**General Program Design for Data Structures Report**

**Project**

**Hangzhou Int.Train System**

**Group Members**

Leighton Madzana

Faith R Kakwere

Gracy Balolage

Ryan Cyril-Zyeele

Primrose Maphosa

**Contents Page**

1. Project Introduction
   1. Abstract
   2. Tools and Software
   3. Implementation Procedure
2. System Design
   1. Skeletal Framework
   2. UML Diagrams
   3. Data Flow Diagrams
3. Testing and Debugging
   1. . Sample outputs and explanations
   2. Bugs and logical errors
4. System analysis
5. Scope of improvement
6. Project Introduction
   1. Abstract

With the breaking of the modern day era, technology has molded every aspect of our livelihood. People all around the globe are looking for faster and more convenient means of getting by with each task of their daily routines. Transport has formed a backbone of the world’s economic, social and political activities, more so trains. Gone are the days when people needed to go to the Train Station to physically book a flight. This can be done online within a couple of minutes.

With this in mind, our team decided to look at what is already in use and see how we could improve its functionality; this led to our choice of a much simpler and more efficient train reservation system. Fully automated both for the client end line and the operational end line ( the ones who control the trains ).

In this project, we focus on train reservations and we implement the booking task by employing the data structures and algorithms that we learnt from the Data Structures and Algorithms course, this semester. These concepts put forth more efficient and thoroughly crafted ways of storing, processing as well as outputting the required information. In this way, our airline reservation system will prove to be a handy tool as far as meeting the modern day demands is concerned. The service allows passengers to view the train schedule and any other related information such as making reservations, cancelling bookings and other such tasks.

TRAIN SYSTEM

RESERVATION

CANCELLATION

UPDATION

* 1. Tools and Software

C++ has native support for object oriented programming, deterministic destruction which is useful for managing resources: this allows system resources to be released when an object is no longer needed, it uses lesser memory as compared to other languages like Java. C++ is also a high-level language allowing direct access to the hardware, making it very important given that we will be manipulating copious amounts of data which requires a substantial amount of computational power.

* 1. Implementation Procedure

Our project focuses on 2 end lines:

1.3.1. Administrative section

The administrative section enables the administrators to manage the system and update the content at regular intervals, the major operations include:

* Create and maintain train schedule.
* View the passenger list.
* View the available seats in the trains..
* Cancel the tickets.
* Updating the train schedule.
  + 1. Passenger section

This portion is mainly for the passenger, allowing him to perform the following functions.

* View all train schedules, timings, fare details and seats availability.
* Book for the tickets.
* View and cancelling of the ticket.

1. System Design

2.1. Skeletal framework

2.1.1. Flow chart

The flow chart below demonstrates the typical process a passenger goes through when making a reservation. The figure furthermore displays a summary of the major functions involved at each level, broken down into a penta-series of subtasks.

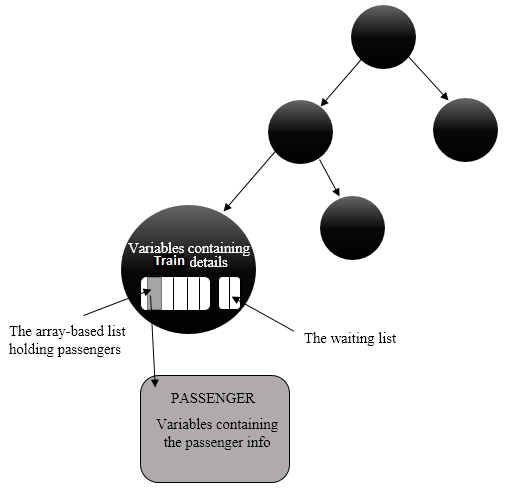
**Passenger’s end line**

**Administrator’s end line**

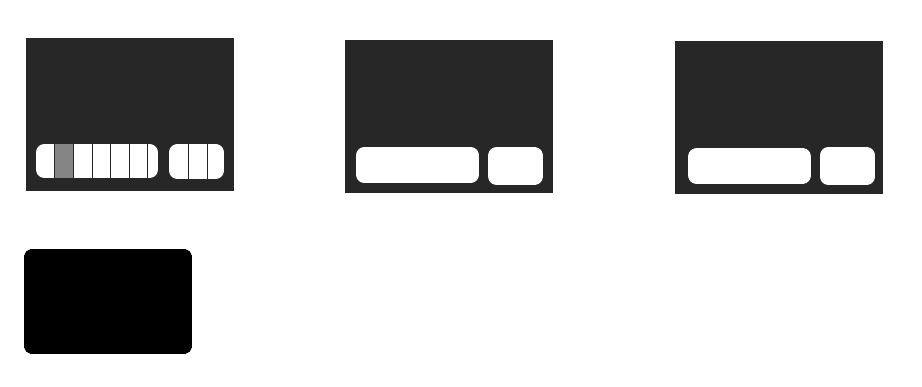
2.1.2. Main Framework Diagram

The operation of our project is solemnly based on this framework (see diagram below). The figure shows an overview of how we envision and implement the application of our Train Reservation System. The links of circles (nodes) resemble the network of nodes which build up a Binary Search tree. The basis of this arrangement is attained from using the train number as the key hence forming this hierarchal structure. Each node is a Train. Within each train, there is a SeatsList object which is executed as an array. Each element of the Seatlist class is a Passenger object which stores the passenger’s information. A WaitingList is also incorporated into the Train. It is also an array, with each cell being a Passenger object.

**Binary Search Tree Implementation**



**Linked List Implementation**



Variables containing train details

The waiting list

The array-based list holding passengers

Variables containing the passenger info

PASSENGER

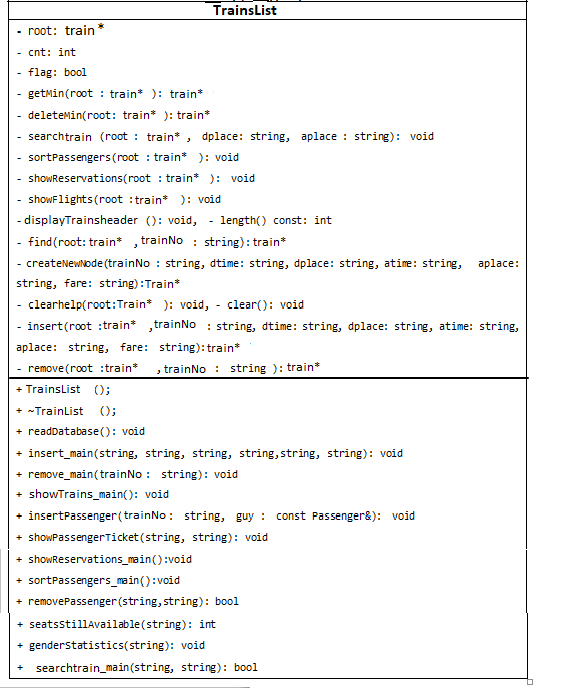
2.1.3. UML diagrams

The following are the UML diagrams for all the classes.

**TrainList Class (Linked-list implementation)**

|  |  |  |
| --- | --- | --- |
| |  | | --- | | **TrainsList** | | * **head: Train|\*** * **tail: Train\*** * **curr: Train\*** * **cnt: int** * **init(): void** * **removeall():void** |   + TrainsList()  + ̴TrainsList()  +readDatabase():void  +append(string, string, string string,string) :void  +searchTrain(dplace : string, aplace: string): bool  +moveToStart(): void  +prep(): void  +next(): void  -length() const: int  -currPos(): int  -insertPassenge(trainNo : string, guy : const Passenger&): void  +showReservation(): void  +showBoardingPass(findpassno : string): bool  -removePassenger(findpassno : string): bool  -seatsAvailable(trainNo : string) : int  -displayTrainsHeader():void  + genderStatistics(trainNo : string):void |

**TrainList Class (Binary Search Tree)**

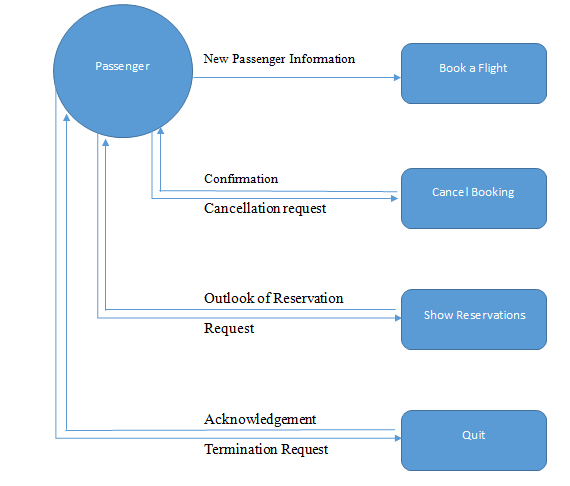
****

from the Train class

2.1.4. Data Flow Diagrams

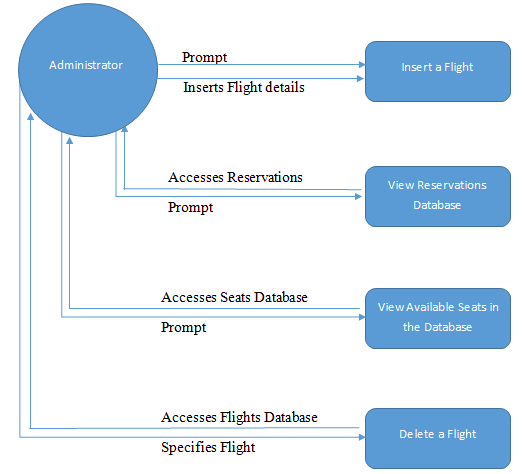
The following figures, illustrate the flow of data for each menu option.

**Passenger’s end line**



Book Train

**Administrator’s end line**



Delete Train

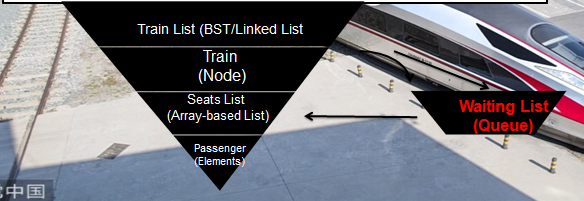
Insert Train

2.2 Functional analysis

2.2.1 Hierarchical Diagram

The diagram below depicts the relationship between the classes contained in our project. The hierarchical shape symbolizes the flow of data or the access of information.

For example, if we are looking for a person in the Train Database, we need to know in which train he is (TrainList), until we get to the bottom to gain access to his information. The TrainList class operates on the top level, it is the class that is the doorway to all the other class.



2.2.2 Main Classes

This diagram shows the 5 main classes within our project.

**TRAINLIST Class**

It is the backbone or root of the whole system. This class is built upon a binary search tree data structure with each node representing a TRAIN class object. The TRAINLIST binary search tree holds all the information of all the trains within the database. It is in the interior of this class, that the admin can insert, delete, search, sort(inorder traversal), or update any of the trains within their database. All the classes that follow have to be accessed through this TRAINLIST doorway.

We also used a linked list as a root of the system. This was just for comparison’s sake. We wanted to analyze its performance against a binary search tree.

**Train Class**

It is an individual Train (node of the binary search tree), which models an airplane. The TRAIN class(or node) contains attributes such as Traincode, cities of travel, maximum number of passengers, etc. This class can only be accessed through the TRAINLIST class. The TRAINclass also interacts with the SeatList class (which will be explained in detail later in this document), by creating an array object. The array symbolizes all the seats within that particular train available for booking. Each train also has a waiting list(queue) to store passengers that might want to book the train after all seats have been booked in the hope that later some seats may be freed after reservation cancelation .

**Seatlist Class**

This is applied as an array-based list since it models the fixed number of seats embodied within aTRAIN. The SeatList class manages the amount of people that can book the train at any given time.

**WaitingList Class**

Given a situation where the SeatList is full, but there might be possibilities of people cancelling their reservation before the departure date. A waiting list is used to store the passengers that might later make up for the possible cancellations. This class implements the array based queue data structure because the size grows and shrinks at any given time, of which in the waiting list, we can only have a certain maximum number of passengers, not exceeding the actual number of passengers in the Seatlist. The WaitingList interacts with the Passenger class as the information about each person has to be showed separately. It also links with the TRAIN class in that; if a situation arises whereby a passenger has to cancel his reservation, we first check if there is any passenger in the waiting list, if yes we dequeue this Passenger object from the WaitingList queue and then overwrite the passenger who cancels his reservation within the SeatList array based list, if no, we proceed with the removal without any replacement.

**Passenger Class**

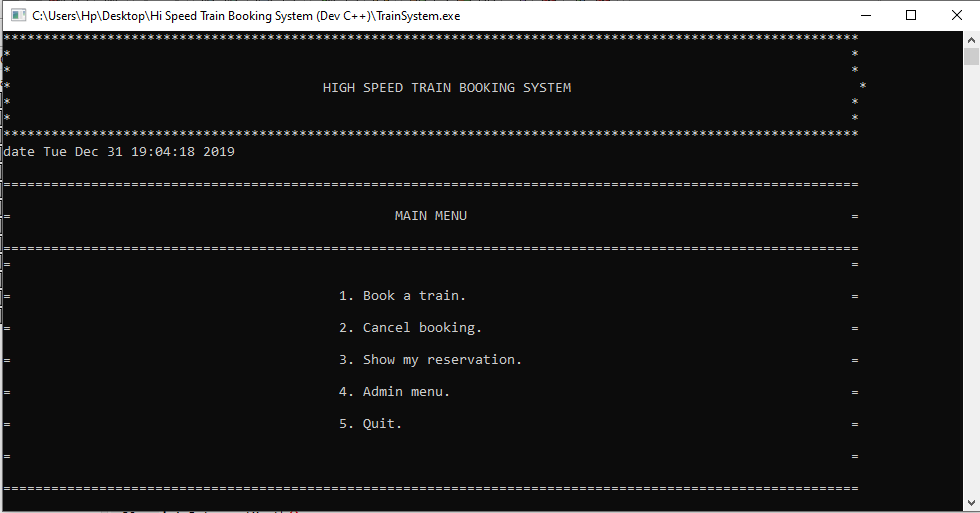
This is a simple person class that holds the information about each individual person within either the SeatList or the WaitingList.

**Date Class**

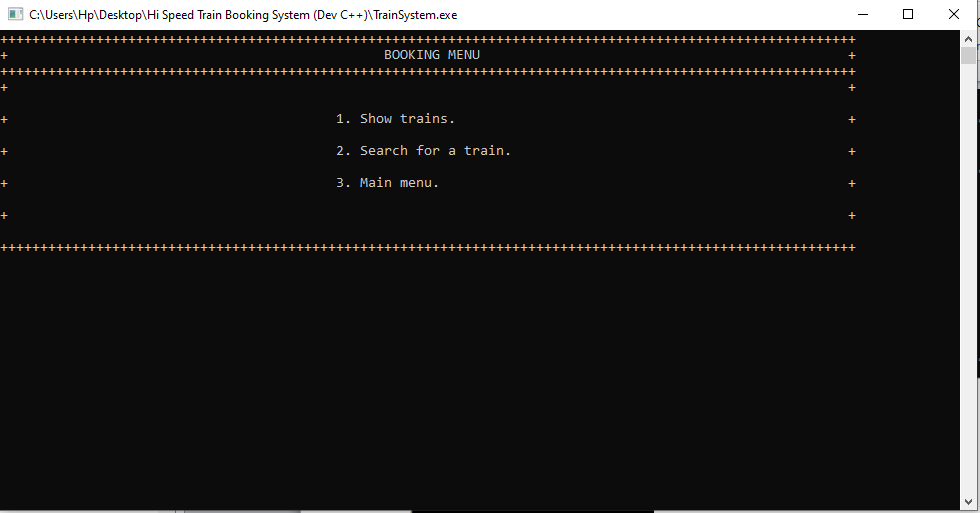
It is just a supplementary class for date input validation from the user.

1. Testing and Debugging
   1. . Sample outputs and explanations

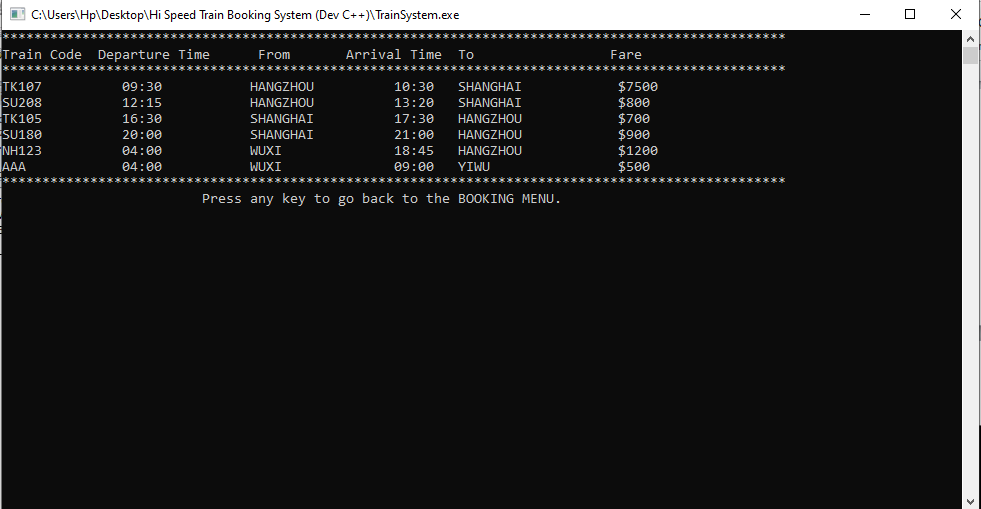
Main menu:



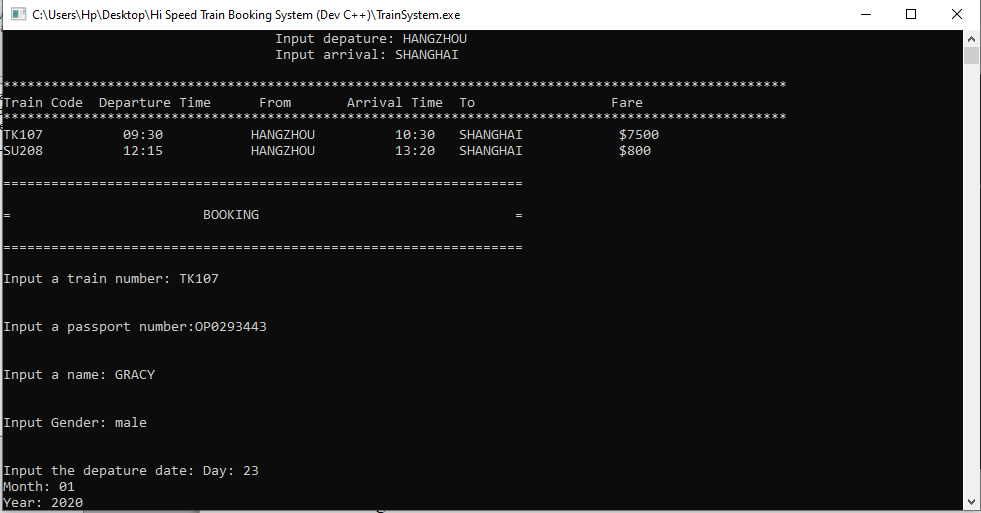
Booking menu:



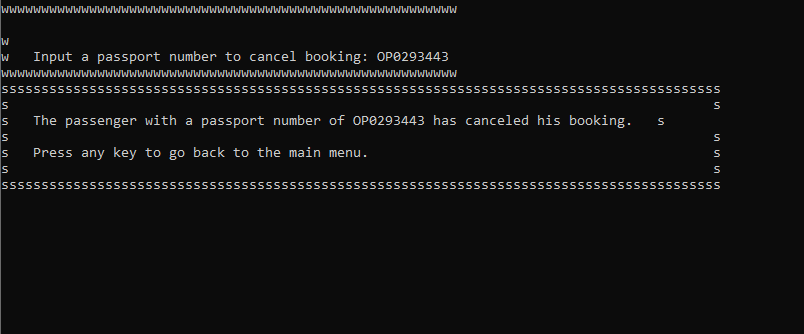
If the user will choose “show Trains”, he’ll get the list of all available Trains:



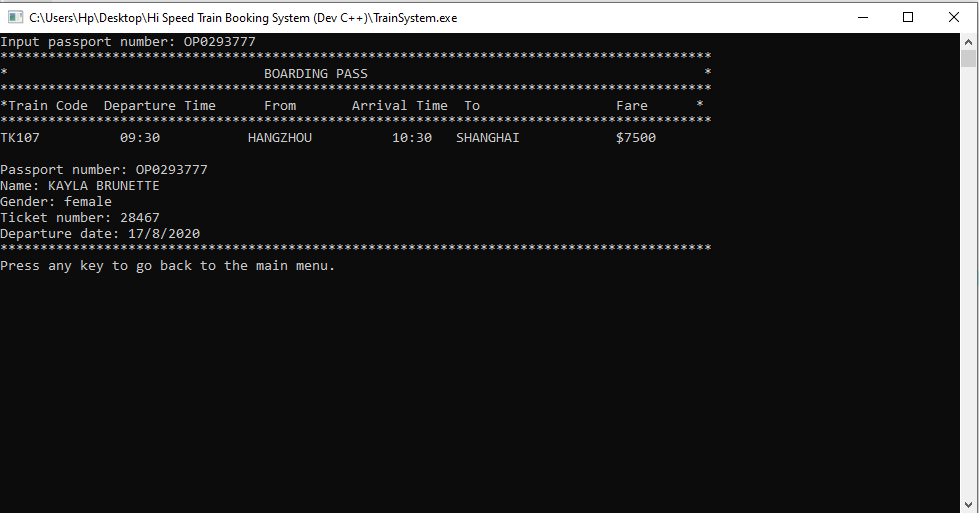
Choosing the second option will lead the user to the booking process.



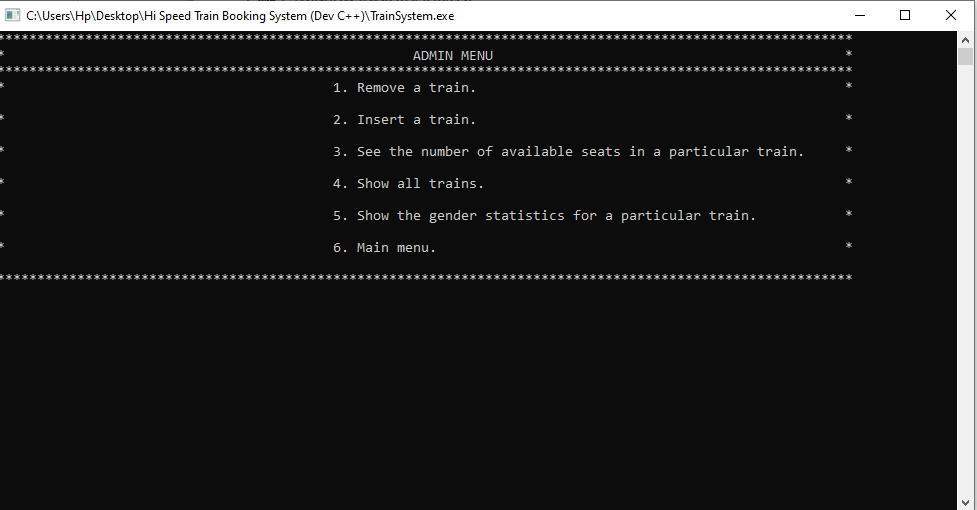
Cancel booking menu:



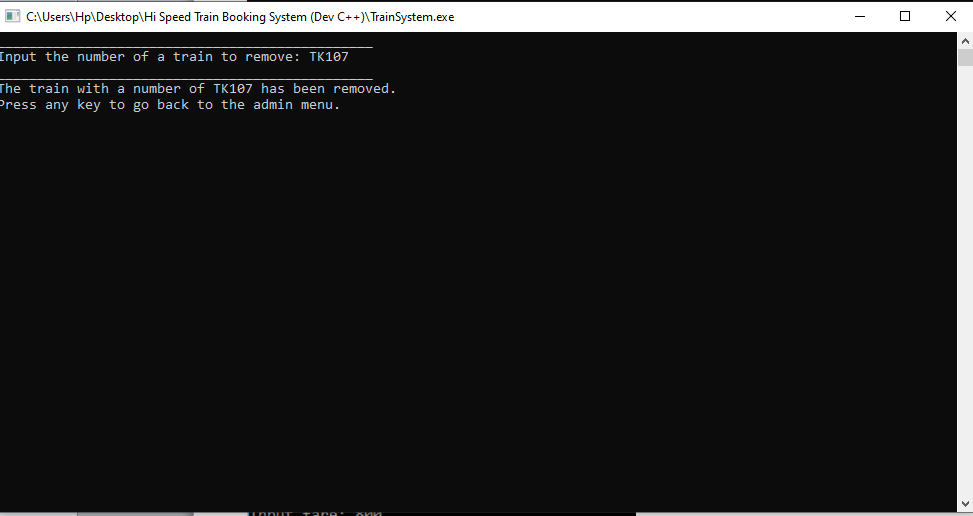
The user can see his reservations by choosing the third option in the main menu and inputting his passport number:



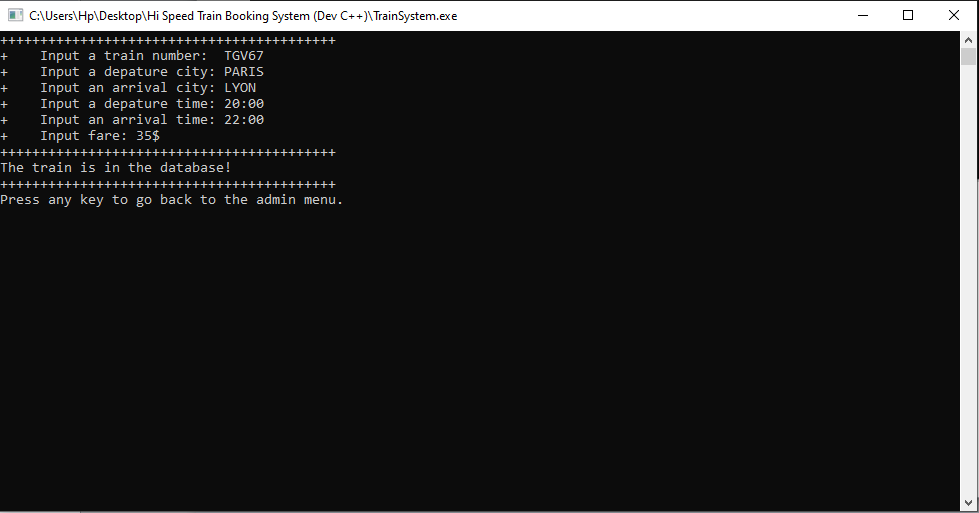
The admin menu:



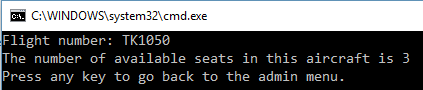
Admin can remove TRAIN from the database:



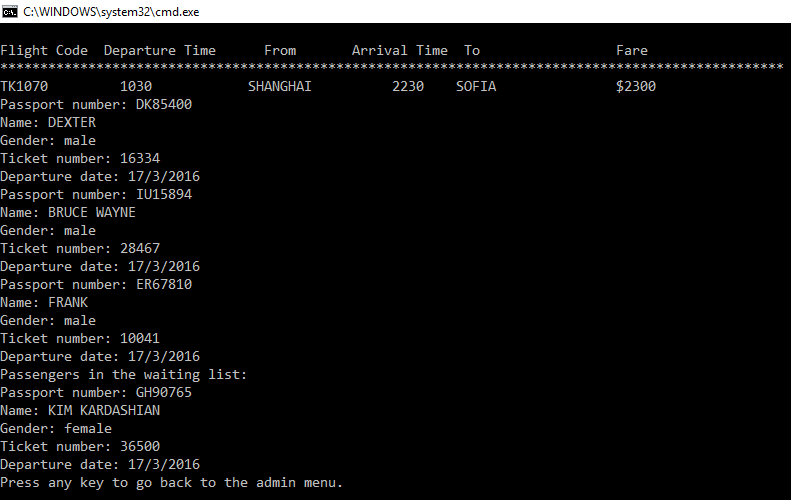
or add them:



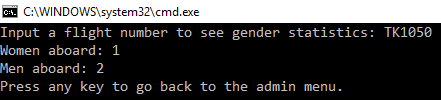
Third option in the admin menu allows to see the number of available seats in the particular TRAIN:



The admin can also see all reservations and people in waiting lists (option 4):



The last option in the admin menu allows to see the gender statistics for a particular train:



* 1. Bugs and logical errors

Logic error: in the beginning, our implementation of the array-based list was completely wrong. For the inputting a passenger into the list, we wanted to set Passenger’s data fields (name, passport number, etc.) with certain values, which is making the whole structure just an array. We understood later, that we should create an object of a Passenger class and insert the object itself into the array-based list.

Bug: The bug surfaced when we tried to remove the last element in a fully filled array-based list; in our implementation after a removal, elements are shifted one step to the left to overwrite the element to be deleted. So after deleting the last element there would be no other elements to fill in that gap so our method was trying to overwrite an out of bounds array index, which was a logic error. The solution was simply to subtract 1 from the internal index of the element being removed

We also encountered logical errors when we were trying to build relationships between the classes, at first we had linked the SeatList class with the WaitingList class, when in actual fact there is no interactions between these two independent class embodied in the TRAIN class. We later resolved it analyzing the core functions of each class.

1. System analysis

Illustrates the data structures tackled within each class, together with the key functions that work on these data structures. It also shows the time complexities that we achieved with each function that we used in our project.

Table 1.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Class Name | Data structure Used | Key Functions  (main functions) | Time Complexites | | |
| Average case | Worst  Case | Best  Case |
| TrainList | Binary Search Tree | Insert | O(d) | O(log n) | O(1) |
| Remove | O(d) | O(log n) | O(1) |
| Search | O(d) | O(log n) | O(1) |
| Sort(inorder) | O(n) | O(n) | O(n) |
| Traversals | O(n) | O(n) | O(n) |
| Linked List | Insert | O(1) | O(1) | O(1) |
| Remove | O(1) | O(1) | O(1) |
| Search | O(n) | O(n) | O(1) |
| Traversal | O(n) | O(n) | O(n) |
| SeatList | Array- based List | Insert | O(n) | O(n) | O(1) |
| remove | O(n) | O(n) | O(1) |
| Search | O(log n) | O(log n) | O(1) |
| Sort | O(nlog n) | O(nlogn) | O(nlogn) |
| Traversals | O(n) | O(n) | O(n) |
| WaitingList | Array-based Queues | enqueue | O(1) | O(1) | O(1) |
| Dequeue | O(1) | O(1) | O(1) |
| traversal | O(n) | O(n) | O(n) |
| Passenger | (Object of SeatList & WaitingList) |  |  |  |  |
| Train | (node of BST) |  |  |  |  |

For the TrainList class we used a binary search tree because each node is a TRAIN hence there is consistent need to be searching the trains. Within the Binary Search tree, the key is the trainnumber, which is then used for searching as well as for sorting when inserting a new TRAIN . A binary search tree is more convenient when searching as compared to other data structures as it has a running time of O(log n) which is much more efficient than O(n) within a Linked List. When outputting sorted information, a binary search tree only requires an inorder traversal costing O(n)in all cases, whereas in other data structures like array-based lists or linked list it will be O(n2) unless much more convenient algorithms are employed.

The array-based List used within the SeatList is also suitable since we will always have a fixed amount of seats within a plane. Accessing any element (i.e. a passenger) can be achieved with constant running time A linked list could have also been used for this task but considering the amount of space it requires as overheads, an array based list seemed more appealing in this regard.

The algorithm engaged for sorting the array-based list is merge sort since so far it has the best running time of O(nlogn). With this approach we had to pay the price for the overhead memory required for a temporary array, but since our line of thought didn’t require an array that is too large for seats within a plane we got by with this algorithm.

We used binary search for searching within the array-based list(sorted) as this yielded a worst (or average) case time complexity of O(logn), and O(1) in the best case. This proved to be more workable, than sequential search with O(n) running time in the worst case.

An array- based queue was the most appropriate data structure when it came to the WaitingList class since the principle of FIFO was very vital. The time and space complexities of this data structure also seemed appealing as they didn’t have any major setbacks.

1. Scope of improvement

Our project is quite efficient, as it uses thoroughly thought through data structures and algorithms to achieve the most efficient time complexities and hence we have achieved the sole purpose of this project. We, however, think that given more time and resources we can try to devise ways to make use of even much better algorithms that will yield time complexities close to the ideal case and even minimum space requirements. To make our system more usable and user friendly, in the future we look forward to add more functions, integrate a GUI, as well as make it accessible online to match up or even surpass the modern day Train Reservation Systems.

For comparison’s sake we had a Linked List implementation as well as a binary search tree. After the execution of our project we noticed that, initially when we had a small amount of information in our database, there wasn’t much of a difference with the running time. A significant change was noted only when the input size increased. The linked list implementation seemed to fall behind. This led to the confirmation that a binary search tree is more efficient for search operations as compared to the linked list. The disadvantage with the binary search tree is its space overheads as each node will require two extra pointers to maintain its structure. But in our project we made sure each node contained required information so as to minimize this space inefficiency. We also had to take note that when inserting the trains into the binary search tree, we randomized the keys values so that they were not sorted at the time of insertion otherwise our binary search tree would end up being unbalanced (like just a linked list) with an inefficient searching time complexities.