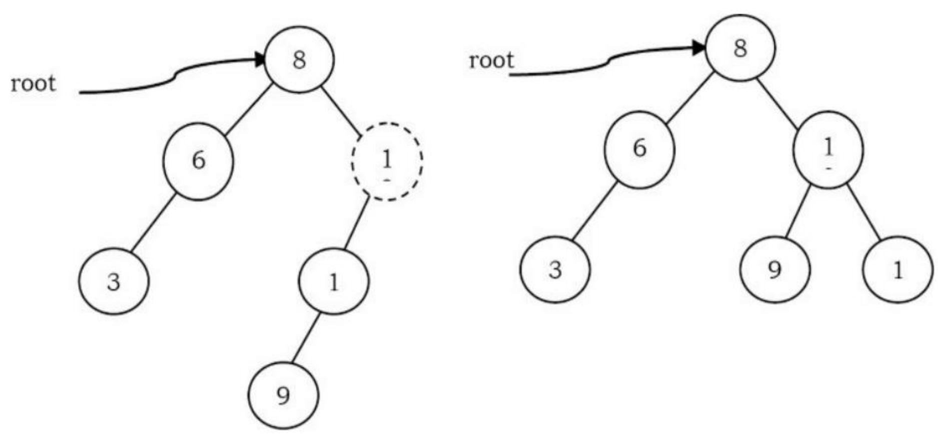
Name	**Hatim Yusuf Sawai**
UID no.	**2021300108**
Experiment No.	**7**

AIM:	**Program on AVL Tree**
Program 1	
PROBLEM STATEMENT:	Write a program to demonstrate insertion into an avl tree using **single & double rotations**
THEORY:	**AVL (Adelson-Velskii & Landis) Tree:** AVL tree is a self-balancing Binary Search Tree (**BST**) where the difference between heights of left and right subtrees cannot be more than **1** for all nodes. That means, the **balancing factor** should be **< 2** or **> -2** for every node of the tree. **Example:** From below binary search trees, the left one is not an AVL tree, whereas the right binary search tree is an AVL tree: 

Advantages of AVL Tree:

Most **BST** operations (insertion, deletion or traversal) take **O(h)** time where **h** is the height of the BST. If we make sure that the height of the tree remains **O(log(n))** after every insertion and deletion, then we can guarantee an upper bound of **O(log(n))** for all these operations. The height of an **AVL** tree is always **O(log(n))** where **n** is the number of nodes in the tree.

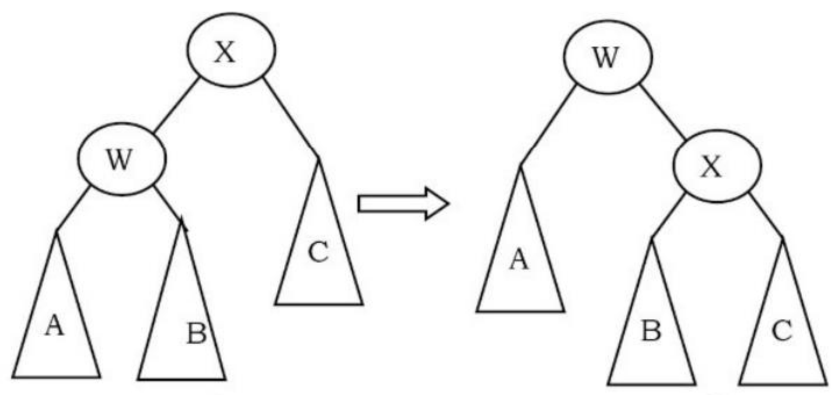
Rotations in AVL Tree:

If the AVL tree property is violated at a node X, it means that the heights of left(X) and right(X) differ by exactly 2. This is because, if we balance the AVL tree every time, then at any point, the difference in heights of left(X) and right(X) differ by exactly 2. Rotations is the technique used for restoring the AVL tree property. This means, we need to apply the rotations for the node X.

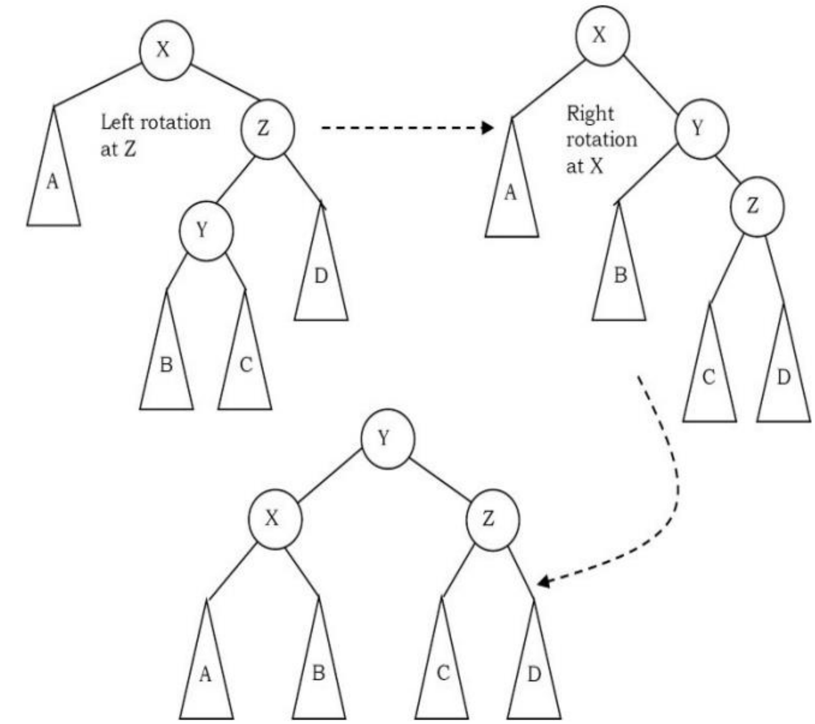
Types of Violations & Corresponding Rotations:

1. An insertion into the **left** subtree of the **left** child of X. **(LL)**

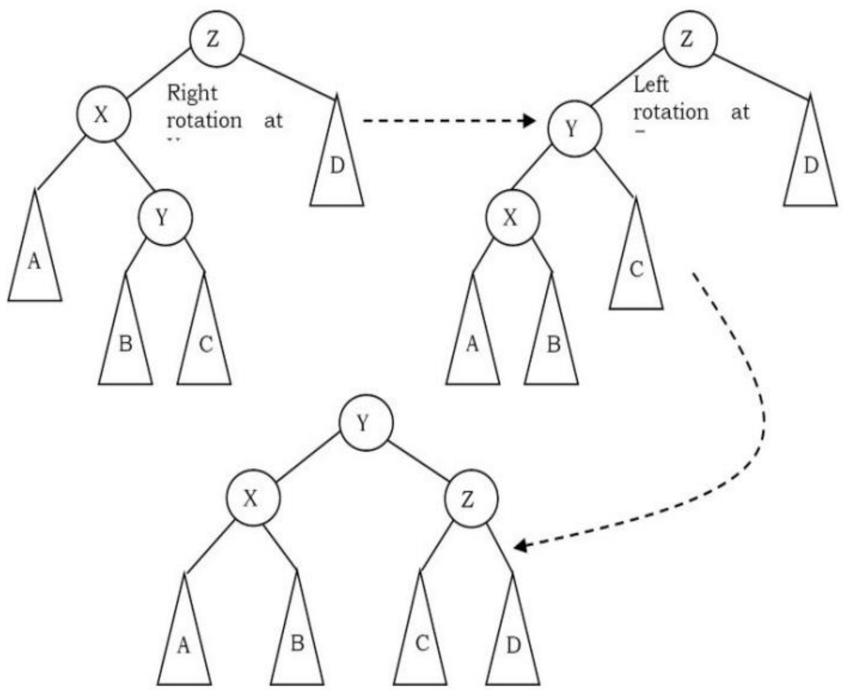
After performing LL rotation, we get tree on right:

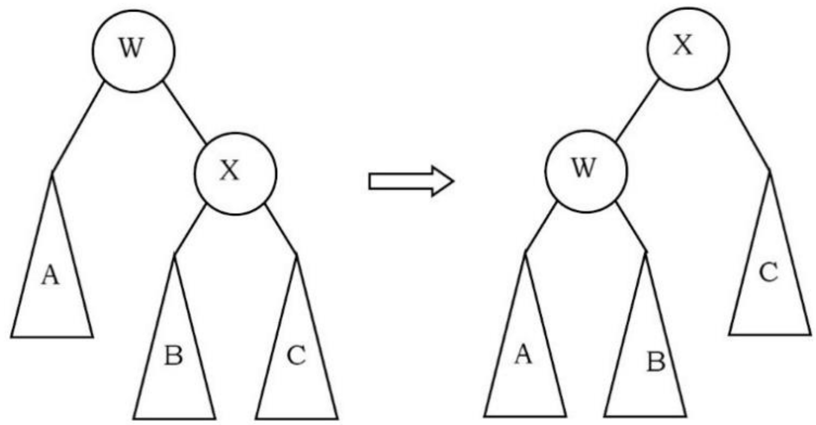


2. An insertion into the **right** subtree of the **left** child of X. **(RL)**
After performing RL rotation, we get tree on right:



3. An insertion into the **left** subtree of the **right** child of X. (**LR)**
After performing LR rotation, we get tree on right:



	4. An insertion into the **right** subtree of the **right** child of X. **(RR)** After performing RR rotation, we get tree on right: 
ALGORITHM:	1. Create AvlTree Class with inner Node class 2. Create 2 ref vars: right, left, int var data & height 3. Initialize right, left to null & height = 1 4. Initialize root node to null **height Method:** 1. if root=null: 2. return 0 3. else return root.height **SingleLL Method:** 1. Initialize node temp to root.left 2. root.left = temp.right 3. temp.right = root 4. Update **height** for both root & temp nodes 5. return temp **SingleRR Method:** 1. Initialize node temp to root.right 2. root.right = temp.left 3. temp.left = root 4. Update **height** for both root & temp nodes 5. return temp

DoubleLR Method:

1. root.*left* = **SingleRR**(root.*left*);
2. return **SingleLL**(root);

DoubleRL Method:

1. root.*right* = **SingleLL**(root.*right*);
2. return **SingleRR**(root);

getBF Method:

1. if root=null:
2. return 0
3. else return **height**(root.left) – **height**(root.right)

insert Method:

1. This method helps to make the recursive call to **inserter** Method

inserter Method:

1. if root is null:
2. root = new node(data)
3. return root
4. if data is less than root.data
5. root.*left* = inserter(root.*left*, data)
6. else if data is more than root.data
7. root.*right* = inserter(root.*right*, data)
8. Else return the root
9. Update **height** for root
10. initialize bal = **getBF**(root)
11. if bal>1 and data<root.left.data:
12. return **SingleLL**(root)
13. if bal<-1 and data>root.right.data:
14. return **SingleRR**(root)
15. if bal>1 and data>root.left.data:
16. return **DoubleLR**(root)
17. if bal<-1 and data<root.right.data:
18. return **DoubleRL**(root)
19. return root

minValue Method:

1. initialize minv equal to root.data
2. loop till root.left is not null:

a. minv = root.left.data

b. root = root.left

3. return minv

PreOrder Method:

1. if root is not null
2. print root.data
3. recurr: PreOrder(root.left)
4. recurr: PreOrder(root.right)

InOrder Method:

1. if root is not null
2. recurr: PreOrder(root.left)
3. print root.data
4. recurr: PreOrder(root.right)

PostOrder Method:

1. if root is not null
2. recurr: PreOrder(root.left)
3. recurr: PreOrder(root.right)
4. print root.data

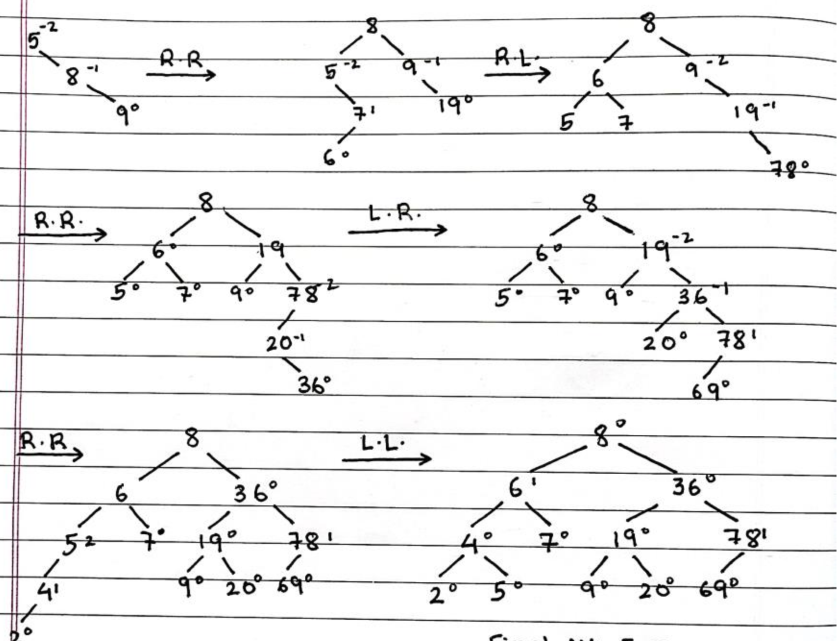
PROBLEM SOLVING:

→ Hatim Sawai

AVL Tree:

→ Insertion using all 4 rotations:

Input: 5,8,9,19,7,6,78,20,36,69,4,2



Final AVL Tree:

PreOrder(Final) = 8,6,4,2,5,7,36,19,9,20,78,69

PROGRAM:

AVLCheck.java:

```
import java.util.Scanner;
import avltreeds.AvlTree;
public class AVLCheck {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        AvlTree avl = new AvlTree();
        int choice,flag;
        int n,d;
        while(true) {
            System.out.println("Select 1 operation:\n1. Insert\t2.
Delete\n3. PreOrder\t4. InOrder\t5. PostOrder");
            choice = sc.nextInt();
            switch(choice) {
                case 1:
```

	`` System.out.print("Enter no. of elements to insert: "); n = sc.nextInt(); System.out.print("Enter the elements: "); for (int i = 0; i < n; i++) { avl.insert(sc.nextInt()); System.out.print("Tree (pre): "); avl.PreOrder(avl.root); System.out.println(); } break; ```

```
                    System.out.print("Enter no. of elements to insert: ");
                    n = sc.nextInt();
                    System.out.print("Enter the elements: ");
                    for (int i = 0; i < n; i++) {
                        avl.insert(sc.nextInt());
                        System.out.print("Tree (pre): ");
                        avl.PreOrder(avl.root);
                        System.out.println();
                    }
                    break;
                case 2:
                    System.out.print("Enter the element to delete: ");
                    d = sc.nextInt();
                    avl.delete(avl.root,d);
                    System.out.print("InOrder: ");
                    avl.InOrder(avl.root);
                    System.out.println();
                    break;
                case 3:
                    System.out.print("PreOrder: ");
                    avl.PreOrder(avl.root);
                    System.out.println();
                    break;
                case 4:
                    System.out.print("InOrder: ");
                    avl.InOrder(avl.root);
                    System.out.println();
                    break;
                case 5:
                    System.out.print("PostOrder: ");
                    avl.PostOrder(avl.root);
                    System.out.println();
                    break;
                case 6:
                    System.out.println("Size of the tree is " +
avl.size(avl.root));
```

```
                break;
            default:
                System.out.println("Invalid choice!");
        }
        System.out.println("Do you want to continue?\n1. Yes\t2. No");
        flag = sc.nextInt();
        if (flag == 2) {
            break;
        }
    }
    sc.close();
  }
}
```

AvlTree.java:

```
package avltreeds;
public class AvlTree {
    class Node {
        int data;
        Node left,right;
        int height;
        Node(int data) {
            this.data = data;
            left = right = null;
            height = 1;
        }
    }
    public Node root=null;
    public int height(Node root) {
        if (root == null)
            return 0;
        return root.height;
    }
    public Node SingleLL(Node root) {
        Node temp = root.left;
        root.left = temp.right;
```

```
        temp.right = root;
        root.height = Math.max(height(root.left), height(root.right)) + 1;
        temp.height = Math.max(height(temp.left), height(temp.right)) +
1;
        return temp;
    }
    public Node SingleRR(Node root) {
        Node temp = root.right;
        root.right = temp.left;
        temp.left = root;
        root.height = Math.max(height(root.left), height(root.right)) + 1;
        temp.height = Math.max(height(temp.left), height(temp.right)) +
1;
        return temp;
    }
    public Node DoubleLR(Node root) {
        root.left = SingleRR(root.left);
        return SingleLL(root);
    }
    public Node DoubleRL(Node root) {
        root.right = SingleLL(root.right);
        return SingleRR(root);
    }
    public int getBF(Node root) {
        if (root == null)
            return 0;
        return height(root.left) - height(root.right);
    }
    public void insert(int data) {
        root = inserter(root, data);
    }
    public Node inserter(Node root,int data) {
        if (root == null) {
            root = new Node(data);
            return root;
        }
```

	`` if (data < root.data) { root.left = inserter(root.left, data); } else if (data > root.data) { root.right = inserter(root.right, data); } else { return root; } root.height = 1 + Math.max(height(root.left), height(root.right)); int bal = getBF(root); if (bal>1 && data<root.left.data) { System.out.println("LL Rotation performed"); return SingleLL(root); } if (bal<-1 && data>root.right.data) { System.out.println("RR Rotation performed"); return SingleRR(root); } if (bal>1 && data>root.left.data) { System.out.println("LR Rotation performed"); return DoubleLR(root); } if (bal<-1 && data<root.right.data) { System.out.println("RL Rotation performed"); return DoubleRL(root); } return root; } public void delete(Node root,int data) { root = deleter(root, data); } public Node deleter(Node root,int data) { if (root == null) { return root; } if (data < root.data) { root.left = deleter(root.left, data); ```

	`` } else if (data > root.data) { root.right = deleter(root.right, data); } else { if (root.left == null) { return root.right; } else if (root.right == null) { return root.left; } root.data = minValue(root.right); root.right = deleter(root.right, root.data); } root.height = Math.max(height(root.left), height(root.right)) + 1; int bal = getBF(root); if (bal>1 && getBF(root.left)>=0) { return SingleLL(root); } if (bal>1 && getBF(root.left)<0) { return DoubleLR(root); } if (bal<-1 && getBF(root.right)<=0) { return SingleRR(root); } if (bal<-1 && getBF(root.right)>0) { return DoubleRL(root); } return root; } public int minValue(Node root) { int minv = root.data; while (root.left != null) { minv = root.left.data; root = root.left; } return minv; } public int size(Node root) { ```

```
        if (root == null)
            return 0;
        else
            return (size(root.left) + 1 + size(root.right));
    }
    public void PreOrder(Node root) {
        if (root!=null) {
            System.out.print(root.data + " ");
            PreOrder(root.left);
            PreOrder(root.right);
        }
    }
    public void InOrder(Node root) {
        if (root!=null) {
            InOrder(root.left);
            System.out.print(root.data + " ");
            InOrder(root.right);
        }
    }
    public void PostOrder(Node root) {
        if (root!=null) {
            PostOrder(root.left);
            PostOrder(root.right);
            System.out.print(root.data + " ");
        }
    }
}
```

OUTPUT:

```
PS D:\Data Structures\Exp7> cd "d:\Data Structures\
Select 1 operation:
1. Insert       2. Delete
3. PreOrder     4. InOrder      5. PostOrder
1
Enter no. of elements to insert: 12
Enter the elements: 5 8 9 19 7 6 78 20 36 69 4 2
Tree (pre): 5
Tree (pre): 5 8
RR Rotation performed
Tree (pre): 8 5 9
Tree (pre): 8 5 9 19
Tree (pre): 8 5 7 9 19
RL Rotation performed
Tree (pre): 8 6 5 7 9 19
RR Rotation performed
Tree (pre): 8 6 5 7 19 9 78
Tree (pre): 8 6 5 7 19 9 78 20
LR Rotation performed
Tree (pre): 8 6 5 7 19 9 36 20 78
RR Rotation performed
Tree (pre): 8 6 5 7 36 19 9 20 78 69
Tree (pre): 8 6 5 4 7 36 19 9 20 78 69
LL Rotation performed
Tree (pre): 8 6 4 2 5 7 36 19 9 20 78 69
Do you want to continue?
1. Yes  2. No
2
PS D:\Data Structures\Exp7>
```

CONCLUSION:	In this experiment, we learned how to insert elements into an avl tree while maintaining the balance using single (LL/RR) or double(LR/RL) rotations on the tree.