EECS 2500 – Fall 2015

Programming Project #2 – Due November 10, 2015 23:59:59

For this assignment, you will read and process a text file containing the complete text of William Shakespeare's *Hamlet*.

Your program will read in the text file, word by word. You are to ignore punctuation and capitalization (i.e., "Castle", "castle?", and "castle" are all equivalent). If you like, you may convert all words to lower case on input, or you can use case-insensitive comparisons – your choice. You will be putting the words into several linked lists, and comparing the performance between approaches.

If, after trimming punctuation and letters, a "word" consists of no letters, then discard the word. Such a case might arise if your word-reading strategy assumes that every word is delimited by spaces, in which case, a dash would constitute a "word", but would not contain any letters, so it would be discarded ("comparisons – your choice" *should* be three words, though your parser might identify "-" as a fourth "word"; however, since it contains no letters, it should be discarded).

For this project, each node in the list will contain TWO pieces of information, in addition to the link to the next node – a `String` for the word stored in the node, and an `int` to hold a count, representing the number of times the word appears in the play.

In your `add` method, when you attempt to add a word that is already on the list, rather than actually creating a new node (containing a duplicate of the word), simply increment the count stored in that node.

Use chapters 9 and 19 of the Liang Java book to help you with the file I/O.

Before processing the lists, you are to make a pass through the file, and time how long it takes you to read and parse the words – this is the "overhead" time. Then, make four MORE passes through the file, building four lists:

Add your words to the following linked lists (build the lists in this order):

1) An unsorted linked list, in which additions always occur at the beginning (i.e., when a word read from the file is not already in the list, a new node, containing the word and a count of 1, becomes the list's first new node.
2) A sorted (alphabetically) linked list.
3) A self-adjusting list in which, when a word is found to be already in the list, the node containing that word is moved to become the first node of the list. For words NOT already in the list, they become the list's first word.
4) A self-adjusting list in which, when a word is found to be already in the list, the list moves back up towards the start of the list by <u>*one node*</u> (rather than all the way to the beginning, as in list #3). In order to implement this functionality, you will need to have a `previous` reference to keep track of the predecessor of the node containing the word in question. Words NOT already in the list become the list's first word.

In the first list, insertions are simple, but searching could require traversing the whole list. Handling a duplicated word (incrementing the count) is trivial.

In the second list, insertions are still pretty simple, but searching should only require traversing half of the list (on average). Incrementing the count is still trivial.

In the third and fourth lists, handling repeated words gets much more complicated, because incrementing a count now also requires moving a node (changing references), but if the self-adjusting nature of the lists helps out enough, the reduced search times may (more than?) make up for this complexity. Moving a repeated word to the beginning of the list is a heavy-handed optimization. Is the more subtle optimization of "moving it back upstream one position" any better?

For the third and fourth lists, produce a list of the words and their counts from first 100 nodes (after the list is built, and you have gathered its statistics [see below]). Copy and paste this list into Excel, and produce a spreadsheet with a line graph showing the counts for the first hundred words of the two lists.

Each of your list classes should provide the following performance metrics:

1) The total number of words in the list (this will be computed by summing the counts in all of the nodes *after* the list is built)
2) The total number of *distinct* words in the list (this will be the number of nodes in the list)
3) The total number of times a comparison was made between the word just read and a node in the list (how many times we checked SOME node to see if it contains the just-read-from-the-file word)
4) The total number of reference changes made. For changes to the list pointer (i.e., the reference variable that points to the first node), consider this a reference change, just like changing the reference in a node.
5) Elapsed time (in decimal seconds – measure time in milliseconds, and divide by 1000).

Your code should be structured such that you open the file, and read it one word at a time, simply to see how long it takes to read and parse the file. Then, close the file, re-open it, re-parse it, building the first list, and report the elapsed time required to build the list (along with the other metrics above). Then close the file, re-open it, and read through it a second time, building the second list, and print the elapsed time for the second method. Repeat this process for each of the four lists.

It would obviously be more efficient for you to instantiate all four lists, and add the same word to each list at once, rather than making four passes through the file, but we would not be able to determine the elapsed time for each list this way. Make the file name a constant, called FILE_NAME, defined early in `main()`, so that I can change the file name once to test your program on some other text file.

Your code should consist of classes for each list, and be well-documented. All output should be to the console (no GUI output).

Submit a 7-zip archive of your ENTIRE Java Workspace directory (and its subfolders) to Blackboard. The name of the folder containing your Eclipse Workspace (and hence the 7-zip file) should be <LastName><comma><Space><FirstName>. I should be able to unzip your workspace, point Eclipse to it, and run your code without modification. You should be using the 32-bit versions of Eclipse Standard (Mars.1), with the JDK 8u60 (even though 8u65/66 has been released). If you need them, they are available at:

http://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/mars/1/eclipse-java-mars-1-win32.zip

http://www.oracle.com/technetwork/java/javase/downloads/java-archive-javase8-2177648.html#jdk-8u65-oth-JPR

Also submit your Excel Spreadsheet (not in the 7-zip) to Blackboard

If you have questions, please let me know ASAP.