CSCI 2275 – Data Structures and Algorithms
Instructor: Hoenigman
Assignment 9
Due Wednesday, November 25, 6pm

Priority Queue Implementation and Analysis

Your local town is hosting this year's Very Very Pregnant People Pageant, which gathers hundreds of women who are very, very pregnant. Unexpectedly during the pageant, every single pregnant woman goes into labor. You live in a small town with only one hospital, so every pregnant person from the pageant heads to the hospital at the same time. How will the doctors decide the order in which the women will be seen? As the hospital's lone programmer, you volunteer to quickly code up a priority queue that will process the pregnant people properly.

As with most things in programming, there are many ways to solve a problem. To make things interesting, you decide to implement this priority queue using three different implementations and compare their respective performances in order to determine the best data structure for the job.

Assignment data

There is a file on Moodle called *patientData2275.csv* that includes a list of patients, time to delivery, and estimated treatment time. The patients with shorter time to delivery are higher priority than those with a longer time to delivery. The first row in the file is the header row, which shows what every column in the file includes. The second row in the file is the first row of data that you need to add to the queue:

Aaliyah, 152, 62

The fields in the header row show:
Name – first name of the patient
Priority – estimated time until baby born in minutes, between 30 and 300.
Treatment – estimated time that patient needs to spend with doctor in minutes, between 30 and 90.

Assignment requirements

Build priority queue using heap, linked list, and STL

You need to implement a priority queue using three approaches – a heap, a linked list, and the built-in C++ priority queue offered in the Standard Template Library. Each of your implementations should have its own .h and .cpp files, similar to the other data structures we have implemented this semester. For the linked list, you can modify the linked list code you wrote earlier this semester. For the STL, you don't need to build a separate class, you can just include the priority queue header in your main driver file.

Read priority queue data from a .csv file
The data that you need to store in your priority queue is in the *patientData_2275.csv* file available on Canvas. The name of that file needs to be handled as a command line argument to your program. There are 880 rows in the file. You can use a vector to store the entire data set when you read in the file.

Process priority queue data
After reading the data from the .csv file, you'll need to build your priority queues using each of your implementations. This means your implementations will need all the required methods to enqueue and dequeue items to and from a priority queue. Your priority queue should prioritize on the time until the baby is born (minimum first). In the case of equal times until the baby is born, prioritize next on the treatment time the patient requires (again minimum first). For example, a patient with an estimated delivery time of 30 minutes and treatment time of 10 minutes will have a higher priority than a patient with an estimated delivery time of 30 minutes and a treatment time of 20 minutes.

Runtime analysis

The different operations on your priority queue should have different runtime performance. For example, to remove an element from the heap should be constant, but there will be operations associated with the minHeapify() process. Building the heap will also involve a certain number of operations. The different implementations – array and linked list – should also have different performances. We expect to see that adding and removing an element with the array heap is faster than adding it the same element to the linked list, generally speaking. The STL may or may not be faster than either of your implementations just because it's a built-in library.

To analyze the runtime performance, you need to perform the build and remove operations 100 times for each implementation and calculate the mean and standard deviations of the runtimes in milliseconds. Your analysis needs to include running the data on files of different sizes so that you can get a picture of how the runtime changes with the size of the input data. There are 880 rows of data. Divide up your files into sets of 100 rows and run tests with 100, 200, 300, and so on. After you've run your tests, produce a graph showing the runtimes for each data structure for each data input size, similar to the graph you produced for Assignment 6.

To run these tests, push the data into the heap and then pop all data from the heap. Repeat the process for each data structure and data length. Once the data has been read in from the file and is stored in a vector, you shouldn't need to reopen the file, you can read from the vector.

Note: if you're running 8 tests 100 times each on three different data structures, it will take some time. Don't wait until the last minute to start these tests.

Code separation into multiple files

Your code needs to be separated into multiple files, such as priorityQueueLL.h, priorityQueueHeap.h, priorityQueueLL.cpp, priorityQueueHeap.cpp, and Assignment9.cpp, similar to other assignments in this class. To build your code, use:

```
g++ -std=c++11 priorityQueueLL.cpp priorityQueueHeap.cpp Assignment9.cpp –o
Assignment9
```

then run your code with

```
./Assignment9 <filename>
```

What to submit

Submit your code and graph as Assignment9.zip to the Assignment 9 link on Canvas.