COMPUTER SCIENCE 21A (SPRING, 2022)
DATA STRUCTURES AND ALGORITHMS

PROGRAMMING ASSIGNMENT 1 – MBTA RED LINE

Overview:

You have been tasked to run a simulation of the Red Line for the MBTA. This will entail populating the stations with trains and passengers from text files, run the simulation, and print out a log that includes the status of the railway's stations and the movement of trains and passengers.

Basic Idea:

You will be given three text files from which you will create the stations, the trains, and the passengers. Each station will maintain queues for passengers who are waiting to travel. Since a station can load and unload passengers onto only 1 train at a time, it will also have queues for any additional trains who have arrived at the station. Trains can hold a limited number of passengers and can only travel in one direction at a time.

Implementation Details:

The provided zip file on LATTE contains 13 files:

- 3 data structures related classes (Node.java, Queue.java, and DoubleLinkedList.java)
- 5 MBTA related classes (Rider.java, Train.java, Station.java, Railway.java, and MBTA.java)
- 2 JUnit test files for you to write tests for your implementations of a queue and doubly linked list.
- 3 Provided JUnit test files that include basic tests for your MBTA classes.

The 3 data structures classes and 5 MBTA related classes are described in detail below. **You must implement the following methods following these exact signatures.** Furthermore, some of these classes include specific fields. **You must also include these in your implementation and leave them as** `public`**.**

Queue.java

This class provides the implementation of a generic circular first-in-first-out queue. In your implementation, you must **use the following 4** `public` **fields:**

`public T[] q` – generic array to store the contents of the queue.

`public int head` – index of the front of the queue, from which you should dequeue.

`public int tail` – index of where **the next element** will be enqueued.

`public int numEntries` – number of entries stored in the queue.

`public Queue(int capacity)` – constructs an empty queue that can hold a specified number of elements. You will find that one line of this method has been provided in the skeleton code. It is how you create a generic array in Java.

`public void enqueue(T element)` – adds an element at the tail of the queue.

`public void dequeue()` – removes the element at the head of the queue. If there is no such element, **you must throw a** `NoSuchElementException`**.**

`public T front()` – returns the element at the head of the queue. If there is no such element, **you must throw a** `NoSuchElementException`**.**

`public int size()` – return the number of elements in the queue.

`public String toString()` – return a String representation of the queue's elements.

Node.java

This class provides the implementation of a doubly linked list node. These nodes should have a pointer to the next node, a pointer to the previous node, and data.

`public Node(T data)` – constructs a doubly linked list node that holds a data field but does not point to any other nodes.

`public void setData(T data)` – sets the data field of this node.

`public void setNext(Node<T> next)` – sets the next pointer of this node.

`public void setPrev(Node<T> prev)` – sets the previous pointer of this node.

`public T getData()` – returns the data stored in this node.

`public Node<T> getNext()` – returns the pointer to the next node or null if one does not exist.

`public Node<T> getPrev()` – returns the pointer to the previous node or null if one does not exist.

`public String toString()` – returns the String representation of this node's element.

DoubleLinkedList.java

This class provides the implementation of a generic **non-circular** doubly linked list.

`public DoubleLinkedList()` – initializes a doubly linked list to have 0 elements.

`public Node<T> getFirst()` – gets the first node in the list or null if one does not exist.

`public void insert(T element)` – adds an element to the end of this list.

`public T delete(T key)` – deletes the first element from this list that matches the provided key. If the provided key does not exist in the list, return null.

`public T get(T key)` – returns the first element in the list that matches the provided key or null if one cannot be found.

`public int size()` – returns the number of elements in the list.

`public String toString()` – returns a String representation of this list's elements.

Rider.java

This class represents a Rider on the red line. A rider should have an ID, starting station, destination station, and a direction.

`public Rider(String riderID, String startingStation, String destinationStation)` – this should construct a Rider with an ID as well as starting and ending stations. A Rider must start by travelling south. You may assume that a Rider will be travelling at least 1 Station.

`public String getStarting()` – returns the name of this Rider's starting station.

`public String getDestination()` – returns the name of this Rider's ending station.

`public String getRiderID()` – returns this Rider's ID.

`public boolean goingNorth()` – returns true if this Rider is northbound. Else, false.

`public void swapDirection()` – swaps the Rider's current direction.

`public String toString()` – returns a String representation of this Rider.

`public boolean equals(Object o)` – checks if this Rider is equal to another Object based on ID.

Train.java

This class represents a Train on the red line. All Trains hold a specified number of passengers in an array, the current number of passengers, the current Station where the Train is, and the Train's current direction. You **must use the following 3** `public` **fields:**

`public static final int TOTAL_PASSENGERS` - the # of passengers that the train can hold, the default is 10 passengers

`public Rider[] passengers` - the Train's passengers.

`public int passengerIndex` - tracks the number of passengers currently in the Train.

`public Train(String currentStation, int direction)` - constructs an empty Train at a given Station going either south (1) or north (0).

`public boolean goingNorth()` - returns true if this Train is northbound. Else, return false.

`public void swapDirection()` - swaps the Train's direction.

`public String currentPassengers()` - returns a String of the current passengers on the Train. For instance, one could return:

```
7SG7IE6K7J7TZLHCHTZW, Porter
0W3E3HYLZ67MQCA6ACQ8, Porter
3A56AC65CK7D12UCE55Y, Porter
```

`public boolean addPassenger(Rider r)` - adds a passenger to the Train as long as (i) the Rider is at the correct Station to enter the Train, (ii) the Train is going in the appropriate direction, and (iii) there is space on the Train. Return true if the addition was completed. Else, return false.

`public boolean hasSpaceForPassengers()` - returns true if the Train has space for additional passengers.

`public String disembarkPassengers()` - This should remove all of the passengers who should be exiting the Train at the Train's current station. It should also return a String of the removed passengers. For instance, one could return:

```
7SG7IE6K7J7TZLHCHTZW, Porter
0W3E3HYLZ67MQCA6ACQ8, Porter
3A56AC65CK7D12UCE55Y, Porter
```

If there are no passengers to remove, return an empty String.

`public void updateStation(String s)` - Updates the name of this Train's current station to be the name of another station.

`public String getStation()` - returns the name of the Train's current Station.

`public String toString()` – returns a String representation of this Train.

Station.java

This class represents a Station on the red line. A Station should track which Trains and Riders are waiting to go north or south. You **must use the following 4** `public` **fields:**

`public Queue<Rider> northBoundRiders` – queue for riders waiting to go north

`public Queue<Rider> southBoundRiders` – queue for riders waiting to go south

`public Queue<Train> northBoundTrains` – queue for trains waiting to go north

`public Queue<Train> southBoundTrains` – queue for trains waiting to go south

`public Station(String name)` – constructs an empty Station with a given name.

`public boolean addRider(Rider r)` – adds a Rider to the appropriate Queue, depending on the Rider's direction, as long as they should be in this Station. Return true if this is possible and false otherwise.

`public String addTrain(Train t)` – moves a Train into this Station, removes all of the passengers who are meant to disembark at this Station, and places the Train in the appropriate Queue depending on its direction. This method should return a String that includes that some passengers were removed at this Station. For instance, one could return:

```
Quincy Adams Disembarking Passengers:
FVPRE0BX09L3OMS2VR87
```

`public Train southBoardTrain()` – This method will prepare a southbound Train of passengers by performing the following steps:

1) Dequeuing a train from the southbound train queue.
2) Adding as many southbound Riders to the Train as possible.
3) Return the train that has been filled or null if there are no waiting southbound Trains.

`public Train northBoardTrain()` – This method will prepare a northbound Train of passengers by performing the following steps:

1) Dequeuing a train from the northbound train queue.
2) Adding as many northbound Riders to the Train as possible.
3) Return the train that has been filled or null if there are no waiting northbound Trains.

`public void moveTrainNorthToSouth()` – changes the direction of the first waiting northbound Train and moves it to the southbound queue. You may assume that the Train will not have any Riders.

`public void moveTrainSouthToNorth()` – changes the direction of the first waiting southbound Train and moves it to the northbound queue. You may assume that the Train will not have any Riders.

`public String stationName()` – returns the name of this Station

`public String toString()` – this should return the name and status of the station. For instance, one could return:

```
Station: Alewife
1 north-bound trains waiting
0 south-bound trains waiting
0 north-bound passengers waiting
1 south-bound passengers waiting
```

`public boolean equals(Object o)` – Checks if a Station is equal to some object based on name.

Railway.java

This class represents the red line railway and is made up of a list of Stations. You **must use the following 2** `public` **fields:**

`public DoubleLinkedList<Station> railway` – a list of the Railway's Stations.

`public String[] stationNames` – the names of the Stations on the Railway. This will contain a maximum of 18 entries corresponding to the 18 stations listed in redLine.txt.

`public Railway()` – constructs an empty Railway.

`public void addStation(Station s)` – adds a Station to the Railway.

`public void addRider(Rider r)` – (i) sets a Rider's direction based on the order of the Stations in the Railway and (ii) adds the Rider to the appropriate Station in the Railway.

`public void addTrain(Train t)` – adds a Train to the appropriate Station in this Railway.

`public void setRiderDirection(Rider r)` – sets a Rider's direction based on the Rider's starting and ending Stations.

`public String simulate()` – This method will execute one simulation of the Railway. You should log the events that occur in the process of your simulation such as the status of each Station, where Trains and Riders are, and when Riders exit a Train. This log should be returned by this method for use in the main class MBTA.java.

During this method, you should traverse the Stations in the Railway north to south and perform the following steps at each Station in this order:

1. Board a (southbound/northbound) train with as many passengers as possible unless the Station is (Braintree/Alewife).
2. Move the boarded trains to their next Stations.
3. Disembark any passengers who are meant to get off at the Stations that the trains were moved to.
4. If the current Station is (Braintree/Alewife) then you should move a (southbound train to go north/northbound train to go south).

Note: you should never have a train switch direction and move from that Station in a single call to this method.

`public String toString()` – returns the Stations list's String representation.

MBTA.java

This class contains your program's main method and should run a simulation of a Railway. The class contains the following 4 fields:

`public static final int NORTHBOUND` – value of the northbound direction (0).

`public static final int SOUTHBOUND` – value of the southbound direction (1).

`static final int TIMES` – the number of times you will run Railway's `simulate()` in the method below `runSimulation()`

`static final Railway r` – the Railway that will be used in the simulation.

`public static void main(String[] args)` – The main method should construct the Railway with the Stations, Riders, and Trains loaded from the provided text files and then run the simulation.

`public static void runSimulation()` – This method runs the simulation using `TIMES` and Railway's `simulate()`.

`public static void initTrains(String trainsFile)` – constructs Trains from an input file and adds them to the Railway.

`public static void initRiders(String ridersFile)` – constructs Riders from an input file and adds them to the Railway.

`public static void initStations(String stationsFile)` – constructs Stations from an input file and adds them to the Railway.

Provided Text Files

You have been provided three text files. Feel free to change them to test your code's functionality.

The format for the redLine.txt file is a list of the 18 unique stations. You are to read them one by one and generate the corresponding Stations and place them within the Railway.

The format for the riders.txt file is:

```
DPK6S7FCGATASW1B02WP     //riderID
Alewife                  //Start station
Park Street              //Destination station
```

Notes:

- Riders will have unique IDs.
- Riders will only have starting and destination stations listed in redLine.txt

The format for the trains.txt file is:

```
Harvard                  //Train starts at this station
1                        //Train direction (1=south, 0=north)
```

Sample Output

In addition to this file and the assignment's skeleton, you will find a text file containing some sample output for the first 3 cycles of the simulation if it was run on the provided input files. Your output may differ because of where in `simulate()` you log a Station's status and whether you move north or south trains first.

JUnit Tests

In this assignment, it will be very important to be rigorously testing the functionality of your data structures. We have provided some basic tests for your MBTA classes, but you **must write your own tests for the data structures in the provided JUnit test files.**

IMPORTANT NOTES

- For the classes in which specific fields are included, **you must be using these fields and leaving them as** `public`**.**
- Due to the nature of the Railway traversal, in one call of `simulate()`, your southbound trains that start further north should pass freely all the way down until the southernmost station as long as there are no trains in the other stations' southbound train queues. On the other hand, northbound trains that start further south will move only 1 stop in each call of `simulate()` even if there are no other trains in other stations' northbound train queues. This behavior is expected.
- There will be no more than 20 people/trains queued for each station in either direction.

- Your `toString()` methods should be written in so that the final output contains a readable log of the flow of passengers and trains between the stations.
- **You are not allowed to use any Java provided data structures except arrays.**
- **You cannot import any Java libraries other than those that are needed to read text files and to throw exceptions in the Queue class:** `java.util.Scanner, java.io.File, java.io.FileNotFoundException, java.util.NoSuchElementException`

Submission Details

- For every non-test method you write, you must provide a **proper JavaDoc** comment using `@param`, `@return`, etc.
- In addition to regular JavaDoc information, please include each **non-test method's runtime.**
- At the top of **every file** that you submit please leave a comment with the following format. Make sure to include **each of these 6 items.**

```
/**
 * <A description of the class>
 * Known Bugs: <Explanation of known bugs or "None">
 *
 * @author Firstname Lastname
 * <your Brandeis email>
 * <Month Date, Year>
 * COSI 21A PA1
 */

>Start of your class here<
```

- Use the provided skeleton Eclipse project.
- Submit your assignment via Latte as a .zip file by the due date. It should contain your solution in an Eclipse project.
- Your Eclipse project should be named **LastnameFirstname-PA1**. To rename a project in Eclipse, right-click on it and choose Refactor > Rename.
- Name your .zip file **LastnameFirstname-PA1.zip**
- Late submissions will not receive credit.