CS271: Data Structures

Priority Queue Worksheet

1. Use your heap class to build a templated priority queue class. The heap should be a private member of the priority queue. Implement the full suite of methods (all that were in the queue class) for the priority queue. The only method which is different is the enqueue; it is different only in its internal operation but not in the public interface. In other words, any application that uses a priority queue class should be able to use the exact same methods from the queue class.

2. We return to the idea of queues as line servers. This time we implement a job shop. Jobs have two initial statistics: an arrival time (int) and a service time (int). Jobs arrive at their appointed arrival time and then must wait for the server to become free to accept that job. The server works on the job for the processing time during which period it is busy and cannot accept new jobs. The goal of the server is to process jobs quickly; one important metric is to minimize the average wait time of each job. Therefore, it can be advantageous to rearrange the order of the jobs to improve this metric. For example, if jobs A and B arrive simultaneously and job A has a processing time of 1 second while job B takes 100 seconds, it makes sense to do job A first since it is short and job B won’t have to wait long. Naturally a priority queue can be used to organize the jobs by order of importance.

One strategy is called FCFS (first come first serve) in which jobs are prioritized according to their arrival time. This is the strategy applied to most human-queue situations (think food line at Wendys) since it is deemed socially unacceptable to let people butt in line before someone who has been waiting longer. Another strategy is SJF (shortest job first) in which jobs are prioritized by their processing time.

Implement a job shop program using a priority queue. Compare a FCFS strategy to a SJF strategy. I have provided two data sets (a small testing set and a larger more realistic set).

3. The priority queue uses comparisons to insert a new item in the queue. This is not a problem for simple types (int, char, etc), but for user defined objects it is necessary to provide overloads of the comparison operators: `operator<` and `operator>`. I created a “job” object in which I defined these two operators.

You will then realize that the only part of the code that changes for SJF and FCFS (and even SJFS) is in these two comparison operators. You can be "efficient" by using the same class and selecting code with an appropriate compiler directive such as:
int Job::operator> ( const Job &j ) const
{
#ifdef FCFS
    insert code here for FCFS comparison
#elif SJFS
    insert code here for SJFS comparison
#else
    // SJF
    insert code here for SJF comparison
#endif
}

Then you can configure your makefile to compile the same code into different executables:

all: testheap sjf fcfs sjfs

testheap: heaptest.cc heap.h heap.cc
g++ -o heaptest heaptest.cc

sjf: JobShop.cc JobShop.h PriorityQueue.h PriorityQueue.cc heap.h heap.cc
g++ -o sjf JobShop.cc

sjfs: JobShop.cc JobShop.h PriorityQueue.h PriorityQueue.cc heap.h heap.cc
g++ -o sjfs -DSJFS JobShop.cc

fcfs: JobShop.cc JobShop.h PriorityQueue.h PriorityQueue.cc heap.h heap.cc
g++ -o fcfs -DFCFS JobShop.cc

4. The format of the input is

    ArrivalTime1  ServiceTime1
    ArrivalTime2  ServiceTime2
    ...

with one job per line. Read input until EOF. The arrival time is a time stamp in seconds that indicates what time the job arrives. The Service Time is the time (in seconds) it takes to complete the job. Simulate only one job processor (not multiple processors like the optional part of Wendys).
Have your program read from stdin. This means it should not open/read any files but instead use `cin >>`. Use the command line to redirect input from a file to your program. That way your program can process different input files without recompiling.

The first line of your program’s output should be the average wait time as a floating point number. The second line of your program should be the max wait time (as a floating point number). You may print optional output following these two lines of output. Please comment out any debugging statements you have about jobs arriving, processing, finishing, etc.

5. **Things to Think About** (optional)

- Create some big data sets to test your program and collect statistics on average wait time. What probability distributions will you use to create your data sets?
- Is SJF the scheme optimal with respect to minimizing average wait time? Can you prove your answer?
- SJF introduces the possibility of ”starvation” which occurs when a large job is perpetually delayed because shorter jobs arrive and get processed first. One possible adaptation is to prioritize by processing time minus wait time. In this way, long jobs are eventually moved to the front of the queue if they have been waiting a long time. We’ll call this SJFS where the S denotes the adaptation to avoid starvation. Implement such a strategy and compare average wait times to FCFS and SJF.
- Can you think of other scheduling strategies that you can implement with a priority queue? With a different kind of structure? What would happen if you used a stack? This mimics the way that some people tackle their own work; take care of what ever has arrived most recently and then work your way backwards.
- Collect statistics on the variance (spread) of wait times and the maximal wait time along with average wait time. Can you plot various strategies to visualize their relative performances?