Unit-6
Tree

Prepared By:

Dhaval Gandhi

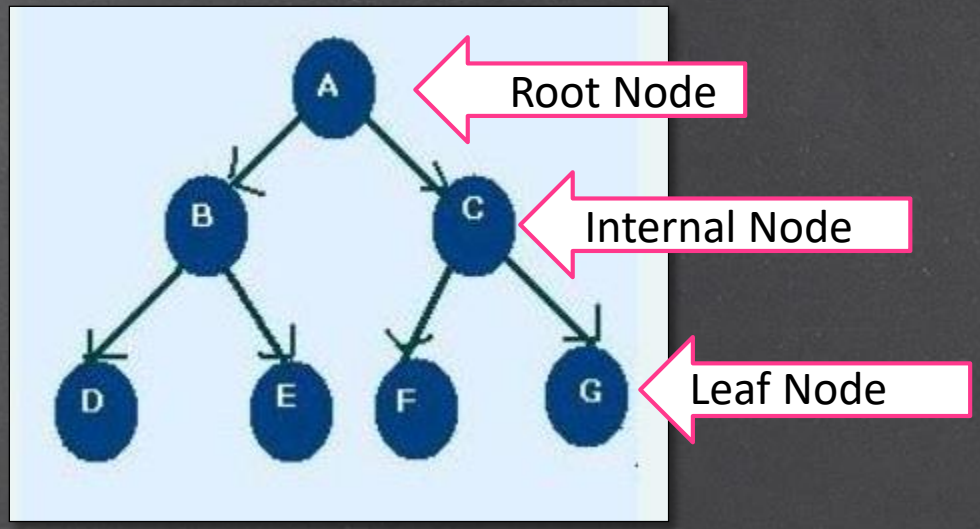
Learning Outcomes

- Non-linear data structures
- Binary trees : Complete Binary Tree,
- Basic Terms: level number, degree, in-degree and out-degree,
- leaf node, similar binary trees, copies of binary trees,
- directed edge, path, depth, General Tree, Forest.
- Binary Search Tree : Insertion of a node in binary tree, Deletion of a node in binary tree, Searching a node in binary tree
- Tree Traversal : Inorder, Preorder, Postorder
- Applications of binary tree

Tree

A Tree is defined as a finite set of one or more nodes such that:

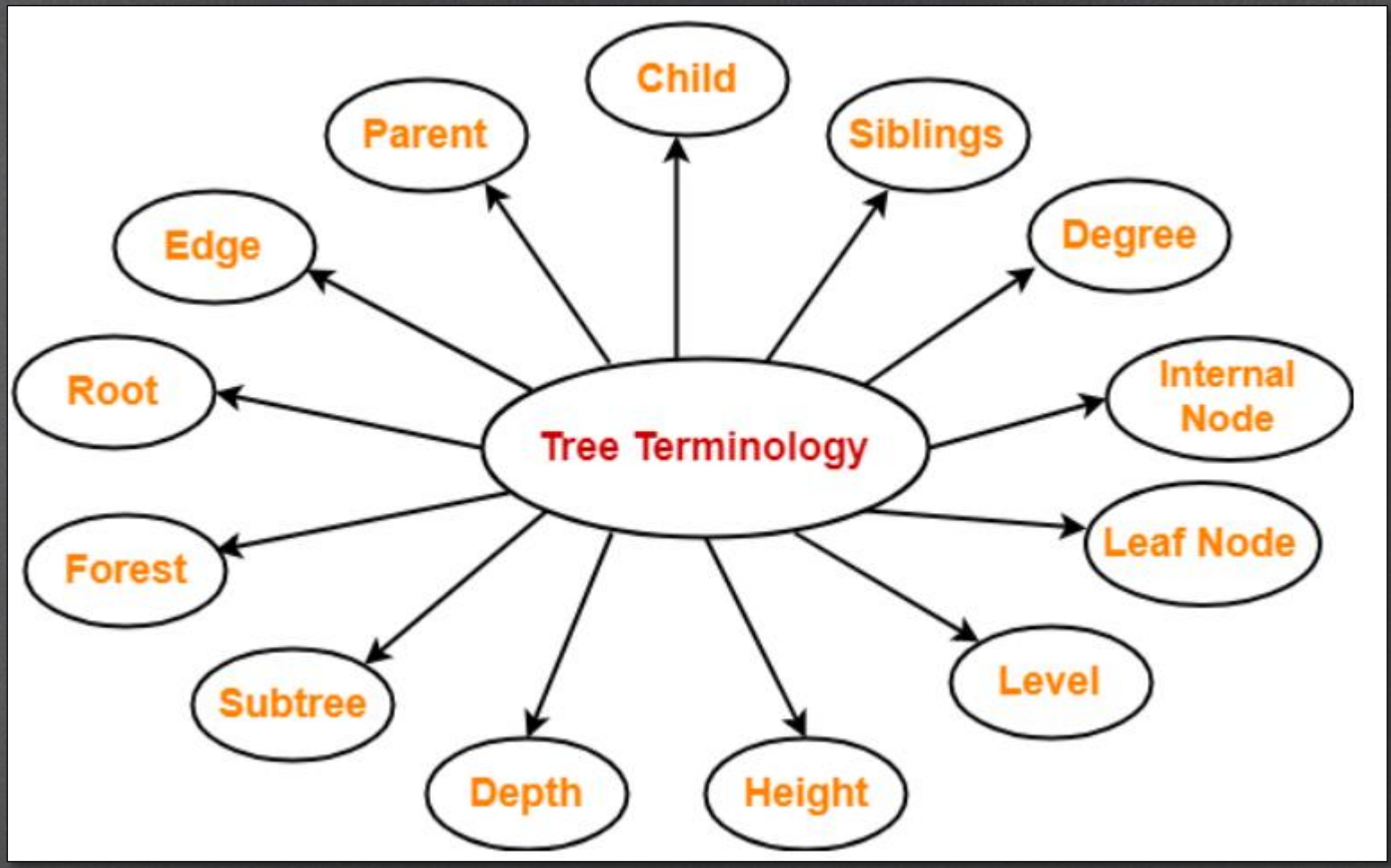
- There is a special node called root node having no predecessor.
- All the nodes in a tree except root node having only one predecessor.
- All the nodes in a tree having 0 or more successors.



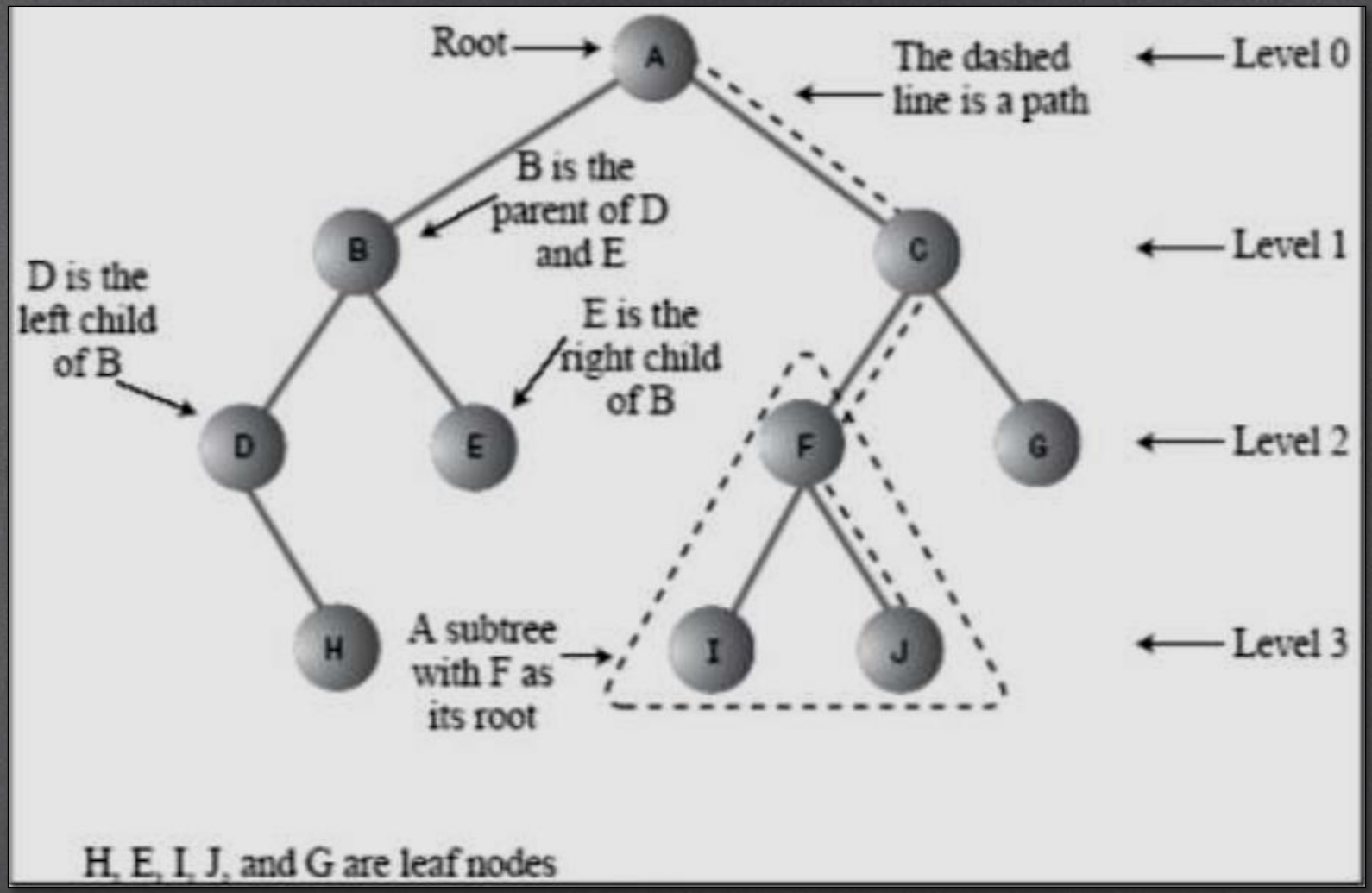
Tree

Node	Predecessor	Successor
A	0	2(B,C)
B	1(A)	2(D,E)
C	1(A)	2(F,G)
D	1(B)	0
E	1(B)	0
F	1(C)	0
G	1(C)	0

Tree Terminology



Tree Terminology



Tree Terminology

In Degree :-

- In a tree number of edges comes in to particular node is called In degree of that particular node.
- Root node having in degree always equals to 0 because it is the first node of tree.

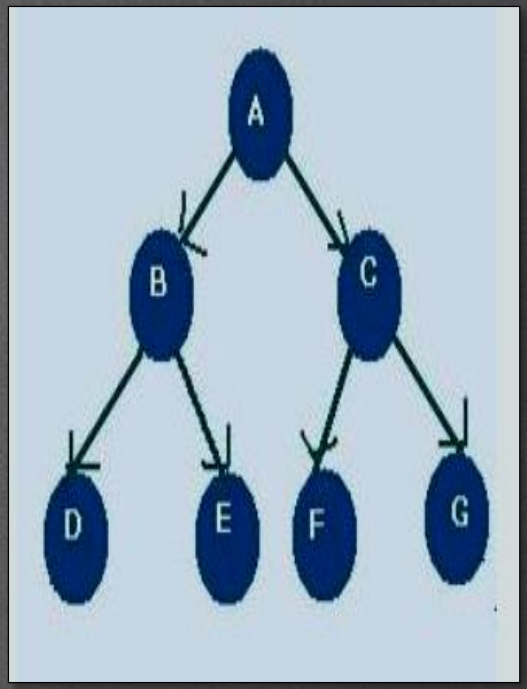
Out Degree :-

- In a tree number of edges comes out from particular node is called Out degree of that particular node.
- Leaf node having out degree always equals to 0 because it is the last node of tree having no further sub tree.

Degree or Total Degree :-

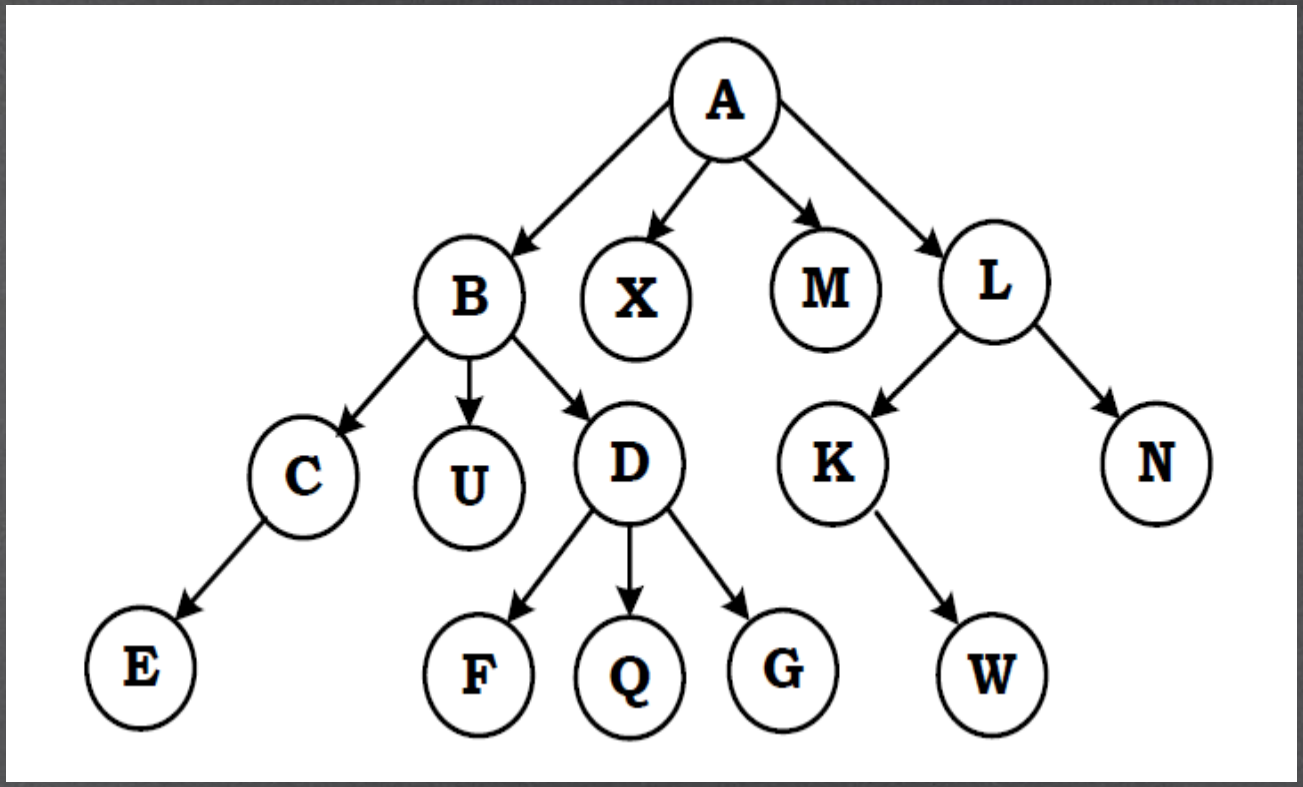
- In a tree the sum of edges comes into particular node and edges comes out from particular node is called degree of that particular node.
- It means Degree is the sum of in degree and out degree. It is also called total degree of node.

Tree Terminology



Node	In Degree	Out Degree	Total Degree
A	0	2	2
B	1	2	3
C	1	2	3
D	1	0	1
E	1	0	1
F	1	0	1
G	1	0	1

Tree Terminology



Tree Terminology

Root Node :-

- A node having in degree equal to 0 is called Root Node.
- The first node from where the tree originates is called as a root node.
- In any tree, there must be only one root node.
- We can never have multiple root nodes in a tree data structure.

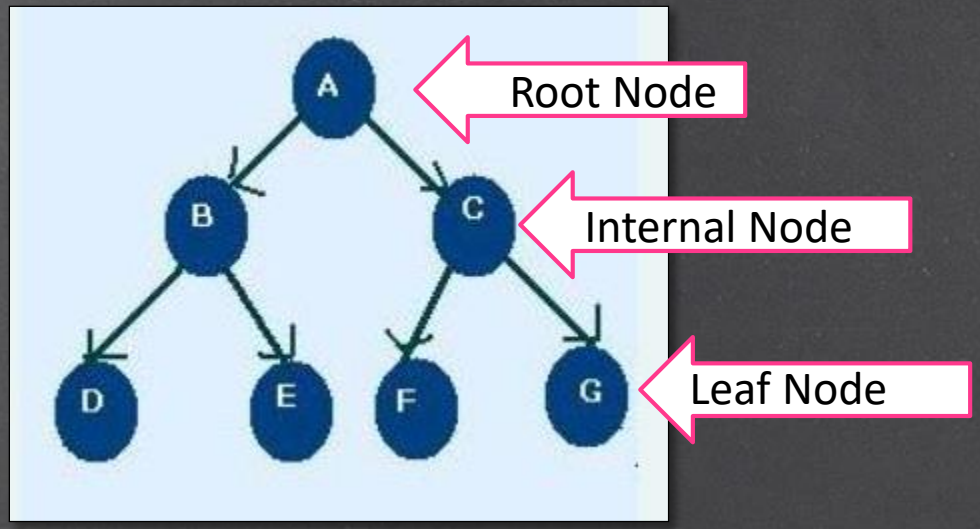
Leaf Node :-

- A node having out degree equals to 0 is called 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.
- Leaf Node does not have any sub tree.

Tree Terminology

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.



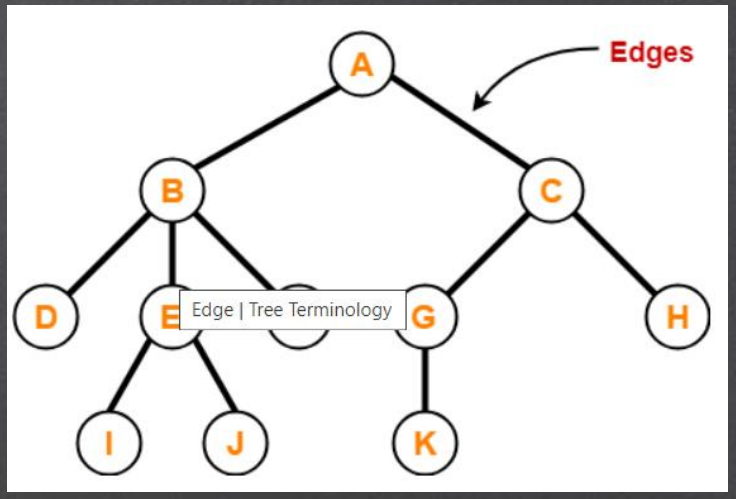
Tree Terminology

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.

Directed Edge :-

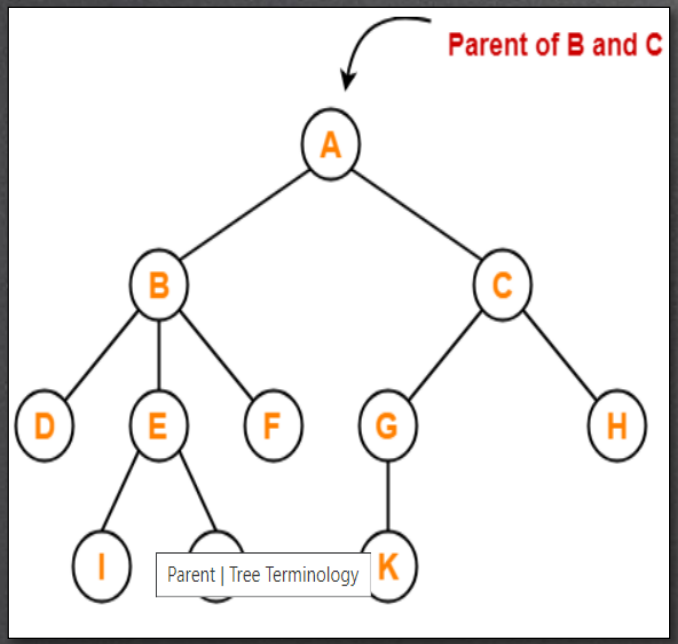
- In a tree an edge which is given direction from one node to another node then it is called directed edge.



Tree Terminology

Parent :-

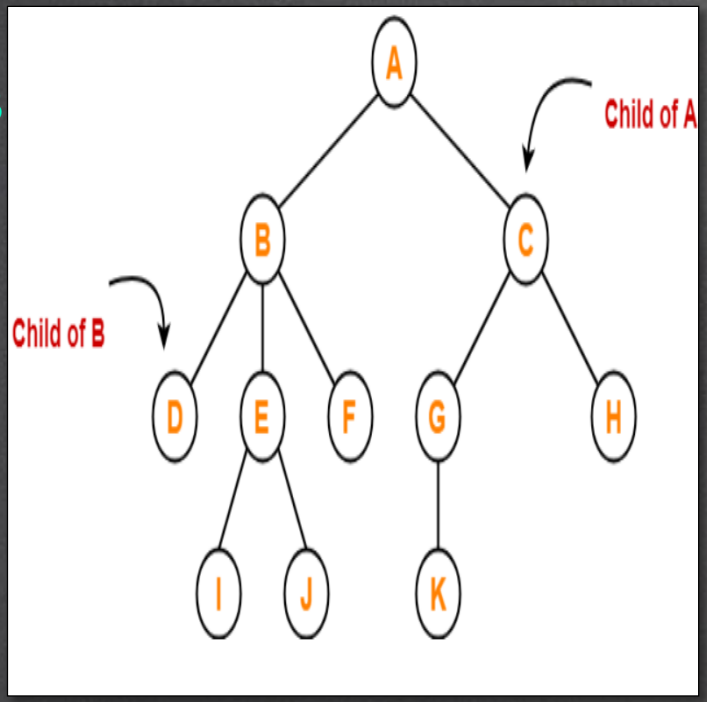
- The node which has a branch from it to any other node is called as a parent node.
- In other words, the node which has one or more children is called as a parent node.
- In a tree, a parent node can have any number of child nodes.
- Node A is the parent of nodes B and C
- Node B is the parent of nodes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



Tree Terminology

Child :-

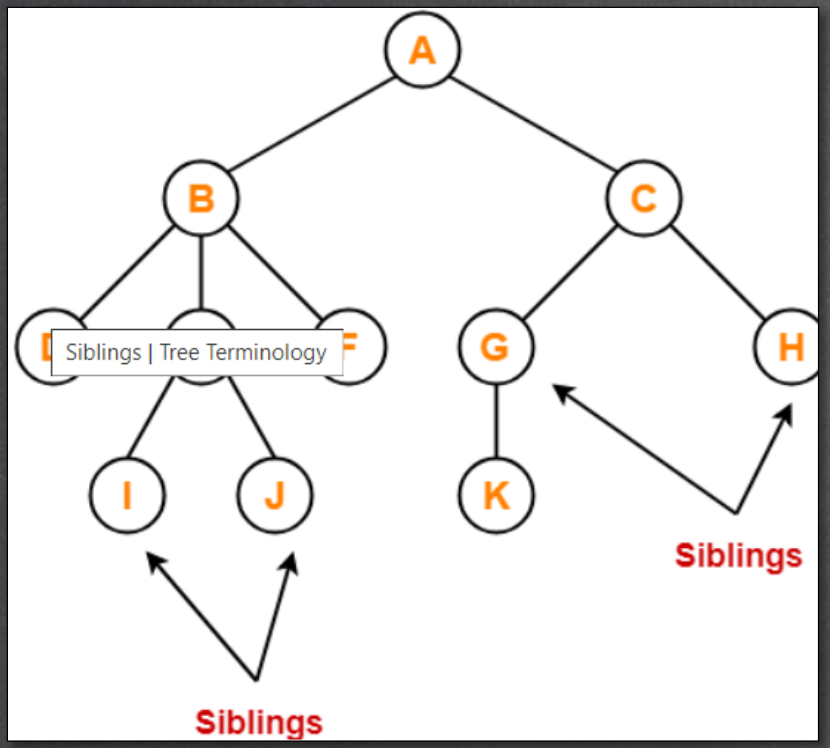
- The node which is a descendant of some node is called as a child node.
- All the nodes except root node are child nodes. In a tree, a parent node can have any number of child nodes.
- Nodes B and C are the children of node A
- Nodes D, E and F are the children of node B
- Nodes G and H are the children of node C
- Nodes I and J are the children of node E
- Node K is the child of node G



Tree Terminology

Siblings :-

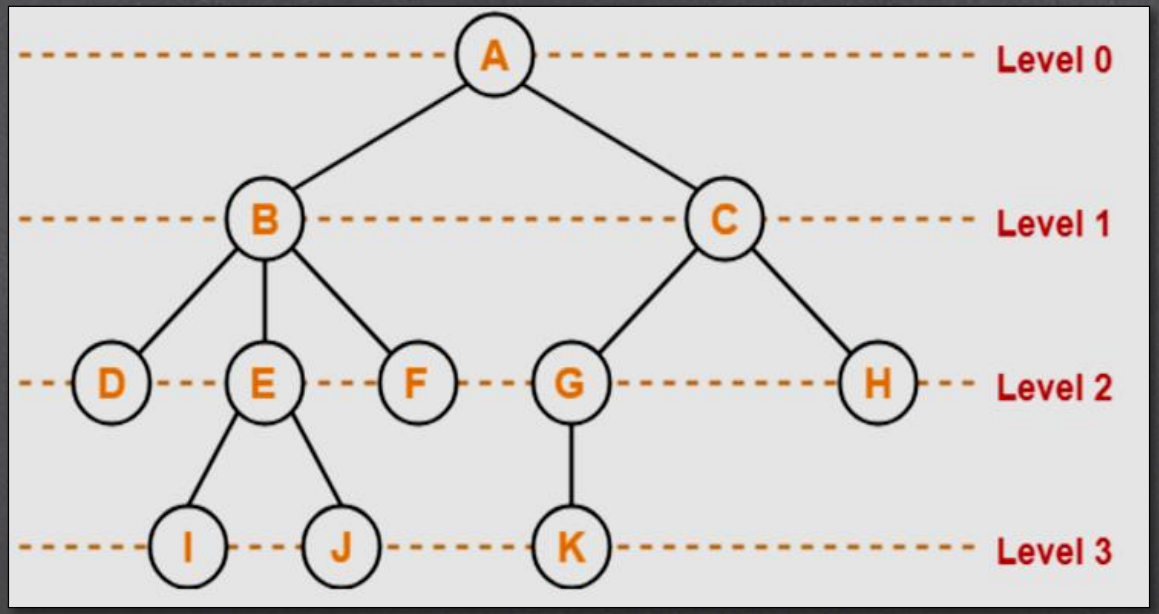
- Nodes which belong to the same parent are called as siblings.
- In other words, nodes with the same parent are sibling nodes.
- Nodes B and C are siblings
- Nodes D, E and F are siblings
- Nodes G and H are siblings
- Nodes I and J are siblings



Tree Terminology

Level :-

- In a tree distance between two nodes represented as level of a tree.
- 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.

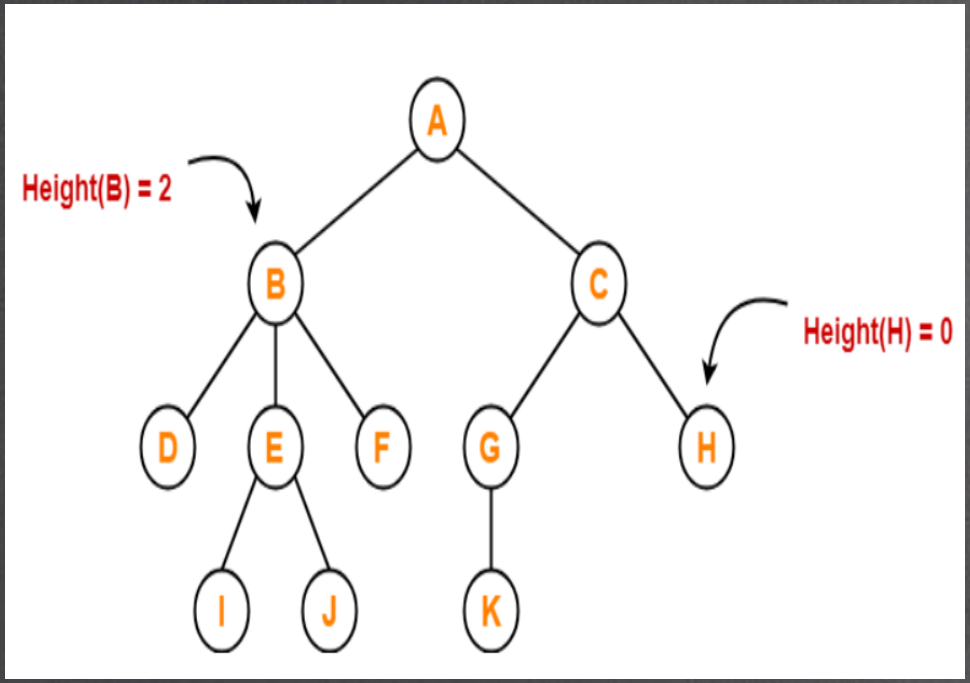


Tree Terminology

Height of a node :-

- 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.
- Height of all leaf nodes = 0

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 , J , K= 0



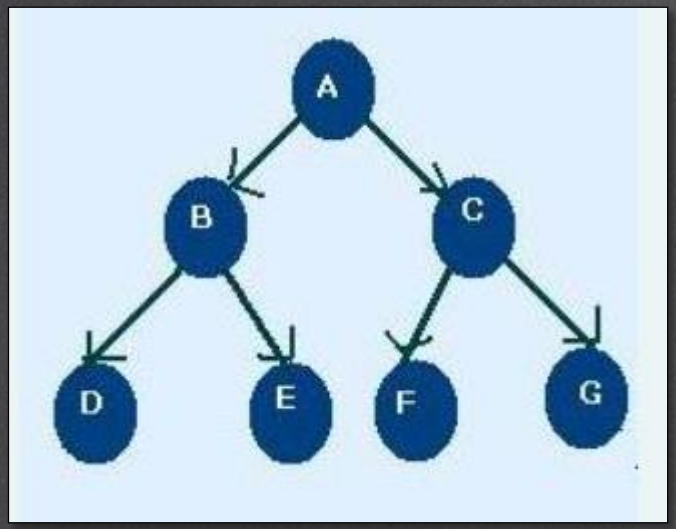
Height of a tree:

- The height of a tree is the height of the root.

Tree Terminology

Weight :-

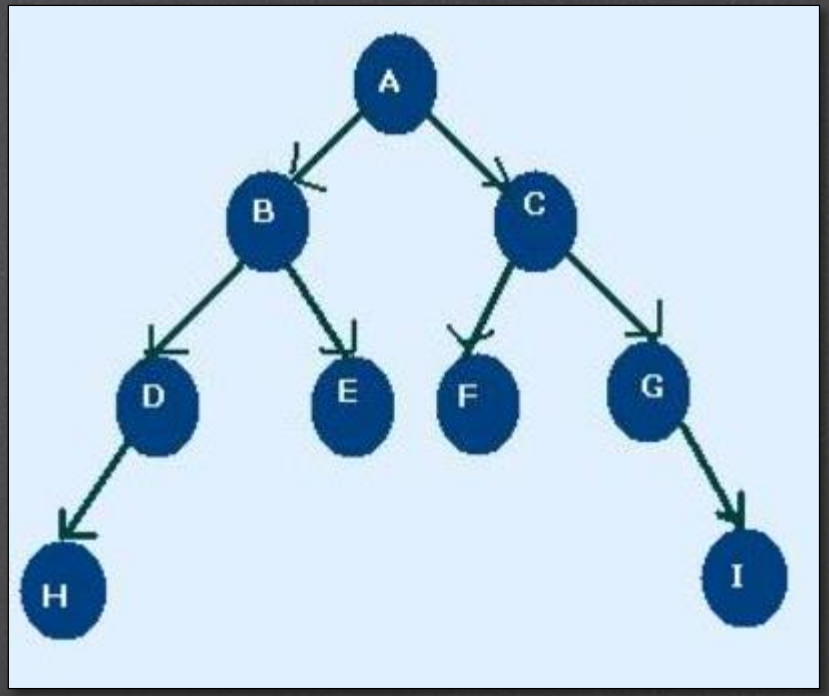
- Total number of leaf nodes available in tree is called weight of a tree.
- Weight of tree is 4
- D, E , F , G



Tree Terminology

Path :-

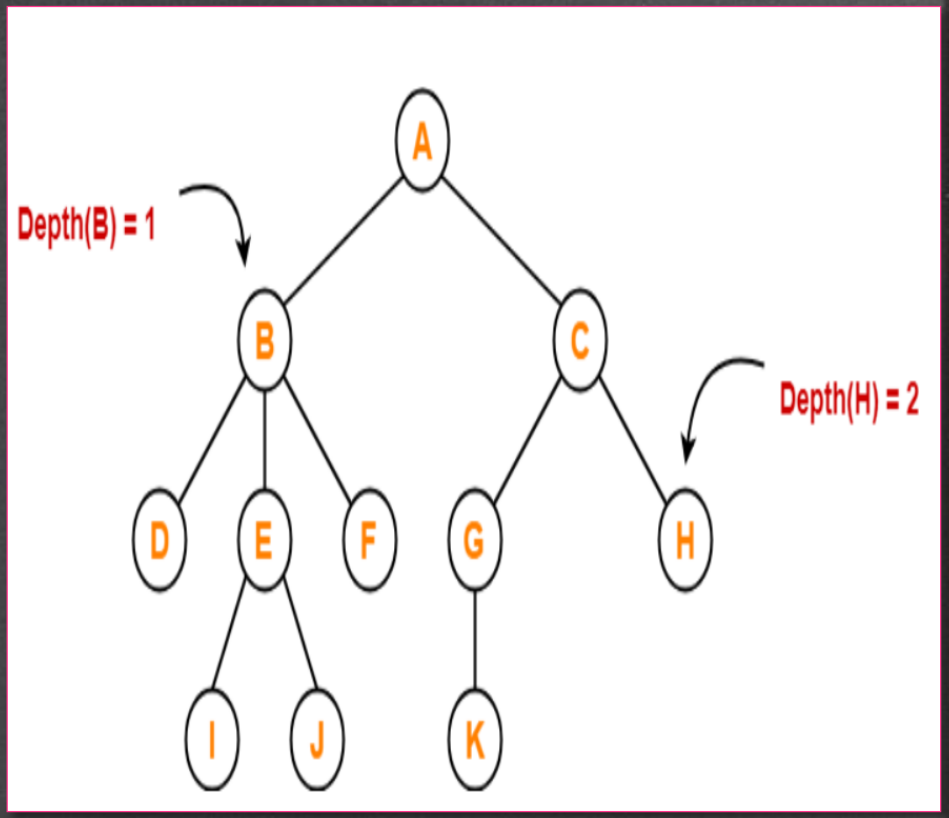
- A sequence of connecting edges from one node to another node is called Path.
- Path from Node A to H is given as A->B->D->H
- Path from Node A to I is given as A->C->G->I



Tree Terminology

Depth :-

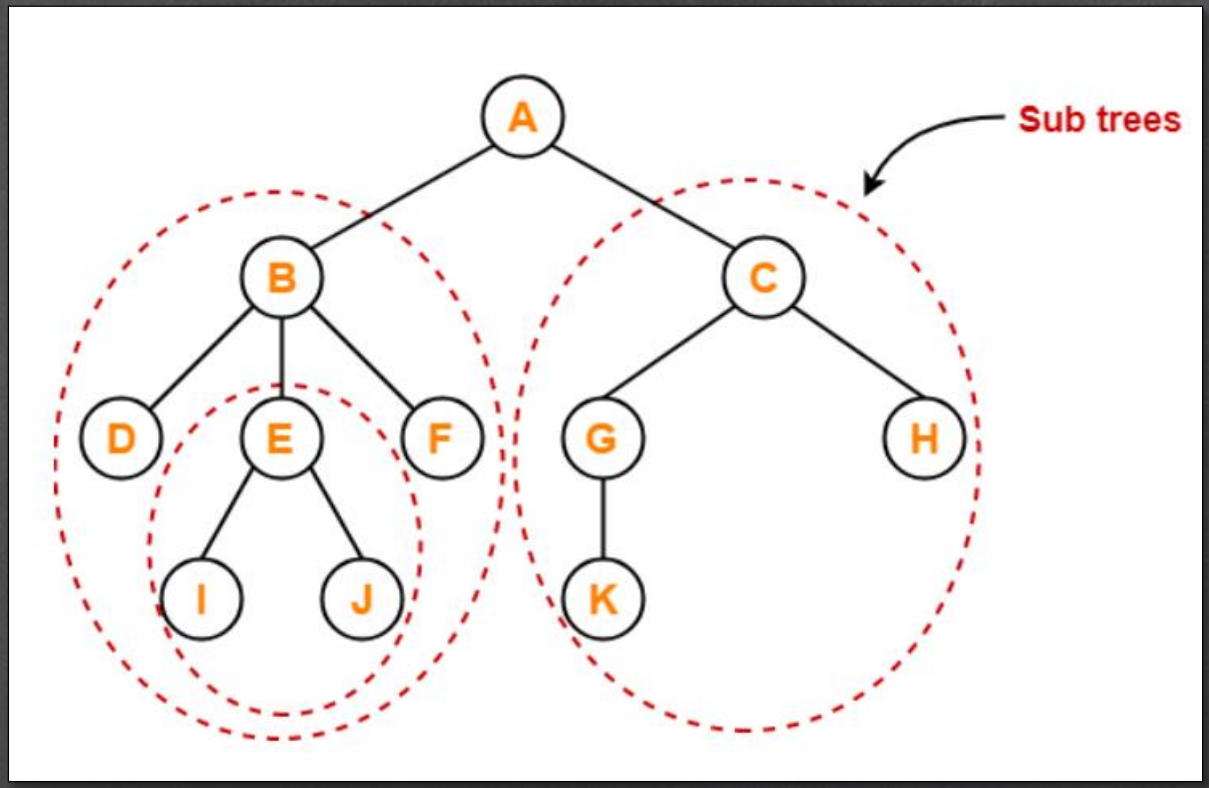
- Total number of edges from root node to a particular node is called as depth of that node.
- Depth of a tree is the total number of edges from root node to a leaf node in the longest path.
- Depth of the root node = 0
- Depth of node A = 0
- Depth of node B = 1
- Depth of node C = 1
- Depth of node 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



Tree Terminology

Subtree :-

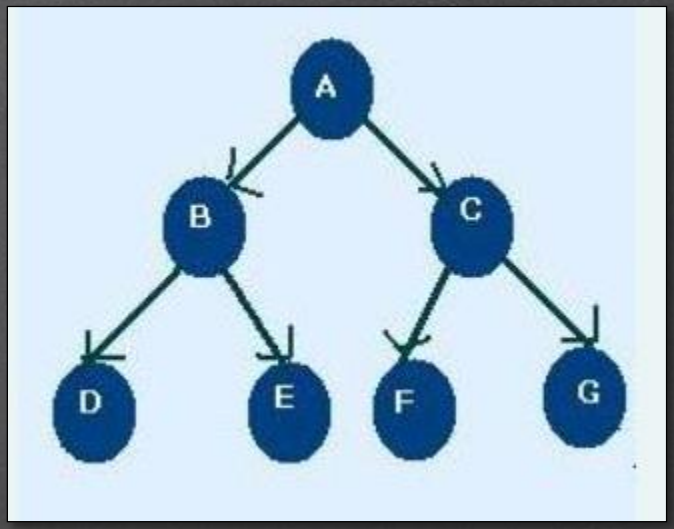
- In a tree, each child from a node forms a subtree recursively.
- Every child node forms a subtree on its parent node.

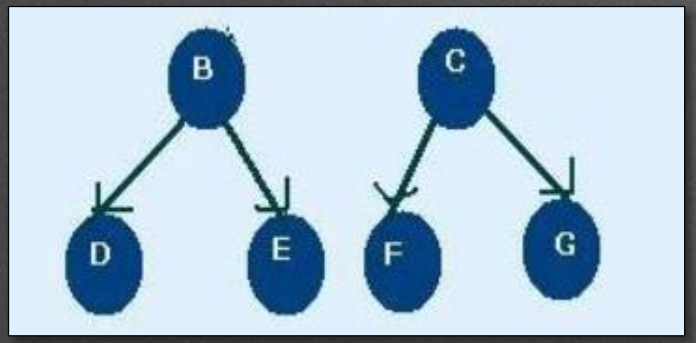


Tree Terminology

Forest :-

- A forest is a collection of two or more disjoint tree.
- Consider Following Figure in which a forest is a collection of two disjoint tree.

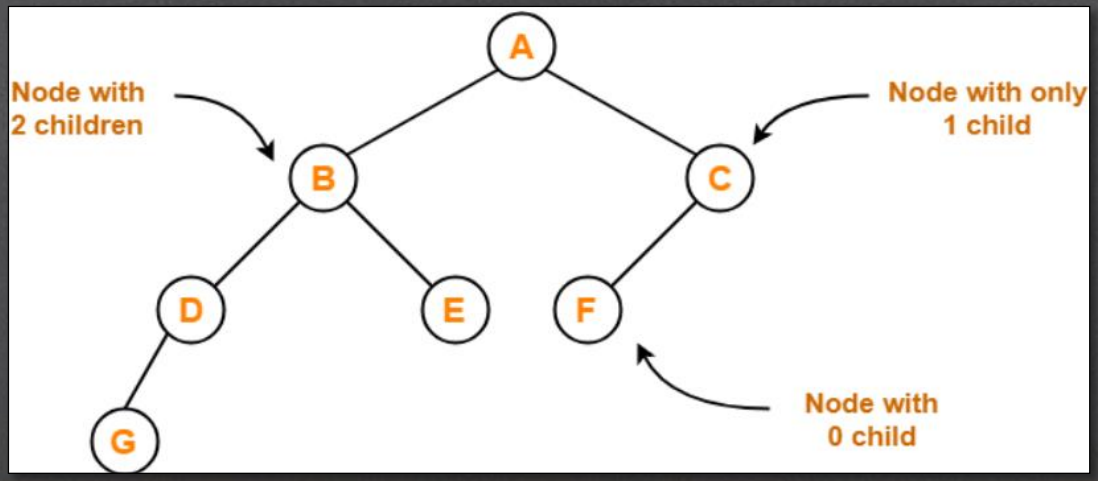




Tree Terminology

Binary Tree :-

- It is a special tree data structure in which each node can have at most 2 children.
- Thus, in Binary tree Each node has either 0 child or 1 child or 2 children.
- A Tree in which out degree of each node is less then or equal to 2 but not less then 0 is called binary tree.
- We can also say that a tree in which each node having either 0, 1 or 2 child is called binary tree.



Tree Terminology

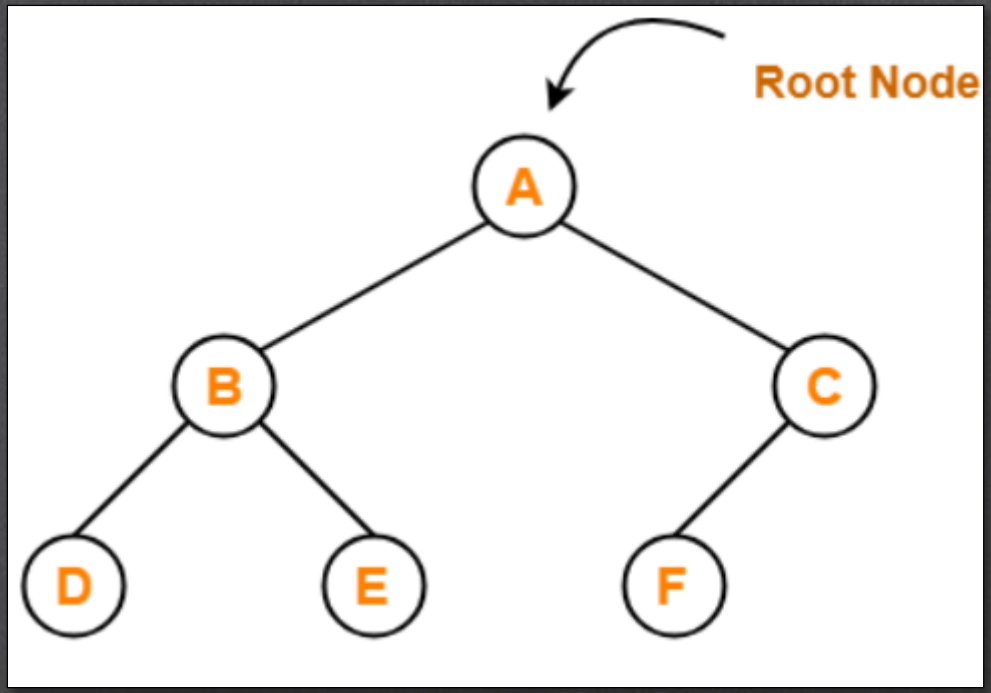
Types of Binary Trees-:-

- Rooted Binary Tree
- Full / Strictly Binary Tree
- Complete / Perfect Binary Tree
- Almost Complete Binary Tree
- Skewed Binary Tree

Tree Terminology

Rooted Binary Tree:-

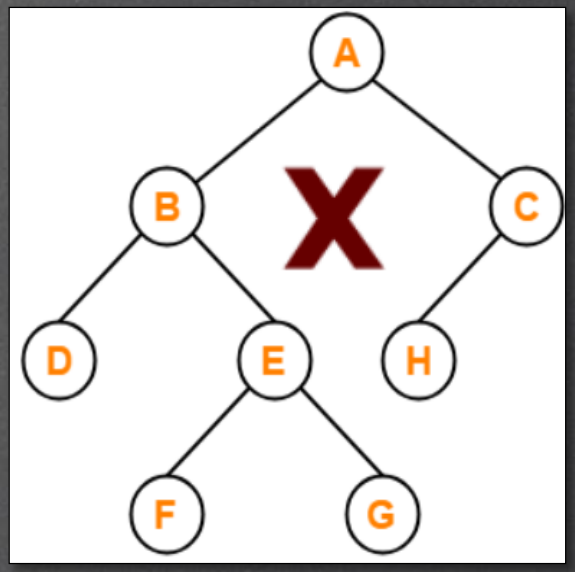
- A rooted binary tree is a binary tree that satisfies the following 2 properties-
- It has a root node.
- Each node has at most 2 children.

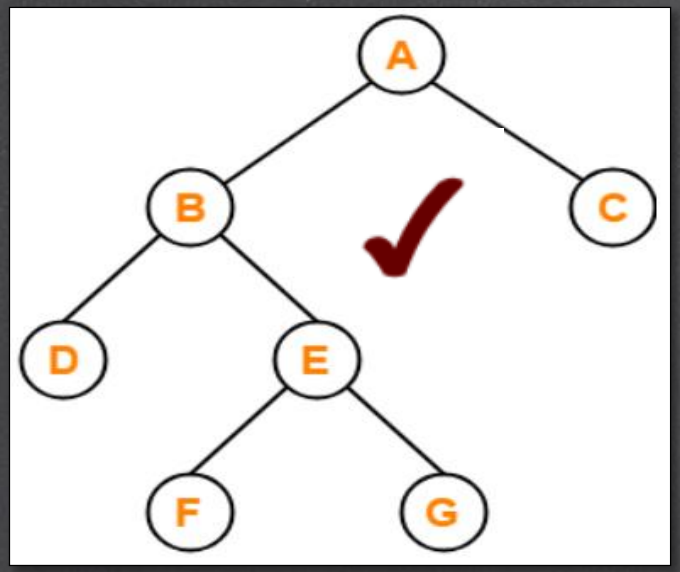


Tree Terminology

Full / Strictly Binary Tree :-

- A binary tree in which every node has either 0 or 2 children is called as a Full binary tree.
- Full binary tree is also called as Strictly binary.
- S.B.T with n leaves contains (2n-1) Nodes.



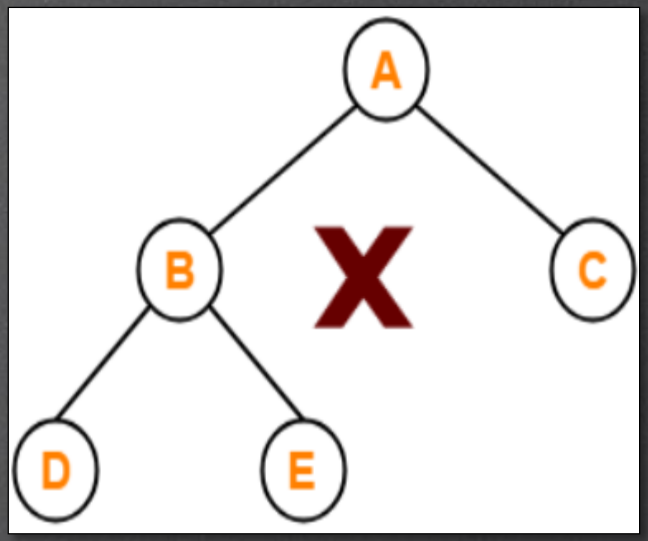


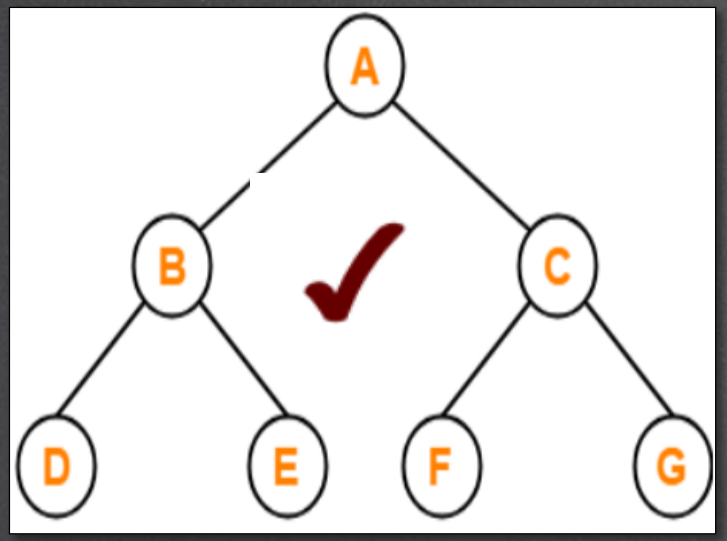
- First binary tree is not a full binary tree. This is because node C has only 1 child.

Tree Terminology

Complete / Perfect Binary Tree :-

- A complete binary tree is a binary tree that satisfies the following 2 properties-
- Every internal node has exactly 2 children.
- All the leaf nodes are at the same level.
- Complete binary tree is also called as Perfect binary tree.



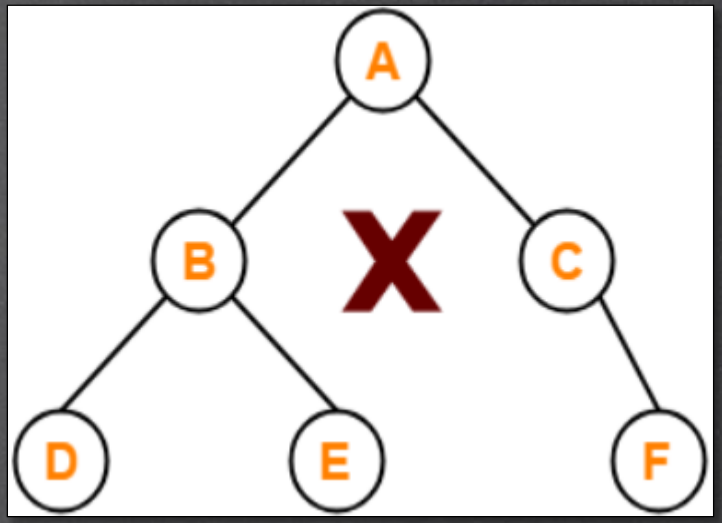


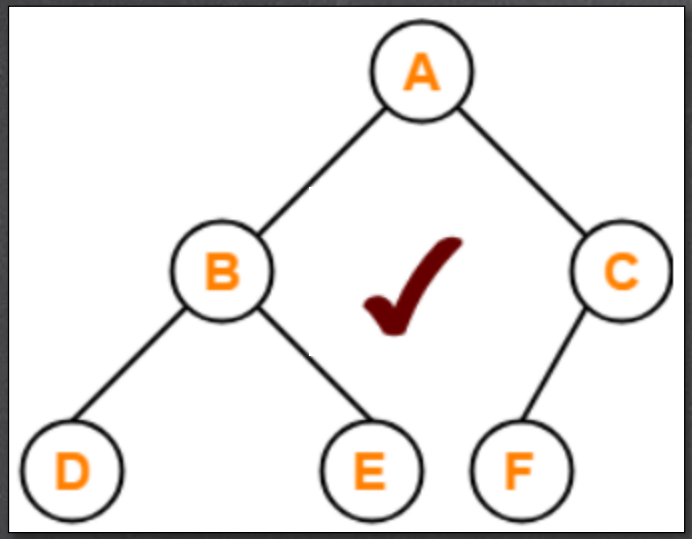
- First binary tree is not a complete binary tree. This is because all the leaf nodes are not at the same level

Tree Terminology

Almost Complete Binary Tree :-

- An almost complete binary tree is a binary tree that satisfies the following 2 properties-
- All the levels are completely filled except possibly the last level.
- The last level must be strictly filled from left to right.



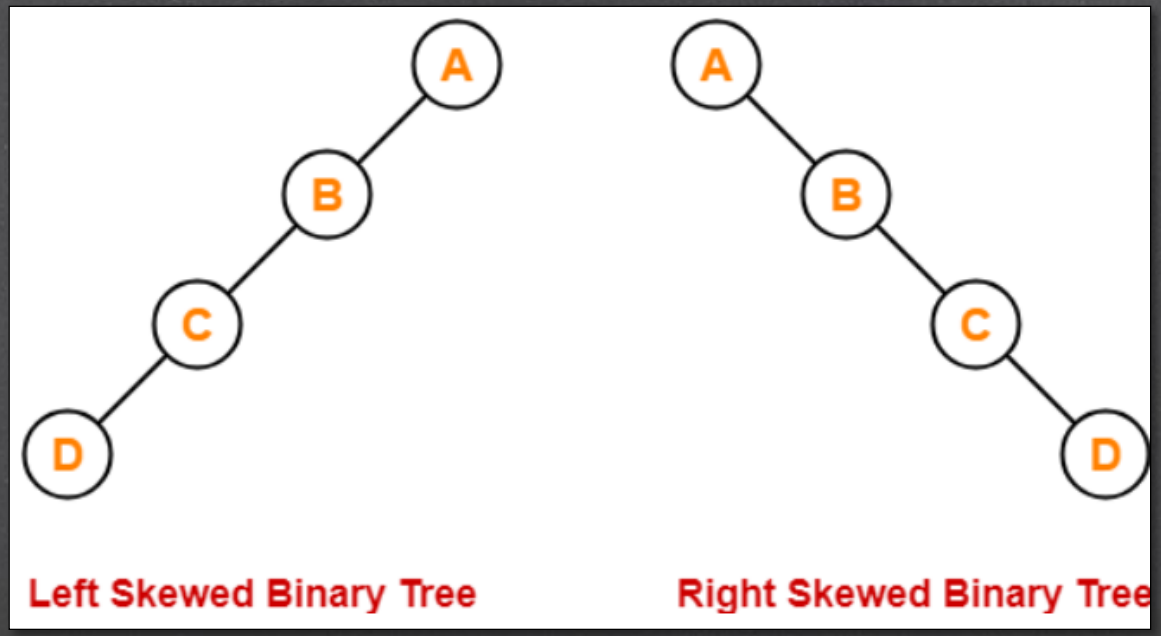


- First binary tree is not an almost complete binary tree. This is because the last level is not filled from left to right.

Tree Terminology

Skewed Binary Tree :-

- A skewed complete binary tree is a binary tree that satisfies the following 2 properties-
- All the nodes except one node has one and only one child.
- The remaining node has no child.
- A skewed binary tree is a binary tree of n nodes such that its depth is (n-1).



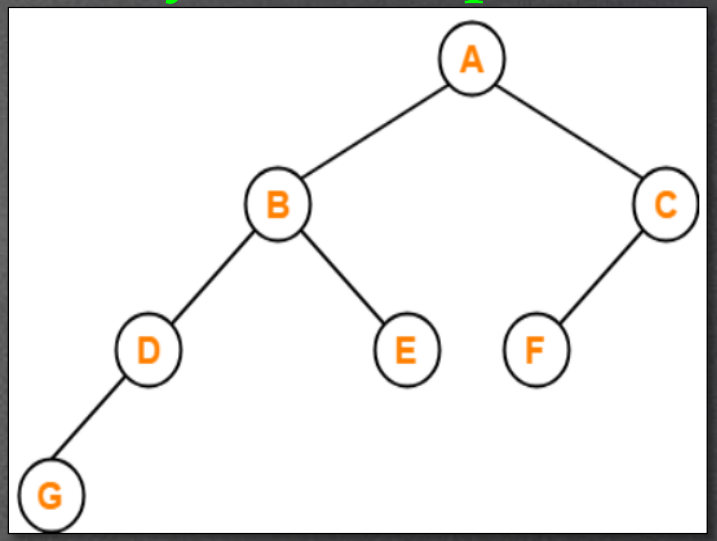
Tree Terminology

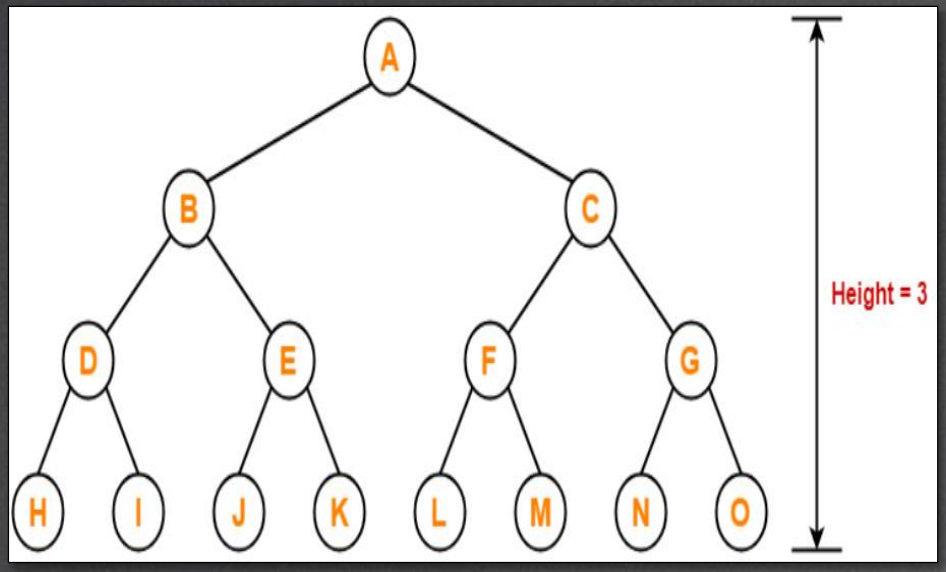
Binary Tree Properties :-

- Minimum number of nodes in a binary tree of height H = H + 1
- Maximum number of nodes in a binary tree of height H = $2^{H+1} - 1$
- Total Number of leaf nodes in a Binary Tree = Total Number of nodes with 2 children + 1
- Maximum number of nodes at any level 'L' in a binary tree= 2^{L}

Tree Terminology

Binary Tree Properties :-



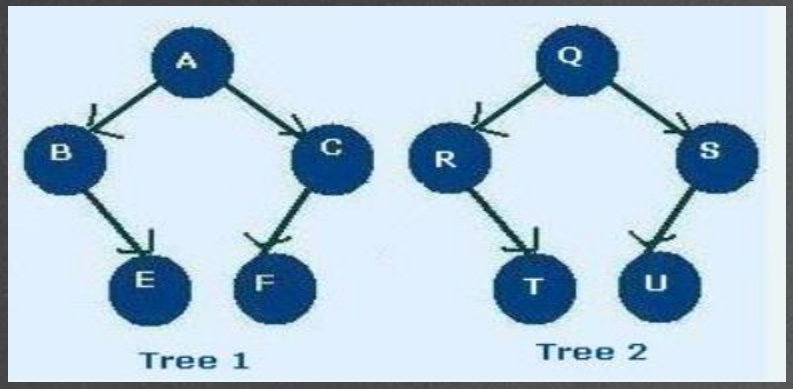


- Number of leaf nodes = 3
- Number of nodes with 2 children = 2
- $2^{3+1} - 1 = 16 - 1 = 15$ nodes
- $2^3 = 8$
- Thus, in a binary tree, maximum number of nodes that can be present at level-3 = 8.

Tree Terminology

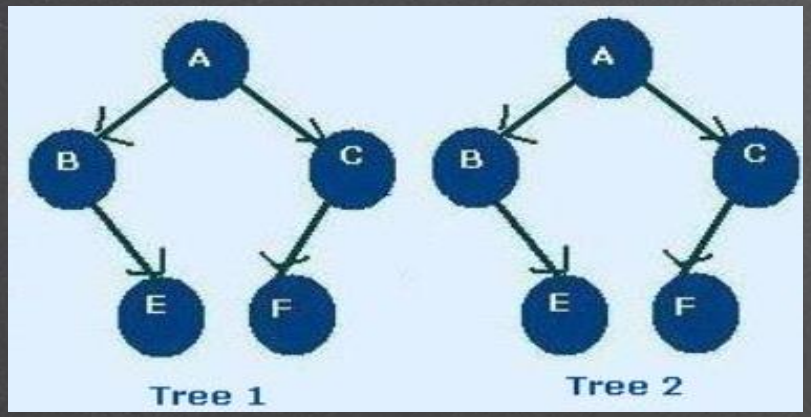
Similar Binary Tree :-

- If two binary trees are similar in structure then they are said to be similar binary trees.



Copies of Binary Tree:-

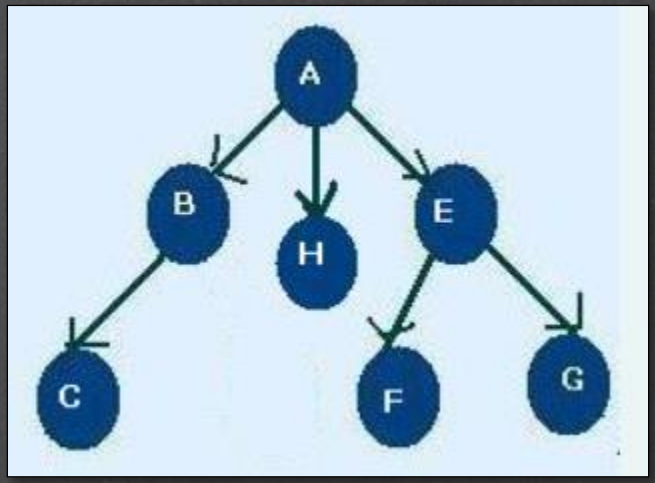
- If two binary trees are similar in structure and their corresponding nodes having same value then they are said to be copies of binary trees.



Tree Terminology

General Tree:-

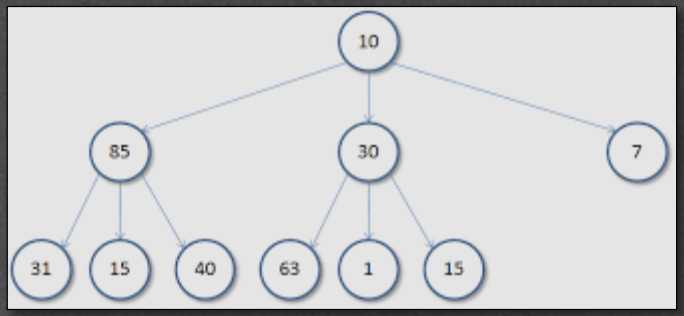
- A Tree in which each node having either 0 or more child nodes is called general tree. So we can say that a Binary Tree is a specialized case of General tree.
- General Tree is used to implement File System.
- Binary tree is a specialized case of general tree.



Tree Terminology

Ternary Tree:-

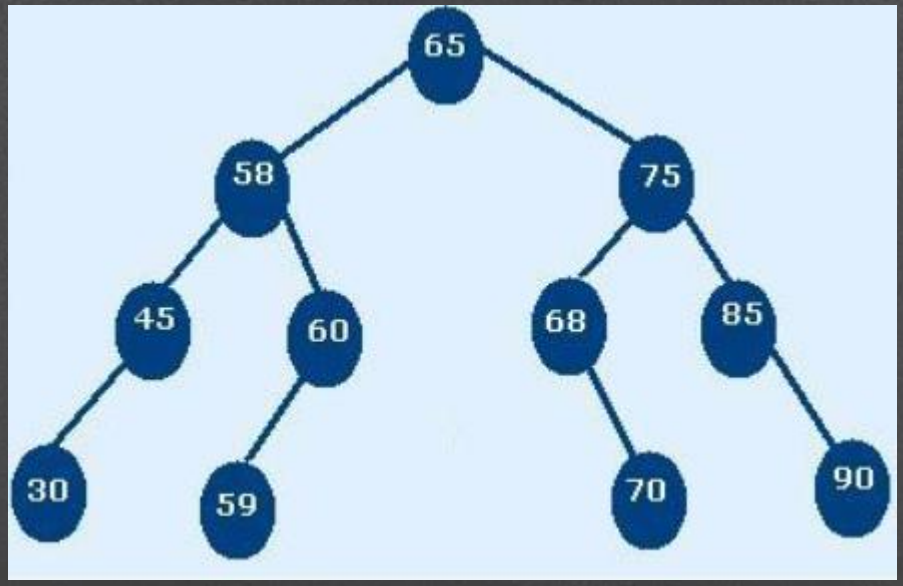
- General Tree in which number of sub trees for any node is not required to 0,1 OR 2.They are highly structured and 3 sub trees per nodes in which case is called a ternary tree.



- **M-Array Tree:-**
- Function in M-directed tree, the out degree of every node <= M is called M-Array Tree.

Binary Search Tree

- Binary Search Tree is a tree in which nodes are arranged such that all the nodes in a left sub tree having values less then root node and all the nodes in a right sub tree having values greater then root node.
- Thus we can say that Binary Search Tree is an ordered binary tree



Binary Search Tree

- Binary Search Tree having two characteristics:
- **All the nodes in a left sub tree having values less then root node.**
- **All the nodes in a right sub tree having values greater then root node.**
- Because of the above mentioned characteristics Binary Search tree allows faster insertion, deletion and searching facility.

Binary Search Tree

Four basic BST operations

1. Traversal
2. Search
3. Insertion
4. Deletion

Binary Search Tree

C representation for Binary tree:

```
struct bnode
{
        int info;
        struct bnode  *left;
        struct bnode  *right;
};
struct bnode *root=NUl
```

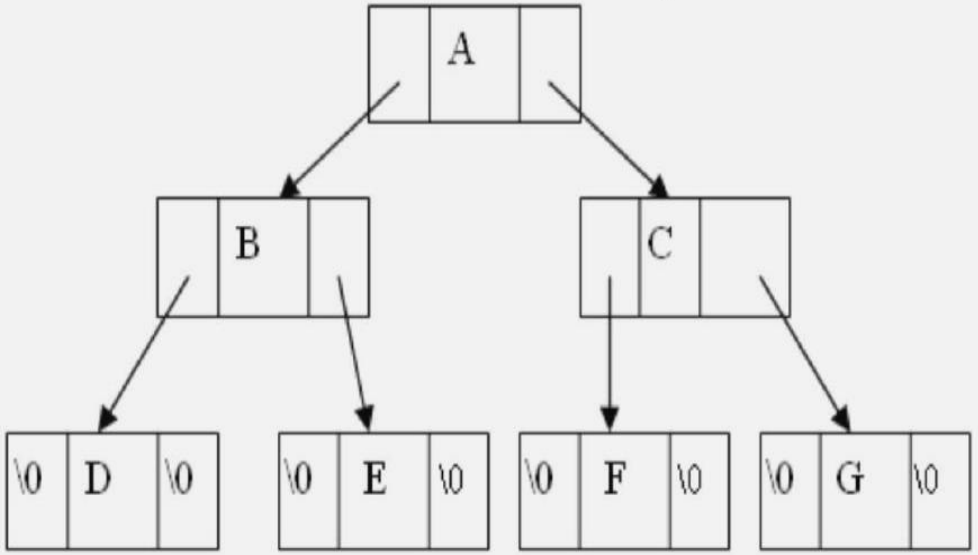


Fig: Structure of Binary tree

Binary Search Tree

```
class Node:
    def __init__(self, info):
        self.lptr = None
        self.rptr = None
        self.info = info
```

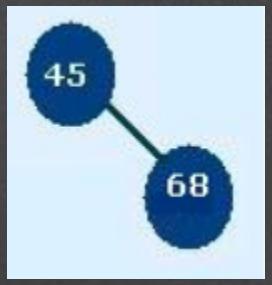
Binary Search Tree

Insertion of a Node in Binary Search Tree :-

45 68 35 42 15 64 78

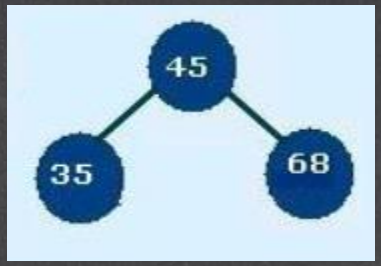
- Step 1: Insert 45:- First element is 45 so it is inserted as a root node of the tree.

- Step 2: Insert 68:-
- First we compare 68 with the root node which is 45. Since the value of 68 is greater then 45 so it is inserted to the right of the root node.

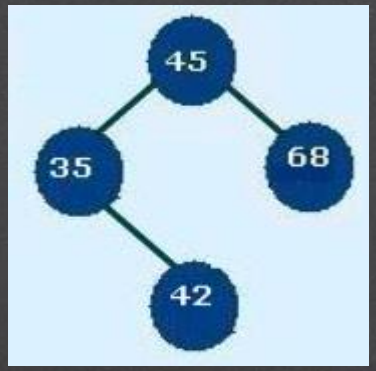


Binary Search Tree

- Step 3:**Insert 35:-** First we compare 35 with the root node which is 45. Since the value of 35 is less then 45 so it is inserted to the left of the root node.

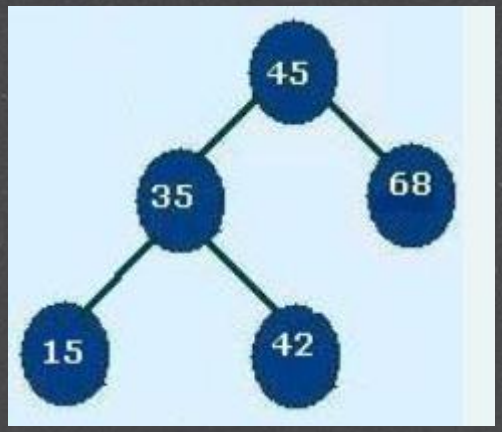


- Step 4:**Insert 42:-** First we compare 42 with the root node which is 45. Since the value of 42 is less then 45 so it is inserted to the left of the root node. But the root node has already one left node 35. So now we compare 42 with 35. Since the value of 42 is greater then 35 we insert 42 to the right of node 35.



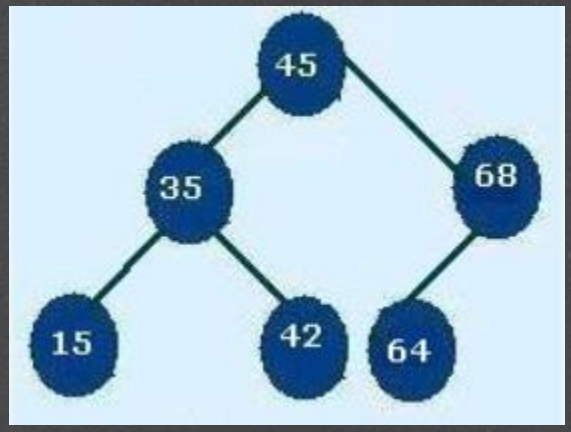
Binary Search Tree

- Step 5: **Insert 15:-**
- First we compare 15 with the root node which is 45. Since the value of 15 is less then 45 so it is inserted to the left of the root node. But the root node has already one left node 35. So now we compare 15 with 35. Since the value of 15 is less then 35 we insert 15 to the left of node 35.



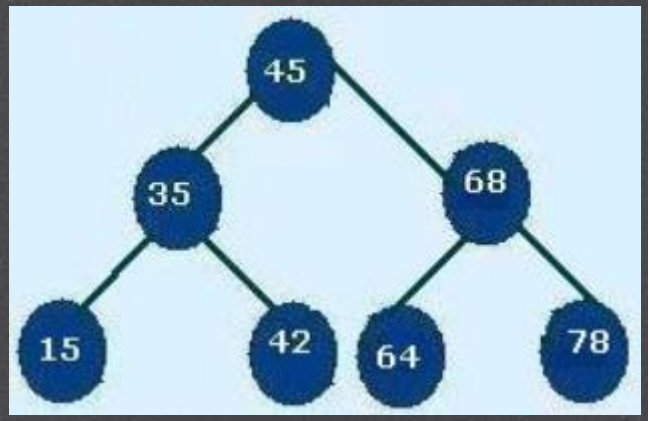
Binary Search Tree

- Step 6: **Insert 64:-**
- First we compare 64 with the root node which is 45. Since the value of 64 is greater then 45 so it is inserted to the right of the root node. But the root node has already one right node 68. So now we compare 64 with 68. Since the value of 64 is less then 68 we insert 64 to the left of node 68.



Binary Search Tree

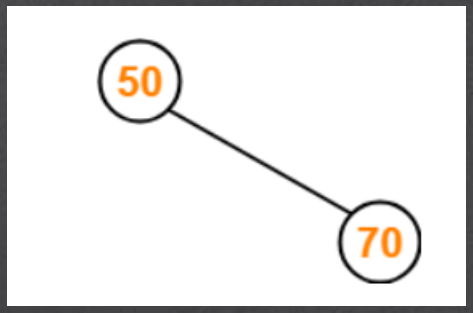
- Step 7: **Insert 78:-**
- First we compare 78 with the root node which is 45. Since the value of 78 is greater then 45 so it is inserted to the right of the root node. But the root node has already one right node 68. So now we compare 78 with 68. Since the value of 78 is greater then 68 we insert 78 to the right of node 68.



Binary Search Tree

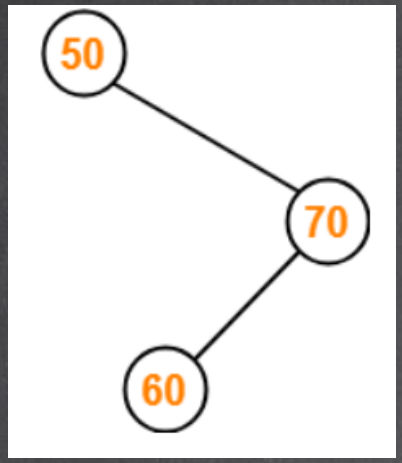
- Construct a Binary Search Tree (BST) for the following sequence of numbers-
- 50, 70, 60, 20, 90, 10, 40, 100
- Step1: Insert 50 :-

- Step2: Insert 70 :-
- As 70 > 50, so insert 70 to the right of 50.

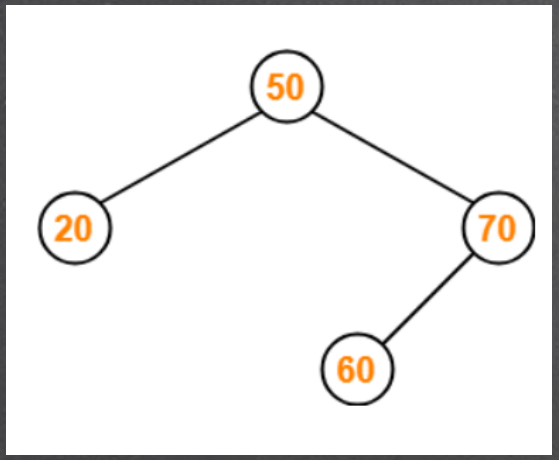


Binary Search Tree

- Step3 : Insert 60 :-
- As 60 > 50, so insert 60 to the right of 50.
- As 60 < 70, so insert 60 to the left of 70.

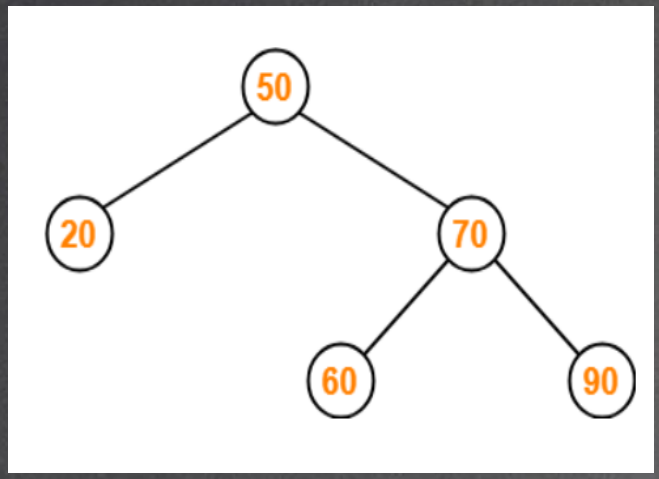


- Step 4 : Insert 20 :-
- As 20 < 50, so insert 20 to the left of 50.

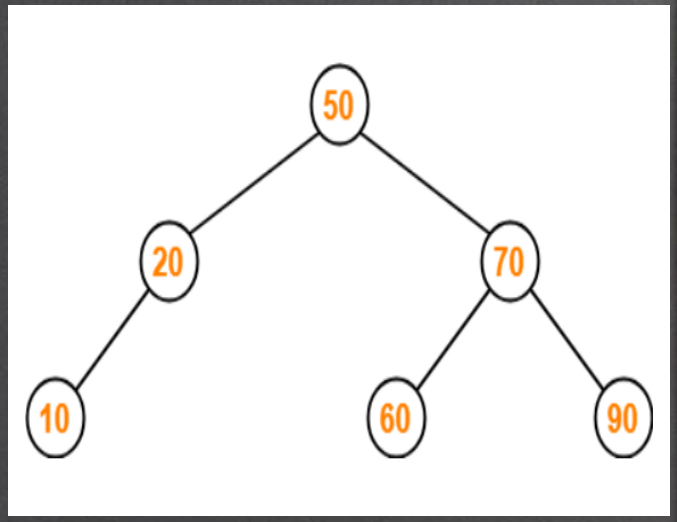


Binary Search Tree

- Step 5 : Insert 90 :-
- As 90 > 50, so insert 90 to the right of 50.
- As 90 > 70, so insert 90 to the right of 70.

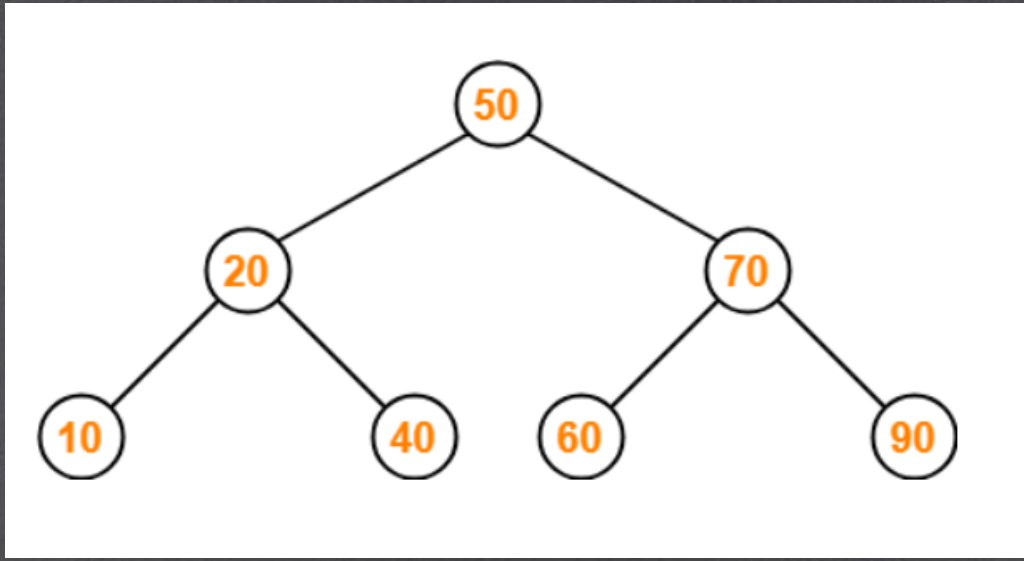


- Step 6 : Insert 10 :-
- As 10 < 50, so insert 10 to the left of 50.
- As 10 < 20, so insert 10 to the left of 20.



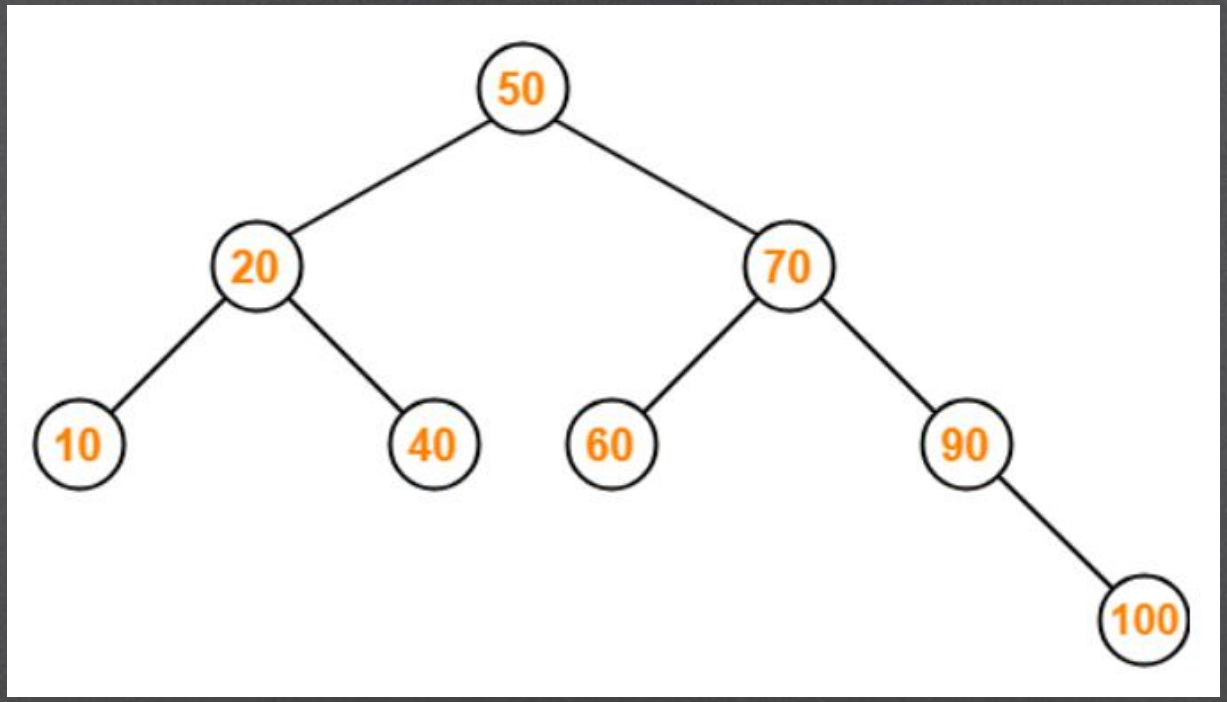
Binary Search Tree

- Step 7 : Insert 40 :-
- As 40 < 50, so insert 40 to the left of 50.
- As 40 > 20, so insert 40 to the right of 20.



Binary Search Tree

- Step 8 : Insert 100 :-
- As 100 > 50, so insert 100 to the right of 50.
- As 100 > 70, so insert 100 to the right of 70.
- As 100 > 90, so insert 100 to the right of 90.

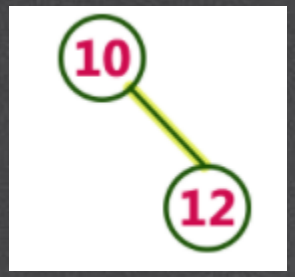


Binary Search Tree

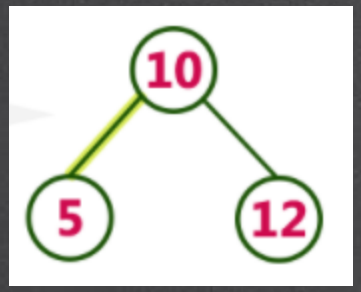
10,12,5,4,20,8,7,15 and 13

- Step 1: Insert 10:-

- Step 2: Insert 12:-
- 12> 10 so Put on R.S.T.



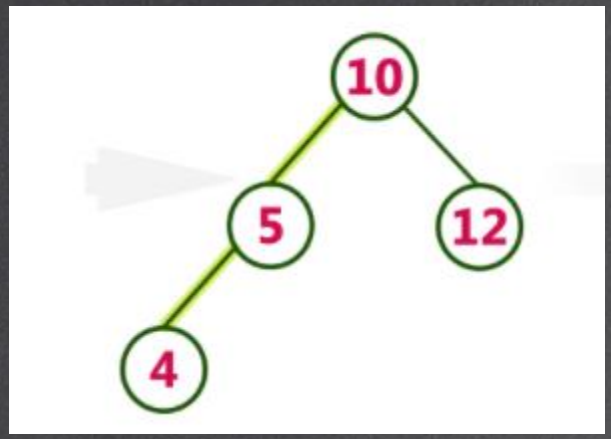
- Step 3: Insert 5:-
- 5< 10 so put on L.S.T.



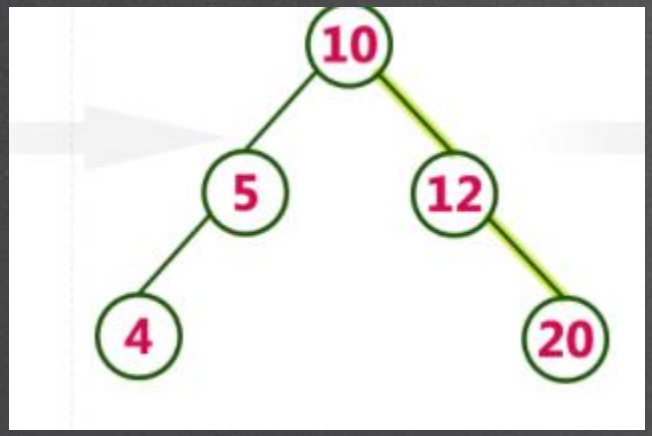
Binary Search Tree

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

- Step 4: Insert 4:-
- 4< 10 so Put on L.S.T.
- 4< 5 so Put on L.S.T. Of 5



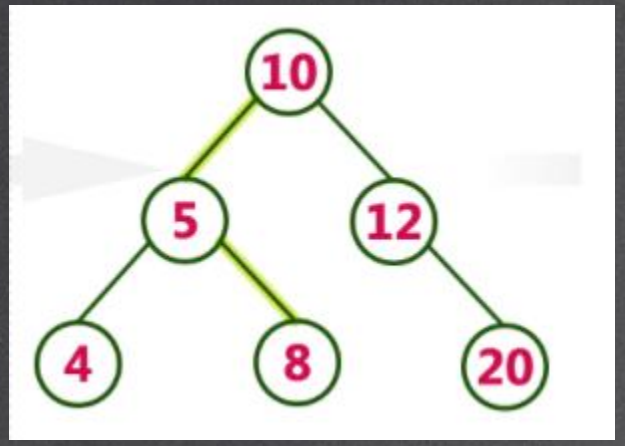
- Step 5: Insert 20:-
- 20>10 so Put on R.S.T.
- 20> 12 so Put on R.S.T. Of 12



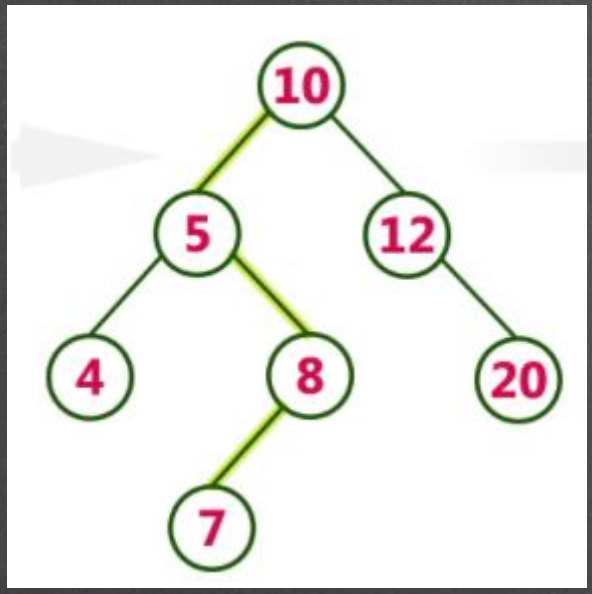
Binary Search Tree

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

- Step 6: Insert 8:-
- 8< 10 so Put on L.S.T.
- 8> 5 so Put on R.S.T. Of 5



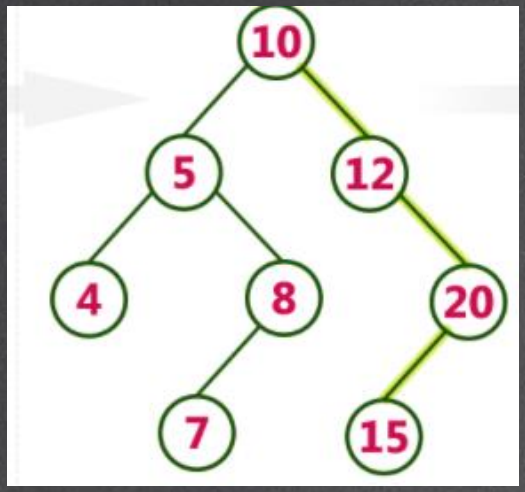
- Step 7: Insert 7:-
- 7< 10 so Put on R.S.T.
- 7 > 5 so Put on R.S.T. Of 5
- 7< 8 so Put on L.S.T. Of 8



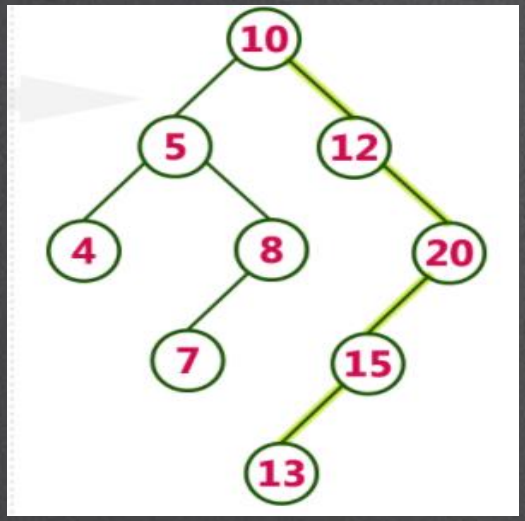
Binary Search Tree

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

- **Step 8: Insert 15:-**



- **Step 9: Insert 13:-**



Home Work

78,58,82,15,66,80,99,5

7,4,10,9,12,2,1

50,55,35,15,52,65,33,47,75,72,27,56

34,58,12,25,45,65,87,30,5,38

Binary Search Tree Insertion Algorithm

Step 1: If ROOT = None then
ROOT=NODE(INFO)
return

Step 2 : If INFO < ROOT.INFO then
If ROOT.LPTR =None then
ROOT.LPTR=NODE(INFO)
Else
ROOT.LPTR.INSERT(INFO)

Step 3: If INFO > ROOT.INFO then
If ROOT.RPTR =None then
ROOT.RPTR=NODE(INFO)
Else
ROOT.RPTR.INSERT(INFO)

Binary Search Tree Insertion Algorithm

```
# Insert method to create nodes
   def insert(self, info):
        if self.info:
            if info < self.info:
                if self.lptr is None:
                    self.lptr = Node(info)
                else:
                    self.lptr.insert(info)
            elif info > self.info:
                if self.rptr is None:
                    self.rptr = Node(info)
                else:
                    self.rptr.insert(info)
        else:
             self.info = info
```

Binary Search Tree Insertion

- Step 1 - Create a new Node 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 new Node.
- Step 4 - If the tree is Not Empty, then check whether the value of new Node is smaller or larger than the node (here it is root node).
- Step 5 - If new Node is smaller than or equal to the node then move to its left child. If new Node 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 new Node as left child if the new Node is smaller or equal to that leaf node or else insert it as right child.

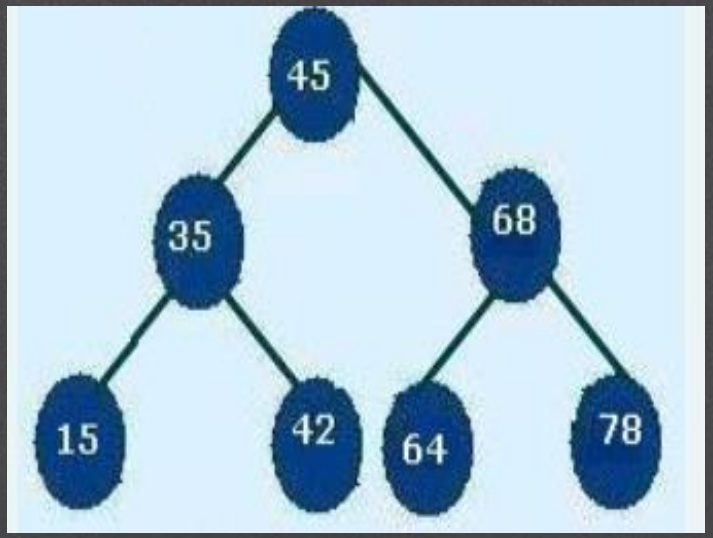
on Program

```
class Node:
    def __init__(self, info):
        self.lptr = None
        self.rptr = None
        self.info = info
# Insert method to create nodes
    def insert(self, info):
        if self.info:
            if info < self.info:
                if self.lptr is None:
                    self.lptr = Node(info)
                else:
                    self.lptr.insert(info)
            elif info > self.info:
                if self.rptr is None:
                    self.rptr = Node(info)
                else:
                    self.rptr.insert(info)
        else:
            self.info = info
# Print the tree
    def Display(self,root):
        if(root==None):
            return
        print(root.info)
        if (root.lptr!=None):
            self.Display(root.lptr)
        if (root.rptr!=None):
            self.Display(root.rptr)
root = Node(12)
```

```
12
6
3
14
20
```

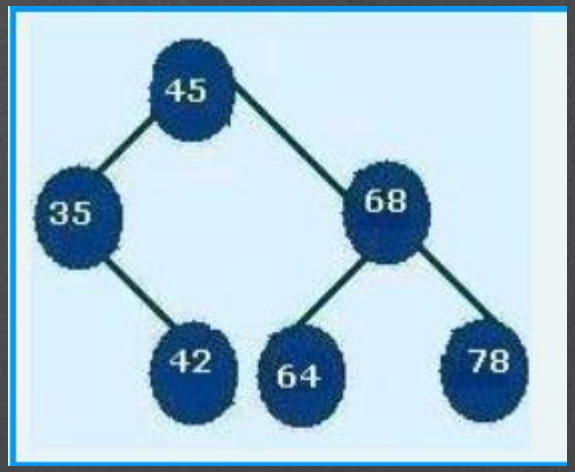
Binary Search Tree Deletion

- In order to delete node from binary search tree we have to consider three possibilities.
- A node to be deleted has no sub tree.
- A node to be deleted has only one sub tree (left or right).
- A node to be deleted has two sub trees (left and right).



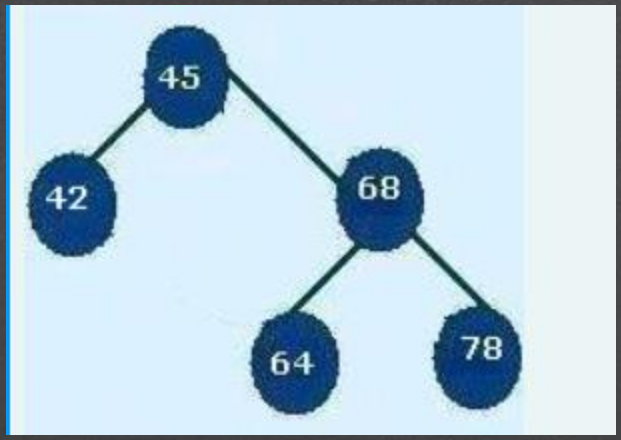
Binary Search Tree Deletion

- Case 1:If a node to be deleted has no sub tree. In such case a node is deleted directly.
- Delete 15
- Here node 15 has no left or right sub tree so we can delete it directly. After deleting node 15 tree looks like as shown below:



Binary Search Tree Deletion

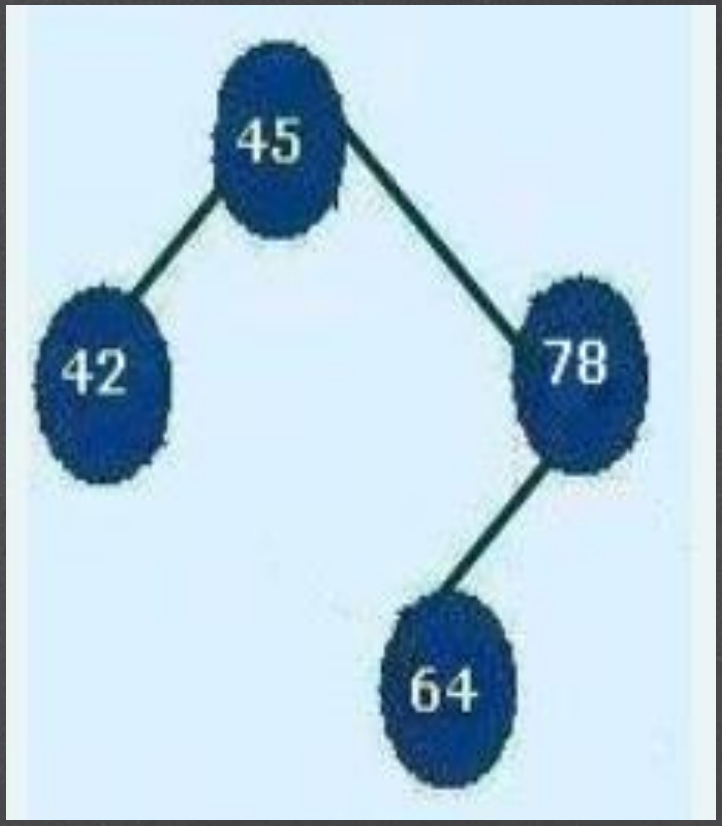
- **Case 2:** If a node to be deleted has only one sub tree either left sub tree or right sub tree. In such case the sub tree of the deleted node is linked directly with the parent of the deleted node.
- **Delete 35:-**Here node 35 has one right sub tree so we link this right sub tree with parent of node 35 which is 45. Now the node which we want to link with node 45 is 42 and value of 42 is less than 45 so we link it as a left sub tree of node 45. After deleting node 35 tree looks like as shown below:



Binary Search Tree Deletion

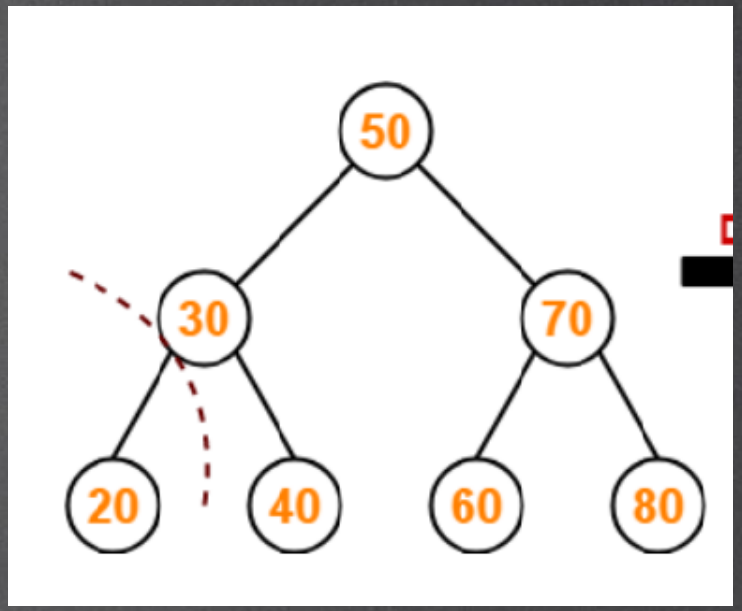
- Possibility 3: If a node to be deleted has two sub trees left as well as right. In such case we have to perform following steps:
- (a) Find in order successor of the node to be deleted.
- (b) Append the right sub tree of the in order successor to its grand parent.
- (c) Replace the node to be deleted with its in order successor.
- Suppose we want to delete node 68. Here node 68 has both left and right sub tree.
- So first we have to find InOrder successor of node 68 which is 78.
- 42 45 64 68 78 Now replace the node to be deleted with it's Inorder successor.
- So we replace node 68 with node 78. After deleting node 68 tree looks like as shown below:

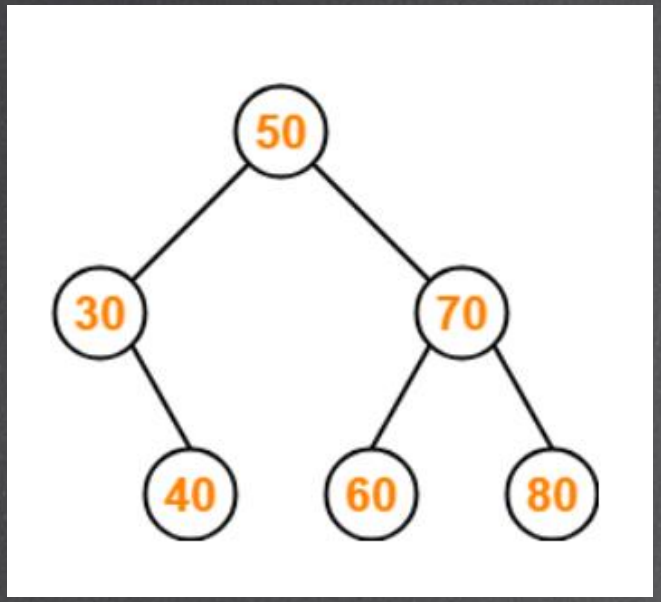
Binary Search Tree Deletion



Binary Search Tree Deletion

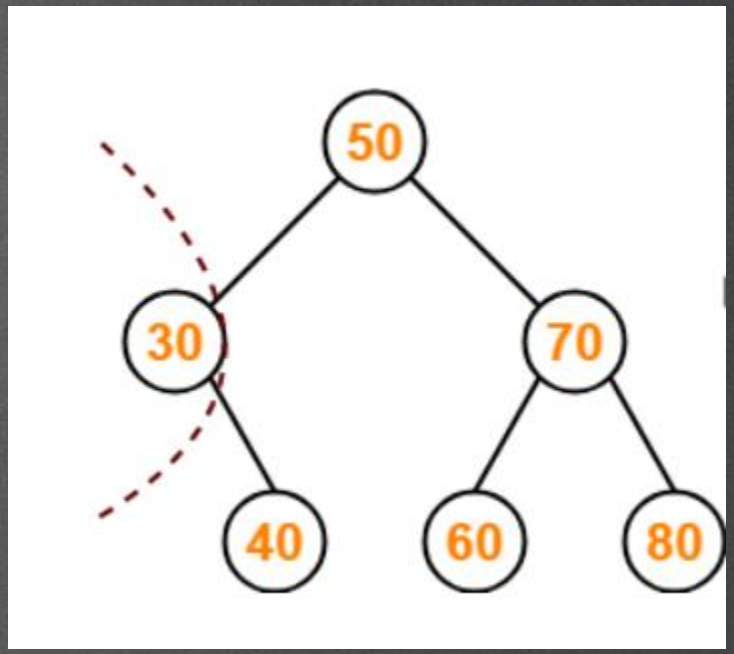
- Delete 20:

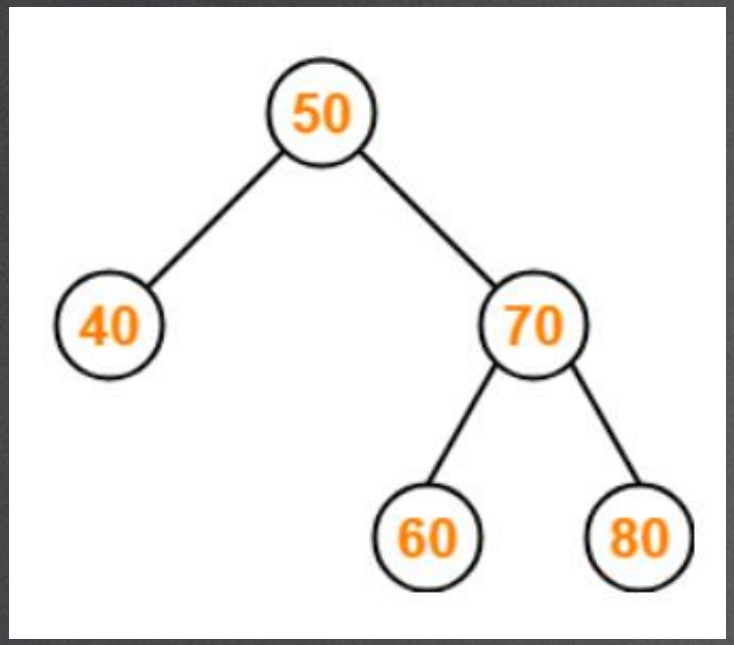




Binary Search Tree Deletion

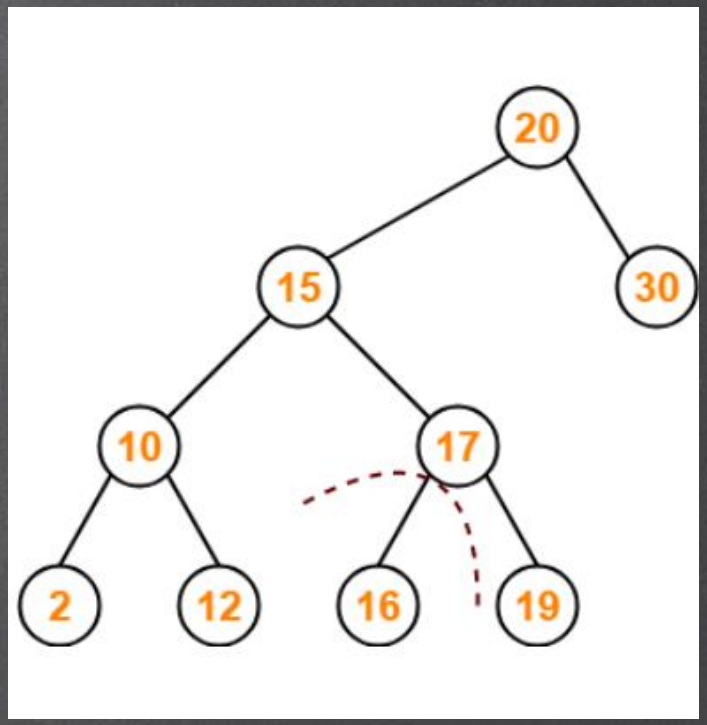
- Delete 30:

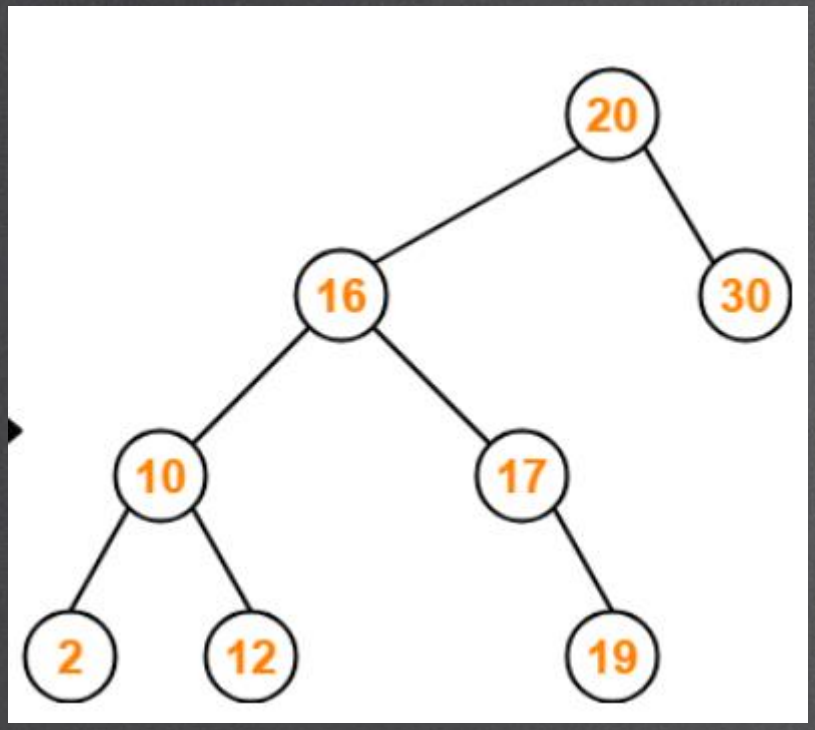




Binary Search Tree Deletion

- Delete 15:
- 2 10 12 15 16 17 19 20 30





Binary Search Tree Deletion

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.

Binary Search Tree Deletion

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.

Binary Search Tree Searching

- In order to searching a node in binary search tree we have to follows the step given below:
- Step 1: First we have to check weather binary search tree is empty or not. If binary search tree is empty then search is unsuccessful.
- Step 2: If binary search tree is not empty then we compare the value of a node to be searched with root node of binary search tree. If both values are equal then search is successful otherwise we have two possibilities.
- (A) If value of the node to be searched is less than the value of root node then we have to search node in left sub tree of root node.
- (B) If value of the node to be searched is greater than the value of root node then we have to search node in right sub tree of root node.
- Step 2 is repeated recursively until node to be searched is found or all the nodes in a binary search tree are compared with the node to be searched.

Binary Search Tree Searching

Algorithm to Search Node in Binary Search Tree:-

Step 1: If ROOT = None then

return

Step 2: If X=ROOT.INFO then

Write "Search is Successful"

Return 1

Step 3: If X < ROOT.INFO then

Call SEARCH (ROOT.LPTR, X)

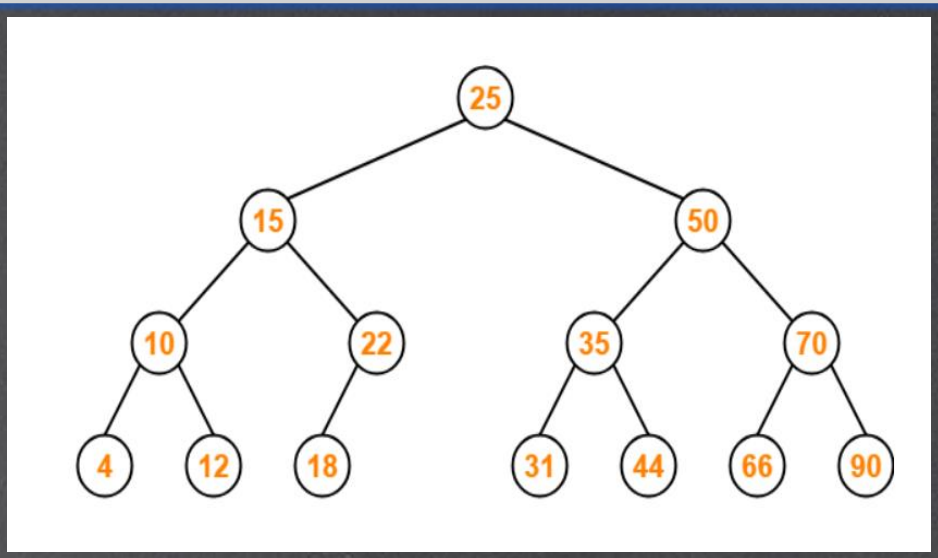
Else

Call SEARCH (ROOT.RPTR, X)

Binary Search Tree Searching

```
# Search the node
   def findval(self,X):
       if X < self.info:
           if self.info is None:
               return str(X)+" Not Found"
           return self.lptr.findval(X)
       elif X > self.info:
           if self.rptr is None:
               return str(X)+" Not Found"
           return self.rptr.findval(X)
       else:
           print(str(self.info) + ' is found')
```

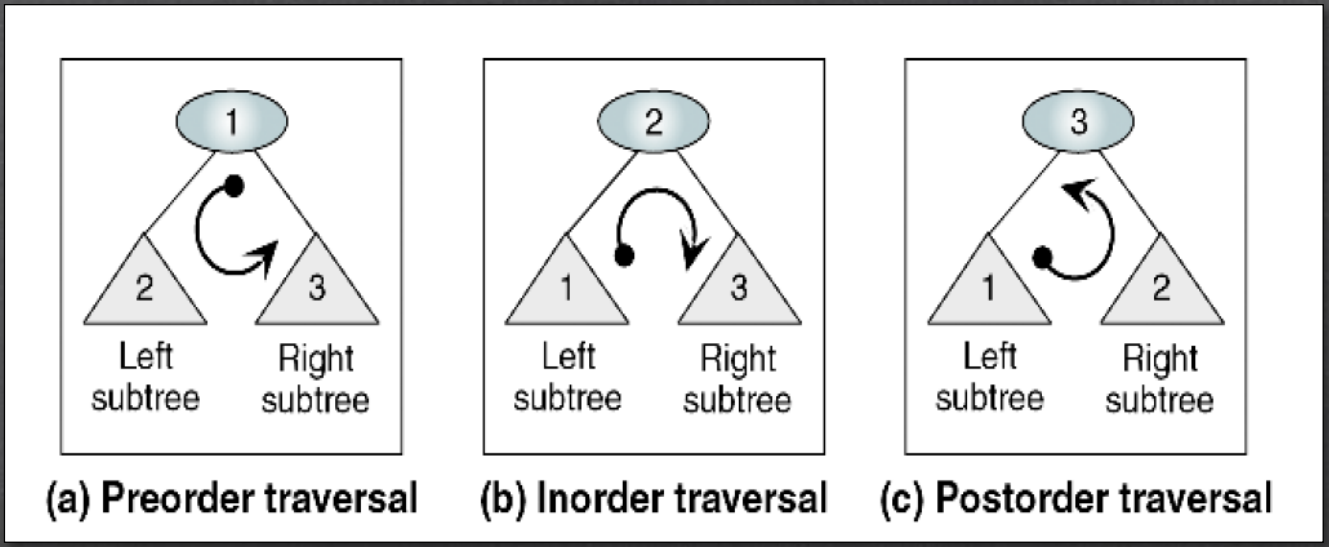
Binary Search Tree Searching



- We start our search from the root node 25.
- As 45 > 25, so we search in 25's right subtree.
- As 45 < 50, so we search in 50's left subtree.
- As 45 > 35, so we search in 35's right subtree.
- As 45 > 44, so we search in 44's right subtree but 44 has no subtrees.
- So, we conclude that 45 is not present in the above BST.

Binary Search Tree Traversal

- Traversal is the method of processing each and every node in the Binary Search Tree exactly once in a systematic manner.
- There are three different types of tree traversal.
- Preorder Traversal (Root-L-R)
- In order Traversal (L-Root-R)
- Post order Traversal(L-R-Root)

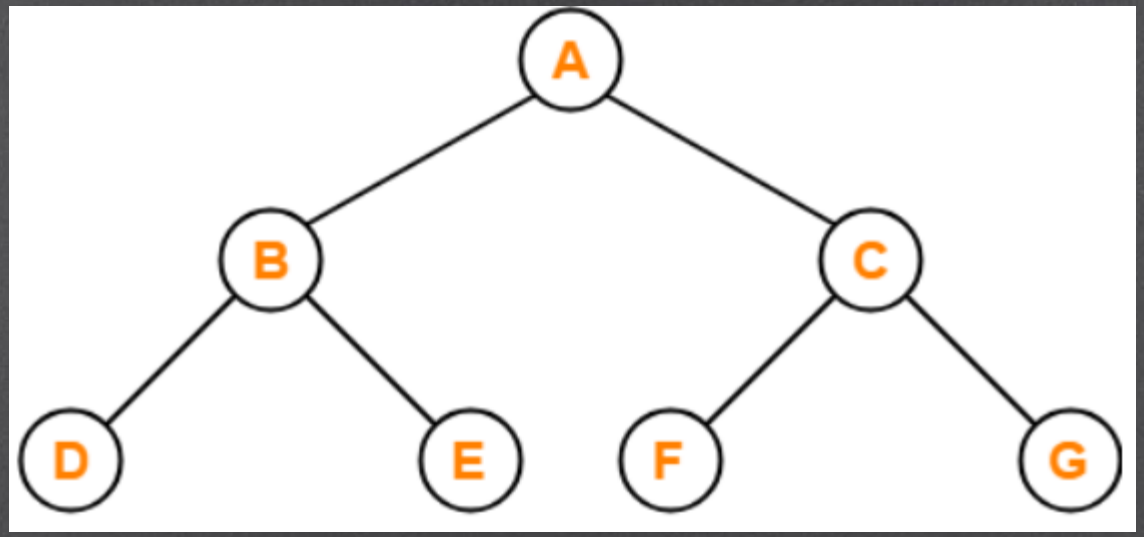


Binary Search Tree Traversal

Preorder Traversal:- (Root → Left → Right)

Steps for Preorder Traversal:

- Process the root node first.
- Traverse the left sub tree of root in preorder.
- Traverse the right sub tree of root in preorder.

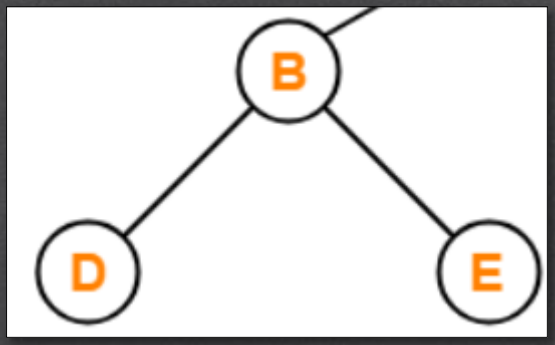


Binary Search Tree Traversal

Preorder Traversal:- (Root → Left → Right)

- Process the root node first. (A)

- Traverse the left sub tree of root in preorder.

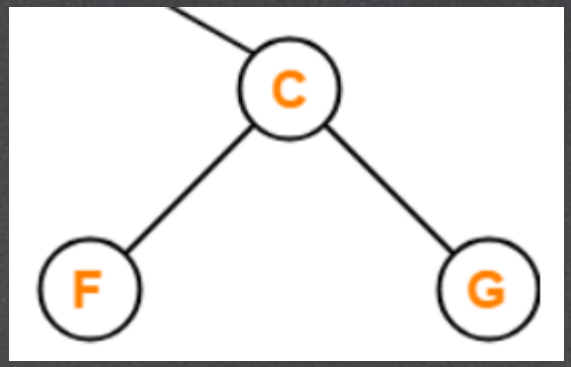


- B-D-E

Binary Search Tree Traversal

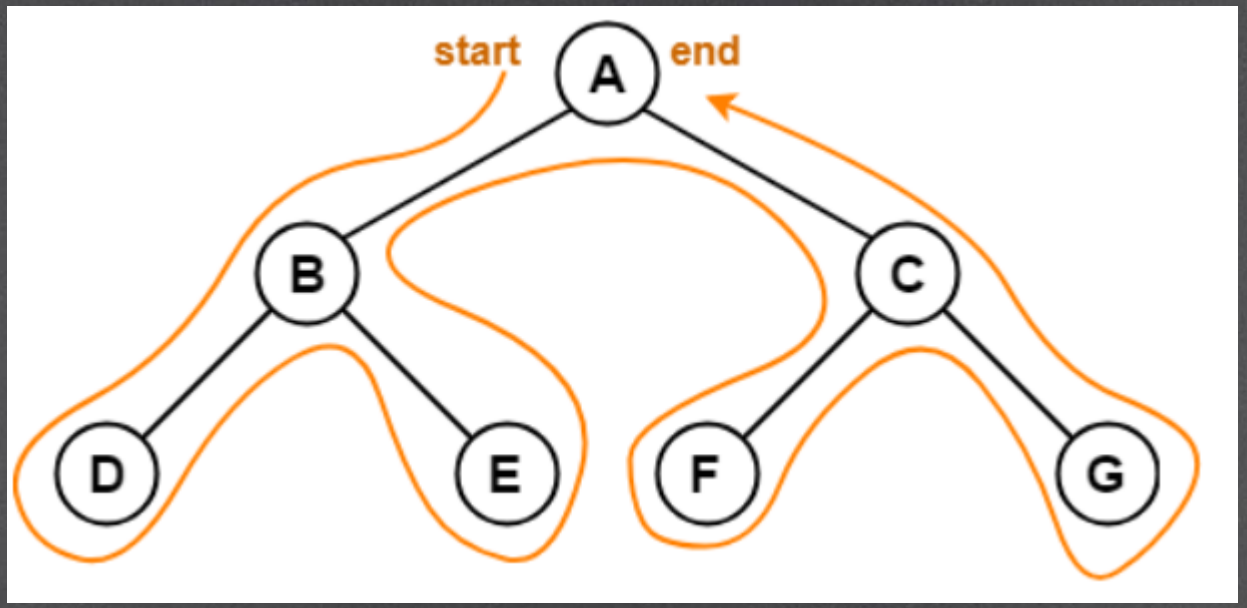
Preorder Traversal:- (Root → Left → Right)

- Traverse the right sub tree of root in preorder.



- C-F-G

Binary Search Tree Traversal



Preorder Traversal : A , B , D , E , C , F , G

Binary Search Tree Traversal

Preorder Traversal Algorithm:-

Step 1: If ROOT = None then
    return
Step 2: Write ROOT.INFO
Step 3: If ROOT.LPTR ≠ NULL then
    Call PREORDER (ROOT.LPTR)
Step 3: If ROOT.RPTR ≠ NULL then
    Call PREORDER (ROOT.RPTR)

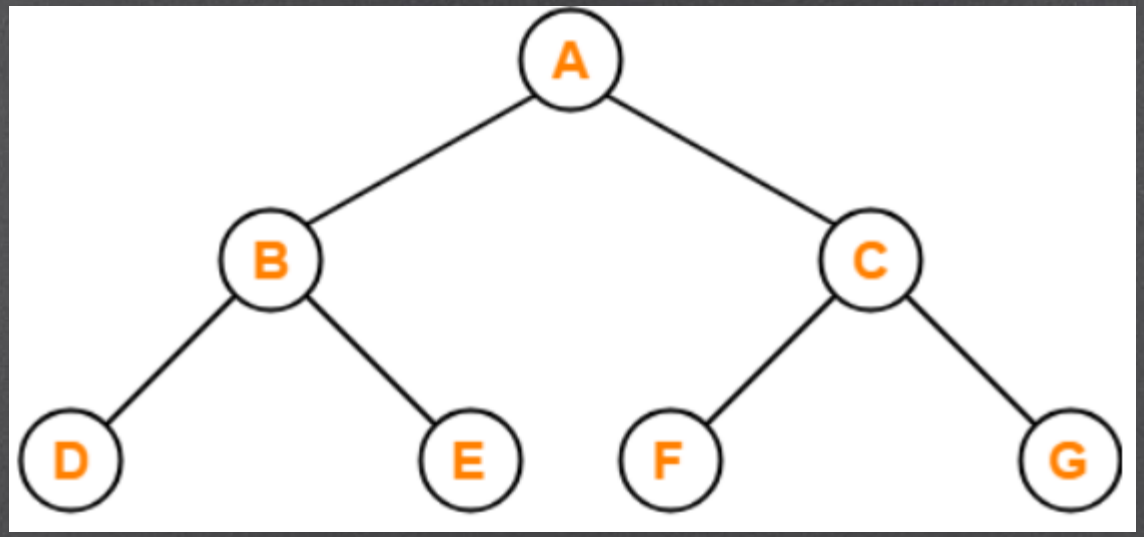
```
# preorder traversal of tree
  def preorder(self,root):
      if(root==None):
          return
      print(root.info)
      if(root.lptr!=None):
          self.preorder(root.lptr)
      if(root.rptr!=None):
          self.preorder(root.rptr)
```

Binary Search Tree Traversal

Inorder Traversal:- (Left → Root → Right)

Steps for Inorder Traversal:

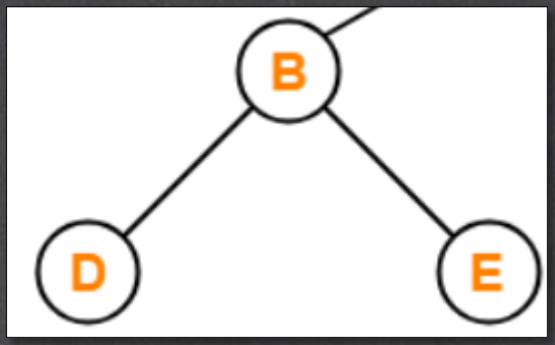
- Traverse the left sub tree in Inorder.
- Process the root node.
- Traverse the right sub tree in Inorder.



Binary Search Tree Traversal

Inorder Traversal:- (Left → Root → Right)

- Traverse the left sub tree of root in Inorder.



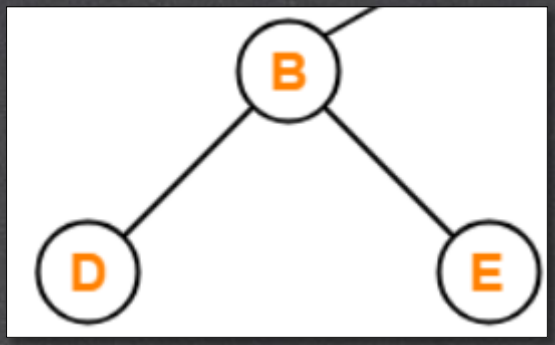
- D-B-E
- Process the root node.

- (A)

Binary Search Tree Traversal

Inorder Traversal:- (Left → Root → Right)

- Traverse the left sub tree of root in Inorder.



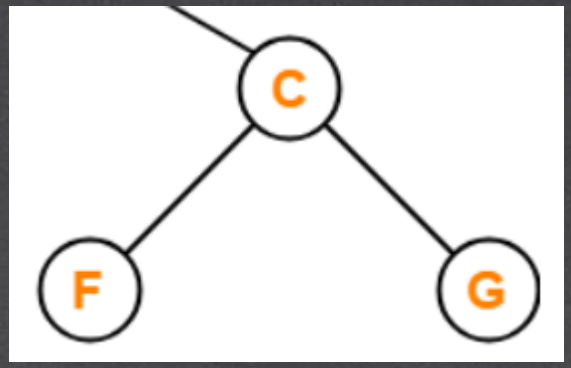
- D-B-E
- Process the root node.

- (A)

Binary Search Tree Traversal

Inorder Traversal:- (Left → Root → Right)

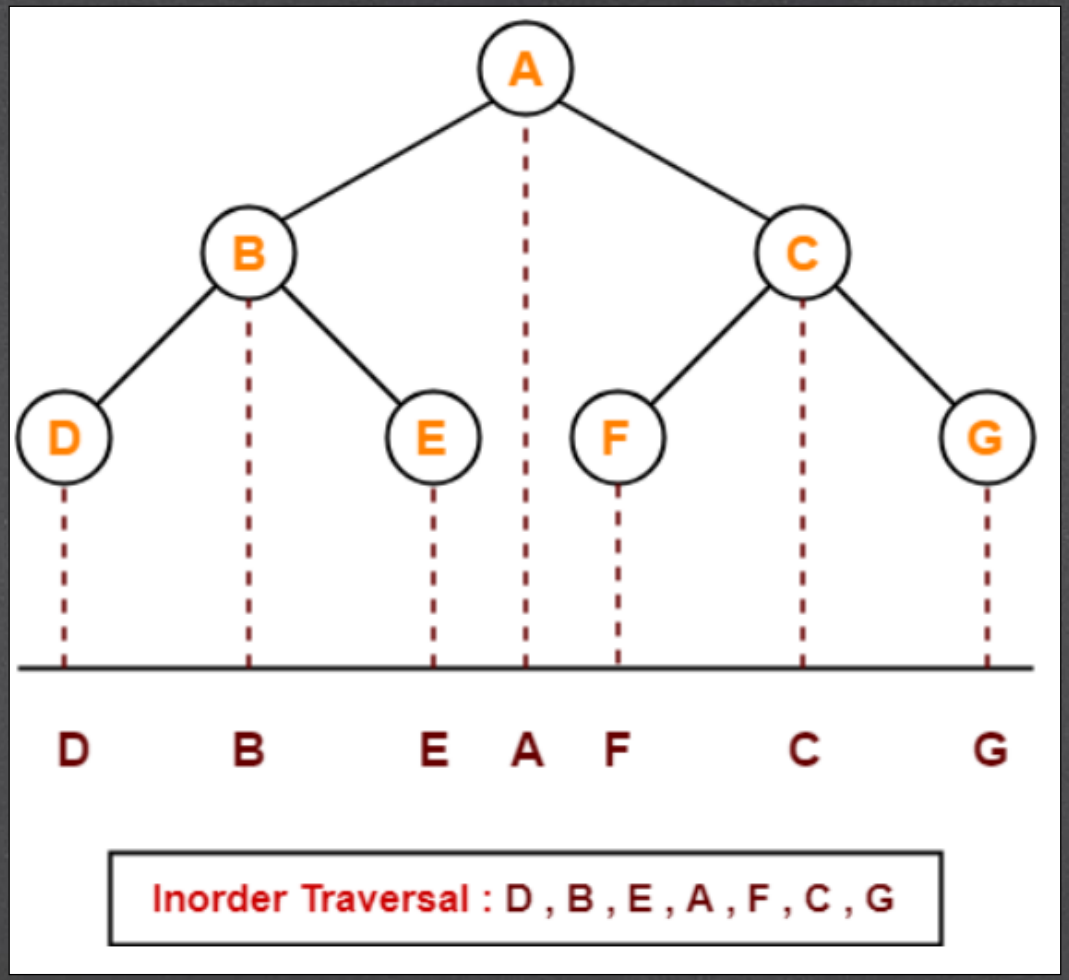
- Traverse the right sub tree of root in Inorder.



- F-C-G

Binary Search Tree Traversal

Inorder Traversal:- (Left → Root → Right)



Binary Search Tree Traversal

Inorder Traversal Algorithm:-

Step 1: If ROOT = None then
return

Step 2: If ROOT.LPTR ≠ NULL then
Call INORDER (ROOT.LPTR)

Step 3: Write ROOT.INFO

Step 4: If ROOT.RPTR ≠ NULL then
Call INORDER (ROOT.RPTR)

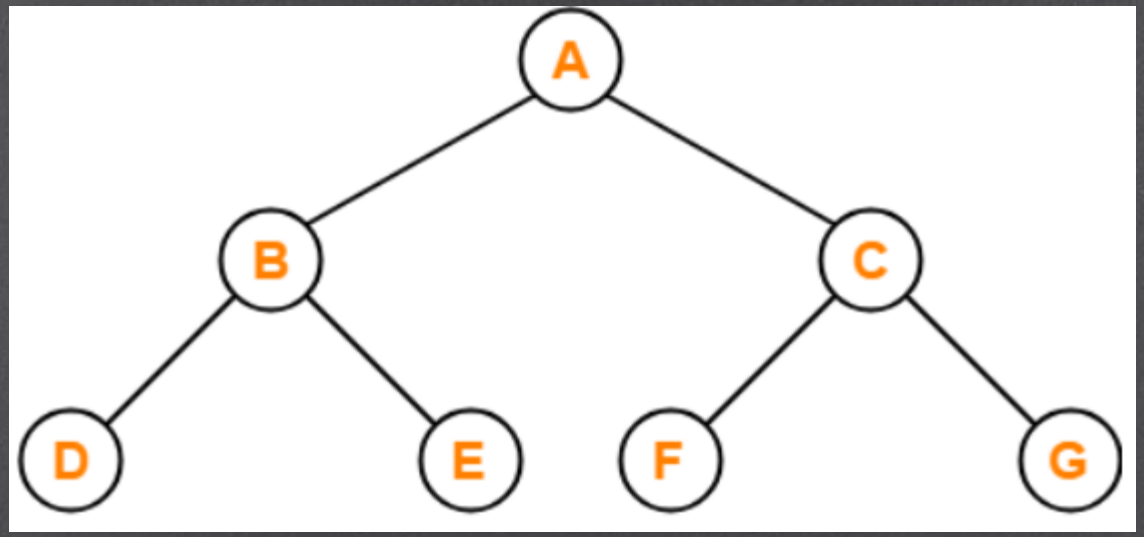
```
# inorder traversal of tree
    def inorder(self,root):
        if(root==None):
            return
        if(root.lptr!=None):
            self.inorder(root.lptr)
        print(root.info)
        if(root.rptr!=None):
            self.inorder(root.rptr)
```

Binary Search Tree Traversal

Postorder Traversal:- (Left → Right → Root)

Steps for Postorder Traversal:

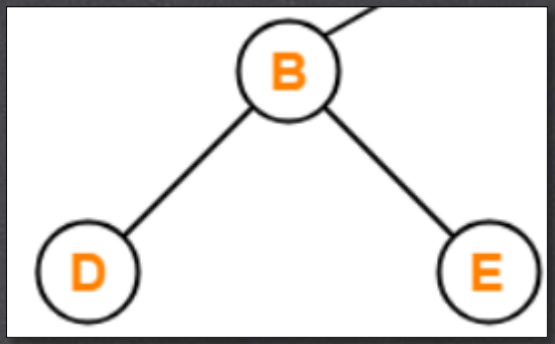
- Traverse the left sub tree in Post order.
- Traverse the right sub tree in Post order.
- Process the root node.



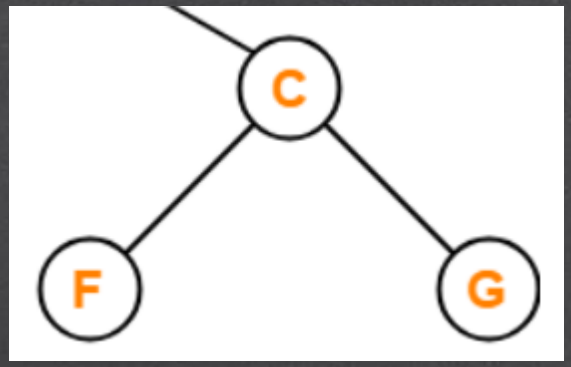
Binary Search Tree Traversal

Postorder Traversal:- (Left → Right → Root)

- Traverse the left sub tree in Post order.



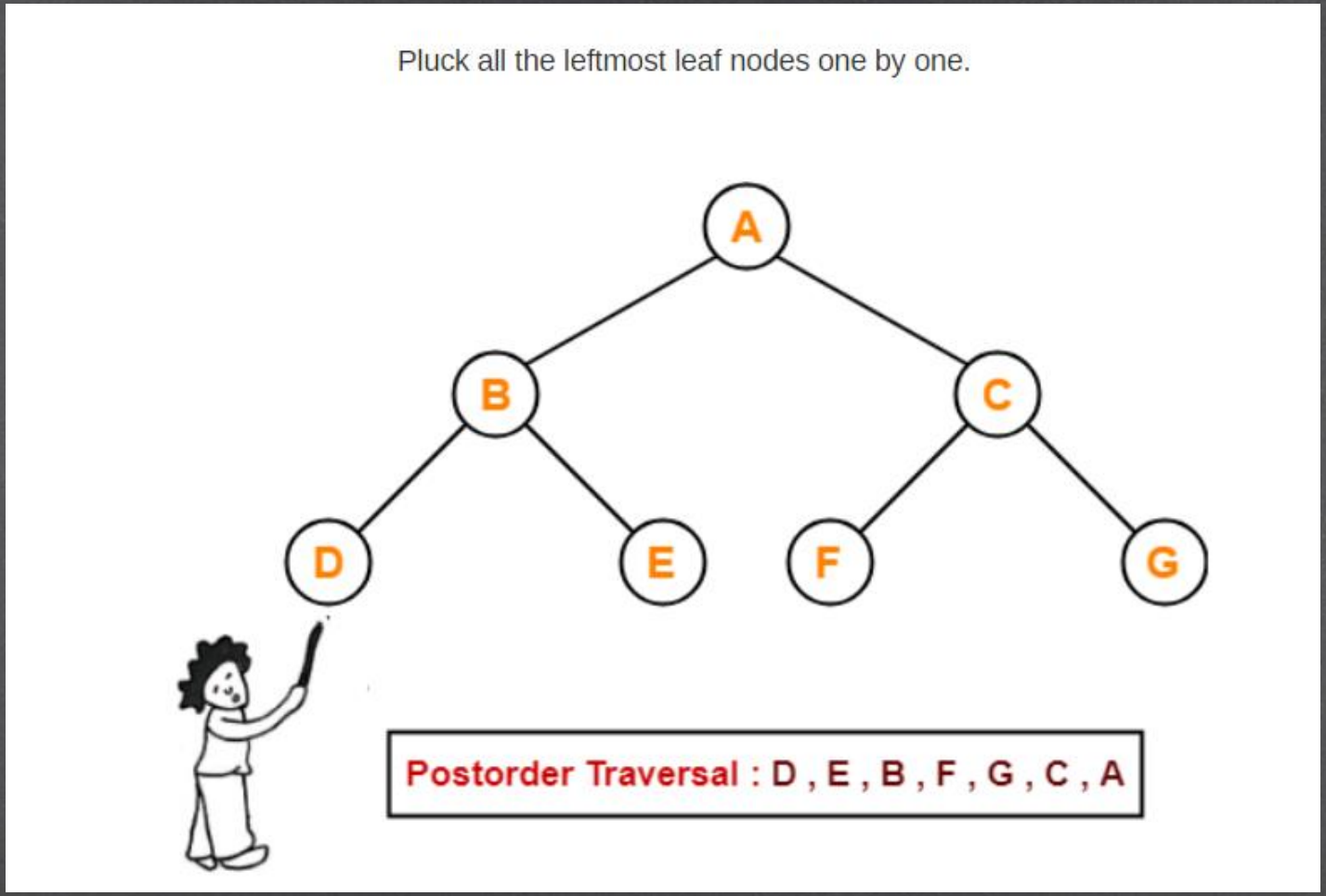
- D-E-B
- Traverse the right sub tree in Post order.



- F-G-C

Binary Search Tree Traversal

Postorder Traversal:- (Left → Right → Root)



Binary Search Tree Traversal

Postorder Traversal Algorithm:-

Step 1: If ROOT = None then
        return
Step 2: If ROOT.LPTR ≠ NULL then
        Call POSTORDER (ROOT.LPTR)
Step 3: If ROOT.RPTR ≠ NULL then
        Call POSTORDER (ROOT.RPTR)
Step 4: Write ROOT.INFO

```
# postorder traversal of tree
    def postorder(self,root):
        if(root==None):
            return
        if(root.lptr!=None):
            self.postorder(root.lptr)
        if(root.rptr!=None):
            self.postorder(root.rptr)
        print(root.info)
```

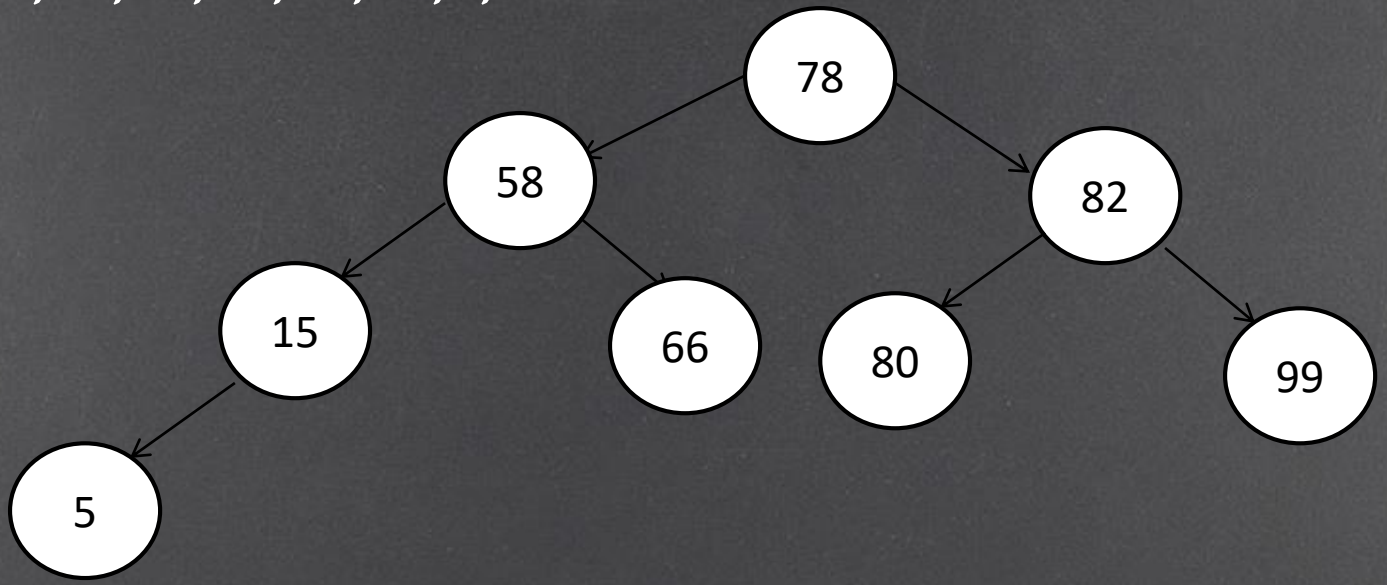
Binary Search Tree Traversal

78,58,82,15,66,80,99,5

7,4,10,9,12,2,1

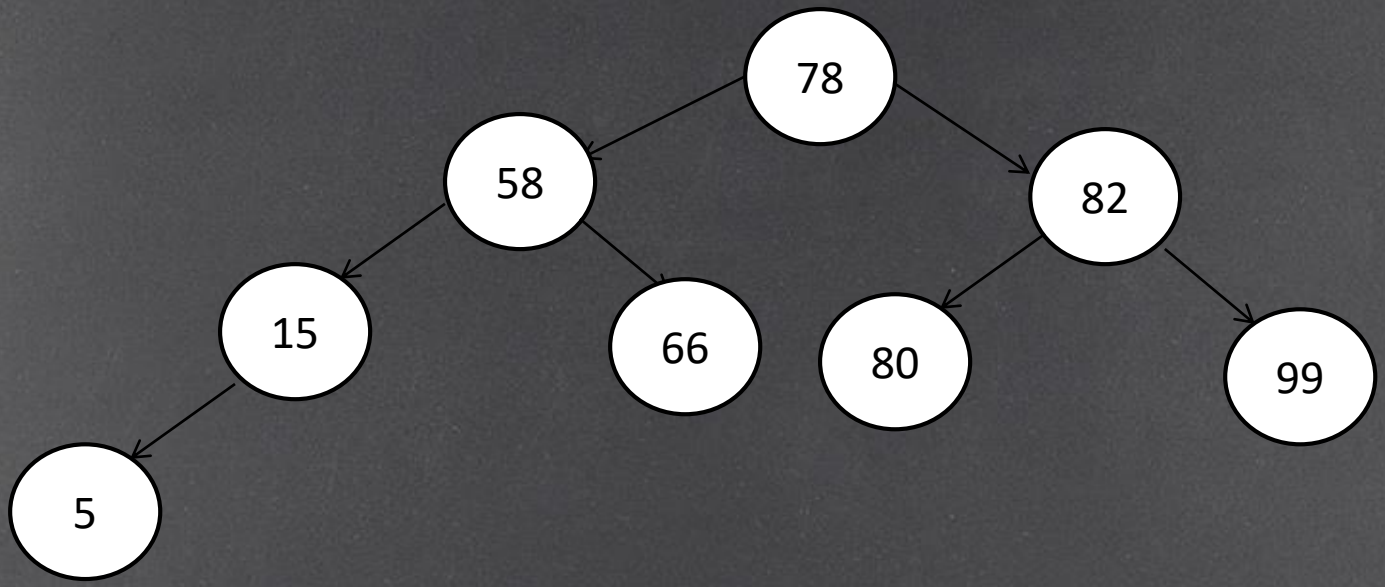
50,55,35,15,52,65,33,47,75,72,27,56

34,58,12,25,45,65,87,30,5,38



Binary Search Tree Traversal

78,58,82,15,66,80,99,5



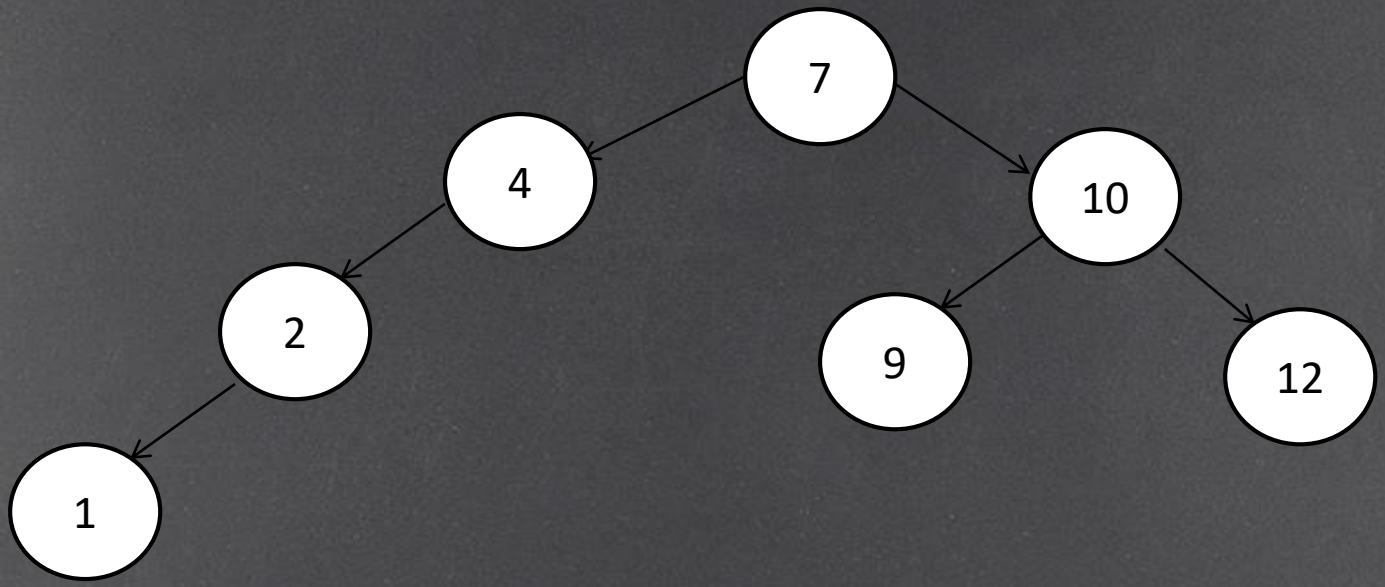
Preorder (Root-L-R):-78 58 15 5 66 82 80 99

Inorder (L-Root-R):- 5 15 58 66 78 80 82 99

Postorder (L-R-Root):- 5 15 66 58 80 99 82 78

Binary Search Tree Traversal

7,4,10,9,12,2,1



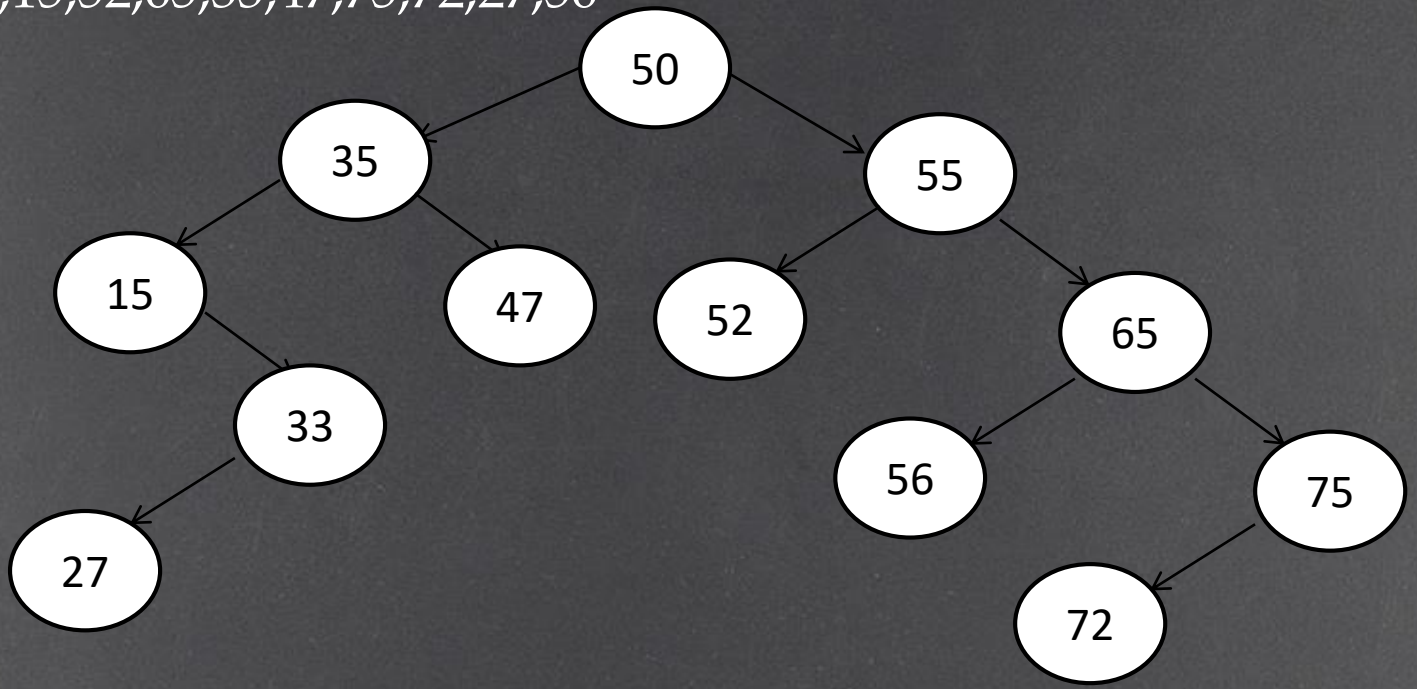
Preorder (Root-L-R):-7 4 2 1 10 9 12

Inorder (L-Root-R):- 1 2 4 7 9 10 12

Postorder (L-R-Root):- 1 2 4 9 12 10 7

Binary Search Tree Traversal

50,55,35,15,52,65,33,47,75,72,27,56



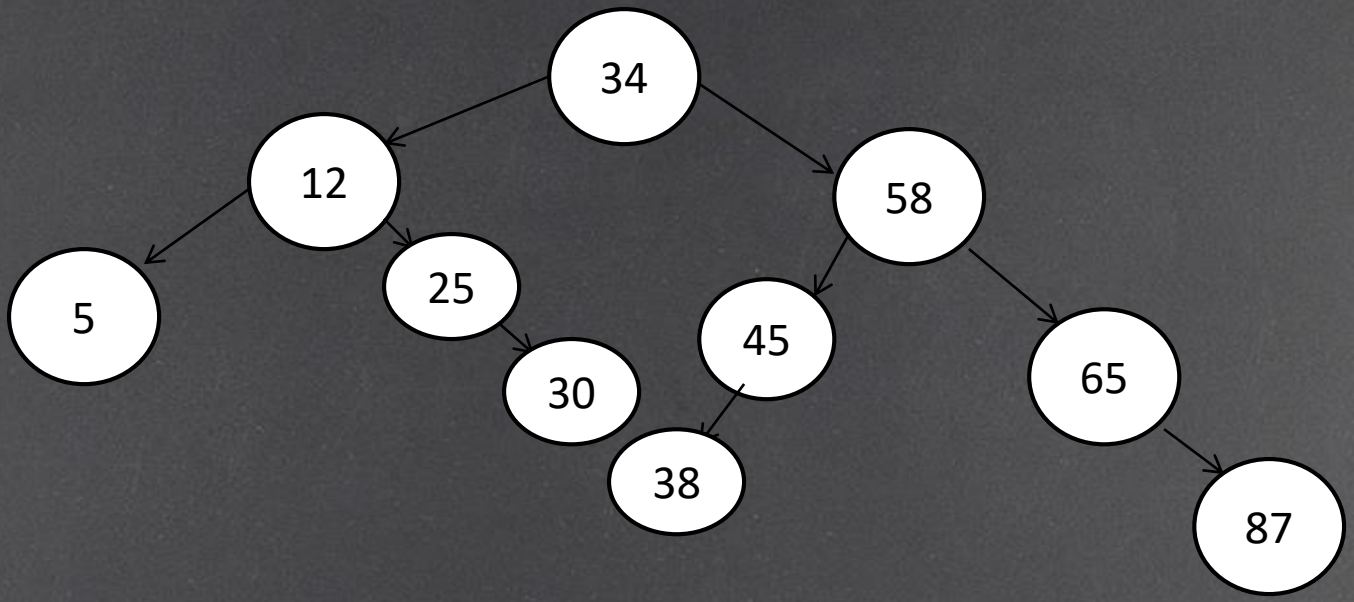
Preorder (Root-L-R):-50 35 15 33 27 47 55 52 65 56 75 72

Inorder (L-Root-R):- 15 27 33 35 47 50 52 55 56 65 72 75

Postorder (L-R-Root):-27 33 15 47 35 52 56 72 75 65 55 50

Binary Search Tree Traversal

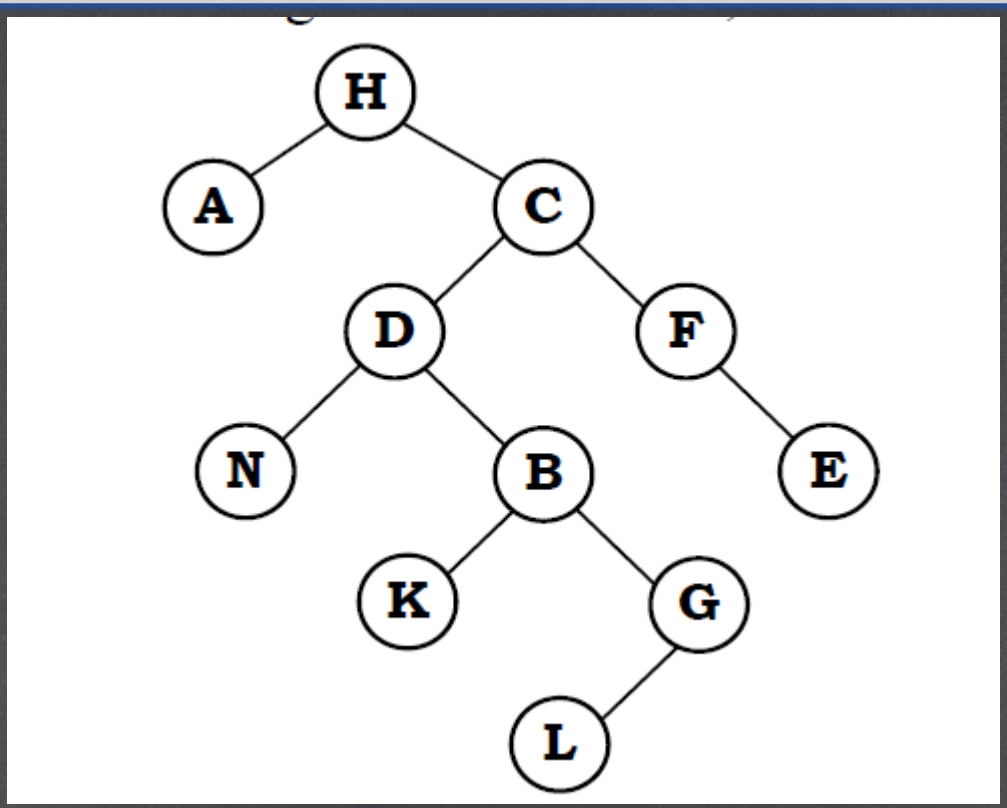
34,58,12,25,45,65,87,30,5,38



Preorder (Root-L-R):-34 12 5 25 30 58 45 38 65 87

Inorder (L-Root-R):- 5 12 25 30 34 38 45 58 65 87

Postorder (L-R-Root):- 5 30 25 12 38 45 87 65 58 34



- In order: AHNDKBLGCFE
- Post Order: ANKLGBDEFCH

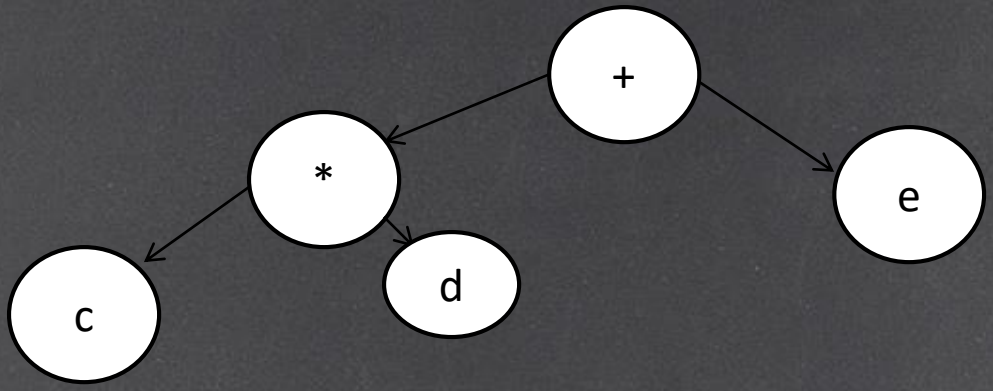
Application of Binary Tree

- It is used in Directory structure of a file store
- It is used in Structure/ manipulation of an arithmetic expressions
- It is used in almost every 3D video game to determine what objects need to be rendered.
- It is widely used in compiler for syntax analysis.
- It is used in implementation of Hash Tree.
- It is used in artificial intelligence.
- It is used to represent more complex data storage having hierarchical relationship among them.
- It is used in almost every high-bandwidth router for storing router-tables.
- It is used in compression algorithms, such as those used by the .jpeg and .mp3 file formats.

Application of Binary Search Tree

- It is used in many search applications where data is constantly entering/leaving, such as the map and set objects in many languages' libraries.
- It is used in Storing a set of names, and being able to lookup based on a prefix of the name. (Used in internet routers.)
- It is used in Storing a path in a graph, and being able to reverse any subsection of the path in O(log n) time. (Useful in travelling salesman problems).
- It is used in Finding square root of given number
- It is used for allows you to do range searches efficiently.

B.T representation of A.E.

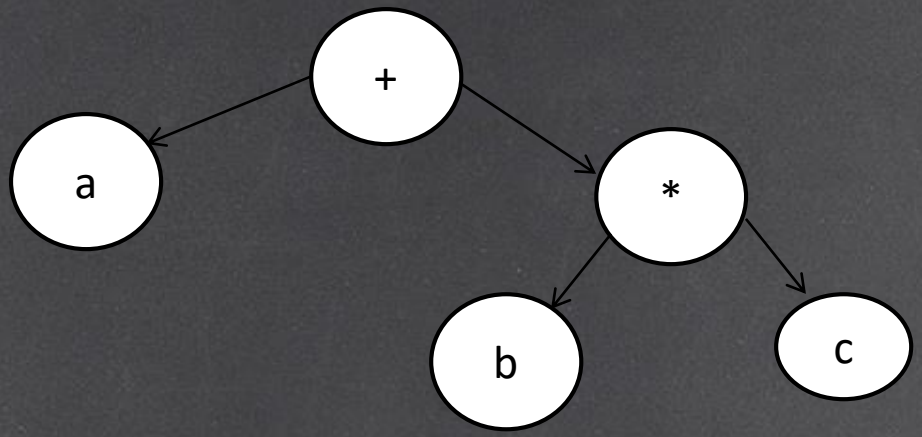


Preorder (Root-L-R):- + * c d e

Inorder (L-Root-R):- c * d + e

Postorder (L-R-Root):- c d * e +

B.T representation of A.E.



Preorder (Root-L-R):- + a * b c

Inorder (L-Root-R):- a + b* c

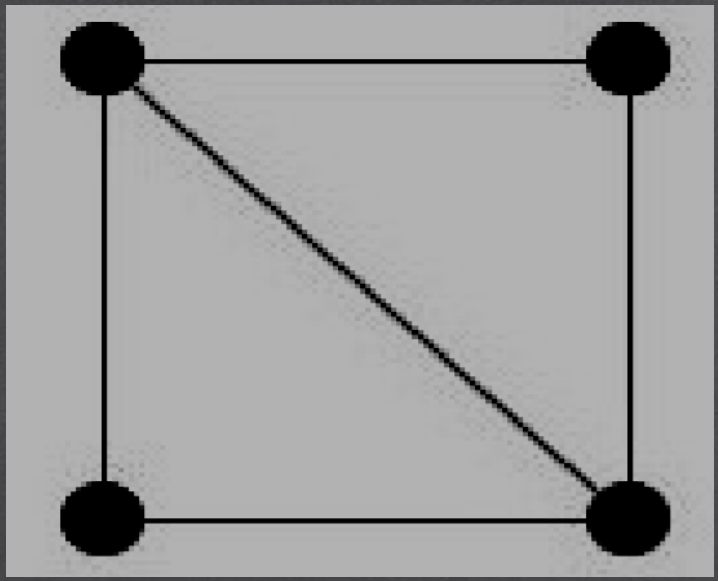
Postorder (L-R-Root):- a b c * +

Binary Search Tree

Time Complexity			
	Array	**Linked List**	**BST**
Search	O(n)	O(n)	O(logn)
Insert	O(1)	O(1)	O(logn)
Remove	O(n)	O(n)	O(logn)

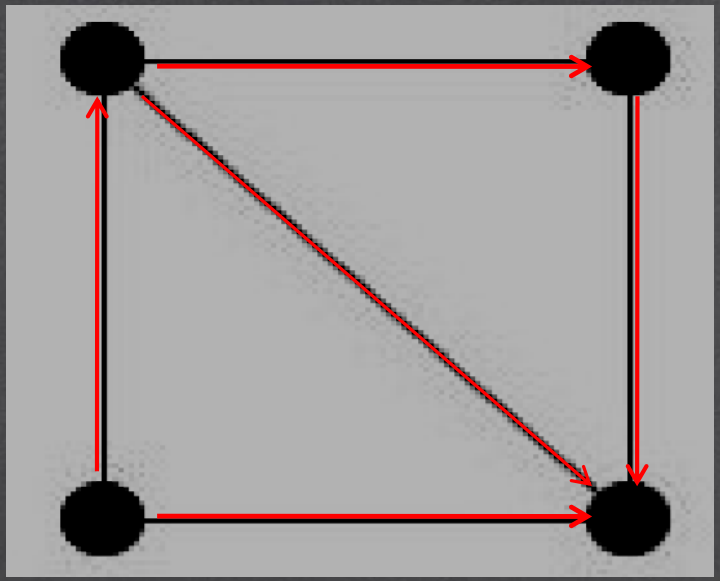
Graph

- It is a set of items connected by edges.
- Each item is called a vertex or node. Trees are just like a special kinds of graphs. Graphs are usually represented by G = (V, E), where V is the set vertices and E is the set of Edges.



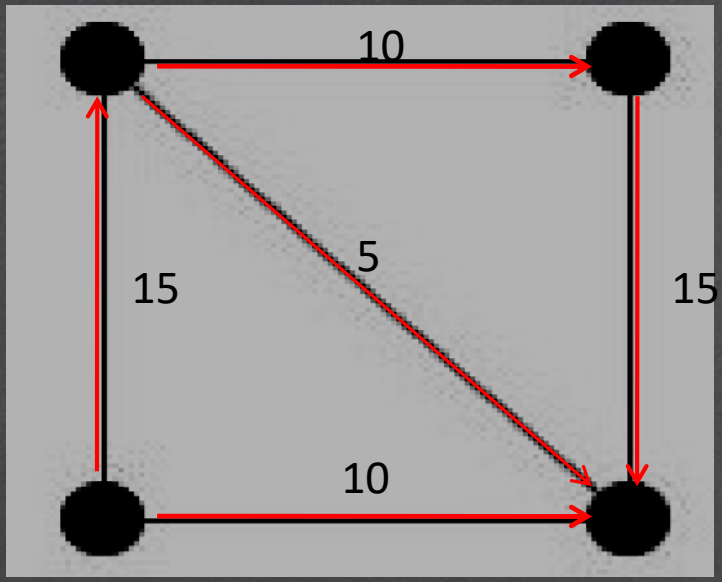
Directed Graph

- It is a graph (set of vertices connected by edges) where edges have a direction associated with them.



Weighted Graph

- It is a graph (set of vertices connected by edges) where each edge of a graph has an associated numerical values is called weight.
- Edge weights are non negative integers.
- They may be directed or undirected.



THANK YOU