Linked List and Lotteries: List of tasks

Haverford CS 106 - Introduction to Data Structures

Lab 3 (due March 19th)

1 Overview

This lab aims to make you practice the linked list data structures, user-defined classes, and command-line input. The overall task is to automate the process of course lotteries at the Bi-Co. It is recommended that you read the DataDescription.pdf before starting the tasks.

2 General guideline

- Complete the pre-lab which is designed to give you some practice with the command line input and making custom objects `Comparable`.
- Make sure you understand what `toString()` is, how you can override the `toString` method, and use it for debugging. You can revisit the lecture slides, which contain an example on how to override an in-built method.
- A proper understanding of the String methods such as `compareTo()` and `split()` may be helpful.
  - If you use a built-in method, please make sure you understand it well.
  - While reading about the method, we encourage you focus on what it does, what inputs it takes, and what output it gives. Using the methods in an example and seeing their behavior in action is also a good way to become familiar with them!
  - It would be great to understand how it does what it's supposed to do. However, this may be complex in some cases and is not required.
- Test your code as frequently as you can while developing the solution. Become your own "red team" and aim to "break" your code rather than confirming it works. Commonly it is advised to test every method independently and make sure that the method does what it is intended to do.

- We recommend you to read through the data and goal description file (*Data Description* from here) at least twice, write down or mark the critical points, think of the bigger picture, and sketch a roadmap of your understanding.
- In the end, please make sure you run and pass the **unmodified** `Verify`.

3 Tasks

3.1 Data design

Please do not use Java's built-in `LinkedList`. In other words, you should build your own Linked List class. The requirement is to build a *doubly* linked list with *sentinels*. Once you understand how the doubly linked list works, you are welcome to experiment with a singly linked list with the same functionality.

Although generic (holds any type of objects) linked lists have numerous advantages, it is acceptable for this lab to implement a linked list made up of a specific type of nodes (those that reflect a student). Note that there are two other alternatives if you feel very comfortable with nodes, classes, and generics:

- You can instead have two classes: a `Node` class (could be generic), and a `Student` class storing information about a student. This is probably the most theoretically sound way of doing the lab, but it is *not required*.
- You can instead have a single `Student` class that acts as both node and a student data structure (this is similar to the recommended way, but the name of the class is `Student` instead of `Node`.) If this makes more sense to you, this way is fine too.

However, since the nodes in our linked list represent a student, a better name for the class that represents a node for this lab may be `StudentNode` rather than `Node` (or `Student`).

The following sections are suggested pathways to accomplish this lab. However, you are welcome to take different approaches as long as you efficiently achieve the correct output.

3.2 `StudentNode` class

The first step is to create a `StudentNode` class. To create a class, you should first think about the fields the `StudentNode` class should have.

3.2.1 Fields or attributes

- A student may have many attributes but we encourage you to keep it relevant for the Bi-Co lottery data. You may want to include `points` as an attribute.
- In addition to the fields about a student, you should add `prev` and `next` fields to reflect being a node. They may point to a sentinel node or previous and next nodes, respectively.
- While creating the fields, think about what access specifier (private, protected, etc.) you would use with the fields you aim to include.
- Once we create the fields, it's generally good to write corresponding getter and setter methods right away.
- You can come back and make changes as necessary.

3.2.2 Constructor and methods

- Write the appropriate constructor(s) and relevant method(s). You might want to think about what input(s) you want for the constructor *Hint: what data type do Opencsv's readers use to represent a student (CSV row)?*
- You may be interested in adapting `CourseInfo`(included in starter code)'s parsing style for parsing preferences for majors and minors.
- You can come back and make changes as necessary.

3.2.3 Calculating points

- Go through `CourseInfo.java`. Make sure you follow comments while reading the code. Once you get an idea of the `CourseInfo.java`, add an appropriate method in `StudentNode` that takes a `CourseInfo` object as an argument and sets `points` field.
- You might want to either modify the original constructor or add a new constructor that sets the `points` field. The new constructor should have an additional parameter to help calculate the `points`.
- Note that the point system is straightforward and not truly "random" in a lottery sense. One of the optional extensions involves tackling this.

3.2.4 Making `StudentNode` be `Comparable`

Recall the ordering of the nodes described in the *Data Description* document. Make the `StudentNode` class implement the `Comparable` interface to achieve the ordering. If you have done the pre-lab, you will have a better idea.

3.3 The Linked List class

3.3.1 Name

Well, consider the saying that "What's in a name?" In computer science, there is a lot in the name. Therefore, pick an appropriate name for this class, taking purpose of this (Java) class and the context of this lab into account. You may like to follow the steps listed below, but you are welcome to take your own approach. As long as you arrive at the correct solution, it's fine.

3.3.2 Fields

- You may have already realized that deciding and declaring the fields/attributes (instance variables) is a good first step to designing a class. You are encouraged to refer to the textbook to learn more about how to design a class.
- List down the fields that a (doubly) linked list typically has. We recommend looking back at the lecture slides and the book if you are unsure what fields a doubly linked list generally has. Remember that the nodes in the linked list would be of type `StudentNode`.
- Since the linked list is for a course, we may want to store which students the lottery prioritizes. Make an instance variable of type `CourseInfo`, so we can store the course's information.

3.3.3 Constructor

- As you may have guessed, a next good step would be to make a constructor that initializes the attributes that you just added.
- To initialize the field of type `CourseInfo`, we should take in the information about the course's priorities as an argument. If you aren't sure about what this data could look like, check the *Data Description* document and the constructor of `CourseInfo`.

3.3.4 Adding students into the Linked List

- This is one of the major components of this lab. You should think carefully about the input and the algorithm to add students in order.
- If you have not, please read the *Ordering and ties* Section of the Data Description document to understand the node's order.

3.3.5 Read students

- Read the csv file in your `Main` class into your linked list. Recall that you can use the `opencsv` library to read in data from csv files – you may want to look into the difference between `CSVReader` and `CSVReaderHeaderAware`.
- To help with testing and debugging the sorted order, you could create another data file with fewer rows than the original data file. You must eventually test your code on the entire input data once you are sure of the implementation's correctness.

3.4 Enabling the command line look up

- Check the program behaves as expected for the command line arguments. The command-line argument syntax can be found in the *Data Description* document. You may assume that the given command-line arguments always follow the syntax described in the aforementioned section.
- The length of `args` for the basic version will never exceed 4, meaning we will only give either one `print` or `find` flag for each call.
- You should also enable the basic version of `--print` and `--find` functionalities. Feel free to implement first whichever one looks easier to you.

4 Writeup

Please include a write-up in your README that explains your design and algorithms. In particular, address the following prompts:

1. Describe the process you followed to arrive at the solution.
2. Describe your StudentNode class design. What variables and methods do you have in it?
3. How do you keep the linked list in sorted order?

4. Describe your implementation of each part of the optional extension done, if any.

5 Optional tasks (5 extra points)

Your submitted work should **NOT** be affected by the optional extensions. That is, you are free to implement these in **separate** methods and try running them for your own simulation, but your submitted work should **output the result without these optional extensions**.

You may find it especially beneficial to commit and push now to ensure that you have a working version you can revert to if needed. Your final push **must** have no compilation errors.

Describe any implementation approaches you take for the extensions in `README`.

1. If there are more students with the same points than the remaining space on the list, adding the first students by how we ordered the linked list is biased for students with earlier lexicographic initials and class years. Yet, we should never over-enroll or over-waitlist beyond the capacity. Here are a couple of suggestions to overcome this issue:
   (a) We could take the under-enrolling approach. If the remaining spots are not enough for students with tied scores, stop enrolling or waitlisting further.
   (b) We could choose `randomly` from the students tied for the remaining space on the list. Their point value must not be changed in the process.
2. The point system currently is not very random. Add a random point value when calculating each node so that there is some randomness in position.
3. If there are multiple students with the given initials, make `find` give an output for each student with the initials.
4. Enable the `mode` (e.g. `all | enrolled | waitlisted`) functionality of the `print` as described in the *Data Description* document.
5. Allow being given multiple flags in one command line at once. The output does not have to be in a particular order as long as the correct output exists somewhere. *Hint: since flags could be repeated, you might find it beneficial to store them.*
6. Implement being given multiple CSV files (i.e., multiple courses' registrations). These would be given first, before any `print` or `find` flags.

6 Acknowledgement

The lab material has partly been inspired by Housing Algorithm Design Studio and Hiring Algorithms modules from EthicalCS.