Cambodia Academy of Digital Technology
Institute of Digital Technology

Department: Computer Science
Specialization: Software Engineering

Final Project Report

Course: Advanced Algorithm
Instructor: Mr. Lay Vathna
& Mr. Korat Natt

Topic: Bus Management System

By

Phon Sokleaphea
Ory Chanraksa
Seang Darong
Sao Visal

24th December 2024

This report details the development of a Bus Management System that supports booking, refunding, and seat management. The system leverages key data structures and algorithms from the Advanced Algorithms course to enhance performance and streamline operations.

Table Of Contents

I. Introduction

1.1. Introduction

The Bus Management System is a C++-based application designed to streamline bus reservations and administrative tasks. It enables users to reserve or cancel seats, view booking history, and manage refunds. Administrators can manage admin accounts, oversee buses, and access user data. The system integrates key programming concepts learned throughout the Advanced Algorithms course, including sorting algorithms, password hashing, and efficient data lookup.

The system is built using Object-Oriented Programming (OOP) principles, ensuring modularity and scalability. Data is managed with C++ data structures such as vectors, linked lists, and hashmaps, with storage handled through the nlohmann/json library for persistent file operations. The project emphasizes efficiency, maintainability, and ease of use, with a focus on seamless user and administrative experiences.

This report documents the system's design, implementation, and the challenges faced during development.

1.2. Objectives

The primary objective of this project was to develop an efficient Bus Management System by applying advanced algorithm concepts learned in the course. The system aims to optimize bus reservations, resource allocation, and provide a seamless user and administrative experience.

Key objectives include:

- **User Experience:** Simplifying bus reservations with a user-friendly interface, providing an easy way to refund or cancel bookings, and enabling users to track their booking history.
- **Administrative Operations:** Allowing administrators to manage bus departures, delete user accounts, and add new admin accounts to ensure smooth system operations.
- **Technical Objectives:** Applying advanced algorithms for performance, utilizing Object-Oriented Programming (OOP) principles for modular and maintainable code, ensuring scalability for future growth, and managing data reliably using the nlohmann/json library. Security measures, like password hashing, were implemented to protect user data.

1.3. Tasks

Ory Chanraksa

- Project Planning and Design: Defined system scope, requirements, and architecture with a modular design and JSON-based database schema.
- User Authentication: Established secure user credentials and processed sensitive information.
- View History: Managed the feature to allow users to view their booking history.
- Waitlist: Proposed a feature for users to enlist for a seat on the bus.

Sao Visal

- Reserve: Handled seat reservations functionality.
- Refund: Managed booking cancellations.

Seang Darong

- Add Admin: Enabled the addition of new administrators.
- View Users: Retrieved user information, including names, emails, and reservations.
- Delete User: Allowed administrators to delete user accounts.

Phon Sokleaphea

- Add Bus: Enabled the addition of new buses.
- View Buses: Retrieved bus information.
- Delete Bus: Allow administrators to delete user buses from the system.

II. Implementation

2.1. Object-Oriented Design

System Class

The System Class is responsible for managing user authentication and session handling. It provides login and signup functionalities, which validate user credentials stored in a JSON file. Upon successful authentication, it creates a **User** object that serves as the interface for interacting with the user's actions, including reservations and refunds.

User Class

The User Class represents individual users in the system, distinguishing between regular users and administrators. Regular users can make seat reservations, check booking statuses, and request refunds. Admin users have additional permissions, such as adding new users or buses. The User class manages a user's booking information, enabling actions like canceling or modifying reservations.

Bus Class

The Bus Class is responsible for handling data related to the buses, such as the bus type, seating capacity, and schedules. It allows users to view available buses, check seat

availability, and reserve seats. The class ensures proper seat management, preventing overbooking by restricting seat reservations when the capacity is reached.

Data Flow and Interactions

The System Class is the entry point, responsible for user authentication. Once the user is authenticated, the system creates a **User** object to manage user-specific tasks. The User Class is responsible for making and managing reservations, viewing bookings, and interacting with the bus system. The Bus Class supports these interactions by providing details about bus availability and seat reservations, ensuring smooth operations of the booking system.

2.2. Helper Libraries/Dependencies

Custom Libraries

- **Menu.hpp**: Contains pre-made menus and input stages for decision-making, along with ASCII art components to improve the visual appeal of the Command Line Interface (CLI). It simplifies menu-driven interactions without requiring file input/output or validation.
- **Validation.hpp**: Provides functions to validate user inputs, such as checking if age, name, and email entries are correct. These functions ensure data integrity during program execution.

Third-Party Libraries:

- **json.hpp**: A lightweight header-only library for handling JSON data in C++. It simplifies parsing, creating, and manipulating JSON objects, offering features like serialization and deserialization to manage structured data effectively.
- **sha1.hpp:** Implements the SHA-1 hashing algorithm in C++. It allows the generation of 160-bit hash values, providing methods for processing input data and retrieving the final hash.

Standard C++ Libraries:

The system utilizes several standard C++ libraries:

- **<iostream> and <fstream>**: Handle input and output operations.
- **<stack>**: Manages data structures.

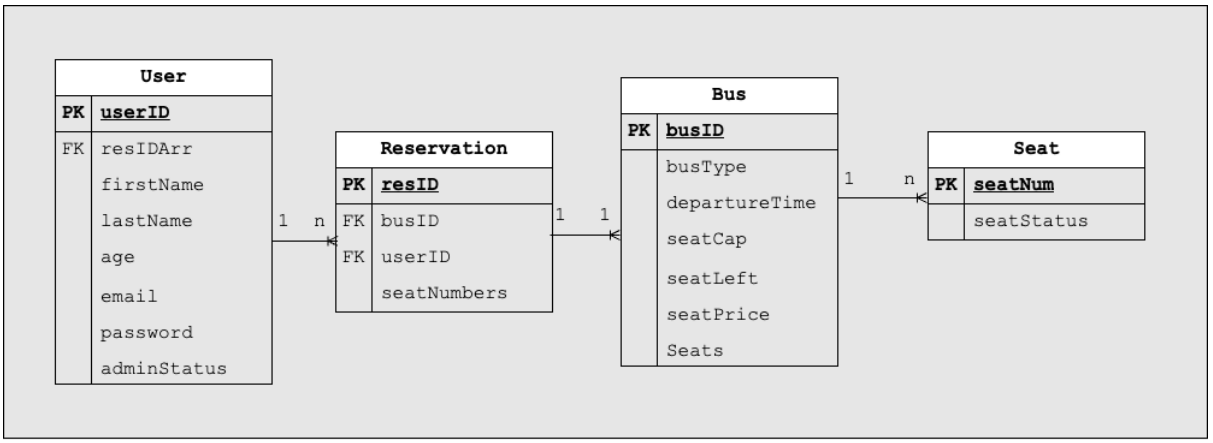
2.3. Database Design

The bus booking system's database is designed to manage key functionalities, including users, buses, routes, and reservations, with data stored in JSON format for easy manipulation. Here's a summary of the high-level components:

Users: Each user has a unique ID, personal details (name, age, email), a hashed password, and associated reservations (resID).
Buses: Each bus entry includes a unique ID, type, route (departure and destination), seat capacity, available seats, seat price, and individual seat status (available or reserved).

Reservations: The reservation system supports both bulk (group bookings) and single reservations. Each record contains the bus ID, reservation ID, seat numbers, and the user ID.
Routes: Each route specifies a starting location and possible destinations.
Relationships:

- **Buses to Routes**: Each bus is linked to a specific route.
- **Buses to Seats**: Buses contain a list of seats with availability status.
- **Users to Reservations**: Users are associated with reservations via the resID attribute.



2.4. Data Structures & Algorithms

For the bus management project, we will use a combination of data structures and algorithms to manage the various functionalities efficiently:

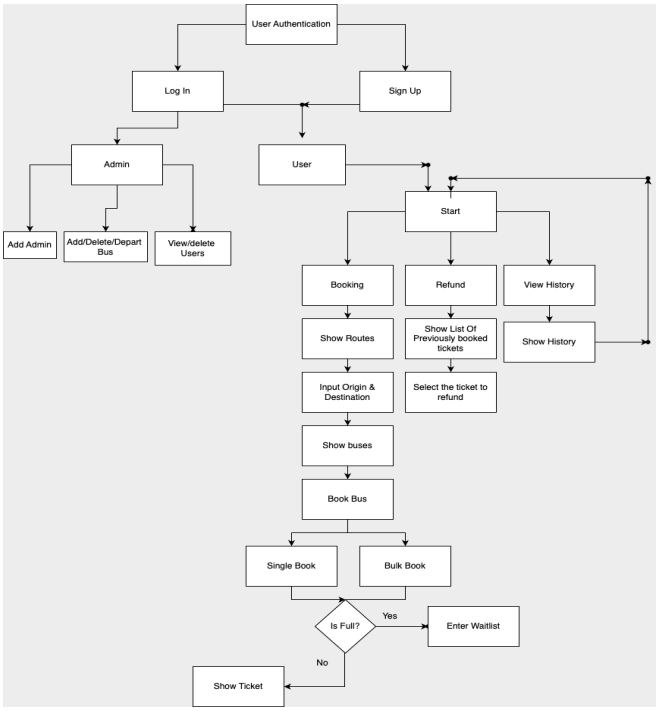
- **Vectors** will store and manage dynamic collections of data, like reservation IDs, users, and seat numbers, allowing for efficient additions with **O(1)** complexity for most operations.
- **Stacks** will help manage history or previous reservations, ensuring the last item added is accessed first, utilizing **O(1)** complexity for push/pop operations.
- **Queues** will handle customer waitlists, processing customers in a First In, First Out (FIFO) order, also with **O(1)** for enqueue/dequeue.
- **Hashmaps (unordered_map)** will provide fast lookups for users or bus data via key-value pairs, offering average constant-time complexity **O(1)** for access.

For algorithms, we'll use:

- **Hashmap-based** search for fast lookups **(O(1))**.
- **Linear search** for arrays, taking **O(n)** time when specific data isn't directly accessible.
- SHA-1 hashing for secure password storage and verification, ensuring data integrity.

Additionally, custom algorithms will handle generating unique IDs for users, buses, and reservations in a format like **X000000**, incrementing from the last ID, operating in constant time **O(1)**. For displaying reservations in reverse order, a stack will efficiently manage and print the latest entries, with an **O(n)** complexity for accessing and printing each item.

2.6. Program Flow



2.7. User Functionalities

User Authentication

The System class handles user authentication with the **authenticateUser()** method, which directs users to either **logIn()** or **signUp()** based on their choice.

- **authenticateUser()**: Displays a menu to the user, then calls either **logIn()** or **signUp()** based on the user's selection, returning a user object after successful authentication.
- **signUp()**: Allows new users to create an account by entering their name, age, email, and password (with **SHA-1 hashing** for security), and generates a unique user ID.
- **logIn()**: Authenticates existing users by verifying their email and hashed password against the database. Incorrect credentials prevent access to the system.

Reserve

1. Data Loading: The **loadData()** method reads data from a **Data.json** file, loading users, buses, reservations, and routes into JSON objects. **Time complexity is O(1).**
2. Displaying Information: destinationMenu() shows available routes by iterating through route data **(O(m x n) complexity)**. And **showAvailableBuses()** returns buses that match a specific origin and destination, iterating through buses **(O(2n) complexity).**
3. Bus Selection: The **selectBus()** method maps user input to a bus index, retrieving the selected bus for modification **(O(1) complexity).**
4. Task Delegation to Bus Class: The **reserveSeat()** and **reserveSeats()** methods manage seat reservations, with **reserveSeat()** checking availability and reserving a single seat

(O(n^2) complexity), and **reserveSeats()** handling multiple seat reservations with a complexity of **O(q × n).**
5. Data Storage: After making reservations, data is updated, and the **Data.json** file is rewritten. Searching for and updating the correct bus and user results in **O(n)** for buses and **O(m)** for users.
6. Finalization: Before writing to the file, final tasks are completed, such as generating a reservation ID and creating a ticket and QR code. This step adds personalization to the user experience.

Refund

1. Ordering the Printing of Tickets: The reservation IDs are stored in a stack, with the latest ID placed at the top. The stack follows the LIFO principle, ensuring the most recent reservations are processed first. **Time Complexity: O(n).**
2. Printing List of Refundable Tickets:A helper method, **refundList()**, is used to identify refundable tickets. Each ticket's status is checked, and valid refundable tickets are returned in a vector (**validRes**). **Time Complexity: O(n) + O(m).**
3. Choice Mapping: The user selects a refundable ticket, which is mapped to the corresponding index in the **resID** array. This allows the system to access and modify the associated reservation data. **Time Complexity: O(1).**
4. Retrieving Seat Numbers and Bus ID: Functions like **getSeatNumsToRefund()** and **getBusIDToRefund()** retrieves the seat numbers and bus ID associated with the reservation ID to be refunded. **Time Complexity: O(2n).**
5. Initializing a Bus Object for Seat Manipulation: Using the bus ID, the system locates the corresponding bus object and constructs it. Depending on whether the refund is for a single or bulk reservation, the system calls the appropriate refund method (**refundSeat()** for single, **refundSeats()** for bulk). **Time Complexity: O(n).**
6. Refund Type and Seat Changes:

- **Single Refund:** A specific seat's status is updated to "available" after it's refunded. **Time Complexity: O(n).**
- **Bulk Refund:** The user can choose to refund all or some of the seats, with edge cases like partially refunded bulk reservations being handled carefully. **Time Complexity: O(n).**

7. Collecting and Storing Data: Once the refund is processed, the bus's seat data is updated and stored. The reservation is marked as inactive, and for bulk refunds, new reservation IDs may be generated depending on whether the remaining seats constitute a single or bulk reservation. **Time Complexity: O(1).**

View History

1. Loading Data and Stack Initialization: loadData() populates **resIDStack** with reservation IDs.**Time Complexity: O(n).**
2. Processing Reservations: Each reservation ID is popped from the stack, and the relevant reservation (bulk or single) is found in the respective JSON array and displayed. **Time Complexity: O(n).**

3. Displaying Details: Bus ID and seat numbers are extracted and passed to **printHistory()**.**Time Complexity: O(1).**

Waitlist

1. Addition to Waitlist: Users can add themselves to the waitlist by specifying a seat number or opting for any available seat. A unique waitlist object is generated for each user. **Time Complexity: O(1) (excluding minor ID generation).**
2. Processing a Waitlist:
When a refund occurs, the system prioritizes waitlisted users based on their request type (specific or any seat). This process involves checking active waitlists and assigning refunded or available seats accordingly. **Time Complexity: O(n × m), due to two nested loops with different iteration counts.**

2.8. Admin Functionalities

The administrative system plays a very important role in managing the whole system. It provides the tools necessary for the admin to control and edit the functionality of the system,ensuring efficiency, security, and smooth operation.The admin functionalities are designed to make managing users and buses simple and organized while keeping the data accurate and secure.

Add Admin

This function allows an admin to add another administrator by collecting details like name, age, email, and password, setting **isAdmin** to true.

Time complexity:

- Password hashing: **O(n)**, where **n** is the password length.
- JSON file handling: **O(m)**, where **m** is the file size.

Overall,The time complexity of this function is **O(n)+O(m)**.

Add Bus

This function allows admins to add buses and the attributes of buses such as bus type, departure time, seats capacity and destination. While buses id are generated through the loop which means if the new buses are added to the data it will generate a new id starting from the last bus id. (e.g last bus id: “B0010” new bus id: “B0011”;

Time complexity:

- Creating the newBus: **O(seatCap)**, where **seatCap** is the number of seats on the new bus.
- Storing data: O(m), where **m** is the total number of buses in the system.

Overall, the time complexity of this function is **O(seatCap + m)**.

Delete Bus

The delete bus is the function that allows the admin to delete buses in the system based on the **busID**. However the admin can't delete the buses that have been booked by the users.

The admin can input the **busID** that they want to delete, the functions will search through data for the matching busID and display the bus information on the screen. If the id doesn't match it will show the message saying "Bus does not exist". If the bus has been booked, the buses cannot be deleted. If there's no booking, the function will ask for confirmation before deleting the bus. Time complexity: **O(m)** where m is the number of buses in the system.

View Bus

The view bus system is the function that allows the admin to view all the bus attributes such as bus id, departure time, seat capacity, bus type and destination. It has two options:

1. View the information of one specific bus by inputting **busID**.
2. View the information of all buses.

If the first option is selected, the system checks for the matching busID and displays the information relating to that bus. If the second option is selected, the information of all the buses will show up. Time complexity: **O(m)** where m is the number of buses in the system. Both of these options have the same time complexity since they are involved in going through data over the buses list.

View User

This function allows the admin to view user data, including name, age, ID, email, encrypted password, and reservations. It has **two** options:

1. View information for one specific user by entering their **userID**.
2. View information for all users.
3. If the first option is chosen, the program checks the provided user ID and displays the matching user's details. If the second option is selected, it displays all users' information. The function handles inputs properly and ensures data readiness.

Time complexity: O(n), where **n** is the number of users. Both options involve searching through or iterating over the user list, resulting in a time complexity of **O(n)**.

Delete User

The admin inputs the **user ID,** and the function searches for the matching user, displaying their information. If the user has an active reservation, they cannot be deleted, and an error message is shown. If there are no reservations, the admin is asked for confirmation before deletion. If canceled, a message is displayed, and exit the program. The time complexity is **O(m+n)**, where **m** is the file size and **n** is the number of users.

III. Outcomes

User Tasks:

- **Bus Search and Route Selection:** Users can easily search for buses and select routes.
- **Reservation:** Seamless seat reservation with unique IDs for tracking and validation.
- **View History:** Users can view their booking history.

Admin Tasks:

- **Bus Management:** Admins can add or delete buses.
- **User Management:** Admins can view and delete user accounts.
- **Admin Account Management:** Admins can add new admin accounts.

Project Achievements:

- **User and Admin Functionality:** Comprehensive tools for both user reservations and admin management.
- **Data Integrity:** Secure handling of user, bus, and reservation data in a structured format.
- **Enhanced User Experience:** Destination-based searches and seat layout visualization.
- **System Efficiency:** Reliable and secure platform with seamless tracking and consistency across operations.

IV. Perspective

4.1. Challenges

Technical Challenges: The team faced difficulties learning new libraries like Nlohmann JSON and SHA1. Time constraints led to a rushed system architecture, with the user class handling too many responsibilities. Database changes, like adding a “status” key to reservations, caused conflicts between branches.
Collaboration Challenges: The team struggled with Git due to a lack of experience, leading to issues with branch management, merging, and conflict resolution. Task prioritization was challenging, and varying skill levels caused delays, requiring more experienced members to assist.

4.2. Reflection

The project highlighted the importance of understanding data structures and algorithms (DSA) for optimizing performance, particularly when working with libraries like Nlohmann JSON. We learned the significance of choosing the right data structures, such as unordered maps, to handle data efficiently. Additionally, the need for careful design in system architecture became clear, as poorly structured classes can lead to maintenance challenges. We also gained experience in database management and the impact of changes on the overall system, emphasizing the importance of planning for scalability and conflict resolution.

V. Future Improvements:

To enhance the system, we propose refining the architecture by redistributing class responsibilities, such as creating a **dedicated admin class**. **Algorithms could be optimized** to reduce inefficiencies, such as eliminating unnecessary data replacement and insertion. The database design can be improved and potentially transitioning to a more structured **SQL** database for better consistency and efficiency.

Additional features include implementing a waitlist system for full buses and transitioning to a **GUI** for improved user experience, possibly using **QT. Real-time syncing** with bus departures and reservations could also be explored for future development.

VI. Conclusion

The bus reservation system successfully provides an efficient and user-friendly platform for both users and administrators. By incorporating features like seat reservations, booking history, and admin management tools, the system streamlines the booking process and ensures smooth operations. Despite challenges with system architecture, algorithm optimization, and collaboration, the team developed a functional solution that met the project requirements. Future improvements, including refining the architecture, optimizing algorithms, and adding additional features like a GUI and real-time syncing, will further enhance the system's capabilities and user experience.

VII. References

Nlohmann, M. (2020). *JSON for Modern C++*. GitHub. Retrieved from https://github.com/nlohmann/json

C++ Standards Committee. (2023). *ISO/IEC 14882:2023 - Information technology - Programming languages - C++*. International Organization for Standardization. Retrieved from https://www.iso.org/standard/79358.html

The C++ Programming Language. (2014). *The Standard C++ Foundation*. Retrieved from https://isocpp.org/

Git. (2023). *Pro Git*. Git SCM. Retrieved from https://git-scm.com/book/en/v2