****

University of Central Punjab

***Data Structure and Algorithms***

***Lab # 09***

**(BINARY SEARCH TREE)**

**Name:** Abdullah Maqbool

**Reg no:** L1F22BSSE0391

**Section:** P4

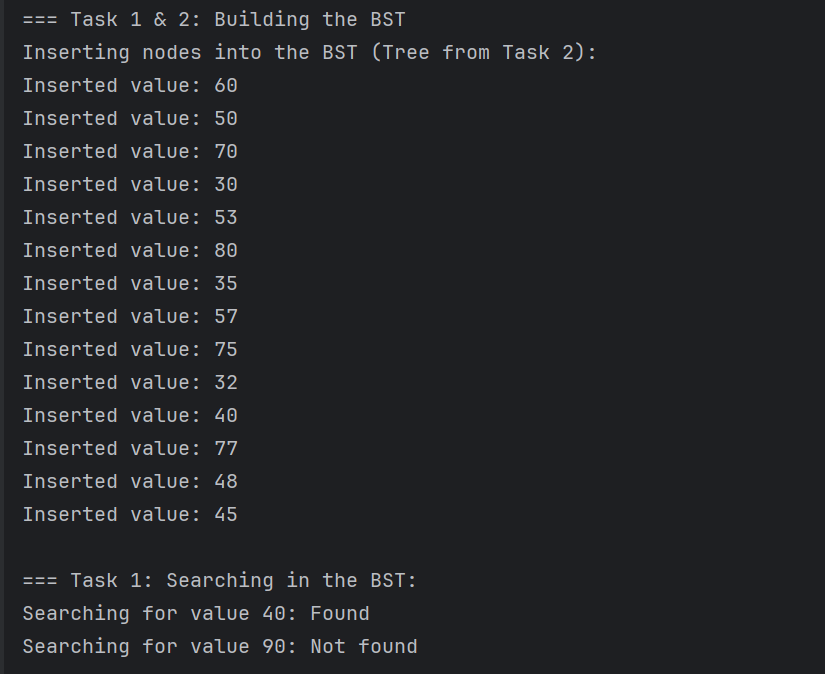
**Submitted to:** Ma`am Javaria Tanveer

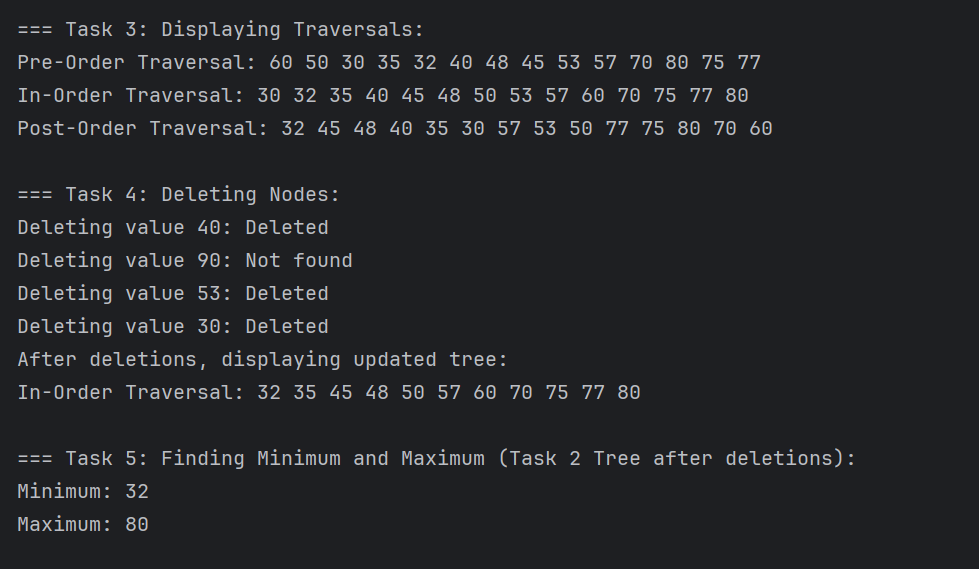
***======================== All Tasks =======================***

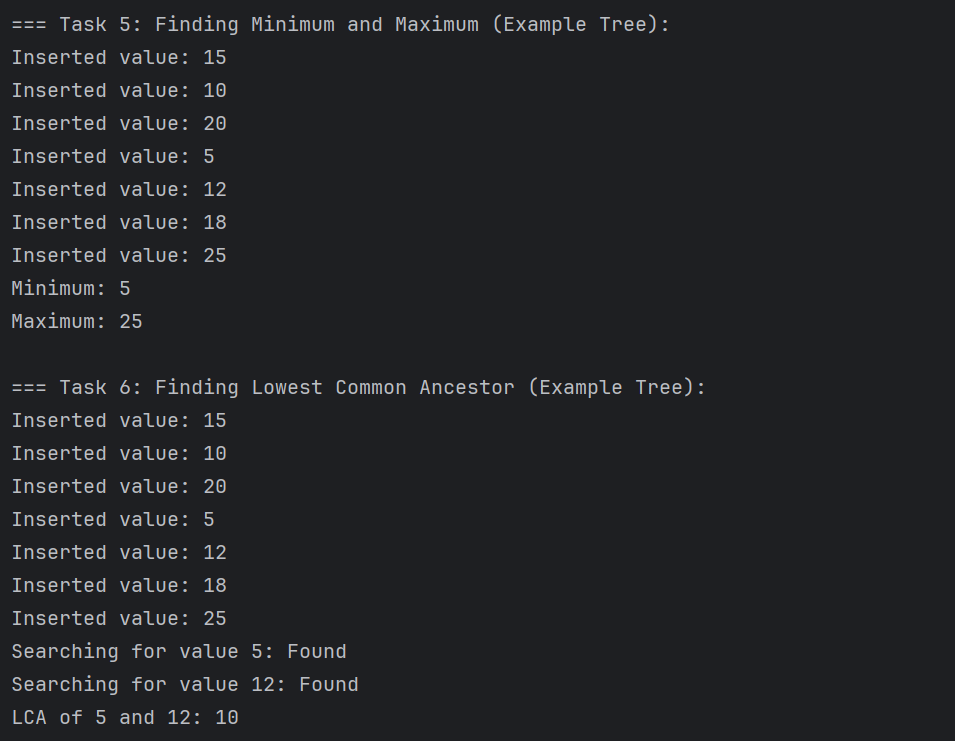
Java Code:

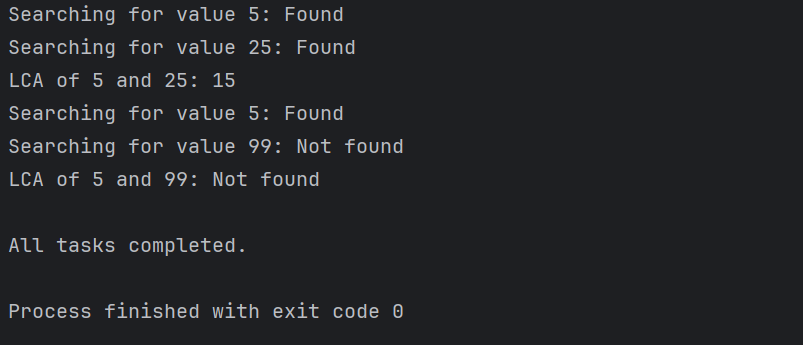
class Node {  
 int key;  
 Node left, right;  
  
 Node(int key) {  
 this.key = key;  
 this.left = null;  
 this.right = null;  
 }  
}  
  
public class BinarySearchTree {  
 private Node root;  
  
 // Constructor  
 public BinarySearchTree() {  
 root = null;  
 }  
  
 // Task 2: Recursive insertNode  
 public void insertNode(int key) {  
 root = insertHelper(root, key);  
 System.*out*.println("Inserted value: " + key);  
 }  
  
 private Node insertHelper(Node node, int key) {  
 if (node == null) {  
 return new Node(key);  
 }  
 if (key < node.key) {  
 node.left = insertHelper(node.left, key);  
 } else if (key > node.key) {  
 node.right = insertHelper(node.right, key);  
 }  
 return node;  
 }  
  
 // Task 1 & 4: Search and Delete Node  
 public boolean search(int key) {  
 boolean found = searchHelper(root, key);  
 System.*out*.println("Searching for value " + key + ": " + (found ? "Found" : "Not found"));  
 return found;  
 }  
  
 private boolean searchHelper(Node node, int key) {  
 if (node == null) {  
 return false;  
 }  
 if (key == node.key) {  
 return true;  
 }  
 if (key < node.key) {  
 return searchHelper(node.left, key);  
 }  
 return searchHelper(node.right, key);  
 }  
  
 public boolean deleteNode(int key) {  
 Node[] result = new Node[1];  
 root = deleteHelper(root, key, result);  
 boolean deleted = result[0] != null;  
 System.*out*.println("Deleting value " + key + ": " + (deleted ? "Deleted" : "Not found"));  
 return deleted;  
 }  
  
 private Node deleteHelper(Node node, int key, Node[] deleted) {  
 if (node == null) {  
 deleted[0] = null;  
 return null;  
 }  
 if (key < node.key) {  
 node.left = deleteHelper(node.left, key, deleted);  
 } else if (key > node.key) {  
 node.right = deleteHelper(node.right, key, deleted);  
 } else {  
 deleted[0] = node;  
 if (node.left == null) {  
 return node.right;  
 } else if (node.right == null) {  
 return node.left;  
 }  
 Node temp = findMin(node.right);  
 node.key = temp.key;  
 node.right = deleteHelper(node.right, temp.key, new Node[1]);  
 }  
 return node;  
 }  
  
 private Node findMin(Node node) {  
 while (node.left != null) {  
 node = node.left;  
 }  
 return node;  
 }  
  
 // Task 3: Traversal Functions  
 public void preOrderTraversal() {  
 System.*out*.print("Pre-Order Traversal: ");  
 preOrderHelper(root);  
 System.*out*.println();  
 }  
  
 private void preOrderHelper(Node node) {  
 if (node == null) return;  
 System.*out*.print(node.key + " ");  
 preOrderHelper(node.left);  
 preOrderHelper(node.right);  
 }  
  
 public void inOrderTraversal() {  
 System.*out*.print("In-Order Traversal: ");  
 inOrderHelper(root);  
 System.*out*.println();  
 }  
  
 private void inOrderHelper(Node node) {  
 if (node == null) return;  
 inOrderHelper(node.left);  
 System.*out*.print(node.key + " ");  
 inOrderHelper(node.right);  
 }  
  
 public void postOrderTraversal() {  
 System.*out*.print("Post-Order Traversal: ");  
 postOrderHelper(root);  
 System.*out*.println();  
 }  
  
 private void postOrderHelper(Node node) {  
 if (node == null) return;  
 postOrderHelper(node.left);  
 postOrderHelper(node.right);  
 System.*out*.print(node.key + " ");  
 }  
  
 // Task 5: Find Minimum and Maximum  
 public void findMinMax() {  
 if (root == null) {  
 System.*out*.println("Tree is empty. No minimum or maximum.");  
 return;  
 }  
 int min = findMinValue(root);  
 int max = findMaxValue(root);  
 System.*out*.println("Minimum: " + min);  
 System.*out*.println("Maximum: " + max);  
 }  
  
 private int findMinValue(Node node) {  
 while (node.left != null) {  
 node = node.left;  
 }  
 return node.key;  
 }  
  
 private int findMaxValue(Node node) {  
 while (node.right != null) {  
 node = node.right;  
 }  
 return node.key;  
 }  
  
 // Task 6: Find Lowest Common Ancestor (LCA)  
 public Integer findLCA(int n1, int n2) {  
 Integer lca = findLCAHelper(root, n1, n2);  
 System.*out*.println("LCA of " + n1 + " and " + n2 + ": " + (lca != null ? lca : "Not found"));  
 return lca;  
 }  
  
 private Integer findLCAHelper(Node node, int n1, int n2) {  
 if (node == null) return null;  
  
 if (node.key > n1 && node.key > n2) {  
 return findLCAHelper(node.left, n1, n2);  
 }  
 if (node.key < n1 && node.key < n2) {  
 return findLCAHelper(node.right, n1, n2);  
 }  
  
 if (search(n1) && search(n2)) {  
 return node.key;  
 }  
 return null;  
 }  
  
 // Main method with direct method calls for all tasks  
 public static void main(String[] args) {  
 // Task 1 & 2: Build the tree from Task 2  
 System.*out*.println("=== Task 1 & 2: Building the BST ");  
 BinarySearchTree bst1 = new BinarySearchTree();  
 System.*out*.println("Inserting nodes into the BST (Tree from Task 2):");  
 bst1.insertNode(60); // Root  
 bst1.insertNode(50);  
 bst1.insertNode(70);  
 bst1.insertNode(30);  
 bst1.insertNode(53);  
 bst1.insertNode(80);  
 bst1.insertNode(35);  
 bst1.insertNode(57);  
 bst1.insertNode(75);  
 bst1.insertNode(32);  
 bst1.insertNode(40);  
 bst1.insertNode(77);  
 bst1.insertNode(48);  
 bst1.insertNode(45);  
 System.*out*.println();  
  
 // Task 1: Search for some values  
 System.*out*.println("=== Task 1: Searching in the BST:");  
 bst1.search(40); // Should be found  
 bst1.search(90); // Should not be found  
 System.*out*.println();  
  
 // Task 3: Display traversals  
 System.*out*.println("=== Task 3: Displaying Traversals:");  
 bst1.preOrderTraversal();  
 bst1.inOrderTraversal();  
 bst1.postOrderTraversal();  
 System.*out*.println();  
  
 // Task 4: Delete nodes  
 System.*out*.println("=== Task 4: Deleting Nodes:");  
 bst1.deleteNode(40); // Node with two children  
 bst1.deleteNode(90); // Non-existent node  
 bst1.deleteNode(53); // Node with one child  
 bst1.deleteNode(30); // Node with two children  
 System.*out*.println("After deletions, displaying updated tree:");  
 bst1.inOrderTraversal();  
 System.*out*.println();  
  
 // Task 5: Find Minimum and Maximum (Using Task 2 tree after deletions)  
 System.*out*.println("=== Task 5: Finding Minimum and Maximum (Task 2 Tree after deletions):");  
 bst1.findMinMax();  
 System.*out*.println();  
  
 // Task 5: Test with example tree from Task 5  
 System.*out*.println("=== Task 5: Finding Minimum and Maximum (Example Tree):");  
 BinarySearchTree bst2 = new BinarySearchTree();  
 bst2.insertNode(15);  
 bst2.insertNode(10);  
 bst2.insertNode(20);  
 bst2.insertNode(5);  
 bst2.insertNode(12);  
 bst2.insertNode(18);  
 bst2.insertNode(25);  
 bst2.findMinMax();  
 System.*out*.println();  
  
 // Task 6: Find LCA (Using example tree from Task 6)  
 System.*out*.println("=== Task 6: Finding Lowest Common Ancestor (Example Tree):");  
 BinarySearchTree bst3 = new BinarySearchTree();  
 bst3.insertNode(15);  
 bst3.insertNode(10);  
 bst3.insertNode(20);  
 bst3.insertNode(5);  
 bst3.insertNode(12);  
 bst3.insertNode(18);  
 bst3.insertNode(25);  
 bst3.findLCA(5, 12); // Should be 10  
 bst3.findLCA(5, 25); // Should be 15  
 bst3.findLCA(5, 99); // Should not be found  
 System.*out*.println();  
  
 System.*out*.println("All tasks completed.");  
 }  
}

Output Screenshots:









**Note: I have written all tasks in a single java file**

***~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ THE END ~~~~~~~~~~~~~~~~~~~~~~~~~~~~***