Experiment **#3**		Student ID	
Date		Student Name	@KLWKS_BOT THANOS

Experiment Title: To implement programs on AVL trees and Red-Black Trees.

Aim/Objective: To understand the concepts and implementation of programs on AVL trees and Red-Black Trees.

Description: To learn about AVL Trees and Red-Black Trees, their balancing techniques, operations, and applications. Students will gain experience in implementing these trees, analyzing their properties, and applying them to solve real-world problems.

Pre-Requisites:

Knowledge: BST, AVL Trees and Red-Black Trees.

Tools: Code Blocks/Eclipse IDE

Pre-Lab:

1. Extend a function to find the height of an AVL tree and verify that the tree is balanced.

Input:

- A pointer or reference to the root node of an AVL tree.

Output:

- The height of the tree as an integer.
- A boolean value (true or false) indicating if the tree is balanced.

Constraints :

The tree contains at most 10^5 nodes.

- **Procedure/Program**:

```
import java.util.Stack;

class TreeNode {
  int val;
  TreeNode left, right;

  TreeNode(int x) {
    val = x;
```

```
        left = right = null;
    }
}

public class TreeBalance {
    static class StackNode {
        TreeNode node;
        int state;

        StackNode(TreeNode node, int state) {
            this.node = node;
            this.state = state;
        }
    }

    static int getHeight(TreeNode node) {
        if (node == null) return -1;
        return 0;
    }

    static boolean isBalanced(TreeNode node) {
        if (node == null) return true;
        int leftHeight = getHeight(node.left);
        int rightHeight = getHeight(node.right);
        return Math.abs(leftHeight - rightHeight) <= 1;
    }

    public static void main(String[] args) {
        TreeNode root = new TreeNode(10);
        root.left = new TreeNode(5);
        root.right = new TreeNode(20);
        root.left.left = new TreeNode(3);
        root.left.right = new TreeNode(7);
        root.right.right = new TreeNode(30);

        Stack<StackNode> stack = new Stack<>();
        stack.push(new StackNode(root, 0));
```

```
    int[] heightMap = new int[1000];
    boolean[] balanceMap = new boolean[1000];
    int nodeCount = 0;

    while (!stack.isEmpty()) {
      StackNode top = stack.pop();
      TreeNode node = top.node;
      int state = top.state;

      if (node == null) continue;

      if (state == 0) {
        stack.push(new StackNode(node, 1));
        stack.push(new StackNode(node.left, 0));
        stack.push(new StackNode(node.right, 0));
      } else {
        int leftHeight = (node.left != null) ? heightMap[node.left.val] : -1;
        int rightHeight = (node.right != null) ? heightMap[node.right.val] : -1;
        int height = Math.max(leftHeight, rightHeight) + 1;
        boolean isBalanced = Math.abs(leftHeight - rightHeight) <= 1;

        heightMap[node.val] = height;
        balanceMap[node.val] = isBalanced;
      }
    }

    int height = heightMap[root.val];
    boolean isBalanced = balanceMap[root.val];

    System.out.println("Height of the tree: " + height);
    System.out.println("Is the tree balanced? " + (isBalanced ? "Yes" : "No"));
  }
}
```

- **Data and Results:**

Data:

The tree contains integer nodes with left-right relationships to analyze.

Result:

The height and balance of the tree are calculated successfully.

- **Analysis and Inferences:**

Analysis:

The balance condition and tree height are checked efficiently here.

Inferences:

The tree structure impacts height and balance status significantly.

2. Write a function that verifies if a given binary tree satisfies the Red-Black Tree properties.

   Input:

   - A Red-Black Tree (represented as a tree structure).

   Output:

   - Print "Valid Red-Black Tree" if the tree satisfies all Red-Black properties, otherwise, print "Invalid Red-Black Tree".

   Constraints:

   The tree contains at most 10^5 nodes.

- **Procedure/Program**:

```
public class RedBlackTreeValidation {

    static class TreeNode {
        int value;
        TreeNode left, right;
        String color;

        TreeNode(int value, String color) {
            this.value = value;
            this.color = color;
            this.left = null;
            this.right = null;
        }
    }

    public static void main(String[] args) {
        TreeNode root = new TreeNode(10, "black");
        root.left = new TreeNode(5, "red");
        root.right = new TreeNode(15, "red");
        root.left.left = new TreeNode(2, "black");
        root.left.right = new TreeNode(7, "black");
```

```
root.right.left = new TreeNode(12, "black");
root.right.right = new TreeNode(20, "black");

TreeNode[] stack = new TreeNode[100];
int top = -1;
stack[++top] = root;
boolean isValid = true;
int blackHeight = -1;

while (top >= 0 && isValid) {
   TreeNode node = stack[top--];
   int currentBlackHeight = 0;

   if (node == null) {
      if (blackHeight == -1) {
         blackHeight = currentBlackHeight;
      } else if (blackHeight != currentBlackHeight) {
         isValid = false;
      }
      continue;
   }

   if (node.color.equals("red")) {
      if ((node.left != null && node.left.color.equals("red")) ||
         (node.right != null && node.right.color.equals("red"))) {
         isValid = false;
      }
   }

   if (node.color.equals("black")) {
      currentBlackHeight++;
   }

   if (node.right != null) {
      stack[++top] = node.right;
   }
   if (node.left != null) {
```

```
            stack[++top] = node.left;
        }
    }

    if (!root.color.equals("black")) {
        isValid = false;
    }

    System.out.println(isValid ? "Valid Red-Black Tree" : "Invalid Red-Black Tree");
  }
}
```

- **Data and Results:**

Data:

The tree contains nodes with values and red-black colors.

Result:

The tree is validated for all red-black tree properties.

- **Analysis and Inferences:**

Analysis:

The black height consistency and red-node rules are evaluated.

Inferences:

A valid red-black tree ensures balanced height and efficiency.

In-Lab:

Problem :

1. Write a function to construct an AVL tree, where nodes are inserted while maintaining the balance property. After each insertion, the tree should self-balance to remain AVL. Print the inorder traversal of the tree after each insertion.

Input:

- An integer `n` ($1 \leq n \leq 10^4$) representing the number of nodes to insert.
- A list of `n` integers where each integer `x` ($1 \leq x \leq 10^6$) represents a value to be inserted into the AVL tree.

Output:

- After each insertion, print the inorder traversal of the tree as space-separated values.

Sample Input:

5
20 15 25 10 5

Sample Output:

20
15 20
15 20 25
10 15 20 25
5 10 15 20 25

Constraints:

- Each insertion should maintain the AVL property, with rotations performed as needed to keep the tree balanced.

- **Procedure/Program**:

```
import java.util.Scanner;

public class AVLTree {

    static class TreeNode {
        int value;
        TreeNode left, right;
        int height;

        TreeNode(int key) {
            value = key;
            left = right = null;
            height = 1;
        }
    }

    public static int getHeight(TreeNode node) {
        if (node == null)
            return 0;
        return node.height;
    }

    public static int getBalance(TreeNode node) {
        if (node == null)
            return 0;
```

```
    return getHeight(node.left) - getHeight(node.right);
}

public static TreeNode leftRotate(TreeNode z) {
    TreeNode y = z.right;
    TreeNode T2 = y.left;

    y.left = z;
    z.right = T2;

    z.height = 1 + Math.max(getHeight(z.left), getHeight(z.right));
    y.height = 1 + Math.max(getHeight(y.left), getHeight(y.right));

    return y;
}

public static TreeNode rightRotate(TreeNode z) {
    TreeNode y = z.left;
    TreeNode T3 = y.right;

    y.right = z;
    z.left = T3;

    z.height = 1 + Math.max(getHeight(z.left), getHeight(z.right));
    y.height = 1 + Math.max(getHeight(y.left), getHeight(y.right));

    return y;
```

```
}

public static TreeNode insert(TreeNode node, int key) {
    if (node == null)
        return new TreeNode(key);

    if (key < node.value)
        node.left = insert(node.left, key);
    else if (key > node.value)
        node.right = insert(node.right, key);
    else
        return node;

    node.height = 1 + Math.max(getHeight(node.left), getHeight(node.right));

    int balance = getBalance(node);

    if (balance > 1 && key < node.left.value)
        return rightRotate(node);

    if (balance < -1 && key > node.right.value)
        return leftRotate(node);

    if (balance > 1 && key > node.left.value) {
        node.left = leftRotate(node.left);
        return rightRotate(node);
    }
```

```
    if (balance < -1 && key < node.right.value) {
        node.right = rightRotate(node.right);
        return leftRotate(node);
    }

    return node;
}

public static void inorder(TreeNode node) {
    if (node != null) {
        inorder(node.left);
        System.out.print(node.value + " ");
        inorder(node.right);
    }
}

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int n = sc.nextInt();
    int[] values = new int[n];

    for (int i = 0; i < n; i++) {
        values[i] = sc.nextInt();
    }

    TreeNode root = null;
```

```
        for (int i = 0; i < n; i++) {
            root = insert(root, values[i]);
            inorder(root);
            System.out.println();
        }

        sc.close();
    }
}
```

- **Data and Results:**

Data:

Input integers are inserted to maintain AVL tree balance.

Result:

Inorder traversal after every insertion shows a balanced AVL tree.

- **Analysis and Inferences:**

Analysis:

AVL rotations ensure height balance after each node insertion operation.

Inferences:

Balanced trees improve search efficiency and maintain logarithmic height.

2. You are given an unbalanced binary search tree, transform it into a Red-Black Tree. The program should read the values, build the binary search tree, and then balance it into a Red-Black Tree while maintaining its properties.

Input:

- A series of integers representing the values to be inserted into the binary search tree.

Output:

- Print the in-order traversal of the tree after balancing.

Sample Input:

- 50 30 70 20 40 60 80

Sample Output:

- 20 30 40 50 60 70 80

Constraints:

- The input tree must be transformed in O(n log n) time, and the Red-Black Tree properties should be verified after the transformation.

- **Procedure/Program**:

```
import java.util.*;

class RedBlackTree {
  enum Color { RED, BLACK }

  static class TreeNode {
    int value;
    Color color;
    TreeNode left, right, parent;
```

```
    TreeNode(int value) {
        this.value = value;
        this.color = Color.RED;
        this.left = this.right = this.parent = null;
    }
}

private static TreeNode TNULL;

public static void initializeTNULL() {
    TNULL = new TreeNode(0);
    TNULL.color = Color.BLACK;
}

private static void leftRotate(TreeNode root, TreeNode x) {
    TreeNode y = x.right;
    x.right = y.left;
    if (y.left != TNULL) {
        y.left.parent = x;
    }
    y.parent = x.parent;
    if (x.parent == null) {
        root = y;
    } else if (x == x.parent.left) {
        x.parent.left = y;
    } else {
        x.parent.right = y;
    }
    y.left = x;
    x.parent = y;
}

private static void rightRotate(TreeNode root, TreeNode x) {
    TreeNode y = x.left;
    x.left = y.right;
    if (y.right != TNULL) {
        y.right.parent = x;
```

```
    }
    y.parent = x.parent;
    if (x.parent == null) {
        root = y;
    } else if (x == x.parent.right) {
        x.parent.right = y;
    } else {
        x.parent.left = y;
    }
    y.right = x;
    x.parent = y;
}

private static void fixInsert(TreeNode root, TreeNode k) {
    TreeNode u;
    while (k.parent.color == Color.RED) {
        if (k.parent == k.parent.parent.left) {
            u = k.parent.parent.right;
            if (u.color == Color.RED) {
                k.parent.color = Color.BLACK;
                u.color = Color.BLACK;
                k.parent.parent.color = Color.RED;
                k = k.parent.parent;
            } else {
                if (k == k.parent.right) {
                    k = k.parent;
                    leftRotate(root, k);
                }
                k.parent.color = Color.BLACK;
                k.parent.parent.color = Color.RED;
                rightRotate(root, k.parent.parent);
            }
        } else {
            u = k.parent.parent.left;
            if (u.color == Color.RED) {
                k.parent.color = Color.BLACK;
                u.color = Color.BLACK;
```

```
                k.parent.parent.color = Color.RED;
                k = k.parent.parent;
            } else {
                if (k == k.parent.left) {
                    k = k.parent;
                    rightRotate(root, k);
                }
                k.parent.color = Color.BLACK;
                k.parent.parent.color = Color.RED;
                leftRotate(root, k.parent.parent);
            }
        }
        if (k == root) {
            break;
        }
    }
    root.color = Color.BLACK;
}

public static void insert(TreeNode root, int value) {
    TreeNode node = new TreeNode(value);
    TreeNode y = TNULL;
    TreeNode x = root;

    while (x != TNULL) {
        y = x;
        if (value < x.value) {
            x = x.left;
        } else {
            x = x.right;
        }
    }
    node.parent = y;
    if (y == TNULL) {
        root = node;
    } else if (value < y.value) {
        y.left = node;
```

```
        } else {
            y.right = node;
        }
        fixInsert(root, node);
    }

    private static void inorder(TreeNode node, List<Integer> result) {
        if (node != TNULL) {
            inorder(node.left, result);
            result.add(node.value);
            inorder(node.right, result);
        }
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int n = scanner.nextInt();

        initializeTNULL();

        TreeNode root = TNULL;

        for (int i = 0; i < n; i++) {
            int value = scanner.nextInt();
            insert(root, value);
        }

        List<Integer> result = new ArrayList<>();
        inorder(root, result);

        for (int value : result) {
            System.out.print(value + " ");
        }
        System.out.println();
    }
}
```

- **Data and Results:**

Data:

A series of integers are inserted into a Red-Black Tree.

Result:

The tree maintains Red-Black properties and prints in-order traversal.

- **Analysis and Inferences:**

Analysis:

Balancing operations ensure all Red-Black Tree properties are satisfied.

Inferences:

Red-Black Trees balance efficiently, ensuring logarithmic time complexity for operations.

Post-Lab:

1. Given the root of an AVL tree and a level k, count the number of nodes at that level.

Input:

- The root node of the AVL tree.
- An integer k ($1 \leq k \leq 10^4$) representing the level in the tree to count nodes.

Output:

- Print the number of nodes at the level k.

- **Procedure/Program**:

```
import java.util.LinkedList;
import java.util.Queue;
import java.util.Scanner;

public class Main {

    static class TreeNode {
        int value;
        TreeNode left, right;

        public TreeNode(int value) {
            this.value = value;
            this.left = this.right = null;
        }
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        int n = sc.nextInt();
        int[] values = new int[n];
        for (int i = 0; i < n; i++) {
            values[i] = sc.nextInt();
        }
```

```
int k = sc.nextInt();

TreeNode root = null;
for (int i = 0; i < n; i++) {
    TreeNode node = new TreeNode(values[i]);
    if (root == null) {
        root = node;
    } else {
        TreeNode temp = root;
        while (true) {
            if (values[i] < temp.value) {
                if (temp.left == null) {
                    temp.left = node;
                    break;
                }
                temp = temp.left;
            } else {
                if (temp.right == null) {
                    temp.right = node;
                    break;
                }
                temp = temp.right;
            }
        }
    }
}

int count = 0;
Queue<TreeNode> queue = new LinkedList<>();
Queue<Integer> levels = new LinkedList<>();
queue.add(root);
levels.add(1);

while (!queue.isEmpty()) {
    TreeNode node = queue.poll();
    int level = levels.poll();
```

```
        if (level == k) {
          count++;
        }

        if (node.left != null) {
          queue.add(node.left);
          levels.add(level + 1);
        }

        if (node.right != null) {
          queue.add(node.right);
          levels.add(level + 1);
        }
      }

      System.out.println(count);
    }
}
```

- **Data and Results:**

Data:

Input values are inserted into an AVL tree to count nodes.

Result:

The program counts the nodes at a specific tree level.

- **Analysis and Inferences:**

Analysis:

Breadth-first search efficiently counts nodes at the desired tree level.

Inferences:

Level counting is efficient with BFS in a binary tree structure.

2. Calculate the maximum depth of a given Red-Black Tree.

Input:

- The root node of a Red-Black tree, represented as a list of values that would form the tree structure.

Output:

- Print the maximum depth as an integer.

- **Procedure/Program**:

```
import java.util.Scanner;

class TreeNode {
   int value;
   String color;
   TreeNode left, right;

   public TreeNode(int value) {
      this.value = value;
      this.color = "red";
      this.left = this.right = null;
   }
}

public class Main {

   public static int maxDepth(TreeNode node) {
      if (node == null) {
         return 0;
      }
      int leftDepth = maxDepth(node.left);
      int rightDepth = maxDepth(node.right);
      return Math.max(leftDepth, rightDepth) + 1;
```

```
}

public static void main(String[] args) {
   Scanner scanner = new Scanner(System.in);

   int n = scanner.nextInt();
   int[] values = new int[n];

   for (int i = 0; i < n; i++) {
      values[i] = scanner.nextInt();
   }

   TreeNode root = null;

   for (int i = 0; i < n; i++) {
      TreeNode node = new TreeNode(values[i]);
      if (root == null) {
         root = node;
      } else {
         TreeNode temp = root;
         while (true) {
            if (values[i] < temp.value) {
               if (temp.left == null) {
                  temp.left = node;
                  break;
               }
               temp = temp.left;
            } else {
               if (temp.right == null) {
                  temp.right = node;
                  break;
               }
```

```
                temp = temp.right;
            }
        }
    }
}

        System.out.println(maxDepth(root));
    }
}
```

- **Data and Results:**

Data:

A series of node values are inserted into the tree.

Result:

The program calculates and prints the maximum depth of tree.

- **Analysis and Inferences:**

Analysis:

Maximum depth is determined using a recursive depth-first approach.

Inferences:

Recursive depth calculation efficiently determines the longest path in tree.

- **Sample VIVA-VOCE Questions (In-Lab):**

1. What is the balance factor in an AVL Tree? How do you calculate it?

It is the difference between the heights of the left and right subtrees of a node. It's calculated as:

$$\text{Balance Factor} = \text{Height of Left Subtree} - \text{Height of Right Subtree}$$

A balance factor between -1 and 1 indicates balance.

2. What is the significance of the red and black properties in Red-Black Trees?

- Nodes are either red or black.
- The root is black.
- No two red nodes can be adjacent.
- Every path from a node to its leaf must have the same number of black nodes. These properties ensure balanced tree height.

3. Can an AVL Tree have more than one node with the same value? How does it handle duplicates?

Typically, duplicates are not allowed. If allowed, they are handled by placing them consistently in one subtree.

\

4. Compare the time complexity of insertion, deletion, and lookup in AVL Trees and Red-Black Trees.

- **AVL Tree:** Insertion, Deletion, and Lookup all take O(log n).
- **Red-Black Tree:** Insertion, Deletion, and Lookup also take O(log n), but Red-Black Trees may perform better due to fewer rotations.

5. What is the difference between an AVL Tree and a Red-Black Tree in terms of balancing and performance?

- AVL Trees enforce stricter balance with more rotations but faster lookups.
- Red-Black Trees are less strict, leading to fewer rotations and better performance for insertions/deletions.

Evaluator Remark (if Any):	**Marks Secured ___out of 50**
	Signature of the Evaluator with Date

Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.