Word Ladder

Overview:

stone
Atone
a**L**one
Clone
clon**S**
c**O**ons
co**N**ns
con**E**s
cone**Y**
Money

Word ladders were invented by Lewis Carroll in 1878, the author of *Alice in Wonderland*. A ladder is a sequence of words that starts at the starting word, ends at the ending word, In a word ladder puzzle you have to change one word into another by altering a single letter at each step. Each word in the ladder must be a valid English word, and must have the same length. For example, to turn stone into money, one possible ladder is given on the left.

Many ladder puzzles have more than one possible solutions. Your program must determine a shortest word ladder. Another path from stone to money is

```
stone   store   shore   chore   choke   choky   cooky   cooey   coney   money
```

Objectives

- Practice implementing and using Linked List based Queue and Stack data structures.
- Gain an understanding of algorithms used for efficient implementation.

Instructions

Your program will accept starting and ending words from the input file called "input.txt". Then, you read the word list in the file "dictionary.txt" and store it in a *HashSet*. Finally, you build a word ladder between the starting and ending words.

There are several ways to solve this problem. One simple method involves using stacks and queues. The algorithm (that you must implement) works as it follows:

> Get the starting word and search through the dictionary to find all words that are one letter different. Create stacks for each of these words, containing the starting word (*push*ed first) and the word that is one letter different. *Enqueue* each of these stacks into a queue. This will create a queue of stacks! Then *dequeue* the first item (which is a stack) from the queue, look at its top word and compare it with the ending word. If they are equal, you are done – this stack contains the ladder. Otherwise, you find all words one letter different from it. For each of these new words create a deep copy of the stack and *push* each word onto the stack. Then *enqueue* those stacks to the queue. And so on. You terminate the process when you reach the ending word or the queue is empty.

You have to keep the track of used words! **Otherwise an infinite loop occurs.**

Example

The starting word is *smart*. Find all words one letter different from *smart*, *push* them into different stacks and store the stacks in the queue. This table represents a queue of stacks.

```
-----------------------------------------
| scart | start | swart | smalt | smarm |
| smart | smart | smart | smart | smart |
-----------------------------------------
```

Now *dequeue* the front of the queue and find all words one letter different from the top word *scart*. This will spawn seven stacks:

```
---------------------------------------------------------
| scant | scatt | scare | scarf | scarp | scars | scary |
| scart | scart | scart | scart | scart | scart | scart |
| smart | smart | smart | smart | smart | smart | smart |
---------------------------------------------------------
```

which we *enqueue* to the queue. The queue size now is 11. Again *dequeue* the front and find all words one letter different from the top word *start*. This will spawn four more stacks:

```
---------------------------------
| sturt | stare | stark | stars |
| start | start | start | start |
| smart | smart | smart | smart |
---------------------------------
```

Add them to the queue. The queue size now is 14. Repeat the procedure until either you find the ending word or such a word ladder does not exist. Make sure you do not run into an infinite loop!

Queue

Implement a Queue data structure using your Linked List. Inserting at the end of the list & removing from the front makes your Linked List a queue.

Stack

Implement a Stack data structure using your Linked List. Adding & removing from the front makes your Linked List a stack.

Dictionary

The dictionary file contains exactly one word per line. However, the dictionary might contain empty lines. The number of lines in the dictionary is not specified.

Output

Your program must output to the console one word ladder from the start word to the end word. Every word in the ladder must be a word that appears in the dictionary. This includes the given start and end words – if they are not in the dictionary, you should print "There is no word ladder between ..." Remember that there may be more than one ladder between the start word and the end word. Your program may output any one of these ladders. The first output word must be the start word and the last output word must be the end word. If there is no way to make a ladder from the start word to the end word, your program must output "There is no word ladder between ..."

For testing purposes, use the provided files “input.txt” and “output.txt”.

Extra credit

Write a method

```
public ArrayList<Stack<String>> buildLadder(String start,
                                            String end,
                                            int ladderLength)
```

in the *WordLadder* class that returns ALL ladders of a given length `ladderLength`, not necessarily the shortest. For example. there are only two shortest ladders between *sail* and *ruin*:

```
sail, rail, rain, ruin
sail, sain, rain, ruin
```

However, there are 47 ladders of length 5. Here are some of them:

```
sail, mail, main, rain, ruin
sail, pail, pain, rain, ruin
sail, bail, rail, rain, ruin
sail, wail, rail, rain, ruin
sail, sain, rain, rein, ruin
sail, jail, rail, rain, ruin
sail, tail, rail, rain, ruin
```

If you feel brave enough to attempt this task, you shall implement so-called *exhaustive search*. In this search you generate all possible stacks of a given length and systematically examine them. You will notice that this method requires considerable computer memory resources. To increase the memory of the JVM (java virtual machine) you use the options `–Xmx` and `–Xms`. Here is an example:

```
java -Xmx500m MainWordLadderDriver
```

where `–Xmx` specifies the max heap size. Heap memory is an internal memory pool dynamically allocated by your application as it's needed. The max heap size for Windows runs from 1.3G to 1.6G depending upon the machine.

Exhaustive examination of all possibilities produces the entire solution space for the problem, which could be exceptionally larger than your computer resources. Therefore, you will get a full credit if your implementation finds all ladders for the following four-letter words:

```
sail ruin
slow fast
monk perl
```

Source:
2001: Owen Astrachan, Duke
2009: Victor S.Adamchik, CMU