AVL Trees

AVL Tree is a self-balancing BST.

Balance Factor in AVL tree = height(left Subtree) - height(right Subtree) and always equal to {-1, 0, 1}

There are 4 cases of rotation in an AVL Tree:

a. LL Case
   - Right Rotate

b. RR Case
   - Left Rotate

c. LR Case
   - Left Rotate
   - Right Rotate

d. RL Case
   - Right Rotate
   - Left Rotate

AVL Tree Code (Insertion)

```
public class AVLTrees {
    static class Node {
        int data, height;
        Node left, right;

        Node(int data) {
            this.data = data;
            height = 1;
        }
    }

    public static Node root;

    public static int height(Node root) {
        if (root == null)
            return 0;
```

```
        return root.height;
    }

    // Right rotate subtree rooted with y
    public static Node rightRotate(Node y) {
        Node x = y.left;
        Node T2 = x.right;

        // rotation using 3 nodes
        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;

        // x is new root
        return x;
    }

    // Left rotate subtree rooted with x
    public static Node leftRotate(Node x) {
        Node y = x.right;
        Node T2 = y.left;

        // rotation using 3 nodes
        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;

        // y is new root
        return y;
    }

    // Get Balance factor of node
    public static int getBalance(Node root) {
        if (root == null)
```

```
            return 0;

        return height(root.left) - height(root.right);
    }

    public static Node insert(Node root, int key) {
        if (root == null)
            return new Node(key);

        if (key < root.data)
            root.left = insert(root.left, key);
        else if (key > root.data)
            root.right = insert(root.right, key);
        else
            return root; // Duplicate keys not allowed

        // Update root height
        root.height = 1 + Math.max(height(root.left), height(root.right));

        // Get root's balance factor
        int bf = getBalance(root);

        // Left Left Case
        if (bf > 1 && key < root.left.data)
            return rightRotate(root);

        // Right Right Case
        if (bf < -1 && key > root.right.data)
            return leftRotate(root);

        // Left Right Case
        if (bf > 1 && key > root.left.data) {
            root.left = leftRotate(root.left);
            return rightRotate(root);
        }

        // Right Left Case
        if (bf < -1 && key < root.right.data) {
            root.right = rightRotate(root.right);
            return leftRotate(root);
```

```
        }

        return root; //returned if AVL balanced
    }


    public static void preorder(Node root) {
        if(root == null) {
            return;
        }

        System.out.print(root.data + " ");
        preorder(root.left);
        preorder(root.right);
    }

    public static void main(String[] args) {
        root = insert(root, 10);
        root = insert(root, 20);
        root = insert(root, 30);
        root = insert(root, 40);
        root = insert(root, 50);
        root = insert(root, 25);

        /*          AVL Tree


                     30
                     / \
                   20  40
                   / \   \
                 10  25  50


        */

        preorder(root);
    }
}
```

AVL Tree Deletion

BST delete is a recursive function in which, after deletion, we get pointers to all ancestors one by one in a bottom up manner. So we don't need a parent pointer to travel up. The recursive code itself travels up and visits all the ancestors of the deleted node.

a. *Perform the normal BST deletion.*
b. *The current node must be one of the ancestors of the deleted node. Update the height of the current node.*
c. *Get the balance factor (left subtree height – right subtree height) of the current node.*
d. *If the balance factor is greater than 1, then the current node is unbalanced and we are either in the Left Left case or Left Right case. To check whether it is Left Left case or Left Right case, get the balance factor of the left subtree. If the balance factor of the left subtree is greater than or equal to 0, then it is Left Left case, else Left Right case.*
e. *If the balance factor is less than -1, then the current node is unbalanced and we are either in the Right Right case or Right Left case. To check whether it is a Right Right case or Right Left case, get the balance factor of the right subtree. If the balance factor of the right subtree is smaller than or equal to 0, then it is Right Right case, else Right Left case.*

```
//for non-empty BST, return the node with MIN data
   public static Node getMinNode(Node root) {
       Node curr = root;
       //MIN data is at left-most node
       while (curr.left != null)
       curr = curr.left;
       return curr;
   }
   public static Node deleteNode(Node root, int key) {
       // perform usual BST delete
       if (root == null) {
           return root;
       }

       // key < root's data => then it lies in left subtree
       if (key < root.data) {
           root.left = deleteNode(root.left, key);
       }
       // key > root's data => then it lies in right subtree
       else if (key > root.data) {
           root.right = deleteNode(root.right, key);
       }
       // key = root's data => then this is the node to be deleted
       else {
            // node with only one child or no child
```

```
        if ((root.left == null) || (root.right == null)) {
            Node temp = null;
            if (temp == root.left)
                temp = root.right;
            else
                temp = root.left;
            // No child case
            if (temp == null)
            {
                temp = root;
                root = null;
            }
            else // One child case
                root = temp; // Copy the contents of
                             // the non-empty child
        }
        else {
            // node with two children: Get the inorder
            // successor (smallest in the right subtree)
            Node temp = getMinNode(root.right);
            // Copy the inorder successor's data to this node
            root.data = temp.data;
            // Delete the inorder successor
            root.right = deleteNode(root.right, temp.data);
        }
    }
    // If the tree had only one node then return
    if (root == null)
        return root;
    // update height of curr node
    root.height = Math.max(height(root.left), height(root.right)) + 1;
    // get balance factor of this node (to check for unbalanced)
    int bf = getBalance(root);
    // If this node becomes unbalanced, then there are 4 cases
    // Left Left Case
    if (bf > 1 && getBalance(root.left) >= 0)
        return rightRotate(root);
    // Left Right Case
    if (bf > 1 && getBalance(root.left) < 0)
    {
```

```
            root.left = leftRotate(root.left);
            return rightRotate(root);
        }
        // Right Right Case
        if (bf < -1 && getBalance(root.right) <= 0)
            return leftRotate(root);
        // Right Left Case
        if (bf < -1 && getBalance(root.right) > 0)
        {
            root.right = rightRotate(root.right);
            return leftRotate(root);
        }
        return root;
    }
```