CSCI-1200 Data Structures — Spring 2018
Homework 5 — Linked Train Cars

In this assignment we examine and manipulate a doubly-linked list structure of different types of train cars: engines, freight cars, passenger cars, dining cars, and sleeping cars. Each engine weighs 150 tons and has power = 3,000 horsepower, freight cars hold different weights (depending on the contents of the car), and all other cars weigh 50 tons. Note that a train can have multiple engines that work together to drive the train, and these engines can be positioned at the front, the back, or even in the middle of the train! Your task for this assignment is to perform a variety of basic train yard operations to connect, disconnect, and reconnect cars into trains in preparation for service. Please read through the entire handout before beginning the assignment.

Train Statistics: Speed and Passenger Comfort

Your first task is to calculate the maximum speed of a specific train (a doubly-linked list of `TrainCars`) on a 2% incline. A very clearly-written tutorial for this real-world problem is available here:

https://web.archive.org/web/20150426114142/http://www.alkrug.vcn.com:80/rrfacts/hp_te.htm

Overly simplified, the formula for the theoretical maximum relates the overall train weight, the combined engine horsepower, and the slope incline:

$$\text{speed } \frac{miles}{hour} = \frac{\text{total horsepower} * 550\frac{lb\text{-}feet}{sec} * 3600\frac{sec}{hr}}{20\frac{lb \text{ tractive effort}}{1\% \text{ grade}} * 2\% \text{ grade} * 5280\frac{feet}{mile} * \text{total weight in tons}}$$

Let's look at an example, using the provided ASCII art printout of a `TrainCar` linked list:

```
     ~~~~
    ||
   ---100   ------101   ------102   ------103   ------104   ------105
 / ENGINE   |       |   |       |   |  dine |   |       |   | sleep |
-oo---oo- + -oo---oo- + -oo---oo- + -oo---oo- + -oo---oo- + -oo---oo-
#cars = 6; total weight = 400; speed on 2% incline = 70.3; avg distance to dining = 1.3; closest engine to sleeper = 5
```

Each `TrainCar` object has a unique ID number displayed in the upper right corner. This number will help us keep track of specific cars when we edit links between train objects. The unmarked cars are simple passenger cars. The statistics printed below the ASCII output detail the total number of cars in the train (including engines), the total weight of the train (in tons), and the calculated maximum speed (in mph) on a 2% incline. You will also need to calculate the final two numbers measuring the comfort level of the passengers riding in the train. First, we would like to know how far the occupants of the passenger cars are from the nearest dining car. Passengers can walk from car to car (forward or backward) in the linked list structure. However, passengers cannot walk through engine or freight cars. The number calculated is the average across all passenger cars. In this example, two passenger cars are 1 car length away from a dining car, and one passenger car is 2 car lengths away. Thus, the average distance to the closest dining car for the above example is $(2 * 1 + 1 * 2)/3 = 1.\overline{3}$. Shorter *average distance to dining* values are more comfortable for passengers. Finally, we measure the quiet comfort of the passengers in the sleeping cars. Since the engines are the noisiest part of the train, we would like to know what is the worst case closest distance to an engine car from any sleeping car in the train. Larger *closest engine to sleeper* values are more comfortable for the passengers. In the event of an "infinte" statistic, you should represent it using a negative value.

Preparing Multiple Freight Trains

The first interesting algorithm we tackle is shipping a large quantity of freight in multiple trains. You will write the function `ShipFreight` that takes in four arguments. The first two arguments are `TrainCar` pointers:

a collection of engines, and a collection of freight cars. The third argument is the required minimum speed to ship this freight and the fourth argument is the maximum number of cars allowed per train. The function returns an STL vector of trains — actually `TrainCar` pointers to the first car / head node in a linked list representing each complete train. Your goal is to ship all of the freight in the fewest number of trains, using the fewest total number of engines. This is a tricky optimization problem with more than one correct answer. For the core assignment you must ensure that your trains achieve the minimum speed and maximum car per train requirements. For extra credit you can improve the algorithm and argue that your solution is indeed optimal (fastest train speed and fewest number of engines and trains) for any provided input.

Here is one possible answer for `ShipFreight` called with 10 engines, 25 freight cars of varying weight, a train length limit of 12 or fewer cars, and a minimum speed of 60 mph. Only 5 engines were necessary and the freight is split into 3 trains. The weight (in tons) of each freight car is displayed in the center of the car.

```
    ~~~~
    ||
   ---106   ------116   ------117   ------118   ------119   ------120   ------121   ------122
 / ENGINE   |   40  |   |   45  |   |   30  |   |   65  |   |   35  |   |   35  |   |   35  |
-oo---oo- + -oo---oo- + -oo---oo- + -oo---oo- + -oo---oo- + -oo---oo- + -oo---oo- + -oo---oo-
#cars = 8, total weight = 435, speed on 2% incline = 64.7
     ~~~~         ~~~~
    ||          ||
   ---107      ---108   ------123   ------124   ------125   ------126   ------127   ------128   ------129   ------130   ------131   ------132
 / ENGINE    / ENGINE   |   50  |   |   85  |   |   50  |   |   45  |   |   40  |   |   85  |   |   85  |   |   45  |   |   45  |   |   65  |
-oo---oo- + -oo---oo- + -oo---oo- + -oo---oo- + -oo---oo- + -oo---oo- + -oo---oo- + -oo---oo- + -oo---oo- + -oo---oo- + -oo---oo- + -oo---oo-
#cars = 12, total weight = 895, speed on 2% incline = 62.8
     ~~~~         ~~~~
    ||          ||
   ---109      ---110   ------133   ------134   ------135   ------136   ------137   ------138   ------139   ------140
 / ENGINE    / ENGINE   |   75  |   |   40  |   |   65  |   |   65  |   |   75  |   |   45  |   |   95  |   |   90  |
-oo---oo- + -oo---oo- + -oo---oo- + -oo---oo- + -oo---oo- + -oo---oo- + -oo---oo- + -oo---oo- + -oo---oo- + -oo---oo-
#cars = 10, total weight = 850, speed on 2% incline = 66.2
```

Managing Train Car Rearrangement

The second significant algorithm you will implement is separating one big passenger train (with two or more engines) into two smaller but similarly fast (and for extra credit, optimally comfortable) trains. The resulting speed of the two smaller trains should be approximately the same as the original large train. First, we'll assume the input train has *exactly* two engines. Here's a sample input train:

```
    ~~~~                                                                                ~~~~
    ||                                                                                  ||
   ---141   ------142   ------143   ------144   ------145   ------146   ------147      ---148   ------149   ------150   ------151
 / ENGINE   |       |   |       |   |  dine |   | sleep |   | sleep |   |       |    / ENGINE   |       |   |  dine |   |       |
-oo---oo- + -oo---oo- + -oo---oo- + -oo---oo- + -oo---oo- + -oo---oo- + -oo---oo- + -oo---oo- + -oo---oo- + -oo---oo- + -oo---oo-
#cars = 11; total weight = 750; speed on 2% incline = 75.0; avg distance to dining = 1.6; closest engine to sleeper = 2
```

After calling the `Separate` function, the original train nodes are split/re-organized into two trains:

```
     ~~~~
    ||
   ---141   ------142   ------143   ------144   ------145
 / ENGINE   |       |   |       |   |  dine |   | sleep |
-oo---oo- + -oo---oo- + -oo---oo- + -oo---oo- + -oo---oo-
#cars = 5; total weight = 350; speed on 2% incline = 80.4; avg distance to dining = 1.5; closest engine to sleeper = 4
                          ~~~~
                         ||
------146   ------147      ---148   ------149   ------150   ------151
| sleep |   |       |    / ENGINE   |       |   |  dine |   |       |
-oo---oo- + -oo---oo- + -oo---oo- + -oo---oo- + -oo---oo- + -oo---oo-
#cars = 6; total weight = 400; speed on 2% incline = 70.3; avg distance to dining = inf; closest engine to sleeper = 2
```

Because the original train in this example had an odd number of non-engine cars, we cannot make the two trains exactly match the speed of the original. Note that not all equal-speed train separations are as simple as cutting one link. Sometimes multiple links must be cut and cars must be pushed around the yard and re-attached in a different configuration. Can you visualize those operations? Draw pictures!

With physical train cars there are real costs associated with each link or unlink of a pair of neighboring cars and with pushing cars around the train yard. The yard needs extra length of track, switches between the tracks, and helper engines and extra staff to complete the operations. We will assume it costs 1 money unit each to: unlink, link, or drag/shift a car one train length of track. We would like to minimize the total

cost of the reconfiguration, but still achieve the best speed-balanced separation of the original train. *Note: There may be multiple different equal-cost solutions for a specific problem! Your code may return any of these solutions.*

Draw at least three different test cases of challenging input to the `Separate` function. Also draw a corresponding valid output configuration for each test case that minimizes the real physical costs associated with the operations. **IMPORTANT: Bring these drawings to Lab 7 – you will need to show these drawings as part of the lab checkpoints.**

First, focus on the implementation of the `Separate` function to handle all input trains with exactly 2 engines. Your code should *not* create any new nodes. Your code should *not* edit the `type` of any `TrainCar` object. We provide a helper function that counts the number of unlink/link operations and the length of track that cars must be dragged or shifted to produce a specific separation result. Your first priority is to separate the original train into two equal (or nearly equal) speed trains. Your second priority is to minimize the cost of unlink, link, and shift operations necessary to complete the separation task. Once the code is working for trains with two engines, you can extend it to work for input trains with more than two engines.

In the above example, note that after separation the passenger comfort values improved in one train but worsened in the other train. In fact, the average distance to dining is now "infinite" because the passengers in car 147 cannot reach the dining car. For extra credit, implement a `SeparateComfort` function that prioritizes creating two equal speed but more comfortable trains (targeting smaller values for *average distance to dining* and/or larger values for *closest engine to sleeper*). The unlink/link/shift cost will likely be greater than the original `Separate` function. Add plenty of test cases to `main.cpp` to demonstrate the success of your extra credit work.

Provided Framework, Implementation Requirements, Hints, & Suggestions

We provide the "`traincar.h`" file with the implementation of the `TrainCar` class. You are not allowed to modify the `traincar.h` file. You will edit the "`traincar_prototypes.h`" file to complete several missing function prototypes that operate on a *train*, a doubly linked list of TrainCar node objects. (Part of your task for this homework is to deduce the exact function prototypes.) You will implement those functions in the "`traincar.cpp`" file. We also provide the "`main.cpp`" file with the ASCII art `PrintTrain` function and a variety of sample tests of the different functions. We use the MersenneTwister pseudo-random number generator (`mtrand.h` and `mtrand.cpp`) — you do not need to understand the details of the `MTRand` class.

You should work on the assignment step-by-step, uncommenting each test case in `main.cpp` as you work. *You are encouraged to use simple recursion to implement some of the functions for this homework.* Compile, test, and fully debug each step before moving on to the next piece of implementation work. You should not modify the provided code except where indicated, as we will be compiling and testing your submitted files with different portions of the instructor solution file. To earn full credit on this homework, your code must also pass the memory error and memory leak checks by Dr. Memory.

Other than the return value of the `ShipFreight` function and test cases in `main.cpp`, you will not use arrays, STL `vector`s, STL `list`s, iterators, or other STL containers in this assignment. Instead, you will be manipulating the low-level custom `TrainCar` objects, and the pointers that link `TrainCar`s to each other.

Submission

Use good coding style and detailed comments when you design and implement your program. You must do this assignment on your own, as described in the "Collaboration Policy & Academic Integrity" handout. If you did discuss this assignment, problem solving techniques, or error messages, etc. with anyone, please list their names in your `README.txt` file.

FINAL NOTE: If you earn 6 points on Test Cases 3, 7, 8, 9 on Submitty by 11:59pm on Wednesday, Feb 28th, you may submit your assignment on Friday, Mar 2nd by 11:59pm without being charged a late day.