EECS 2500 – Fall 2015

Programming Project #4 – Due: December 12, 2015 23:59:59

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

Just as in the last program, your code will read in a 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 the same four linked lists, with the same rules, AND into a binary search tree, and reporting the performance metrics for all five. The goal here is to compare the performance of the binary tree to that of the four linked lists (in particular, the goal is to compare the tree against the fastest of the four lists, whichever that one is).

If, after trimming leading and trailing 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 parser might identify "-" as a fourth "word").

In addition to the four linked lists from Project #2, you are to build a Binary Search Tree, with each distinct word in a node.

Each node in the tree will contain TWO pieces of information, in addition to the two child pointers (which *any* binary search tree node would have) – 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. It will not help you to implement the parent-node pointer, so don't worry about that (*z.p* in the slides)

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

Use the pseudocode provided in class as a starting point. Note: That pseudocode is NOT ready-to-go as-is – you'll want to adjust it a bit. Don't create a new node to hold the word you're trying to insert until you determine that it's not already in the tree. If it IS already in the tree, just increment that node's counter and `return`. If it's NOT already in the tree, then build a new node and hang it on the tree as a new leaf as appropriate.

You tree class should provide the following performance metrics:

1) The total number of words in the tree (this will be computed by summing the counts in all of the nodes once the tree is built)
2) The total number of distinct words in the tree (this will be the number of nodes in the tree)
3) The total number of times a comparison was made between the word just read and a node in the tree (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 tree's root pointer (i.e., the reference variable that points to the root node), consider this a reference change, just like changing a

child reference in a node. Note: this will only happen once, when you create the first node for "ACT"

5) The total elapsed time to build the tree (measured in milliseconds, divided by 1000 to give decimal seconds with three decimal places) .

In order to count the number of nodes and total words (the counts IN the nodes), you'll want to do a tree traversal AFTER the tree is built. I suggest a recursive in-order traversal. We covered this code in class. Rather than just printing the contents of a node, your traversal code should pick up "1" (for the node counter) as well as the counter IN the node (the number of times that word appeared). Do not just keep a pair of counters to count words and nodes as you go – just build the tree, and then count words and nodes AFTER the tree is built.

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, and finally, for the binary tree.

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.