**Program 1: Word Ladders**

**Word Ladders**

Your assignment is to build an application that discovers word ladders. The word ladder application accepts two words of the same length and then attempts to find the shortest word ladder connecting the two. Each word in the ladder should differ from the previous one by just one letter, and every single rung should be some word in the dictionary. If there are two or more shortest paths, then you only need to print one of them.

Here is a sample run:

We found a path from **slow** to **fast** and here it is:

slow

flow

flaw

flat

fiat

fist

fast

Input: use the test cases shown in TestLadder.cpp.

**Implementation Sketch**

The word ladder application should use a standard algorithm known as breadth-first search.

Use a doubly linked list as a queue to store partial ladders that represent possibilities to explore. The ladders are appended in order of length so that as you remove from the front, you will consider shorter ladders before longer ones. (Note, the doubly linked list implementation will be convenient for a later assignment.)

You first append a length-one ladder containing only the start word, then append all the length-two ladders that contain the start word plus a word that is one letter away from the start word, followed by the length-three ladders that further extend with a word that is two letters away, and so on. When a ladder is popped from the front, if the last word for that ladder is the target destination word, your job is done and you have found a complete ladder. It must be the shortest one because you appended ladders in order of increasing length. If you exhaust the queue of possibilities without having found a completed ladder, then no ladder exists. (Note that two words of differing length can't possibly have a ladder, so you can eliminate this case without searching.)

Let's make the algorithm a bit more concrete with some pseudo-code:

create initial ladder (just start word) and enqueue it on the partialSolution queue

while queue is not empty

extract front ladder from queue (this is shortest partial ladder)

if last word of this ladder is target destination word

**print completed ladder and return**

else for each unused word in dictionary that differs by one char from last word

{ create copy of partial ladder

extend this ladder by appending new word

append this ladder onto end of the partialSolution queue

}

}

**Example: finding a ladder from oops to pops**

1. Put the ladder consisting of the single word “oops” on the queue.
2. **Remove** (from the front of the queue) the first list in the queue. Call this list **ladder** and the last item in the **ladder** we'll call **current**.
3. Generate all possible one letter changes of **current** which:
   1. appear in the dictionary
   2. haven't been generated before

For example, if current is oops, we'd generate bops cops fops hops kops lops mops pops sops tops wops oohs oots. Now, for each word we generated, make a new ladder that is equal to the previous ladder plus the new generated word. Put each of these ladders at the end of the queue and mark each of these words as having now been generated. At this time, your queue will contain: [oops,bops][oops,cops][oops,fops][oops,hops][oops,kops][oops,lops]

[oops,mops][oops,pops][oops,sops][oops,tops] [oops,wops][oops,oohs] [oops,oots]

Since we will repeat this over and over, after one more iteration, we'll have:   
**[oops,cops][oops,fops][oops,hops][oops,kops][oops,lops] [oops,mops][oops,pops][oops,sops][oops,tops][oops,wops]**

**[oops,oohs][oops,oots][oops,bops,boas][oops,bops,bobs] [oops,bops,bods][oops,bops,bogs][oops,bops,boos] [oops,bops,boss][oops,bops,bots][oops,bops,bows]**

**[oops,bops,boys**

And after another iteration we'll have: 

[oops,fops][oops,hops][oops,kops][oops,lops][oops,mops][oops,pops][oops,sops][oops,tops]

[oops,wops][oops,oohs] [oops,oots][oops,bops,boas][oops,bops,bobs] [oops,bops,bods][oops,bops,bogs][oops,bops,boos][oops,bops,boss][oops,bops,bots] [oops,bops,bows][oops,bops,boys] [oops,cops,caps][oops,cops,ceps][oops,cops,cups] [oops,cops,cobs][oops,cops,cods] [oops,cops,cogs][oops,cops,cols][oops,cops,cons] [oops,cops,coos][oops,cops,coss] [oops,cops,cots][oops,cops,cows][oops,cops,coys] [oops,cops,cope][oops,cops,copy]

Notice that all the ladders of length 2 are before the ladders of length 3 in the queue. This ensures us that when we finally find our goal word, we know that it was the shortest possible ladder.

**Implementation Requirements:**

* You must implement a queue using doubly linked list (using prev/next pointers) for the list of partial solutions.
* You must implement a “findWord” method which uses a binary search to find the word in an dictionary array.
* You may use vectors to implement the ladders.
* You must parameterize similar methods so you have one version that has multiple behaviors rather than multiple versions of similar code.
* You must have enough outputting capabilities to do the following:
  + Determine how many words were read from dictionary.txt
  + Be able to display all the words which are one-away from a given word
  + Be able to display the contents of the partialSolution list.

This is critical for your own sanity. If you come to me for help, I will insist you have these debugging tools.

* You must use the style guidelines.
* During debugging, use a command like cin>> response so that the user can control the rate of execution. For example, a segment of the code might look something like:

//cout << toString("current ladder", thisLadder);

ofs << "This ladder size" << thisLadder.size() << endl;

char c;

//cin >> c;

string lastWord = thisLadder[thisLadder.size() - 1];

//cout << "lastWord " << lastWord << endl;

if (lastWord == targetWord){

cout << toString("Congrats", thisLadder);

ofs << toString("Congrats", thisLadder);

return true;

}

* Consider using ostringstream to allow every data structure to print itself easily.

string Node::toString(){

ostringstream oss;

oss << "[";

for (int i = 0; i < (int) ladder.size(); i++){

oss << ladder[i];

if (i != ladder.size()-1)

oss << ",";

}

oss << "]";

return oss.str();

}

**Input/Output**

**Input:** use the test cases shown in TestLadder.cpp.

**Output**

* If a ladder is possible, show the shortest ladder found.
* If a ladder is impossible, state why:
  + Words not found in dictionary
  + Words not the same length
  + No ladder was found
* You must display debug outputs to a file. p1SampleOut.txt shows what it might look like.

**Looking for an extra challenge?**

1. Preprocess the words when they are read in so that the only ``good'' words are tried as one-letter-off matches, not every word is tried. Instead, only try words that are in fact one letter off. This will involve comparing all the words when they are read in and somehow keeping track of which words are one letter apart. This will make the program run much more quickly since not all words will be considered as candidates.
2. If you picked a random word from the dictionary and then picked another random word of the same size from the dictionary, can you find a ladder from one word to the other? Run this random experiment lots of times (you decide what 'lots' means). What percentage of random words can form ladders? What percentage of random [3, 4, 5, 6]-letter words form ladders? Do the results surprise you?