Re-submit Assignment

PriorityQueue Template, Lab Assignment 11

Due Nov 15 by 11:59pm **Points** 100 **Submitting** a file upload **File Types** cpp and h

There are 3 parts to this one, with 3 CPPs and one H. But the CPPs are all copy/paste/markups of code already written by you or provided in a module, each with not very much change. The H file is the main programming focus, because that's all new.

Part 1, PriorityQueue.h Template

Implement a heap-based priority queue named **PriorityQueue** in a template named **PriorityQueue.h**. Apply the public interface shown in this module. Fully test it with a test driver CPP, but do not submit the CPP -- just the H.

A good test-driver test is a copy-pop output listing of the contents of your priority queue (without assertions). Do that *after* writing your copy constructor. Also use your own struct-based objects as **typename V** in some of your testing.

Another good test is to fill a priority queue with values of your choosing, and then copy-pop its contents into an array. That array should be ordered hi-to-lo. Check its contents with expecteds, actuals, and assertions.

Normally we would not use **cin** or **rand** for input to a test driver. But this is a good place to make an exception. You could test your priority queue by pushing a large number of random integers onto it and (a) asserting the size, and (b) asserting the contents through a copy-pop loop. Just don't output the expected and actual values!

HINT: Be careful in the pop algorithm -- do **not** *compare the right child to anything if there is no right child! Just because the left child passes the <=siz test does not mean that the right child does...*

WARNING #1: For priority queues of C++ data types where you have no control over how less-than comparisons are made, do **not** *attempt lo-to-hi ordering. You'll need your own struct-defined objects with reverse logic in their operator< for that to work.*

WARNING #2: Use only ***less-than*** *comparisons of* ***typename V*** *values in the H, because that's all you'll be defining when you use your own struct-defined objects as values.*

Submit just **PriorityQueue.h**.

Part 2, Timing Test (Fast)

Run a timing test for your **PriorityQueue** to confirm that push and pop are both O(log n). Refer back to the module on big oh, and the timing test code provided, and adapt it. Write separate CPPs, named **PriorityQueue.push.cpp** and **PriorityQueue.pop.cpp**. Make sure that to include confirmations that (1) the priority queue to be tested has n values in it, and (2) the same priority queue is arranged in hi-to-lo order after the timing test ends. Use the Q&A section of this module to discuss how to do these confirmations.

For each CPP and for each timed cycle, first build a priority queue of size n using a for-loop and push. Confirm its size. *Then*:

1. Start the timer.
2. Do the "for (int rep = 0; rep < reps; rep++)" loop, pushing (or popping) more values.
3. Stop the timer.

Confirm that the priority queue is ordered hi-to-lo by popping its values and using assertion in a for-loop.

NOTE: You are supposed to be timing push and pop onto a queue that already has n values in it. You are NOT supposed to be timing the pushing and popping of n values for a queue that is initially empty. NOR are you supposed to be timing the pushing and popping of n values for a queue that already has n values in it.

WARNING: Do NOT let array-doubling happen in the timed part of the test. The first thing that happens with the first push in the times reps loop should NOT be an array resizing.

Submit **PriorityQueue.push.cpp** and **PriorityQueue.pop.cpp**.

Part 3, BetterSimulation.cpp

In Lab Assignment 7, Part 2, you wrote **Simulation.cpp** -- a server simulation with randomized arrivals and a wait queue. Maybe you did not notice at the time, but it used a *very* inefficient way to check if any servers were done servicing their customer. It used this O(n) process (where n is the number of servers):

```
// handle all services scheduled to complete at this clock time
for each server
  if the server is busy
    if the service end time of the customer that it's now serving equals the clock time
      set this server to idle
```

Of course, n is very small, so it's not *really* a timing issue. But it *is* inefficient coding. That was fine at the time, because we didn't even know about big oh yet. But now we do, and now we have priority queues.

The better way to do this is to keep a separate list of scheduled end-of-service times, with the server number and the end-of-service time, like this sample, arranged lo-to-hi:

```
server | time for end-of-service
------ + ----------------------
   2   |         10
   1   |         12
   4   |         17
   5   |         17
```

(TIP: During testing, output the above table using copy-pop, to make sure it's got each server appearing no more then once and only if it's busy, that all end-of-service times are >= the clock time and are in order lo-to-hi..)

Then instead of asking *each* server *each* minute "are you busy?" and "are you done yet?", we could just look at this above list and check the top value. Until the top value's end-of-service time equals the clock time, there's nothing to do -- no more bothering each server every minute.

The list is called an "**event queue**". It's a lo-to-hi priority queue. Its contents are **service events**, which you should model in a struct with server number and end-of-service time. Push a new **service event** onto the **event queue** every time you calculate a new end-of-service time for the customer whose service is starting. Then you can replace the "handle all services..." code block with this:

```
// handle all services scheduled to complete at this clock time
while event queue is not empty and its top's end-of-service time equals the clock time
  set the top service event's server to idle
  pop the just-used service event off of the event queue
```

The **service end time** will no longer be needed in the customer object's struct, so remove that from the struct. Use only the new priority queue to manage end-of-service events. So in the "**// for idle servers, move customer from wait queue and begin service**" code block, reinterpret "**set service end time to current clock time PLUS "random service interval"**" so that it creates a new end-of-service event and pushes it onto the priority queue, instead of setting that no-longer-present attribute of the customer object.

WARNING: Do NOT write a separate H file for a lo-to-hi priority queue. Reverse the logic in the service event operator< to force lo-to-hi.

OUTPUT. The output should be ***exactly the same*** as Lab Assignment 7, Part 2's, except for the following ***addition***:

Since you know the timing of the next end-of-service event (if there is one), put somewhere in the minute's output something like "Next end-of-service event *at* XX minutes" or "Next end-of-service event *in* YY minutes" or "No scheduled end-of-service events at this time". Or use copy-pop to output the whole (nicely formatted) queue, so you can see what's coming up for the servers. Make sure you use the priority queue for this output!

Submit this as **BetterSimulation.cpp.** And if you use any of your own H files for DynamicArray or Queue or whatever, submit those H files too.

Lab Assignment Rubric

Criteria	Ratings							Pts
Fully accurate results, following all specifications **view longer description**	Works the first time. 70.0 pts	Works on the 2nd try 65.0 pts	Works on the 3rd try 60.0 pts	Works after 4 or more tries. 50.0 pts	Doesn't work after 2 weeks. Partial credit. 20.0 pts	Not submitted within two weeks of the due date. 0.0 pts	Work is not original -- appears to be a marked-up copy of the work of another or previous student. 0.0 pts	70.0 pts
Submits all work on time, fully complete if not fully correct. **view longer description**	Submitted on time 20.0 pts	Submitted on time, but one or more files are missing or not correctly named. 16.0 pts	Submitted on time, but with missing identification in one or more submitted CPP or H files. 15.0 pts	Submitted on time but not fully complete. 10.0 pts	Late or wholly incomplete! 0.0 pts			20.0 pts
Well-organized and professional quality code. **view longer description**	Fully meets expectations 10.0 pts	Mostly meets expectations, just needs to be a bit more careful. 8.0 pts	Many areas are well done, but there are a lot of areas that need work. 6.0 pts	Getting there, but needs to be a lot better. 3.0 pts	Needs a lot of work. See the instructor for guidance. 0.0 pts			10.0 pts
								Total Points: 100.0