**//Java program that performs insert, delete, search, traversal operations in AVL Tree**

```
import  java.util.Scanner;
class Node {
        int  data;
        int  h;
        Node left;
        Node right;
        public Node() {
                data = 0;
                h = 0;
                left = null;
                right = null;
        }
        public Node(int value) {
                this.data = value;
                h = 0;
                left = null;
                right = null;
        }
}

class ConstructAVLTree {
        private Node root;
        public ConstructAVLTree() {
                root = null;
        }
        public void insert(int  value) {
                root = insertRec(root, value);
        }
        public void delete(int key) {
                root = deleteRec(root, key);
                System.out.println("After deleting " + key + " Height = "+ getHeight(root));
        }
        private int getHeight(Node node) {
                return node==null ? -1 : node.h;
        }
        private int max(int l, int r) {
                return l>r? l : r;
        }
```

```
private Node insertRec(Node root, int value) {
	if(root==null)
		root = new Node(value);
	else if(value < root.data) {
		root.left = insertRec(root.left, value);
		if(getHeight(root.left) - getHeight(root.right) == 2)
			if(value < root.left.data)
				root = rotateWithLeftChild(root);
			else
				root = doubleWithLeftChild(root);
	}
	else if(value > root.data) {
		root.right = insertRec(root.right, value);
		if(getHeight(root.right) - getHeight(root.left) == 2)
			if(value > root.right.data)
				root = rotateWithRightChild(root);
			else
				root = doubleWithRightChild(root);
	}
	else
		;   //inserting a duplicate element is discarded
	root.h = max(getHeight(root.left), getHeight(root.right)) + 1;
	return root;
}

private Node deleteRec(Node root, int key) {
	if (root == null) {
			return null;
	}
	if (key < root.data) {
			root.left = deleteRec(root.left, key);
	}
	else if (key > root.data) {
			root.right = deleteRec(root.right, key);
	}
	else {   // Node to be deleted found
			if (root.left != null && root.right != null) {    // Two children
				Node minNode = findMin(root.right);
				root.data = minNode.data;
				root.right = deleteRec(root.right, minNode.data);
			}
			else {   // One or no child
				root = (root.left != null) ? root.left : root.right;
			}
	}
```

```
        if (root != null) {
                        root.h = Math.max(getHeight(root.left), getHeight(root.right)) + 1;

                        // Balance the tree
                        if (getHeight(root.left) - getHeight(root.right) == 2) {
                                if (getHeight(root.left.left) >= getHeight(root.left.right)) {
                                        root = rotateWithLeftChild(root);
                                }
                                else {
                                        root = doubleWithLeftChild(root);
                                }
                        }
                        else if (getHeight(root.right) - getHeight(root.left) == 2) {
                                if (getHeight(root.right.right) >= getHeight(root.right.left)) {
                                        root = rotateWithRightChild(root);
                                }
                                else {
                                        root = doubleWithRightChild(root);
                                }
                        }
        }
        return root;
}  //end of deleteRec() method

// Find the minimum node in a tree
private Node findMin(Node root) {
        if (root == null || root.left == null) {
                        return root;
        }
        return findMin(root.left);
}

private Node rotateWithLeftChild(Node k2) {
        Node k1 = k2.left;
        k2.left = k1. right;
        k1.right = k2;
        k2.h = max(getHeight(k2.left), getHeight(k2.right)) + 1;
        k1.h = max(getHeight(k1.left), k2.h) + 1;
        return k1;
}
```

```
private Node rotateWithRightChild(Node k1) {
        Node k2 = k1.right;
        k1.right = k2.left;
        k2.left = k1;
        k1.h = max(getHeight(k1.left), getHeight(k1.right)) + 1;
        k2.h = max(getHeight(k2.right), k1.h)+1;
        return k2;
}
private Node doubleWithLeftChild(Node k3) {
        k3.left = rotateWithRightChild( k3.left );
        return rotateWithLeftChild( k3 );
}
private Node doubleWithRightChild(Node k1) {
        k1.right = rotateWithLeftChild( k1.right );
        return rotateWithRightChild( k1 );
}
public void inorderTraversal() {
        inorderTraversal(root);
}
private void inorderTraversal(Node root) {
        if (root != null) {
                        inorderTraversal(root.left);
                        System.out.print(root.data+" ");
                        inorderTraversal(root.right);
        }
}
public void preorderTraversal() {
        preorderTraversal(root);
}
private void preorderTraversal(Node root) {
        if (root != null) {
                        System.out.print(root.data+" ");
                        preorderTraversal(root.left);
                        preorderTraversal(root.right);
        }
}
public void postorderTraversal() {
        postorderTraversal(root);
}
private void postorderTraversal(Node root) {
        if (root != null) {
                        postorderTraversal(root.left);
                        postorderTraversal(root.right);
                        System.out.print(root.data+" ");
        }
}
```

```
	public void search(int key) {
		Node n = searchRec(root, key);
		if(n == null) System.out.println("Key is not found in the Tree");
		else System.out.println("Key is found");
	}
	private Node searchRec(Node root, int key) {
		if (root==null || root.data==key) return root;
		else if(key < root.data) return searchRec(root.left, key);
		else return searchRec(root.right, key);
	}
}
public class AVLTree {
	public static void main(String[] args) {
		Scanner sc = new Scanner(System.in);
		ConstructAVLTree  avl = new ConstructAVLTree();
		while(true) {
			System.out.println("1. Insert an element");
			System.out.println("2. Delete an element");
			System.out.println("3. Search for an element");
			System.out.println("4. Inorder Traversal");
			System.out.println("5. Preorder Traversal");
			System.out.println("6. Postorder Traversal");
			System.out.println("7. Exit");
			System.out.println("Enter your choice: ");
			int  ch = sc.nextInt();
			switch(ch) {
				case 1:  System.out.println("Enter the element to be inserted:");
					int item = sc.nextInt();
					avl.insert(item);
					break;
				case 2:  System.out.println("Enter the element to be deleted:");
					int key = sc.nextInt();
					avl.delete(key);
					break;
				case 3:  System.out.println("Enter the key element to search");
					4key = sc.nextInt();
					avl.search(key);
					break;
				case 4:  System.out.println("Inorder Traversal:");
					avl.inorderTraversal();
					System.out.println();
					break;
				case 5:  System.out.println("Preorder Traversal:");
					avl.preorderTraversal();
					System.out.println();
					break;
```

```
                    case 6:  System.out.println("Postorder Traversal:");
                             avl.postorderTraversal();
                             System.out.println();
                             break;
                    case 7:  System.exit(0);
                }
            }
        }
}
```