Trees

IT is a type of data structure. A data structure that shows relationship between parent and child is called a Tree.

Properties:

- Trees have Vertices & Edges. The circles are called vertices and the lines connecting two circles is called edges.
- There are no cycles in tree.
- If there are n vertices, there will be n-1 edges.

Binary Tree:

“Bi” means two. Each parent will have at either ‘0’ children or ‘2’ children. If a parent has 3 children, then it will not be a Binary Tree but ‘nary’ tree. A Binary Tree is a hierarchical data structure in which each node has at most two children generally referred as left child and right child.

Each node contains three components:

1. Pointer to left subtree
2. Pointer to right subtree
3. Data Element

Binary Tree: Common Terminologies

Root: Topmost node in a tree.

Parent: Every node (excluding a root) in a tree is connected by a directed edge from exactly one other node. This node is called a parent.

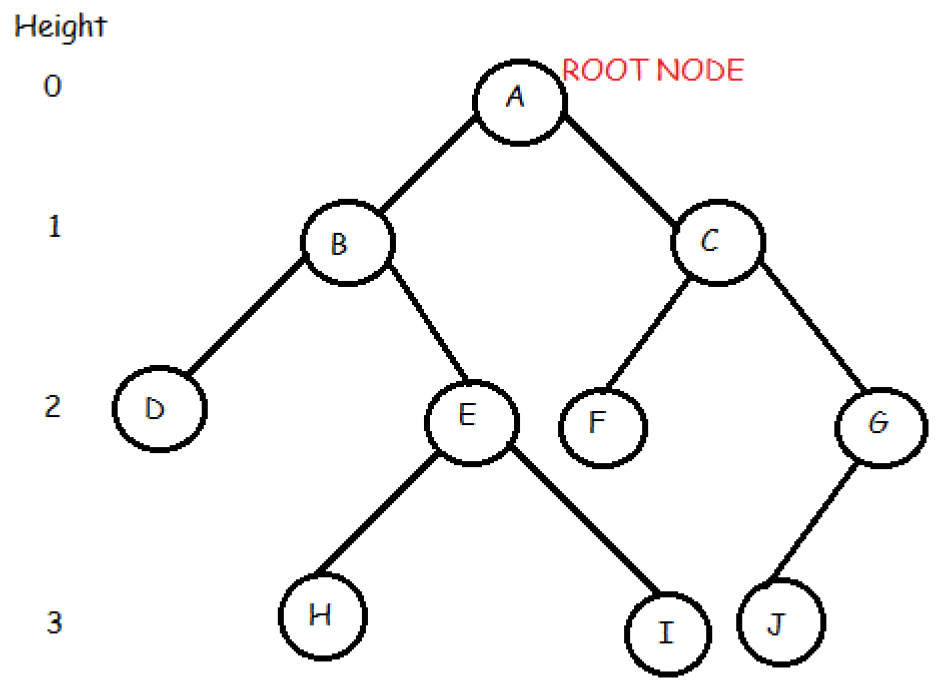
Child: A node directly connected to another node when moving away from the root.

Leaf/External Node: Node with no children.

Internal Node: Node with at least one child.

Depth of a Node: Number of edges from root to the node.

Height of a Node: Number of edges from the node to the deepest leaf. Height of the tree is the height of the root.



In the above Binary Tree, we see that root node is A. The tree has 10 nodes with 5 internal nodes, i.e., A, B, C, E, G and 5 external nodes, i.e., D, F, H, I, J. The height of the tree is 3. B is the parent of D and E while D and E are children of B.

Advantages of Trees

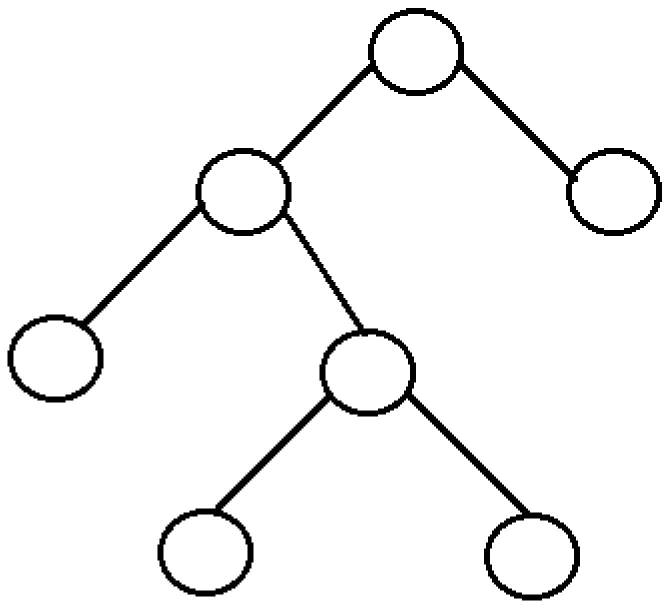
Trees are so useful and frequently used, because they have some very serious advantages:

- Trees reflect structural relationships in the data.
- Trees are used to represent hierarchies.
- Trees provide an efficient insertion and searching.
- Trees are very flexible data, allowing to move subtrees around with minimum effort.

Types of Binary Trees (Based on Structure)

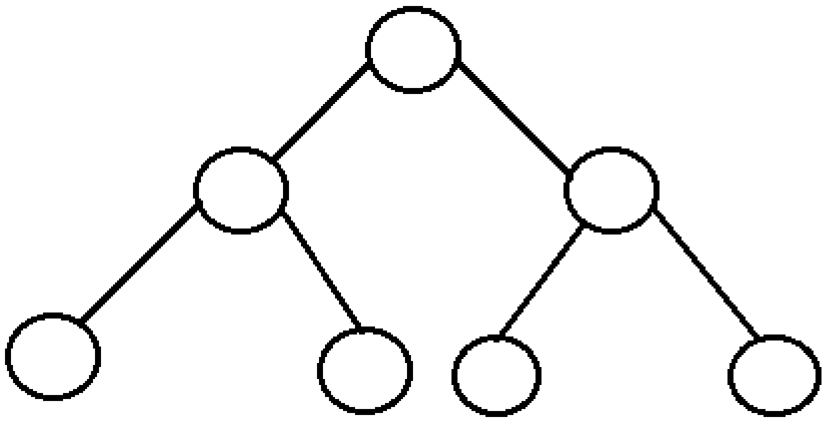
Rooted Binary Tree: It has a root node and every node has at most two children.

Full Binary Tree: It is a tree in which except leaf node, every node in the tree has either 0 or 2 children. Node which do not have any children are called Leaf Nodes.



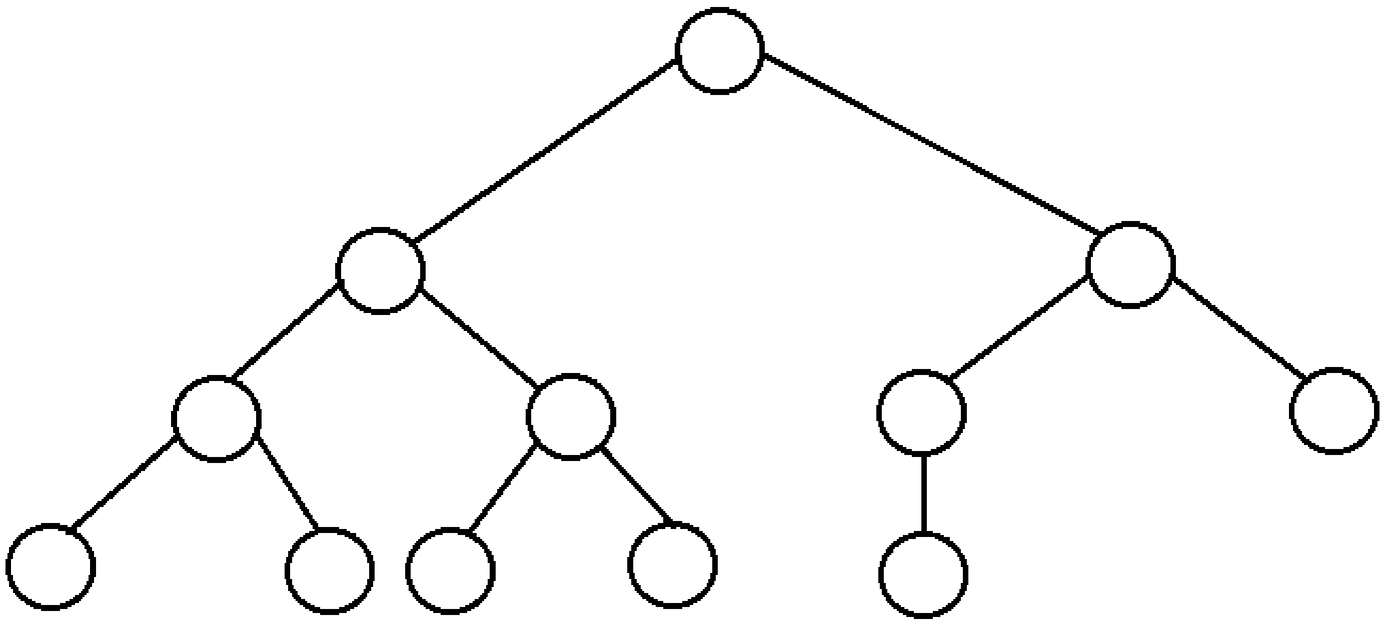
- The number of nodes, n, in a full Binary Tree is at least n = 2h – 1, and atmost n = $2^{h+1} - 1$, where h is the height of the tree.
- The number of leaf nodes l, in a full Binary Tree is number, L of internal nodes + 1, i.e., l = L+1.

Perfect Binary Tree: It is a Binary Tree in which all interior nodes have two children and all leaves have the same depth or same level.



- A perfect Binary Tree with l leaves has n = 2l-1 nodes.
- In perfect full Binary Tree, l = 2h and n = 2h+1 - 1 where, n is number of nodes, ‘h’ height of tree and l is number of leaf nodes

Complete Binary Tree: It is a Binary Tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

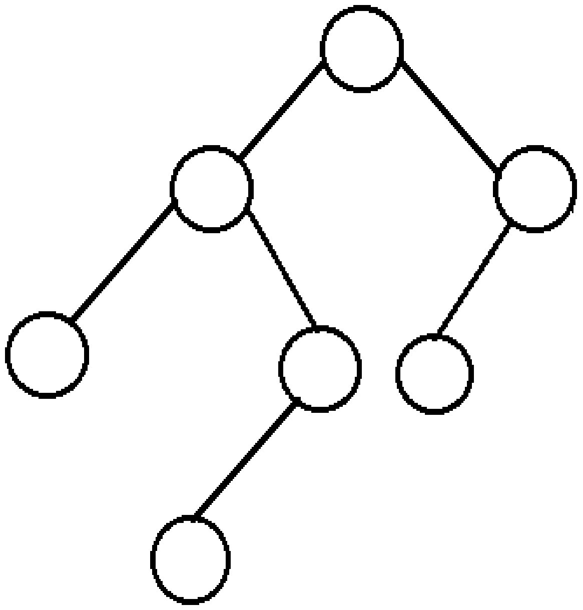


- The number of internal nodes in a complete Binary Tree of n nodes is floor(n/2).

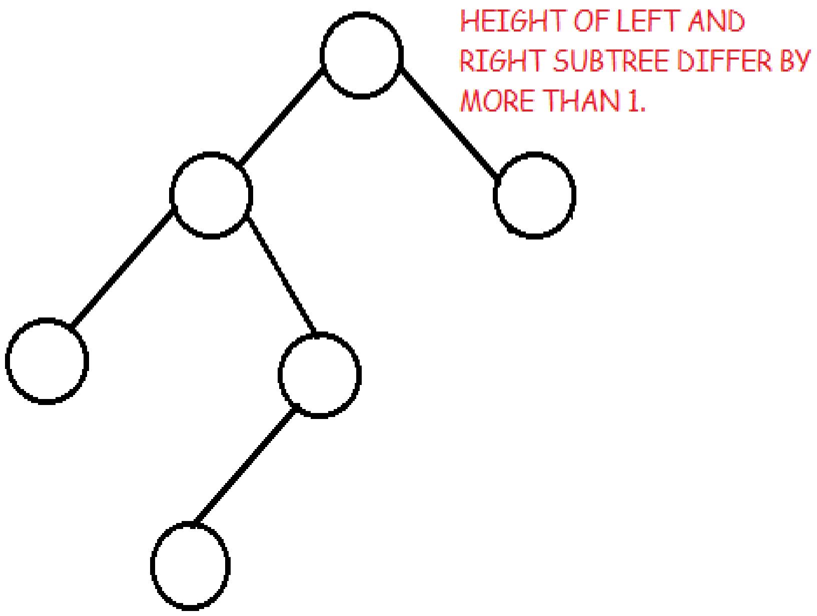
Balanced Binary Tree: A Binary Tree is height balanced if it satisfies the following constraints:

1. The left and right subtrees' heights differ by at most one, AND
2. The left subtree is balanced, AND
3. The right subtree is balanced

An empty tree is height balanced.



HEIGHT BALANCED BINARY TREE



NOT A HEIGHT BALANCED BINARY TREE

- The height of a balanced Binary Tree is O (log n) where n is number of nodes.

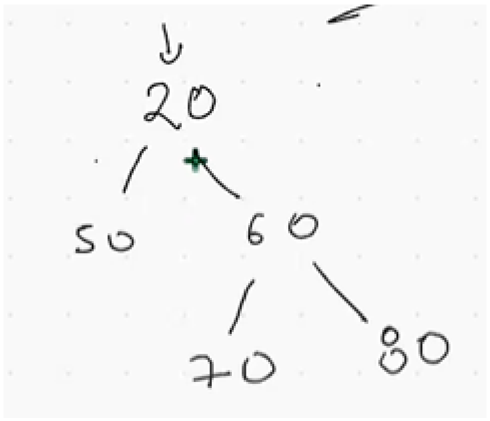
Traversals:

1) Preordered Traversal (Root L R)

2) In-Order Traversal (L Root R)

3) Post-Order Traversal (L R Root)

Pre-Ordered Traversal:



In a binary tree we will always have root, in this case root = 20. In a pre-ordered traversal, we print the root, then we will go recursively to the left and then towards right.

At root = 20, we will print the root and then we will go recursively to left and print 50. Then we will go to recursively to the left of 50 as it is empty we will go to right. Since right is also empty, we will go back to 20.

20, 50

We will go recursively towards left and print 70. Since there is nothing after 70, we will go towards right. As right has no element, we will go to the 60 and print 80 that is on it’s right.

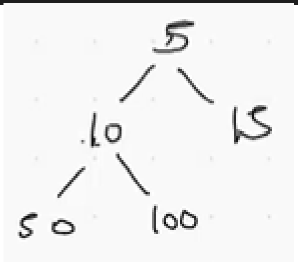
20, 50, 60, 70, 80.

We can write the code as:

```
def preorder(root):
        if root is none:
                return
        print(root.data)

        preorder(root.left)
        preorder(root.right)
```

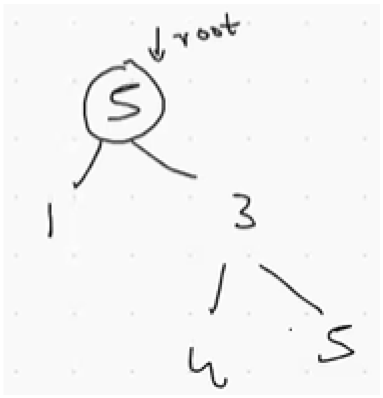
CODE:

```python
class Node:
    def __init__(self):
        self.data = data
        self.left = None
        self.right = None

if __name__ == "__main__":
    root = Node(5)
    root.left = Node(10)
    root.right = Node(15)

    root.left.right = Node(100)
    root.left.left = Node(50)
```



This is the tree that we have created with the above code.

Understanding using the below tree.



Root = 5, using stack, the initial value will be 5.

If root is not none, it will print the root value. The recursion starts and the code will be paused at line 4. It will call another recursion root.left so the root wil become 1. It will print (root.data) and will come to line 4 and get paused.

5, 1

At line 4, the code will go for root.left. there is no value on the left hence root is None, so the program will return to preorder(root.right). There is no value at right as well, so it will return again.

As there is nothing to print the program will go back to the root = 5. Now, the code will run for preorder (root.right). The value at the right of root = 5 is 3. The program will print 3.

5, 1, 3

Now, root = 3, and the root.left will be 4 and it will print 4.

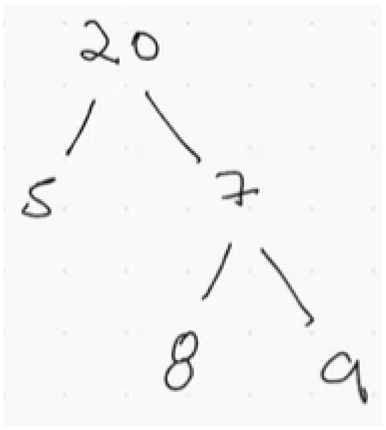
5, 1, 3, 4

Now, the root = 4. Since there is no value at left or right of the root, it will return to root = 3. The program has finished the left of the root, now the program will go to the right of the tree and will print 5.

5, 1, 3, 4, 5

Inorder Traversal:

```
def inorder(root):
    if root is None:
        return

    inorder(root.left)
    print(root.data)
    inorder(root.right)
```



Left—Root—Right order for inorder traversal.

At 20, first, we will go to left = 5. At 5, we will go to it's left. Sicne it is empty, we will return 5. We have done exploring the left, now, we will print the root 5.

5

Then to the right there is no element so it will return to 5. For 5, left, root and right is done so it will return to 20. For 20, we have explored the left part, so now, the program will check for root value = 20 and print it.

5, 20

After root = 20, the program will now explore the right of root = 20. On the right we have right = 7, so the root will shift to 7. Now, at root = 7, the left would be checked first. Left = 8 and now root = 8. At root = 8, there is no value on the left hence it will return to root= 8. The root will be print as per the left, root and right principle.

5, 20, 8

Once 8 is printed, the value will return to 7 and so will the root. Now, root = 7 and the program will print 7.

5, 20, 8, 7

The program will go to right, and hence the root will be shifted to 9. The program will check for left of 9. Since there are no elements on the left, the program will come back and print 9 as it is the root value.

5, 20, 8, 7, 9