

## **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;
y.height = Math.max(height(y.left), height(y.right)) + 1;
x.height = Math.max(height(x.left), height(x.right)) + 1;
x.height = Math.max(height(x.left), height(x.right)) + 1;
y.height = Math.max(height(y.left), height(y.right)) + 1;
```



```
return 0;
   return new Node (key);
root.height = 1 + Math.max(height(root.left), height(root.right));
   return rightRotate(root);
    return leftRotate(root);
   root.left = leftRotate(root.left);
   return rightRotate(root);
   return leftRotate(root);
```



```
preorder(root.left);
preorder(root.right);
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.

```
public static Node getMinNode(Node root) {
    curr = curr.left;
        root.left = deleteNode(root.left, key);
```



```
((root.left == null) || (root.right == null)) {
        if (temp == root.left)
        Node temp = getMinNode(root.right);
int bf = getBalance(root);
   return rightRotate(root);
```



```
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;
}
```

## APNA COLLEGE