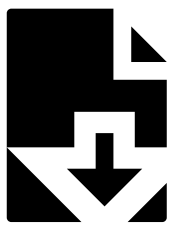
Assignment 2A: WordLadder

Assignment by Chris Piech, Marty Stepp, and Victoria Kirst. Originally based on a problem by Julie Zelenski and Jerry Cain.

DUE WEDNESDAY, JULY 11 AT 5PM PDT

This problem practices using stacks and queues.

Links:



Starter Code

We provide a ZIP archive with a starter project that you should download and open with Qt Creator. You will edit and turn in only the following file. The ZIP contains other files/libraries; do not modify these. Your code must work with the other files unmodified.

- **wordladder.cpp**, the C++ code for your solution

demo JAR

If you want to further verify the expected behavior of your program, you can download the following provided sample solution demo JAR and run it. If the behavior of our demo in any way conflicts with the information in this spec, you should favor the spec over the demo.

"How do I run the assignment solution demos?"

Our assignments offer a solution 'demo' that you can run to see the program's expected behavior. On many machines, all you have to do is download the .jar file, then double-click it to run it. But on some Macs, it won't work; your operating system will say that it doesn't know how to launch the file. If you have that issue, download the file, go to the Downloads folder in your Finder, right-click on the file, and click Open, and then press Confirm.

Some Mac users see a security error such as, "cs106b-hw2-wordladder-demo.jar can't be opened because it is from an unidentified developer." To fix this, go to System Preferences → Security & Privacy. You will see a section about downloaded apps. You should see a prompt asking if you would like to allow access to our solution JAR. Follow the steps and then you should be able to open the demo.

If all else fails, you could run the demo JAR from a terminal. Every operating system allows you to open a "terminal" or "console" window for typing raw system commands. Open your operating system's terminal or console window (Google if you need to learn how to do this), and then type:

```
cd DIRECTORY_WHERE_DEMO_JAR_IS_STOREDjava –jar JAR_FILE_NAME
```

For example, on a Mac machine, if your user name is jsmith12 and you have saved a demo JAR named hw2.jar in your Documents/106b directory, you would type:

```
cd /users/jsmith12/Documents/106bjava –jar hw2.jar
```

Or on a Windows machine, if your user name is jsmith12 and you have saved a demo JAR named hw2.jar in your Documents/106b directory, you would type:

```
cd C:\users\jsmith12\Documents\106bjava –jar hw2.jar
```

Program Description:

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 the word "code" to the word "data". Each changed letter is underlined as an illustration:

code → c<u>a</u>de → ca<u>t</u>e → <u>d</u>ate → dat<u>a</u>

There are many other word ladders that connect these two words, but this one is the shortest. There might be others of the same length, but none with fewer steps than this one.

In this problem, write a program that repeatedly prompts the user for two words and finds a minimum-length ladder between the words. You must use the Stanford **`Stack`** and **`Queue`** collections from Chapter 5, along with following a particular provided algorithm to find the shortest word ladder.

Example Logs of Execution:

Here is an example log of interaction from your program (with user input bolded). Your output must match this format exactly to earn full credit. (We are very picky.)

```
Welcome to CS 106B/X Word Ladder!
Please give me two English words, and I will convert the
first into the second by modifying one letter at a time.

Dictionary file name: dictionary.txt

Word 1 (or Enter to quit): code
Word 2 (or Enter to quit): data
A ladder from data back to code:
data date cate cade code

Word 1 (or Enter to quit):
Have a nice day.
```

Example log of execution

Notice that the word ladder prints out in reverse order, from the second word back to the first. If there are multiple valid word ladders of the same length between a given starting and ending word, your program would not need to generate exactly the ladder shown in this log, but you must generate one of minimum length.

Here are some additional expected output files to compare. Also, your program's graphical Console window has a File → Compare Output feature for checking your output. Use it!

- test #1 (code - data)
- test #2 (smalldict1.txt)
- test #3 (work - play; sleep - awake; etc.)
- test #4 (metal - azure; etc.)

Implementation Details:

Your code should **ignore case**; in other words, the user should be able to type uppercase, lowercase, mixed case, etc. words and the ladders should still be found and displayed in lowercase. You should also check for several kinds of **user input errors**, and not assume that the user will type valid input. Specifically, you should check that both words typed by the user are valid words found in the dictionary, that they are the same length, and that they are not the same word. If invalid input occurs, your program should print an error message and re-prompt the user. See the logs of execution on the course web site for examples of proper program output for such cases.

You will need to keep a **dictionary** of all English words. We provide a file **dictionary.txt** that contains these words, one per line. The file's contents look something like the following (abbreviated by ... in the middle in this spec):

```
aa
aah
aahed
...
zyzzyvas
zzz
zzzs
```

partial contents of dictionary.txt file

Your program should prompt the user to enter a dictionary file name and use that file as the source of the English words. (If the user types a file name that does not exist, reprompt them; see the second execution log further down this page.) Read the file a single time in your program, and choose an efficient collection to store and look up words. Note that you should not ever need to loop over the dictionary as part of solving this problem.

Finding a word ladder is a specific instance of a shortest-path problem of finding a path from a start position to a goal. Shortest-path problems come up in routing Internet packets, comparing gene mutations, and so on. The strategy we will use for finding a shortest path is called *breadth-first search* ("BFS"), a search process that expands out from a start position, considering all possibilities that are one step away, then two steps away, and so on, until a solution is found. BFS guarantees that the first solution you find will be as short as any other.

For word ladders, start by examining ladders that are one step away from the original word, where only one letter is changed. Then check all ladders that are two steps away, where two letters have been changed. Then three, four, etc. We implement the breadth-first algorithm using a **queue** to store partial ladders that represent possibilities to explore. Each partial ladder is a **stack**, which means that your overall collection must be a **queue of stacks**.

Here is a partial **pseudocode** description of the algorithm to solve the word-ladder problem:

```
how to find a word ladder from word w1 to w2:
    create a queue of stacks, initially containing only a single stack storing {w1}.
    repeat until queue is empty or w2 is found:
        dequeue a stack s.
        for each valid unused English word w
                that is a "neighbor" (differs by 1 letter)
                of the word on top of s:
            create a new stack s2 whose contents are the same as s,
                    but with w added on top,
            and add s2 to the queue.
```

The following list attempts to illustrate the progress of the overall algorithm when used to find a word ladder from "cat" to "dog". We begin with a queue containing only a single stack containing the starting word, cat. Then we enqueue all of its neighbors; suppose those neighbors in the dictionary are cot, cad,

and car. Then we enqueue the neighbors of cot, and then the neighbors of cad, and so on, each as the tops of larger and larger stacks. Eventually we find a path to the destination word, "dog", or we exhaust all possible stack paths and give up.

- { {**cat**} }
- { {cat,cot}, {cat,cad}, {cat,car} }
- { {cat,cad}, {cat,car}, {cat,cot,dot}, {cat,cot,cog}, {cat,cot,con} }
- { {cat,car}, {cat,cot,dot}, {cat,cot,cog}, {cat,cot,con}, {cat,cad,bad} }
- { {cat,cot,dot}, {cat,cot,cog}, {cat,cot,con}, {cat,cad,bad}, {cat,car,bar}, {cat,car,war} }
- { {cat,cot,cog}, {cat,cot,con}, {cat,cad,bad}, {cat,car,bar}, {cat,car,war}, {cat,cot,dot,**dog**} }

Finding neighbor words: Part of the above algorithm describes finding each "neighbor" of a given word. A neighbor of a given word *w* is a word of the same length as *w* that differs by exactly 1 letter from *w*. For example, "date" and "data" are neighbors. It is not appropriate to look for neighbors by looping over the entire dictionary every time; that would be too slow. Instead use two nested loops: one that goes through each character index in the word, and one that loops through the letters of the alphabet from **a**-**z**, replacing the character in that index position with each of the 26 letters in turn. For example, when examining neighbors of "date", you'd try:

- **aate**, **bate**, **cate**, ..., **zate** ← *all possible neighbors where only the 1st letter is changed*
- **date**, **dbte**, **dcte**, ..., **dzte** ← *all possible neighbors where only the 2nd letter is changed*
- ...
- **data**, **datb**, **datc**, ..., **datz** ← *all possible neighbors where only the 4th letter is changed*

The following diagram attempts to depict this process. Think of it as "spinning" each of the four letters of "code" through the alphabet A-Z, looking for valid English dictionary words.

```
   .   .   .   .
   a   m   b   c
   b   n   c   d
+---+---+---+---+
| c | o | d | e |
+---+---+---+---+
   d   p   e   f
   e   q   f   g
   .   .   .   .
```

Note that many of the possible words along the way (aate, dbte, datz, etc.) are not valid English words. Your algorithm has access to an English **dictionary**, and each time you generate a word using this looping process, you should look it up in the dictionary to make sure that it is actually a legal English word.

Another subtle issue is that you **do not reuse words** that have been included in a previous shorter ladder. For example, suppose that you have add the partial ladder cat → cot → cog to the queue. Later on, if your code is processing ladder cat → cot → con, one neighbor of con is cog, so you might want to

examine cat → cot → con → cog. But doing so is unnecessary because your goal is to find the shortest valid word ladder, and you have already found one from cat to cog without the unnecessary word con. So you should not enqueue the longer ladder in your algorithm. If you follow this rule properly, it will ensure that as soon as you've enqueued a ladder ending with a specific word, you've found a minimum-length path from the starting word to the end word in the ladder, so you will ever have to enqueue that end word again. To implement this strategy, keep track of the set of words that have already been used in any ladder, and ignore those words if 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**.

It is helpful to test your program on smaller dictionary files first to find bugs or issues related to your dictionary or word searching. We have provided files named **smalldict1.txt** through **smalldict3.txt** that you can try. Here is a sample log of execution using a smaller dictionary file:

```
Welcome to CS 106B/X Word Ladder!
Give me two English words, and I will change the first
into the second by changing one letter at a time.

Dictionary file name: notfound.txt
Unable to open that file.  Try again.
Dictionary file name: oops where am I.txt
Unable to open that file.  Try again.
Dictionary file name: smalldict1.txt

Word 1 (or Enter to quit): ghost
Word 2 (or Enter to quit): boo
The two words must be the same length.

Word 1 (or Enter to quit): marty
Word 2 (or Enter to quit): keith
The two words must be found in the dictionary.

Word 1 (or Enter to quit): kitty
Word 2 (or Enter to quit): kitty
The two words must be different.

Word 1 (or Enter to quit): dog
Word 2 (or Enter to quit): cat
A ladder from cat back to dog:
cat cot cog dog

Word 1 (or Enter to quit): cat
Word 2 (or Enter to quit): dog
A ladder from dog back to cat:
dog cog cot cat

Word 1 (or Enter to quit): byebye
Word 2 (or Enter to quit):
Have a nice day.
```

Example log of execution #2

Style Details:

As in other assignments, you should follow our **Style Guide** for information about expected coding style. You are also expected to follow all of the general style constraints emphasized in the Homework 1 specs, such as the ones about good problem decomposition, parameters, using proper C++ idioms, and commenting. The following are additional points of emphasis and style constraints specific to this problem:

Algorithms: You should follow the general algorithms as described in this document and should not substitute a very different algorithm. In particular, you should follow the breadth-first approach for finding word ladders as described in this spec. You should also not write a recursive algorithm for finding word ladders.

Collections: You are expected to make intelligent decisions about what kind of collections from the Stanford C++ library to use at each step of your algorithm, as well as using those collections properly. As much as possible, pass collections by reference, because passing them by value makes an expensive copy of the collection. Do not use pointers, arrays, or STL containers on this program.

Frequently Asked Questions (FAQ):

For each assignment problem, we receive various frequent student questions. The answers to some of those questions can be found below.

Q: Can I work with a partner on this assignment?
A: Yes! This is a pair assignment. You can also choose to work individually. If you work in a pair, submit only a single copy of your solution, not two.

Q: Is it okay if my partner does Part A, and I do Part B?
A: We do not recommend that. You're both responsible for knowing and understanding all of the material covered by this assignment. You are much less likely to understand the material if you didn't practice it. We highly recommend that you sit together with your partner to do the work for both parts. Another style that works well is to each write code for both parts separately, then meet together to compare them and produce a best solution that is a mix of the two.

Q: Can I use recursion on this assignment?
A: No. Do not use recursion (functions that call themselves).

Q: How do I construct a compound collection? It doesn't compile.
A: You need a space between nested **<>**.

```
Vector<Vector<int>> foo;    // no
Vector<Vector<int> > foo;   // yes
```

Q: Am I allowed to use a `Lexicon`?
A: Yes, though you are not required to do so.

Q: How can I loop over all of the letters in the alphabet?

A:

```
for (char ch = 'a'; ch <= 'z'; ch++) { ... }
```

Q: Why is my program veeeeeeery slow?
A: Usually this happens for one of two reasons: Either you have chosen the wrong data structure somewhere, or you have not properly followed the pseudocode algorithm we have you. A common bug is that students will loop over the entire dictionary to look for words. You should not need to do that. The purpose of the dictionary is to look up words, to test whether something is a valid word or not; not to loop over. If you want to loop over all possible "neighbor" words that are one letter away from a given word, do this by looping over the indexes of the string and over the letters a-z, as described in the pseudocode; not by looping over the dictionary.

Q: How many seconds is my program allowed to take? Mine finds the ladder in ___ seconds. Is that fast enough?
A: There is no specific time limit we test you on. But you should choose good collections and write your algorithm efficiently. If you do not, you may lose points. You could try running our demo solution to see how long that takes on your machine to get some idea of an appropriate runtime, though yours might not match this exactly.

Q: Why does my program spit out a VERY long (20-30+ word) ladder even for simple cases?
A: Maybe you're accidentally adding words to an existing collection rather than making a copy of it and adding to the copy. Run with a very small input, then print out your collections along the way and make sure everything is what you expect.

Q: If the word ladder is invalid for multiple reasons, which error message should I print?
A: It is up to you.

Q: I already know a lot of C/C++ from my previous programming experience. Can I use advanced features, such as pointers, on this assignment?
A: No; you should limit yourself to using the material that was taught in class so far.

Q: Can I add any other files to the program? Can I add some classes to the program?
A: No; you should limit yourself to the files and functions in the spec.

(Optional) Possible Extra Features:

Though your solution to this assignment must match all of the specifications mentioned previously, it is allowed and encouraged for you to add extra features to your program if you'd like to go beyond the basic assignment. Note that our motivation for allowing extra features is to encourage your creativity, not inflate your grade; so if any points are given for your extra features, they will be minimal. The purpose is to explore the assignment domain further, not to disrupt the class grading curve. Make sure to see the notes below about how to separate your extra code and how to turn it in properly.

Here are some example ideas for extra features that you could add to your program:

- **Allow word ladders between words of different length:** The typical solution forbids word ladders between words that are not the same length. But as an extra feature, you could make it so that it is considered legal to add or remove a single letter from your string at each hop along the way. This would make it possible to, for example, generate a word ladder from "car" to "cheat": car, cat, chat, cheat.
- **Allow word ladder end-points to be outside the dictionary:** Generally we want our word ladders to consist of words that are valid English words found in the dictionary. But it can be fun to allow only the start and end words to be non-dictionary words. For example, "Marty" is not an English word, but if you did this extra feature, you could produce a word ladder from "Marty" to "curls" as: marty, party, parts, carts, cards, curds, curls.
- **Other:** If you have your own creative idea for an extra feature, ask your SL and/or the instructor about it.

Indicating that you have done extra features: If you complete any extra features, then in the comment heading on the top of your program, please list all extra features that you worked on and where in the code they can be found (what functions, lines, etc. so that the grader can look at their code easily).

Submitting a program with extra features: Since we use automated testing for part of our grading process, it is important that you submit a program that conforms to the preceding spec, even if you want to do extra features. If your feature(s) cause your program to change the output that it produces in such a way that it no longer matches the expected sample output test cases provided, you should submit two versions of your program file: a first one with the standard file name without any extra features added (or with all necessary features disabled or commented out), and a second one whose file name has the suffix **-extra.cpp** with the extra features enabled. Please distinguish them in by explaining which is which in the comment header. Our turnin system saves every submission you make, so if you make multiple submissions we will be able to view all of them; your previously submitted files will not be lost or overwritten.
