EECS 2500 – Fall 2021 – Lab #2

Due November 5, 2021 23:59:59

For this assignment, you will read and process a text file containing the complete text of William Shakespeare's *Hamlet*. After getting the program working with *Hamlet*, you will repeat your run with different files, to see how the size of the file (and the size of the vocabulary used IN the file) influences the amount of work done. After gathering this other data, you will prepare a formal lab report of your findings.

Your program will read in the text file, word by word. You are to ignore leading and trailing 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, 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 space-delimited parser might identify "-" as a fourth "word"; however, since it contains no letters, it should be discarded). Do NOT use `.replaceAll()` (or anything else) to remove *all* punctuation – that would remove *internal* punctuation as well; you are to trim *leading* and *trailing* punctuation only. Strings like "don't" have internal punctuation, and are not to be modified.

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 file.

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 12 and 17 of the Liang Java book (10th edition) to help you with the file I/O.

Before processing the lists, you are to make two passes through the file. Don't measure anything about the first pass – this pass is purely for letting the O/S and the JVM go through the file, filling buffers, etc. On the SECOND pass, 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 (singly-) 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. This approach does the least amount of work on an *addition*.
2) A sorted (alphabetically) linked list. This list should have shorter average search times, because once you've determined where a word should be, if it's not there, there is no use in searching the *rest* of the list (if you're trying to add "absolute", and you reach "absolve", you know that "absolute" isn't in the list, and there's no sense checking the remainder *of* the list. You do, however, have to create a new node and link it in between the two existing nodes (or perhaps this is a new first node, or a new last node).
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. The idea here is that the more-frequently-used words will stay closer to the

beginning of the list. Moving a repeated word *all the way* back to the beginning of the list is something of a “heavy-handed” adjustment, which leads us to the fourth list…

4) A self-adjusting list in which, when a word is found to be already in the list, the word’s position moves back up towards the start of the list *by one node* (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 (not a doubly-linked list) 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 all the way 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? These are just some of the questions your investigation should explore (and your report should document)!

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 four list classes must 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; not by keeping track of the count as you go)
2) The total number of *distinct* words in the list (this will be the number of nodes in the list) – the file’s “vocabulary” – also to be computed at the end, rather than on-the-fly.
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). These are caked “key comparisons”
4) The total number of node / list 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. If you use a variable to traverse the list, do NOT count changes to that variable; only to node pointers and/or list pointers.
5) Elapsed time (in decimal seconds with three places – measure time in milliseconds, and divide by 1000, so your output will look like “12.345 seconds”)

Your code should be structured such that you open the file, and read it one word at a time, simply to fill the O/S buffers, and then close the file, re-open it, and make a second pass to see how long it takes to read and parse the file after that. 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 so many 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 you 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 "2500 - <LastName><comma><Space><FirstName>.7z". I should be able to unzip your workspace, point Eclipse to it, and run your code without modification. You should be using same versions of Eclipse and the JDK as in the first lab Also submit your Excel Spreadsheet (not in the 7-zip) to Blackboard. Name your excel spreadsheet and your report the same way – you will submit a .7z, a .docx, and an .xlsx file.

Once you have your program running for Hamlet, I will provide additional text files for you to process it against. You are to gather the statistics for each file, put the data into Excel, produce graphs showing the relationship(s) between file length (words), vocabulary, total run time, and searches and reference changes made. All of this will go into a document (Word; no .rtf, .pdf, etc.) that you will submit as a lab report along with your project. Submit the lab report OUTSIDE of your 7-zip file, but name the report "2500 - <LastName><comma><Space><Firstname>-Report.docx"

If you have questions, please let me know ASAP.

Implementation requirements:

1) Put all of your code in the default package (don't create any packages for this project). Ignore Eclipse's warning about the "discouraged" use of the default package. For our purposes, it's fine.

2) Your list classes must all implement this `ListInterface` (not the one in the book):

```
// ListInterface.java
//
// This interface lists all of the methods that the four test lists in this project
// have to support -- adding a word to the list (aka insert), and getters for the
// total number of key comparisons and reference changes, as well as the counts
// for distinct and total words (after the lists have been built)
//
public interface ListInterface
{
    public void add(String word);   // add this word to the linked list

    public long getKeyCompare();    // Get the number of key comparisons
    public long getRefChanges();    // Get the number of reference changes

    public int  getDistinctWords(); // Get the # of distinct words on the list
    public int  getTotalWords();    // Get the total number of ALL words
}
```

3) Your lists must also be extensions of this abstract class:

```java
// BaseList.java
//
// This class is just a starting point for the four lists that will be used in this project.
// Because the only thing that will differentiate them is the WAY we do the additions to the
// list, we can put all of the code that's common to ALL of them here, and use inheritance
// to build the four specific lists.  We declare this one as being abstract simply to keep
// from accidentally instantiating it -- it doesn't HAVE its add method.
//
// By forcing this layer of inheritance into the process, every list's class just consists
// of a one-line constructor (to call this one), and its add() method.

public abstract class BaseList implements ListInterface
{
    LLNode list;
    long   refChanges;
    long   keyCompare;

    BaseList() // constructor: empty list, counters set to zero
    {
        list = null;
        refChanges = 0;
        keyCompare = 0;
    }

    @Override
    public long getRefChanges()
    {
        // How many reference changes did we do (how much structural work)?
        //
        return refChanges;
    }

    @Override
    public long getKeyCompare()
    {
        // How many key comparisons (how much work was done looking for things on the list)?
        //
        return keyCompare;
    }

    @Override
    public int getDistinctWords()
    {
        // How many nodes are there on the list?  Each corresponds to a unique word
        //
        int count = 0;       // Alternate coding:
        LLNode p = list;     //
        while (p != null)    // for (LLNode p = list; p != null; p = p.getNext()) count++;
        {                    // return count;
            count++;         //
            p = p.getNext(); //
        }                    //
        return count;        //
    }

    @Override
    public int getTotalWords()
    {
        // How many TOTAL words? That's the sum of the counts in each node.
        //
        int count = 0;
        LLNode p = list;
        while (p != null)
        {
            count += p.getCount();
            p = p.getNext();
        }
        return count;
       }
}
```

By using this abstract class and the interface, you can create your individual lists as an array, and process them in a loop, rather than with sequential code:

```
ListInterface[] Lists = new ListInterface[4]; // By creating the lists as
                                              // an array, we can iterate
                                              // with a subscript
Lists[0] = new List1(); // Unsorted, insertions at beginning, no self-optimization
Lists[1] = new List2(); // Sorted linked list
Lists[2] = new List3(); // Unsorted, heavy-handed self-adjusting (moves repeated
                        //word to the front of the list)
Lists[3] = new List4(); // Unsorted, conservative self-adjusting (moves repeated
                        //word one position towards front of list)
//
String[] ListNames = {"Unsorted", "Sorted", "Self-Adj (Front)",
                      "Self-Adj (By One)"};
```

This will also facilitate modularity, which will help with the NEXT assignment, which will build on this one.

4) Consider the following characters to be punctuation (remember, you are to trim the leading and trailing punctuation *only*):

```
!@#$%^&*()_+-=[]\{}|;':"`~,./<>?
```

5) Your output will be in the following format:

```
 #    List Name      Run Time  Vocabulary  Total Words   Key Comp     Ref Chgs
-- ----------------- --------  ----------  ----------- ------------ ------------
 1 Unsorted          9999.999    9999999   999999999   99999999999  W
 2 Sorted            9999.999    9999999   999999999   99999999999  99999999999
 3 Self-Adj (Front)  9999.999    9999999   999999999   99999999999  99999999999
 4 Self-Adj (By One) 9999.999    9999999   999999999   99999999999  99999999999
```

You must use `System.out.printf()` to produce your individual output lines (the header and separator lines may be done with `System.out.println()`, of course).