CS14 Summer 2016
Programming Assignment 2

Word Ladder

Due Monday July 4 at 1:00 AM

Word Ladder

A word ladder is a word game. The player is given a start word and an end word and must come up with a "ladder" of words where each word in the ladder is no more than one letter different than the word before it and the word after it.

Here are a few word ladder examples:

- style stale state slate plate prate crate craze crazy
- brave braze blaze blame flame flams flats feats fears hears heart
- stack stark stars stirs sties stied steed stead steak
- black (*black is both the start and end word*)

For this assignment, you will write a program to find the smallest word ladder given a start word and an end word. To keep things simple, you will only be working with 5 letter words.

You can download the dictionary of 5 letter words we will be using to grade your program here: words5.dict.

The program will take as command line arguments the name of the dictionary file, the start word, and the end word and output to standard output the word ladder, e.g. ($a.out words5.dict brave heart). The output must start with the start word and end with the end word, and output no other characters other than whitespace. We will have a program that will take your output in this format as its input and verify you have a valid word ladder.

There are many algorithms out there to solve this problem including some that are much more efficient than the one we will be using. This assignment has been designed to give you practice implementing and using stacks and queues, therefore you must use the following algorithm.

Algorithm - Find Word Ladder

```
Create a stack of strings.
Push the start word on this stack.
Create a queue of stacks.
Enqueue this stack.

While the queue is not empty
   For each word in the dictionary
      If a word is one letter different (in any position) than the top string of the front stack
         If this word is the end word
           You are done! The front stack plus this word is your word ladder. Don't forget to output this word ladder.
         Make a copy of the front stack.
         Push the found word onto the copy.
         Enqueue the copy.
   Dequeue front stack.
```

Implementation

You will implement your own queue and stack. These classes must follow the ADT specifications given in lecture meaning the public interface must be the same and all public member function must run in O(1) time in the worst case. The only exceptions to the O(1) rule are the "Big 3" functions, copy constructor, overloaded assignment operator, and destructor. Since you will be copying stacks, you may need to define these functions yourself to avoid shallow copies. Here is a link to some notes on copy constructors if you don't have your CS12 textbook anymore: Copy Constructor info.

You must also implement a WordLadder class with the following public interface:

- **WordLadder(const string &listFile)** - Constructor that passes in the name of the dictionary file.
- **void outputLadder(const string &start, const string &end)** - passes in the start and end words and outputs to standard output the word ladder.

Storing the dictionary

The dictionary is in a file. You will want to read in the dictionary once and store it. Which of the "list" data structures we've learned so far should we choose to store this dictionary in: vector, list, queue, or stack? Once you begin to understand the algorithm, you will find that you can make this algorithm run much faster if you take out of the dictionary, or at least ignore, any word you've added to a potential word ladder.

You are required to use STL's list or slist (whichever is the most efficient possible given your algorithm) to store the dictionary. Do not make your own linked list as I want you to have practice using the STL list along with their iterator. Why is a linked list a good choice here?

Object Oriented Programming

There is a lot of freedom in the design of this assignment and since writing the actual queue and stack implementation is simple (I give a lot of it in class), a large portion of the points will be dedicated to your OOP skills.

Modular design

What modules should you create and what should each module be responsible for? You should have several private member functions in the WordLadder class. When you are done with this assignment, you should be able to directly take out your queue and stack classes and use them to implement a queue or stack for some other data structure by simply changing the type of data stored in the node class or changing the type of the array. A queue or stack class should know nothing about WHAT it is actually queuing or stacking, nor should a queue or stack class contain data specific statistics.

Notes on grading

Much of the grading of this assignment will be based on your OOP and general programming skills. Be sure to use good programming techniques where possible (use of classes, functions, good parameter passing techniques, error checking, encapsulation, object oriented programming). Make sure your program is user friendly.

You must at least use the compilation flags -W -Wall -Werror. Programs will be graded under linux on bell.cs.ucr.edu. Programs that do not compile will receive a zero. Be sure to download and re-test your submission from iLearn.

Suggestions to follow while coding

- Follow a modular programming style. Write one function for your program and completely test it before moving to the next function. This will isolate your debugging because most of the time the bug will be in the newly created module. However, if you forget a test case in an old module and do not completely test it, a new module can reveal old bugs. Watch out for this.

What to turn in

You should **ALWAYS** turn in what you have thus far on our program at least **6** hours before the due date (by 7 PM). Then continue to work on your program and turn in more current versions as you get them working.

You must turn in all .cc and .h files. In addition, you must follow any guidelines and include any information that is stated on the syllabus about at-home programming assignments. DO NOT forget to place a class_file_header.txt (filled in) at the top of each of your source files. Without this, your submission will not be graded. Since the student makes no claim that the submission represents academic work completed solely by the student. Points will be deducted from your score if this is added, by explicit instructor permission, after the due date.

DO NOT, I repeat, DO NOT turn in any files that are not required for the final compilation of your assignment. Do not turn in your temporary .cc files or your saved/old revisions of your .cc files or your debug folders. If you turn in files that aren't required for your assignment, it hinders the grading processes. You will be docked points if you turn in extraneous files.

A reminder about collaboration on home programming assignments

Please remember, at-home programming assignments are not lab assignments and you may not team-code with your lab partner or any other individual. Limited collaboration may be acceptable, but programs must represent YOUR OWN original work. Sharing code or team-coding are strictly not allowed. Copying code from ANY source (any book, current or past students, past solutions (inluding your own past solutions), web, etc) is strictly not allowed even with citation. Collaboration may consist of discussing the general approach to solving the problem, but should not involve communicating in code or even pseudo-code. Students may help others find bugs. Your code MUST be unique -- the odds of randomly producing similar code is very low. Computing, like surgery or driving a car or playing golf, can only be learned by doing it yourself!