Assignment 7 – ER Priority Queue

In this assignment, you will implement an emergency room priority queue by using a linked list. To do so, you will have to modify a linked list data structure in C to add patients to the queue with a defined priority.

You will also develop a main method to test your priority queue by receiving input from the standard input and by posting output to the standard output.

Programming environment and preliminary instructions

For this assignment, please ensure your work executes correctly on the Linux machines in ELW B238. You are welcome to use your own laptops and desktops for much of your programming; if you do this, give yourself one or two days before the due date to iron out any bugs in the C programs you have uploaded to a lab workstation. (Bugs in this kind of programming tend to be platform specific, and something that works perfectly at home may end up crashing on a different hardware configuration.)

Start your Assignment 7 by copying the files provided in **`/home/rbittencourt/seng265/a7`** into your **`a3`** directory inside the working copy of your local repository.

Your program will be run from the Unix command line, and will take no command-line parameters. The input data should come from the standard input, and the output data should be sent to the standard output. Any code that differs from these requirements will result in zero grade.

What you must submit

For this assignment, we will not use git as a submission tool. Instead, you will submit your source code files through Brightspace, via Assignment 7 in the Brightspace course site. *Please submit only the* **`list.c`** *and* **`pq-tester.c`** *files.* You should not submit test files, since we will use our own test files to test your source code. Our test files will be similar to the test files you will find in **`/home/rbittencourt/seng265/a7/pq-tests`**, but will have a different number of enqueue and dequeue commands just to be sure you are implementing your functions correctly. Attach **`list.c`** *and* **`pq-tester.c`** to the BrightSpace Assignment 7 page. Remember to click submit afterward. You should receive a notification that your assignment was successfully submitted.

CRITICAL: Any compile or runtime errors will result in a zero grade (if the tester crashes, it will not be able to award you any points for any previous tests that may have passed). Make sure to compile and run your program in the lab workstation environment (ELW B238) before submitting it!

Individual work

This assignment is to be completed by each individual student (i.e., no group work). Naturally you will want to discuss aspects of the problem with fellow students, and such discussion is encouraged. **However, sharing of code fragments is strictly forbidden without the express written permission of the course instructor (Roberto).** If you are still unsure regarding what is permitted or have other questions about what constitutes appropriate collaboration, please contact the course instructor as soon as possible. (Code-similarity analysis tools will be used to examine submitted work.) The URLs of significant code fragments you have found online and used in your solution must be cited in comments just before where such code has been used.

Description of the task

You will implement a priority queue scheme for an emergency room by using a linked list data structure. You will use the linked list **`add_with_priority`** function to *enqueue* patients in the priority queue, and the **`remove_front`** function to *dequeue* patients from the priority queue. The **`add_with_priority`** function is not implemented in the **`list.c`** file, and it is your job to correctly implement it according to the description below.

In the emergency room, higher priority means a smaller number than lower priority. For instance, a patient with priority 1 has a higher priority than a patient with priority 2. A new patient with higher priority must come before patients with lower priority in the linked list. Also, a new patient with the same priority as other patients must come after those patients in the linked list, and before patients with lower priority.

You will also implement functions inside **`pq-tester.c`** to interpret (parse) the *enqueueing* and *dequeueing* operations that will come from the standard input (or, indirectly, from a cat command over the input test files), and to deallocate the dynamic memory for the linked list: **`read_lines`**, **`tokenize_line`**, and **`deallocate_memory`**.

You must not provide filenames to the program, nor hardcode input and output file names.

Implementing your program

The C program you will develop in this assignment:

- Reads the input from **`stdin`** with the patients that arrive to the emergency room or leave the emergency room, and interprets each line as either an enqueueing or a dequeueing operation;
- For each enqueueing operation with a patient that comes from **`stdin`**, adds the patient's struct to the priority queue (which is based on a linked list), by inserting it after all patients with higher or same priority than the given patient;
- For each dequeueing operation from **`stdin`**, removes the first patient from the linked list;
- Dumps the ordered list of patients into **`stdout`**;
- Deallocated the dynamic memory for the linked list.

Assuming your current directory **`a3`** contains your **`pq-tester.c`** file with your **`main()`** function, that after compiling and linking your code you have the executable file **`pq-tester`** in the same directory, and that a **`pq-tests`** directory containing the assignment's test files is also in the current directory, then the command to run your executable will be:

```
% cat pq-tests/input01.txt | ./pq-tester
```

In the command above, you will pipe the input query text from the **`pq-tests/input01.txt`** to the **`pq-tester`** executable and the output will appear on the console. You may want to capture the output to a temporary file, and then compare it with the expected output. The **`diff`** command allows comparing two files and showing the differences between them.

```
% cat pq-tests/input01.txt | ./pq-tester > temp.txt
% diff pq-tests/output01.txt temp.txt
```

On the other hand, we will test your program with commands like the following one-liner. So, to test your code like we will, you have to use the following command structure with two pipes.

```
% cat pq-tests/input01.txt | ./pq-tester | diff pq-tests/output01.txt -
```

The ending `hyphen/dash` informs **`diff`** that it must compare **`pq-tests/output01.txt`** contents with the input piped into **`diff`**'s **`stdin`**. We will do this with other test files in the **`pq-tests`** directory.

Start with simple cases (for example, the one described in this write-up). In general, lower-numbered tests are simpler than higher-numbered tests. Refrain from writing the program all at once, and budget time to anticipate for when "things go wrong".

Exercises for this assignment

You may develop your code the way that suits you best. Our suggestions here are more for facilitating your learning than as a requirement for your work. You may not need to do the exercises in the item 1 below if you want to practice deeper problem solving skills. But, in case you get stuck, you may look at them as a reference. On the other hand, if Python programming seems difficult to you, you may use them as a script to learn and practice.

1. Write your program **`pq-tester.c`** program in the **`a3`** directory.

a. In the **`read_lines`** function, read the input from **`stdin`** by using either **`fgets()`** or **`getline()`**. Use a while loop to read all lines. If you wish, test your loop by printing the read lines into **`stdout`** with **`printf`**;
b. Now that you are reading each line in **`read_lines`**, call **`tokenize_line`** inside the former function to tokenize and interpret each line. If you wish, test your tokenizing loop by printing the read tokens into **`stdout`** with **`printf`**;
c. Use **`strcmp`** and conditional instructions inside your tokenizing loop to distinguish between *enqueue* and *dequeue* commands. Use local variables to store patient data in *enqueue* commands, and after reading all tokens, build the Patient struct. Either return a valid Patient pointer with *enqueue* commands or remove the first patient from the list with *dequeue* commands;
d. Back in **`read_lines`**, use the list **`add_with_priority`** and **`remove_front`** functions to either enqueue or dequeue patients in the list.
e. Check the **`dump`** results in the standard output, if they work as expected;
f. Recall to deallocate memory before you finish your program by correctly implementing the **`deallocate_memory`** function;
g. Test your program using the **`diff`** tool with the first test input file as explained in the section above; check whether your results are correct by comparing your output file with the first output file as explained in the section above;
h. Is your code working with this small example? So now test with the following input and output tester files, which contain more complex tests;
i. Have you respected the modularization requested inside the **`pq-tester.c`** starter file, and have commented your code during the development? If not, now it would be to fix this and document your code as well, in case you want an "A" grade.

Evaluation

Our grading scheme is relatively simple and is out of 100 points. Assignment 7 grading rubric is split into six parts.

1) Modularization - 10 points - the code should have the appropriate modularization as requested inside the **`pq-tester.c`** starter file, dividing the larger task into simpler tasks;
2) Documentation - 10 points - code comments (enough comments explaining the hardest parts (loops, for instance), no need to comment each line), function comments (explain function purpose, parameters and return value if existent), adequate indentation;
3) Compiling - 5 points - Code compiling with no warnings when using **`gcc -Wall -std=c11 ...`**;
4) Memory deallocation - 5 points – Correctly deallocating (freeing) dynamic memory used for the linked list before the program ends;
5) Tests: Part 1 - 10 points - passing test 1 in **`pq-tests`** directory: **`input01.txt`** as input and **`output01.txt`** as the expected output;
6) Tests: Part 2 - 20 points - passing test 2 in **`pq-tests`** directory: **`input02.txt`** as input and **`output02.txt`** as the expected output;
7) Tests: Part 3 - 20 points - passing test 3 in **`pq-tests`** directory: **`input03.txt`** as input and **`output03.txt`** as the expected output;
8) Tests: Part 4 - 20 points - passing test 4 in **`pq-tests`** directory: **`input04.txt`** as input and **`output04.txt`** as the expected output.

We will only assess your final submission sent up to the due date (previous submissions will be ignored). On the other hand, late submissions after the due date will not be assessed.