Assignment 2

COMP 250 Winter 2020

posted: Tuesday, Feb. 11 2020
due: Tuesday, Feb. 25, 2020 at 23:59

Learning Objectives

This assignment aims at building on what we have seen in assignment 1 to start storing information and navigating it using complex data structures. Unlike in assignment 1, you will actually have to make decisions on how you want to solve the problems at hand, and while we will see some methods to tackle them in class, most of this discussion will be held at a conceptual level. Your task is to translate your conceptual understanding of those concepts into Java. Watch out: the output of this assignment is not *deterministic*, which means you can get different outcomes between executions. Unlike for assignment 1, or for your assignments in COMP 202, it will not be immediately obvious whether your code works. Make sure to think about this and test your program thoroughly! In terms of specific concepts, this assignment tests all your knowledge on object-oriented programming covered in A1, as well as doubly linked list, navigating between nodes on lists, rearranging lists, and sorting.

General Instructions

- **Submission instructions**
  - Late assignments will be accepted up to 2 days late and will be penalized by 10 points per day. Note that submitting one minute late is the same as submitting 23 hours late. We will deduct 10 points for any student who has to resubmit after the due date (i.e. late) irrespective of the reason, be it wrong file submitted, wrong file format was submitted or any other reason. This policy will hold regardless of whether or not the student can provide proof that the assignment was indeed "done" on time.
  - Don't worry if you realize that you made a mistake after you submitted: you can submit multiple times but **only the latest submission will be evaluated**. We encourage you to submit a first version a few days before the deadline (computer crashes do happen and myCourses may be overloaded during rush hours).
  - Please store all your files in a folder called "Assignment2", zip the folder and submit it to MyCourses. Inside your zipped folder, there must be the following files.
    * `TrainNetwork.java`
    * `TrainLine.java`
    * `TrainStation.java`

Do not submit any other files, especially .class files. Any deviation from these requirements may lead to lost marks

- It does not matter whether or not you create a package to store all these classes. It is up to you to decide whether you'd like to have a package or not.
- You are given class templates to complete. You can only change the code of these classes within the methods containing the comments "YOUR CODE GOES HERE". You cannot change any method headers unless a comment specifies you can, and even then, the only changes you can make are to add exception handling. If you change a method header, we might not be able to grade you and you may receive a grade of zero. Read the comments in the templates carefully.
- Requests to evaluate the assignment manually shall not be entertained, so please make sure that you follow the instruction closely or your code may fail to pass the automatic tests. Note that for this assignment, you are NOT allowed to import any other class (including for example `ArrayList` or `LinkedList`). The template code comes with two imports in the class `TrainLine`. You cannot change them nor add any new ones. **Any failure to comply with these rules will give you an automatic 0.**
- We have included with these instruction a tester class called `TrainRide`, which will help you test your network. You are welcome to add more tests to it. This tester includes a network of 15 stations over three lines, but your code can be evaluated on another network. We will also release (later) a very light MiniTester, which is a mini version of the final tester used to evaluate your assignment and will call every method of the assignment. If your code fails those tests, it means that there is a mistake somewhere. Even if your code passes those tests, it may still contain some errors. We will test your code on a much more challenging set of examples. We therefore highly encourage you to modify the tester class and expand it.
- You will automatically get 0 if your code does not compile.
- Failure to comply with any of these rules will be penalized. If anything is unclear, it is up to you to clarify it by asking either directly a TA during office hours, or on the discussion board on Piazza.

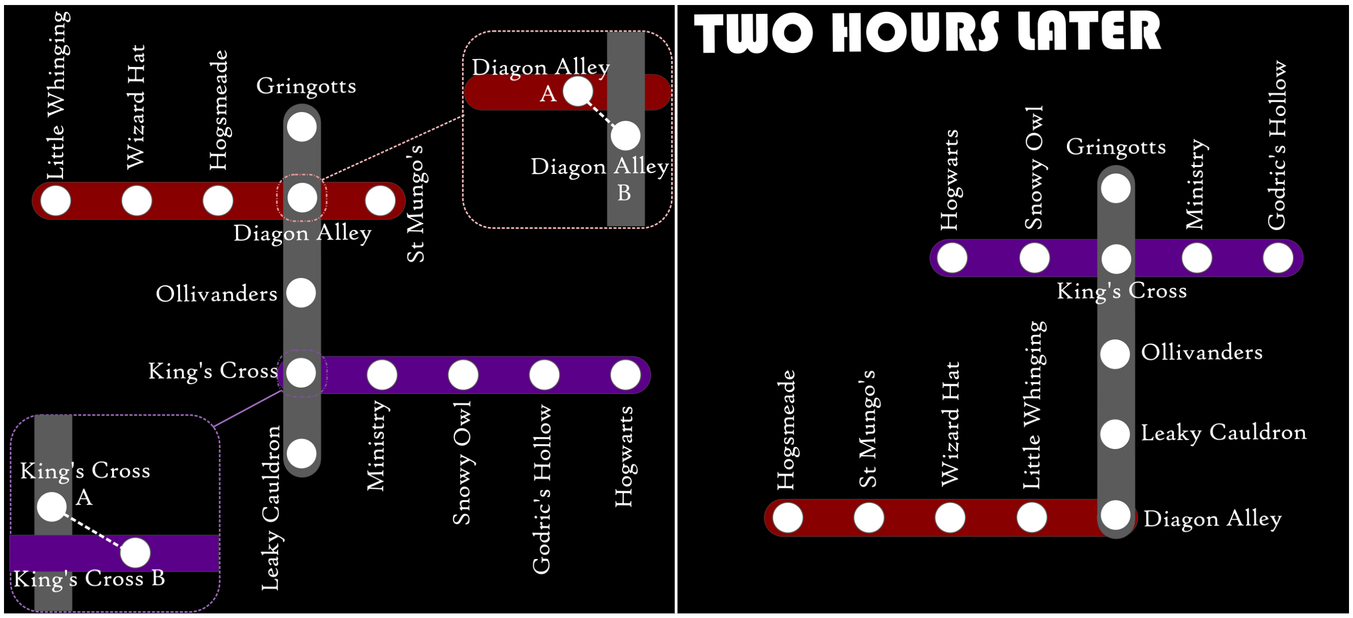

Figure 1: A train network plan (left), before and after a shuffling (right)

A Confusing Train Ride

Due to a significant increase in its student population, especially in its Machine Learning department, the single direct train going to Hogwarts has been replaced by a network of 15 stations distributed over three lines. However, this has revealed to not exactly be an improvement. Just like the staircases within the castle, the stations get bored and, to fight this boredom, rearrange themselves every 2 hours. However, luckily for you, the stations have not yet started switching lines; they always remain on the same line, but the order of the stations within a line gets shuffled.

Your task is to implement this shuffling train network and simulate the trip of an unfortunate traveler. You will be given Java templates to complete. Follow the instructions for each class closely.

Here are a few things you should know about the network:

- Traveling between two stations takes an hour.
- All stations are unique and only exist on one line.
- Transfer is possible between specific stations of different lines, which are indicated by having the same name but followed by a unique letter, identifying a distinct platform. Those remain two distinct stations, but that offer the possibility of traveling from one to the other. Transfer is always two-way; if it is possible to transfer from station A to B, then transfer is possible from station B to A.
- If transferring is possible, the passenger is forced to transfer, unless the passenger would be transferring to a station he just transferred from. for example, if I station from station London - A to London - B, your next step is not to transfer from London - B to London - A.

- Shuffling occurs every two hours. During a shuffling event, the order of the stations within a line changes. The left and right terminals might change. The transfer stations are part of the shuffling, but they stations they transfer to remain unaffected.
- You can assume there is only one traveler on the network at a time, and that every line has a single train which is magically waiting for the traveler at the appropriate transfer station.

You should now be ready to start the assignment, which is divided in four classes, as follows:

[**0 points**] First, you are given a class `TrainStation`. A `TrainStation` encodes the stations of the network. A station is part of a `TrainLine`, which you can imagine as a doubly linked list. It has the following fields:

- `TrainStation` left : the next station on the left
- `TrainStation` right : the next station on the right
- `boolean` rightTerminal : true if the station is at the right end of the line
- `boolean` leftTerminal : true if the station is at the left end of the line
- `String` name : the name of the station.
- `TrainLine` line: the `TrainLine` this station belongs to.
- `boolean` hasConnection: true if the train station connects to another line. This is the only public field.
- `TrainStation` transfersToStation : the station object on the other line, if this transfer exists.
- `TrainLine` transfersToLine : the line object you can transfer to at this station.

All those fields, except `hasConnection`, are private. As such, you are provided with `get` and `set` methods for all the private fields. The class also comes with two constructors, as well as an `equals` method for comparing stations. **Do not modify the TrainStation class.**

[**65 points**] You are also given a class `TrainLine`. A `TrainLine` contains stations that move around. It has the following fields:

- `TrainStation` leftTerminus : the terminal station on the left
- `TrainStation` rightTerminus : the terminal station on the right
- `String` lineName : the name of the line.
- `boolean` goingRight: true if the train is going from the left to the right (assuming node 0 is at the left, and the last node at the right). You can assume there is only one train on the line which magically awaits for you at the transfer station, so the direction of the line is the direction of this train.
- public `TrainStation[]` lineMap : an array of `TrainStation` which encodes the *map* of the line. in that array, the station at index 0 is the left-most station of the line.

A constructor is provided, as well as `equals` method and a helper class `StationNotFoundException`. You are also provided with a method `toString` which converts the lineMap to a String for printing purposes. Finally, a function `shuffleArray` shuffles the lineMap for you.

Your task is to implement the following methods:

- `public int getSize()` : this method returns an integer equal to the number of stations on the line.
- `public TrainStation findStation(String name)` : this method take as input the name of a station, and searches through the line to return the `TrainStation` of this name. All station names are unique.
  - Iterate over the line until you find a station of the right name.
  - If the station is not found, throw a `StationNotFoundException`
- `public TrainStation getNext(TrainStation station)` : takes as input a station and returns the next station of the line.
  - There is only one train on the line, it always goes in the same direction, until it hits a terminal station, then it turns around.
  - Use the `goingRight` field to know in which direction the train is going.
  - if the station is not on this line, throw a `StationNotFoundException`.
  - **You cannot use the lineMap to find the next station.**
- `public TrainStation travelOneStation(TrainStation current, TrainStation previous)` : takes as input the previous and the current station and returns the next station, but while also considering line transfers. Line transfers count as a station change and take the same time as a standard move between stations. So if you are at a station that has the option of transferring, `travelOnestation` should return the station transferred to, and this should count as one time step of one hour.
  - Trains do not like passengers. If you have the opportunity to transfer, you must, unless transferring brings you back to the station you just arrived from (condemning you to an eternal ping-pong between the two).
  - If a valid transfer is available, return the station you transferred to. Otherwise, return the next station on the usual path of the line, computed with `getNext`.
  - If the `current` argument to `travelOneStation` is not on this train line, throw a `StationNotFoundException`
- `public TrainStation[] getLineArray()` : returns an array of the train stations on the line, in order from the left terminal (index 0 of the array) to the right terminal (last index of the array). This array is completely independent from the `lineMap`. The idea is to use this function to update the `lineMap`.
- `public void shuffleLine()` : shuffles the station on the line.

  - You are provided with a `shuffleArray` method, which takes as input an array of TrainStations generated with `getLineArray`, and updates the `lineMap` to a shuffled version of this array.
  - Once you updated the `lineMap` using the provided method, reorder the stations of the line so that their order matches that of the `lineMap`.
  - tips: remember to keep track of the terminal stations, and to update the `TrainStation` objects.

- `public void sortLine()` : sorts the stations of the line in increasing alphabetical order, and updates the `lineMap` (using `getLineArray`). Note that for clarity, we make every station name in `TrainRide` start with a number. Numbers are included in the alphabetical comparison. You can use any of the algorithms covered in class, namely bubble sort, insertion sort, or selection sort. No matter what you use, you need to implement it yourself. Tip: you can make a helper `swap` function to make your life easier.

[**35 points**] You are also given a class `TrainNetwork`. A `TrainNetwork` contains an array of train lines. You are asked to implement the following functions:

- `public TrainLine getLineByName(String lineName)` : this method take as input the name of a Line and returns a line of that name, otherwise throws a LineNotFoundException (helper class provided).
- `public void dance()` : shuffles all the lines using `shuffleLine`.
- `public void undance()` : sorts all the lines using `sortLine`.
- `public int travel(String startStation, String startLine, String endStation, String endLine)` : the key function of the program. It takes as input coordinates for departure and arrival and simulates a trip. Please follow the instructions closely.
  1. Obtain departure station and line objects from the name strings provided as parameters to the method. Store them in the variables `curLine` and `curStation`.
  2. Iterate over the train network starting at the departure station in the provided `while` loop, updating `curStation` and `curLine` . You can change the termination condition.
  3. Keep track of the number of stations visited. This number is equal to the number of hours spent on the train. Assume line transfers always take an hour.
  4. Remember the network `dances` every **two** hours.
  5. At each iteration, keep track of the current station and the previous station. Tip: you do need to keep track of the previous station even if it is not obvious why. Remember how transferring works.
  6. at each iteration, check whether you have arrived at destination by comparing the objective name to the name of the current station.

7. Once you have reached the destination, or after 168 hours of traveling, stop iterating and return an integer equal to the number of hours spent traveling.
8. If the station cannot be found, assume you stayed on the train for a week before giving up, and return 168.
9. Do not throw exceptions in this method.

[**0 points**] You are also provided with a pre-written class `TrainRide`, which instantiates the different objects to a full train network (see the illustration). You can make modifications to the lines or itinerary to test out your code, but the contents of this class will not be graded.

All methods of this assignment are connected and as such, a part of your grade will come from methods that call other methods. This means that one method working incorrectly can lead to a lot of lost marks. Make sure you test your assignment thoroughly by trying the `TrainRide`, which is a syntactic and logical test of every method. Double-check that your results make sense given the rules stated in this handout.

Good luck!