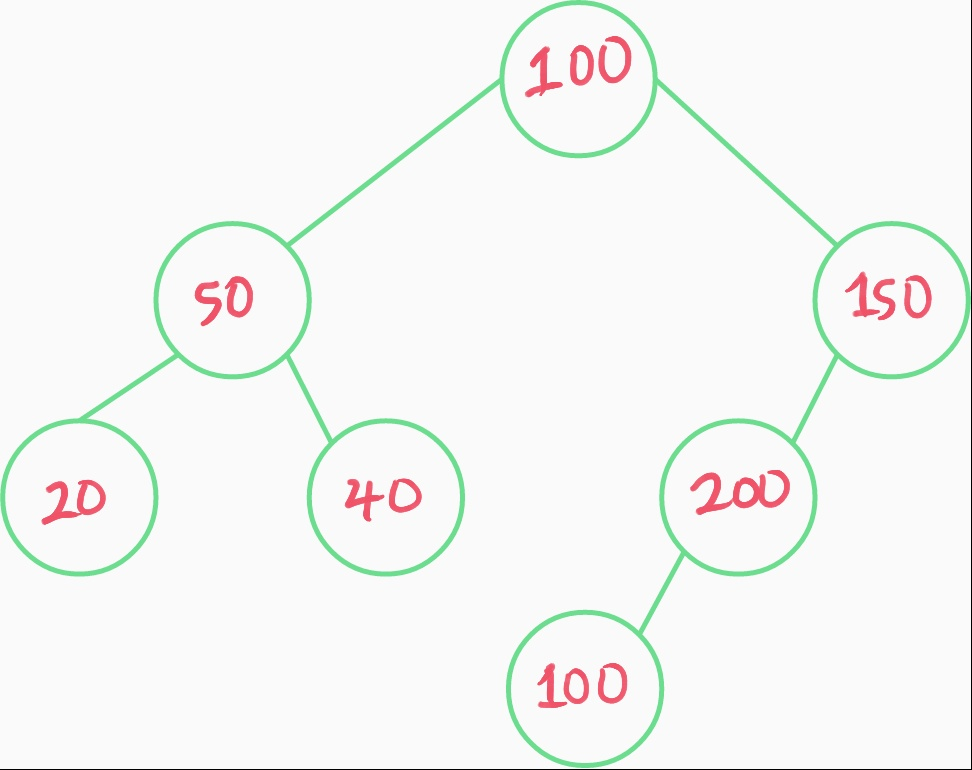
CS-146 Homework-4 Due: June-28 at 11:59 PM

1. Write a program to implement a generic binary search tree. This class has a private member variable called root.
2. Implement a public method for the above class that returns the height of the binary tree.
3. Write a private method in BinarySearch called passedTest. This method accepts a BinaryNode and determines if the difference of height of the right subtree and left subtree is greater than 1. If it is greater than 1, it should return false and otherwise it should return true.
4. Write a private method called allPassed in BinarySearchTree that returns true if all the nodes are satisfying the passedTest condition as defined in problem-3. Note that you may traverse the tree using any technique that you like.
5. Implement the function printInOrder.
6. Implement a method that returns a String that contains the PostOrder traversal of the binary search tree.
7. Implement a public method that return the number of nodes in the binary tree.
8. A given post order traversal is shown below. What will be the pre-order traversal?

Post-order: 20, 40, 50, 100, 200, 150, 100

**Pre-order: 100, 50, 20, 40, 150, 200, 100**

Binary Tree Diagram

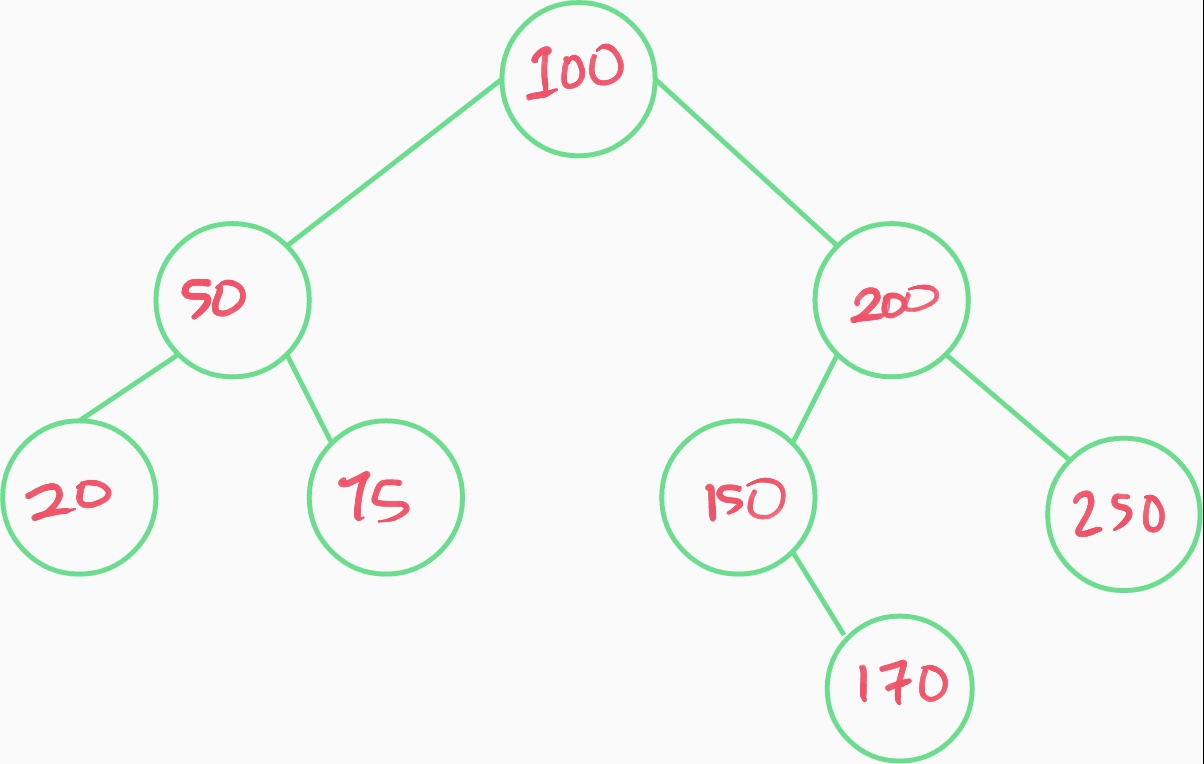


1. The pre-order traversal of a binary search tree is given below. Show the post order traversal.

Pre-order: 100, 50, 20, 75, 200, 150, 170, 250

**Post-order: 20, 75, 50, 170, 150, 250, 200, 100**

Binary Search Tree Diagram



1. Use the BST program and test the code. Show the screen shot of the post order traversal and pre-order traversals.
2. Given the following numbers in the given order, show the red black tree. Show the steps as you flip colors or do any rotations.

100, 200, 150, 170, 165, 180, 220, 163, 164

Link To Image: [GoogleDrive](https://drive.google.com/file/d/1KqEJQrYbOnpkhlq4TilxKGutpw7u0vHV/view?usp=sharing)

\*Hi Professor, I tried embedding the image here, but it was barely visible so please check out the link to the image and let me know if there’s any problem. Thank you!

1. Show the pre-order traversal of this red black tree while showing the color of each node in the pre-order traversal.
2. Write the red black tree code and insert the above numbers. Show the screen shot of the pre-order traversal of the resulting tree. Distinguish the colors by writing a \* next to the black color values. Compare the result with the previous question.
3. Given the following numbers in the given order, show the AVL tree. Show the steps as you do any rotations.

100, 200, 150, 170, 165, 180, 220, 163, 164

Google Drive:

<https://drive.google.com/file/d/12zNI-ee550aDX3_yakaV5NEaGm1q3ayp/view?usp=sharing>

1. Show the pre-order traversal of this AVL tree.
2. Write the AVL tree code and insert the above numbers. Show the screen shot of the pre-order traversal of the resulting tree. Compare the result with the previous question.

Source Code (for clearer viewing, I have it provided under my repository at <https://github.com/wen1225/CS146-HW4.git>

)

RBT

package Tree;  
  
public class RedBlackTree <T extends Comparable<T>> {  
 private RedBlackNode<T> root;  
 private RedBlackNode<T> nil;  
 private enum COLOR { **BLACK**, **RED**; }  
 private static class RedBlackNode<T extends Comparable<T>> {  
 //Member Variables  
 T key;  
 COLOR color;  
 RedBlackNode<T> parent;  
 RedBlackNode<T> left;  
 RedBlackNode<T> right;  
 }  
 public RedBlackTree() {  
 nil = new RedBlackNode<>();  
 //nil node is always black  
 nil.color = COLOR.**BLACK**;  
 nil.left = null;  
 nil.right = null;  
 nil.parent = null;  
  
 /\* why this way doesn't work? but root = nil does? hmmm  
 root = new RedBlackNode<>();  
 root.color = COLOR.BLACK;  
 //when root node is first created, it has no child, so point both to nil  
 root.left = nil;  
 root.right = nil;  
 //parent of root node is always nil node  
 root.parent = nil;  
 \*/  
 root = nil;  
  
 }  
 public void printTree() {  
 print(this.root, "", true);  
 }  
 private void print(RedBlackNode<T> root, String indent, boolean last) {  
 if (root != nil) {  
 System.**out**.print(indent);  
 if (last) {  
 if (root.parent == null) {  
 System.**out**.print("ROOT-");  
 }  
 else {  
 System.**out**.print("R----");  
 }  
 indent += " ";  
 }  
 else {  
 System.**out**.print("L----");  
 indent += "| ";  
 }  
 String c = (root.color == COLOR.**RED**) ? "RED" : "BLACK";  
 System.**out**.println(root.key + "(" + c + ")");  
 print(root.left, indent, false);  
 print(root.right, indent, true);  
 }  
 }  
 private void rotateLeft(RedBlackNode<T> x) {  
 RedBlackNode<T> y = x.right;  
 x.right = y.left;  
 if (y.left != nil) {  
 y.left.parent = x;  
 }  
 y.parent = x.parent;  
 if (x.parent == null) {  
 this.root = y;  
 }  
 else if (x == x.parent.left) {  
 x.parent.left = y;  
 }  
 else {  
 x.parent.right = y;  
 }  
 y.left = x;  
 x.parent = y;  
 }  
 private void rotateRight(RedBlackNode<T> x) {  
 RedBlackNode<T> y = x.left;  
 x.left = y.right;  
 if (y.right != nil) {  
 y.right.parent = x;  
 }  
 y.parent = x.parent;  
 if (x.parent == null) {  
 this.root = y;  
 }  
 else if (x == x.parent.right) {  
 x.parent.right = y;  
 }  
 else {  
 x.parent.left = y;  
 }  
 y.right = x;  
 x.parent = y;  
 }  
 public void insert(T key) {  
 insertNode(key);  
 }  
 private void insertNode(T key) {  
 RedBlackNode<T> newNode = new RedBlackNode<>();  
 newNode.key = key;  
 //if insert is called, means newNode is not the root node, so its parent is not nil.  
 //"I guess I could still technically let it point to nil, but we'll see"  
 newNode.parent = null;  
 newNode.left = nil;  
 newNode.right = nil;  
 //new node insertion must always be red.  
 newNode.color = COLOR.**RED**;  
  
 //Create temporary node pointers  
 RedBlackNode<T> y = null;  
 //this.root refers to the caller's (our RBT object) root node. So, point x to the caller's root.  
 RedBlackNode<T> x = this.root;  
  
 while (x != nil) {  
 y = x;  
 //if newNode's key < root's key  
 if (newNode.key.compareTo(x.key) < 0) {  
 x = x.left;  
 }  
 else if (newNode.key.compareTo(x.key) > 0) {  
 x = x.right;  
 }  
 else {  
 //do nothing with duplicate keys  
 }  
 }  
  
 newNode.parent = y;  
 if (y == null) {  
 root = newNode;  
 }  
 else if (newNode.key.compareTo(y.key) < 0) {  
 y.left = newNode;  
 }  
 else {  
 y.right = newNode;  
 }  
  
 if (newNode.parent == null) {  
 newNode.color = COLOR.**BLACK**;  
 return;  
 }  
  
 //Fix possible violation  
 insertFixUp(newNode);  
 }  
 public void insertFixUp(RedBlackNode<T> currNode) {  
 RedBlackNode<T> tempNode;  
 while (currNode.parent.color == COLOR.**RED**) {  
 if (currNode.parent == currNode.parent.parent.right) {  
 tempNode = currNode.parent.parent.left;  
 if (tempNode.color == COLOR.**RED**) {  
 tempNode.color = COLOR.**BLACK**;  
 currNode.parent.color = COLOR.**BLACK**;  
 currNode.parent.parent.color = COLOR.**RED**;  
 currNode = currNode.parent.parent;  
 }  
 else {  
 if (currNode == currNode.parent.left) {  
 currNode = currNode.parent;  
 rotateRight(currNode);  
 }  
 currNode.parent.color = COLOR.**BLACK**;  
 currNode.parent.parent.color = COLOR.**RED**;  
 rotateLeft(currNode.parent.parent);  
 }  
 }  
 else {  
 tempNode = currNode.parent.parent.right;  
 if (tempNode.color == COLOR.**RED**) {  
 tempNode.color = COLOR.**BLACK**;  
 currNode.parent.color = COLOR.**BLACK**;  
 currNode.parent.parent.color = COLOR.**RED**;  
 currNode = currNode.parent.parent;  
 }  
 else {  
 if (currNode == currNode.right) {  
 currNode = currNode.parent;  
 rotateLeft(currNode);  
 }  
 currNode.parent.color = COLOR.**BLACK**;  
 currNode.parent.parent.color = COLOR.**RED**;  
 rotateRight(currNode.parent.parent);  
 }  
 }  
 if (currNode == root) {  
 break;  
 }  
 }  
 root.color = COLOR.**BLACK**;  
 }  
 private void transplant(RedBlackNode<T> u, RedBlackNode<T> v) {  
 if (u.parent == null) {  
 root = v;  
 }  
 else if (u == u.parent.left) {  
 u.parent.left = v;  
 }  
 else {  
 u.parent.right = v;  
 }  
 v.parent = u.parent;  
 }  
 public void delete(T key) {  
 deleteNode(this.root, key);  
 }  
 private void deleteNode(RedBlackNode<T> n, T key) {  
 RedBlackNode<T> z = nil;  
 RedBlackNode<T> x, y;  
 while (n != nil) {  
 if (n.key == key) {  
 z = n;  
 }  
 if (n.key.compareTo(key) <= 0) {  
 n = n.right;  
 }  
 else {  
 n = n.left;  
 }  
 }  
 if (z == nil) {  
 System.**out**.println("Node not found.");  
 return;  
 }  
 y = z;  
 COLOR yInitialColor = y.color;  
 if (z.left == nil) {  
 x = z.right;  
 transplant(z, z.right);  
 }  
 else if (z.right == nil) {  
 x = z.left;  
 transplant(z, z.left);  
 }  
 else {  
 y = min(z.right);  
 yInitialColor = y.color;  
 x = y.right;  
 if (y.parent == z) {  
 x.parent = y;  
 }  
 else {  
 transplant(y, y.right);  
 y.right = z.right;  
 y.right.parent = y;  
 }  
 transplant(z, y);  
 y.left = z.left;  
 y.left.parent = y;  
 y.color = z.color;  
 }  
 if (yInitialColor == COLOR.**BLACK**)  
 deleteFixUp(x);  
 }  
 //"I dont get why we need to pass n tho, why can't we just call it w/ no arg?"  
 public RedBlackNode<T> min(RedBlackNode<T> n) {  
 while (n.left != nil)  
 n = n.left;  
 return n;  
 }  
 public void deleteFixUp(RedBlackNode<T> x) {  
 RedBlackNode<T> s;  
 while (x != root && x.color == COLOR.**BLACK**) {  
 if (x == x.parent.left) {  
 s = x.parent.left;  
 if (s.color == COLOR.**RED**) {  
 s.color = COLOR.**BLACK**;  
 x.parent.color = COLOR.**RED**;  
 rotateLeft(x.parent);  
 s = x.parent.right;  
 }  
 if (s.left.color == COLOR.**BLACK** && s.right.color == COLOR.**BLACK**) {  
 s.color = COLOR.**RED**;  
 x = x.parent;  
 }  
 else {  
 if (s.right.color == COLOR.**BLACK**) {  
 s.left.color = COLOR.**BLACK**;  
 s.color = COLOR.**RED**;  
 rotateRight(s);  
 s = x.parent.right;  
 }  
 s.color = x.parent.color;  
 x.parent.color = COLOR.**BLACK**;  
 s.right.color = COLOR.**BLACK**;  
 rotateLeft(x.parent);  
 x = root;  
 }  
 }  
 else {  
 s = x.parent.left;  
 if (s.color == COLOR.**RED**) {  
 s.color = COLOR.**BLACK**;  
 x.parent.color = COLOR.**RED**;  
 rotateRight(x.parent);  
 s = x.parent.left;  
 }  
 if (s.left.color == COLOR.**BLACK** && s.right.color == COLOR.**BLACK**) {  
 s.color = COLOR.**RED**;  
 x = x.parent;  
 }  
 else {  
 if (s.left.color == COLOR.**BLACK**) {  
 s.right.color = COLOR.**BLACK**;  
 s.color = COLOR.**RED**;  
 rotateLeft(s);  
 s = x.parent.left;  
 }  
 s.color = x.parent.color;  
 x.parent.color = COLOR.**BLACK**;  
 s.left.color = COLOR.**BLACK**;  
 rotateRight(x.parent);  
 x = root;  
 }  
 }  
 }  
 x.color = COLOR.**BLACK**;  
 }  
 public void printPreOrder() {  
 preOrderTraversal(root);  
 System.**out**.println();  
 }  
 private void preOrderTraversal(RedBlackNode<T> root) {  
 if (root == null)  
 return;  
 else {  
 System.**out**.println(root.key + " " + root.color);  
 if (root.left != nil) {  
 preOrderTraversal(root.left);  
 }  
 if (root.right != nil) {  
 preOrderTraversal(root.right);  
 }  
 }  
 }  
 public static void main(String[] args) {  
 RedBlackTree<Integer> tree = new RedBlackTree<>();  
 tree.insert(100);  
 tree.insert(200);  
 tree.insert(150);  
 tree.insert(170);  
 tree.insert(165);  
 tree.insert(180);  
 tree.insert(220);  
 tree.insert(163);  
 tree.insert(164);  
  
 //tree.printTree();  
 tree.printPreOrder();  
 }  
}

BST

/\*  
\* Author: Wen Yao Ho  
\* Class: CS146, HW4  
\* Professor: Faramarz Mortezaie  
\*/  
  
package Tree;  
  
public class BinarySearchTree<T extends Comparable<? super T>> {  
 private BinaryNode<T> root;  
  
 /\* PUBLIC METHODS \*/  
 public BinarySearchTree() { root = null; }  
 public void add(T value) {  
 root = insert(root, value);  
 }  
 public BinaryNode<T> search(BinaryNode<T> node, T value) {  
 if (node == null || node.value.compareTo(value) == 0)  
 return node;  
 else {  
 if (value.compareTo(node.value) < 0)  
 return search(node.leftNode, value);  
 else  
 return search(node.rightNode, value);  
 }  
 }  
 public int treeHeight() {  
 return height(root);  
 }  
 public void printInOrder() {  
 inOrderTraversal(root);  
 System.**out**.println();  
 }  
 public void printPostOrder() {  
 postOrderTraversal(root);  
 System.**out**.println();  
 }  
 public void printPreOrder() {  
 preOrderTraversal(root);  
 System.**out**.println();  
 }  
 public int numberOfNodes() {  
 //Any traversal method is fine because we have to count every node anyways.  
 //T(n) = O(n)  
 return countAllNodes(root);  
  
 }  
  
 /\* PRIVATE METHODS \*/  
 private BinaryNode<T> insert(BinaryNode<T> node, T value) {  
 if (node == null)  
 return new BinaryNode<>(value);  
 else {  
 if (value.compareTo(node.value) < 0) {  
 node.leftNode = insert(node.leftNode, value);  
 }  
 else if (value.compareTo(node.value) > 0) {  
 node.rightNode = insert(node.rightNode, value);  
 }  
 else {  
 //do nothing with duplicate values  
 }  
 }  
 return node;  
  
 }  
 private int height(BinaryNode<T> root) {  
 if (root == null)  
 return 0;  
 else {  
 //Each recursive call means height + 1. That's why we don't explicitly +1 when root is null  
 int leftHeight = height(root.leftNode) + 1;  
 int rightHeight = height(root.rightNode) + 1;  
 return Math.**max**(leftHeight, rightHeight);  
 }  
 }  
 private boolean passedTest(BinaryNode<T> node) {  
 if (node == null)  
 return false;  
 int leftHeight = height(node.leftNode);  
 int rightHeight = height(node.rightNode);  
 return Math.**abs**(leftHeight - rightHeight) <= 1;  
 }  
 private boolean allPassed(BinaryNode<T> node) {  
 boolean isLeftPass = passedTest(node.leftNode);  
 boolean isRightPass = passedTest(node.rightNode);  
 return isLeftPass == isRightPass;  
 }  
 private void inOrderTraversal(BinaryNode<T> root) {  
 if (root == null)  
 return;  
 else {  
 //If leftNode has child, go deeper to the left  
 if (root.hasLeftChild()) {  
 inOrderTraversal(root.leftNode);  
 }  
 //If here, means we're at the leaf and no more left child,  
 //so print the value at the leaf  
 System.**out**.print(root.value + " ");  
 //Then, check if leaf has right child, and repeat  
 if (root.hasRightChild()) {  
 inOrderTraversal(root.rightNode);  
 }  
 }  
 }  
 private void postOrderTraversal(BinaryNode<T> root) {  
 if (root == null)  
 return;  
 else {  
 if (root.hasLeftChild()) {  
 postOrderTraversal(root.leftNode);  
 }  
 if (root.hasRightChild()) {  
 postOrderTraversal(root.rightNode);  
 }  
 System.**out**.print(root.value + " ");  
 }  
 }  
 private void preOrderTraversal(BinaryNode<T> root) {  
 if (root == null)  
 return;  
 else {  
 System.**out**.print(root.value + " ");  
 if (root.hasLeftChild()) {  
 preOrderTraversal(root.leftNode);  
 }  
 if (root.hasRightChild()) {  
 preOrderTraversal(root.rightNode);  
 }  
 }  
 }  
 private int countAllNodes(BinaryNode<T> root) {  
 //Traversal method doesn't matter because we need to visit every node. O(n).  
 //I used post-order. "sum" is always 1, so sum always increment by 1 for each  
 //recursive call  
 if (root == null)  
 return 0;  
 else {  
 int sum = 1;  
 if (root.hasLeftChild()) {  
 sum += countAllNodes(root.leftNode);  
 }  
 if (root.hasRightChild()) {  
 sum += countAllNodes(root.rightNode);  
 }  
 return sum;  
 }  
 }  
  
 private static class BinaryNode<T extends Comparable<? super T>> {  
 T value;  
  
 BinaryNode<T> leftNode;  
 BinaryNode<T> rightNode;  
  
 BinaryNode() {}  
 BinaryNode(T value) {  
 this(value, null, null);  
 }  
 BinaryNode(T value, BinaryNode<T> leftNode, BinaryNode<T> rightNode) {  
 this.value = value;  
 this.leftNode = leftNode;  
 this.rightNode = rightNode;  
 }  
  
 boolean hasLeftChild() { return leftNode != null; }  
 boolean hasRightChild() { return rightNode != null; }  
 }  
 public static void main(String[] args) {  
 /\* Question 2  
 //Empty tree, T(1) = O(1) => height = 0  
 BinarySearchTree<Integer> root = new BinarySearchTree<>();  
 System.out.println(root.treeHeight());  
  
 //Right-skewed tree of height 5, T(n) = O(n) => height = 5  
 root.add(1);  
 root.add(2);  
 root.add(3);  
 root.add(4);  
 root.add(5);  
 System.out.println(root.treeHeight());  
 \*/  
  
 /\* Question 3  
 //Empty tree, T(1) = O(1) => height = 0  
 BinarySearchTree<Integer> root = new BinarySearchTree<>();  
 System.out.println(root.treeHeight());  
  
 //left tree height = 0, right tree height = 1. Height difference = 1  
 //expect true.  
 root.add(1);  
 root.add(2);  
  
 //left tree height = 0, right tree height = 2. Height difference = 2 > 1,  
 //expect false;  
 root.add(3);  
  
 System.out.println(root.passedTest(root.root));  
 \*/  
  
 /\* Question 4  
 //Empty tree, T(1) = O(1) => height = 0  
 BinarySearchTree<Integer> root = new BinarySearchTree<>();  
  
 //complete tree in the making...difference of height in each node is not > 1. So  
 // expect allPassed to return true  
 root.add(6);  
 root.add(3);  
 root.add(17);  
 root.add(2);  
 root.add(5);  
 root.add(10);  
 root.add(18);  
 root.add(1);  
  
 System.out.println(root.passedTest(root.root));  
 \*/  
  
 /\* Question 5  
 //Empty tree, T(1) = O(1) => height = 0  
 BinarySearchTree<Integer> root = new BinarySearchTree<>();  
  
 root.add(50);  
 root.add(12);  
 root.add(60);  
 root.add(7);  
 root.add(30);  
 root.add(55);  
 root.add(70);  
 root.add(4);  
 root.add(54);  
 root.add(58);  
 root.add(100);  
 root.add(3);  
 root.add(6);  
 root.add(2);  
 root.add(52);  
  
 //expect tree to be printed in sorted order so:  
 // 2 3 4 6 7 12 30 50 52 54 55 58 60 70 100  
 root.printInOrder();  
 \*/  
  
 /\*Question 6  
 //Empty tree, T(1) = O(1) => height = 0  
 BinarySearchTree<Integer> root = new BinarySearchTree<>();  
  
 root.add(50);  
 root.add(12);  
 root.add(60);  
 root.add(7);  
 root.add(30);  
 root.add(55);  
 root.add(70);  
 root.add(4);  
 root.add(54);  
 root.add(58);  
 root.add(100);  
 root.add(3);  
 root.add(6);  
 root.add(2);  
 root.add(52);  
  
 //expect tree to be printed such order:  
 // 2 3 6 4 7 30 12 52 54 58 55 100 70 60 50  
 root.printPostOrder();  
 \*/  
  
 /\* Question 7  
 //Empty tree, T(1) = O(1) => height = 0  
 BinarySearchTree<Integer> root = new BinarySearchTree<>();  
  
 root.add(50);  
 root.add(12);  
 root.add(60);  
 root.add(7);  
 root.add(30);  
 root.add(55);  
 root.add(70);  
 root.add(4);  
 root.add(54);  
 root.add(58);  
 root.add(100);  
 root.add(3);  
 root.add(6);  
 root.add(2);  
 root.add(52);  
  
 //expect number of nodes to be: 15  
 System.out.println(root.numberOfNodes());  
 \*/  
  
 // Question 10  
 BinarySearchTree<Integer> root = new BinarySearchTree<>();  
  
 root.add(100);  
 root.add(50);  
 root.add(20);  
 root.add(75);  
 root.add(200);  
 root.add(150);  
 root.add(170);  
 root.add(250);  
  
 root.printPostOrder();  
 root.printPreOrder();  
 }  
}