Unit-IV:

TREES: Introduction, Terminology, Representation of Trees, Binary Trees, The Abstract Data Type, Properties of Binary Tress, Binary Tree Representations, Binary Tree Traversals. Binary Search Trees, Definition, Searching a Binary Search Tree, Insertion into a Binary Search Tree, Deletion from a Binary Search Tree, Height of Binary Search Tree.

INTRODUCTION:

- Tree is an example of a non-linear data structure.

Definition:

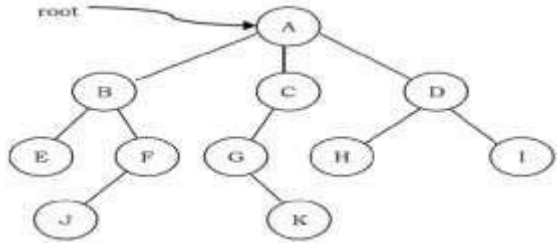
A **tree** is a hierarchical representation of a finite set of one or more data elements called nodes such that:

- There is a special node called the root of the tree.
- The nodes other than the root node form an ordered pair of disjoint sub trees(left and right sub trees).

=>. Each node can have at most one link coming into it.

=>. Trees are recursive structures. Each child node is itself the root of a sub tree.

=>.At the bottom of the tree are leaf nodes, which have no children.

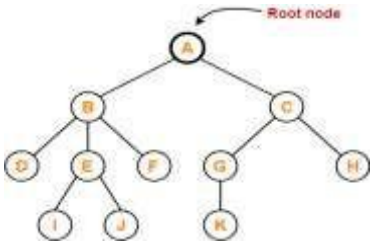


TERMINOLOGY:

1. Root-

- The first node from where the tree originates is called as a **root node**.
- In any tree, there must be only one root node.

Example-

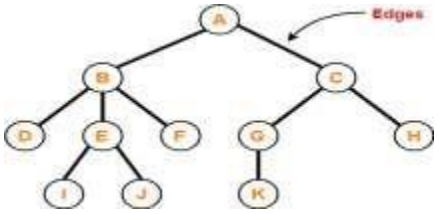


Here, node A is the only root node.

2. Edge-

- The connecting link between any two nodes is called as an **edge**.
- In a tree with n number of nodes, there are exactly (n-1) number of edges.

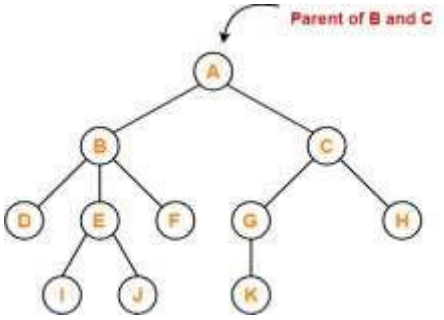
Example-



3. Parent-

- The node which has a branch from it to any other node is called as a **parent node**.
- In other words, the node which has one or more children is called as a parent node.
- In a tree, a parent node can have any number of child nodes.

Example-



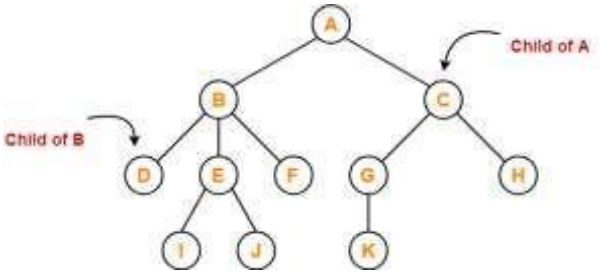
Here,

- Node A is the parent of nodes B and C
- Node B is the parent of nodes D, E and F
- Node C is the parent of nodes G and H
- Node E is the parent of nodes I and J
- Node G is the parent of node K

4. Child-

- The node which is a descendant of some node is called as a **child node**.
- All the nodes except root node are child nodes.

Example-



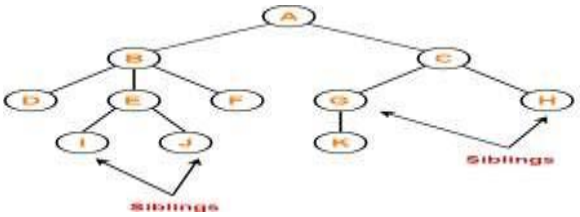
Here,

- Nodes B and C are the children of node A
- Nodes D, E and F are the children of node B
- Nodes G and H are the children of node C
- Nodes I and J are the children of node E
- Node K is the child of node G

5. Siblings-

- Nodes which belong to the same parent are called as **siblings**.
- In other words, nodes with the same parent are sibling nodes

Example



Here,

- Nodes B and C are siblings
- Nodes D, E and F are siblings
- Nodes G and H are siblings
- Nodes I and J are siblings

6. Degree-

- **Degree of a node** is the total number of children of that node.
- **Degree of a tree** is the highest degree of a node among all the nodes in the tree.

Example-

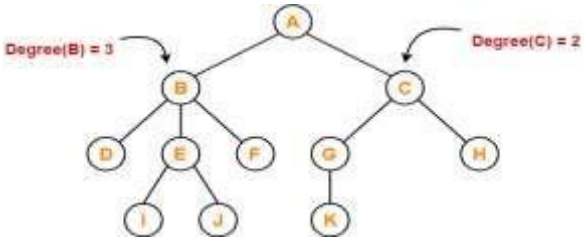
Here,

- Degree of node A = 2
- Degree of node B = 3
- Degree of node C = 2
- Degree of node D = 0
- Degree of node E = 2
- Degree of node F = 0
- Degree of node G = 1
- Degree of node H = 0
- Degree of node I = 0
- Degree of node J = 0
- Degree of node K = 0

7. Internal Node-

- The node which has at least one child is called as an **internal node**.
- Internal nodes are also called as **non-terminal nodes**.
- Every non-leaf node is an internal node.

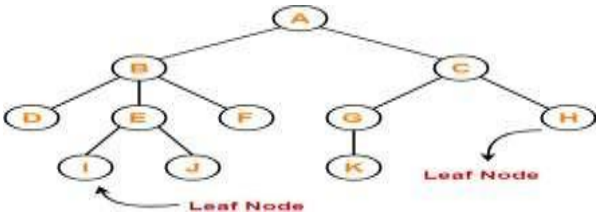
Example-



8. Leaf Node-

- The node which does not have any child is called as a **leaf node**.
- Leaf nodes are also called as **external nodes** or **terminal nodes**.

Example-

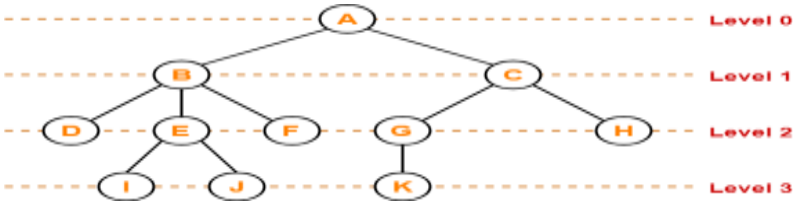


Here, nodes D, I, J, F, K and H are leaf nodes.

9. Level-

- In a tree, each step from top to bottom is called as **level of a tree**.
- The level count starts with 0 and increments by 1 at each level or step.

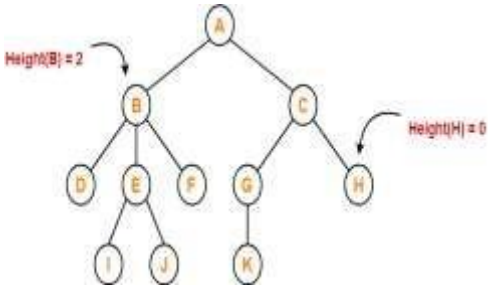
Example-



10. Height-

- Total number of edges that lies on the longest path from any leaf node to a particular node is called as **height of that node**.
- **Height of a tree** is the height of root node or maximum level of a tree is called height of a tree
- **Or total number of levels – 1.**
- Height of all leaf nodes = 0

Example-



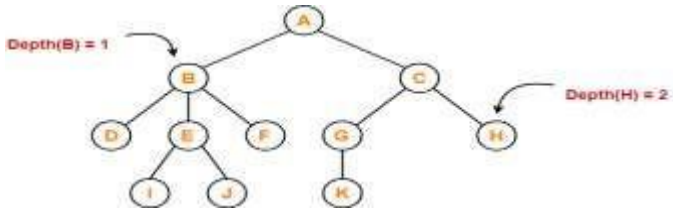
Here,

- Height of node A = 3
- Height of node B = 2
- Height of node C = 2
- Height of node D = 0
- Height of node E = 1
- Height of node F = 0
- Height of node G = 1
- Height of node H = 0
- Height of node I = 0
- Height of node J = 0
- Height of node K = 0

11. Depth-

- Total number of edges from root node to a particular node is called as **depth of that node**.
- **Depth of a tree** is the total number of edges from root node to a leaf node in the longest path.
- Depth of the root node = 0
- The terms “level” and “depth” are used interchangeably.

Example-



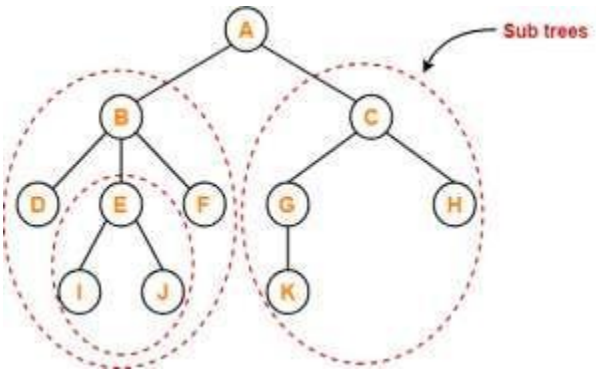
Here,

- Depth of node A = 0
- Depth of node B = 1
- Depth of node C = 1
- Depth of node D = 2
- Depth of node E = 2
- Depth of node F = 2
- Depth of node G = 2
- Depth of node H = 2
- Depth of node I = 3
- Depth of node J = 3
- Depth of node K = 3

12. Subtree-

- In a tree, each child from a node forms a **subtree** recursively.
- Every child node forms a subtree on its parent node.

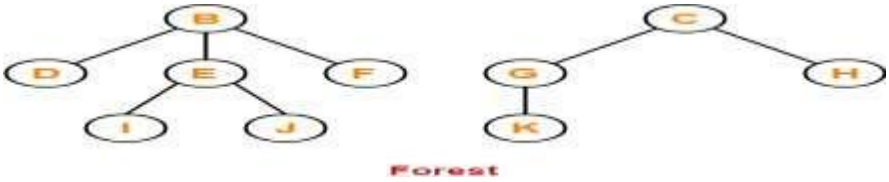
Example-



13. Forest-

A forest is a set of disjoint trees.

Example-



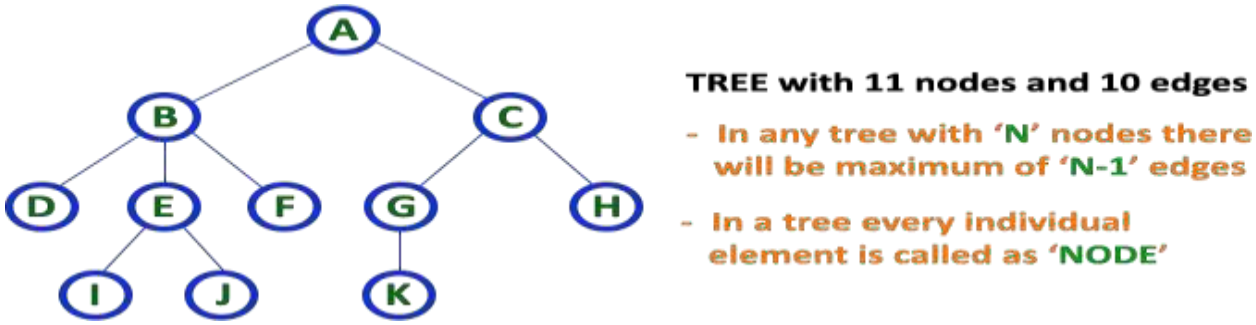
Forest

REPRESENTATION OF TREES:

A tree data structure can be represented in two methods. Those methods are as follows...

1. **List Representation**
2. **Left Child - Right Sibling Representation**

Consider the following tree...

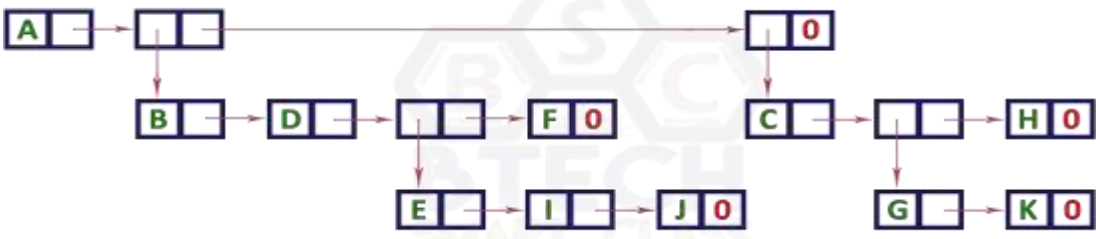


1. List Representation

In this representation, we use two types of nodes

- one for representing the node with data called 'data node' and
- another for representing only references called 'reference node'.
- We start with a 'data node' from the root node in the tree. Then it is linked to an internal node through a 'reference node' which is further linked to any other node directly. This process repeats for all the nodes in the tree.

The above example tree can be represented using List representation as follows...

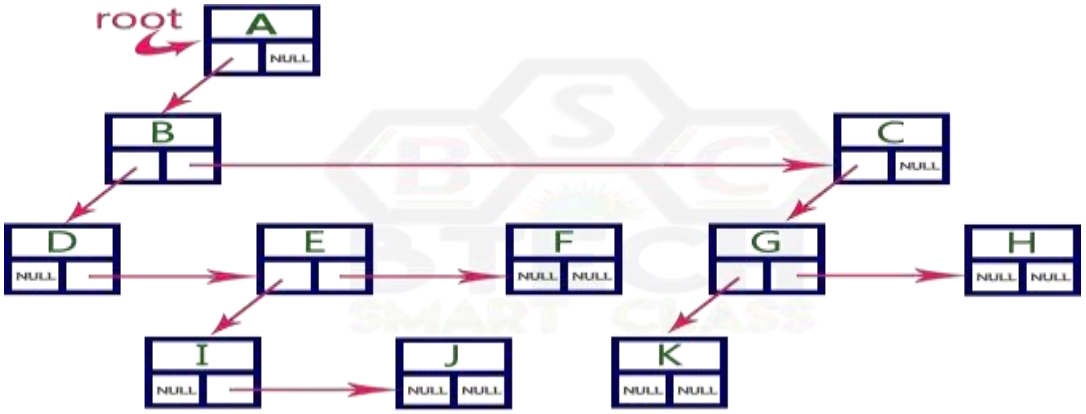


2. Left Child - Right Sibling Representation

In this representation, we use a list with one type of node which consists of three fields namely

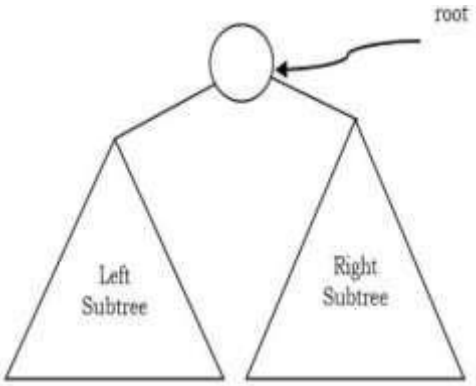
- Data field,
- Left child reference field
- and Right sibling reference field.
- Data field stores the actual value of a node, left reference field stores the address of the left child and right reference field stores the address of the right sibling node. Graphical representation of that node is as follows...

In this representation, every node's data field stores the actual value of that node. If that node has left a child, then left reference field stores the address of that left child node otherwise stores NULL. If that node has the right sibling, then right reference field stores the address of right sibling node otherwise stores NULL.
The above example tree can be represented using Left Child - Right Sibling representation as follows...



Binary Trees:

In general, tree nodes can have any number of children. In a binary tree, each node can have at most two children. A binary tree is either **empty** or consists of a node called the **root** together with two binary trees called the **left subtree** and the **right subtree**



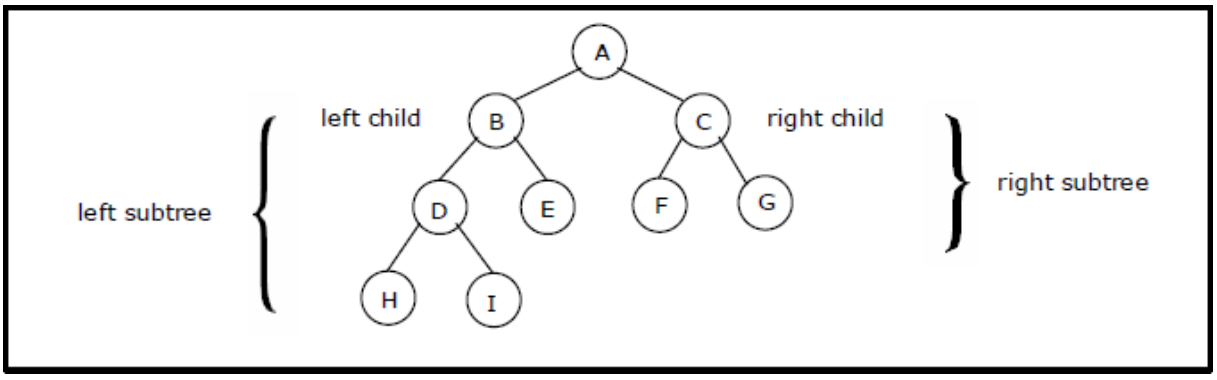


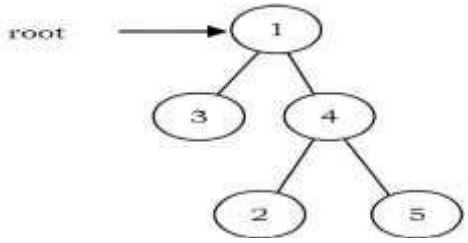
Fig: example for binary tree

Types of Binary Trees:

Strictly Binary Tree/Fully Binary Tree:

A strictly binary tree is binary tree in which every node must have two children except the leaf node.

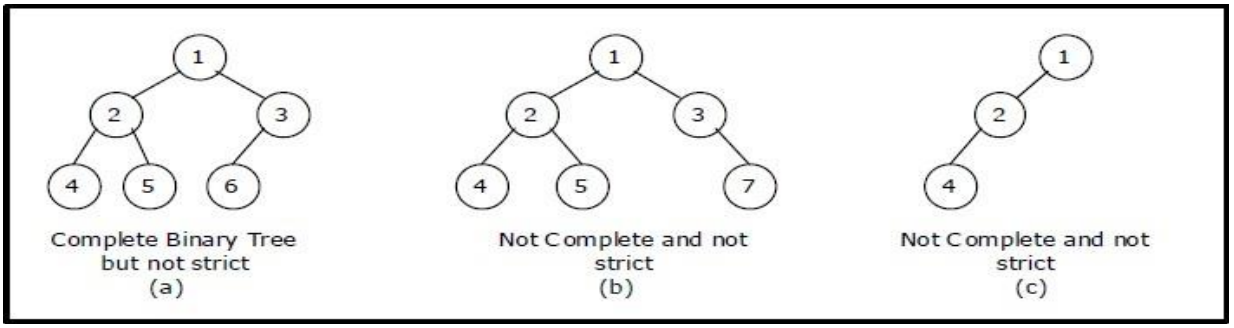
- Strictly binary tree is also called as **Full Binary Tree** or **Proper Binary Tree** or **2-** tree



- A strictly binary tree with n leaves always contains (2n-1) nodes.

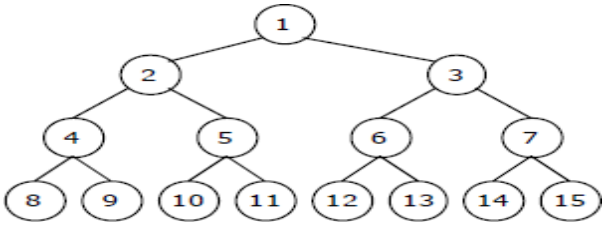
Complete Binary Tree:

A complete binary tree is a binary tree in which except the last level the remaining levels are completely filled and arrange the nodes from the left to right.



Perfect Binary Tree/Fully complete Binary Tree:

A binary tree is called perfect binary tree if each node has exactly two children and all leaf nodes are at the same level

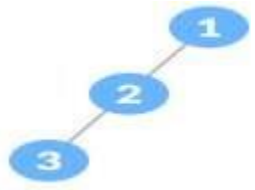


Degenerated Binary Tree or pathological Binary Tree or skewed binary tree :

A degenerate or pathological tree is the tree having a single child either left or right. Thus, there are three types of skewed binary tree: **left-skewed binary tree** , **right-skewed binary tree** or mixed skewed binary tree.

a. Left skewed degenerated BT:

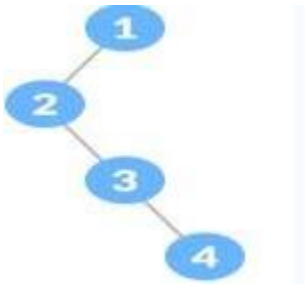
Each internal node has only one left child.



b. Right skewed degenerated BT:

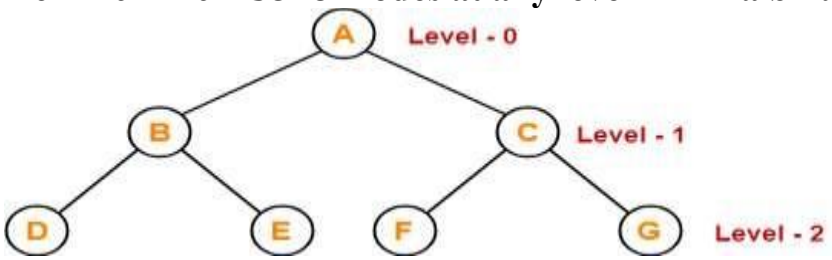
Each internal node has only one left child

c. mixed skewed degenerated BT:



PROPERTIES OF BINARY TRESS:

- **Maximum number of nodes at any level 'L' in a binary tree = 2^L**

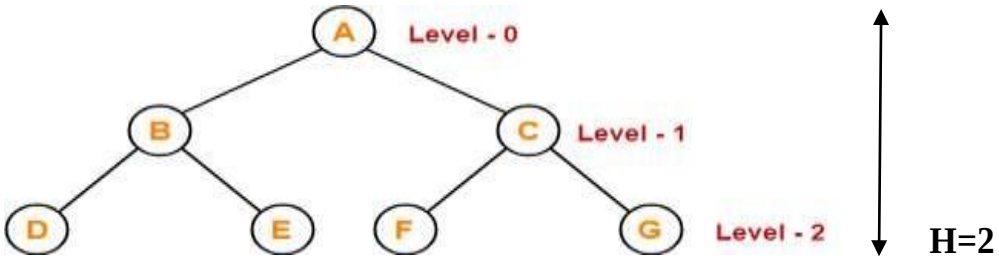


Maximum number of nodes at level-0 in a binary tree $=2^0 =1$

Maximum number of nodes at level-1 in a binary tree $=2^1 =2$

Maximum number of nodes at level-2 in a binary tree $=2^2 =4$

- **Maximum number of nodes in a binary tree of height H= $2^{H+1} – 1$**



Since, there are h levels we need to add all nodes at each level

$2^0 + 2^1 + 2^2 + \cdots + 2^h$ (series I n G.P)

$= 2^{H+1} – 1.$

Maximum number of nodes in a binary tree of height 2

$= 2^{2+1} – 1$

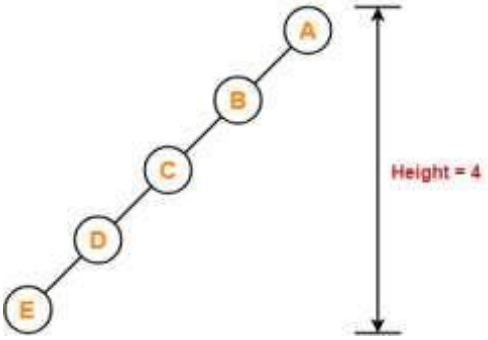
$= 8 – 1$

= 7 nodes

Thus, in a binary tree of height = 2, maximum number of nodes that can be inserted = 7

- **Minimum number of nodes in a binary tree of height H= H + 1**

To construct a binary tree of height = 4, we need at least 4 + 1 = 5 nodes.



- **Minimum height of a BT H= $\lceil \log(n+1) \rceil - 1$**

Finding a height we have to use maximum no. of nodes at height H

$n = 2^{H+1} - 1.$

$n+1 = 2^{H+1}$

$H+1 = \log_2(n+1)$

$H = \log_2(n+1) - 1$

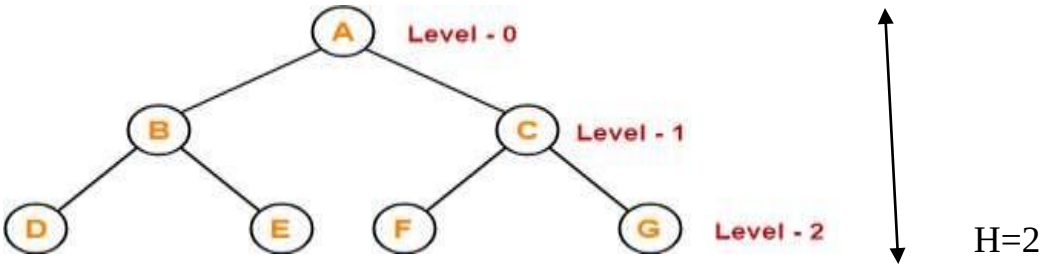
- **Maximum height H=n-1**

Finding min height we have to consider min no .of nodes at height H

N= H+1

H=n-1

- **If h= height of a BT then maximum no .of leaves=2^h**



H=2

In above max no.of leaves =2^2 =4 leave nodes

- **If a binary tree contains m nodes at level L ,it contains at most 2m nodes at level L+1.**

From above example at level 0 we have one node. So, m=1

At L+1=2m=2(1)=2

At L+1 means level 1 we have 2 nodes.

- **Total no. of edges in a full binary tree with n node is n-1**

From above example, the tree contains 7 nodes .so total edges are =n-1=7-1=6 edges

- **Total no. of external nodes =total no. of internal nodes + 1**

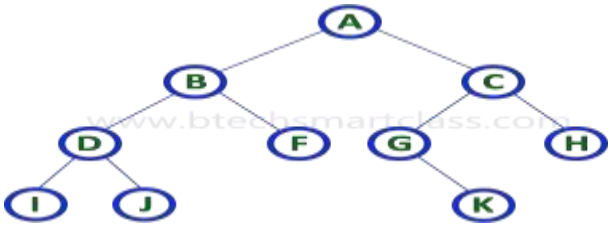
From above example, the tree contains 3 internal nodes are 3 the external nodes = internal nodes + 1=3+1=4 external nodes

BINARY TREE REPRESENTATIONS:

A binary tree data structure is represented using two methods. Those methods are as follows...

1. **Array Representation**
2. **Linked List Representation**

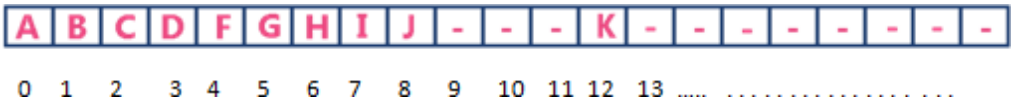
Consider the following binary tree...



1. Array Representation of Binary Tree

In array representation of a binary tree, we use one-dimensional array (1-D Array) to represent a binary tree.

- The root of the tree is always stored at index zero
- A node occupies index I then its left child stored at **2*i +1 and** the right child stored at **2*i +2**
  Consider the above example of a binary tree and it is represented as follows..



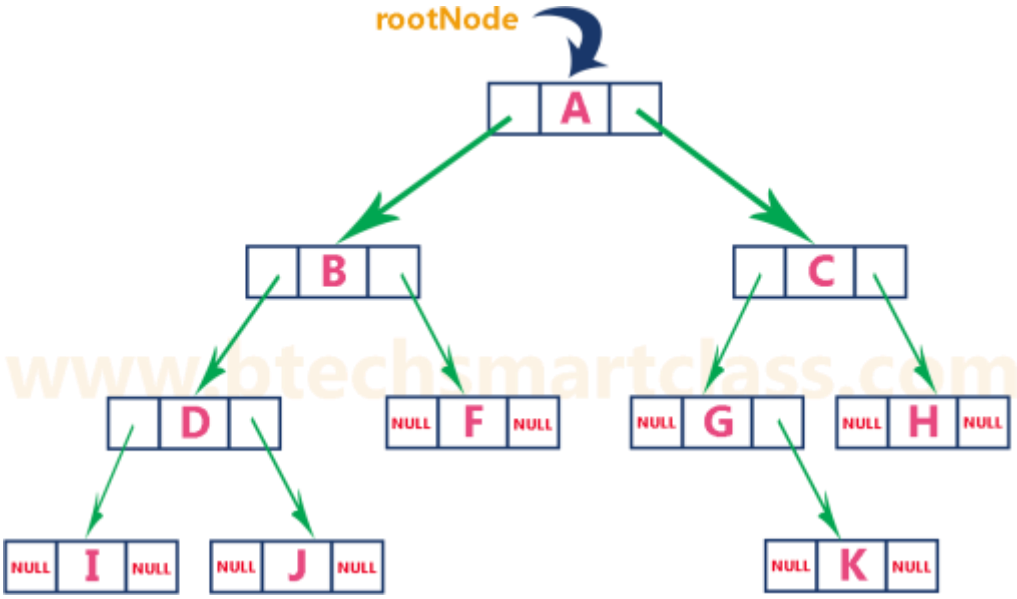
2. Linked List Representation of Binary Tree

In a linked list representation every node consists of three fields.

- **First field for storing left child address**
- **Second field for storing actual data**
- **Third field for storing right child address.**

In this linked list representation, a node has the following structure...

The above example of the binary tree represented using Linked list representation is shown as follows...



The structure of a binary search tree is

```
  Struct BT
{
   Struct BT * leftchild;
   Int data;
   Struct BT * rightchild;
} *new,*root;
```

Binary tree operations or Abstract Data Type:

Binary tree as an Abstract data type because which contains various operations.

Abstract Data Type Binary tree( )

{

Objective: binary tree is a collection of nodes in which each node exactly contains at most two children

Operations:

Create( ): used for creating the BT

Insert( ): adding new node into the tree.

Delete( ): removing a node from the tree.

Traversal ( ): visiting each node in the tree exactly once

}

Create( ) operation on binary tree: used for creating new node into the BT

Algorithm:

Struct node * create( ) // function returns addess of a node

Begin

1. Declare data variable x
2. Allocate memory to the tree

3.read x value and if you want not enter any child value then you can give simply x value as -1

4.else

4.1 give some value of x and assign to the new node data field

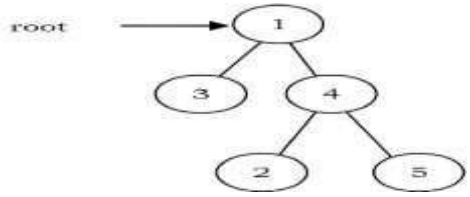
New->data = x

4.2 for creating left subtree call create ( ) recursively, i.e., new->left = create( )

4.3 for creating right subtree call create ( ) recursively,i.e., new->right= create( )

5.return new node

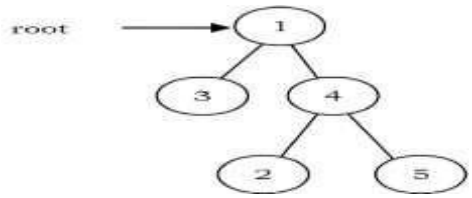
End



Insertion operation on BT:

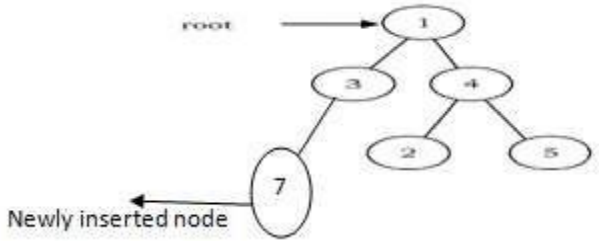
In a binary tree we can insert a new node either left or right of a given node.i.e in BT nodes are inserted into the tree based on user desire.

Example:



Consider the inserted node is 7, for inserting the new node into the tree user has to give his choice mean node is attached to the left or right side of root node 1 .if he give choice as left of root node 1 then again he has to give a choice whether the new node insert as left or right child of the node 3.if he give answer as attach left child of 3 then insert new node as the left

Child of node 3.



.

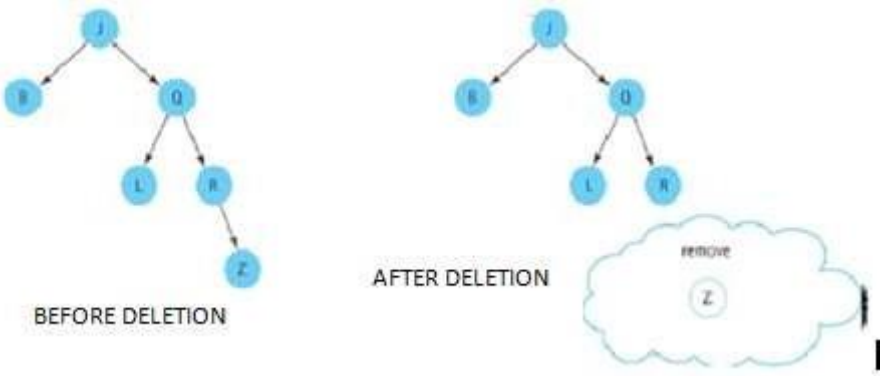
Deletion operation on BT:

Removing a node from the tree is known as deletion operation

- Deletion operation on BT can be performed in three ways.
    1. Delete a node with no child
    2. Delete a node with one child
    3. Delete a node with two children

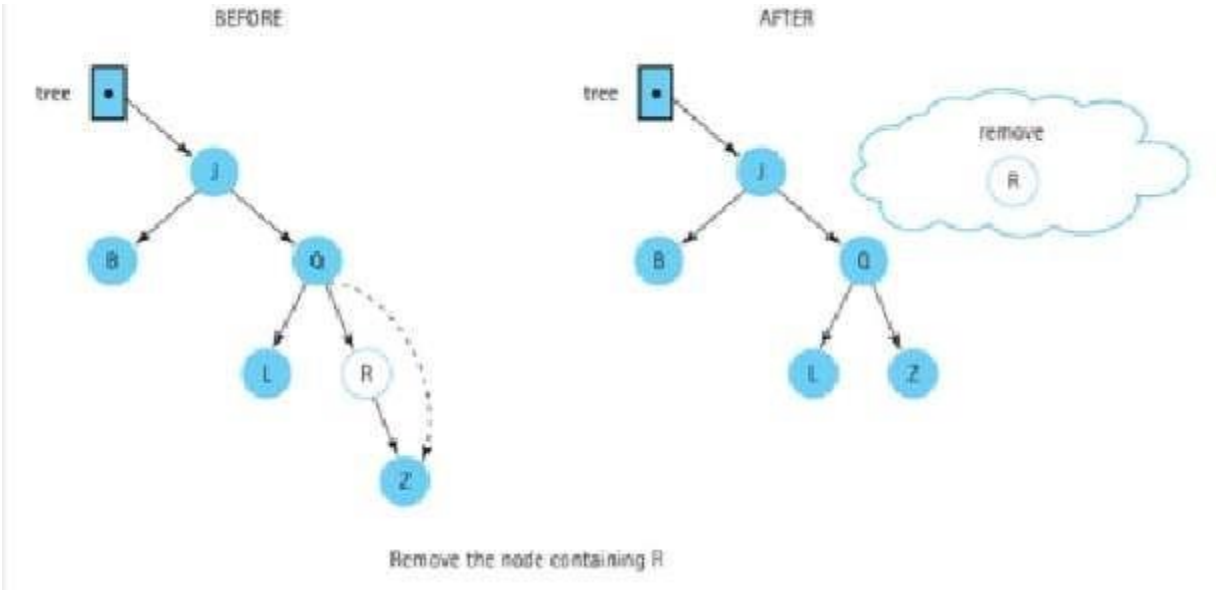
Delete a node with no child:

Delete a node with no child does not affect the tree because no child means the node is a leaf node so we can easily remove the node from the tree and appropriate link of its parent set to null.



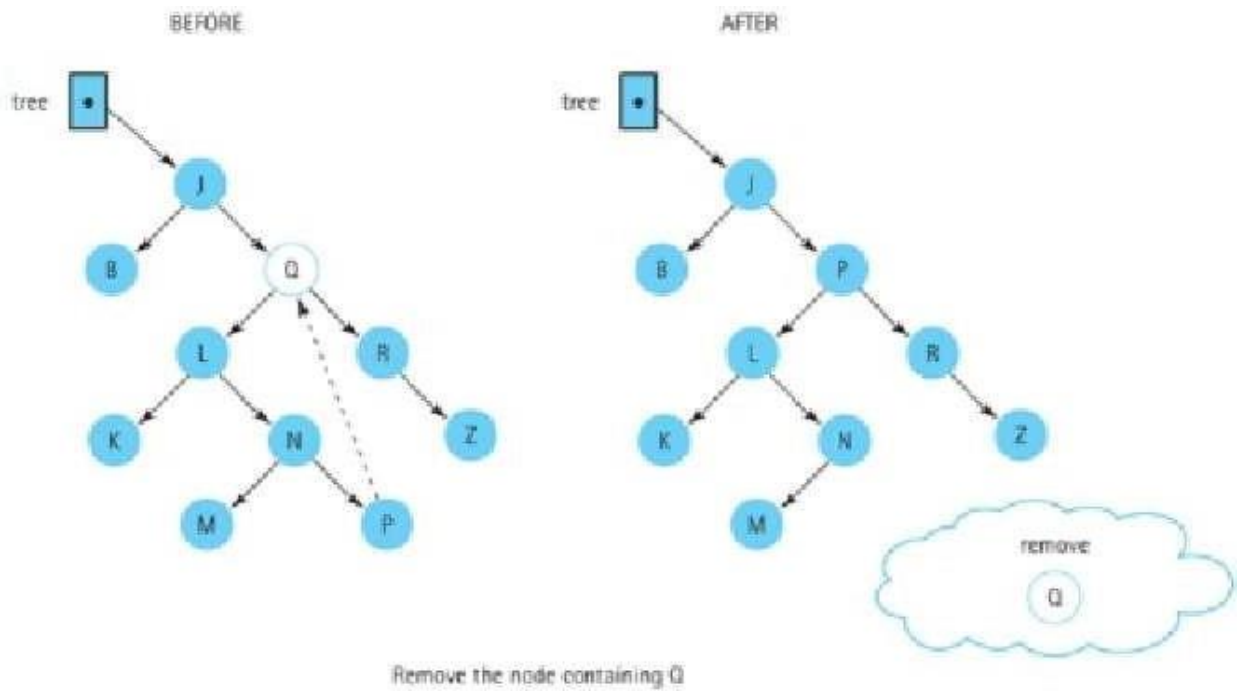
Delete a node with one child:

When delete the node with one child then the child of the deleted node is become the child of the deleted node parent.



Delete a node with two children:

Node with two children means it contains both left and right node. When delete the node with two children then replace deleted node with maximum value of the its left sub tree or minimum value of its right sub tree.



Binary Tree Traversal Techniques:

A tree traversal is a method of visiting every node in the tree. Traversal operation also called display operation.

- The traversal in BT involves three kinds of basic activities such as
  1. Visiting the root node (N)
  2. Visiting the left sub tree (L)
  3. Visiting the right sub tree (R)
- Based on the order in which these three operations are performed the traversal divid into three types.
  1. Preorder (*NLR*) Traversal
  2. Inorder (*LNR*) Traversal
  3. Post order (*LRN*) Traversal
- Level Order Traversal : This method is inspired from Breadth First Traversal (BFS of Graph algorithms).

Preorder (*NLR*) Traversal:

In a preorder traversal, each root node is visited before its left and right sub trees are traversed. Preorder search is also called backtracking. The steps for traversing a binary tree in preorder traversal are:

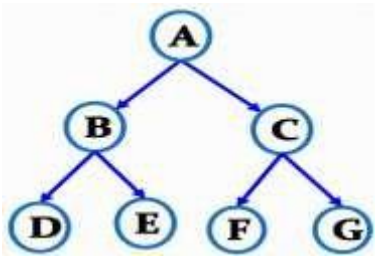
1. Visit the root.
2. Visit the left sub tree, using preorder.

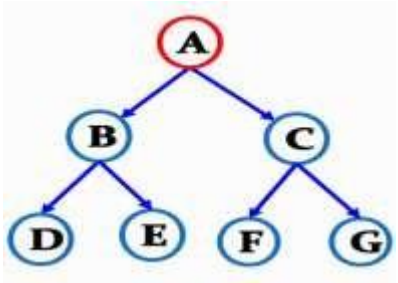
3. Visit the right sub tree, using preorder.

The algorithm for preorder traversal is as follows:

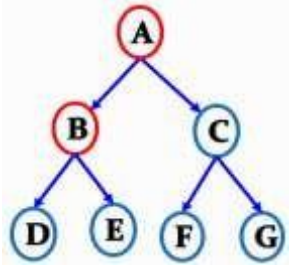
```
void preorder(node *root)
begin
  if( root != NULL )
  begin
     Display root -> data;
     Preorder (root -> left child);// call recursively
    Preorder (root -> right child); // call recursively
  end
end
```

Example:

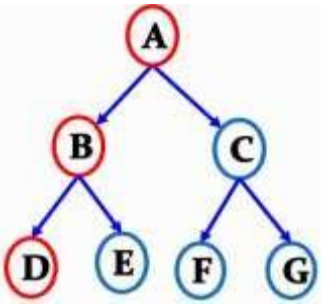




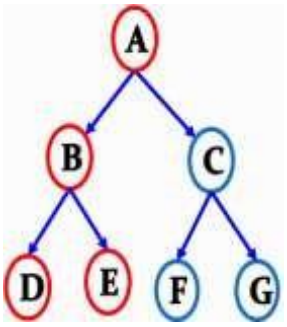
displayed A



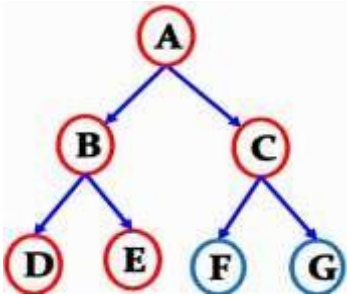
displayed A,B



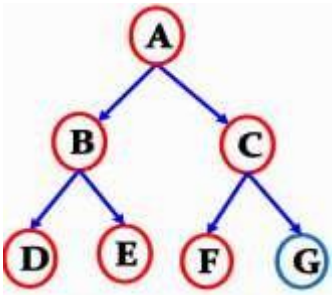
displayed A,B,D



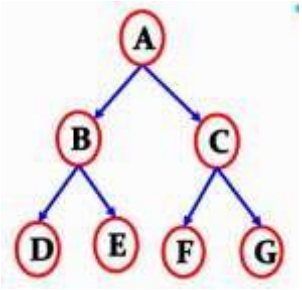
Displayed A,B,D,E



display A,B,D,E,C



displayed A,B,D,E,C,F



Displayed A, B, D, E, C, F, G

- The preorder traversal of above tree : A,B,D,E,C,F,G

Inorder (*LNR*) Traversal:

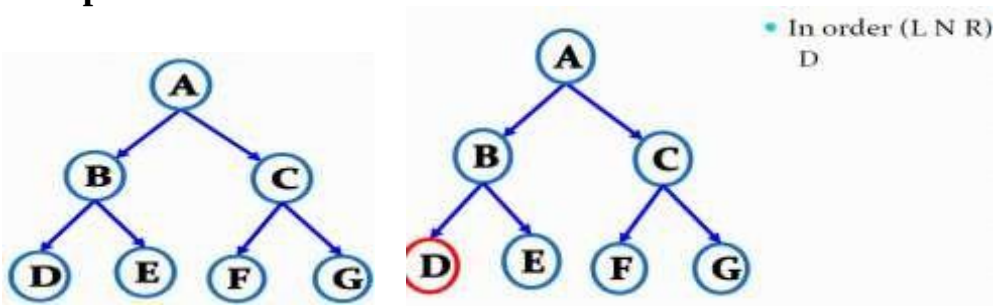
In the case of inorder traversal, the root of each subtree is visited after its left subtree has been traversed but before the traversal of its right subtree begins. The steps for traversing a binary tree in inorder traversal are:

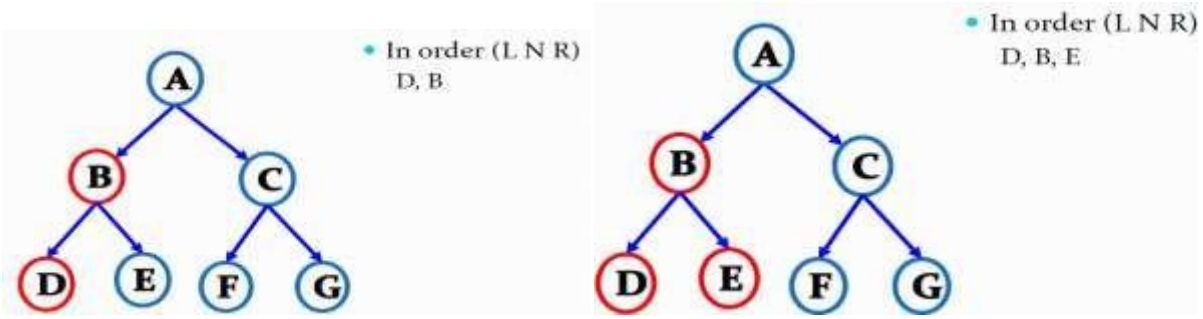
1. Visit the left subtree, using inorder.
2. Visit the root.
3. Visit the right subtree, using inorder.

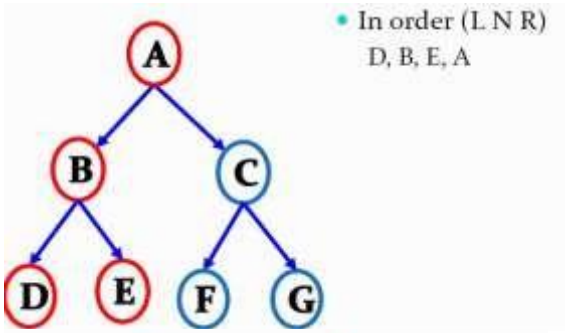
The algorithm for inorder traversal is as follows:

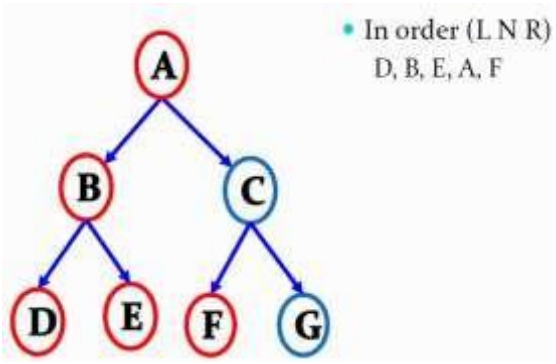
```
void inorder(node *root)
begin
if(root != NULL)
begin
 in order(root->left child);

Print root -> data;
in order(root->right child);
end
end
```

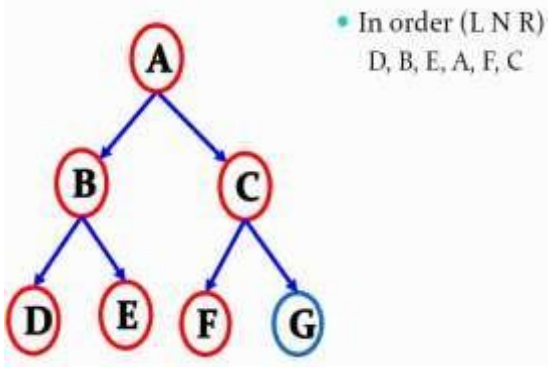
example:

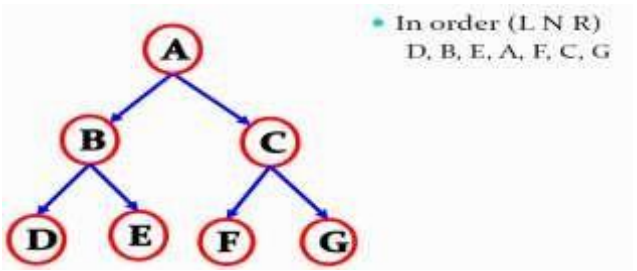












The inorder traversal of above graph is D,B,E,A,F,C,G

Post order (LRN) Traversal:

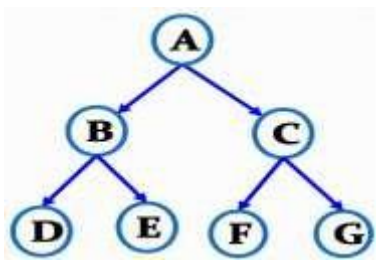
In a postorder traversal, each root is visited after its left and right subtrees have been traversed. The steps for traversing a binary tree in postorder traversal are:

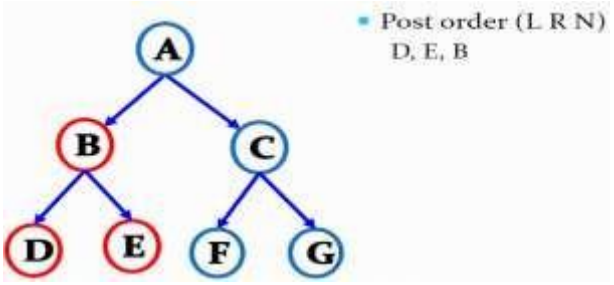
1. Visit the left subtree, using postorder.
2. Visit the right subtree, using postorder
3. Visit the root.

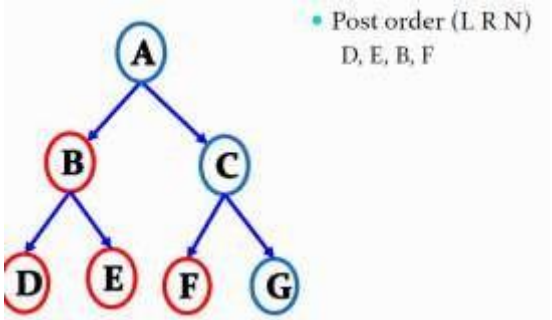
The algorithm for postorder traversal is as follows:

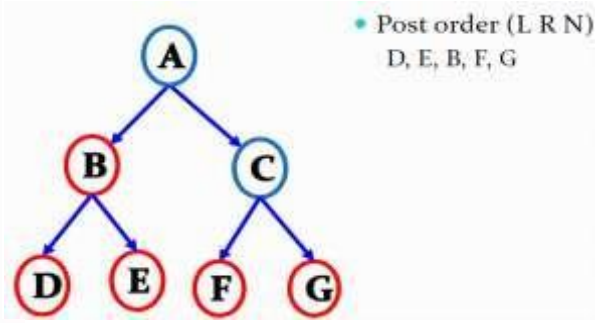
```
void postorder(node *root)
begin
  if( root != NULL )
begin
  postorder (root -> left child);
  postorder (root -> right child);
  print (root -> data);
end
end
```

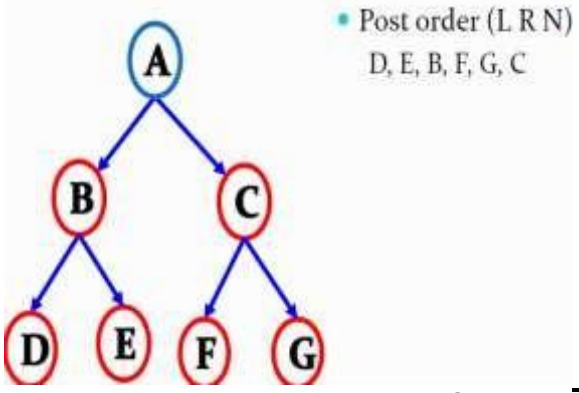
example:

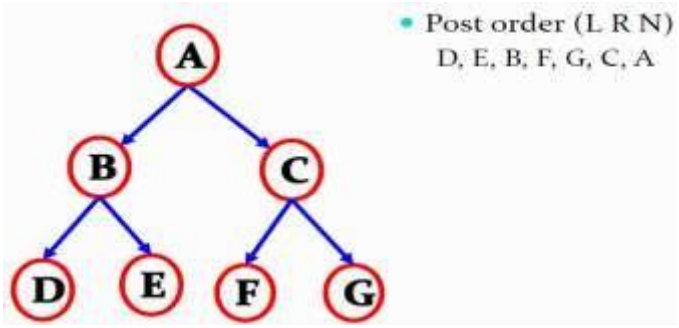












The POST order traversal of above graph is D,E,B,F,G,C,A

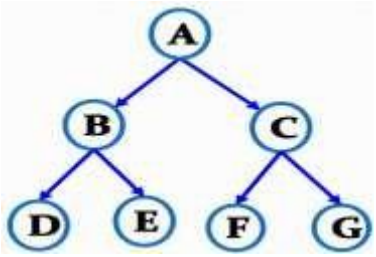
Level order Traversal:

In a level order traversal, the nodes are visited level by level starting from the root, and going from left to right. The level order traversal requires a queue data structure. So, it is not possible to develop a recursive procedure to traverse the binary tree in level order. This is nothing but a breadth first search technique in graphs.

The algorithm for level order traversal is as follows:

```
Void  levelorder()
  begin
    int j;
for(j = 0; j < ctr; j++)
begin
if(tree[j] != NULL)
print tree[j] -> data;
end
end
```

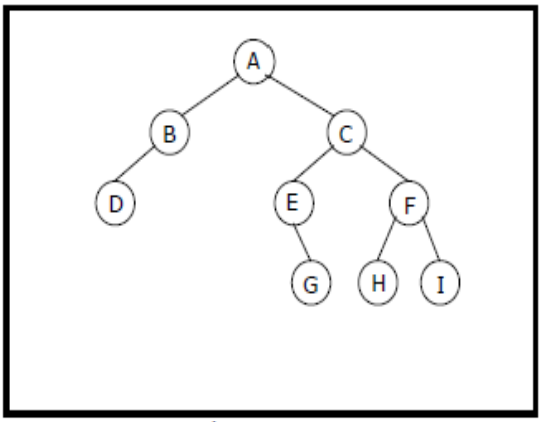
Example:



level order traversal of a tree is: A,B,C,D,E,F,G

Example 1:

Traverse the following binary tree in pre, post, inorder and level order.



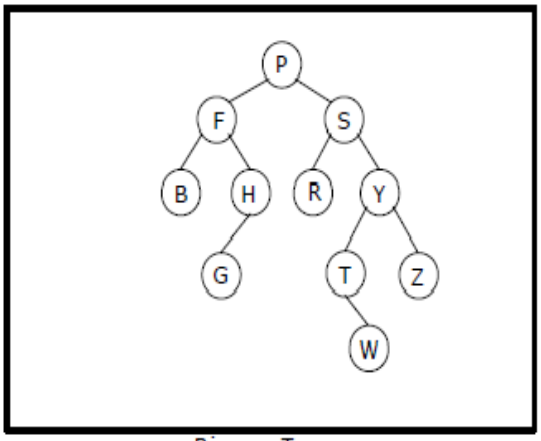
Binary Tree

- Preorder traversal yields:
  A, B, D, C, E, G, F, H, I
- Postorder traversal yields:
  D, B, G, E, H, I, F, C, A
- Inorder traversal yields:
  D, B, A, E, G, C, H, F, I
- Level order traversal yields:
  A, B, C, D, E, F, G, H, I

Pre, Post, Inorder and level order Traversing

Example 2:

Traverse the following binary tree in pre, post, inorder and level order.



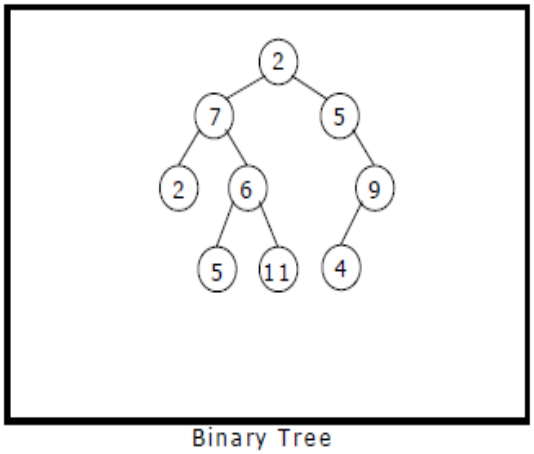
Binary Tree

- Preorder traversal yields:
  P, F, B, H, G, S, R, Y, T, W, Z
- Postorder traversal yields:
  B, G, H, F, R, W, T, Z, Y, S, P
- Inorder traversal yields:
  B, F, G, H, P, R, S, T, W, Y, Z
- Level order traversal yields:
  P, F, S, B, H, R, Y, G, T, Z, W

Pre, Post, Inorder and level order Traversing

Example 3:

Traverse the following binary tree in pre, post, inorder and level order.



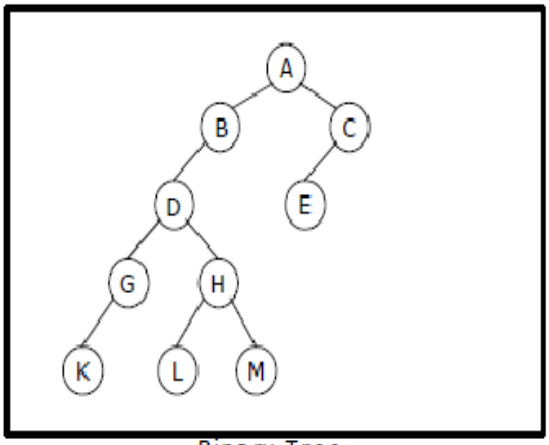
Binary Tree

- Preorder traversal yields:
  2, 7, 2, 6, 5, 11, 5, 9, 4
- Postorder travarsal yields:
  2, 5, 11, 6, 7, 4, 9, 5, 2
- Inorder travarsal yields:
  2, 7, 5, 6, 11, 2, 5, 4, 9
- Level order traversal yields:
  2, 7, 5, 2, 6, 9, 5, 11, 4

Pre, Post, Inorder and level order Traversing

Example 4:

Traverse the following binary tree in pre, post, inorder and level order.



Binary Tree

- Preorder traversal yields:
  A, B, D, G, K, H, L, M, C, E
- Postorder travarsal yields:
  K, G, L, M, H, D, B, E, C, A
- Inorder travarsal yields:
  K, G, D, L, H, M, B, A, E, C
- Level order traversal yields:
  A, B, C, D, E, G, H, K, L, M

Pre, Post, Inorder and level order Traversing

Building (construction)Binary Tree from Traversal Pairs:

Sometimes it is required to construct a binary tree if its traversals are known. From a Single traversal it is not possible to construct unique binary tree. However any of the two traversals are given then the corresponding tree can be drawn:

1. In order and preorder
2. In order and postorde
3. Preorder and post order

The basic principle for formulation is as follows:

If the preorder traversal is given, then the first node is the root node. If the post order traversal is given then the last node is the root node. Once the root node is identified, all

the nodes in the left sub-trees and right sub-trees of the root node can be identified using in order.

Same technique can be applied repeatedly to form sub-trees. It can be noted that, for the purpose mentioned, two traversal are essential out of which one should be in order traversal and another preorder or post order using these combinations corresponding tree can be drawn uniquely; alternatively, given preorder and post order traversals, binary tree cannot be obtained uniquely.

Construct a binary tree from inorder and preorder:

Example 1:

Construct a binary tree from a given preorder and in order sequence:

Preorder: A B D G C E H I F

In order: D G B A H E I C F

Solution:

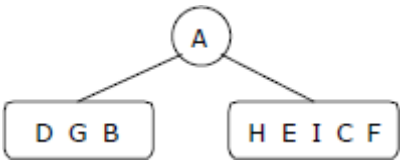
*From Preorder sequence **A** B D G C E H I F, the root is: A*

From In order sequence *D G B **A** H E I C F*, we get the left and right sub trees:

Left sub tree is: D G B

Right sub tree is: H E I C F

The Binary tree up to this point looks like:

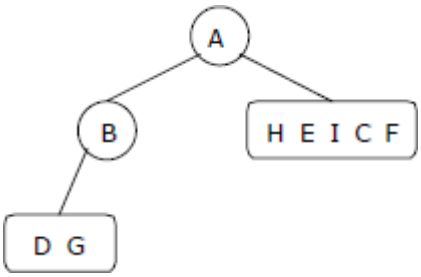


To find the root, left and right sub trees for D G B:

*From the preorder sequence **B** D G, the root of tree is: B*

From the in order sequence D G **B,** we can find that D and G are to the left of B.

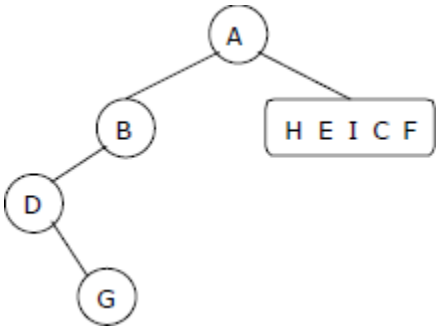
The Binary tree upto this point looks like:



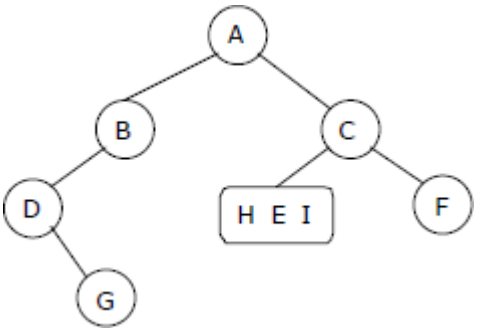
To find the root, left and right sub trees for D G:

From the preorder sequence **D** *G, the root of the tree is: D* . From the in order sequence ***D** G*, we can find that there is no left node to D and G is at the right of D.

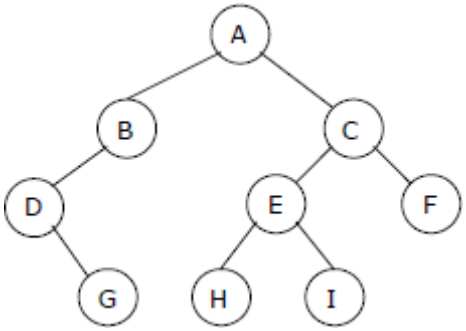
The Binary tree upto this point looks like:



To find the root, left and right sub trees for H E I C F:
*From the preorder sequence **C** E H I F, the root of the left sub tree is: C*
From the in order sequence *H E I **C** F*, we can find that H E I are at the left of C and F is at the right of C.
The Binary tree up to this point looks like:



To find the root, left and right sub trees for H E I:
*From the preorder sequence **E** H I, the root of the tree is: E*
From the in order sequence *H **E** I*, we can find that H is at the left of E and I is at the right of E.
The Binary tree up to this point looks like:



Construct a binary tree from inorder and postorder:

Construct a binary tree from a given postorder and inorder sequence:
Inorder: D G B A H E I C F
Postorder: G D B H I E F C A

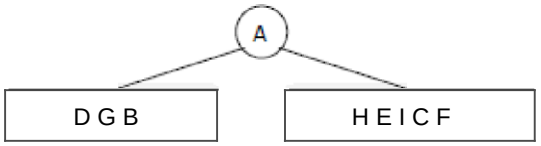
Solution:

From Postorder sequence C D B H I E F C A, the root is: *A*

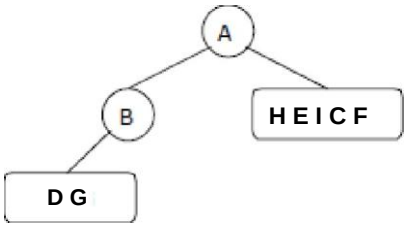
From inorder sequence D G B A H E F C F, we get the left and right subtrees:

Left sub tree is. D G B
Right sub tree is. H E IC F

The Binary tree up to this point looks like:



Ta find the root, left and right sub trees for D G B:
From the preorder sequence B D G, the root of tree is: B
From the in order sequence D G B, we can find that D and G are to the left of B.
The Binary tree up to this point looks like:

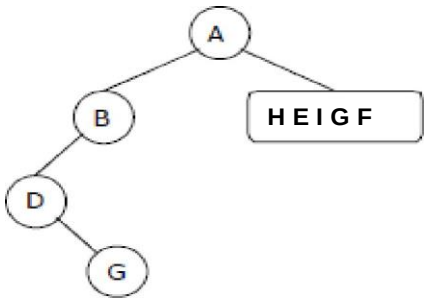


To find the root, left and right sub trees for D G:

*From the postorder sequence G **D**, the root of the tree is: D*

From the in-order sequence D we can find that is no left subtree for D and G is to the right of D.

The Binary tree up to this point looks like.

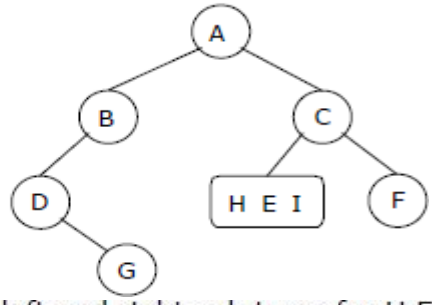


To find the root, left and right sub trees for H E I C F:

*From the postorder sequence H I E F **C**, the root of the left sub tree is: C*

From the inorder sequence H E I ***C*** F, we can find that H E I are to the left of C and F is the right subtree for C.

The Binary tree upto this point looks like:

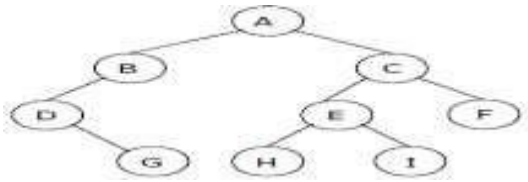


To find the root, left and right sub trees for H E I:

*From the postorder sequence H I **E**, the root of the tree is: E*

From the inorder sequence *H* **E** *I*, we can find that H is left subtree for E and I is to the right of E.

The Binary tree upto this point looks like:



Construct a binary tree from preorder and postorder

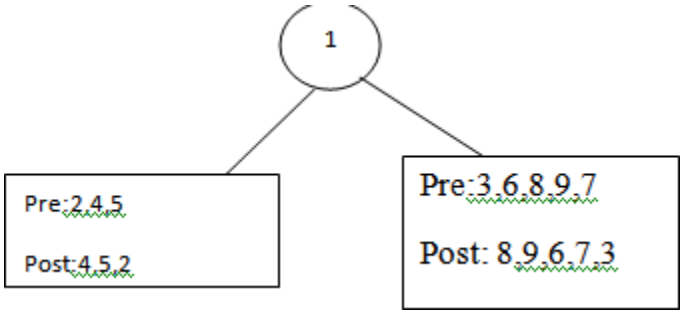
Construct a binary tree from a given postorder and inorder sequence:
preorder:1,2,4,5,3,6,8,9,7
Postorder: 4,5,2,8,9,6,7,3,1
Solution:

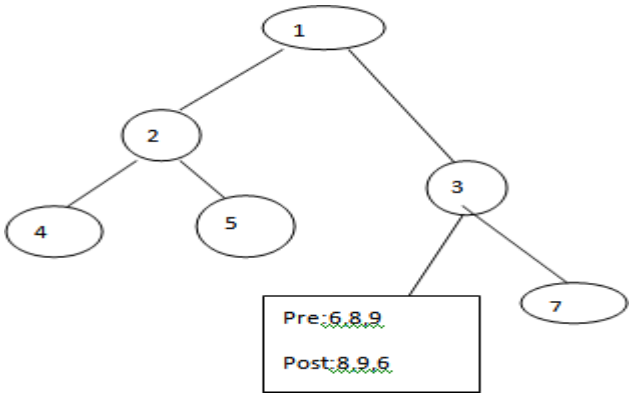
1. We know that root is first element in pre-order traversal and last element in post order traversal. therefore, the root node is 1
2. Then we locate the next element (successor) of root node in preorder sequence, which must be the left child of the root node. Now since 2 is root node of the left subtree, all the nodes before 2 in post order sequence must be present in the left sub tree of the root node

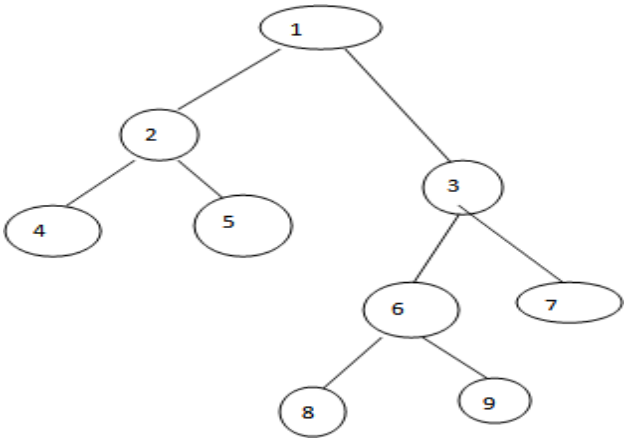
i.e {4,5,2} and all the nodes after 2 (except the last) must be present in right subtree i.e{8,9,6,7,3}.

Left sub tree:
Preorder:2,4,5
Postorder:4,5,2

Right sub tree:
Preorder: 3,6,8,9,7
Postorder:8.9,6,7,3


1
Pre:2,4,5
Post:4,5,2
Pre:3,6,8,9,7
Post: 8,9,6,7,3


1
2
3
4
5
7
Pre:6,8,9
Post:8,9,6


1
2
3
4
5
6
7
8
9

Binary Search Trees:

In a binary tree, every node can have a maximum of two children but there is no need to maintain the order of nodes basing on their values. In a binary tree, the elements are arranged in the order they arrive at the tree from top to bottom and left to right

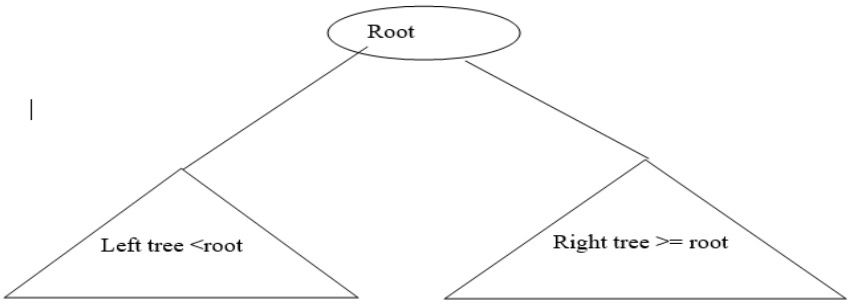
- To enhance the performance of binary tree, we use a special type of binary tree known as **Binary Search Tree**. Binary search tree mainly focuses on the search operation in a binary tree. Binary search tree can be defined as follows...

Definition or BST property:

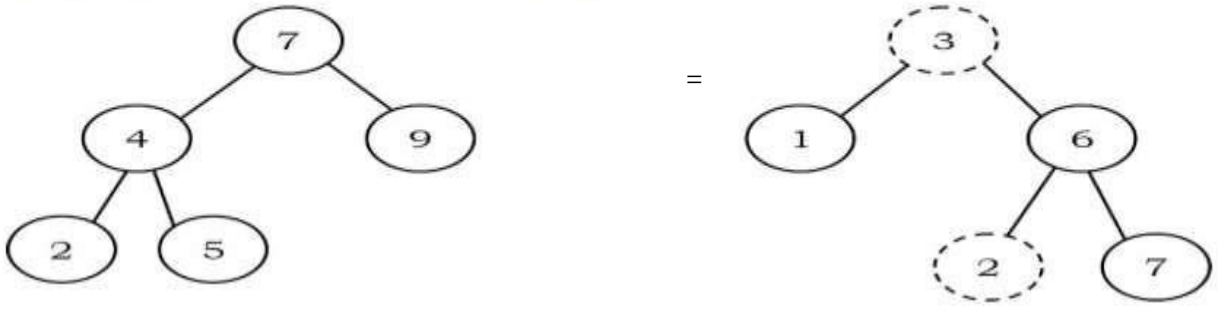
In binary search trees, all the left sub tree node values should be less than the root node data value and all the right sub tree node values should be greater than root node data value. This is called binary search tree property.

- This property should be satisfied at every node in the tree.
  - The left sub tree of a node contains only nodes with keys less than the its root nodes key.
  - The right sub tree of a node contains only nodes with keys equal or greater than the its root nodes key.
  - Both the left and right sub trees must also be binary search trees.
- Here the values can also be known as keys.
- The BST also known as order binary tree or sorted binary tree
- Every binary search tree is a binary tree but every binary tree need not to be binary search tree

```
left subtree (keys) < node (key) ≤ right subtree (keys)
```



Example: The left tree is a binary search tree and the right tree is not a binary search tree (at node 6 it's not satisfying the binary search tree property).



Operations on a Binary Search Tree:

The following operations are performed on a binary search tree...

1. **Search**
2. **Insertion**
3. **Deletion**
4. **Find min/find max**

Search Operation in BST:

The searching operation is used to find whether a given value is present in the tree or not.

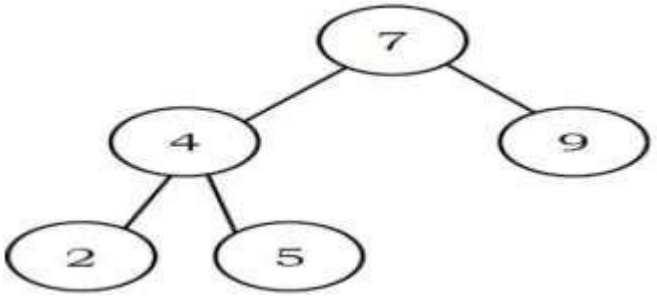
In a binary search tree, the search operation is performed with **O(log n)** time complexity. The search operation is performed as follows..

- **Step 1 -** Read the search element from the user.
- **Step 2 -** Compare the search element with the value of root node in the tree.

- **Step 3 -** If both are matched, then display "Given node is found!!!" and terminate the functi **Step 4 -** If both are not matched, then check whether search element is smaller or larger than that root node value.
- **Step 5 -** If search element is smaller, then continue the search process in left subtree.

- **Step 6-** If search element is larger, then continue the search process in right subtree.
- **Step 7 -** Repeat the same until we find the exact element or until the search element is compared with the leaf node
- **Step 8 -** If we reach to the node having the value equal to the search value then display "Element is found" and terminate the function.
- **Step 9 -** If we reach to the leaf node and if it is also not matched with the search element, then display "Element is not found" and terminate the function.

Example:



considered search key= 4

Now compare search key with root node here root value is 7 and search key value 4 and 4< 7 so go for searching 4 to the left sub tree of the root .here left sub tree root value and search key values are matched so, the search key is found.

Insertion Operation in BST:

adding a new node at the correct position means that the new node should not violate the property of BST. In a binary search tree, the insertion operation is performed with **O(log n)** time complexity. In binary search tree, new node is always inserted as a leaf node.

Algorithm:

The insertion operation is performed as follows...

Step 1 – Create a newNode with given value and set its **left** and **right** to **NULL**.

Step 2 - Check whether tree is Empty.

Step 3 - If the tree is **Empty**, then set **root** to **newNode**.

Step 4 - If the tree is **Not Empty**, then check whether the value of newNode is **smaller** or **larger** than the node (here it is root node).

Step 5 - If newNode is **smaller** than **or equal** to the node then move to its **left** child. If newNode is **larger** than the node then move to its **right** child.

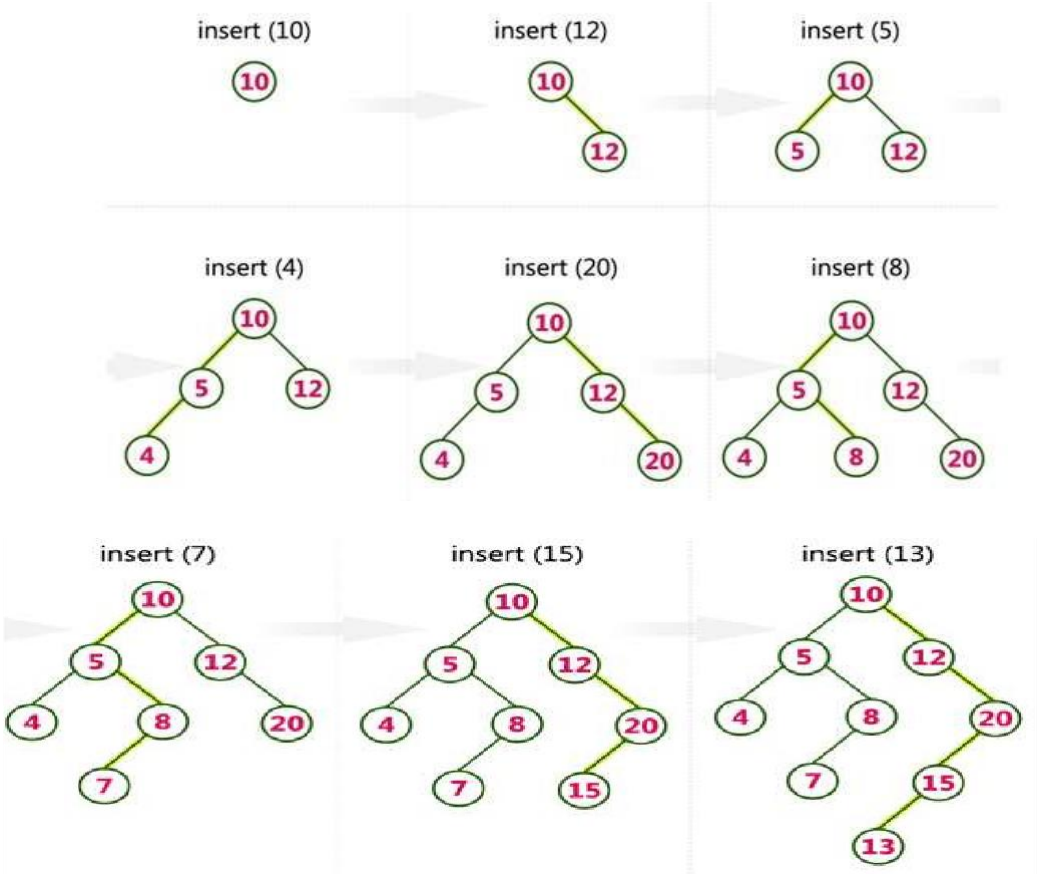
Step 6- Repeat the above steps until we reach to the **leaf** node (i.e., reaches to NULL).

Step 7 - After reaching the leaf node, insert the newNode as **left child** if the newNode is **smaller or equal** to that leaf node or else insert it as **right child**.

Example: Construct a Binary Search Tree by inserting the following sequence of numbers...

- ***10,12,5,4,20,8,7,15 and 13***

- Above elements are inserted into a Binary Search Tree as follows...



Deletion Operation in BST:

In a binary search tree, the deletion operation is performed with **O(log n)** time complexity. Deleting a node from Binary search tree includes following three cases...

- **Case 1: Deleting a Leaf node (A node with no children)**
- **Case 2: Deleting a node with one child**
- **Case 3: Deleting a node with two children**

Algorithm:

Case 1: Deleting a leaf node

We use the following steps to delete a leaf node from BST...

- **Step 1 - Find** the node to be deleted using **search operation**
- **Step 2 -** Delete the node using **free** function (If it is a leaf) and terminate the function.

Case 2: Deleting a node with one child

We use the following steps to delete a node with one child from BST...

- **Step 1 - Find** the node to be deleted using **search operation**
- **Step 2 -** If it has only one child then create a link between its parent node and child node.
- **Step 3 -** Delete the node using **free** function and terminate the function.

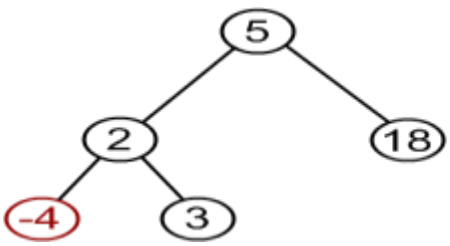
Case 3: Deleting a node with two children

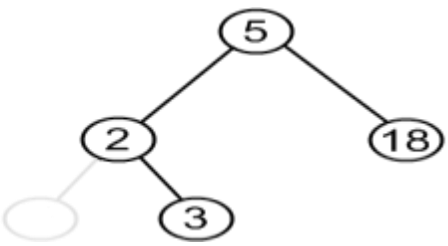
We use the following steps to delete a node with two children from BST...

- **Step 1 - Find** the node to be deleted using **search operation**
- **Step 2 -** If it has two children, then find the **largest** node in its **left subtree** (OR) the **smallest** node in its **right subtree**.
- **Step 3 - Swap** both **deleting node** and node which is found in the above step.
- **Step 4 -** Then check whether deleting node came to **case 1** or **case 2** or else goto step 2
- **Step 5 -** If it comes to **case 1**, then delete using case 1 logic.
- **Step 6-** If it comes to **case 2**, then delete using case 2 logic.
- **Step 7 -** Repeat the same process until the node is deleted from the tree.

Case 1: Deleting a Node that has no children:

We can simply remove the deleted node without any issue. This case is quite simple. Algorithm sets corresponding link of the parent to NULL and disposes the node.

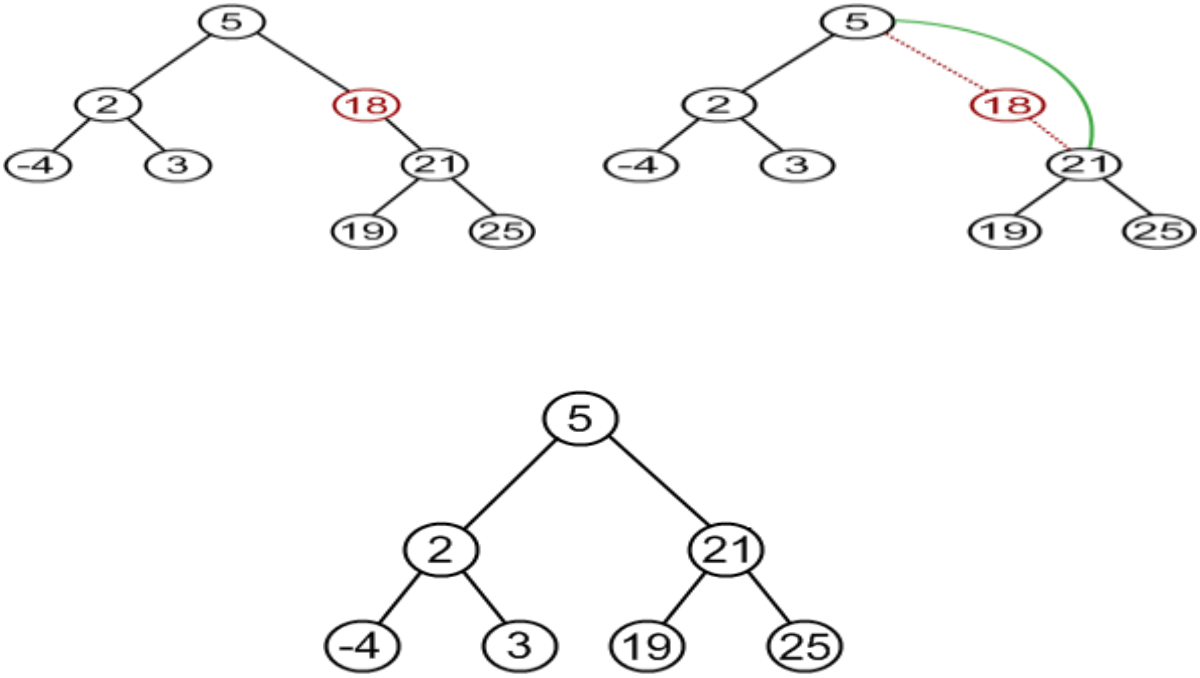




Case 2:Node to be removed has one child:

child will replace the position of the parent. Node with one child means it contains either left node or right node.

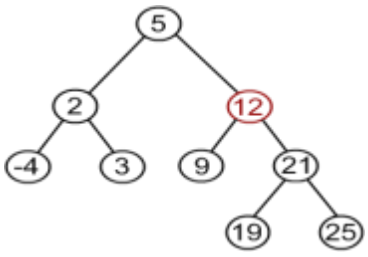
- If the deleted node is left child of its parent then the deleted node child becomes the left child of the deleted node's parent.
- Correspondingly, if the deleted node is right child of its parent then the deleted node child becomes the right child of the deleted node's parent.
- It this case, node is cut from the tree and algorithm links single child (with it's sub tree) directly to the parent of the removed node.
- **Example: remove 18 from the BST**



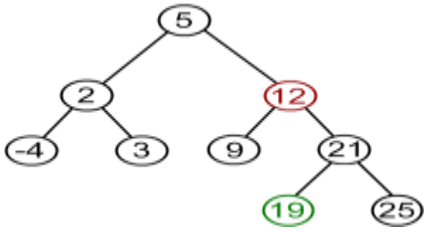
Case 3:Node to be removed has two children:

To handle this case, replace the node's value with its in-order predecessor (largest value in the left sub tree) or in order successor (smallest value in the right sub tree).

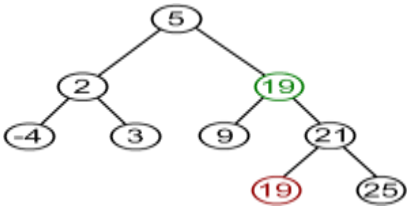
Example. Remove 12 from a BST.

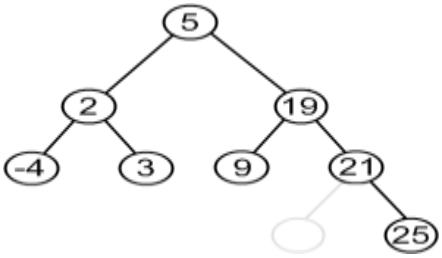


Find minimum element in the right sub tree of the node to be removed. In current example it is 19.



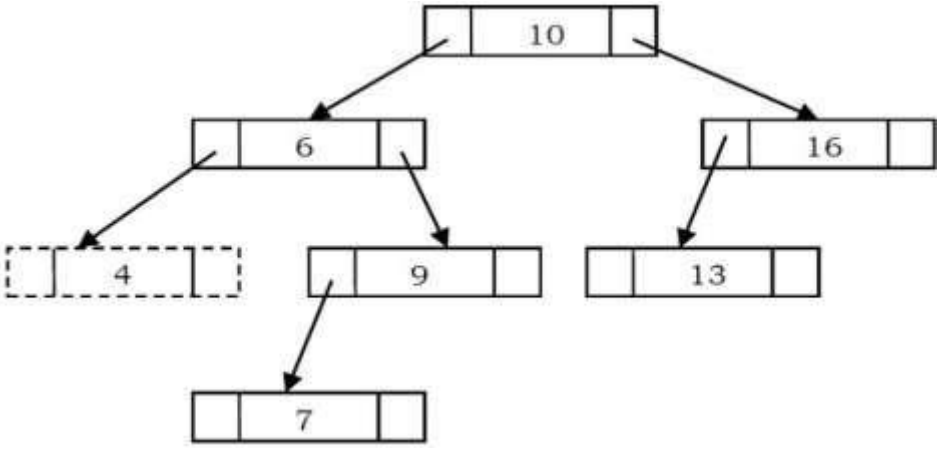
Replace 12 with 19. Notice, that only values are replaced, not nodes. Now we have two nodes with the same value and Remove 19 from the left subtree





Finding Minimum Element in Binary Search Trees

In BSTs, the minimum element is the left-most node, which does not has left child. In the BST below, the minimum element is **4**.



Algorithm:

Find min(struct node * root)

Begin

If(root= =Null)

Return root

Else if(root->left = =Null)

Return root;

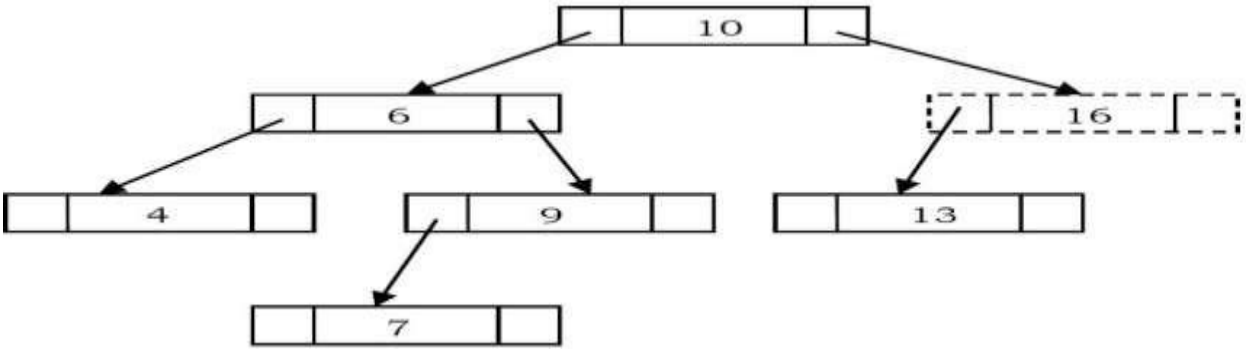
Else

Return findmin(root->left)

end

Finding Maximum Element in Binary Search Trees

In BSTs, the maximum element is the right-most node, which does not have right child.
In the BST below, the maximum element is **16**.



Algorithm:

Find max(struct node * root)

Begin

If(root= =Null)

Return root

Else if(root->right = =Null)

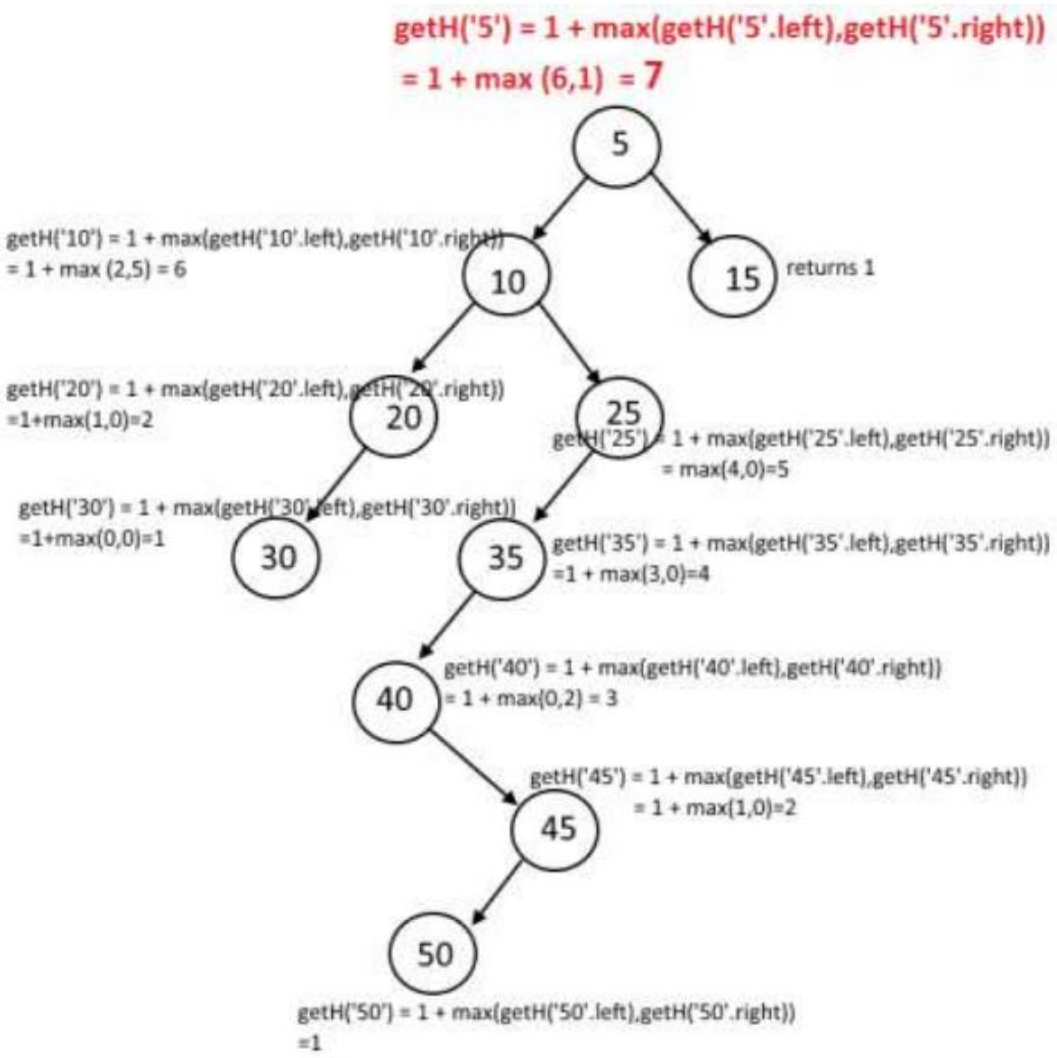
Return root;

Else

Return findmin(root->right)

End

Height of a Binary Search Tree:



Algorithm to find maximum depth or height of a binary tree as following:

- Get the height of left sub tree, say left Height
- Get the height of right sub tree, say right Height
- Take the Max (left Height, right Height) and add 1 for the root and return
- Call recursively.