COMPUTER SCIENCE 21A (SPRING, 2019)
DATA STRUCTURES AND ALGORITHMS

PROGRAMMING ASSIGNMENT 1

Due Sunday, February 17th @ 11:30pm

- Your assignment should be submitted via Latte as a .zip file by the date it is due.
- Name your .zip file LastnameFirstname-PA1.zip
- Each class should have the proper header (see the Additional Notes at the bottom)
- Use the provided template.
- Late submissions will not receive credit.

Overview:

You have been tasked to run a simulation of the Red Line for the MBTA. Using all the knowledge you've learned so far in class, they want you to create a simplified version of the railway, add trains and passengers, and then run the simulation.

Basic Idea:

You will be given three text files with which to generate the stations, the trains, and the passengers. The stations have queues for the trains and the passengers and can only have one train in the station at a time. Trains can hold a limited number of passengers and can only travel in one direction at a time. Your goal is to populate the stations with trains and passengers, then run your simulation and print out a log of all the actions that occur during the simulation, including passenger and train movement.

Requirements:

In the PA1.zip file, you will find 9 classes: these are grouped into two main categories, fundamental data structures and MBTA classes. The data structures are Node.java, Queue.java, and DoubleLinkedList.java. The MBTA classes are Rider.java, Train.java, Station.java, Railway.java, and MBTA.java (client code). The first part of this assignment is to create all of your data structures and test their functionality. The second part of this assignment is to create all of the MBTA classes and implement the simulation. **You should complete your data structures in three to four days and spend the rest on the MBTA part**. The classes included are as follows:

Node.java

- Fields
  - `T element`
  - `Node<T> next`
  - `Node<T> prev`
- Constructor
  - `Node()` – default, creates null node
  - `Node(<T> element, Node<T> next, Node<T> prev)`
- Methods
  - `void setElement(T element)` – set the element
  - `void setNext(Node<T> next)` – set the next node
  - `void setPrev(Node<T> next)` – set the previous node
  - `Node<T> getNext()` – return the next node
  - `Node<T> getPrev()` – return the previous node
  - `T getElement()` – return the element at the node
  - `String toString()` – return the String representation of the element

Queue.java

- Fields
  - `T[] q`
  - `int head`
  - `int tail`
  - `int length`
  - `int size`
- Constructor
  - `Queue(int size)`
- Methods
  - `void enqueue(T element)` – add an element to the tail
  - `T dequeue()` – remove the head element
  - `T peek()` – return the head element without dequeueing
  - `int length()` – return the length (not the size of the array)
  - `String toString()`

DoublyLinkedList.java

- Fields
  - `Node<T> head`
  - `Node<T> tail`
  - `int length`
- Constructor
  - `DoubleLinkedList()`
- Methods
  - `Node<T> getHead()` – return head node
  - `Node<T> getTail()` – return tail node
  - `Node<T> insert(T key)` – insert to end of the list
  - `void delete(T key)` – delete a given key
  - `T get(T key)` – return the element with the key

    - `int length()` – return the length
    - `String toString()`

Rider.java (MBTA)

- Fields
    - `String riderID`
    - `String startingStation`
    - `String destinationStation`
    - `int direction` – 1:south-bound, 0:north-bound
- Constructor
    - `Rider(String riderID, String startingStation, String destinationStation)`
- Methods
    - `String getStarting()` – return the name of the starting station
    - `String getDestination()` – return the name of the destination station
    - `String getRiderID()` – return the riderID
    - `int getDirection()` – return the direction
    - `void setDirection()` – this will be set by the Railway class (below)
    - `String toString()`
    - `boolean equals(Object s)` – this should compare riderIDs

Train.java

- Fields
    - `Rider[] passengers`
    - `String currentStation`
    - `final int TOTAL_PASSENGERS` – the # of passengers that the train can hold, the default is 10 passengers
    - `int direction` – 1:south-bound, 0:north-bound
    - `int passengerIndex` – this should be used to verify if more passengers can be added or not
- Constructor
    - `Train(String currentStation, int direction)`
- Methods
    - `int getDirection()`
    - `void setDirection(int direc)` – you will need to reset the direction when the train reaches the northernmost or southernmost station and turns around
    - `String currentPasengers()` – return the current passenger(s) in the train
    - `boolean addPassenger(Rider r)` – return `true` if a passenger is added successfully, `false` if not. You must make sure that the rider is supposed to be on the train.
    - `boolean canAddPassenger()` – return `true` if a passenger can be added, `false` if not
    - `String removePassenger(Station s)` – This should return a string of all the passengers removed at a station
        - Example:
          7SG7IE6K7J7TZLHCHTZW, Porter

0W3E3HYLZ67MQCA6ACQ8, Porter
3A56AC65CK7D12UCE55Y, Porter

  - `void updateStation(String s)` – this should update the `currentStation` whenever the train is moved between stations
  - `String getStation()` – return the station the train is currently at
  - `String toString()`

Station.java

- Fields
  - `String name` – name of the station
  - `Queue<Rider> northWaiting` – queue for riders waiting to go north
  - `Queue<Rider> southWaiting` – queue for riders waiting to go south
  - `Queue<Train> northBoundTrains` – queue for trains to go north
  - `Queue<Train> southBoundTrains` – queue for trains to go south
- Constructor
  - `Station(String name)`
- Methods
  - `void addWaitingRider(Rider r)` – adds a waiting rider to the appropriate queue, depending on direction
  - `String addTrain(Train t)` – this method takes a `Train` input, figures out the direction it is moving in, removes the passengers that are supposed to disembark at the current station, and then places it into the correct `Train` queue. The string returned is the `Train.toString()`.
  - `Train southBoardTrain()` – this method first will dequeue a train from the south-bound queue (if possible), and while the train has space for passengers, it will add riders from the south-bound rider queue. Finally, the method will return the filled train.
  - `Train northBoardTrain()` – see above.
  - `void moveTrainNorthToSouth()` – dequeue a train from the north-bound queue, change the direction field, and place it into the south-bound queue
  - `void moveTrainSouthToNorth()` – see above.
  - `String stationName()` – return the name of the station
  - `String toString()` – this should return the name and status of the station (how many people are waiting to rider in each direction, how many trains are waiting in each direction)
  - `boolean equals(Object s)` – this should check if the names of the stations are equal

Railway.java

- Fields
  - `DoubleLinkedList<Station> railway`
  - `String[] stations` – this should be initialized to the number of stations in the redLine.txt file (18 stations)
  - `int stationIndex` – helps with initializing the `String[] stations`
- Constructor
  - `Railway()`

- Methods
  - `void addStation(Station s)` – insert the Station into your `DoubleLinkedList<Station> railway` and update the index in `String[] stations` with the name of the station
  - `void addRider(Rider r)` – after receiving the rider, you need to use the `String[] stations` and `calcStation()` to calculate the direction and set the direction of the rider. Then, you should insert the rider into the correct starting station within `railway`.
  - `void addTrain(Train t)` – insert the `Train t` into the proper station within `railway`.
  - `int calcStation(String station1, String station2)` – given two `Station` names, figure out if the person is travelling N to S or S to N. Hint: `stations[0]` should be the northernmost station.
  - `String simulate()`

    This is the "simulation" part of the program. You should have a `String` that is logging the changes in the stations and the trains (if passengers are being dropped off, picked up, etc.)

    Starting from the northernmost station, you will traverse the `DoubleLinkedList<Station> railway` and shift one train from the north-bound and south-bound queue in the respective direction for each station. The order in which the operations should occur is:
    1. Unload any passengers on the trains in the given station
    2. Board any passengers in the given station
    3. Dequeue the fronts of the north-bound and south-bound train queues
    4. Move the trains to the next station in the correct direction
    5. If the station is one of the ones at the end (Alewife and Braintree), then instead of moving the north train/south train more north/south (which is impossible, as they are the last stop), turn them around using the `moveTrainNorthToSouth()` or `moveTrainSouthToNorth()` methods.

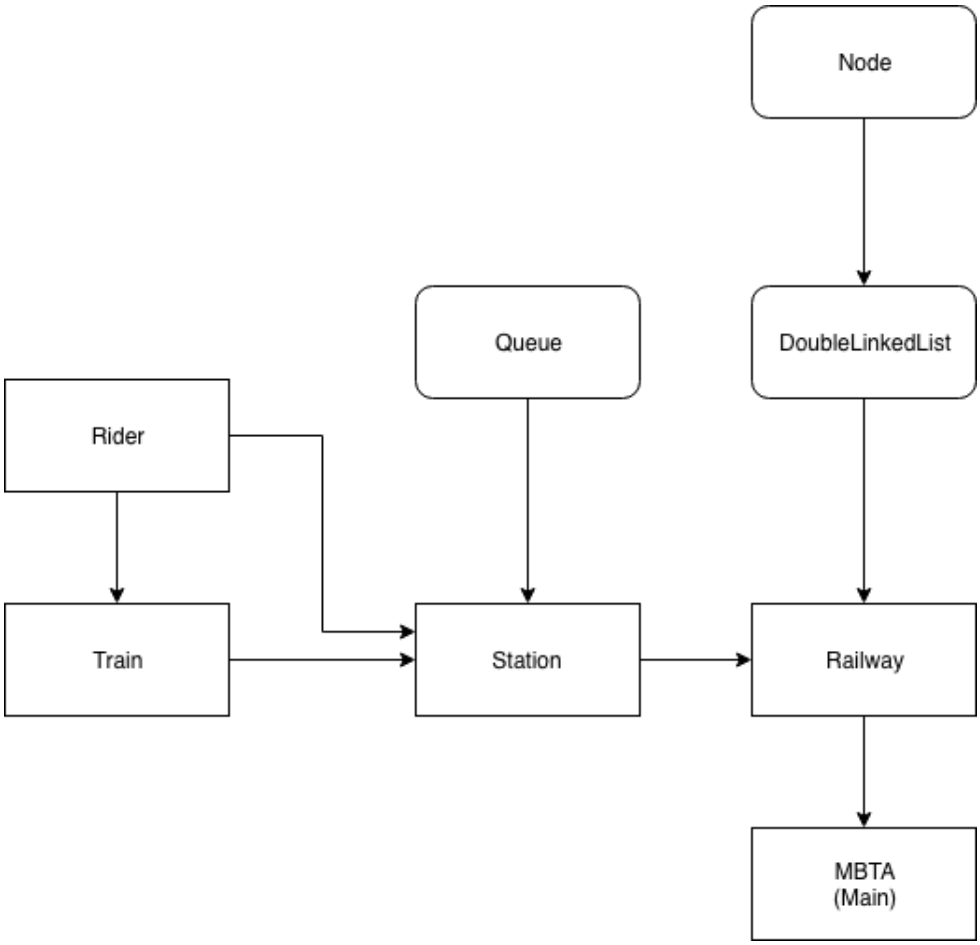
    This should continue down the entire `railway` until the end is reached. Then, you should return the log to print in the Main method. See the bottom of this assignment for a sample output.

  - `String toString()` – return the `DoubleLinkedList<Station> railway` as a String

MBTA.java (Main method)

- Fields
  - `final int HOURS` – this is the number of cycles that you will run the `simulate()` method, one pass through of the system is equal to one cycle
  - `Scanner s`
  - `Railway r`
- Methods

- `main` – Your main method should initiate the Railway, stations, riders, and trains by parsing the given text files and constructing the respective objects. Finally, you should call a method to run the simulation in the Railway class.
- `void initiateRiders(String filename)` – you should construct the `Riders` as you parse the file, and pass them into the `addRider` method of your `Railway r`.
- `void initiateStations(String filename)` – same idea as `initiateRiders`
- `void initateTrains(String filename)` – same idea as `initiateRiders`



Overall Program Structure

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 stations. You are to read them one by one and generate the corresponding Stations and place them within the DLL Railway.

The format for the riders.txt file is:

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

The format for the trains.txt file is:

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

Testing

In this assignment, it will be very important to be testing the functionality of your data structures. For example, this means testing the insertion and deletion of your double linked lists. We have provided some basic test for your MBTA classes, but you will be writing your own tests for the data structures. Your data structure tests should test every single method and also be fairly rigorous and exhaustive, meaning they should test various edge cases (be sure to test your structures when they're empty!)

Grading

The grade breakdown will be as follows:

- Data Structures – 50 points
  - Node – 6 points
  - Node Test – 6 points
  - Doubly Linked List – 10 points
  - Doubly Linked List Test – 6 points
  - Queue – 8 points
  - Queue Test – 4 points
- MBTA Classes – 50 points
  - Rider – 5 points
  - Train – 5 points
  - Station – 15 points
  - Railway – 20 points
  - MBTA – 5 points

IMPORTANT

- In this assignment you must use the fields we have provided, as this is how we are going to evaluate your program's functionality (besides printing the log of the railway system).
- Because of the nature of the Railway traversal, your north-bound trains that start further north (if there are no other trains in the other Station queues) should pass freely all the way down until the southern-most station. This is expected.
- There will be no more than 20 people/trains queued for each station in either direction, therefore don't have to worry about resizing your queues.

- Your `toString()` methods should be written in such a way that makes the final output readable and allows the reader to trace the flow of passengers and trains between the stations.
- You should be commenting **every single method with Java Docs and its runtime!**

Additional Notes

- Properly heading your code – **this must be at the top of your code**

```
//first_name last_name
//email@brandeis.edu
//Month Date, Year
//PA#
/* Explanation of the program/class */
/* Known Bugs: explain bugs/null pointers/etc. */

>Start of your class here<
```

- Java Docs
  - The way to create a Java Doc is to type:
    `/** + return`
  - Depending on your method, the Java Doc may also create additional things, such as `@param`, `@throws`, or `@return`. You should properly label the parameters and returns. See the given code for examples.
  - If you want to read more on Java Docs, you can find that here.