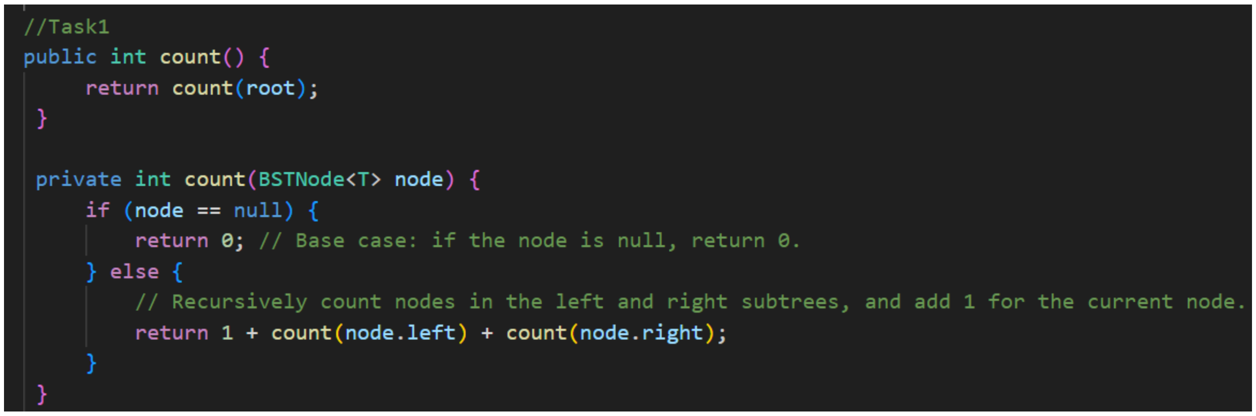
Name: IBRAHIM ALSHAYEA, ID: 202176470, Date: 3/9/2024

Task1: I implemented a method called count() which counts the number of nodes in a binary search tree (BST). This method calls a private recursive method count(BSTNode<T> node). This recursive method traverses the tree, incrementing the count for each node encountered. The base case returns 0 when the current node is null, and the recursive case sums the counts of the left and right subtrees, adding 1 for the current node. This results in an accurate count of all nodes in the tree.

Task1 Code:

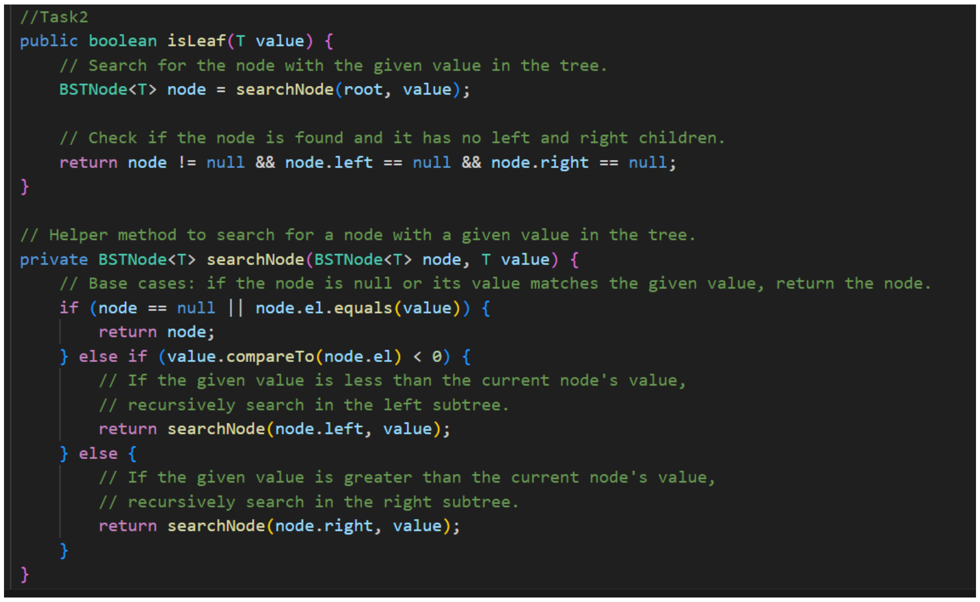


```java
//Task1
public int count() {
     return count(root);
 }

 private int count(BSTNode<T> node) {
     if (node == null) {
         return 0; // Base case: if the node is null, return 0.
     } else {
         // Recursively count nodes in the left and right subtrees, and add 1 for the current node.
         return 1 + count(node.left) + count(node.right);
     }
 }
```

Task2: I implemented a method called isLeaf(T value) which determines whether a node with a given value is a leaf in the binary search tree (BST). This method first searches for the node with the given value in the tree using a helper method called searchNode(BSTNode<T> node, T value). If the node is found and it has no left and right children, the method returns true, indicating that the node is a leaf. Otherwise, it returns false. This approach ensures that we accurately identify leaf nodes in the BST.

Task2 Code:



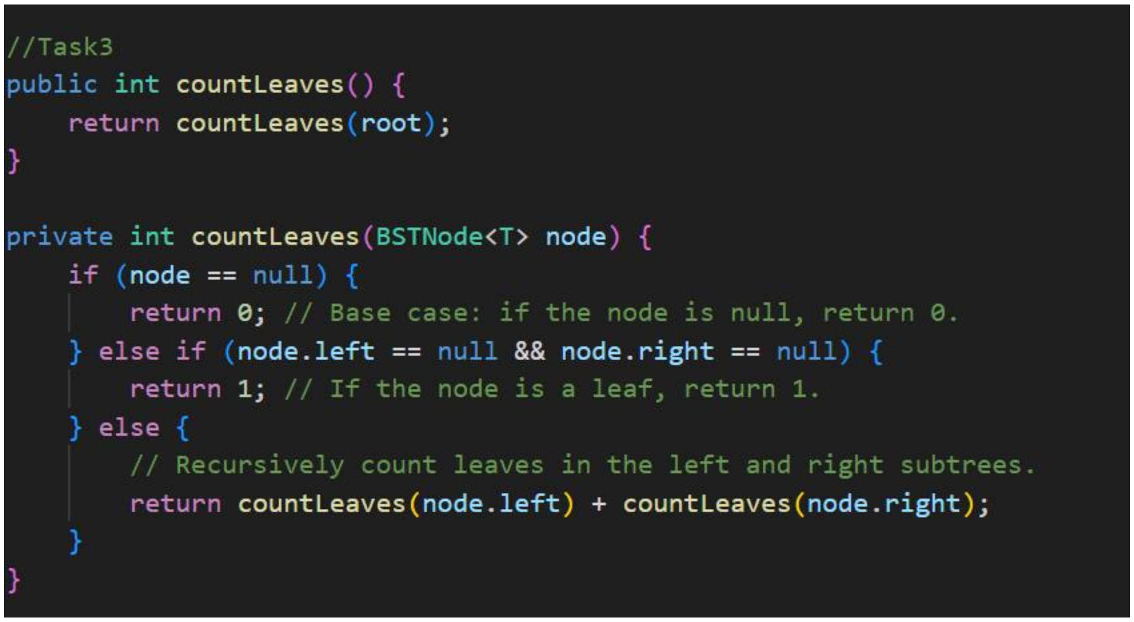
```java
//Task2
public boolean isLeaf(T value) {
    // Search for the node with the given value in the tree.
    BSTNode<T> node = searchNode(root, value);

    // Check if the node is found and it has no left and right children.
    return node != null && node.left == null && node.right == null;
}

// Helper method to search for a node with a given value in the tree.
private BSTNode<T> searchNode(BSTNode<T> node, T value) {
    // Base cases: if the node is null or its value matches the given value, return the node.
    if (node == null || node.el.equals(value)) {
        return node;
    } else if (value.compareTo(node.el) < 0) {
        // If the given value is less than the current node's value,
        // recursively search in the left subtree.
        return searchNode(node.left, value);
    } else {
        // If the given value is greater than the current node's value,
        // recursively search in the right subtree.
        return searchNode(node.right, value);
    }
}
```

Task3: I implemented a method called countLeaves() to count the number of leaf nodes in the binary search tree (BST). This method recursively traverses the tree, incrementing the count each time a leaf node is encountered. If the current node is null or a leaf node, the method returns the appropriate count. Otherwise, it recursively counts the leaf nodes in the left and right subtrees and returns their sum. This approach ensures an accurate count of leaf nodes in the BST.

Task3 Code:

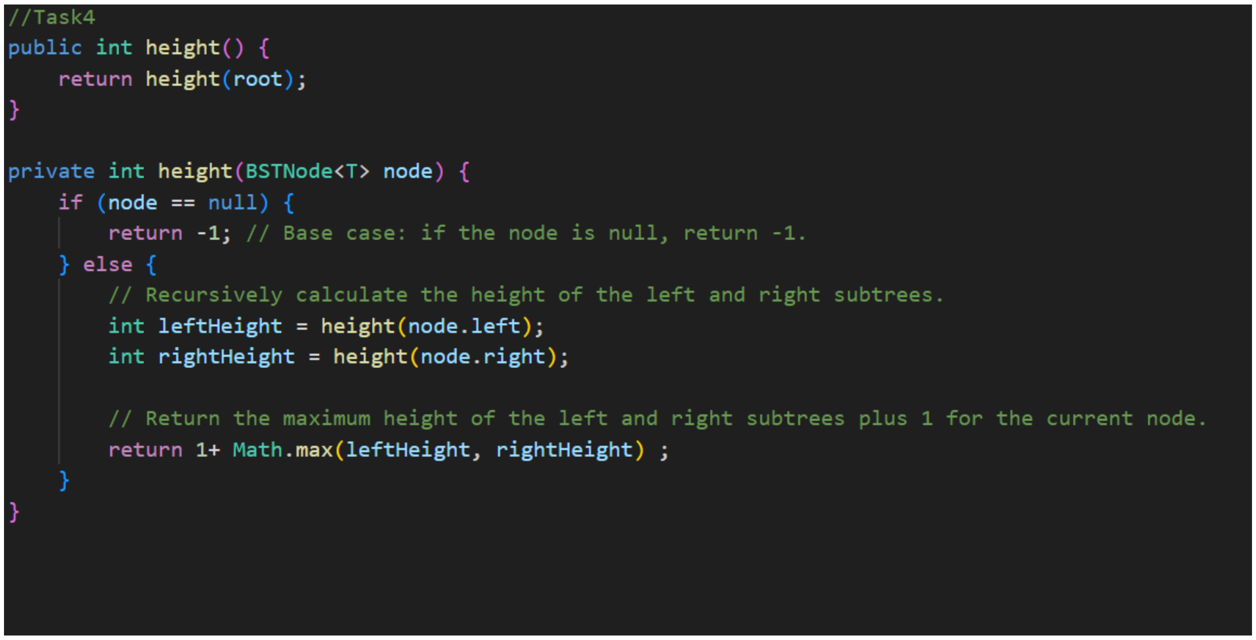


```java
//Task3
public int countLeaves() {
    return countLeaves(root);
}

private int countLeaves(BSTNode<T> node) {
    if (node == null) {
        return 0; // Base case: if the node is null, return 0.
    } else if (node.left == null && node.right == null) {
        return 1; // If the node is a leaf, return 1.
    } else {
        // Recursively count leaves in the left and right subtrees.
        return countLeaves(node.left) + countLeaves(node.right);
    }
}
```

Task4: I implemented a method called height() to calculate the height of the binary search tree (BST). This method recursively traverses the tree to determine its height. If the current node is null, indicating an empty subtree, the method returns -1. Otherwise, it recursively calculates the heights of the left and right subtrees. The height of each subtree is determined by taking the maximum height between the left and right subtrees and adding 1 to account for the current node. Finally, the method returns the height of the tree rooted at the given node. This recursive approach ensures an accurate calculation of the height of the BST.

Task4 Code:



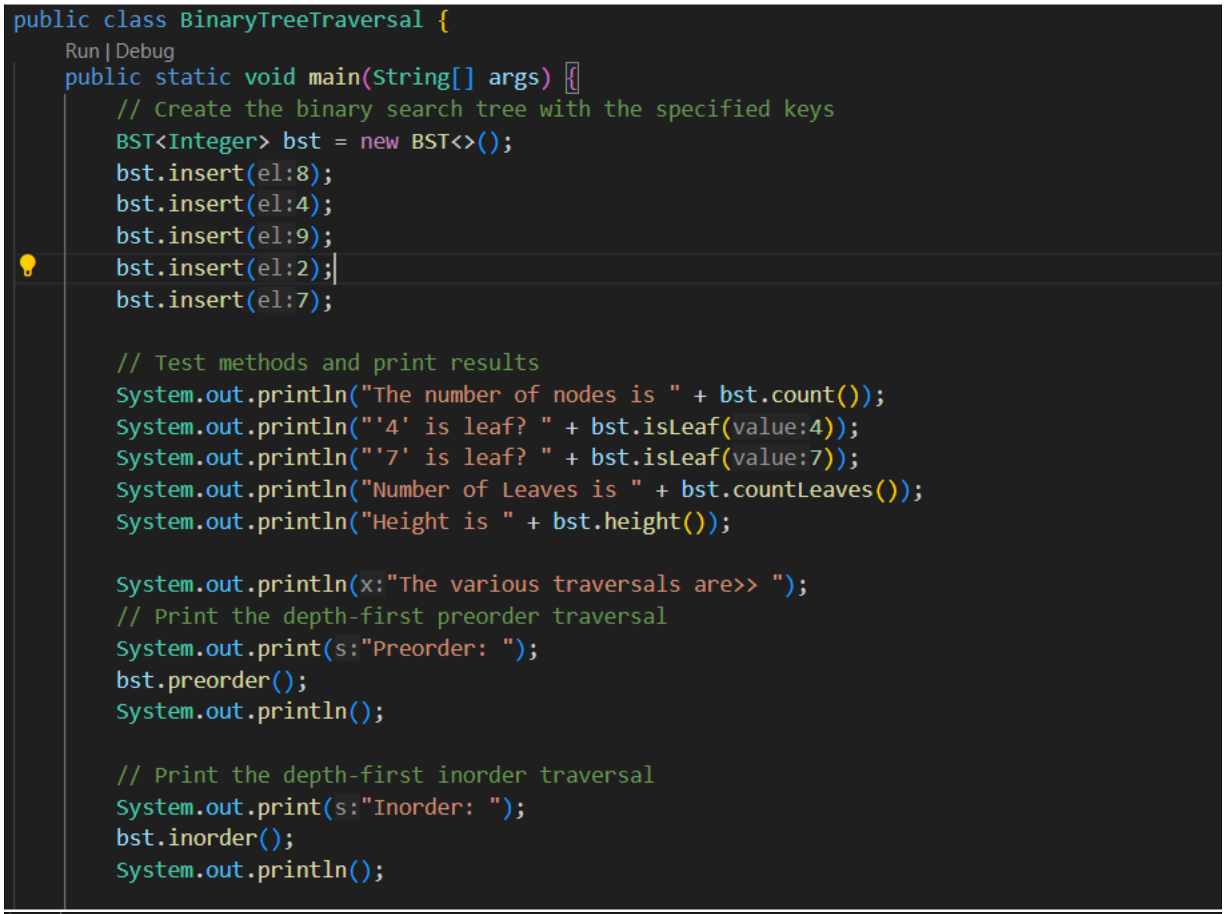
```java
//Task4
public int height() {
    return height(root);
}

private int height(BSTNode<T> node) {
    if (node == null) {
        return -1; // Base case: if the node is null, return -1.
    } else {
        // Recursively calculate the height of the left and right subtrees.
        int leftHeight = height(node.left);
        int rightHeight = height(node.right);

        // Return the maximum height of the left and right subtrees plus 1 for the current node.
        return 1+ Math.max(leftHeight, rightHeight) ;
    }
}
```

Task5 : I created a Java program named BinaryTreeTraversal to demonstrate the creation of a binary search tree (BST). The program begins by constructing a BST with predefined integer values. It then assesses various properties of the tree, such as node count, leaf node identification, leaf node count, and tree height. Finally, it prints the depth-first preorder, inorder, and postorder traversals, as well as the breadth-first traversal of the tree. This program serves as a practical example of BST operations and traversal techniques.

Task5 Code:

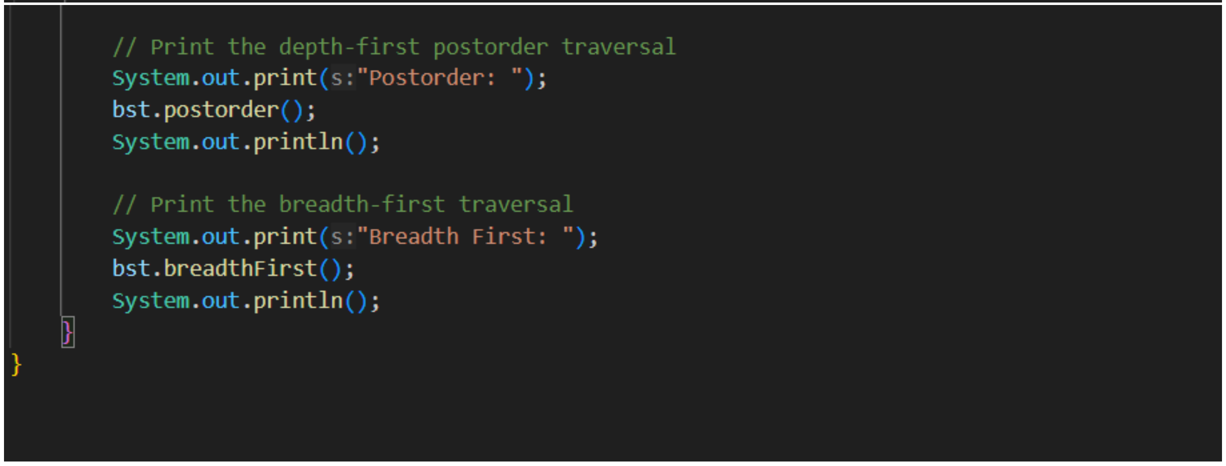


```
public class BinaryTreeTraversal {
    Run | Debug
    public static void main(String[] args) {
        // Create the binary search tree with the specified keys
        BST<Integer> bst = new BST<>();
        bst.insert(el:8);
        bst.insert(el:4);
        bst.insert(el:9);
        bst.insert(el:2);
        bst.insert(el:7);

        // Test methods and print results
        System.out.println("The number of nodes is " + bst.count());
        System.out.println("'4' is leaf? " + bst.isLeaf(value:4));
        System.out.println("'7' is leaf? " + bst.isLeaf(value:7));
        System.out.println("Number of Leaves is " + bst.countLeaves());
        System.out.println("Height is " + bst.height());

        System.out.println(x:"The various traversals are>> ");
        // Print the depth-first preorder traversal
        System.out.print(s:"Preorder: ");
        bst.preorder();
        System.out.println();

        // Print the depth-first inorder traversal
        System.out.print(s:"Inorder: ");
        bst.inorder();
        System.out.println();
```



```
        // Print the depth-first postorder traversal
        System.out.print(s:"Postorder: ");
        bst.postorder();
        System.out.println();

        // Print the breadth-first traversal
        System.out.print(s:"Breadth First: ");
        bst.breadthFirst();
        System.out.println();
    }
}
```