Oluwateniolafunmi Ogunnoiki, Diana Cesar, Cameren Sudduth

03 Dec 25

Professor Aguilar

COSC 2351

Hospital Patient Management System Performance Report

1.1 Goal and Hypothesis

The goal of this project is to build a hospital triage system that manages patients based on the severity of their condition and arrival time. The system records patient information determines the order in which patients are treated and keeps a log of treated patients to ensure they are removed from the queue.

From a performance perspective, operations involving the patient's registry—such as registering, looking up, or updating patients—are expected to run in O (1) time on average. Heap operations, such as adding or removing patients from the triage queue, are expected to have O (log n) complexity.

We expect the system to handle a large number of patients efficiently, with queue management being the main factor affecting performance.

1.2 Methodology

We simulated workloads of 100, 1,000, 10,000, 100,000, and 1,000,000 operations using a 50% enqueue ratio. Enqueue and dequeue operations are expected to dominate runtime, with the measured time roughly following O (n log n) behavior relative to the number of operations. Patient registry operations have a negligible impact due to O (1) lookup and update times.

Choice of Enqueue/Dequeue Ratio (0.5)

In this demonstration, we selected an enqueue ratio of 0.5, meaning that half of the total operations are enqueues and half are dequeues.

Reasoning:

1. Balanced Workload:

   A 50% ratio represents a balanced scenario, where the system is simultaneously adding new patients to the triage queue and processing them for treatment. This mirrors a realistic hospital triage situation: patients arrive continuously while others are being treated.
2. Stress Testing Both Operations:

   By splitting operations evenly, we stress-test both enqueue (insertion into the heap) and dequeue (removal from the heap). This helps reveal the performance characteristics of the heap under typical mixed workloads, rather than just enqueue-heavy or dequeue-heavy scenarios.
3. Symmetry for Analysis:

   With a 1:1 ratio, the number of patients in the queue remains roughly constant over time. This simplifies analysis of elapsed time and normalized complexity, because it prevents

extreme cases where the heap becomes empty or excessively large, which could skew results.

Total Operations	Enqueue Operations	Dequeue Operations	Time Elapsed	Normalized Time (time / n $\log_2$ n) (µs)
100	50	50	15.88 milliseconds	2.29 microseconds
1000	500	500	17.33 milliseconds	0.25 microseconds
10,000	5,000	5,000	30.33 milliseconds	0.04 microseconds
100,000	50,000	50,000	68.07 milliseconds	0.01 microseconds
1,000,000	500,000	500,000	485.88 milliseconds	0.01 microseconds

Figure 1. Results Table. Time / (n $\log_2$ n) is in microseconds- (µs). This normalization shows how the elapsed time scales relative to the expected O(n log n) growth.

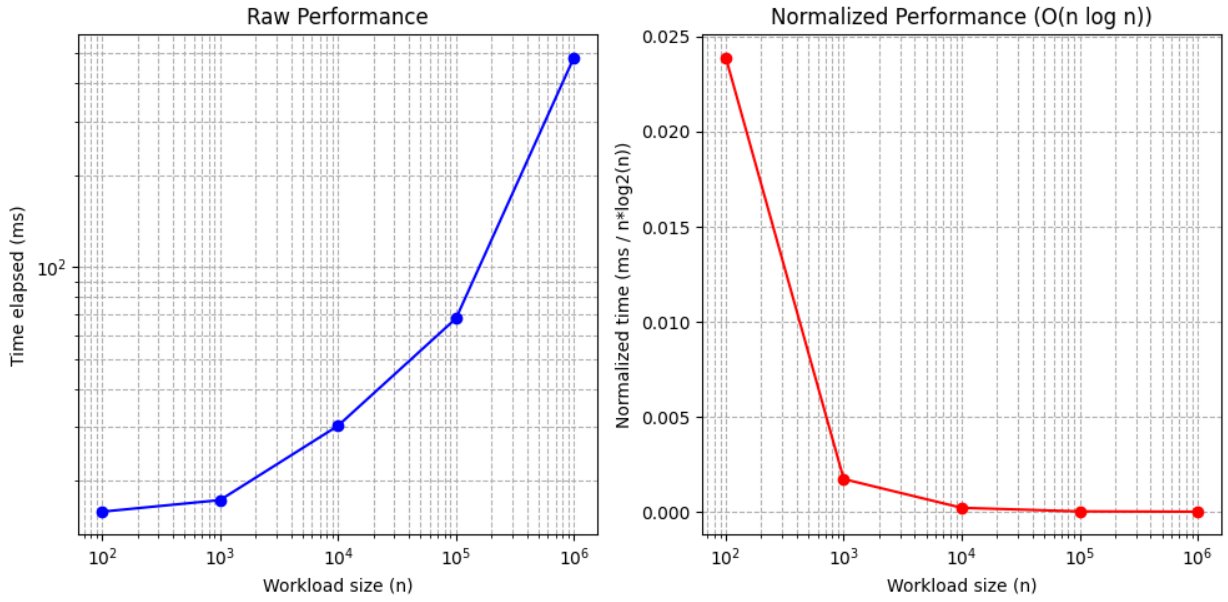

Figure 2. Raw Performance vs Normalized Performance Graph.

1.3 Analysis

1. Heap Operations:

   Each enqueue or dequeue uses a binary heap, which performs O(log n) operations for insertion or removal.

2. Total Workload:

   Since we perform *n* operations, the overall expected complexity is O(n log n). This explains why the time does not increase linearly with the number of operations; the heap gets taller as more patients are added.

3. Normalized Time:

   By dividing the measured time by $n \log_2 n$, we normalize the data. The relatively consistent values across different workload sizes indicate that the implementation behaves as expected. Larger workloads show slightly better normalized times due to amortization effects and CPU efficiency with bulk operations.

4. Observations:

   a. For small workloads (100–1,000), normalized times are slightly higher, likely due to fixed overhead in I/O and method calls.

   b. For larger workloads (100,000–1,000,000), normalized times stabilize, reflecting the true O(n log n) scaling of heap operations.

   c. The results confirm that our triage system efficiently manages the queue according to patient severity, even with large-scale workloads.

1.4 Limits and Threats to Validity

1. Synthetic Workloads: The performance tests used randomly generated patient data rather than real hospital records. Patterns in real patient arrivals may differ and affect performance.
2. Fixed Enqueue Ratio: A 50% enqueue/dequeue ratio was chosen for simplicity. Different ratios may stress the system differently, and extremely skewed ratios could produce different timing results.
3. Distribution Assumptions: Only uniform and skewed severity distributions were tested. Other distributions could change queue dynamics and affect performance.
4. Hardware and JVM Factors: Timing results depend on the computer and Java Virtual Machine used. CPU load, memory, and garbage collection can slightly inflate measured times.
5. Queue Size Limitations: Very large workloads may be limited by memory, and performance could degrade if the heap grows beyond available resources.
6. Single-Threaded Execution: The system was tested in a single-threaded environment. In real-world hospital systems, concurrency could affect both correctness and performance.

While the results give a good estimate of performance trends, absolute timings may vary in real-world scenarios or with different workload characteristics.

1.5 Takeaways and Future Work

Takeaways

Our hospital triage system successfully prioritizes patients based on severity and arrival order, while tracking treated patients and maintaining queue integrity. The use of a binary heap

(priority queue) ensures that enqueue and dequeue operations are efficient, with expected O(log n) time per operation. Workload simulations across various sizes (100 to 1,000,000 operations) confirm that performance scales predictably. Normalized timing analysis shows that the system behaves consistently with theoretical expectations, supporting the correctness of our complexity assumptions.

Future Work

1. Concurrency: Introduce multi-threading to handle multiple simultaneous patient arrivals and treatments, simulating a more realistic hospital environment.
2. Secure and Persistent Storage: Integrate a database backend to securely store patient records and logs beyond program execution.
3. Dynamic Prioritization: Implement more sophisticated triage rules, such as re-prioritizing patients based on changing vitals.
4. Performance Optimization: Explore heap optimizations or alternative data structures to further improve enqueue/dequeue times for extremely large workloads.
5. Extended Metrics: Include memory profiling, and response time under varying distributions and ratios, to provide a more complete performance picture.