**Homework 4 - Hash Tables**

**Assignment**

**Introduction**

A world famous Hampton University history professor wants you to settle [a very old debate](http://en.wikipedia.org/wiki/Shakespeare_authorship_question) on who wrote Shakespeare's plays, [Shakespeare](http://en.wikipedia.org/wiki/William_Shakespeare) or [Sir Francis Bacon](http://en.wikipedia.org/wiki/Francis_Bacon)? You protest that this question is surely outside your area of expertise. "Oh, no," chuckles the historian, stroking his snowy white beard. "I need a Computer Scientist! Someone who can efficiently compare any two documents in terms of their word frequency."

Authors tend to use some words more often than others. For example, Shakespeare used "thou" more often than Bacon. The professor believes 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.

The professor gave you copies of Shakespeare's writing ([Hamlet](http://courses.cs.washington.edu/courses/cse373/14sp/homework4/hamlet.txt)) and Bacon's writing ([The New Atlantis](http://courses.cs.washington.edu/courses/cse373/14sp/homework4/the-new-atlantis.txt)), 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 more books is out of the question! Thus, for Phase B you should download and analyze several more works, as many works as you feel is necessary to support your conclusion. [Project Gutenberg](http://www.gutenberg.org/wiki/Main_Page) is a good place to look.

**Interesting Facts**

Word-frequency analysis plays a central role in providing the input data for [tag clouds](http://en.wikipedia.org/wiki/Tag_cloud). There are many uses for tag clouds, such as indicating words that are more common in some writing (e.g., someone's blog) than they are more generally (e.g., on all web pages).

Word-frequency analysis also plays an important role in Cryptanalysis, the science of breaking secretly encoded messages. The first mention of using the frequency distribution of a language to break codes was in a 14-volume Arabic encyclopedia written by al-Qalqashandi in 1412. The idea is attributed to Ibn al-Duraihim, the brilliant 13th century Arab Cryptanalyst. If you are interested in cryptography, be sure to check out [The Code Book](http://www.simonsingh.net/the-code-book) by Simon Singh. This is great introduction to cryptography and cryptanalysis.

Think computer science is all fun and games? [The Codebreakers](http://david-kahn.com/book-david-kahn-code-codebreakers-cryptography.htm), by David Kahn, is a fascinating look at many of the problems solved by crypotanalysis, from breaking WWII secret messages to the very question of who wrote Shakespeare's works!

**The Assignment**

This assignment includes implementation of Hashtables, tests, and programs manipulating those structures. You will also be collecting experimental data and completing the homework questions. A large part of your work will involve using the WordCount program. WordCount should output the frequency of each word in the document, starting with the most frequent words and resolving ties in alphabetical order as shown below:

970 the

708 and

666 of

632 to

521 at

521 i

521 into

466 a

444 my

391 in

383 buffalo

...

Note that the actual printing code, along with many other parts, are provided. The printing format should exactly match what is shown above (use the provided printing code). Do not make WordCount print any other extra outputs such as number of words, time it takes to run, etc. Your WordCount should work as specified when given correct input parameters as shown below. Print an appropriate error message and terminate the program (System.exit) if an incorrect or incomplete set of parameters are given. As provided, Wordcount.java processes two parameters. If you call WordCount from the command line, it should take parameters as follows (in eclipse you will just add the argument listed below):

java WordCount [-s | -o] <Input file name>

* Argument 1: DataCounter implementation. -s for Hashtable with separate chaining,

-o for Hashtable with open addressing.

* Argument 2: Input file name.

**Homework Questions**

Submit a Microsoft Word document, answering the following questions:

1. Testing.
   1. How did you design your tests & what properties did you test?
   2. What boundary cases did you consider?
2. Conduct an experiment to determine which DataCounter implementation (HashTable\_SC, HashTable\_OP) is better for large input texts.
   1. Describe your experimental setup:
      1. Inputs used
      2. How you collected timing information
      3. Any details that would be needed to replicate your experiments
   2. Experimental Results

(Place your graphs and tables of results here).

* 1. Interpretation of Experimental Results
     1. What did you expect about the results and why?
     2. Did your results agree with your expectations? Why or Why not?
     3. According to your experiment, which Hashtable implementation, separate chaining or open addressing, is better?

1. Conduct experiments to determine if changing the hash function affects the runtime of your HashTable.
   1. Brief description of your hash functions
   2. Experimental Results

(Place your graphs and tables of results here).

Experiment with at least 2 hash functions (2 Hashing functions = 2 experiments depending on how you measured the runtime). Don’t forget to give each graph a title and label the axes.

* 1. Interpretation. Your expectations and why? Did it match your results? If not, why?

1. Using Correlator, does your experimentation suggest that Bacon wrote Shakespeare's plays? Show at least one (you can experiment with more texts if you want) correlation value for each of:
   1. Shakespeare's work compared to Shakespeare's work
   2. Bacon's work compared to Bacon's work
   3. Shakespeare's work compared to Bacon's work. According to the results of your experiments, did Bacon write Shakespeare's plays?
2. Include a description of how your project goes "above and beyond" the basic requirements (if it does).
3. Improvement
   1. Which parts of the project were most difficult?
   2. How could the project be better?

Question 2 above will ask you to design and run some experiments to determine which implementations are faster for various inputs. Answering these questions will require writing additional code to run the experiments, collecting timing information and producing result tables and graphs, together with relatively long answers. Do not wait until the last minute! Insert tables and graphs in your homework as appropriate, and be sure to give each one a title and label the axes for the graphs.

Question 3 above you will need to write a second hashing function. To exaggerate the difference between the two hash functions, you would want to compare a very simple hash function with a decent one (the one used in StringHasher).

All graphs and tables should be inserted into your homework file. For all experimental results, you should provide detailed interpretation, especially when the results do not match your expectations.

For more stable results, use average runtimes of multiple runs as shown in the example timing code below. Java optimizes repeated operations, so it runs faster at the end of the loop. Throw away several runs at the beginning of the loop to encounter this effect (JVM warmup).

private static double getAverageRuntime(String[] args)

{

double totalTime = 0;

for (int i = 0; i < NUM\_TESTS; i++)

{

long startTime = System.currentTimeMillis();

WordCount.main(args);

long endTime = System.currentTimeMillis();

if (i >= NUM\_WARMUP) // Throw away first NUM\_WARMUP runs to encounter JVM warmup

{

totalTime += (endTime - startTime);

}

}

return totalTime / (NUM\_TESTS-NUM\_WARMUP); // Return average runtime.

}

**Programming**

**From Blackboard**

Download the project from Blackboard to get started:

HashTables.zip

Unzip and place the HashTables folder in your workspace folder. From eclipse, import it using file -> import -> General -> Existing projects into workspace -> select HashTables as root directory. Note that the provided code will not compile or run correctly until you complete "1. Getting Started" below. Here are brief descriptions of the main provided files in roughly the order suggested that you read them.

DataCount.java

A simple container for an object and an associated count.

DataCounter.java

Like a dictionary, except it maintains a count for each data item. The element is not required to be element type E to be "comparable". Instead, constructors will take function objects of type Comparator and use them to do comparisons. Also notice that a DataCounter provides an iterator SimpleIterator<DataCount>.

SimpleIterator.java

The iterator that a DataCounter must provide. Do not use Java's iterator type because it obligates you to certain rules that are difficult to implement (if curious, read about concurrent modification exceptions).

Comparator.java

The type for function objects that do comparisons between pairs of elements. Constructors of DataCounter implementations should take it as an argument, as illustrated in Hashtable\_SC.java.

DataCountStringComparator.java

An implementation of Comparator that orders two DataCount objects. It requires that you correctly implement StringComparator.

WordCount.java

Processes a text file and prints out the words from the file in frequency order.

FileWordReader.java

Handles input files, converting each word into lower case and removing punctuation, so that every string you process will contain just the 26 lowercase English letters.

Hasher.java

Interface for a function object your hashtable implementation will want to take in as a parameter.

Note that you have been provided several other files in addition to the ones listed above. These other files are mostly full of stubs that you must fill in (there are instructions in the comments inside of those files). You will need to implement classes in the HashTables package including making several changes to WordCount.java. You should NOT modify the ProvidedCode package at all. You should \*NOT\* modify the provided structures, such as package names, file names, interfaces of public methods and constructors, etc. You also should \*NOT\* move the provided files to other packages. However, you may add new files (add files to packages \*other\* than ProvidedCode), and you may add new packages to the project as needed. You can add or modify private methods, and you can add public methods and constructors as long as doing so does not violate the style guidelines (i.e. too many unnecessary methods/classes, new method exposes sensitive internal data of the class).

**Implementation**

You will work on the simpler and helpful subproblem of computing the frequency of words in a text and compute a numeric value that quantifies the "similarity" of two texts. You will also perform some simple experiments to determine which of the Hashtable implementations performs better. For this assignment somewhat less guidance has been given on how to organize your code and implement your algorithms. It is up to you to make good design decisions.

1. Getting Started

After this step, running:

java WordCount [-s | -o] <filename>

For the code to compile and generate the correct output, you need to first implement the following:

* StringComparator: Used by both data counters and sorting algorithms. Because of how the output must be sorted in the case of ties, your implementation should return a negative number when the first argument to compare alphabetically comes before the second argument. Do not use any String comparison provided in the Java library; the only String methods you are allowed to use are length and charAt.
* WordCount.getCountsArray: The provided code returns with an error. Your code should use the argument's iterator to retrieve all the elements and put them in a new array. The code you write is using (not implementing) a SimpleIterator.

1. Adding Data Counter Implementations

Provide two additional implementations of DataCounter as described below. You should provide the methods defined in DataCounter. Follow the instructions & hints found inside of each file listed below.

* StringHasher: To use your HashTable in WordCount, you need to hash strings. Implement your own hashing strategy for strings. Do not use Java's hashCode method. As in StringComparator, the only String methods you are allowed to use are length and charAt.

* HashTable\_SC: You need to implement the HashTable with separate chaining. You are strongly encouraged, but not required, to use inner class for the node object. If you choose to use an external class for the node object, make sure you include the file in your submission.
* HashTable\_OA: You need to implement another Hashtable with open addressing. You only need to choose one from these three hashing strategies: linear probing, quadratic probing, or double hashing.

Both of your HashTable should rehash as appropriate (use an appropriate load factor), and its capacity should always be a prime number. Your HashTable should grow, and it should be able to handle an input size larger than the sample text files given to you (It should be able to grow at least up to 200,000).

1. Document Correlation

How to quantify the similarity between two texts is a large area of study. You should use the

simple approach below. Add code to Correlator.java to implement this simple approach:

* Calculate word counts for the two documents and use each document's length to create normalized frequencies so that frequencies can be meaningfully compared between documents of different lengths (i.e., use frequency percentages).
* Ignore words whose normalized frequencies are too high or too low to be useful. A good starting point is to remove or ignore words with normalized frequencies above 0.01 (1%) or below 0.0001 (0.01%) in both documents.
* Then, 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. This should be done using a \*SINGLE\* iterator. This means only 1 iterator being used in Correlator.java, \*NOT\* 1 iterator per DataCounter (You should call dataCounter.getIterator() just once in Correlator.java). Hint: Take advantage of DataCounter's methods.
* The final value of this running sum will be your difference metric, which corresponds to the square of the Euclidean distance between two vectors in the space of shared words in two documents. Note that this ignores any word that does not appear in both documents, which is probably the biggest weakness of this metric.
* Correlator.java should output a single number which is the computed correlation as described above (Do \*NOT\* print any extra output other than a single number printed from the provided printing code).
* Comparing Hamlet and The New Atlantis computes a similarity of 5.657273669233966E-4. Your answer should be very close to this one (like 5.65727366923\*\*\*\*E-4, where \* can be any number).

Correlator.java should accept command line parameters indicating which DataCounter to use and the filenames

java Correlator [-s | -o] <filename1> <filename2>

1. Tests

Similar with homework 3, be sure to test your solutions thoroughly and to turn in your testing code. Part of the grading will involve thorough testing including any difficult cases.

**What to Submit**

Important reminders:

* You are allowed to add methods and classes, but you should not change the names & interfaces of the given packages, classes and public methods and constructors. Also do not move the given classes to other packages. (Adding is ok, deleting & modifying public interfaces is not allowed)
* Make sure ALL the files that you add, including test files are in the HashTables package.
* You should not edit the files in the package ProvidedCode.
* You should not use any Java libraries in your implementation (using methods from the Math package is ok), except in your tests and timing code.
* Do not change the output (printing) format of WordCount and Correlator (used for grading). Also, do not produce any extra output other that what is printed using the provided printing code (for example, do not print the runtime in WordCount).
* For all subclasses of DataCounter, do not override the toString method of DataCounter (used for grading).
* Make sure your code is properly documented (recall the Programming Guidelines).

You should submit the following files:

* Correlator.java
* HashTable\_OA.java
* HashTable\_SC.java
* StringComparator.java
* StringHasher.java
* WordCount.java
* Any additional Java files needed, if any
* The Java files you used to test your Hashtable implementations
* The Java files you used to time your Hashtable implementations
* A Microsoft Word document containing answers to the Homework Questions.

**Above and Beyond**

You may do any or all of the following; pick ones you find interesting. Please be sure that you finished your project completely before attempting any of these. You should start on these only if you have time left after thoroughly testing all your code and carefully checking all your homework answers. Submit your extra credit files separately in extra.zip, and provide a very detailed explanation for each of your implementations in question 5 of the homework questions.

1. AVL tree: Implement AVL tree as an additional implementations of DataCounter. AVLTree should be self-balancing search tree.
2. Generic: Change the Hashtable implemetation to accept generic type rather than String type. It required changing some files in the ProvidedCode packages.