/** * AVLTree class represents a self-balancing binary search tree. * The tree is constructed using Node objects. * The Left Right rotations are performed using the Node class. */ class AVLTree { Node root; // Root node of the tree

```
/**
 * Returns the height of the specified node.
 *
 * @param N the node whose height is to be calculated
 * @return the height of the node, or 0 if the node is null
 */
int height(Node nd) {
    if (nd == null)
        return 0;
    return nd.height;
}


/**
 * Calculates the balance factor of a given node in the AVL tree.
 * The balance factor is defined as the difference between the heights of the left and right
 *
 * @param N the node for which the balance factor is to be calculated
 * @return the balance factor of the node
 */
int getBalance(Node nd) {
    if (nd == null)
        return 0;
    return height(nd.left) - height(nd.right);
}


/**
 * Performs a right rotation on the specified node.
 *
 * @param y the node to be rotated
 * @return the new root node after rotation
 */
Node rightRotate(Node y) {

    Node x = y.left;        Node temp = x.right;

    // Perform rotation
    x.right = y;        y.left = temp;

    // Update heights
    y.height = Math.max(height(y.left), height(y.right)) + 1;
```

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

    // Return new root
    return x;
}

/**
 * Performs a left rotation on the specified node.
 *
 * @param x the node to be rotated
 * @return the new root node after rotation
 */
Node leftRotate(Node x) {
    Node y = x.right;        Node temp = y.left;

    // Perform rotation
    y.left = x;        x.right = temp;

    // Update heights
    x.height = Math.max(height(x.left), height(x.right)) + 1;
    y.height = Math.max(height(y.left), height(y.right)) + 1;

    // Return new root
    return y;
}

/**
 * Inserts a new node with the specified key into the AVL tree.
 *
 * @param node the root node of the tree
 * @param key the key value of the node to be inserted
 * @return the root node of the tree after insertion
 */
Node insert(Node node, int key) {
    // Perform BST insertion
    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 // Duplicate keys not allowed
        return node;

    // Update height of this ancestor node
```

```java
    node.height = 1 + Math.max(height(node.left), height(node.right));

    // Get the balance factor of this ancestor node
    // to check whether this node became unbalanced
    int balance = getBalance(node);

    // If this node becomes unbalanced, then there are 4 cases

    /* SINGLE ROTATION Cases */
    // LL Case
    if (balance > 1 && key < node.left.key)
        return rightRotate(node);

    // RR Case
    if (balance < -1 && key > node.right.key)
        return leftRotate(node);
    /* SINGLE ROTATION END Cases */

    /* DOUBLE ROTATION Cases */
    // LR Case
    if (balance > 1 && key > node.left.key) {
        node.left = leftRotate(node.left);
        return rightRotate(node); // rightRotate(node) to get new root node
    }

    // RL Case
    if (balance < -1 && key < node.right.key) {
        node.right = rightRotate(node.right);
        return leftRotate(node); // leftRotate(node) to get new root node
    }
    /* DOUBLE ROTATION END Cases */

    // No changes return the (unchanged) node pointer
    return node;
}

/**
 * Performs a pre-order traversal of the AVL tree starting from the given node.
 * Prints the keys of the nodes in the traversal order.
 *
 * @param node the starting node for the traversal
 */
void preOrder(Node node) {
    if (node != null) {
        System.out.print(node.key + " ");
        preOrder(node.left);
```

```
        preOrder(node.right);
    }
}
```

}