CSCI 320, Life Beyond Python, Spring 2022
Project 6: A doubly linked list in C++

Project description

In this project you will implement a doubly linked list with bare-bones functionality in C++. There are two parts, the public interface, which I specify, and the rest of the implementation, where you are free to express yourself.

Smart pointers

Working with pointers is much less common in C++ than in C, since C++ provides container objects and strings are also a class. If you must use pointers, do not use C-style pointers ("nekkid pointers") but instead use the "smart pointers" introduced in C++ 11. Smart pointers were introduced to reduce the likelihood of the common types of C pointer errors. With smart pointers you can usually structure your code in such a way that avoids calls to `delete`.

The two most common types of smart pointers are **unique pointers**, `std::unique_ptr`, and **shared pointers**, `std::shared_ptr`. Both are wrappers around C-style pointers that control access to the underlying pointer.

If we use a unique pointer, there is one and only one pointer that points to a location in memory at any given time. This avoids the problem where two pointers point to the same chunk of memory and the memory is freed through one of the pointers so the other now points to an unallocated chunk of memory. Unique pointers can also be used to prevent memory leaks—when a unique pointer goes out of scope, C++ knows it is safe to free the associated memory. We discussed unique pointers in class with this example.

If we use shared pointers, then multiple shared pointers may point to the same location in memory. You cannot explicitly free the associated memory—C++ will take care of that. C++ tracks the number of pointers that point to the same chunk of memory. As shared pointers go out of scope, this count of pointers to the same object is decremented. When the count reaches zero C++ knows it is safe to free the associated memory.

In this particular case it is a bit tricky to ensure the pointer use count is ultimately zero because the pointers to a node are in its predecessor and successor, not the node itself. Because the destructor sequence for nodes is not obvious I have given you destructor code for the `NODE` and `LIST` classes that works

The class interface

You may **not** change private variables into public variables.

What to do

Create your functions in a file named `list.hpp`. Submit your file `list.hpp` to the autograder on Gradescope for grading.