RV College of Engineering®
Mysore Road, RV Vidyaniketan Post,
Bengaluru - 560059, Karnataka, India

DEPARTMENT OF INFORMATION SCIENCE AND ENGINEERING

Data Structures and Applications – IS233AI

Experiential Learning (Lab)

REPORT

on

Smart Online Assessment & Analytics Platform

Submitted by

Nishchith S	**1RV24IS085**
S Mohith Aradhya	**1RV24IS102**

Under the guidance of

Prof. Rekha B. S.

Assistant Professor, Department of Information Science and Engineering

In partial fulfilment for the award of degree of

Bachelor of Engineering

in

Department of Information Science and Engineering

2025-2026

RV College of Engineering®

Mysore Road, RV Vidyaniketan Post,Bengaluru - 560059, Karnataka, India

CERTIFICATE

Certified that the project work titled **Smart Online Assessment & Analytics Platform** is carried out by **Nishchith S (1RV24IS085)** and **S Mohith Aradhya (1RV24IS102)** who are bonafide students of RV College of Engineering, Bengaluru, in partial fulfilment for the award of degree of **Bachelor of Engineering in Information Science and Engineering** of the **Visvesvaraya Technological University**, Belagavi during the academic year 2025-2026. It is certified that all corrections/suggestions indicated for the Internal Assessment have been incorporated in the report deposited in the departmental library. The report has been approved as it satisfies the academic requirements in respect of experiential learning work prescribed by the institution for the said degree.

Signature of Guide	**Signature of Head of the Department**
Prof. Rekha B. S.	**Dr. G S Mamatha**

External Viva

	Name of Examiners	**Signature with Date**
1		
2		

RV College of Engineering®

Mysore Road, RV Vidyaniketan Post,Bengaluru - 560059, Karnataka, India

DECLARATION

We , **Nishchith S and S Mohith Aradhya,** students of Third semester B.E., department of Information Science Engineering, RV College of Engineering, Bengaluru, hereby declare that Experiential Learning (Lab) titled ***'Smart Online Assessment & Analytics Platform'*** has been carried out by us and submitted in partial fulfilment for the award of degree of **Bachelor of Engineering** in **Information Science and Engineering** during the academic year 2025-26

We also declare that any Intellectual Property Rights generated out of this project carried out at RVCE will be the property of RV College of Engineering®, Bengaluru and we will be one of the authors of the same.

Place: Bengaluru

Date:

Name	Signature
1. Nishchith S (1RV24IS085)	
2. S Mohith Aradhya (1RV24IS102)	

ACKNOWLEDGEMENT

It needs more than these few words to express the immense gratitude and profound thanks to the people given all these valuable comments, suggestions and encouragements making cruise smoothly, along the path and reward the effort with success.

I would like to express sincere gratitude to my mentor, Prof. Rekha B. S., Assistant Professor, Department of Information Science & Engineering, RV College of Engineering®, Bengaluru, for her valuable guidance, expert review and her encouragement in choosing this domain and her constant support for the project.

I would also like to express my sincere gratitude to Dr. G.S. Mamatha, Professor and Head of the Department, Information Science & Engineering, RV College of Engineering®, Bengaluru, for the encouragement and support for the project.

I thank Dr. K N Subramanya, Principal, RV College of Engineering®, Bengaluru, for his valuable inputs for

the project and for his moral support.

I also thank my family members for their constant support and encouragement.

Lastly, I would like to thank my professors, staff of ISE Department who gave and their valuable suggestions in the implementation of my project.

Nishchith S and S Mohith Aradhya

Information Science & Engineering

RV College of Engineering,

Bengaluru-59

ABSTRACT

AssessIQ is a Data Structures and Algorithms (DSA)–centric online assessment and analytics platform developed to manage test execution, student evaluation, and performance analysis with strong algorithmic guarantees. Modern online assessment systems often prioritize user interface and backend frameworks, while relying on ad-hoc logic for managing attempts, submissions, and analytics. Such approaches lead to scalability issues, race conditions, inconsistent dashboards, and inefficient computation as the number of users increases. This project addresses these limitations by designing the entire assessment lifecycle around classical data structures and well-defined algorithms rather than treating DSA as an auxiliary component.

The core functionality of AssessIQ is built upon fundamental data structures including hash maps, sets, arrays, stacks, queues, and heaps. Hash maps are used extensively for constant-time lookup of users, test attempts, roles, and question metadata, significantly reducing database dependency and request latency. Sets enforce uniqueness constraints such as single-attempt policies, prevent duplicate submissions, track answered and review-flagged questions, and manage submission lock tokens in constant time. Arrays provide an efficient and cache-friendly structure for storing question banks, multiple-choice options, and student responses, enabling fast sequential evaluation and scoring.

Stacks are employed to implement reversible question navigation and review workflows, allowing students to move backward and forward through questions with O(1) time complexity per operation. Queues are used to serialize concurrent test submissions and evaluation requests, ensuring fairness, eliminating race conditions, and decoupling submission intake from background evaluation. Heaps (priority queues) support efficient leaderboard generation and real-time ranking by maintaining top-K results without requiring expensive full sorting operations. Each algorithm is carefully analyzed with respect to time and space complexity to justify the choice of data structure and ensure scalability.

The backend is implemented using Supabase with PostgreSQL, where relational constraints, row-level security, and B-tree indexing complement the in-memory DSA logic to preserve data integrity

and consistency. A single source of truth is maintained through the test_attempts table, from which all dashboards and analytics are derived. By combining classical data structures with practical system design principles, AssessIQ demonstrates how DSA concepts can be applied to solve real-world engineering problems in online assessment platforms. The project serves as a strong academic example of algorithm-driven system design suitable for technical evaluation, university project submissions, and scalable deployment.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Online assessment systems have become a critical component of modern education, enabling scalable evaluation, performance tracking, and analytics for large numbers of learners. However, many assessment platforms are designed with a strong emphasis on user interface and backend frameworks while relying on ad-hoc or naïve logic to manage test attempts, submissions, and result computation. As the number of users and concurrent interactions grows, such systems often suffer from duplicate submissions, race conditions, inconsistent dashboards, and inefficient analytics. These issues arise primarily due to the absence of well-structured data management strategies and the improper use of fundamental data structures, leading to performance bottlenecks and reduced system reliability.

This project addresses these challenges by designing and implementing **AssessIQ**, a data structures and algorithms–centric online assessment and analytics platform. Rather than treating DSA concepts as an afterthought, the system is engineered around core data structures that govern every stage of the assessment lifecycle, including attempt creation, question navigation, submission handling, evaluation, and result visualization. The focus of the project is not on developing a complex web application, but on demonstrating how classical data structures can be applied to solve real-world problems in assessment systems with clear correctness and performance guarantees.

The primary objective of the project is to design and implement an assessment pipeline that:

● Ensures single-attempt enforcement and duplicate prevention using set-based membership checks.

● Manages efficient question traversal and reversible navigation through stack-based history tracking.

● Serializes concurrent submissions and evaluations using queue-based processing to avoid race conditions.

● Supports fast evaluation, aggregation, and leaderboard generation using arrays, hash maps, and heaps.

From a DSA perspective, the project emphasizes how deliberate choices—such as using hash maps for constant-time user and attempt lookups, sets for enforcing uniqueness constraints, stacks for reversible navigation, queues for fair submission processing, and heaps for efficient ranking—directly influence system correctness, time complexity, and scalability. The backend is implemented using Supabase with PostgreSQL, where indexed queries and relational constraints complement the in-memory data structures to preserve data integrity. The scope of the system is intentionally focused on educational use, targeting small to medium-scale assessments, and is treated as a learning-oriented prototype that demonstrates the practical application of data structures and algorithms in a realistic engineering context rather than a commercial-grade product.

.

CHAPTER 2

SYSTEM DESIGN

The overall system follows a **modular, data-structures-driven architecture** with three conceptual phases: **attempt initialization, submission and evaluation processing, and analytics and visualization**. Each phase is designed around well-defined data structures and algorithms to ensure correctness, scalability, and synchronization. The backend is implemented using **JavaScript with Supabase (PostgreSQL + Row Level Security)**, while the frontend is built using **HTML, CSS, and Vanilla JavaScript** to orchestrate test navigation, submission, and dashboard rendering.

System Architecture

At a high level, the system is divided into the following components:

● **A backend service that:**

- Manages authentication and role-based access control.
- Validates and initializes test attempts with single-attempt enforcement.
- Stores submissions and evaluation results in a centralized test_attempts table.
- Provides indexed queries for dashboards and analytics.
- Maintains consistency using relational constraints and transaction safety.

● **A client-side assessment engine that:**

- Loads question banks and metadata.
- Manages in-test navigation, progress tracking, and review states.
- Serializes submissions and communicates with the backend.
- Renders student and teacher dashboards with real-time updates.

The separation between assessment logic (attempts, evaluation, analytics) and presentation logic (UI rendering) ensures modularity, maintainability, and clarity of system responsibilities.

Backend Module Organization

The backend logic is organized conceptually as:

- **Attempt_Manager**
  - Validates test start requests.
  - Enforces single-attempt rules using set-based checks and database uniqueness constraints.
  - Issues and manages submission lock tokens.
- **Evaluation_Engine**
  - Processes queued submissions.
  - Computes scores, accuracy, and time taken.
  - Updates the test_attempts table as the single source of truth.
- **Analytics_Service**
  - Retrieves attempt data using indexed queries.
  - Computes aggregates such as averages, completion rates, and rankings.
  - Supports leaderboard generation using heap-based ranking logic.
- **Access_Control**
  - Enforces role-based permissions using Supabase RLS policies.
  - Ensures students and teachers access only authorized data.

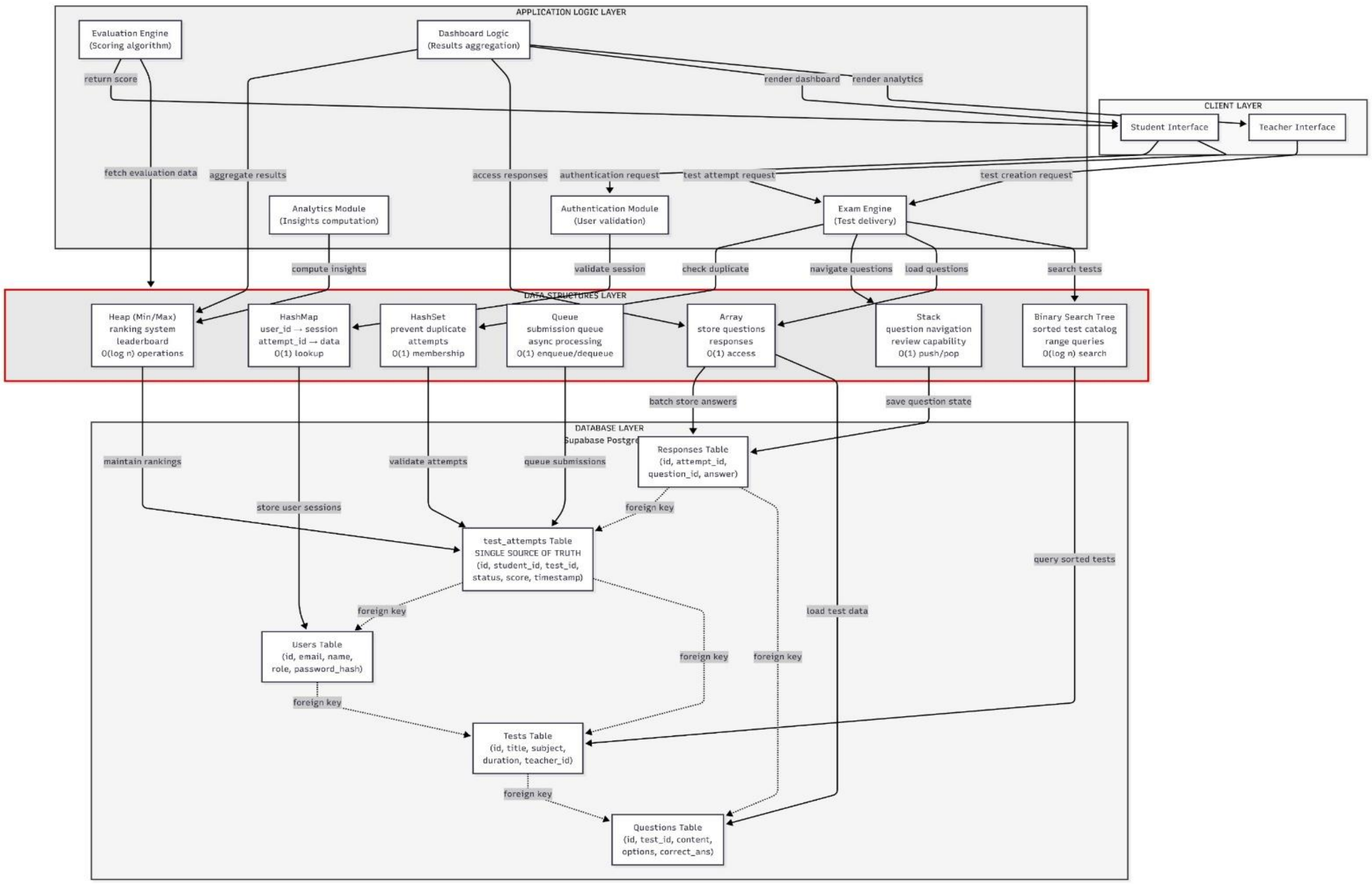
APPLICATION LOGIC LAYER
Evaluation Engine (Scoring algorithm)
Dashboard Logic (Results aggregation)
return score
render dashboard
render analytics
CLIENT LAYER
Student Interface
Teacher Interface
fetch evaluation data
aggregate results
access responses
authentication request
test attempt request
test creation request
Analytics Module (Insights computation)
Authentication Module (User validation)
Exam Engine (Test delivery)
compute insights
validate session
check duplicate
navigate questions
load questions
search tests
DATA STRUCTURES LAYER
Heap (Min/Max) ranking system leaderboard O(log n) operations
HashMap user_id → session attempt_id → data O(1) lookup
HashSet prevent duplicate attempts O(1) membership
Queue submission queue async processing O(1) enqueue/dequeue
Array store questions responses O(1) access
Stack question navigation review capability O(1) push/pop
Binary Search Tree sorted test catalog range queries O(log n) search
batch store answers
save question state
DATABASE LAYER
Supabase Postgre
Responses Table (id, attempt_id, question_id, answer)
maintain rankings
validate attempts
queue submissions
store user sessions
foreign key
test_attempts Table SINGLE SOURCE OF TRUTH (id, student_id, test_id, status, score, timestamp)
query sorted tests
load test data
Users Table (id, email, name, role, password_hash)
foreign key
foreign key
foreign key
foreign key
Tests Table (id, title, subject, duration, teacher_id)
foreign key
Questions Table (id, test_id, content, options, correct_ans)

Fig.1 System Architechture

Data Structures and Operations

The system is intentionally centered around a small set of fundamental data structures.

1. Hash Maps for Fast Lookup

Hash maps are used on the client side to cache user profiles, role information, question metadata, and attempt states. These structures enable constant-time lookup for authentication checks, question retrieval, and evaluation logic, significantly reducing repeated database access.

2. Sets for Uniqueness and State Tracking

Sets are used to:

- Prevent duplicate attempts by storing (studentId, testId) pairs.
- Track answered and review-flagged questions during a test.
- Manage active submission lock tokens.

Set membership checks ensure O(1) validation and eliminate costly linear scans or redundant database queries.

3. Arrays for Sequential Processing

Arrays store:

- Question banks and options.
- Student responses during test attempts.
- Batches of attempt records fetched from the database.

Arrays enable efficient index-based access and cache-friendly sequential traversal during evaluation and aggregation.

4. Stacks for Navigation and Reversibility

Stacks implement reversible question navigation and review workflows. Each navigation action pushes the current question onto the stack, while backtracking operations pop from the stack. This structure ensures O(1) navigation operations and preserves the student's traversal context without recomputation.

5. Queues for Submission and Evaluation Serialization

Queues are used to serialize:

- Test submissions from multiple students.
- Background evaluation and result processing tasks.

A FIFO queue ensures fairness, avoids race conditions, and decouples submission intake from evaluation logic. This design prevents concurrent writes and maintains predictable system behavior under load.

6. Heaps for Ranking and Leaderboards

Heaps (priority queues) are used to generate leaderboards by ranking students based on score and submission time. Heap-based top-K extraction avoids expensive full sorting operations and supports efficient real-time updates as new results arrive.

Conceptual Flow of the System

1. **Attempt Initialization**
   - Student requests to start a test.
   - System checks set membership and database constraints.
   - Lock token is issued and attempt state is initialized.
2. **In-Test Interaction**
   - Questions are navigated using stack-based history.
   - Answered and review-marked questions are tracked using sets.
   - Progress is computed in constant time.
3. **Submission and Evaluation**
   - Submission is enqueued for processing.
   - Evaluation engine dequeues and processes responses.
   - Results are stored in the test_attempts table.
4. **Analytics and Dashboard Rendering**
   - Dashboards query indexed attempt data.
   - Aggregates and leaderboards are computed using arrays and heaps.
   - Results are rendered consistently from a single source of truth.

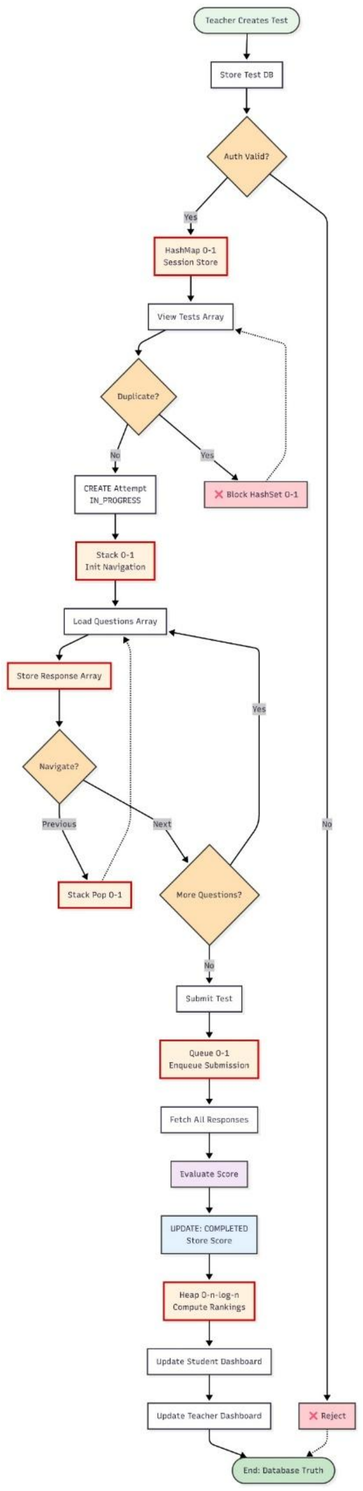

Fig.2: Methodology

CHAPTER 3

IMPLEMENTATION DETAILS

The AssessIQ system is engineered as a decoupled, modular web application consisting of a client-side assessment engine and a backend service built on Supabase. The design prioritizes algorithmic correctness, modularity, and maintainability, with every critical workflow explicitly driven by classical data structures and algorithms. The architecture separates attempt management, evaluation, and analytics into independent logical units, ensuring clarity of responsibility and ease of extension.

1. Backend Architecture (Logic Layer)

The backend acts as the authoritative computation and persistence layer, handling attempt validation, submission storage, evaluation results, and analytics queries. It is designed around a single-source-of-truth model using PostgreSQL, with data-structure-driven logic applied both in memory and at the database level.

A. Attempt Initialization and Validation

Each test attempt is validated using a combination of in-memory set membership checks and database-level uniqueness constraints. The (studentId, testId) pair is verified against an active attempts set to prevent duplicate or concurrent attempts in constant time. Upon successful validation, a unique lock token is generated and stored in an active lock set to ensure idempotent submissions. All validation operations execute in O(1) average time using hash maps and sets, minimizing database overhead.

B. Submission Handling and Evaluation Pipeline

Student submissions are serialized using a FIFO queue to prevent race conditions and ensure fair processing. Each submission is dequeued and evaluated independently using a single-pass array traversal over student responses. Question metadata is retrieved through hash maps, enabling constant-time access to correct answers during evaluation. Computed results such as score, accuracy, and time taken are persisted in the test_attempts table, which serves as the only authoritative data source for analytics and dashboards.

C. Analytics and Dashboard Query Layer

Dashboards retrieve attempt data through indexed queries on student and test identifiers, ensuring logarithmic query performance. Leaderboards are generated using heap-based priority queues to efficiently extract top-performing students without full dataset sorting. Aggregated metrics such as averages and completion rates are computed on demand using array traversal, eliminating stale cached values and ensuring consistency.

2. Frontend Architecture (Assessment Engine)

The frontend operates as an orchestration layer for DSA-driven workflows, managing test interaction, navigation, submission, and visualization. It communicates securely with the backend via Supabase APIs while maintaining responsive user interaction.

A. In-Test Navigation and State Management

Question navigation history is implemented using a stack, allowing constant-time backward and forward traversal. Answered and review-flagged questions are stored in sets to enable instant membership checks and progress calculation. Question banks and answer options are maintained in arrays, providing efficient index-based access and sequential rendering.

B. Submission Orchestration

Before submission, the frontend verifies the presence of a valid lock token to prevent duplicate requests. Submissions are dispatched asynchronously to avoid blocking the user interface. Immediate confirmation is provided to the student while evaluation proceeds independently in the backend.

C. Dashboard Rendering and Visualization

Student dashboards display active tests, completed attempts, and performance summaries derived directly from backend query results. Teacher dashboards present class-level analytics and leaderboards computed using heap-based ranking logic. Visualization components render analytical summaries without embedding business logic in the UI.

3. Security, Integrity, and Configuration

Row-Level Security (RLS) policies enforce role-based access control, ensuring that students and teachers access only authorized records. Database constraints such as unique keys and foreign keys provide an additional safety layer against duplicate attempts and orphaned data. Scoring rules, time limits, and evaluation parameters are configuration-driven, allowing system reuse without modifying core logic.

4. Design Rationale

From an implementation perspective, AssessIQ demonstrates how classical data structures directly influence system behavior. Hash maps enable fast lookup, sets enforce correctness, stacks support reversible navigation, queues serialize concurrency, and heaps provide scalable ranking. The system is intentionally scoped as an educational prototype, emphasizing clarity, correctness, and algorithmic rigor over framework complexity, making it suitable for academic evaluation and technical demonstration.

CHAPTER 4
RESULTS AND DISCUSSION

Given the educational scope and prototype-oriented design of AssessIQ, evaluation emphasizes correctness of data-structure-driven workflows, handling of edge cases, and qualitative system reliability rather than large-scale stress benchmarking. The results demonstrate that deliberate use of classical data structures successfully enforces correctness, prevents race conditions, and enables consistent analytics, while also revealing areas where performance optimization is required.

Execution Outcomes and Correctness

On small to medium-scale test environments involving approximately 50–300 students and tests containing 10–50 questions, the system consistently enforces single-attempt constraints using set-based membership checks combined with database-level uniqueness constraints. Concurrent duplicate submissions are prevented through lock-token validation and queue-based serialization, ensuring that each attempt is evaluated exactly once. Stack-based navigation enables smooth backward and forward traversal of questions in constant time, while array-based evaluation guarantees that each response is processed exactly once. Leaderboard correctness is validated by comparing heap-based top-K rankings with full array sorting, producing identical ordering for ranked results. Dashboard views remain consistent because all metrics are derived exclusively from the test_attempts table, which acts as the single source of truth.

Handling Edge Cases

The system design explicitly addresses several critical edge cases. Concurrent submissions arriving within a short time window are serialized using a FIFO queue, preventing race conditions and inconsistent writes. Duplicate attempts are blocked at both the application level using sets and at the database level using unique constraints, providing layered protection. Partial or incomplete submissions are handled gracefully, with unanswered questions excluded from scoring without affecting evaluation stability. Tests with larger question sets continue to evaluate predictably due to linear array traversal. These scenarios demonstrate effective application of DSA principles, where sets enforce uniqueness, queues serialize access, and arrays provide stable time complexity.

Performance Observations

Performance remains stable for typical academic usage. Attempt validation executes in constant time using hash maps and sets, while evaluation complexity grows linearly with the number of questions, which is acceptable for standard assessments. Heap-based leaderboard generation avoids repeated full sorting and enables efficient top-K extraction. However, dashboard refresh operations that aggregate large numbers of attempts introduce moderate overhead, indicating potential benefits from incremental aggregation or selective caching in future versions.

Comparison with Naïve Alternatives

Compared to naïve implementations, the DSA-driven approach provides clear advantages. Linear scans for detecting duplicate attempts are replaced by constant-time set membership checks. Full dataset sorting for ranking is avoided through heap-based priority queues. Direct concurrent writes are replaced with queue-based serialization, eliminating race conditions. Stack-based navigation provides efficient and intuitive backtracking compared to recomputing navigation paths on each action. These comparisons confirm that the chosen data structures significantly improve correctness and efficiency.

Potential Improvements

While the system meets its educational objectives, further enhancements can improve scalability and responsiveness. Incremental leaderboard updates could reduce recomputation overhead, background evaluation workers could increase throughput during peak submission periods, and adaptive caching strategies could improve dashboard performance for large cohorts. Overall, the results confirm that AssessIQ effectively demonstrates the practical application of classical data structures and algorithms in an online assessment system. The project achieves a strong balance between correctness, clarity, and performance, making it suitable for academic evaluation and DSA-focused system design demonstrations.

CHAPTER 5
CONCLUSION

This project demonstrates how classical data structures and algorithms can be effectively applied to the design of a robust online assessment and analytics system. By structuring the entire assessment lifecycle around fundamental DSA concepts—such as hash maps for fast lookup, sets for uniqueness enforcement, stacks for reversible navigation, queues for serialized submission handling, arrays for efficient evaluation, and heaps for ranking—the system achieves correctness, consistency, and predictable performance without relying on ad-hoc logic.

The effectiveness of AssessIQ is well-aligned with its intended academic scope. The system successfully enforces single-attempt constraints, prevents race conditions during concurrent submissions, supports intuitive question navigation, and produces consistent dashboards derived from a single source of truth. These outcomes highlight how careful data structure selection directly influences system reliability and scalability. While the platform is not optimized for enterprise-scale deployments, it meaningfully demonstrates how DSA principles translate into real-world engineering decisions rather than remaining theoretical constructs.

In terms of practical applicability, such a system is particularly useful for:
● Educational institutions conducting small to medium-scale online assessments.
● Academic environments where correctness, transparency, and explainability are more important than UI Complexity
● Teaching scenarios where students are required to demonstrate the application of data structures and algorithms in realistic system design problems.

Future work and potential enhancements include optimizing leaderboard updates through incremental heap maintenance, introducing background evaluation workers for higher throughput, and adding richer analytics such as percentile-based performance analysis. Additional improvements may include adaptive caching strategies for dashboards and support for larger concurrent user loads. Overall, the project serves as a strong bridge between core DSA coursework and applied system design, illustrating that fundamental structures such as maps, sets, stacks, queues, and heaps remain central to building reliable and scalable modern software systems.

REFERENCES

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022.

[2] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Boston, MA, USA: Addison-Wesley, 2011.

[3] M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, *Data Structures and Algorithms in Java*, 6th ed. Hoboken, NJ, USA: Wiley, 2014.

[4] S. Sahni, *Data Structures, Algorithms, and Applications in Java*, 2nd ed. New York, NY, USA: McGraw-Hill, 2014.

[5] A. Silberschatz, H. F. Korth, and S. Sudarshan, *Database System Concepts*, 7th ed. New York, NY, USA: McGraw-Hill, 2019.

[6] H. Garcia-Molina, J. D. Ullman, and J. Widom, *Database Systems: The Complete Book*, 2nd ed. Upper Saddle River, NJ, USA: Pearson, 2008.

[7] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*, 2nd ed. San Francisco, CA, USA: Morgan Kaufmann, 2012.

[8] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[9] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. San Francisco, CA, USA: Morgan Kaufmann, 1993.

[10] D. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, 3rd ed. Boston, MA, USA: Addison-Wesley, 1997.