CS 210 Program 2

Objectives

- Complete a templated linked list data structure
- Use operator overloading
- Use an iterator to traverse the linked list structure
- Implement a Priority Queue using the linked list structure

Steps Overview

1. Complete the Node template class
2. Complete the ListIterator template class
3. Complete a LinkedList template class
4. Implement a Priority Queue using the LinkedList template class

Download starter files

Get p3.zip. Create your project.

Step 1: Node.h

Our linked list structure will be singly linked with head and tail pointers. This means access will be through the head pointer for almost all functions. A template will allow us to vary the data type stored in the Node. Here's the code for a Node that stores only int data. Convert it to a generic type template and store it in a file called **Node.h**. (Yes, the Node class is all inline functions and is in a header file.)

To complete the header file, add the compiler directives: #ifndef NODE_H, #define NODE_H, #endif

```
// compiler directives here
// template<class T>          // uncomment this line
class Node {
    public:
        int data;             // change int to T
        Node *next;

        Node(int newData) {   // change int to T
            data = newData;
            next = nullptr;
        }
    };
// compiler directive here
```

Step 2: ListIterator.h

To create the iterator, we're going to do more than just create a pointer and traverse the list. The class is complicated, but hang in there as we'll explain it as we go.

1). Create a new file called ListIterator.h

Add the compiler directives for LIST_ITERATOR_H in similar fashion as we did for Node.h.

2). Bring in the Node class header:

```
    #include "Node.h"
```

3). Add the template header and private pointer:

```
    template<class T>
    class ListIterator {
    private:
        Node<T> *itptr;    // Pointer to traverse the linked list
    public:
        // Add here the functions described below
    };
```

4). Add the overloaded constructors

4a). The no-arg constructor initializes the iterator's internal pointer to `nullptr` (roughly equivalent to `NULL`). This allows us to get a new iterator by the statement:
`ListIterator<sometype> p`

```
    ListIterator () { itptr = nullptr; }
```

4b). The next constructor initializes the iterator's internal pointer to the passed in argument's internal pointer. This allows us to get a new iterator by the statement: `ListIterator<sometype> p(q)` where `q` is a `ListIterator` object.

`const` instructs the compiler to verify that no modifications to `q` are allowed in this constructor.

`ListIterator<T>&` is a *pass-by-reference* parameter. It is not a pointer, but it does hold a memory address.

```
    ListIterator (const ListIterator<T>& q) { itptr = q.itptr; }
```

4c). The third constructor initializes the iterator's internal pointer to a `Node<T>` object. This allows us to get a new iterator by the statement: `ListIterator<sometype> p(someNodePointer)` where `q` is a `Node<T>` object.

```
    ListIterator (Node<T> *q) { itptr = q; }
```

5). Add overloaded operators prefix (++p) and postfix (p++)

To traverse the list simply by incrementing the pointer is not enough. We have to define what "traversing the list" means. C++ allows *operator overloading* which allows a class to specify how operators applied to the class type will perform. READ about operator overloading before continuing! If you just can't help yourself, then here's a much too brief summary:

Given function header:	`X operator+(const X&) const`
Given variables:	`X aa, bb;`
Given addition operation:	`aa + bb ==implies==> aa.operator+(bb);`

5a). Type conversions performed on both argumentsA prefix increment, ++p, **first advances** the pointer to the next object in the list before returning it's value. The function header parts are:

`ListIterator&` means to *return this ListIterator reference itself*

`operator++` means the '++' increment operator is being overloaded

`()` indicates that the object being reference is the **right**-hand-side object of the operator [e.g., ++p]

```
ListIterator& operator++ () { // prefix, ++p
        itptr = itptr->next;
        return *this;
    }
```

5b). A postfix increment, p++, means to return the **current** value of the pointer and **afterward** to advance it to the next object in the list. The current value is saved, the pointer incremented, then the saved (old) value is returned.

`ListIterator` means to *return by value,* i.e., make a copy and return it

`operator++` means the '++' increment operator is being overloaded

`(int)` indicates that the object being reference is the **left**-hand-side object of the operator [e.g., p++], and `int` indicates an increment by one.

```
ListIterator operator++ (int) { // postfix, p++
        ListIterator<T> tmp = *this; // make a copy of current state
        ++*this; // call prefix to update the state, ++(*this)
        return tmp; // return the copy of the now former state
    }
```

6). Add overloaded operators equality operators

To compare two ListIterators, we check the internal pointers. Now we can write: `p != q` where `p` and `q` are ListIterator objects. The same goes for `p == q` .

```
bool operator== (ListIterator<T> q) {return itptr == q.itptr; }
bool operator!= (ListIterator<T> q) {return itptr != q.itptr; }
```

7). Add overloaded operator dereference

The ListIterator is traversing Node<T> objects, each of which stores data for the list. This data element is what the user added to the list (the user can be unaware of how the data is wrapped in a node and attached to other nodes to make a list). Here we access the data element of the current node referenced by itptr when the ListIterator object is dereferenced.

`T&` means to *return a variable of type T reference* `operator*` means to overload the dereference operator

`()` indicates that the object being reference is the **right**-hand-side object of the operator [e.g., *p]

```
T& operator* () {return  itptr->data; }
```

That completes the `ListIterator` class.

Step 3: LinkedList.h

Add LinkedList.h to your project if you haven't already. Add p2driver.cpp.

Templates require that the class is implemented in the header file. This class will have all inline functions so no header file is required.

If you've completed Steps 1 and 2, you should be able to compile the driver (recommended: use the makefile). Though p2 will run, it will not be correct until you've located the "TODO" statements which indicate the remaining methods you will write. You'll want to add (many) more test cases.

Use good development techniques: incrementally compiling and running the code. Study the methods already implemented for clues on how to complete the remaining methods.

Step 4: LinkedListPQ.h

Implement `PriorityQueue.h` in a class called `LinkedListPQ`. This class will have all inline functions so only the header file is required.

Test your code—consider using a tester like what you had for Program 1, Phase 2.

Submtting your work

Upload your files individually or as a .zip file to Gradescope / Program 2. The autograder will be enabled 3-5 days before the due date. Meanwhile, test, test, test!   Separate autograding tests will test the LinkedList class and the Priority Queue implementation.

Files:

```
Node.h
ListIterator.h
LinkedList.h
LinkedListPQ.h
```

Cheating policy

Don't cheat. Don't share your code. Don't copy another source. Don't deny yourself the opportunity to learn your craft. Don't put yourself in a position to fail the course by cheating.

Do your own work. Do go slow enough to absorb the language constructs introduced in this assignment. Do learn it—these concepts will appear on a future test and/or quiz.