CSC 207 **Assignment 5: Doublets** **Spring 2011**

Assigned: Tuesday 1 March 2011

Due: Tuesday 8 March 2011

Topics: Polymorphism, Collections, Maps, Sets, Queues, Stacks

Collaboration: This homework assignment may be completed individually or in pairs.

Submission: Follow the instructions for submitting programs via P-Web and handing in a printed copy. Be sure to generate unit tests and a statement justifying their sufficiency.

Introduction

In 1879, Lewis Carroll (author of *Jabberwocky* and *Through the Looking-Glass*, among other classics), documented a word puzzle he termed "Doublets," which you may have seen. He wrote:

> DEAR VANITY [FAIR], —Just a year ago last Christmas, two young ladies—smarting under that sorest scourge of feminine humanity, the having "nothing to do"—besought me to send them "some riddles." But riddles I had none at hand, and therefore set myself to devise some other form of verbal torture which should serve the same purpose. The result of my meditations was a new kind of Puzzle—new at least to me—which, now that it has been fairly tested by a year's experience and commended by many friends, I offer to you, as a newly-gathered nut, to be cracked by the omnivorous teeth which have already masticated so many of your Double Acrostics.
>
> The rules of the Puzzle are simple enough. Two words are proposed, of the same length; and the Puzzle consists in linking these together by interposing other words, each of which shall differ from the next word in *one letter only*. That is to say, one letter may be changed in one of the given words, then one letter in the word so obtained, and so on, till we arrive at the other given word. The letters must not be interchanged among themselves, but each must keep to its own place. As an example, the word "head" may be changed into "tail" by interposing the words "heal, teal, tell, tall." I call the two given words "a Doublet," the interposed words "Links," and the entire series "a Chain," ...
>
> It is perhaps, needless to state that it is *de rigeuer* that the links should be English words, such as might be used in good society. ...
>
> I am told that there is an American game involving a similar principle. I have never seen it, and can only say of its inventors, *"pereant qui ante nos nostra dixerunt!"*

Your task in this assignment will be to build the underlying data structures that will allow you to find a chain of links yielding a solution to Carroll's doublets for any pair of words. In the next assignment, you'll complete the search algorithm and do some preliminary experiments with it.

Links

Your first task is to build a data structure that represents the allowable links for each word. For example, we should be able to efficiently determine that the candidate plays from *heal* include *heel*, *hell*, *heat*, *heap*, *real*, *hear*, *deal*, *zeal*, *seal*, *peal*, *head*, *veal*, and *meal*, while the candidates from *glass* are *grass*, *glads*, *class*, *gloss*.

Create a class called `Links` that contains such a structure. The constructor should take a file name as a parameter and call a private helper method to populate the links data with word information from the file, which should have one word per line. Include a member method that takes a string a returns a collection of the valid connections (i.e., candidate plays) from the given word or `null` if there are none.

Note: Building this structure may have a high time complexity. However, the structure itself should facilitate the low complexity operations that will be necessary to play the game.

Note: Java supports labeled `continue` statements.

Chains

Create a class `Chain` representing an immutable sequence of link words. It should support adding a new word to the (end) of the chain, as well as retrieving the last word in the chain (i..e., to see if it is the desired word). Because the objects are immutable, adding a word should return a new `Chain`.

Your class should be an `Iterable<String>`, so that you can easily look over the contents (in order) and ultimately print the solution. Include a `length` method as well.

When searching for a chain between the doublets, it will be important not to repeat any words (otherwise your solution will be unnecessarily long). Thus, to ensure your chain does not repeat words, you should include a boolean method `contains` to determine *efficiently* whether a given word appears in the chain.

Note: You will likely need to use `clone` operations to ensure immutability. Because `clone` returns an `Object`, you will likely need to cast it to the type being cloned.

Note: You can do this with the collection types we have studied. However, for optional code economy, examine the Java Collections framework for a single collection data structure that does nearly everything required.

Managing Chains

Starting from the first doublet, there are many possible links that could lead to a solution. For instance in finding a valid chain for Carroll's doublet from *head* to *tail*, you may decide to use a link from *head* to *heal,* or you might decide instead to use a link from *head* to *hear*.

Which one should you choose? If you are playing cereberally, you might choose the link that seems most promising. However, it could end up being a dead-end, requiring you to resort to some alternative link.

Computationally, this process requires storing the alternatives you intend perhaps to try later if your search should come up short. Of course, you'd also need a method to retrieve one of those alternatives for examination. This should sound much like the capabilities of two collections you have studied recently featuring slightly different spins on "put one in" and "take one out" capabilities. Because the behavior can differ, we may want to use them interchangeably to guide our search,

Interface

To take advantage of polymorphism, create a `ChainManager` interface that supports two core methods: `add`, which simply adds a chain for future consideration, and `next`, which returns (and removes) a chain.

In addition to these, we may be interested in tracking the performance of the manager. Include two other methods, `numNexts`, which returns the number of times `next` has been called, and `maxSize`, which returns the the largest number of chains the manager has stored so far.

Implementations

Create two implementations of `ChainManager`, one called `QueueChainManager` that treats chains in a FIFO fashion, and another called `StackChainManager` that treats chains in a LIFO fashion. The `next` method should return `null` when there are no more chains left to consider.

Note: `java.util.LinkedList` *IS-A* `java.util.Queue`. Your `QueueChainManager` implementation should only use `Queue` methods.

Data

You can find several word list files (derived from SCOWL) on the MathLAN at `~weinman/courses/CSC207/misc/english-words.*.txt`. The number in the file name indicates the frequency percentile of the word usage. Thus, `english-words.10.txt` is a shorter list, but its words are more common than those who appear (additionally) in `english-words.20.txt`. The lists are strict subsets of one another.

Acknowledgments While the original puzzle is attributable to Lewis Carroll, the inspiration for the problem as a programming assignment is due to Owen Astrachan's 2001 "Word Ladder Nifty Assignment", http://www.cs.duke.edu/csed/poop/ladder