Task 1:

Write Algo for AVL tree

1. **Insertion**:
   * Insert the node as you would in a regular Binary Search Tree.
   * After insertion, check the balance factor of each node.
   * If any node’s balance factor is outside the range [-1, 1], perform the necessary rotation to balance the tree.
2. **Balancing**:
   * **Left-Left Case (LL)**: If the left subtree is taller and the left child of the left subtree is unbalanced, perform a **right rotation**.
   * **Right-Right Case (RR)**: If the right subtree is taller and the right child of the right subtree is unbalanced, perform a **left rotation**.
   * **Left-Right Case (LR)**: If the left subtree is taller and the right child of the left subtree is unbalanced, perform a **left rotation** followed by a **right rotation**.
   * **Right-Left Case (RL)**: If the right subtree is taller and the left child of the right subtree is unbalanced, perform a **right rotation** followed by a **left rotation**.
3. **Height Update**:
   * After every operation (insertion or rotation), update the height of the affected nodes.
4. **Searching**:
   * Searching works like in a regular Binary Search Tree.

Task 2:

Write code for AVL tree

class AVLTree {

// Node structure for AVL Tree

class Node {

int key, height;

Node left, right;

Node(int d) {

key = d;

height = 1;

}

}

private Node root;

// Get the height of a node

int height(Node N) {

if (N == null) return 0;

return N.height;

}

// Get the balance factor of a node

int getBalance(Node N) {

if (N == null) return 0;

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

}

// Right rotate the subtree rooted at node

Node rightRotate(Node y) {

Node x = y.left;

Node 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 the new root

return x;

}

// Left rotate the subtree rooted at node

Node leftRotate(Node x) {

Node y = x.right;

Node 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 the new root

return y;

}

// Insert a node in the AVL tree

Node insert(Node node, int key) {

// 1. Perform the normal BST insertion

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 // Duplicate keys are not allowed

return node;

// 2. Update the height of the current node

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

// 3. Get the balance factor of this node

int balance = getBalance(node);

// If this node becomes unbalanced, then there are 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 the (unchanged) node pointer

return node;

}

// Method to insert a key in the AVL tree

public void insert(int key) {

root = insert(root, key);

}

// Method to print the tree in Inorder traversal

public void inorder(Node root) {

if (root != null) {

inorder(root.left);

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

inorder(root.right);

}

}

// Main method

public static void main(String[] args) {

AVLTree tree = new AVLTree();

// Inserting nodes into the AVL Tree

tree.insert(10);

tree.insert(20);

tree.insert(30);

tree.insert(15);

tree.insert(25);

System.out.println("Inorder traversal of the AVL Tree:");

tree.inorder(tree.root); // Display the AVL tree in inorder

}

}

Task 3: Insert operation task

 **BST Insert**: Insert the new node like you would in a regular Binary Search Tree (BST).

 **Update Heights**: After the insertion, update the height of the current node.

 **Check Balance Factor**: Calculate the balance factor of the current node.

 **Apply Rotations**:

* If the balance factor is > 1, it indicates a **Left-Left (LL)** imbalance, so perform a **Right Rotation**.
* If the balance factor is < -1, it indicates a **Right-Right (RR)** imbalance, so perform a **Left Rotation**.
* If the balance factor is > 1 but the **left child’s right subtree** is taller, it indicates a **Left-Right (LR)** imbalance, so perform a **Left Rotation** followed by a **Right Rotation**.
* If the balance factor is < -1 but the **right child’s left subtree** is taller, it indicates a **Right-Left (RL)** imbalance, so perform a **Right Rotation** followed by a **Left Rotation**.

class AVLTree {

// Node class to represent each node in the tree

class Node {

int key, height;

Node left, right;

Node(int d) {

key = d;

height = 1; // New node is initially added at leaf

}

}

private Node root;

// Get height of the node

int height(Node N) {

if (N == null) return 0;

return N.height;

}

// Get balance factor of a node

int getBalance(Node N) {

if (N == null) return 0;

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

}

// Right rotate the tree rooted at 'y'

Node rightRotate(Node y) {

Node x = y.left;

Node 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 the new root

return x;

}

// Left rotate the tree rooted at 'x'

Node leftRotate(Node x) {

Node y = x.right;

Node 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 the new root

return y;

}

// Insert a new key into the AVL tree

Node insert(Node node, int key) {

// 1. Perform normal BST insertion

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 {

// Duplicate keys are not allowed

return node;

}

// 2. Update the height of the ancestor node

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

// 3. Get the balance factor of this node (to check if it became unbalanced)

int balance = getBalance(node);

// If this node becomes unbalanced, then there are 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 the (unchanged) node pointer

return node;

}

// Method to insert a key into the AVL tree

public void insert(int key) {

root = insert(root, key);

}

// Inorder traversal to print the AVL Tree

public void inorder(Node root) {

if (root != null) {

inorder(root.left);

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

inorder(root.right);

}

}

// Main method to test the insertion

public static void main(String[] args) {

AVLTree tree = new AVLTree();

// Insert nodes into the AVL Tree

tree.insert(10);

tree.insert(20);

tree.insert(30);

tree.insert(15);

tree.insert(25);

// Display the AVL Tree in Inorder Traversal

System.out.println("Inorder traversal of the AVL Tree:");

tree.inorder(tree.root); // Output will be in sorted order

}

}