Day Task 1:

Write Algo for AVL tree

1. create a node

2. check if tree is empty or not

3. if tree is empty the inserted node will be the root node.

4. if tree is not empty , do a binary search tree insertion op and also check the balance factor of the node.

5. if the balance factor exceeds 1, we should do rotations on the heavy weighted tree and repeat the insertion from step 4 onwards.

Task 2:

Write code for AVL tree

package July12;  
  
// Node class for AVL Tree  
class AVLNode {  
 int key, height;  
 AVLNode left, right;  
  
 public AVLNode(int d) {  
 key = d;  
 height = 1; // new node is initially added at leaf  
 }  
}  
  
// AVL Tree class  
public class AVLTree {  
 private AVLNode root;  
  
 // Get height of node  
 int height(AVLNode N) {  
 return (N == null) ? 0 : N.height;  
 }  
  
 // Get balance factor  
 int getBalance(AVLNode N) {  
 return (N == null) ? 0 : height(N.left) - height(N.right);  
 }  
  
 // Right rotate  
 AVLNode rightRotate(AVLNode y) {  
 AVLNode x = y.left;  
 AVLNode T2 = x.right;  
  
 // Perform rotation  
 x.right = y;  
 y.left = T2;  
  
 // Update heights  
 y.height = Math.*max*(height(y.left), height(y.right)) + 1;  
 x.height = Math.*max*(height(x.left), height(x.right)) + 1;  
  
 return x;  
 }  
  
 // Left rotate  
 AVLNode leftRotate(AVLNode x) {  
 AVLNode y = x.right;  
 AVLNode T2 = y.left;  
  
 // Perform rotation  
 y.left = x;  
 x.right = T2;  
  
 // Update heights  
 x.height = Math.*max*(height(x.left), height(x.right)) + 1;  
 y.height = Math.*max*(height(y.left), height(y.right)) + 1;  
  
 return y;  
 }  
  
 // Insert a key into the AVL Tree  
 AVLNode insert(AVLNode node, int key) {  
 // 1. Perform normal BST insertion  
 if (node == null) return new AVLNode(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 keys not allowed  
  
 // 2. Update height  
 node.height = 1 + Math.*max*(height(node.left), height(node.right));  
  
 // 3. Get balance factor  
 int balance = getBalance(node);  
  
 // 4. Balance the tree (4 cases)  
 // Left Left Case  
 if (balance > 1 && key < node.left.key)  
 return rightRotate(node);  
  
 // Right Right Case  
 if (balance < -1 && key > node.right.key)  
 return leftRotate(node);  
  
 // Left Right Case  
 if (balance > 1 && key > node.left.key) {  
 node.left = leftRotate(node.left);  
 return rightRotate(node);  
 }  
  
 // Right Left Case  
 if (balance < -1 && key < node.right.key) {  
 node.right = rightRotate(node.right);  
 return leftRotate(node);  
 }  
  
 return node;  
 }  
  
 // Public insert method  
 public void insert(int key) {  
 root = insert(root, key);  
 }  
  
 // In-order traversal  
 public void inorderTraversal() {  
 inorderHelper(root);  
 System.*out*.println();  
 }  
  
 private void inorderHelper(AVLNode node) {  
 if (node != null) {  
 inorderHelper(node.left);  
 System.*out*.print(node.key + " ");  
 inorderHelper(node.right);  
 }  
 }  
  
 // Main method to test the AVL tree  
 public static void main(String[] args) {  
 AVLTree tree = new AVLTree();  
  
 int[] keys = { 10, 20, 30, 40, 50, 25 };  
  
 for (int key : keys) {  
 tree.insert(key);  
 }  
  
 System.*out*.println("In-order traversal of the AVL Tree:");  
 tree.inorderTraversal();  
 }  
}

output:

In-order traversal of the AVL Tree:

10 20 25 30 40 50

Task 3:

Write algo for Read Black tree insertion

Insert an Element - Red Black Tree −

1. Check tree is empty. If empty, then insert new node - color Black. (Because Root Node - Black in color)

2. else if Tree - not empty then insert new node as leaf node to the end and color - Red.

3. If parent of new node is Red and its neighbours(parent’s) node is also Red,

then Flip the color of the both neighbour and Parent and Grandparents (If it is not Root Node Otherwise Flip the color of the Parent and neighbour only) i.e., Black.

4. If parent of new node is Red and its neighbours(parent’s) node is empty or NULL,

then Rotate (either Left-Left or Left-Right rotation) the new node and parent.

5. we have two types of rotation

- Left Left Rotation and

- Left Right Rotation.

6. we apply Rotation in some conditions only.

The conditions are −

- If parent of new node is Red and neighbour node is empty or NULL, then rotate left or right rotation.

- In Left-Left Rotation flip the color of the parent and grandparent.

Make the parent as Grandparent and grandparent as child

Task  4:

Wap to insert an element in red black tree

package July12;  
  
  
// AVL Tree class  
public class AVLTree {  
 private AVLNode root;  
  
 // Get height of node  
 int height(AVLNode N) {  
 return (N == null) ? 0 : N.height;  
 }  
  
 // Get balance factor  
 int getBalance(AVLNode N) {  
 return (N == null) ? 0 : height(N.left) - height(N.right);  
 }  
  
 // Right rotate  
 AVLNode rightRotate(AVLNode y) {  
 AVLNode x = y.left;  
 AVLNode T2 = x.right;  
  
 // Perform rotation  
 x.right = y;  
 y.left = T2;  
  
 // Update heights  
 y.height = Math.*max*(height(y.left), height(y.right)) + 1;  
 x.height = Math.*max*(height(x.left), height(x.right)) + 1;  
  
 return x;  
 }  
  
 // Left rotate  
 AVLNode leftRotate(AVLNode x) {  
 AVLNode y = x.right;  
 AVLNode T2 = y.left;  
  
 // Perform rotation  
 y.left = x;  
 x.right = T2;  
  
 // Update heights  
 x.height = Math.*max*(height(x.left), height(x.right)) + 1;  
 y.height = Math.*max*(height(y.left), height(y.right)) + 1;  
  
 return y;  
 }  
  
 // Insert a key into the AVL Tree  
 AVLNode insert(AVLNode node, int key) {  
 // 1. Perform normal BST insertion  
 if (node == null) return new AVLNode(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 keys not allowed  
  
 // 2. Update height  
 node.height = 1 + Math.*max*(height(node.left), height(node.right));  
  
 // 3. Get balance factor  
 int balance = getBalance(node);  
  
 // 4. Balance the tree (4 cases)  
 // Left Left Case  
 if (balance > 1 && key < node.left.key)  
 return rightRotate(node);  
  
 // Right Right Case  
 if (balance < -1 && key > node.right.key)  
 return leftRotate(node);  
  
 // Left Right Case  
 if (balance > 1 && key > node.left.key) {  
 node.left = leftRotate(node.left);  
 return rightRotate(node);  
 }  
  
 // Right Left Case  
 if (balance < -1 && key < node.right.key) {  
 node.right = rightRotate(node.right);  
 return leftRotate(node);  
 }  
  
 return node;  
 }  
  
 // Public insert method  
 public void insert(int key) {  
 root = insert(root, key);  
 }  
  
 // In-order traversal  
 public void inorderTraversal() {  
 inorderHelper(root);  
 System.*out*.println();  
 }  
  
 private void inorderHelper(AVLNode node) {  
 if (node != null) {  
 inorderHelper(node.left);  
 System.*out*.print(node.key + " ");  
 inorderHelper(node.right);  
 }  
 }  
  
 // Main method to test the AVL tree  
 public static void main(String[] args) {  
 AVLTree1 tree = new AVLTree1();  
  
 int[] keys = { 10, 20, 30, 40, 50, 25 };  
  
 for (int key : keys) {  
 tree.insert(key);  
 }  
  
 System.*out*.println("In-order traversal of the AVL Tree:");  
 tree.inorderTraversal();  
 }  
}

output: