```java
package assign2;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

/**
 * COSC 2P03 Assignment 2
 *
 * Created by Matt Laidman on September 29, 2014.
 * Student Number 5199807
 *
 * This program will create two Binary Search Trees from a given input text file; one standard BST, and
 * one threaded BST. Only successor (right) threading was implemented.
 *
 * If the program attempts to add a node to the tree whose key already exists it will increase a counter,
 * C, rather than attempt to add a duplicate entry as to maintain Binary Search Tree compliance.
 *
 * When the traversals are run, it will out put the counter next to the word in the list.
 * Output format: word - #
 *
 * The outputs of the traversals of the trees will occur in the following order:
 *
 *  1. Recursive Pre-Order traversal of the un-threaded BST.
 *  2. Recursive In-Order traversal of the un-threaded BST.
 *  3. Recursive Post-Order traversal of the un-threaded BST.
 *  4. Iterative In-Order traversal of the un-threaded BST.
 *  5. In-Order traversal of the threaded BST.
 *  6. Recursive In-Order traversal of the threaded BST.
 *
 * ****************** Valid Input *********************
 *
 * N/A - File location is hardcoded in. Test data is saved to "dat\input.txt".
 *     - (See String variable "INPUT" in constructor)
 *
 * ***************** Global Variables *****************
 *
 * N/A
 *
 *
 * @author Matt Laidman (5199807)
 * @version 1.0 (October 1, 2014)
 */

public class TreeCompare {

    /**
     * Constructor locates the file and initiates the comparing of the traversals
     *
     * Variable INPUT is String representation of location of file containing test data.
     */

    public TreeCompare ( ) {

        String INPUT = "dat/input.txt";

        Scanner fileS = null;
        try {
            fileS = new Scanner(new File(INPUT));
        } catch (FileNotFoundException e) {
            System.out.println("File not found at: \"" + INPUT + "\"");
        }
        if (fileS == null) {
            System.out.println("File not found at: \"" + INPUT + "\"");
        } else {
            doCompare(fileS);
        }
    }

    /**
     * doCompare method calls the necessary functions to read in the test data, and builds
```

```
 * both the Binary Search Tree and the Threaded Binary Search Tree.
 *
 * It then calls the traversal functions in the order required by the assignment.
 *
 * @param reader    - The Scanner ready to read the input file
 */

private void doCompare (Scanner reader) {

    LNode words = parseFile(reader);                        // Get words from reader and save to
    reader.close();                                         // list

    TNode tree = buildBST(words);                           // Build BST from words
    TTNode tTree = buildTBST(words);                        // Build Threaded BST from words

    System.out.println();
    System.out.println("Recursive Pre-Order Traversal (Word - Count)\n");
    recursivePre(tree);                                     // Output recursive pre-order traversal
    System.out.println();                                   // of BST

    System.out.println("Recursive In-Order Traversal (Word - Count)\n");
    recursiveIn(tree);                                      // Output recursive in-order traversal
    System.out.println();                                   // of BST

    System.out.println("Recursive Post-Order Traversal (Word - Count)\n");
    recursivePost(tree);                                    // Output recursive post-order traversal
    System.out.println();                                   // of BST

    System.out.println("Iterative In-Order Traversal (Word - Count)\n");
    iterativeIn(tree);                                      // Output iterative in-order traversal
    System.out.println();                                   // of BST

    System.out.println("In-Order Threaded Traversal (Word - Count)\n");
    threadedIn(tTree);                                      // Output in-order traversal
    System.out.println();                                   // of threaded BST

    System.out.println("Recursive In-Order Traversal (Threaded Tree) (Word - Count)\n");
    tRecursiveIn(tTree);                                    // Output recursive in-order traversal
}                                                           // of threaded BST

/**
 * tRecursiveIn function is the recursive In-Order traversal of the threaded BST.
 *
 * @param tree      - The BST to traverse
 */

private void tRecursiveIn (TTNode tree) {

    if (tree.left != null) {                                // If current node has left child,
        tRecursiveIn(tree.left);                            // recursive call left.
    }
    System.out.println(tree.key + " - " + tree.C);          // Print node contents (visit)
    if (tree.right != null && !tree.thread) {               // If current has right (non-thread)
        tRecursiveIn(tree.right);                           // child, recursive call right.
    }
}

/**
 * threadedIn function is the threaded In-Order traversal of the threaded BST.
 *
 * @param tree      - The BST to traverse
 */

private void threadedIn (TTNode tree) {

    TTNode ptr = tree;
    TTNode qtr;

    if (ptr != null && ptr.key != null) {                   // If root exists
        while (ptr.left != null) {                          // While ptr has left child,
            ptr = ptr.left;                                 // point to left child.
```

```
        }
        while (ptr != null) {                               // (Successor Algorithm) While ptr is
            System.out.println(ptr.key + " - " + ptr.C);    // not null, print node contents (visit)
            qtr = ptr;
            ptr = ptr.right;                                // Point to right child
            if (ptr != null && !qtr.thread) {
                while (ptr.left != null) {                  // While ptr has left child
                    ptr = ptr.left;                         // Point to left child
                }
            }
        }
    }
}

/**
 * iterativeIn function is the iterative In-Order traversal of the un-threaded BST.
 *
 * @param tree      - The BST to traverse
 */

private void iterativeIn (TNode tree) {

    AStack aStack = new AStack();                           // Create stack (array) to store nodes
    TNode ptr = tree;

    while (!aStack.isEmpty() || ptr != null) {              // While stack is not empty
        if (ptr != null) {                                  // If ptr is not null
            aStack.push(ptr);                               // Push onto stack
            ptr = ptr.left;                                 // Point to left child
        } else {                                            // Else if ptr is null
            ptr = aStack.pop();                             // Pop node off stack and set to ptr
            System.out.println(ptr.key + " - " + ptr.C);    // Print node contents (visit)
            ptr = ptr.right;                                // Point to right child
        }
    }
}

/**
 * recursivePost function is the recursive Post-Order traversal of the un-threaded BST.
 *
 * @param tree      - The BST to traverse
 */

private void recursivePost (TNode tree) {

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

/**
 * recursiveIn function is the recursive In-Order traversal of the un-threaded BST.
 *
 * @param tree      - The BST to traverse
 */

private void recursiveIn (TNode tree) {

    if (tree.left != null) {                                // If tree has left child
        recursiveIn(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
        recursiveIn(tree.right);                            // Recursive call right
    }
}
```

```java
/**
 * recursivePre function is the recursive Pre-Order traversal of the un-threaded BST.
 *
 * @param tree      - The BST to traverse
 */

private void recursivePre (TNode tree) {

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

/**
 * buildTBST function initiates the tAdd function to add each word in words list to the threaded tree
 *
 * @param words     - The words to add to threaded tree.
 * @return          - The root node of the threaded BST.
 */

private TTNode buildTBST (LNode words) {

    TTNode tree = new TTNode();
    if (words == null) {                                    // If there are no words in list
        System.out.println("No data to build tree.");
        return null;
    } else {
        while (words != null) {                             // Else while there are words in list
            tree = tAdd(tree, new TTNode(words.key));       // Call tAdd with each word
            words = words.next;
        }
    }
    return tree;                                            // Return built threaded BST
}

/**
 * tAdd function adds each word to the BST and threads the tree as it is built.
 *
 * @param tree      - The tree to add the node to.
 * @param item      - The item to add to the tree.
 * @return          - The root node of the completed tree.
 */

private TTNode tAdd (TTNode tree, TTNode item) {

    if (tree.key == null) {                                 // If empty tree
        return item;                                        // Return item as root
    }
    TTNode ptr = tree;
    TTNode qtr = null;
    while (ptr != null) {                                   // Find where to add node
        qtr = ptr;                                          // While there are items in the tree
        if (item.key.compareTo(ptr.key) == 0) {             // If item is equal to node in tree
            break;                                          // Stop loop
        } else if (item.key.compareTo(ptr.key) < 0) {       // If item is less than node
            ptr = ptr.left;                                 // Go left
        } else if (!ptr.thread) {                           // Else if right is not a successor
            ptr = ptr.right;                                // Go right
        } else {                                            // Else location to add found
            break;                                          // Stop loop
        }
    }
    if (item.key.compareTo(qtr.key) == 0) {                 // If item is equal to current node
        qtr.C++;                                            // Increase count
    } else if (item.key.compareTo(qtr.key) < 0) {           // If item less than current node
        qtr.left = item;                                    // Current node left child to item
        item.thread = true;                                 // Flag item as right threaded
```

```
        item.right = qtr;                                       // Item right child to current node
    } else if (qtr.thread) {                                    // Else (greater than and threaded)
        item.thread = true;                                     // Flag item as right threaded
        qtr.thread = false;                                     // Flag current node as not threaded
        item.right = qtr.right;                                 // Item right thread to current right
        qtr.right = item;                                       // Current right child to item
    } else {                                                    // Else (greater than and not threaded)
        qtr.right = item;                                       // Current right child to item
    }
    return tree;                                                // Return completed tree
}

/**
 * buildBST method initiates the recursive add function to add each word in words to the BST.
 *
 * @param words     - The list of words from the input file to add to the BST.
 * @return          - The root node of the completed BST.
 */

private TNode buildBST (LNode words) {

    TNode tree = new TNode();
    if (words == null) {                                        // If no words in list
        System.out.println("No data to build tree.");
        return null;                                            // Return empty tree
    } else {
        while (words != null) {                                 // While there are words in list
            tree = add(tree, new TNode(words.key));             // Call add with each word
            words = words.next;
        }
    }
    return tree;                                                // Return root node of the completed BST
}

/**
 * add function recursively locates the position in the BST to add item to, and then sets the
 * pointers as the recursion exits.
 *
 * @param tree      - The tree to add item to.
 * @param item      - The item to add to the tree.
 * @return          - The root node of the BST.
 */

private TNode add(TNode tree, TNode item) {

    if (tree == null || tree.key == null) {                    // If tree is empty
        return item;                                            // Return item as root
    } else if (item.key.compareTo(tree.key) == 0) {             // If item equals node in tree
        tree.C++;                                               // Increase count
    } else if (item.key.compareTo(tree.key) < 0) {              // If item is less than node in tree
        tree.left = add(tree.left, item);                       // Recursive call with left child
    } else {                                                    // Else (if item is greater than node)
        tree.right = add(tree.right, item);                     // Recursive call with right child
    }
    return tree;
}

/**
 * parseFile function calls the appropriate method to read in the data from the test file
 * to a node list and then calls and then calls the noSpecialChars method with each word in the
 * list to 'clean' (remove all illegal characters) from each word.
 *
 * @param reader    - The Scanner ready to read in the words.
 * @return          - A node list containing the 'cleaned' words from the data file.
 */

private LNode parseFile (Scanner reader) {

    LNode words = getWords(reader);                             // Call getWords with Scanner and set to
    LNode ptr = words;                                          // a node list.
```

```
        if(ptr == null) {                                       // If node list is empty,
            System.out.println("No data given in file.");       // Empty data file
        } else {
            while (ptr != null){                                // While list has words
                ptr.key = noSpecialChars(ptr.key);              // Call noSpecialChars with each word
                ptr = ptr.next;
            }

        }
        return words;                                           // Return 'cleaned' list
    }

    /**
     * getWords function gets each word from the Scanner and returns a node list containing each word.
     *
     * @param reader    - The Scanner ready to read in words.
     * @return          - Node list containing each word in file.
     */

    private LNode getWords (Scanner reader) {

        LNode words = new LNode(reader.next());                 // Read in first word
        LNode ptr = words;
        while (reader.hasNext()) {                              // While there are words to read in
            ptr.next = new LNode(reader.next());                // Set next node to word
            ptr = ptr.next;
        }

        return words;                                           // Return completed list
    }

    /**
     * noSpecialChars function 'cleans' a String from all illegal characters by checking each char
     * against the validChar boolean function.
     *
     * @param word      - The String to clean.
     * @return          - The 'cleaned' String.
     */

    private String noSpecialChars (String word) {

        String tWord = "";
        for (char c : word.toCharArray()) {                     // For each char in word
            if (validChar(c)) {                                 // If char is valid char
                tWord = tWord + c;                              // Append to new word
            }
        }
        return tWord;                                           // Return new word
    }

    /**
     * validChar function checks a passed char against an array of invalid characters.
     * Invalid chars are: . , - ( ) ;
     *
     * @param c         - The char to check
     * @return          - True if char is valid, false if char is invalid.
     */

    private boolean validChar (char c) {

        char[] invalids = {'.', ',', '-', '(', ')', ';'};
        for (char i : invalids) {                               // For each char in invalids array
            if (c == i) {                                       // If c is equal to invalid char
                return false;                                   // Return false
            }
        }
        return true;                                            // Otherwise return true.
    }

    public static void main(String[] args) {new TreeCompare();}
}
```