ECCS 2671: Data Structures and Algorithms I

Fall Semester 2025

Project 1: Doubly Linked List Implementation

Due: Monday, September 29th, 11:59 PM

This is a pairwise group assignment (please speak to us if considering working alone). In order to complete this assignment, be sure to accept the GitHub Classroom assignment, clone the repo, and produce a local copy of the cloned repo on your computer. Decide who wants to complete the member functions to implement for Student 1 and who wants to complete the member functions in the list of Student 2 (we made sure to balance them so there is not a significant difference between the two) and record your names in the .cpp file that is submitted and commit the change. Branch the repo using either Student-1 or Student-2 as part of the branch name to distinguish the branch, and commit to your branch once your member functions are complete (you are encouraged to divide your work into multiple commits, for example, after you have successfully tested a couple of member functions and know you have a working partial solution), and then perform a pull request for your teammate's review prior to each merging the code. The branches must be merged back to the main branch before the final push is done, which submits the assignment. Please refer to the GitHub guide for branching and pull requests. The breakdown of your individual grade will be 90% for your individual member function implementations and 10% for following the branching and pull requests procedures, and overall class functionality in the code. This homework is only submitted through GitHub by pushing the commits your team make. The deadline is **Monday, September 29**.

I want you to be successful in this course, and there are many opportunities for assistance. I encourage questions on homework, programming assignments, projects, and even during exams (in those circumstances the amount of clarity given in the answer may not be complete, but it still may be helpful!). Achieving grades that do not represent your own work is not true success. Stress, lack of sleep, demands beyond being a student, and completing assignments at the last minute can contribute to poor decision-making. If you find yourself tempted to cross the line, I encourage you to speak with me so we can discuss ways to enhance your learning experience and make you truly successful.

Project Objectives:

1. Learn how to implement doubly linked list.
2. Improve C++ programming.
3. Learn to determine an appropriate algorithmic approach to a problem.
4. Learn to develop computer programs that are implementations of various algorithms and use multiple data structures

---

Linked lists are a linear data structure that can be used for many applications, including as an implementation for other data structures (as we will see). A ***train*** is an excellent analogy for a linked list because it's a linear collection of cars connected in a sequence. A **doubly linked list** fits this model perfectly, as each car (CarNode) is connected to both the car in front of it and the one behind it.

1. **Dynamic** – Trains can have cars added or removed easily, just as cars can be added to or deleted from a playlist. This process is memory efficient. When adding to the end of the list (the tail), it only requires constant time to add a car. Nodes in a linked list can be reordered easily as well.

2. **Low Overhead** – Linked lists do not have a lot of additional memory overhead required for maintaining the data structure.

3. **Simple** – Working with linked lists is easier than some other more complex data structures.

Most modern train yard control systems need to assemble and reconfigure trains made up of many different car - locomotives, passenger coaches, dining cars, cargo cars, and cabooses. When working with a train composition, it is common to need both "next car" and "previous car" operations: for example, to traverse the train from the engine to the caboose, or in reverse. This needs for bidirectional traversal maps perfectly onto the two-way links of a doubly linked list, where each car (node) points to both its next and previous neighbor.

One practical constraint of using a linked list to model a train is the lack of random access. Unlike array-based models—such as a static list of cars indexed by position - it is not possible to instantly access, say, the 5th car by index; the system must step car-by-car through the linked list to reach the desired position.
The train composition must maintain a consistent and meaningful order—typically, the first car (the engine or lead car) is at the head of the list, followed by cars in their coupled order, until the last car (commonly the caboose) is at the tail. Each car should have an associated sequence number (e.g., Car 1, Car 2, ..., Car n) reflecting its place on the train. This numbering and the overall sequence must be updated carefully every time cars are added or removed to ensure that the train's structure remains correct and consistent for operational and reporting needs.

Implementation Task:
You are tasked with implementing a `TrainCompositionManager` class that manages a train as a doubly linked list of car nodes. Each node represents a car, containing relevant details such as its unique car number, type, and weight, as well as pointers to the next and previous cars in the sequence.

Data Structure
Train `CarNode` should include:

- **`int carNumber`**: A unique integer representing the car's position in the train.
- **`string carName`**: A specific identifier for the car (e.g., "Silver Streak Car", "Grand View Diner").
- **`string carType`**: `(e.g., Locomotive, Passenger, Cargo, Dining, Caboose)`: A general category for the car (e.g., "Passenger"). This is for classification and is different from the `carName`.
- **`double weightTons`**: The weight of the train car. This value must be between a lower and upper limit.
  - **Lower limit**: **10 tons** (to ensure stability).
  - **Upper limit**: **143 tons** (representing the standard gross rail load of 286,000 pounds).
- **`string manufacturerName`**: The name of the company that built the car. This can be an empty string if the manufacturer is unknown.
- **`CarNode* next`**: A pointer to the next `CarNode` node in the sequence.
- **`CarNode* prev`**: A pointer to the previous `CarNode` node in the sequence.

Your `TrainCompositionManager` will provide a suite of operations to manipulate and inspect the train:

The following member functions should be implemented by **Student 1**:

```
1. CarNode* getCarNode(const string carName);
2. void addCar(const string addedCarName, const string carType, const double weightTons);
3. void addCar(const string addedCarName, const int carNumber, const string carType,
   const double weightTons);
4. void deleteCar(const string deletedCarName);
5. int getCarNum(const string searchedCarName);
6. vector<string> getAllCarNamesByType(const string carType);
```

The following member functions should be implemented by **Student 2**:

```
1. CarNode* getCarNode(const int carNumber);
```

2. `void addCar(const string addedCarName, const int carNumber, const string carType, const double weightTons, const string manufacturerName);`
3. `void deleteLastCar();`
4. `void deleteCar(const int carNumInList);`
5. `string getCarName(const int carNumInList);`
6. `int getCarTypeCount(const string carType);`

There are several features fully implemented for you that do not require any modification, including the class body, main (although you may choose to comment out some of the commands in main to debug member functions as you implement them), the constructor, destructor, a simple getter function to return the number of cars in the Train, and a friend function of our class that overloads the insertion operator to print out the contents in the Train object. This example of operator overloading in C++ uses the concept of a friend function, which can access private (and protected) members of a class even though it is not a member function of the class. It is needed so the output stream objects play nicely with our linked list class objects. Operator overloading is not part of this course and you will not be tested on this topic; it is just for your information. Feel free to add examples to the main to test your code more comprehensively.

It is highly recommended each student tests their member functions as they go, using cout statements to view the state of certain variables as needed. Comment out necessary statements in main while debugging.

For **Student 1**, a brief description of what each member function should do is given below:

1. `CarNode* getCarNode(const string carName);`

   Traverses the linked list to find the first car with the given `carName` (string) and returns a pointer to the node if present, or nullptr otherwise. It's a **private helper function**.

2. `void addCar(const string addedCarName, const string carType, const double weightTons);`

   Adds a new `carNode` with the given `addedCarName`, `carType, and weightTons` at the head of the train. All other car numbers should be incremented since this new car is now the head (car number 1). Also be sure to update the number of cars on the list. Since no manufacturer is given, let it be an empty string.

3. `void addCar(const string addedCarName, const int carNumber, const string carType, const double weightTons);`

   Adds a new `carNode` with the given parameters at the position specified by `carNumber` (if `carNumber` is valid given the list, between 1 and the number of cars). All Cars at `carNumber` and beyond (because the new car will supplant the place of the old car at `carNumber`) should have their car numbers incremented and all pointers should be handled correctly. Note that if `carNumber` < 2, place the car at the head. If `carNumber` > `numCars`, it should be placed at the tail. Since no manufacturer is given, let it be an empty string. Be sure that your code can handle all special cases (empty list, single node list, placing at head, placing at tail, etc.).

4. `void deleteCar(const string deletedCarName);`

   Deletes a `carNode` based on the car name, if present. If the car isn't in the train, do nothing. Be sure to handle all special cases (empty list, deleting the only node in a single node list, deleting at the head, deleting at the tail, deleting in the middle, etc.). Be sure to handle the pointers around the node to be deleted correctly (no memory leaks or dangling pointers), and be sure to decrement the appropriate car numbers after deleting the node.

5. `int getCarNum(const string searchedCarName);`

   Searches for the car in the Train with `searchedCarName` and returns its car number if it is found. If there is no car with the searched car name, then the function returns -1 (must return an int).

6. `vector<string> getAllCarNamesByType(const string carType);`

   This function traverses the linked list and returns a vector containing the `carName` of every car that matches the given `carType`. If no cars of that type exist, it should return an empty vector. This is useful for identifying and listing all specific cars of a certain type, like all "Passenger" cars.

For **Student 2**, a brief description of what each member function should do is given below:

1. `CarNode* getCarNode(const int carNumber);`

   Traverses the linked-list to find the node with the given `carNumber` (int). It returns head if `carNumber` < 2, tail if `carNumber` ≥ `numCars`, or the first node pointer to a node whose car number is at least the value of `carNumber`. It's a **private helper function**.

2. `void addCar(const string addedCarName, const int carNumber, const string carType, const double weightTons, const string` **`manufacturerName`**`);`

   Should add a new `CarNode` with the given parameters, at the specified `carNumber` (if `carNumber` is valid given the list, between 1 and the number of cars). All cars at `carNumber` and beyond (because the new car will supplant the place of the old car at `carNumber`) should have their car numbers incremented and all pointers should be handled correctly. Note that if `carNumber` < 2, the car should be placed at the head, or if `carNumber` > `numCars`, it should be placed at the tail. Be sure that your code can handle all special cases (empty list, single node list, placing at head, placing at tail, etc.).

3. `void deleteLastCar();`

   Should delete the last car in the Train (at the tail).

4. `void deleteCar (const int carNumInList);`

   Deletes a `CarNode` based on its position in the list (given by `carNumInList`). It should delete the first car if `carNumInList` < 2 or the last node if `carNumInList` ≥ `numCars`. Be sure to handle the pointers around the node to be deleted correctly (no memory leaks or dangling pointers) and be sure to decrement the appropriate car numbers after deleting the node.

5. `string getCarName(const int carNumInList);`

   Searches for the car at the given car number `carNumInList` and return its name. It should return the name of the head if `carNumInList` < 2, and the tail if `carNumInList` ≥ `numCars`. For the special case of an empty list, return the string "Empty Train" (must return a string).

6. `int getCarTypeCount(const string carType);`

This function traverses the entire linked list and returns the total number of cars that match the given `carType` string. If no cars of that type are found, it should return 0.

Side Notes:

1. **Examples of Train cars manufacturers:**

Examples of train car manufacturers include:

- **Bombardier** (now part of Alstom): A Canadian company known for a wide range of rail vehicles, including locomotives, metro trains, and passenger coaches.
- **Siemens Mobility**: A German company that manufactures high-speed trains (like the ICE), regional trains, and locomotives.
- **Kawasaki Heavy Industries**: A Japanese manufacturer that produces various rail vehicles, including subway cars and commuter trains, often used in places like New York City.
- **CRRC Corporation**: A large Chinese state-owned company that is the world's largest rolling stock manufacturer. It produces a variety of trains, including high-speed trains, locomotives, and freight wagons.
- **General Motors** (Electro-Motive Diesel): While primarily known for its automobiles, GM's EMD division was a major manufacturer of diesel-electric locomotives in the United States.
- **Alstom**: A French multinational company that produces a variety of railway rolling stock, including high-speed trains (TGV). It recently acquired Bombardier's rail division.

2. **Can train cars from different manufacturers be in one train?**

Yes, train cars from different manufacturers can be in one train. This is a common practice in the railroad industry, especially for **freight trains**. Railroad companies often own and operate a diverse fleet of cars from various manufacturers, and they also lease cars from other companies.

A single train can be composed of:

- Locomotives from a company like General Motors (Electro-Motive Diesel).
- Freight cars from multiple different manufacturers.
- Specialized cars, such as those from companies like Kawasaki or Alstom, that are designed for specific tasks.

3. Difference between CarName and CarType.

`carName` is a specific identifier for a particular car, while `carType` is a general category for a car. Think of it like this: a car's **`carName`** is its unique name, such as "Silver Streak Car" or "Grand View Diner". Its **`carType`** is its functional classification, such as "Locomotive," "Passenger," or "Dining".

For example, a train might have multiple cars of the same `carType`, like several "Passenger" cars . However, each of those cars could have a unique `carName`, allowing you to differentiate between them. The `getCarNode` function for Student 1 uses the `carName` to find a specific car, not its general type.

Sample `CarNode`:

Here are some sample `CarNodes` based on the data structure provided in the project.

1. **Locomotive (Engine) Node**
    - **`carNumber`**: 1
    - **`carName`**: "The Iron Horse"
    - **`carType`**: "Locomotive"
    - **`manufacturerName`**: "General Motors"
    - **`weightTons`**: 130.0 (within the 10-143 ton range)
    - **`next`**: (points to the next car)
    - **`prev`**: (nullptr, since it's the head of the list)
2. **Passenger Car Node**
    - **`carNumber`**: 2
    - **`carName`**: "Silver Streak Car"
    - **`carType`**: "Passenger"
    - **`manufacturerName`**: "Bombardier"
    - **`weightTons`**: 55.5 (within the 10-143 ton range)
    - **`next`**: (points to the next car)
    - **`prev`**: (points to the previous car)
3. **Dining Car Node**
    - **`carNumber`**: 3
    - **`carName`**: "Grand View Diner"
    - **`carType`**: "Dining"
    - **`manufacturerName`**: "CRRC"
    - **`weightTons`**: 75.2 (within the 10-143 ton range)
    - **`next`**: (points to the next car)
    - **`prev`**: (points to the previous car)
4. **Caboose Node**
    - **`carNumber`**: 4
    - **`carName`**: "Red Caboose"
    - **`carType`**: "Caboose"
    - **`manufacturerName`**: "" (empty string since none is given)
    - **`weightTons`**: 25.0 (within the 10-143 ton range)
    - **`next`**: (nullptr, since it's the tail of the list)
    - **`prev`**: (points to the previous car)