Programming Project 3 Part 1: I'll link to like if linking liking move

Overview:

This project will be divided into two parts: The first is to implement a templated singly linked list class, the second will be to use your linked list to facilitate some simple AI for maze traversal. The first part will be due in a week's time. The second part will be assigned at that point and will be due the week after.

For this first part you will be provided with 3 files: list.h, studentinfo.h, main.cpp. As with all instructions read everything carefully, if anything is confusing please ask me or post on the discussion forum.

list.h:

You will need to implement the member functions for a singly linked list class. This list (all linked lists for that matter) will be very similar to what we discussed in lecture. So, clearly understanding the how and the why behind the lecture's linked list implementation will go a long way in completing these functions. I cannot emphasize enough the importance of drawing/planning everything out before attempting to implement any of the functions, otherwise you can easily put down erroneous code. Any program with extensive use of pointers can be extremely difficult to debug after the fact, prevention is generally the best approach to anything, but is even truer in this case.

List's member variables are provided to you. There are two Node pointers, head and tail, and an integer, size. If there are Nodes in the list the head must always point to the first Node, the tail must always point to the last Node in the list, and size must always be the same as the number of Nodes in the list. Your implementations must keep these invariants consistent with the intended state of the list when your functions complete. This is different from the lecture's linked list where there was only a head pointer.

The Node class is strictly internal to the List and so defined within the list itself, its definition is complete and nothing more is needed in its implementation.

list's functions:

1. List(): Default constructor. This should construct an empty List, the member variables should be initialized to reflect this state.
2. ~List(): Destructor. The list dynamically allocates nodes, that means when we destruct your list we need to ensure we deallocated the nodes appropriately to avoid memory leaks.
3. void printItems() const: Traverses the list and prints the items in the list in a single line, followed by a newline. printItems should indicate the Front and Rear of the list for example suppose our list contains the strings "Cash", "Shell", and "Ruby", printItems will display in the console:

   ```
   Front Cash Shell Ruby Rear
   ```

4. bool isEmpty() const: returns boolean value indicating if the list is empty or not.
5. void addToFront(Type item): Adds item to a new Node at the Front of the list. Updates head, tail, and size accordingly. Must appropriately handle cases in which the list is empty and if there are nodes already in the list.

6. void addToRear(Type item): Adds item to a new Node at the Rear of the list. Updates head, tail, and size accordingly. Must appropriately handle cases in which the list is empty and if there are nodes already in the list.
7. void addItem(int index, Type item): Given an index, this function adds the item to a new Node at the index. Updates head, tail, and size accordingly. If the index is less than or equal to zero add the item to the front. If the index is greater than or equal to the size of the list then add it to the rear. Otherwise add the item at the index indicated.
8. Type getFront() const: returns the item in the Node at the front of the list without modifying the list. The function cannot be called if the list is empty.
9. Type getRear() const: returns the item in the Node at the rear of the list without modifying the list. The function cannot be called if the list is empty.
10. Type getItem(index item) const: returns the item in the Node in the index place of the list.
11. int getSize() const: returns the size of the list.
12. int findItem(Type item): Searches the list to see if the item is currently in the list. If it is, the function returns the index of the item, otherwise it returns -1;
13. bool deleteFront():Removes the first item in the list, returns true if the item was deleted, false otherwise. Updates head, tail, and size accordingly. Must appropriately manage cases where the list is empty or has one or more items.
14. bool deleteRear() : Removes the last item in the list, returns true if the item was deleted, false otherwise. Updates head, tail, and size accordingly. Must appropriately manage cases where the list is empty or has one item, or has two or more items.
15. bool deleteItem(int index): Removes the item at the index of the list, returns true if the item was deleted false otherwise. Updates head, tail, and size accordingly. Must check to see if the index is inbounds.

studentinfo.h:

There are only two functions here where you return your name and ID, they are used in the automated testing. Please make sure you modify these functions to do what they intend.

main.cpp:

Where you test your work.

Tips:

Start Early! Do not wait until the last minute to work on this; you will run out of time. There is relatively little benefit in trying to complete the project in a single sitting. It is better in terms of time management and also more conducive to learning (due to the way the brain works) to pace out your efforts over a period of days rather than cramming it all in one go.

As always, once you put these files into your project make sure everything compiles without issue before making any changes. For ease of testing and development you can make your member variables public. Of course, don't forget to change it back to private prior to submission.

Compile and submit. The code that is provided to you should compile without error given an empty main. You should make sure you can do so before doing anything else. When you make significant headway, make sure what you have compiles and then submit it, that way if for any reason when you hit the deadline and you can't compile you at least have the previous working code to fall back on. Never submit anything that does not compile. Code that does not compile is worth less than code that does but has half the amount of work; in industry it is worth nothing.

Have test cases before executing any code. For any function, in planning how you want to implement that function, spend time jotting down possible ways to test or break that function. Make sure you have tests that visit all areas of code, meaning if you have if-else statements, you have a test case that will enter each of the branches. Not only that, make sure you know ahead of time what it is you expect to happen based on what you understand the function should be doing. That way you can compare "What is actually happening" with "What you intend". An unfortunate fact-of-life for programmers is that these two are seldom the same. It takes considerable practice to be able to effectively delineate between the two notions; we have the tendency to bias one with the other.

Submission:

For this part of the project you submit 2 files:

list.h studentinfo.h

You will have your own main.cpp for testing, but do not include it with your submission. Combine everything into a zip file name <lastname>_<id>.zip, for example nguyen_123456.zip. Note that when you resubmit on canvas it will postpend your file name with a number indicating its submission order, this is ok. If I take these 8 files, I must be able to compile them using VS2017 without any errors or warnings. Be sure to you do not introduce any compilation or link errors.