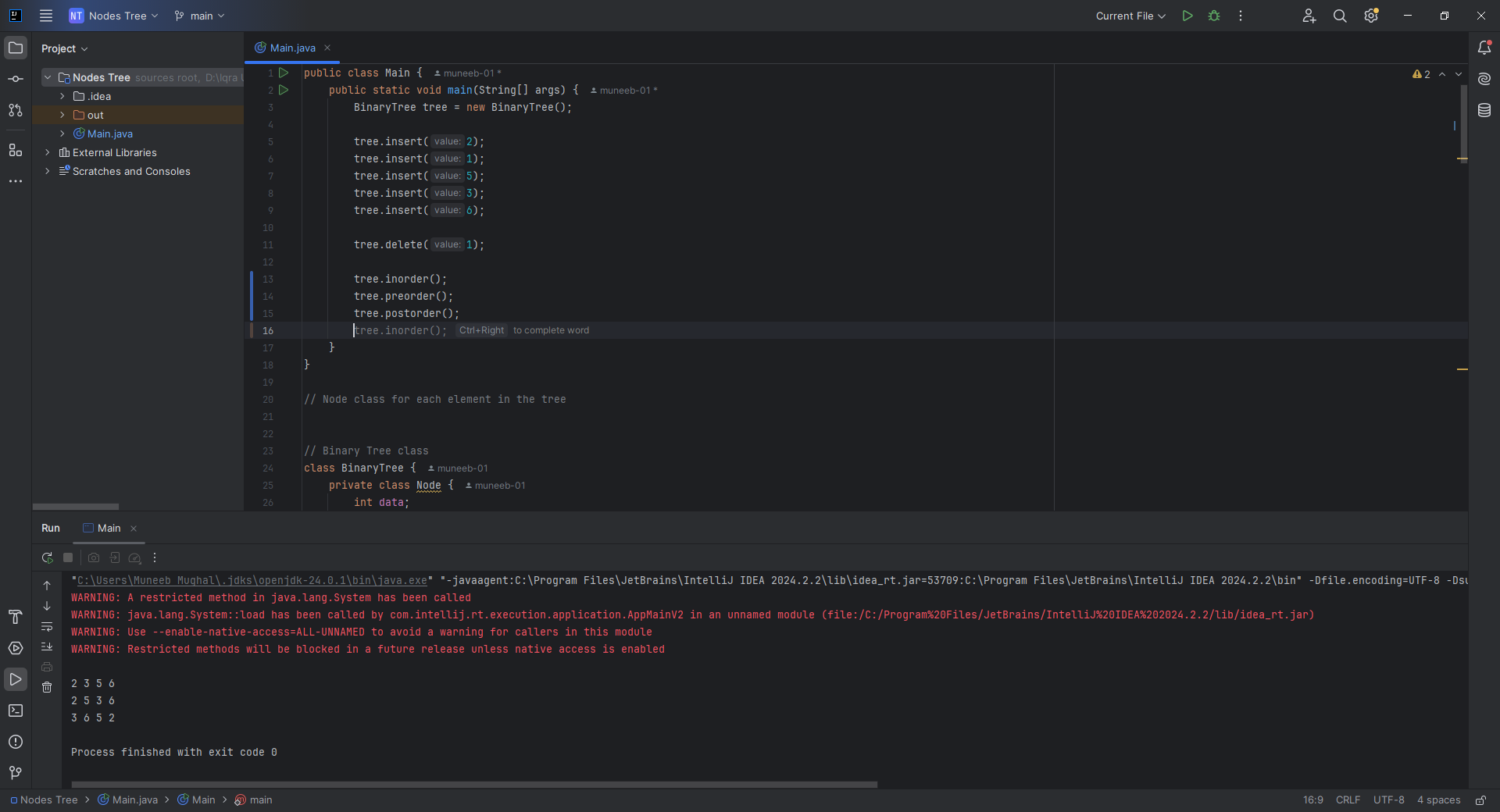
**CLASS TASK All:**

**CODE:**

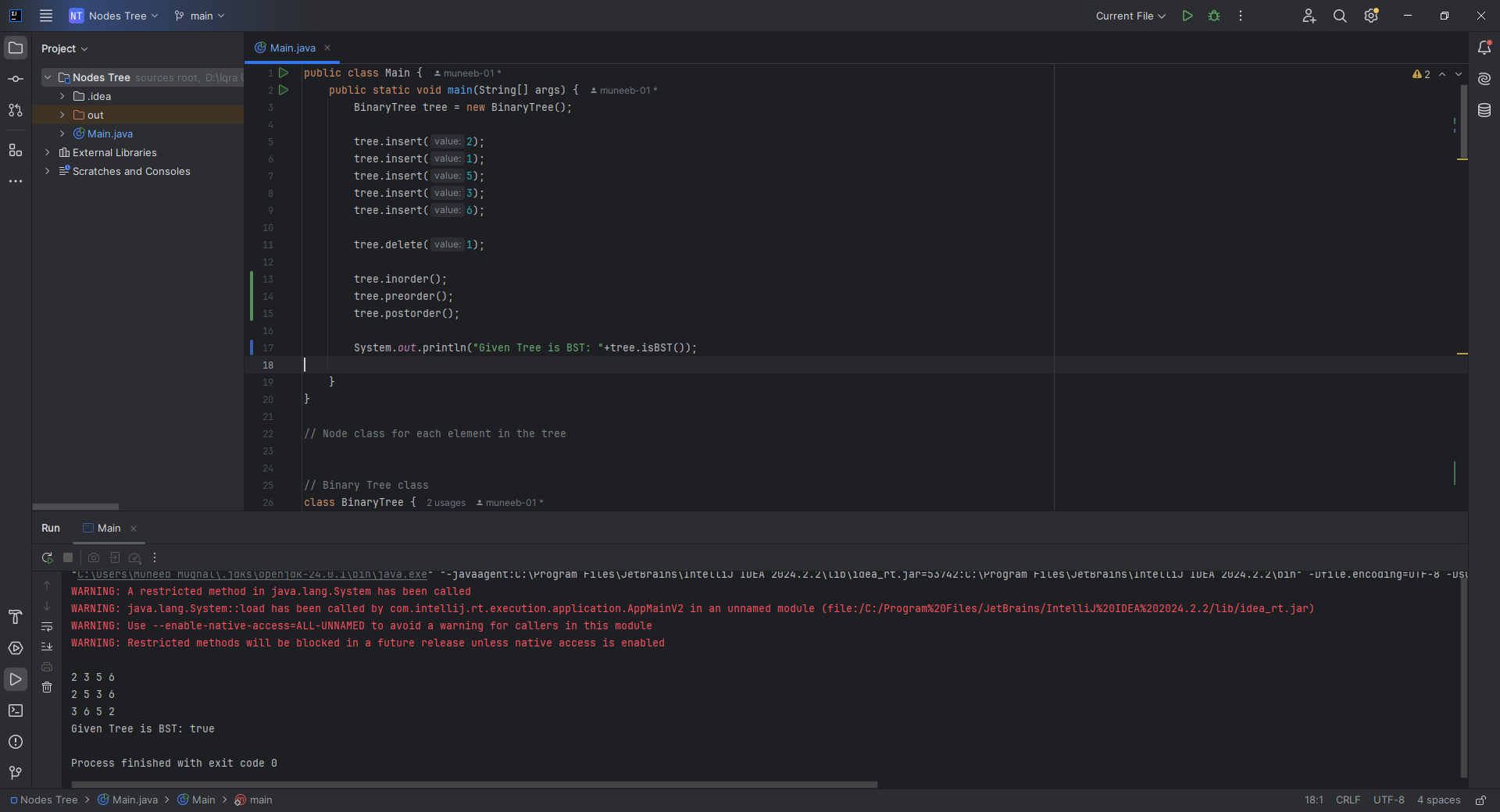
public class Main {  
 public static void main(String[] args) {  
 BinaryTree tree = new BinaryTree();  
  
 tree.insert(2);  
 tree.insert(1);  
 tree.insert(5);  
 tree.insert(3);  
 tree.insert(6);  
  
 tree.delete(1);  
 }  
}  
  
// Node class for each element in the tree  
  
  
// Binary Tree class  
class BinaryTree {  
 private class Node {  
 int data;  
 Node left, right;  
  
 Node(int value) {  
 data = value;  
 left = right = null;  
 }  
 }  
 Node root;  
  
 void insert(int value) {  
 root = insertRec(root, value);  
 }  
  
 // Recursive BST insert logic  
 Node insertRec(Node root, int value) {  
 if (root == null) {  
 return new Node(value);  
 }  
 if (value < root.data)  
 root.left = insertRec(root.left, value);  
 else if (value > root.data)  
 root.right = insertRec(root.right, value);  
 return root;  
 }  
  
 // Inorder traversal: Left → Root → Right  
 void inorder() {  
 inorderRec(root);  
 System.*out*.println();  
 }  
  
 void inorderRec(Node node) {  
 if (node != null) {  
 inorderRec(node.left);  
 System.*out*.print(node.data + " ");  
 inorderRec(node.right);  
 }  
 }  
  
 // Preorder traversal: Root → Left → Right  
 void preorder() {  
 preorderRec(root);  
 System.*out*.println();  
 }  
  
 void preorderRec(Node node) {  
 if (node != null) {  
 System.*out*.print(node.data + " ");  
 preorderRec(node.left);  
 preorderRec(node.right);  
 }  
 }  
  
 // Postorder traversal: Left → Right → Root  
 void postorder() {  
 postorderRec(root);  
 System.*out*.println();  
 }  
  
 void postorderRec(Node node) {  
 if (node != null) {  
 postorderRec(node.left);  
 postorderRec(node.right);  
 System.*out*.print(node.data + " ");  
 }  
 }  
 void delete(int value) {  
 root = deleteRec(root, value);  
 }  
  
 Node deleteRec(Node root, int value) {  
 if (root == null) return null;  
  
 if (value < root.data) {  
 root.left = deleteRec(root.left, value);  
 } else if (value > root.data) {  
 root.right = deleteRec(root.right, value);  
 } else {  
 // Node to be deleted found  
  
 // Case 1: No child  
 if (root.left == null && root.right == null) {  
 return null;  
 }  
  
 // Case 2: One child  
 if (root.left == null) return root.right;  
 if (root.right == null) return root.left;  
  
 // Case 3: Two children  
 Node successor = findMin(root.right);  
 root.data = successor.data;  
 root.right = deleteRec(root.right, successor.data);  
 }  
  
 return root;  
 }  
  
 Node findMin(Node node) {  
 while (node.left != null) {  
 node = node.left;  
 }  
 return node;  
 }  
  
}



**Home Task 1:** Checking if a Tree is a BST

**CODE:**

boolean isBST() {  
 return isBSTUtil(root, Integer.*MIN\_VALUE*, Integer.*MAX\_VALUE*);  
}  
  
private boolean isBSTUtil(Node node, int min, int max) {  
 if (node == null) return true;  
 if (node.data <= min || node.data >= max) return false;  
 return isBSTUtil(node.left, min, node.data) && isBSTUtil(node.right, node.data, max);  
}

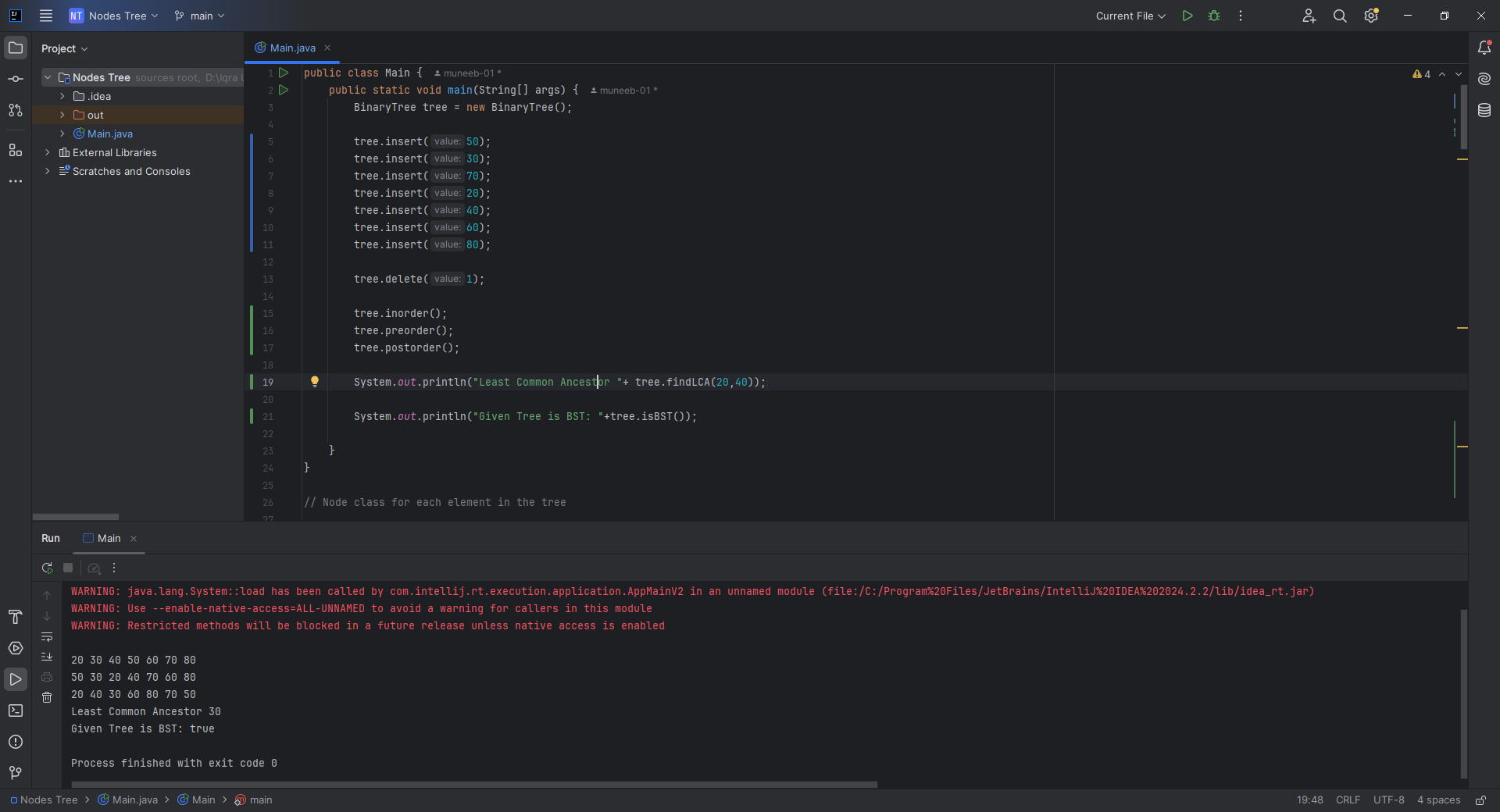


**Home Task 2:** Finding the Lowest Common Ancestor

**CODE:**

int findLCA(int n1, int n2) {  
 Node lcaNode = findLCARec(root, n1, n2);  
 return lcaNode != null ? lcaNode.data : -1; // Return -1 if no LCA found  
}  
  
private Node findLCARec(Node node, int n1, int n2) {  
 if (node == null) return null;  
  
 // If both n1 and n2 are smaller, LCA lies in left  
 if (n1 < node.data && n2 < node.data) {  
 return findLCARec(node.left, n1, n2);  
 }  
  
 // If both n1 and n2 are greater, LCA lies in right  
 if (n1 > node.data && n2 > node.data) {  
 return findLCARec(node.right, n1, n2);  
 }  
  
 // Otherwise, this node is the LCA  
 return node;

}



**Home Task 3: BST TO DLL**

**CODE:**

class BinaryTree {  
 public class Node { // <-- make public here  
 int data;  
 Node left, right;  
  
 Node(int value) {  
 data = value;  
 left = right = null;  
 }  
 }  
 Node root;  
  
 private Node head = null;  
 private Node prev = null;  
  
 public Node bstToDoublyLinkedList() {  
 head = null;  
 prev = null;  
 convertBSTtoDLL(root);  
 return head;  
 }  
  
 private void convertBSTtoDLL(Node curr) {  
 if (curr == null) return;  
  
 convertBSTtoDLL(curr.left);  
  
 if (prev == null) {  
 head = curr;  
 } else {  
 prev.right = curr;  
 curr.left = prev;  
 }  
 prev = curr;  
  
 convertBSTtoDLL(curr.right);  
 }  
  
 public void printDoublyLinkedList(Node head) {  
 Node curr = head;  
 while (curr != null) {  
 System.*out*.print(curr.data);  
 if (curr.right != null) System.*out*.print(" <-> ");  
 curr = curr.right;  
 }  
 System.*out*.println();  
 }  
  
 public void insert(int val) {  
 root = insertRec(root, val);  
 }  
  
 private Node insertRec(Node node, int val) {  
 if (node == null) return new Node(val);  
 if (val < node.data) node.left = insertRec(node.left, val);  
 else if (val > node.data) node.right = insertRec(node.right, val);  
 return node;  
 }  
}  
  
public class Main {  
 public static void main(String[] args) {  
 BinaryTree tree = new BinaryTree();  
 int[] values = {10, 5, 15, 2, 8, 12, 20};  
 for (int v : values) {  
 tree.insert(v);  
 }  
  
 BinaryTree.Node head = tree.bstToDoublyLinkedList();  
 tree.printDoublyLinkedList(head);  
 }  
}

