

**TRIMESTER 1 2019/2020**

**TDS2111 – Data Structures and Alogrithms**

**High-Speed Train Reservation System**

Tutorial Section: DS1F

Eugene Low Chun Chuan 1161103561

David Ang Huang Qiang 1151105412

Justin John Rajoo 1161103503

FACULTY OF INFORMATION SCIENCE & TECHNOLOGY

MULTIMEDIA UNIVERSITY MALAYSIA

3 OCTOBER 2019

|  |  |  |  |
| --- | --- | --- | --- |
| **Project Evaluation Sheet**  This form is to appended to the final report for submission. | | | |
| **High-Speed Train Reservation System** by **David Ang Huang Qiang** | | | |
|  | **%** | |  |
| 1. **Final Report** | **7** | | **Remark** |
| 1. Introduction / Problem Statements |  | / 1 |  |
| 1. ADT Specification |  | / 2 | Draft |
|  | / 1 | Final |
| 1. Implementation Details  * Justification for each data structure chosen, e.g., why do you consider the stack data structure, but not queue? * Justification for each algorithm chosen, e.g., why do you implement binary search, rather than other searching algorithms? |  | / 2 |  |
| 1. Program Screenshots |  | / 1 |  |
| 1. Implementation Codes (in a CD) |
| B. **Program Source Code** | **11** | |  |
| 1. Basic functionalities, including add, edit, delete, search, sort, display  * No compilation/run-time errors * File handling, usability, user friendliness |  | / 5 |  |
| 1. Data Structures  * Any two data structures learned in TDS2111, including linked-list, stack, queue, hash tables, etc. * Array implementation is acceptable, but there will be no mark given in this section. |  | / 2 |  |
| 1. Good Programming Practices  * Code readability and cleanliness, consistent indentation * Commendation and documentation * Self-explanatory naming convention for variables and function names |  | / 2 |  |
| 1. Extras  * Self-defined header files, separate files for classes, exception handling, aesthetic design * Additional functionalities, etc. |  | / 2 |  |
| 1. **Program Demonstration** | **2** | |  |
| Question and Answer |  | / 2 |  |
| **Total** |  | **/ 20** |  |

# **Introduction**

The High-Speed Train Reservation System was developed to help manage the data for the train station. This reservation system is able to accept keyed in data and will also be able to store the data in the system without any delay. The system will also be able to generate an output when needed to. This system has two modes which are the user mode and admin mode. The first mode which is the user mode will allow users to book ticket, search ticket, cancel ticket and modify booking through the train reservation system. The second mode which is the admin mode will allow the admin to sort records in a text file according to train id.

# **Problem Statements**

This High-Speed Train Reservation System was created to replace the old system that the train station was using. The previous old system had no specific way to keep and store records of ticket purchases.

In the previous old system, users that wanted to purchase a train ticket would have to queue up in front of a ticket counter. This proofed to be inefficient because users would have to queue up for a long time. Users would also sometimes not know how many tickets are available before queueing up to purchase a ticket. This would sometimes lead to tickets being sold out and them wasting their time queuing up for no reason.

Another problem with the old system is that users were not able to modify their ticket. After making their purchase at the bus terminal ticket counter users were not able to change certain details about their ticket. For example, if the user had provided the wrong departure time, they were not allowed to change the time. They would have to buy a new ticket in order to fix the departure time.

Finally, the old system also made it hard to sort file. In the old system sorting files had to be done manually. This would not only take a lot of time but sometimes it would also cause the files to be sorted wrongly due to human error.

# **ADT Specification**

**Data:**

First\_Name : customer’s first name

Last\_Name : customer’s last name

IC : customer’s identity card number

Departure : train depart location

Destination : location that the train reached

Date : date for the customer to travel

Month : month for the customer to travel

Year : year for the customer to travel

Time : time for the customer to travel

Seat\_No : the seat that the customer select in train

Notes : additional notes

**Opeartions:**

Add () : add customer’s details into file

modify () : modify ticket details

deleteticket () : ticket cancellation

search () : search for booking tickets

sort () : display train accordingly by identity number

display () : display available seats

# **Implementation Details**

**Using Linked-List to Display and Book Seat**

The reason being user need constant-time insertions/deletions from the list. User want to be able to book ticket in the middle of the list (such as a priority queue).

If we implement array-based in our system, we have to manually declare the size of the array every single time we add or decrease the number of seats. If we declared a large size of array, the unused storage will be wasted.

**Using Binary Search Tree to Search Customer Record**

The answer is that trees can be traversed in their natural order very efficiently. If you need to visit every element of a dictionary in alphabetical order, a tree can support this directly, where a hash table cannot.

Binary search trees are collections that can efficiently maintain a dynamically changing dataset in sorted order, for some "sortable" type.\*

Having a sorted array is useful for many tasks because it enables binary search to be used, to efficiently locate elements. The problem with a sorted array is that elements can't be inserted and removed efficiently.

The binary search tree is a different way of structuring data so that it can still be binary searched (or a very similar procedure can be used), but it's easier to add and remove elements. Instead of just storing the elements contiguously from least to greatest, the data is maintained in many separate chunks, making adding an element a matter of adding a new chunk of memory and linking it to existing chunks.

Binary search trees support everything you can get from a sorted array: efficient search, in-order forward/backwards traversal from any given element, predecessor /successor element search, and max /min queries, with the added benefit of efficient inserts and deletes. With a self-balancing binary search tree (BST), all of the above run in logarithmic time.

Firstly, we open the file .Next, fill and insert the binary tree and close the file. When a customer enters a key word for this instance their name, it will display the records of the customer.

**Using File Handling to add, delete and modify customers’ ticket.**

When customer input their details, a file is created individually and written into it. The file name is saved as their identity card (IC) number. To modify customer’s ticket, they would have to input their IC. Their file is removed and new file is created with the new records written in it. To delete their ticket, they would insert IC and the file will be removed.

# **Program Screenshots**



















