CT5057 – Algorithms & Data Structures

Assignment 1

Date of Completion: 7th May 2025

Table of Contents

Introduction

This documentation details the comprehensive design and development processes of a flight scheduling application labelled as “Airlines FSP”, emphasizing the analysis and implementation of data structures and algorithms. This includes arrays, hash maps and trees combined with searching and sorting algorithms within the program. Highlighted by Shaffer (2001) in his preface, data structures play a crucial title in the optimization of code efficiency, especially as the complexity of tasks rise which the difficulty of data directly proportionally increases. Data structures provide the simplification of managing data for intended use. Furthermore, the inclusion of algorithms, which requires problem solving and logical reasoning is a necessity to generate solutions for simplistic or complex problems, embedding mathematical models to analyse the interaction of these algorithms against these problems for improved results (Hopcroft, Ullman and Aho, 1983). The topic is explored further in **Airlines FSP Development & Algorithmic Analysis**. Both computational definitions are essential to highlight the primary challenge to implement a fully functional flight scheduling program while ensuring optimal algorithmic efficiency in terms of space and time for high quality performance and justify their integration within the application.

Overview on Airlines FSP (Flight Scheduling Application)

The Airlines Flight Scheduling Program (FSP) is designed to efficiently manage passenger and flight information. A notable example of this software is Sabre Schedule Manager, integrated by 60% of the leading airlines globally, including Air Serbia and Alaska Airlines (Sabre, 2020). Primary stakeholders are general airline services, focused with flight planning, crew scheduling and real-time updating however, this application will incorporate limited essential features for passengers. The mandatory conditions for the application:

- All flights must be collected and printed.
- Passengers must be able to book and remove their availability for a flight.
- The scheduling system must handle operations using one searching algorithm and one sorting algorithm, incorporating efficient use of data structures.
- Relevant passenger information on booked flight must be accessible

Numerous tools and programming concepts were introduced and utilized to contribute with the design and development phases of Airlines FSP, discussed in the following section.

Programming Language & Tools

Decisions were made, particularly regardless the selection between Java and Python, two notorious high-level programming languages. While Python accumulates less memory space, reduces lines of code and offers superior code implementations compared to Java, the latter is more robust and strictly adheres to object-oriented programming principles, a notable weak point with Python, specifically with encapsulation (Khoirom et al, 2020). Both languages are viable options; however, due to personal preference and a heavy emphasis on OOP required, Java was chosen. Object-oriented programming offers the creation of relevant entities and features for the application. These entities or classes along with their behaviour and attributes are mentioned and designed using Unified Modelling Language diagrams discussed and evaluated in **Visual Data Representation Diagrams (UML)**. An online diagramming tool called LucidChart was used. IntelliJ IDEA IDE by JetBrains specifically for development in Java was chosen and used for the entirety of the Airlines FSP project.

System Requirements, Analysis & Design

Functional and non-functional requirements were generated, analysed and derived from the previously stated mandatory conditions while considering the usefulness and feasibility of each requirement for the main stakeholder – the passenger. After generation and analysis, several data representation diagrams were made to visually represent the involvement of requirements/features with entity relationships within the program along with key data attributes, behaviours and these interactions with one another for developer comprehension.

Functional & Non-Functional Requirements

Functional Requirements	**Non-Functional Requirements**
Must display all flights and relevant information for passengers.	Should display this information in a simplistic and organised manner + load up quicky.
Must allow passenger to select a flight using flight number.	Should load up chosen flight as quick as possible after user inputs flight number.
Must allow passenger to book a flight and choose class category + assigned with a seat number + access booked information.	Should support multiple passengers booking data.
Must allow passenger to cancel a booked flight.	Should remove a passenger quickly
Must add passenger to a waitlist if seat capacity in class category is full.	A passenger should be assigned a status: booked or waitlisted
Must book waitlisted passenger if passenger in booked flight cancels.	Passenger information should be collected and printed accurately

The program must be functional	The program should continue to operate after an action is committed (booking, loading flights, etc)

The formulated requirements above value the viewpoint of the sole primary stakeholder which are potential passengers seeking to book a flight therefore the fundamental functionality must be the feature to successfully a flight if there are available seats. Additional functionalities should include flight management, class and seating scheduling arrangements, accessibility to flight and passenger information derived from the mandatory conditions highlighted in **Overview on Airlines FSP (Flight Scheduling Program)**.

<u>Visual Data Representation Diagrams (UML)</u>

These Unified Modelling Language diagrams below documents the complete application system of Airlines FSP, defining basic interactions between classes and their functionality.

Use-Case Diagram

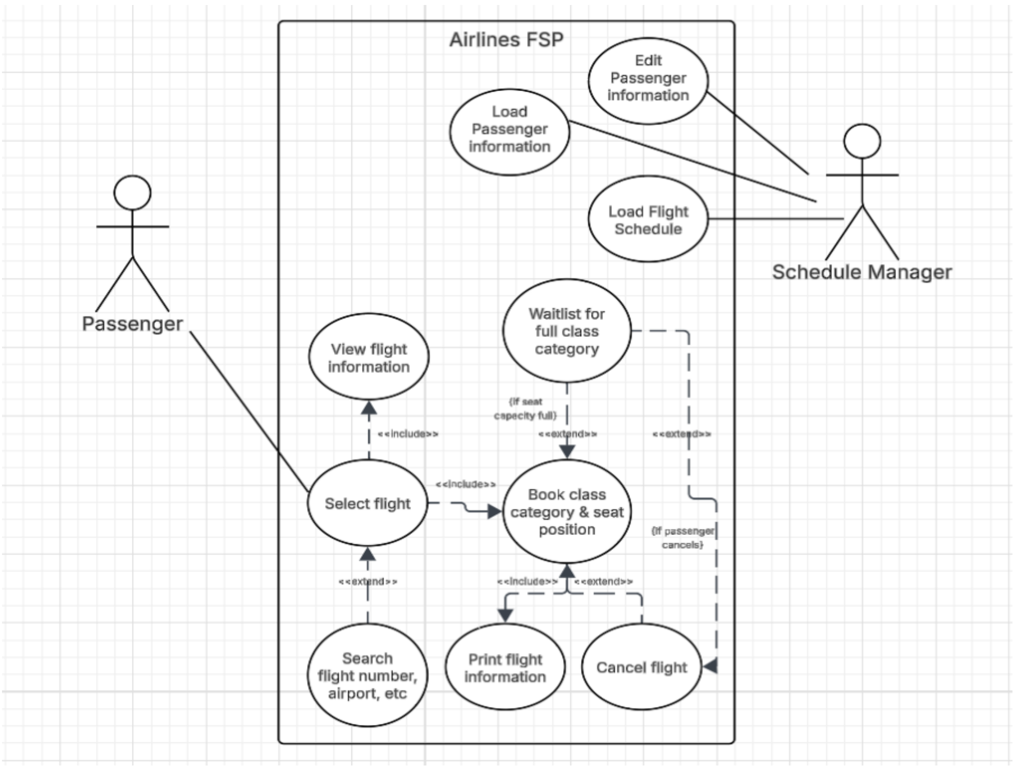


Figure 1 – Airlines FSP Use Case Diagram

Class Diagram

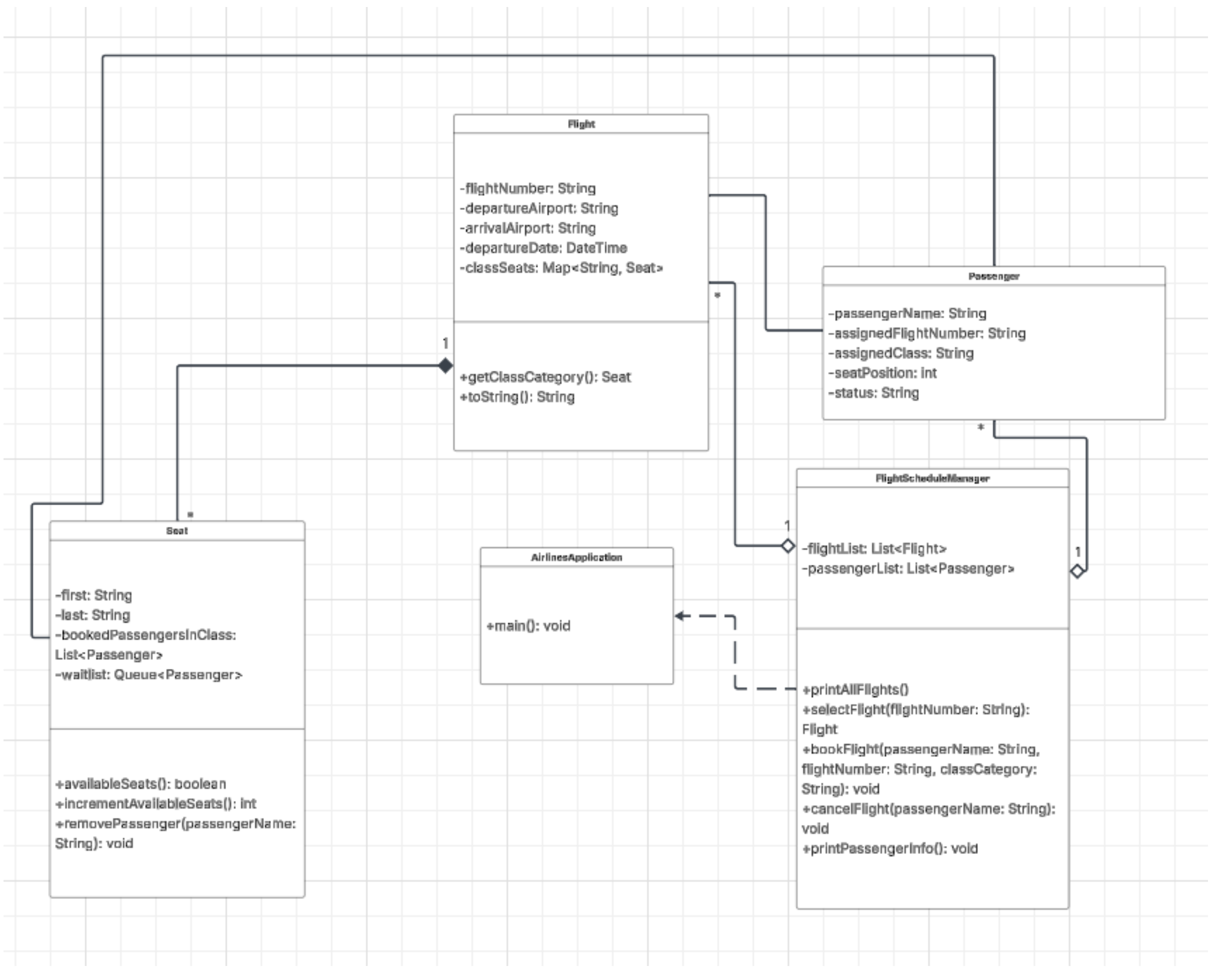
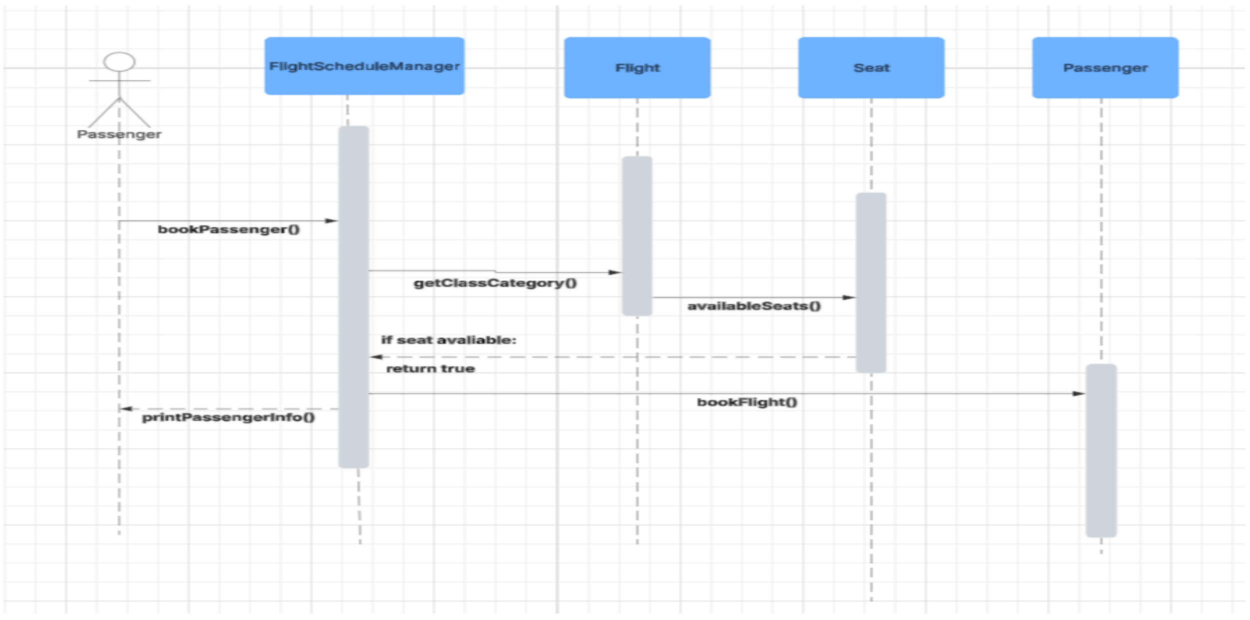

Flight
-flightNumber: String
-departureAirport: String
-arrivalAirport: String
-departureDate: DateTime
-classSeats: Map<String, Seat>
+getClassCategory(): Seat
+toString(): String
Passenger
-passengerName: String
-assignedFlightNumber: String
-assignedClass: String
-seatPosition: int
-status: String
FlightScheduleManager
-flightList: List<Flight>
-passengerList: List<Passenger>
+printAllFlights()
+selectFlight(flightNumber: String): Flight
+bookFlight(passengerName: String, flightNumber: String, classCategory: String): void
+cancelFlight(passengerName: String): void
+printPassengerInfo(): void
AirlinesApplication
+main(): void
Seat
-first: String
-last: String
-bookedPassengersInClass: List<Passenger>
-waitlist: Queue<Passenger>
+availableSeats(): boolean
+incrementAvailableSeats(): int
+removePassenger(passengerName: String): void
1
*
*
1
1
*

Figure 2 – Airlines FSP Class Diagram

Sequence Diagram



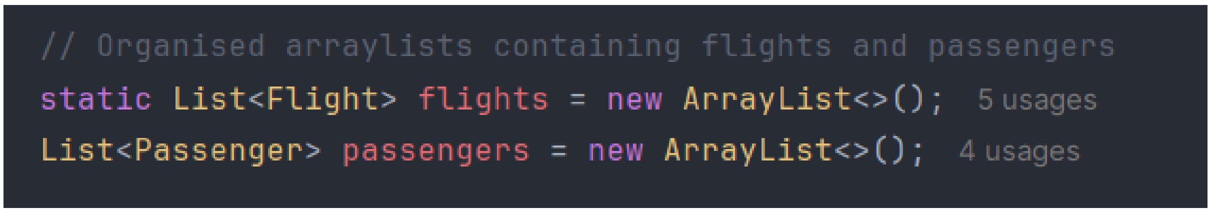
Airlines FSP Development & Algorithmic Analysis

Throughout the development on Airlines Flight Scheduling Program (FSP), several minor adjustments and mechanisms were configured that deviates from the early system requirements and UML diagrams as ongoing challenges generated. The adoption of Comma Separated Values (CSV) files to receive, modify and store passenger and flight information was one notable adjustment compared to implementing traditional relational database. It resulted in incorporating additional logic in certain appropriate classes to read and write data to the CSV files however they automatically operate simplistic features when reading and writing data. Furthermore, the data structures and algorithms involved with the intention of incorporating both searching and sorting algorithms for scheduling efficiency logic. Each data structure and algorithm were integrated based on assigned functionality during scheduling. Several methodologies are discussed when analysing the efficiency of each algorithm, comparing each methodology based on performance metrics to review the effectiveness of implementing each algorithm and data structures within the application.

Implementation of Data Structures

Every data structure, from primitive to non-primitive types, used in Airlines FSP were essential for managing and manipulating data for efficient use.

Lists with ArrayList Implementation

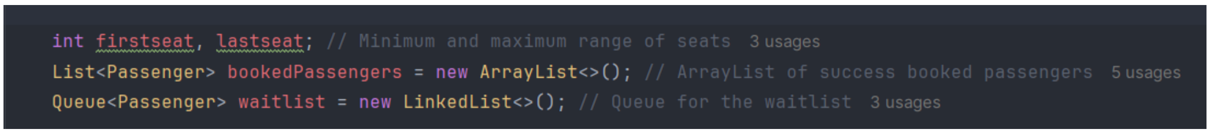


```
// Organised arraylists containing flights and passengers
static List<Flight> flights = new ArrayList<>();  5 usages
List<Passenger> passengers = new ArrayList<>();  4 usages
```

Figure 4 – Java lists to hold flights and passenger data

This data structure was implemented with the ArrayList class to maintain the records of flights, booked passengers in a specific class and all passengers, including those in the waitlist. This approach offered flexibility for functional programming, ensuring reliable dynamic sizing to continuously add and remove passenger information. The time and space complexity for adding elements is O(1) however these metrics are reduced to O(n) when elements are removed due to shift in elements (GeeksForGeeks, 2016). Within the scenario of adding and removing passengers, excluding flight schedules which are only accessed and printed, it is accepted however HashMap could offer more efficiency to add and remove passengers from the passenger CSV file by incorporating key-value pair (passenger ID paired with passenger name). An approach like this would delivers improved time and space complexity to remove elements compared to ArrayList implementation. No independent passenger identification was organised rather passengers were identified solely by their booked flight number. This implementation was acceptable, but approach could be better.

Queue with LinkedList Implementation

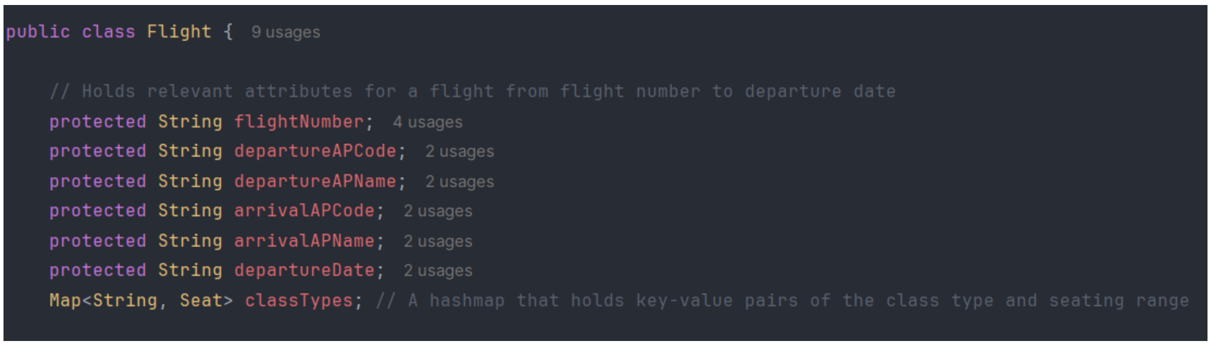


```
int firstseat, lastseat; // Minimum and maximum range of seats  3 usages
List<Passenger> bookedPassengers = new ArrayList<>(); // ArrayList of success booked passengers  5 usages
Queue<Passenger> waitlist = new LinkedList<>(); // Queue for the waitlist  3 usages
```

Figure 5 – “Seat” class attributes: Integers, List and Queue

The Queue Interface in Java utilizes the LinkedList class to establish and manage the waitlist for passengers when the seating capacity in a booked class category reaches its limit. This data structure is ideal when handling waitlists, operating on a First In First Out (FIFO) Order system. It ensures fair booking in the order of arrival in waitlist however a downside is the time and space complexity of O(n) to iterate and search for elements as the queue interface strictly adheres to FIFO, reducing flexibility and efficiency when complex algorithms are involved which do not follow FIFO. On average, time and space complexity of for adding and removing passengers using offer() and poll() functions is O(1), additionally benefitting with dynamic memory (GeeksForGeeks, 2016). As the waitlist grows, the assignment of pointers increases leading to increased memory usage to search in the waitlist due to the linear search nature of the LinkedList implementation, adding complexity compared to an ArrayList. Currently, the implementation of this data structure is acceptable for managing basic waitlist operations but a change in the implementation class to ArrayDeque class could enhance performance significantly as it allows elements to be removed at both ends due to array-based implementation (Bloch, J, 2018). In this case, the following operation is not necessary for a basic waitlist but rather for complex structures.

HashMap Implementation

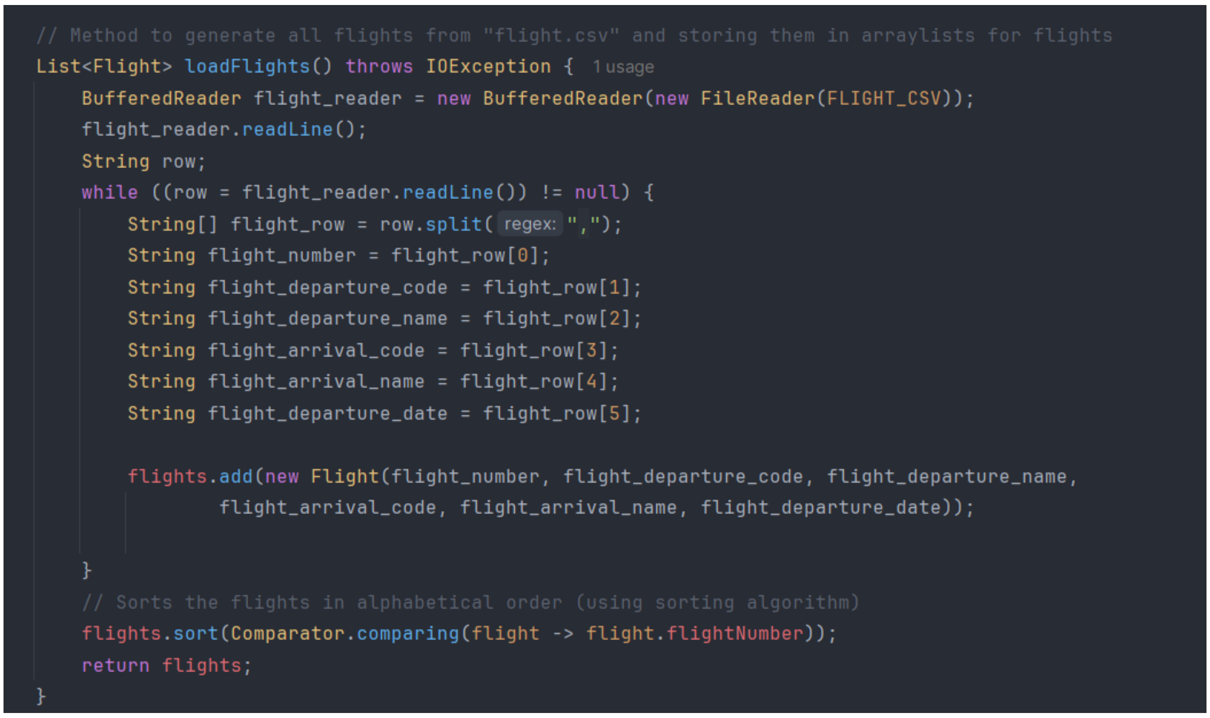


```
public class Flight {  9 usages

    // Holds relevant attributes for a flight from flight number to departure date
    protected String flightNumber;  4 usages
    protected String departureAPCode;  2 usages
    protected String departureAPName;  2 usages
    protected String arrivalAPCode;  2 usages
    protected String arrivalAPName;  2 usages
    protected String departureDate;  2 usages
    Map<String, Seat> classTypes; // A hashmap that holds key-value pairs of the class type and seating range
```

Figure 6 – Contains strings and a HashMap in a “Flight” class

The purpose was to associate the class category name paired with their corresponding seating range for user to specify the class category they wish to book (First Class/Business/Economy) and a function ‘getClassType()’ would return the instance of the “Seat” class. The retrieval time is O(1) due to the usage of key-value pairs therefore duplicates are prevented but this reason is the same explanation why the space complexity is O(n) which consumes more memory (Goodrich, Tamassia and Goldwasser, 2014). Mentioned in **Lists with ArrayList Implementation**, utilizing a HashMap for storing passenger information would be more useful than class categories and seat range pairs. It provides good OOP principle of encapsulation but sacrifices memory space. As an alternative, an EnumMap could be implemented instead of a HashMap (Bloch, J, 2018).

Sorting Algorithm Analysis (TimSort Algorithm)



```
// Method to generate all flights from "flight.csv" and storing them in arraylists for flights
List<Flight> loadFlights() throws IOException {  1 usage
    BufferedReader flight_reader = new BufferedReader(new FileReader(FLIGHT_CSV));
    flight_reader.readLine();
    String row;
    while ((row = flight_reader.readLine()) != null) {
        String[] flight_row = row.split( regex: ",");
        String flight_number = flight_row[0];
        String flight_departure_code = flight_row[1];
        String flight_departure_name = flight_row[2];
        String flight_arrival_code = flight_row[3];
        String flight_arrival_name = flight_row[4];
        String flight_departure_date = flight_row[5];

        flights.add(new Flight(flight_number, flight_departure_code, flight_departure_name,
                flight_arrival_code, flight_arrival_name, flight_departure_date));

    }
    // Sorts the flights in alphabetical order (using sorting algorithm)
    flights.sort(Comparator.comparing(flight -> flight.flightNumber));
    return flights;
}
```

Figure 7 – function to read, sort and collect all flights from “flight.csv” file

Implementation In Program

After collecting flight schedule data from the flight CSV file and organising each flight into a list, the list of flights is sorted in alphabetical order by applying the “List.sort(Comparator.comparing)” method which employs TimSort – a hybrid sorting algorithm that combines Merge Sort and Insertion Sort. In this scenario, the flight list is:

- Partially sorted using Insertion Sort.
- Divided into small “runs” which are decomposed lists containing the flights.
- The runs are merged using Merge Sort, accomplishing the sorting process

TimSort offers a time complexity of O(n log n), which is useful compared to other sorting algorithms like Selection and Insertion Sort with slower time complexities of O(n^2). Although Merge sort could have been an acceptable option, TimSort has a better best case time complexity of O(n) compared to O(n log n) from Merge Sort (Auger, Nicaud and Pivoteau, 2015). The is the most viable option to use before applying Binary Search.

Theoretical Analysis: Time Complexity

From combining Insertion and Merge Sort algorithms, it applies the time complexity of Insertion sort to the unsorted list, resulting in a slow time complexity of O(n^2) however, due to partial sorting, the time complexity improves to O(n). The list is split into runs where Merge sort is executed on the partially sorted list, providing a time complexity of O(n log n) therefore the overall time complexity of the sorting algorithm: O(n) + O(n log n) = O(n log n).

Average Case: O(n log n) – the merging process from Merge Sort on the partially sorted flight list dominates Insertion Sort.

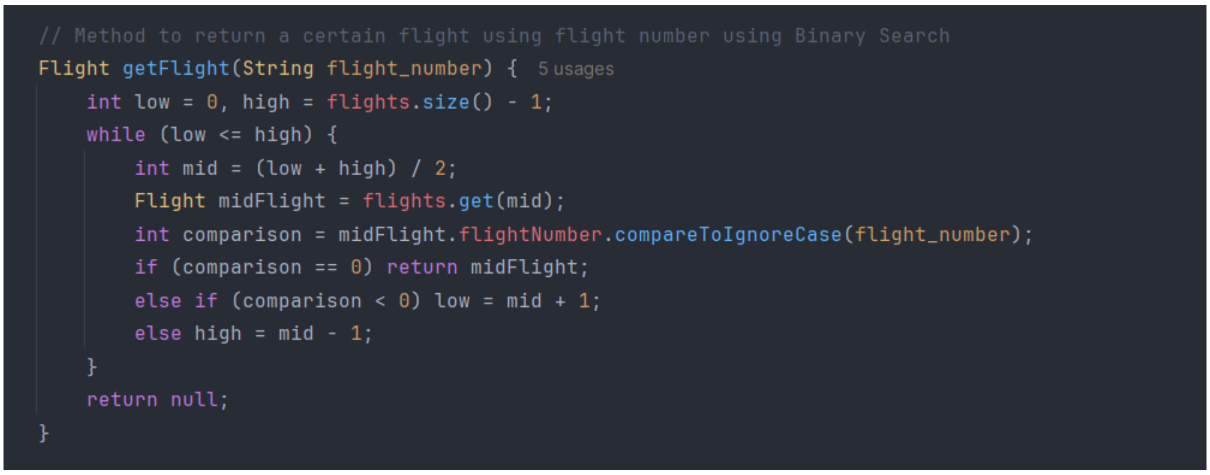
Best Case: O(n) – the flight list is entirely sorted, requiring minimal to zero implementation of Merge Sort.

Worst Case: O(log n) -> the merge sort algorithm is significantly engaged with sorting.

Theoretical Analysis: Space Complexity

Derived from the space complexity of Merge Sort, which requires memory space to partially or completely sort runs. This gives a space complexity of O(n).

Searching Algorithm Analysis (Binary Search)



```java
// Method to return a certain flight using flight number using Binary Search
Flight getFlight(String flight_number) {  5 usages
    int low = 0, high = flights.size() - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        Flight midFlight = flights.get(mid);
        int comparison = midFlight.flightNumber.compareToIgnoreCase(flight_number);
        if (comparison == 0) return midFlight;
        else if (comparison < 0) low = mid + 1;
        else high = mid - 1;
    }
    return null;
}
```

Figure 8 – function to get specific flight from flight number using Binary Search

Implementation In Program

The "getFlight()" function applies Binary Search to locate a specific flight using the flight number, provided as a parameter. Given that the built-in Java TimSort sorted the list of flights by flight number prior, employing Binary Search is appropriate and an exact value is provided and reused. The following function operates by:

- Receives the flight number entered by the user.
- Commits a Binary Search on the sorted list of flights where the middle element is compared with flight number, if the flight numbers match the flight is returned.
- If flight number is alphabetically lower (e.g. EA739 < ZT473), the search continues with the left side.

- If flight number is alphabetically higher (e.g. EA739 > AA221), the search continues with the right side.

- This process is repeated until the flight number is found or declared null.

- The function returns the flight or returns null.

Binary Search possesses a time complexity of O(log n) and a space complexity of O(1) which more efficient than linear search with a time complexity of O(n) (Lin, 2019). This is useful long-term as the list of flights grows but the flight schedule is fixed in this certain scenario.

Theoretical Analysis: Time Complexity

Assuming the list was sorted prior because if it is not sorted, the searching algorithm will fail. The algorithm applies a divide and conquer approach, constantly dividing the search space by 2 where "middle_value = left + (right – left)/2" using the "compareTo()" method to compare alphabetical characters when dealing with strings (Bloch, J, 2018)

Average Case: O(log n) - total iterations between flight numbers would be O(log n) as one comparison would be O(m log n) where m shows average length in a flight number string

Best Case: O(1) - flight number specified is found in the middle of the list

Worst Case: O(log n) -> flight number specified not found in the list or found after final comparison therefore no significant change in m

Theoretical Analysis: Space Complexity

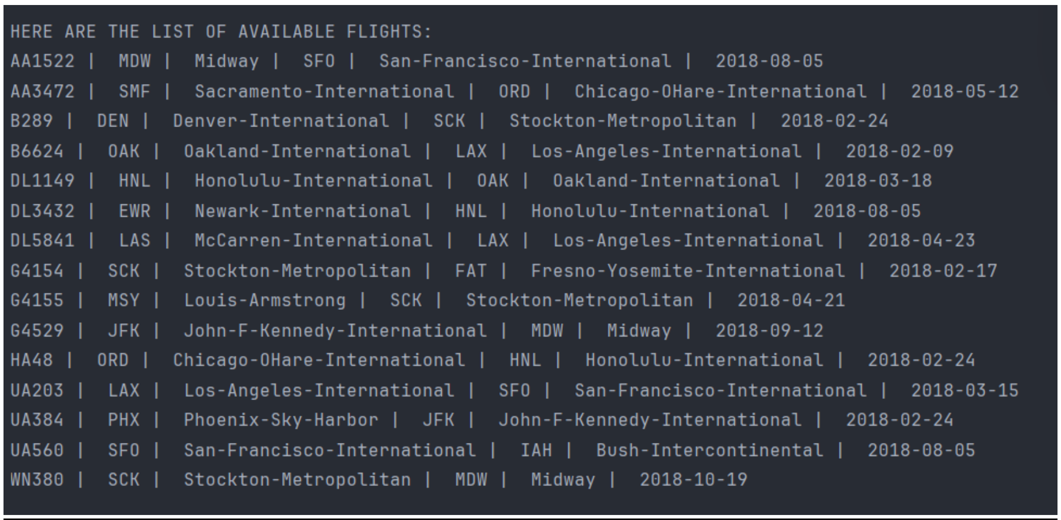
The number of flights stored in the list is O(n) but the algorithm uses constant values for left, right and middle indexes with no additional data structures therefore space complexity: O(1)

Testing

Unit and Integration Testing - Test Suite

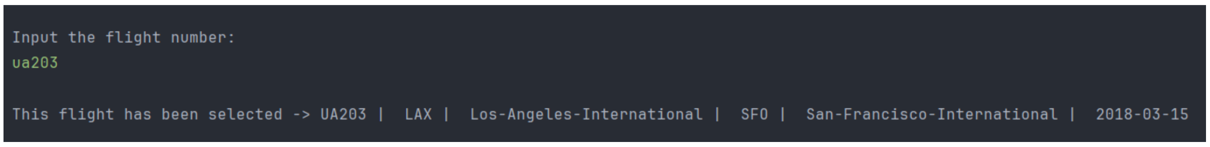
Test ID	Test Action	Inputs Required	Expected Results	Actual Results	Status
001	Load and print flights from flight CSV file	User should select the option to book a flight	Flights are read from CSV file and printed	Flights are read from CSV file and printed	Passed

			shown in Figure 9		
002	Find and print specific flight using flight number	User inputs flight number after requesting to book a flight	Specified flight is printed shown in Figure 10	Specified flight is printed	Passed
003	Save passenger in CSV file with relevant information after booking	Passenger inputs name, flight number and class category	Passenger information saved in passenger CSV file	Passenger information partially saved in passenger CSV file	Failed
004	Load passengers from CSV file with relevant information	Passenger requests passenger information to be printed	Passenger information is printed	Passenger information is printed	Passed
005	Passengers placed in waitlist	Passenger chooses to book class but seat range full	Passenger stored in waitlist + status is deemed as “Waitlisted”	Passenger stored in waitlist + status is deemed as “Waitlisted”	Passed



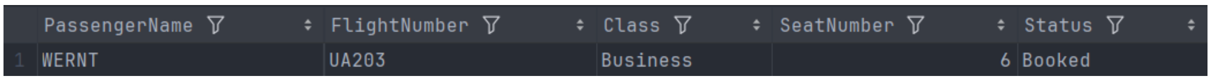
```
HERE ARE THE LIST OF AVAILABLE FLIGHTS:
AA1522 |  MDW |  Midway |  SFO |  San-Francisco-International |  2018-08-05
AA3472 |  SMF |  Sacramento-International |  ORD |  Chicago-OHare-International |  2018-05-12
B289 |  DEN |  Denver-International |  SCK |  Stockton-Metropolitan |  2018-02-24
B6624 |  OAK |  Oakland-International |  LAX |  Los-Angeles-International |  2018-02-09
DL1149 |  HNL |  Honolulu-International |  OAK |  Oakland-International |  2018-03-18
DL3432 |  EWR |  Newark-International |  HNL |  Honolulu-International |  2018-08-05
DL5841 |  LAS |  McCarren-International |  LAX |  Los-Angeles-International |  2018-04-23
G4154 |  SCK |  Stockton-Metropolitan |  FAT |  Fresno-Yosemite-International |  2018-02-17
G4155 |  MSY |  Louis-Armstrong |  SCK |  Stockton-Metropolitan |  2018-04-21
G4529 |  JFK |  John-F-Kennedy-International |  MDW |  Midway |  2018-09-12
HA48 |  ORD |  Chicago-OHare-International |  HNL |  Honolulu-International |  2018-02-24
UA203 |  LAX |  Los-Angeles-International |  SFO |  San-Francisco-International |  2018-03-15
UA384 |  PHX |  Phoenix-Sky-Harbor |  JFK |  John-F-Kennedy-International |  2018-02-24
UA560 |  SFO |  San-Francisco-International |  IAH |  Bush-Intercontinental |  2018-08-05
WN380 |  SCK |  Stockton-Metropolitan |  MDW |  Midway |  2018-10-19
```

Figure 9 – flight schedule collected from CSV file and printed



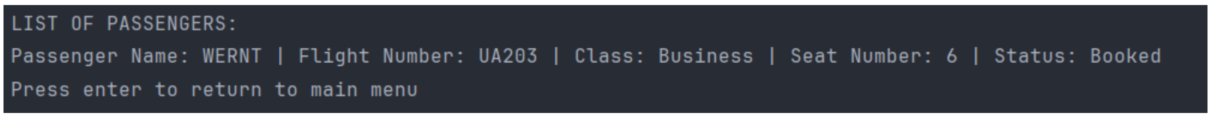
```
Input the flight number:
ua203

This flight has been selected -> UA203 |  LAX |  Los-Angeles-International |  SFO |  San-Francisco-International |  2018-03-15
```

Figure 10 – specified flight is printed using flight number



	PassengerName	FlightNumber	Class	SeatNumber	Status
1	WERNT	UA203	Business	6	Booked

Figure 11 – passenger information stored in passenger CSV file



```
LIST OF PASSENGERS:
Passenger Name: WERNT | Flight Number: UA203 | Class: Business | Seat Number: 6 | Status: Booked
Press enter to return to main menu
```

Figure 12 – reads and prints passenger CSV file

Conclusion

In conclusion, this project was a success however Airlines FSP had many bugs at the end but fulfilled multiple mandatory features while applying the necessary data structures and algorithms. Possible improvements include more efficient data structures using algorithms.

Reference List

- Aho Alfred V and Aho (1983). *Data Structures and Algorithms*. Pearson Education India.

- Bloch, J. (2018). *Effective Java*. Boston: Addison-Wesley.

- GeeksforGeeks (2016). *List Interface in Java with Examples - GeeksforGeeks*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/list-interface-java-examples/.

- Goodrich, M.T., Tamassia, R. and Goldwasser, M.H. (2014). *Data Structures and Algorithms in Java*. Wiley Global Education.

- Sabre (2020). *Schedule Manager*. [online] Sabre. Available at: https://www.sabre.com/products/suites/network-planning-and-optimization/schedule-manager/.

- Shaffer, C.A. (2010). *A practical introduction to data structures and algorithm analysis*. Upper Saddle River, N.J.: Prentice Hall.

- Storer, J.A. (2012). *An Introduction to Data Structures and Algorithms*. Springer Science & Business Media.

- Lin, A., 2019. Binary search algorithm. WikiJournal of Science, 2(1), pp.1-13.

- Auger, N., Nicaud, C. and Pivoteau, C., 2015. Merge Strategies: From Merge Sort to TimSort.