CS 202
Assignment #14

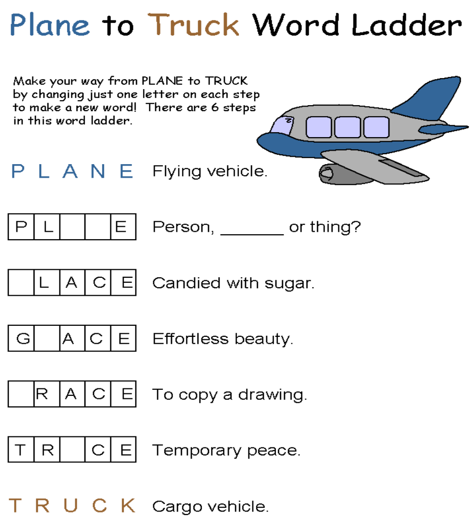
Purpose:	Learn to about templates, stacks, and queues using linked lists
Due:	Tuesday (5/07)
Points:	200

Assignment:

Design and implement a series of C++ classes to implement a stack and queue as follows:

- ***linkedStack*** to implement the stack using a linked list (*from previous assignment*)
- ***linkedQueue*** to implement the queue with a linked list

A main will be provided that can be used to test the classes.


Plane to Truck Word Ladder
Make your way from PLANE to TRUCK by changing just one letter on each step to make a new word! There are 6 steps in this word ladder.
P L A N E Flying vehicle.
P L _ _ E Person, _______ or thing?
_ L A C E Candied with sugar.
G _ A C E Effortless beauty.
_ R A C E To copy a drawing.
T R _ C E Temporary peace.
T R U C K Cargo vehicle.

A *word ladder*[1] is a word game. A word ladder puzzle begins with two words. To solve the puzzle, a chain of words is found to link the two words, in which each successive word in a chain or ladder differ by only one letter. Create class that will find and display a word ladder between two 5-letter words. All words must be five letters and only lower-case). The word ladder is ***not*** a template class.

The word ladder game requires a dictionary. The word dictionary file name should be read from the command line. The word file will contain 5-letter words and may include additional unrelated information (which must be ignored). For example, to start the ladder program:

```
        ed-vm% ./ladder -w words.dat
```

If no command line arguments are provided, the program should display a usage message (i.e., **`"Usage: ./ladder -w <wordsFile>"`**). If the specifier (**`-w`**) is not provided or incorrect, and error message should be displayed (i.e., **`"Error, invalid word file specification"`**). If the file does not exist or is not provided, the program should display an error message (i.e, **`"Error, unable to open input file: none.txt"`**).

Make File:

You will need to develop a make file. You should be able to type:

```
make
```

Which should create the executables. The makefile should build both executables; one for the testing (i.e., testQueue) and the other for the word ladder program (i.e., ladder).

Submission:

- Submit a compressed zip file of the program source files, header files, and makefile via the on-line submission by 23:50.

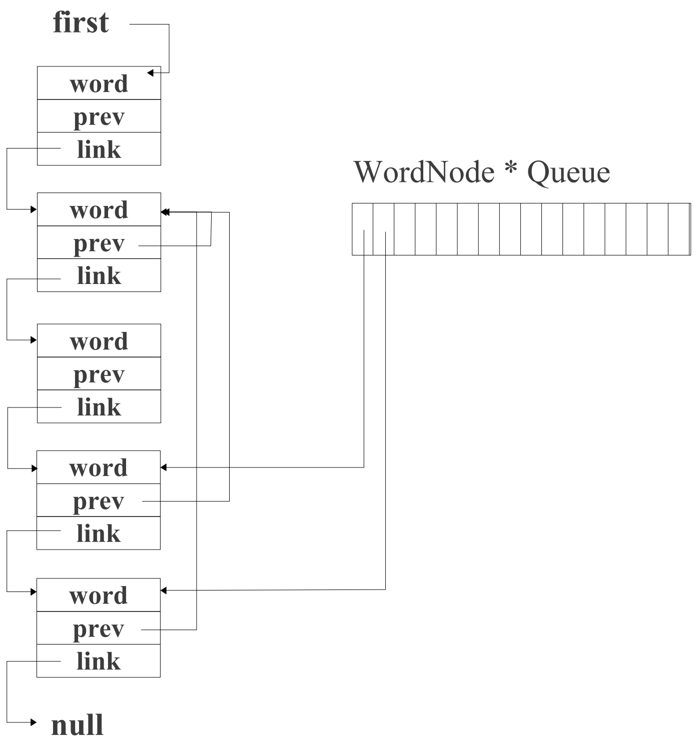
All necessary files must be included in the ZIP file. The grader will download, uncompress, and type **make** (so you must have a valid, working *makefile*).

1 For more information, refer to: http://en.wikipedia.org/wiki/Word_ladder

Word Ladder Algorithm

The following is an outline of the steps required to find a word ladder between two words.

- A structure is created for a linked list that includes the *word*, *prev*, and *link* fields. The *word* is a string, and the *prev* and *link* fields are pointers to a word node.
- A dictionary of 5-letter words is placed in a linked list using the *word* and *link* fields. The *prev* field is set to NULL.
- Create a queue of pointers to *wordNode*.
    - The linked list will contain all the words. The queue and stack will contain pointers to the word nodes (where the words are).
- Enqueue a pointer to the first word on the queue.
    - In order to distinguish this word as the first word, the *prev* pointer should be set pointing to itself.
- Repeat the following:
    - Dequeue an element (pointer to a *wordNode*).
    - Search the dictionary to find all the words that are one letter apart from the dequeued word.
        - If one of these words is the target word, done (ladder found).
        - Otherwise, enqueue a pointer to the word → if it has not been enqueued before.
            - When a word is enqueued, the *prev* field should be set to the previous word (that was dequeued).
            - Use the *prev* field determine if the word has already been enqueued.
        - If the last word is taken off the queue (i.e., queue is now empty), there is no path to the target word.
- The ladder can be printed using the *prev* fields. The *prev* pointer in the *wordNode* stores the links that allow the ladder to be recreated. The *prev* links point from the last word to the first word. In order to display the word ladder in the correct order, a stack should be used to reverse the order.



When the target word is found, the *prev* field is used track the word back the start word. Since this is in reverse, a stack is used to change the order.

Main Program

In addition, develop a client program (i.e., main) to use the word ladder class to display a word ladder. The main should perform the following key functions:

- The main program (ladder.cpp) should read the dictionary file name from the command line (as noted previously). The main should use the word ladder class to check the file and, if good, read the dictionary. Otherwise, the main should display an error message.
- The program should then prompt for a first or start word. The program should ensure the word is in the dictionary and, if not, re-prompt until a valid word is entered. If an empty string is entered, the program should be terminated.
- The program should then prompt for a second or end word. The program should ensure the word is in the dictionary and, if not, re-prompt until a valid word is entered.
- When both words are available, the program should attempt to find a word ladder. If a word ladder can be found, it should be displayed (in order). If not, an appropriate message should be displayed.

Refer to the example executions for examples of the error messages and formatting of the output,

Class Descriptions

- Word Ladder Class

  The word ladder class will implement the word ladder functionality and the words dictionary with a linked list and include the specified functions. Since this is ***not*** a template class, a header file and implementation file will be required. We will use the below word node structure.

```
struct wordNode {
      string info;
      wordNode *prev;
      wordNode *link;
};
```

wordLadder
-count: int
-wordNode *first
-wordNode *ladderEnd
+wordLadder()
+~wordLadder()
+readDictionary(const char *): bool
+findLadder(const string, const string): bool
+validWord(const string) const: bool
+printLadder() const: void
+resetLadder(): void
-isOneApart(const string, const string) const: bool
-insertWord(const string newItem): void
-searchPtr(const string& searchItem) const: wordNode *

Function Descriptions

- The *wordLadder()* constructor function will initialize class variables (NULL or 0 as appropriate). The *~wordLadder()* destructor function should free any allocated memory for both the ladder and the dictionary.

- The *readDictionary(const char *)* function should attempt to open and read the passed dictionary file name. If the function is unable to open the file, a *false* should be returned. The function should read all words in the provided dictionary file (using only the first five letters) and create a linked list of *wordNode's* with ***first*** as the pointer to the first word and ***count*** as the number of words in the dictionary.

- The *findLadder(const string, const string)* function should find a word ladder, if possible, between the first word (5-letter word) and the second string (5-letter word). The function should return true if a ladder can be found and false otherwise. The class pointer, *ladderEnd*, should be set to the final node (last word) in a linked list of words in the word ladder.

- The *resetLadder()* function should reset the *prev* field of every *wordNode* to NULL.

- The *validWord(const string)* function should attempt to find the passed word in the dictionary. If the word is found in the dictionary, the function should return true and false otherwise.

- The *printLadder()* function should print the word ladder, including the header. Since the word ladder will be in reverse, the print ladder function should use a stack to reverse the printing order to print the ladder from the first word.

- The private *isOneApart(const string, cons string)* function should determine if the passed words are one letter apart of not. If they are, return true and false otherwise.

- The private *insertWord(const string)* function can be used by the *readDictionary* function to insert words in a linked list.

- The private function *searchPtr(const string&)* should search the dictionary for the passed word and return the address of the node (in the linked list) if found and NULL otherwise.

- Linked List Stack Class
  The linked stack class will implement a stack with a linked list including the specified functions. We will use the following node structure definition.

```
template <class myType>
struct nodeType {
      myType info;
      nodeType<myType> *link;
};
```

linkedStack<myType>
-nodeType<myType> *stackTop
-count: int
+operator == (const linkedStack<myType>&): bool
+operator != (const linkedStack<myType>&): bool
+linkedStack()
+linkedStack(const linkedStack<myType>&)
+~linkedStack()
+deleteStack(): void
+isStackEmpty() const: bool
+push(const myType&): void
+peek() const: myType
+pop(): myType
+stackCount() const: int
+stackSum() const: myType
-rSum(nodeType<myType> *) const: myType
-copyStack(const linkedStack<myType>&): void

Function Descriptions

- The *linkedStack()* constructor should initialize the stack to an empty state (*stackTop* = NULL and *count*=0). The *linkedStack(const linkedStack<myType>&)* copy constructor should create a new, deep copy of the passed stack. The *~linkedStack()* destructor should delete the stack (including releasing all the allocated memory).

- The *isStackEmpty()* function should determine whether the stack is empty, returning *true* if the stack is empty and *false* if not.

- The *push(const myType&)* function will add the passed item to the top of the stack.

- The *pop()* function return and remove the top item from the stack. The *peek()* function will return the current top of the stack (without changing the stack). If the stack is empty, an error message should be displayed.

- The *stackCount()* function should return the count of items currently on the stack.

- The *stackSum()* function should return the sum of all items currently on the stack. The *stackSum()* function should use the recursive *rSum()* function.

- The overloaded *operator == (const linkedStack<myType>&)* should return *true* if the current and passed stacks are the same and *false* otherwise. This will require comparing the data values for each element in each stack. The overloaded *operator != (const linkedStack<myType>&)* should return *false* if the current and passed stacks are the same and *true* otherwise.

- <u>Linked List Queue Class</u>
  The linked queue class will implement a queue with a linked list including the specified functions. We will use the following node structure definition.

```
template <class Type>
struct queueNodeType
{
    Type info;
    queueNodeType<Type> *link;
};
```

linkedQueue<Type>
-queueNode<Type> *queueFront
-queueNode<Type> *queueRear
-count: int
+operator = (const linkedQueue<Type>&): linkedQueue<Type>&
+linkedQueue()
+linkedQueue(const linkedQueue<Type>& otherQueue)
+~linkedQueue()
+isQueueEmpty() const: bool
+initializeQueue(): void
+enqueue(const Type& newItem): void
+front() const: Type
+back() const: Type
+dequeue(): Type
+queueCount(): int
+printQueue(): void
-copyQueue(const linkedQueue<Type>& otherQueue): void

<u>Function Descriptions - Queue</u>

- The *initializeQueue()* function will create and initialize a new, empty queue. The *isEmptyQueue()* function should determine whether the queue is empty, returning *true* if the queue is empty and *false* if not.

- The *enqueue(const Type& newItem)* function will add the passed item to the back of the queue. The *front()* function will return the current front of the queue and the *back()* function return the current back of the queue .

- The *dequeue()* function will remove and return the front item from the queue. If the queue is empty, nothing should happen.

- The *linkedQueueType()* constructor should initialize the queue to an empty state (*queueFront* = NULL and *queueRear* = NULL). The *linkedQueueType(const linkedQueueType<Type>&)* copy constructor should create a new, deep copy of the passed queue. The *~linkedQueueType()* destructor should delete the queue (including releasing all the allocated memory).

- The overloaded *operator = (const linkedQueueType<Type>&)* should create a new, deep copy of the queue.
- The private *copyQueue(const linkedQueueType<Type>&)* function is used by the copy constructor and the overloaded operator = *(const linkedQueueType<Type>&)* .
- The *printQueue()* function should print the current elements of queue. The *queueCount()* function should return the current count of elements in the queue.

Refer to the example executions for output formatting. Make sure your program includes the appropriate documentation. See Program Evaluation Criteria for CS 202 for additional information. ***Note, points will be deducted for especially poor style or inefficient coding.***

Example Execution:

Below is an example program execution for the test queue main program.

```
ed-vm%
ed-vm% ./testQueue
--------------------------------------------------
CS 202 - Assignment #14
Basic Testing for Stacks and Queues

--------------------------------------------------
Queue Operations - Integers:

Queue 0:  1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Queue 1 (odds): 1 3 5 7 9 11 13 15 17 19

Queue 2 (evens): 2 4 6 8 10 12 14 16 18 20

Queue 3 (copy of odds): 1 3 5 7 9 11 13 15 17 19
Queue 3 (now empty):

Queue 4 (copy of odds): 1 3 5 7 9 11 13 15 17 19
Queue 4 (count): 10
Queue 4 (first item): 1
Queue 4 (last item): 19

--------------------------------------------------
Queue Operations - Floats:

Queue 5 (floats, original): 1.5 2.5 3.5 4.5 5.5 6.5 7.5

Queue 6 (floats, original): 1.5 2.5 3.5 4.5 5.5 6.5 7.5

Queue 7 (floats, original): 1.5 2.5 3.5 4.5 5.5 6.5 7.5

Queue 5 (floats, modified): 4.5 5.5 6.5 7.5 1.5 2.5 3.5
Queue 5 (count): 7
Queue 5 (first item): 4.5
Queue 5 (last item): 3.5

Queue 6 (floats, modified): 1.5 2.5 3.5 4.5 5.5 6.5 7.5
Queue 6 (count): 7
Queue 6 (first item): 1.5
Queue 6 (last item): 7.5
--------------------------------------------------
Game Over, thank you for playing.
ed-vm%
```

Below is an example program execution for the test queue main program.

```
ed-vm%
ed-vm% ./ladder
Usage: ./ladder -w <wordFile>
ed-vm%
ed-vm% ./ladder -w
Error, must provide word file name.
ed-vm%
ed-vm% ./ladder -x words.dat
Error, invalid word file specifier.
ed-vm%
ed-vm%
./ladder -w words
Error, unable to read word file: words
ed-vm%
ed-vm%
ed-vm% ./ladder -w words.dat
CS 202 Word Ladder Program.

--------------------
Enter First Word: money
Enter Second Word: stone

Word Ladder:
money
coney
cones
cores
corns
coins
chins
chine
shine
shone
stone
--------------------
Enter First Word: angel
Enter Second Word: devil
There is no path from angel to devil
--------------------
Enter First Word:
ed-vm%
ed-vm%
ed-vm%  ./ladder -w knuth.dat
CS 202 Word Ladder Program.

--------------------
Enter First Word: cthc
Error, first word is not in dictionary.
Enter First Word: catcher
Error, first word is not in dictionary.
Enter First Word: cha
Error, first word is not in dictionary.
Enter First Word: chase
Enter Second Word: cacth
Error, second word is not in dictionary.
Enter Second Word: catch

Word Ladder:
chase
cease
pease
peace
peach
perch
parch
```

```
patch
catch
--------------------
Enter First Word: would
Enter Second Word: could

Word Ladder:
would
could
--------------------
Enter First Word: tears
Enter Second Word: fears

Word Ladder:
tears
fears
--------------------
Enter First Word: showss
Error, first word is not in dictionary.
Enter First Word: shows
Enter Second Word: reedad
Error, second word is not in dictionary.
Enter Second Word: ready

Word Ladder:
shows
shews
sheds
seeds
reeds
reads
ready
--------------------
Enter First Word:
ed-vm%
ed-vm%  ./ladder -w words.dat
CS 202 Word Ladder Program.

--------------------
Enter First Word: plane
Enter Second Word: truck

Word Ladder:
plane
place
glace
grace
trace
track
truck
--------------------
Enter First Word: train
Enter Second Word: plane

Word Ladder:
train
brain
braid
brand
bland
blank
plank
plane
--------------------
Enter First Word:
ed-vm%
```