**DSA – ASSIGNMENT 20**

💡 1. Given a binary tree, your task is to find subtree with maximum sum in tree.

**Examples:**

Input1 :

1

/   \\

2      3

/ \    / \

4   5  6   7

Output1 : 28

As all the tree elements are positive, the largest subtree sum is equal to sum of all tree elements.

Input2 :

1

/    \\

-2      3

/ \    /  \

4   5  -6   2

Output2 : 7

Subtree with largest sum is :

-2

/ \

4   5

Also, entire tree sum is also 7.

**Solution. :-**

* Define a recursive function findMaxSubtreeSum(node) that takes a node of the binary tree as input and returns the maximum sum of any subtree rooted at that node.
* Inside the findMaxSubtreeSum function:
  + If the node is null, return 0 (base case).
  + Recursively calculate the maximum sum of the left subtree by calling findMaxSubtreeSum(node.left).
  + Recursively calculate the maximum sum of the right subtree by calling findMaxSubtreeSum(node.right).
  + Calculate the current sum as the sum of the node's value, the sum of the left subtree, and the sum of the right subtree: currentSum = node.value + leftSubtreeSum + rightSubtreeSum.
  + Return the maximum value among the current sum, the sum of the left subtree, and the sum of the right subtree: return max(currentSum, leftSubtreeSum, rightSubtreeSum).
* In the main function:
  + Initialize a variable maxSum to a very small value (e.g., negative infinity).
  + Traverse the binary tree in any order (preorder, inorder, or postorder) and for each node:
    - Calculate the maximum sum of the subtree rooted at that node by calling findMaxSubtreeSum(node).
    - If the calculated maximum sum is greater than maxSum, update maxSum with the new value.
  + Return maxSum as the result.

**class TreeNode:**

**def \_\_init\_\_(self, value):**

**self.value = value**

**self.left = None**

**self.right = None**

**def findMaxSubtreeSum(node):**

**if node is None:**

**return 0**

**leftSubtreeSum = findMaxSubtreeSum(node.left)**

**rightSubtreeSum = findMaxSubtreeSum(node.right)**

**currentSum = node.value + leftSubtreeSum + rightSubtreeSum**

**return max(currentSum, leftSubtreeSum, rightSubtreeSum)**

**def findMaxSubtreeSumInTree(root):**

**if root is None:**

**return 0**

**maxSum = float('-inf')**

**def traverse(node):**

**nonlocal maxSum**

**if node is None:**

**return**

**maxSum = max(maxSum, findMaxSubtreeSum(node))**

**traverse(node.left)**

**traverse(node.right)**

**traverse(root)**

**return maxSum**

**root = TreeNode(1)**

**root.left = TreeNode(2)**

**root.right = TreeNode(3)**

**root.left.left = TreeNode(4)**

**root.left.right = TreeNode(5)**

**root.right.left = TreeNode(6)**

**root.right.right = TreeNode(7)**

**maxSubtreeSum = findMaxSubtreeSumInTree(root)**

**print(maxSubtreeSum)**

💡2. Construct the BST (Binary Search Tree) from its given level order traversal.

**Example:**

Input: arr[] = {7, 4, 12, 3, 6, 8, 1, 5, 10}

Output: BST:

7

/    \\

4     12

/  \\     /

3   6  8

/   /   \

1  5   10

**Solution. :-**

* Define a TreeNode class with attributes for value, left, and right.
* Create a function constructBST(levelOrder) that takes the level order traversal array as input and returns the root of the constructed BST.
* Inside the constructBST function:
  + If the levelOrder array is empty, return None.
  + Create a variable root and assign it the first element of the levelOrder array.
  + Create two empty lists: leftSubtree and rightSubtree.
  + Iterate over the remaining elements in the levelOrder array:
    - If the current element is less than root, append it to the leftSubtree list.
    - If the current element is greater than root, append it to the rightSubtree list.
  + Recursively call constructBST on the leftSubtree list to construct the left subtree of root. Assign the result to root.left.
  + Recursively call constructBST on the rightSubtree list to construct the right subtree of root. Assign the result to root.right.
  + Return root.

**class TreeNode:**

**def \_\_init\_\_(self, value):**

**self.value = value**

**self.left = None**

**self.right = None**

**def constructBST(levelOrder):**

**if not levelOrder:**

**return None**

**root = TreeNode(levelOrder[0])**

**leftSubtree = []**

**rightSubtree = []**

**for i in range(1, len(levelOrder)):**

**if levelOrder[i] < root.value:**

**leftSubtree.append(levelOrder[i])**

**else:**

**rightSubtree.append(levelOrder[i])**

**root.left = constructBST(leftSubtree)**

**root.right = constructBST(rightSubtree)**

**return root**

**levelOrder = [7, 4, 12, 3, 6, 8, 1, 5, 10]**

**root = constructBST(levelOrder)**

**# Function to print the BST**

**def printBST(root):**

**if root is None:**

**return**

**print(root.value)**

**printBST(root.left)**

**printBST(root.right)**

**printBST(root)**

💡 3. Given an array of size n. The problem is to check whether the given array can represent the level order traversal of a Binary Search Tree or not.

**Examples:**

Input1 : arr[] = {7, 4, 12, 3, 6, 8, 1, 5, 10}

Output1 : Yes

For the given arr[], the Binary Search Tree is:

7

/    \\

4     12

/  \\     /

3   6  8

/   /   \

1  5   10

Input2 : arr[] = {11, 6, 13, 5, 12, 10}

Output2 : No

The given arr[] does not represent the level order traversal of a BST.

**Solution. :-**

* Create a function isValidLevelOrder(arr) that takes the array as input and returns a boolean indicating whether the array represents a valid level order traversal of a BST.
* If the array is empty, return True (an empty array is considered a valid level order traversal).
* Set the variable n to the length of the array.
* Initialize an empty stack stack and set the variable root to the first element of the array.
* Iterate over the array from index 1 to n - 1:
  + If the current element is less than root, return False (the array cannot represent a BST).
  + If the stack is not empty and the current element is greater than the top element of the stack, pop elements from the stack until the stack is empty or the current element is less than the top element.
    - Set root to the last popped element.
  + Push the current element onto the stack.
* Return True if the loop completes without returning False.

**def isValidLevelOrder(arr):**

**if not arr:**

**return True**

**n = len(arr)**

**stack = []**

**root = arr[0]**

**for i in range(1, n):**

**if arr[i] < root:**

**return False**

**while stack and arr[i] > stack[-1]:**

**root = stack.pop()**

**stack.append(arr[i])**

**return True**

**arr = [7, 4, 12, 3, 6, 8, 1, 5, 10]**

**isValid = isValidLevelOrder(arr)**

**print(isValid)**