Joanna Klukowska
joannakl@cs.nyu.edu

Programming Project 5: Counting Words

Relative Performance of a Sorted Linked List and a Binary Search Tree

Due date: May 2, 11:55PM EST.

You may discuss any of the assignments with your classmates and tutors (or anyone else) but all work for all assignments must be entirely your own. Any sharing or copying of assignments will be considered cheating. If you get significant help from anyone, you should acknowledge it in your submission.

You are responsible for every line in your program: you need to know what it does and why.

If you are considering use of any Java API classes that are not discussed in CSCI101 curriculum and have not been covered yet in CSCI102, ask your instructor first. The objective of these projects is to practice the material that has been discussed in class and the solutions are not always the most efficient ones.

Objectives

The goal of this project is for you to master (or at least get practice on) the following tasks:

- implementation of a reference based sorted list,
- implementation of a reference based binary search tree with recursive methods.

Problem Description

Given an input text file compute a list of all unique words and count how many times they occur. Repeat this with linked list and binary search tree as the choice of the data structure for storing words to compare their relative performance.

The program running time and word count information are written to the standard output (console). The user specified number of words together with their counts are written to the output file whose name should be provided on the command line.

Problem Solution

You need to implement almost all of the code from scratch, although a lot of it is provided in the textbook and the lecture notes (feel free to reuse any code that is posted on the course website or listed in the book - make sure to provide proper credit).

Input/Output

The program is not interactive. The only input is provided on the command line. The program should expect three command line arguments:

- the name of the input file,
- an integer value indicating the cutoff value for the word counts that should be printed to the output file,
- and the name of the output file. The program needs to validate all command line parameters.

If they are missing or invalid, the program should print an error message and terminate.

The input file is a text file that contains an arbitrary collection of words. The program should use the provided FileParser class to obtain a list of clean words from the input. Read the documentation for that class to learn how to use it. The program should catch and handle the exceptions that `FileParser` class throws.

The output file should contain the list of the most frequently occurring words together with their counts. The second parameter on the command line specifies the cutoff value of the words that should be printed to the output file - only the words whose count is above or equal to the cutoff value should be printed to the output file. The structure of the output file should be as follows:

```
count1   word1
count2   word2
...
```

The words should be printed in alphabetical order (not sorted by counts).

Processing of data

Populating the data structures Once your program uses FileParser to obtain the list of all the words from the input file, it should store them in two different data structures: 1) a sorted linked list, 2) a binary search tree. The nodes in the list/tree should store unique words (no repetitions allowed) together with their counts. As the program goes through the list of all words provided by the FileParses object it needs to decide if the next word already exists in the structures or not. If it does exist, the count for the corresponding word should be incremented. If it does not exist, a new entry should be created with count 1 and added to each structure.

Computing the most frequent words In order to find the `N` most frequent words in the input file, your program needs to find the N words with the highest counts in both data structures. This task should be accomplished by pruning the list and the tree. Traverse each data structure and remove all nodes that contain words whose count is below the cutoff value. After pruning both structures should contain identical words. The words remaining in one of the structures should be then printed to the output file (do not print the content of both structures to the output file).

Performance measurements The program should time how long it takes to 1) construct a structure given an ArrayList containing all the words from the input file, 2) prune the structure given the cutoff value. The program should display to the console all information regarding the timing and sizes of the structures at each stage. The output should be printed in the following format:

```
INFO: Reading file took 294462 ms (~   0.294 seconds).
INFO: 218082 words read.

Processing using Sorted Linked List
INFO: Creating index took 12746524 ms (~  12.747 seconds).
INFO: 11321 words stored in index.
INFO: Pruning index took 11389 ms (~   0.011 seconds).
INFO: 250 words remaining after pruning.

Processing using Recursive BST
INFO: Creating index took 74390 ms (~   0.074 seconds).
INFO: 11321 words stored in index.
INFO: Pruning index took 18046 ms (~   0.018 seconds).
INFO: 250 words remaining after pruning.
```

Implementation

The class that runs this program should be named `MostFrequentWords` - this class is responsible for validating command line parameters, parsing the input file (using FileParses object), creating the two data structures to store the words, performing all the timing, printing data to the console, and printing data to the output file. Given the number of tasks, you should modularize this class and provide methods that perform some of the task outside of the `main()`.

Your program also needs to implement the `SortedLinkedList` and `BST` classes. They do not need to be generic, but they may if you wish. You need to decide what methods have to be provided by each of the classes. You may find it useful to specify a common interface that both classes implement (since they are providing exactly same service to the calling program). You will also need to define classes to represent nodes for the list and for the tree.

You may find it useful to provide a class that represents an object composed of a word and its count. This is not mandatory, but should simplify implementation of the list and the tree.

Working on This Assignment

You should start right away! Unlike in the previous projects, you need to write significant amount of the code. Even if you are using the implementations that are provided in the book/lectures you still have to make them work with your program and validate their correctness. You can modularize the work by implementing and testing one structure at a time.

Grading

20 points - development and correctness of `MostFrequentWord` class

30 points - development and correctness of `SortedLinkedList` class (and any helper classes that it needs, like the node)

30 points - development and correctness of `BST` class (and any helper classes that it needs, like the node)

20 points - documentation and program style

Documentation has to be specified using Javadoc. Program style includes (but is not limited to):

Formatting

- indentation - should be consistent,
- spacing - use white space to separate sections of code, but do not put two blank lines between each line of code),
- maximum line length - it should fit on a fairly small screen, for many programmers 80-100 characters should be the max for the line length - don't let it wrap,
- use of parenthesis to make it clear what operations are performed in what order
- use of curly braces to indicate where one block ends and other begins
- ordering of the declarations of the class members: fields, constructors, methods

Naming conventions

- class and interface names start with a single uppercase letter
- package names use lower case letters
- constants use all upper case letters
- other identifiers use camel case

Self documenting code

- use names that represent the variables/methods
- use full name - abbreviations may make sense when you first write them, but you will forget what they stand for and others will never know

Coding

- don't declare variables that are never used
- don't use loops with conditions that are always true and then use a break to terminate
- don't write methods that do too many things - each method should perform a single concise task
- don't write the same code in multiple places - turn it into a method

How and What to Submit

Your should submit all source code files (the ones with .java extensions only) in a single **zip** file to NYU Classes.

Do not submit the text files that were posted as sample input files.

Once you submit the program make sure that you receive a confirmation email from NYU Classes. Read through it to double check that correct files were uploaded. Keep this email as your proof of submission until you receive your grade for the assignment.