

TASK 1

- First, let's find the Big O value of the functions. Our `addPatient(Patient p)` function works in $O(1)$ time because when we add a patient to the list, we add it at the end, and since we have the last patient's data (tail), we can add it directly to the end of it. If there was no tail, it would take $O(n)$ time and this would increase the cost because we had to count one by one from the beginning to access the latest data. `removePatient(int id)` runs in $O(n)$ time because it needs to navigate through all the data to know which patient to delete. The `findPatient(int id)` function also runs in $O(n)$ time because it needs to look at all patients one by one to find the patient.
- If we had added an `ArrayList` here, `findPatient(int id)` would still have $O(n)$ complexity because it would have to navigate through all the data, and the `removePatient(int id)` function would still run in $O(n)$ time, but when we deleted a data in the `ArrayList`, it would have to shift all the data to the left of it, and this is an extra cost. The `addPatient(Patient p)` function would go from $O(1)$ to $O(n)$ because when we add data to the end of the `ArrayList`, if the capacity of that array is full, Java first creates an array that is twice the length of that array (this is called resizing) and first copies the array to the newly created array one by one and adds the data we want to add to the end, and this would have a bad effect on the code because it has $O(n)$ time complexity, but `LinkedList` doesn't have this capacity issue, it always adds in constant time.

TASK 2

- The logic of the queuing system is first in, first out. This is called FIFO system. This type of structure is used because it should be a more fair structure to work in queuing systems. The issue is that in structures such as bank queues, hospital queues, bread queues, first in, first out is very important. In this case, it guarantees that everyone receives service and thus a fair structure is provided.
- If we had used `Stack` instead of `Queue`, the LIFO system would have been activated instead of the FIFO system, and this is something we never want because there would never be a guarantee that the first one in would be first out. For example, a patient arriving at 09.00 would always be at the end of the queue, a patient arriving at 09.30 would be ahead of the first patient, and a patient arriving at 10.00 would be ahead of the patient arriving at 09.30, and perhaps those people would never be able to get their turn, because each new patient would come to the front of the queue, and in this case, it would be a very unfair structure (The technical name of this situation in the literature is "Starvation").

- Now let's talk about time complexities. First of all, the time complexity of the `enqueue(TreatmentRequest request)` function is $O(1)$ because since I can access the end of the list thanks to `tail`, I can dequeue the last element in $O(1)$ time. The `dequeue()` function is also $O(1)$ because since I can reach the top of the list thanks to the `head`, I can pull the first element in $O(1)$ time. It is more appropriate to use Queue here since the time complexity of adding and removing in both Queue and Stack structures requires $O(1)$ time, but requires sequential processing logic in things such as hospital systems and banking systems.

TASK 3

- First, let's start with what Stack is. Stack's logic is LIFO, meaning last in, first out, and since the last discharged person is generally important in discharge records, it is important to know the last discharged person and add or remove them. Such logic is actually known as "History Log" and internet history is actually managed with it (For example, show the last website I visited). Since only the top part of the list is dealt with in such structures, a single "top" variable will be sufficient.
- If we had used Queue instead of Stack, FIFO logic would come into play and in order to find the last person discharged, we would either need a "tail" variable indicating the end of the queue, or we would have to go through the list one by one from the beginning, which would be $O(n)$ instead of $O(1)$. If we wanted to delete the data of the last discharged person, even if we had "tail", we would have to go to the end of the list because we needed to access the data before that, and it would always be Big $O(n)$, which is a very undesirable situation. Stack is much more suitable for these situations.
- The time complexity of this Stack structure is that both the `push(DischargeRecord record)` function and the `pop()` function have a time complexity of $O(1)$. Because the process is always done from the beginning and we have a variable called "top", our access to the beginning is always $O(1)$.

TASK 4

- I used two different queues in my code. One is a priority queue (for emergency patients) and the other is a normal queue (for normal patients). If there are patients in the emergency queue, those patients will be removed first, followed by normal patients. Regarding time complexity, the `addTreatmentRequest(TreatmentRequest request)` function adds to the end of the specified queue in $O(1)$ time, based on the `isPriority` condition, because we have the `tail` data, so we

can access and add the data at the very end in $O(1)$ time. The `processTreatment()` function, on the other hand, retrieves patients from the priority queue first if there are any, otherwise it retrieves patients from the normal queue if there are any. Both retrieve in $O(1)$ time because we have the `head` data, so we can access and add the data at the very beginning in $O(1)$ time. For sorting, I used Merge Sort, which is the fastest sorting algorithm I know, and this sorting algorithm has a time complexity of $O(n \log n)$.

- Using two different priority queues significantly speeds up the process. If we wanted to do it with a single queue, we would have to check the entire sequence from the beginning every time a patient was added, which would add extra workload. However, the drawback of this system is that currently, patients are only separated into urgent/non-urgent categories, but if urgent patients also need to be sorted from 1 to 10, more queues would be created, making it unmanageable in the future. Furthermore, if we were to use Merge Sort to keep the list continuously sorted, there would be an extra cost to sort each time a patient is added. If we used a Heap-Based Priority Queue structure, the list would be continuously sorted, data insertion/removal operations would be much faster, and it would have $O(\log n)$ time complexity. We could check all patients in a single list, and even if they had urgency statuses from 1 to 100, they could all be checked in one list.