CSC 115: Fundamentals of Programming II
Assignment #5: Binary Search Trees

Due date

Wednesday, April 1, 2015 at 9:00 pm via submission to conneX.

How to hand in your work

Submit the requested files through the Assignment #5 link on the CSC 115 conneX site. Please ensure you follow all the required steps for submission (including confirming your submission).

Learning outcomes

When you have completed this assignment, you should have become acquainted with:

- How to create a *reference-based implementation of a binary search tree* for strings.
- Using a binary search tree to *create a word-reference report*.

Word-reference report

A common task when processing text files is producing a *word-reference report*. This is an alphabetical list of words from the text file, and beside each word is printed the line numbers on which the words appeared in the original text file. For example, consider the following four-line text file:

```
Humpty Dumpty
Sat on a wall.
Humpty dumpty
had a great fall.
```

If we filter out common words (such as "a", "the", "in", etc.), and convert all words to lower case, and eliminate punctuation, then a report as follows is produced:

```
dumpty: 1 3
fall: 4
great: 4
had: 4
humpty: 1 3
```

```
on: 2
sat: 2
wall: 2
```

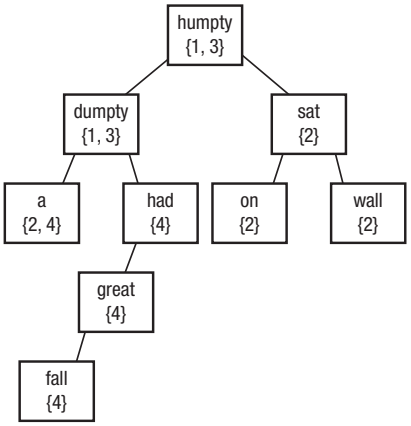
One way we can build this report is to make use of a binary search tree (BST) where the item stored in each tree node is a *word references* object (*WordRefs*). Each *WordRefs* object contains a string and a list of Integers, and therefore the search key for each *TreeNode* will be its *WordRef*'s string. When each individual word in the file is read in, the algorithm must:

- Locate the *WordRefs* item in the BST by its word.
- Add the current line number for the individual word to the end of the *WordRefs* list of Integers.

Note that retrieving the *WordRefs* object from the BST does not remove that object from the tree. Put differently, we do not need to re-insert the *WordRefs* object after adding the current line number. (If a word appears more than once in same line, we can list the line number for the word as many times as the word appears in the line.)

Once all words in the file have been read the algorithm then removes from the BST those *WordRefs* objects corresponding to words in some *exclusion* list (e.g, the words "a", "the", "i", "it", "is".).

Here is one representation of the BST for the file given above but before deletion of the word "a":



After removing the node corresponding the word "a", the final report is now produced via an in-order traversal of nodes in this BST while printing the value of the *WordRefs* object in each node, one word per input line.

Your work for the assignment will be to complete the functionality needed within an implementation of a Binary Search Tree (*BSTRefBased*).

What you are to write

1. You are provided with these files. The only files that may be modified are ***in bold/italicized letters***:
    a. *AbstractBinaryTree.java*
    b. BSTRefBased.java
    c. BSTIterator.java
    d. *TreeException.java*
    e. *TreeNode.java*
    f. *WordReference.java*
    g. *WordRefs.java*
    h. *Test inputs and outputs in the test/ directory.*

2. The code in *WordReference.java* already performs all of the input and output for you. It will open a file, read in each word, convert words to lower case, remove punctuation, keep track of the current line number, and invoke tree operations as needed. See item 8 below for a description of how the program will be executed at the command line.

3. Write your solution to *BSTRefBased.java*. You do not need a separate tester class, but you must write testing code for this class with a *main()* method for the class. Two tests have already been provided to get you started. You should use the textbook's description of BST-specific operations as the basis of your own implementation (i.e., steps 5, 6 and 7 below).

4. For each of the methods already in *BSTRefBased.java* (and which you are therefore not to modify), provide a comment before the method that describes how the operation is implemented by the method. Please note that your comments cannot be copy-and-pastes from the comments in AbstractBinaryTree.

5. Write your implementations of *void insert(String word)* and *TreeNode insertItem(TreeNode r, String word)* within *BSTRefBased.java*. The former method is public (and may be invoked from outside the class), while the latter may only be accessed within *BSTRefBased*.

6. Write your implementations of *WordRefs retrieve(String word)* and *TreeNode retrieveItem(TreeNode r, String word)* within *BSTRefBased.java*. The former method is public (and may be invoked from outside the class), while the latter may only be accessed within *BSTRefBased*.

7. Write your implementations of *void delete(String word), TreeNode deleteItem(TreeNode r, String word)*, *TreeNode deleteNode(TreeNode node*), *TreeNode findLeftMost(TreeNode node)*, and *TreeNode deleteLeftMost(TreeNode node)* within *BSTRefBased.java*. Only the first of these is public (and may be invoked from outside the class), while all the

others may only be accessed from within *BSTRefBased*.

8. Write your complete implementation of *BSTIterator.java*. Use the textbook's description of such an iterator to guide your implementation. You may also use code presented in lectures as the basis of your solution.

9. When you are finished your own tests of your work, please use the tests given in the *tests/* directory. There are eight numbered tests. For example, to run the third test you would execute:

   ```
   java WordReference tests/in03.txt
   ```

   and the results produced must match the contents of tests/out03.txt

10. In all of your work please follow the coding conventions posted at conneX (i.e., a document in the "Lectures & stuff" section).

Files to submit (three in total):

1. *BSTRefBased.java*
2. *BSTIterator.java*
3. *A file named test_output.txt (which also includes a description of what each test is checking followed by the test input and output)*

Grading scheme

- "A" grade: An exceptional submission demonstrating creativity and initiative going above and beyond the assignment requirements. The program runs without any problems and implements the required classes and methods in those classes. Public methods are tested with output showing tests and their results. Any extra work appears in class files having the name *<file>Extra.java*, and identified within the file (i.e., Class comment) are details of your extension to the assignment demonstrating creativity and initiative.

- "B" grade: A submission completing the requirements of the assignment. The program runs without any problems and implements the required classes and methods in those classes. Public methods are tested with output showing tests and their results.

- "C" grade: A submission completing most of the requirements of the assignment. The program runs with some problems but does implement the required classes and methods, and yet might not have the expected output. Methods are tested with output showing tests and their results.

- "D" grade: A serious attempt at completing requirements for the assignment. The program runs with major problems, or does not contain implementations of all required classes and methods.

- "F" grade: Either no submission given, the submission does not compile, or submission represents very little work.