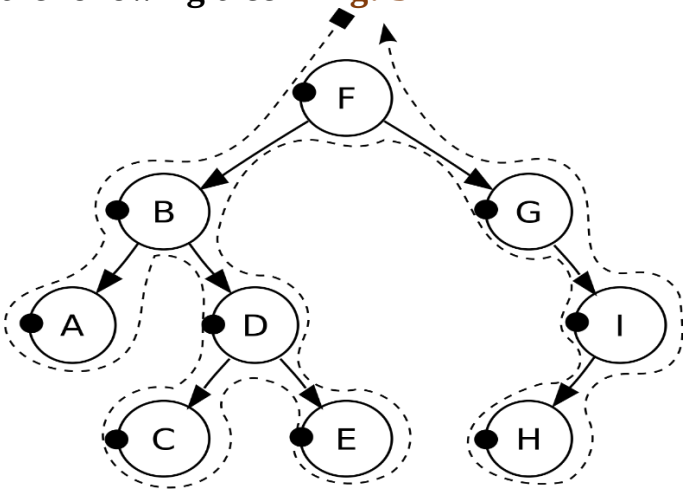
Step 4: PRE-ORDER(TREE.RIGHT) /*Traverse the right sub-tree recursively*/
[END OF LOOP]

Step 5: END

→According to pre-order algorithm visit the root node is visited first, then left sub-tree of root node is visited recursively, then visit right sub-tree of the root node recursively i.e. in the following sequence a binary tree is visited using pre-order traversal algorithm:

In-order traversal sequence: *Visit the root→Left sub-tree→right sub-tree*

Example→1: consider the following tree in fig: 3



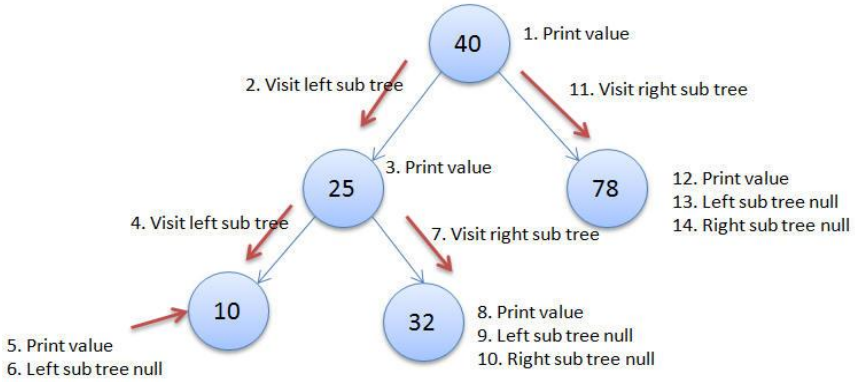
/*In the above Fig: 3 follow the dotted line and visit the node when you find black dot*/

According to the above sequence at first root node of the tree in Fig: 3 is visited.
Output: F
→Then visit the left sub-tree of root node F completely, so visit root of the left sub-tree of F i.e. node B

Output: F B
→Then visit left sub-tree of B, so visit A
Output: F B A
→Then visit root of right sub-tree of B i.e. D
Output: F B A D
→Then visit left sub-tree of D i.e. C
Output: F B A D C
→Then go to right sub-tree of D, so visit E
Output: F B A D C E
→Now left-sub tree of root node F completed, so go to right sub-tree of the root node F
→Then visit root of right sub-tree of F i.e. G
Output: F B A D C E G
→The left sub-tree of G is empty, so go to right sub-tree of G and visit its root i.e. I
Output: F B A D C E G I
→Visit the left sub-tree of I i.e. H
Output: F B A D C E G I H
→There is no right sub-tree of I
→Now there is no more nodes to visit, so stop here

Therefore pre-order traversal of the binary tree in Fig: 3 is→F B A D C E G I H

Example→2: consider the following binary tree in Fig: 4



The above PREORDER traversal gives: **40, 25, 10, 32, 78**

Therefore pre-order traversal sequence of the binary tree in Fig: 2 is: 40 25 10 32 78

Post-order traversal algorithm:

Algorithm POST-ORDER (TREE) /* 'TREE' is address of root node of a binary tree*/

Step 1: Repeat Steps 2 to 4 while TREE != NULL

Step 2: POST-ORDER(TREE -> LEFT) /*Traverse the left sub-tree recursively*/

Step 3: POST-ORDER(TREE -> RIGHT) /*Traverse the right sub-tree recursively*/

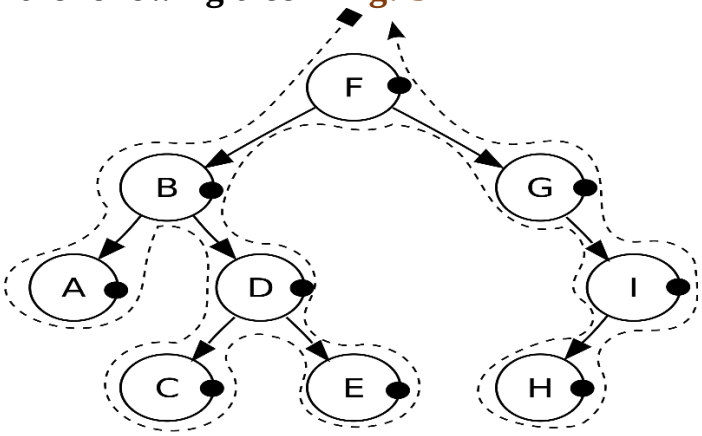
Step 4: Print TREE. INFO /*print information of root node*/
[END OF LOOP]

Step 5: END

→According to post-order algorithm at first visit left sub-tree of the root node recursively, then right sub-tree of root node is visited recursively, then visit the root node finally i.e. in the following sequence a binary tree is visited using post-order traversal algorithm:

post-order traversal sequence: *Left sub-tree→right sub-tree→visit the root*

Example→1: consider the following tree in fig: 5

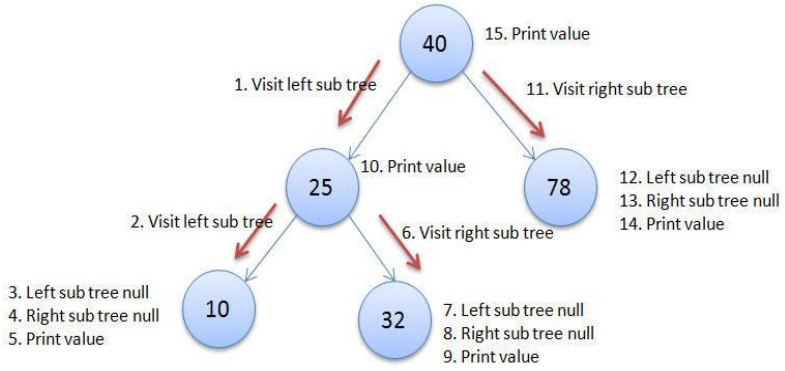


/*In the above Fig: 5 follow the dotted line and visit the node when you find black dot*/

According to the above sequence at first left sub-tree of the root node Fig: 5 is visited recursively i.e. the left most leaf node A is visited first.
Output: A
→Then visit the right sub-tree B rooted at D, so visit left sub-tree of D i.e. C
Output: A C
→Then visit right sub-tree of D, so visit E
Output: A C E
→Then visit D and B respectively
Output: A C E D B
→Now left sub-tree of F is over, so visit right sub-tree of F rooted at G
→There is no left sub-tree of G so go to its right sub-tree rooted at I
→Visit left sub-tree of I i.e. the node H
Output: A C E D B H
→There is no right sub-tree of I , so visit I
Output: A C E D B H I
→Then visit G
Output: A C E D B H I G
→Now left-sub tree and right sub-tree of the root node F is over, so finally visit the root node F
Output: A C E D B H I G F
→Now there is no more nodes to visit, so stop here

Therefore post-order traversal of the binary tree in Fig: 5 is→ A C E D B H I G F

Example→2: consider the following binary tree in Fig: 6



The above POSTORDER traversal gives: **10, 32, 25, 78, 40**

Therefore post-order traversal sequence of the binary tree in Fig: 6 is: 10 32 25 78 40