CSCI-1200 Data Structures — Spring 2019
Homework 10 — Job Prioritization

In this assignment we will write a couple of classes to help with job management. These jobs could represent anything, for example in disaster recovery a central dispatch might receive many work orders which come in without being sorted by when they were requested or what their priority is.

Description of Classes

Every job is represented by an instance of the *Job* class. Every Job has a timestamp (the number of seconds since midnight January 1st, 1970 UTC - see `http://en.wikipedia.org/wiki/Unix_time` for more background), an urgency, and a unique ID. The timestamps are numbers that use `std::time_t`. To make it easier to understand these times, we have provided code in an output operator that can print Jobs and includes a more readable date and time. The ID is automatically handled by the Job constructor. The Job copy constructor will alter IDs that are copied to help you track your efficiency. *Note: The gmtime() function used by the Job output operator depends on your computer. You might see "UTC" or "GMT" at the end of your timestamps on your computer, but they will have the correct suffix of "GMT" when run on Submitty.*

Another hint: The includes can get pretty messy in this assignment. The recommended way to handle using the Job class is to "Job.h" in your .cpp files but **do not** include them in your other .h files. C++ linking gets very complicated, but hopefully this piece of advice will prevent issues for most students. If you are curious about more advanced problems you may want to consult `http://www.cs.rpi.edu/academics/courses/spring19/csci1200/hw/10_job_prioritization/more_about_includes.pdf` .

A higher urgency indicates a more important job. In the disaster recovery example, a very high urgency might indicate a task like repairing power to a hospital. A common task is thus to ask "what is the highest urgency job that hasn't been dispatched?" This will be handled by a priority queue class, *UrgentQueue*.

However, if only high urgency jobs are serviced, it is possible that some low urgency jobs will sit unaddressed for very long periods of time. To counter this, a dispatcher may want to sometimes ask "what is the oldest job that hasn't been dispatched?" This will be handled by a priority queue class, *TimeQueue*.

You must implement both queue classes "elegantly" (they should be efficient and use good style). Whenever a Job is added to one queue, it should also be added to the other queue but you should not create a new Job. Instead you should use Job pointers in most of your code. Whenever a job is removed from one queue, it should be removed from the other queue. Removing a job represents the job being dispatched. *Hint: There are several orderings that could create a valid heap. However if your output does not match Submitty's it usually means that one or more of your operations was inefficient.*

You need to support any operations that we call in the two queue classes. `remove_from_index(position)` takes in the position in a queue's representation and erases that value. Without this, we would have to do an expensive find, but as mentioned in Lecture 22.9 we can use "hooks", which in this case are just the locations in the heap representation. By storing these in advance, we eliminate the need to search for a specific value's location within our queue.

Arguments and Commands

`main.cpp` takes 2 arguments, the name of an input file and the name of an output file. Command parsing is already implemented.

Examples of the input are provided and are named `test1.txt` etc. The expected output for these examples is provided in `out_test1.txt` etc. We will not ask you to "print-next-" or "remove-next-" commands on an empty queue.

A description of the commands is on the next page.

- *add-job urgency timestamp*: *urgency* and *timestamp* are numbers. This command adds a job to the priority queues.
- *print-by-urgent*: This prints the contents of the UrgentQueue in a left-to-right BFS.
- *print-by-time*: This prints the contents of the TimeQueue in a left-to-right BFS.
- *print-next-urgent*: This prints the information about the most urgent job.
- *print-next-age*: This prints the information about the oldest job.
- *remove-next-urgent*: This removes the most urgent job from the queues.
- *remove-next-age*: This removes the oldest job from the queues.

Provided Files and Submission

We will use our own version of `main.cpp`, `Job.cpp`, and `Job.h` even if you submit your own. You should submit five files: `README.txt`, `UrgentQueue.h`, `UrgentQueue.cpp`, `TimeQueue.h`, and `TimeQueue.cpp`

Use good coding style and detailed comments when you design and implement your program. Please use the provided template `README.txt` file for any notes you want the grader to read, including work for extra credit. You must do this assignment on your own, as described in the "Collaboration Policy & Academic Integrity". If you did discuss the problem or error messages, etc. with anyone, please list their names in your `README.txt` file.

If you have at least 6 points on test cases 3 and 7 by Wednesday 11:59:59PM, you can submit on Friday without being charged a late day. Submissions must still be received by Saturday 11:59:59PM to be graded.

Extra Credit

For extra credit, write a single priority queue class ElegantQueue and use two instances of it to solve the assignment. In addition to your `ElegantQueue.h` and `ElegantQueue.cpp`, you should submit a version of the provided files with the following names to demonstrate your queue in action: `main_extra.cpp`, `Job_extra.h`, and `Job_extra.cpp` . These files should only have the minimal changes necessary to use your new queue class. Your implementation should still be similarly "elegant". Describe your design in the README section for extra credit.