```java
package SplayTree;

import List.LNode;

/**
 * SplayTree class is a Binary Search Tree data structure that implements single and double, left and
 * right, AVL rotations to 'splay' a tree to the root any time that it is accessed. If an item is added
 * to the tree more than once; a counter, c, of TNode is incremented by one. Implemented operations are:
 *
 * insert       Insert a String into the tree.
 * find         Locate a String in the tree and promote it to the root.
 * inOrder      Perform an In-Order (Symmetric-Order) traversal of the tree.
 * preOrder     Perform a Pre-Order traversal of the tree.
 * delete       Find a String in the tree, promote it to the root, then remove it.
 *
 * When an inOrder/preOrder traversal is performed, the word will be printed to the console with it's
 * respective number of occurrences next to it (word - #).
 *
 * ****************** Valid Input ******************
 *
 * words        An LNode linear linked-list of Strings.
 *
 * *************** Public Operations *****************
 *
 * insert       Insert a given String into the tree.
 * find         Locate a given String in the tree and promote it to the root.
 * inOrder      Perform an In-Order traversal of the tree, output to console.
 * preOrder     Perform a Pre-Order traversal of the tree, output to console.
 * 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 (November 2, 2014)
 */

public class SplayTree {

    private TNode data;                                         // Private TNode root pointer

    /**
     * Public constructor to call private buildTree function with list of words.
     *
     * @param words     The list of words to build the tree from.
     */

    public SplayTree(LNode words) {

        buildTree(words);                                       // Call private buildTree function
    }

    /**
     * Public insert function to call private insert function with data.
     *
     * @param item      The String to add to the tree.
     * @return          The tree with item added.
     */

    public TNode insert (String item) {

        data = insert(data, item);                              // Call private insert function
        return data;
    }

    /**
     * Public find function to call private find function with data.
     *
     * @param item      The String to find in the tree.
     * @return          The tree with item as the root.
```

```
 */

@SuppressWarnings("unused")
public TNode find (String item) {

    data = find(data, item);                                // Call private find function
    return data;
}

/**
 * Public inOrder function to call private inOrder function with data.
 */

public void inOrder ( ) {

    inOrder(data);                                          // Call private inOrder function
}

/**
 * Public preOrder function to call private preOrder function with data.
 */

public void preOrder ( ) {

    preOrder(data);                                         // Call private preOrder function
}

/**
 * Public delete function to call private delete function with data.
 *
 * @param item      The String to find and remove from tree.
 * @return          The tree with item removed.
 */

public TNode delete (String item) {

    data = delete(data, item);                              // Call private delete function
    return data;
}

/**
 * Private delete calls the find function to locate the item in the tree and promote it to the root.
 * If there are multiple occurrences of item in the tree, the TNode's count, c, is decremented,
 * otherwise it is removed from the tree.
 *
 * @param tree      The tree to remove item from.
 * @param item      The String to remove form the tree.
 * @return          The tree with item removed.
 */

private TNode delete (TNode tree, String item) {

    if (tree == null) {                                     // If null tree, return null
        return null;
    }
    tree = find(tree, item);                                // Find item in tree
    if (tree.key.compareTo(item) != 0) {                    // If item not in tree
        return tree;
    }
    if(tree.c > 1) {                                        // If multiple occurrences
        tree.c--;                                           // Decrement c
        return tree;
    }
    if (tree.left != null && tree.right != null) {          // If node has left and right children
        TNode ptr = find(tree.right, successor(tree).key);
        ptr.left = tree.left;                               // Promote successor to root of right
        return ptr;                                         // Attach left subtree to successor
    } else if (tree.left != null) {                         // If right null child
        return tree.left;                                   // Point around root
    } else {                                                // If left null child
        return tree.right;                                  // point around root
```

```
    }
}

/**
 * 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 preOrder function recursively performs a Pre-Order traversal of the tree.
 *
 * @param tree      The tree to traverse.
 */

private void preOrder (TNode tree) {

    System.out.println(tree.key + " - " + tree.c);          // Print node contents (visit)
    if (tree.left != null) {                                // If tree has left child7
        inOrder(tree.left);                                 // Recursive call left
    }
    if (tree.right != null) {                               // If tree has right child
        inOrder(tree.right);                                // Recursive call right
    }
}

/**
 * Private inOrder function recursively performs an In-Order traversal of the tree.\
 *
 * @param tree      The tree to traverse.
 */

private void inOrder (TNode tree) {

    if (tree.left != null) {                                // If tree has left child7
        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 calls the find function to locate item in tree. If item is returned as
 * the root, the TNode's count is incremented. Otherwise, the String's insertion point will have beer
 * promoted to the root and the item will be added to the tree. The function will then call the find
 * function on the item again to promote it to the root.
 *
 * @param tree      The tree to add item to.
 * @param item      The String to add to the tree.
 * @return          The tree with item added.
 */

private TNode insert (TNode tree, String item) {

    TNode pres, prev;
```

```
    if (tree == null) {                                     // If null tree, return new TNode
        tree = new TNode(item);
        return tree;
    }
    tree = find(tree, item);                                // Attempt to find item in tree
    if (item.compareTo(tree.key) == 0) {                    // If new root is item
        tree.c++;                                           // Increment c
        return tree;
    }
    pres = tree;
    prev = pres;
    if (item.compareTo(pres.key) < 0) {                     // Locate insertion point
        pres = pres.left;
    } else {
        pres = pres.right;
    }
    if (pres == null) {                                     // If no left/right child
        if (item.compareTo(prev.key) < 0) {
            prev.left = new TNode(item);                    // Add to root left child
        } else {
            prev.right = new TNode(item);                   // Add to root right child
        }
    } else {                                                // Otherwise add to root's child
        if (item.compareTo(pres.key) < 0) {
            pres.left = new TNode(item);                    // Add as left child
        } else {
            pres.right = new TNode(item);                   // Add as right child
        }
    }
    tree = find(tree, item);                                // Find item again
    return tree;
}

/**
 * Private find function locates a given String in the tree and calls the private splay function
 * with the appropriate TNodes along the way to promote it, or it's would-be parent to the root.
 *
 * @param tree      The tree to locate item in.
 * @param item      The String to locate.
 * @return          The tree with item as root.
 */

private TNode find (TNode tree, String item) {

    TNode grandparent, parent, ptr;
    if (tree == null) {                                     // If null tree, return null
        return null;
    }

    if (item.compareTo(tree.key) == 0) {                    // If current root is item, return tree
        return tree;
    } else {
        ptr = tree;                                         // Point to root
        while (true) {
            parent = ptr;                                   // parent points to ptr
            if (item.compareTo(ptr.key) < 0 && ptr.left != null) {
                ptr = ptr.left;                             // Point left if less than
            } else if (item.compareTo(ptr.key) > 0 && ptr.right != null) {
                ptr = ptr.right;                            // Point right if greater than
            }
            if (parent == ptr) {                            // Return ptr if root equal
                return ptr;
            } else if (item.compareTo(ptr.key) == 0) {
                return splay(ptr, parent, null);            // Return splay if first child equal
            }
            grandparent = parent;                           // grandparent points to parent
            parent = ptr;                                   // parent points to ptr
            if (item.compareTo(ptr.key) < 0 && ptr.left != null) {
                ptr = ptr.left;                             // Point left if less than
            } else if (item.compareTo(ptr.key) > 0 && ptr.right != null) {
                ptr = ptr.right;                            // Point right is greater than
```

```
            }
            if (parent != ptr) {                            // If three TNodes to splay
                ptr = splay(ptr, parent, grandparent);      // Point to splay of three TNodes
            } else {                                        // Otherwise two TNodes to splay
                return splay(parent, grandparent, null);    // Return splay of two TNodes
            }
            if (item.compareTo(ptr.key) == 0) {
                return ptr;                                 // If ptr equals item return ptr as root
            }
        }
    }
}

/**
 * Private splay function 'splays' the given ptr with respect to its parent and grandparent to the
 * local root by calling the appropriate AVL rotations. If grandparent is null, only a single
 * rotation is performed on the parent TNode.
 *
 * @param ptr           The TNode to splay to the top.
 * @param parent        The parent of the TNode
 * @param grandparent   The grandparent of the TNode
 * @return              The TNode splayed to the top.
 */

private TNode splay (TNode ptr, TNode parent, TNode grandparent) {

    if (grandparent == null) {                              // If null grandparent pointer
        if (ptr == parent.left) {
             return singleR(parent);                        // Single right rotation with parent
        } else {
            return singleL(parent);                         // Single left rotation  with parent
        }
    } else {                                                // Otherwise three TNodes
        if (parent == grandparent.left && ptr == parent.right) {
            return doubleR(grandparent);                    // If left-right; double right rotation
        } else if (parent == grandparent.right && ptr == parent.left) {
            return doubleL(grandparent);                    // If right-left; double left rotation
        } else if (parent == grandparent.left && ptr == parent.left) {
            return doubleSR(grandparent);                   // If left-left; double single right
        } else {                                            // Else right right
            return doubleSL(grandparent);                   // Double single left rotation
        }
    }
}

/**
 * Private singleR function performs the single right AVL rotation on a given TNode and its left
 * child.
 *
 * @param tree      The TNode to rotate.
 * @return          The rotated TNode.
 */

private TNode singleR (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
    return ptr;                                             // Return current TNode
}

/**
 * Private doubleR functions performs the double right AVL rotation by calling the singleL and
 * singleR functions.
 *
 * @param tree      The TNode to rotate.
 * @return          The rotated TNode
 */

private TNode doubleR (TNode tree) {
```

```
    tree.left = singleL(tree.left);                         // Left rotation on left child
    return singleR(tree);                                   // Right rotation on current
}

/**
 * Private doubleSR function performs the double single right AVL-like rotation for the top-down
 * implementation of a splay tree by calling the singleR function twice with the appropriate TNodes.
 *
 * @param tree      The TNode to rotate.
 * @return          The rotated TNode.
 */

private TNode doubleSR (TNode tree) {

    tree.left = singleR(tree.left);                         // Right rotation on left child
    return singleR(tree);                                   // Return right rotation on current TNod
}

/**
 * Private singleL function performs the single left AVL rotation on a given TNode and its right
 * child.
 *
 * @param tree      The TNode to rotate.
 * @return          The rotated TNode.
 */

private TNode singleL (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
    return ptr;                                             // Return current TNode
}

/**
 * Private doubleL function performs the double left AVL rotations by calling the singleL and singleR
 * functions.
 *
 * @param tree      The TNode to rotate.
 * @return          The rotated TNode.
 */

private TNode doubleL (TNode tree) {

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

/**
 * Private doubleSL function performs the double single left AVL-like rotation for the top-down
 * implementation of a splay tree by calling the singleL function twice with the appropriate TNodes.
 *
 * @param tree      The TNode to rotate.
 * @return          The rotated TNode.
 */

private TNode doubleSL (TNode tree) {

    tree.right = singleL(tree.right);                       // Left rotation on right child
    return singleL(tree);                                   // Return left rotation of current
}

/**
 * Private buildTree function adds each word given in the list to the tree. SplayTreeException is
 * thrown if a null list is given.
 *
 * @param words     The list of words to add to the tree.
 */

private void buildTree (LNode words) {
```

```
        if (words == null) {                                    // If null list, throw exception
            throw new SplayTreeException();
        }
        while (words != null) {                                 // While list has words
            data = insert(words.key);                           // Add words to tree
            words = words.next;                                 // Get next word
        }
    }
}
```