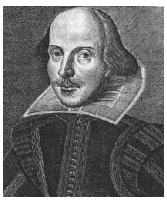
Last Updated: Jan 22, 2013



CSE 332 Winter 2013: Project 2 - Shake-n-Bacon

Outline

Due Dates and Turn-In

- Find a partner: As soon as possible, you must notify us before Wednesday Jan 30 if you plan on working with a partner
- Phase A: Wednesday Feb 6, 11PM
- Phase B: Tuesday Feb 19, 11PM

Use the course dropbox to submit your assignment electronically.

Working with a Partner

You are strongly encouraged, but not required, to work with a partner of your own choosing for this project. You may work with somebody you already know or use the course message board to help find a partner. No more than two students total may form a team. You may divide the work however you wish, under three conditions:

- You document each team member's effort in your write-up.
- You work together and make sure you *both* understand the answers to all write-up questions.
- You understand how your partner's code is structured and how it works.

Other logistics:

- Turn-in only **one** submission per team (i.e., only one set of Phase A code, only one set of Phase B code, only only one Writeup). It would help the course staff if the same partner submits all of these via their Catalyst account, but this is not required.
- Both names should appear in all files, although it is fine to attribute one partner as a primary author on a particular piece of code.

Test all of your code together to be sure it properly integrates. Start this early, and do not attempt to merge your code on the due date. You will very likely experience problems with integration. You may wish to set up an SVN repository to make collaboration easier. If so, contact the course staff regarding project space.

Team members will receive the same project grade unless there is an *extreme circumstance* and you notify us *in advance of the deadline*.

If you plan to work with a partner, one partner MUST send Daniel an email using this link. In your email, provide the following information for BOTH partners: full name, UWNetID, CSE Email address.

Introduction and Word-Frequency Analysis

You have just been approached by a world famous UW history professor. He would like 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. In Phase A, you will work on the simpler and helpful subproblem of computing the frequency of words in a text. In Phase B, you will compute a numeric value that quantifies the "similarity" of two texts.

Perhaps your work can be applied to the works attributed to Shakespeare and Bacon to settle the ancient debate. The professor has provided you with 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.

When working with document correlations, it is often desirable to work only with the roots of words. In this way, "sleeps", "sleeping", and "sleep" are all considered to be the same word. This process is called *word stemming*, and is used in search engines and many other contexts. However, proper stemming is difficult. Dropping an 's' from the end of a word seems natural but turns "bus" into "bu". Dropping punctuation seems like a good idea but it equates "it's" and "its". Implementing a decent stemming algorithm (such as Porter Stemming) is above and beyond work for this project.

The code provided to you does not really do stemming, but it does normalize the input as follows:

- It converts all letters to lower-case, so "An" and "an" are the same word.
- It removes all punctuation from words, despite this being occasionally wrong.

Hence every string you process will contain just the 26 lowercase English letters (although this doesn't deeply impact the project).

Learning Objectives

For this project, you will:

- Be introduced to `DataCounter`, a variant of the `Dictionary` ADT, and understand the strengths of various `DataCounter` implementations
- Gain proficiency with heaps, AVL trees, and hashtables
- Implement and use two idioms related to writing generic and reusable code: function objects and iterators
- Implement sorting algorithms
- Gain experience with unit testing and the JUnit tool
- Gain preliminary experience with timing code and evaluating experimental data

The Assignment

This assignment consists of two phases. Each phase includes implementation of data structures, JUnit tests, and programs manipulating those structures. Phase B also involves collecting some experimental data and completing write-up questions.

Provided Code

You need several files provided in this single zip file:

project2files.zip

Create a new Eclipse project with these files. The files are described below, including how the code is organized. Note the provided code will not compile or run correctly until you complete the first part of Phase A.

We believe the code provided to you is correct (let us know of problems). You will need to add several additional files and make several changes to `WordCount.java`. Here are brief descriptions of the provided files in *roughly* the order we suggest understanding 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 (instead of storing an arbitrary value). We do not require the element type `E` to be "comparable". Instead, constructors for implementations will take function objects of type `Comparator` and use them to do comparisons. Also notice that a `DataCounter` provides an iterator `SimpleIterator<DataCount<E>>`.

`SimpleIterator.java`
: The iterator that a `DataCounter` must provide. We do not use Java's iterator type because doing so obligates us to certain rules that are difficult to implement properly (if curious, read about "concurrent modification exceptions").

`Comparator.java`
: The type for function objects that do comparisons between pairs of some element type. Constructors of `DataCounter` implementations should take a `Comparator` as an argument, as illustrated in `BinarySearchTree`.

`BinarySearchTree.java`
: An implementation of `DataCounter` using a (you guessed it) binary search tree. You should not modify this file, but it is a good example of how to do function objects and iterators. The iterators you write will not be as difficult.

`WordCount.java`
: The main file for Phase A. You will need to make several changes to this file. It processes a file and prints out the words from the file in frequency order. More detail is provided in the description below. In Phase B, you will write a different main class.

`TestBinarySearchTree.java`
: A class containing JUnit tests for `BinarySearchTree`. This is just an example that may or may not help to inspire you while writing test files for the data structures you implement.

`GStack.java`
: Same generic stack interface you saw in Project 1.

`FileWordReader.java`
: Handles input, converting each word into lower-case and removing punctuation.

`DataCountStringComparator.java`
: An implementation of `Comparator` that orders two `DataCount<String>` objects in the proper order for Phase A. It requires that you you correctly implement `StringComparator`, as detailed below.

`PriorityQueue.java`
: Interface your heap will implement. Your heap should use a function object for comparisons, just like the `DataCounter` implementations.

`Hasher.java`
: Interface for a function object your hashtable implementation will want to take as a parameter.

Phase A Programming

Running the `WordCount` program should output the frequency of each word in the document, starting with the most frequent words and resolving ties in alphabetical order. For example, output might look like this:

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

Note the actual printing code, along with many other parts, are provided. Getting a correct version of the output will require only a few additions. You will then implement different data-counters and sorting routines to see different ways to solve this problem. We next explain all the command-line options you need to support, but we do not recommend that you start by implementing all of these. Instead, go one step at a time, as outlined below, and add command-line options when it is convenient.

Command-Line Arguments

Overall, the `WordCount` program you submit for Phase A takes arguments as follows:

```
java WordCount [ -b | -a | -m | -h ] [ -is | -hs | -os | -k <number>] <filename>
```

- The first argument is which `DataCounter` implementation to use: `-b` for unbalanced binary search tree, `-a` for AVL tree, `-m` for move-to-front list, or `-h` for hashtable. Because hashtable is part of Phase B, your Phase A submission can ignore this parameter or print an appropriate error message.
- The second argument is which sorting method to use: `-is` for insertion sort (given to you, once you provide string-comparison), `-hs` for heapsort, `-os` for other, or `-k` followed by a number for printing only the top-k words rather than sorting all of them. Because `-os` and `-k` are part of Phase B, your Phase A submission can ignore these parameters or print an appropriate error message.
- The third argument is the input file to process.

For example, at the end of phase A, your code should work with command lines like the following:

```
java WordCount -b -is filename

java WordCount -a -hs filename
```

You will need to modify `Wordcount.java` to process these parameters. As provided, it only processes a single parameter for the filename.

Getting Started

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

- Provide a `GArrayStack` implementation. It is needed by the iterator for `BinarySearchTree`. You can probably just copy this file from your Project 1 solution.
- Provide a `StringComparator` class that implements `Comparator<String>`. This comparator is used by the provided code for both data-counters and sorting. 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` comes first alphabetically. Do *not* use any `String` comparison provided in Java's standard library; the only `String` methods you should use are `length` and `charAt`.
- Implement the static method `getCountsArray` in `WordCount`. 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`. If you have trouble with casting, take another look at these notes on generic arrays.

At this point your program should produce the correct output (implicitly using options `-b` and `-is`, since you do not yet process command-line parameters).

Adding Other Data-Counter Implementations

Provide two additional implementations of `DataCounter<E>` as described below. Provide appropriate JUnit tests for each data structure implementation. Use the appropriate implementation based on the first command-line argument.

- **AVL tree:** Create a class `AVLTree<E>` that implements `DataCounter<E>` using AVL trees. You need to provide only the methods defined in `DataCounter`. Your implementation *must be a subclass of* `BinarySearchTree<E>` and must use inheritance and calls to superclass methods to avoid unnecessary duplication or copying of functionality. Helpful Hints:
  - Create a subclass of `BSTNode`, perhaps named `AVLNode`.
  - Because all tree nodes must be instances of `AVLNode`, you will need to use overriding to replace the `incCount` method such that it creates `AVLNode` instances instead of `BSTNode` instances.
  - Do not attempt to "replace" the `left` and `right` fields in `BSTNode` with new `left` and `right` fields in `AVLNode`. This will instead mask the super-class fields (i.e., the resulting node would actually have four node fields, with code accessing one pair or the other depending on the type of the references used to access the instance). Such masking will lead to highly perplexing and erroneous behavior. Instead, continue using the existing `BSTNode` `left` and `right` fields. Cast their values to `AVLNode` whenever necessary in your `AVLTree`. This may feel wrong, and it may be quite a few casts, but is a reasonable solution given the goal of reusing methods from the BinarySearchTree implementation.
- **Move-to-front list:** Create a class `MoveToFrontList<E>` that implements `DataCounter<E>` using a linked list as follows:
  - The list is typically not sorted.
  - Add new items (with a count of 1) to the front of the list.
  - Whenever an existing item has its count incremented by `incCount` or retrieved by `getCount`, *move it to the front of the list*. That means you delete the node from its current list position and make it the first node in the list.
  - You need to implement an iterator. The iterator should *not* move elements to the front.

  Although this data structure has O(n) worst-case time operations, it does well when some words are much more frequent than others (because the frequent ones will end up near the beginning of the list). The write-up questions will explore this a bit further.

Adding a Heap and Heapsort

Provide an implementation of `PriorityQueue<E>` in class `FourHeap<E>`. Your data structure should be like the binary heap we studied, except nodes should have ***four*** children instead of two. Only leaves and at most one other node will have fewer children. You should use an efficient array-based implementation. (Hint: Complete written homework #2 before attempting this. As in homework #2, consider beginning with array index 0. Your index arithmetic should be simple.) Develop appropriate JUnit tests.

Now provide a different sorting algorithm using a priority queue. This algorithm is called *heapsort* and works as follows:

- Insert each element to be sorted in the priority queue.
- Then remove each element, storing the results in order in the array that should hold the sorted result.

Put your algorithm in a static method `heapSort` in `WordCount.java`. For full credit, your sorting routine should be generic like `insertionSort` (the two should have the same arguments).

Adjust `WordCount` to use your `heapSort` method when the second command-line argument is `-hs`.

Submission

Submit all your new files, named exactly as follows:

- `GArrayStack.java`
- `StringComparator.java`
- `AVLTree.java`
- `MoveToFrontList.java`
- `FourHeap.java`
- `WordCount.java` (the only provided file you should modify)
- Additional Java files you created for testing

Make sure your code is properly documented (recall the Programming Guidelines). Note that the write-up questions are all submitted with Phase B, but some of them refer to work you did in Phase A. You may wish to start on them early.

Phase B Programming

In Phase B, you will provide an additional counter implementation, an additional sorting implementation, and support for printing only the *k* most-frequent words in the file. You will then write a different main program that computes the similarity of two files. Finally, you will perform some simple experimentation to determine which implementations are fastest. In this phase, 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.

Another Counter and Sorter

Include the two implementations described below, together with appropriate JUnit tests.

- **Hashtable:** Create a class `Hashtable<E>` that implements `DataCounter<E>` using a hashtable. You need to provide only the methods defined in `DataCounter`. You may implement any kind of hashtable discussed in class; the only restriction is that it should not restrict the size of the input domain (i.e., it must accept any key) or the number of inputs (i.e., it must grow as necessary). You should use this implementation with the `-h` option. Hint: To make your hashtable generic using the function-object approach, have the constructor take both a `Comparator<E>` and a `Hasher<E>`.
- **Sorting:** Implement either quicksort or mergesort and use your implementation with the `-os` option. As with the provided insertionsort and your heapsort, your sort code should be generic.

To use your hashtable for `WordCount`, you will need to be able to hash strings. Implement your own hashing strategy using `charAt` and `length`. Do *not* use Java's `hashCode` method.

Top *k* Words

Extend `WordCount` such that if the second argument is `-k`, then the third argument is a number (and the fourth argument is the filename). When the `-k` option is used, `WordCount` should print out only the *k* most frequent words. If the text has fewer than *k* distinct words, then print all word frequencies. If there are ties for the *k* most frequent words, resolve them alphabetically so that you still print exactly *k* words. You should still print the words in the same order: most-frequent first with ties resolved alphabetically.

An easy way to implement this feature would be to sort all the words by frequency and then just print *k* of them. This approach finds the *k* most frequent words in time *O(n log n)*, where *n* is the number of distinct words. You will need to implement this simple approach for comparison purposes in your experimentation (see the write-up questions), but `WordCount` should be configured to use a more efficient approach that runs in time *O(n log k)* (assuming *k* is less than *n*).

Helpful Hints:

- Use a priority-queue, but never put more than *k* elements into it.

- Efficiently tracking the *k* most-frequently occurring words will require a *different* comparator than you used when implementing heapsort.

Document Correlation

How to quantify the similarity between two texts is a large area of study. We will use a simple approach, leaving more advanced models such as Latent Semantic Indexing to Above and Beyond. 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 or fractions).
- Ignore words whose frequencies are too high or too low to be useful. A good starting point is to remove words with normalized frequencies above 0.01 (1%) or below 0.0001 (0.01%) in both documents. Feel free to play with these percentages when determining who wrote Shakespeare's plays, but for grading purposes report results and submit code using these percentages.
- 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.
- 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 ignores any word that does not appear in both documents, which is probably the biggest weakness of this metric.

Write a class `Correlator` with a `main` method that computes the correlation between two documents. Its only output should be a single floating-point number which is the correlation computed as described above. Command-line flags should specify which `DataCounter` to use, as in `WordCount`:

```
java Correlator [ -b | -a | -m | -h ] <filename1> <filename2>
```

For example, your correlator should work with command lines like the following:

```
java Correlator -b filename1 filename2

java Correlator -h filename1 filename2
```

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

One implementation comparing *Hamlet* to *The New Atlantis* computes a similarity of 5.657273669233966E-4. An answer close to this suggests you are probably on the right track.

Experimentation

The write-up questions 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 and collect timing information. For example, writing code that runs word-counting for various values of *k* will be much easier than manually running your program with different values of *k*.

Submission

Submit ***all* your files**, whether or not you have changed them. This includes the provided interfaces, everything from Phase A, and **at least** (please submit all files needed to run your code):

- `Hashtable.java`
- `Correlator.java`
- `Wordcount.java`
- Any additional Java files you created for testing and experimentation.

You should fix bugs and design weakness in your Phase A code. Doing so will help your grade, but not as much as having a correct and elegant Phase A submission.

Write-Up Questions (Due with Phase B)

Submit a report answering the following questions. Note that the final 3 questions require writing code, collecting data, and producing results tables and/or graphs together with relatively long answers. So do not wait until the last minute.

You may submit a `txt` file with separate image files for your charts. You may also insert your charts inline in an `html` or `pdf` file. Formatting is not a component of your grade, as long as the report is clear and readable.

1. Who is in your group? (only one writeup needs to be submitted per group).
2. What assistance did you receive on this project? Include anyone or anything *except* your partner, the course staff, and the printed textbook.
3. How long did the project take? Which parts were most difficult? How could the project be better?
4. What "above and beyond" projects did you implement? What was interesting or difficult about them? Describe how you implemented them.
5. How did you design your JUnit tests? What properties do they test? What do they not test? What boundary cases did you consider?
6. Why does the iterator for a binary search tree need an explicit stack? If the iterator were re-written for use especially with AVL trees, how could you avoid the possibility of resizing the stack in the midst of iteration (which may require changing the interface for `GStack`)?
7. If a `DataCounter`'s iterator returned elements in the order needed for printing the most-frequent words first, then you would not need a `PriorityQueue`. For each `DataCounter` implementation, could such an iterator be implemented efficiently? Explain your answers.
8. For your `Hashtable` to be *correct* (not necessarily *efficient*), what must be true about the arguments to the constructor?
9. For `WordCount`, which `DataCounter` implementation and sorting implementation tend to produce the fastest results for large input texts? How did you determine this? Are there (perhaps contrived) texts that would produce a different answer, especially considering how `MoveToFrontList` works? Does changing your hashing function affect your results? Describe the experiments you ran to determine your answer and submit experimental results (e.g., in a table). Relevant experimental set-up should describe *at least* the texts used and how you collected timing information, but should probably include additional details that would be needed to replicate your experiments.
10. For `WordCount`, design and conduct experiments to determine whether it is faster to use your *O(n log k)* approach to finding the top *k* most-frequent words or the simple *O(n log n)* approach (using the fastest sort you have available). The relative difference in speed surely depends on *k*, so produce a graph showing the time for the two approaches for various values of *k* ranging from very small (i.e., 1) to very large (i.e., *n*). How could you modify your implementation to take advantage of your experimental conclusions?
11. Using `Correlator`, does your experimentation suggest that Bacon wrote Shakespeare's plays? We do not need a fancy statistical analysis. This question is intended to be fun and simple. Give a 1-2 paragraph explanation.
12. If you worked with a partner:
    a. Describe the process you used for developing and testing your code. If you divided it into pieces, describe that. If you worked on everything together, describe the actual process used. For example, discuss how long you talked about what, in what order you wrote and tested the code, and how long that required.
    b. Describe each group member's contributions/responsibilities in the project.
    c. Describe at least one good thing and one bad thing about the process of working together.

Above and Beyond (Due with Phase B)

You may do any or all of the following; pick ones you find interesting. Submit your 'above and beyond' files separately in a zip file.

1. *Alternate Hashing Strategies:* Implement both closed and open addressing and perform experimentation to compare performance. Also design additional hashing functions and determine which affects performance

more: the cost of hashing, the collision-avoidance of hashing, or your addressing strategy.

2. *Excluded Words:* Adjust `WordCount` and `Correlator` to take an additional file that contains words that should be ignored. For example, the file might contain very common words (like `the`, `a`, and `she`) or it might be another text (so `WordCount` would then report words not appearing in the other text). There are multiple ways to implement this feature: you could skip words-to-be-ignored as you process the input file, or when sorting the elements, or when printing them. Which is fastest and why do you think this is? What data structure did you use to determine efficiently if a word should be ignored and why is this efficient? Note that for top-*k*, you should still print *k* words (assuming there are *k* words that should not be ignored). How does this affect the ways you can implement this skip-some-words feature?
3. *Introspective Sort:* Introspective sort is an unstable quicksort variant which switches to heapsort for inputs which would result in a $O(n^2)$ running-time for normal quicksort. Thus, it has an average-case and a worst-case runtime of O(n log n), but generally runs faster than heapsort even in the worst case. Implement IntrospectiveSort, and give a sample input which would result in a quadratic runtime for normal quicksort (using a median-of-3 partitioning scheme).
4. *Word Stemming:* The introduction discussed how it is difficult to remove suffixes, plurals, verb tenses, etc., from words, but it is important when performing text analysis (e.g., for web search engines). One common algorithm is Porter Stemming. Implement this algorithm. Try to do so without looking at the source code posted at the link. If you do look at their source code, cite it properly.
5. *Word Co-Occurrence:* A phenomenon even more interesting than word frequency is word co-occurrence. Create a new `WordCount` that counts the frequency of pairs of words. Your code should insert as a pair any two words which appear within *c* words of each other, where *c* is a command-line argument (try 10 for test runs).
6. *Better Correlation Models:* Research better ways to quantify the correlation between two text documents and what their advantages/disadvantages are. Describe what you found and implement an algorithm more sophisticated than the simple one in `Correlator`. Explain how your algorithm affects your conclusions.

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).
