CIS 291 – Data Structures in C++ - Spring 2005
Homework #7
HW #7 due: THURSDAY, March 24th, BEGINNING of lecture

Purpose:
To read about, think about, and practice with linked implementations of ADT's.

How this will be turned in:
Use **~st10/291submit**, called from the directory on cs-server where the files you wish to submit are stored.

Homework #7
On the public course web page, you will find the **node.cpp** and **node.h** files mentioned in lecture.

You will recall that we also discussed using **node** in a linked implementation of the **stack** ADT, which is very similar to the 2-3-05 stack pseudo-UML (see Week 3 Lab Exercise and Lecture #2 links) **except** that it is no longer fixed-capacity. That is, methods **get_capacity** and **is_full** have been removed, as have been all references to a fixed capacity or references to these methods.

An implementation that should reflect what we discussed conceptually in lecture is also available on the public course web page: **stack.h**, **stack.cpp**, and **test_stack.cpp**. (Make sure you have the versions with last-modified date of **3-10-05**!)

(Note that, because dynamic allocation is involved, this includes an explicitly defined **copy constructor** and **destructor**. If you read these implementations carefully, you should be able to get an idea of what they do.

* sometimes the copy constructor is called implicitly, and sometimes explicitly --- I hope to discuss this further later on. But, in the meantime, it is always written in the "pattern" shown here, with exactly one argument as shown. Its usual purpose is to (in the words of Savitch and Main, p. 56) "to initialize a new object as an exact copy of an existing object".

  You can see an example of syntax that ends up calling the copy constructor within **test_stack.cpp**.

* you don't generally call the destructor directly --- but it is called automatically in a number of cases, such as when you end the function in which an object was declared.

  It, as the name implies, destroys the object --- frees all of its memory properly when the object in question is no longer needed.

)

0. NOTE!!! The INTENTION here is that you CAREFULLY READ and STUDY the files provided --- do you understand node's implementation? Do you understand stack's implementation?

1. (warm up) Keep in mind, we have a number of files involved for **test_stack.cpp** to link and load successfully! Do you know what all of them are? To make sure:

* copy all of the files that you need to run **test_stack.cpp** into a new directory on cs-server named **291hw07**.

* Add a **cout** statement to **test_stack.cpp** that prints out your name, to personalize your output.

* Then, compile it, run it, make sure that all it well, and then redirect its output into a file **291hw07_1**:

  test_stack > 291hw07_1

Submit your **291hw07_1** file using **~st10/291submit**; this will show (hopefully) that you've remembered everything needed to run a program using a linked stack.

(It also shows why we are almost at the point where **make files** would be worth the effort to make and set up...)

2. Now, implement a **linked** implementation of our **queue** ADT; make sure that your implementation MEETS the specifications of the **2-3-05** queue pseudo-UML (posted with HW #3), except with **all** references to fixed-capacity --- including methods **get_capacity** and **is_full** --- REMOVED.

It must also use the **node** class appropriately.

(ASK ME if you are not clear what is being required above!!!!!)

Create **queue.h**, **queue.cpp**, and **test_queue.cpp** that is a queue-appropriate testing main function patterned (carefully!) after the **test_stack.cpp** posted along with this assignment. (You should NOT be "removing" any tests, although you will of course modify them to be appropriate for queues; you may ADD additional tests if you wish.)

When you have tested these, debugged them, and are satisfied with them, run **test_queue** and redirect its output into a file **291hw07_2**:

test_queue > 291hw07_2.

A few **additional requirements/comments**:

* careful reading and thought is the intent here; comments that still read like stack comments may result in point deductions.

* what does a linked list that is going to serve as a queue need? Hint: there is more than one way to do this, but some alternatives are a lot harder to work with than others...

* As you are writing your **copy constructor**, look carefully at the stack version, of course. But also consider: *how should you set rear for the new copy?* This is probably the trickiest part of this method (and there are easier and harder approaches for attempting it...)

* **POINTER/LINKED LIST RULES TO LIVE BY:**
  * If there is ANY chance --- ANY chance at all! --- that a pointer might be NULL, then you need to MAKE SURE that it is NOT null **before** you try to follow it.

    (failure to do so can lead to lovely run-time errors --- segmentation faults, bus errors, fun stuff.)

  * For any operation on a linked list, double check your implementation:
    * does it work in the **empty** list case (if that is applicable)?
    * does it work in a single-element case?
    * does it work in a multi-element case?

  * When it doubt, hand-walk through your code, drawing the lovely box-and-arrow pictures...

When you are done, submit the following files using **~st10/291submit**:

291hw07_1
queue.h, **queue.cpp**, **test_queue.cpp**, **291hw07_2**