```java
package AVLTree;

import List.LNode;

/**
 * AVLTree class is a Binary Search Tree data structure that implements single and double, left and
 * right, AVL rotations to maintain a Height Balance factor of no more than 1. If an item is added to
 * the tree more than once; a counter, c, of TNode in incremented by one. Implemented operations are:
 *
 * insert       Insert a String into the tree.
 * inOrder      Perform an In-Order (Symmetric-Order) traversal of the tree.
 * isAVL        Checks if each node in the tree is AVL compliants, returns true or false.
 * delete       Find a node in the tree and remove it.
 *
 * When an inOrder traversal is performed, the word will be printed to the console with it's respective
 * number of occurrences nest to it (word - #).
 *
 * ******************* Valid Input ******************
 *
 * words        An LNode linear linked-list of Strings.
 *
 * *************** Public Operations *****************
 *
 * insert       Insert a given String into the tree.
 * inOrder      Perform an In-Order traversal of the tree, output to console.
 * isAVL        Returns true of the tree is AVL compliant, false if not.
 * delete       Find and remove a given string from the tree.
 *
 * ***************** Global Variables *****************
 *
 * data         Private variable for root pointer.
 *
 * @author Matt Laidman (5199807)
 * @version 1.0 (October 21, 2014)
 */

public class AVLTree {

    private TNode data;                                         // Private variable to store root pointer

    /**
     * Public constructor to call private buildTree function with list of words.
     *
     * @param words     An LNode linear linked-list storing words to add to tree.
     */

    public AVLTree (LNode words) {

        buildTree(words);                                       // Call buildTree method.
    }

    /**
     * Public insert method to call private insert function to add a String into the tree.
     *
     * @param item      String to add to tree.
     * @return          Root of tree with String added.
     */

    public TNode insert (String item) {

        data = insert(data, item);                              // Call private insert function.
        return data;                                            // Return root of tree
    }

    /**
     * Public inOrder method to call private method with private data variable.
     */

    public void inOrder ( ) {

        inOrder(data);
```

```
}

/**
 * Public isAVL function to call private isAVL with private data variable.
 *
 * @return          True if tree is AVL compliant, false if not.
 */

public boolean isAVL ( ) {

    return isAVL(data);
}

/**
 * Public delete to call private delete to remove a String from the tree.
 *
 * @param item      The String to remove from the tree.
 * @return          The root node of the tree.
 */

public TNode delete (String item) {

    data = delete(data, item);                              // Call private delete method
    return data;                                            // Return root node of tree
}

/**
 * Private delete function to locate and remove a String from the tree. Recursively navigates the
 * tree to target node, checks left and ride side heights as the recursion pops and calls appropriate
 * AVL rotation functions.
 *
 * @param tree      The local root of the tree to delete from (absolute root originally).
 * @param item      The String to locate in the tree and remove.
 * @return          The root TNode of the tree.
 */

private TNode delete (TNode tree, String item) {

    if (tree == null) {                                     // If node is null
        return null;                                        // Return null
    }
    if (item.compareTo(tree.key) < 0) {                     // If item is less than Node
        tree.left = delete(tree.left, item);                // Recursive call with left subtree
        if (height(tree.right) - height(tree.left) == 2) {  // If right subtree heavier
            if (height(tree.right.right) >= height(tree.right.left)) {
                tree = singleLeft(tree);                    // Right side (outside) of subtree
            } else {                                        // heavier, call singleLeft, otherwise
                tree = doubleLeft(tree);                    // call doubleLeft.
            }
        } else {                                            // Update height
            tree.height = Math.max(height(tree.right), height(tree.left)) + 1;
        }
    } else if (item.compareTo(tree.key) > 0) {              // If item is greater than Node
        tree.right = delete(tree.right, item);              // Recursive call with right subtree
        if (height(tree.left) - height(tree.right) == 2) {  // If left subtree heavier
            if (height(tree.left.left) >= height(tree.left.right)) {
                tree = singleRight(tree);                   // Left side (outside) of subtree
            } else {                                        // heaver, call singleRight, otherwise
                tree = doubleRight(tree);                   // call doubleRight
            }
        } else {                                            // Update height
            tree.height = Math.max(height(tree.right), height(tree.left)) + 1;
        }
    } else if (tree.left != null && tree.right != null) {   // If node has left and right children
        tree.key = successor(tree).key;                     // Copy successor key to key
        tree.right = delete(tree.right, tree.key);          // Delete successor
        if (height(tree.right) - height(tree.left) == 2) {  // If right subtree heavier
            if (height(tree.right.right) >= height(tree.right.left)) {
                tree = singleLeft(tree);                    // Right side (outside) of subtree
            } else {                                        // heaver, call singleLeft, otherwise
                tree = doubleLeft(tree);                    // call doubleLeft.
```

```
            }
        } else {                                                // Update height
            tree.height = Math.max(height(tree.right), height(tree.left)) + 1;
        }
    } else {                                                    // Node to delete has one or no children
        if (tree.left != null) {                                // If there is left child
            tree = tree.left;                                   // Copy left child to Node
        } else {
            tree = tree.right;                                  // Otherwise copy right child to Node
        }
    }
    return tree;                                                // Return node
}

/**
 * Private successor function performs the successor algorithm (right once, left as far as possible)
 * on a given node and returns it's successor
 *
 * @param tree      The TNode to locate successor of.
 * @return          The successor of the given TNode.
 */

private TNode successor (TNode tree) {

    tree = tree.right;                                          // Right once
    if (tree == null) {                                         // If null tree, return null
        return null;
    }
    while (tree.left != null) {                                 // Left as far as possible
        tree = tree.left;
    }
    return tree;                                                // Return successor
}

/**
 * Private isAVL function recursively checks every node in the tree to ensure that the entire tree
 * is AVL compliant (height balance of 1).
 *
 * @param tree      The current TNode to check.
 * @return          True if tree is AVL, false if not.
 */

private boolean isAVL (TNode tree) {

    return (tree == null) || (Math.abs(height(tree.left) - height(tree.right)) <= 1 &&
            isAVL(tree.left) && isAVL(tree.right));             // Check heights and call left and right
}

/**
 * Private inOrder function recursively performs an In-Order traversal of the tree (LVR).
 *
 * @param tree      The current TNode to enummerate.
 */

private void inOrder (TNode tree) {

    if (tree.left != null) {                                    // If tree has left child
        inOrder(tree.left);                                     // Recursive call left
    }
    System.out.println(tree.key + " - " + tree.c);              // Print node contents (visit)
    if (tree.right != null) {                                   // If tree has right child
        inOrder(tree.right);                                    // Recursive call right
    }
}

/**
 * Private insert function recursively adds a string to it's appropriate spot on the tree and
 * initiates the AVL rotations to rebalance the tree.
 *
 * @param tree      The TNode to add to.
 * @param item      The String to add.
```

```
 * @return          The tree with TNode added.
 */

private TNode insert (TNode tree, String item) {

    if (tree == null) {                                     // If tree is empty
        tree = new TNode(item);                             // Return item as root
    } else if (item.compareTo(tree.key) < 0) {              // If item less than current key
        tree.left = insert(tree.left, item);                // Recursive call left
        if (height(tree.left) - height(tree.right) == 2) {  // If left subtree heavier
            if (item.compareTo(tree.left.key) < 0) {        // If item added to left
                tree = singleRight(tree);                   // Single right rotation
            } else {
                tree = doubleRight(tree);                   // Otherwise double right rotation
            }
        } else {                                            // Update tree height
            tree.height = 1 + Math.max(height(tree.left), height(tree.right));
        }
    } else if (item.compareTo(tree.key) > 0) {              // If item less than current key
        tree.right = insert(tree.right, item);              // Recursive call right
        if (height(tree.right) - height(tree.left) == 2) {  // If right subtree heavier
            if (item.compareTo(tree.right.key) > 0) {       // If item greater than current key
                tree = singleLeft(tree);                    // Single left rotation
            } else {
                tree = doubleLeft(tree);                    // Otherwise double left rotation
            }
        } else {                                            // Update tree height
            tree.height = 1 + Math.max(height(tree.right), height(tree.left));
        }
    } else {                                                // If item is equal to current key
        tree.c++;                                           // Increase count
    }
    return tree;                                            // Return TNode
}

/**
 * Private singleRight function performs the single left to right AVL rotation on the tree, while
 * maintaining the binary search property.
 *
 * @param tree      The TNode to perform the rotation on.
 * @return          The TNode after the rotation.
 */

private TNode singleRight (TNode tree) {

    TNode ptr = tree.left;                                  // Current points left to left child's
    tree.left = ptr.right;                                  // right child
    ptr.right = tree;                                       // Left child points right to current
    tree.height = Math.max(height(tree.left), height(tree.right))+1;
    ptr.height = Math.max(height(ptr.left), tree.height)+1; // Update heights
    return ptr;                                             // Return current TNode
}

/**
 * Private doubleRight function performs the double left to right AVL rotation on the tree, while
 * maintaining the binary search property. This is accomplished using a single left rotation and a
 * single right rotation, from the respective singleLeft and singleRight functions.
 *
 * @param tree      The TNode to perform the rotation on.
 * @return          The TNode after the rotation.
 */

private TNode doubleRight (TNode tree) {

    tree.left = singleLeft(tree.left);                      // Left rotation on left child
    return singleRight(tree);                               // Right rotation on current
}

/**
 * Private singleLeft function performs the single right to left AVL rotation on the tree, while
 * maintaining the binary search property.
```

```
     *
     * @param tree      The TNode to perform the rotation on.
     * @return          The TNode after the rotation.
     */

    private TNode singleLeft (TNode tree) {

        TNode ptr = tree.right;                                 // Current points right to right child's
        tree.right = ptr.left;                                  // left child
        ptr.left = tree;                                        // Right child points left to current
        tree.height = Math.max(height(tree.left), height(tree.right))+1;
        ptr.height = Math.max(height(ptr.right), tree.height)+1;// Update heights
        return ptr;                                             // Return current TNode
    }

    /**
     * Private doubleLeft function performs the double right to left AVL rotation on the tree, while
     * maintaining the binary search property. This is accomplished using a single right rotation and a
     * single left rotation, from the respective singleRight and singleLeft functions.
     *
     * @param tree      The TNode to perform the rotation on.
     * @return          The TNode after the rotation.
     */

    private TNode doubleLeft (TNode tree) {

        tree.right = singleRight(tree.right);                   // Right rotation on right child
        return singleLeft(tree);                                // Left rotation on current
    }

    /**
     * Private height function returns the height of a given TNode in the tree. If the TNode is null, -1
     * is returned.
     *
     * @param node      The TNode to get the height of.
     * @return          The height of the TNode in the tree.
     */

    private int height(TNode node) {

        if (node == null) {                                     // If null TNode
            return -1;                                          // Return -1
        } else {
            return node.height;                                 // Otherwise return height
        }
    }

    /**
     * Private buildTree function builds the binary search tree by calling the insert function with each
     * word in the supplied LNode list of words.
     *
     * @param words     The list of words to build tree from.
     */

    private void buildTree (LNode words) {

        if (words == null) {                                    // If null word list
            throw new AVLTreeException();                       // Throw AVLTreeException
        }
        while (words != null) {                                 // While there are words in list
            data = insert(words.key);                           // Insert word into tree
            words = words.next;                                 // Next word
        }
    }

}
```