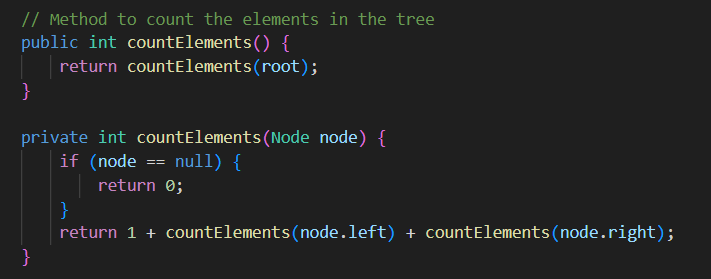
REPORT LAB 6

**Problem 1: Counting Elements in a Binary Tree**

**Purpose**: To count the total number of nodes in a binary tree.

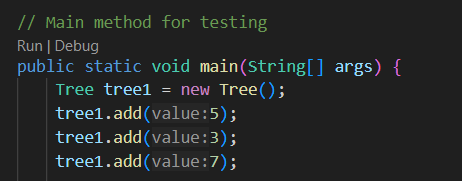
**Implementation**: The method **countElements** is added to the **Tree** class. It uses a helper method **countElements(Node node)** to recursively traverse the tree, starting from the root, and counts each node.



**Functionality**:

* If the current node is null, it returns 0.
* Otherwise, it returns 1 plus the count of the left and right subtrees.
* This method ensures that all nodes in the tree are counted, providing the total number of elements.

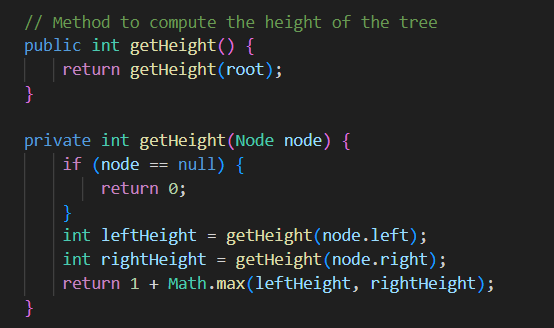
Using sample tree1 to test:



Output: 

**Problem 2: Computing the Height of a Binary Tree**

**Purpose**: To compute the height (or depth) of a binary tree, which is the number of edges on the longest path from the root to a leaf.



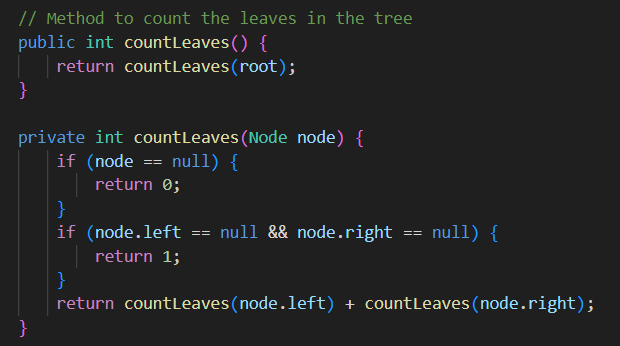
**Functionality**:

* If the current node is null, it returns 0.
* Otherwise, it calculates the heights of the left and right subtrees and returns the greater height plus one.
* This method ensures that the height of the tree is correctly calculated.

Using tree1 to test, we have the output: 

**Problem 3: Counting Leaves in a Binary Tree**

**Purpose**: To count the number of leaf nodes in a binary tree. A leaf node is a node with no children.



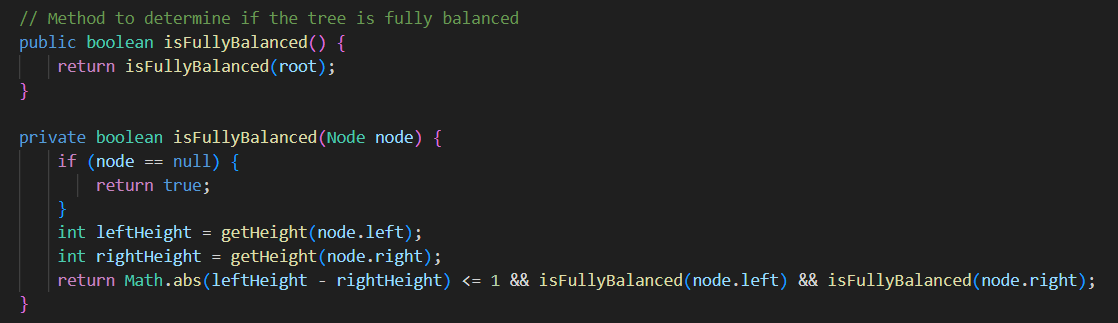
**Functionality**:

* If the current node is null, it returns 0.
* If the current node has no children, it returns 1.
* Otherwise, it returns the sum of the leaf counts from the left and right subtrees.
* This method ensures that all leaves are counted, providing the total number of leaf nodes.

Using tree1 to test, we have the output: 

**Problem 4: Checking if a Binary Tree is Fully Balanced**

**Purpose**: To determine whether a binary tree is fully balanced. A tree is fully balanced if the heights of the left and right subtrees of every node differ by at most one.



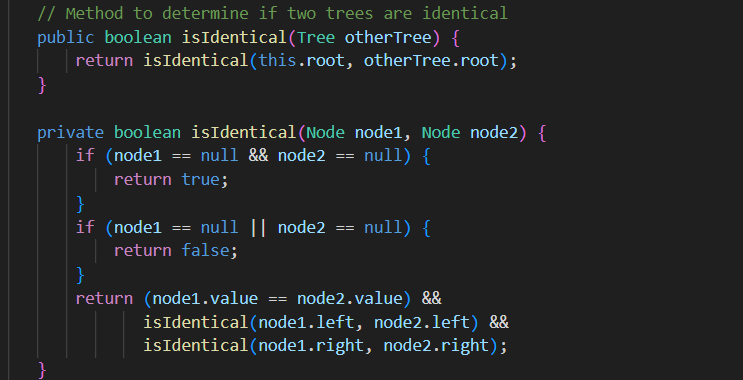
**Functionality**:

* If the current node is null, it returns true.
* It calculates the heights of the left and right subtrees.
* It checks if the height difference is at most one and if both subtrees are fully balanced.
* This method ensures that the balance condition is verified for the entire tree.

Continute using tree1 to test, we have the output: 

**Problem 5: Checking if Two Binary Trees are Identical**

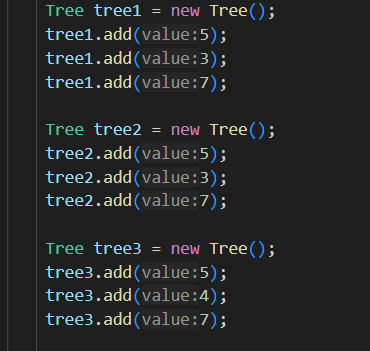
**Purpose**: To determine whether two binary trees are identical. Two trees are identical if they are structurally identical and have the same node values.



**Functionality**:

* If both nodes are null, they are considered identical.
* If one node is null and the other is not, they are not identical.
* If the values of the current nodes are different, they are not identical.
* Otherwise, it checks the left and right subtrees recursively.
* This method ensures that both trees are compared node by node.

This time, we use tree1 to check if it is identical with tree2 and tree3:

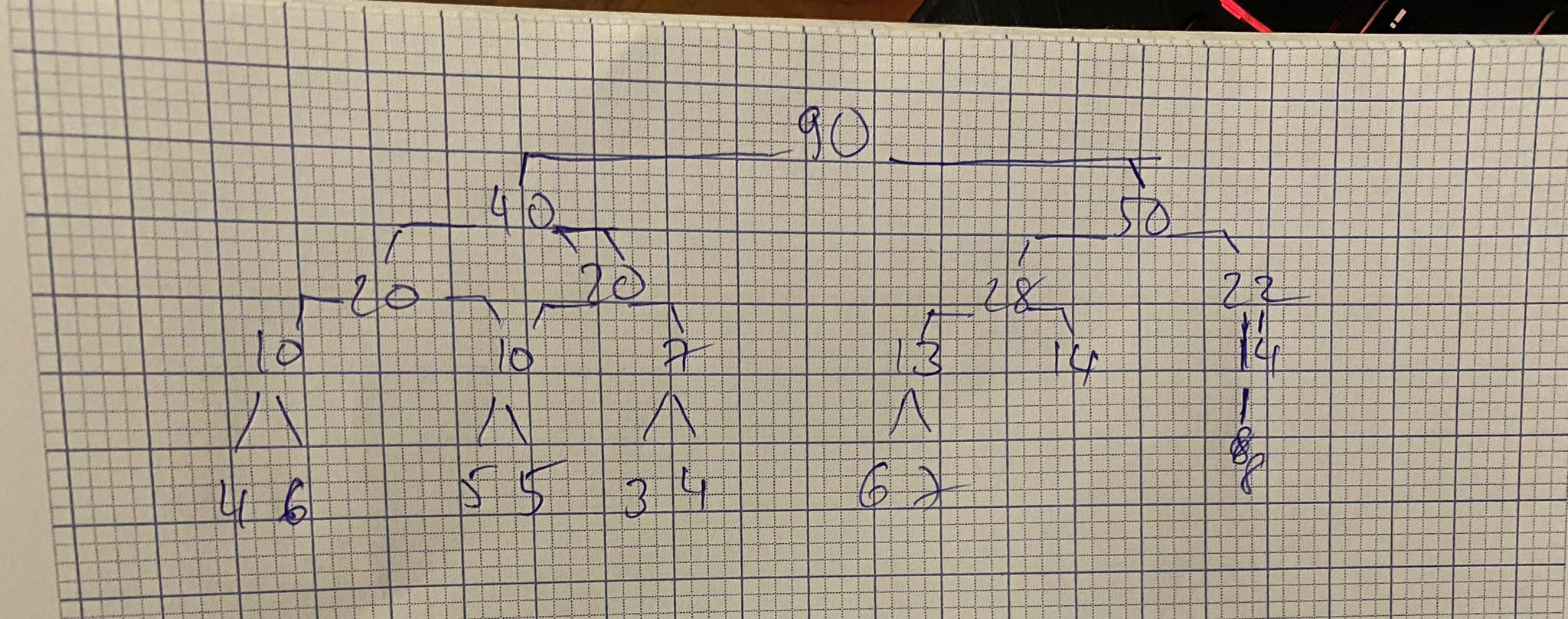


Output:



Problem6:

“I am a student at International University. My name is TA THI PHUONG THAO. I am working on a DSA lab”



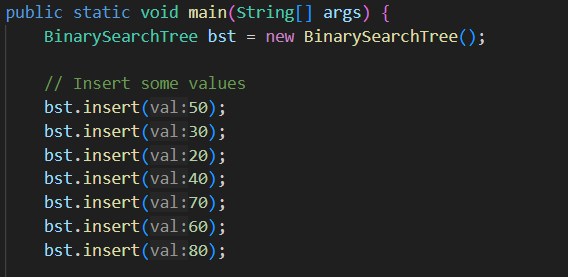
TreeApp.java

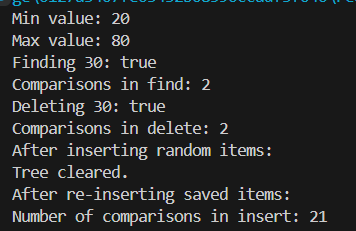
For those functions in this TreeApp, we need to create a class name binary search tree (BST) and add methods in it

Here are functions and its purposes:

* **insert(int val):** Inserts a new value into the binary search tree.
* **find(int val):** Searches for a value in the binary search tree.
* **delete(int val):** Deletes a value from the binary search tree.
* **clear():** Clears the binary search tree.
* **insertRandom(int numItems, int range):** Inserts a specified number of random values into the tree within a given range.
* **findMin():** Finds the minimum value in the binary search tree.
* **findMax():** Finds the maximum value in the binary search tree.
* **traverseAndSave():** Traverses the tree and saves items into an array.
* **getComparisons():** Retrieves the number of comparisons made during the last operation.
* **Other helper methods:** These methods aid in the recursive implementation of insert, find, delete, and traversal operations.

Here we have the inputs are:



The output:   


The choice of traversal method affects the order in which nodes are saved and reinserted, leading to potential variations in the resulting tree structure. However, if the binary search tree maintains its sorting property, the overall relationship between elements should remain consistent across different traversals.

* **In-order traversal:** In this traversal, the nodes are visited in sorted order. Saving the items in an array using in-order traversal and then reinserting them into the tree would result in the same binary search tree structure, assuming no balancing operations are performed during insertion.
* **Pre-order traversal:** In this traversal, the root node is visited first, followed by the left subtree, and then the right subtree. Saving the items in an array using pre-order traversal and then reinserting them into the tree would result in a tree that maintains the same overall shape as the original tree, but the exact structure may differ due to the order in which nodes are reinserted.
* **Post-order traversal:** In this traversal, the left subtree is visited first, followed by the right subtree, and then the root node. Saving the items in an array using post-order traversal and then reinserting them into the tree would likely result in a different tree structure compared to the original tree, as nodes are reinserted in a different order.