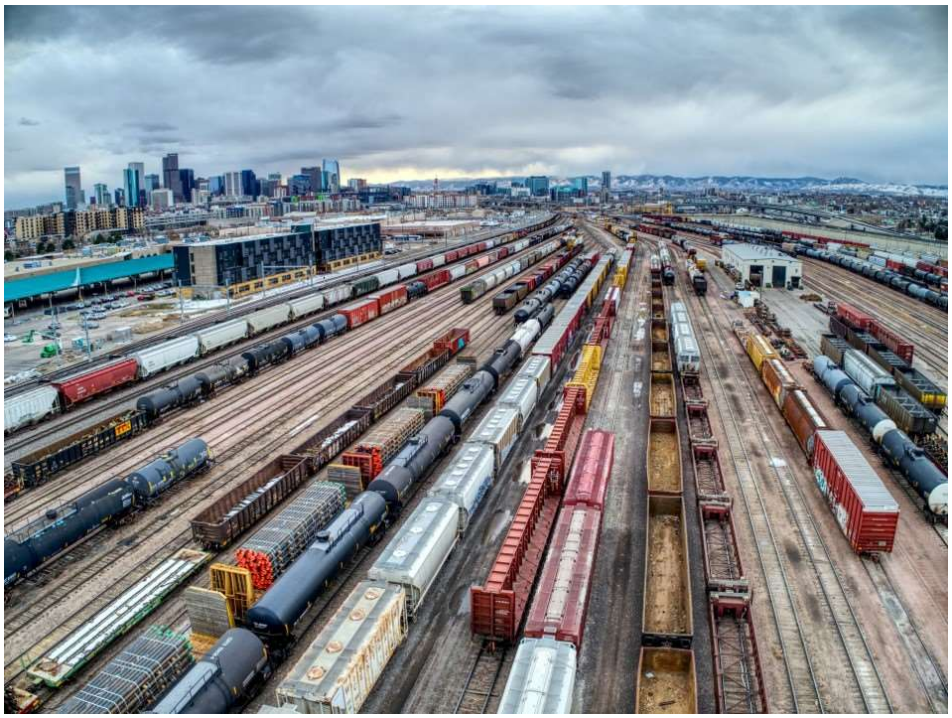
Prepared by: Wallace Mak

COMP2011 Assignment 3: Trains

Introduction

Choo-choo! All aboard! Yes, this assignment is about trains. You will model them with doubly-linked lists and implement various functions which manipulate them. You will practice dynamic memory usage, pointers, and linked lists. It is challenging, so you are **highly recommended to start early**. If you need clarification of the requirements, please feel free to post on the Piazza. However, to avoid cluttering the forum with repeated/trivial questions, please do **read all the given code, webpage description, sample output, and latest FAQ (refresh this page regularly) carefully before you post your questions**. Also please be reminded that we won't debug any student's assignment for the sake of fairness.



About academic integrity

We value academic integrity very highly. Please read the Honor Code section on our course webpage to make sure you understand what is considered as plagiarism and what the penalties are. The following are some of the highlights:

- Do NOT try your "luck" - we use sophisticated plagiarism detection software to find cheaters. We also review codes for potential cases manually.
- The penalty (for **BOTH** the copier and the copiee) is not just getting a zero in your assignment. Please read the Honor Code thoroughly.
- Serious offenders will fail the course immediately, and there may be additional disciplinary actions from the department and university, upto and including expulsion.

Downloads

Download the starting skeleton code HERE. Please download, unzip it, and add all the extracted files to your IDE now, as we will refer to the given source code from time to time in the description below. Your task is to complete 11 functions in todo.cpp, and submit it to ZINC.

Note that this is a C++ program that consists of multiple source files, therefore you should compile all those cpp files according to what you have learned in the labs/lectures.

Here are some rules that you should follow:

- You should NOT modify given.cpp and pa3.h at all.
- You can modify the main function to add your own test cases. However, make sure your submitted todo.cpp can compile with the original unmodified versions of main.cpp, pa3.h, and given.cpp.
- You are NOT allowed to include any additional library.
- You are NOT allowed to use any global variable nor static variable like "static int x".
- You are NOT allowed to use "auto".
- You may use any library function that can be used without including additional libraries. (note: you must make sure it works on ZINC yourself as different environments may have different requirements).
- You may create and use your own helper functions defined in todo.cpp itself.
- You may use any kind of loops and local variables/arrays in general for this assignment.

There is no demo program for this assignment as everything should already be well-defined and sufficiently described. Instead we have provided you with many examples and test cases and their corresponding outputs for you to check your understanding and program. You may find the test cases about them in the sample output section.

Overview

We use a doubly-linked list to represent a train. For that, a node struct is defined in pa3.h:

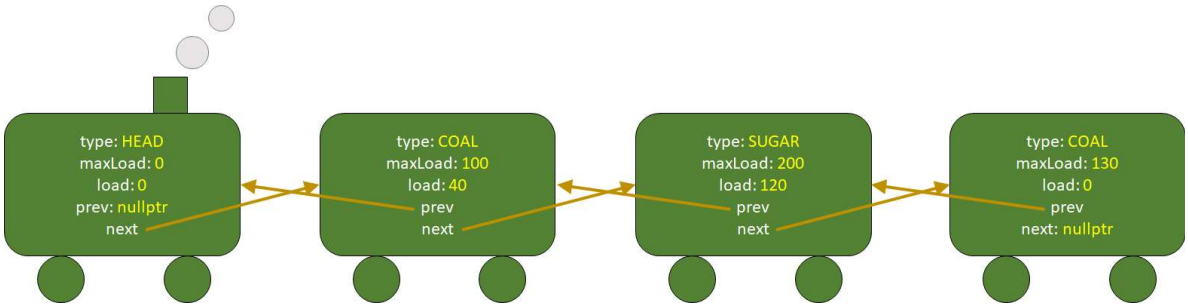
```
struct TrainCar //a node in the doubly-linked list that represents the train - it is therefore a car in the tr
{
    CarType type;
    int maxLoad; //maximum possible load of cargo for this car; always 0 for HEAD
    int load; //current load of cargo for this car; always 0 for HEAD
    TrainCar* prev; //previous pointer
    TrainCar* next; //next pointer
};
```

The `CarType` is defined as follows.

```
enum CarType {HEAD, OIL, COAL, WOOD, STEEL, SUGAR}; //either HEAD for the train head or the type of the cargo
```

Our train always has exactly one HEAD car at the beginning of the list, and then possibly followed by cargo cars which can either carry OIL, COAL, WOOD, STEEL, or SUGAR.

For example, here is an illustration of a train of 4 train cars which is represented by a doubly-linked list of 4 nodes:



(p.s. unfortunately not all programmers are artists... please use your imagination to imagine that it is a beautiful train! :))

- The first car (i.e. the first node) is always a HEAD (train head), and its load and maximum load should always be set to 0 because it is not supposed to carry any cargo at all. As the first node of the linked list, the prev pointer is always set to nullptr. The next pointer points to the next car/node (if any). If the train only consists of the train head and no other cars, then the next pointer should be set to nullptr.
- The second car/node in this example is a COAL car that carries COAL. The maximum load is 100 units while the current load is 40 units. Naturally, the current load can never exceed the maximum load which is at least 1. (except for the head car, as mentioned, of which both the current load and maximum load are always 0) It also cannot be negative. The next pointer points to the third car/node while the prev pointer points to the first car/node, as you can see from the arrows in the picture.
- The third car/node in this example is a SUGAR car that carries SUGAR. The maximum load is 200 units while the current load is 120 units. What the prev and next pointers in a doubly-linked list point to should be obvious to you now - you may also refer to the picture above.
- The fourth car/node in our example is also our last car/node in the train. It currently doesn't have any load of COAL so its current load is 0 while the maximum load is 130 units. As the last car/node, its next pointer is set to nullptr.

The `printTrain` function, which prints the given train, is given to you. You can read its implementation in the given.cpp file. For the example given above, the output would be as follows (2 lines):

```
[HEAD] -> [COAL:40/100] -> [SUGAR:120/200] -> [COAL:0/130]
In reverse: [COAL:0/130] <- [SUGAR:120/200] <- [COAL:40/100] <- [HEAD]
```

The first line shows the train printed from the head node to the last node, while the second line shows the same train printed from the last node to the head node. You are expected to read the given code to understand how that is printed. We will be using this simple text representation in the rest of this assignment description.

Tip: if your program can print the first line but not the second line (the reverse printing of the train), it probably means that some of the prev pointers in your doubly-linked list is not properly set.

Also, in the description below, "position" of a car/node refers to the position of it in the linked list (counting from 0, starts at HEAD). For the example above, the "[HEAD]" node is at position 0, the [COAL:40/100]" node is at position 1, the "[SUGAR:120/200]" node is at position 2, and the "[COAL:0/130]" node is at position 3.

Functions to Implement

This section describes the functions that you need to implement for this assignment in todo.cpp. While we try to be as detailed as possible in the following description, you should check the given test cases in `main` and the corresponding sample output to verify your understanding, especially if you cannot understand the requirements completely just by reading the webpage description alone.

```
TrainCar* createTrainHead();
```

Dynamically create and return a train head car node. As it is currently the only car in the train, set both the prev and next pointers to nullptr. Friendly reminder: check the test cases. For this function, you may refer to the very first test case.

```
bool addCar(TrainCar* head, int position, CarType cargoType, int maxLoad);
```

Attempt to dynamically create and insert a new train car node of the specific `cargoType` with 0 current load and the specified `maxLoad` to an existing train headed by the specified `head` at the specified `position` (refer to the following examples to see how the position is used). It should return true if the addition is successful. The addition only fails when the given `cargoType` is HEAD because we would never add a train head with this function, or the given `position` is not larger than 0 because we won't be adding a new car before the train head, or the given `position` is larger than the length of the train (see examples below), or the given `maxLoad` is zero or negative. When it fails, do nothing to the train and simply return false.

However, **in this and all other functions that you need to implement, you can assume the `head` parameter given always points to a valid train head node of a valid existing train** in all our test cases.

Example 1:

- Original train which has its train head node pointed by `train` (which will be passed to the function as parameter `head`):

```
[HEAD]
In reverse: [HEAD]
```

- After calling `addCar(train, 1, OIL, 100)` which returns true:

```
[HEAD] -> [OIL:0/100]
In reverse: [OIL:0/100] <- [HEAD]
```

Example 2:

- Original train which has its train head node pointed by `train`:

```
[HEAD] -> [OIL:0/100]
In reverse: [OIL:0/100] <- [HEAD]
```

- `addCar(train, 0, SUGAR, 200)` should simply return false.
- `addCar(train, 3, SUGAR, 200)` should simply return false. (length of the train is 2, and 3 is larger than that)
- `addCar(train, 1, SUGAR, 0)` should simply return false.
- After calling `addCar(train, 1, SUGAR, 200)` which returns true:

```
[HEAD] -> [SUGAR:0/200] -> [OIL:0/100]
In reverse: [OIL:0/100] <- [SUGAR:0/200] <- [HEAD]
```

Example 3:

- Original train which has its train head node pointed by `train`:

```
[HEAD] -> [OIL:0/100]
In reverse: [OIL:0/100] <- [HEAD]
```

- After calling `addCar(train, 2, SUGAR, 200)` which returns true:

```
[HEAD] -> [OIL:0/100] -> [SUGAR:0/200]
In reverse: [SUGAR:0/200] <- [OIL:0/100] <- [HEAD]
```

```
bool deleteCar(TrainCar* head, int position);
```

Attempt to remove a car node at the specified `position`. It should return true if the deletion is successful. The deletion only fails when the given `position` is invalid (less than 1 or not larger than the car length, see examples below). When it fails, do nothing to the train and simply return false.

It should go without saying that you need to release any dynamically allocated memory allocated to the car node removed.

Example 1:

- Original train which has its train head node pointed by `train`:

```
[HEAD] -> [OIL:0/100] -> [SUGAR:0/200]
In reverse: [SUGAR:0/200] <- [OIL:0/100] <- [HEAD]
```

- `deleteCar(train, 0)` should simply return false. We won't delete the train head with this function.
- `deleteCar(train, 3)` should simply return false. The length of the train is 3, so there is no node to delete at position 3.
- After calling `deleteCar(train, 1)` which returns true:

```
[HEAD] -> [SUGAR:0/200]
In reverse: [SUGAR:0/200] <- [HEAD]
```

Example 2:

- Original train which has its train head node pointed by `train`:

```
[HEAD] -> [OIL:0/100] -> [SUGAR:0/200]
In reverse: [SUGAR:0/200] <- [OIL:0/100] <- [HEAD]
```

- After calling `deleteCar(train, 2)` which returns true:

```
[HEAD] -> [OIL:0/100]
In reverse: [OIL:0/100] <- [HEAD]
```

`bool swapCar(TrainCar* head, int a, int b);`

Swap train cars at the given positions `a` and `b`. It should return true if the swap is successful. The swap only fails when any of the given positions is invalid, just like the `deleteCar` function. When it fails, do nothing to the train and simply return false.

Swap can also be successfully performed if `a` is the same as `b` as long as they are valid positions, although the train won't apparently change.

Example 1:

- Original train which has its train head node pointed by `train`:

```
[HEAD] -> [OIL:0/100] -> [SUGAR:0/200]
In reverse: [SUGAR:0/200] <- [OIL:0/100] <- [HEAD]
```

- `swapCar(train, 0, 1)` should simply return false.
- `swapCar(train, 1, 3)` should simply return false.
- After calling `swapCar(train, 1, 2)` which returns true:

```
[HEAD] -> [SUGAR:0/200] -> [OIL:0/100]
In reverse: [OIL:0/100] <- [SUGAR:0/200] <- [HEAD]
```

  Note that calling `swapCar(train, 2, 1)` would have the same effect.

`void sortTrain(TrainCar* head, bool ascending);`

Sort the cargo train cars by their current loads ascendingly if `ascending` is true or descendingly if `ascending` is false. The train head should not be included in the sorting as the train head must always be at position 0.

For simplicity, assume no two cargo cars in the same train have the same current load in all test cases that test this function.

Example 1:

- Original train which has its train head node pointed by `train`:

```
[HEAD] -> [SUGAR:30/400] -> [OIL:10/100] -> [COAL:40/200] -> [STEEL:20/300] -> [COAL:5/500]
In reverse: [COAL:5/500] <- [STEEL:20/300] <- [COAL:40/200] <- [OIL:10/100] <- [SUGAR:30/400] <- [HEAD]
```

- After calling `sortTrain(train, true)`:

```
[HEAD] -> [COAL:5/500] -> [OIL:10/100] -> [STEEL:20/300] -> [SUGAR:30/400] -> [COAL:40/200]
In reverse: [COAL:40/200] <- [SUGAR:30/400] <- [STEEL:20/300] <- [OIL:10/100] <- [COAL:5/500] <- [HEAD]
```

- After calling `sortTrain(train, false)`:

```
[HEAD] -> [COAL:40/200] -> [SUGAR:30/400] -> [STEEL:20/300] -> [OIL:10/100] -> [COAL:5/500]
In reverse: [COAL:5/500] <- [OIL:10/100] <- [STEEL:20/300] <- [SUGAR:30/400] <- [COAL:40/200] <- [HEAD]
```

`bool load(TrainCar* head, CarType type, int amount);`

Attempt to load the specified `amount` of cargo of the specified `type` for the train (i.e. add new cargo to the train). It should return true if the loading is successful. The loading only fails if there is not enough free space for the specified cargo type in the train. When it fails, do nothing to the train and simply return false. When there are multiple cars of that cargo type, the loading always tries to fill the cargo cars that are closer to the train head first. Refer to the examples below.

You can assume the `amount` is always positive and the `type` won't be HEAD. (i.e. no checking of those parameters is needed)

Example 1:

- Original train which has its train head node pointed by `train`:

```
[HEAD] -> [SUGAR:30/400] -> [OIL:10/100] -> [COAL:40/200] -> [STEEL:20/300] -> [COAL:5/500]
In reverse: [COAL:5/500] <- [STEEL:20/300] <- [COAL:40/200] <- [OIL:10/100] <- [SUGAR:30/400] <- [HEAD]
```

- `load(train, WOOD, 1)` should simply return false because the train doesn't even have any WOOD cargo car.
- `load(train, SUGAR, 380)` should simply return false because the train can only carry at most 370 additional units of SUGAR.
- After calling `load(train, SUGAR, 300)` which returns true:

```
[HEAD] -> [SUGAR:330/400] -> [OIL:10/100] -> [COAL:40/200] -> [STEEL:20/300] -> [COAL:5/500]
In reverse: [COAL:5/500] <- [STEEL:20/300] <- [COAL:40/200] <- [OIL:10/100] <- [SUGAR:330/400] <- [HEAD]
```

Example 2:

- Original train which has its train head node pointed by `train`:

```
[HEAD] -> [SUGAR:30/400] -> [OIL:10/100] -> [COAL:40/200] -> [STEEL:20/300] -> [COAL:5/500]
In reverse: [COAL:5/500] <- [STEEL:20/300] <- [COAL:40/200] <- [OIL:10/100] <- [SUGAR:30/400] <- [HEAD]
```

- After calling `load(train, COAL, 510)` which returns true:

```
[HEAD] -> [SUGAR:30/400] -> [OIL:10/100] -> [COAL:200/200] -> [STEEL:20/300] -> [COAL:355/500]
In reverse: [COAL:355/500] <- [STEEL:20/300] <- [COAL:200/200] <- [OIL:10/100] <- [SUGAR:30/400] <- [HEAD]
```

  In this example, the first COAL car is filled with 160 units of additional COAL first, then the second COAL car is filled with the remaining 350 units of COAL.

`bool unload(TrainCar* head, CarType type, int amount);`

Attempt to unload the specified `amount` of cargo of the specified `type` for the train (i.e. remove cargo from the train). It should return true if the unloading is successful. The unloading only fails if there is not enough cargo to remove for the specified cargo type in the train. When it fails, do nothing to the train and simply return false. When there are multiple cars of that cargo type, the unloading always tries to unload the cargo cars that are closer to the train tail first. Refer to the examples below.

You can assume the `amount` is always positive and the `type` won't be HEAD.

Example 1:

- Original train which has its train head node pointed by `train`:

```
[HEAD] -> [SUGAR:30/400] -> [OIL:10/100] -> [COAL:40/200] -> [STEEL:20/300] -> [COAL:5/500]
In reverse: [COAL:5/500] <- [STEEL:20/300] <- [COAL:40/200] <- [OIL:10/100] <- [SUGAR:30/400] <- [HEAD]
```

- `unload(train, WOOD, 1)` should simply return false because the train doesn't even have any WOOD cargo car.
- `unload(train, SUGAR, 31)` should simply return false because the train only has 30 units of SUGAR.
- After calling `unload(train, SUGAR, 29)` which returns true:

```
[HEAD] -> [SUGAR:1/400] -> [OIL:10/100] -> [COAL:40/200] -> [STEEL:20/300] -> [COAL:5/500]
In reverse: [COAL:5/500] <- [STEEL:20/300] <- [COAL:40/200] <- [OIL:10/100] <- [SUGAR:1/400] <- [HEAD]
```

Example 2:

- Original train which has its train head node pointed by `train`:

```
[HEAD] -> [SUGAR:30/400] -> [OIL:10/100] -> [COAL:40/200] -> [STEEL:20/300] -> [COAL:5/500]
In reverse: [COAL:5/500] <- [STEEL:20/300] <- [COAL:40/200] <- [OIL:10/100] <- [SUGAR:30/400] <- [HEAD]
```

- After calling `unload(train, COAL, 15)` which returns true:

```
[HEAD] -> [SUGAR:30/400] -> [OIL:10/100] -> [COAL:30/200] -> [STEEL:20/300] -> [COAL:0/500]
In reverse: [COAL:0/500] <- [STEEL:20/300] <- [COAL:30/200] <- [OIL:10/100] <- [SUGAR:30/400] <- [HEAD]
```

In this example, 5 units of COAL are removed from the the last COAL car first, then 10 units of COAL are remove from the second last COAL car.

`void printCargoStats(const TrainCar* head);`

Print the one-line cargo stats to the console terminal with cout. The exact format required will be explained with the following example.

You can assume the train has at least one cargo car in all the test cases that test this function.

Example 1:

- Original train which has its train head node pointed by `train`:

```
[HEAD] -> [SUGAR:30/400] -> [OIL:10/100] -> [COAL:40/200] -> [STEEL:0/300] -> [COAL:5/500]
In reverse: [COAL:5/500] <- [STEEL:0/300] <- [COAL:40/200] <- [OIL:10/100] <- [SUGAR:30/400] <- [HEAD]
```

- `printCargoStats(train)` should print the following line to the console terminal with cout:

```
SUGAR:30/400,OIL:10/100,COAL:45/700,STEEL:0/300
```

As you can see, it simply tells the user, for each cargo type that exists in the train, the total current load and the total maximum load. For example, there are in total 45 units of COAL while at most 700 units can be carried, hence we have "COAL:45/700".

The printing order of the cargo types depend on their first appearances in the train. In this example, since the SUGAR car appears first, the stats for SUGAR are printed first. The stats for OIL are printed second as OIL appears the second in the train. Finally, because COAL first appears earlier than STEEL in the train, the stats for COAL are printed before STEEL. Stats for WOOD are not printed because there is no WOOD car at all in this example.

The stats for different cargo types are separated by exactly one comma ",". There should be no empty spaces at all in the whole line. There is also exactly one new-line character `endl` at the end of the line.

Following all the rules listed above, there can only be one version of the printed line. Any difference in your output will be considered as incorrect. Since all grading will be automatically done by output comparison and we will not allow any student to modify their submitted code after the submission deadline, please check carefully if your code can output the stats with the exact format as described. You can compare your output with our sample output for the given test cases which test this function, or run them on ZINC to make sure (friendly reminder: ZINC server will be very busy when the deadline approaches, please do not expect you can get the grading reports for checking quickly during that time; probably you will need to manually compare your output with our sample output carefully then if you start late).

`void divide(const TrainCar* head, TrainCar* results[CARGO_TYPE_COUNT]);`

Create new trains of single cargo types from the given train. The cars in the new trains must be dynamically created. (i.e. in the new trains, do not just put/point to any existing cars of the given train - you need to create new copies of the cars) The original train should not be modified. We will use the following examples to explain what you need to do.

You can assume the train has at least one cargo car in all the test cases that test this function.

Example 1:

- Original train which has its train head node pointed by `train`:

```
[HEAD] -> [SUGAR:30/400] -> [OIL:10/100] -> [COAL:40/200] -> [STEEL:0/300] -> [COAL:5/500]
In reverse: [COAL:5/500] <- [STEEL:0/300] <- [COAL:40/200] <- [OIL:10/100] <- [SUGAR:30/400] <- [HEAD]
```

- `divide(train, results)` will perform the following:
  1. Find out that how many are cargo types are there in total for this train. For this example, there are 4, in this order according to their first appearances in the train: SUGAR, OIL, COAL, STEEL
  2. Generate a new train for each of the cargo types, in the order determined in the previous step. Therefore, for this example, there should be 4 new trains created for this example: a SUGAR train first, then an OIL train, then a COAL train, and finally a STEEL train.
  3. For the SUGAR (the first cargo type) train, you will need to create a new train head, and then add a new SUGAR car of the same load and maximum load as the only SUGAR car we have in the given train:

     ```
     [HEAD] -> [SUGAR:30/400]
     In reverse: [SUGAR:30/400] <- [HEAD]
     ```

     Make the first result pointer (results[0]) point to the head car node of it.
  4. For the OIL (the second cargo type) train, you will need to create a new train head, and then add a new OIL car of the same load and maximum load as the only OIL car we have in the given train:

```
[HEAD] -> [OIL:10/100]
In reverse: [OIL:10/100] <- [HEAD]
```

Make the second result pointer (results[1]) point to the head car node of it.

5. For the COAL (the third cargo type) train, you will need to create a new train head, and then add two new COAL cars of the same loads and maximum loads as the two OIL cars we have in the given train. Note that we keep their relative ordering in the new train as you can see:

```
[HEAD] -> [COAL:40/200] -> [COAL:5/500]
In reverse: [COAL:5/500] <- [COAL:40/200] <- [HEAD]
```

Make the third result pointer (results[2]) point to the head car node of it.

6. For the STEEL (the fourth cargo type) train, you will need to create a new train head, and then add a new STEEL car of the same load and maximum load as the only STEEL car we have in the given train:

```
[HEAD] -> [STEEL:0/300]
In reverse: [STEEL:0/300] <- [HEAD]
```

Make the fourth result pointer (results[3]) point to the head car node of it.

7. Set all the remaining result pointers to nullptr. Since there are only 4 new trains created in this example, only the first 4 result pointers will point to the new trains, you should therefore set results[4] to nullptr. (note: CARGO_TYPE_COUNT is 5, so there are only 5 results pointers)

- If we print the results array with the following code which is also used in our test cases as you can see in main.cpp:

```
cout << "Divide results:" << endl;
for(int i=0; i<CARGO_TYPE_COUNT; i++)
{
    cout << "results[" << i << "]:" << endl;
    if(results[i] == nullptr)
        cout << "nullptr" << endl;
    else
        printTrain(results[i]);
}
```

The following would be the output:

```
Divide results:
results[0]:
[HEAD] -> [SUGAR:30/400]
In reverse: [SUGAR:30/400] <- [HEAD]
results[1]:
[HEAD] -> [OIL:10/100]
In reverse: [OIL:10/100] <- [HEAD]
results[2]:
[HEAD] -> [COAL:40/200] -> [COAL:5/500]
In reverse: [COAL:5/500] <- [COAL:40/200] <- [HEAD]
results[3]:
[HEAD] -> [STEEL:0/300]
In reverse: [STEEL:0/300] <- [HEAD]
results[4]:
nullptr
```

As usual, you may use the other examples in the given test cases in main.cpp to verify/help with your understanding.

`TrainCar* optimizeForMaximumPossibleCargos(const TrainCar* head, int upperBound);`

Find a subset of the cargo cars which can carry the maximum amount of cargo (type doesn't matter for this part) that is not more than the given `upperBound`, then create a new train with those cargo cars and return it. The cars in the returned train must be dynamically created. (i.e. in the returned train, do not just put/point to any existing cars of the given train - you need to create new copies of the cars) The original train should not be modified. We will use the following examples to explain what you need to do.

You can assume the train has at least one cargo car in all the test cases that test this function.

For simplicity, you can assume there will only be exactly one solution in each of the test cases for this function.

Example 1:

- Original train which has its train head node pointed by `train`:

```
[HEAD] -> [SUGAR:30/400] -> [COAL:10/100] -> [COAL:25/200]
In reverse: [COAL:25/200] <- [COAL:10/100] <- [SUGAR:30/400] <- [HEAD]
```

- There are 8 possible trains that can be generated with those cargo cars (we always keep the same relative ordering among them, so there can only be exactly 8 of them in this example):
  1. ```
     [HEAD] -> [SUGAR:30/400] -> [COAL:10/100] -> [COAL:25/200]
     In reverse: [COAL:25/200] <- [COAL:10/100] <- [SUGAR:30/400] <- [HEAD]
     ```
     which carries 65 units of cargo in total.
  2. ```
     [HEAD] -> [SUGAR:30/400] -> [COAL:10/100]
     In reverse: [COAL:10/100] <- [SUGAR:30/400] <- [HEAD]
     ```
     which carries 40 units of cargo in total.
  3. ```
     [HEAD] -> [SUGAR:30/400] -> [COAL:25/200]
     In reverse: [COAL:25/200] <- [SUGAR:30/400] <- [HEAD]
     ```
     which carries 55 units of cargo in total.
  4. ```
     [HEAD] -> [COAL:10/100] -> [COAL:25/200]
     In reverse: [COAL:25/200] <- [COAL:10/100] <- [HEAD]
     ```
     which carries 35 units of cargo in total.
  5. ```
     [HEAD] -> [SUGAR:30/400]
     In reverse: [SUGAR:30/400] <- [HEAD]
     ```
     which carries 30 units of cargo in total.
  6. ```
     [HEAD] -> [COAL:10/100]
     In reverse: [COAL:10/100] <- [HEAD]
     ```
     which carries 10 units of cargo in total.
  7. ```
     [HEAD] -> [COAL:25/200]
     In reverse: [COAL:25/200] <- [HEAD]
     ```
     which carries 25 units of cargo in total.
  8. ```
     [HEAD]
     In reverse: [HEAD]
     ```
     which carries 0 unit of cargo in total.
- `optimizeForMaximumPossibleCargos(train, 100)` should create and return a new train:

  ```
  [HEAD] -> [SUGAR:30/400] -> [COAL:10/100] -> [COAL:25/200]
  In reverse: [COAL:25/200] <- [COAL:10/100] <- [SUGAR:30/400] <- [HEAD]
  ```
  which carries 65 units of cargo in total.
- `optimizeForMaximumPossibleCargos(train, 35)` should create and return a new train:

  ```
  [HEAD] -> [COAL:10/100] -> [COAL:25/200]
  In reverse: [COAL:25/200] <- [COAL:10/100] <- [HEAD]
  ```
  which carries 35 units of cargo in total.
- `optimizeForMaximumPossibleCargos(train, 36)` should create and return a new train:

  ```
  [HEAD] -> [COAL:10/100] -> [COAL:25/200]
  In reverse: [COAL:25/200] <- [COAL:10/100] <- [HEAD]
  ```
  which carries 35 units of cargo in total.
- `optimizeForMaximumPossibleCargos(train, 40)` should create and return a new train:

```
[HEAD] -> [SUGAR:30/400] -> [COAL:10/100]
In reverse: [COAL:10/100] <- [SUGAR:30/400] <- [HEAD]
```

which carries 40 units of cargo in total.

- `optimizeForMaximumPossibleCargos(train, 5)` should create and return a new train:

```
[HEAD]
In reverse: [HEAD]
```

which carries 0 unit of cargo in total.

Note that you should not assume the maximum number of cars the given train can have, since the train is a linked list.

```
void deallocateTrain(TrainCar* head);
```

Release/deallocate all the memory allocated for all the cars in the given train.

Tip: If this function is crashing your program for all the test cases on ZINC, you may consider leaving its implementation empty so that you can get the majority of points from test cases that don't check for memory leak.

Sample Output

Your finished program should produce the same output as our sample output for the test cases. User input, if any, is omitted in the files. You can run them locally or run them online on ZINC anytime. Please note that sample output, naturally, does not show all possible cases. It is part of the assessment for you to design your own test cases to test your program. **Be reminded to remove any debugging message** that you might have added before submitting your final submission.

There are 22 given test cases of which the code can be found in the given main function. These 22 test cases are first run without any memory leak checking (they are numbered #1 - #22 on ZINC). Then, the same 22 test cases will be run again, in the same order, with memory leak checking (those will be numbered #23 - #44 on ZINC). For example, test case #24 on ZINC is actually the given test case 2 (in the given main function) run with memory leak checking.

Each of the test cases run without memory leak checking (i.e., #1 - #22 on ZINC) is worth 1 mark. The second run of each test case with memory leak checking (i.e., #23 - #44 on ZINC) is worth 0.2 mark. The maximum score you can get on ZINC, before the deadline, will therefore be 22*(1+0.2) = 26.4.

About memory leak and other potential errors
Memory leak checking is done via the `-fsanitize=address,leak,undefined` option (related documentation here) of the g++ compiler on Linux (it won't work on Windows for the versions we have tested). Check the "Errors" tab (next to "Your Output" tab in the test case details popup) for errors such as memory leak. Other errors/bugs such as out-of-bounds, use-after-free bugs, and some undefined-behavior-related bugs may also be detected. You will get 0 mark for the test case if there is any error there. Note that if your program has no errors detected by the sanitizers, then the "Errors" tab may not appear. If you wish to check for memory leak yourself using the same options, you may follow our Checking for memory leak yourself guide. It is known that the memory leak detection tools may not be able to detect all the leaks a program has - however for automatic grading purpose we will only deduct marks for leaks that it can detect.

After the deadline
We will have 22 additional test cases which won't be revealed to you before the deadline. Together with the 22 given test cases, there will then be 44 test cases used to give you the final assignment grade. All 44 test cases will be run two times as well: once without memory leak checking and once with memory leak checking. The assignment total will therefore be 44*(1+0.2) = 52.8. Details will be provided in the marking scheme which will be released after the deadline.

Submission

Deadline

23:59:00, Nov 30th (Tue)

Please submit the **todo.cpp** only to ZINC. For assignment 3, we will check for memory leak in your program in some of the test cases. ZINC usage instructions can be found here.

Notes:

- You may submit your file multiple times, but only the last submission will be graded. **You do NOT get to choose which version we grade.** If you submit after the deadline, late penalty will be applied according to the submission time of your last submission.
- Submit early to avoid any last-minute problem. Only ZINC submissions will be accepted.
- The ZINC server will be very busy on the last day especially in the last few hours, so you should expect you would get the grading result report not-very-quickly. However, as long as your submission is successful, we would grade your latest submission with all test cases after the deadline.
- In the grading report, pay attention to various errors reported. For example, **under the "make" section, if you see a red cross, click on the STDERR tab to see the compilation errors.** You must fix those before you can see any program output for the test cases

below.
- Make sure you submit the correct file yourself. You can download your own file back from ZINC to verify. Again, **we only grade what you uploaded last to ZINC**.

Compilation Requirement

It is **required** that your submissions can be compiled and run successfully in our online autograder ZINC. If we cannot even compile your work, it won't be graded. Therefore, for parts that you cannot finish, just put in dummy/empty implementation so that your whole program can be compiled for ZINC to grade the other parts that you have done.

Late submission policy

There will be a penalty of -1 point (applied to the assignment score which will be scaled to have a maximum of 100 point) for every minute you are late. For instance, since the deadline of the assignment is 23:59:00 on Nov 30th, if you submit your solution at 1:00:00 on Dec 1st, there will be a penalty of -61 points for your assignment. However, the lowest grade you may get from an assignment is zero: any negative score after the deduction due to late penalty (and any other penalties) will be reset to zero.

FAQ

You should check this section a day before the submission deadline to make sure you haven't missed out any clarification, even if you have already submitted your work to ZINC by then.

- Q: My code doesn't work / there is an error, here is the code, can you help me to fix it?
- A: As the assignment is a major course assessment, to be fair, you are supposed to work on it on your own and we should not finish the tasks for you. We might provide some very general hints to you, but we shall not fix the problem or debug for you.

- Q: My program gives the correct output on my computer, but it gives a different one on ZINC. What may be the cause?
- A: Usually inconsistent strange result (on different machines/platforms, or even different runs on the same machine) is due to relying on uninitialized hence garbage values, missing return statements, accessing out-of-bound array elements, improper use of dynamic memory, or relying on library functions that might be implemented differently on different platforms (such as pow() in cmath).
  You may find a list of common causes and tips on debugging in the notes here.
  In this particular PA, it is probably related to misuse of dynamic memory. Good luck with bug hunting!