COSC 2P95 – Lab Exercise 5 – Section 01 – Data Structures

Lab Exercises are uploaded via the Sakai page.
Since you need to submit both a sample execution and your source files, package it all up into a .zip to submit.

This week, we'll be creating a slightly more advanced data structure than what we wrote for the lab.
In the lab, you wrote a simple queue. This time, you'll create a *priority queue*.

The principle of a priority queue is pretty simple:

- All entries now have both record data and a priority
- Depending on the application, a higher priority may be signified by a lower priority value, or a higher priority value (for this task, assume that lower values take precedence)
- When two entries have the same priority values, they behave as a standard FIFO queue
    - i.e. the element that's been waiting the longest gets dibs
- When two entries have different priority values, the more important entry goes first

For example, consider a print spooler:

- Normally, you'd expect pages to be printed on a first-come, first-serve basis
    - But what happens when a very high-priority print job comes through?
    - Basically, that higher-priority job 'jumps the queue'

Priority queues are useful for all sorts of things, including graph algorithms, job (and process) scheduling, and event simulation.

- For example, consider event simulation:
    - Each entry/event has a corresponding time also within the node
    - When trying to follow along with the events that take place during an execution, you need only take the next item with the lowest timestamp
        - This is one of those cases where it very much makes sense for a lower priority value to reflect a higher effective priority

There are different ways to implement a priority queue. The simplest is to write a regular queue, except instead of always appending new entries to the tail end, bump them up whenever they would otherwise follow an element with a higher priority value. Include *some* mechanism for knowing when the PQ is empty.

Requirements:

- The PQ holds PRecords, as defined in the lab
- You may use an array-based or linked implementation
- Though real PQs typically also have some form of a 'peek', and the ability to raise/lower priorities (to resequence the elements), all you need implement for this task are `enqueue` (i.e. *add*), and `dequeue` (i.e. *remove*)
- Write a simple program that repeats until you choose to quit
    - You may choose to add an entry, specifying the time/priority, and the string to add to the entry
    - You may choose to dequeue and display the 'next' element (i.e. the oldest member of the PQ, amongst those with the lowest priority value)

Requirements for Submission:

Besides your source files, simply include a sample execution demonstrating your program.
Remember that things like memory leaks or forgetting to include a header file would be reflected poorly by the evaluation.