**DSA – ASSIGNMENT 21**

💡1. You are given a binary tree. The binary tree is represented using the TreeNode class. Each TreeNode has an integer value and left and right children, represented using the TreeNode class itself. Convert this binary tree into a binary search tree.

Input:

10

/ \\

2 7

/ \

8 4

Output:

8

/ \\

4 10

/ \

2 7

**Solution. :-**

* Define a TreeNode class with attributes for value, left, and right.
* Create a function inOrderTraversal(root, values) that performs an in-order traversal of the binary tree and appends the node values to the values list.
* Inside the inOrderTraversal function:
  + If the root is None, return.
  + Recursively call inOrderTraversal on the left subtree (root.left).
  + Append the root.value to the values list.
  + Recursively call inOrderTraversal on the right subtree (root.right).
* Create a function constructBST(values) that takes the list of node values as input and returns the root of the constructed binary search tree.
* Inside the constructBST function:
  + If the values list is empty, return None.
  + Calculate the middle index of the values list: mid = len(values) // 2.
  + Create a new TreeNode with the value at the middle index: root = TreeNode(values[mid]).
  + Recursively call constructBST on the left half of the values list and assign the result to root.left.
  + Recursively call constructBST on the right half of the values list and assign the result to root.right.
  + Return root.
* Use the given binary tree to construct the sorted list of values using the inOrderTraversal function.
* Use the sorted list of values to construct the binary search tree using the constructBST function.

**class TreeNode:**

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

**self.value = value**

**self.left = None**

**self.right = None**

**def inOrderTraversal(root, values):**

**if root is None:**

**return**

**inOrderTraversal(root.left, values)**

**values.append(root.value)**

**inOrderTraversal(root.right, values)**

**def constructBST(values):**

**if not values:**

**return None**

**mid = len(values) // 2**

**root = TreeNode(values[mid])**

**root.left = constructBST(values[:mid])**

**root.right = constructBST(values[mid+1:])**

**return root**

**# Given binary tree**

**root = TreeNode(10)**

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

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

**root.left.left = TreeNode(8)**

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

**# Perform in-order traversal to obtain sorted values**

**values = []**

**inOrderTraversal(root, values)**

**# Construct binary search tree from sorted values**

**bstRoot = constructBST(values)**

**# Function to print the BST**

**def printBST(root):**

**if root is None:**

**return**

**printBST(root.left)**

**print(root.value)**

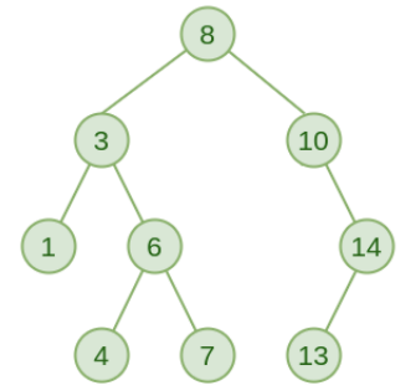
**printBST(root.right)**

**printBST(bstRoot)**

💡 2. Given a Binary Search Tree with all unique values and two keys. Find the distance between two nodes in BST. The given keys always exist in BST.

**Example:**

Consider the following BST:



**Input-1:**

n = 9

values = [8, 3, 1, 6, 4, 7, 10, 14,13]

node-1 = 6

node-2 = 14

**Output-1:**

The distance between the two keys = 4

**Input-2:**

n = 9

values = [8, 3, 1, 6, 4, 7, 10, 14,13]

node-1 = 3

node-2 = 4

**Output-2:**

The distance between the two keys = 2

**Solution. :-**

* Define a TreeNode class with attributes for value, left, and right.
* Create a function findDistance(root, node1, node2) that takes the root of the BST, and the two nodes whose distance needs to be calculated as input.
* Inside the findDistance function:
  + If root is None, return 0.
  + If both node1 and node2 are less than root.value, recursively call findDistance on the left subtree: return findDistance(root.left, node1, node2).
  + If both node1 and node2 are greater than root.value, recursively call findDistance on the right subtree: return findDistance(root.right, node1, node2).
  + If node1 is less than root.value and node2 is greater than root.value, it means node1 and node2 are on different sides of the current root. In this case, return the sum of the distances from the root to node1 and node2: return findDistance(root, node1, None) + findDistance(root, None, node2).
  + If node1 is equal to root.value, return the distance from root to node2 by recursively searching the right subtree: return findDistance(root.right, None, node2) + 1.
  + If node2 is equal to root.value, return the distance from root to node1 by recursively searching the left subtree: return findDistance(root.left, node1, None) + 1.
* Use the given BST values to construct the BST using the TreeNode class.
* Call the findDistance function with the root of the BST and the two target nodes to get the distance between them.

**class TreeNode:**

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

**self.value = value**

**self.left = None**

**self.right = None**

**def findDistance(root, node1, node2):**

**if root is None:**

**return 0**

**if node1 and node2:**

**if node1.value < root.value and node2.value < root.value:**

**return findDistance(root.left, node1, node2)**

**elif node1.value > root.value and node2.value > root.value:**

**return findDistance(root.right, node1, node2)**

**else:**

**return findDistance(root, node1, None) + findDistance(root, None, node2)**

**elif node1 and node1.value == root.value:**

**return findDistance(root.right, None, node2) + 1**

**elif node2 and node2.value == root.value:**

**return findDistance(root.left, node1, None) + 1**

**# Given BST values**

**bstValues = [8, 3, 1, 6, 4, 7, 10, 14, 13]**

**node1Value = 6**

**node2Value = 14**

**# Construct the BST**

**root = None**

**for value in bstValues:**

**if root is None:**

**root = TreeNode(value)**

**else:**

**current = root**

**while True:**

**if value < current.value:**

**if current.left is None:**

**current.left = TreeNode(value)**

**break**

**else:**

**current = current.left**

**else:**

**if current.right is None:**

**current.right = TreeNode(value)**

**break**

**else:**

**current = current.right**

**# Find the distance between node1 and node2**

**node1 = None**

**node2 = None**

**def findNode(root, value):**

**if root is None:**

**return None**

**if root.value == value:**

**return root**

**if value < root.value:**

**return findNode(root.left, value)**

**else:**

**return findNode(root.right, value)**

**node1 = findNode(root, node1Value)**

**node2 = findNode(root, node2Value)**

**distance = findDistance(root, node1, node2)**

**print("The distance between the two keys =", distance)**

💡 3. Write a program to convert a binary tree to a doubly linked list.

Input:

10

/ \\

5 20

/ \\

30 35

Output:

5 10 30 20 35

**Solution. :-**

* Define a TreeNode class with attributes for value, left, and right.
* Create a function binaryTreeToDLL(root) that takes the root of the binary tree as input and returns the head of the doubly linked list.
* Inside the binaryTreeToDLL function:
  + If root is None, return None.
  + Recursively convert the left subtree into a doubly linked list and assign the result to leftList.
  + Recursively convert the right subtree into a doubly linked list and assign the result to rightList.
  + Create a new node current with the value of root.
  + Set current.left to leftList (the tail of the left subtree list).
  + Set current.right to rightList (the head of the right subtree list).
  + If leftList is not None, update leftList.right to current.
  + If rightList is not None, update rightList.left to current.
  + Return the head of the doubly linked list, which is leftList if it exists, otherwise current.
* Use the given binary tree values to construct the binary tree using the TreeNode class.
* Call the binaryTreeToDLL function with the root of the binary tree to convert it to a doubly linked list.
* Traverse the doubly linked list from the head to the tail, and print the values.

**class TreeNode:**

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

**self.value = value**

**self.left = None**

**self.right = None**

**def binaryTreeToDLL(root):**

**if root is None:**

**return None**

**leftList = binaryTreeToDLL(root.left)**

**rightList = binaryTreeToDLL(root.right)**

**current = TreeNode(root.value)**

**current.left = leftList**

**if leftList is not None:**

**leftList.right = current**

**if rightList is not None:**

**rightList.left = current**

**while rightList is not None and rightList.left is not None:**

**rightList = rightList.left**

**current.right = rightList**

**return leftList if leftList is not None else current**

**# Given binary tree**

**root = TreeNode(10)**

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

**root.right = TreeNode(20)**

**root.right.left = TreeNode(30)**

**root.right.right = TreeNode(35)**

**# Convert binary tree to doubly linked list**

**head = binaryTreeToDLL(root)**

**# Traverse the doubly linked list**

**current = head**

**while current is not None:**

**print(current.value, end=" ")**

**current = current.right**

💡 4. Write a program to connect nodes at the same level.

Input:

1

/ \\

2 3

/ \ / \

4 5 6 7

Output:

1 → -1

2 → 3

3 → -1

4 → 5

5 → 6

6 → 7

7 → -1

**Solution. :-**

* Define a TreeNode class with attributes for value, left, right, and nextRight (the next node at the same level).
* Create a function connectNodes(root) that takes the root of the binary tree as input.
* Inside the connectNodes function:
  + If root is None, return.
  + Create a queue to perform a level order traversal of the binary tree.
  + Enqueue the root node to the queue.
  + While the queue is not empty, perform the following steps:
    - Initialize a variable prev to None. This will keep track of the previously visited node at the same level.
    - Initialize a variable levelSize to the size of the queue.
    - Iterate levelSize times to process all the nodes at the current level:
      * Dequeue a node from the front of the queue and assign it to current.
      * If prev is not None, update prev.nextRight to current.
      * Assign current to prev for the next iteration.
      * Enqueue the left and right children of current (if they exist) to the queue.
  + After processing all the nodes at the current level, update prev.nextRight to None to indicate the end of the level.
  + Repeat the above steps until all levels of the binary tree are processed.
* Use the given binary tree values to construct the binary tree using the TreeNode class.
* Call the connectNodes function with the root of the binary tree to connect the nodes at the same level.
* Traverse the connected nodes and print their values along with their nextRight values.

**from collections import deque**

**class TreeNode:**

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

**self.value = value**

**self.left = None**

**self.right = None**

**self.nextRight = None**

**def connectNodes(root):**

**if root is None:**

**return**

**queue = deque()**

**queue.append(root)**

**while queue:**

**levelSize = len(queue)**

**prev = None**

**for \_ in range(levelSize):**

**current = queue.popleft()**

**if prev is not None:**

**prev.nextRight = current**

**prev = current**

**if current.left is not None:**

**queue.append(current.left)**

**if current.right is not None:**

**queue.append(current.right)**

**prev.nextRight = None**

**# Given binary tree**

**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)**

**# Connect nodes at the same level**

**connectNodes(root)**

**# Traverse the connected nodes and print their values along with nextRight values**

**current = root**

**while current is not None:**

**node = current**

**while node is not None:**

**print(node.value, end=" ")**

**node = node.nextRight**

**print("-1")**

**current = current.left**