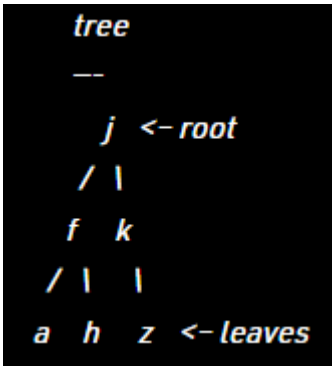
Name	Virinchi Sadashiv Shettigar
UID no.	2021300118
Experiment No.	7

AIM:	Program on AVL Tree
PROBLEM STATEMENT:	Write a program to demonstrate insertion into an avl tree using single & double rotations
THEORY:	**Tree** A tree is a popular data structure that is non-linear in nature. Unlike other data structures like an array, stack, queue, and linked list which are linear in nature, a tree represents a hierarchical structure. The ordering information of a tree is not important. A tree contains nodes and 2 pointers. These two pointers are the left child and the right child of the parent node. Let us understand the terms of tree in detail. **Root:** The root of a tree is the topmost node of the tree that has no parent node. There is only one root node in every tree. **Edge:** Edge acts as a link between the parent node and the child node. **Leaf:** A node that has no child is known as the leaf node. It is the last node of the tree. There can be multiple leaf nodes in a tree. Subtree: The subtree of a node is the tree considering that particular node as the root node. Depth: The depth of the node is the distance from the root node to that particular node. **Height:** The height of the node is the distance from that node to the deepest node of that subtree. **Height of tree:** The Height of the tree is the maximum height of any node. This is same as the height of root node.



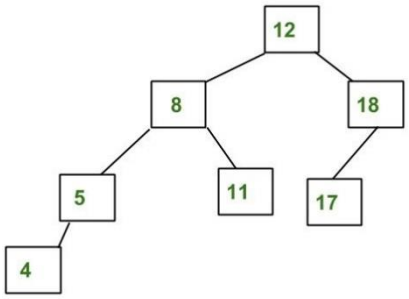
The types of Trees in the Data Structure.

1. Binary tree.
2. Binary Search Tree.
3. AVL Tree.
4. B-Tree.

AVL TREE

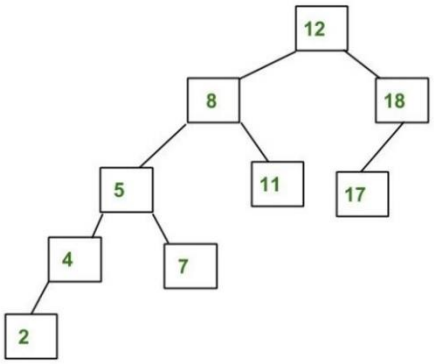
AVL(Adelson-Velskii and Landis) tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.

Example of AVL Tree:



The above tree is AVL because the differences between heights of left and right subtrees for every node are less than or equal to 1.

Example of a Tree that is NOT an AVL Tree:



The above tree is not AVL because the differences between the heights of the left and right subtrees for 8 and 12 are greater than 1.

Why AVL Trees?

- Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take O(h) time where h is the height of the BST.
- The cost of these operations may become O(n) for a skewed Binary tree. 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.

Insertion in AVL Tree:

To make sure that the given tree remains AVL after every insertion, we must augment the standard BST insert operation to perform some re-balancing.

Following are two basic operations that can be performed to balance a BST without violating the BST property (keys(left) < key(root) < keys(right)).

1. Left Rotation
2. Right Rotation

Steps to follow for insertion:

Let the newly inserted node be w

1. Perform standard BST insert for w.
2. Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the child of z that comes on the path from w to z and x be the grandchild of z

that comes on the path from w to z.

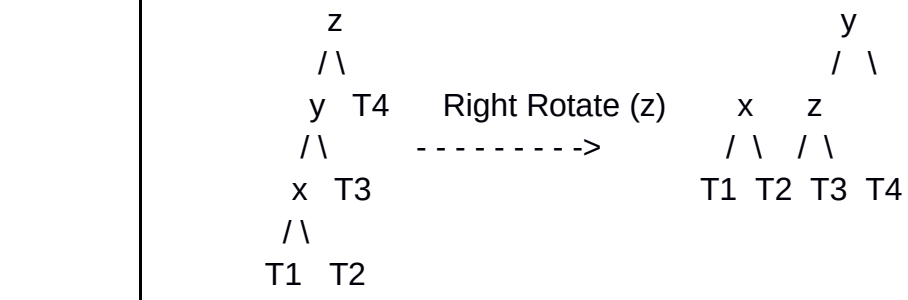
3. Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that need to be handled as x, y and z can be arranged in 4 ways.
4. Following are the possible 4 arrangements:

a. y is the left child of z and x is the left child of y (Left Left Case)
b. y is the left child of z and x is the right child of y (Left Right Case)
c. y is the right child of z and x is the right child of y (Right Right Case)
d. y is the right child of z and x is the left child of y (Right Left Case)

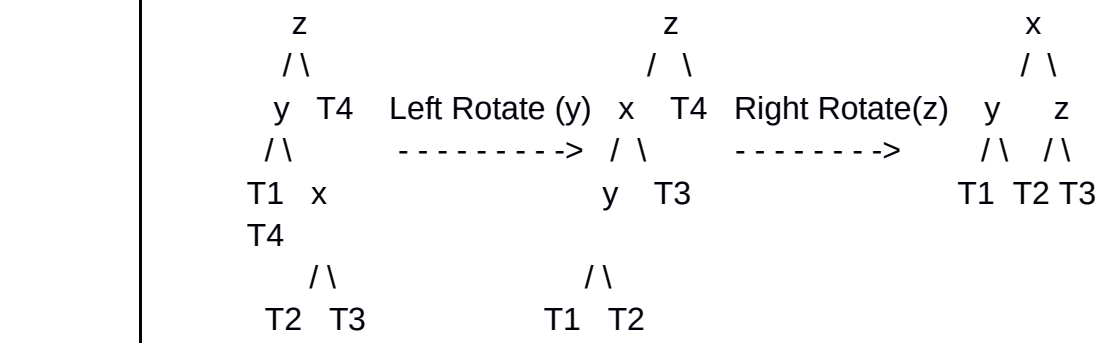
Following are the operations to be performed in above mentioned 4 cases. In all of the cases, we only need to re-balance the subtree rooted with z and the complete tree becomes balanced as the height of the subtree (After appropriate rotations) rooted with z becomes the same as it was before insertion.

A. <u>Left Left Case</u>

T1, T2, T3 and T4 are subtrees.



B. <u>Left Right Case</u>



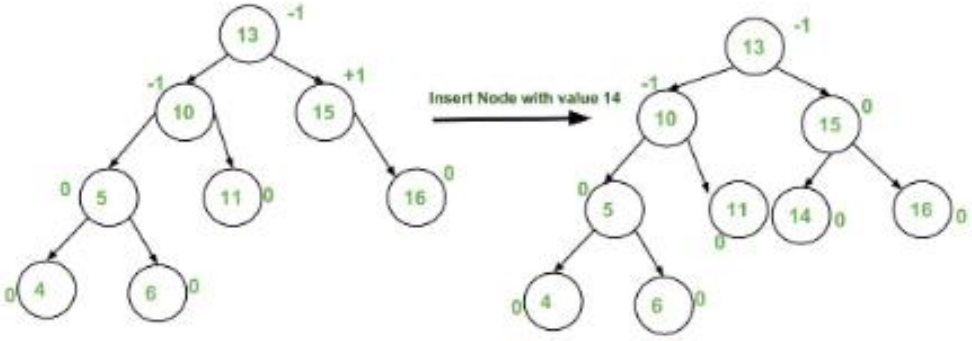
C. <u>Right Right Case</u>

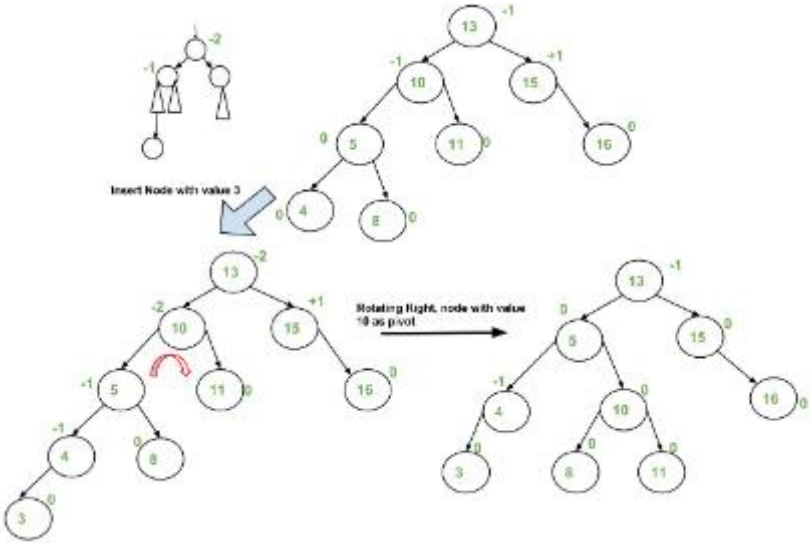
```
   T1   y     Left Rotate(z)       z      x
  /  \         - - - - - - - ->      / \    / \
 T2   x                            T1  T2 T3  T4
      / \
     T3  T4
```

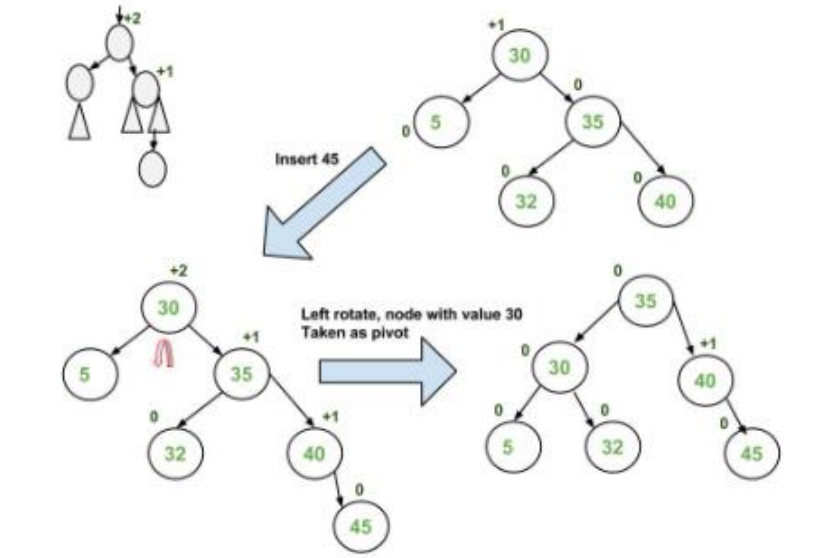
D. Right Left Case

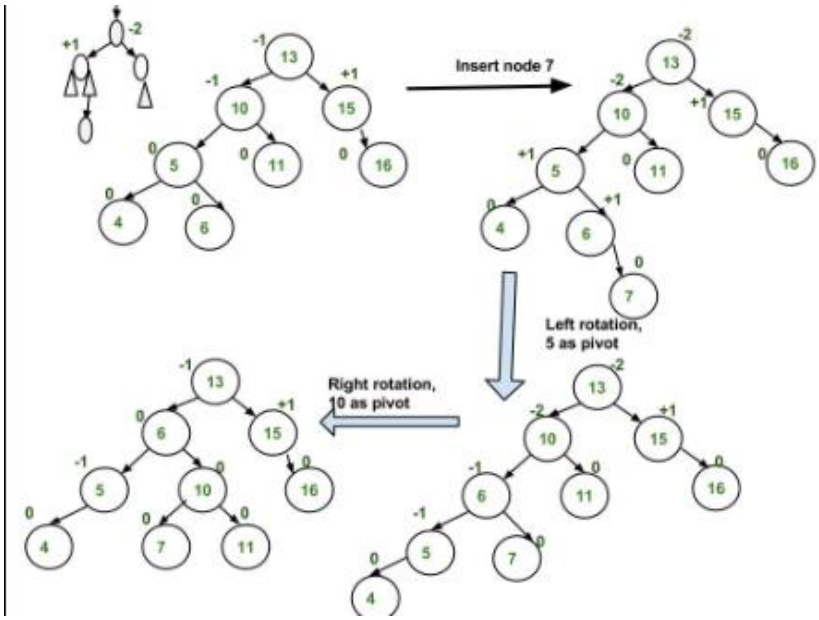
```
      z                              z                              x
     / \                            / \                           /  \
    T1   y   Right Rotate (y)      T1   x     Left Rotate(z)     z     y
      / \    - - - - - - - - - ->     / \      - - - - - - - ->    / \   / \
     x   T4                         T2   y                        T1  T2  T3
    T4
     / \                            / \
    T2   T3                        T3   T4
```

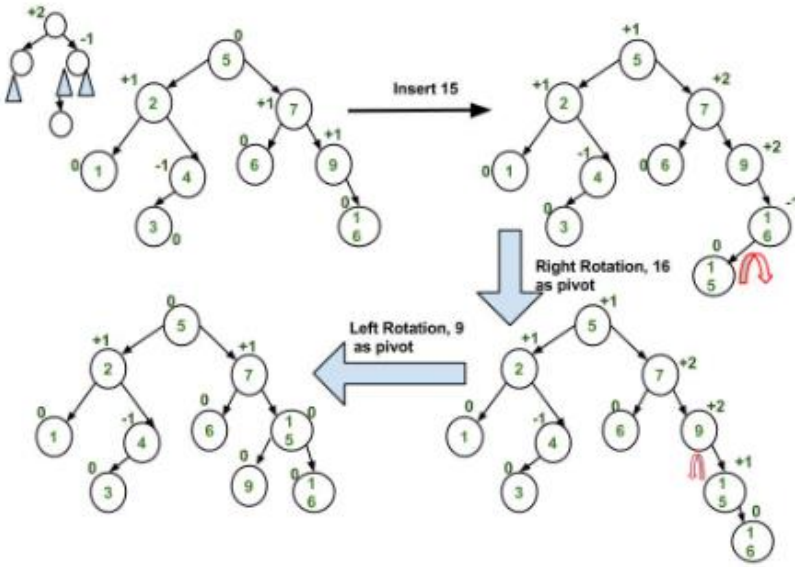
Illustration of Insertion at AVL Tree:






Insert 45
Left rotate, node with value 30
Taken as pivot


Insert node 7
Left rotation,
5 as pivot
Right rotation,
10 as pivot



ALGORITHM:

Class Exp7

1. Create new object of AVL
2. Enter number of nodes in AVL Tree
3. Enter all the nodes and insert them simultaenously
4. Print Preorder of the tree

Class AVL

- Class Node
  - Int height, key
  - Node left, right
  - Constructor Node(int d)
    - Key = d
    - Height = 1
- Node root
- Int height(Node n)
  - If n equals null, return 0 else return height of n
- Int max(int a, int b)
  - return a if a > b else b
- Node LL(Node a)
  1. Node b = a.left
  2. Node c = b.left
  3. If(b.right equals null), right of b equals a and left of a becomes null
  4. Else left of a becomes right of b and right of b equals a
  5. Height of c equals max of height of left of c and height of right of c +1
  6. Height of a equals max of height of left of a and height of right of a +1
  7. Height of b equals max of height of left of b and height of right of b +1
  8. return b

Node LR(Node a)

1. Node b = a.left
2. Node c = b.right
3. If(c.right equals null), right of c equals a and left of a becomes null
4. Else left of a becomes right of c and right of c equals a
5. If(c.left equals null), left of c equals b and right of b becomes null
6. Else right of b becomes left of c and left of c equals b
7. Height of a equals max of height of left of a and height of right of a +1
8. Height of b equals max of height of left of b and height of right of b +1
9. Height of c equals max of height of left of c and height of right of c +1
10. return c

Node RR(Node a)

1. Node b = a.right
2. Node c = b.right
3. If(b.left equals null), left of b equals a and right of a becomes null
4. Else right of a becomes left of b and left of b equals a
5. Height of c equals max of height of left of c and height of right of c +1
6. Height of a equals max of height of left of a and height of right of a +1
7. Height of b equals max of height of left of b and height of right of b +1
8. return b

Node RL(Node a)

1. Node b = a.right
2. Node c = b.left
3. If(c.left equals null), left of c equals a and right of a becomes null
4. Else right of a becomes left of c and left of c equals a
5. If(c.right equals null), right of c equals b and left of b becomes null
6. Else left of b becomes right of c and right of c equals b
7. Height of a equals max of height of left of a and height of right of a +1
8. Height of b equals max of height of left of b and height of right of b +1
9. Height of c equals max of height of left of c and height of right of c +1
10. return c

Int getBalance(Node N)

1. If N equals null, return 0, else return height of left of N – height of right of N

Void insert(int key)

1. Root = insert(root, key)

Node insert(Node node, int key)

1. If node equals null, return new object of Node class(key)
2. If key < key of node, left of node equals insert(node.left, key)
3. Else if key > key of node, right of node equals insert(node.right, key)
4. Else, return node
5. Height of node equals max of height of left of node and height of right of node +1
6. Int balance equals getbalance(node)
7. If balance > 1 and key < key of left of node, return LL(node)
8. Else if balance > 1 and key > key of left of node, return LR(node)
9. Else if balance < 1 and key > key of right of node, return RR(node)
10. Else, return RL(node)
11. Return node

Void Inorder()

1. Call inoder(root)

Void Inoder(Node root)

1. Call Inorder(root.left)
2. Print data of root
3. Call Inoder(root.right)

Void Preorder()

1. Call Preorder(root)

Void Preorder(Node root)

1. Print data of root
2. Call Preorder(root.left)
3. Call Preorder(root.right)

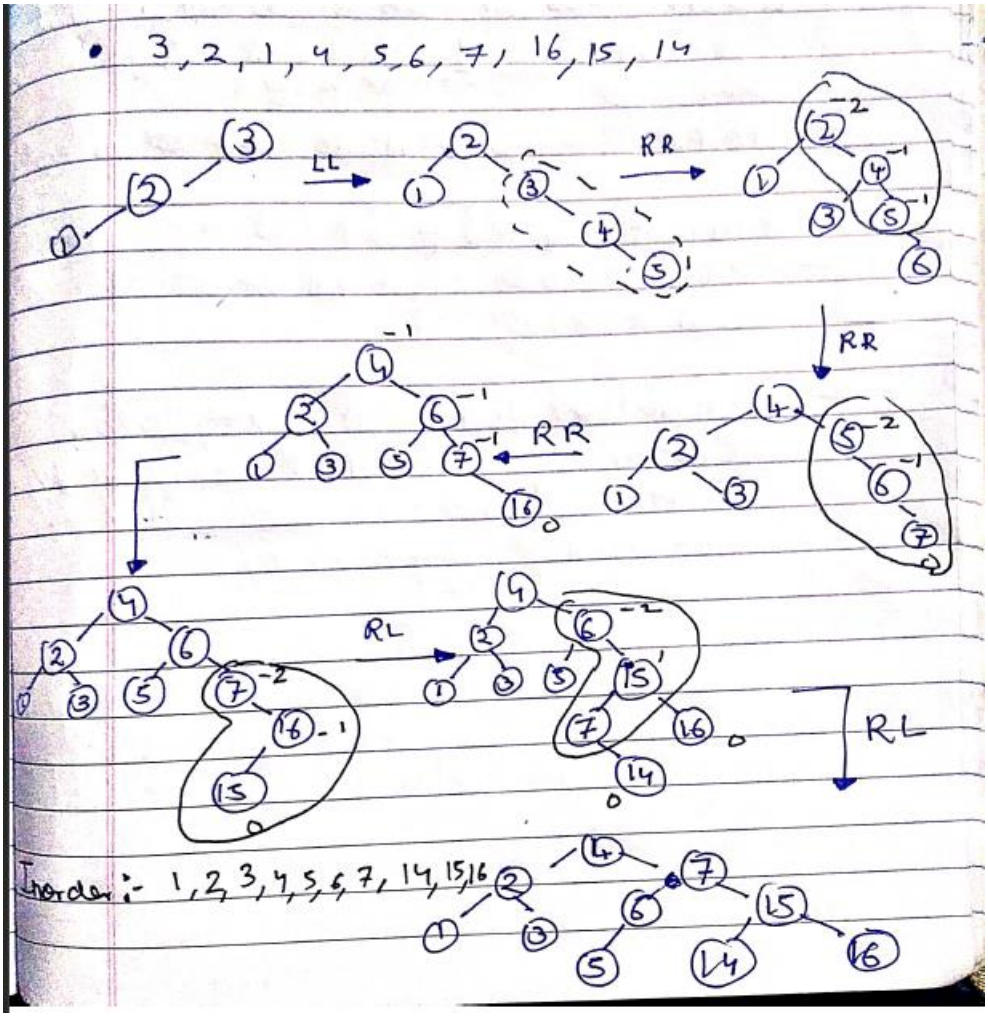
Void Postorder()

1. Call Postorder(root)

Void Postorder(Node root)

1. Call Postorder(root.left)
2. Call Postorder(root.right)
3. Print data of root

PROBLEM-SOLVING:



PROGRAM:

```
import java.util.*;

class AVL{
   class Node{
      int key, height;
      Node left, right;
      Node(int d){
         key = d;
         height = 1;
      }
   }
   Node root;
   int height(Node N){
      if (N == null)
         return 0;
      return N.height;
```

```
}
int max(int a, int b){

    if(a>b){
        return a;
    }
    else{
        return b;
    }
}
Node LL(Node a){
    Node b = a.left;
    Node c = b.left;
    if(b.right == null){
        b.right = a;
        a.left = null;
    }
    else{
        a.left = b.right;
        b.right = a;
    }
    System.out.print("LL rotation\n");
    c.height = max(height(c.left), height(c.right)) + 1;
    a.height = max(height(a.left), height(a.right)) + 1;
    b.height = max(height(b.left), height(b.right)) + 1;
    return b;
}
Node RR(Node a){
    Node b = a.right;
    Node c = b.right;
    if(b.left == null){
        b.left = a;
        a.right = null;
    }
    else{
        a.right=b.left ;
        b.left = a;
    }
    System.out.print("RR rotation\n");
    c.height = max(height(c.left), height(c.right)) + 1;
    a.height = max(height(a.left), height(a.right)) + 1;
    b.height = max(height(b.left), height(b.right)) + 1;
    return b;
}
Node LR(Node a){
    Node b = a.left;
```

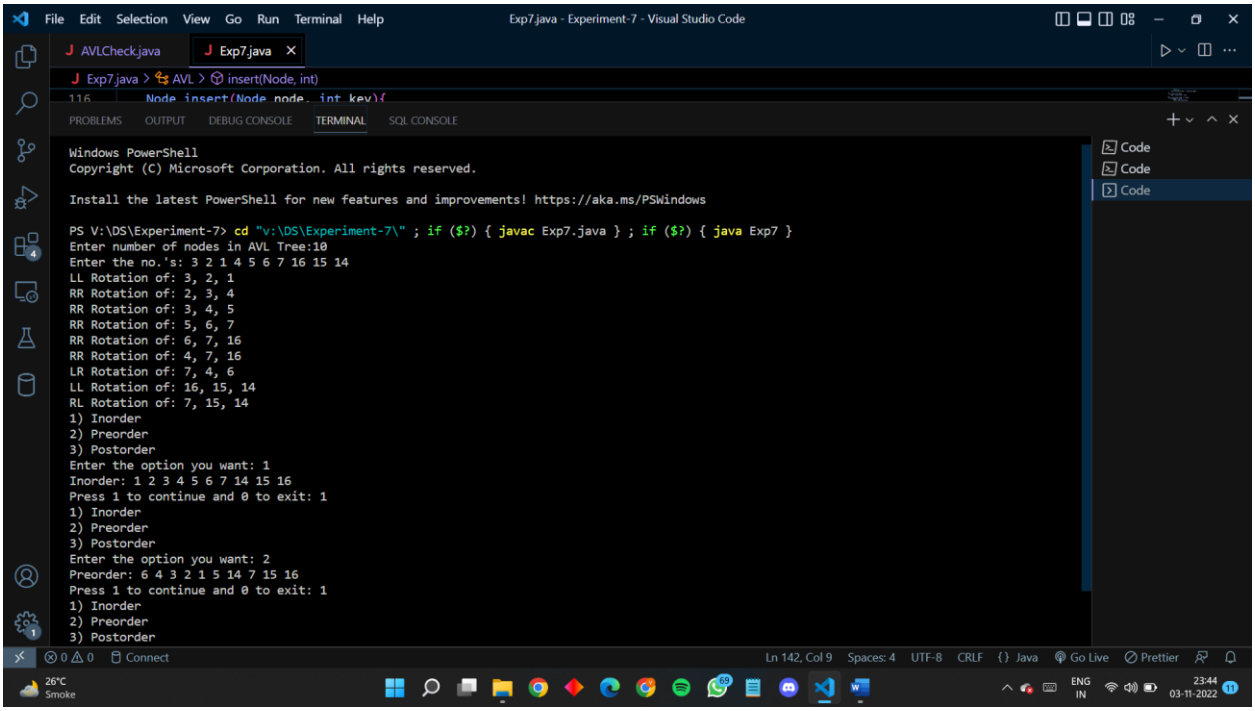
```
    Node c = b.right;
    if(c.right == null){
        c.right = a;
        a.left=null;
    }
    else{
        a.left = c.right;
        c.right = a;
    }
    if(c.left == null){
        c.left = b;
        b.right=null;
    }
    else{
        b.right = c.left;
        c.left = b;
    }

    System.out.print("LR rotation\n");
    a.height = max(height(a.left), height(a.right)) + 1;
    b.height = max(height(b.left), height(b.right)) + 1;
    c.height = max(height(c.left), height(c.right)) + 1;
    return c;
}
Node RL(Node a){
    Node b = a.right;
    Node c = b.left;
    if(c.left == null){
        c.left= a;
        a.right=null;
    }
    else{
        a.right = c.right;
        c.right = a;
    }
    if(c.right == null){
        c.right = b;
        b.left=null;
    }
    else{
        b.left = c.right;
        c.right = b;
    }
    System.out.print("RL rotation\n");
    a.height = max(height(a.left), height(a.right)) + 1;
    b.height = max(height(b.left), height(b.right)) + 1;
```

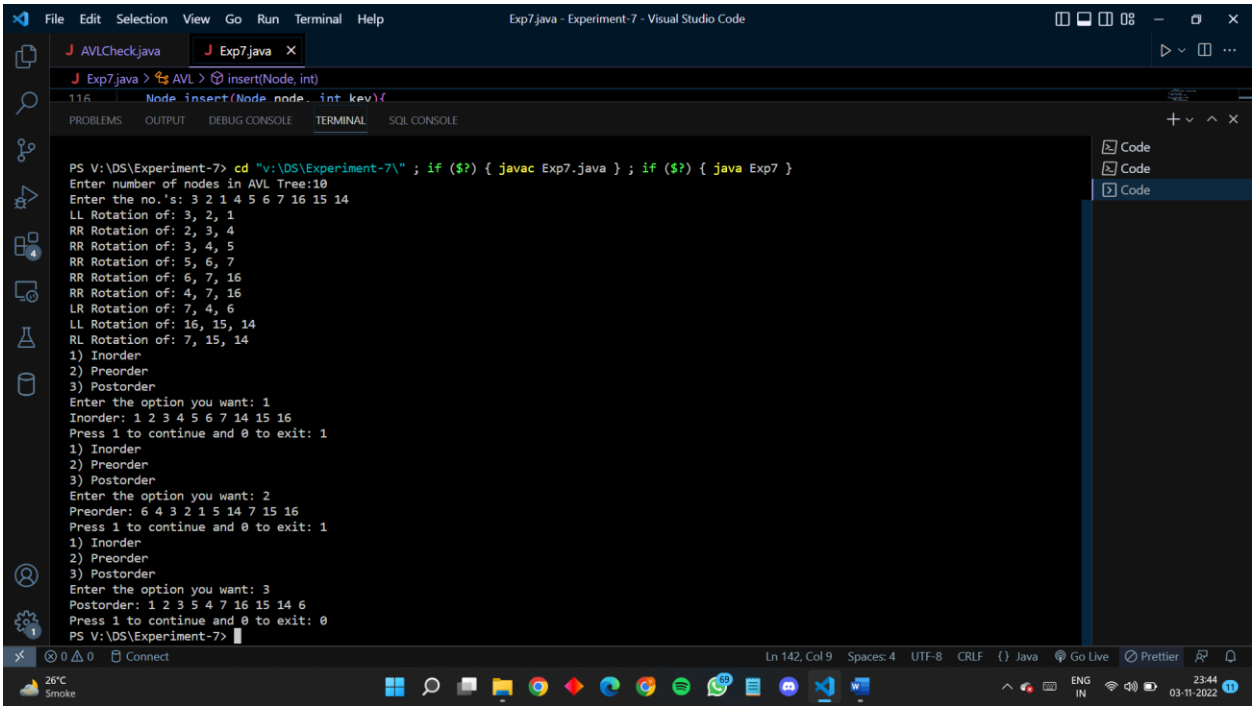
```
    c.height = max(height(c.left), height(c.right)) + 1;
    return c;
}
int getBalance(Node N){
    if (N == null)
        return 0;
    return height(N.left) - height(N.right);
}
void insert(int key){
    root = insert(root, key);
}
Node insert(Node node, int key){
    if (node == null)
        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
        return node;
    node.height = 1 + max(height(node.left), height(node.right));
    int balance = getBalance(node);
    if (balance > 1 && key < node.left.key){
        return LL(node);
    }
    if (balance < -1 && key > node.right.key){
        return RR(node);
    }
    if (balance > 1 && key > node.left.key){
        return LR(node);
    }
    if (balance < -1 && key < node.right.key){
        return RL(node);
    }
    return node;
}
void inorder(Node node){
    if (node != null){
        inorder(node.left);
        System.out.print(node.key + " ");
        inorder(node.right);
    }
}
void inorder(){
    inorder(root);
}
```

```
    void preorder(Node root){
            if(root==null){
                return;
            }
            System.out.print(root.key+" ");
            preorder(root.left);
            preorder(root.right);
        }
    void preorder(){
        preorder(root);
    }
        void postorder(Node root){
            if(root==null){
                return;
            }
            postorder(root.left);
            postorder(root.right);
            System.out.print(root.key+" ");
        }
        void postorder(){
        postorder(root);
    }
}

class Exp7{
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        AVL tree = new AVL();
        System.out.print("Enter number of nodes in AVL Tree:");
        int n = sc.nextInt();
        System.out.print("Enter the no.'s: ");
        for(int i = 0; i < n; i++){
            tree.insert(sc.nextInt());

        }
        int flag=1,choice;
        while(flag==1){
        System.out.print("1) Inorder\n2) Preorder\n3) Postorder");
        System.out.print("\nEnter the option you want: ");
        choice=sc.nextInt();
        switch(choice){
        case 1:
        System.out.print("Inorder: ");
        tree.inorder();
        break;
        case 2:
```

```
        System.out.print("Preorder: ");
        tree.preorder();
        break;
        case 3:
        System.out.print("Postorder: ");
        tree.postorder();
        break;
        default:
        System.out.print("Invalid choice");
        break;
        }
        System.out.print("\nPress 1 to continue and 0 to exit: ");
        flag=sc.nextInt();
        if(flag==0){
        break;
        }
    }
    sc.close();
  }
}
```

OUTPUT:





CONCLUSION:	From this experiment, I learned about the AVL tree. I also learned about balance factor and insertion of data in an AVL tree through the four cases of rotation that are LL, RR, RL, LR and was able to implement it.