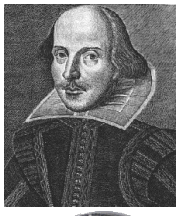

CSE 373 Homework 4: Shake-n-Bacon

Outline

Due Dates, Turn-In, and Rules

Find partner: Use the Discussion Board to find a partner & Complete this Catalyst Survey by 11:00PM Wednesday May 06, 2015

Code: Due 11:00PM Wednesday May 13, 2015

Read What to Turn In before you submit — poor submissions may lose points.

Turn in your assignment here, ***ONE*** submission per team

- Complete this project by yourself or with a partner.
- Remember Collaboration Policy, Grading Policy, and Programming Guidelines before you begin working on the project. In particular, note that the write-up you turn in is worth a substantial portion of the grade!

Working with a Partner

You are strongly encouraged, but not required, to work with a partner. You can use the Discussion Board to find a partner, and you should complete this Catalyst Survey by the above due date if you wish to work as a team. No more than two students may form a team. You may divide the work however you wish, but place the names of both partners at the top of all files (You may attribute one partner as the primary author). Team members will receive the same project grade unless there is an *extreme circumstance* (notify us *before the deadline*).

If you work on a team, you should:

1. Turn in only ***ONE*** submission including the write-up per team.
2. Work together and make sure you *both* understand the answers to all write-up questions.
3. Understand how your partner's code is structured and how it works.
4. Test all of your code together to be sure it properly integrates.

Introduction to Word Frequency Analysis

A world famous UW history professor wants you to settle a very old debate on who wrote Shakespeare's plays, Shakespeare or Sir 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) 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 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 is a good place to look.

The Assignment

This assignment includes implementation of Hashtables, tests, and programs manipulating those structures. You will also be collecting experimental data and completing write-up 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.

Provided Code

Download the project to get started:

CSE373_hw4.zip

Unzip and place the CSE373_hw4 folder in your workspace folder. From eclipse, import it using file -> import -> General -> Existing projects into workspace -> select CSE373_hw4 as root directory. We believe the provided code is correct, but let us know if you see any problems. Note that the provided code will not compile or run correctly until you complete "1. Getting Started" in Programming section. Here are brief descriptions of the main provided files in *roughly* the order we suggest 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. We do not require the 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. We do not use Java's iterator type because it obligates us 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 `shake-n-bacon` 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).

Programming

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. We purposely give somewhat less guidance 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`.

2. 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 discussed in the class. 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 discussed in the class. 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 as discussed in the class), 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).

3. Document Correlation

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

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).
- Comparing documents can be fun. Feel free to post links to interesting text examples on the course message board.

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

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

4. 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.

5. Experiment & Write-up

Submit a `README.pdf` file, answering the questions in this template README file: (`README.docx`)

Question 3 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 `README.pdf` as appropriate, and be sure to give each one a title and label the axes for the graphs. Place all your timing code into the package `writeupExperiment`. Be careful not to leave any write-up related code in the `shake_n_bacon` package (such as printing runtime in `WordCount`). To prevent losing points due to the modifications made for the write-up experiments, we recommend that you copy all files that need to be modified for the experiments into the package `writeupExperiment`, and start working from there. Files in different packages can have the same name, but when editing be sure to check if you are using the correct file!

You need to write a second hashing function for Question 4. 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 `README` file. For all experimental results, we would like to see a detailed interpretation, especially when the results do not match your expectations. For more stable results, we recommend averaging the runtime 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(NUM_WARMUP <= i) { // Throw away first NUM_WARMUP runs to encounter JVM warmup
totalTime += (endTime - startTime);
}
}
return totalTime / (NUM_TESTS-NUM_WARMUP); // Return average runtime.
}
```

Turn in Information

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 `shake_n_bacon` 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).

Code:

- `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
- `README.pdf`, containing answers to the Write-Up 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. We advise you to start on these only if you have time left after thoroughly testing all your code and carefully checking all your writeup answers. Recall that any extra credit you complete is noted and kept separate in the gradebook and may or may not be used to adjust your grade at the end of the quarter, as detailed in the course grading policy. Submit your extra credit files separately in `extra.zip`, and provide a very detailed explanation for each of your implementations in question 6 of `README.pdf`.

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 to change some files in the providedCode packages.

Interesting Tidbits

- Word-frequency analysis plays a central role in providing the input data for tag clouds. 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 by Simon Singh. This is great introduction to cryptography and cryptanalysis.

- Think computer science is all fun and games? The Codebreakers, 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!

Acknowledgments

A variant of this project was the third and final project in CSE326 for many years. The first word counter appeared during Spring 2000 when 326 was taught by the famous Steve Wolfman. The snowy-bearded historian appeared in Autumn 2001, courtesy of Adrien Treuille. The assignment was subsequently tweaked over the years by Matt Cary, Raj Rao, Ashish Sabharwal, Ruth Anderson, David Bacon, Hal Perkins, and Laura Effinger-Dean. Dan Grossman adapted it to be the second project in CSE332. James Fogarty stayed up late during the snow-pocalypse of 2012 to clarify confusions experienced in previous offerings (which had been carefully detailed by Ruth Anderson).