Lab 10

Due: 8 April

Handout 10
CSC 062: Spring, 2007
2 April

Word Ladders

A word ladder is a type of puzzle invented by Lewis Carroll in 1878. The idea is to transform one word to another a step at a time by forming a sequence of words each of which differs from its predecessor by a single letter at a time. A simple example is the word ladder from cat to dog:

$$\text{cat} \rightarrow \text{cot} \rightarrow \text{cog} \rightarrow \text{dog}$$

Other examples include code to data:

$$\text{code} \rightarrow \text{cade} \rightarrow \text{cate} \rightarrow \text{date} \rightarrow \text{data}$$

and tears to smile

$$\text{tears} \rightarrow \text{sears} \rightarrow \text{spars} \rightarrow \text{spare} \rightarrow \text{spire} \rightarrow \text{spile} \rightarrow \text{smile}$$

Your program will ask the user to enter a start and a destination word and then your program is to find a word ladder between them if one exists. By using an algorithm known as breadth-first search, you are guaranteed to find the shortest such sequence. The user can continue to request other word ladders until they are done.

Here is some sample output of the word ladder program:

```
Enter start word (RETURN to quit): work
Enter destination word: play
Found ladder: work fork form foam flam flay play
Enter start word (RETURN to quit): awake
Enter destination word: sleep
Found ladder: awake aware sware share shire shirr shier sheer sheep sleep
Enter start word (RETURN to quit): airplane
Enter destination word: tricycle
No ladder found.
```

Solving Word Ladders

Finding a word ladder is an instance of a shortest path problem, where 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 packet routing, robot motion planning, social networks, studying gene mutations, and more. One approach for finding a shortest path is to use a breadth-first search. A breadth-first search searches outward from the start in a radial fashion until it hits the goal. For word ladders, this means first examining those ladders that represent "one hop" (i.e. one changed letter) from the start. If any of these reach the destination, we're done. If not, the search now examines all ladders that add one more hop (i.e. two changed letters). By expanding the search radially at each step, all one-hop ladders are examined before two-hops, and three-hop ladders only considered if none of the one-hop nor two-hop ladders worked out, thus the algorithm is guaranteed to find the shortest successful ladder.

As we saw earlier in the term, a breadth-first is typically implemented using a queue. The queue is used 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. As a result, ladders will be dequeued in order of increasing length. The algorithm operates by dequeueing the front ladder from the queue and determining if it reaches the goal. If it does, you have a complete ladder, and it is the shortest. If not, you take that partial ladder, make copies and extend it to reach words that are one more hop away, and then enqueue those extended ladders onto the queue to be examined later. If you exhaust the queue of possibilities without having found a completed ladder, you can conclude that no ladder exists.

Let's make the algorithm a bit more concrete with some pseudocode:

```
create initial ladder (just start word) and enqueue it
  while queue is not empty
      dequeue first ladder from queue (this is shortest partial ladder)
      if top word of this ladder is the destination word
          return completed ladder
      else for each word in lexicon that differs by one char from top word
           and has not already been used in some other ladder
               create copy of partial ladder
               extend this ladder by pushing new word on top
               enqueue this ladder at end of queue
```

A few of these tasks deserve a bit more explanation. For example, you will need to find all the words that differ by one letter from a given word. A simple loop can change the first letter to each of the other letters in the alphabet and ask the lexicon if that transformation results in a valid word. Repeat that for each letter position in the given word and you will have discovered all the words that are one letter away.

Another issue that is a bit subtle is the restriction that you not re-use words that have been included in a previous ladder. This is an optimization that avoids exploring redundant paths (and will make your program run much, much faster!). For example, if you have previously tried the ladder cat → cot → cog and are now processing cat → cot → con, you would find the word cog one letter away from con, so looks like a potential candidate to extend this ladder. However, cog has already been reached in an earlier (and thus shorter) ladder, and there is no point in re-considering it in a longer ladder. The simplest way to enforce this is to keep track of the words that have been used in any ladder (using yet another lexicon!) and ignore those words when they come up again. This technique is also helpful to avoid getting trapped in an infinite loop building a circular ladder such as cat → cot → cog → bog → bat → cat.

Since additions to the ladder always occur at one end, a stack is a natural abstraction to represent a word ladder. The elements in the stack are simply strings. When you need to make a copy of a stack, remember that the assignment operator works as expected for all our container classes, and thus just assigning one stack to another will create a new stack with the same contents. 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.

Implementation Hints

A stack is ideal for storing a word ladder. The first entry pushed on the stack is the starting word and each subsequent word is pushed on top. You'll be building lots of extensions of a given stack, so be sure to copy the stack (e.g., using assignment) before pushing on the new element.

A queue is just what's needed to keep track of the partial ladders under consideration. The ladders are enqueued (and thus dequeued) in order of length so as to find the shortest option first.

For this program, we also supply Lexicon, a special-purpose class for storing a word list. It will be useful for the dictionary of English words as well as tracking those words that have already been examined to avoid redundantly processing them. Read the interface file `lexicon.h` in the starter project for details on how to create a new lexicon, add words to it, check if a word exists, and so forth. Behind the scenes, the lexicon employs a complex representation that is extremely efficient and compact to give you blindingly fast access to a list of over 100,000 words. However, you don't need to know any of that in order to use it! (We'll see how we can implement this later in the term.)

Word ladder task breakdown

This program requires only between one and two pages of code, but it still benefits from a step-by-step development plan to keep things moving along smoothly.

1. Try out the demo program. Play with the demo, `wordLadder.exe` (which is in the startup folder), just for fun and to see how it works from a users perspective.
2. Get familiar with the provided classes. Stacks and queues should be familiar, but it will be helpful look up the documentation in the STL (see the documentation web page), as you'll find that `push` is used

to enqueue something on a queue and `pop` is used to remove the front element. `Lexicon` will be new to you. Reading the comments in the header files before you start will help you get oriented in how to use these classes correctly. The files `stringOps` and `map` are helper classes used in defining `Lexicon`. You do not even need to read them. All of your code will go in `wordLadder.cpp`.

3. Carefully design the algorithm and the data structures. Be sure you understand the breadth-first algorithm and the various data types you will be using. Simulate it by hand to make sure you understand what should happen. Note that since the items in the queue are stacks, you have a nested template - be careful!

4. Dictionary handling. Set up a Lexicon object with the large dictionary read from our data file, `lexicon.dat`. Write a function that will iteratively construct strings that are one letter different from a given word and run them by the dictionary to determine which strings are words. Why not add some testing code that lets the user enter a word and prints a list of all words that are one letter different so you can verify this is working?

5. Implement breadth-first search. 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 or so words) to make it easier for you to test and trace your algorithm while you are in development. Do your stress tests using the large dictionary only after you know it works in the small test environment.

Like the last assignment, this assignment was adapted from one given by Julie Zelenski, which was itself based on a suggestion by Owen Astrachan.