DSA REPORT

To Sir Umer Usman

JUNE 15, 2025

Table of Contents

Chat Application Project Report

1. Project Description

This project delivers a console-based chat application developed in Java, mimicking core functionalities of popular messaging platforms. It enables multiple users to sign up, log in, exchange text messages, view chat histories, and manage their accounts. A central design principle was the effective application of various fundamental Data Structures and Algorithms (DSAs) to ensure efficient data storage, retrieval, and manipulation, with data persistence managed through text files.

2. Data Structures Used

The application extensively utilizes the following data structures:

- **HashMap<String, User>**: In UserDatabase for storing all user accounts.
- **HashMap<String, Queue<Message>>**: Within the User class, for managing messages sent to and received from specific chat partners.
- **Queue<Message>**: Implemented using LinkedList, for storing messages in chronological (FIFO) order within individual chats.
- **LinkedList<String>**: In UserDatabase for maintaining a history of recently logged-in usernames.
- **Stack<Message>**: Used temporarily in Message to retrieve messages in reverse chronological (LIFO) order.
- **BinarySearchTree (Custom Implementation)**: In UserDatabase for storing usernames to facilitate alphabetical sorting.

3. Why Specifically These Data Structures for Specific Purposes

The selection of each data structure was driven by specific functional and performance requirements:

- **HashMap for userDatabase (UserDatabase)**:
    - **Purpose:** To store and retrieve User objects by their username (which acts as a unique key).

    - **Justification:** HashMap provides near O(1) average-case time complexity for put, get, and containsKey operations. This is crucial for rapid user authentication during login and efficient lookup of message recipients, ensuring a responsive user experience even with many users.
- **HashMap of Queues for sentMessages and receivedMessages (User)**:
    - **Purpose:** To organize chat messages, where each key represents a unique chat partner, and the value is a sequence of messages exchanged with that partner.
    - **Justification:** The outer HashMap allows for quick O(1) average-case access to a specific conversation (e.g., all messages with "Alice"). The inner Queue (implemented by LinkedList) is ideal for storing messages in First-In, First-Out (FIFO) order. This naturally preserves the chronological flow of a conversation, as messages are added to the end and retrieved from the front, reflecting how conversations unfold. LinkedList specifically offers efficient O(1) insertion and deletion at both ends.
- **LinkedList for recentLogins (UserDatabase)**:
    - **Purpose:** To maintain a small, fixed-size history of the most recently logged-in users.
    - **Justification:** LinkedList is chosen because it excels at efficient O(1) insertions at the tail (addLast) and deletions from the head (removeFirst). This perfectly suits the requirement of adding new logins to the "most recent" end and easily removing the "oldest" login when the list exceeds its maximum size.
- **Stack for getMessagesReverseChronological (Message)**:
    - **Purpose:** To display a specific number of messages in reverse chronological order (newest first).
    - **Justification:** After retrieving messages in chronological order (using Queue logic), pushing them onto a Stack and then popping them achieves Last-In, First-Out (LIFO) behavior. This means the last

    message pushed (the most recent) will be the first one popped, effectively reversing the order for display. It's a simple and direct way to achieve this specific display requirement.

- **BinarySearchTree for usernameBST (UserDatabase)**:
    - **Purpose:** To store all usernames in a sorted manner and retrieve them alphabetically.
    - **Justification:** A Binary Search Tree provides efficient O(log N) average-case time complexity for insertion and deletion, and its inorderTraversal() method naturally yields elements in sorted order (O(N) for traversal). This is a clean and explicit way to manage a sorted list of all users, particularly useful for administrative views or search functionalities requiring ordered lists.

4. Applications of Project

This project serves as an excellent foundation and demonstration for several real-world applications and concepts:

- **Learning and Educational Tool:** Provides a clear, practical example of how various core data structures (HashMaps, Queues, Stacks, BSTs) are implemented and used together in a single application.
- **Basic Messaging System Prototype:** Forms the rudimentary backend logic for any messaging application, illustrating user registration, authentication, and chat history management.
- **Foundation for GUI Applications:** The logical structure and data handling components are directly transferable to a graphical user interface (GUI) application, requiring only a new UI layer (e.g., Swing, JavaFX).
- **Introduction to Data Persistence:** Demonstrates a simple method of saving application data to files, a critical step for any software that needs to retain information between sessions.
- **Backend for Simple Social Platforms:** The user management and messaging core could be extended to support profiles, friendships, and posts, forming the basis of a lightweight social network.

5. Scalability

The current console-based chat application, while demonstrating robust DSA principles, has inherent limitations regarding scalability for real-world deployment:

- **In-Memory Data Loading:** At startup, all user accounts and messages are loaded into memory. This becomes a significant bottleneck and performance issue as the number of users and messages grows, leading to high memory consumption and slow startup times.
- **File I/O for Persistence:** Data persistence relies on reading and rewriting entire text files (accounts.txt, Messages.txt) for updates (e.g., deleteUser, rewriteAccountsFile). This is highly inefficient for large datasets, as it involves frequent, time-consuming disk operations.
- **Single-Machine Operation:** The application is designed for single-user, local console interaction. It lacks network capabilities, meaning users cannot chat across different computers or in real-time.
- **No Concurrent Access Handling:** There's no mechanism to handle multiple users attempting to read from or write to the data files simultaneously. In a multi-user environment, this would lead to data corruption.
- **Limited Search Performance:** While HashMap provides quick user lookups, searching messages by keyword requires iterating through all messages, which is O(N) where N is the total number of messages. This is inefficient for large volumes of chat data.

To scale this application, significant architectural changes would be required, moving beyond simple file persistence and single-process execution.

6. Improvements That Can Be Made

Many improvements can enhance the functionality, robustness, and user experience of this application:

- **Graphical User Interface (GUI):** Implement a modern GUI using Swing, JavaFX, or a web framework (e.g., with HTML/CSS/JavaScript and a backend server) to provide a more intuitive and visually appealing user experience.

- **Database Integration:** Replace the current text file persistence with a robust relational database (e.g., MySQL, PostgreSQL, SQL Server) or a NoSQL database (e.g., MongoDB, Firebase). This would offer:
  - **True Scalability:** Efficiently handle large datasets with optimized queries.
  - **Concurrency Control:** Built-in mechanisms for managing simultaneous reads and writes from multiple users.
  - **Data Integrity:** Enforce data types, relationships, and constraints.
  - **Security:** Better mechanisms for data security, backups, and recovery.
- **Network Communication (Real-time Chat):** Implement client-server architecture using Java Sockets or a framework like Netty to enable real-time messaging between users connected over a network.
- **Enhanced Security:**
  - **Stronger Password Hashing:** Use industry-standard, computationally intensive hashing algorithms like BCrypt or Argon2, which incorporate salting and stretching to resist brute-force and rainbow table attacks.
  - **Input Sanitization:** Implement rigorous input validation and sanitization to prevent common vulnerabilities like injection attacks (though less critical for a text-file based console app, it's a good practice).
- **Group Chats:** Extend the messaging logic to allow users to create and participate in group conversations.
- **User Profiles and Status:** Allow users to update more profile details (e.g., display name, profile picture URL, custom status messages).
- **File/Media Sharing:** Implement functionality to send and receive files, images, or other media.
- **Notifications:** Add system notifications for new messages, online status changes of friends, etc.

- **Improved Search Functionality:** Implement inverted indexes or use database-specific full-text search capabilities for faster and more flexible keyword searching across messages.
- **Asynchronous Operations:** For database or network operations, use asynchronous programming models to keep the UI (if GUI is added) responsive.
- **Error Handling and Logging:** Implement a more sophisticated error handling strategy with dedicated logging frameworks (e.g., Log4j, SLF4j) for better debugging and monitoring.
- **Unit and Integration Testing:** Develop comprehensive test suites to ensure the correctness and reliability of all features, especially after modifications.

7. Limitations

The current version of the application has several limitations due to its design choices and scope:

- **No Real-Time Communication:** This is a batch-processing chat application. Messages are exchanged and viewed by reloading data, not in real-time. There is no concept of messages instantly appearing for the receiver.
- **No Network Functionality:** Users can only interact with the application locally on the same machine. It does not support chatting with users on different computers.
- **Limited Concurrency Support:** The file-based persistence model does not inherently support concurrent writes from multiple users, which would lead to data corruption if multiple instances tried to write simultaneously.
- **Vulnerability to Data Loss/Corruption:** Relying solely on plain text files for persistence is prone to issues like data corruption if the application crashes during a write operation, or manual file tampering.
- **Security Concerns (Password Hashing Basic):** While SHA-256 hashing is used, it lacks salting, making it vulnerable to rainbow table attacks. The

application stores plain text passwords in memory (for comparison before hashing), which is not ideal.

- **Poor Scalability:** As discussed in Section 5, the file-based approach and in-memory loading prevent the application from scaling to a large number of users or messages.
- **No Offline Messaging:** Messages are only delivered when both sender and receiver are active within the application's current session or when data is reloaded. There's no concept of messages waiting for an offline user.
- **Basic UI:** The console-based interface is functional but not user-friendly or intuitive for typical chat application users.
- **No Input Validation Beyond Basics:** While some input validation (e.g., password length, email format) is present, comprehensive validation and sanitization against malicious input are not fully implemented.
- **Limited Error Recovery:** Error handling is basic, primarily printing messages to the console or showing JOptionPane dialogs. More robust error recovery mechanisms are absent.