Data Structures and Algorithms Submission Report: Priority-Driven Ticketing

1. Problem-Solution Fit: The Need for Prioritization

Problem Statement

In high-volume customer support environments, critical issues (e.g., system outages) often get buried beneath numerous routine or low-priority requests. Staff require a dynamic, real-time queue that eliminates manual sorting and guarantees that the most urgent tasks are presented first, thereby minimizing business risk and meeting service level agreements (SLAs).

Solution: Justification of Priority Queue (PQ)

The **Priority Queue** data structure is the optimal choice for this problem as it inherently maintains a collection of elements prioritized by a key metric (ticket severity).

- **Justification:** The PQ ensures $O(1)$ access to the highest-priority element (the next ticket the staff should process). This contrasts sharply with simple list structures which would require $O(n)$ search time or $O(nlogn)$ sorting every time a new element is added.

2. Priority Queue Implementation Details

The Priority Queue logic is implemented in **JavaScript** on the Staff Dashboard to process data fetched from Firebase Firestore.

Sorting Logic

The prioritization utilizes a two-tiered comparison to ensure both urgency and fairness (First-In, First-Out).

Tier	Priority Metric	Rule	Data Source

Primary	**Ticket Severity**	Highest priority is addressed first.	`priority` field (Mapped: High (3) > Medium (2) > Low (1)).
Secondary	**Submission Time (FIFO)**	If priorities are equal, the oldest ticket is addressed first.	`created_at` field (Timestamp).

Pseudocode for Ticket Comparison

The core sorting algorithm utilizes an array sort with a custom comparator function that implements the PQ logic:

```
// Comparator Function (A - B)
function compareTickets(A, B) {
   // 1. Primary Sort: Priority (Descending Order: High (3) is greater than Low (1))
   if (A.priorityValue !== B.priorityValue) {
      return B.priorityValue - A.priorityValue;
   }
   // 2. Secondary Sort: Timestamp (Ascending Order: Older time is smaller value)
   return A.created_at.seconds - B.created_at.seconds;
}
```

3. Efficiency & Optimization

Time and Space Complexity

The implementation currently uses the built-in JavaScript `Array.prototype.sort()` method on the collection of tickets fetched from Firestore, effectively simulating the PQ output.

Operation	**Implementation**	**Time Complexity**	**Notes on Efficiency**

Enqueue	`array.push()` followed by `array.sort()`.	O(nlogn)	While O(1) for simple insertion, the full sort dominates the complexity.
Dequeue	`array[0]` access.	O(1)	Retrieval of the most urgent ticket is instantaneous.
Space Complexity	Array storage for all fetched tickets.	O(n)	Space requirement scales linearly with the number of tickets.

Note on Optimality

While the current approach (Array + Sort) provides correct prioritization, for very large datasets ($n \gg 1000$), a pure heap-based Priority Queue implementation would yield O(logn) for Enqueue, offering better performance scalability than O(nlogn).

4. Implementation Accuracy and Innovation

Implementation Accuracy

- **Correctness:** The custom comparison function accurately applies both priority and FIFO rules to generate a valid priority-sorted queue.
- **Data Integrity:** The use of Firebase `Timestamp` for `created_at` ensures high-precision, non-client-manipulable data necessary for the accurate FIFO tie-breaker logic.

Innovation

- **Real-World Application:** The project demonstrates the practical application of the Priority Queue to a ubiquitous business problem—resource allocation and urgent task management—thereby optimizing staff operational flow in a real-time environment.
- **Integration with Web Services:** The seamless integration of a DSA concept (PQ) with a real-time database (Firestore) illustrates a modern, service-oriented approach to data processing and display.

5. Summary of Deliverables

The **Data Structures and Algorithms** component is delivered via:

1. **Code:** JavaScript implementation of the `sortTickets` function on the `staff_dashboard.html` file, demonstrating the PQ logic.
2. **Report:** This document provides the formal explanation and complexity analysis.
3. **Demo:** A running demo showcasing the automatic reordering of the staff queue when tickets of varying priorities are submitted in quick succession.