Introduction

CONGRATS! You have just graduated from SJSU! You have just been approached by a world famous history professor. He would like you to settle a centuries-old debate on who wrote Shakespeare's plays, Shakespeare or Sir Francis Bacon? You protest that this question is surely outside of your area of expertise. "Oh, no," chuckles the historian, stroking his snowy white beard. "I need a Computer Scientist!"

For this project, you will:

Be introduced to DataCounter, a variant of the DictionaryADT, and understand the strengths of various DataCounter implementations

- Gain more practice with AVL trees
- Become comfortable with implementing a HashTable
- Understand some of the complexities of word stemming
- Learn how to benchmark your code

Word Frequency Analysis

Authors tend to use some words more often than others. For example, Shakespeare used "thou" more often than Bacon. The professor believes that a "signature" can be found for each author, based on frequencies of words found in the author's works, and that this signature should be consistent across the works of a particular author but vary greatly between authors. He wants you to come up with a way of quantifying the difference between two written works, and to use your technique on several of Shakespeare's and Bacon's works to settle the ancient debate!

The professor has provided you with copies of Shakespeare's (Hamlet) and Bacon's writing (The New Atlantis), which he has painstakingly typed by hand, from his antique, leather-bound first-editions. Being good scientists, however, you quickly realize that it is impossible to draw strong conclusions based on so little data, and asking him to type two more books is out of the question! Thus, you should download and analyze several more works, as many works as you feel is necessary to support your conclusion.

Word Stemming

When dealing with document correlations, it is often desirable to work only with the roots of words. That way, "sleeps", "sleeping", and "sleep" are all considered to be the same word. This process is called word stemming, and is used in most real-world search engines. For this assignment, you only need to follow two simple rules:

Convert all the words to lowercase ("An" and "an" are the same word)
Remove all punctuation ("end." and "end" are the same word)
The supplied class FileWordReader includes code to do this processing for you.

Word stemming is a fairly complex topic. What these rules do is not so much word stemming as input normalization; you do not try to undo conjugations or other morphology. Fancier word stemming such as removing 's' from the end of a word can lead to erroneous results (such as "bu" from "bus") and require special logic. Even our simple rules cause problems; for instance, "it's" and "its" are now the same word. Implementing a better stemming algorithm (like Porter Stemming) is above and beyond work.

Signature Generation

A fundamental part of your work lies in computing the "signature" of a document. The professor has provided you with a sample WordCount program that reads in a document and counts the number of times that a stemmed word appears, assuming that the document's words are already stemmed. The output of this program looks like this:

```
970 the
708 and
666 of
632 to
521 at
521 i
521 into
466 a
444 my
391 in
383 buffalo
...
```

where the number in column 1 is the frequency that the corresponding string in column 2 occurs in the text. Note that the WordCount program sorts its output primarily in decreasing order by frequency count, secondarily by alphabetical order. The ordering would be identical if it had sorted by frequency fraction first (i.e. frequency_count/num_total_words).

Document Correlation

Document correlation is a reasonably large area of study. Perhaps its most visible application is in search engines which rank the correlation of webpages to a set of keywords that you provide. One model often used for correlating documents is Latent Semantic Indexing (LSI) where a collection of documents is considered together (rather than independently) and a word's usefulness is determined by how frequently it appears in the documents (for instance, "the" isn't very useful because it appears in most documents).

We will not be doing LSI. We will do a simpler document comparison:

- Calculate word counts for the two documents and normalize the frequencies so that they can be meaningfully compared between different documents (hint: use frequency percentages or fractions.)

- As in LSI, remove words whose relative frequencies are too high or too low to be useful to your study. A good starting point is to remove words with normalized frequencies above 0.01 (1%) and below 0.0001 (0.01%), but feel free to play around with these numbers. You should, however, report results with these frequencies so we can compare outputs with a standard set of numbers.

- For every word that occurs in both documents, take the difference between the normalized frequencies, square that difference, and add the result to a running sum.

- The final value of this running sum will be your difference metric. This metric corresponds to the square of the Euclidean distance between the two vectors in the space of shared words in the document. Note that this metric assumes that words not appearing in both documents do not affect the correlation.

Partners

You are encouraged (although not required) to work with a partner of your own choosing for this project. No more than two students total can be on a team together.

Requirements

There are five steps in this project:

1. Write two DataCounter dictionary implementations (AVL, Hash)
2. Modify WordCount to be able use your DataCounter implementations, and to select the implementation at runtime.
3. Write a document correlator that will print a difference score between two documents
4. Benchmark your data structures and correlator
5. Analyze and write up the results of your performance tests

   The analysis and writeup will be significantly longer in this project. Be sure to allocate time for it. It is worth 1/3 of your grade, and you will not be able to do it in an hour or two.

DataCounter Implementations

For this assignment, you must implement two data structures (you may reuse previous implementations):

AVL tree – **Your implementation MUST extend BinarySearchTree included in the zip files regardless if you choose to modify the existing code for BinarySearchTree.**
Hash table – **You must implement your own hash code for the hash table. Do not use String.hashCode()**

Bothof these data structures must implement the DataCounter interface, which is a specialized version of DictionaryADT. You do not need to implement remove in any of these structures. You can implement any hash tables discussed in class; the only restriction is that it should not restrict the size of the input domain or the number of inputs (i.e. your hash table must grow).

WordCount

The WordCount program will read in a text file and tally up all the words that appear in it. The WordCount program given to you currently uses an unbalanced binary search tree as its backing DataCounter implementation and contains an insertion sort. You may base your WordCount program on it, or write your own. You need to add additional DataCounter implementations.

The commandline form for WordCount will be as follows:

java WordCount [-b | -a | -h] [-frequency | -num_unique] <filename>

- -b Use an Unbalanced BST to implement the DataCounter
- -a Use an AVL Tree
- -h Use a Hashtable
- -frequency Print all the word/frequency pairs, ordered by frequency, and then by the words in lexicographic order
- -num_unique Print the number of unique words in the document. This is the total number of distinct (different) words in the document. Words that appear more than once are only counted as a single word for this statistic.

It is fine to require that one of -b, -a, or -h must be specified for your program to run. Your program should not crash, however, if given an invalid command line.

Note that for the -frequency option, you need to produce words ordered primarily by frequency and secondarily by lexicographic (i.e., alphabetical) order. For example:

```
43 these
42 upon
42 your
41 after
41 into
40 said
39 many
39 more
38 an
```

The sample WordCount program does this sorting using Insertion Sort... More on sorting projects next time!

Document Correlator

The Document Correlator should take in 2 documents and return the correlation (difference metric calculation) between them. You may want to use the WordCount class given to you to implement the backend of the Correlator, though doing so is not necessary. For the basic requirements, you must design an algorithm that does the comparison specified in section IIIc Document Correlation. This program should also take command line flags to specify which backing data structure to use. The commandline structure should be:

Usage: java Correlator [-b | -a | -h] <filename1> <filename2>

- -b Use an Unbalanced BST in the backend
- -a Use an AVL Tree in the backend
- -h Use a Hashtable in the backend

Benchmarks

Since we are implementing two DataCounter dictionaries in this project, it is natural to ask "which is faster." Benchmarking and profiling are really the only reliable ways to judge this since there are many many hidden assumptions in the way you write your code that will add unexpected constants to your program. Hopefully you will do some exploration in this assignment and prove to yourself that you really can't predict what will affect program runtime too much (go through and try to optimize away little things like how many assignments you do, how many if statements you execute, etc. and see how much or little this affects your program).

When generating (or reading) benchmarks, you must ask yourself the following questions:

- What am I measuring? Speed is too vague. Does it mean full program runtime? What if my program waits for user input? Does it matter?
- Why am I measuring this and why should anyone be interested in it? Full program runtime of an interactive user app where the users fall asleep while running the code isn't really interesting data.
- What methodology will I use to measure my program? Does it actually measure what I want?
- What are the sources of error? Is the error big enough to matter? Are my results still reliable?

This set of questions actually applies to any analysis.

You are required to design benchmarks that measure the attributes listed below. You may also include any other data that you feel necessary to draw conclusions from the benchmarks in your analysis.

- How fast is java WordCount -frequency compared to count.sh and count.pl?
- How much difference in speed do your different DataCounters make in the correlator and/or the wordcount?

There are a few tools available to you for benchmarking. The simplest ones are:

- The Unix time command.
- System.nanoTime() or System.currentTimeMillis() (record the time at two different places in your program and subtract to get running time)

Both methods have their strengths and weaknesses (for instance, time must measure your process creation times, and JVM startup times). These strengths and weaknesses will exhibit themselves differently depending on where and how these tools are used. In your analysis, you will need to cite the known sources for errors in your benchmarks and justify why they don't matter for your measurements, or somehow create a correction for your measurement. Essentially, you must convince us that your benchmark is measuring something that makes sense and that your analysis can be based off the collected data.

For example, to time count.sh or count.pl, you can do the following:

```
 time ./count.sh your-file
```

The syntax is the same for count.pl. Use the User time value that time returns.

README.txt

Your README.txt file needs to answer the following questions:

1. Who is in your group?
2. How long did the project take?
3. Before you started, which data structure did you expect would be the fastest?
4. Which data structure is the fastest? Why were you right or wrong?
5. In general, which DataCounter dictionary implementation was "better": trees or hash tables? Note that you will need to define "better" (ease of coding, ease of debugging, memory usage, disk access patterns, runtime for average input, runtime for all input, etc).
6. Are there cases in which a particular data structure performs really well or badly in the correlator? Enumerate the cases for each data structure.
7. Give a one to two paragraph explanation of whether or not you think Bacon wrote Shakespeare's plays based on the data you collected. No fancy statistical analysis here (formal analysis comes later); keep it fun and simple.
8. Writeup your benchmarks (this is long). Your mission is to convince us that your benchmark makes sense and that we should be interested in it if we are trying to ascertain which data structure is better suited for your input. You will need to answer at least the following (all formal analysis should answer something similar):

- What are you measuring?
- What is the definition of "better" given your measurement?
- Why is the measurement interesting in determining which is the superior algorithm for this project?
- What was your method of benchmarking?
- What were the sources of errors?
- What were your results?
- How did you interpret your results?
- What were your conclusions?
- Are there any interesting directions for future study?

  You may attach this in a separate, non-plain-text file. If you do use a non-plaintext format please ensure that the file extension matches the format. Also, please consider using PDFs in place of proprietary formats (i.e .doc(x), .xls(x), .pages, .numbers, etc.) which will just cause your grader extra headaches.

9. What did you enjoy about this assignment? What did you hate? Could we have done anything better?

Files and Sample Code

Sample texts and starter code are available on the course web site in file project3files.zip. (Right-click on the link to download the file if a single click doesn't work.) You can also get texts of your own at Project Gutenberg, which has thousands of books as plain text files. Their mission is to provide electronic versions of many popular public domain texts. Check it out! Note that these files contain some header text that is not part of the actual play or book. For accurate results you should remove everything that is not part of the work. Extra text occurs at the beginning and end or each file, and sometimes at the beginning of each act of a play. Try running your word-counting program on the King James Bible. (Guess which word comes up more frequently in the Bible: "he" or "she?"... and by a factor of what?). Also, if you have any special requests for texts or other cool files you'd like to have added to the test files, email the course staff .

In addition, your history professor has provided some code which he wrote (these days, everybody knows how to program!). You may use it if you wish, although your code must follow the provided DataCounter interface:

- DataCounter - Specification of an interface for a DataCounter. Your classes must implement this interface. (Note that DataCounter is a dictionary that is specialized for this particular task, so it isn't as generalized as some of the ADTs we've seen in the past. This is primarily for performance reasons.)
- BinarySearchTree - Specification and implementation of an unbalanced binary search tree class. Use of the provided BinarySearchTree implementation is optional (you may choose to implement your own), but your AVL and Splay tree classes must inherit from your BinarySearchTree implementation.
- BinarySearchTree.BSTNode- Specification and implementation of a binary search tree node (used in the BinarySearchTree class).
- FileWordReader - A class that reads in a file and does simple word stemming.
- WordCount - A simple program that reads words from a FileWordReader and tallies their frequency in a DataCounter.
- count.sh - A Unix shell script to compute word counts. Usage: ./count.sh your-file
- count.pl - Similar to count.sh, except in Perl.

Turnin

Code Turn-in

For the code turn-in, you are expected to turn in all your code (AVLTree, HashTable, BinarySearchTree (if you modified it), WordCount, Correlator, your unit tests, and any other code of interest).

Note that the code you submit for the Code turn-in should be functional, fully-tested code. Include your (and your partner's) real name in every file you turn in. **If you don't you will lose points.**

Turn in a zip with your and your partner's names in the title as you did with Project 2 with the individual files inside. **If you don't you will lose points.**
If you turn in a file, ensure that it has the appropriate extension. Renaming a .doc to .txt does not make it a plain text file, and causes problems. If you miss this you will lose points.
Please remember to remove debugging output before submitting your code.
Writeup and Benchmarking turn-in

For the final checkin, include all of the above, your README.txt, your benchmark analysis. You may turn in some format other than a plain-text file, especially for the analysis part. However, please do not turn in a file format that is not easily portable. **Consider using PDFs.**

One partner needs to turn in.

VIII. Grading Breakdown

Each part of the assignment will be worth (approximately) the following percentages. Please budget your time appropriately.

55%	Program correctness (including boundary cases) and error-free compilation
45%	README.txt and answers to writeup questions, including benchmarking

Credits

This assignment is has become a fixture in my old UW data structures class, and has evolved a little or a lot over the years!