ECE250: Lab Project 1
Doubly Linked Sentinel List
Due Date: Monday, September 24, 2018- 11:00 PM

Purpose

The purpose of the projects is to help you learn the course material and to help you begin to implement your own personal library of tools. Many of the subsequent projects will rely on previous ones (for example, you may be asked to specifically use your linked list classes to implement more complex data structures).

Requirements

In this sub-project, you will implement two classes:

1. a doubly linked list class with sentinels: `Double_sentinel_list`, and
2. a nested doubly linked node class: `Double_node`.

A doubly linked list bound by two sentinels and containing three nodes is shown in Figure 1. The empty doubly linked list bounded by two sentinels is shown in Figure 2, the nodes marked **S** being the sentinel nodes.

Figure 1. A doubly linked list with three nodes bounded by two sentinels.

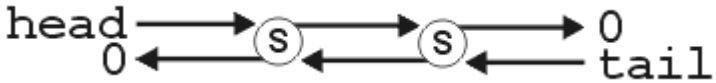

Figure 2. An empty doubly linked list with two sentinels.

Class Specifications

UML Class Diagram

Double_sentinel_list
− list_head:Double_node − list_tail:Double_node − list_size:Integer
+ create():Double_sentinel_list + create(in dsl:Double_sentinel_list):Double_sentinel_list + size():Integer + empty():Boolean + front():Type

```
+ back():Type
+ begin():Double_node
+ end():Double_node
+ rbegin():Double_node
+ rend():Double_node
+ find( in obj:Type ):Double_node
+ count( in obj:Type ):Integer
+ swap( inout list:Double_sentinel_list )
+ =( in rhs:Double_sentinel_list ):Double_sentinel_list
+ push_front( in obj:Type )
+ push_back( in obj:Type )
+ pop_front()
+ pop_back():Type
+ erase( in obj:Type ):Integer
+ destroy()
```

The skeleton code for this class and utility classes are given on LEARN.

Description

This class stores a finite list of *n* (zero or more) elements stored in doubly linked nodes. The following are the properties of this class:

- If there are zero elements in the list, the list is said to be *empty*.
- Each element is stored in an instance of the `Double_node<Type>` class.
- At all times, the head and tail pointers store the addresses of the head and tail *sentinel* nodes, respectively.
- The previous pointer of the head sentinel always points to `nullptr`.
- The next pointer of the tail sentinel always points to `nullptr`.
- If the list is empty, the next pointer of the head sentinel node is assigned the address of the tail sentinel; otherwise, the next pointer of the head sentinel node is assigned the address of the first node in the linked list (the front node).
- If the list is empty, the previous pointer of the tail sentinel node is assigned the address of the head sentinel; otherwise, the previous pointer of the tail sentinel node is assigned the address of the last node in the linked list (the back node).
- The next pointer of the *k*th node ($1 \leq k < n$) stores the address of the ($k + 1$)st node, the next pointer of the *n*th is assigned to the address of the tail sentinel.
- The previous pointer of the *k*th node ($1 < k \leq n$) stores the address of the ($k - 1$)st node, and the previous pointer of the first node is assigned the address of the head sentinel.

Member Variables of Double Sentinel List

There are three member variables:

- Two pointers to `Double_node<Type>` objects, referred to as the *head pointer* and *tail pointer*, respectively, and
- An integer referred to as the *list size* which equals the number of elements in the list.

Member Functions of Double Sentinel List

Constructors

`Double_sentinel_list()`

The constructor creates two instances of a `Double_node<Type>` (called the *sentinels*). The head and tail pointers are set to point to one of the sentinels, each. The values stored in these nodes is not important, you can use the default value or whatever values you want. The previous and next pointers of the head sentinel should be `nullptr` and the address of the tail sentinel, respectively. The previous and next pointers of the tail sentinel should be the address of the head sentinel and `nullptr`, respectively. The node count is set to `0`. (**O**(1))

Destructor

`~Double_sentinel_list()`

The destructor must delete each of the nodes in the list including the sentinels. (**O**(*n*))

Copy Constructor

`Double_sentinel_list( Double_sentinel_list const &list )`

The copy constructor must create a new doubly linked list with a copy of all of the nodes within the linked list pass as the argument `list` with the values stored in the same order. The linked list passed as an argument may not be changed. Once a copy is made, any change to the original linked list must not affect the copy. (**O**(*n*))

Move Constructor

`Double_sentinel_list( Double_sentinel_list &&list )`

The move constructor must create a new doubly linked list with all the nodes found within the linked list passed as an argument `list` with the values stored in the same order. It is assumed that the destructor will immediately be called on the argument linked list as soon as this constructor finishes, so all the nodes in the argument linked list can be used in this newly created linked list. The argument linked list should be updated to one that is empty. (This is most easily done by initializing this linked list as an empty linked list and then calling `swap`.) (**O**(1))

Accessors

This class has these accessors:

`int size() const;`

Returns the number of items in the list. (**O**(1))

`bool empty() const;`

Returns `true` if the list is empty, `false` otherwise. (**O**(1))

`Type front() const;`

Retrieves the object stored in the node pointed to by the next pointer of the head sentinel. This function throws a `underflow` if the list is empty. (**O**(1))

`Type back() const;`

Retrieves the object stored in the node pointed to by the previous pointer of the tail sentinel. This function throws a `underflow` if the list is empty. (**O**(1))

`Double_sentinel_list<Type>::Double_node *begin() const;`

Returns the address stored by the next pointer of the head sentinel node. (**O**(1))

`Double_sentinel_list<Type>::Double_node *end() const;`

Returns the address of the tail sentinel node. (**O**(1))

`Double_sentinel_list<Type>::Double_node *rbegin() const;`

Returns the address stored by the previous pointer of the tail sentinel node. (**O**(1))

`Double_sentinel_list<Type>::Double_node *rend() const;`

Returns the address of the head sentinel node. (**O**(1))

`Double_sentinel_list<Type>::Double_node find( Type const & ) const;`

Returns the address of the first node in the linked list storing a value equal to the argument; if none is found, return `end()`. (**O**(*n*))

`int count( Type const & ) const;`

Returns the number of nodes in the linked list storing a value equal to the argument. (**O**(*n*))

The member functions `begin()` and `end()` allows a user to iterate through the entries of a linked list as follows:

```
       List<int> my_list;
       list.push_front( 0 );
       // Insert and possibly remove other entries into the list

       // Iterate through the linked list from the first entry to the
last
       //  - the type 'auto' lets the compiler determine the type for
you!
        for ( auto *ptr = list.begin(); ptr != list.end(); ptr = ptr-
>next() ) {
                std::cout << ptr->value() << ' ';
        }
```

The member functions `rbegin()` and `rend()` allows a user to iterate through the entries of a linked list **in reverse order** as follows:

```
       // Iterate through the linked list in reverse order
       // from the last entry back down to teh first entry
       //  - the type 'auto' lets the compiler determine the type for
you!
        for ( auto *ptr = list.rbegin(); ptr != list.rend(); ptr =
ptr->previous() ) {
                std::cout << ptr->value() << ' ';
        }
```

If you did not use `auto` (short for automatic type declarations), you would have to explicitly state that the type is `typename Double_sentinel_list<int>::Double_node`. In this case, because the compiler knows the return type of the `begin()` member function, it can use the signature of that member function to deduce the appropriate type for `ptr`.

Mutators

This class has these mutators:

`void swap( Double_sentinel_list &list );`

The swap function swaps all the member variables of this linked list with those of the argument `list`. (**O**(1))

`Double_sentinel_list &operator=( Double_sentinel_list const &rhs );`

The assignment operator makes a copy of the argument (the right-hand side of the assignment) and then swaps the member variables of this node doubly linked sentinel list those of the copy. (**O**($n_{lhs} + n_{rhs}$))

`Double_sentinel_list &operator=( Double_sentinel_list &&rhs );`

The move operator moves the nodes in the argument (the right-hand side of the assignment) linked list to this linked list, changing the argument linked list into an empty list. (**O**(1))

`void push_front( Type const &new_value );`

Creates a new `Double_node<Type>` storing the argument `new_value`, the next pointer of which is set to the next pointer of the sentinel and the previous pointer is set to point to the sentinel. The next pointer of the sentinel and the previous pointer of what was the first node are set to this new node. (**O**(1))

`void push_back( Type const &new_value );`

Similar to `push_front`, this places a new node at the back of the list storing the argument `new_value`. (**O**(1))

`void pop_front();`

Delete the first non-sentinel node at the front of the linked list and update the previous and next pointers of any other node (including possibly the sentinels) within the list as necessary. Throw an `underflow` exception if the list is empty. (**O**(1))

`void pop_back();`

Similar to `pop_front`, delete the last non-sentinel node in the list. This function throws an `underflow` if the list is empty. (**O**(1))

```
int erase( Type const &value );
```

Delete all the nodes in the linked list that have a value equal to the argument `value` (use == to test for equality with the retrieved element). Update the previous and next pointers of any other node (including possibly the sentinels) within the list. Return the number of nodes that were deleted. (**O**(*n*))

Compilation

In Linux, you will have to use `g++ -std=c++11 Double_sentinel_list_driver.cpp`.

How to Test Your Program

You will need to implement the methods of the doubly sentinel list class in the `Double_sentinel_list.h` file. Make sure that your test cases cover most functionality for the `Double_sentinel_list` class as well as boundary cases. We use drivers and tester classes for automated marking, and provide them for you to use. We also provide you with some basic test cases, that can serve as samples for you to create more comprehensive test cases. You can find the testing files on LEARN. To compile and run your code with an input test file (e.g., `test.in`):

- `g++ -std=c++11 Double_sentinel_list_driver.cpp -o Double_sentinel_list_driver`
- `./Double_sentinel_list_driver < test.in`

Program Submission

Once you have completed your solution and tested it comprehensively, you need to build a compressed file, in tar.gz format, which should contain the following:

- `Double_sentinel_list.h`

Build your tar file using the UNIX tar command as given below:

- `tar -cvzf xxxxxxxx_pn.tar.gz Double_sentinel_list.h`

where `xxxxxxxx` is your UW user id (e.g., jsmith) and `n` is the project number which is 1 for this project. All characters in the file name must be lower case. Submit your tar.gz file using LEARN, in the drop box corresponding to this project

What You May Do

You may:

- Make multiple submissions—only the last is kept and marked.

- Declare and define other classes and other member functions not specified in the project description; however, if these appear in other files, those files must be included with your submission.
- You may even, though you must be very cautious, overload the given member functions or add additional parameters to those member functions so long as those extra parameters have default values and the functions as specified in the project may be called with no issues or ambiguities.
- You may use the following standard library functions:

Library Header File	Functions and Classes
`#include <iostream>`	All functions and classes.
`#include <cassert>`	`assert( predicate );`

`#include <algorithm>`	`std::min( T, T );`
	`std::max( T, T );`
	`std::swap( T&, T& );`

`#include <cmath>`	All functions.
`#include <cfloat>`	All constants.
`#include <climits>`	All constants.

Compiler and Testing

All testing will be performed on the Unix server `ecelinux` and it is a requirement that your submissions compile and run on that machine. Source code which compiles using a Windows IDE but does not compile under `g++` will receive a mark of 0.

Late Projects

If you were late for your submission, you can still submit the project two hours after the deadline. You will receive a maximum grade of 50% in the project.

Program Documentation and Style

The following programming style is required for all projects.

Your name and ID must appear at the top of all files which you have created or modified.

Write clear and understandable code. Improve the clarity of your code by using vertical and horizontal spacing, meaningful variable names, proper indentation and comments.

Precede each function with comments indicating:

- What it does
- What each parameter is used for
- Assumptions that it makes
- How it handles errors

Exactly one project will be marked for comments and style. You will not be told which project was marked, nor when the marking has taken place.

Failing project grades

If you receive a mark of less than 70% in any of the projects, you have the opportunity to meet with the lab instructor. You should first determine what went wrong with your submission and try to fix as much as you can. When you meet with the lab instructor, you will discuss what went wrong, and discuss strategies for moving forward. Depending on your responses, you can receive a grade of up to 70%

Acknowledgment

All credits goes to Mr. Douglas Wilhelm Harder.