**Question 1 (chapter 25)**

**Write a program that uses a GUI to allows us ot manipulate a BST and has the following:**

* **label and input text box to enter an integer value.**
* **label and output text box to display result (list of tree nodes)**
* **button to insert**
* **button to delete**
* **button to show Inorder**
* **button to show Preorder**
* **Button to show Postorder.**

**Test your program on the following:**

**insert 12**

**insert 23**

**insert 56**

**insert 45**

**delete**

**insert 10**

**insert 13**

**insert 47**

**Inorder**

**Preorder**

**Postorder**

*Code*

*Tree Class*

//based on lecture slide chapter 25

package application;

import java.util.Collection;

public interface Tree<E> extends Collection<E> {

public boolean search(E e);

public boolean insert(E e);

public boolean delete(E e);

public int getSize();

public default void inorder() {

}

public default void postorder() {

}

public default void preorder() {

}

@Override

public default boolean isEmpty() {

return this.size() == 0;

}

@Override

public default boolean contains(Object e) {

return search((E)e);

}

@Override

public default boolean add(E e) {

return insert(e);

}

@Override

public default boolean remove(Object e) {

return delete((E)e);

}

@Override

public default int size() {

return getSize();

}

@Override

public default boolean containsAll(Collection<?> c) {

return false;

}

@Override

public default boolean addAll(Collection<? extends E> c) {

return false;

}

@Override

public default boolean removeAll(Collection<?> c) {

return false;

}

@Override

public default boolean retainAll(Collection<?> c) {

return false;

}

@Override

public default Object[] toArray() {

return null;

}

@Override

public default <T> T[] toArray(T[] array) {

return null;

}

}

*BST Class*

//based on lecture slide chapter 25

package application;

public class BST<E> implements Tree<E> {

protected TreeNode<E> root;

protected int size = 0;

protected java.util.Comparator<E> c;

private String output;

public BST() {

this.c = (e1, e2) -> ((Comparable<E>)e1).compareTo(e2);

}

public BST(java.util.Comparator<E> c) {

this.c = c;

}

public BST(E[] objects) {

this.c = (e1, e2) -> ((Comparable<E>)e1).compareTo(e2);

for (int i = 0; i < objects.length; i++)

add(objects[i]);

}

@Override

public boolean search(E e) {

TreeNode<E> current = root; // Start from the root

while (current != null) {

if (c.compare(e, current.element) < 0) {

current = current.left;

}

else if (c.compare(e, current.element) > 0) {

current = current.right;

}

else

return true; // Element is found

}

return false;

}

@Override

public boolean insert(E e) {

if (root == null)

root = createNewNode(e);

else {

TreeNode<E> parent = null;

TreeNode<E> current = root;

while (current != null)

if (c.compare(e, current.element) < 0) {

parent = current;

current = current.left;

}

else if (c.compare(e, current.element) > 0) {

parent = current;

current = current.right;

}

else

return false;

if (c.compare(e, parent.element) < 0)

parent.left = createNewNode(e);

else

parent.right = createNewNode(e);

}

size++;

return true;

}

public void setOutput(String s) {

output = s;

}

public String getOutput() {

return output;

}

protected TreeNode<E> createNewNode(E e) {

return new TreeNode<>(e);

}

@Override

public void inorder() {

output = " ";

inorder(root);

}

protected void inorder(TreeNode<E> root) {

if (root == null) return;

inorder(root.left);

System.out.print(root.element + " ");

output += root.element + " ";

inorder(root.right);

}

@Override

public void postorder() {

output = "";

postorder(root);

}

protected void postorder(TreeNode<E> root) {

if (root == null) return;

postorder(root.left);

postorder(root.right);

System.out.print(root.element + " ");

output += root.element + " ";

}

@Override

public void preorder() {

output = "";

preorder(root);

}

protected void preorder(TreeNode<E> root) {

if (root == null) return;

System.out.print(root.element + " ");

output += root.element + " ";

preorder(root.left);

preorder(root.right);

}

public static class TreeNode<E> {

protected E element;

protected TreeNode<E> left;

protected TreeNode<E> right;

public TreeNode(E e) {

element = e;

}

}

@Override

public int getSize() {

return size;

}

public TreeNode<E> getRoot() {

return root;

}

public java.util.ArrayList<TreeNode<E>> path(E e) {

java.util.ArrayList<TreeNode<E>> list =

new java.util.ArrayList<>();

TreeNode<E> current = root;

while (current != null) {

list.add(current);

if (c.compare(e, current.element) < 0) {

current = current.left;

}

else if (c.compare(e, current.element) > 0) {

current = current.right;

}

else

break;

}

return list;

}

@Override

public boolean delete(E e) {

TreeNode<E> parent = null;

TreeNode<E> current = root;

while (current != null) {

if (c.compare(e, current.element) < 0) {

parent = current;

current = current.left;

}

else if (c.compare(e, current.element) > 0) {

parent = current;

current = current.right;

}

else

break;

}

if (current == null)

return false;

if (current.left == null) {

if (parent == null) {

root = current.right;

}

else {

if (c.compare(e, parent.element) < 0)

parent.left = current.right;

else

parent.right = current.right;

}

}

else {

TreeNode<E> parentOfRightMost = current;

TreeNode<E> rightMost = current.left;

while (rightMost.right != null) {

parentOfRightMost = rightMost;

rightMost = rightMost.right;

}

current.element = rightMost.element;

if (parentOfRightMost.right == rightMost)

parentOfRightMost.right = rightMost.left;

else

parentOfRightMost.left = rightMost.left;

}

size--;

return true;

}

@Override

public java.util.Iterator<E> iterator() {

return new InorderIterator();

}

private class InorderIterator implements java.util.Iterator<E> {

private java.util.ArrayList<E> list =

new java.util.ArrayList<>();

private int current = 0;

public InorderIterator() {

inorder();

}

private void inorder() {

inorder(root);

}

private void inorder(TreeNode<E> root) {

if (root == null) return;

inorder(root.left);

list.add(root.element);

inorder(root.right);

}

@Override

public boolean hasNext() {

if (current < list.size())

return true;

return false;

}

@Override

public E next() {

return list.get(current++);

}

@Override

public void remove() {

if (current == 0)

throw new IllegalStateException();

delete(list.get(--current));

list.clear();

inorder();

}

}

@Override

public void clear() {

root = null;

size = 0;

}

}

*BSTPane Class (Java Fx )*

*//based on lecture slide chapter 25*

*package application;*

*import javafx.application.Application;*

*import javafx.event.ActionEvent;*

*import javafx.geometry.Pos;*

*import javafx.stage.Stage;*

*import javafx.scene.Scene;*

*import javafx.scene.control.Button;*

*import javafx.scene.control.Label;*

*import javafx.scene.control.TextField;*

*import javafx.scene.layout.BorderPane;*

*import javafx.scene.layout.HBox;*

*public class BSTPane extends Application {*

*private BST<Integer> tree = new BST<>();*

*private Label lbOutput = new Label();*

*private TextField tfRes;*

*private Label lbInput = new Label ();*

*private TextField tfInput;*

*private Button btInsert;*

*private Button btDelete ;*

*private Button btInorder;*

*private Button btPreorder;*

*private Button btPostorder;*

*private int value ;*

*@Override*

*public void start(Stage primaryStage) {*

*BorderPane pane = new BorderPane();*

*//Output*

*lbOutput = new Label("Output:");*

*tfRes = new TextField();*

*tfRes.setPrefColumnCount(35);*

*tfRes.setDisable(true);*

*HBox hBox2 = new HBox(5);*

*hBox2.getChildren().addAll(lbOutput,tfRes);*

*hBox2.setAlignment(Pos.CENTER);*

*pane.setBottom(hBox2);*

*//Input*

*lbInput = new Label("Input:");*

*tfInput = new TextField();*

*tfInput.setPrefColumnCount(3);*

*tfInput.setAlignment(Pos.BASELINE\_RIGHT);*

*HBox hBox1 = new HBox (3);*

*hBox1.getChildren().addAll(lbInput,*

*tfInput);*

*hBox1.setAlignment(Pos.CENTER);*

*pane.setTop(hBox1);*

*// Button*

*btInsert = new Button("Insert");*

*btDelete = new Button("Delete");*

*btInorder = new Button("Inorder");*

*btPreorder = new Button ("Preorder");*

*btPostorder = new Button ("Postorder");*

*HBox hBox = new HBox(8);*

*hBox.getChildren().addAll(btInsert, btDelete,btInorder,btPreorder,btPostorder );*

*hBox.setAlignment(Pos.CENTER);*

*pane.setCenter(hBox);*

*//Set on Action*

*btInsert.setOnAction(e -> insertValue (e));*

*btDelete.setOnAction(e -> delValue (e));*

*btInorder.setOnAction(e -> setInorder (e));*

*btPostorder.setOnAction(e -> setPost (e));*

*btPreorder.setOnAction(e -> setPreorder(e));*

*Scene scene = new Scene(pane, 550, 175);*

*primaryStage.setTitle("BST");*

*primaryStage.setScene(scene);*

*primaryStage.show();*

*}*

*private void setPost(ActionEvent e) {*

*tfRes.setText("");*

*System.out.println("\n");*

*tree.postorder();*

*tfRes.setText("Postorder: " + tree.getOutput());*

*}*

*public void displayTree() {*

*if (tree.getRoot() != null) {*

*displayTree(tree.getRoot());*

*}*

*}*

*private void displayTree(BST.TreeNode<Integer> root) {*

*tfRes.setText(" ");*

*String e1 = "";*

*for (int e : tree) {*

*e1 += String.valueOf(e + " ");*

*}*

*tfRes.setText(e1 + " ");*

*}*

*private void setPreorder(ActionEvent e) {*

*System.out.println("\n");*

*tree.preorder();*

*tfRes.setText("Preorder: " + tree.getOutput());*

*}*

*private void setInorder(ActionEvent e) {*

*System.out.println("\n");*

*tree.inorder();*

*tfRes.setText("Inorder: " +tree.getOutput());*

*}*

*private void delValue(ActionEvent e) {*

*value = Integer.parseInt(tfInput.getText());*

*if (!tree.search(value)) {*

*tfRes.setText(value + " is not in the tree");*

*}*

*else if (tree.search(value)){*

*tree.delete(value);*

*displayTree();*

*tfRes.setText(value + " is deleted in the tree");*

*}*

*}*

*private void insertValue(ActionEvent e) {*

*value = Integer.parseInt(tfInput.getText());*

*if(tree.search(value)) {*

*tfRes.setText(value + " is already in the tree");*

*}*

*else {*

*tree.insert(value);*

*displayTree();*

*}*

*}*

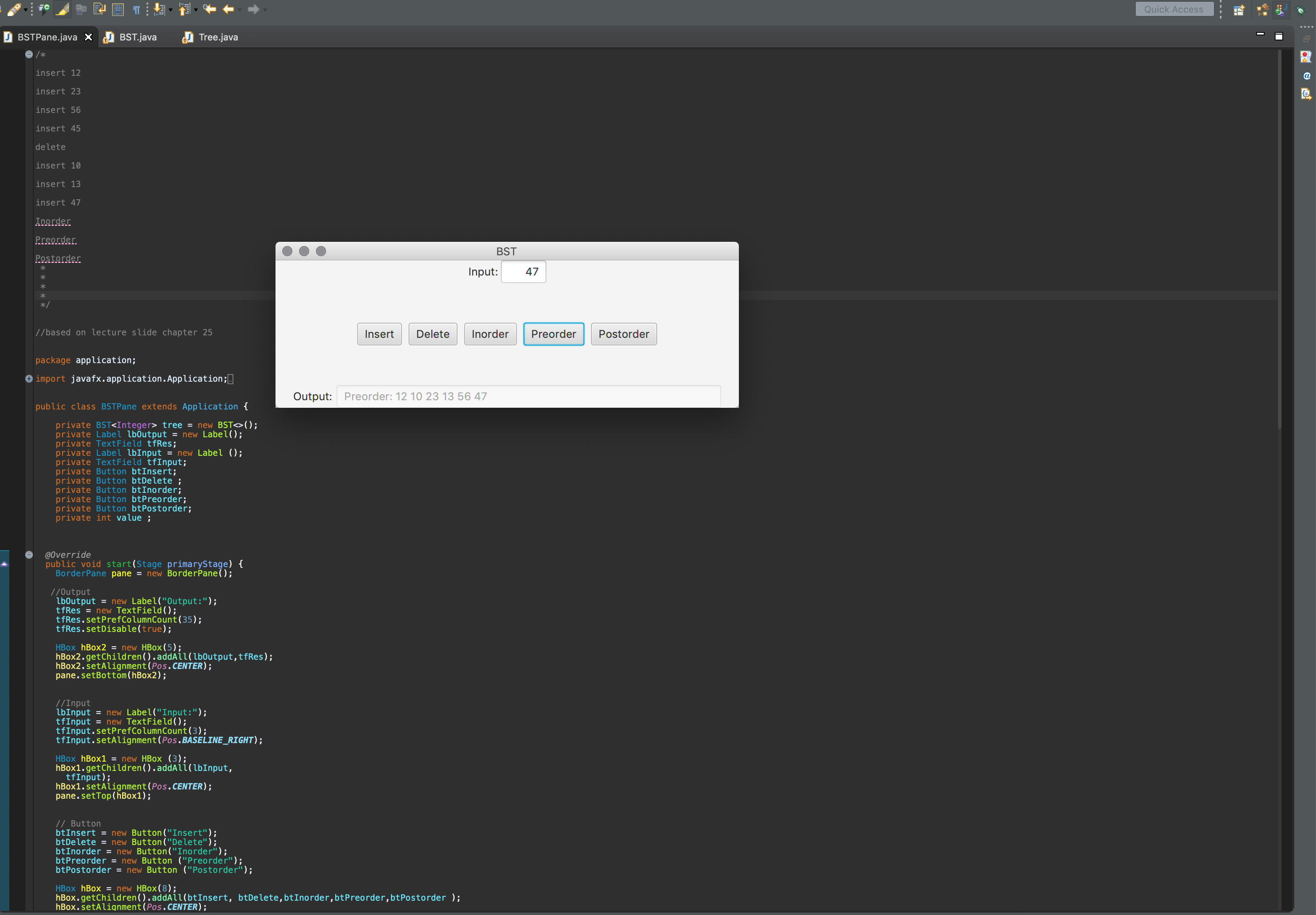
*public static void main(String[] args) {*

*launch(args);*

*}*

*}*

*Screenshots*

**

**Question 2( chapter 26)**

**Write a test program that randomly generates 100 numbers and inserts them in an AVL tree, search for a value, then reshuffles the numbers and then perform a search on the same value. Did the reshuffle change any thing?**

Based on the program output no changes were found.

*Code*

*AVL tree class*

//based on lecture slide 26

package application;

import application.BST.TreeNode;

public class AVLTree<E> extends BST<E> {

public AVLTree() {

}

public AVLTree(java.util.Comparator<E> c) {

super(c);

}

public AVLTree(E[] objects) {

super(objects);

}

@Override

protected AVLTreeNode<E> createNewNode(E e) {

return new AVLTreeNode<E>(e);

}

@Override

public boolean insert(E e) {

boolean successful = super.insert(e);

if (!successful)

return false;

else {

balancePath(e);

}

return true;

}

private void updateHeight(AVLTreeNode<E> node) {

if (node.left == null && node.right == null)

node.height = 0;

else if (node.left == null)

node.height = 1 + ((AVLTreeNode<E>)(node.right)).height;

else if (node.right == null)

node.height = 1 + ((AVLTreeNode<E>)(node.left)).height;

else

node.height = 1 +

Math.max(((AVLTreeNode<E>)(node.right)).height,

((AVLTreeNode<E>)(node.left)).height);

}

private void balancePath(E e) {

java.util.ArrayList<TreeNode<E>> path = path(e);

for (int i = path.size() - 1; i >= 0; i--) {

AVLTreeNode<E> A = (AVLTreeNode<E>)(path.get(i));

updateHeight(A);

AVLTreeNode<E> parentOfA = (A == root) ? null :

(AVLTreeNode<E>)(path.get(i - 1));

switch (balanceFactor(A)) {

case -2:

if (balanceFactor((AVLTreeNode<E>)A.left) <= 0) {

balanceLL(A, parentOfA);

}

else {

balanceLR(A, parentOfA);

}

break;

case +2:

if (balanceFactor((AVLTreeNode<E>)A.right) >= 0) {

balanceRR(A, parentOfA);

}

else {

balanceRL(A, parentOfA);

}

}

}

}

private int balanceFactor(AVLTreeNode<E> node) {

if (node.right == null)

return -node.height;

else if (node.left == null)

return +node.height;

else

return ((AVLTreeNode<E>)node.right).height -

((AVLTreeNode<E>)node.left).height;

}

private void balanceLL(TreeNode<E> A, TreeNode<E> parentOfA) {

TreeNode<E> B = A.left;

if (A == root) {

root = B;

}

else {

if (parentOfA.left == A) {

parentOfA.left = B;

}

else {

parentOfA.right = B;

}

}

A.left = B.right;

B.right = A;

updateHeight((AVLTreeNode<E>)A);

updateHeight((AVLTreeNode<E>)B);

}

private void balanceLR(TreeNode<E> A, TreeNode<E> parentOfA) {

TreeNode<E> B = A.left;

TreeNode<E> C = B.right;

if (A == root) {

root = C;

}

else {

if (parentOfA.left == A) {

parentOfA.left = C;

}

else {

parentOfA.right = C;

}

}

A.left = C.right;

B.right = C.left;

C.left = B;

C.right = A;

updateHeight((AVLTreeNode<E>)A);

updateHeight((AVLTreeNode<E>)B);

updateHeight((AVLTreeNode<E>)C);

}

private void balanceRR(TreeNode<E> A, TreeNode<E> parentOfA) {

TreeNode<E> B = A.right;

if (A == root) {

root = B;

}

else {

if (parentOfA.left == A) {

parentOfA.left = B;

}

else {

parentOfA.right = B;

}

}

A.right = B.left;

B.left = A;

updateHeight((AVLTreeNode<E>)A);

updateHeight((AVLTreeNode<E>)B);

}

private void balanceRL(TreeNode<E> A, TreeNode<E> parentOfA) {

TreeNode<E> B = A.right;

TreeNode<E> C = B.left;

if (A == root) {

root = C;

}

else {

if (parentOfA.left == A) {

parentOfA.left = C;

}

else {

parentOfA.right = C;

}

}

A.right = C.left;

B.left = C.right;

C.left = A;

C.right = B;

updateHeight((AVLTreeNode<E>)A);

updateHeight((AVLTreeNode<E>)B);

updateHeight((AVLTreeNode<E>)C);

}

@Override

public boolean delete(E element) {

if (root == null)

return false;

TreeNode<E> parent = null;

TreeNode<E> current = root;

while (current != null) {

if (c.compare(element, current.element) < 0) {

parent = current;

current = current.left;

}

else if (c.compare(element, current.element) > 0) {

parent = current;

current = current.right;

}

else

break;

}

if (current == null)

return false;

if (current.left == null) {

if (parent == null) {

root = current.right;

}

else {

if (c.compare(element, parent.element) < 0)

parent.left = current.right;

else

parent.right = current.right;

balancePath(parent.element);

}

}

else {

TreeNode<E> parentOfRightMost = current;

TreeNode<E> rightMost = current.left;

while (rightMost.right != null) {

parentOfRightMost = rightMost;

rightMost = rightMost.right;

}

current.element = rightMost.element;

if (parentOfRightMost.right == rightMost)

parentOfRightMost.right = rightMost.left;

else

parentOfRightMost.left = rightMost.left;

balancePath(parentOfRightMost.element);

}

size--;

return true;

}

@Override

public boolean search(E e) {

TreeNode<E> current = root;

while (current != null) {

if (c.compare(e, current.element) < 0) {

current = current.left;

}

else if (c.compare(e, current.element) > 0) {

current = current.right;

}

else

return true;

}

return false;

}

/\*\* AVLTreeNode is TreeNode plus height \*/

protected static class AVLTreeNode<E> extends BST.TreeNode<E> {

protected int height = 0; // New data field

public AVLTreeNode(E o) {

super(o);

}

}

}

*BST class*

*//based on lecture slide chapter 25*

*package application;*

*public class BST<E> implements Tree<E> {*

*protected TreeNode<E> root;*

*protected int size = 0;*

*protected java.util.Comparator<E> c;*

*private String output;*

*public BST() {*

*this.c = (e1, e2) -> ((Comparable<E>)e1).compareTo(e2);*

*}*

*public BST(java.util.Comparator<E> c) {*

*this.c = c;*

*}*

*public BST(E[] objects) {*

*this.c = (e1, e2) -> ((Comparable<E>)e1).compareTo(e2);*

*for (int i = 0; i < objects.length; i++)*

*add(objects[i]);*

*}*

*@Override*

*public boolean search(E e) {*

*TreeNode<E> current = root; // Start from the root*

*while (current != null) {*

*if (c.compare(e, current.element) < 0) {*

*current = current.left;*

*}*

*else if (c.compare(e, current.element) > 0) {*

*current = current.right;*

*}*

*else*

*return true; // Element is found*

*}*

*return false;*

*}*

*@Override*

*public boolean insert(E e) {*

*if (root == null)*

*root = createNewNode(e);*

*else {*

*TreeNode<E> parent = null;*

*TreeNode<E> current = root;*

*while (current != null)*

*if (c.compare(e, current.element) < 0) {*

*parent = current;*

*current = current.left;*

*}*

*else if (c.compare(e, current.element) > 0) {*

*parent = current;*

*current = current.right;*

*}*

*else*

*return false;*

*if (c.compare(e, parent.element) < 0)*

*parent.left = createNewNode(e);*

*else*

*parent.right = createNewNode(e);*

*}*

*size++;*

*return true;*

*}*

*public void setOutput(String s) {*

*output = s;*

*}*

*public String getOutput() {*

*return output;*

*}*

*protected TreeNode<E> createNewNode(E e) {*

*return new TreeNode<>(e);*

*}*

*@Override*

*public void inorder() {*

*output = " ";*

*inorder(root);*

*}*

*protected void inorder(TreeNode<E> root) {*

*if (root == null) return;*

*inorder(root.left);*

*System.out.print(root.element + " ");*

*output += root.element + " ";*

*inorder(root.right);*

*}*

*@Override*

*public void postorder() {*

*output = "";*

*postorder(root);*

*}*

*protected void postorder(TreeNode<E> root) {*

*if (root == null) return;*

*postorder(root.left);*

*postorder(root.right);*

*System.out.print(root.element + " ");*

*output += root.element + " ";*

*}*

*@Override*

*public void preorder() {*

*output = "";*

*preorder(root);*

*}*

*protected void preorder(TreeNode<E> root) {*

*if (root == null) return;*

*System.out.print(root.element + " ");*

*output += root.element + " ";*

*preorder(root.left);*

*preorder(root.right);*

*}*

*public static class TreeNode<E> {*

*protected E element;*

*protected TreeNode<E> left;*

*protected TreeNode<E> right;*

*public TreeNode(E e) {*

*element = e;*

*}*

*}*

*@Override*

*public int getSize() {*

*return size;*

*}*

*public TreeNode<E> getRoot() {*

*return root;*

*}*

*public java.util.ArrayList<TreeNode<E>> path(E e) {*

*java.util.ArrayList<TreeNode<E>> list =*

*new java.util.ArrayList<>();*

*TreeNode<E> current = root;*

*while (current != null) {*

*list.add(current);*

*if (c.compare(e, current.element) < 0) {*

*current = current.left;*

*}*

*else if (c.compare(e, current.element) > 0) {*

*current = current.right;*

*}*

*else*

*break;*

*}*

*return list;*

*}*

*@Override*

*public boolean delete(E e) {*

*TreeNode<E> parent = null;*

*TreeNode<E> current = root;*

*while (current != null) {*

*if (c.compare(e, current.element) < 0) {*

*parent = current;*

*current = current.left;*

*}*

*else if (c.compare(e, current.element) > 0) {*

*parent = current;*

*current = current.right;*

*}*

*else*

*break;*

*}*

*if (current == null)*

*return false;*

*if (current.left == null) {*

*if (parent == null) {*

*root = current.right;*

*}*

*else {*

*if (c.compare(e, parent.element) < 0)*

*parent.left = current.right;*

*else*

*parent.right = current.right;*

*}*

*}*

*else {*

*TreeNode<E> parentOfRightMost = current;*

*TreeNode<E> rightMost = current.left;*

*while (rightMost.right != null) {*

*parentOfRightMost = rightMost;*

*rightMost = rightMost.right;*

*}*

*current.element = rightMost.element;*

*if (parentOfRightMost.right == rightMost)*

*parentOfRightMost.right = rightMost.left;*

*else*

*parentOfRightMost.left = rightMost.left;*

*}*

*size--;*

*return true;*

*}*

*@Override*

*public java.util.Iterator<E> iterator() {*

*return new InorderIterator();*

*}*

*private class InorderIterator implements java.util.Iterator<E> {*

*private java.util.ArrayList<E> list =*

*new java.util.ArrayList<>();*

*private int current = 0;*

*public InorderIterator() {*

*inorder();*

*}*

*private void inorder() {*

*inorder(root);*

*}*

*private void inorder(TreeNode<E> root) {*

*if (root == null) return;*

*inorder(root.left);*

*list.add(root.element);*

*inorder(root.right);*

*}*

*@Override*

*public boolean hasNext() {*

*if (current < list.size())*

*return true;*

*return false;*

*}*

*@Override*

*public E next() {*

*return list.get(current++);*

*}*

*@Override*

*public void remove() {*

*if (current == 0)*

*throw new IllegalStateException();*

*delete(list.get(--current));*

*list.clear();*

*inorder();*

*}*

*}*

*@Override*

*public void clear() {*

*root = null;*

*size = 0;*

*}*

*}*

*AVLDriver (Main Class)*

*package application;*

*import java.util.ArrayList;*

*import java.util.Collections;*

*import java.util.HashSet;*

*import java.util.Random;*

*import java.util.Set;*

*public class AVLDriver {*

*private static AVLTree<Integer> tree = new AVLTree<>();*

*public static void main(String[] args) {*

*// Creating Arraylist of Random Integers upto 100*

*ArrayList <Integer> arr = creatingArray(100);*

*System.out.println("Display Random Elements:");*

*dispayArray(arr);*

*//Method for inserting each elements to AVL Tree*

*insertValues (arr);*

*//Traverse the elements in the tree*

*System.out.println("Traverse through the Elements");*

*displayTree();*

*//get random value for search from arraylist*

*int value = getRandom(arr);*

*System.out.println("Random value: " + value);*

*//perform search*

*System.out.println("Is this random value exsit in the tree: " +tree.search(value));*

*Collections.shuffle(arr);*

*//After shuffling the elements*

*System.out.println("\nAfter Shuffling");*

*System.out.println("Display Random Elements: (after shuffling)");*

*dispayArray(arr);*

*insertValues(arr);*

*System.out.println("Traverse through the Elements (after shuffling)");*

*displayTree();*

*System.out.println("Same Random value: " + value);*

*System.out.println("Is this random value exsit in the tree: " +tree.search(value));*

*}*

*private static void displayTree() {*

*for (int e: tree) {*

*System.out.print(e + " ");*

*}*

*System.out.println("\nThe number of nodes is " + tree.getSize());*

*}*

*private static void dispayArray(ArrayList<Integer> arr) {*

*for (int i = 0; i<arr.size(); i++) {*

*System.out.print(arr.get(i) + " ");*

*}*

*System.out.println("");*

*}*

*private static void insertValues(ArrayList<Integer> arr) {*

*//clear any previous values*

*tree.clear();*

*for (int i = 0; i<arr.size(); i++) {*

*tree.insert(arr.get(i));*

*}*

*}*

*private static int getRandom(ArrayList<Integer> arr) {*

*int randIndex = new Random ().nextInt(arr.size());*

*return arr.get(randIndex);*

*}*

*private static ArrayList<Integer> creatingArray(int size) {*

*ArrayList<Integer> list = new ArrayList<Integer>();*

*Random rand = new Random ();*

*for (int i=0; i<size; i++) {*

*int x = rand.nextInt(1000);*

*list.add(x);*

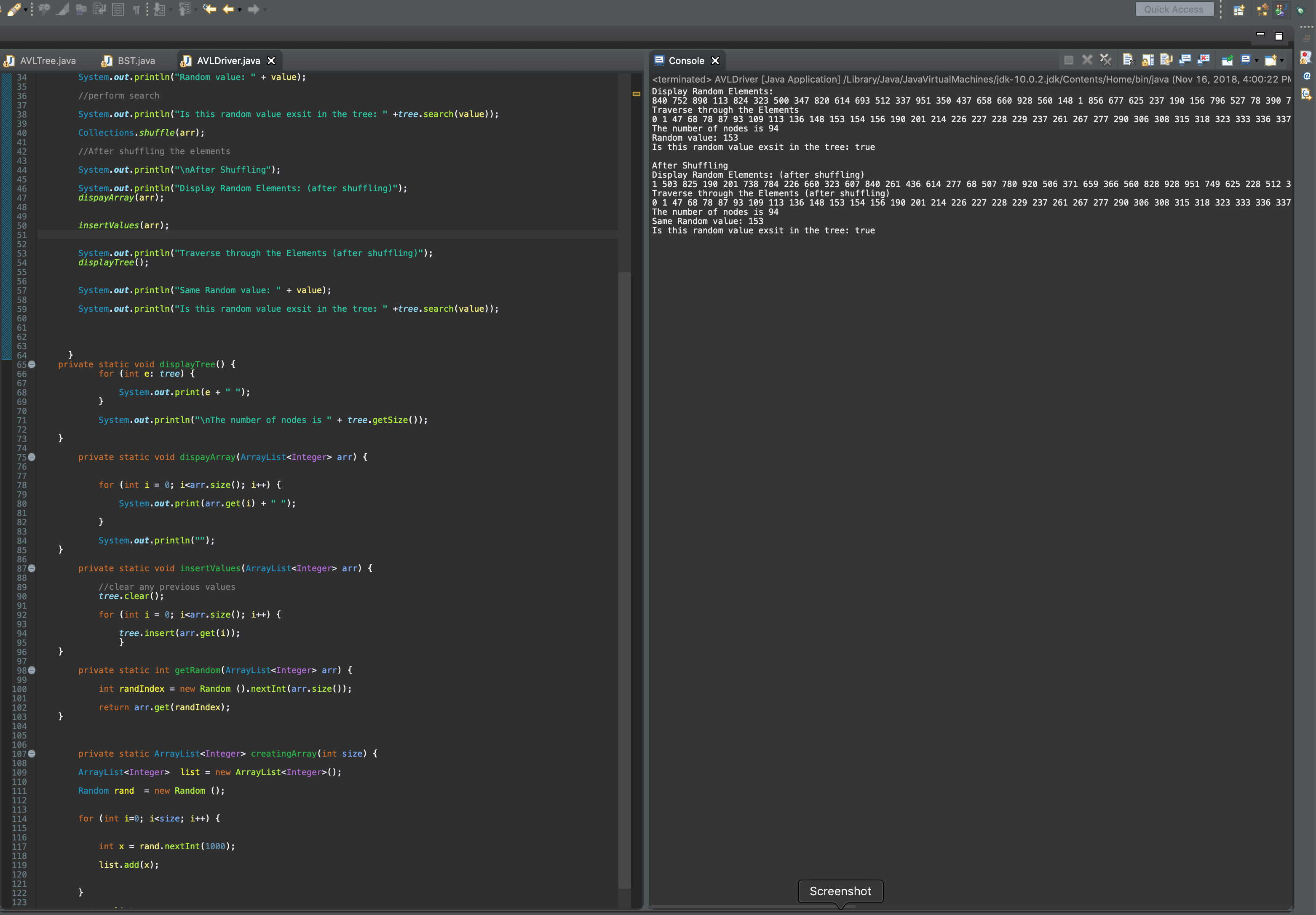
*}*

*return list;*

*}*

*}*

*Screenshots*

**