**Word Ladders**

(This assignment was copied from Stanford and modified slightly)

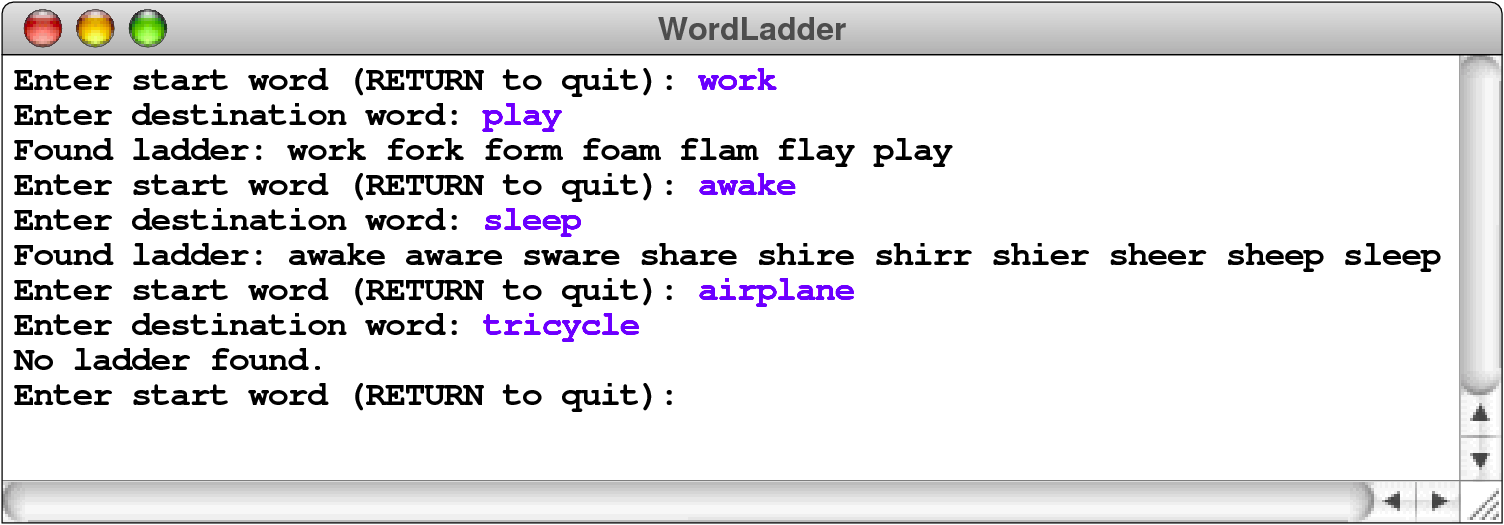
A ***word ladder*** is a connection from one word to another formed by changing one letter at a time with the constraint that at each step the sequence of letters still forms a valid word. For example, here is a word ladder connecting **"code"** to **"data"**.

**code** → **core** → **care** → **dare** → **date** → **data**

That word ladder, however, is not the shortest possible one. Although the words may be a little less familiar, the following ladder is one step shorter:

**code** → **cade** → **cate** → **date** → **data**

Your job in this problem is to write a program that finds a minimal word ladder between two words. Your code will make use of several of the ADTs along with a powerful algorithm called breadth-first search to find the shortest such sequence. Here, for example, is a sample run of the word-ladder program in operation:



**A sketch of the word ladder implementation**

Finding a word ladder is a specific instance of a ***shortest-path problem****,* in which the challenge is to find the shortest path from a starting position to a goal. Shortest-path problems come up in a variety of situations such as routing packets in the Internet, robot motion planning, determining proximity in social networks, comparing gene mutations, and more.

One strategy for finding a shortest path is the classic algorithm known as ***breadth-first search****,* which is a search process that expands outward from the starting position,considering first all possible solutions that are one step away from the start, then allpossible solutions that are two steps away, and so on, until an actual solution is found.Because you check all the paths of length 1 before you check any of length 2, the firstsuccessful path you encounter must be as short as any other.

For word ladders, the breadth-first strategy starts by examining those ladders that are one step away from the original word, which means that only one letter has been changed. If any of these single-step changes reach the destination word, you’re done. If not, you can then move on to check all ladders that are two steps away from the original, which means that two letters have been changed. In computer science, each step in such a process is called a ***hop****.*

The breadth-first algorithm is typically implemented by using a queue to store partial ladders that represent possibilities to explore. The ladders are enqueued in order of increasing length. The first elements enqueued are all the one-hop ladders, followed by the two-hop ladders, and so on. Because queues guarantee first-in/first-out processing, these partial word ladders will be dequeued in order of increasing length.

To get the process started, you simply add a ladder consisting of only the start word to the queue. From then on, the algorithm operates by dequeueing the ladder from the front of the queue and determining whether it ends at the goal. If it does, you have a complete ladder, which must be minimal. If not, you take that partial ladder and extend it to reach words that are one additional hop away, and enqueue those extended ladders, where they will be examined later. If you exhaust the queue of possibilities without having found a completed ladder, you can conclude that no ladder exists.

It is possible to make the algorithm considerably more concrete by implementing it in

***pseudocode****.* The pseudocode for the word-ladder problem appears in Figure 1.

**Figure 1. Pseudocode implementation of the word-ladder algorithm**

*create an empty queue.*

*add a ladder containing the start word to the end of the queue*

**while (***the queue is not empty***) {**

*dequeue a ladder from the queue*

**if (***the final word in this ladder is the destination word***) {**

*return this ladder as the solution*

**}**

**for (***each word in lexicon that differs from last word in ladder by 1 letter***) {**

**if (***that word has not already been used in a ladder***) {**

*create a copy of the current ladder*

*add the new word to the end of the copy*

*add the new ladder to the end of the queue*

**}**

**}**

**}**

*report that no word ladder exists.*

As is generally the case with pseudocode, several of the operations that are expressed in English need to be fleshed out a bit. For example, the loop that reads

***for (****each word in lexicon that differs from last word in latter by 1 letter****)***

is a *conceptual* description of the code that belongs there. It is, in fact, unlikely that this idea will correspond to a single **for** loop in the final version of the code. The basic idea, however, should still make sense. What you need to do is iterate over all the words that differ from the current word by one letter. One strategy for doing so is to use two nested loops; one that goes through each character position in the word and one that loops through the letters of the alphabet, replacing the character in that index position with each of the 26 letters in turn. Each time you generate a word using this process, you need to look it up in the lexicon of English words to make sure that it is actually a legal word.

Another issue that is a bit subtle is the restriction that you not reuse words that have been included in a previous ladder. One advantage of making this check is that doing so reduces the need to explore redundant paths. For example, suppose that you have previously added the partial ladder

**cat** → **cot** → **cog**

to the queue and that you are now processing the ladder

**cat** → **cot** → **con**

One of the words that is one hop away from **con**, of course, is **cog**, so you might be tempted to enqueue the ladder

**cat** → **cot** → **con** → **cog**

Doing so, however, is unnecessary. If there is a word ladder that begins with these four words, then there must be a shorter one that, in effect, cuts out the middleman by eliminating the unnecessary word **con**. In fact, as soon as you’ve enqueued a ladder ending with a specific word, you never have to enqueue that word again.

The simplest way to implement this strategy is to keep track of the words that have been used in any ladder (which you can easily do using another lexicon) and ignore those words when they come up again. Keeping track of what words you’ve used also eliminates the possibility of getting trapped in an infinite loop by building a circular ladder, such as

**cat** → **cot** → **cog** → **bog** → **bag** → **bat** → **cat**

One of the other questions you will need to resolve is what data structure you should use to represent word ladders. Conceptually, each ladder is just an ordered list of words, which should make your mind scream out “ArrayList!” (Given that all the growth is at one end, stacks are also a possibility, but ArrayList's will be more convenient when you are trying to print out the results.) The individual components of the **ArrayList** are of type **String**.

**Implementing the application**

At this point, you have everything you need to start writing the actual Java code to get this project done. It’s all about leveraging various classes—you’ll find your job is just to coordinate the activities of Queues, ArrayList's, and Lexicons (a class you will create) necessary to get the job done. The finished assignment requires less than a page of code, so it’s not a question of typing in statements until your fingers get tired. It will, however, certainly help to think carefully about the problem before you actually begin that typing.

As always, it helps to plan your implementation strategy in phases rather than try to get everything working at once. Here, for example, is one possible breakdown of the tasks:

* *Task 1- Create and test the* ***Lexicon*** *(a word list) class.* This class should include the below methods:

|  |  |
| --- | --- |
| **Constructor** | |
| [**Lexicon()**](https://www.stanford.edu/class/cs106b/cppdoc/Lexicon-class.html#Constructor:Lexicon) | Creates an empty lexicon. |
| [**Lexicon(fileName)**](https://www.stanford.edu/class/cs106b/cppdoc/Lexicon-class.html#Constructor:Lexicon) | Initializes a new lexicon by reading the file and adding all of its words to the lexicon. |
| **Methods** | |
| [**add(word)**](https://www.stanford.edu/class/cs106b/cppdoc/Lexicon-class.html#Method:add) | Adds the specified word to the lexicon. |
| [**contains(word)**](https://www.stanford.edu/class/cs106b/cppdoc/Lexicon-class.html#Method:contains) | Returns **true** if **word** is contained in the lexicon. |
| [**isEmpty()**](https://www.stanford.edu/class/cs106b/cppdoc/Lexicon-class.html#Method:isEmpty) | Returns **true** if the lexicon contains no words. |
| [**getOneLetterAwayWords(word)**](https://www.stanford.edu/class/cs106b/cppdoc/Lexicon-class.html#Method:mapAll) | Returns an ArrayList of words in the lexicon that differ from word by one letter. |
| [**size()**](https://www.stanford.edu/class/cs106b/cppdoc/Lexicon-class.html#Method:size) | Returns the number of words contained in the lexicon. |
| [**toString()**](https://www.stanford.edu/class/cs106b/cppdoc/Lexicon-class.html#Method:toString) | Converts the lexicon to a printable string representation. |

• *Task 2—Play around with the lexicon*. Before you write the word ladder application, you might experiment with a simpler program that uses the lexicon in simpler ways. For example, you might write a program that reads in a word and then prints out all the English words that are one letter away.

• *Task 3—Think carefully about your algorithm and data-structure design.* Be sure you understand the breadth-first algorithm and what data types you will be using.

• *Task 4—Implement the breadth-first search algorithm.* Now you’re ready for the meaty part. The code is not long, but it is dense, and all those templates will conspire to trip you up. We recommend writing some test code to set up a small dictionary (with just ten words or so) to make it easier for you to test and trace your algorithm while you are in development. Test your program using the large dictionary only after you know it works in the small test environment.

Note that breadth-first search is not the most efficient algorithm for generating minimal word ladders. As the lengths of the partial word ladders increase, the size of the queue grows exponentially, leading to exorbitant memory usage when the ladder length is long and tying up your computer for quite a while examining them all. Later in this course, we will touch on improved search algorithms.