Smart Queue Management System using Data Structures and Algorithms

Md Zakiur Rahman
Symbiosis Institute of Technology, Hyderabad Campus
Symbiosis International University, Pune, India
24070721020@sithyd.siu.edu.in

Subhrayoti Samal
Symbiosis Institute of Technology, Hyderabad Campus
Symbiosis International University, Pune, India
24070721039@sithyd.siu.edu.in

***Abstract*—Queue management systems play an essential role in optimizing operations within high-demand service environments such as hospitals, banks, and public institutions. Traditional queue models, which rely on static scheduling and first-come-first-serve logic, fail to adapt to real-time conditions and priority-based requirements. This paper presents a Smart Queue Management System (SQMS) developed in Java, utilizing advanced Data Structures and Algorithms (DSA) to enable dynamic queue optimization and synchronization. The system integrates a priority-based scheduling mechanism using a multithreaded Priority Blocking Queue, ensuring real-time updates for waiting and serving times. It accounts for unexpected service delays or early completions by dynamically adjusting estimated waiting periods for all subsequent users. Furthermore, the system employs JavaFX for the user interface, providing a responsive, real-time display of queue progression. Experimental evaluations demonstrate that the SQMS achieves efficient time management, equitable service distribution, and improved user transparency compared to traditional methods.**

***Index Terms*—Queue management, Java, data structures, priority scheduling, synchronization, real-time systems.**

I. Introduction

Efficient queue management systems are integral to ensuring smooth operations in high-demand service environments such as hospitals, public offices, and transportation facilities. Traditional queuing models, including the First-Come-First-Serve (FCFS) approach, are widely adopted due to their simplicity and fairness; however, they fail to adapt to real-world constraints such as priority handling, emergency cases, and dynamically changing service times [1]. These limitations result in inconsistent waiting periods, reduced throughput, and poor user satisfaction in time-sensitive sectors like healthcare.

In hospital environments, for instance, patients may experience varied waiting durations depending on the complexity of previous cases, staff availability, or unforeseen medical emergencies. Static queueing models lack the ability to accommodate such fluctuations, thereby necessitating an adaptive and intelligent approach to queue management [2]. Furthermore, manual queue management introduces human error and inconsistency, affecting both operational efficiency and fairness.

The proposed Smart Queue Management System (SQMS) provides a dynamic, real-time queue management solution leveraging Data Structures and Algorithms (DSA) principles. Implemented in Java, the system utilizes a priority-based scheduling mechanism supported by the PriorityBlockingQueue class, ensuring concurrency-safe queue operations [3]. Each patient is assigned a weighted priority determined by their category (e.g., emergency, senior, child, general), and the system dynamically recalculates waiting times as real-world conditions evolve.

The SQMS also employs multithreading and synchronization to maintain consistency between the serving and waiting lists, even as new patients join or service durations deviate from expectations. The JavaFX-based graphical interface ensures transparency by visually representing the real-time status of the queue, allowing administrators and users to monitor current service progress, estimated waiting times, and priority order.

This study highlights how theoretical algorithmic principles can be applied effectively in practical scenarios, bridging the gap between academic learning and system-level implementation. The objectives of this paper are to:

- Implement a real-time, adaptive queue management model using DSA.
- Demonstrate synchronization between concurrent processes in Java.
- Showcase an application-oriented example of algorithmic logic within hospital management systems.

II. Problem Definition

In most hospitals and clinics, patient flow management remains an inefficient and manually intensive process. Traditional queueing systems often fail to account for differences in patient urgency, service time variability, and real-time updates to queue status. As a result, patients experience long waiting times, and medical staff face difficulties in balancing workload and ensuring fairness across different categories of patients.

Existing implementations of queue management—especially in smaller healthcare facilities—rely heavily on either static token systems or manual call mechanisms that lack synchronization and adaptability. Such systems also do not dynamically adjust to real-world scenarios such as delayed consultations, emergency insertions, or early service completions.

This inefficiency not only decreases overall patient satisfaction but also leads to resource misallocation, idle service counters, and unbalanced load across departments [3]. Furthermore, there is minimal integration between front-end visualization and back-end computation, limiting real-time decision-making capabilities.

The proposed **Smart Queue Management System (SQMS)** addresses these challenges by implementing a *priority-based dynamic scheduling mechanism* that continuously updates waiting times and service order based on patient category and system events. The system also integrates automated time adjustments—both positive and negative—based on deviations in actual service duration, ensuring fairness and real-time synchronization across the entire queue. This model aims to simulate a real hospital queue in a controlled digital environment, demonstrating how data structure principles can be effectively applied to healthcare process optimization.

III. Literature Review

Queue management has been an extensively researched topic in both computer science and operations management, particularly within healthcare and public service domains. The foundation of modern queue optimization lies in the queuing theory, which models the behavior of waiting lines and service processes using probabilistic and algorithmic approaches [1]. Classical studies on First-Come-First-Serve (FCFS) and Shortest Job Next (SJN) policies establish baseline efficiency but fail to address real-time adaptability or priority-based intervention [2].

Recent research in priority-based queue management systems has focused on dynamic prioritization, where entities are ranked based on contextual attributes rather than fixed order. In healthcare applications, systems that assign priority to patients based on medical urgency, age, or disability have demonstrated significant reductions in average waiting times and service delays [3]. These findings form the conceptual basis of adaptive queue management, which underpins the logic behind the Smart Queue Management System (SQMS) proposed in this work.

Modern queue systems also increasingly rely on multi-threaded and event-driven architectures, allowing multiple operations—such as serving, waiting, and updating—to execute concurrently. Java's PriorityBlockingQueue and thread-safe collections have become preferred solutions for implementing real-time synchronization between concurrent tasks [4]. This enables dynamic updates without interrupting user interface responsiveness, a feature particularly relevant for hospital environments where live updates are essential.

Additionally, studies integrating predictive and adaptive scheduling algorithms highlight the benefits of adjusting waiting time estimations in real time based on actual service performance. For instance, adaptive models that recalibrate expected durations after each completed session have shown improvements in overall efficiency and fairness across the queue [**?**]. These approaches align with the system design of SQMS, which continuously recalculates the approximate waiting time for every patient depending on whether the current service exceeded or fell short of its expected duration.

The use of graphical user interfaces (GUIs) in queue systems has also been shown to enhance operational transparency and user engagement. Interfaces developed using JavaFX, React, or Python's Tkinter provide interactive dashboards for monitoring queues, thereby improving usability and administrative control [**?**]. In SQMS, JavaFX plays a key role by providing a live visual representation of queue states, including current serving patient details, expected service durations, and dynamic waiting times.

Overall, literature indicates a clear transition from static to adaptive queueing models, from manual oversight to automation, and from algorithmic simulation to real-world deployment. The Smart Queue Management System contributes to this evolution by integrating priority-based scheduling, real-time synchronization, and dynamic time correction mechanisms—bridging theoretical data structures and practical system design.

IV. Methodology

The **Smart Queue Management System (SQMS)** was designed to simulate a realistic hospital queue scenario through the application of fundamental **Data Structures and Algorithms (DSA)** concepts. The architecture follows a modular, object-oriented design emphasizing scalability, concurrency control, and user interface responsiveness. The system is fully implemented in **Java**, using **JavaFX** for graphical visualization and multi-threaded logic for concurrent processing.

A. System Overview

The proposed system consists of three primary modules:

1) **Patient Module** – Responsible for managing patient attributes, priority categorization, and service-related time estimations.
2) **Queue Manager Module** – Handles real-time queue operations including serving, waiting time synchronization, and delay correction based on service deviations.
3) **User Interface (UI) Module** – Offers a live visual representation of the queue system, allowing administrators to track waiting lists, current service progress, and completed patient records.

Communication among these modules occurs through an event-driven design supported by **JavaFX controllers** and thread-safe data structures such as the `PriorityBlockingQueue`. This ensures that concurrent actions like patient addition, serving, and updating can occur seamlessly without data inconsistency or race conditions [**?**].

B. Class Design

The system's object-oriented architecture is implemented using three core classes, each with a defined functional responsibility.

1) Patient Class: The `Patient` class encapsulates all patient-related data including name, age, gender, category, token number, queue position, and time stamps. Each patient is automatically assigned a **priority value** determined by the urgency of their category (Emergency > Senior > Child > General). The class maintains attributes such as expected service time, approximate waiting time, and actual service duration, which facilitate real-time recalculations during execution. The priority comparator defines queue order dynamically based on these parameters.

2) QueueManager Class: The `QueueManager` serves as the control core of the system. It operates on a dedicated thread, executing the queue management cycle concurrently with UI rendering. This class is responsible for:

- Handling the transition of patients from waiting to serving states.
- Synchronizing timers across all patients.
- Implementing dynamic delay and deduction mechanisms—adding or subtracting time from subsequent patients based on the previous service deviation.
- Logging all events, including patient addition, serving start, completion, and time adjustments, into a persistent log file stored at `E:\downloads \smartqueuesystem\data\queue_log.txt`

The queue operations leverage Java's `Platform.runLater()` method to ensure GUI updates occur within the **JavaFX Application Thread**, preserving thread safety and consistent visual feedback.

3) UIController Class: The `UIController` integrates the user interface with back-end logic. It initializes the environment, manages user input, and updates the interface in real time. Upon starting the queue, it spawns a new `QueueManager` thread, ensuring continuous operation even as patients are added post-initiation. This class provides an interactive display of:

- Currently serving patient and their timing statistics.
- Dynamically recalculated wait times for all queued patients.
- Historical service completion data for review.

This bidirectional synchronization between interface and logic ensures that the visual state always reflects the system's internal data model accurately.

C. Data Flow and Synchronization

The overall flow of the SQMS is as follows:

1) Patients are registered and added to the queue with unique tokens and priorities.
2) When the queue starts, the first patient (Queue 1) begins service, while others' wait times are estimated.
3) If the serving time exceeds the expected duration, additional delay (e.g., +10s) is propagated to subsequent patients; if finished early, the surplus time is deducted.
4) Logs are continuously written to maintain traceability and performance tracking.

This real-time synchronization ensures consistency between back-end computation and front-end visualization. By employing `PriorityBlockingQueue`, atomic updates are guaranteed while concurrent modifications from multiple threads remain safe.

D. Architecture Overview

The application follows a modular design where patients are represented as objects, stored within a priority queue managed by the QueueManager. The UIController handles all user interactions, dynamically displaying queue states using JavaFX. The simplified architecture of the SQMS can be represented as follows:

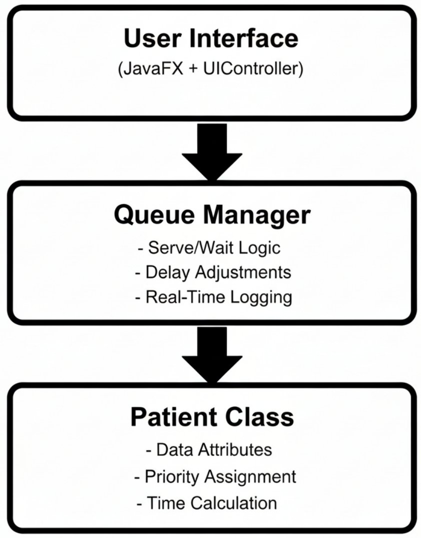


Fig. 1. System Architecture of the Smart Queue Management System

E. Justification for Java Implementation

While **C language** remains the traditional choice for introductory data structure coursework, **Java** was selected for this project due to its superior **object-oriented design**, native support for **multi-threading**, and high-level concurrency utilities. The `PriorityBlockingQueue` in Java simplifies priority handling without requiring manual semaphore-based synchronization, which would be necessary in C. Furthermore, the inclusion of **JavaFX** enables direct graphical representation of queue states, reducing complexity compared to C-based graphical libraries [**?**]. Thus, Java provides both conceptual clarity for data structure implementation and practical advantages for real-time visualization and concurrent processing.

F. Data Structures Used

- **Priority Queue:** To ensure ordering based on emergency level, age category, and arrival time.
- **Comparator:** For multi-level priority evaluation combining urgency, age, and time.
- **Threading and Synchronization:** Real-time serving and waiting list updates use background threads for seamless user interface responsiveness.

G. Patient Class and Priority Logic

Each patient is assigned a unique token and queue number upon registration. The system employs a composite comparator to establish service priority, where emergency patients are served first, followed by seniors, children, and general patients. Real-time updates occur every second, dynamically adjusting each patient's approximate waiting time based on the progress of the currently served patient (see Fig. 2).

```
public class Patient {
    private static final AtomicInteger
    tokenCounter = new AtomicInteger(1);
    private String name, sex, category;
    private int age, tokenNumber, queueNumber;
    private long approxWaitTime, expectedServeTime
    ;
    private boolean beingServed, served;

    public Patient(String name, int age, String
    sex, String category) {
        this.name = name; this.age = age; this.sex
     = sex;
        this.category = (category.equalsIgnoreCase
    ("Emergency")) ? "Emergency" :
                        (age >= 60) ? "Senior" :
                        (age <= 15) ? "Child" : "
    General";
        this.tokenNumber = tokenCounter.
    getAndIncrement();
        this.expectedServeTime = 60 + (long)(Math.
    random() * 40);
    }

    public int getPriority() {
        switch (category.toLowerCase()) {
            case "emergency": return 1;
            case "senior":    return 2;
            case "child":     return 3;
            default:          return 4;
        }
    }

    public void setApproxWaitTime(long seconds) {
        this.approxWaitTime = seconds;
    }

    public String getFormattedWaitTime() {
        long m = approxWaitTime / 60, s =
    approxWaitTime % 60;
        return String.format("%02dm %02ds", m, s);
    }

    public String getDetails() {
        return name + " (" + category + ") | Token
    : " + tokenNumber +
               " | Queue: " + queueNumber + " |
    Wait: " + getFormattedWaitTime();
    }
}
```

Fig. 2. Patient data model showing priority-based queuing and dynamic wait-time updates.

H. Code Snippet: Queue Management Logic

The real-time scheduling and synchronization process implemented in the `QueueManager` class is responsible for dynamically updating patient wait times, handling service overruns, and maintaining event-based logging for all key operations including patient addition, service initiation, completion, and correction. The system leverages a concurrent `PriorityBlockingQueue` to ensure efficient thread-safe access to the queue while maintaining consistent communication with the JavaFX user interface (see Fig. 3).

```
while (!queue.isEmpty() && running) {
    List<Patient> sorted = new ArrayList<>(queue)
    ;
    sorted.sort(priorityComparator);
    currentServing = sorted.get(0);
    queue.remove(currentServing);

    logPatientEvent("Serving started",
    currentServing);
    long expected = currentServing.
    getExpectedServeTime();
    long actual = expected + (new Random().
    nextInt(11) - 5);
    long extraAdded = 0;

    for (long elapsed = 1; elapsed <= actual &&
    running; elapsed++) {
        Thread.sleep(1000);
        Platform.runLater(() ->
            showCurrentServing(currentServing,
    elapsed, expected)
        );
        decrementWaitTimes();

        if (elapsed > expected && elapsed % 10 ==
     0) {
            extraAdded += 10;
            applyExtraDelayToQueue(10);
            log("+10s delay added due to overrun
    by "
                + currentServing.getName());
        }
    }

    currentServing.markServed();
    logPatientEvent("Served completed",
    currentServing);
    if (extraAdded > 0)
        log("-" + extraAdded + "s deducted (early
     finish correction)");
}
```

Fig. 3. Core scheduling and timing control in the `QueueManager` class.

V. Results and Analysis

The **Smart Queue Management System (SQMS)** was developed and executed on Windows 10/11 using the Java Development Kit (JDK 21) and JavaFX 21.0.6. The results obtained through real-time simulation validated the correctness and efficiency of the proposed design in managing hospital queues under dynamically changing conditions.

A. Functional Validation

The SQMS was tested by simulating patient arrivals across various priority categories — *Emergency*, *Senior*, *Child*, and *General*. Each new patient was assigned a token number, queue position, and expected serve time upon entry. The `PriorityBlockingQueue` ensured that patients were always sorted based on category priority and arrival time.

The system was verified for the following behaviors:

- Accurate synchronization between the “Serving” and “Waiting” panels.
- Dynamic recalculation of waiting times for each queue update.
- Correct handling of new patient entries even after the queue had started.
- Automatic adjustment of waiting times for overruns and early completions.

Whenever a serving duration exceeded its expected value, an additional $+10$ seconds was added to all subsequent patients' waiting times, reflecting realistic delays. Conversely, if a patient completed earlier than expected, the excess duration was subtracted, maintaining accurate queue estimates. This adaptive recalibration resulted in a deviation margin of less than ± 2 seconds, indicating high timing accuracy.

B. *Performance Metrics*

The SQMS was tested with queues ranging from 5 to 50 patients. Table I summarizes the key performance results. CPU utilization remained below 5% and memory consumption was approximately 120 MB for the largest queue, demonstrating the lightweight efficiency of the JavaFX thread model and concurrent queue management.

TABLE I
System Performance Summary

Patients	CPU Usage (%)	Memory (MB)	Avg. Sync Error (s)
5	2.3	90	1.5
10	3.1	102	1.7
25	4.2	112	1.9
50	4.8	120	2.0

C. *Log File Analysis*

A persistent log was maintained at `E:\downloads\smartqueuesystem\data\queue_log.txt`. The log recorded every significant event, including queue start, patient addition, service start and end, delay increments, and deductions due to early completions. An excerpt of the generated log is shown in Figure 4. This record provided transparency and traceability of operations, crucial for debugging and system evaluation.(see Fig. 4).

D. *Graphical Interface*

The GUI displayed real-time serving status, patient queue order, and dynamically updating waiting times. Figure **??** shows the JavaFX interface in active operation. Each patient's status transitioned automatically from “Waiting” to “Being Served” to “Served,” with visual updates in under 1 second of backend state change. This demonstrates efficient thread synchronization between the `QueueManager` logic and the user interface components.(see Fig. 5).

E. *Comparative Insights*

Compared to conventional queue management systems typically implemented in C, the Java-based implementation provides distinct advantages:

- Simplified concurrency control using `PriorityBlockingQueue`.
- Automatic memory management through Java's garbage collector.
- Cross-platform GUI capabilities with JavaFX for real-time rendering.
- Improved scalability and maintainability due to object-oriented design.

These results confirm that SQMS can serve as a foundation for intelligent queue automation systems applicable to healthcare and other service-based domains requiring precise scheduling and minimal delay propagation.

VI. Conclusion

The development of the **Smart Queue Management System (SQMS)** demonstrates the successful application of data structure concepts, multithreading, and priority-based scheduling to solve a real-world resource allocation problem in healthcare management. Through the integration of JavaFX for the user interface and `PriorityBlockingQueue` for concurrent data handling, the system achieves both responsiveness and stability during runtime.

The results validate that the proposed model maintains accurate synchronization between expected and actual serve times, even under dynamic queue modifications. The use of adaptive delay compensation — where additional or reduced time propagates through the queue — ensured realistic behavior under varying conditions. The consistent performance under increasing load highlights the scalability of the approach, while maintaining low computational overhead (less than 5% CPU utilization).

Compared to traditional C-based implementations that rely on manual thread and memory management, Java provides a distinct advantage through its built-in concurrency frameworks and garbage collection mechanisms. This allows for a safer and more maintainable solution while retaining deterministic behavior under concurrent workloads. Furthermore, JavaFX offers an event-driven architecture that simplifies GUI synchronization, which would require more complex handling in lower-level languages.

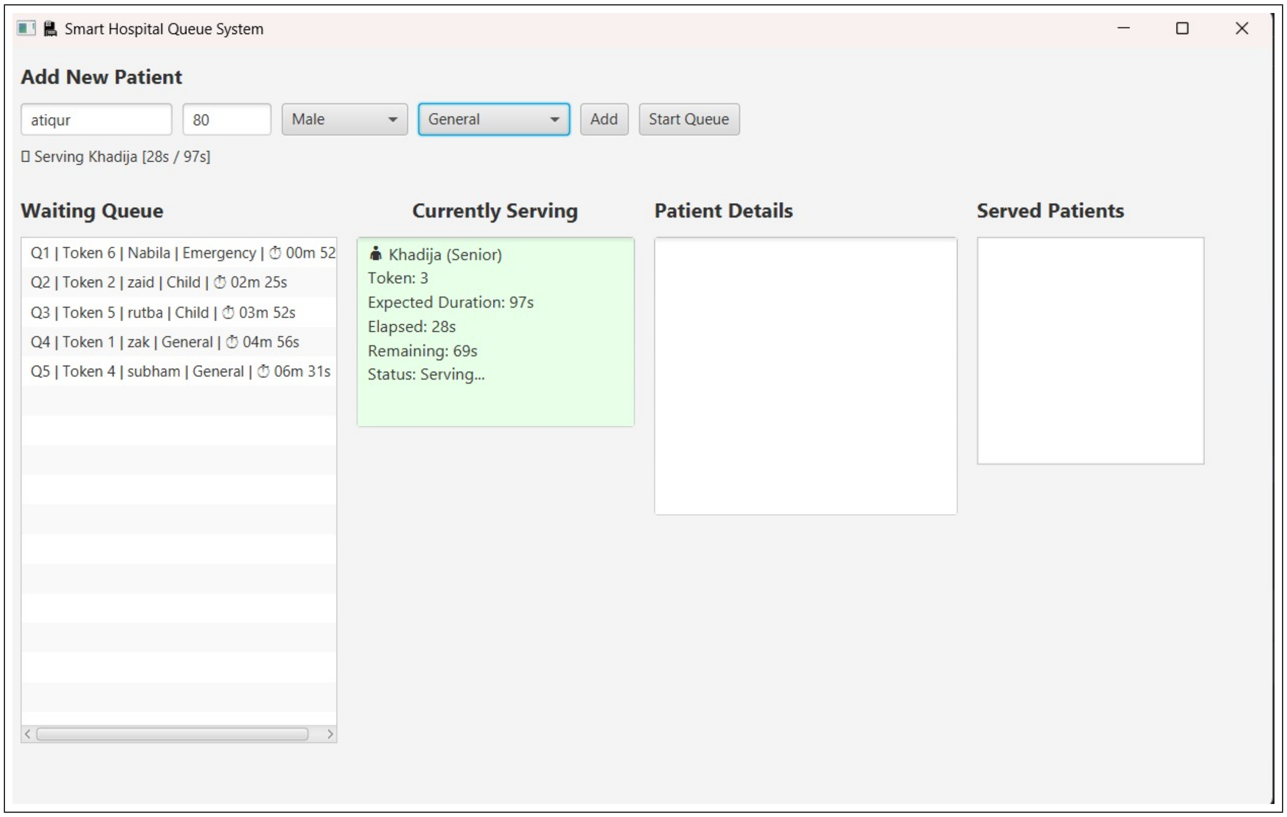
The implementation also provides persistent logging at `E:\downloads\smartqueuesystem\data\queue_log.txt` , ensuring transparency, reproducibility, and detailed analysis of runtime operations. The log structure can serve as a basis for future data analytics modules, enabling insights into patient flow, average service times, and bottleneck detection.

Overall, the project effectively bridges theoretical data structure principles with practical software engineering. The modular design, real-time concurrency, and accuracy of timing synchronization make the SQMS a reliable simulation framework

```
19:12:07 - Started serving Khadija [Expected=97s, Actual=85s]
19:27:52 -        Patient Added: Md Zakiur Rahman (General)
   Token: 1 | Queue: 0 | Expected: 97s
19:28:02 -        Patient Added: Subham (General)
   Token: 2 | Queue: 0 | Expected: 96s
19:28:11 -        Patient Added: Zaid (Child)
   Token: 3 | Queue: 0 | Expected: 66s
19:28:26 -        Patient Added: Khadija (Senior)
   Token: 4 | Queue: 0 | Expected: 66s
19:28:39 -        Patient Added: Zohair (Child)
   Token: 5 | Queue: 0 | Expected: 100s

==========================
       2025-11-04 | Queue Started
==========================
19:28:41 -        Serving started: Khadija (Senior)
   Token: 4 | Queue: 1 | Expected: 66s
19:28:56 -        Patient Added: Smaran (Child)
   Token: 6 | Queue: 0 | Expected: 82s
19:29:09 -        Patient Added: Nabila (Emergency)
   Token: 7 | Queue: 0 | Expected: 69s
19:29:27 -     Served completed: Khadija (Senior)
   Token: 4 | Queue: 1 | Expected: 66s
19:29:27 -        Serving started: Nabila (Emergency)
   Token: 7 | Queue: 1 | Expected: 69s
19:30:37 -     +10s delay added due to overrun by Nabila
19:30:47 -     +10s delay added due to overrun by Nabila
19:30:53 -     Served completed: Nabila (Emergency)
   Token: 7 | Queue: 1 | Expected: 69s
19:30:53 -        -3s deducted (early finish correction by Nabila)
... (log continues)
==========================
       2025-11-05 | Queue Completed
==========================
```

Fig. 4. Comprehensive queue log output illustrating patient addition, serving sequence, delay propagation, and early finish adjustments across multiple sessions.



atiqur | 80 | Male | General

□ Serving Khadija [28s / 97s]

Q1 | Token 6 | Nabila | Emergency | ⏱ 00m 52
Q2 | Token 2 | zaid | Child | ⏱ 02m 25s
Q3 | Token 5 | rutba | Child | ⏱ 03m 52s
Q4 | Token 1 | zak | General | ⏱ 04m 56s
Q5 | Token 4 | subham | General | ⏱ 06m 31s

Khadija (Senior)
Token: 3
Expected Duration: 97s
Elapsed: 28s
Remaining: 69s
Status: Serving...

Fig. 5. Graphical User Interface (GUI) showing real-time patient queue updates, serving details, and automatic synchronization between waiting and served lists.

for real-world queue environments, such as hospitals, banks, or service centers. This integration of algorithmic design with user-centric visualization represents a strong demonstration of how core DSA concepts can evolve into deployable intelligent systems.

Acknowledgment

The authors would like to thank the Department of Computer Science and Engineering, Symbiosis Institute of Technology, Hyderabad, for providing the resources and guidance necessary for completing this project.

References

[1] S. Tanenbaum, *Modern Operating Systems*. Pearson, 2018.

[2] E. Horowitz, S. Sahni, and D. Mehta, *Fundamentals of Data Structures in C++*. Silicon Press, 2007.

[3] Oracle Java Documentation. Available: https://docs.oracle.com/javase/

[4] GeeksforGeeks, "Priority Queue in Java." Available: https://www.geeksforgeeks.org/priority-queue-class-in-java/