CS 251: Intermediate Software Design

Program Assignment 3
Part 1 (AQueue) Due Friday, Feb 23rd, 2007
Part 2 (LQueue) Due Friday, Mar 2nd, 2007

A queue is an *Abstract Data Type* (ADT) that implements a priority queue with "first-in, first-out" (FIFO) behavior. Common operations on a queue include *enqueue()*, *dequeue()*, *front()*, *is_empty()*, and *is_full()*. This part of your programming assignment focuses upon building and using an *array* and *linked list* implementations of ADT `Queue`:

1. *Array queue* (`AQueue`) – the first queue will use your `Array` class. The trick is to implement a "circular" queue with a "dummy node" so that you never need to copy data unnecessarily and so that you will remove special case checks in the code.
2. *Linked list queue* `LQueue` – the second queue will use a circular linked list, which is "unbounded" (at least in principle...) and uses dynamic memory and a "dummy node" for the circular queue. This will be more challenging to write correctly than `AQueue` since it requires you understand how C++ linked lists work.

You'll need to implement STL-like iterators for both types of queues.

Part 1 – AQueue

The first implementation you will write is a queue that reuses the class `Array` you implemented for your first assignment. The `enqueue()`, `dequeue()`, and `front()` methods explicitly check whether the queue is empty or full and throw exceptions if their preconditions aren't held. It therefore isn't necessary for clients to call `is_empty()` or `is_full()` before adding, removing, or viewing a queue element.

Graduates taking the class need to implement a bidirectional iterator for `AQueue`. Undergraduates taking the class just need to implement a forward iterator, though you can implement a bidirectional iterator if you'd like. Both graduates and undergraduates should use standard C++ library generic algorithms to implement their `AQueue` classes.

Part 2 – LQueue

A limitation of the `AQueue` implementation of the ADT Queue is that queues cannot grow beyond their initial size. Your second implementation will therefore write an queue using a circular linked list that allocates memory for new queue nodes dynamically. Note that this change only affects the queue representation, but does not affect the queue interface.

To simplify the `enqueue()`, `dequeue()`, and `LQueue_Iterator` logic, please add a dummy node to your `LQueue` implementation. This will remove all special-case checks in your code.

Graduate students will need to implement the following enhancement to `LQueue`:

- *A free list* – Implement a free list by overloading class-specific `operator new` and `operator delete` in `LQueue_Node`. This free list will cache previously allocated nodes in a `static` data member in class `LQueue_Node`. Note that your `LQueue::enqueue()` and `LQueue::dequeue()` methods must *not* know anything about the free list!

Getting Started

You can get the "shells" for the program from `www.cs.wustl.edu/~schmidt/cs251/assignment3`. The `Makefile`, `AQueue.cpp`, `AQueue-test.h`, `LQueue.cpp`, and `LQueue-test.h` files are written for you. All you need to do is edit the `AQueue.cpp` and `LQueue.cpp` files to add the methods that implement the `AQueue` and `LQueue` ADTs. Note that you'll need to reuse the files from your `Array` implementation for `AQueue`.

If you are an undergraduate student please use the shells that are in the `ugrad` directory at the URL above. If you are a graduate student please use the shells that are in the `grad` directory at the URL above.