Computer Science Fundamentals II Due on Friday, November 15, 2019

Assignment 3

CS1027A *University of Western Ontario*

Learning Outcomes

By completing this assignment, you will gain skills in:

- The solution of problems through the use of queues and
- The implementation of doubly linked lists.

1 Introduction

LACER (London Area Cargo Express Railway) is a new railroad company that is trying to make their mark with cargo deliveries around southern Ontario. They are devoted to making frequent, on-time deliveries around the London area. LACER plans to implement a new idea for connecting train cars that allows for more efficiency in loading, storing, transporting, and unloading of goods. LACER is hiring **you** as Lead Programmer to help coordinate the logistics of their trains. In this assignment you will create a Java program that determines an optimal configuration of a train given a set of train cars.

You are given a series of input files, each listing the train cars that need to be connected in order to form a train. The order of the train cars listed in an input file is not necessarily the order that the train cars will be in after being connected to the train. Each train car carries a specific product (represented as a color) and has a weight capacity (represented as a number), as shown in Figure 1. We have provided code for you that will read this information from the input files and store them in a *queue*.

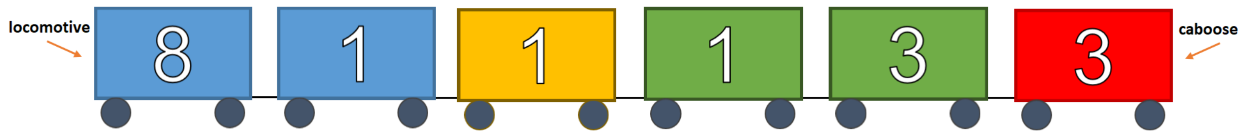

Figure 1: An example of a train consisting of 6 train cars. The front of the train is called the *locomotive* and the back of the train is called the *caboose*. Each train car has a product (color) and a weight capacity (number). For example, the *locomotive* train car has the color blue and the number 8.

The basic idea of an optimal train car ordering is the following: when considering any two adjacent train cars a and b, either a and b share the same color **or** a and b share the same number (see Figure 1).

1.1 Connecting Train Cars

You will have access to a *queue* of train cars that was read from an input file. LACER has created a set of rules to produce an optimal ordering of train cars from this input *queue*:

- The first train car from the input *queue* should become the *locomotive*.
- For each additional train car from the input *queue*, start at the *caboose* and move towards the *locomotive*, trying to connect the train car to the train:
  - If connecting the train car as the new *caboose*, **either** the color **or** the number of the train car must match the old *caboose*.
  - If connecting the train car in the middle of the train, the train car must have a color or number match to **both** the previous train car **and** the next train car.
  - Never add the new train car as the new *locomotive*.

1.2 The Backup Queue

Your program will loop through the given train cars one-by-one and attempt to connect them to the train. **However, sometimes you might find that a train car cannot be connected anywhere in the train using the optimal ordering described above.** You must maintain a backup *queue* of train cars that could not be connected to the train yet, following these rules:

- There are no rules regarding adjacent *TrainCars* in the backup *queue*.
- Train cars from the input *queue* that can not be connected to the train should be added to the backup *queue*.
- Train cars from the backup *queue* should be connected to the train later (if possible).
- Prioritize adding train cars from the backup *queue* (if possible) over the input *queue*.
- Since the backup *queue* is a **queue**, you can only consider the *front* of the backup *queue* when trying to attach train cars.

Your program should stop trying to connect train cars when:

- The input *queue* and backup *queue* are both empty or
- The input *queue* is empty and the *front* train car of the backup *queue* cannot be added.

Note that for each input file, there is exactly 1 correct solution on how to form the train. Additionally, for some of the input files, the correct answer will have some leftover *TrainCars* in the backup *queue*.

2 Classes to Implement

A description of the classes that you need to implement in this assignment is given below. You can implement more classes, if you want. You **cannot** use any static instance variables. You **cannot** use Java's provided *LinkedList* class or *Queue* class or any of the other Java classes from the Java library that implements collections.

2.1 TrainCar.java

This class represents a train car and has the following private instance variables:

- *int number.* This variable stores the number of the train car.
- *String color.* This variable stores the color of the train car.

This class needs to provide the following public methods:

- *TrainCar(int number, String color).* Creates a new train car with the specified number and color.
- *void setNumber(int number).* Sets the number of the train car.
- *void setColor(String Color).* Sets the color of the train car.
- *int getNumber().* Returns the number of the train car.
- *String getColor().* Returns the color of the train car.
- *String toString().* Returns a *String* representation of the train car of the form: $< color, number >$.

You can implement other methods in this class, if you want to, but they must be declared as private.

2.2 Train.java

This class represents a doubly linked list, where each *DoubleNode* in the doubly linked list has a pointer to the previous *DoubleNode* and the next *DoubleNode.* The data contained in each *DoubleNode* is a *TrainCar.* Some methods have already been provided to print out the values of the *Train.*
This class will have the following private instance variable:

- *DoubleNode<TrainCar> locomotive.* This points to the front of the train.
- *DoubleNode<TrainCar> caboose.* This points to the rear of the train.

You must implement the following methods in this class:

- *Train().* Creates a new train where the *locomotive* is *null* and the *caboose* is *null.*
- *boolean addCar(DoubleNode<TrainCar> newCar).* This method attempts to add a *TrainCar* to the *Train* using the rules described in Section 1.1. If the *TrainCar* is successfully added to the *Train*, this method returns true; otherwise, this method returns false.

You can write more methods in this class, if you want to, but they must be declared as private.

2.3 TrainYard.java

This class represents a train yard where multiple train cars might be connected to form a train. Some methods have already been provided to read in each *TrainCar* from an input file a store them in a *queue*. You need to make the following method:

- *void buildTrain().* This method determines whether to connect a *TrainCar* from the input *queue* or the backup *queue* to the *Train* using the rules described in Section 1.2.

Your program must catch any exceptions that are thrown by the provided code. For each exception caught, an appropriate message must be printed. The message must explain what caused the exception to be thrown.

You can write more methods in this class, if you want to, but they must be declared as private.

3 Command Line Arguments

TrainYard.java reads the name of an input file from the command line. You can run the program with the following command:

- *java TrainYard nameOfTrainFile*

where *nameOfTrainFile* is the name of the file containing the *TrainCars.*

To get Eclipse to supply a command line argument to your program open the "Run -> Run Configurations..." menu item. Make sure that the "Java Application ->TrainYard" is the active selection on the left-hand side. Select the "Arguments" tab. Enter the name of the file for the map in the "Program arguments" text box.

4 Classes Provided

You can download several Java classes from the course website's "Assignments" tab that will get you started on this assignment. You are encouraged to study the given code to learn how it works. Below is a description of these classes.

4.1 ArrayQueue.java

This class implements a simple array based *queue* where *front* is fixed at index 0. The public methods that you might use from this class are the following:

- *void enqueue(T element).* This method adds *element* to the *rear* of the *queue.*
- *T dequeue() throws EmptyCollectionException.* This method removes an element from the *front* of the *queue.*
- *T first() throws EmptyCollectionException.* This method returns the element at the *front* of the *queue.*
- *boolean isEmpty().* This method returns *true* if the *queue* is empty; otherwise, this method returns *false.*

- *int size().* This method returns the number of elements stored in the *queue*.

4.2 DoubleNode.java

This class represents a single node in a doubly linked list. The public methods that you might use from this class are the following:

- *DoubleNode<T> getNext().* This method returns the next node in a doubly linked list.
- *DoubleNode<T> getPrevious().* This method returns the previous node in a doubly linked list.
- *void setNext(DoubleNode<T> node).* This method sets the next node in a doubly linked list.
- *void setPrevious(DoubleNode<T> node).* This method sets the previous node in a doubly linked list.
- *T getElement().* This method returns the element contained in a *DoubleNode<T>*.

4.3 Other Classes Provided

EmptyCollectionException.java, QueueADT.java.

5 Sample Input Files Provided

You are given **12** input files that contain lists of *TrainCars* that you can use to test your program. For students using Eclipse, put all of these files in the same folder where the *.classPath* and *.project* files are. If your program can not find the input files, you have not put them in the correct location.

6 Submission

Submit all of your .java files to OWL. **Do not** put the code inline in the textbox. **Do not** submit your *.class* files. If you do this and do not submit your *.java* files your program cannot be marked. **Do not** submit a compressed file with your Java classes (.zip, .rar, .gzip, ...). **Do not** put a ”package” command at the top of your Java classes.

7 Non-Functional Specifications

1. **Assignments are to be done individually and must be your own work.** Software will be used to detect cheating.

2. You can assume that the data in the input files are correct. So no error checking of any type is required.

3. Include comments in your code in **javadoc** format. Add javadoc comments at the beginning of your classes indicating who the author of the code is and giving a brief

description of the class. Add javadoc comments to the methods and instance variables. Read information about javadoc in the second lab for this course.

4. Include comments in each Java file describing key parts of your program. Comments should be grammatically correct, concise, and easy to understand.
5. Use Java coding conventions and good programming techniques. For example:
   (a) Use Java conventions for naming variables and constants.
   (b) Write readible code: good indentation, appropriate white spaces, etc., are required.
6. Make sure your code runs using Eclipe's Java, even if you do not use Eclipse to write your code.

8 Grading Criteria

- Total Marks: [20]
- Functional Specifications:
  - [2] Train.java
  - [1] TrainCar.java
  - [1] TrainYard.java
  - [12] 12 Tests
- Non-Functional Specifications:
  - [1] Meaningful variable names, private instance variables
  - [1] Code readability and indentation
  - [2] Code comments

Assignment files (*Train.java*, *TrainCar.java*, and *TrainYard.java*) are to be submitted to OWL by 11:55pm on November 15th (a Friday).