Chapter 5: Tree

5.1 Tree Basics

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.

5.1.1 Introduction to Tree Data Structure

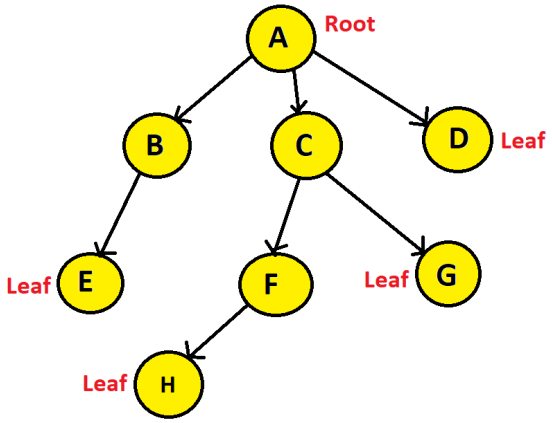
We have covered a number of data structures so far in this course. However, trees are extremely unique as some feats can only be achieved efficiently using trees. Following are some real life applications of trees:

- Continuous Sorting can be achieved using trees. Imagine you have a sorted array of 1 million data. Now you have to add 1000 more data into it. Then comes the responsibility of sorting the entire array again. However, using one variation of trees (Binary Search Tree), we can efficiently insert data later in a sorted manner without having to sort all the data again and again.
- Trees are widely used in folder/file system structure. For example, your desktop folder is the root of the tree and in your desktop you have multiple folders, and in those folders, you have some files. These files are the leaf nodes of the tree.
- Trees are used in electrical circuit designing and electricity transmission. For example, suppose you have a main center point from where all the electricity is generated and from there on it will be spread into branches until it reaches every home, office or industry.
- Router/Computer Network algorithms construct trees of the locations across the network to determine the route that data packets must follow to reach their destination efficiently.

5.1.2 Terminologies of a Tree

5.1.2.1 Root/Leaf/Non-Leaf Node:

Each tree consists of one or more nodes that are interlinked. The topmost node is called the **root** node. Below the root there are one or more nodes. The nodes residing at the bottom, or in other words the nodes 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.



Here we can see that node A is the root node, Nodes E, H, G, and D are the leaf nodes. Nodes A, B, F, and C are non-leaf nodes. The root node and the non-leaf nodes can be considered as internal nodes.

5.1.2.2 Parent/Child:
The direction of the tree is from top to bottom. Which means Node B is the immediate successor of node A, and Node A is the immediate predecessor of node B. Therefore, node B is the child of A, whereas, A is the parent of B.

5.1.2.3 Siblings:
All the nodes having the same parent are known as siblings. Therefore, B, C, and D are siblings, and F and G are siblings.

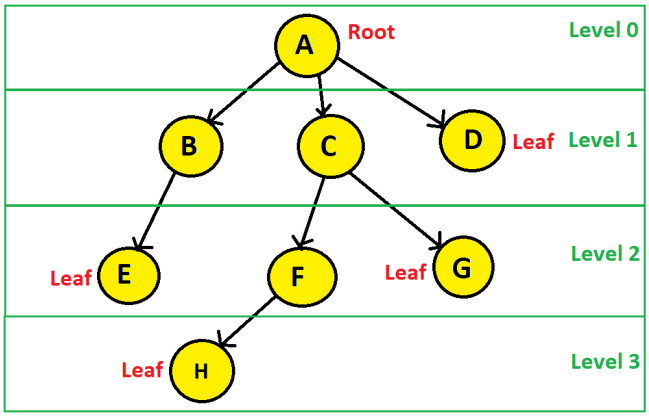
5.1.2.4 Edge/Path:
The link between two nodes is called an edge. A path is known as the consecutive edges from the source node to the destination node. So, if we asked what is the path from node A to E? The answer would be A→B→E. A tree having n number of nodes will have (n-1) number of edges. Here, we have 8 nodes and 7 edges in total.

5.1.2.5 Degree:
The number of children of a node is known as degree. The degree of node A is 3, node B is 1 and Node E is 0.

5.1.2.6 Depth:
The length of the path from a node to the root node is known as the depth. The depth of nodes E, F, and G is 2; depth of B, C, and D is 1; depth of A is 0; depth of H is 3.

5.1.2.7 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 four paths: A→B→E, A→C→F→H, A→C→G, and A→D. Of these four paths, A→C→F→H is the longest path. Hence, the height of Node A is 3.

5.1.2.8 Level:
Each hierarchy starting from the root is known as the level.



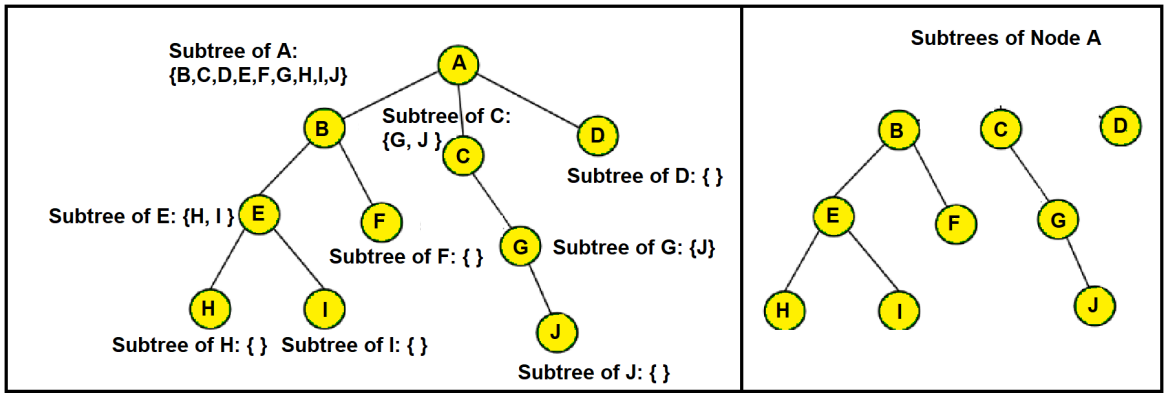
From the above figure, we can see that the level of node A is 0; level of nodes B, C, and D is 1; level of nodes E, F, and G is 2; and level of node H is 3.

Points to remember:

- *The depth and height of a node may not be the same. The depth of A is 0, whereas, the height of A is 3.*
- *Level of a node == Depth of that node.*

5.1.2.9 Subtree:

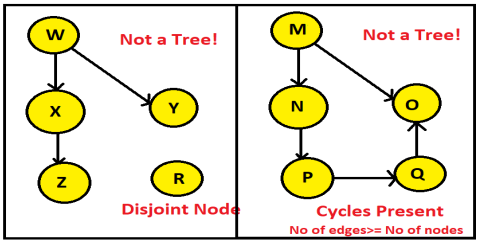
A tree that is the child of a Node.



Any tree can be further divided into subtrees with respect to a particular node. Here node A has three subtrees that are shown on the right side.

5.1.3 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 number of nodes will have n or more edges if it contains one or more cycles. This means, for a tree, no of edges == no of nodes - 1.



5.1.4 Tree Coding - Tree Construction using Linked List (Dynamic Representation)

A tree can be represented using a linked list (Dynamic Representation) or an array (Sequential Representation). Here we shall see how to dynamically represent trees.

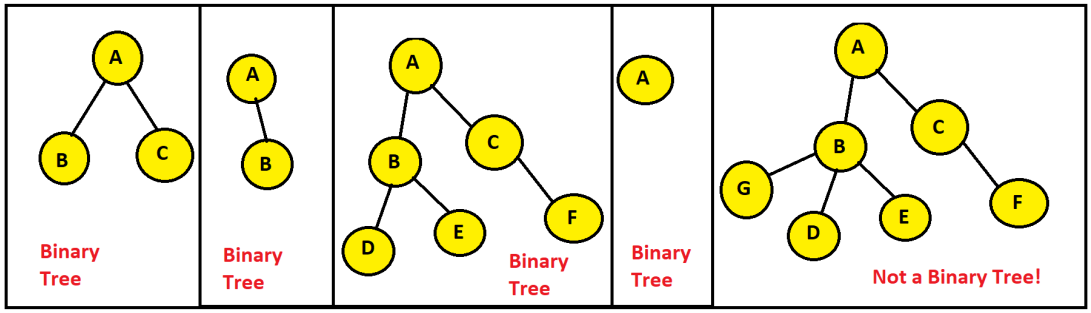
1. Create a class called `Node` with the following data members:
   - `elem`: to store the value of the node.
   - an array named `children`: to store the references of all the children.

2. The first node created becomes the root of the tree

3. After creating each child node, their reference must be stored at their parent node's children array.

5.2 Binary Tree

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



5.3 Characteristics of a Binary Tree

- Each node in a binary tree can have at most two child nodes
- If the number of internal nodes is n, number of external nodes is n+1, number of edges is 2n, number of internal edges is n-1, number of external edges is n+1.
- 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}

5.4 Binary Tree Traversal: Pre-order, In-order, Post-order

Pre-order

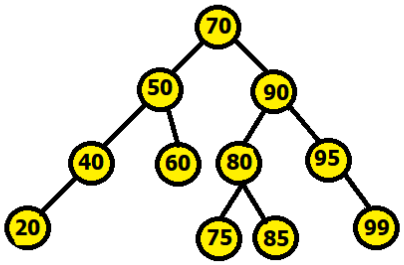
Whenever a node is visited for the **first** time, its element is printed. We start from the root and print its element. Then its left subtree is traversed. If a node does not have a left child, we return to that node After all the nodes of its entire left subtree have been traversed thrice, we head back to the root node for the second time and check if it has any right child, and if it does not have a right child either, we again return to that node After all the nodes of its entire left subtree have been traversed thrice, we head back to the root node for the third time and then go towards its parent node. After all the nodes of its entire left subtree have been traversed twice, we head back to the root node. After that, its right subtree is traversed. The root is printed at first.

In-order

Whenever a node is visited for the **second** time, its element is printed. We start from the root and traverse its left subtree. If a node does not have a left child, we return to that node for the second time and print its element. Then we check if it has any right child, and if it does not have a right child either, we again return to that node for the third time and then go towards its parent node. After all the nodes of its entire left subtree have been traversed thrice, we head back to the root node for the second time and print its element. The root is printed after all the nodes of its left subtree are printed. After that, its right subtree is traversed. After all the nodes of its entire right subtree have been traversed thrice, we head back to the root node for the third time. **In-order traversal also sorts the data in ascending order.**

Post-order

Whenever a node is visited for the **third** time, its element is printed. We start from the root and traverse its left subtree. If a node does not have a left child, we return to that node for the second time and check if it has any right child, and if it does not have a right child either, we again return to that node for the third time, print its element and then go towards its parent node. After all the nodes of its entire left subtree have been traversed thrice, we head back to the root node and then traverse its right subtree. After all the nodes of its entire right subtree have been traversed thrice, we head back to the root node for the third time and print its element. The root is printed at the last.



Pre-Order: 70, 50, 40, 20, 60, 90, 80, 75, 85, 95, 99

In-Order: 20, 40, 50, 60, 70, 75, 80, 85, 90, 95, 99

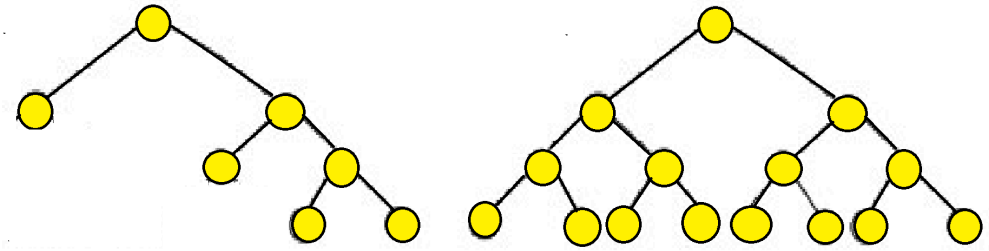
Post-Order: 20, 40, 60, 50, 75, 85, 80, 99, 95, 90, 70

5.5 Types of a Binary Tree

5.5.1 Full/Strict Binary Tree

In a full binary tree, internal nodes (every node except the leaf nodes) have two children. How can we identify a full binary tree? 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

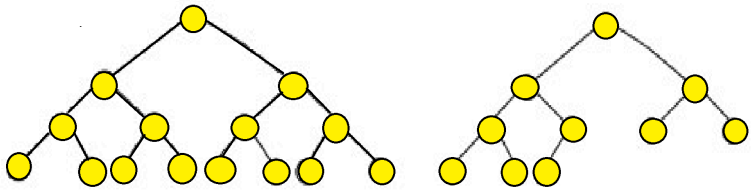
Here, in the leftmost tree, the no of internal nodes is 3 and the no of leaf nodes is 4. Again in the rightmost tree, the no of internal nodes is 7 and the no of leaf nodes is 8.

5.5.2 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 possibly reside on the left side.

How can we identify a complete binary tree?

- If all the internal nodes (every node except the leaf nodes) have two childs, then it is a complete binary tree.
- If any of the internal nodes has only one child, the child must reside in the left side and not in the right side.

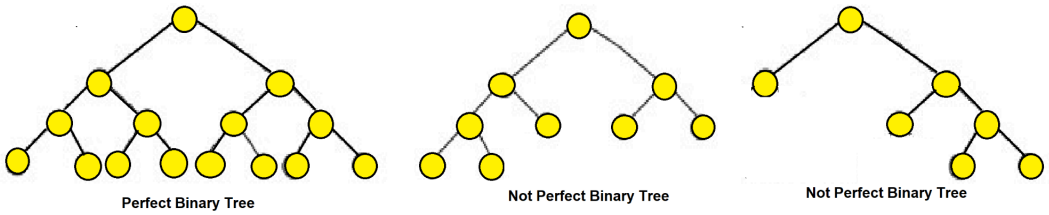


Complete Binary Tree

In the leftmost tree, all internal nodes have two children. Therefore, it is a complete binary tree. In the rightmost tree, all internal nodes except one have two children. The only internal node that has one child, has its child residing on its left side. Therefore, it is also a complete binary tree.

5.5.3 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.



Perfect Binary Tree

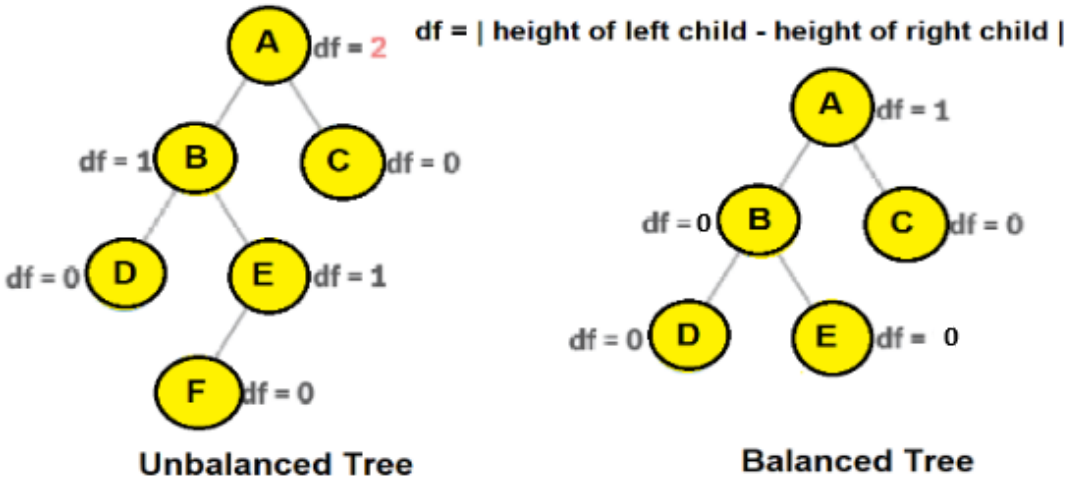
Not Perfect Binary Tree

Not Perfect Binary Tree

In the tree in the middle and rightmost position, all the leaf nodes are not on the same level.

5.5.4 Balanced Binary Tree

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



Unbalanced Tree

Balanced Tree

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 in between their left and right subtrees.

Points to remember:

- *Every perfect binary tree is also a full binary tree, but every full binary tree is not a perfect binary tree.*

5.6 Binary Tree Coding

5.6.1 Tree Construction using Array (Sequential Representation)

Have you ever thought about how to represent binary trees in your program? If you can recall, we have been using linked lists so far to represent trees. So, there are two ways to represent binary trees:

1. Dynamic Representation (Using Linked List)
2. Sequential Representation (Using Array)

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

Sequential Representation (Using Array) Conditions:

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

Pseudocode of Array representation of a Binary Tree:

```
FUNCTION createBinaryTree(arr, index)
    IF index >= length of arr OR arr[index] IS NULL
        RETURN NULL
    ENDIF

    node ← new TreeNode(arr[index])
    node.left ← createBinaryTree(arr, 2 * index)
    node.right ← createBinaryTree(arr, 2 * index + 1)
    RETURN node
END FUNCTION
```

5.6.2 Level, Height, and Depth Finding

```
Depth = number of edges from root to the node

FUNCTION findDepth(root, targetValue, currentDepth)
   IF root IS NULL
      RETURN -1
   ENDIF

   IF root.elem == targetValue
      RETURN currentDepth
   ENDIF

   leftDepth ← findDepth(root.left, targetValue, currentDepth + 1)
   IF leftDepth ≠ -1
      RETURN leftDepth
   ENDIF

   RETURN findDepth(root.right, targetValue, currentDepth + 1)
END FUNCTION

Height = longest path from the node to a leaf

FUNCTION getHeight(node)
   IF node IS NULL
      RETURN -1
   ENDIF

   leftHeight ← getHeight(node.left)
   rightHeight ← getHeight(node.right)
   RETURN max(leftHeight, rightHeight) + 1
END FUNCTION
```

5.6.3 Number of Nodes Finding

```
FUNCTION countNodes(root)
   IF root IS NULL
      RETURN 0
   ENDIF

   RETURN 1 + countNodes(root.left) + countNodes(root.right)
END FUNCTION
```

5.6.4 Identifying Tree Types: Full, Complete and Perfect

Full Binary Tree or Not	Complete Binary Tree or Not

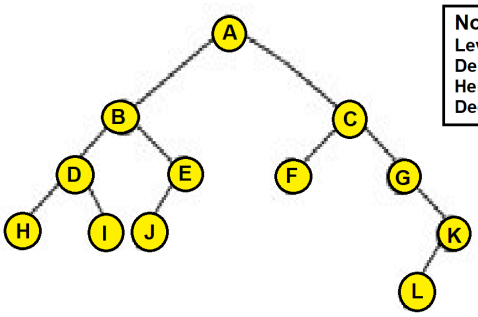
Every node has 0 or 2 children.
 	
FUNCTION isFullTree(root)
  IF root IS NULL
  RETURN TRUE
  ENDIF
 	
  IF (root.left IS NULL AND root.right IS NULL)
  RETURN TRUE
  ENDIF
 	
  IF (root.left IS NOT NULL AND root.right IS NOT NULL)
  RETURN isFullTree(root.left) AND isFullTree(root.right)
  ENDIF
 	
  RETURN FALSE
END FUNCTION
	FUNCTION isComplete(root, index, totalNodes)
	  IF root IS NULL
	  RETURN TRUE
	  ENDIF
	
	  IF index > totalNodes
	  RETURN FALSE
	  ENDIF
	
	  RETURN isComplete(root.left, 2 * index, totalNodes) AND
	  isComplete(root.right, 2 * index + 1, totalNodes)
	END FUNCTION

Perfect Binary Tree or Not	Balanced Binary Tree or Not

FUNCTION isPerfect(root, depth, level)
  IF root IS NULL
  RETURN TRUE
  ENDIF
 	
  IF root.left IS NULL AND root.right IS NULL
  RETURN depth == level + 1
  ENDIF
 	
  IF root.left IS NULL OR root.right IS NULL
  RETURN FALSE
  ENDIF
 	
  RETURN isPerfect(root.left, depth, level + 1) AND
  isPerfect(root.right, depth, level + 1)
END FUNCTION
 	
You can get depth using:
 	
depth = getHeight(root) + 1
	FUNCTION isBalanced(root)
	  IF root IS NULL
	  RETURN (TRUE, -1)
	  ENDIF
	
	  (leftBalanced, leftHeight) ← isBalanced(root.left)
	  (rightBalanced, rightHeight) ← isBalanced(root.right)
	
	  currentBalanced ← leftBalanced AND rightBalanced AND
	  abs(leftHeight - rightHeight) ≤ 1
	
	  currentHeight ← max(leftHeight, rightHeight) + 1
	
	  RETURN (currentBalanced, currentHeight)
	END FUNCTION

Exercises

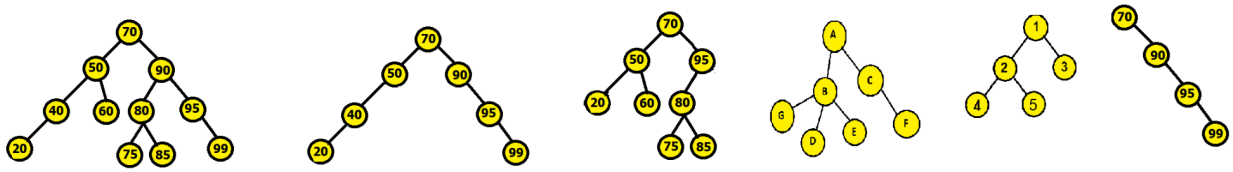
5.1: Find the level, depth, height and degree of the specified nodes of the following tree.



Node A	Node D	Node F	Node C
Level:	Level:	Level:	Level:
Depth:	Depth:	Depth:	Depth:
Height:	Height:	Height:	Height:
Degree:	Degree:	Degree:	Degree:

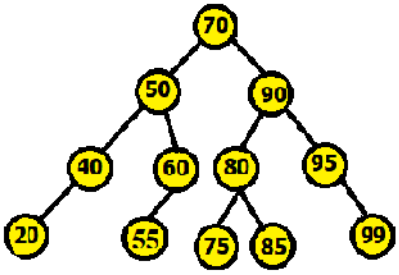
Node E	Node G	Node L
Level:	Level:	Level:
Depth:	Depth:	Depth:
Height:	Height:	Height:
Degree:	Degree:	Degree:

5.2: Identify which of the following trees are full, complete, perfect and balanced.



5.3: Write the code to construct a tree of height 3 and minimum number of 9 nodes. Use your imagination while designing the tree.

5.4: Traverse the following trees in pre-order, in-order and post-order and print the elements. Show both simulation and code.

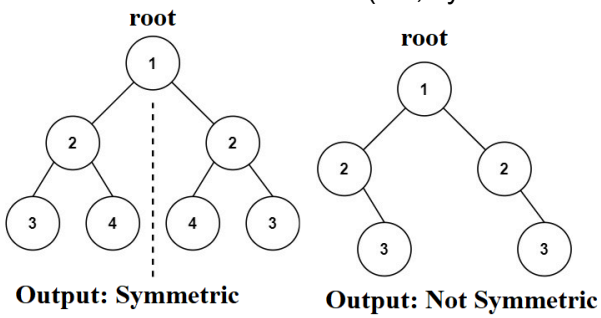


Pre-order:
In-order:
Post-order:

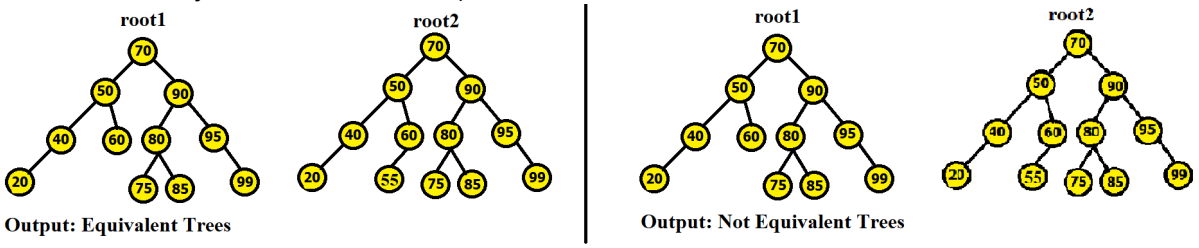
5.5: Consider the following array and convert it into a binary tree. Show simulation and code.

```
[None, 15, 25, 35, 10, 35, 15, 18, None, None, None, 33, None, 5, None, 19, None, None, None, 16]
```

5.6: Write a Python function **isSymmetric(root)** that takes the root of a binary tree, check whether it is a mirror of itself (i.e., symmetric around its center).



5.7: Write a Python function **isIdentical(root1, root2)** that takes the roots of two binary trees, check whether they are identical or not).



5.8: Given two binary trees, find if both of them are identical or not.

Input:

```
    1          1
  /   \      /   \
 2     3    2     3
```

Output: Yes

Explanation: There are two trees both having 3 nodes and 2 edges, both trees are identical having the root as 1, left child of 1 is 2 and right child of 1 is 3.

Input:

```
    1        1
  /   \    /   \
 2     3  3     2
```

Output: No

Explanation: There are two trees both having 3 nodes and 2 edges, but both trees are not identical.

5.9: Given a binary tree, convert it into its mirror.

Input:

```
      10
     /  \
    20   30
   /  \
  40  60
```

Output: 30 10 60 20 40

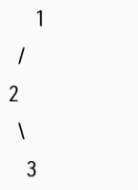
Explanation: The tree is

```
      10                  10
    /    \  (mirror)  /    \
   20    30    =>    30    20
  /  \                /  \
 40  60              60   40
```

The inroder traversal of mirror is 30 10 60 20 40.

5.10: Given a binary tree, find if it is height balanced or not. A tree is height balanced if the difference between heights of left and right subtrees is not more than one for all nodes of the tree.

Input:



Output: 0

Explanation: The max difference in height of left subtree and right subtree is 2, which is greater than 1. Hence unbalanced

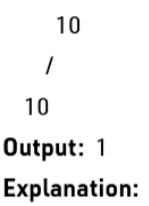
Input:

Output: 1

Explanation: The max difference in height of left subtree and right subtree is 1. Hence balanced.

5.11: Given a binary tree, check whether all of its nodes have the value equal to the sum of their child nodes.

Input:



Output: 1

Explanation: Here, every node is sum of its left and right child.

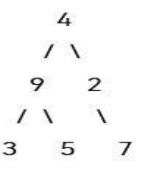
Input:

Output: 0

Explanation: Here, 1 is the root node and 4, 3 are its child nodes. 4 + 3 = 7 which is not equal to the value of root node. Hence, this tree does not satisfy the given conditions.

5.12: Given a binary tree, find the largest value in each level.

Input :



Output : 4 9 7

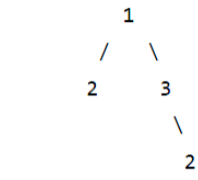
Explanation :

There are three levels in the tree:

1. {4}, max = 4
2. {9, 2}, max = 9
3. {3, 5, 7}, max=7

5.13: Given a binary tree, check if it has duplicate values.

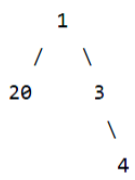
Input : Root of below tree



Output : Yes

Explanation : The duplicate value is 2.

Input : Root of below tree

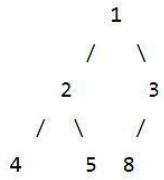


Output : No

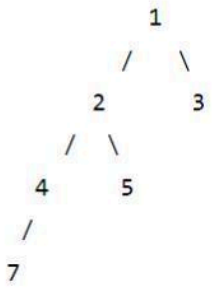
Explanation : There are no duplicates.

5.14: Given a root of a binary tree, and an integer k, print all the nodes which are at k distance from root. Distance is the number of edges in the path from the source node (Root node in our case) to the destination node.

For example, in the below tree, 4, 5 & 8 are at distance 2 from root.

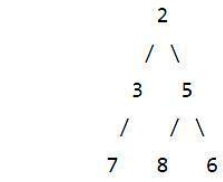


5.15: Given a binary tree and a key, write a function that prints all the ancestors of the node with the key in the given binary tree. For example, if the given tree is following binary Tree and the key is 7, then your function should print 4, 2, 1.



5.16: Given a binary tree, print all the nodes having exactly one child. Print "-1" if no such node exists.

Input:

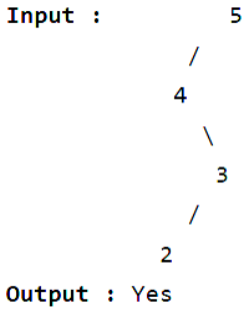


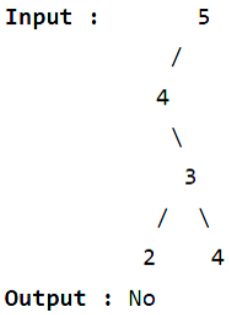
Output: 3

Explanation:

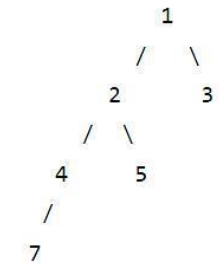
There is only one node having single child that is 3.

5.17: Given a binary tree, check whether it is a skewed binary tree or not. A skewed tree is a tree where each node has only one child node or none.

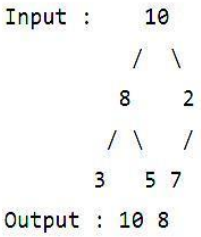




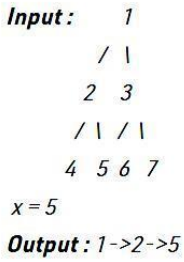
5.18: Given a binary tree, find its minimum depth. The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node. For example, the minimum depth of the below Binary Tree is 2.



5.19: Given a binary tree, print all nodes that are full nodes. Full nodes are nodes which have both left and right children as non-empty.



5.20: Given a binary tree with distinct nodes(no two nodes have the same data values). The problem is to print the path from root to a given node **x**. If node **x** is not present then print "No Path".



5.21: Evaluate a given binary expression tree representing algebraic expressions. A binary expression tree is a binary tree, where the operators are stored in the tree's internal nodes, and the leaves contain constants. Assume that each node of the binary expression tree has zero or two children. The supported operators are +, -, * and /.
For example, the value of the following expression tree is 28.