Report for LAB 2

Mafalda de Macedo Pinto Candal

Deadline: 16th October 23:59

Task description

LAB 2 set the challenge create a program that helps manage a doctor's office waiting room, by organizing patient admissions, and treatments based on priority. Patients are added to a waiting list with their name, age, and priority. Those with the highest priority are treated first, but in cases of equal priority, patients are treated in the order they arrived. The program offers options to add new patients [N], treat the highest-priority patient [T], display the current waiting list [L], and exit [Q].

Structures and Algorithms

To represent the waiting list, I made use of a linked list, implemented through the struct Patient, where each node holds a patient's details (name, age, priority) and a pointer to the next patient (next). The linked list (1) has a dynamic size, allowing for efficient dynamic memory management, where we can add or remove patients as needed without worrying about a fixed size, and (2) allows for adding and removing nodes efficiently.

The insertion complexity of the linked list is O(n). If the new patient has the highest priority, the insertion happens at the front in O(1) time. However, for patients with lower priorities, the list must be traversed to find the correct position, leading to **O(n)** time complexity in the worst case. On the other hand, the **removal complexity** is always **O(1)** because removing the highest-priority patient from the front only requires updating the head pointer.

While I discovered possibly more efficient data structures like Binary Heap, Balanced Binary Search Tree which reach O(log n) insertion complexity, I did not feel like my knowledge was enough to comfortably implement them and learn the most through this assignment. And given the scale of the problem, a linked list is a practical choice for managing a small to medium number of patients, balancing ease of implementation with adequate performance for this assignment.

Once again, modularity was a priority of mine when writing this program, there are four helper functions: GetPatientDetails(), Admit(), GiveTreatment() and CheckWaitingList().

I first attempted to create a single function to carry out command N, which included creating a patient and inserting them into the queue. I thought that the one-step process would contribute to simplicity and to reduce functions calls, but at last, I considered that having two functions—GetPatientDetails() and Admit()—attributed a unique, more manageable task to each helper function. This separation improves modularity, and, while this is a stand-alone assignment, it would allow to (1) reuse the patient creation logic in other contexts, (2) simplify testing by isolating functionality, and (3) make the code easier to maintain and modify.

The function GiveTreatment() removed the patient with the highest priority from the list – command T – and CheckWaitingList() allows to print the patient waiting list – command L. I also initially had a IsWaitingListEmpty() function, but I considered that it was not used enough to be worth creating a function. The improvement in readability was minimum and it seemed redundant.

Evaluation of the program

Writing and modifying this code was a 2-steps-forward-and-1-step-back type of process. I frequently ran “make test” to understand how my code was running and I also tested many of the commands for an empty list (the most apparent edge case). Among the errors I encountered, I can name: (1) having way too many getchar() which I eliminated by adding a space in my scanf() expression, (2) not freeing the memory allocated when running the command “Q”, which I rectified by running the function GiveTreatment() until all memory had been freed, (3) having an unnecessary tail pointer, which I later removed because the list was ordered by priority and therefore it was not needed, and (4) I missed incrementing the count when the first patient was added. I fixed this by ensuring the count was updated inside the condition that handles the first patient.

Conclusion and Self-Reflection

This assignment much expanded my understanding of linked lists, as well as other data structures which I researched when choosing the main structural element for my assignment. While I found the guidelines of the assignment very straightforward and the commands easy to abstract, it was the function Admit() that was the most difficult to achieve. Because it used the struct Patient and the priority system, it took me a little while to grasp the dynamics of the pointers (the various “base” or “special” case that I needed to account for). I’d say this was a process of research followed by a period of trial-and-error. In the end, it was a logical process that proved quite achievable and enjoyable with the help of the class presentations.

As I struggled with finding my initial direction in this assignment, I would appreciate discussing the practical uses of each data structures and its different implementations (e.g. a queue can be implemented using many different data structures). I’d also love to hear more about Lorenzo’s experience as a programmer and the creative and artistic element to coding. Thank you.