BINUS INTERNATIONAL

DATA STRUCTURE

OBJECT ORIENTED PROGRAMMING

FINAL PROJECT REPORT

DIMSUM (Digital Interactive MRT Schedule Update Manager)

Athallah Raja Mustafa - 2802537552

Barri Nur Pratama - 2802501142

Jason Franto Fong - 2802557781

Supervisors:

Dr. Maria Seraphina Astriani, S.Kom., M.T.I.

Jude Joseph Lamug Martinez MCS

TABLE OF CONTENTS

CHAPTER 1

1.1. Background

We think that the current MRT system in Jakarta is full of potential but must be far more efficient. For some of the key issues for commuters, these include uncertainty of train arrival times and waiting time for this, which can lead to unnecessary anxiety, inefficiencies in trips, and less reliance on public transportation. By implementing our proposed real-time MRT arrival system, we will be able to address these concerns directly. With present, real-time information on train arrival and departure, commuters will be more able to schedule their journeys with precision and confidence.

This improvement will also substantially reduce waiting time, allowing commuters to get to stations precisely on time for their trains, rather than showing up too early or waiting in uncertainty. Such predictability in public transportation timetabling can significantly lower travel-related anxiety, enhance the satisfaction of passengers, and, most significantly, boost overall usage of the MRT system. A system that accommodates a smoother, more reliable commuting experience can convince Jakarta residents to switch from private to public transport, which could in turn make the city less crowded and with a smaller carbon footprint (PT MRT Jakarta, n.d.).

It cannot be overstressed how crucial 'real-time' management of public transport is. It performs a critical role of enhancing not just the transit service reliability and efficiency, but also the comfort and trust of the passenger. Through offering accurate and informative transit information, travelers can make better travel decisions. This sense of empowerment can lead to higher commuter satisfaction, reduction in travel anxiety, and higher levels of trust in the public transportation system.

One such example is the implementation of the "OneBusAway" system in Chicago. In this experiment, the effect of providing real-time bus arrival details to the passengers through a mobile application was examined. The result of this study was extremely positive. Many commuters articulated that they were much more satisfied with their daily commute. Appreciation of reduced waiting time brought improved mood and travel experience, and in a few cases even the convenience of trip planning led to higher use of public transport. People believed that commutation became more convenient and less stressful if they had timely information at handy disposal.

Our vision is to replicate and build upon the success of such systems as OneBusAway by developing a similar real-time information system exclusively for the MRT system in Jakarta. We believe that this technological innovation can change the way one looks at and utilizes public transport in the city. Gone will be the days when the MRT will be perceived as an alternative or less convenient option, but as a reliable and efficient option for daily commutes. With our real-time arrival system, we're not only streamlining operations but also providing a nicer and less stressful ride experience. In the long run, we envision a shift in commuter behavior, a switch where there are more commuters who want to make the MRT their preferred mode of transportation because it is convenient and reliable. This would not only be good for individual commuters but also for the entire Jakarta urban mobility ecosystem.

In short, optimizing the MRT system through real-time tracking is not just a tech upgrade, it's the way towards Jakarta's streamlined public transportation and meeting the growing demands of its metropolitan population. Cutting wait times, removing frustration, and providing correct information enable us to increase MRT ridership and make a more eco-friendly, accessible, and commuter-friendly city.

1.2. Problem Description

Mass Rapid Transit or MRT has been a lot of help for people to travel, and do their duties. People can travel long distances within minutes with the MRT. MRT is one form of public transportation, so it will also need a scheduling system. scheduling in MRT needs to be precise so people can determine when the train will arrive furthermore preventing tardiness in their activities. An MRT scheduling system is crucial for ensuring efficient train operations, minimizing delays, and optimizing passenger flow. The system must handle:

1. Train Timetabling & Scheduling
   - Optimizing train frequency to match peak and off-peak demand.
   - Adjusting schedules dynamically in case of delays or disruptions.
   - Coordinating arrivals and departures across multiple stations.
2. Real-time Monitoring & Updates
   - Handling train delays, breakdowns, or emergencies.
   - Communicating schedule changes to passengers via digital displays or apps.
3. User-Friendly Interfaces
   - Providing commuters with an easy-to-use platform for checking schedules.
   - Allowing staff to modify schedules as needed.

A simple mistake in designing the MRT schedule might lead to fatal problems. It may cause wrong departure time for trains, tardiness in people's activities, or worse, accidents. Here is the application design for our MRT scheduling system, complete with user interaction. The table outlines various system functions and specifies whether the passengers and managers have access to each feature.

FEATURE	PASSENGER	MANAGER
VIEW FULL SCHEDULE	YES	YES
VIEW EACH STATION'S SCHEDULE	YES	YES
FIND NEXT ARRIVING TRAIN	YES	NO
ADD TRAIN SCHEDULE(S)	NO	YES

RESCHEDULE TRAIN(S)	**NO**	**YES**
DELAY TRAIN(S)	**NO**	**YES**
CANCEL TRAIN(S)	**NO**	**YES**

1.3. Solution

To address the problem, we make use of three different data structures, each chosen based on its efficiency in handling different aspects of MRT scheduling. The three data structures we choose are: TreeMap, HashMap, and PriorityQueue.

Data Structures Used

1. **TreeMap**

Treemap is a data structure (often represented by a Red-Black Tree) that has a key-value pair. This could be beneficial in the case of:

- The key can represent the time.
- The value can represent the train name/ID

By doing this, multiple trains can traverse through the system, meaning that values can for example be: Train 001, but point to multiple unique keys (which are the time).
Some advantages of using the TreeMap data structure in implementing the functions of the MRT scheduling system are as such:

- A TreeMap will maintain the order of the train schedules, whether we are adding or searching.
- A key lookup in TreeMap is expected to have a time complexity of O(log n), which is very efficient (Downey, 2017).
- It doesn't use a hash function like HashMap, and avoids the difficulty in designing or choosing a good one (Downey, 2017).
- By preserving the order, TreeMap eliminates the need of sorting which can hinder the performance like other types of data structures.

The efficiency of a TreeMap is further enhanced by its use of Red-Black Tree balancing. As explained in Data Structures and Algorithms in Java, "Adding red-black balancing to a binary tree has only a small negative effect on average performance, and avoids worst-case performance when the data is already sorted" (Lafore, 2017).

In conclusion, A TreeMap is an optimal data structure for handling train scheduling due to its automatic sorting, efficient retrieval, and ability to handle dynamic updates efficiently. While HashMap provides fast lookups, it does not maintain order, making it less suitable for time-based scheduling. Meanwhile, PriorityQueue is useful for processing the next train efficiently but will not be as efficient in viewing full schedules or modifying specific data. Therefore, a TreeMap is a very promising data structure that can be used in creating a program that can manage train operations efficiently.

2. HashMap

“Hashmaps are an essential part of many algorithms as they allow associative retrieval of n elements in average constant time and have a space complexity of O(n)” (Brehm, 2019). The HashMap can be implemented as such:
Scenario: Retrieve the Schedule for "Train A"

- Use the Train-ID "Train A" as the key to retrieve the list of station-time pairs.
- Value: [{”Bundaran HI“, "5:12"}, {"Senayan", "5:20"}, {"Istora", "6:12"}]
- Display the schedule:
- "Train A" arrives at:
- "Bundaran HI" at 5:12
- "Senayan" at 5:20
- "Istora" at 6:12

In this implementation, the Value or the bucket stores a type of list that contains the station and time. This is to find at which station the train is at at a specific time.
Advantages in using HashMap are:

- Insertion, searches, and deletions on average have the time complexity of O(1).
- Train schedules can be retrieved immediately from the corresponding bucket without scanning all entries.
- Efficient retrieval allows quick station-based filtering, which is good for the function of view station schedule.
- We can implement an arraylist for the buckets to allow easy appending for new station-time pairs.

In conclusion, the use of HashMap for core operations like retrieving, adding, modifying, or removing train schedules outperforms TreeMap and PriorityQueue due to its constant-time efficiency and direct key-based access. Although there might be some conditions where the HashMap can be slower due to hash collisions from inefficient designs.

3. Priority Queue

A Priority Queue is an essential data structure for train scheduling. It helps efficiently handle the order of train arrivals, departure, delays based on priority not just time. According to Koffman and Wolfgang (2021), "A priority queue is a data structure in which only the highest priority item is accessible. During insertion, the position of an item in the queue is based on its priority relative to the priorities of other items in the queue." This characteristic makes it particularly useful for handling train departures, arrivals, and delays.

Advantages:

- Handling train delays dynamically: When a train is delayed, a priority queue can dynamically reorder the schedule based on urgency and availability; such as comparing the amount of delays (the larger, the more priority).
- The priority queue will have fast access to the next data in the queue (this can be the next arriving train itself), this means the time complexity is O(1).

Example Use Case:

- Insert delay-related tasks into the PriorityQueue:
- {"Train A", "Delayed by 15 minutes", "ST-001", Priority: 2}
- {"Train B", "Delayed by 30 minutes", "ST-002", Priority: 1}
- {"Train C", "Delayed by 5 minutes", "ST-003", Priority: 3}
- Process tasks in order of priority:
- First: Handle the 30-minute delay of Train B at "ST-002".
- Next: Process the 15-minute delay of Train A at "ST-001".
- Finally: Process the 5-minute delay of Train C at "ST-003".

In conclusion, A PriorityQueue is a great choice for managing train schedules, particularly for functions that require prioritization based on departure times. Unlike a HashMap, which provides fast lookups but does not maintain order, a PriorityQueue ensures that the next departing train is always accessible. “The hash table is inappropriate as there is no convenient way to find the minimum element, so hashing merely adds complications, and does not improve performance over, say, a linked list" (Aho et al., 1983). This means that while a HashMap is great at searching specific trains, it is not ideal for efficiently retrieving the next train arriving without scanning all elements. By using a PriorityQueue, the system can efficiently manage departures, reschedules, and delays.

4. Research Procedure

1. Define Functions & Requirements

Each function in the train scheduling application, as outlined in the design phase, will be carefully assessed to determine its specific requirements and the most suitable data structure for optimal performance.

2. Implement Each Function Using Different Data Structures

To evaluate efficiency, each function will be implemented using TreeMap, HashMap, and PriorityQueue. This will allow for a detailed analysis of time and space complexity on all the different functions.

3. Testing

- **Input Types:** Train schedules, train codes, station names, and train time delays.
- **Dataset Sizes:** Small (10), Medium (100), Large (1000).
- **Operations Tested:** Insertion, search, retrieval, update, and deletion.
- **Performance Metrics:** Execution time (ms) and memory usage (bytes).

4. Comparing Results & Determine the Best Combination

Based on the test results, the most efficient data structures will be strategically combined. The final system will incorporate a combination of the three data structures chosen, leveraging HashMap for fast lookups, TreeMap for maintaining an ordered schedule, and priority queues for handling prioritized tasks such as delays, this will ensure optimal efficiency for all the different functions of the system.

CHAPTER 2

2.1. Project Specification

DIMSUM is an application that focuses on data structures that seeks to enhance Jakarta's MRT system via real-time updates of train arrivals. The system provides minimal waiting times for commuters, predictable travel, and encourages heightened utilization of public transport.

Objectives:

- Create a real-time system of MRT arrival monitoring.
- Enhance commuter experience by removing uncertainty and waiting time.
- Increase utilization of public transport by enhancing reliability.

Key Features:

- Real-Time Data Processing: Utilizes efficient data structures to handle real-time MRT arrival data.
- Predictive Analytics: Analyzes patterns to predict train arrivals and expected delays.

Technical Specifications:

- Programming Language: Java
- Data Structures Used: Hashmap, Treemap, Priority Queue

Expected Outcomes:

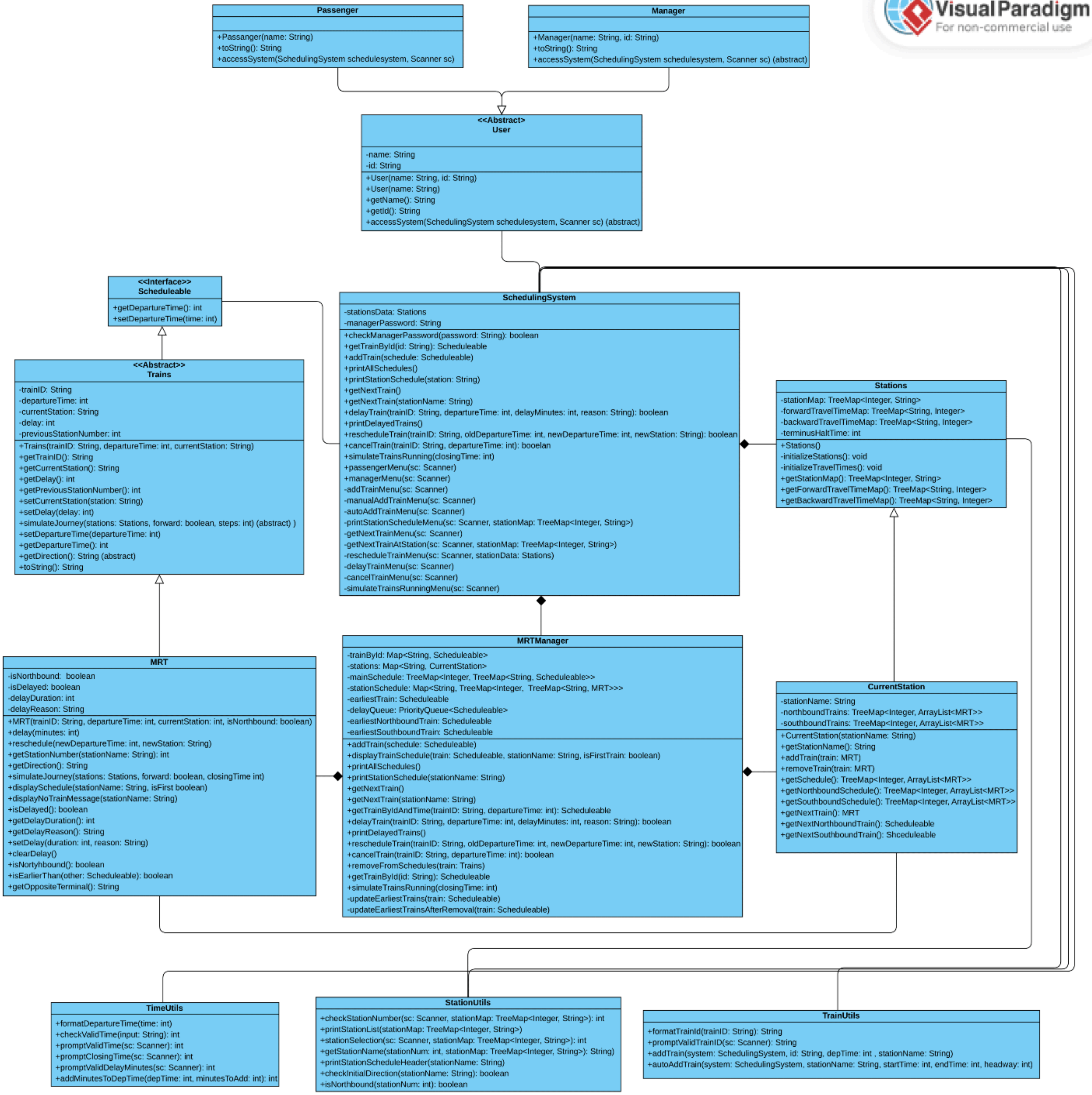
- Improved commuter satisfaction due to minimal waiting time.
- Ridership growth in the MRT through enhanced reliability.
- An extensible system that can be utilized for other transportation systems.

2.2. Solution Design

Here is the class diagram for our MRT scheduling system, complete with both the manager and user interactions. The class diagram includes the various functions we have used both for the manager and user sides.

Class Diagram of Completed System

Here is a link to the PDF file for a clearer view of the class diagram: https://drive.google.com/file/d/1BGjTrXK81MUQ9SVlxXhmDQoAwxUo4LCy/view?usp=sharing


Passenger
+Passanger(name: String)
+toString(): String
+accessSystem(SchedulingSystem schedulesystem, Scanner sc)
Manager
+Manager(name: String, id: String)
+toString(): String
+accessSystem(SchedulingSystem schedulesystem, Scanner sc) (abstract)
<<Abstract>
User
-name: String
-id: String
+User(name: String, id: String)
+User(name: String)
+getName(): String
+getId(): String
+accessSystem(SchedulingSystem schedulesystem, Scanner sc) (abstract)
<<Interface>>
Scheduleable
+getDepartureTime(): int
+setDepartureTime(time: int)
<<Abstract>>
Trains
-trainID: String
-departureTime: int
-currentStation: String
-delay: int
-previousStationNumber: int
+Trains(trainID: String, departureTime: int, currentStation: String)
+getTrainID(): String
+getCurrentStation(): String
+getDelay(): int
+getPreviousStationNumber(): int
+setCurrentStation(station: String)
+setDelay(delay: int)
+simulateJourney(stations: Stations, forward: boolean, steps: int) (abstract))
+setDepartureTime(departureTime: int)
+getDepartureTime(): int
+getDirection(): String (abstract)
+toString(): String
SchedulingSystem
-stationsData: Stations
-managerPassword: String
+checkManagerPassword(password: String): boolean
+getTrainById(id: String): Scheduleable
+addTrain(schedule: Scheduleable)
+printAllSchedules()
+printStationSchedule(station: String)
+getNextTrain()
+getNextTrain(stationName: String)
+delayTrain(trainID: String, departureTime: int, delayMinutes: int, reason: String): boolean
+printDelayedTrains()
+rescheduleTrain(trainID: String, oldDepartureTime: int, newDepartureTime: int, newStation: String): boolean
+cancelTrain(trainID: String, departureTime: int): booelan
+simulateTrainsRunning(closingTime: int)
+passengerMenu(sc: Scanner)
+managerMenu(sc: Scanner)
-addTrainMenu(sc: Scanner)
-manualAddTrainMenu(sc: Scanner)
-autoAddTrainMenu(sc: Scanner)
-printStationScheduleMenu(sc: Scanner, stationMap: TreeMap<Integer, String>)
-getNextTrainMenu(sc: Scanner)
-getNextTrainAtStation(sc: Scanner, stationMap: TreeMap<Integer, String>)
-rescheduleTrainMenu(sc: Scanner, stationData: Stations)
-delayTrainMenu(sc: Scanner)
-cancelTrainMenu(sc: Scanner)
-simulateTrainsRunningMenu(sc: Scanner)
Stations
-stationMap: TreeMap<Integer, String>
-forwardTravelTimeMap: TreeMap<String, Integer>
-backwardTravelTimeMap: TreeMap<String, Integer>
-terminusHaltTime: int
+Stations()
-initializeStations(): void
-initializeTravelTimes(): void
+getStationMap(): TreeMap<Integer, String>
+getForwardTravelTimeMap(): TreeMap<String, Integer>
+getBackwardTravelTimeMap(): TreeMap<String, Integer>
MRT
-isNorthbound: boolean
-isDelayed: boolean
-delayDuration: int
-delayReason: String
+MRT(trainID: String, departureTime: int, currentStation: int, isNorthbound: boolean)
+delay(minutes: int)
+reschedule(newDepartureTime: int, newStation: String)
+getStationNumber(stationName: String): int
+getDirection(): String
+simulateJourney(stations: Stations, forward: boolean, closingTime int)
+displaySchedule(stationName: String, isFirst boolean)
+displayNoTrainMessage(stationName: String)
+isDelayed(): boolean
+getDelayDuration(): int
+getDelayReason(): String
+setDelay(duration: int, reason: String)
+clearDelay()
+isNortyhbound(): boolean
+isEarlierThan(other: Scheduleable): boolean
+getOppositeTerminal(): String
MRTManager
-trainById: Map<String, Scheduleable>
-stations: Map<String, CurrentStation>
-mainSchedule: TreeMap<Integer, TreeMap<String, Scheduleable>>
-stationSchedule: Map<String, TreeMap<Integer, TreeMap<String, MRT>>>
-earliestTrain: Scheduleable
-delayQueue: PriorityQueue<Scheduleable>
-earliestNorthboundTrain: Scheduleable
-earliestSouthboundTrain: Scheduleable
+addTrain(schedule: Scheduleable)
+displayTrainSchedule(train: Scheduleable, stationName: String, isFirstTrain: boolean)
+printAllSchedules()
+printStationSchedule(stationName: String)
+getNextTrain()
+getNextTrain(stationName: String)
+getTrainByIdAndTime(trainID: String, departureTime: int): Scheduleable
+delayTrain(trainID: String, departureTime: int, delayMinutes: int, reason: String): boolean
+printDelayedTrains()
+rescheduleTrain(trainID: String, oldDepartureTime: int, newDepartureTime: int, newStation: String): boolean
+cancelTrain(trainID: String, departureTime: int): boolean
+removeFromSchedules(train: Trains)
+getTrainById(id: String): Scheduleable
+simulateTrainsRunning(closingTime: int)
-updateEarliestTrains(train: Scheduleable)
-updateEarliestTrainsAfterRemoval(train: Scheduleable)
CurrentStation
-stationName: String
-northboundTrains: TreeMap<Integer, ArrayList<MRT>>
-southboundTrains: TreeMap<Integer, ArrayList<MRT>>
+CurrentStation(stationName: String)
+getStationName(): String
+addTrain(train: MRT)
+removeTrain(train: MRT)
+getSchedule(): TreeMap<Integer, ArrayList<MRT>>
+getNorthboundSchedule(): TreeMap<Integer, ArrayList<MRT>>
+getSouthboundSchedule(): TreeMap<Integer, ArrayList<MRT>>
+getNextTrain(): MRT
+getNextNorthboundTrain(): Scheduleable
+getNextSouthboundTrain(): Shceduleable
TimeUtils
+formatDepartureTime(time: int)
+checkValidTime(input: String): int
+promptValidTime(sc: Scanner): int
+promptClosingTime(sc: Scanner): int
+promptValidDelayMinutes(sc: Scanner): int
+addMinutesToDepTime(depTime: int, minutesToAdd: int): int
StationUtils
+checkStationNumber(sc: Scanner, stationMap: TreeMap<Integer, String>): int
+printStationList(stationMap: TreeMap<Integer, String>)
+stationSelection(sc: Scanner, stationMap: TreeMap<Integer, String>): int
+getStationName(stationNum: int, stationMap: TreeMap<Integer, String>): String)
+printStationScheduleHeader(stationName: String)
+checkInitialDirection(stationName: String): boolean
+isNorthbound(stationNum: int): boolean
TrainUtils
+formatTrainId(trainID: String): String
+promptValidTrainID(sc: Scanner): String
+addTrain(system: SchedulingSystem, id: String, depTime: int , stationName: String)
+autoAddTrain(system: SchedulingSystem, stationName: String, startTime: int, endTime: int, headway: int)

Discussion on Solution Scheme and Data Structure Implementation

The decision to implement the three different data structures in this way is further explained by the data and analysis provided in Chapter 3.

DIMSUM is designed to be an efficient MRT scheduling system that manages train schedules, user roles, and station data in the Jakarta MRT network. The system supports two main roles: passengers (which are the users that can only see the schedules), and the managers (this role allows adding schedules, delaying, rescheduling, or even cancelling).

The architecture of the program leverages object-oriented principles such as encapsulation, inheritance, and polymorphism to enhance the program's modularity and maintainability.

Core components of the program:

1. **User management logic:** This part includes the abstract User class, and both Manager and Passenger classes, where these two classes are generalized into the User abstract class by using inheritance relationship.
2. **Train scheduling logic:** This part handles the creation of a train object represented by the MRT class, where MRT is a subclass of the more general Trains abstract class, and the Trains abstract class implements the interface Schedulables, this ensures that all train types provide scheduling-related methods. Another class in this logic is the MRTManager class, this class manages the collection of MRT objects, and is responsible for updating schedules, managing delays, and cancelling.
3. **Station management:** This part manages the station data such as travel times, and also storing the name of the station a specific train object is in. The station management logic consists of classes Stations and CurrentStation, where CurrentStation inherits the Stations class.
4. **Utility classes:** These classes provide static methods that can help the program in processing the data. The classes are: TimeUtils, StationUtils, TrainUtils. The functions provided inside these classes are true to its name, if for example, the train scheduling logic needs to add delay in minutes into the format of HHMM, the TimeUtils class will help in achieving that. By incorporating these classes we can reduce code duplication and ensure consistent data handling for the whole program.
5. **Scheduling system logic:** This part only consists of the SchedulingSystem class which manages the main user interface. This class enables the user to interact with the program, giving data input and getting the outputs they want. The SchedulingSystem class gives error handling, menu logic, and references to important classes.

Data Structure Implementation:

1. Java Collections

   Treemap:

   This is used specifically for strong schedules and station data.

   Example: TreeMap <Integer, String> for mapping station numbers to names.

   Example: TreeMap<Integer, ArrayList<MRT>> for mapping departure times to lists of trains at a station.

   Advantage: TreeMap keeps keys sorted, allowing efficient range queries and schedule lookups.

HashMap:
Used for quick lookup of trains and stations by ID or name.
Example: HashMap<String, Schedulable> for mapping train IDs to train objects.

PriorityQueue:
Used for relaying information of delayed trains, where the train with the largest delay will have the highest priority, hence it can be processed first.

ArrayList:
Used for storing lists of trains scheduled at the same time or for holding collections of stations.

2. Custom Classes
   Stations:
   It encapsulates all the station names and travel times by using TreeMaps for immutability and fast access.

   Current Station:
   It extends stations and adds dynamic scheduling for northbound and southbound trains using TreeMaps of ArrayLists.

3. Utility Classes
   Time Utils, Train Utils, Station Utils:
   Provide static methods for validation and formatting which ensures integrity of the data throughout the system.

Main Algorithms and Functions

1. Train Scheduling and Lookup
   Adding a Train:
   When a manager adds a train, the system verifies the train ID and departure time and validates it. It then inserts the train into the appropriate TreeMap(s) for both global and per-station schedules.

2. Delay and Rescheduling
   Delay Algorithm:
   When a train is delayed, the system will update the delay field and moves the train into a new time slot created/provided in the TreeMap, ensuring all the schedules remain consistent.

   Rescheduling:
   Removes the train from its old time slot and inserts it into a new time slot. It then updates all relevant data structures.

3. User Interaction
   Role-Based Menus:
   The main loop in Main class presents different menus and options based on whether the user is either a passenger or manager.

   Input Validation:
   All user inputs (times, IDs, station numbers) are validated/processed beforehand using utility classes.

4. Station and Time Management
   Travel Time Calculation:
   The system uses the “travel time” maps in Station to compute the journey durations and simulate train movement.

   Station Selection:
   Users can select stations by either number or name. With the help of validation, invalid choices are prevented.

In conclusion, this solution scheme makes use of Java’s collection framework for efficient data storage and retrieval, this is achieved by testing the implementation for three different data structures: HashMap, TreeMap, and PriorityQueue. The solution also applies object-oriented design that improves the maintainability, readability, and reusability of the code; protects the integrity of data by abstraction and encapsulation; and also ensures the scalability of the system for future enhancements.

CHAPTER 3

3.1. Test Results

Tables

Input size: 10

Operation Type	Data Structure	Input Size	Runtime (ms)	Memory Usage (KB)
Add schedules	TreeMap	10	1.0835	-1.5625
Add schedules	HashMap	10	0.80867	12.33906
Add schedules	PriorityQueue	10	1.1391	1290.2578
Print all schedules	TreeMap	10	0.26632	0.296875
Print all schedules	HashMap	10	0.02039	0.0
Print all schedules	PriorityQueue	10	0.0814	1290.2891
Print for station	TreeMap	10	0.21336	0.109375
Print for station	HashMap	10	0.02277	0.005469
Print for station	PriorityQueue	10	0.0491	1290.3359
Find next train	TreeMap	10	0.82032	0.4375
Find next train	HashMap	10	0.972390	5.55859375
Find next train	PriorityQueue	10	0.0427	0.0313

Reschedule train	TreeMap	10	2.36694	26.8828125
Reschedule train	HashMap	10	0.0271	0.009375
Reschedule train	PriorityQueue	10	0.0594	0.0
Delay train	TreeMap	10	0.67786	0.296875
Delay train	HashMap	10	0.01704	0.0
Delay train	PriorityQueue	10	0.0812	1290.3359
Cancel train	TreeMap	10	0.22785	0.25
Cancel train	HashMap	10	0.02325	0.0046875
Cancel train	PriorityQueue	10	0.0462	0.0

Input size: 100

Operation Type	Data Structure	Input Size	Runtime (ms)	Memory Usage (KB)
Add schedules	TreeMap	100	2.5739	14.609375
Add schedules	HashMap	100	1.17748	106.109375
Add schedules	PriorityQueue	100	1.2761	1290.2578
Print all schedules	TreeMap	100	0.474	1.671875
Print all schedules	HashMap	100	0.02437	0.0

Print all schedules	PriorityQueue	100	0.2075	1290.2891
Print for station	TreeMap	100	0.39754	0.109375
Print for station	HashMap	100	0.02468	0.00546875
Print for station	PriorityQueue	100	0.0473	1290.2969
Find next train	TreeMap	100	0.7658	0.4375
Find next train	HashMap	100	1.26802	5.5296875
Find next train	PriorityQueue	100	0.0367	0.03125
Reschedule train	TreeMap	100	0.4143	-0.03125
Reschedule train	HashMap	100	0.02296	0.009375
Reschedule train	PriorityQueue	100	0.0531	0.0
Delay train	TreeMap	100	0.8999	-0.71875
Delay train	HashMap	100	0.01888	0.0
Delay train	PriorityQueue	100	0.1128	1290.3359
Cancel train	TreeMap	100	0.3270	10.171875
Cancel train	HashMap	100	0.02137	0.0046875
Cancel train	PriorityQueue	100	0.0640	0.0

Input size: 1000

Operation Type	Data Structure	Input Size	Runtime (ms)	Memory Usage (KB)
Add schedules	TreeMap	1000	7.6554	90.3515625
Add schedules	HashMap	1000	3.63095	1042.1375
Add schedules	PriorityQueue	1000	3.4810	1290.2578
Print all schedules	TreeMap	1000	1.1053	2.359375
Print all schedules	HashMap	1000	0.02892	0.0
Print all schedules	PriorityQueue	1000	0.4291	1290.2891
Print for station	TreeMap	1000	1.2442	0.109375
Print for station	HashMap	1000	0.02892	0.0
Print for station	PriorityQueue	1000	0.0356	0.03125
Find next train	TreeMap	1000	1.616	0.3671875
Find next train	HashMap	1000	0.98961	5.54375
Find next train	PriorityQueue	1000	0.0449	0.03125
Reschedule train	TreeMap	1000	0.6681	0.0859375
Reschedule train	HashMap	1000	0.02191	0.0
Reschedule train	PriorityQueue	1000	0.0510	0.0

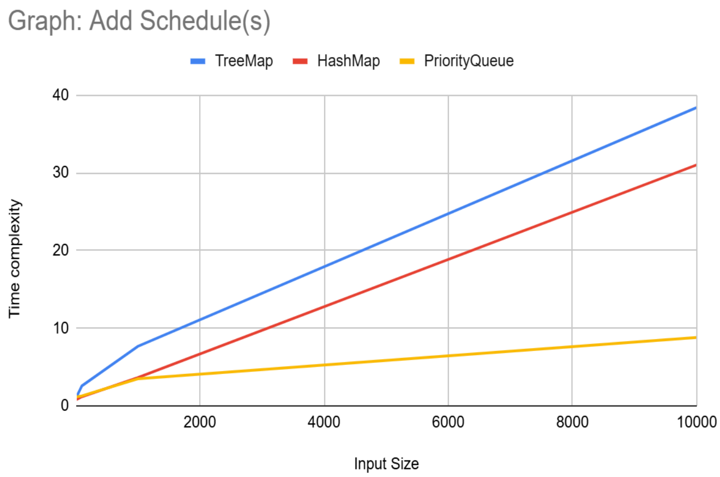
Delay train	TreeMap	1000	1.3193	-0.6015625
Delay train	HashMap	1000	0.01765	0.0
Delay train	PriorityQueue	1000	0.1242	1290.3359
Cancel train	TreeMap	1000	0.5214	10.1484375
Cancel train	HashMap	1000	0.0173	0.0
Cancel train	PriorityQueue	1000	0.0499	0.0

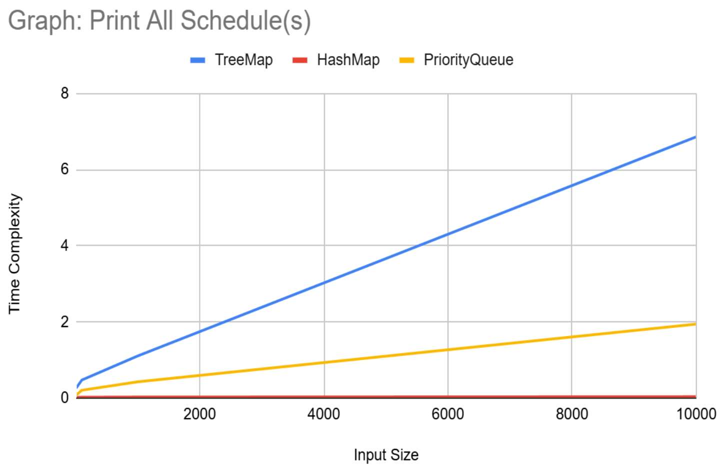
Input size: 10000

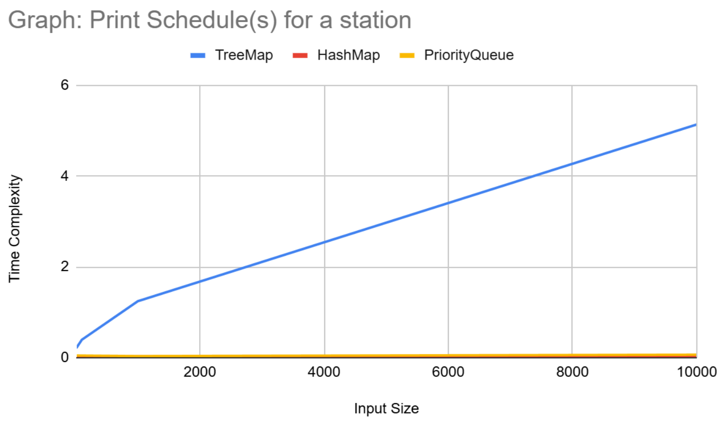
Operation Type	Data Structure	Input Size	Runtime (ms)	Memory Usage (KB)
Add schedules	TreeMap	10000	38.4217	792.8359375
Add schedules	HashMap	10000	31.0266	10507.9594
Add schedules	PriorityQueue	10000	8.8116	5177.28125
Print all schedules	TreeMap	10000	6.8607	2.671875
Print all schedules	HashMap	10000	0.03766	0.0
Print all schedules	PriorityQueue	10000	1.9452	1290.2890
Print for station	TreeMap	10000	5.13684	-732.6015625
Print for station	HashMap	10000	0.02614	0.0

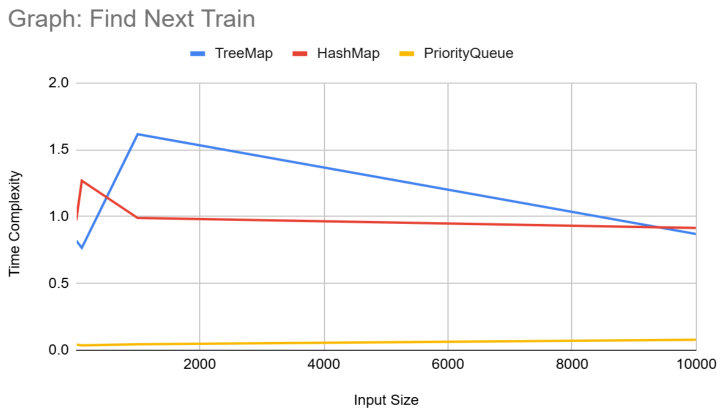
Print for station	PriorityQueue	10000	0.06728	1290.2656
Find next train	TreeMap	10000	0.86956	-732.453125
Find next train	HashMap	10000	0.91434	5.54375
Find next train	PriorityQueue	10000	0.0785	0.03125
Reschedule train	TreeMap	10000	0.40202	-732.6015625
Reschedule train	HashMap	10000	0.01816	0.0
Reschedule train	PriorityQueue	10000	0.06912	0.0
Delay train	TreeMap	10000	0.82298	-733.390625
Delay train	HashMap	10000	0.01645	0.0
Delay train	PriorityQueue	10000	0.08244	1290.3359
Cancel train	TreeMap	10000	1.11443	10.1328125
Cancel train	HashMap	10000	0.02574	0.0
Cancel train	PriorityQueue	10000	0.0626	0.0

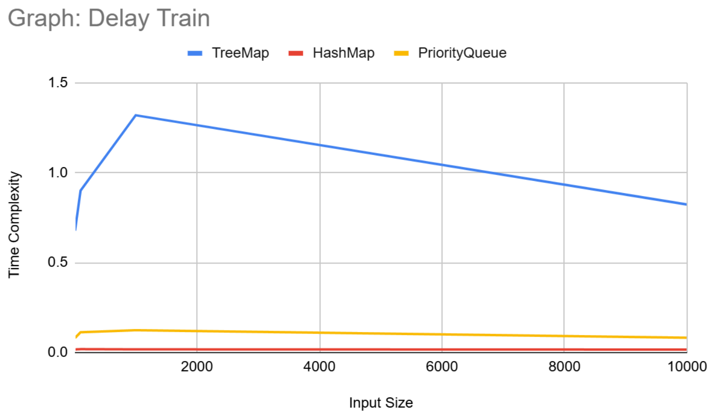
Graphs

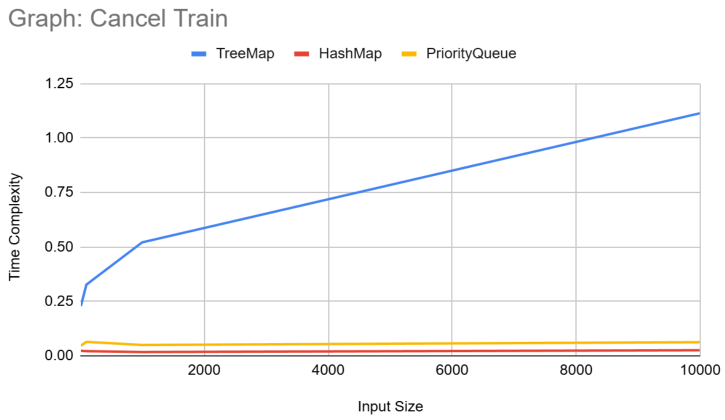

Graph: Add Schedule(s)


Graph: Print All Schedule(s)


Graph: Print Schedule(s) for a station


Graph: Find Next Train


Graph: Delay Train


Graph: Cancel Train

3.2. Complexity Analysis

HashMap

1. **Add Schedule(s)**
   - Time Complexity: O(k), basically O(n) constant.
   - Space Complexity: O(k)
   - This operation creates a list of “k” stations and inserts it into the train map. HashMap insertion is O(1), but the overall cost comes from adding each station.
2. **Print All Schedule(s)**
   - Time Complexity: O(n × m)
   - Space Complexity: O(1)
   - The function loops through all “n” trains and prints their “m” stations. It doesn't create new data structures, only reads existing ones.
3. **Print Schedule(s) for a Certain Station**
   - Time Complexity: O(n × m)
   - Space Complexity: O(1)
   - It checks each train and each of its stations to find matches for the target station. Since it only prints results, no extra memory is needed.
4. **Print Next Departing Train**
   - Time Complexity: O(n × m)
   - Space Complexity: O(1)
   - This operation scans every train and station to find the earliest departure time. Time parsing is constant, and only a few variables are used to track the result.

5. **Reschedule Train(s)**

   ○ Time Complexity: O(m)

   ○ Space Complexity: O(1)

   ○ It quickly looks up the train, then searches through its list of “m” stations to find and update the one that needs rescheduling.

6. **Delay Train(s)**

   ○ Time Complexity: O(m)

   ○ Space Complexity: O(1)

   ○ After finding the train, it loops through all “m” stations and adjusts their times. The function works directly on existing data.

7. **Cancel Train(s)**

   ○ Time Complexity: O(1)

   ○ Space Complexity: O(1)

   ○ Removing a train from the HashMap takes constant time. There is no iteration or memory allocation required.

Summary of Performance Characteristics

- Most Efficient (O(1)):
  Canceling entire trains and accessing specific train schedules.

- Moderately Efficient (O(m)):
  Adding new schedules, rescheduling stations, and delaying trains.

- Least Efficient (O(n × m)):
  Printing all schedules, finding schedules for specific stations, and identifying the next departing train.

Space Usage Overview

Most operations use constant space, O(1), as they only work with existing data. The only exception is when adding a new train schedule, which temporarily uses O(k) space to store the list of new stations. The total memory needed to store all train data is O(n × m), representing all trains and their stations.

TreeMap

1. Add Schedules(s)

The add schedule(s) function in the TreeMap implementation takes four parameters: departureTime: int, trainID: String, stationName: String, testing: Boolean. This method has two main logics, one is the function .computeIfAbsent(), and the other is .put().

.computeIfAbsent() is used for checking whether the TreeMap already has the same departure time data, if not it will create a new TreeMap {trainID, stationName}, and this will be .put() inside of another TreeMap with the departure time as the key. The results will look like this: {departureTime, {trainID, stationName}}. This is so that the data maintains a sorted departureTime. Another logic using .put() but is for another TreeMap called isOnTime, which stores data of train delays.

- Time complexity for .computeIfAbsent(): O(log n)
- Time complexity for .put() or insert function for TreeMap: O(log n)
- Overall: O(log n + log n) = **O(log n)**

Space complexity of the method itself is: **O(1)**

2. Print all schedule(s)

The print all schedule(s) method in the TreeMap implementation is the same method as to printing schedules for a specific station, but for printing all the schedules, we just input null for the selectedStation parameter.

The method has a nested loop. The outer loop iterates through all the departure time in the outer TreeMap. The functions used are: entry.getKey() (this is to get all the departureTime) and entry.getValue() (this is to get the inner TreeMap). The inner loop iterates through the inner TreeMap and uses the same functions to get the trainID and stationName.

Let n = number of departure time, and m = number of trains for that specific departure time. The time complexity should then be O(n * m). But because in real situations, the number of trains that have the same departure time should be limited (not that big). Therefore, in most cases the time complexity should be **O(n).** Meanwhile, for space complexity since it only uses local variables, it will be constant: **O(1).**

3. Print schedule(s) for a specific station

As mentioned before, for TreeMap implementation, the method for printing schedule(s) for a specific station is the exact same as printing all the schedule(s). Hence, **the time complexity and space complexity does not change: O(n) and O(1) respectively.**

4. Find next train

The find next train method basically takes the first key of the TreeMap by using the .firstKey() function. Since TreeMaps maintains a sorted order for their keys, taking the earliest departure time equates to O(1) time complexity.

Another function is to get the data of that first train itself, this is done by using .get() which requires the same time complexity as adding the schedules which are O(log n + log n) or simply **O(log n) time complexity.** Meanwhile, space complexity is the same, since the method itself does not need to store all the train's data, hence it has **O(1)** space complexity.

5. Reschedule a train

The reschedules train method has parameters for: trainID, oldDepTime (the current departure time data already in the TreeMap), newDepTime (departure time data that will replace the old one), newStation (this can be optional, but it can replace the old station's data).

The functions in this method consist of adding and removing functions, as mentioned before these operations take **O(log n) time**, where n is the number of schedules. Meanwhile, for space complexity, it is **O(1)** since it does not create a new data structure, but just uses local variables.

6. Delay a train

The delay train method takes parameters: trainID, oldDepTime, and delayMinutes (the amount of delay in minutes). This method is similar to rescheduling a train as it takes the trainID and the current departure time and finds it in the TreeMap. The difference is that the delayMinutes is being added mathematically into the current departure time (oldDepTime), by using another helper method, but that takes O(1) time complexity. So the time complexity is still dominated by .get(), and .remove() functions which takes **O(log n) time complexity.**

Same as any other methods, the space complexity for delaying a train is **O(1)** since it does not create a new data structure to store additional data.

7. Cancel a train

The cancel train method has parameters trainID and oldDepTime. From here we can see that reschedule/delay/cancel all has similar logic to it. For cancelling a train it only needs to find the corresponding data given by input. After it has been found in the TreeMap using the .containsKey() and .get() functions, it is then removed from the TreeMap using .remove(). All these operations take **O(log n) time complexity.** The space complexity for canceling a train is also **O(1)** just like the other methods.

Summary

The TreeMap based scheduling system utilizes Red-Black trees to ensure efficient and balanced operations. The Red-Black tree has a self-balancing structure which is why key methods such as adding or removing takes O(log n) time complexity. This allows quick scheduling and data retrievals. Due to this self-balancing nature, Red-Black trees can prevent self degradation, this means that its operations will always stay at O(log n), unlike hashmap for example, which due to collision, the time complexity can differ. All of these characteristics are well suited for our goal in creating an MRT scheduling system.

PriorityQueue

1. **Add Schedule(s)**

The Add Schedule(s) method is a combined method of addStation and addTrain function. The inputs in this function will be inserted into a list of stations called schedule. Insertion to a priority queue takes O(log n) time while the space takes O(n) byte to use. Whereas inserting the station that has a priority queue called trainQueue as an instance into the list of stations called schedule, it took O(n) for time and O(1) for space. Since the add schedule method is a combined method between these two methods, it will take O(n log n) time and O(n) space.

2. **Print All Schedule(s)**

Print All Schedule(s) returns all stations inside the schedule list and a copy of the Priority Queue inside the station class. Since it returns a copy of the priority queue it takes O(n) time and space to operate.

3. **Print Schedule(s) for a Certain Station**

This method has the same operation as the Print All Schedule(s), the difference is it only returns some specific stations with its trainQueue. Print Schedule(s) for a Certain station uses O(n) time and space to run since it also returns a copy of Priority Queue (trainQueue).

4. **Print Next Departing Train**

The print next departing train method used the peek() function inside the Priority Queue class. This method prints the next departing train inside a station. The peek() method uses O(1) time and space to run and so is this method.

5. **Reschedule Train(s)**

The Reschedule Train(s) method takes an int called newPriority and a Train as an input. This method removes the exact train then changes its priority with the inputted priority after that it adds the train again into the priority queue. The method combines a deletion and insertion so that it uses O(n + log n) => O(n) time to run and O(1) space since there’s no change of size inside the queue.

6. **Delay Train(s)**

Delay Train(s) method Train and int called delay as an input. This method removes the exact train and changes its departureTime then it loops through the queue to find the closest departure time of a train to the train’s delayed departure time and it adds the trains priority with the closest train’s priority. After that the method inserts this train again into the queue. Considering the deletion, loop and insertion, it takes O(n + n + log n) => O(n) time to run and since it doesn’t change the size of the queue it uses O(1) space to run.

7. **Cancel Train(s)**

Cancel Train(s) method removes a train from a queue. The deletion method took O(n) time and O(1) space to run. In the table above the space doesn't show a negative value (deletion). This occasion happened because of the Collection Internal Structure itself.

Collections like ArrayList or PriorityQueue may not shrink their internal arrays when removing elements. The backing array remains the same size until it is explicitly trimmed or the collection implementation resizes it.

Conclusion of the Analysis

According to the data and analysis we got, the implementation of our MRT schedule manager can achieve the most efficient time complexity by combining these three data structures for specific functions.

1. Utilizing TreeMap for the global schedule → TreeMap<departureTime, TreeMap<String, Train>>

   We chose this to preserve the order of departure time when printing the schedule, this is due to the self-balancing structure of TreeMap that orders its key.

2. Creating two HashMaps for efficient and fast lookups by referring to the trainID, and another one for linking station name to the station object → HashMap<String, Train>, HashMap<String, Station>

   By implementing this, our schedule manager can enable O(1) lookups by referring to trainID, and also constant time for finding the station object (which has information of trains in that specific station), this is done by looking at the key, which is the station name (type String).

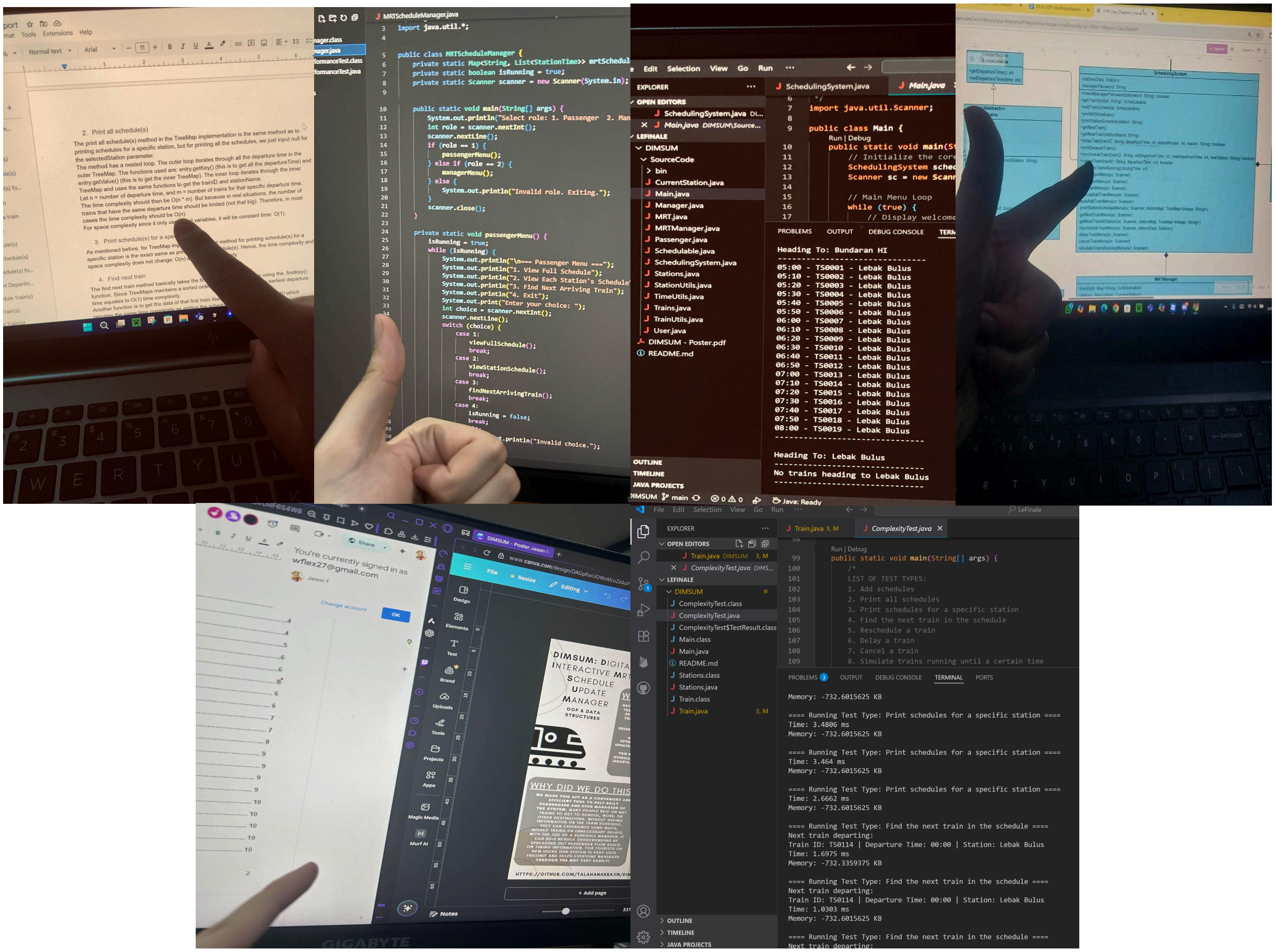
3. The use of PriorityQueue is of course going to be focusing on delaying a train schedule. This is done by creating a queue called delayQueue → PriorityQueue<Train>

   This is so that we can prioritize trains that have the largest delay, and we can print these trains in order from the largest delay. By organizing delayed trains this way, train managers can easily identify and address the most critical disruption in the system.

Other implementations of data structure can be added as well to help in achieving the best results of time and space complexity. Nevertheless, this is already a strong foundation to follow for our completed system.

CHAPTER 4

4.1. Documentation



4.2. Appendix

Github:

https://github.com/talahanakrajin/DIMSUM.git

Poster:

https://www.canva.com/design/DAGpKwUOWoM/oZil4oP53IU-JLo8ZtHmkQ/edit?utm_content=DAGpKwUOWoM&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton

4.3. References

PT MRT Jakarta. (n.d.). Jadwal keberangkatan MRT. Jakarta MRT.

Aho, A. V., Hopcroft, J. E., & Ullman, J. D. (1983). *Data structures and algorithms*. Addison-Wesley.

Boyar, J., & Larsen, K. (1999). The seat reservation problem. Algorithmica, 25(4), 403–417.

Brehm, W. (2019). Hash tables with pseudorandom global order. INFOCOMP Journal of Computer Science, 18(1), 20–25.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to algorithms (3rd ed.). MIT Press.

Downey, A. B. (2017). Think data structures: Algorithms and information retrieval in Java. O'Reilly Media.

Ferris, B., Watkins, K., & Borning, A. (2010). OneBusAway: Results from providing real-time arrival information for public transit. Transportation Research Record, 2143(1), 180–188.

Koffman, E. B., & Wolfgang, P. A. (2021). Data structures: Abstraction and design using Java (4th ed.). John Wiley & Sons.

Lafore, R. (2017). Data structures and algorithms in Java. Pearson.

Weiss, M. A. (2010). Data structures and problem-solving using Java (4th ed.). Addison-Wesley.

Winona State University. (2016). *Proceedings of the 16th Winona Computer Science Undergraduate Research Symposium* (pp. 35-36). Winona State University.

Wu, Y., & Liu, Y. (2020). Optimization of seat allocation with fixed prices: An application of the discrete choice model. Journal of Revenue and Pricing Management, 19(3), 201–218.