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

The Bus Management System is designed to be an efficient management solution, developed entirely in C++ to demonstrate the practical application of advanced programming and algorithmic concepts. The system allows users to reserve single or multiple seats, cancel or refund previous reservations, and view their booking history. On the administrative side, it provides administrators with the ability to manage admin accounts, oversee departing buses, modify bus configurations, and access detailed user logs and data. This comprehensive design ensures both a seamless user experience and robust administrative control.

This project incorporates skills acquired throughout the 10 weeks of coursework in the Advanced Algorithms course. Concepts such as sorting algorithms, password-hashing for security, and efficient data lookup are directly integrated into the source code to ensure robustness and efficiency. Additionally, data structures are employed to store data persistently and facilitate efficient operations. These include vector arrays, linked lists, queues, and hashmaps, all implemented and optimized using C++.

The system is built using Object-Oriented Programming (OOP) principles to ensure modularity, scalability, and long-term maintainability. By structuring the application into multiple interacting classes, such as System, User, and Bus, each responsible for specific functionalities, the design promotes clear separation of concerns. This approach, combined with the powerful features of C++, allows for easier maintenance, testing, and future enhancements.

To manage data storage and persistence, the system leverages the nlohmann/json library, a popular C++ library for handling JSON data. This library simplifies the process of reading and writing user and bus data to and from files. It plays a crucial role in ensuring seamless data storage for bus details, user information, and reservations.

Throughout the development process, the system's architecture and class design have been carefully considered to ensure efficiency, scalability, and ease of use. This report documents the development of the Bus Management System, covering the design decisions, implementation details, and technical challenges faced during the creation of the system.

II. Objectives

The primary objective of this project was to develop an efficient solution to a real-world problem by applying concepts and techniques learned in the Advanced Algorithms course. The Bus Management System was designed to streamline the processes of managing reservations, optimizing bus resource allocation, and ensuring a user-friendly experience for both passengers and administrators. Below are the specific objectives of the project:

2.1. User Experience/Operations Objectives

The following are the core user actions which enables them to do various common purchases and booking related areas.

Simplify Bus Reservations:

Enable users to reserve single or multiple seats via a simple command-line interface. The system eliminates the need for manual processes, offering a straightforward and efficient way to book tickets.

Facilitate Refund Handling:

Provide users with an option to cancel or refund previous bookings easily. This feature ensures that users are capable of maintaining their rights over their past reservations, and for the robustness and flexibility of the system as a whole.

Track Booking History:

Users can view their booking history, including details like dates, seat numbers, and bus. This feature aims to provide the user the ability to keep track of their bookings recent or previous.

2.2. Administrative Experience/Operations Objectives

The following are the key functionalities designed for administrators, providing them with control over the system's operations:

Scheduling of Bus Departures:

Provide administrators with the functionality to manage bus departures effectively. This includes assigning buses to specific departure times and ensuring proper alignment with users who have reservations on those buses.

User Deletion

Enable administrators to delete user accounts along with their associated booking data, ensuring accurate and streamlined system management.

Addition of new administrators

Allow existing administrators to register new admin accounts, expanding the administrative team as needed to handle system operations efficiently.

2.3. Technical Objectives

Application of Course Concepts

Utilize advanced algorithms and techniques learned during the coursework to enhance system performance and reliability. This includes implementing efficient sorting and searching algorithms, optimizing resource usage, and employing various data structures like vectors, queues, linked lists, and hash maps for seamless data management.

Implement Object Oriented Programming Principles (OOP)

Employ OOP principles to ensure a modular, maintainable, and reusable codebase rather than procedural programming. Key design patterns and encapsulated classes enable the system to remain organized and extensible for future development.

Scalability and Extensibility

Design a system architecture that can scale with increasing user demand and integrate additional features in the future.

Robust Data Management

Utilize libraries such as nlohmann/json for reliable and structured data handling. This approach ensures the system can effectively store, retrieve, and update data while maintaining consistency and reliability.

Secure User Data

Implement security measures like password hashing to protect user credentials. Additionally, safeguard sensitive information to uphold user privacy and prevent unauthorized access to critical system functions.

III. Methodology

3.1. Choosing the project

After reviewing a list of potential project ideas, we carefully considered each option to determine the most suitable project for development. We ultimately decided to build a **Bus Management System** under the broader category of travel management systems. The decision was based on several key criteria:

Useful real world application:

The chosen project should address real-world problems by offering practical and efficient digital solutions. It should streamline and automate processes, reducing manual effort and enhancing usability for both end-users and system operators. This ensures that the project has tangible benefits and relevance beyond the academic context.

Ease of Demonstration

Given the development constraints, the project must be easy to demonstrate using a command-line interface (CLI). This ensures that the core functionality can be showcased effectively without the need for complex setups or external dependencies, making it accessible for testing and evaluation.

Challenging Enough for Learning and Application of Advanced Concepts:

While the project is straightforward, it also presents significant challenges that allow us to apply key concepts from the Advanced Algorithms course. These include the implementation of sorting algorithms, optimization techniques, and the use of various data structures such as vectors, queues, and hash maps to manage reservations and bus schedules efficiently. The system also requires handling user data securely, further emphasizing the practical application of security principles such as password hashing.

Real-World Business Context

The project should address a practical problem faced in real-world scenarios, making it relevant and relatable. While the initial implementation may be limited to basic features, the core idea should have the potential to be adapted for broader applications or real-world deployment.

3.2. System Architecture Analysis

The initial step was to determine the system's capabilities and define the operations it should allow users to perform. By analyzing these core functions and processes, we could design the necessary components, modules, data structures, and interactions for the system.

3.2.1. Research and Analysis

In the early stages of planning, we determined that the system should consist of two distinct components. One part is designed for users to interact with, enabling them to reserve seats and perform related operations. The other part focuses on administrative functions, granting administrators high-level control over the system's resources, including buses, their routes, and user management.

User Functionality

To better understand what users are capable of doing in a bus reservation program, we conducted research into reputable web and mobile applications to identify common features and best practices.

Key findings:

- Users can reserve seats by specifying their origin and destination, which then displays suitable buses for selection.
- After selecting a bus, users can book specific seats and proceed with payment.
- Refund functionality allows users to cancel and refund specific bookings as needed.
- Logged-in users can access their **booking history**, providing a detailed record of past reservations.

From this research, we finalized the **three core user operations**:

1. **Reserve Seat(s)**
2. **Refund Bookings**
3. **View Booking History**

Administrative Functionality

To complement user functionalities, we decided to include administrative features for managing system resources. Since web applications typically do not expose administrative tools for analysis, we relied on brainstorming and understanding typical requirements for such systems.

The **admin interface** supports high-level control over:

- **Bus Resources:** Creating and modifying bus configurations (e.g., seating layouts).
- **Bus Departures:** Manually scheduling and managing bus departures.
- **User Management:** Adding or removing admin accounts, deleting user accounts, and retrieving user details.

These findings helped us finalize the primary administrative operations, which include:

1. **Add New Administrators**
2. **Manage Bus Departures**
3. **Add/Delete Bus Configurations**
4. **Delete Users**
5. **Retrieve User Information**

3.2.2. Choosing the System Architecture

The decision making on the system architecture to implement was a very crucial initial step into the development stages of this project. What we wanted was a system that is robust, scalable, fits with the CLI interface implementation and Object Oriented Design.

Procedural Programming Vs. Object Oriented Programming

At first, we considered utilizing a procedural approach to system architecture, as we had done in previous programming projects. However, to create a system that is more scalable, maintainable, and easier to understand, we opted for an **object-oriented programming** (OOP) approach. This allowed us to design the system using classes such as System, User, and Bus, promoting modularity and clear separation of responsibilities.

Data Storage

When addressing data storage concerns, we explored various options, including plain text files, JSON files, and databases like SQLite or MySQL. Ultimately, we decided to use JSON for data storage as it strikes a balance between the simplicity of plain text files and the complexity of a full-fledged database. **JSON** offers a structured, meaningful format that is easy to parse and manipulate with the help of the **nlohmann/json** library while being lightweight and efficient for the scale of this project.

Data Structures and Algorithms

We narrowed down different data structures and algorithms according to their complexity of implementation and usefulness. Thus, the system leverages a variety of data structures and algorithms to ensure efficient operation. **Queues** are used to manage a waitlist, **Hashmaps** are utilized for fast lookups of user information and reservation details. Additionally, **sorting algorithms** are employed to manage and display lists of buses to choose or view a list of history. Moreover, a **hashing algorithm** was implemented to encrypt user passwords.

Error Handling and Validation

To ensure system robustness, error handling mechanisms were incorporated to manage unexpected **inputs** or **system states**. Validation checks were implemented for user inputs, such as ensuring seat numbers, and **correct login credentials**. This prevents errors from propagating and enhances the user experience.

Security

Security was a key consideration, particularly since users are required to log in to perform operations such as making reservations or processing refunds. This raised concerns about protecting users' sensitive information, such as passwords, from being exposed or accessed by developers. To address this, we implemented password **hashing** techniques to securely encrypt user passwords, ensuring that sensitive data remains protected and reducing the risk of unauthorized access.

User Friendly CLI Design

Since the project is CLI-based, special attention was given to designing a user-friendly interface. **Commands** and **prompts** were structured intuitively to minimize the learning curve for users and administrators while ensuring smooth navigation **forward and back** through system functionalities.

3.3. System Architecture Definition

After careful considerations and analysing small but significant details, we have constructed a detailed outline of the system architecture that is going to be utilized to build this project from the ground up ensuring robustness and efficiency while, also leveraging the skills acquired during the 10 weeks of coursework for the Advanced Algorithm Course.

Before the technical deep-dive, we shall introduce the **file structure** for this project:

```
├── README.md
├── doc
│   ├── Bus.md
│   ├── Menu.md
│   ├── Nlohmann-tutorial.md
│   ├── Nlohmann.md
│   ├── Route.md
│   ├── SHA1.md
│   ├── System.md
│   ├── User.md
│   └── Validation.md
├── main
```

```
|   ├── main.cpp

├── test
|   ├── test.cpp
|   ├── test.json
|   └── userTest.json
└── utils
    ├── database
    |   └── Data.json
    ├── header
    |   ├── Bus.hpp
    |   ├── System.hpp
    |   ├── User.hpp
    |   ├── menu.hpp
    |   └── validation.hpp
    └── libs
        ├── json.hpp
        └── sha1.hpp

8 directories, 22 files
```

3.3.1. Object Oriented Design

To oppose the usage of traditional procedural programming, we have decided to apply OOP to this program. By analysing the needs of this system, the team carefully constructed different classes with clear and logical responsibilities. Briefly, we concluded with 3 main classes: **System**, **User** and **Bus**.

System Class:

The **System class** handles user authentication, including login and signup processes. It interacts with a JSON file to securely store and retrieve user data, managing user details such as name, age, email, password, and admin status. The class returns a User object for further operations.

+ Attributes and Methods

```
class System // Responsibility: Handles user authentication (login/signup)
{
private:
    /*User Data JSON object to use for helper method*/
    /*****************************************/
    // Core functions
    /*****************************************/
    int logInOrSignUp();
    User *logIn();
```

```
    User *signUp();
    bool isAdmin();
    /*****************************************/
    // Helper Functions for sign up
    /*****************************************/
    void loadUser();
    string inputFirstName();
    string inputLastName();
    int inputAge();
    string inputEmail();
    string inputPassword();
    string confirmPassword(string);
    string generateUserID();
    /*****************************************/
    // Helper Functions for log in
    /*****************************************/
    string getID(string);
    int getAge(string);
    string getFirstName(string);
    string getLastName(string);
    bool getAdminStatus(string);
    vector<string> getResID(string);
    string getEmail();
    string getPassword(string);
    /*****************************************/
    /*User Data JSON object to use for helper methods*/
    json userData;
    /*****************************************/

public:
    System() {}                // Default-construct `User` object
    User *authenticateUser(); // user authentication process (return a user obj)
};
```

+ Flow

Step	Description
1. Authentication	**authenticateUser()** prompts the user to choose between login or signup.
2. Login Process	**logIn()** verifies the email and password, retrieves user details, and returns a User object.

3.Signup Process	**signUp()** collects user details, validates inputs, saves new user data, and creates a User object to return.
4.Helper Methods	Validates inputs (e.g., name, email), hashes passwords, and assists in retrieving or saving user details.
5.Data Management	Reads from and writes to a JSON file **(Data.json)** for user data storage and retrieval.

+ Public Methods

Method	Description	Returns
User* authenticateUser()	Prompts the user to log in or sign up. Routes to the appropriate method.	A pointer to the authenticated user
int logInOrSignUp()	Displays a menu for login or signup options.	1 for login or 2 for signup
User* logIn()	Handles login by validating credentials and retrieving user details.	A pointer to the User object
User* signUp()	Handles signup by validating inputs, saving data, and creating a new user.	A pointer to the new User object

+ Private Helper Methods (Signup)

Method	Description	Returns
void loadUser()	Loads user data from Data.json into memory.	N/A
string inputFirstName()	Prompts and validates the user's first name.	A valid first name
string inputLastName()	Prompts and validates the user's last name.	A valid last name
int inputAge()	Prompts and validates the user's age.	A valid age

string inputEmail()	Prompts and validates the user's email address.	A valid email
string inputPassword()	Prompts and validates the user's password for strength.	A valid password
string confirmPassword()	Prompts for password confirmation and ensures it matches the original password.	The hashed password
string generateUserID()	Generates a unique user ID based on the existing number of users in Data.json.	A new user ID

+ Private Helper Methods (Login)

Method	Description	Returns
vector<string> getResID()	Retrieves the reservation IDs associated with the given email.	A vector of reservation IDs
bool getAdminStatus()	Checks whether the user with the given email has admin privileges.	true if admin, else false
string getID()	Retrieves the user ID associated with the given email.	A user ID
string getEmail()	Retrieves the email associated with the logged-in user.	A user email
string getPassword()	Retrieves the hashed password associated with the given email.	The hashed password
int getAge()	Retrieves the age of the user with the given email.	The user's age
string getFirstName()	Retrieves the first name of the user with the given email.	The user's first name
string getLastName()	Retrieves the last name of the user with the given email.	The user's last name

+ Example Usage

```
System system;
User *authenticatedUser = system.authenticateUser();
```

```
// If user is logged in or signed up successfully
authenticatedUser.operation();
```

User Class

The **User class** represents a user of the bus reservation system, handling both normal users and admins. The class supports functionality such as managing reservations, handling user details, and enabling admin-level actions.

+ Attributes and Methods

```
class User
{
private:
   // user attributes ============================================================
   string userID;
   string name;
   string lastName;
   string firstName;
   int age;
   string email;
   string password;
   bool isAdmin;
   vector<string> resID; // Stores reservation IDs

   // For initially loading data ================================================
   json data;
   json users;
   json buses;
   json reservations;
   json routes;

   // For partially changed data to be put back to file =====================
   json busToModify;
   json modifiedUser;

   // Helper Methods
   // Working with data
   void loadData();

   // show and input from, to
```

```
    void destinationMenu();
    string inputFrom();
    string inputTo(string);

    // show buses and bus selection based on search
    vector<int> showAvailableBuses(string, string);
    int printBus(json, string, string, int, int *);
    Bus selectBus(vector<int>);

    // After working with the bus object(reserve seat....)
    void generateResID(int seatNum);
    void generateTicket(int);
    void showQRCode();
    void storeData();

public:
    User() = default; // Default constructor

    // Constructor to initialize user data after authentication
    User(string UID, string fn, string ln, string n, int a, string em, string pswd,
bool aS, vector<string> rID)
    {
        userID = UID;
        firstName = fn;
        lastName = ln;
        name = n;
        age = a;
        email = em;
        password = pswd;
        isAdmin = aS;
        resID = rID;
    }

    // Set and Get methods
    string getUID() { return userID; }
    string getName() { return name; }
    string getFirstName() { return firstName; }
    string getLastName() { return lastName; }
    int getAge() { return age; }
    string getEmail() { return email; }
    string getPassword() { return password; }
    bool getAdminStatus() { return isAdmin; };

    // check and redirect the user to be admin or normal user
    void checkUserType();
```

```
    // Core User Methods
    void reserve();      // Method for reserving a bus ticket
    void refund();       // Method for refunding a reservation
    void viewHistory(); // Method for viewing reservation history

    // Core Admin Methods
    void addAdmin();
    void addBus();
    void changeBusSettings();
    void getAllUsers();

    // Helper Functions
    void printUser();
};
```

+ Flow

Step	Description
1. Initialization	**Constructor** initializes user attributes like userID, firstName, lastName, email, etc., and loads associated data.
2. User Type Check	**checkUserType()** determines if the user is an admin or a normal user and redirects accordingly.
3. Data Loading	**loadData()** loads data from files (users, buses, reservations, and routes) into JSON objects.
4. Destination Selection	**destinationMenu()** shows the destination menu, inputFrom() takes the starting point, and inputTo(string) takes the destination.
5. Bus Search and Selection	**showAvailableBuses()** lists buses matching the search, printBus() displays bus details, and selectBus() allows bus selection.
6. Seat Reservation	**reserve()** reserves seats and generates a reservation ID (**generateResID()**) and ticket (**generateTicket()**).
7. Data Storage	**storeData()** saves modified data, including reservations and buses, back to the files.
8. Admin Functions	Admins can use **addAdmin()**, **addBus()**, **changeBusSettings()**, and **getAllUsers()** to manage buses and users.

9. Ticket Management	**refund()** handles ticket refunds, and **viewHistory()** displays the user's reservation history.
10. QR Code Generation	**showQRCode()** generates a QR code for payment.

+ Private Attributes

Attribute	Type	Description
userID	string	Unique identifier for the user.
name	string	Full name of the user.
firstName	string	First name of the user.
lastName	string	Last name of the user.
age	int	Age of the user.
email	string	Email address of the user.
password	string	Password for user authentication.
isAdmin	bool	Indicates whether the user has admin privileges.
resID	vector<string>	Stores reservation IDs associated with the user.
data	json	Complete dataset loaded from the data file.
users	json	Subset of user data from the complete dataset.
buses	json	Subset of bus data from the complete dataset.
reservations	json	Subset of reservation data from the complete dataset.
routes	json	Subset of route data from the complete dataset.
busToModify	json	Stores the data of a bus being modified.

modifiedUser	json	Stores partially changed user data for updates.

+ Public Methods

Constructors

Method	Description
User()	Default constructor that initializes an empty User object.
User(string UID, string fn, string ln, string n, int a, string em, string pswd, bool aS, vector<string> rID)	Parameterized constructor to initialize user attributes after authentication.

Getters

Method	Return Type	Description
getUID()	string	Returns the unique user ID.
getName()	string	Returns the full name of the user.
getFirstName()	string	Returns the first name of the user.
getLastName()	string	Returns the last name of the user.
getAge()	int	Returns the age of the user.
getEmail()	string	Returns the email of the user.
getPassword()	string	Returns the password of the user.
getAdminStatus()	bool	Returns whether the user is an admin.

Core Methods

Method	Description
checkUserType()	Checks if the user is an admin or a normal user and redirects accordingly.

reserve()	Handles the process of reserving a bus ticket.
refund()	Manages the process of refunding a bus reservation.
viewHistory()	Displays the reservation history of the user.

Admin Methods

Method	Description
addAdmin()	Adds a new administrator to the system.
addBus()	Adds a new bus to the system.
changeBusSettings()	Allows administrators to modify bus settings.
getAllUsers()	Retrieves and displays all users in the system.

Helper Methods

Data Handling

Method	Description
loadData()	Loads data from the specified data file into JSON objects.
storeData()	Saves modified data back to the file.

Reservation Support

Method	Description
destinationMenu()	Displays available routes for user selection.
inputFrom()	Accepts and validates the origin of a route from the user.
inputTo(string)	Accepts and validates the destination of a route from the user.
showAvailableBuses(string, string)	Displays buses available for a selected route.

printBus(json, string, string, int, int *)	Displays details of a bus if it matches the selected route criteria.
selectBus(vector<int>)	Prompts the user to select a bus and returns the chosen Bus object.

Reservation Finalization

Method	Description
generateResID(int)	Generates a unique reservation ID for the selected seat based on the type of booking.
generateTicket(int)	Generates a ticket for the reserved seat.
showQRCode()	Displays a QR code for the reservation..

Bus Class

The **Bus class** represents a bus in the bus reservation system. It handles attributes related to bus details, seat availability, and allows for seat reservations

Attributes and Methods

```
class Bus
{
private:
   // Attributes
   string busType; // Type of bus (e.g., luxury, economy)
   string dpTime;  // departure time
   string busID;   // Unique ID for the bus
   json route;     // route of bus
   int seatCap;    // Total seat capacity
   int seatLeft;   // remaining seats
   int seatPrice;  // Price per seat
   json seats;

   bool isSeatAvailable(int seat); // Checks if a specific seat is available
   vector<int> numOfSeatsChanged;

public:
   // Constructor
    Bus(string type, string dpTime, string id, json route, int cap, int rem, int
price, json seats)
```

```
    {
        busType = type;
        this->dpTime = dpTime;
        busID = id;
        this->route = route;
        seatCap = cap;
        seatLeft = rem;
        seatPrice = price;
        this->seats = seats;
    };

    // Core Bus Methods
    void printHistory(vector <string>);
    void printBusInfo();
    void showSeatLayout(); // Displays seat layout of the bus
    json reserveSeat();    // Reserves a single seat
    json reserveSeats();   // Reserves multiple seats
    int getSeatLeft() { return this->seatLeft; };
    vector<int> getSeatNumChanges() { return this->numOfSeatsChanged; };
};
```

+ Flow

Step	Description
1. Initialization	Constructor initializes attributes such as busType, dpTime, busID, route, seatCap, seatLeft, seatPrice, and seats.
2. Seat Availability	**isSeatAvailable(int seat)** checks if a specific seat is available.
3. Seat Reservation	**reserveSeat()** reserves a single seat, and **reserveSeats()** reserves multiple seats.
4. Seat Updates	Tracks changes to seat availability using **numOfSeatsChanged**.
5. Seat Layout Display	**showSeatLayout()** displays the current seat layout.
6. Bus Information	printBusInfo() outputs the bus details, such as type, capacity, and price.

+ Private Attributes

Attribute	Type	Description
busType	string	Type of the bus (e.g., luxury, standard).
dpTime	string	Departure time of the bus.
busID	string	Unique identifier for the bus.
route	json	The route information of the bus, including origin and destination.
seatCap	int	Total seat capacity of the bus.
seatLeft	int	Remaining available seats on the bus.
seatPrice	int	Price per seat.
seats	json	JSON array representing the seats and their availability.
numOfSeatsChanged	vector<int>	List of seat numbers that have been changed (reserved).

+ Public Methods

Constructor

Method	Description

Bus(string type, string dpTime, string id, json route, int cap, int rem, int price, json seats)	Parameterized constructor to initialize a Bus object with given bus details.

Getters

Method	Return Type	Description
getSeatLeft()	int	Returns the number of remaining seats available.
getSeatNumChanges()	vector<int>	Returns a list of seat numbers that have been reserved.

Core Methods

Method	Description
showSeatLayout()	Displays the seat layout of the bus, showing available and unavailable seats.
reserveSeat()	Prompts the user to reserve a single seat and updates the seat status.
reserveSeats()	Prompts the user to reserve multiple seats and updates the seat statuses.
printBusInfo()	Prints the bus details including bus ID, type, route, and seat availability.

Helper Methods

Method	Description
isSeatAvailable(int seat)	Checks if a specific seat is available for reservation based on its seat number.

3.3.2. Helper libraries/Dependencies

This project utilizes several libraries, including custom libraries developed specifically for this program, third-party libraries created by other developers and built-in standard C++ libraries.

Independently made Libraries:

+ Menu.hpp:

The menu.hpp file contains pre-made decision making menus/input stages of this program that does not require the usage of reading and writing or validation of the inputs to stored files. This is simply a library of pre-made menus that can be called as a function in the main classes to use for decision making stages. Additionally, there are also ASCII art components used to add additional visual appeal to the bland Command Line Interface (CLI). Below are some example of functions that are in this file:

```
int logInOrSignUpMenu()
{
   int authChoice;
   do
   {
       cout << "\n\n\t\t\t  USER AUTHENTICATION\n";
       cout << "1. Log in\n";
       cout << "2. Sign Up\n";
       cout << "Enter (1/2): ";
       cin >> authChoice;
       if (cin.fail() || authChoice < 1 || authChoice > 2)
       {
           cout << "\nPlease enter again...\n\n";
           clearInput();
       }
       else
       {
           break;
       }
   } while (1);
   return authChoice;
}
void printThanks()
{
```

```
    cout << "\n\n";
    cout << "    _____ _   _    _    _   _ _ __ __   _____  _   _ \n"
            "   |_   _| | | |  / \\  | \\ | | |/ / \\ \\ / / _ \\| | | |\n"
            "     | | | |_| | / _ \\ |  \\| | ' /   \\ V / | | | | | |\n"
            "     | | |  _  |/ ___ \\| |\\  | . \\    | || |_| | |_| |\n"
            "     |_| |_| |_/_/   \\_\\_| \\_|_|\\_\\   |_| \\___/ \\___/"
         << endl;
    cout << " _____ ___  ____    ____   ___   ___  _  _____ _   _  ____ \n"
            "|  ___/ _ \\|  _ \\  |  _ \\ / _ \\ / _ \\| |/ /_ _| \\ | |/ ___|\n"
            "| |_ | | | | |_) | | |_) | | | | | | | ' / | ||  \\| | |  _ \n"
            "|  _|| |_| |  _ <  |  __/| |_| | |_| | . \\ | || |\\  | |_| |\n"
            "|_|   \\___/|_| \\_\\ |_|    \\___/ \\___/|_|\\_\\___|_| \\_|\\____|\n"
         << endl;
    cout << "      __        _____ _____ _   _   _   _ ____     _ \n"
            "     \\ \\      / /_ _|_   _| | | | | | | / ___|   | |\n"
            "      \\ \\ /\\ / / | |  | | | |_| | | | | \\___ \\   | |\n"
            "       \\ V  V /  | |  | | |  _  | | |_| |___) |  |_|\n"
            "        \\_/\\_/  |___| |_| |_| |_|  \\___/|____/   (_)\n"
         << endl;
}
```

+ Validation.hpp:

The validation.hpp file is used for checks of inputted data to see whether what was inputted is valid. It consists of different functions that are made to validate different types of inputs at the various stages of the program such as age, name and valid email inputs. Below are some functions of this file:

```
bool isEmailValid(string email)
{
    for (int i = 0; i < email.size(); i++)
    {
        if (email[i] == '@')
        {
            string emailEndCheck = email.substr(i, email.size());
            if (emailEndCheck == "@gmail.com")
            {
                return true;
            }
            else
            {
                cout << "\nError: Email is invalid\n";
                return false;
            }
```

```
        }
    }
    return false;
}
```

```
bool isEmailAvailable(string email)
{
    ifstream readData(dataFile);
    if (!readData.is_open())
    {
        cerr << "\nError: Failed to open file, Path: " << dataFile << " \n";
        return false;
    }

    // Parse the JSON data
    json allData;
    readData >> allData;
    readData.close();

    // Iterate through the users to check for email
    for (const auto &user : allData["users"])
    {
        if (user.contains("email") && user["email"] == email)
        {
            cout << "\nError: Email is taken\n";
            return false;
        }
    }

    // Email is available
    return true;
}
```

Third Party Libraries

+ json.hpp

The project utilizes the json.hpp library, developed by Niels Lohmann, for seamless JSON handling in C++. This lightweight and powerful header-only library simplifies parsing, creating, and manipulating JSON objects. It enables intuitive usage with features like JSON serialization and deserialization, making it ideal for managing structured data. Below is a demonstration of the usage of this library:

```
#include <iostream>
```

```cpp
#include <fstream>
#include "json.hpp"

using json = nlohmann::json;

int main() {
    // Open the JSON file
    std::ifstream inputFile("data.json");
    if (!inputFile.is_open()) {
        std::cerr << "Error: Could not open file." << std::endl;
        return 1;
    }

    // Parse the JSON file into a json object
    json jsonData;
    inputFile >> jsonData;
    inputFile.close();

    // Access and display specific data
    std::cout << "Bus ID: " << jsonData["busID"] << std::endl;
    std::cout << "Seats Available: " << jsonData["seatLeft"] << std::endl;

    // Display all the data in the file
    std::cout << "\nComplete Data:\n" << jsonData.dump(4) << std::endl;

    return 0;
}
```

+ sha1.hpp

This library is a lightweight, header-only implementation of the SHA-1 hashing algorithm in C++. **SHA-1 (Secure Hash Algorithm 1)** generates a 160-bit hash value typically represented as a 40-character hexadecimal number. This implementation provides a **SHA1** class for processing input data and computing the hash. It supports operations like resetting the state, processing bytes or blocks of data, and retrieving the final hash as either a 32-bit integer array or a byte array. Below is a usage example of this library:

```cpp
#include <iostream>
#include <iomanip>
#include "sha1.hpp" // Include the library

int main() {
    std::string password = "hello_world";
    std::string hashed = hashPassword(password); // Call the hashPassword function
    std::cout << "Original: " << password << "\n";
```

```
    std::cout << "Hashed: " << hashed << "\n"; // Display the hashed result
    return 0;
}
```

```
Output

Original: hello_world
Hashed: 2aae6c35c94fcfb415dbe95f408b9ce91ee846ed
```

Standard C++ Libraries:

The Bus Management System leverages several standard built-in C++ libraries to enable core functionality. Common libraries such as **<iostream>** and **<fstream>** handle input/output operations, forming the foundation for approximately 90% of the codebase. Additionally, the system employs less commonly used libraries like **<stack>** for managing data structures, **<map>** for efficient key-value data storage (e.g., mapping buses to their schedules), **<algorithm>** for advanced data processing, and **<limits>** for handling edge cases in numeric computations. These additional libraries enhance the system's functionality, enabling complex features like route optimization, data validation, and error handling. (For more details about the C++ Standard Library, please refer to the official documentation linked in the reference section below).

3.3.3. Database Design

The database for the bus booking system is designed to support key functionalities such as managing buses, routes, seats, and reservations. The structure reflects a hierarchical and relational approach, with data stored in **JSON** format for easy retrieval and modification. Below is a breakdown of the database design and its high level components:

+ High Level Objects

Users

Each user is represented by a record containing personal details and their associated reservations. Attributes include:

- **id:** Unique identifier for the user.
- **name:** First and last name of the user.
- **age:** The user's age.
- **email:** Contact email for the user.
- **password:** A hashed password for user authentication.
- **isAdmin:** Indicates whether the user has admin privileges.
- **resID:** A list of reservation IDs associated with the user.

Example:


```
{
  "users": [
    {
      "id": "U000001",
      "name": {
        "firstName": "John",
        "lastName": "Doe"
      },
      "age": 30,
      "email": "john.doe@example.com",
      "password": "hashedPassword",
      "isAdmin": false,
      "resID": []
    }
  ]
}
```


Buses

The buses collection stores details about each bus, including its type, route, departure time, seating capacity, and the status of individual seats. Each bus entry contains:

- **busType**: Specifies the type of bus (e.g., Standard, Luxury, Sleeper).
- **departureTime**: Timestamp indicating when the bus departs.
- **id**: A unique identifier for each bus.
- **route**: Contains the departure and destination locations.
- **seatCap**: The total number of seats on the bus.
- **seatLeft**: The number of available seats.
- **seatPrice**: Price per seat for the journey.
- **seats**: An array representing individual seat details, including:
    - **seatNum**: The seat number.
    - **status**: Indicates whether the seat is "available" or "reserved".

Example:


```
{
  "busType": "Standard Bus",
  "departureTime": "2024-12-01 06:00",
  "id": "B0001",
  "route": {
    "from": "Phnom Penh",
    "to": "Siem Reap"
  },
  "seatCap": 6,
```

```
  "seatLeft": 6,
  "seatPrice": 10,
  "seats": [
    { "seatNum": 1, "status": "available" },
    { "seatNum": 2, "status": "available" }
  ]
}
```

Reservations

The reservations collection manages both bulk and individual reservations:

- **bulkReservations**: Used for group bookings, each record contains:
  - busID: The ID of the bus being reserved.
  - id: A unique identifier for the reservation.
  - seatNumber: An array of seat numbers reserved.
  - userID: The ID of the user making the reservation.
- **singleReservations**: For individual seat bookings, each record contains:
  - busID: The ID of the bus being reserved.
  - id: A unique identifier for the reservation.
  - seatNumber: The specific seat number reserved.
  - userID: The ID of the user making the reservation.

Example:

```
{
  "bulkReservations": [
    {
      "busID": "B0003",
      "id": "RB000000",
      "seatNumber": [1, 2, 3],
      "userID": "U000001"
    }
  ],
  "singleReservations": [
    {
      "busID": "B0002",
      "id": "R000000",
      "seatNumber": 4,
      "userID": "U000001"
    }
  ]
}
```

Routes

The routes collection defines possible travel routes. Each record includes:

- **from**: The starting location.
- **to**: A list of possible destinations from the starting location.

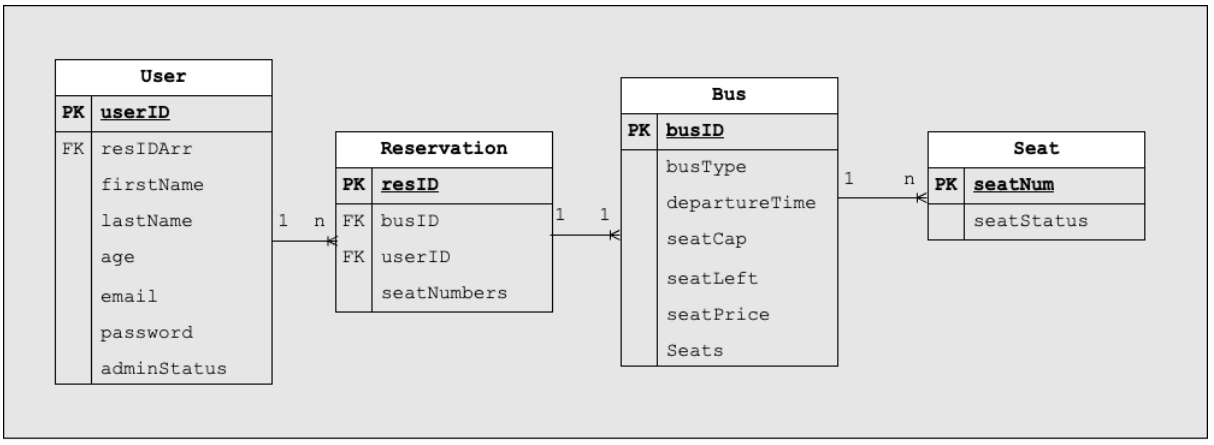
Example:

```
{
  "from": "Phnom Penh",
  "to": ["Siem Reap", "Battambang", "Kampot"]
}
```

+ Relationships

- **Buses** to **Routes**: A bus is linked to a route, indicating where it travels from and to.
- **Buses** to **Seats**: Each bus contains a list of seats, each with a status (either reserved or available). This design helps manage the seating arrangements and availability of seats on each bus.
- **Users** to **Reservations**: Users are linked to their reservations through the resID attribute, which references either single or bulk reservations.

Entity Relationship Diagram:



3.3.4. Data Structures

This Bus Management Program implements various data structures from primitive to decently complex data structures. This implementation was thought of to utilize performance and complexity. The data structures include vector, stack, queue and hashmaps/hash tables.

Vector

Vector is a powerful data structure in C++ that allows for robust storage and access of data. Throughout the development stage of this project we implemented this data structure more

than other data structures due to its ease of use and completeness. However, there are times where a vector array is not compatible or efficient (will discuss later).

+ Example:

```
// A vector of seatNumbers
vector <int> seatNumbers;

// A vector of seat objects
vector <json> seats;

// Adding a new user;
vector <json> currentUsersList;
vector <json> newUser;
currentUsersList.push_back(newUser);

// Adding a new reservation ID to a user

User user;
user.currentResIDs.push_back("R000005");
```

Using a vector ensures that common operations such as push_back() have an average time complexity of **O(1)**. Although, in rare cases that the vector needs to resize itself it can have a time complexity of **O(n).**

Stack

A **Stack** is a data structure that follows the **Last In, First Out (LIFO)** principle, meaning that the last element added to the stack is the first one to be removed. This structure is ideal for scenarios where it is necessary to process elements in reverse order, such as traversing through items in a history or undo operations.

+ Example:

```
stack<string> resIDStack;
// Push all reservation IDs into the stack
for (auto &res : user.resID)
{
   resIDStack.push(res);
}
```

Since stack follows the **Last In - First Out** Principle, it means that it always and only will work with the top most element. Therefore, the time complexity in this data structure is **O(1)** for both **push()**, **pop()** and **top()**.

Queue

This data structure is utilized for functions that require a First In - First Out (FIFO) principle. For a bus management system, this data structure comes in handy when working with a waitlist, a list of awaiting customers.

+ Example:

```
queue<waitlist> waitlistQueue;

waitListQueue.enqueue(user);
if (busSeatAvailable())
{
   waitListQueue.dequeue(user);
}
```

Since stack follows the **First In - First Out** Principle, meaning that it always and only will work with the head element. Therefore, the time complexity in this data structure is **O(1)** for both **enqueue()** and **dequeue()**.

Hashmaps

A **hashmap** (or **unordered_map** in C++) is a powerful data structure designed for efficient data storage and retrieval. It stores data as **key-value pairs**, where each key is unique and associated with a corresponding value. The primary advantage of a hashmap is its ability to quickly locate a value using its key, thanks to the use of a **hash function**. The hash function computes a **hash value** based on the key, which is then used to determine the storage location (or **bucket**) for the associated value.

In our program, we leverage the **nlohmann json** library, which utilizes std::unordered_map (a built-in C++ hashmap implementation) under the hood. This ensures that key lookups and value retrievals are performed quickly and efficiently, allowing our program to handle large amounts of data with minimal performance overhead.

+ Example:

```
// Create a simple unordered_map with string keys and int values
std::unordered_map<std::string, int> userAge;

// Adding data
userAge["Alice"] = 25;
userAge["Bob"] = 30;
```

```cpp
// Accessing a value by key
std::cout << "Alice's age: " << userAge["Alice"] << std::endl;
```

This method enables **constant-time complexity (O(1))** for most key lookups, making the hashmap an extremely efficient data structure for accessing values quickly.

+ Analysis on using hashmap in the nlohmann json library:

Given the following json structure:

```
"bus":{
   "busType": "Standard Bus",
   "departureTime": "2024-12-01 06:00",
   "id": "B0001",
   "route": {
     "from": "Phnom Penh",
     "to": "Siem Reap"
   },
   "seatCap": 6,
   "seatLeft": 6,
   "seatPrice": 10,
   "seats": [
     { "seatNum": 1, "status": "available" },
     { "seatNum": 2, "status": "available" },
     { "seatNum": 3, "status": "available" },
     { "seatNum": 4, "status": "reserved" },
   ]
 }
```

In general, a key lookup in this library has a constant-time complexity O(1). However, when traversing through levels to access a specific data point, the time complexity adds up. For example, retrieving the key **"busType"** is an O(1) operation. But to get the origin of the bus, we first need to access the **"route"** key (O(1)) and then the **"from"** key (O(1)) at the next level, resulting in an overall time complexity of O(L), where L is the number of levels you need to traverse.

Another case arises when searching through an array. For instance, to find the seat that is reserved, we access the "seats" array in O(1) time and then perform a linear search to find a seat where the "status" is "reserved". This search operation has a time complexity of O(n), where n is the number of elements in the "seats" array.

Thus, using this library, the time complexity is generally:

- **O(1)** for direct access.
- **O(L)** for accessing a value at a level > 1.

- **O(n)** when performing a linear search in an array.

3.3.5. Algorithms

+ Searching Algorithms

Hashmap Search

By using a hashmap (or unordered_map), we can perform lookups in **average O(1) time**. The hashmap allows us to efficiently retrieve values associated with a specific key through hashing, making it ideal for fast and direct lookups.

Linear Search

Linear search is used when the key is unknown or embedded within an array. In this approach, we examine each element in the array one by one until we find the target. This results in an **O(n)** time complexity.

Combination of Hashmap and Linear Search

Using a hashmap optimizes search operations by providing **O(1)** access to data via high-level keys, preventing the need for deep, unknown searches. For example, in a hierarchical structure with **n** elements, a linear search could take **O(n)** time, as each element must be checked one by one. Without a hashmap, if you need to search across multiple levels (e.g., **m** levels), the time complexity could grow exponentially, approaching **O(m * n)**. However, by using a hashmap, we avoid this deep search, ensuring the operation remains **O(1)** for direct access and **O(n)** only when performing a linear search within an array.

+ Secure Hashing Algorithm 1 (SHA-1)

Secure Hash Algorithm 1 (SHA-1) is a cryptographic hashing algorithm widely used in various applications to ensure data integrity and security. In the context of the bus management project, SHA-1 plays a pivotal role in securely managing sensitive user information, particularly in handling passwords.

SHA-1 is designed to transform an input (such as a user's password) into a fixed-size, unique 160-bit hash value, often represented as a 40-character hexadecimal string. This transformation is a one-way process, meaning the original input cannot be retrieved from the hash value. Here's how it operates within the bus management system:

In the project, when users register or update **their passwords,** the system applies the **SHA-1 algorithm** to the plain-text password. This ensures that passwords are not stored in plain text in the system's database, significantly reducing the risk of exposing sensitive information in the event of a data breach.

Step-by-Step Hashing Process

- **Pre-processing**: The algorithm begins by padding the input password to ensure its length is a multiple of 512 bits. Padding involves appending a 1 bit, followed by zero bits, and finally, the length of the original password.
- **Dividing into Blocks**: The padded input is divided into 512-bit blocks.
- **Initialization**: SHA-1 uses five fixed 32-bit words (initial hash values), which are updated as the algorithm processes the input.
- **Main Hashing Loop**: For each 512-bit block, the algorithm performs a series of logical and bitwise operations over 80 rounds. These operations include XOR, AND, OR, and bit rotations, which mix the input data thoroughly to produce the final hash value.
- **Output**: After processing all the blocks, the five 32-bit words are concatenated to form the final 160-bit hash value.

Verification During Login

When a user logs in, the system hashes the input password with SHA-1 and compares it with the stored hash in the database. If the two hash values match, the system grants access, ensuring secure and reliable password verification.

+ Custom Algorithms/Solutions

Generating a new ID

Since entities such as Bus, User and Reservation have their ID as Primary Keys, we need to implement a solution to generate new IDs for the ever growing database to accommodate new users.

The Identification for these entities have the following format: **“X000000”**. The “X” is to identify each type of ID (User, Bus, or Reservation) and the following 6 digits are for the unique number. Our task is to implement an algorithm where given this base string and the database, generate a new ID by incrementing the lastly produced ID by one.

The following is the code snippet of the solution to this problem:

```
int nextID = objArr.size();
string nextID_string = to_string(nextID);
string baseID = "X000000";
int start = baseID.size() - nextID_string.size();
int j = 0;
for (int i = start; i < baseID.size(); i++)
{
     baseResID[i] = nextID_string[j];
     j++;
}
```

The solution yields a time complexity of O(n) where **n = baseID.size() - start**, however, since baseID.size() = 7 thus, **n < 7**. Due to n being lesser than 7 (a generally small number), this implementation can be considered a constant-time operation **O(1)**.

Display a list of previous bookings ordered by latest

Since reservationID's are appended to the back of a vector array we can implement a stack data structure to display the lastly added item first.

The solution works as follows:

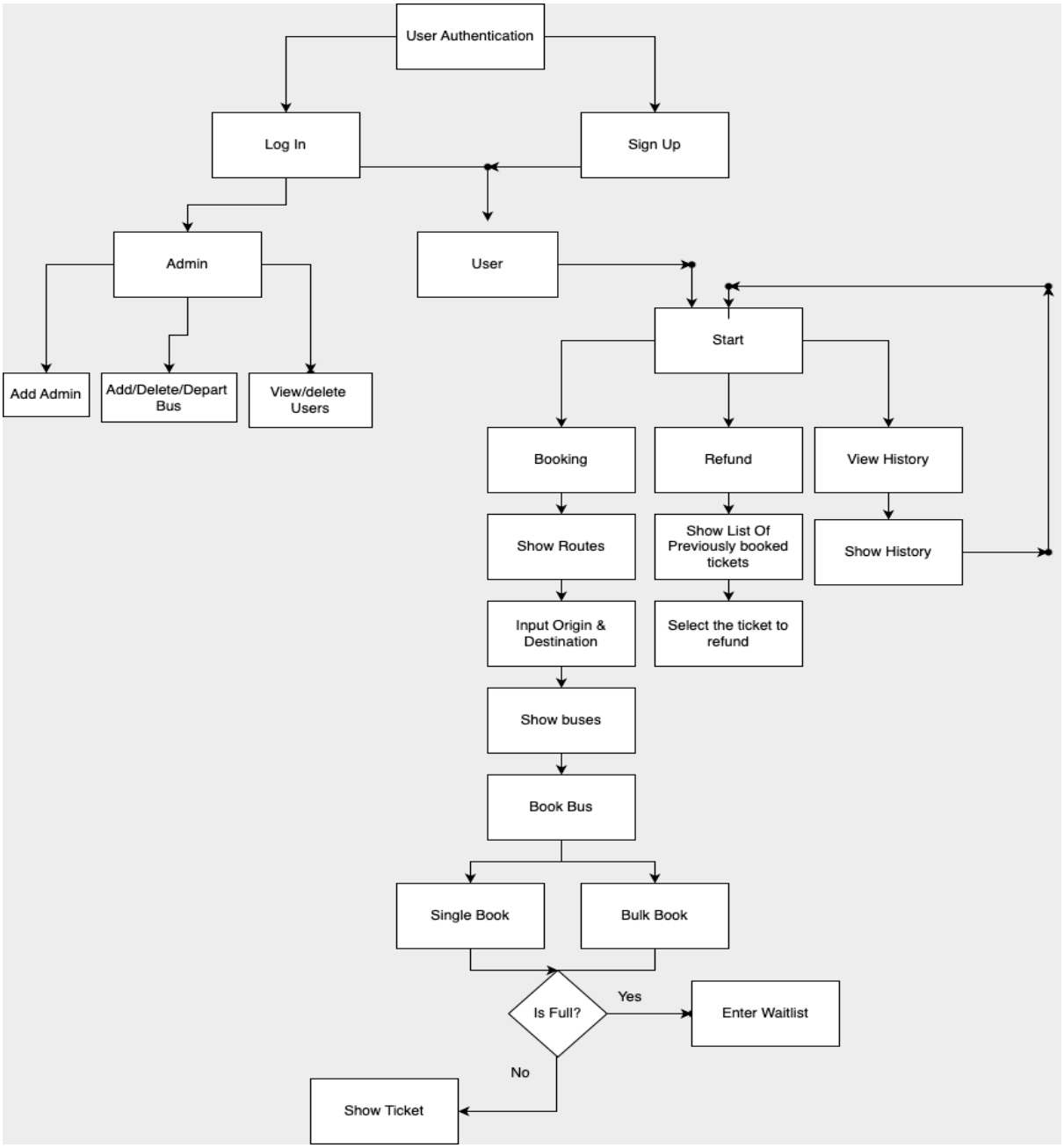
```
stack<string> resIDStack;

// Push all reservation IDs into the stack
for (auto &res : user.resID)
{
   resIDStack.push(res);
}

while (!resIDStack.is_empty())
{
   processAndPrintResID(resIDStack.top());
   resIDStack.pop();
}
```

Here, using the stack efficiently handles the problem by ensuring that we access the latest booking first, without needing to manually reverse the order of the reservations. The time complexity of this solution is **O(n)**, where **n** is the number of reservations, as we need to push and pop each reservation ID exactly once.

Note: these are only some of the algorithms that are made for general purposes. In 3.5. Development Process we will document more in-depth on specific solutions and approaches.

3.3.6. Program Flow

Below is a high level look at how the program as a whole looks when all the components defined above work together:



3.4. Task Delegation & Collaboration

Efficient task delegation and collaboration were critical to this project. By breaking down the tasks into manageable components and utilizing version control, the team ensured that development was streamlined, conflicts were minimized, and progress was well-documented.

3.4.1. Task Delegation

To ensure smooth development and equal contribution from all team members, tasks were divided based on relevancy of related branching operations of each task. Below is an overview of the task delegation process:

Project Planning and Design:

Assigned to **Ory Chanraksa**, the **Project Planning and Design** phase focused on defining the system's scope, requirements, and overall architecture to ensure a solid foundation for development. This involved identifying key entities like **User**, **Bus**, and **Reservation**, along with their relationships, and outlining core functionalities such as user management, seat reservations, and booking history. A modular design approach was adopted to decouple components, making the system more maintainable and scalable. Additionally, a JSON-based database schema was created to store and efficiently manage data, with relationships optimized for quick lookups and updates.

User Authentication:

To begin the development of this project, we have pinpointed a big concern which is that for users to do any of the main operations there needs to be data that is associated with that user since we are not building a guest-based system, there must be user credentials and sensitive information to be secured and processed. Thus, the initial step was to create a **"User Authentication"** feature. This task was assigned to **Ory Chanraksa** to figure out this initial step so that consequential processes ahead will be developed smoothly.

User Functionalities:

As previously defined the user has 3 core actions: **Reserve**, **Refund**, **View History**.
Therefore, 2 of the team members are assigned to complete these functions.

- The **Reserve** operation is a method of the user where it will handle all things related to the reservation of seat(s) on a bus. This task was assigned to **Sao Visal**.
- Similarly, the **Refund** operation is also another method of the user class where it does the reverse of the reserve method therefore, we thought it was reasonable to delegate this task to **Sao Visal**.
- The **View History** operation was handled by **Ory Chanraksa** as it was fairly straightforward and would be the third function to be completed after reserve and refund, assuring that those two worked properly.

Admin Functionalities:

As previously defined the user has 5 core actions:**Add Admin**, **Add Bus, Delete Bus**, **View User, Delete User**. Therefore, 2 of the team members are assigned to complete these functions.

- The **Add Admin** operation is a method of the admin where it will handle the addition of a new admin. This task was assigned to **Seang Darong**.
- The **Add Bus** operation [TO WRITE].
- The **Delete Bus** operation [TO WRITE].
- The **View Users** operation is a function to get user information like name, email user id, and password reservation.This function has an option to get one user by input a userID or get all the existing user information . This task was assigned to **Seang Darong**.

- The **Delete User** operation is a method of the admin where it allows the admin to delete a user by their ID.This task was assigned to **Seang Darong**.

3.4.2. Collaboration and Git Version Control

+ Collaboration Strategy:

Effective collaboration was facilitated through the use of Git as the version control system. Below are the key practices employed:

Branching Strategy

The team adopted a branching strategy to ensure parallel development. Each team member worked on their respective features in dedicated branches (e.g., feature/user-authentication, feature/reserve, feature/admin-addAdmin) to avoid conflicts in the main codebase.

Commit Best Practices

Commits were made frequently, with clear and descriptive messages (e.g., Add algorithm for ID generation, Fix seat availability logic). This made it easier to track changes and debug issues.

Pull Requests and Code Reviews

Changes were merged into the main branch only after submitting pull requests. Team members reviewed each other's code to ensure quality, identify potential bugs, and promote knowledge sharing.

Collaboration Tools

The team used platforms like **GitHub** for repository hosting, issue tracking, and discussions. This centralized all collaboration efforts and maintained transparency.

+ Collaboration Workflow

User Authentication

The first branch created, **origin/user-authentication**, focused on implementing user authentication, enabling users to log in or sign up securely. It laid the foundation for subsequent features requiring user accounts.

Admin User Management

The **origin/user-authentication-admin** branch extended the authentication functionality to include admin-specific actions. This branch primarily focused on managing user-related tasks, such as viewing or modifying user data.

Admin Bus Management

Building on the admin functionality, **origin/user-authentication-admin-busRelated** was created to implement bus-related administrative tasks. This included actions like adding or removing buses, ensuring a seamless workflow for bus management.

Reservation and Refund

The **origin/feature/reserve** branch introduced essential features for the reservation and refund process. This functionality was pivotal in managing bookings and processing refunds efficiently.

User History

To allow users to review their past bookings, the **origin/feature/viewHistory** branch was created. This feature added value by enabling users to track their booking history easily.

UI Improvements

The **origin/UI-improvement** branch focused on enhancing the user interface by adding colors, error messages, and ASCII art. These changes were aimed at improving the overall user experience and making the application visually appealing.

Testing

The **origin/test** branch served as a testing environment where new features were merged and tested for compatibility and stability before being pushed to the main branch.

Final Merging

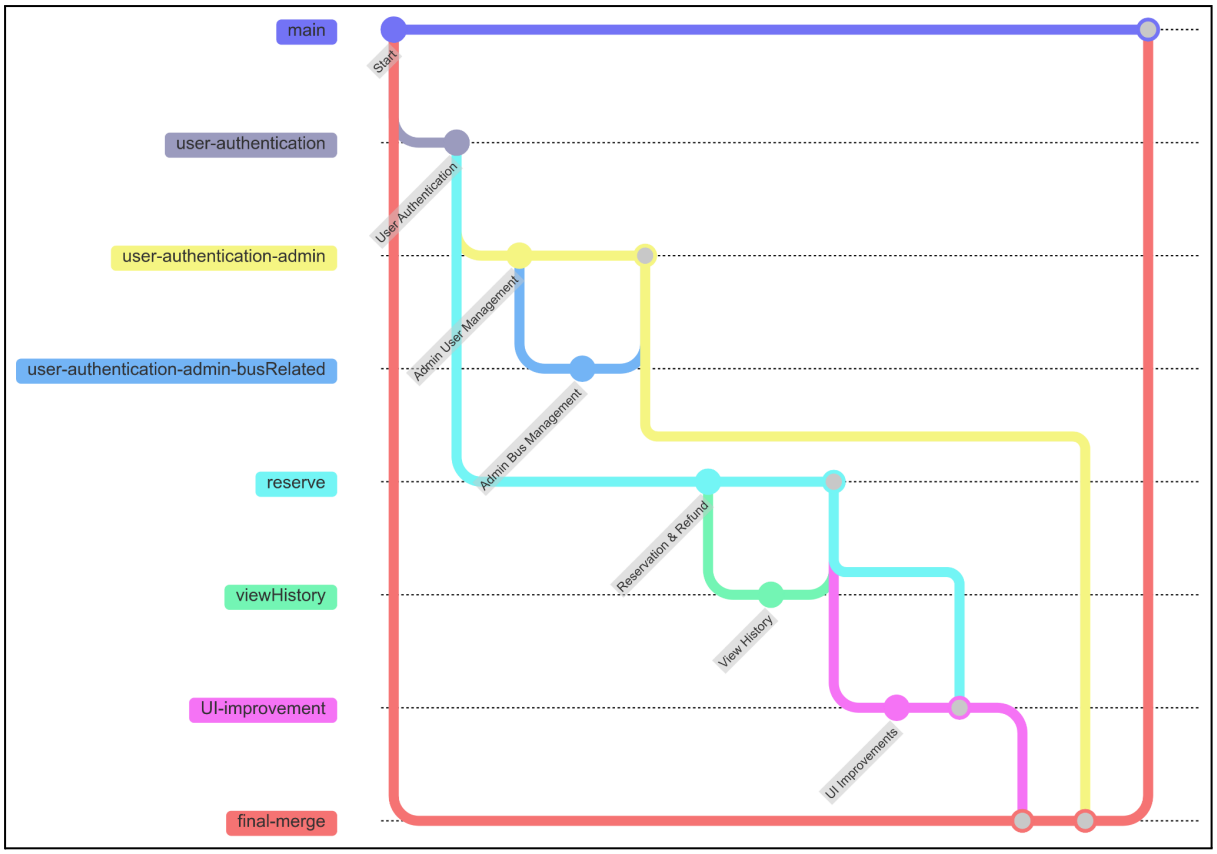
The **origin/final-merge** branch acted as a third backup during the merging phase. It was used to merge main2 with test, ensuring all updates were consolidated and stable before final deployment.

Backup Branches

To ensure code safety and stability, the **origin/main2** branch was maintained as a backup for the main code. It provided a reliable fallback during development, especially when testing or merging new features.

Main Branch

Finally, the **origin/main** branch housed the production-ready code. After all features were thoroughly tested and merged, this branch contained the stable version of the application ready for deployment.



3.5. Development Process

3.5.1. User Functionalities

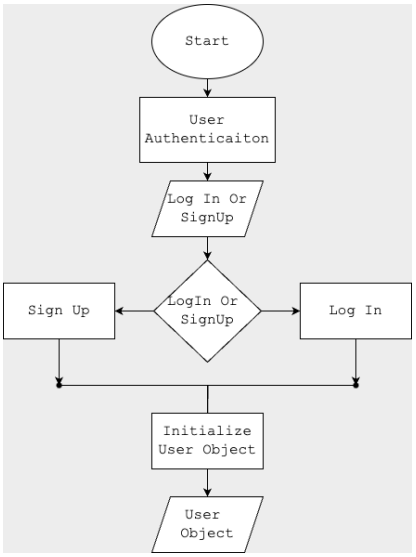
Note: The functions below are documented based on their logical flow, including input handling, decision-making, algorithms, and database interactions. Minor error-handling functionalities are not detailed, as they are fundamental and built into the system's robust architecture. (Refer to 3.3.2. Helper Libraries/Dependencies for validation techniques.)

+User Authentication

To access the program, users must either log in or sign up. This functionality is managed by the **System** class. As mentioned earlier, the **System** class handles user authentication, ensuring that the provided credentials are valid for operations involving sensitive data or registering new users into the database for future reservations. Referring to the class definition above, we have designed a primary public method, **authenticateUser()**, which serves as an abstraction layer for two additional methods: **logIn()** and **signUp()**. Below is a detailed explanation of the functionality and flow of each method, along with their supporting processes.

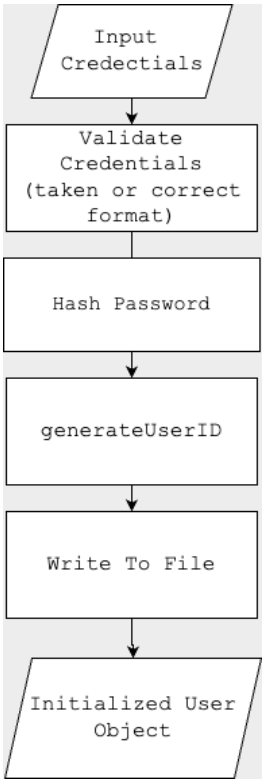
authenticateUser()

The authenticateUser() method serves as a large encapsulation function that abstracts away from its conditional states. It begins by presenting a menu to the user, allowing them to choose whether to log in or sign up. From that choice, this function will delegate each task to the private methods **logIn()** or **signUp()** based on a conditional switch statement. After the completion of either of the methods, this function will return a user object. The returned object is initialized with user attributes derived from the database via the constructor.



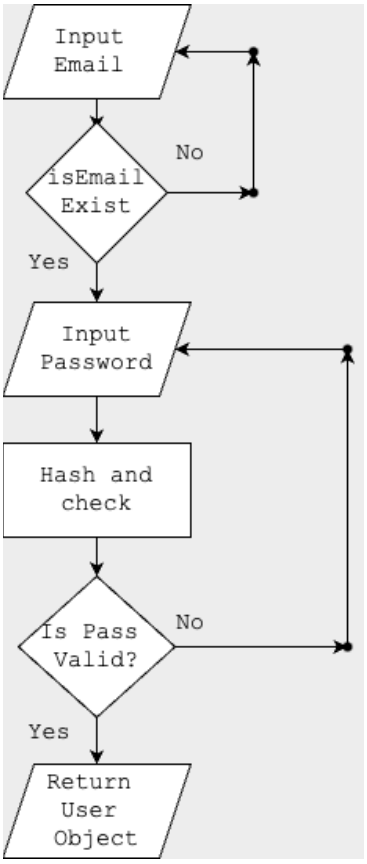
signUp()

The **signUp()** function is to facilitate the authentication of new users; it includes basic sign up capabilities such as inputting name, age, email and password as well as password confirmation. Additionally, at this stage of the program it employs the significant **SHA-1** hashing algorithm to encrypt the password. The last stage of this method is to generate a unique user ID for that newly signed in user.



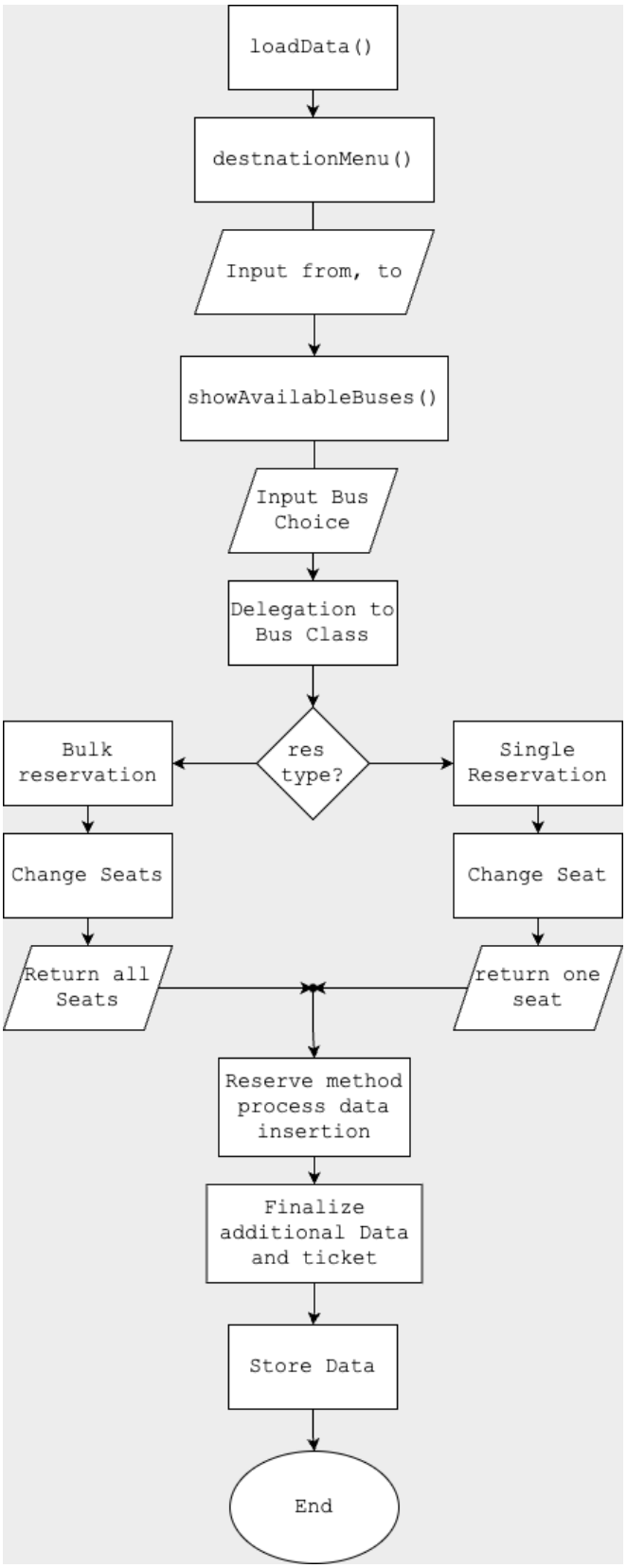
logIn()

This function is responsible for the authentication of users who are intending to access the program. This function gets 2 inputs: **Email** and **Password**. The email is checked in the database for its validity and the inputted password is hashed with the **SHA-1** hashing algorithm to then be compared to the hashed password associated with that email in the database. If entered credentials are not correct, the system will allow entrance therefore, prohibiting unauthenticated users to do operations and securely protect the program.



+ Reserve

The reserve method is responsible for displaying information, bus selection and making reservations by delegating seat reservations to the bus class and storing data to the database.



Before documenting each method's flow, it is best practice to explain how each method gets its data for access and manipulation. The private method responsible for this is the method **loadData()** function.

Void loadData()

It works by reading from the Data.json file and storing everything read from that file to a json object called Data. Then, from this Data object we parses each of its high level objects, thus we get:

- Json users = data[“users”]
- Json buses = data[“buses”]
- Json reservations = data[“reservations”]
- Json routes = data[“routes”]

```
ifstream readData(dataFilePath);
if (!readData.is_open())
{
    cerr << "Couldn't open file" << endl;
    return;
}
json allData;
readData >> allData;
this->data = allData;
this->users = allData["users"];
this->buses = allData["buses"];
this->reservations = allData["reservations"];
this->routes = allData["routes"];
```

Time Complexity: O(1).

1. Displaying Information

Void destinationMenu()

This function is for displaying routes that are available. It works by iterating through the routes object and using “from” as the base it then will find the destinations that originate from that origin.

```
for (const auto &route : routes)
    {
        for (const auto &to : route["to"])
        {
            cout << "*\t" << route["from"] << "\t -------> \t" << to << "\t\t*" <<
endl;
        }
    }
```

This implementation results in a time complexity of **O(m x n)** where **m** represents the number of origins and **n** represents the number destinations based on the origin.

Vector <int> showAvailableBuses(string origin, string destination)

This method takes in 2 parameters: **origin** and **destination** and returns an array of indexes. This function uses a loop that iterates through all the buses and uses the printBus() function to print the bus that matches the origin and destination. When a bus matches the origin and destination, the printBus() function will return the index of that bus if not it will return -1. After searching through all the bus, all the indexes returned/push_backed onto the array will then be validated by only getting the valid indexes (!-1). The validated array will then be returned for further operation.

```
for (const auto &bus : buses)
   {
       allIndex.push_back(printBus(bus, f, t, busIdx)); //));
       busIdx++;
   }
// .....
for (int i : allIndex)
   {
       if (i != -1)
       {
           validIndex.push_back(allIndex[j]);
       }
       j++;
   }
return validIndex;
```

Due to the iteration of the buses array (n buses) and validating the indexes (size = numberOfBuses or n) thus, **Time complexity: O(2n)**.

2. Bus Selection

Bus selectBus(vector <int> busIdx)

This function is used for the prompt to get a bus choice and map the choice to the correct bus to modify/book. It simply maps the inputted integer to the index of the busIdx array and the value of that index will be the index of the bus that is to be modified. Example, If **choice is 1** (selecting the first bus in the list) then we decrement that inputted choice by 1 we get 0 and get the element in the busIdx parameter, for instance 3 which means that we are going to modify the 4th bus in the buses array or in the database. Lastly, get the data of that bus which is at **buses[chosenBusIdx]** and fill the constructor for a bus object for it to use for seat modification when returned.

```
    int choice;
    cout << "Select a bus \n> ";
    cin >> choice;
    choice = choice - 1;
    int busIndex = busIdxArr[choice];

    busToModify = buses[busIndex];
// below contains direct access of data of that bus for the constructor
```

Since there are no loops and all individual steps are O(1), the overall complexity is O(1). Even though multiple constant-time operations are involved, they don't scale with input size, so **Time Complexity: O(1).**

3. Task delegation to Bus Class

The reserve method delegates tasks to the Bus class for itself to display its own information with the method showSeatLayout() and to change seat statuses. Based on the inputted reservation decision, the bus class can utilize two methods: reserveSeat() or reserveSeats().

Reservation Type

The reserve function handles the direction of reservation type based on a booking type menu.

```
if (bType == 1)
    {
        chosenSeat = bus.reserveSeat();
    }
    else if (bType == 2)
    {
        seatsOfBus = bus.reserveSeats();
    }
```

Time complexity: O(1).

Void showSeatLayout()

This method will display a representation of the bus seat layout. It loops through the seats object of that bus to print rows and columns of seats enclosed in brackets **“[n]”** as well as mark taken seats as **“[X]”.**

```
for (const auto &seat : seats)
    {
        int seatNum = seat["seatNum"];
        if (seat["status"] == avail)
        {
```

```
            cout << "[ " << seatNum << " ]";
        }
        else
        {
            cout << "[ X ]";
        }
        if (seatNum % 2 == 0)
        {
            cout << "\t";
        }
        if (seatNum % 4 == 0)
        {
            cout << "\n";
        }
    }
```

Time complexity: O(n). Where **n** represents the number of seats there are on the bus.

Json reserveSeat()

This function is capable of prompting the user to input a seat number that they wish to reserve while also handling the checking of inputting at a taken seat.

```
    int seatNum;
    while (1)
    {
        cout << "Select a seat\n> ";
        cin >> seatNum;
        if (isSeatAvailable(seatNum))
        {
            break;
        }
        else
        {
            cout << "Error: Seat unavailable\n";
        }
    }

    json seatToReturn;
    for (auto &seat : seats)
    {
        if (seat["seatNum"] == seatNum)
        {
            seat["status"] = "reserved";
            seatToReturn = seat;
```

```
            break;
        }
    }

    return seatToReturn;
```

The main time constraint of this method is the isSeatAvailable() function that loops that linearly searches the array for that seatNumber and checks its status and the for loop to set the status of that seat to “reserved”. This implementation yields a **Time Complexity: O(n^2)**. This is because these 2 functions loop through the same number of seats from the seats array.

Json reserveSeats()

This function allows the user to input multiple seat numbers for bulk reservation. It ensures that each seat entered is valid and not already reserved. The function prompts the user to input seat numbers one by one, verifies their availability, and reserves them if they are free. Any attempts to reserve already-taken seats are flagged, and the user is prompted to re-enter a valid seat number. The process continues until all desired seats are successfully reserved.

```
vector<int> seatNumberArr;
vector<json> seatsArr;
int seatNum;
cout << "Enter the number of seats to book\n> ";
int numberOfSeatToBook;
cin >> numberOfSeatToBook;
for (int i = 0; i < numberOfSeatToBook; i++)
{
    while (1)
    {
        cout << "Seat: " << i + 1 << ": ";
        cin >> seatNum;
        if (isSeatAvailable(seatNum))
        {
            seatNumberArr.push_back(seatNum);
            break;
        }
        else
        {
            cout << "Seat Unavailable\n";
        }
    }
}
int i = 0;
for (auto &seat : seats)
{
```

```
        if ((i < seatNumberArr.size()) && (seat["seatNum"] == seatNumberArr.at(i)))
        {
            seat["status"] = "reserved";
            i++;
        }
        if (i >= seatNumberArr.size())
        {
            break;
        }
    }
    return seats;
```

Since this method takes in the number of seats to book (q) and iterates q times to gather input while checking the availability of each seat, the first part of the function has a time complexity of **O(q × n)**, where n is the number of total seats. Once all inputs are gathered, the function runs through the seats array to update the status of the selected seats, which has a time complexity of **O(n)**. Therefore, the overall **Time Complexity**:**O(q × n)**, as the input loop dominates.

4. Collecting and Storing Data

Since the reserveSeat() and reserveSeats() methods of the Bus class only return the seat or seats json object, we will have to insert and access data for accurate file writing.

+ Collecting Data

Single Reservation

This implementation only returns one singular seat thus, we need to search through the seats object of the busToModify object to insert that seat to all the seats after the function completes its execution.

```
        chosenSeat = bus.reserveSeat();
        int seatIdx = 0;
        for (const auto &seat : busToModify["seats"])
        {
            if (seat["seatNum"] == chosenSeat["seatNum"])
            {
                busToModify["seats"][seatIdx] = chosenSeat;
                break;
            }
            seatIdx++;
        }
        // changing some attributes
        int seatLeft = busToModify["seatLeft"];
```

```
        seatLeft--;
        busToModify["seatLeft"] = seatLeft;
```

The procedure above involves the usage of a loop to linearly search the seats array to insert/replace that seat. Thus, we receive a **Time Complexity: O(n)**. n represents the number of seats there are on the bus.

Bulk Reservation

In contrast to the single reservation method, the **reserveSeats()** method returns all the seats of that bus after multiple seats were changed.

```
        seatsOfBus = bus.reserveSeats();
        int seatLeft = bus.getSeatLeft();
        cout << seatLeft << endl;
        busToModify["seats"] = seatsOfBus;
        busToModify["seatLeft"] = seatLeft;
        seatsChangedArr = bus.getSeatNumChanges();
```

Due to direct access, this operation yields a **Time Complexity: O(1).**

+ Data Storage

This stage involves ensuring all changed and unchanged data is finalized before storing to the Data.json file. It includes finding the user and replacing it with the modifiedUser object. Finding the Bus and replacing it with modified values to ensure data integrity when written back to the database.

```
    int busIdx = 0;
    for (const auto &bus : buses)
    {
        if (bus["id"] == busToModify["id"])
        {
            break;
        }
        busIdx++;
    }

    buses[busIdx] = busToModify;
    modifiedUser["age"] = this->age;
    modifiedUser["email"] = this->email;
    modifiedUser["id"] = this->userID;
    modifiedUser["isAdmin"] = this->isAdmin;
    modifiedUser["name"]["firstName"] = this->firstName;
    modifiedUser["name"]["lastName"] = this->lastName;
```

```
modifiedUser["password"] = this->password;
modifiedUser["resID"] = this->resID;

int userIdx = 0;
for (const auto &user : users)
{
    if (user["id"] == this->userID)
    {
        break;
    }
    userIdx++;
}

users[userIdx] = modifiedUser;
data["users"] = users;
data["buses"] = buses;
data["reservations"] = reservations;

ofstream storeFile(dataFilePath);
if (!storeFile.is_open())
{
    cerr << "Couldn't open file";
}

storeFile << data.dump(4);
storeFile.close();
```

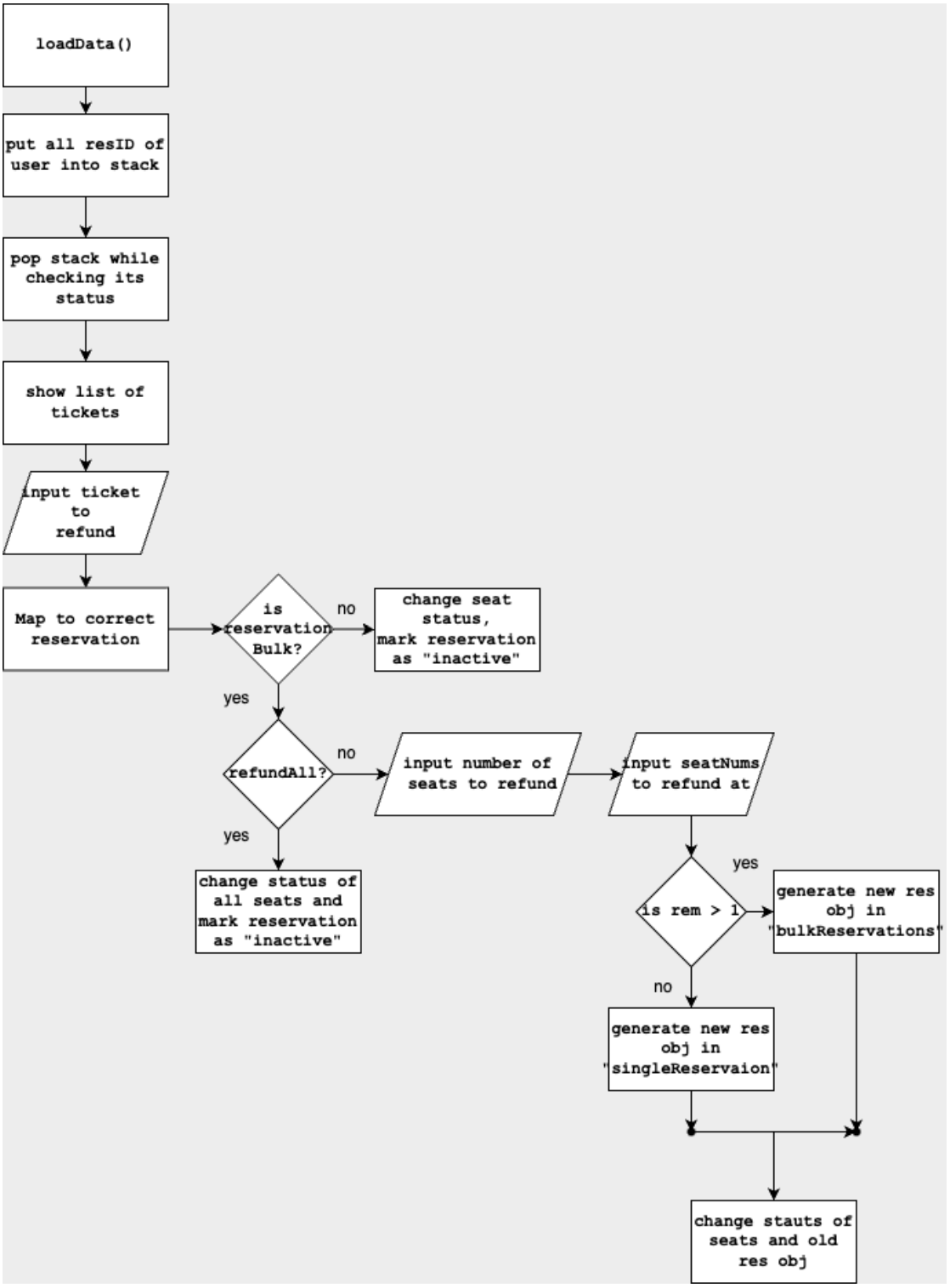
Due to the use of 2 separate loops to find the correct user and the correct bus. This function yields a **Time Complexity: O(n) + O(m). n** for the number of buses and **m** for the number of users.

5. Finalization:

This stage occurs prior to the file-writing phase and is responsible for finalizing the data and process. It includes tasks such as generating ***(refer to 3.3.5. Algorithms)*** and appending a reservation ID to the user, generating a ticket, and creating a QR code. While the QR code serves no practical function within the system, it is included as a fun and optional feature, adding a creative touch to the reservation process. The primary focus of this stage is to ensure all relevant data is properly associated and ready for storage, enhancing the user experience with an added layer of personalization.

Refund

This feature is responsible for facilitating the possibility of a user choosing to cancel previous reservations. Due to the fact that, the before mentioned Reserve method situating for 2 main types: Single Reservations and Bulk Reservations. Thus, the refund process must accommodate these 2 demands while preserving the integrity of the database ensuring no data loss, affecting other users or reservations.



1. Ordering the printing of tickets

As part of our objective to create a user-friendly interface, we decided to use a **stack** data structure to store the reservation ID in which the latest of this array will then be at the top of the stack. The top will then be processed first thus, following the **LIFO** principle.

```
for (string r : this->resID)
{
    resIDStack.push(r);
}
```

Time Complexity: O(n).

2. Printing list of refundable tickets

This method contains the helper method **refundList(),** in which it is responsible for printing a list of refundable tickets, i.e. resID with status **"active".** This Function will then return the indexes of that resID of that user that is refundable. Within this helper method lies another method called **printRefund()**. Using a loop, the topmost element is passed into the **printRefund()** function to process if the ticket is refundable, if true return the index of the resID, if false then return "" (which will get refined later for choice mapping).

```
while (!resIDStack.empty())
{
    allRes.push_back(printRefund(resIDStack.top()));
    resIDStack.pop();
}
for (string i : allRes)
{
    if (i != "")
    {
        validRes.push_back(i);
    }
}
```

After this function is finished, it will return a vector array validRes. Ex: validRes = [1,4,5] which indicates that the **second**, **fifth**, and **sixth** element in the resID array of the user are refundable.

Time Complexity: O(n) + O(m).

3. Choice Mapping

This process is going to validate the choice made by the user by viewing the refund list or list of refundable tickets labelled numerically 1,2,3. However, the choice 1,2,3 needs to be verified and mapped correctly to the appropriate index of the reservationID to then access associated data of it to change other objects as well, i.e. busID and seatNumbers. The function **inputRefund()** function returns the choice entered by the user. This choice will get decremented and use the result to access the element at that index.

```
int choice = inputRefund();
string resIDToRefund = refundableResID.at(choice - 1);
```

Time Complexity: O(1).

4. Retrieving Seat Numbers and Bus to refund at:

A function called **getSeatNumsToRefund()** will take in the resIDToRefund as an argument and search for the seat numbers booked associated with that reservationID. Another function **getBusIDToRefund()** will retrieve the busID via the resIDToRefund.

```
vector<int> seatNumsToRefund = getSeatNumsToRefund(resIDToRefund);
string busIDToRefund = getBusIDToRefund(resIDToRefund);
```

Time Complexity: O(2n).

5. Initializing a bus object for seat manipulation

By having the 2 data above, we now can perform the refund. However, this task is delegated to the Bus class to change its own seat status and other attributes. Thus, we need to loop through the array of buses to find the bus with that ID. Finally, we can construct a bus object with the attributes from the bus json object.

```
for (auto &bus : buses)
        {
            if (bus["id"] == busIDToRefund)
            {
                busToModify = bus;
                string busType = busToModify["busType"];
                string dpTime = busToModify["departureTime"];
                string busID = busToModify["id"];
                json route = busToModify["route"];
                int seatCap = busToModify["seatCap"];
                int seatLeft = busToModify["seatLeft"];
                int seatPrice = busToModify["seatPrice"];
```

```
                json seats = busToModify["seats"];
                Bus bus(busType, dpTime, busID, route, seatCap, seatLeft, seatPrice,
seats);
                if (!isResIDBulk(resIDToRefund))
                {
                    bus.refundSeat(seatNumsToRefund);
                }
                else
                {
                    bus.refundSeats(seatNumsToRefund);
                }
            }
}
```

6. Refund Type and Seat Changes

By identifying the ID to see if it is a single or bulk type, we can determine the path that the refund process can go.

+ Single Refund

```
int seatToRefund = seatNumsToRefund.at(0);
   for (auto &seat : seats)
   {
       if (seat["seatNum"] == seatToRefund)
       {
           seat["status"] = "available";
           seatNumsChanged.push_back(seat["seatNum"]);
           break;
       }
   }
   return seats;
```

Time Complexity: O(n). n represents the number of seats on the bus.

+ Bulk Refund

Since this process needs to handle multiple cases and edge cases, we need a menu to prompt the user for their choice. The user can choose to refund all seats or partially.
Refunding All Seats

```
       int i = 0;
       for (auto &seat : seats)
       {
           cout << i << " ";
```

```
                    if (i < seatNumsToRefund.size() && (seat["seatNum"] ==
seatNumsToRefund.at(i)))
            {
                seat["status"] = "available";
                i++;
            }
            if (i >= seatNumsToRefund.size())
            {
                this->seatLeft += seatNumsToRefund.size();
                break;
            }
        }
        return seats;
```

Time Complexity: O(n). n represents the number of seats on the bus.

Refunding partial seats:

For this case, we had to handle situations and different data manipulation. To easily demonstrate these cases, let N be the number of seats that are refundable and n be the number of seats that they user choose to refund. We have 2 cases: N - n > 1 and N - n = 1. "**N - n > 1"** indicates that the remaining seats that the wish to retain is larger than 1 or in other words it is still a bulk reservation. However, for the case of **"N - n = 1",** this means that there are only 1 remaining seat thus, making it a single reservation.

```
while (1)
        {
            cout << "\033[36m\nEnter number of seats to refund \033[0m\n\n> ";
            cin >> numOfSeatsToRefund;
              if (numOfSeatsToRefund > seatNumsToRefund.size() || numOfSeatsToRefund
<= 0)
            {
                cout << invalidInputMessage;
            }
            else
            {
                break;
            }
        }

        for (int i = 0; i < numOfSeatsToRefund; i++)
        {
            while (1)
            {
                cout << "\033[36mSeat: \033[0m ";
```

```
                cin >> seatNum;
                isSeatNumExist = false;
                for (int i = 0; i < seatNumsToRefund.size(); i++)
                {
                    if (seatNumsToRefund[i] == seatNum)
                    {
                        isSeatNumExist = true;
                        break;
                    }
                }
                if (isSeatNumExist)
                {
                    break;
                }
                else
                {
                    cout << invalidInputMessage;
                }
            }
            wantedSeatNumbers.push_back(seatNum);
        }
        this->wantedSeatNums = wantedSeatNumbers;
        seatNumsChanged = seatNumsToRefund;
        for (int i = seatNumsChanged.size() - 1; i >= 0; --i) // Iterate backwards
        {
            for (int j = 0; j < wantedSeatNumbers.size(); ++j)
            {
                if (seatNumsChanged[i] == wantedSeatNumbers[j])
                {
                        seatNumsChanged.erase(seatNumsChanged.begin() + i); // Erase
safely
                    break;                                                  // Exit
inner loop to prevent further comparisons
                }
            }
        }
        int i = 0;
        for (auto &seat : seats)
        {
                    if  (i  <  wantedSeatNumbers.size()  &&  (seat["seatNum"]  ==
wantedSeatNumbers.at(i)))
            {
                seat["status"] = "available";
                i++;
            }
```

```
            if (i >= wantedSeatNumbers.size())
            {
                this->seatLeft += wantedSeatNumbers.size();

                return seats;
            }
        }
```

7. Collecting and Storing Data

Back to the refund method situated in the User class, both of the Bus's refund methods will return the entire seats object of that bus.

```
json seatsOfModifiedBus = bus.refundSeat(seatNumsToRefund);

busToModify["seats"] = seatsOfModifiedBus;
int seatLeft = busToModify["seatLeft"]; // gain one seat
seatLeft++;
busToModify["seatLeft"] = seatLeft;
json singleRes = reservations["singleReservations"];
for (auto &res : singleRes)
{
       if (res["id"] == resIDToRefund)
       {
              res["status"] = "inactive";
              break;
        }
}
reservations["singleReservations"] = singleRes;
storeData();
seccessRefundMenu();
```

```
json seatsOfModifiedBus = bus.refundSeats(seatNumsToRefund);
busToModify["seats"] = seatsOfModifiedBus;
busToModify["seatLeft"] = bus.getSeatLeft();
vector<int> seatNumRemaining = bus.getSeatNumChanges();
generateNewResIDForRefund(busIDToRefund, seatNumRemaining);
json bulkRes = reservations["bulkReservations"];
for (auto &res : bulkRes)
{
       if (res["id"] == resIDToRefund)
       {
           res["status"] = "inactive";
           break;
          }
```

```
}
reservations["bulkReservations"] = bulkRes;
storeData();
```

After the method returns the seats of the bus, we modify some minor attributes. After that the data is stored using the storeData() function. Additionally, for bulk refunds if the user chooses to only partially refund some seats, then the **generateNewResID()** will then take remainingSeats array size to determine whether to generate a new reservation object at "bulkReservation" or "singleReservation" or not do anything.

Time Complexity: O(1). This is due to direct access of Data and push_back() method to append a new reservation object.

View History

This is an important feature that allows users to see their reservation history without filtering whether it has departed or awaiting or refunded. This section heavily utilizes the **Stack** data structure which follows the **LIFO** principle to display reservations in reverse order or "sorted" by last booked.

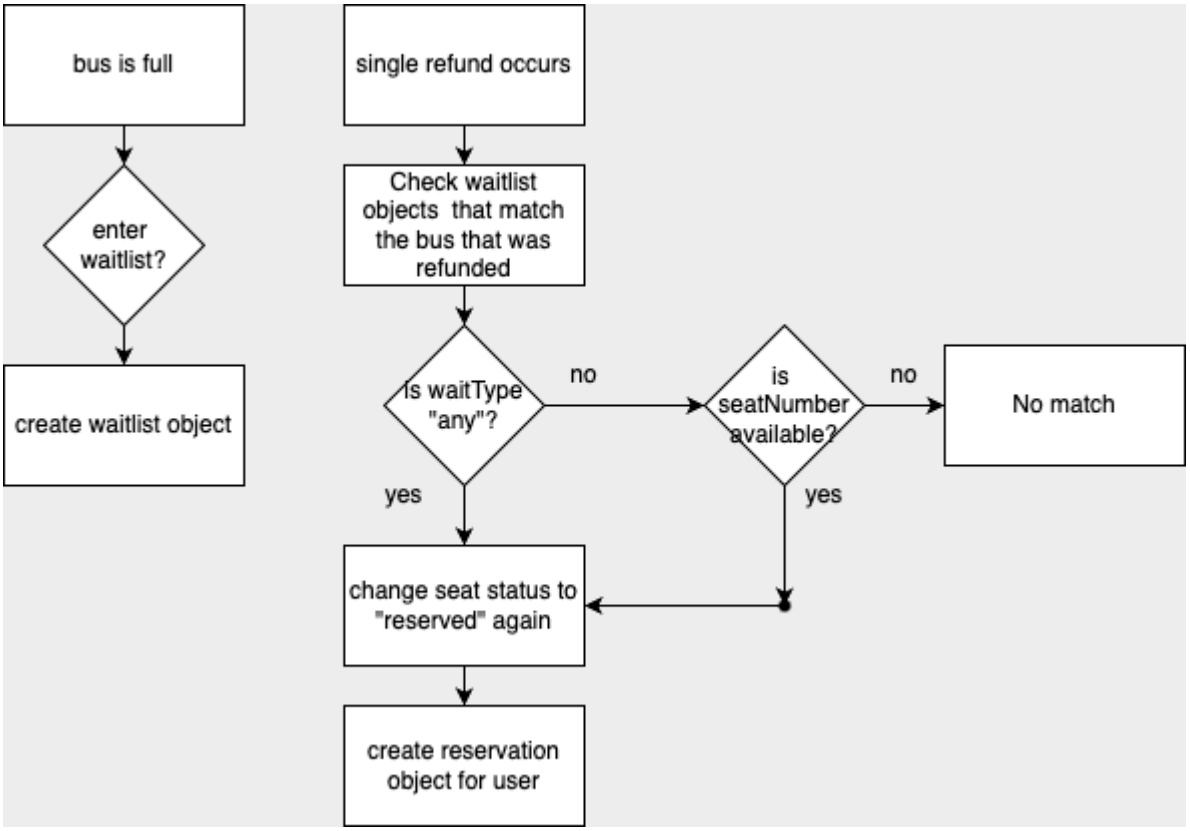
```
    loadData();
    stack<string> resIDStack;

    // Push all reservation IDs into the stack
    for (auto &res : this->resID)
    {
        resIDStack.push(res);
    }

    // Process each reservation ID from the stack
    cout << "\n USER RESERVATION HISTORY \n\n";
    cout << "\nORDERED BY LATEST\n\n";
    while (!resIDStack.empty())
    {
        string topResID = resIDStack.top();

        if (isResIDBulk(topResID))
        {
            json bulkRes = reservations["bulkReservations"];
            for (const auto &res : bulkRes)
            {
                if (res["id"] == topResID)
                {
                    string busID = res["busID"];
                    vector<int> seatNums = res["seatNumber"];
```

```
                printHistory(seatNums, busID);
            }
        }
    }
    else
    {
        json singleRes = reservations["singleReservations"];
        for (const auto &res : singleRes)
        {
            if (res["id"] == topResID) // Fixed comparison
            {
                string busID = res["busID"];
                vector<int> seatNum;
                seatNum.push_back(res["seatNumber"]);
                printHistory(seatNum, busID);
            }
        }
    }

    resIDStack.pop();
}
```

This implementation involved the use of the stack data structure, however, reservationIDs had to be pushed to the stack and then popped. Thus, we get a **Time Complexity: O(2n)** however, the push() and pop() method of the stack is only O(1).

Waitlist

The waitlist feature is the most recent update to the system. For now, it only accommodates single waitlists. This feature is situated in its own class called the **"Waitlist"** class. This class is responsible for the addition of users to the waitlist and processing the action of assigning a seat to the waitlisted user. This part of the program utilizes a queue-like structure, however, disregarding a strict queue. By using an array to store all waitlists with additions to the waitlist being at the back, thus the first elements in that array are prioritized resulting in a queue-like characteristic or a structure that follows the FIFO principle.



1. Addition to Waitlist

This step occurs when a bus is fully reserved hence, prompting the user to have a choice to add themselves to the waitlist by inputting the seat number that they want to be placed or in a case where the user doesn't require specification to an exact seat one can choose to waitlist for any seats.

```
int seatNum;
int waitType;
cout << "\nWould you like to waitlist for a specific seat or any?" << endl;
cout << "1. Specific\n";
cout << "2. Any Seat\n\n> ";
cin >> waitType;
if (waitType == 1)
{
    while (1)
    {
        cout << "Enter seat Number\n\n> ";
        cin >> seatNum;
        cout << "The cap of bus is : " << busToModify["seatCap"] << endl;
        if (seatNum <= busToModify["seatCap"] && seatNum > 0)
        {
            break;
```

```
            }
            else
            {
                cout << invalidInputMessage;
            }
        }
        vector<int> seatNums;
        seatNums.push_back(seatNum);
        generateWaitlistObj(1, 1, seatNums, 1);
    }
    else
    {
        cout << "You have been added to the waitlist\n";
        vector<int> empty;
        generateWaitlistObj(1, 2, empty, 1);
    }

// generateWaitlistObj;

if (bType == 1 && wType == 1) // single and specific
    {
        json sW;
        string baseID = "W000000";
        int lastID = sW.size();
        string lastID_string = to_string(lastID);
        int start = baseID.size() - lastID_string.size();
        int j = 0;
        for (int i = start; i < baseID.size(); i++)
        {
            baseID[i] = lastID_string[j];
            j++;
        }
        time_t currentTime = time(nullptr);
        string time = ctime(&currentTime); // Convert to string
        time.pop_back();
        json newWaitObj;
        newWaitObj["id"] = baseID;
        newWaitObj["waitType"] = "specific";
        newWaitObj["userID"] = userID;
        newWaitObj["busID"] = busID;
        newWaitObj["size"] = 1;
        newWaitObj["seatNumber"] = seatNums.at(0);
        newWaitObj["time"] = time;
        newWaitObj["status"] = "active";
        waitList["singleWaitList"].push_back(newWaitObj);
```

```
}
else if (bType == 1 && wType == 2) // single and any
{
    json sW;
    string baseID = "W000000";
    int lastID = sW.size();
    string lastID_string = to_string(lastID);
    int start = baseID.size() - lastID_string.size();
    int j = 0;
    for (int i = start; i < baseID.size(); i++)
    {
        baseID[i] = lastID_string[j];
        j++;
    }
    time_t currentTime = time(nullptr);
    string time = ctime(&currentTime); // Convert to string
    time.pop_back();
    json newWaitObj;
    newWaitObj["id"] = baseID;
    newWaitObj["waitType"] = "any";
    newWaitObj["userID"] = userID;
    newWaitObj["busID"] = busID;
    newWaitObj["size"] = 1;
    newWaitObj["seatNumber"] = nullptr;
    newWaitObj["time"] = time;
    newWaitObj["status"] = "active";
    waitList["singleWaitList"].push_back(newWaitObj);
}
```

This process is fairly straightforward by just adding a waitlist object. The processing of these objects will be covered later on. This step of the program yields a **Time Complexity: O(1)**. (disregarding minor ID generation).

2. Processing a Waitlist

This step occurs when a refund happens to ensure that waitlisted users are prioritized over regular reserving users. Moreover, it adds an automation process to the system in which if the seat refunded happens to be a seat that is waitlisted for or after the refund a seat is left open thus, the user who are waitlisting for any seat on that bus will be prioritized first. By utilizing a queue-like structure we ensure that the first user to waitlist for that bus will be prioritized first. Ex: if a user waitlists for seat 7 first and another user waitlists for any single seat on that bus prior to the other user, then the user with specific the seat request will get prioritized first and vice versa.

```
loadData();
```

```
   json singleWait = waitList["singleWaitList"];
   int sWIdx = 0;
   for (auto &sW : singleWait)
   {
         if (sW["status"] == "active" && sW["busID"] == busID && sW["waitType"] ==
"any")
       {
           string userID = sW["userID"];
           int busIdx = 0;
           for (auto &bus : buses)
           {
               if (bus["id"] == busID)
               {
                   bus["seats"][seatRefunded - 1]["status"] = "reserved";
                   int seatLeft = bus["seatLeft"];
                   seatLeft--;
                   bus["seatLeft"] = seatLeft;

                   vector<int> seatNum;
                   seatNum.push_back(seatRefunded);
                   generateReservationFromWaitlist(busID, userID, seatNum);
                   sW["status"] = "inactive";
                   waitList["singleWaitList"][sWIdx] = sW;
                   storeDataWaitList();
                   return true;
               }
           }
       }
        else if (sW["status"] == "active" && sW["busID"] == busID && sW["waitType"]
== "specific")
       {
           int specificSeatNum = sW["seatNumber"];
           for (auto &bus : buses)
           {
                       if (bus["id"] == busID && bus["seats"][specificSeatNum -
1]["status"] == "available")
               {
                   int seatLeft = bus["seatLeft"];
                   seatLeft--;
                   bus["seatLeft"] = seatLeft;
                   bus["seats"][specificSeatNum - 1]["status"] = "reserved";
                   vector<int> seatNum;
                   seatNum.push_back(specificSeatNum);
                   generateReservationFromWaitlist(busID, userID, seatNum);
                   sW["status"] = "inactive";
```

```
                    waitList["singleWaitList"][sWIdx] = sW;
                    storeDataWaitList();
                    return true;
                }
            }
        }
        sWIdx++;
    }
    return false;
```

Due to the usage of 2 loops but with different numbers of elements per loop we conclude that this method has a **Time Complexity: O(n X m)**.

3.5.2. 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 admin to add another administrator to the system by collecting necessary information like name,firstname,lastname,age,email,password,overall its function is similar to sign up for the normal user but we have to set the **bool isAdmin** to true.

```
void User::addAdmin()
{
    cout << "\t\t\t\tNEW ADMIN\n\n";
    string fName = inputFirstName();
    string lName = inputLastName();
    int age = inputAge();
    string email = inputEmail();
    string pass = inputPassword();
    string passCf = confirmPassword(pass);
    passCf = hashPassword(passCf);

    ifstream readFile(dataFile);
    if (!readFile.is_open()){
        cerr <<"\n Error cannot open:"<<dataFile;
    }
    json allData;
    readFile >> allData;
    readFile.close();

    json userData = allData["users"];
    int lastID = userData.size();
    string fullID = "U000000";
    string lastID_string = to_string(lastID);
    int start = fullID.size() - lastID_string.size();
```

```
    for (int i = 0; i < lastID_string.size(); i++)
    {
        fullID[start + i] = lastID_string[i];
    }

    json newUser;
    newUser["id"] = fullID;
    newUser["name"]["firstname"] = fName;
    newUser["name"]["lasttname"] = lName;
    newUser["age"] = age;
    newUser["email"] = email;
    newUser["password"] = passCf;
    newUser["isAdmin"] = true;
    newUser["resID"] = json::array();
    allData["users"].push_back(newUser);

    ofstream writeFile(dataFile);
    if (!writeFile.is_open())
    {
        cerr << "Error: cannot open file" << dataFile;
    }
    writeFile << allData.dump(4);
    writeFile.close();

    cout << "Admin user added successfully!" << endl;
}
```

Time complexity of this fucntion are:

Hashing password **O(n)** where **n** is the length of the password

Json file **O(m)** where **m** is the size of the json file.

Overall, this function combines various techniques and libraries to achieve its functionality. One of the key components is **JSON Handling** using the nlohmann/json library. In this project, JSON is used as our database to store data such as buses, users, reservations, and more. JSON is a lightweight and easy-to-read format, making it widely used for data storage and exchange.In this function json is used for reading data to load all the existing users from the database and add the new admin into the users list.and 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 get all the user data like name , age , ID ,email ,encrypted password, reservation.This function is divided into 2 functions as well. The first one is to get data of one user at a time only by inputting the Specific user ID. On the other hand the second option is to get all the existing users from the database including all their information.

```
void User::getAllUsers()
{
   int option;
   string userID;
   bool isFound = false;
   loadData();

   // Option selection loop
   while (true)
```

```
    {
        cout << "Please input option (1/2):" << endl;
        cout << "1/ View one user by input their ID." << endl;
        cout << "2/ View all the existing users." << endl;
        cout << "Input here: ";
        cin >> option;

        // Check if the option is valid
        if (option < 1 || option > 2)
        {
            cout << "Invalid option, please try again." << endl;
            continue;
        }
        break; // Exit the loop if option is valid
    }

    // Option 1: View a user by ID
    if (option == 1)
    {
        while (true)
        {
            cout << "Please input user ID: ";
            cin >> userID;

            // Validate userID length and format
            if (userID.length() != 7)
            {
                cerr << "Invalid length: " << userID << endl;
                continue;
            }
            if (userID[0] != 'U')
            {
                cerr << "Invalid format: Does not start with 'U'" << endl;
                continue;
            }

            // for search for the user
            isFound = false;
            for (auto it = users.begin(); it != users.end(); ++it)
            {
                if ((*it)["id"] == userID)
                {
                    isFound = true;
                    // Print user information if found
                    cout << "\n\nUser found:\n"
                         << endl;
                    cout << endl
                         << endl
                         << "----------------------Users-----------------------" <<
endl
                         << endl;
                    cout << "ID: " << (*it)["id"] << endl;
                    cout << "Name: " << (*it)["name"]["firstName"] << " " <<
(*it)["name"]["lastName"] << endl;
                    cout << "Age: " << (*it)["age"] << endl;
                    cout << "Email: " << (*it)["email"] << endl;
                    cout << "Is Admin: " << (*it)["isAdmin"] << endl;
                    cout << "Password: " << (*it)["password"] << endl;
                    cout << "Reservation: " << (*it)["resID"] << endl;
```

```
                    cout << endl
                         << "_____________________________________________________" <<
endl
                         << endl;
                    break; // Break loop after finding the user
                }
            }

            if (!isFound)
            {
                cout << "User does not exist." << endl
                     << "Please try again" << endl;
                continue;
            }
            break;
        }
    }
    // Option 2: View all users
    else
    {
        cout << endl
             << endl
             << "----------------------All Users-------------------------" << endl
             << endl;
        for (const auto &user : users)
        {
            cout << "ID: " << user["id"] << endl;
            cout << "Name: " << user["name"]["firstName"] << " " << =
user["name"]["lastName"] << endl;
            cout << "Age: " << user["age"] << endl;
            cout << "Email: " << user["email"] << endl;
            cout << "Is Admin: " << user["isAdmin"] << endl;
            cout << "Password: " << user["password"] << endl;
            cout << "Reservation: " << user["resID"] << endl;
            cout << endl
                 << "_____________________________________________________" << endl
                 << endl;
        }
    }
}
```

The function starts by asking the user to choose between two options:

1. Viewing information for one specific user.
2. Viewing information for all users.

If the user selects the first option, the program will ask them to provide details about the user ID they want to see. It will check if the input is correct and search through the database. When it finds the matching user, it will display their full details.

If the second option is chosen, the program will go through the entire database and show the details of all users in a clear format. The function makes sure the data is ready before starting and handles any incorrect inputs properly.

The time complexity of the function is **O(n),** where **n** is the number of users in the database. For Option 1, in the worst case, the program will search through the entire user list to find a match, resulting in a time complexity of **O(n)**. For Option 2, the program iterates through all the users to display their information, which also results in a time complexity of **O(n)**.

Delete User

This function allows the admin to perform a deletion of a user from the database based on their ID.This functionThe function ensures that a user is only deleted if they meet specific conditions, such as having no active reservations.

```
void User::deleteUser()
{
   string userID;
   cout << "Please input the user ID you want to delete: " << endl;
   cin >> userID;

   loadData();

   bool isFound = false;
   char confirm;

   for (auto it = users.begin(); it != users.end(); ++it)
   {
       if ((*it)["id"] == userID)
       {
           isFound = true;

           cout << "User found:" << endl;
           cout << "ID: " << (*it)["id"] << endl;
           cout << "Name: " << (*it)["name"]["firstName"] << " " <<
(*it)["name"]["lastName"] << endl;
           cout << "Age: " << (*it)["age"] << endl;
           cout << "Email: " << (*it)["email"] << endl;
           cout << "Is Admin: " << (*it)["isAdmin"] << endl;
           cout << "Password: " << (*it)["password"] << endl;
           cout << "Reservation: " << (*it)["resID"] << endl;

           if (!(*it)["resID"].empty())
           {
               cout << "Cannot delete user " << userID << " because they have
active reservations." << endl;
               break;
           }

           cout << "Are you sure you want to delete this user with ID " << userID
<< " (y/n)? ";
           cin >> confirm;

           if (confirm == 'y' || confirm == 'Y')
           {

               users.erase(it);
               cout << "User deleted successfully." << endl;

               data["users"] = users;

               ofstream writeData(dataFilePath);
               if (!writeData.is_open())
               {
                   cerr << "Error: Unable to save changes to file." << endl;
               }
               else
```

```
                {
                    writeData << data.dump(4);
                    writeData.close();
                    cout << "Changes saved to file." << endl;
                }
            }
            else
            {

                cout << "User deletion canceled." << endl;
            }
            break;
        }
    }

    if (!isFound)
    {
        cout << "User does not exist." << endl;
    }
}

void User::viewAllBus()
{
    cout << endl
         << endl
         << "----------------------All Buses-------------------------" << endl
         << endl;
    loadData();
    for (const auto &bus : buses)
    {
        cout << "Bus Type: " << bus["busType"] << endl;
        cout << "ID: " << bus["id"] << endl;
        cout << "Departure Time: " << bus["departureTime"] << endl;
        cout << "Route: " << bus["route"]["from"] << " to " << bus["route"]["to"] <<
endl;
        cout << "Seat Cap: " << bus["seatCap"] << endl;
        cout << "Seat Left: " << bus["seatLeft"] << endl;
        cout << "Seat Price: " << bus["seatPrice"] << endl;
        cout << endl
             << endl
             << "_____________________________________________________" << endl
             << endl;
        ;
    }
}
```

The function begins by asking the administrator for the user ID of the user they wish to delete.after the admin input the ID it will loop to find the user that match with that ID and then display all the information of the user and check if the user have any reservation if the user already have a reservation it will display error message if the user have no reservation it will ask for confirmation to prevent any accident after confirm the user will be deleted from the database and if the use cancel the deletion it will display the message and exit the program.

The time complexity of this function is **O(m+n)** where **m** is the size of the file and **n** is the number of users in the file.

3.6. Final Review and Submission

UI improvements

After completing the logical development phase and integrating all functionalities, the focus shifted to refining the user interface to enhance the overall user experience. **Colors** were added to text using **ANSI codes**, improving readability and emphasizing important information. The layout was refined by adjusting **spacing** within prompts and outputs, ensuring a clean and organized appearance. To create a more immersive experience, a **loading bar** was introduced, simulating the behavior of real-world applications. Additionally, **ASCII art** was incorporated to visually separate different sections of the program, such as the authentication phase, reservation phase, and history view. These UI enhancements contributed to a more polished, intuitive, and professional final product.

Testing

The testing phase focused on ensuring the program's robustness and functionality by identifying and addressing potential bugs and input errors. Each feature was thoroughly tested using a range of valid and invalid inputs to verify error handling and edge-case scenarios. Specific attention was given to the reservation, refund, authentication, and administrative functionalities to confirm they operated as intended without causing any crashes or unexpected behavior. Through this process, all detected issues were resolved, ensuring a seamless user experience.

Preparation for Demonstration

To prepare for the project demonstration, several scenarios were pre-configured to highlight the program's core functionalities. Buses with varying seat capacities were created to showcase the reservation process, including booking available seats and demonstrating waitlisting for fully booked buses. Pre-existing reservations were set up to display the refund process, emphasizing the automatic allocation of refunded seats to the next eligible user on the waitlist.

For the administrative features, buses were added to the system, and the “View Bus” option was used to verify and demonstrate the newly added buses. Similarly, new admin accounts were created, and the “View Users” feature was used to display all users in the system. Demonstrations were also prepared for deletion functionalities, showing how buses, users, or admins could be removed from the system while maintaining system integrity. These preparations ensured that all features were clearly and effectively showcased during the demonstration.

Submission

The submission phase involved finalizing the main branch of the project, ensuring that all code was merged and verified for completeness and stability. A detailed project report was prepared, documenting the development process, features, testing outcomes, and final implementation. The finalized project, along with the report, was submitted to the lecturer for evaluation, marking the completion of the development process.

IV. Challenges

4.1. Technical Challenges

During the process of developing the project, we encounter plenty of challenges ranging from learning new libraries and dependencies to finding the most efficient solutions/algorithms to apply.

Learning new Libraries/Dependencies

To accomplish our goal of creating an efficient and robust bus management system, we had to seek elsewhere for third-party libraries to complete certain tasks. As best practice, we opt to learn and understand in-depthly about the libraries that we include to make this project possible, the libraries consists of the Nlohmann JSON Library, Secure Hash Algorithm (SHA1) and standard C++ libraries and are less frequently used such as <stack> and <limit>.

Nlohmann JSON Library

As previously mentioned, this project heavily depends on this library to manage and manipulate data from our Data.json file. With this library comes its own defined methods to configure data as well as change and manipulate to our desire. Under the hood, this library utilizes the **std::unordered_map** which is a built in data structure in C++, however, none of our team members are familiar with it therefore, we had to learn this data structure and understand it and some of unique techniques/methods to access data within the JSON object.

Defining the system architecture

A system needs its backbone to function effectively therefore, designing a highly robust architecture for the system is the most important step. This step included the thought of designing the classes that will be utilized, database design, external dependencies, file structure, and collaboration techniques. This stage of the development, had to be rushed due to time constraints and allow for the assignment of tasks later on to be developed in time. Thus, we built a system architecture that meets our needs enough; however, we do acknowledge that itself is not the most robust and balanced architecture that we could have come up with.

Classes Responsibilities:

Due to limited hands-on experience, the system architecture defined above was not perfect. It had its limitations and flaws, such as the user class having too many responsibilities such that it handles both the user attributes as well as handling the ever growing operations that the user can do without delegating these tasks to other helper classes much except for delegation to the bus class for seat(s) reservations. Moreover, there was an administrative side to this

program, we made a mistake to not make another or an inherited class of User to do admin operations thus, making User.hpp an excessive bundle of all “Human related” operations.

Modifying the Database:

Throughout the development process, we figured at certain stages there needs to be a value or an attribute to a json object directly in the database. Issues such as needing a **“status”** key for a reservation to define whether the reservation is awaiting or has been refunded. This created conflicts with other teammate's branches and functionalities.

4.2. Collaboration Challenges

Due to the team’s lack of experience with Git version control, we encountered many challenges where we have to learn it and implement it at the same time which leads to lots of errors and setbacks.

Lack of Git experience

The team’s unfamiliarity with Git created difficulties in managing version control efficiently. Mistakes in basic operations, such as creating or switching branches, slowed progress and disrupted workflows.

Learning curve

With no prior exposure to Git, the team had to simultaneously learn and implement its workflows, which often led to confusion. Understanding concepts like branching, merging, and conflict resolution while actively working on the project proved to be challenging.

Task prioritization issues

At times, the team struggled to agree on which tasks to focus on, particularly when faced with tight deadlines or overlapping objectives. This led to delays in completing essential deliverables.

Skill gaps

The team had varying levels of expertise, which caused delays in some areas of the project. Less experienced members needed extra time to learn, while more skilled members had to step in and assist, which temporarily reduced overall productivity.

V. Outcomes

The bus reservation system successfully delivers a comprehensive platform that streamlines the booking process for users while providing robust management tools for administrators. Users benefit from features such as searching for buses, selecting routes, and reserving tickets with ease. The system generates unique IDs for reservations, users, and buses, ensuring seamless tracking and validation of bookings.

Administrators are equipped with functionalities to manage buses, users, and the addition of admin accounts, enabling efficient system management. The platform ensures data integrity by securely handling user, bus, and reservation data in a structured format while tracking changes and maintaining consistency across all operations.

The integration of features such as destination-based searches and seat layout visualization enhances the overall user experience, making the reservation process intuitive and interactive. By providing tailored functionalities for both regular users and administrators, the system delivers a reliable, secure, and efficient solution for bus reservations and management.

VI. Future Improvements

To address the challenges encountered and enhance the system's functionality, several future improvements are proposed:

Refinement of System Architecture

The architecture can be made more modular and robust by redistributing class responsibilities. For instance, separating user operations into multiple helper classes and creating a dedicated admin class or inheriting it from the User class can reduce complexity and improve scalability.

Refinement of Algorithms/Solutions

Due to time constraints, our thought process was just to build a system that works and think about optimizations later on. Thus, some methods are unnecessarily complex, inefficient and costly such as not returning a Bus object after reserving resulting in the need to replace and insertion of data that expends time complexities. Therefore, in the future, we wish to recreate some of the inefficient methods to better display our team’s capabilities instead of a rushed end-product.

Improved Database Design

The JSON-based database could be optimized with additional attributes such as a "status" key for reservations to avoid ad-hoc modifications during development. Transitioning to a more structured database system, like SQL, could further enhance data consistency and query efficiency.

Enhanced Collaboration Practices

To mitigate issues with version control, the team should focus on mastering Git workflows through dedicated training sessions. This includes practicing branching, merging, and conflict resolution to streamline development processes and reduce setbacks.

+ Additional Features

Waitlist System:

Implement a waitlisting system where full buses allow for users to have a priority queue over the chance of a seat getting refunded and that user having a chance of securing a seat/seats in the wanted bus.

Graphical User Interface (GUI) Implementation:

Instead of the limitations of Command Line Interface (CLI), we can further improve the user experience with the superior GUI. Initially, we wanted to create a GUI implementation,

however, due to time constraints and varying skill levels among developers we opted for the CLI instead. In the future, if the codebase continues to still be in C++ we would like to utilize a GUI library such as QT.

Dynamic real-time utilization:

As of current, our program is still relatively manual with no real-time syncing with bus departures and reservations thus, making this program to enable real-world time would be a big plus and can be commercialized.

VII. Conclusion

The bus reservation system successfully addresses the primary objectives of providing a streamlined booking process for users while offering comprehensive management tools for administrators. By allowing users to easily search for buses, select routes, and reserve seats, the system enhances the overall travel experience. Administrators benefit from efficient functionalities such as managing buses, users, and making administrative changes, which ensures smooth operations of the platform.

Despite encountering technical and collaboration challenges throughout the development, such as limited experience with Git and the complexity of managing a JSON-based database, the team managed to deliver a functional system. The learning curve provided valuable insights into both technical tools and team coordination, contributing to the team's growth and experience.

The system's future potential includes improvements in its architecture, database design, and user interface, all of which would further optimize its performance and scalability. Enhanced version control practices, task management tools, and improved testing frameworks will address current development challenges, leading to smoother collaboration and more efficient workflows in future iterations. Overall, this project provides a solid foundation for a bus reservation system that can be built upon to meet the evolving needs of users and administrators alike.

VIII. 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