VISVESVARAYA TECHNOLOGICAL UNIVERSITY

JNANA SANGAMA, BELAGAVI – 590 018

A Mini Project Report on

AIRLINE TICKETING SYSTEM

Submitted in partial fulfillment of the requirements as a part of the File Structures Laboratory for the award of degree of ***Bachelor of Engineering in Information Science and Engineering*** *of Visvesvaraya Technological University, Belagavi*

Submitted by

SHARANYA RP **1RN20IS143**
SIDHARTH S PAI **1RN20IS160**

Project Guide

Dr. Nirmalkumar S Benni

Associate Professor
Dept. of ISE, RNSIT

Faculty Incharge

Dr. Prakasha S

Associate Professor
Dept. of ISE, RNSIT

Department of Information Science and Engineering

RNS Institute of Technology

Channasandra, Dr. Vishnuvardhan Road, RR Nagar Post,
Bengaluru – 560 098

2022 – 2023

RNS Institute of Technology

Channasandra, Dr. Vishnuvardhan Road, RR Nagar Post,
Bengaluru – 560 098

DEPARTMENT OF INFORMATION SCIENCE & ENGINEERING

CERTIFICATE

This is to certify that the Mini project report entitled ***AIRLINE TICKETING SYSTEM*** has been successfully completed by **SIDHARTH S PAI** bearing USN **1RN20IS160** and **SHARANYA RP** bearing USN **1RN20IS143**, presently VI semester students of **RNS Institute of Technology** in partial fulfillment of the requirements as a part of the File Structures Laboratory for the award of the degree **Bachelor of Engineering in Information Science and Engineering** under **Visvesvaraya Technological University, Belagavi** during academic year 2022 – 2023. It is certified that all corrections/suggestions indicated for Internal Assessment have been incorporated in the report deposited in the departmental library. The mini project report has been approved as it satisfies the academic requirements as a part of File Structures Laboratory for the said degree.

Dr. Nirmalkumar S Benni	**Dr. Prakasha S**	**Dr. Suresh L**
Project Guide	**Faculty Incharge**	**Professor and HOD**

External Viva

Name of the Examiners	**Signature with date**
1. _____	_____
2. _____	_____

DECLARATION

We, **SIDHARTH S PAI [USN:1RN20IS160]** and **SHARNAYA RP [USN:1RN20IS143]** student of VI Semester BE, in Information Science and Engineering, RNS Institute of Technology hereby declare that the **AIRLINE TICKETING SYSTEM** has been carried out by us and submitted in partial fulfillment of the requirements for the VI Semester degree of Bachelor of Engineering in Information Science and Engineering of Visvesvaraya Technological University, Belagavi during academic year 2022-2023.

Place : Bengaluru

Date : 06/07/2023

SHARANYA RP
(1RN20IS143)

SIDHARTH S PAI

(1RN20IS160)

ABSTRACT

The main purpose of designing the Airline Ticketing system is to deal with all the data management and methodologies related to booking a ticket. It includes creating, editing, viewing or deleting a ticket. All of the common needs of a consumer while booking tickets manually are covered by this project.

A lot of different methods are used here and the requirements of a user are met phenomenally. From selecting the seat to confirming their reservation, most of the things are covered in this project. This software efficiently collects all the data required in the reservation of a ticket and processes it to generate a ticket after checking all the content comprehensively.

It is easy to use and very user friendly with step by step procedural inputs and outputs. Overall, this project can be used in a very wide scope and has potential to improve the lives of a common user who likes to travel a lot.

Testing plays a vital role in ensuring the system's reliability and effectiveness. Various types of testing, including unit testing, integration testing, system testing, performance testing, security testing, usability testing, and regression testing, are conducted to verify functionality, performance, security, and user-friendliness. The system aims to provide a convenient and efficient platform for users to book and manage their travel arrangements.

.

ACKNOWLEDGMENT

The fulfillment and rapture that go with the fruitful finishing of any assignment would be inadequate without specifying the people who made it conceivable, whose steady direction and support delegated the endeavors with success.

We would like to profoundly thank the **Management** of **RNS Institute of Technology** for providingsuch a healthy and vibrant environment to conduct the project work.

We would like to express our sincere thanks to our beloved Principal **Dr. Ramesh Babu H S** for his support and inspiration towards the attainment of knowledge.

We wish to place on record our words of gratitude to **Dr. Suresh L,** Professor and Head of the Department, Information Science and Engineering, for being the enzyme and master mind behind our project work.

We would also like to thank our mini project guide **Dr. Nirmalkumar S Benni,** Associate Professor,Department of ISE, RNSIT, Bengaluru, for her valuable input, suggestions, time, and guidance.

We would like to thank all other teaching and non-teaching staff of Information Science & Engineering who have directly or indirectly helped me to conduct the project work.

Place: Bengaluru

Date:06/07/2023

Sharanya RP
(1RN20IS143)
Sidharth S Pai
(1RN20IS160)

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

The Airline Ticketing System is a stand-alone application. It provides a user-friendly, interactive Menu Driven Interface (MDI). All data is stored in files for persistence. The project uses one file to hold the records.

1.1 Introduction to File Structure

A file structure is a combination of representations for data in files and of operations for accessing the data. A file structure allows applications to read, write, and modify data. It might also support finding the data that matches some search criteria or reading through the data in some order. An improvement in file structure design may make an application hundreds of times faster. The details of the representation of the data and the implementation of the operations determine the efficiency of the file structure for the application.

1.1.1 About the File

When we talk about a file on disk or tape, we refer to a collection of bytes stored there. A file, when the word is used in this sense, physically exists. A disk drive may contain hundreds, even thousands of these physical files.

From the standpoint of an application program, a file is somewhat like a telephone line connection to a telephone network. The program can receive bytes through this phone line or send bytes down it, but it knows nothing about where these bytes come from or where they go. The program knows only about its end of the line. Even though there may be thousands of physical files on a disk, a single program is usually limited to the use of only about 20 files.

The application program relies on the OS to take care of the details of the telephone switching system. It could be that bytes coming down the line into the program originate from a physical file they come from the keyboard or some other input device. Similarly, bytes the program sends down the line might end up in a file, or they could appear on the terminal screen or some other output device. Although the program doesn't know where the bytes are coming from or where they are going, it does know which line it is using. This line is usually referred to as the logical file, to distinguish it from the physical files on the disk or tape.

1.1.2 Various Kinds of storage of Fields and Records

A field is the smallest, logically meaningful, unit of information in a file.

Field Structures

The four most common methods as shown in Fig. 1.1 of adding structure to files to maintain the identity of fields are:

- Force the fields into a predictable length.
- Begin each field with a length indicator.
- Place a delimiter at the end of each field to separate it from the next field.
- Use a "keyword=value" expression to identify each field and its contents.

Method 1: Fix the Length of Fields

In the above example, each field is a character array that can hold a string value of some maximum size. The size of the array is 1 larger than the longest string it can hold. Simple arithmetic is sufficient to recover data from the original fields.

The disadvantage of this approach is adding all the padding required to bring the fields up to a fixed length, makes the file much larger. We encounter problems when data is too long to fit into the allocated amount of space. We can solve this by fixing all the fields at lengths that are large enough to cover all cases, but this makes the problem of wasted space in files even worse. Hence, this approach isn't used with data with large amount of variability in length of fields, but where every field is fixed in length if there is very little variation in field lengths.

Table 1.1 Fix the Length of Fields

Ames******Mary******123*******Maple*****Stillwater***OK**74075
Mason*****Alan*******90********Eastgate***Ada********OK**74820

Method 2: Begin Each Field with a Length Indicator

We can count to the end of a field by storing the field length just ahead of the field. If the fields are not too long (less than 256 bytes), it is possible to store the length in a single byte at the start of each field. We refer to these fields as length-based.

Table 1.2 Begin Each Field with a Length Indicator

04Ames04Mary09123 Maple10Stillwater02OK0574075
05Mason04Alan1190 Eastgate03Ada02OK0574820

Method 3: Separate the Fields with Delimiters

We can preserve the identity of fields by separating them with delimiters. All we need to do is choose some special character or sequence of characters that will not appear within a field and then insert that delimiter into the file after writing each field. White-space characters (blank, new line, tab) or the vertical bar character, can be used as delimiters.

Table 1.3 Separate the Fields with Delimiters

04Ames04Mary09123 Maple10Stillwater02OK0574075
05Mason04Alan1190 Eastgate03Ada02OK0574820

Method 4: Use a "Keyword=Value" Expression to Identify Fields

It is the first structure in which a field provides information about itself. It is easy to tell which fields are contained in a file. Even if we don't know ahead of time which fields the file is supposed to contain. It is also a good format for dealing with missing fields. If a field is missing, this format makes it obvious, because the keyword is simply not there. It is helpful to use this in combination with delimiters, to show division between each value and the keyword for the following field, but this also wastes a lot of space

A record can be defined as a set of fields that belong together when the file is viewed in terms of a higher level of organization.

Table 1.4 Use a "Keyword=Value" Expression to Identify Fields

04Ames04Mary09123 Maple10Stillwater02OK0574075
05Mason04Alan1190 Eastgate03Ada02OK0574820

Record Structures

The most often used methods for organizing records of a file as shown in Fig 1.2 and Fig 1.3 are:

- Require the records to be predictable number of bytes in length.
- Require the records to be predictable number of fields in length.
- Begin each record with a length indicator consisting of a count of the number of bytes that the record contains.
- Use a second file to keep track of the beginning byte address for each record.
- Place a delimiter at the end of each record to separate it from the next record.

Method 1: Use Fixed-Length Records

A fixed-length record file is one in which each record contains the same number of bytes. In the field and record structure shown, we have a fixed number of fields, each with a predetermined length, that combine to make a fixed-length record.

Fixing the number of bytes in a record does not imply that the size or number of fields in the record must be fixed. Fixed-length records are often used as containers to hold variable numbers of variable-length fields. It is also possible to mix fixed and variable-length fields within a record.

Table 1.5 Fixed length records (Fixed-Length fields)

Ames****Mary****123Maple***Stillwater*****OK***74075
Mason***Alan****90*Eastgate**Ada*********OK***74820

Table 1.6 Fixed length records (variable length fields)

Ames****Mary****123Maple***Stillwater*****OK***74075
Mason***Alan****90*Eastgate**Ada*********OK***74820

Method 2: Make Records a Predictable Number of Fields

Rather than specify that each record in a file contains some fixed number of bytes, we can specify that it will contain a fixed number of fields. In the figure below, we have 6 contiguous fields and we can recognize fields simply by counting the fields modulo 6.

Table 1.7 Make Records a Predictable Number of Fields

Ames****Mary****123Maple***Stillwater*****OK***74075
Mason***Alan****90*Eastgate**Ada*********OK***74820

Method 3: Begin Each Record with a Length Indicator

We can communicate the length of records by beginning each record with a filed containing an integer that indicates how many bytes there are in the rest of the record. This is commonly used to handle variable-length records.

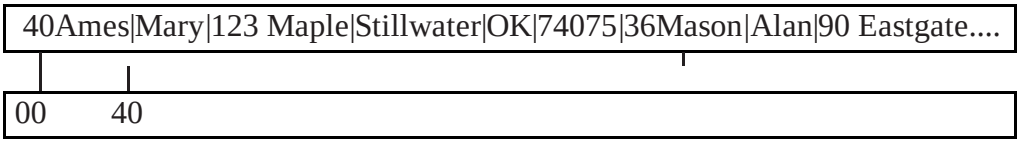
Table 1.8 Begin Each Record with a Length Indicator

40Ames|Mary|123 Maple|Stillwater|OK|74075|36Mason|Alan|90 Eastgate....

Method 4: Use an Index to Keep Track of Addresses

We can use an index to keep a byte offset for each record in the original file. The byte offset allows us to find the beginning of each successive record and compute the length of each record. We look up the position of a record in the index, then seek to the record in the data file.

Table 1.9 Use index to store the start address of each record



Index file

Method 5: Place a Delimiter at the End of Each Record

It is analogous to keeping the fields distinct. As with fields, the delimiter character must not get in the way of processing. A common choice of a record delimiter for files that contain readable text is the end-of-line character (carriage return/ new-line pair or, on UNIX systems, just a new-line character: \n). Here, we use a # character as the record delimiter.

Table 1.10 Place a Delimiter at the End of Each Record

40Ames|Mary|123 Maple|Stillwater|OK|74075|36Mason|Alan|90 Eastgate....

1.1.3 Application of File Structure

Relative to other parts of a computer, disks are slow. 1 can pack thousands of megabytes on a disk that fits into a notebook computer.

The time it takes to get information from even relatively slow electronic random-access memory (RAM) is about 120 nanoseconds. Getting the same information from a typical disk takes 30 milliseconds. So, the disk access is a quarter of a million times longer than a memory access. Hence, disks are very slow compared to memory. On the other hand, disks provide enormous capacity at much less cost than memory. They also keep the information stored on them when they are turned off.

Tension between a disk's relatively slow access time and its enormous, nonvolatile capacity, is the driving force behind file structure design. Good file structure design will give us access to all the capacity without making our applications spend a lot of time waiting for the disk.

Method 5: Place a Delimiter at the End of Each Record

It is analogous to keeping the fields distinct. As with fields, the delimiter character must not get in the way of processing. A common choice of a record delimiter for files that contain readable text is the end-of-line character (carriage return/ new-line pair or, on UNIX systems, just a new-line character: \n). Here, we use a # character as the record delimiter.

Table 1.11 Place a Delimiter at the End of Each Record

40Ames|Mary|123 Maple|Stillwater|OK|74075|36Mason|Alan|90 Eastgate....

1.1.4 Application of File Structure

Relative to other parts of a computer, disks are slow. 1 can pack thousands of megabytes on a disk that fits into a notebook computer.

The time it takes to get information from even relatively slow electronic random-access memory (RAM) is about 120 nanoseconds. Getting the same information from a typical disk takes 30 milliseconds. So, the disk access is a quarter of a million times longer than a memory access. Hence, disks are very slow compared to memory. On the other hand, disks provide enormous capacity at much less cost than memory. They also keep the information stored on them when they are turned off.

Tension between a disk's relatively slow access time and its enormous, nonvolatile capacity, is the driving force behind file structure design. Good file structure design will give us access to all the capacity without making our applications spend a lot of time waiting for the disk.

Chapter 2

SYSTEM ANALYSIS

Systems analysis the process of observing systems for troubleshooting or development purposes. It is applied to information technology, where computer-based systems require defined analysis according to their makeup and design.

2.1 Analysis of Application

The airline ticketing system is a software application designed for booking and managing airline tickets. It features a client-server architecture and utilizes a database for data storage. The system offers functionalities such as user registration, flight search, ticket reservation, payment processing, booking management, and ticket cancellation.

2.2 Structure used to Store the Fields and Records

The structure used to store fields and records is not provided in the given code. However, based on the code, it appears that the data is stored in separate text files (e.g., Login_Details.txt, Booking_Details.txt, Seat_Details.txt, Payment_Details.txt) rather than using a specific structure in the code.

If selected for modify option, the System will ask to enter the Vehicle no. which is to be modified and thereby after entering the Vehicle no., the corresponding details of the vehicle are also displayed and then option to modify the fields is asked to be entered. The user can then give all the related information and press enter. And the updated or modified values will be reflected in the file.

Operations Performed on a File

1. Reading Flight Information:

The system can read flight information from a file, such as flight schedules, available seats, boarding cities, destination cities, and ticket prices.

This information can be used to populate the search results when users are looking for available flights.

2. Updating Flight Information:

The system may need to update flight information in the file, such as when a flight is fully booked or cancelled.

This operation ensures that the system maintains up-to-date and accurate flight details.

3. Storing User Information:

When a user registers or creates an account, their information (e.g., username, password) can be stored in a file for future reference.

This allows the system to authenticate users during login and maintain a record of registered users.

4. Storing Booking Details:

After a user successfully selects a seat and completes the booking process, the booking details (e.g., user information, flight details, payment information) can be stored in a file.

This operation allows the system to keep a record of all bookings made by users.

5. Retrieving Booking History:

The system can read the file containing booking details to retrieve a user's booking history.

This allows users to view their past bookings and facilitates administrative tasks, such as generating reports or managing bookings.

6. Managing Seat Availability:

The system can track and update seat availability in a file as users make bookings.

This ensures that users can only select available seats and prevents double bookings.

7. Logging System Activities:

The system can log various activities, such as user logins, flight searches, bookings, and payment transactions, in a file.

Logging provides an audit trail for troubleshooting, monitoring system usage, and generating reports.

Indexing Used :

Hashing

Hashing is the transformation of a string of characters into a usually shorter fixed-length value or key that represents the original string. Hashing is used to index and retrieve items in a database because it is faster to find the item using the shorter hashed key than to find it using the original value. It is also used in many encryption algorithms. A hash table is a collection of items which are stored in such a way as to make it easy to find them later. Each position of the hash table, often called a slot, can hold an item and is named by an integer value starting at 0. For example, we will have a slot named 0, a slot named 1, a slot named 2, and so on. Initially, the hash table contains no items, so every slot is empty.

Bucket − A hash file stores data in bucket format. Bucket is considered a unit of storage. A

bucket typically stores one complete disk block, which in turn can store one or more records.

Hash Function − A hash function, h, is a mapping function that maps all the set of search keys K to the address where actual records are placed. It is a function from search keys to bucket addresses.

Hashing enables faster data retrieval. There are 2 types of hashing, these are static hashing and dynamic hashing. In this project static hashing has been implemented. In static hashing, when a search key value is provided, the hash function always computes the same address. For example, if mod-11 hash function is used, then it shall generate only 11 values. The output address shall always be same for that function. The number of buckets for the program remains unchanged at all the times.

Advantages of hashing

Records need not be sorted after any of the transaction. Hence the effort of sorting is reduced in this method. Since block address is known by hash function, accessing any record is very faster. Similarly updating or deleting a record is also very quick. This method can handle multiple transactions as each record is independent of other. i.e.; since there is no dependency on storage location for each record, multiple records can be accessed at the same time. Even if collisions occur it is still quick to go reach the correct location using hashing than with linear searching. It is suitable for online systems like net-banking, ticket booking etc. Some of the applications are:

- In cryptography hash functions can be used to hide information. One such application is in digital signatures where a so-called secure hash is used to describe a large document with a small number of bits.
- A one-way hash function is used to hide a string, for example for password protection. Instead of storing passwords in plain text, only the hash of the password is stored. To verify whether a password entered is correct, the hash of the password is compared to the stored value. These signatures can be used to authenticate the source of the document, ensure the integrity of the document as any change to the document invalidates the signature, and prevent repudiation where the sender denies signing the document.
- String commitment protocols use hash functions to hide to what string a sender has committed so that the receiver gets no information. Later, the sender sends a key that allows the receiver to reveal the string. In this way, the sender cannot change the string once it is committed, and the receiver can verify that the revealed string is the committed string. Such protocols might be used to flip a coin across the internet: The sender flips a coin and commits the result. In the meantime, the receiver calls heads or

tails, and the sender then sends the key, so the receiver can reveal the coin flip.

- Hashing can be used to approximately match documents, or even parts of documents. Fuzzy matching hashes overlapping parts of a document and if enough of the hashes match, then it concludes that two documents are approximately the same. Big search engines look for similar documents so that on search result pages they don't show the many slight variations of the same document. It is also used in spam detection, as spammers make slight changes to email to bypass spam filters or to push up a document's content rank on a search results page. Geneticists use it to compare sequences of genes fragments with a known sequence of a related organism to assemble the fragments into a reasonably accurate genome.
- Hashing is used to implement hash tables. In hash tables one is given a set of keys K α and needs to map them to a range of integers so they can have stored at those locations in an array. The goal is to spread the keys out across the integers as well as possible to minimize the number of keys that collide in the array. You should not confuse the terms hash function and hash table. They are two separate ideas, and the latter uses the former.

The hashing algorithm is called the hash function. The hash function is used to index the original value or key and then used later each time the data associated with the value or key is to be retrieved. Thus, hashing is always a one-way operation. If a hash function produces the same hash for 2 different values, it is known as collision. A hash function that offers an extremely low risk of collision may be considered acceptable.

The hash function that has been used in this project is Summation and reminder method.

If the table size is 10 and all of the keys sum to five. In this case, the choice of hash function and table size needs to be carefully considered. The best table sizes are prime numbers. One problem though is that keys are not always numeric as its common for them to be strings. A possible solution: add up the ASCII values of the characters in the string to get a numeric value and then perform the Summation and reminder method.

Summation and reminder method: The sum of all the digits of the key in its numeric form is first calculated. Then the sum is then shrunk to the table size by performing a modulus operation on it with the maximum size of the hash table. This technique of summing up and bringing down the key to the smaller hash table size format is called Summation and reminder.

Linear Probing: The technique used to resolve collision is linear probing. Linear probing is a scheme in computer programming for resolving collisions in hash tables, data structures for maintaining a collection of key–value pairs and looking up the value associated with a given key. Along with quadratic probing and double hashing, linear probing is a form of open

addressing. When the hash function causes a collision by mapping a new key to a cell of the hash table that is already occupied by another key, linear probing searches the table for the closest following free location and inserts the new key there. Lookups are performed in the same way, by searching the table sequentially starting at the position given by the hash function, until finding a cell with a matching key or an empty cell. In linear probing, to insert a key k, it first checks h (k) and then checks each following location in succession, wrapping around as necessary, until it finds an empty location. That is, the probe is h (k, i) = (h (k) + i) mod m. Each position determines a single probe sequence, so there are only m possible probe sequences.

The problem with linear probing is that keys tend to cluster. It suffers from primary clustering: Any key that hashes to any position in a cluster (not just collisions), must probe beyond the cluster and adds to the cluster size. Worse yet, primary clustering not only makes the probe sequence longer, it also makes it more likely that it will be lengthen further.

Chapter 3

SYSTEM DESIGN

3.1 User Interface

The User Interface or UI refers to the interface between the application and the user. Here, the UI is menu-driven, that is, a list of options (menu) is displayed to the user and the user is prompted to enter an integer corresponding to the choice that represents the desired operation to be performed.

3.1.1 Insertion of a Record

When the admin wants to add a new ticket to the system, they would select the option "1" from the admin menu. Following this, a new screen is displayed to prompt the admin to enter the details of the ticket. The required information for the ticket includes:

Ticket ID: A unique identifier for the ticket.

Passenger Name: The name of the passenger associated with the ticket.

Boarding City: The city or airport where the passenger will board the flight.

Destination City: The city or airport where the passenger will reach their destination.

Seat: The seat number or seat identifier assigned to the passenger.

Ticket Class: The class of the ticket, such as economy, business, or first class.

Once the admin enters all the necessary values, a "record inserted" message is displayed to confirm the successful addition of the ticket. The admin is then prompted to press any key to return to the menu screen.

3.1.2 Display of a Record

To display the records stored in the system, the user (admin or otherwise) selects option "3" from the menu. If there are no records available, a "no records found" message is displayed to inform the user.

When displaying the records, suitable headings are provided to organize the information. For each ticket displayed, all the field details are shown. This includes the following information:

Ticket ID: The unique identifier for the ticket.

Passenger Name: The name of the passenger associated with the ticket.

Boarding City: The city or airport where the passenger will board the flight.

Destination City: The city or airport where the passenger will reach their destination.

Seat: The seat number or seat identifier assigned to the passenger.

Ticket Class: The class of the ticket, such as economy, business, or first class.

By displaying the field details for each ticket, the user can view the relevant information stored in the system..

3.1.3 Deletion of a Record

When the user wants to delete a ticket from the airline ticketing system, they would select option "4" from the menu. Following this, the user is prompted to enter the ticket ID of the ticket they want to delete. If there are no records in the system that match the entered ticket ID, a "no records found" message is displayed to inform the user that no matching ticket was found.

If a record with the specified ticket ID is found, the details of that particular ticket, such as the ticket ID, passenger name, boarding city, destination city, seat, and ticket class, are displayed to confirm that the correct ticket is being deleted.

After displaying the ticket details, a "record deleted" message is shown to indicate that the ticket has been successfully removed from the system.

In both cases (no records found or record deleted), the user is prompted to press any key to return to the menu screen, allowing them to continue using other functionalities of the airline ticketing system.

3.1.4 Modification of a Record

When the user wants to modify a ticket in the airline ticketing system, they would select option "2" from the menu. Following this, the user is prompted to enter the ticket ID of the ticket they want to modify.

If the entered ticket ID does not have a matching record, a "record not found" message is displayed, indicating that no ticket with that ID exists in the system. The user is then prompted to press any key to return to the menu screen.

If a record with the specified ticket ID is found, the existing ticket data is displayed with relevant headings. The user is then prompted to enter the updated details for the ticket, such as the ticket ID, passenger name, boarding city, destination city, seat, and ticket class. After accepting all the updated values, a "record updated" message is displayed to confirm that the ticket has been successfully modified.

Once again, the user is prompted to press any key to return to the menu screen, allowing them to continue using other functionalities of the airline ticketing system.

Chapter 4

IMPLEMENTATION

Implementation is the process of defining how the system should be built, ensuring its working and meets quality standards. It is a systematic and structured approach for effectively integrating a software-based service or component into the requirements of end users.

4.1 About C++

4.1.1 Structures:

This Ticket structure allows storing and managing ticket information in a structured manner, making it easier to work with airline reservations and related data.

The struct Ticket provides a convenient way to store and access ticket information. Each field is of type string, allowing flexibility in storing textual data.

```
struct Ticket {
    string id;
    string name;
    string boardingCity;
    string destinationCity;
    string seat;
    string ticketClass;
};
```

1. id: A string field representing the ticket ID. It is used to uniquely identify the ticket.
2. name: A string field that stores the name of the passenger associated with the ticket.
3. boardingCity: A string field indicating the boarding city or airport for the flight.
4. destinationCity: A string field representing the destination city or airport for the flight.
5. seat: A string field holding the seat number or seat identifier assigned to the passenger.
6. ticketClass: A string field specifying the ticket class, such as "economy," "business," or "first class."

4.1.2 File Handling

Files form the core of this project and are used to provide persistent storage for user entered information on disk. The open () and close () methods, as the names suggest, are defined in the C++ Stream header file fstream.h, to provide mechanisms to open and close files. The physical file handles used to refer to the logical filenames, are also used to ensure files exist before use and that existing files aren't overwritten unintentionally. The 2 types of files used are data files and index files. open () and close () are invoked on the file handle of the file to be opened/closed. open () takes 2 parameters- the filename and the mode of access. close () takes no parameters.

4.1.3 Character Arrays and Character functions

Character arrays are used to store the record fields to be written to data files. They are also used to store the ASCII representations of the integer that are written to the data file. Character functions are defined in the header file ctype.h and also in string.h. Some of the functions used include:

- toupper() – used to convert lowercase characters to uppercase characters.
- itoa() – to store ASCII representations of integers in character arrays
- atoi() – to convert an ASCII value into an integer.
- strlen() – returns the length of the character array
- strcpy() – copies one string to another
- strcat() – concatenates one string to the end of the next

4.2 Implemented code

4.2.1 Insertion Module

The read() function (Figure. 4.1) takes a new record as input to the data file if the key number entered is not a duplicate. Values are inserted after computing the hash byte offset address and linear probing is done to find the actual position of placement of record.

```
 void createTicket() {
Ticket ticket;
cout << "Enter passenger name: ";
cin.ignore();
getline(cin, ticket.name);
cout << "Enter boarding city: ";
getline(cin, ticket.boardingCity);
cout << "Enter destination city: ";
getline(cin, ticket.destinationCity);
cout << "\n1) Economy\n2) Business\n\nPick class (1-2): ";
      string x;
      int y;
```

```
for (int i = 0; i > -1; i++)
{
        cin >> x;
        if (x == "1") {
                ticket.ticketClass += "Economy";
                y = 1;
                break;
        }
        else if (x == "2") {
                ticket.ticketClass += "Business";
                y = 2;
                break;
        }
        else
        {
                cout << "\n\nInvalid input! Please enter again (1-2): ";
        }
}
```

4.2.2 Display Module

The `printBookedTickets()` function displays a list of booked tickets. It iterates through the `ticketIndex` data structure and prints the ticket details, including the ticket ID, passenger name, boarding city, destination city, and seat. After displaying each ticket, it waits for 5 seconds and then clears the screen. This function provides an overview of the booked tickets in a readable format.

```
void printBookedTickets() {
   cout << "List of booked tickets:" << endl;
   for (const auto& pair : ticketIndex) {
      const Ticket& ticket = pair.second;
      cout << "Ticket ID: " << ticket.id << endl;
      cout << "Passenger Name: " << ticket.name << endl;
      cout << "Boarding City: " << ticket.boardingCity << endl;
      cout << "Destination City: " << ticket.destinationCity << endl;
      cout << "Seat: " << ticket.seat << endl;
      cout << "--------------------" << endl;
      Sleep(5000);
      system("cls");
   }}
```

4.2.3 Deletion Module

The deleteTicket() function enables users to delete a ticket by providing its ID. It first prompts the user to enter a ticket ID. If the ID is found in the ticketIndex, the corresponding ticket is removed from the data structure. The ticket details are then updated in a file specified by Booking_Details file. If the update is successful, a confirmation message is displayed, and the screen is cleared. However, if there is an error saving the ticket details, an error message is shown. In case the entered ticket ID is not found in the ticketIndex, a message indicating the absence of the ticket is displayed

```
void deleteTicket() {
    string ticketID;
  cout << "Enter ticket ID: ";
  cin >> ticketID;

  // Check if the ticket ID exists
  if (ticketIndex.find(ticketID) != ticketIndex.end()) {
     // Remove the ticket from the hash table
     ticketIndex.erase(ticketID);

     // Update the ticket details in the file
     ofstream bookingFile(BOOKING_DETAILS_FILE);
     if (bookingFile) {
        for (const auto& pair : ticketIndex) {
           const Ticket& ticket = pair.second;
              bookingFile << ticket.id << " " << ticket.name << " " << ticket.boardingCity
  << " "
                    << ticket.destinationCity << " " << ticket.seat << endl;
        }
        bookingFile.close();
        cout << "Ticket deleted successfully!" << endl;
        Sleep(3500);
        system("cls");
     } else {
        cout << "Error saving ticket details." << endl;
     }
  } else {
     cout << "Ticket not found." << endl;
  }
}
```

4.2.4 Modify Module

The code is a function editTicket() that allows users to modify the details of a ticket. It prompts for a ticket ID, checks if the ID exists, and if so, asks for new passenger name, boarding city, and destination city. It then reads seat details from a file, displays available seats, and prompts the user to choose a seat. The chosen seat is marked as occupied, and the seat details are updated in the file. Finally, the ticket details are saved in another file, and success/error messages are displayed accordingly

```
void editTicket() {
string ticketID;
cout << "Enter ticket ID: ";
cin >> ticketID;
   if (ticketIndex.find(ticketID) != ticketIndex.end()) {
   Ticket& ticket = ticketIndex[ticketID];
   cout << "Enter new passenger name: ";
   cin.ignore();
   getline(cin, ticket.name);
   cout << "Enter new boarding city: ";
   getline(cin, ticket.boardingCity);
   cout << "Enter new destination city: ";
   getline(cin, ticket.destinationCity);fstream Seat;
      int count = 0, delimit = 0;
      string line;
      SeatRecord Seats[10];
      Seat.open("Seat_Details.txt");
      while (getline(Seat, line))
      {
            count++;
      }
      Seat.close();
      Seat.open("Seat_Details.txt");
      for (int j = 0; j < count; j++)
      {
            string line1;
            getline(Seat, line1);
            for (int i = 0; i > -1; i++)
            {
                  char temp;
                  temp = line1[i];
                  if (temp == '-')
                  {
                        delimit = i;
                        break;
                  }
```

```
                Seats[j].RowA += temp;
            }
            for (int i = delimit + 1; i > -1; i++)
            {
                char temp;
                temp = line1[i];
                if (temp == '-')
                {
                    delimit = i;
                    break;
                }
                Seats[j].RowB += temp;
            }
            for (int i = delimit + 1; i > -1; i++)
            {
                char temp;
                temp = line1[i];
                if (temp == '-')
                {
                    delimit = i;
                    break;
                }
                Seats[j].RowC += temp;

}
        }
        Seat.close();
        string RowName, SeatNum, FinalSeatNum;
        bool Flag = false;
        cout << "\n\nA" << "\t" << " B" << "\t" << " C" << "\t\n";
        for (int i = 0; i < 10; i++)
        {
            cout << "\n\n" << Seats[i].RowA << "\t" << Seats[i].RowB << "\t" <<
Seats[i].RowC << "\t\n";
        }
        cout << "\n\n=====================================\n";
        for (int x = 0; x > -1; x++)
        {
            cout << "\n\nEnter the row name that you chose: ";
            cin >> RowName;
            if (RowName != "A" && RowName != "B" && RowName != "C")
            {                       cout << "\n\nInvalid row, please try again. ";
            }
            else
                break;
```

```
}
for (int x = 0; x > -1; x++)
{
	check7:
	cout << "\n\nEnter the seat number that you chose: ";
	cin >> SeatNum;
	if (SeatNum == "XX")
	{
		cout << "\n\nInvalid seat, please try again! \n";
		goto check7;
	}
	for (int i = 0; i < 10; i++)
	{
		if (RowName == "A")
		{
			if (SeatNum == Seats[i].RowA)
			{
				Seats[i].RowA = "XX";
				Flag = true;
				break;
			}
		}
		else if (RowName == "B")
		{
			if (SeatNum == Seats[i].RowB)
			{
				Seats[i].RowB = "XX";
				Flag = true;
				break;
			}
		}
		else if (RowName == "C")
		{
			if (SeatNum == Seats[i].RowC)
			{
				Seats[i].RowC = "XX";
				Flag = true;
				break;
			}
		}
	}
	if (Flag == false)
	{
		cout << "\nSeat number not found, please try again. \n";
```

```
                }
                else
                        break;
        }
                FinalSeatNum = RowName + SeatNum;
                ofstream Del;
                Del.open("Seat_Details.txt", ios::trunc);
                Del.close();
                Del.open("Seat_Details.txt", ios::trunc);
                for (int i = 0; i < 10; i++)
                {
                        Del << Seats[i].RowA << "-" << Seats[i].RowB << "-" << Seats[i].RowC
<< "-\n";
                }
                Del.close();
      // Update the ticket details in the file
      ofstream bookingFile(BOOKING_DETAILS_FILE);
      if (bookingFile) {
         for (const auto& pair : ticketIndex) {
            const Ticket& t = pair.second;
            bookingFile << t.id << " " << t.name << " " << t.boardingCity << " "
                    << t.destinationCity << " " << t.seat << endl;
         }
         bookingFile.close();
         cout << "Ticket updated successfully!" << endl;
         Sleep(3500);
         system("cls");
      } else {
         cout << "Error saving ticket details." << endl;
      }
   } else {
      cout << "Ticket not found." << endl;
   }
}
```

4.2.5 Hashing Module

There are many methods to compute the hash value of the key. These methods are called hash functions. The unordered_map data structure is used to store tickets indexed by ticket ID. The unordered_map in C++ is implemented using hashing to provide efficient lookup and retrieval of elements based on their keys. In this code, the ticketIndex variable is an unordered_map that stores Ticket structures with ticket IDs as keys. The use of unordered_map allows for quick retrieval of ticket information based on the ticket ID.

Code: // Hash table to store tickets indexed by ticket ID

```
unordered_map<string, Ticket> ticketIndex;
struct SeatRecord {
        string RowA;
        string RowB;
        string RowC;
};
```

Code: To add the ticket details into hash table and store in Seat_Details.txt file.

```
Del.open("Seat_Details.txt", ios::trunc);
                for (int i = 0; i < 10; i++)
                {
                        Del << Seats[i].RowA << "-" << Seats[i].RowB << "-" << Seats[i].RowC
<< "-\n";
                }
                Del.close();
      ticket.seat=FinalSeatNum;
   // Add the ticket to the hash table
   ticketIndex[ticket.id] = ticket;
```

4.3 Testing

Software Testing is the process used to help identify the correctness, completeness, security and quality of the developed computer software. Testing is the process of technical investigation and includes the process of executing a program or application with the intent of finding errors.

4.3.1 Unit Testing

The unit testing is the process of testing the part of the program to verify whether the program is working correct or not. In this part the main intention is to check the each and every inputs which we are inserting to our file. Here the validation concepts are used to check whether the program is taking the inputs in the correct format or not.

Functions (or features) are tested by feeding them input and examining the output. Functional testing ensures that the requirements are properly satisfied by the project. This type of testing is not concerned with how processing occurs, but rather, with the results of processing. During functional testing, Black Box Testing technique is used in which the internal logic of the system being tested is not known to the tester.

Table 4.1 Unit Testing Test Cases

Cases	Description	Input Data	Expected O/P	Actual O/P	Status
1	Test user registration	Register a new user with valid information.	User registration is successful.	User registration is successful.	PASS
2	Test user login	Log in with valid user credentials.	User login is successful.	User login is successful.	PASS
3	Test ticket creation	Create a ticket with valid information.	Ticket creation is successful.	Ticket creation is successful	PASS
4	Test ticket payment	Make a payment for a ticket.	Ticket payment is successful.	Ticket payment is successful	PASS
5	Test ticket cancellation	Cancel a ticket	Ticket cancellation is successful	Ticket cancellation is successful	PASS
6	Test flight search	Search for available flights with specific criteria.	List of available flights matching the search criteria.	List of available flights matching the search criteria.	PASS

7	Test Ticket Status Update	Update the status of a ticket	Ticket status is updated successfully.	Ticket status is updated successfully.	PASS
8	Test ticket details update	Update the details of a ticket.	Ticket details are updated successfu lly.	Ticket details are updated succe ssfully.	PASS

4.3.2 Integration Testing

Integration testing is also taken as integration and testing this is the major testing process where the units are combined and tested. Its main objective is to verify whether the major parts of the program is working fine or not. This testing can be done by choosing the options in the program and by giving suitable inputs it is tested.

Table 4.2 Integration testing test cases

Cases	Description	Input	Expected O/P	Actual O/P	Status
1	Registering a new user and logging in	Register a new user and log in with the same credentials	Successful registration and login.	Successful registration and login.	PASS
2	Creating a ticket and making a payment	Log in, create a ticket, and make a payment.	Successful ticket creation and payment.	Successful ticket creation and payment.	PASS
3	Creating a ticket without logging in	Attempt to create a ticket without logging in.	User not logged in, authentication required..	User not logged in, authentication required.	PASS
4	Making a payment without creating a ticket	Log in and try to make a payment without creating a ticket	No ticket found, please create a ticket first.	No ticket found, please create a ticket first.	PASS
5	Checking the ticket status after successful payment	Log in, create a ticket, make a payment, and check the ticket status	Ticket status is "Paid" or "Confirmed".	Ticket status is "Paid" or "Confirmed".	PASS
6	Checking the ticket status before making a payment	Log in, create a ticket, and check the ticket status.	Ticket status is "Pending" or "Unpaid"..	Ticket status is "Pending" or "Unpaid".	PASS

7	Searching for available flights	Search for available flights with specific criteria.	List of available flights matching the search criteria.	List of available flights matching the search criteria.	PASS
8	Updating ticket details	Log in, create a ticket, and update its details.	Ticket details updated successfully.	Ticket details updated successfully.	PASS
9	Cancelling a ticket	Log in, create a ticket, and cancel it	Ticket cancelled successfully	Ticket cancelled successfully	PASS

4.3.3 System Testing

System testing is defined as testing of a complete and fully integrated software product. This testing falls in black-box testing wherein knowledge of the inner design of the code is not a pre-requisite and is done by the testing team. System testing is done after integration testing is complete. System testing should test functional and non-functional requirements of the software.

Table 4.3 System Testing test cases

Cases	Description	Input Data	Expected O/P	Actual O/P	Status
1	Test user registration and login:	Register new user, login with registered credentials.	Successful registration and login.	Successful registration and login.	PASS
2	Test flight search and selection	Search and select a flight	Correct display of available flights and successful selection.	Correct display of available flights and successful selection.	PASS
3	Test ticket booking and payment	Book a ticket and complete payment.	Successful ticket booking and payment process.	Successful ticket booking and payment process.	PASS
4	Test ticket cancellation	Cancel a booked ticket	Successful ticket cancellation	Successful ticket cancellation	PASS

4.4 Discussion of Results

All the features provided in the project and its operations have been presented as screenshots.

4.4.1 Front page

The below figure shows the front page of the terminal when the project is started.

Figure 4.1 Front Page

4.4.2 Registration page

The user has to first register and create an account to use this system by setting an username and a password.

Figure 4.2 Registration Page

4.4.3 Login page

The use has to login into the airline ticketing system using the username and password which was previously set.

Figure 4.3 Login Page

4.4.4 Main menu page

This main menu page will have different operations that you can perform on the airline ticketing system.

Figure 4.4 Main menu Page

4.4.5 Create a new ticket

This page is used to create a new ticket by filling the necessary details and then the page followed by seat selection where the user can select the available row and seat.

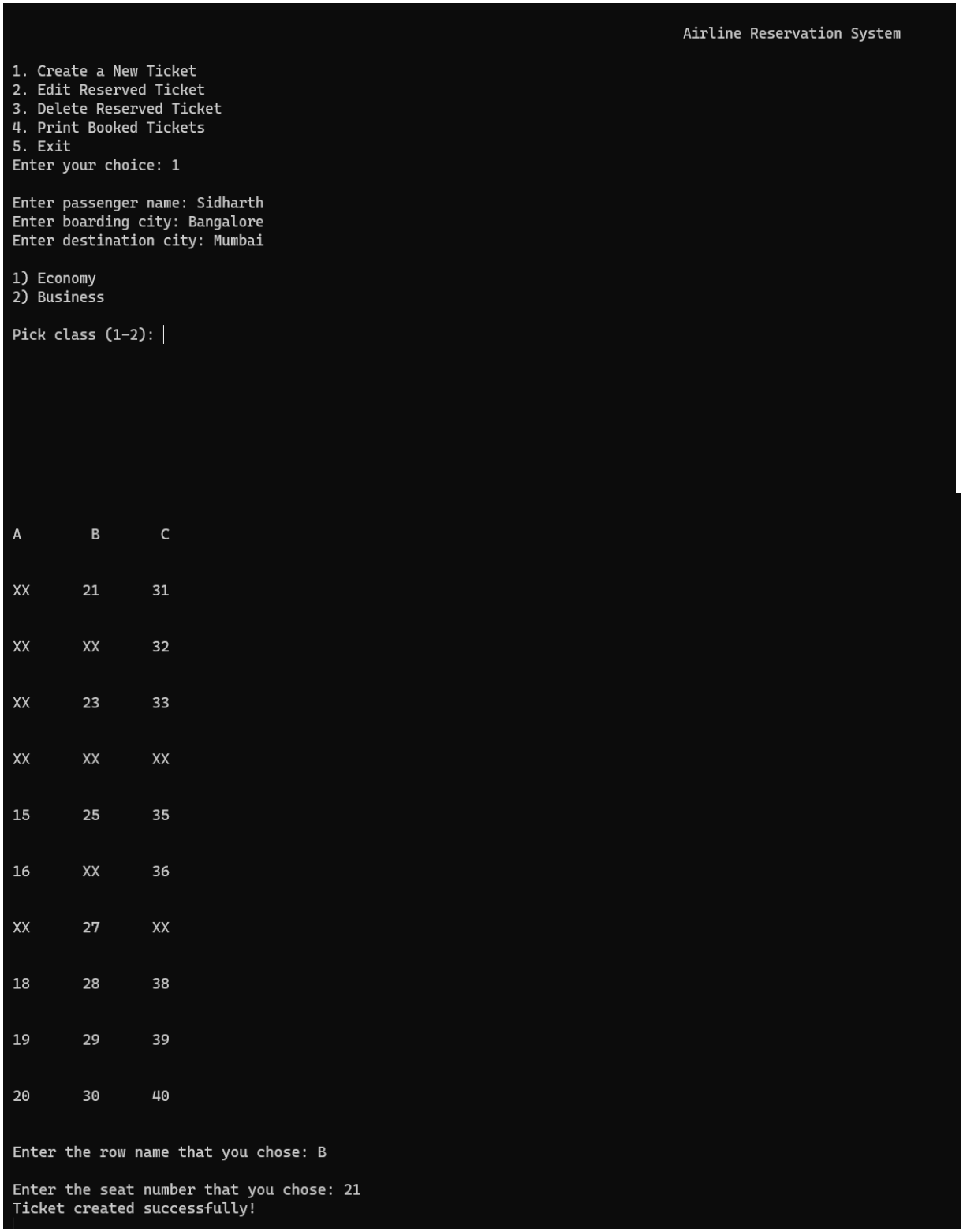


Figure 4.5 New ticket Page

4.4.6 Payment details

This page is used for the where payment is done by entering the card number and other details to complete the creation process.

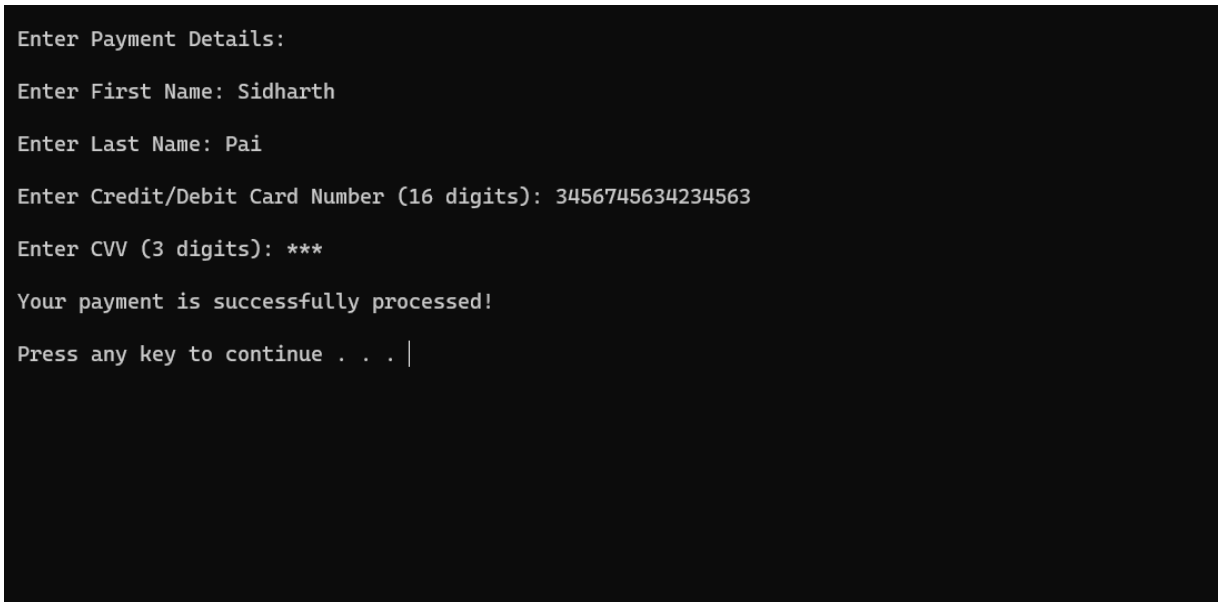


Figure 4.6 Payment Page

4.4.7 Display ticket

This page prints the already created airplane tickets showing all the details of the ticker holder.

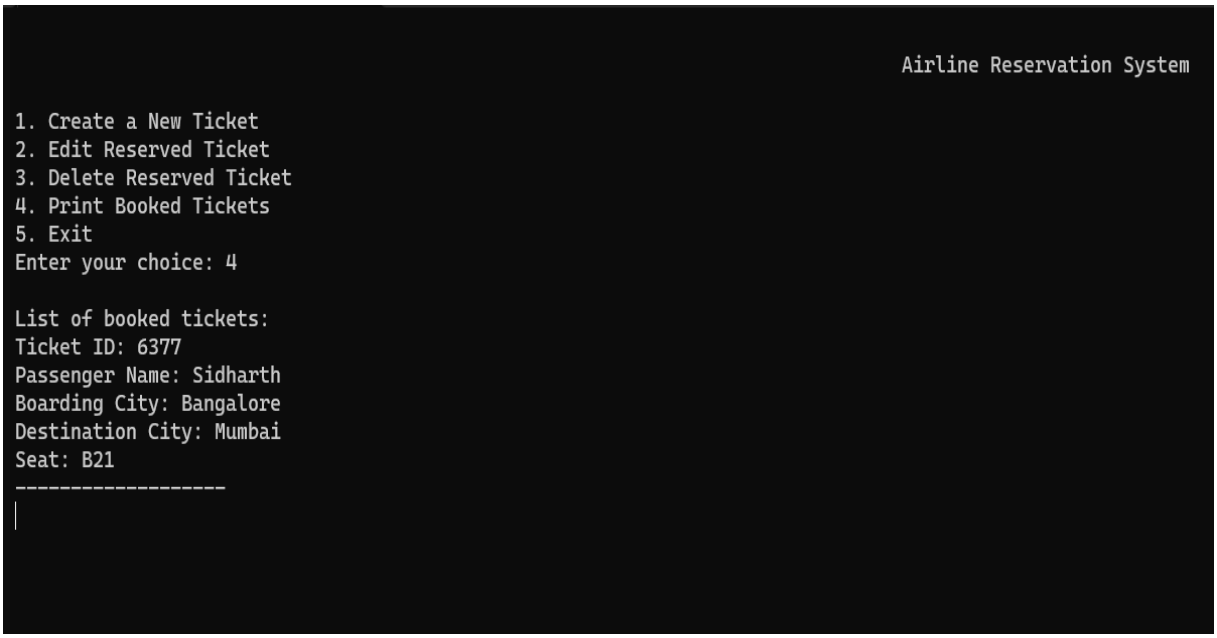


Figure 4.7 Display Page

4.4.8 Edit Ticket

This page shows that the user can edit the existing ticket and change all the details including seat number all over again .

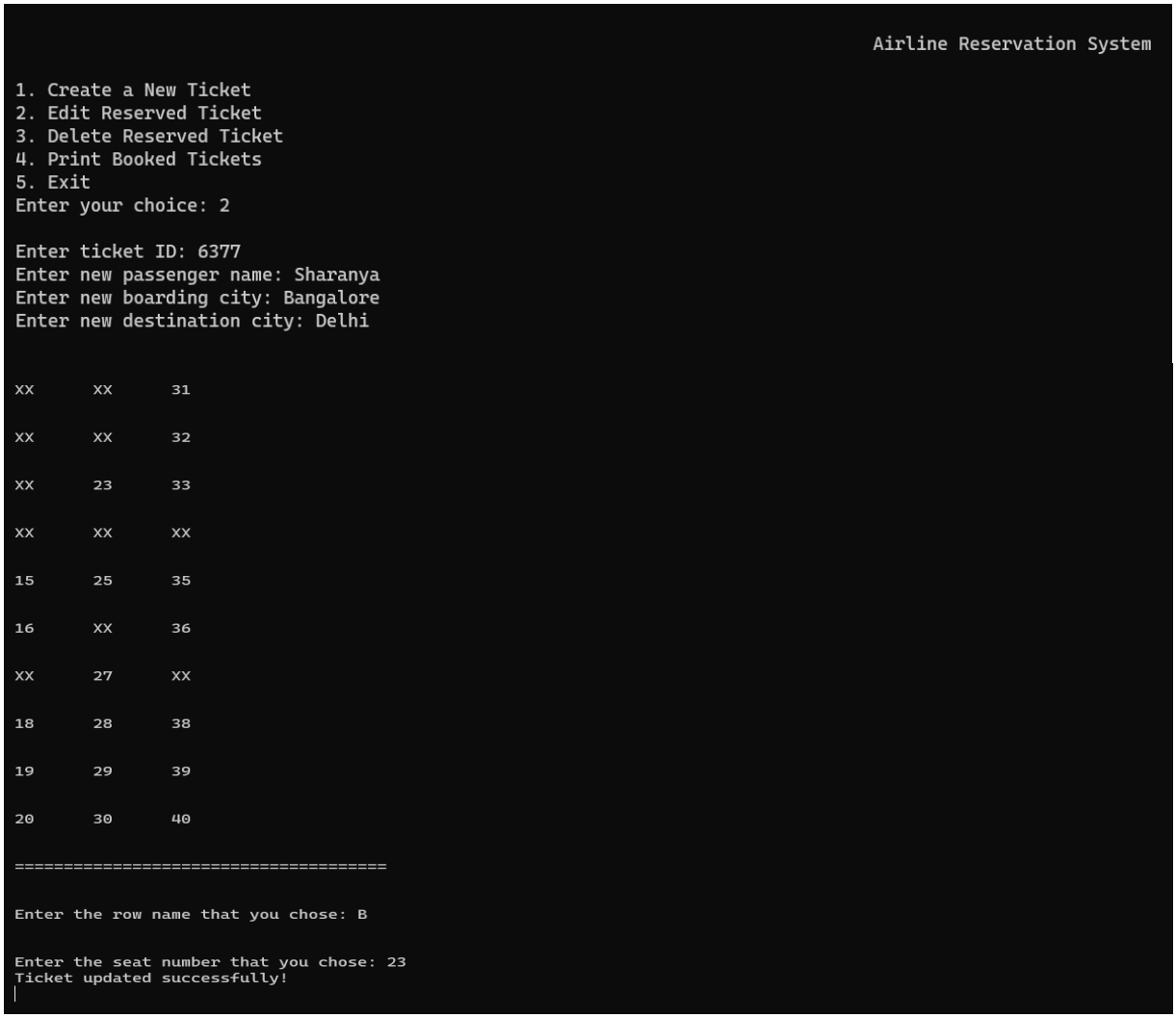


Figure 4.8 Edit ticket Page

4.4.9 Print edited ticket

This prints the ticket details after editing the existing one but the ticket ID remains the same.

Figure 4.9 Print ticket Page

4.4.10 Delete a ticket

This displays that we can enter the ticket ID and then delete the entire ticket.

Figure 4.10 Delete ticket Page

4.5 File Contents

4.5.1 Login Details File

Whenever someone logs into the airline system, their username and password are recorded and stored in the "Login_Details.txt" file. The administrator, or admin, has the ability to access and view the contents of this file.

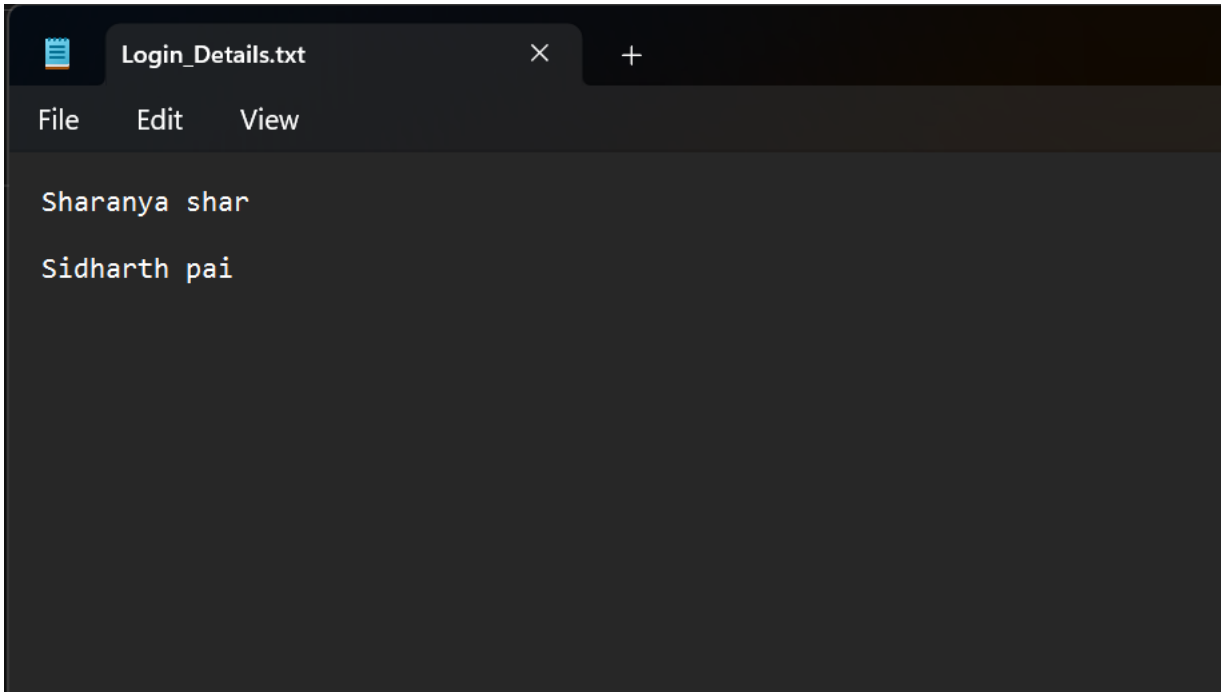


Figure 4.11 Login details file

4.5.2 Payment Details File

This file contains payment information related to customers who have booked airline tickets. The admin, or administrator, has the ability to access and view the contents of this file.

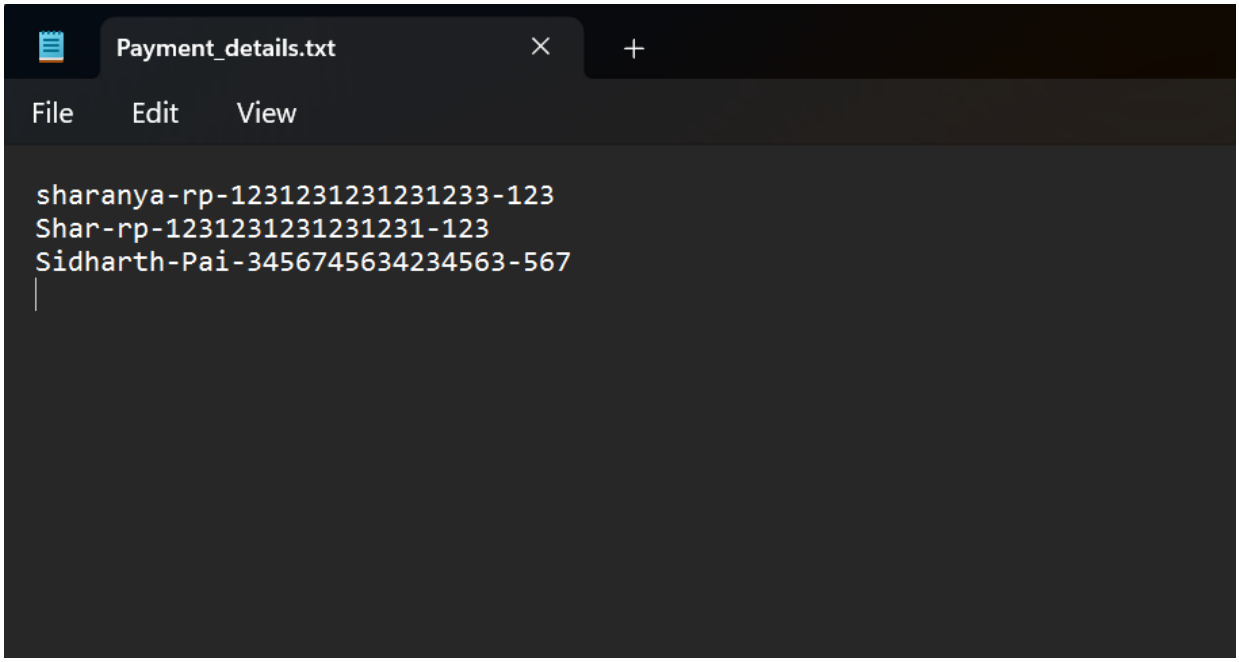


Figure 4.12 Payment details file

4.5.3 Seat Details File

This file contains information about the seats that customers have booked when purchasing tickets. The admin, or administrator, has the ability to access and view the contents of this file. When customers book airline tickets, their seat selections and related details are recorded and stored in the "Seat_Details" file.

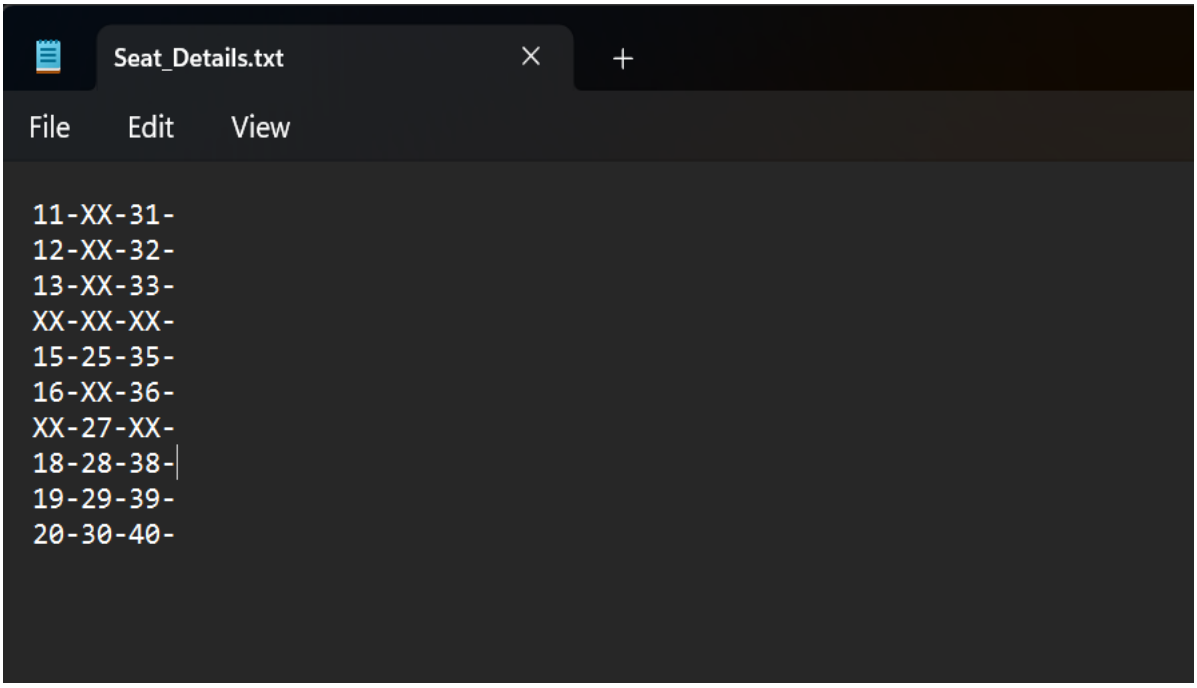


Figure 4.13 Seat details file

4.5.3 Booking Details File

This information includes the ticket ID, customer name, source (departure location), destination (arrival location), and the assigned seat number.

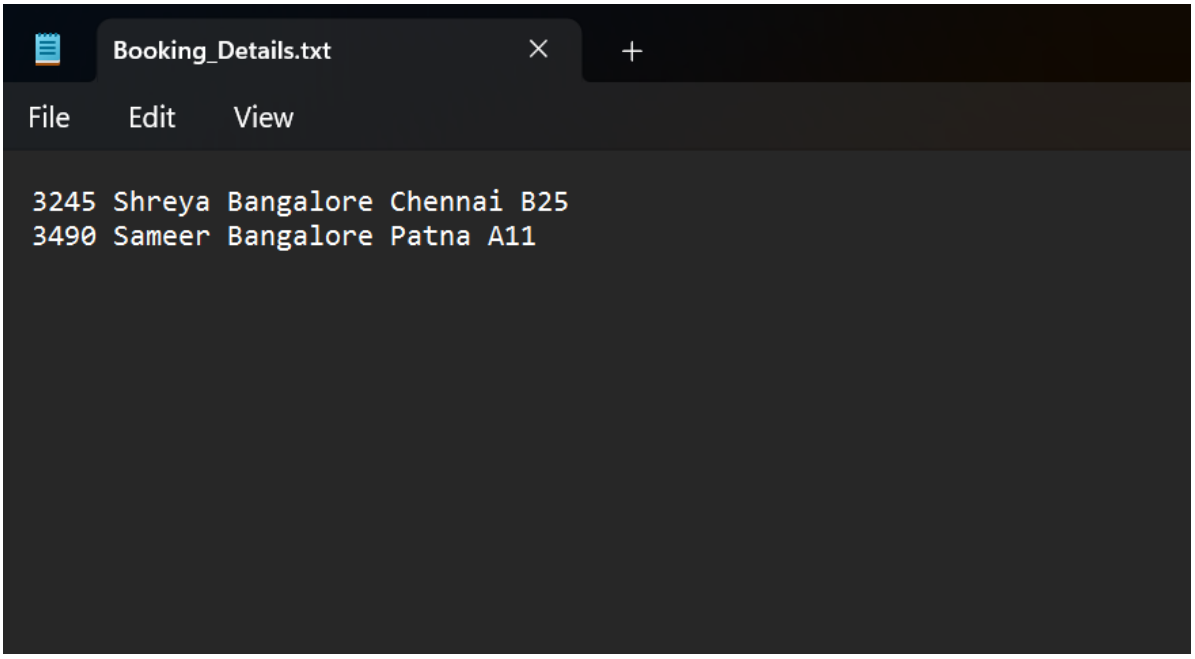


Figure 4.14 Booking details file

Chapter 5

CONCLUSION AND FUTURE ENHANCEMENTS

Conclusion

The project demonstrates a simple implementation of a ticket booking system. It allows users to register, login, and create tickets for various destinations. The project emphasizes user authentication and provides a basic interface for managing tickets.

Throughout the implementation, the project covers essential programming concepts such as file handling, user input validation, and data storage. It showcases the use of file operations to store and retrieve user login details and ticket information.

However, it's important to note that the provided code and algorithm represent a simplified version of a ticket booking system. In real-world scenarios, additional features like ticket availability checks, seat selection algorithms, and payment integrations would be necessary. Furthermore, robust error handling, data validation, and security measures would need to be implemented for a production-grade application.

Overall, this project serves as a starting point for building a more comprehensive and sophisticated ticket booking system, providing a foundation for further enhancements and customization based on specific requirements.

Future Enhancements

- Email Notifications: Adding an email notification system would keep users informed about their bookings, including confirmation emails, reminders, and updates. This feature would improve communication with users and provide them with important information regarding their reservations.
- Booking History and User Profiles: Creating a booking history for each user would allow them to view their past reservations and easily track their travel history. Additionally, implementing user profiles would enable users to manage their personal information, preferences, and notification settings.
- Social Media Integration: Integrating social media features would allow users to share their ticket bookings or invite friends to join them. This could help increase the system's visibility and attract new users through word-of-mouth referrals.
- Multilingual Support: Adding multilingual support would make the ticket booking system accessible to users from different regions and language backgrounds. It would enhance the user experience and make the application more inclusive.

REFERENCES

[1] https://www.predictiveanalyticstoday.com/top-airline-reservation-system/

[2] https://www.capterra.com/airline-reservation-system-software/

[3] https://stackoverflow.com

[4] https://www.getapp.com/hospitality-travel-software/airline-reservation-system/

[5] https://en.wikipedia.org/wiki/Airline_reservations_system

[6] https://www.provab.com/airline-reservation-system.html