I**D: hrajranj**

**Day17 – 10th July 2025**

**Task 1:**

**Write Algo for AVL tree**

**Ans:**

AVL search Algo

1 − Create a node

2 − Check if tree is empty

3 − If tree is empty, new node is root node.

4 − not empty, perform Binary Search Tree insertion operation and check balancing factor of the node in the tree.

5 − Suppose balancing factor > apply rotations on node and resume insertion from Step 4.

**Task 2:**

**Write code for AVL tree**

**Hint: try to insert nodes**

**While inserting get the balance of the tree**

**Create 2 methods for left rotate and right rotate**

**Try to insert**

**Finally display**

**Ans:**

import java.util.\*;

class Node {

   int key, height;

   Node left, right;

   Node (int d) {

      key = d;

      height = 1;

   }

}

public class AVLTree {

   Node root;

   int height (Node N) {

      if (N == null)

         return 0;

      return N.height;

   }

   int max (int a, int b) {

      return (a > b) ? a : b;

   }

   Node rightRotate (Node y) {

      Node x = y.left;

      Node T2 = x.right;

      x.right = y;

      y.left = T2;

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

      x.height = max (height (x.left), height (x.right)) + 1;

      return x;

   }

   Node leftRotate (Node x) {

      Node y = x.right;

      Node T2 = y.left;

      y.left = x;

      x.right = T2;

      x.height = max (height (x.left), height (x.right)) + 1;

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

      return y;

   }

   int getBalance (Node N) {

      if (N == null)

         return 0;

      return height (N.left) - height (N.right);

   }

   Node insert (Node node, int key) {

      if (node == null)

         return (new Node (key));

      if (key < node.key)

         node.left = insert (node.left, key);

      else if (key > node.key)

         node.right = insert (node.right, key);

      else

         return node;

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

      int balance = getBalance (node);

      if (balance > 1 && key < node.left.key)

         return rightRotate (node);

      if (balance < -1 && key > node.right.key)

         return leftRotate (node);

      if (balance > 1 && key > node.left.key) {

         node.left = leftRotate (node.left);

         return rightRotate (node);

      }

      if (balance < -1 && key < node.right.key) {

         node.right = rightRotate (node.right);

         return leftRotate (node);

      }

      return node;

   }

   void printTree(Node root){

   if (root == null)

      return;

   if (root != null) {

      printTree(root.left);

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

      printTree(root.left);

   }

}

   public static void main(String args[]) {

      AVLTree tree = new AVLTree();

      tree.root = tree.insert(tree.root, 5);

      tree.root = tree.insert(tree.root, 15);

      tree.root = tree.insert(tree.root, 44);

      tree.root = tree.insert(tree.root, 35);

      tree.root = tree.insert(tree.root, 65);

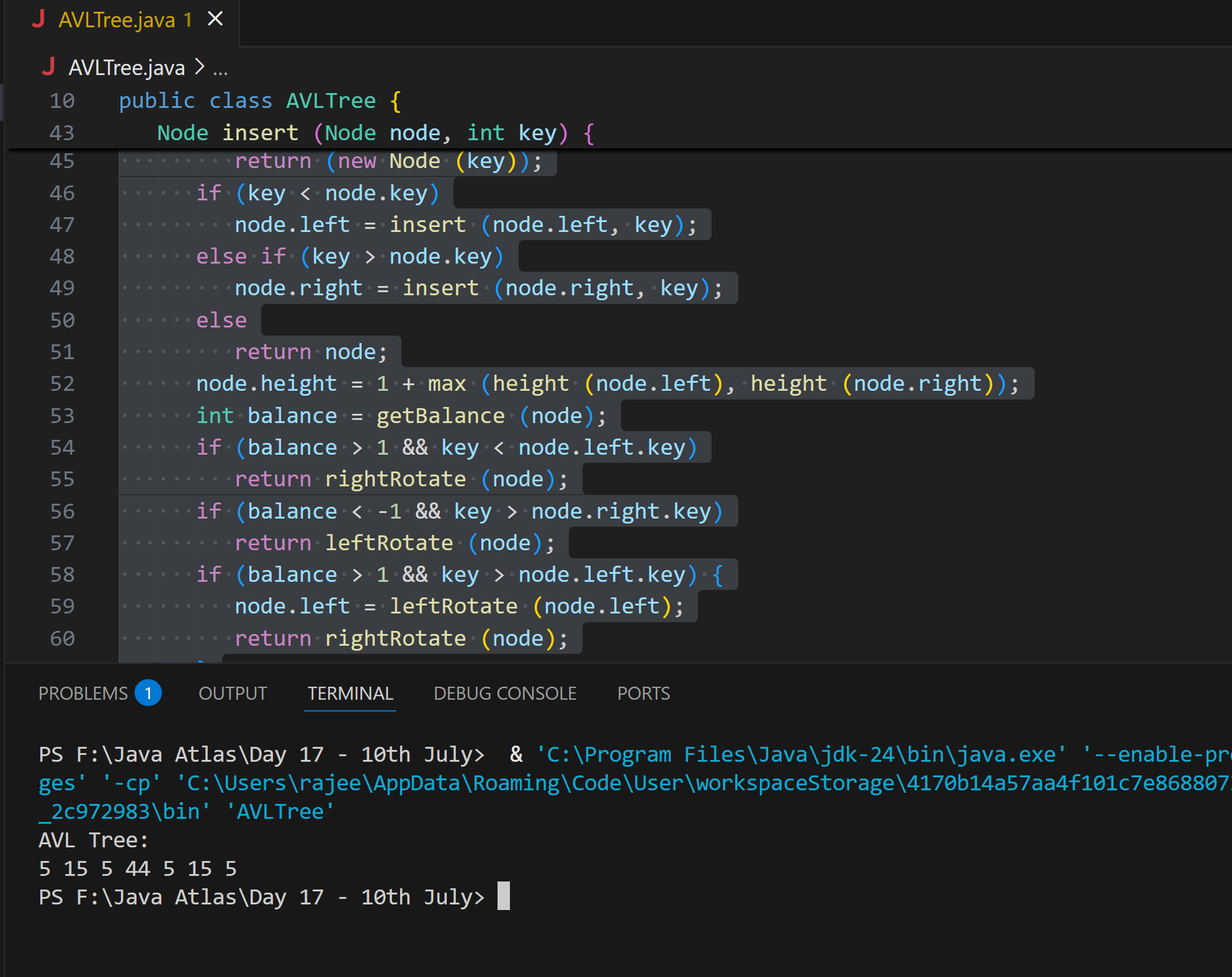
      tree.root = tree.insert(tree.root, 78);

      System.out.println("AVL Tree: ");

      tree.printTree(tree.root);

   }

}

****

**Task 3:**

**Write algo for Read Black tree insertion**

**Ans:**

A **Red-Black Tree (RBT)** is a self-balancing Binary Search Tree with the following properties:

**Red-Black Tree Properties:**

1. Every node is either **red** or **black**.
2. The **root** is always **black**.
3. **Red nodes** cannot have **red children** (no two reds in a row).
4. Every path from a node to its leaf/null descendants has the **same number of black nodes**.
5. New nodes are always inserted as **red**.

**Insertion Algorithm**

**Step-by-Step:**

**Input**: Root node root, value to insert key

1. **BST Insert**:
   * Insert the new node just like in a **normal Binary Search Tree (BST)**.
   * Color the newly inserted node as **RED**.
2. **Fix Violations**:
   * While the **parent** of the inserted node is **RED**, do the following:

**Fixing Violations:**

Let’s define:

* N: Newly inserted node
* P: Parent of N
* G: Grandparent of N
* U: Uncle of N

**Case 1: Uncle U is RED**

* Recolor P and U to **BLACK**
* Recolor G to **RED**
* Set N = G and continue (go up the tree)

**Case 2: Uncle U is BLACK or NULL**

There are 4 subcases depending on N's position:

**▶ Case 2a: Left-Left (N is left child of P, and P is left child of G)**

* **Right Rotate G**
* Swap colors of P and G

**▶ Case 2b: Right-Right (N is right child of P, and P is right child of G)**

* **Left Rotate G**
* Swap colors of P and G

**▶ Case 2c: Left-Right (N is right child of P, P is left of G)**

* **Left Rotate P** → becomes Left-Left
* Apply Case 2a

**▶ Case 2d: Right-Left (N is left child of P, P is right of G)**

* **Right Rotate P** → becomes Right-Right
* Apply Case 2b

**🏁 Final Step:**

* After all fixing, always **recolor the root to BLACK**.

**Task  4:**

**Wap to insert an element in red black tree**

**Ans:**

class RedBlackTree {

    static final boolean RED = true;

    static final boolean BLACK = false;

    class Node {

        int data;

        boolean color;

        Node left, right, parent;

        Node(int data) {

            this.data = data;

            this.color = RED;

        }

    }

    private Node root;

    // Left rotate

    private void leftRotate(Node x) {

        Node y = x.right;

        x.right = y.left;

        if (y.left != null) 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;

    }

    // Right rotate

    private void rightRotate(Node y) {

        Node x = y.left;

        y.left = x.right;

        if (x.right != null) x.right.parent = y;

        x.parent = y.parent;

        if (y.parent == null) root = x;

        else if (y == y.parent.left) y.parent.left = x;

        else y.parent.right = x;

        x.right = y;

        y.parent = x;

    }

    // Insert a new node

    public void insert(int data) {

        Node newNode = new Node(data);

        root = bstInsert(root, newNode);

        fixViolation(newNode);

    }

    // BST insert

    private Node bstInsert(Node root, Node node) {

        if (root == null) return node;

        if (node.data < root.data) {

            root.left = bstInsert(root.left, node);

            root.left.parent = root;

        } else if (node.data > root.data) {

            root.right = bstInsert(root.right, node);

            root.right.parent = root;

        }

        return root;

    }

    // Fix Red-Black Tree violations after insert

    private void fixViolation(Node node) {

        Node parent = null;

        Node grandparent = null;

        while (node != root && node.parent.color == RED) {

            parent = node.parent;

            grandparent = parent.parent;

            if (parent == grandparent.left) {

                Node uncle = grandparent.right;

                if (uncle != null && uncle.color == RED) {

                    // Case 1: Recolor

                    parent.color = BLACK;

                    uncle.color = BLACK;

                    grandparent.color = RED;

                    node = grandparent;

                } else {

                    if (node == parent.right) {

                        // Case 2: Left Rotation

                        node = parent;

                        leftRotate(node);

                    }

                    // Case 3: Right Rotation

                    parent.color = BLACK;

                    grandparent.color = RED;

                    rightRotate(grandparent);

                }

            } else {

                Node uncle = grandparent.left;

                if (uncle != null && uncle.color == RED) {

                    parent.color = BLACK;

                    uncle.color = BLACK;

                    grandparent.color = RED;

                    node = grandparent;

                } else {

                    if (node == parent.left) {

                        node = parent;

                        rightRotate(node);

                    }

                    parent.color = BLACK;

                    grandparent.color = RED;

                    leftRotate(grandparent);

                }

            }

        }

        root.color = BLACK;

    }

    // Inorder traversal

    public void inorder() {

        inorderHelper(root);

    }

    private void inorderHelper(Node node) {

        if (node != null) {

            inorderHelper(node.left);

            System.out.print(node.data + "(" + (node.color == RED ? "R" : "B") + ") ");

            inorderHelper(node.right);

        }

    }

    // Main method

    public static void main(String[] args) {

        RedBlackTree tree = new RedBlackTree();

        int[] values = {10, 20, 30, 15, 25, 5, 1};

        for (int val : values) {

            tree.insert(val);

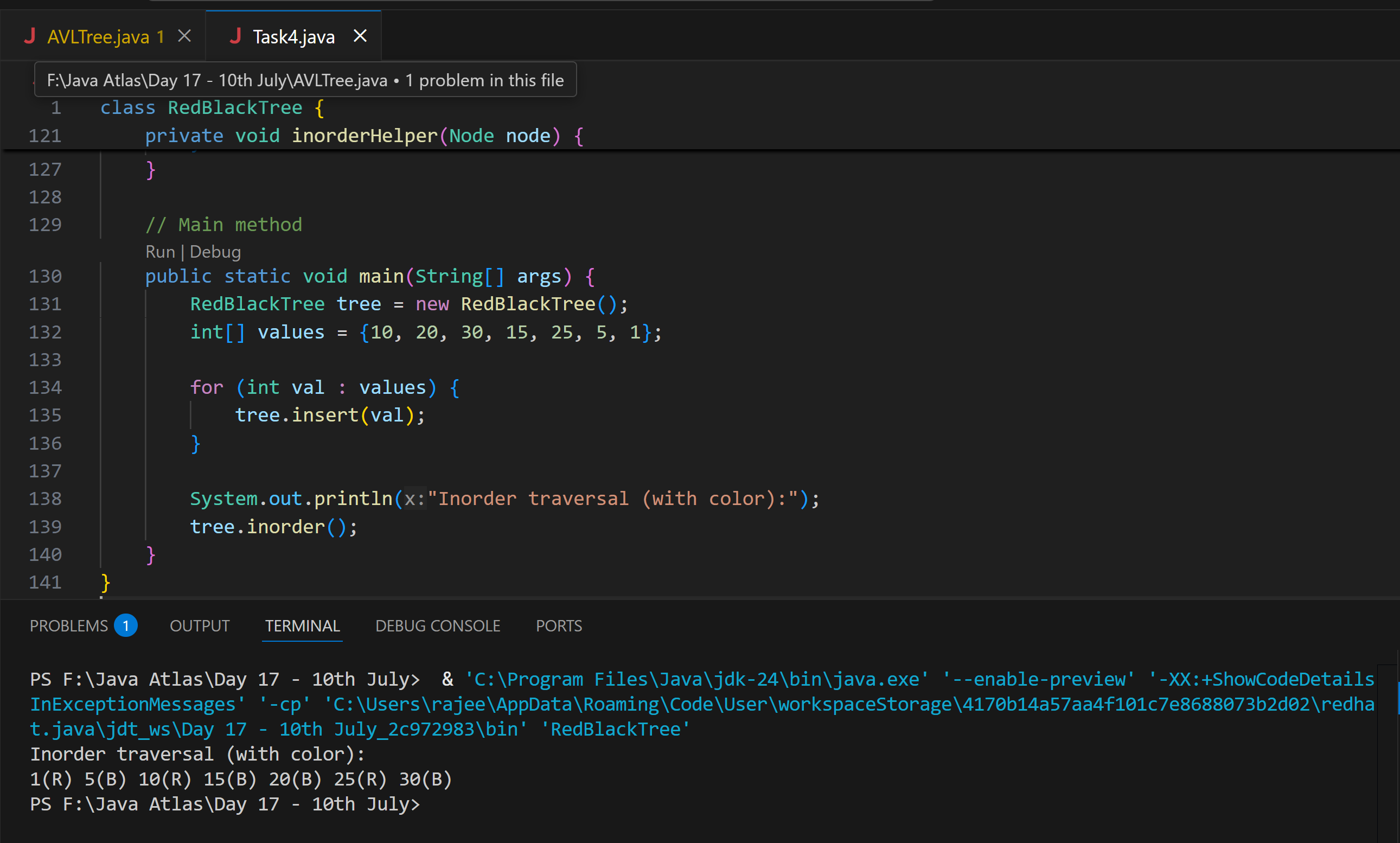
        }

        System.out.println("Inorder traversal (with color):");

        tree.inorder();

    }

}

****

**AVL qn:**

Why are AVL trees considered balanced, and how does this impact performance?

* 1. They rearrange nodes to ensure maximum depth for fast access.
  2. They allow unbalanced growth in left subtrees for quick insertion.
  3. They maintain a balance factor (height difference) of at most 1 betweensubtrees to ensure O(log n) operations**.**
  4. They replicate nodes for redundancy, enabling constant-time deletion.

What is the maximum height of an AVL tree with p nodes?

a) p

b) log(p)

c) log(p)/2

d) p⁄2

What maximum difference in heights between the leafs of a AVL tree is possible?

a) log(n) where n is the number of nodes

b) n where n is the number of nodes

c) 0 or 1

d) atmost 1 (as avl is self balanced)

Why to prefer red-black trees over AVL trees?

a) Because red-black is more rigidly balanced

b) AVL tree store balance factor in every node which costs space

c) AVL tree fails at scale

d) Red black is more efficient