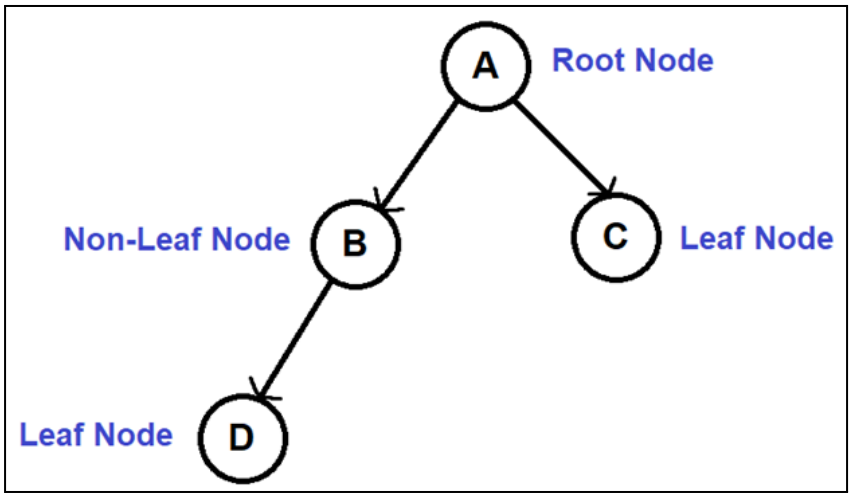
Tree

Tree is a non-linear data structure that allows us to organize, access, and update data efficiently by representing non-linear data in a form of hierarchy.

Terminologies of a Tree:

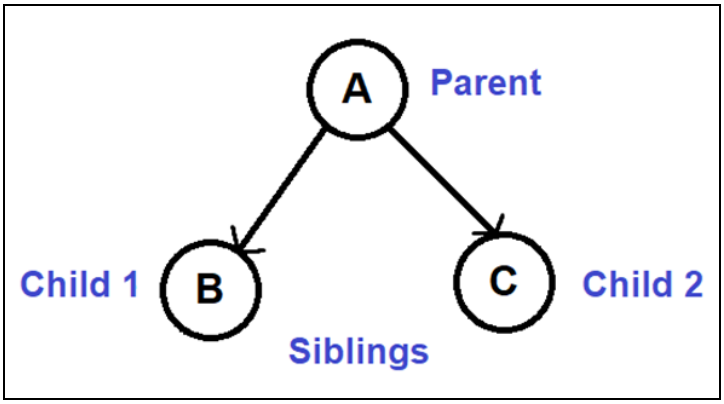
- Root/Leaf/Non-Leaf Node:

Each tree consists of one or more nodes that are connected. The root node is the topmost node (or the node without any parent). Below the root, there are one or more nodes. The nodes residing at the bottom (or those that do not lead to any other node) are called leaf nodes. If we have access to the root node, we have access to the entire tree.



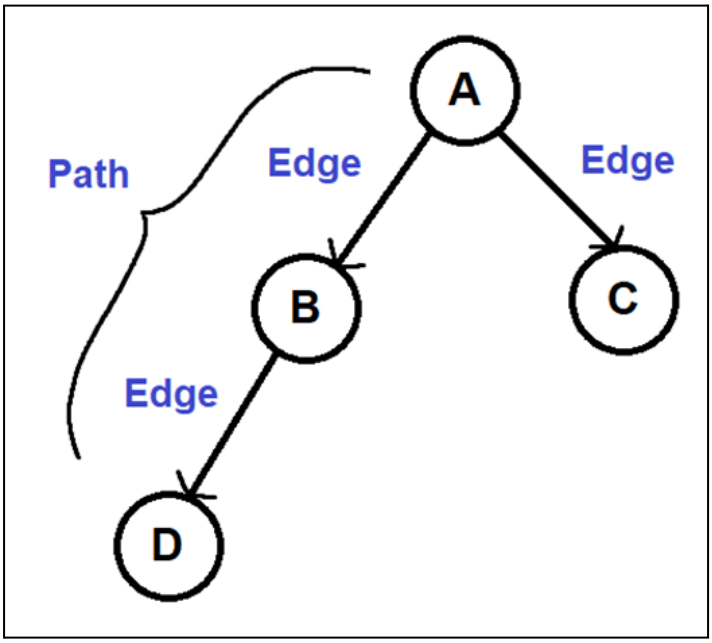
- Parent/Child/Siblings:

A node is a parent when it is an immediate predecessor of another node and a node is a child when it is an immediate successor of another node. Hence, node A is the parent of node B and C whereas node B and C are the children of node A. All the nodes having the same parent are known as siblings. Therefore, nodes B and C are siblings.



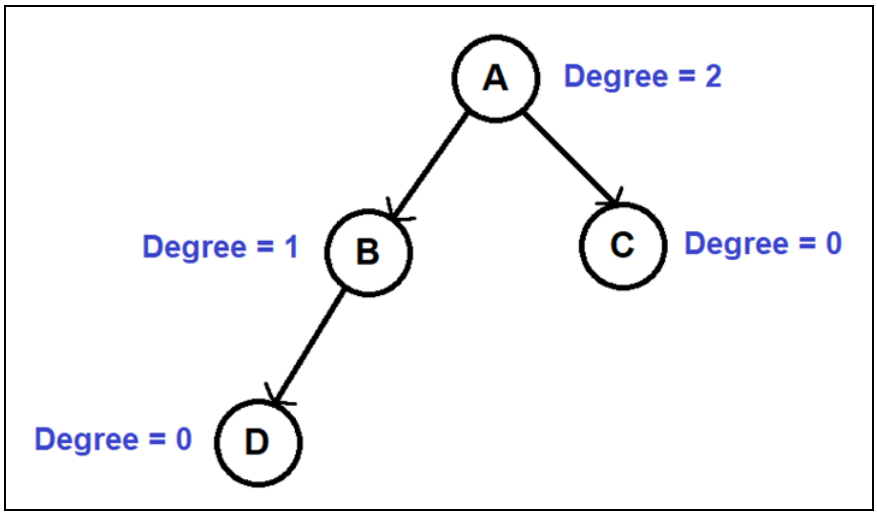
- Edge/Path:

The link between two nodes is called an edge. A-B, A-C, and B-D are edges. A path is known as the consecutive edges from the source node to the destination node. So, if we asked what is the path from node A to D? The answer would be A→B→D.



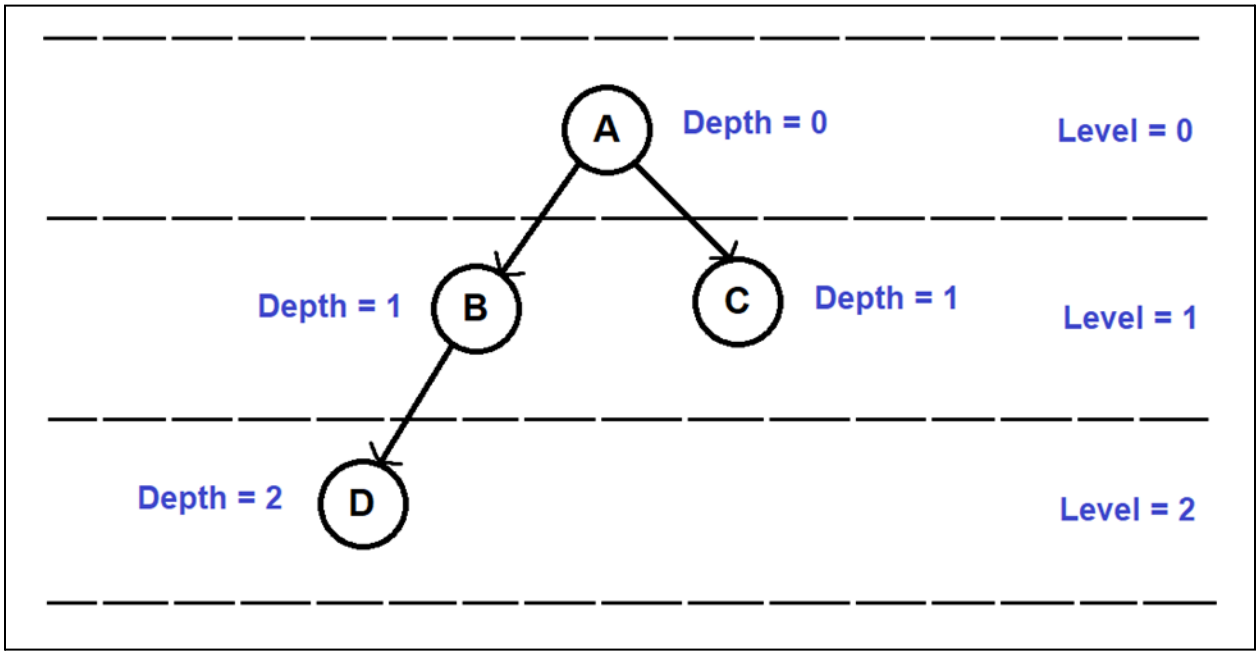
- Degree:

The number of children of a node is known as the degree. Node A has 2 children hence it has a degree of 2.



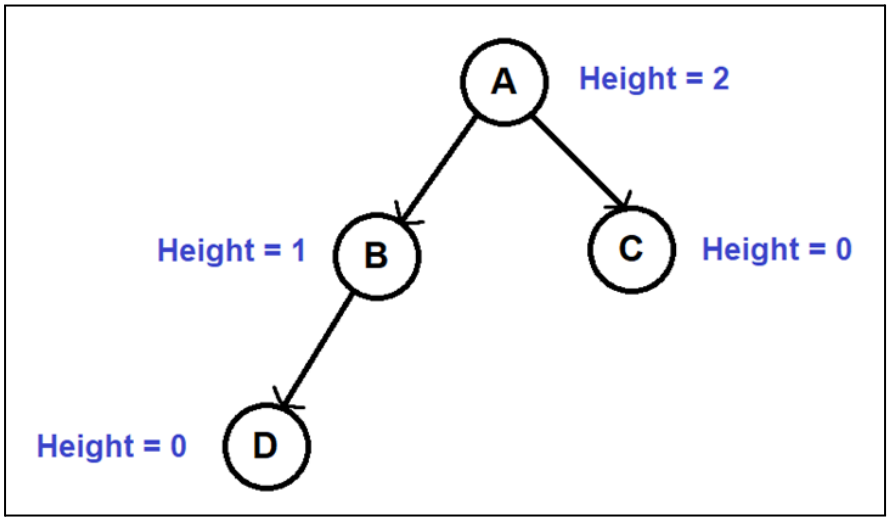
- Depth/Level:

The length of the path from a node to the root node is known as the depth. The depth of node D is 2 as the length of the path from node D to root node A is 2. Each hierarchy starting from the root is known as the level. For any node, level == depth.



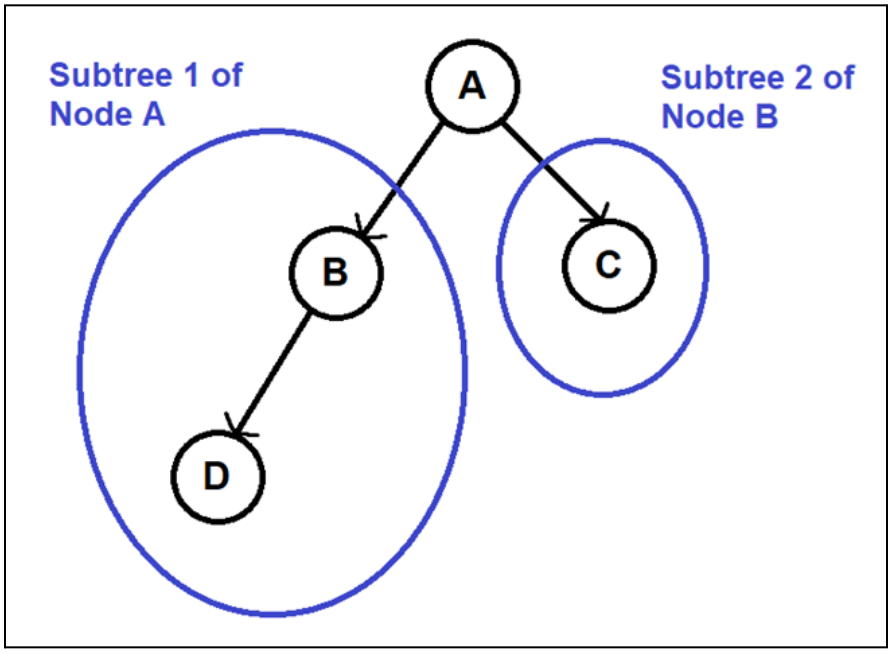
- Height:

The length of the longest path from a node to one of its leaf nodes is known as the height. From node A to the leaf nodes there are two paths: A→B→D and A→C. Of these four paths, A→B→D is the longest path. Hence, the height of Node A is 2. The height of a tree means the height of the root node.



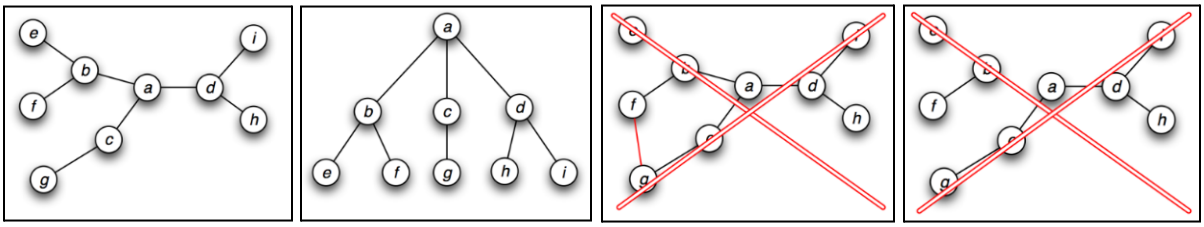
- Subtree:

A subtree is a tree that is a child of a node. Any tree can be further divided into subtrees with respect to a particular node.



Characteristics of a Tree:

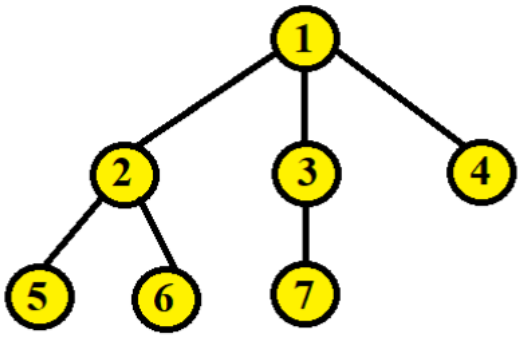
- A tree must be continuous and not disjoint, which means every single node must be traversable starting from the root node.
- A tree cannot have a cycle
- A tree having n nodes must have n-1 edges



Tree implementation using Linked List:

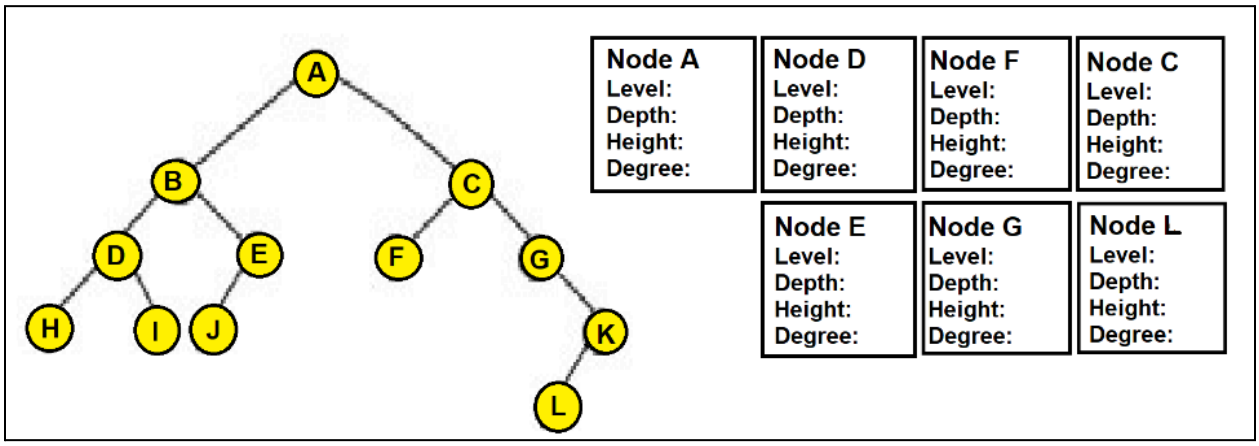
```
class Node:
        def __init__(self, elem):
                self.elem = elem
                self.children = [ ]

root = Node(1)
root.children += [Node(2)]
root.children += [Node(3)]
root.children += [Node(4)]
root.children[0].children += [Node(5)]
root.children[0].children += [Node(6)]
root.children[1].children += [Node(7)]
```



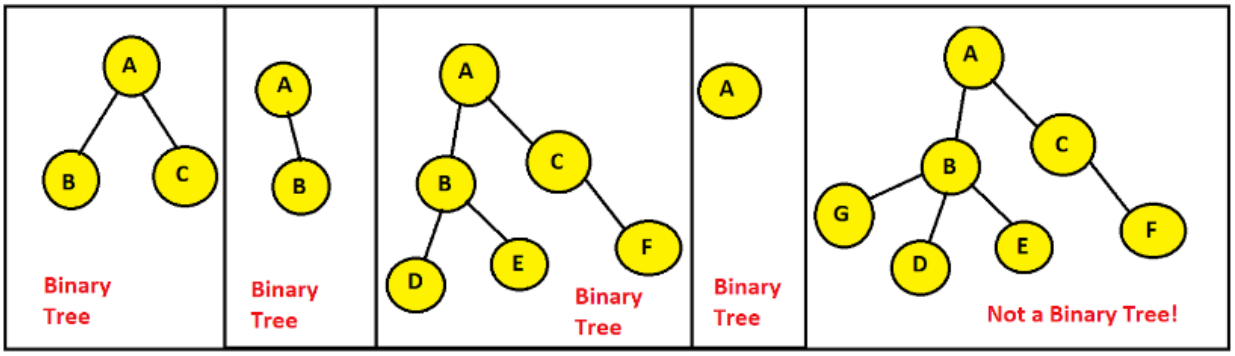
Follow-up Question:

Q) Find the level, depth, height, and degree of the specified nodes of the following tree.



Binary Tree:

A tree is a binary tree if every single node of the tree has at most 2 child nodes.



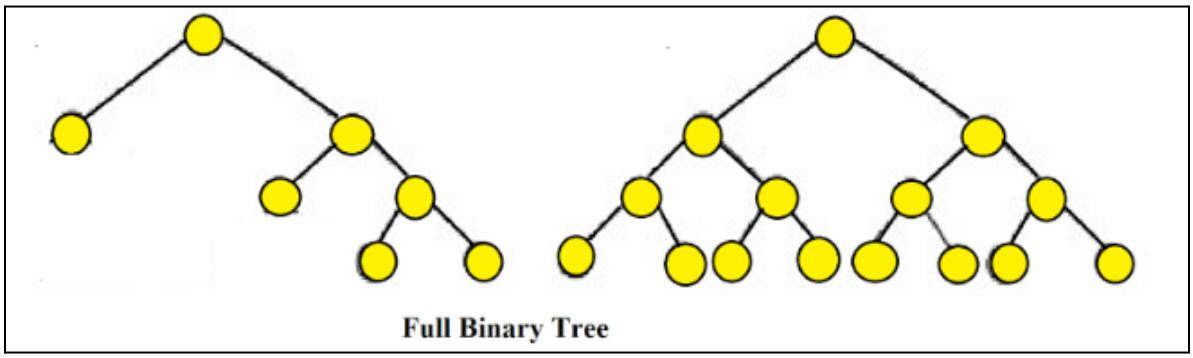
Characteristics of a Binary Tree:

- Each node in a binary tree can have at most two child nodes
- The maximum number of nodes possible in a binary tree of height 'h' is: $2^{h+1} - 1$
- The maximum number of nodes at level i is: 2^{i}

Types of Binary Tree: [Optional]

- Full/Strict Binary Tree:

In a full binary tree, every node has 0 or 2 children. Any binary tree that maintains the following condition is a full binary tree: No of leaf nodes = no of internal nodes + 1

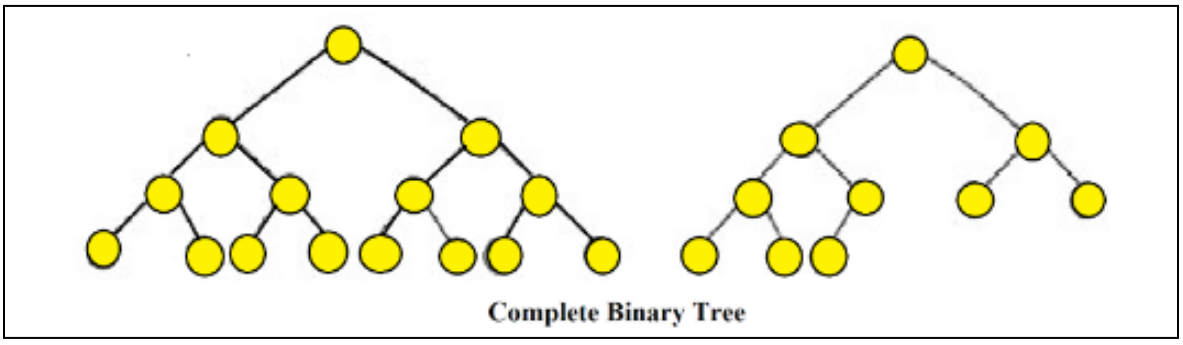


Full Binary Tree

Here, in the leftmost tree, the no of internal nodes is 3 and the no of leaf nodes is 4. Again in the rightmost tree, the no of internal nodes is 7 and the no of leaf nodes is 8.

- Complete Binary Tree:

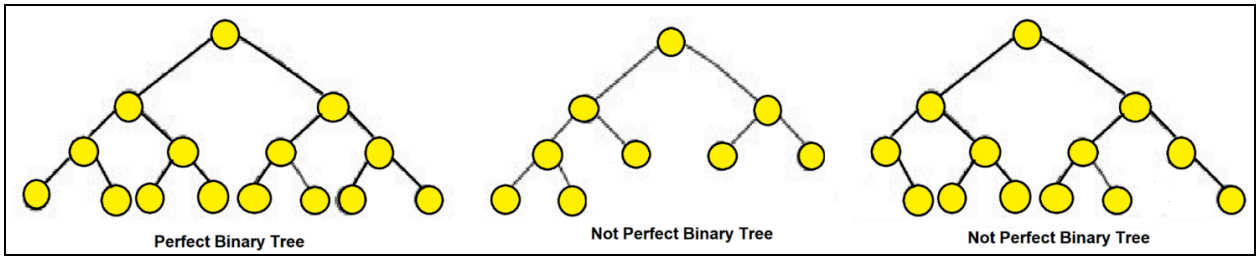
In a complete binary tree, all the levels are filled entirely with nodes, except the lowest level of the tree. Also, in the lowest level of this binary tree, every node should reside on the left side.



Complete Binary Tree

- Perfect Binary Tree:

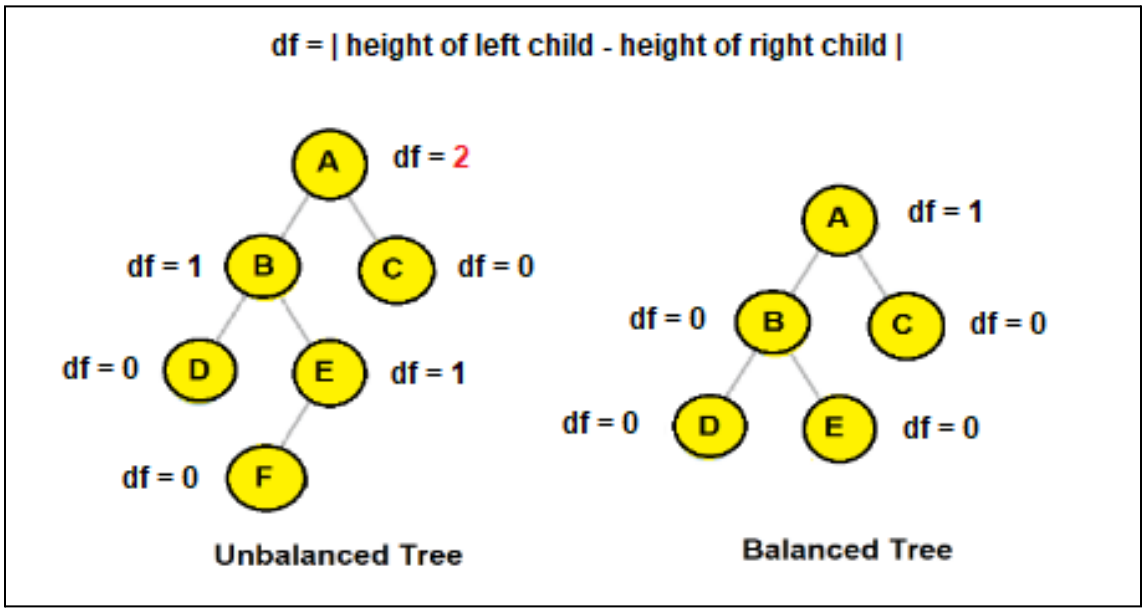
In a perfect binary tree, every internal node has exactly two child nodes and all the leaf nodes are at the same level.



In the tree in the middle, all the leaf nodes are not on the same level. In the rightmost tree, not all internal nodes have two children.

- Balanced Binary Tree:

In a balanced binary tree, the height of any node's left and right subtree will not differ by more than 1. Balanced binary trees are also referred to as height-balanced binary trees.



In the leftmost tree, the height of Node A's left subtree is 2 and right subtree is 0. Therefore, the difference between these two is 2. On the other hand, in the rightmost tree, no nodes have a height difference of more than 1 between their left and right subtrees.

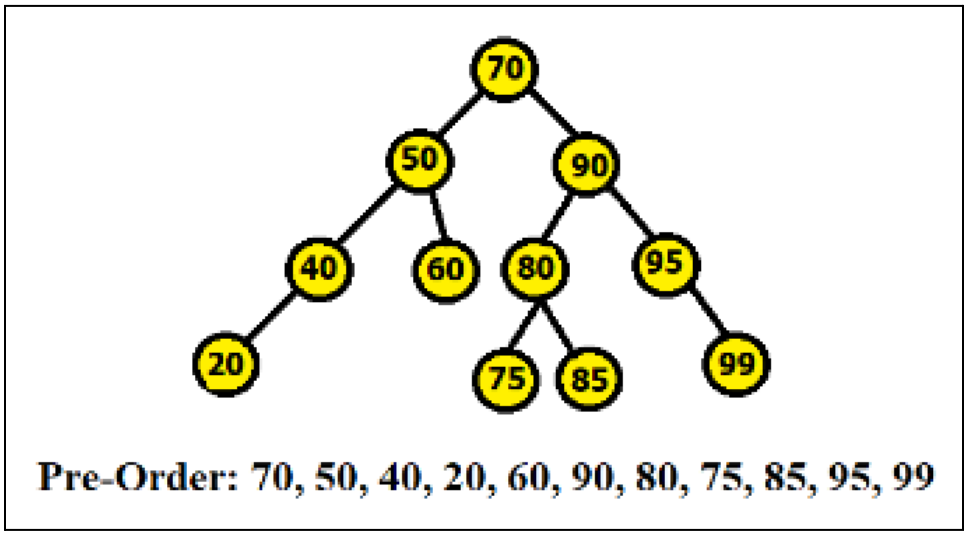
Note: When there is only 1 node in a subtree, the height of that subtree is 1. When there is no node in a subtree, the height of that subtree is considered as -1

Binary Tree Traversal:

- Pre-order:

In pre-order traversal, we start at the root node and print its value. Next, we move to the left child of the root and repeat the process: print the node's value and then move to its left child. If a node has no left child, we return to that node and check for a right child. If there is a right child, we move to it and repeat the process. If a node has no children, we return to its parent and check for the right child. This process continues until all nodes are visited. (The values of the nodes are printed when they are visited for the first time)

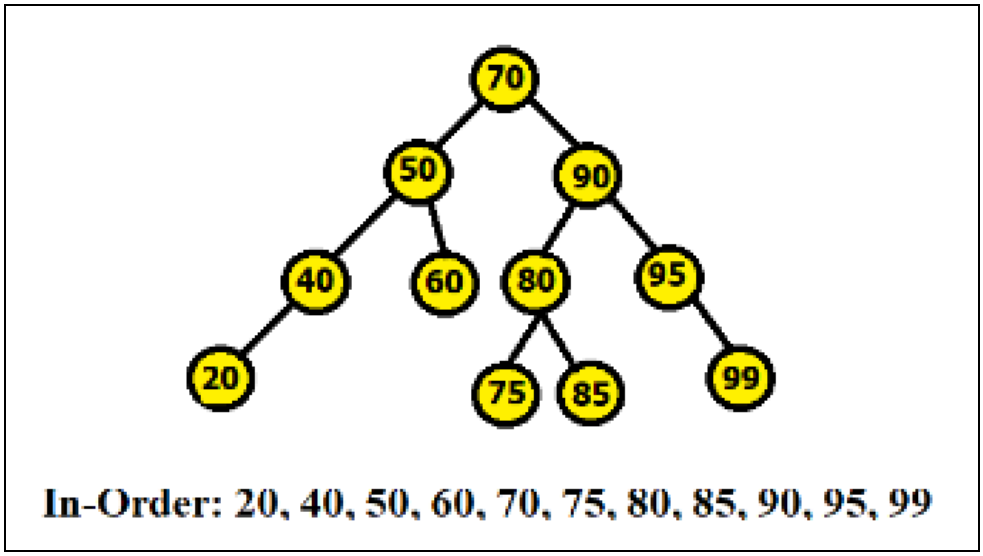
Visit Order: **Root, Left, Right**



- In-order:

In in-order traversal, we start at the root and first traverse its left subtree. When we reach a node without a left child, we visit that node for the first time and print its value. Then, we check for the right child. If there is no right child, we also return to the parent node and print its value. After printing a node, we check its right child and repeat the process. This means we always print the node's value after visiting its left subtree and before visiting its right subtree. The root node is printed after all the nodes in its left subtree have been printed, and then we traverse its right subtree. (The values of the nodes are printed when they are visited for the second time.)

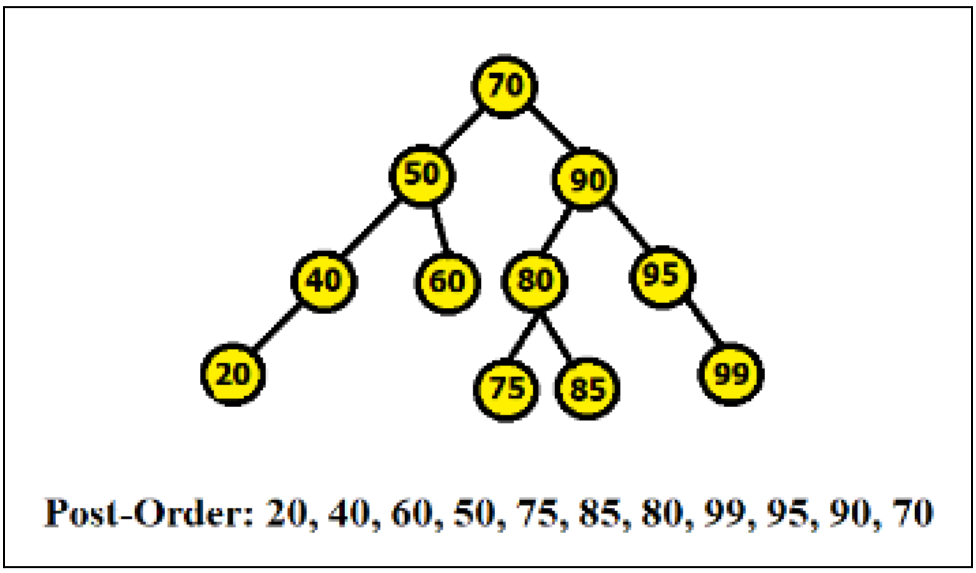
Visit Order: **Left, Root, Right**



- Post-order:

In post-order traversal, we start by traversing the left subtree of the root. For each node, we first explore its left subtree. If a node does not have a left child, we check if it has a right child. If it has no right child either, we return to the node and print its value as we visit it for the third time. After printing the node, we go back to its parent. This process continues until all nodes in the left subtree are visited and printed. Next, we traverse the right subtree in the same manner. Once the right subtree is completely processed and all nodes are printed, we return to the root node and print its value as the last step. In post-order traversal, the root is printed only after all nodes in both its left and right subtrees have been visited and printed. (The values of the nodes are printed when they are visited for the third time.)

Visit Order: **Left, Right, Root**



Binary Tree Coding:

Binary Tree implementation:

There are two ways to represent binary trees:

i) Dynamic Representation (Using Linked List)
ii) Sequential Representation (Using Array)

We have already covered the dynamic representation of trees. Now let us look at how to represent binary trees sequentially.

Sequential Representation (Using Array) Conditions:

- If the height of the binary tree is h, an array of maximum 2^{h+1} length is required.
- The root is placed at index 1.
- Any node that is placed at index i, will have its left child at 2i and its right child at 2i+1.

```
class Node:
    def __init__(self, elem):
        self.elem = elem
        self.left = None
        self.right = None
```

```
#Binary Tree from Array

def tree_construction(arr, i, n):
    root = None
    if i < n:
        if (arr[i] != None):
            root = Node(arr[i])
            root.left = tree_construction(arr, 2*i, n)
            root.right = tree_construction(arr, 2*i + 1, n)
    return root

arr = [None, "A", "B", "C", "D", "E", "F", None]
root = tree_construction(arr, 1, len(arr))
```

```
#Array from Binary Tree

arr = [None] * 2**(h+1)
def array_construction(root, i):
        if root == None:
                return None
        else:
                arr[i] = root.elem
                array_construction(root.left, 2*i)
                array_construction(root.right, 2*i + 1)

array_construction(root, 1)
```

Level, Height, Depth Finding:

```
def get_level(root, elem, level=0):
        if root is None:
                return -1  # Return -1 if the element is not found
        if root.elem == elem:
                return level
        # Search in the left subtree
        left_level = get_level(root.left, elem, level + 1)
        if left_level != -1:
                return left_level
        # If not found in the left subtree, search in the right subtree
        return get_level(root.right, elem, level + 1)

print(get_level(root, 0, 4))
```

```
def get_height(root):
        if root is None:
                return -1
        else:
                return 1 + max(get_height(root.left), get_height(root.right))
```

Since, level == depth, get_level function will also work for getting the depth of any node.

Identifying Tree Types: Full, Complete and Perfect: [Optional]

```
#Checking Complete Tree

def count_nodes(root):
        if root is None:
                return 0
        return 1 + count_nodes(root.left) + count_nodes(root.right)

def is_complete(root, i, nodes):
        if root is None:
                return True
        if i > nodes:
                return False
        return is_complete(root.left, 2*i, nodes) and
                 is_complete(root.right, 2*i + 1, nodes)

if is_complete(root, 1, count_nodes(root)):
        print("Complete Binary Tree")
else:
        print("Not a Complete Binary Tree")
```

```
#Checking Full Tree

def is_full(root):
        if root is None:
                return True
        if root.left is None and root.right is None:
                return True
        if root.left is not None and root.right is not None:
                return is_full(root.left) and is_full(root.right)
        return False
if is_full(root):
        print("Full Binary Tree")
else:
        print("Not a Full Binary Tree")
```

```
#Checking Perfect Tree

def calculate_depth(node):
        depth = 0
        while node is not None:
                depth += 1
                node = node.left
        return depth - 1

def is_perfect(root, depth, level=0):
        if root is None:
                return True
        if root.left is None and root.right is None:
                return (depth == level)
        if root.left is None or root.right is None:
                return False
        return is_perfect(root.left, depth, level + 1) and
                 is_perfect(root.right, depth, level + 1)
```

```
if is_perfect(root, calculate_depth(root)):
        print("Perfect Binary Tree")
else:
        print("Not a Perfect Binary Tree")
```