**Flight Reservation System**

**Formal Report**

DATA STRUCTURES & ALGORITHMS

By

CSY22054 & CSY22091

[**1. Introduction 2**](#_gfkhcz395e1r)

[**2. Detailed Description 2**](#_qmy10suhe8nm)

[**3. Design: 3**](#_eh0mi4fmyx1r)

[**3.1 ADT: FlightGrapImpl 3**](#_p5znqy46jked)

[3.2 ADT: Flight 5](#_gi51jml5f3ma)

[3.3 ADT: Airport 6](#_1vh5n2h0twbr)

[3.4 ADT: Passenger 8](#_wp3ltr60bcfx)

[3.5 Dijkstra's Algorithm Implementation 9](#_pyqomn4z9aux)

[**4. Testing 10**](#_gd3e2jijhvho)

[**5. Conclusion 12**](#_3n288ijdloy)

[**6. References 12**](#_istqmlmp5x33)

# Introduction

The Flight Reservation System is a Java-based application designed to manage flight information, airport details, flight bookings, and more. This report provides an overview of the system's functionality, including the main components and key features.

# 2. Detailed Description

When we started thinking about how we were going to build this project we thought about how an actual airport works and we asked ourselves what the airport needs and what it does not need so that it can function. After collecting all the necessities we came up with the following ADTs:  
Airport, AvailableFlight, Flight, Passenger and App. Helping us build the project we also created a graph, FlightGraph, and an implementation for it, FlightGraphImpl. Finally we have our test case ADTs, AirportTest, AvailableFlightTest, FlightTest, PassengerTest, FlightGraphTest, FlightGraphImplTest, PerformanceTest that are mentioned more towards the end of this report.

# 3. Design:

## **3.1 ADT: FlightGrapImpl**

* addAirport(String airportCode, String airportName)

Preconditions:

airportCode and airportName should not be null. The airport code should be unique.

Postconditions:

A new airport is added to the FlightGraphImpl instance with the provided code and name.

* addFlight(String departureAirport, String destinationAirport, String flightNumber, String departureTime, String arrivalTime, int totalSeats, double price)

Preconditions:

DepartureAirport and destinationAirport should be valid airports in the system.

FlightNumber should be unique.

TotalSeats should be greater than 0.

Price should be greater than or equal to 0.

Postconditions:

A new flight is added to the FlightGraphImpl instance connecting the specified departure and destination airports.

* searchForFlights(String departureAirport, String destinationAirport, String date)

Preconditions:

departureAirport and destinationAirport should be valid airports in the system.

date should be a valid date format.

Postconditions:

Returns a list of AvailableFlight objects representing available flights between the specified airports on the given date.

* bookFlight(String flightNumber, String passengerName, int seatNumber)

Preconditions:

flightNumber should correspond to an existing flight.

passengerName should not be null or empty.

seatNumber should be a valid seat number.

Postconditions:

If the booking is successful, a seat on the specified flight is booked for the passenger.

* deleteFlight(String flightNumber)

Preconditions:

flightNumber should correspond to an existing flight.

Postconditions:

If the flight is found, it is removed from the system.

## **3.2 ADT: Flight**

Constructor

Preconditions:

FlightNumber, departureTime, arrivalTime, totalSeats, and price should not be null.

TotalSeats and price should be greater than 0.

Postconditions:

A new Flight object is created with the provided details.

* bookSeat(Passenger passenger)

Preconditions:

Passenger should not be null.

Postconditions:

If there are available seats, the passenger is added to the list of booked passengers, and the remaining seats are updated.

## **3.3 ADT: Airport**

Constructor

Preconditions:

airportCode and airportName should not be null.

Postconditions:

A new Airport object is created with the provided code and name.

* addFlight(Flight flight)

Preconditions:

Flight should not be null.

Postconditions:

The provided flight is added to the list of flights associated with the airport.

* hasFlight(String flightNumber)

Preconditions:

flightNumber should not be null.

Postconditions:

Returns true if a flight with the specified flight number is found in the list of flights; otherwise, returns false.

* getFlightTo(String destinationAirportCode)

Preconditions:

destinationAirportCode should not be null.

Postconditions:

Returns the first flight from the airport to the specified destination airport code or null if no such flight is found.

* removeFlight(String flightNumber)

Preconditions:

flightNumber should not be null.

Postconditions:

If a flight with the specified flight number is found, it is removed from the list of flights associated with the airport. Returns true if removal is successful; otherwise, returns false.

## 

## **3.4 ADT: Passenger**

Constructor

Preconditions:

name should not be null or empty.

seatNumber should be a valid seat number.

Postconditions:

A new Passenger object is created with the provided name and seat number.

## 

## **3.5 Dijkstra's Algorithm Implementation**

(1)The searchForFlights method implements Dijkstra's algorithm to find the shortest path between airports based on the total prices. It uses priority queues and maps to track distances and previous nodes. We thought it was a cool idea to implement the Dijkstra’s algorithm for the reason that it fits like a glove in our situation so we studied it and implemented it into our project. (2)

while (!priorityQueue.isEmpty()) {

String currentAirport = priorityQueue.poll();

Airport current = airports.get(currentAirport);

for (Flight flight : current.getFlights()) {

double newPrice = totalPrices.get(currentAirport) + flight.getPrice();

if (newPrice < totalPrices.get(flight.getDestination().getAirportCode())) {

totalPrices.put(flight.getDestination().getAirportCode(), newPrice);

previousNodes.put(

flight.getDestination().getAirportCode(),

currentAirport);

priorityQueue.add(flight.getDestination().getAirportCode());

}

}

}

# 

# 

# 4. Testing

The chosen ADT for performance testing is **FlightGraphImpl**.

**Results**:

10 Airports: Theoretical Complexity: O((V + E) \* log(V)) = O((10 + 10) \* log(10)) = O(20 \* 2) = O(40)

Actual Performance: Measured time ≈ 1ms

100 Airports: Theoretical Complexity: O((V + E) \* log(V)) = O((100 + 100) \* log(100)) = O(200 \* 2) = O(400)

Actual Performance: Measured time ≈ 1ms

1000 Airports: Theoretical Complexity: O((V + E) \* log(V)) = O((1000 + 1000) \* log(1000)) = O(2000 \* 3) = O(6000)

Actual Performance: Measured time ≈ 9ms

**Data Generation**:

The data was generated using a loop and random values for airports and flights. This approach ensures a more controlled and reproducible test environment.

**Observations:**

* As expected, the theoretical complexity indicates that the operation should take more time as the number of airports increases.
* The actual measured time depends on various factors and may not precisely follow the theoretical complexity due to real-world considerations.
* If the measured time scales roughly in line with the theoretical complexity, it suggests that the algorithm implementation is performing as expected.

# 5. Conclusion

In summary, the Flight Reservation System is a Java-based application that efficiently manages flight information and bookings. The report covers key components, including FlightGraphImpl, Flight, Airport, and Passenger, outlining essential operations such as adding airports and flights, searching for available flights, and booking seats. The system ensures integrity through well-defined preconditions and postconditions. This concise report serves as a comprehensive guide to the data structures supporting the Flight Reservation System.

# 6. References

1. M. Sambol, “Dijkstra's algorithm in 3 minutes”, USA, 2014. [Online].

Available: <https://www.youtube.com/watch?v=_lHSawdgXpI>

1. M. Noto and H. Sato, "A method for the shortest path search by extended Dijkstra algorithm," Smc 2000 conference proceedings. 2000 ieee international conference on systems, man and cybernetics. 'cybernetics evolving to systems, humans, organizations, and their complex interactions' (cat. no.0, Nashville, TN, USA, 2000, pp. 2316-2320 vol.3, doi: 10.1109/ICSMC.2000.886462.

<https://ieeexplore.ieee.org/abstract/document/886462>