Data Structures & Algorithms:

Flight Scheduling Application

Nathan Hewett

August 2022

Table of Contents

Font Conventions

This report contains the following typographical conventions.

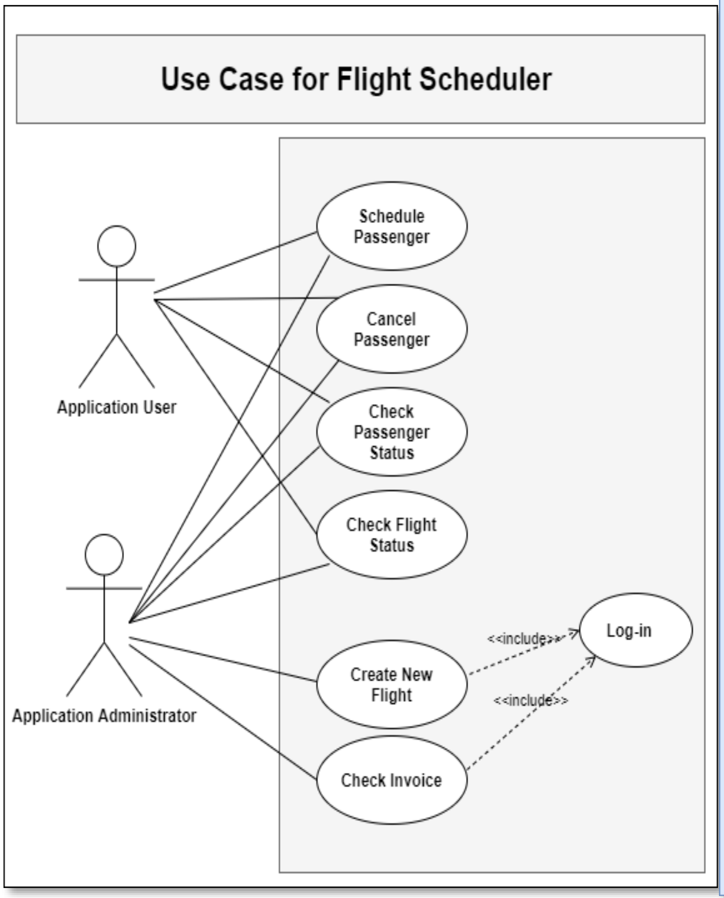
Italic:	Used for filenames, pathnames, pseudo code and emphasis.
`Consolas:`	Used for program code, methods, commands, and database contents.

1 Introduction

This application has been created to provide an efficient command line platform for flight operations, including; the scheduling and cancelling of passengers, allocating waiting lists, and retrieving statuses and invoice details. The data of which, is stored within accompanying text files. To ensure the optimal performance of the program, specific data structures and algorithms have been selected that produce minimal computation. As such, the time complexity of each algorithm has been analysed and compared, in an attempt to bring all operations as close to constant time as possible. Likewise, the space efficiency has been considered to ensure unnecessary memory allocation is not consumed. However, because this application has been developed for a computer, the space efficiency is less essential than that of the time complexity, due to the large resources of available memory. When analysing efficiency, the complexity is described using the *Big O* notation. This is effectively the measure of growth of a function, in terms of the number of operations it requires (Mishra and Garg, 2008, pp.363 – 364). Generally, the best-case complexity is considered unimportant due to its rarity and overall effect on the program. However, the worst-case is significant, due to the potentially detrimental effect of the application's performance (Ford and Topp, 2005, pp.125 - 131). Therefore, when selecting an appropriate algorithm or data structure, the worst-case performance has been a significant factor. These efficiencies are discussed further when demonstrating their uses within the application.

2 Application Design

The programming language used for this project is Java; JDK:17. To provide modularity and scalability, the program has been structured using the principles of *Object-Orientated Programming*. Furthermore, the software development cycle chosen is *Object-Orientated Development,* as this method emphasises the creation of objects in driving the process of development (Britton and Doake, 2005, pp.7 - 10). As demonstrated in the *Figure 1; Use Case Diagram,* this application is intended for two user categories. The first of which can perform all operations involving the scheduling and cancelling of passengers, with the addition of retrieving the statuses of passengers and flights. The second user type can perform all the aforementioned functions, with further features including; creating new flights and checking invoice-information. The behaviour of the application is displayed in the *Figure 2; Sequence Diagram*, which demonstrates the communication and interaction of objects. To visualise the specific behavioural aspects, the *Figure 3 Activity Diagram* demonstrates the associated processes required for each function. This assists in analysing the functions from a non-technical aspect (Kimmel, 2005, pp.48 – 77). As shown, all operations are performed from the Application Menu*,* and in most cases will receive input from the user to insert or retrieve information from stored data. The Application Menu class is created using the *Singleton* design-*pattern*, therefore ensuring there can only be one instance of the program's menu (Ford and Topp, 2005, pp.98 - 99). *Unified Modelling Language* (UML) diagrams visualise the arrangement of classes, including inheritance (Kimmel, 2005, pp.102 - 114). A UML overview of all classes is viewable in *Figure 4.* As exhibited, generally all subclasses are inherited from either the **`AirlineObject`** Abstract class, or **`Database`** Interface. This implementation provides objects for all flight entities, with allocated classes to perform the entities' database functions. *Figure 5* demonstrates **`AirlineObject`** classes with associated methods. Likewise, *Figure 6* displays database functionalities, with the addition of **`DepartureDatesRadixSort`** and **`InvoiceBinarySearchTree`***,* which contain *Radix* sorting and a *Binary Search Tree*, respectively.



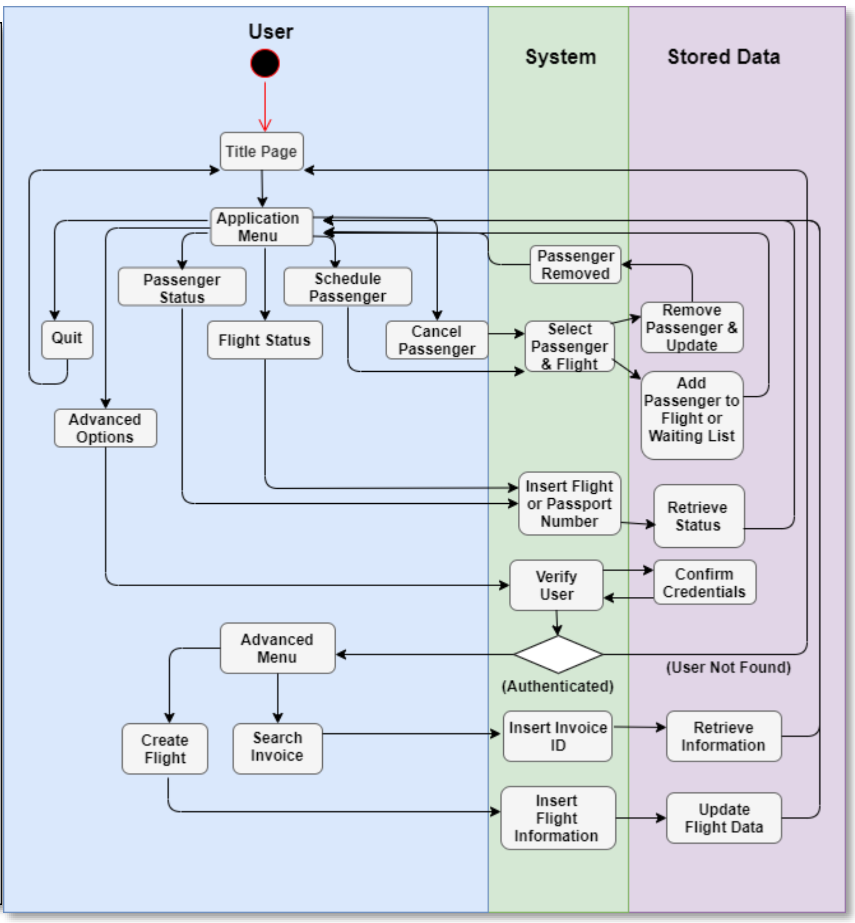


Figure 1: Use Case Diagram

Figure 2: Activity Diagram

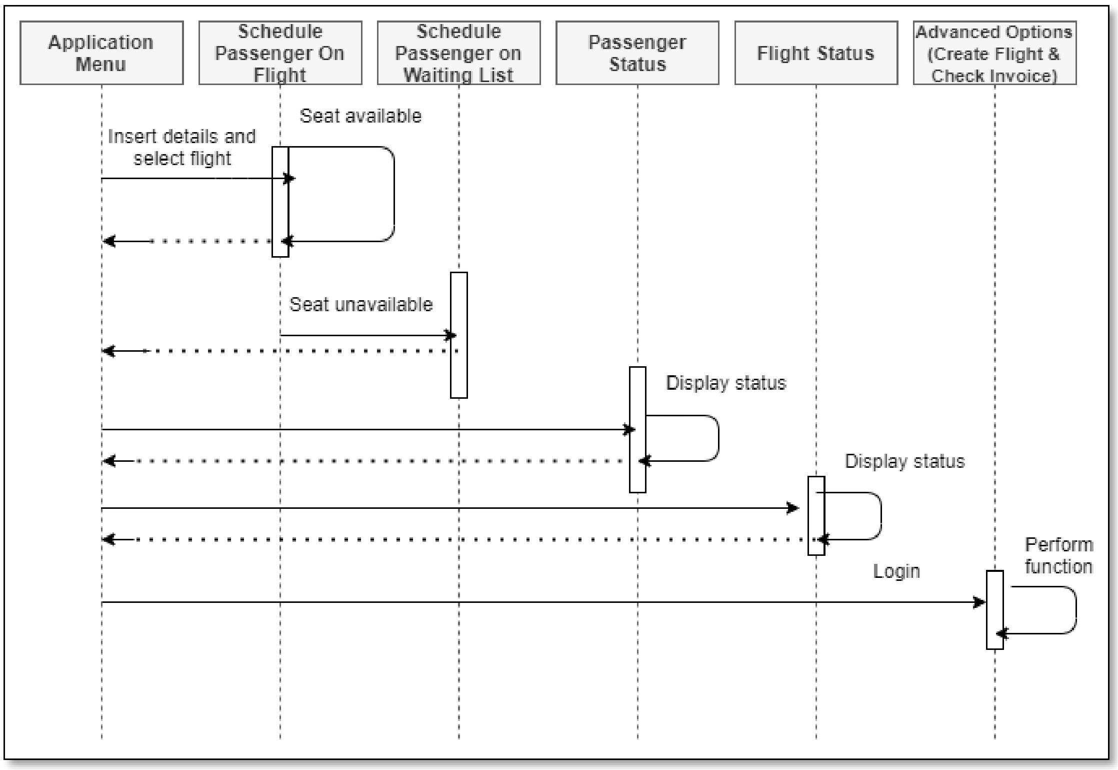

Application Menu
Schedule Passenger On Flight
Schedule Passenger on Waiting List
Passenger Status
Flight Status
Advanced Options (Create Flight & Check Invoice)
Insert details and select flight
Seat available
Seat unavailable
Display status
Display status
Login
Perform function

Figure 3: Sequence Diagram

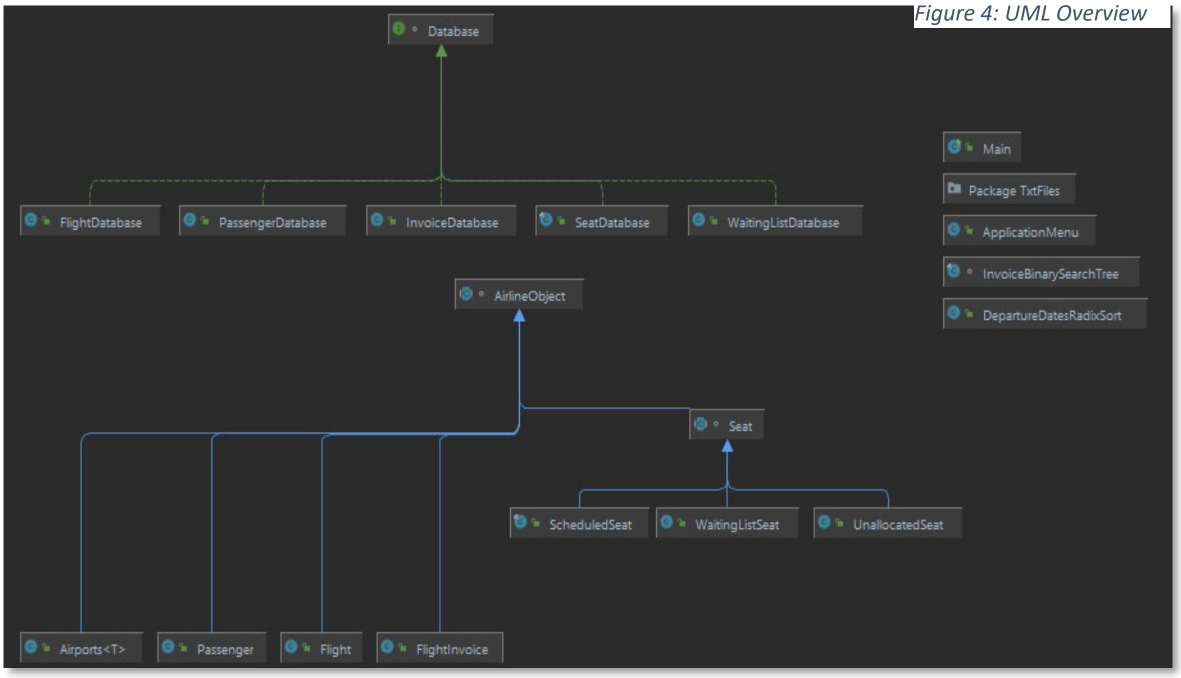
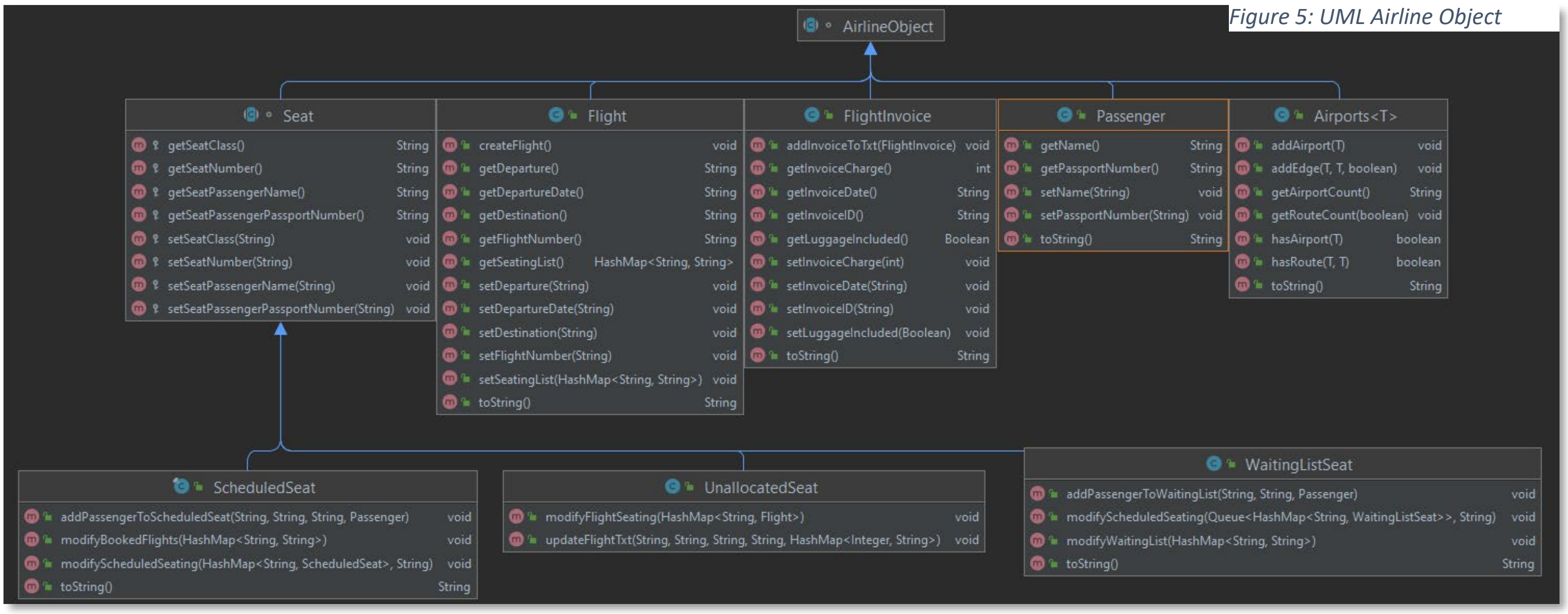
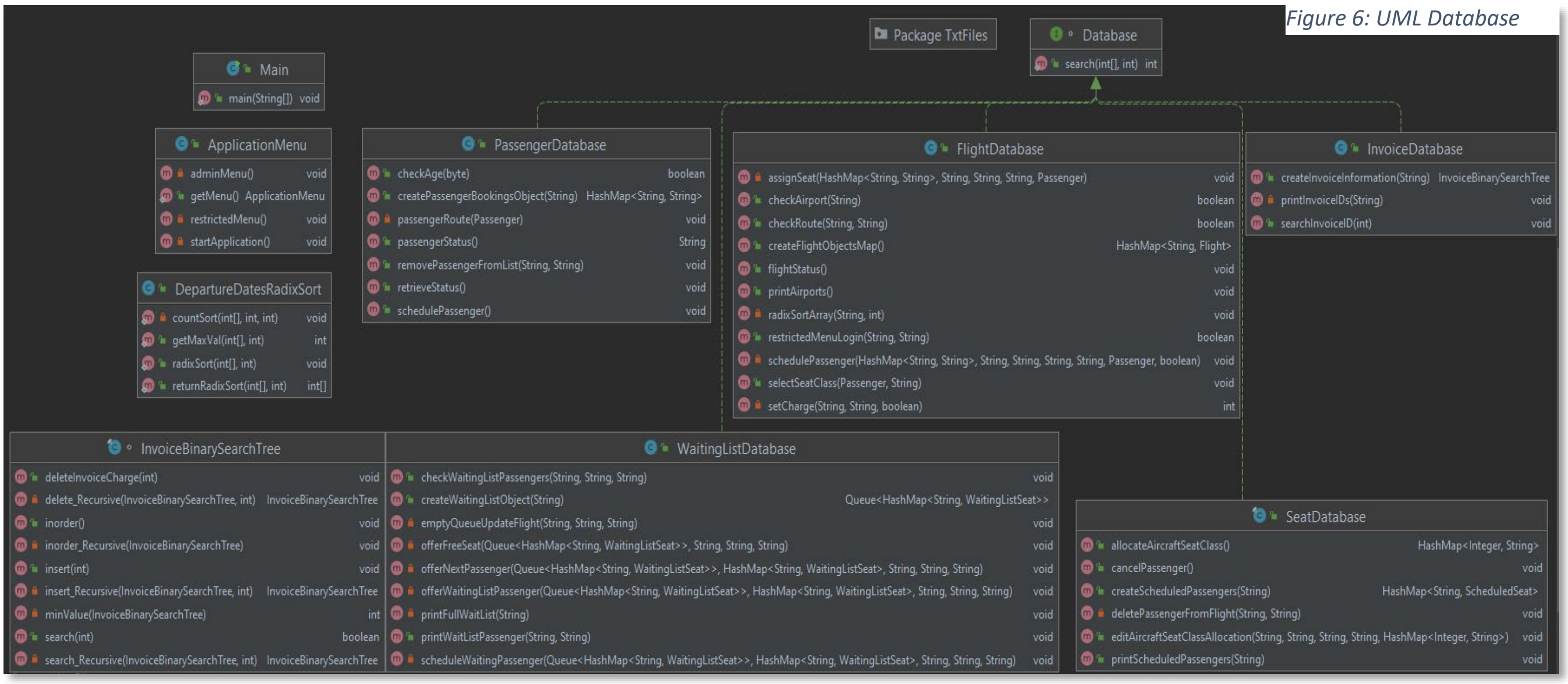


Figure 4: UML Overview



AirlineObject
Seat
getSeatClass() String
getSeatNumber() String
getSeatPassengerName() String
getSeatPassengerPassportNumber() String
setSeatClass(String) void
setSeatNumber(String) void
setSeatPassengerName(String) void
setSeatPassengerPassportNumber(String) void
Flight
createFlight() void
getDeparture() String
getDepartureDate() String
getDestination() String
getFlightNumber() String
getSeatingList() HashMap<String, String>
setDeparture(String) void
setDepartureDate(String) void
setDestination(String) void
setFlightNumber(String) void
setSeatingList(HashMap<String, String>) void
toString() String
FlightInvoice
addInvoiceToTxt(FlightInvoice) void
getInvoiceCharge() int
getInvoiceDate() String
getInvoiceID() String
getLuggageIncluded() Boolean
setInvoiceCharge(int) void
setInvoiceDate(String) void
setInvoiceID(String) void
setLuggageIncluded(Boolean) void
toString() String
Passenger
getName() String
getPassportNumber() String
setName(String) void
setPassportNumber(String) void
toString() String
Airports<T>
addAirport(T) void
addEdge(T, T, boolean) void
getAirportCount() String
getRouteCount(boolean) void
hasAirport(T) boolean
hasRoute(T, T) boolean
toString() String
ScheduledSeat
addPassengerToScheduledSeat(String, String, String, Passenger) void
modifyBookedFlights(HashMap<String, String>) void
modifyScheduledSeating(HashMap<String, ScheduledSeat>, String) void
toString() String
UnallocatedSeat
modifyFlightSeating(HashMap<String, Flight>) void
updateFlightTxt(String, String, String, String, HashMap<Integer, String>) void
WaitingListSeat
addPassengerToWaitingList(String, String, Passenger) void
modifyScheduledSeating(Queue<HashMap<String, WaitingListSeat>>, String) void
modifyWaitingList(HashMap<String, String>) void
toString() String

Figure 5: UML Airline Object



Package TxtFiles
Database
search(int[], int) int
Main
main(String[]) void
ApplicationMenu
adminMenu() void
getMenu() ApplicationMenu
restrictedMenu() void
startApplication() void
DepartureDatesRadixSort
countSort(int[], int, int) void
getMaxVal(int[], int) int
radixSort(int[], int) void
returnRadixSort(int[], int) int[]
PassengerDatabase
checkAge(byte) boolean
createPassengerBookingsObject(String) HashMap<String, String>
passengerRoute(Passenger) void
passengerStatus() String
removePassengerFromList(String, String) void
retrieveStatus() void
schedulePassenger() void
FlightDatabase
assignSeat(HashMap<String, String>, String, String, String, Passenger) void
checkAirport(String) boolean
checkRoute(String, String) boolean
createFlightObjectsMap() HashMap<String, Flight>
flightStatus() void
printAirports() void
radixSortArray(String, int) void
restrictedMenuLogin(String, String) boolean
schedulePassenger(HashMap<String, String>, String, String, String, String, Passenger, boolean) void
selectSeatClass(Passenger, String) void
setCharge(String, String, boolean) int
InvoiceDatabase
createInvoiceInformation(String) InvoiceBinarySearchTree
printInvoiceIDs(String) void
searchInvoiceID(int) void
InvoiceBinarySearchTree
deleteInvoiceCharge(int) void
delete_Recursive(InvoiceBinarySearchTree, int) InvoiceBinarySearchTree
inorder() void
inorder_Recursive(InvoiceBinarySearchTree) void
insert(int) void
insert_Recursive(InvoiceBinarySearchTree, int) InvoiceBinarySearchTree
minValue(InvoiceBinarySearchTree) int
search(int) boolean
search_Recursive(InvoiceBinarySearchTree, int) InvoiceBinarySearchTree
WaitingListDatabase
checkWaitingListPassengers(String, String, String) void
createWaitingListObject(String) Queue<HashMap<String, WaitingListSeat>>
emptyQueueUpdateFlight(String, String, String) void
offerFreeSeat(Queue<HashMap<String, WaitingListSeat>>, String, String, String) void
offerNextPassenger(Queue<HashMap<String, WaitingListSeat>>, HashMap<String, WaitingListSeat>, String, String, String) void
offerWaitingListPassenger(Queue<HashMap<String, WaitingListSeat>>, HashMap<String, WaitingListSeat>, String, String, String) void
printFullWaitList(String) void
printWaitListPassenger(String, String) void
scheduleWaitingPassenger(Queue<HashMap<String, WaitingListSeat>>, HashMap<String, WaitingListSeat>, String, String, String) void
SeatDatabase
allocateAircraftSeatClass() HashMap<Integer, String>
cancelPassenger() void
createScheduledPassengers(String) HashMap<String, ScheduledSeat>
deletePassengerFromFlight(String, String) void
editAircraftSeatClassAllocation(String, String, String, String, HashMap<Integer, String>) void
printScheduledPassengers(String) void

Figure 6: UML Database

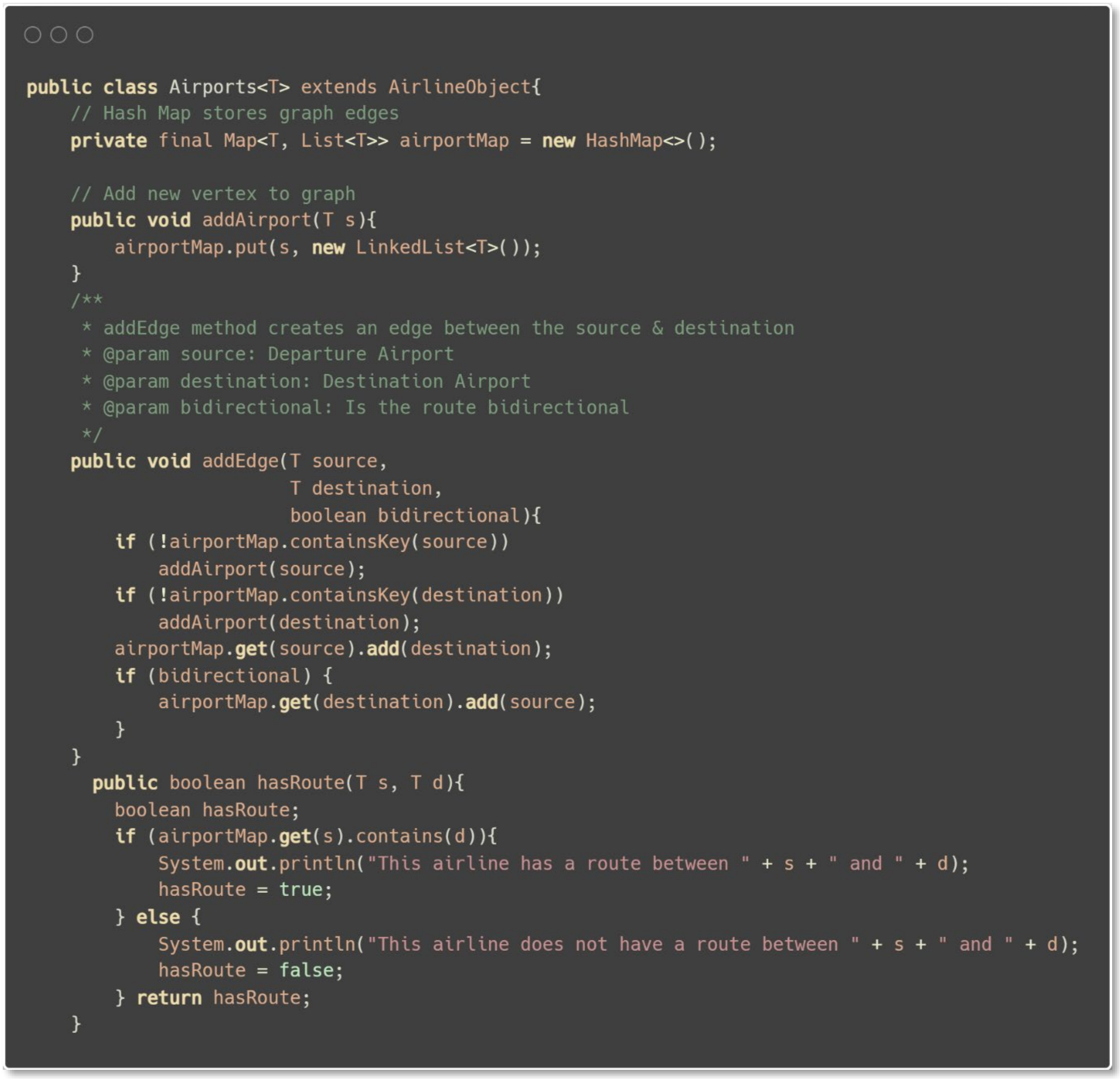
3 Data Structures

3.1 Hash Map

Hash Maps are implemented frequently throughout the program, in most cases to store retrieved data from text files. Much like *Hash Tables*, they enable the mapping of keys and values by storing and retrieving the values from *buckets*. This is determined by a hashing method (Goodrich and Tamassia, 2006, p.372 - 373). However, unlike Hash Tables, Hash Maps are unsynchronised which prevents sharing between multiple threads. This un-synchronisation requires less memory, thus enhances its performance. Furthermore, research by Pereira *et al* (2016, pp.18-19), concludes that Hash Maps are more energy efficient than Hash Tables. Therefore, as there is no requirement for multi-threading in this application, Hash Maps are the chosen implementation. Hash Maps are highly efficient, providing a worst-case time complexity of *O(n)* and average complexity of *O(1),* for searching, insertion and deletion. Furthermore, it's performance is only affected by two factors; *initial capacity* (the number of buckets when created) and *load factor* (the measurement of how full the Hash Map becomes before rehashing; defaulted at 0.75). Once reaching its threshold, the capacity doubles and the contents are re-hashed (Oracle, no date). This results in a cost to both time and space complexity. Therefore, their implementation within this program is to receive an initial capacity of retrieved data, of which is not greatly increased. For example; the mapping of seat classes to their number. This prevents the likelihood of re-hashing, thus ensuring the expected average time-complexity is produced. Other data structures considered for this task include Java's *Doubly Linked List's*, which provides a better worst-case insertion and deletion time complexity of *O(1)*, albeit with a slower average case search time-complexity of; *O(n)* (Downey, 2016, pp.41 - 50). However, the Hash Map is often the preferred choice in this program due to the requirement of pairing data to keys and values.

3.2 Graph

To provide a structure that enables connections between various airports, a *Graph* has been selected. This structure is ideal as it consists of connected *Vertices* and *Edges*, hence can be allocated to airports and their routes. A Graph can be represented using two methods; *Adjacency Matrix,* and *Adjacency Lists.* With the Adjacency Matrix implementation, Vertices are expressed as adjacent to one another. Whereas with an Adjacency List, all Edges of a Vertex are contained within an associated list. To decide on the optimal Graph implementation for this application, the required functions and their corresponding time complexities have been compared. The predominant purposes of this Graph implementation are to hold airport locations, their routes and to enable querying. When adding data to the Graph, both Graph types produce a complexity of constant time; *O(1)* for adding Edges (E). However, the Adjacency List model fairs better than the Matrix for adding Vertices (V); with time complexities of *O(1)* and *O(|V+|E|),* respectively. Furthermore, the Adjacency List demands a lower space complexity. Although, when it comes to querying the data, the Adjacency Matrix performs better than the List implementation; with complexities of *O(1)* and*, O(|V|),* respectively (Singh and Sharma, 2012, pp.179 - 183). Adding Edges and Vertices to the Graph will occur far more frequently than querying for this application, thus the Adjacency List is the chosen implementation. As visualised in *Figure 7,* the Graph is implemented using a Hash Map for the Adjacency List. The Vertices are stored as keys, and all associated Edges are stored as values within a Linked List. All datatypes are applicable within the graph, due the inclusion of *Generics* (Ford and Topp, 2005, pp.147 - 148). This has been included to enable future modifications to provide features such as; Edge weighting to account for distances between airport locations.

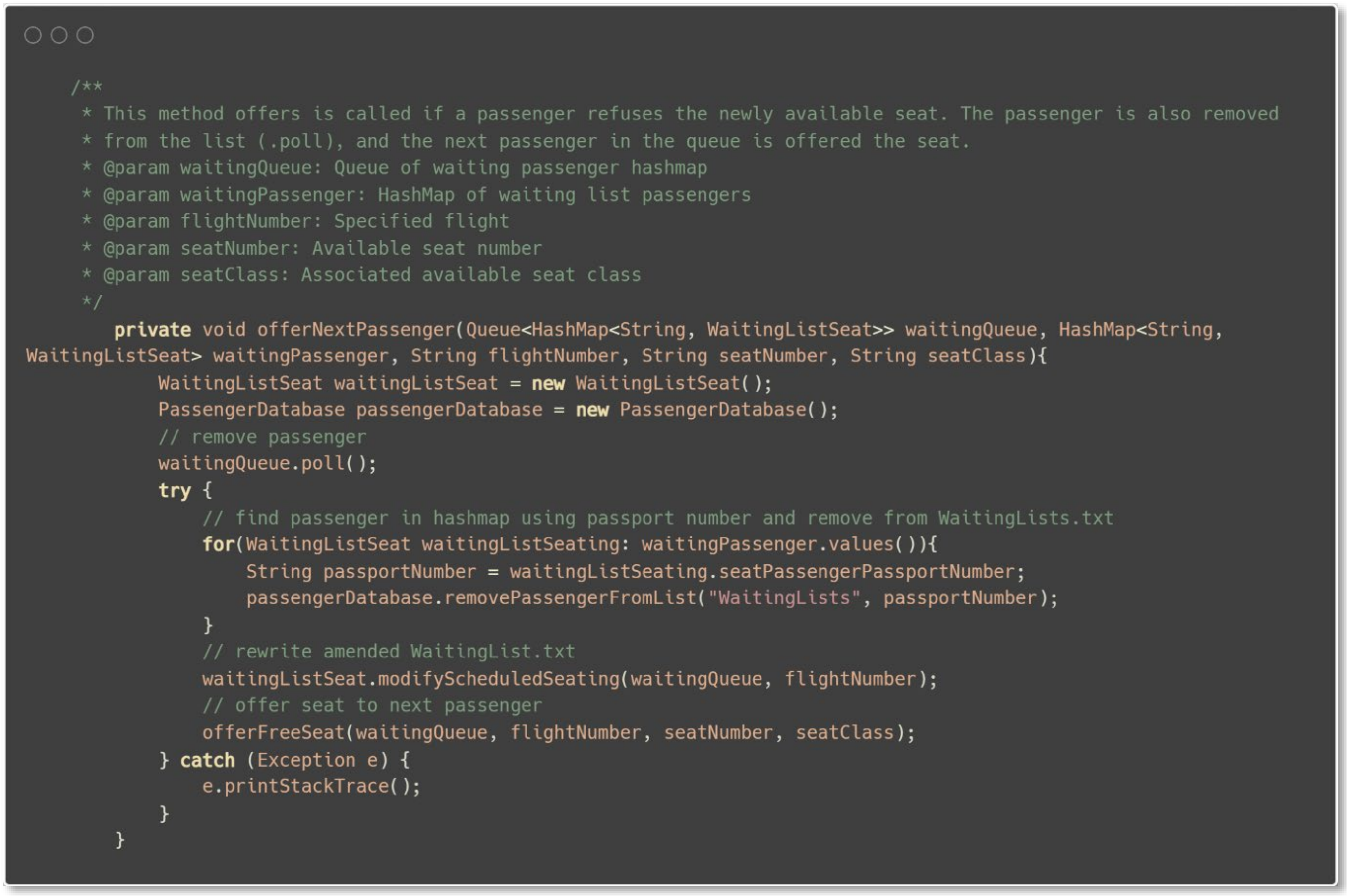


```java
public class Airports<T> extends AirlineObject{
    // Hash Map stores graph edges
    private final Map<T, List<T>> airportMap = new HashMap<>();

    // Add new vertex to graph
    public void addAirport(T s){
        airportMap.put(s, new LinkedList<T>());
    }
    /**
     * addEdge method creates an edge between the source & destination
     * @param source: Departure Airport
     * @param destination: Destination Airport
     * @param bidirectional: Is the route bidirectional
     */
    public void addEdge(T source,
                        T destination,
                        boolean bidirectional){
        if (!airportMap.containsKey(source))
            addAirport(source);
        if (!airportMap.containsKey(destination))
            addAirport(destination);
        airportMap.get(source).add(destination);
        if (bidirectional) {
            airportMap.get(destination).add(source);
        }
    }
      public boolean hasRoute(T s, T d){
        boolean hasRoute;
        if (airportMap.get(s).contains(d)){
            System.out.println("This airline has a route between " + s + " and " + d);
            hasRoute = true;
        } else {
            System.out.println("This airline does not have a route between " + s + " and " + d);
            hasRoute = false;
        } return hasRoute;
    }
```

Figure 7: Adjacency List Graph

3.3 Queue

The *Queue* data structure provides the optimal solution for inserting and removing waiting-list passengers, with an average and worst-case time complexity of *O(1).* This time complexity is more efficient than that of other data structures, such as an Array or Tree, which would incur a complexity of *O(n)* for the same task. The Queue will ensure the first insertion into its structure will remain at the front, utilising the *First In First Out* principle (Ford and Topp, 2005, pp.389 - 426). Therefore, when placing a passenger into the waiting list, the first entry will be the first scheduled onto an available seat. This is not the case for a *Stack* data structure, which places the most recent insertion to the head of the structure, using the *First in Last Out* principle. *Figure 8* displays an example of a Queue containing numerous Hash Maps of Waiting List Seat information. This method removes a passenger that refuses an available seat and then offers it to the subsequent passenger. To ensure an empty waiting list does not return the exception: **`NoSuchElementFoundException`**, the method: **`.remove()`** has been replaced with **`.poll().`** Thus, returning *Null* when the waiting list becomes empty, ensuring the unclaimed seat is returned to the flight's available seats.



```java
    /**
     * This method offers is called if a passenger refuses the newly available seat. The passenger is also removed
     * from the list (.poll), and the next passenger in the queue is offered the seat.
     * @param waitingQueue: Queue of waiting passenger hashmap
     * @param waitingPassenger: HashMap of waiting list passengers
     * @param flightNumber: Specified flight
     * @param seatNumber: Available seat number
     * @param seatClass: Associated available seat class
     */
        private void offerNextPassenger(Queue<HashMap<String, WaitingListSeat>> waitingQueue, HashMap<String,
WaitingListSeat> waitingPassenger, String flightNumber, String seatNumber, String seatClass){
            WaitingListSeat waitingListSeat = new WaitingListSeat();
            PassengerDatabase passengerDatabase = new PassengerDatabase();
            // remove passenger
            waitingQueue.poll();
            try {
                // find passenger in hashmap using passport number and remove from WaitingLists.txt
                for(WaitingListSeat waitingListSeating: waitingPassenger.values()){
                    String passportNumber = waitingListSeating.seatPassengerPassportNumber;
                    passengerDatabase.removePassengerFromList("WaitingLists", passportNumber);
                }
                // rewrite amended WaitingList.txt
                waitingListSeat.modifyScheduledSeating(waitingQueue, flightNumber);
                // offer seat to next passenger
                offerFreeSeat(waitingQueue, flightNumber, seatNumber, seatClass);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
```

Figure 8: Waiting List Queue

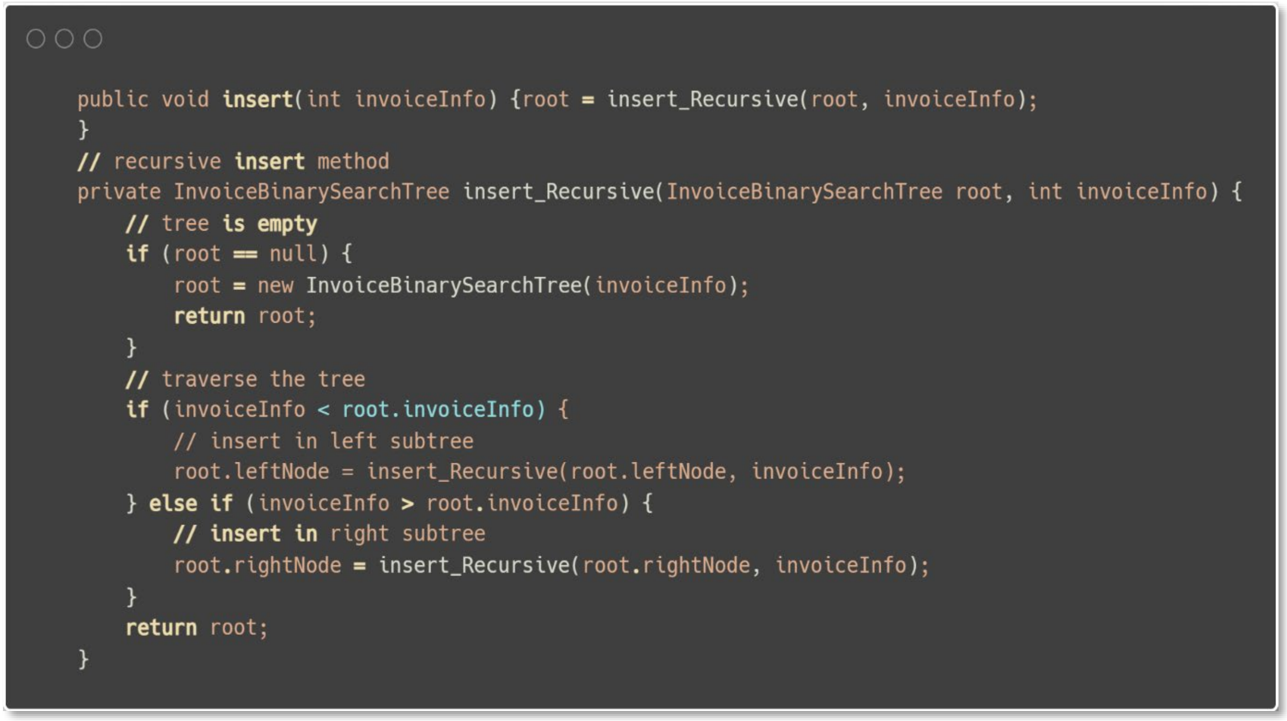
3.4 Binary Search Tree

The **`InvoiceDatabase`** class contains a method for searching invoices. This relies on the Binary Search Tree (BST) data structure, contained within class; **`InvoiceBinarySearchTree`** . This structure has been chosen, as it sorts values upon insertion of the data. As displayed in *Figure 9,* for each node, the node to the left and right will be less than, or more than itself, respectively. Therefore, the structure of the tree will remain organised and can be easily traversed. Pushpa and Vinod (2007, p.237) state that a height-balanced BST is the optimal data structure for storing data, when the complete tree is stored in primary memory. Furthermore, a BST inserts and deletes with a worst-case and average time complexity of *O(n),* and *O(log(n)).* This time complexity is less efficient than that of Linked Lists at; *O(1).* However, due to the Binary Search Tree's inclusion of sorting upon insertion, there is no further requirement to perform later sorting, at additional computational cost. An example of this is viewable in *Table 1.*

Table 1: Table depicting a comparison of Insertion, Sorting and Searching Time Complexities

	Average Total for Insert, Sort and Searching of Elements	**Worst-Case Total for Insert, Sort and Searching of Elements**
Binary Search Tree	*O(log(n)) + O(log(n))*	*O(n) + O(n)*
Linked List + Quicksort + Binary Search	*O(1) + O(n log(n) + O(log(n))*	*O(1) + O(n^2) + O(log(n))*

Once sorted, a *Binary Search* can be performed on the BST with the worst-case and average time-complexities of *O(n)* and *O(log(n)).* This is marginally worse than the standard time complexity of a Binary Search, due to the possibility of the tree becoming unbalanced (Ford and Topp, 2005, pp.530 - 551). Therefore, moving forward this data structure should be replaced with a self-balancing tree such as a *Red Black Tree* (RBT). This tree ensures all roots, external nodes and red-node children are coloured black. Furthermore, all external nodes have the same number of black coloured descendants, minus one, referred to as; *black depth.* These rules ensure the worst-case time complexity for all operations will remain at *O(log(n))* (Goodrich and Tamassia, 2006, pp.463 - 480).



```
public void insert(int invoiceInfo) {root = insert_Recursive(root, invoiceInfo);
}
// recursive insert method
private InvoiceBinarySearchTree insert_Recursive(InvoiceBinarySearchTree root, int invoiceInfo) {
    // tree is empty
    if (root == null) {
        root = new InvoiceBinarySearchTree(invoiceInfo);
        return root;
    }
    // traverse the tree
    if (invoiceInfo < root.invoiceInfo) {
        // insert in left subtree
        root.leftNode = insert_Recursive(root.leftNode, invoiceInfo);
    } else if (invoiceInfo > root.invoiceInfo) {
        // insert in right subtree
        root.rightNode = insert_Recursive(root.rightNode, invoiceInfo);
    }
    return root;
}
```

Figure 9: Binary Search Tree

4 Sorting Algorithms

4.1 Radix Sort

Sorting algorithms, such as; *Selection Sort*, *Insertion Sort, Quicksort* and *Bubble Sort,* all provide a polynomial worst-case time efficiency of *O(n^2)*. A study by Al-Kharabsheh *et al* (2013, pp.120 - 124) assessed the performance of each algorithm, when sorting a dataset of 10,000 elements, producing 2227, 1605, 489, 1133 milliseconds, respectively. Notably, the Quicksort algorithm performs significantly faster in this test, due to its ability to pick a pivot element and divide and conquer the remaining elements. Thus, producing an average time complexity of *O(n log(n)).* Furthermore, even when including the performance of *Mergesort,* which produces a worst-case time complexity of *O(n log(n)),* Quicksort still performs better during testing. When combining the concepts of Mergesort with Insertion Sort to create *Timsort,* the best-case time complexity is improved from *O(n log(n))* to *O(n)*. As such, *Java* and *Python* utilise Timsort as their default sorting algorithm (Goel, *et al,* 2019, pp.1 - 7). Mergesort, Quicksort and Timsort were the initial algorithms considered for the task of sorting flight departure dates. However, due to the departure dates being permanently fixed at eight-digits, the time complexity can be further enhanced with the utilisation of *Radix Sort.* As displayed in *Figure 10,* this algorithm sorts its elements by grouping digits of the same-place value, then by their numerical order. Therefore, this provides an average and worst-case time complexity of *O(nk),* where *n* = amount of values & *k* = length of the value's digits (Goodrich and Tamassia, 2006, pp.516 - 519). As this sorting algorithm is solely utilised on dates, *k* will always equal eight. The space complexity of this algorithm is slower than most at; *O(n+k),* however as previously discussed this is less important for applications with large memory resources. This algorithm is viewable in *Figure 11.*

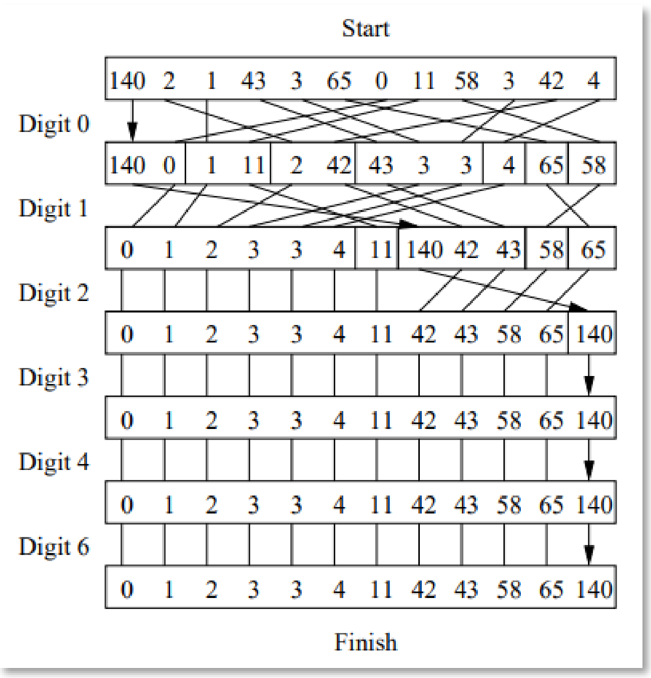
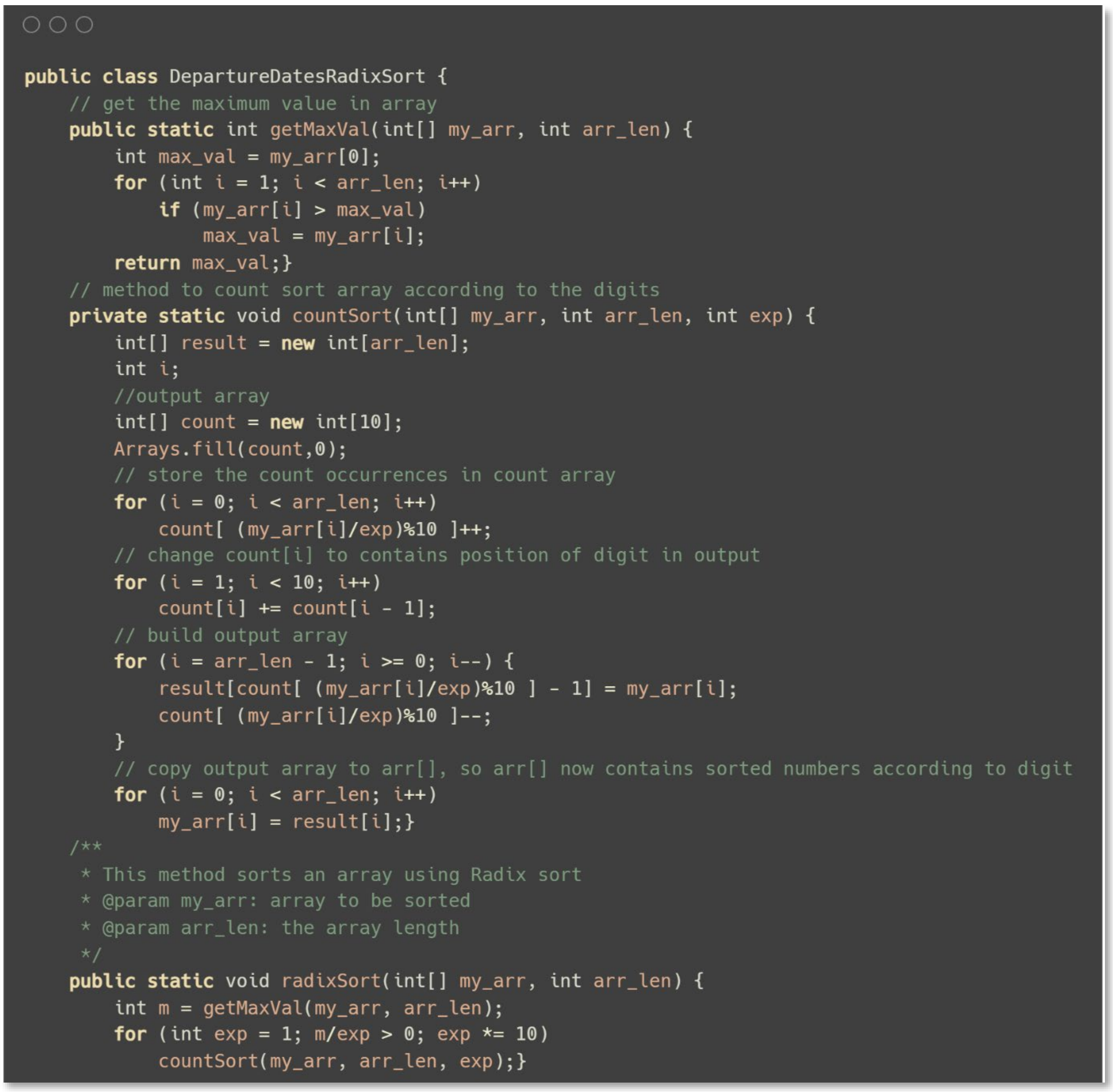


Figure 10: An illustration depicting Radix Sort, (Downey, 2016)



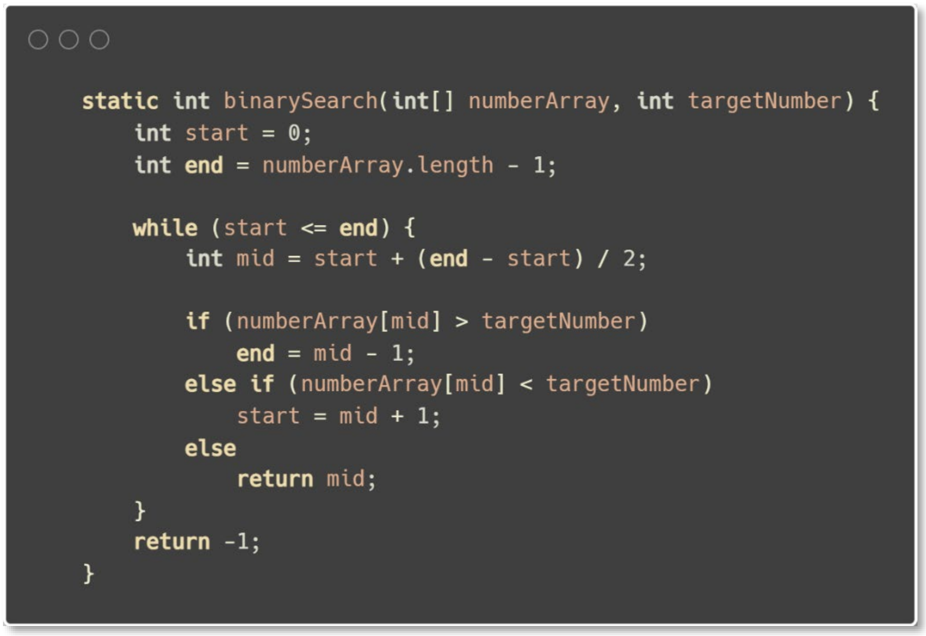
```
public class DepartureDatesRadixSort {
    // get the maximum value in array
    public static int getMaxVal(int[] my_arr, int arr_len) {
        int max_val = my_arr[0];
        for (int i = 1; i < arr_len; i++)
            if (my_arr[i] > max_val)
                max_val = my_arr[i];
        return max_val;}
    // method to count sort array according to the digits
    private static void countSort(int[] my_arr, int arr_len, int exp) {
        int[] result = new int[arr_len];
        int i;
        //output array
        int[] count = new int[10];
        Arrays.fill(count,0);
        // store the count occurrences in count array
        for (i = 0; i < arr_len; i++)
            count[ (my_arr[i]/exp)%10 ]++;
        // change count[i] to contains position of digit in output
        for (i = 1; i < 10; i++)
            count[i] += count[i - 1];
        // build output array
        for (i = arr_len - 1; i >= 0; i--) {
            result[count[ (my_arr[i]/exp)%10 ] - 1] = my_arr[i];
            count[ (my_arr[i]/exp)%10 ]--;
        }
        // copy output array to arr[], so arr[] now contains sorted numbers according to digit
        for (i = 0; i < arr_len; i++)
            my_arr[i] = result[i];}
    /**
     * This method sorts an array using Radix sort
     * @param my_arr: array to be sorted
     * @param arr_len: the array length
     */
    public static void radixSort(int[] my_arr, int arr_len) {
        int m = getMaxVal(my_arr, arr_len);
        for (int exp = 1; m/exp > 0; exp *= 10)
            countSort(my_arr, arr_len, exp);}
```

Figure 11: Radix Sort

5 Searching Algorithms

5.1 Binary Search

To search for values within a sorted array, the method: **`binarySearch`** has been implemented within the **`Database`** Interface*,* displayed in *Figure 12.* This performs effective binary searching of pre-sorted departure dates. The method is Static, enabling Binary Searching to be easily accessible anywhere within the program (Deitel and Deitel, 2012, p.130). Unlike the Binary Search utilised in the Binary Search Tree, the average and worst-case time complexity will always be *O(log(n)).* This is preferable when compared to a linear search of *O(n).* Nonetheless, this could be improved upon with the utilisation of *Interpolation,* ensuring a complexity of; *O(log log (n)).* However, due to the data not being distributed uniformly, this searching method would not achieve this efficiency and is not currently implemented (Mehlhorn and Tsakalidis, 1993, pp.621 - 634).

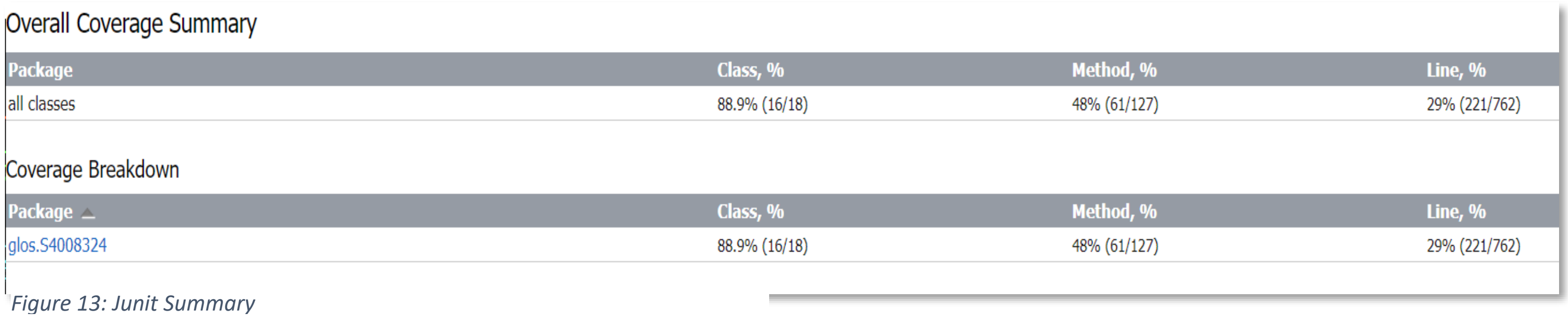


```java
static int binarySearch(int[] numberArray, int targetNumber) {
    int start = 0;
    int end = numberArray.length - 1;

    while (start <= end) {
        int mid = start + (end - start) / 2;

        if (numberArray[mid] > targetNumber)
            end = mid - 1;
        else if (numberArray[mid] < targetNumber)
            start = mid + 1;
        else
            return mid;
    }
    return -1;
}
```

Figure 12: Binary Search

6 Software Tests

Throughout development, incremental testing has been performed both manually and with the *Junit* framework. Unit testing aides in de-bugging and provides improvements in code quality (Appel, 2015, pp.2 - 4). The coverage breakdown in *Figure 13* demonstrates that within the program, 89.9% of classes and 48% of methods are unit-tested. An example of a unit-test for Radix Sort is viewable in *Figure 14*. Furthermore, *Table 2* displays additional manual testing.



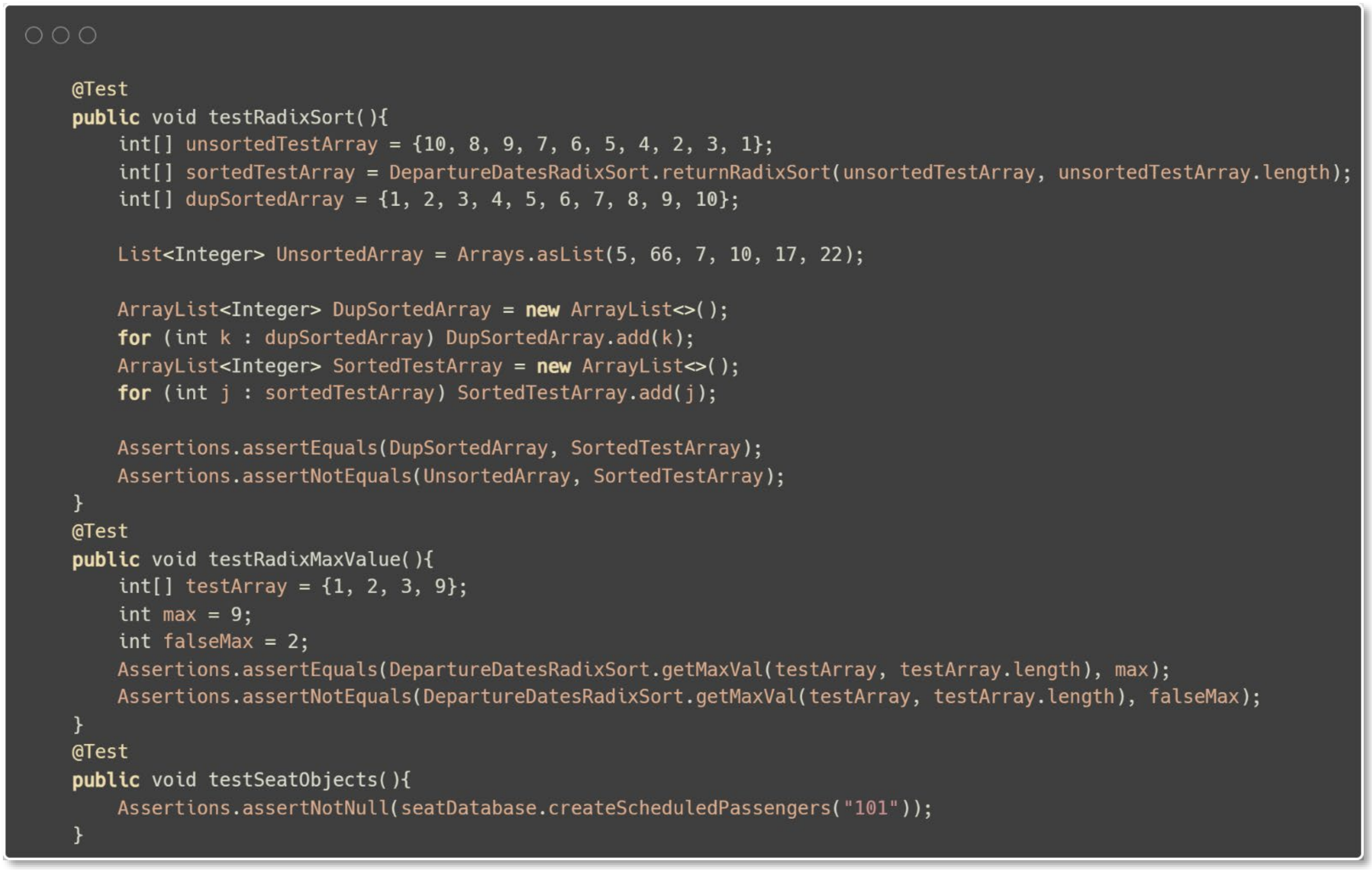
Overall Coverage Summary

Package	Class, %	Method, %	Line, %
all classes	88.9% (16/18)	48% (61/127)	29% (221/762)

Coverage Breakdown

Package ▲	Class, %	Method, %	Line, %
glos.S4008324	88.9% (16/18)	48% (61/127)	29% (221/762)

Figure 13: Junit Summary



```
    @Test
    public void testRadixSort(){
        int[] unsortedTestArray = {10, 8, 9, 7, 6, 5, 4, 2, 3, 1};
        int[] sortedTestArray = DepartureDatesRadixSort.returnRadixSort(unsortedTestArray, unsortedTestArray.length);
        int[] dupSortedArray = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

        List<Integer> UnsortedArray = Arrays.asList(5, 66, 7, 10, 17, 22);

        ArrayList<Integer> DupSortedArray = new ArrayList<>();
        for (int k : dupSortedArray) DupSortedArray.add(k);
        ArrayList<Integer> SortedTestArray = new ArrayList<>();
        for (int j : sortedTestArray) SortedTestArray.add(j);

        Assertions.assertEquals(DupSortedArray, SortedTestArray);
        Assertions.assertNotEquals(UnsortedArray, SortedTestArray);
    }
    @Test
    public void testRadixMaxValue(){
        int[] testArray = {1, 2, 3, 9};
        int max = 9;
        int falseMax = 2;
        Assertions.assertEquals(DepartureDatesRadixSort.getMaxVal(testArray, testArray.length), max);
        Assertions.assertNotEquals(DepartureDatesRadixSort.getMaxVal(testArray, testArray.length), falseMax);
    }
    @Test
    public void testSeatObjects(){
        Assertions.assertNotNull(seatDatabase.createScheduledPassengers("101"));
    }
```

Figure 14: Testing Radix Sort

Table 2: Manual Testing

Class	**Testing Method**	**Result**	**Further Action**
`FlightDatabase`	Retrieve flight information from flights.txt	Seating is returning in String format not HashMap	Need to convert to HashMap format and re-test
`FlightDatabase`	Retrieve flight information from flights.txt – with properties .load method	Seating is placed into a HashMap and set into Flight object	N/A
`ScheduledSeat`	Ensure seating data is updating to text file correctly	Issues with spacing	Re-organised the layout and spacing
`FlightDatabase`	Ensure Radix sorting method passes correct data	Issues with array type.	ArrayList has been converted to array using; **`.stream()mapToInt(I -> i).toArray()`**
`Main`	Conduct a full test of all features and methods	Issues when returning to admin menu and selecting options. Scanner missing lines	Cursor in incorrect position. Use .**`nextLine`** method to set
`Main`	Conduct a full test of all features and methods	Working correctly	N/A

7 Conclusion

Throughout each development iteration, all implemented algorithms and data structures have been reflected upon and explored for further improvements. For instance, algorithms such as Radix Sort have been later introduced to lessen the time complexity. This proved difficult in some instances, such as converting dynamic Array Lists to a static Array required to perform the algorithm. Regardless, it has provided a benefit to the overall program. Such that, throughout all the algorithms and data structure operations utilised, the worst-case time complexity is *O(n).* With the one exception of Radix Sort's time complexity at *O(nk).* However, due to *k* being fixed at the eight-digit date value, this is more efficient than alternative sorting algorithms. The program contains no instances of worst-case complexities of; super linear *O(n log (n))),* polynomial *O(n^2),* exponential *O(2^n),* or factorial growth *O(n!).* Therefore, this application is scalable and can efficiently compute larger quantities of data. Moreover, improvements to reduce time complexities have been explored; such as converting the Binary Search Tree to a Red Black Tree. However, if memory resources were to become reduced, the discussed algorithms should be re-assessed for space-complexity optimisation. For the purpose of this assignment, text files have been implemented for data storage. Moving forward, a more appropriate form of data management will be required, such as a *Relational Database Management System* (RDMS). Furthermore, this application requires processing of a user's personal information; therefore, it is necessary that security measures are implemented. These include the use of password encryption, hashing algorithms and if introducing an RDMS; query parametrisation.

8 References

Al-Kharabsheh, K., *et al* (2013) ‘Review on sorting algorithms: A comparative study’. *International Journal of Computer Science and Security,* 7(3), pp. 120 -126. Available at: (PDF) Review on Sorting Algorithms A Comparative Study (researchgate.net) (Accessed: 4 April 2022).

Appel, F. (2015) *Testing with JUnit: Master high-quality software development driven by unit tests.* Birmingham: Packt Publishing Ltd, pp. 1-5.

Britton, C., and Doake, J. (2005) *A student guide to object-orientated programming.* Oxford: Elsevier Ltd, pp. 7 – 10.

Deitel, P., and Deitel, H. (2012) *Java: How to program.* 9th edn. Essex: Pearson Education Limited, pp. 110 – 140.

Downey, A. (2016) *Think data structures: Algorithms and information retrieval in Java.* Needham: Green Tea Press, pp. 30 – 60.

Ford, W., and Topp, W. (2005) *Data structures with Java.* New Jersey: Pearson Education Inc, pp. 95 – 555.

Goel, J., *et al.* (2019) ‘Comparative analysis and implementation of popular sorting algorithm’. *International Conference on Advances in Engineering Science Management & Technology,* pp. 1 – 7, doi:10.2139/ssrn.3403975

Kimmel, P. (2005) *UML Demystified: A self-teaching guide.* California: McGraw-Hill/Osborne, pp. 17 – 115.

Mehlhorn, K., and Tsakalidis, A. (1993) ‘Dynamic interpolation search’ *Journal of the ACM, 40(3),* pp. 621 – 634, doi:10.1145/174130.174139

Mishra, A., and Garg, D. (2008) ‘Selection of best sorting algorithm’. *International Journal of Intelligent Information Processing,* 2(2), pp. 363 – 369. Available at: (PDF) Selection of Best Sorting Algorithm | Deepak Garg - Academia.edu (Accessed: 2 April 2022).

Oracle (no date) ‘Class HashMap<K,V>’. Available at: HashMap (Java Platform SE 7) (oracle.com) (Accessed: 17 March 2022).

Pereira, R., Couto, M., Cunha, J., Fernandes, P., and Saraiva, J. (2016) ‘The influence of the Java collection framework on overall energy consumption’ *IEEE / ACM 5th International Workshop on Green Sustainable Software,* pp. 15 – 21, doi:10.1109/GREENS.2016.011

Pushpa, S., and Vinod, P. (2007) 'Binary search tree balancing methods: A critical study'. *International Journal of Computer Science and Network Security,* 7(8), pp. 237 – 243. Available at: IJCSNS - International Journal of Computer Science and Network Security (Accessed: 4 April 2022).

Singh, H. and Sharma, R. (2012) 'Role of adjacency matrix & adjacency list in graph theory' *International journal of Computers & technology, 3(1),* pp. 179 – 183, doi:10.24297/ijct.v3i1c.2775

9 Appendix B: User Guide

Logged Out Screen:

- Press enter key to start the application

Application Menu:

- Schedule a passenger by pressing 'S'
  - Enter passenger name, passport number and age
  - Enter departure and destination airport
  - Select seat class
  - Select Seat number
    - If there is no seat available, passenger is given the option to be added to waiting list

- Cancel a passenger by pressing 'C'
  - Enter passenger passport number
  - Enter flight number
    - Passenger will be removed, and any waiting list passengers will be offered the available seat, otherwise the seat will be returned to the flight

- View a passengers status by pressing 'P'
  - Enter passenger passport number
    - Passenger bookings and waiting lists will become viewable

- View a flight status by pressing 'F'
  - Enter flight number
    - Flight information will become available

- For advanced menu press 'A'
  - Enter administrator username and password

        - To create new flight detail, enter ‘S’
            - Enter all departure, destination, and seating requirements
                - Flight information will be created

        - To retrieve an invoice, enter ‘I’
            - Enter invoice number
                - Invoice information will be displayed

- To quit press ‘Q’