**Algo for avl tree**

1. Create node
2. Check if tree is empty r no
3. If tree us empty the node will become the root node
4. If tree is not empty do binary search tree insertion and check the balance factor for the node
5. If the balance factors exceeds 1 we should do rotations on the heavy weighted tree and repeat the insertion from step 4

**Pseudo code**

function insert(node, 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 // duplicate

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

balance = height(node.left) - height(node.right)

if balance > 1 and key < node.left.key:

return rotateRight(node) // LL

if balance < -1 and key > node.right.key:

return rotateLeft(node) // RR

if balance > 1 and key > node.left.key:

node.left = rotateLeft(node.left)

return rotateRight(node) // LR

if balance < -1 and key < node.right.key:

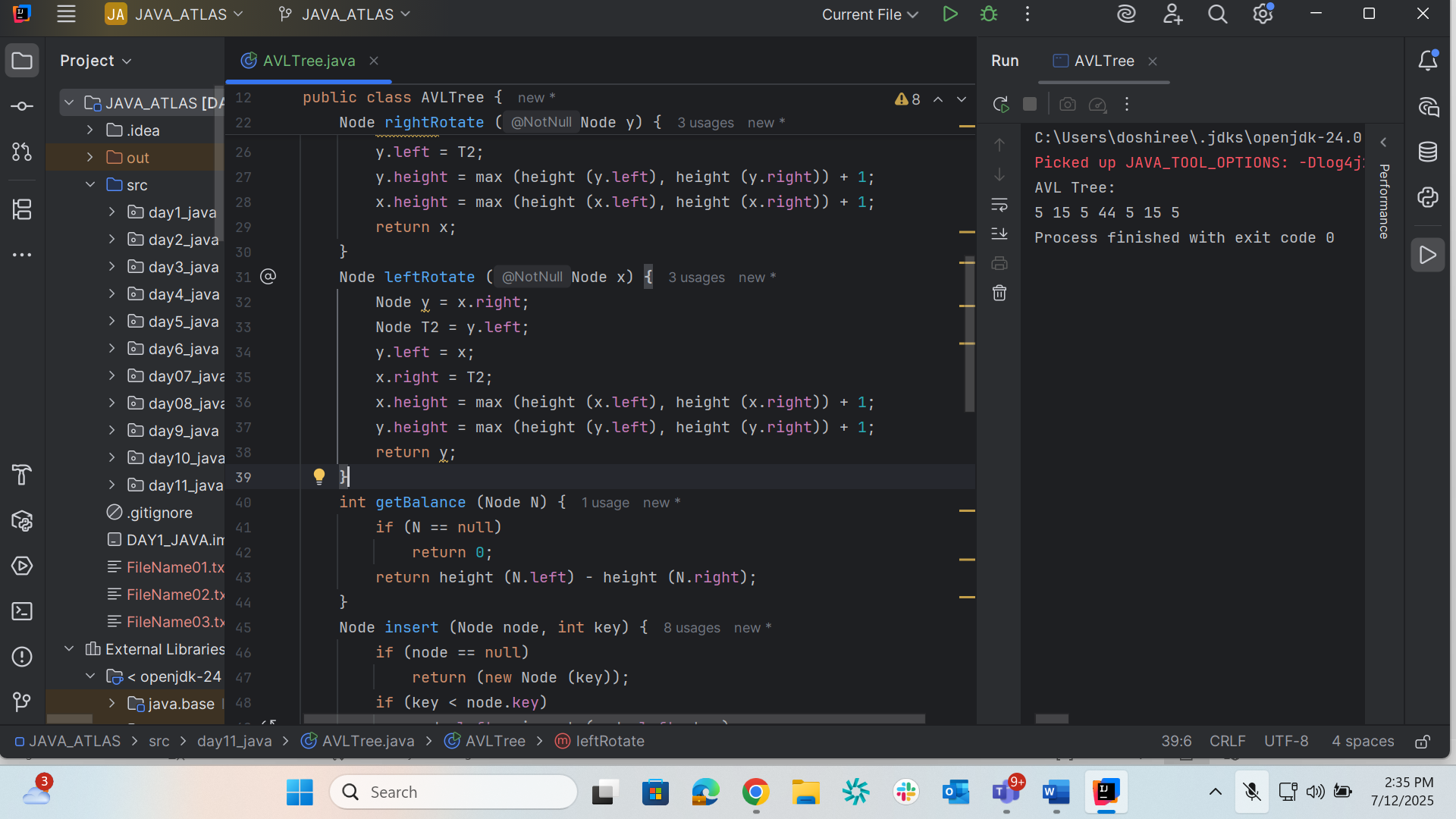
node.right = rotateRight(node.right)

return rotateLeft(node) // RL

return node

**code**

**package day11\_java;  
  
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);  
  
 }  
}**



**Red black graph**

| **Property #** | **Rule** |
| --- | --- |
| 1 | Node color: red or black |
| 2 | Root is black |
| 3 | Leaf NIL nodes are black |
| 4 | No red node has a red child |
| 5 | All root–leaf paths have same number of black nodes (black-height) |