**DSA711S DATA STRUCTURES AND ALGORITHMS 2**

***Group/Team should edit this document to suit their Group Assignment project***

***The format of this MS Word file should be: Font: Calibri, Font size 12, line spacing: single***

**GROUP ASSIGNMENT TEAM LIST AND PROJECT SUBMISSION TEMPLATE**

*The Assignment must be done as a group, but each student/group member must submit the MS Word file on elearning and also send the file to the email:*[*postgraduatementor@gmail.com*](mailto:postgraduatementor@gmail.com) *on or before 23h59pm on Tuesday, 6 May 2025.*

**There is no limit to the number of pages of your MS Word document.**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Sn** | **Name** | **Student**  **Number** | **Specialization** | **Programme** | **Cell Phone Number** | **PT/**  **FT** | **Contributions made by each student/team member** |
| 1 | Mario Uushunga **(TL)** | 223023795 | Computer Science (Software Development) | Full-Time | 0812963838 | FT | Section B (5) B-trees (Implementation and search for key 8) |
| 2 | Prince Itope | 223102792 | Computer Science (Software Development) | Full-Time | 0817615116 | FT | Section B (2) Binary Search Tree (Postorder transversal) |
| 3 | Diago Flavio De Oliveira | 223086525 | Computer Science (Software Development) | Full-Time | 0812878222 | FT | Section B (4) Red Black Tree (Insert the nine elements and display Postorder transversal) |
| 4 | Benjamin Nehoya | 223056553 | Computer Science (Software Development) | Full-Time | 0814066550 | FT | Section B (3) AVL (Insert the nine elements) |
| 5 | Gospel Nwagbara | 223116866 | Computer Science (Software Development) | Full-Time | 0816317160 | FT | Section B (1) Binary Search Tree (Inserting Node 3) |

**SUBMITTED BY :** Mario Uushunga

**DATE**:

9 May 2025

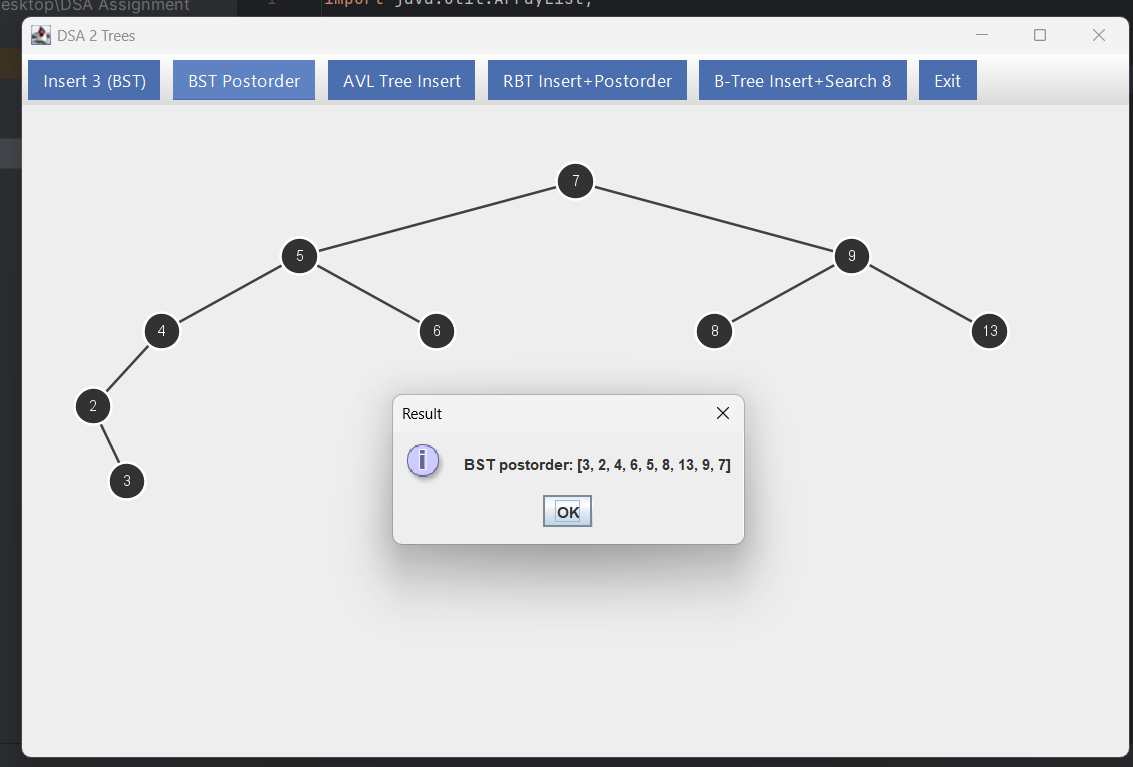
**SESSION B**

import javax.swing.\*;  
import javax.swing.border.EmptyBorder;  
import javax.swing.plaf.basic.BasicButtonUI;  
import java.awt.\*;  
import java.awt.event.\*;  
import java.util.ArrayList;  
import java.util.List;  
  
public class Main {  
 enum TreeType { *BST*, *AVL*, *RBT*, *BTREE* }  
  
 static List<Object> *avlSnapshots* = new ArrayList<>();  
 static List<Object> *rbtSnapshots* = new ArrayList<>();  
 static List<Object> *tttSnapshots* = new ArrayList<>();  
  
 private final BST bst = new BST();  
 private final AVLTree avl = new AVLTree();  
 private final RBTree rbt = new RBTree();  
 private final BTree btree = new BTree();  
 private final TreePanel treePanel;  
  
 public Main() {  
 int[] fullSeq = {7,5,9,4,6,8,13,2,3};  
 for (int v : fullSeq) {  
 btree.insert(v);  
 rbt.insert(v);  
 }  
 JFrame frame = new JFrame("DSA 2 Trees");  
 frame.setDefaultCloseOperation(JFrame.*EXIT\_ON\_CLOSE*);  
 frame.setSize(900,600);  
 frame.setLocationRelativeTo(null);  
 treePanel = new TreePanel();  
 frame.add(treePanel, BorderLayout.*CENTER*);  
 JToolBar toolbar = new JToolBar();  
 toolbar.setFloatable(false);  
 toolbar.setBorder(new EmptyBorder(5,5,5,5));  
 String[] labels = {  
 "Insert 3 (BST)",  
 "BST Postorder",  
 "AVL Tree Insert",  
 "RBT Insert+Postorder",  
 "B-Tree Insert+Search 8",  
 "Exit"  
 };  
 for (int i = 0; i < labels.length; i++) {  
 JButton btn = createButton(labels[i]);  
 final int idx = i;  
 btn.addActionListener(e -> onAction(idx));  
 toolbar.add(btn);  
 if (i < labels.length - 1) toolbar.addSeparator(new Dimension(10,0));  
 }  
 frame.add(toolbar, BorderLayout.*NORTH*);  
 frame.setVisible(true);  
 animateBSTCreation(new int[]{7,5,9,4,6,8,13,2}, 300);  
 }  
  
 private JButton createButton(String text) {  
 JButton btn = new JButton(text);  
 btn.setFont(new Font("Segoe UI", Font.*PLAIN*,14));  
 btn.setForeground(Color.*WHITE*);  
 btn.setBackground(new Color(75,110,175));  
 btn.setOpaque(true);  
 btn.setUI(new BasicButtonUI());  
 btn.setBorder(BorderFactory.*createEmptyBorder*(6,12,6,12));  
 btn.setCursor(Cursor.*getPredefinedCursor*(Cursor.*HAND\_CURSOR*));  
 btn.addMouseListener(new MouseAdapter() {  
 public void mouseEntered(MouseEvent e) { btn.setBackground(new Color(95,130,195)); }  
 public void mouseExited(MouseEvent e) { btn.setBackground(new Color(75,110,175)); }  
 });  
 return btn;  
 }  
  
 private void onAction(int idx) {  
 switch (idx) {  
 case 0:  
 if (bst.contains(3)) {  
 JOptionPane.*showMessageDialog*(treePanel, "3 is already in the BST.","Info",JOptionPane.*INFORMATION\_MESSAGE*);  
 } else {  
 animateBSTCreation(new int[]{7,5,9,4,6,8,13,2,3}, 300);  
 }  
 break;  
 case 1:  
 List<Integer> post = bst.getPostorderList();  
 treePanel.animateTraversal(TreeType.*BST*, bst.getRoot(), post, 300, () ->  
 JOptionPane.*showMessageDialog*(treePanel, "BST postorder: " + post,"Result",JOptionPane.*INFORMATION\_MESSAGE*)  
 );  
 break;  
 case 2:  
 animateAVLRotations();  
 break;  
 case 3:  
 int[] seq = {7,5,9,4,6,8,13,2,3};  
 animateRBTCreation(seq, 300);  
 int totalRbt = *rbtSnapshots*.size() \* 300 + 200;  
 new Timer(totalRbt, e -> {  
 List<Integer> rpost = rbt.getPostorderList();  
 treePanel.animateTraversal(TreeType.*RBT*, rbt.getRoot(), rpost, 300, () ->  
 JOptionPane.*showMessageDialog*(treePanel, "RBT postorder: " + rpost,"Result",JOptionPane.*INFORMATION\_MESSAGE*)  
 );  
 }) {{ setRepeats(false); }}.start();  
 break;  
 case 4:  
 int[] seq2 = {7,5,9,4,6,8,13,2,3};  
 animate23TreeCreation(seq2, 300);  
 int totalTtt = *tttSnapshots*.size() \* 300 + 200;  
 new Timer(totalTtt, e -> {  
 List<Integer> path = btree.getSearchPath(8);  
 treePanel.animateTraversal(TreeType.*BTREE*, btree.getRoot(), path, 300, () ->  
 JOptionPane.*showMessageDialog*(treePanel, "B-Tree search path: " + path,"Result",JOptionPane.*INFORMATION\_MESSAGE*)  
 );  
 }) {{ setRepeats(false); }}.start();  
 break;  
 case 5:  
 System.*exit*(0);  
 break;  
 }  
 }  
  
 private void animateBSTCreation(int[] seq, int delay) {  
 BST tmp = new BST();  
 List<Object> stages = new ArrayList<>();  
 for (int v : seq) {  
 tmp.insert(v);  
 stages.add(*cloneBST*(tmp.getRoot()));  
 }  
 bst.root = tmp.getRoot();  
 treePanel.animateStages(TreeType.*BST*, stages, delay);  
 }  
  
 private void animateAVLRotations() {  
 int[] seq = {7,5,9,4,6,8,13,2,3};  
 *avlSnapshots*.clear();  
 AVLTree tmp = new AVLTree();  
 for (int v : seq) {  
 *avlSnapshots*.add(*cloneAVL*(tmp.getRoot()));  
 tmp.insert(v);  
 *avlSnapshots*.add(*cloneAVL*(tmp.getRoot()));  
 }  
 avl.root = tmp.getRoot();  
 treePanel.animateStages(TreeType.*AVL*, *avlSnapshots*, 300);  
 }  
  
 private void animateRBTCreation(int[] seq, int delay) {  
 *rbtSnapshots*.clear();  
 RBTree tmp = new RBTree();  
 for (int v : seq) {  
 *rbtSnapshots*.add(*cloneRBT*(tmp.getRoot()));  
 tmp.insert(v);  
 *rbtSnapshots*.add(*cloneRBT*(tmp.getRoot()));  
 }  
 rbt.root = tmp.getRoot();  
 treePanel.animateStages(TreeType.*RBT*, *rbtSnapshots*, delay);  
 }  
  
 private void animate23TreeCreation(int[] seq, int delay) {  
 *tttSnapshots*.clear();  
 BTree tmp = new BTree();  
 for (int v : seq) {  
 *tttSnapshots*.add(*clone23*(tmp.getRoot()));  
 tmp.insert(v);  
 *tttSnapshots*.add(*clone23*(tmp.getRoot()));  
 }  
 btree.setRoot(tmp.getRoot());  
 treePanel.animateStages(TreeType.*BTREE*, *tttSnapshots*, delay);  
 }  
  
 public static BST.Node cloneBST(BST.Node n) {  
 if (n == null) return null;  
 BST.Node c = new BST.Node(n.key);  
 c.left = *cloneBST*(n.left);  
 c.right = *cloneBST*(n.right);  
 return c;  
 }  
  
 public static AVLTree.Node cloneAVL(AVLTree.Node n) {  
 if (n == null) return null;  
 AVLTree.Node c = new AVLTree.Node(n.key);  
 c.h = n.h;  
 c.left = *cloneAVL*(n.left);  
 c.right = *cloneAVL*(n.right);  
 if (c.left != null) c.left.p = c;  
 if (c.right != null) c.right.p = c;  
 return c;  
 }  
  
 public static RBTree.Node cloneRBT(RBTree.Node n) {  
 if (n == RBTree.*NIL*) return RBTree.*NIL*;  
 RBTree.Node c = new RBTree.Node(n.key);  
 c.color = n.color;  
 c.left = *cloneRBT*(n.left);  
 c.right = *cloneRBT*(n.right);  
 if (c.left != RBTree.*NIL*) c.left.p = c;  
 if (c.right != RBTree.*NIL*) c.right.p = c;  
 return c;  
 }  
  
 public static BTree.Node clone23(BTree.Node n) {  
 if (n == null) return null;  
 BTree.Node c = new BTree.Node();  
 c.leaf = n.leaf;  
 c.n = n.n;  
 System.*arraycopy*(n.keys, 0, c.keys, 0, n.n);  
 for (int i = 0; i <= n.n; i++) {  
 c.C[i] = *clone23*(n.C[i]);  
 }  
 return c;  
 }  
  
 public static void main(String[] args) {  
 SwingUtilities.*invokeLater*(Main::new);  
 }  
}

import javax.swing.\*;  
import java.awt.\*;  
import java.util.List;  
  
public class TreePanel extends JPanel {  
 Main.TreeType type;  
 Object root;  
  
 private List<Object> stages;  
 private int stageIndex;  
 private Timer stageTimer;  
  
 private List<Integer> travList;  
 private int travIndex;  
 private Timer travTimer;  
  
 private int highlightKey = -1;  
 private Runnable onComplete;  
  
 public void setTree(Main.TreeType t, Object r) {  
 stopTimers();  
 type = t;  
 root = r;  
 highlightKey = -1;  
 repaint();  
 }  
  
 public void animateStages(Main.TreeType t, List<Object> s, int delay) {  
 stopTimers();  
 type = t;  
 stages = s;  
 stageIndex = 0;  
 highlightKey = -1;  
 stageTimer = new Timer(delay, e -> {  
 if (stageIndex >= stages.size()) {  
 stageTimer.stop();  
 } else {  
 root = stages.get(stageIndex++);  
 repaint();  
 }  
 });  
 stageTimer.setInitialDelay(delay);  
 stageTimer.start();  
 }  
  
 public void animateTraversal(  
 Main.TreeType t,  
 Object r,  
 List<Integer> seq,  
 int delay,  
 Runnable callback  
 ) {  
 stopTimers();  
 type = t;  
 root = r;  
 travList = seq;  
 travIndex = 0;  
 highlightKey = -1;  
 onComplete = callback;  
 travTimer = new Timer(delay, e -> {  
 if (travIndex >= travList.size()) {  
 travTimer.stop();  
 highlightKey = -1;  
 repaint();  
 if (onComplete != null) onComplete.run();  
 } else {  
 highlightKey = travList.get(travIndex++);  
 repaint();  
 }  
 });  
 travTimer.setInitialDelay(delay);  
 travTimer.start();  
 }  
  
 private void stopTimers() {  
 if (stageTimer != null && stageTimer.isRunning()) stageTimer.stop();  
 if (travTimer != null && travTimer.isRunning()) travTimer.stop();  
 }  
  
 @Override  
 protected void paintComponent(Graphics g) {  
 super.paintComponent(g);  
 if (root == null) return;  
 Graphics2D g2 = (Graphics2D) g;  
 g2.setRenderingHint(RenderingHints.*KEY\_ANTIALIASING*,  
 RenderingHints.*VALUE\_ANTIALIAS\_ON*);  
 g2.setStroke(new BasicStroke(2));  
 int w = getWidth();  
 switch (type) {  
 case *BST*:  
 drawCircleTree((BST.Node) root, w/2, 60, w/4, g2, new Color(50,50,50));  
 break;  
 case *AVL*:  
 drawCircleTree((AVLTree.Node) root, w/2, 60, w/4, g2, new Color(34,139,34));  
 break;  
 case *RBT*:  
 drawCircleRBT((RBTree.Node) root, w/2, 60, w/4, g2);  
 break;  
 case *BTREE*:  
 drawBTree((BTree.Node) root, 0, w, 60, g2);  
 break;  
 }  
 }  
  
 private void drawCircleTree(BST.Node n, int x, int y, int xo,  
 Graphics2D g, Color fill) {  
 if (n.left != null) {  
 int cx = x - xo, cy = y + 60;  
 connectCircles(g, x, y, cx, cy, 15);  
 drawCircleTree(n.left, cx, cy, xo/2, g, fill);  
 }  
 if (n.right != null) {  
 int cx = x + xo, cy = y + 60;  
 connectCircles(g, x, y, cx, cy, 15);  
 drawCircleTree(n.right, cx, cy, xo/2, g, fill);  
 }  
 drawCircleNode(g, x, y, 15, n.key, fill);  
 }  
  
 private void drawCircleTree(AVLTree.Node n, int x, int y, int xo,  
 Graphics2D g, Color fill) {  
 if (n.left != null) {  
 int cx = x - xo, cy = y + 60;  
 connectCircles(g, x, y, cx, cy, 15);  
 drawCircleTree(n.left, cx, cy, xo/2, g, fill);  
 }  
 if (n.right != null) {  
 int cx = x + xo, cy = y + 60;  
 connectCircles(g, x, y, cx, cy, 15);  
 drawCircleTree(n.right, cx, cy, xo/2, g, fill);  
 }  
 drawCircleNode(g, x, y, 15, n.key, fill);  
 }  
  
 private void drawCircleNode(Graphics2D g, int x, int y, int r,  
 int key, Color fill) {  
 Color bg = (highlightKey == key ? Color.*ORANGE* : fill);  
 g.setColor(bg);  
 g.fillOval(x - r, y - r, 2\*r, 2\*r);  
 g.setColor(Color.*WHITE*);  
 g.drawOval(x - r, y - r, 2\*r, 2\*r);  
 String s = String.*valueOf*(key);  
 FontMetrics fm = g.getFontMetrics();  
 g.drawString(s,  
 x - fm.stringWidth(s)/2,  
 y + fm.getAscent()/2 - 2  
 );  
 }  
  
 private void connectCircles(Graphics2D g,  
 int x1, int y1, int x2, int y2, int r) {  
 double dx = x2 - x1, dy = y2 - y1;  
 double d = Math.*hypot*(dx, dy);  
 if (d < 1) return;  
 double ux = dx/d, uy = dy/d;  
 int sx = (int)Math.*round*(x1 + ux\*r);  
 int sy = (int)Math.*round*(y1 + uy\*r);  
 int ex = (int)Math.*round*(x2 - ux\*r);  
 int ey = (int)Math.*round*(y2 - uy\*r);  
 g.setColor(Color.*DARK\_GRAY*);  
 g.drawLine(sx, sy, ex, ey);  
 }  
  
 private void drawCircleRBT(RBTree.Node n, int x, int y, int xo,  
 Graphics2D g) {  
 if (n == RBTree.*NIL*) return;  
 int cy = y + 60;  
 if (n.left != RBTree.*NIL*) {  
 int cx = x - xo;  
 connectCircles(g, x, y, cx, cy, 15);  
 drawCircleRBT(n.left, cx, cy, xo/2, g);  
 }  
 if (n.right != RBTree.*NIL*) {  
 int cx = x + xo;  
 connectCircles(g, x, y, cx, cy, 15);  
 drawCircleRBT(n.right, cx, cy, xo/2, g);  
 }  
 Color fill = n.color ? Color.*RED* : Color.*BLACK*;  
 drawCircleNode(g, x, y, 15, n.key, fill);  
 }  
  
 private int computeSubtreeWidth(BTree.Node n, int slotW) {  
 if (n == null) return slotW;  
 int total = 0;  
 for (int i = 0; i <= n.n; i++) {  
 total += computeSubtreeWidth(n.C[i], slotW);  
 }  
 return Math.*max*(n.n \* slotW, total);  
 }  
  
 private void drawBTree(BTree.Node n, int ax, int aw, int y,  
 Graphics2D g) {  
 if (n == null) return;  
 int slotW = 60, h = 30, vgap = 60;  
 int[] cW = new int[n.n + 1], cX = new int[n.n + 1];  
 int totalW = 0;  
 for (int i = 0; i <= n.n; i++) {  
 cW[i] = computeSubtreeWidth(n.C[i], slotW);  
 totalW += cW[i];  
 }  
 int startX = ax + (aw - totalW)/2;  
 for (int i = 0, x = startX; i <= n.n; i++) {  
 cX[i] = x + cW[i]/2;  
 x += cW[i];  
 }  
 int boxW = n.n \* slotW;  
 int left = cX[0], right = cX[n.n];  
 int center= (left + right)/2;  
 int boxX = center - boxW/2;  
 g.setColor(Color.*WHITE*);  
 g.fillRect(boxX, y, boxW, h);  
 for (int i = 0; i < n.n; i++) {  
 if (highlightKey == n.keys[i]) {  
 g.setColor(Color.*YELLOW*);  
 g.fillRect(boxX + i\*slotW, y, slotW, h);  
 }  
 }  
 g.setColor(Color.*BLACK*);  
 g.drawRect(boxX, y, boxW, h);  
 for (int i = 1; i < n.n; i++) {  
 int sx = boxX + i\*slotW;  
 g.drawLine(sx, y, sx, y + h);  
 }  
 FontMetrics fm = g.getFontMetrics();  
 g.setColor(Color.*BLACK*);  
 for (int i = 0; i < n.n; i++) {  
 String s = String.*valueOf*(n.keys[i]);  
 int tx = boxX + i\*slotW + (slotW - fm.stringWidth(s))/2;  
 int ty = y + (h - fm.getHeight())/2 + fm.getAscent();  
 g.drawString(s, tx, ty);  
 }  
 if (!n.leaf) {  
 for (int i = 0; i <= n.n; i++) {  
 if (n.C[i] != null) {  
 int px = boxX + i\*slotW;  
 connectCircles(g,  
 px, y + h,  
 cX[i], y + h + vgap,  
 slotW/4  
 );  
 drawBTree(n.C[i],  
 startX + sum(cW,0,i),  
 cW[i],  
 y + h + vgap,  
 g);  
 }  
 }  
 }  
 }  
  
 private int sum(int[] arr, int start, int end) {  
 int s = 0;  
 for (int i = start; i < end; i++) s += arr[i];  
 return s;  
 }  
}

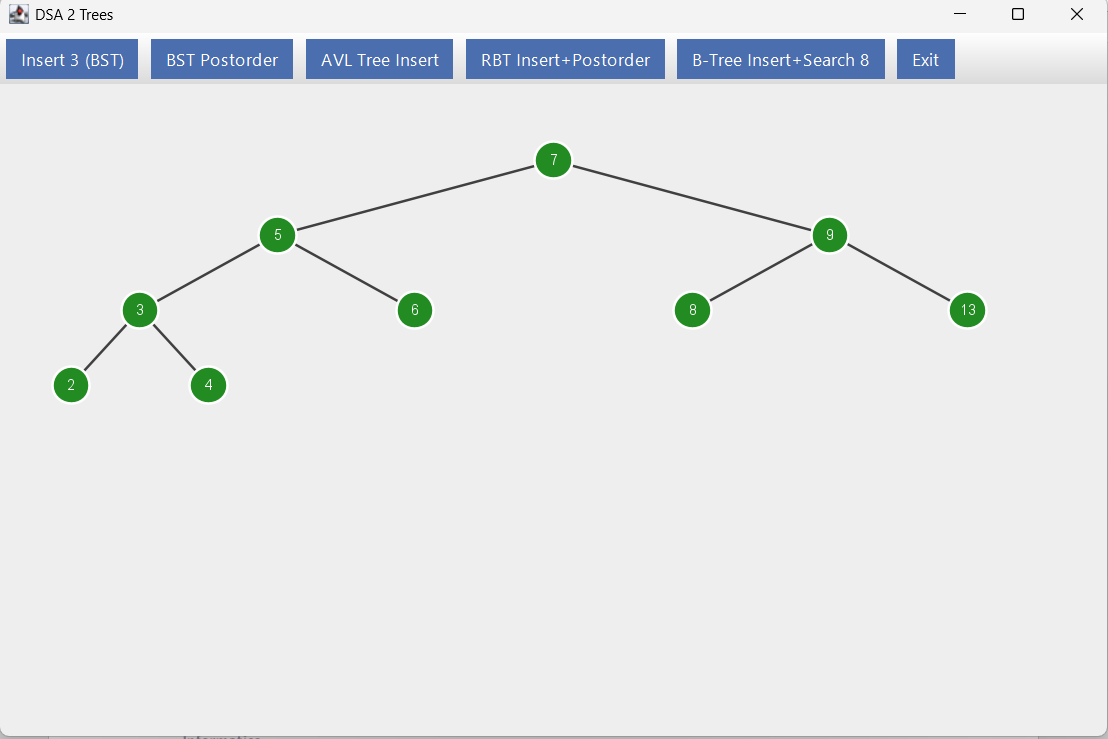
**Question 1 and Question 2**

import java.util.ArrayList;  
import java.util.List;  
  
public class BST {  
 public static class Node {  
 public int key;  
 public Node left, right;  
 public Node(int k) {  
 key = k;  
 left = right = null;  
 }  
 }  
  
 Node root;  
  
 public void insert(int k) {  
 root = insertRec(root, k);  
 }  
  
 private Node insertRec(Node node, int k) {  
 if (node == null) {  
 return new Node(k);  
 }  
 if (k < node.key) {  
 node.left = insertRec(node.left, k);  
 } else {  
 node.right = insertRec(node.right, k);  
 }  
 return node;  
 }  
  
 public boolean contains(int k) {  
 return containsRec(root, k);  
 }  
  
 private boolean containsRec(Node node, int k) {  
 if (node == null) return false;  
 if (node.key == k) return true;  
 return (k < node.key)  
 ? containsRec(node.left, k)  
 : containsRec(node.right, k);  
 }  
  
 public Node getRoot() {  
 return root;  
 }  
  
 public List<Integer> getPostorderList() {  
 List<Integer> out = new ArrayList<>();  
 postorderRec(root, out);  
 return out;  
 }  
  
 private void postorderRec(Node node, List<Integer> out) {  
 if (node == null) return;  
 postorderRec(node.left, out);  
 postorderRec(node.right, out);  
 out.add(node.key);  
 }  
}

****

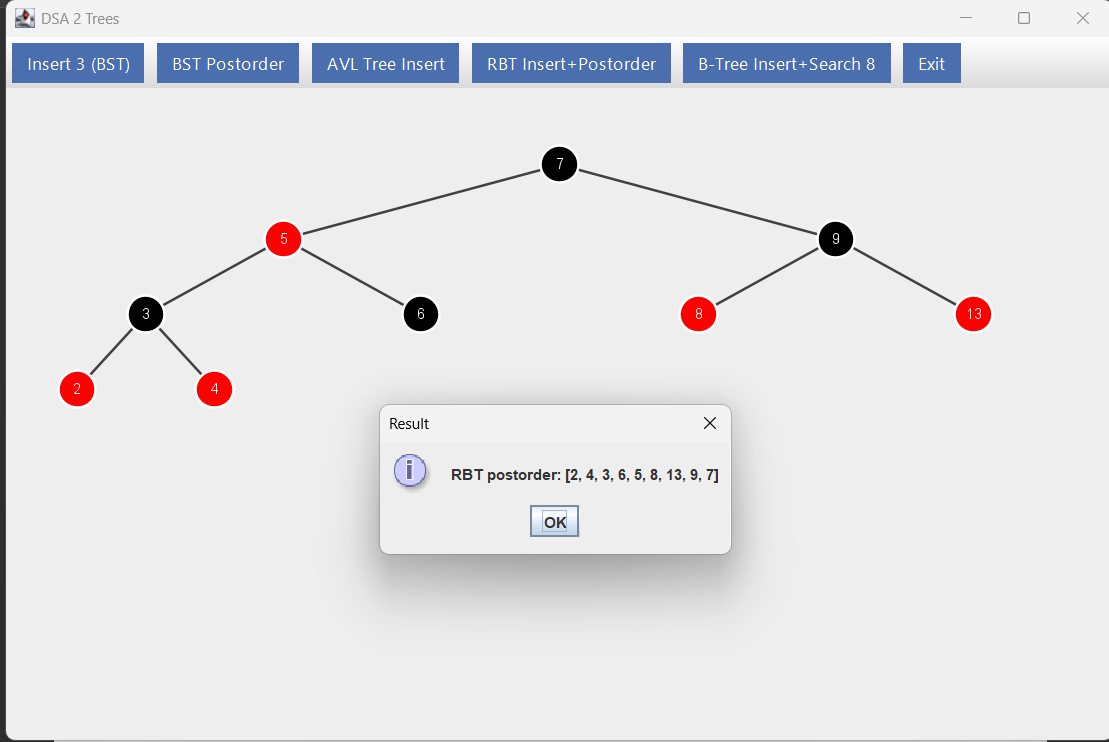
**Question 3**

import java.util.\*;  
  
public class AVLTree {  
 public static class Node {  
 public int key, h;  
 public Node left, right, p;  
 public Node(int k) {  
 key = k;  
 h = 1;  
 left = right = p = null;  
 }  
 }  
  
 Node root;  
  
 public void insert(int k) {  
 root = insertRec(root, k, null);  
 }  
  
 private Node insertRec(Node node, int k, Node parent) {  
 if (node == null) {  
 Node n = new Node(k);  
 n.p = parent;  
 return n;  
 }  
 if (k < node.key) {  
 node.left = insertRec(node.left, k, node);  
 } else {  
 node.right = insertRec(node.right, k, node);  
 }  
 node.h = 1 + Math.*max*(height(node.left), height(node.right));  
 return balance(node);  
 }  
  
 private int height(Node n) {  
 return n == null ? 0 : n.h;  
 }  
  
 private int balanceFactor(Node n) {  
 return height(n.left) - height(n.right);  
 }  
  
 public Node rotateLeftSub(Node x) {  
 Node y = x.right;  
 Node T2 = y.left;  
 y.left = x;  
 x.right = T2;  
 y.p = x.p;  
 x.p = y;  
 if (T2 != null) T2.p = x;  
 x.h = 1 + Math.*max*(height(x.left), height(x.right));  
 y.h = 1 + Math.*max*(height(y.left), height(y.right));  
 Main.*avlSnapshots*.add(Main.*cloneAVL*(root));  
 return y;  
 }  
  
 public Node rotateRightSub(Node y) {  
 Node x = y.left;  
 Node T2 = x.right;  
 x.right = y;  
 y.left = T2;  
 x.p = y.p;  
 y.p = x;  
 if (T2 != null) T2.p = y;  
 y.h = 1 + Math.*max*(height(y.left), height(y.right));  
 x.h = 1 + Math.*max*(height(x.left), height(x.right));  
 Main.*avlSnapshots*.add(Main.*cloneAVL*(root));  
 return x;  
 }  
  
 private Node balance(Node node) {  
 int bf = balanceFactor(node);  
 if (bf > 1) {  
 if (balanceFactor(node.left) < 0) {  
 node.left = rotateLeftSub(node.left);  
 }  
 node = rotateRightSub(node);  
 } else if (bf < -1) {  
 if (balanceFactor(node.right) > 0) {  
 node.right = rotateRightSub(node.right);  
 }  
 node = rotateLeftSub(node);  
 }  
 return node;  
 }  
  
 public void recalcHeights() {  
 recalc(root);  
 }  
  
 private int recalc(Node n) {  
 if (n == null) return 0;  
 n.h = 1 + Math.*max*(recalc(n.left), recalc(n.right));  
 return n.h;  
 }  
  
 public Node find(Node n, int k) {  
 if (n == null) return null;  
 if (n.key == k) return n;  
 return k < n.key ? find(n.left, k) : find(n.right, k);  
 }  
  
 public Node getRoot() {  
 return root;  
 }  
}

****

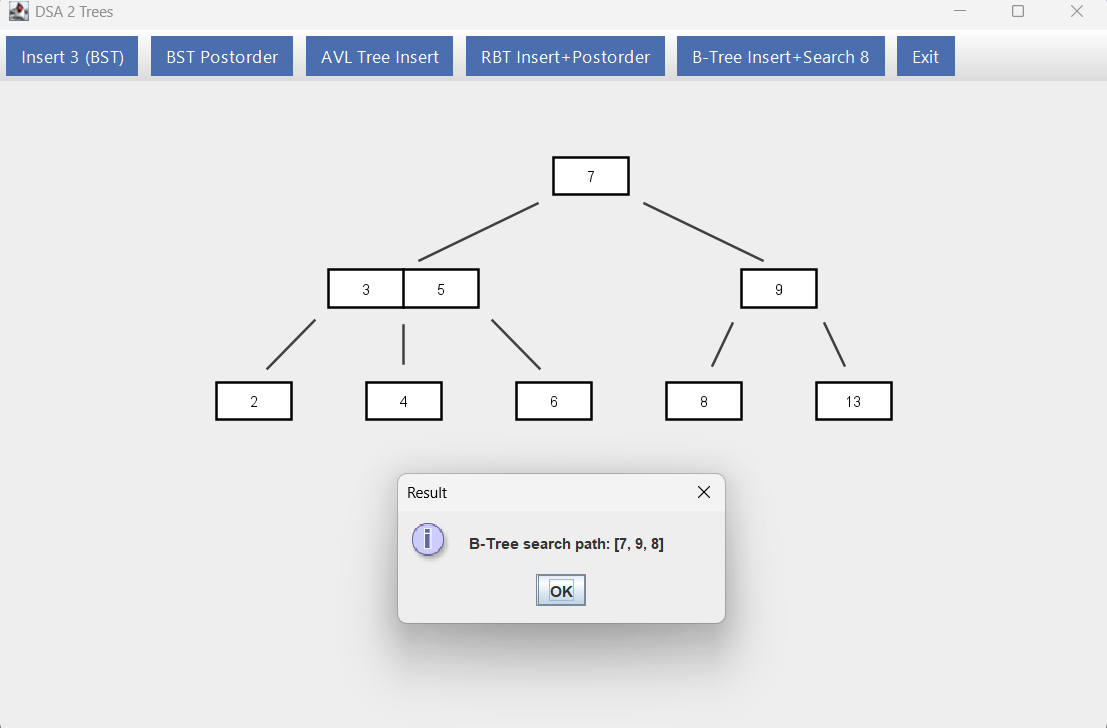
**Question 4**

import java.util.ArrayList;  
import java.util.List;  
  
public class RBTree {  
 static final boolean *RED* = true;  
 static final boolean *BLACK* = false;  
  
 public static class Node {  
 int key;  
 boolean color;  
 Node left, right, p;  
 public Node(int k) {  
 key = k;  
 color = *RED*;  
 left = right = p = *NIL*;  
 }  
 public Node(int k, boolean c) {  
 this(k);  
 color = c;  
 }  
 }  
  
 public static final Node *NIL* = new Node(0, *BLACK*);  
 static {  
 *NIL*.left = *NIL*.right = *NIL*.p = *NIL*;  
 }  
  
 Node root = *NIL*;  
  
 public void insert(int key) {  
 Node z = new Node(key), y = *NIL*, x = root;  
 while (x != *NIL*) {  
 y = x;  
 x = key < x.key ? x.left : x.right;  
 }  
 z.p = y;  
 if (y == *NIL*) {  
 root = z;  
 } else if (key < y.key) {  
 y.left = z;  
 } else {  
 y.right = z;  
 }  
 z.left = *NIL*;  
 z.right = *NIL*;  
 fixUp(z);  
 }  
  
 void fixUp(Node z) {  
 while (z.p.color == *RED*) {  
 if (z.p == z.p.p.left) {  
 Node y = z.p.p.right;  
 if (y.color == *RED*) {  
 z.p.color = *BLACK*;  
 y.color = *BLACK*;  
 z.p.p.color = *RED*;  
 Main.*rbtSnapshots*.add(Main.*cloneRBT*(root));  
 z = z.p.p;  
 } else {  
 if (z == z.p.right) {  
 z = z.p;  
 leftRotate(z);  
 Main.*rbtSnapshots*.add(Main.*cloneRBT*(root));  
 }  
 z.p.color = *BLACK*;  
 z.p.p.color = *RED*;  
 Main.*rbtSnapshots*.add(Main.*cloneRBT*(root));  
 rightRotate(z.p.p);  
 Main.*rbtSnapshots*.add(Main.*cloneRBT*(root));  
 }  
 } else {  
 Node y = z.p.p.left;  
 if (y.color == *RED*) {  
 z.p.color = *BLACK*;  
 y.color = *BLACK*;  
 z.p.p.color = *RED*;  
 Main.*rbtSnapshots*.add(Main.*cloneRBT*(root));  
 z = z.p.p;  
 } else {  
 if (z == z.p.left) {  
 z = z.p;  
 rightRotate(z);  
 Main.*rbtSnapshots*.add(Main.*cloneRBT*(root));  
 }  
 z.p.color = *BLACK*;  
 z.p.p.color = *RED*;  
 Main.*rbtSnapshots*.add(Main.*cloneRBT*(root));  
 leftRotate(z.p.p);  
 Main.*rbtSnapshots*.add(Main.*cloneRBT*(root));  
 }  
 }  
 }  
 root.color = *BLACK*;  
 Main.*rbtSnapshots*.add(Main.*cloneRBT*(root));  
 }  
  
 void leftRotate(Node x) {  
 Node y = x.right;  
 x.right = y.left;  
 if (y.left != *NIL*) y.left.p = x;  
 y.p = x.p;  
 if (x.p == *NIL*) {  
 root = y;  
 } else if (x == x.p.left) {  
 x.p.left = y;  
 } else {  
 x.p.right = y;  
 }  
 y.left = x;  
 x.p = y;  
 }  
  
 void rightRotate(Node x) {  
 Node y = x.left;  
 x.left = y.right;  
 if (y.right != *NIL*) y.right.p = x;  
 y.p = x.p;  
 if (x.p == *NIL*) {  
 root = y;  
 } else if (x == x.p.right) {  
 x.p.right = y;  
 } else {  
 x.p.left = y;  
 }  
 y.right = x;  
 x.p = y;  
 }  
  
 public Node getRoot() {  
 return root;  
 }  
  
 public List<Integer> getPostorderList() {  
 List<Integer> out = new ArrayList<>();  
 post(root, out);  
 return out;  
 }  
  
 private void post(Node n, List<Integer> L) {  
 if (n == *NIL*) return;  
 post(n.left, L);  
 post(n.right, L);  
 L.add(n.key);  
 }  
}

****

**Question 5**

import java.util.ArrayList;  
import java.util.List;  
  
public class BTree {  
 // Default to 2–3 tree (max 2 keys per node); change to 3 for 3–4 tree, etc.  
 public static int *MAX\_KEYS* = 2;  
  
 public static class Node {  
 int[] keys;  
 Node[] C;  
 int n;  
 boolean leaf;  
  
 public Node() {  
 keys = new int[*MAX\_KEYS* + 1];  
 C = new Node[*MAX\_KEYS* + 2];  
 n = 0;  
 leaf = true;  
 }  
 }  
  
 private Node root;  
  
 public BTree() {  
 root = new Node();  
 }  
  
 public Node getRoot() {  
 return root;  
 }  
  
 public void setRoot(Node r) {  
 root = r;  
 }  
  
 public void insert(int k) {  
 SplitResult r = insertRec(root, k);  
 if (r != null) {  
 Node s = new Node();  
 s.leaf = false;  
 s.n = 1;  
 s.keys[0] = r.key;  
 s.C[0] = r.left;  
 s.C[1] = r.right;  
 root = s;  
 }  
 }  
  
 private SplitResult insertRec(Node node, int k) {  
 if (node.leaf) {  
 int i = node.n - 1;  
 while (i >= 0 && k < node.keys[i]) {  
 node.keys[i+1] = node.keys[i];  
 i--;  
 }  
 node.keys[i+1] = k;  
 node.n++;  
 if (node.n <= *MAX\_KEYS*) return null;  
 return splitNode(node);  
 } else {  
 int i = 0;  
 while (i < node.n && k > node.keys[i]) i++;  
 SplitResult r = insertRec(node.C[i], k);  
 if (r == null) return null;  
 for (int j = node.n - 1; j >= i; j--) {  
 node.keys[j+1] = node.keys[j];  
 }  
 node.keys[i] = r.key;  
 for (int j = node.n; j >= i+1; j--) {  
 node.C[j+1] = node.C[j];  
 }  
 node.C[i] = r.left;  
 node.C[i+1] = r.right;  
 node.n++;  
 if (node.n <= *MAX\_KEYS*) return null;  
 return splitNode(node);  
 }  
 }  
  
 private SplitResult splitNode(Node node) {  
 int midIndex = node.n / 2;  
 int midKey = node.keys[midIndex];  
 Node left = new Node();  
 Node right = new Node();  
 left.leaf = node.leaf;  
 right.leaf = node.leaf;  
 left.n = midIndex;  
 for (int j = 0; j < midIndex; j++) {  
 left.keys[j] = node.keys[j];  
 }  
 right.n = node.n - midIndex - 1;  
 for (int j = 0; j < right.n; j++) {  
 right.keys[j] = node.keys[midIndex + 1 + j];  
 }  
 if (!node.leaf) {  
 for (int j = 0; j <= left.n; j++) {  
 left.C[j] = node.C[j];  
 }  
 for (int j = 0; j <= right.n; j++) {  
 right.C[j] = node.C[midIndex + 1 + j];  
 }  
 }  
 return new SplitResult(midKey, left, right);  
 }  
  
 private static class SplitResult {  
 int key;  
 Node left, right;  
 SplitResult(int key, Node l, Node r) {  
 this.key = key;  
 this.left = l;  
 this.right = r;  
 }  
 }  
  
 public List<Integer> getSearchPath(int k) {  
 List<Integer> path = new ArrayList<>();  
 searchPath(root, k, path);  
 return path;  
 }  
  
 private void searchPath(Node node, int k, List<Integer> path) {  
 int i = 0;  
 while (i < node.n && k > node.keys[i]) i++;  
 if (i < node.n) path.add(node.keys[i]);  
 else if (node.n > 0) path.add(node.keys[node.n - 1]);  
 if (node.leaf) return;  
 searchPath(node.C[i], k, path);  
 }  
}

****