Program Description

For Part I, the challenge was to complete insertion and deletion functions for a doubly linked list. I also had to implement a copy constructor, assignment operator, and output operator for a DoublyLinkedList class. The final goal was to turn the DoublyLinkedList class into a template class.

For Part II, using the templated doubly linked list, I designed a program to store student records, read records from a file, search for a specific student's record, and output all student records.

Purpose of the Assignment

The main goal of this assignment was to build a strong understanding of LinkedLists and to get experience implementing LinkedLists in our programs. I learned how LinkedLists are very valuable when dealing with insertions and deletions of sorted data. I also learned how different operations on linked lists work, such as insert before, insert after, delete before, delete after, and insert orderly.

Data Structures Description

In this assignment, I learned about the Single and doublyLinkedList data structures (although we only implemented doublylinkedlists). Depending on the Linked List type, each node held an object, a pointer to the next node, and in the case of a doubly Linked List: had a previous node. The DoublyLinkedList class held values for the header and trailer nodes. It provided function interfaces to insert and remove nodes at various positions (before or after). I also overloaded the output and assignment operators for this custom class, as well as created an initial constructor.

LinkedLists are very useful in C++ when dealing with large amounts of sorted data. In the case of a vector or an array, inserting an element at a the current point can be a linear runtime process, due to having to shift each following element. The LinkedList however can insert an element at the current position in constant runtime. The same goes for deletion from the list.

Algorithm Description

Inside SimpleDoublyLinkedList.cpp I implemented four functions: *insert_before, insert_after, delete_before, delete_after*. All four of these functions run with a constant runtime O(1). Since these functions know the current position in the doubly linked list, there is a constant number of operations to perform to add or remove a node.

To *insert_before*, a new node was initialized with data, and a previous pointer to the current node's previous, as well as a next pointer assigned to the current node. The previous node's next pointer was assigned the the value of the new node. The current node's previous pointer also pointed to the new node. This algorithm would also check to make sure that there was a previous node. In the case that there isn't a previous node, the insert function would skip assigning previous->next = n.

To *insert_after*, a new node was initialized with data, a previous pointer assigned to the current node, and a next pointer assigned to the current node's next pointer. The next node's previous pointer was assigned to the address of the new node. The next pointer on the current node was assigned to the new node. This function would also check to make sure there was a next node, in the case that there wasn't, the function would act accordingly.

To *remove_before*, I had to check that there was a previous node, and that the previous node, had a previous node. If the previous node didn't have a previous node, then the current

node would be the first node in the linked list. Pointers would be assigned accordingly and the node before the current node would be deleted.

To *remove_after*, I check that there was a next node, and that the next node had a next node. If next node didn't have a next node, then the function would behave accordingly. Pointers would be reassigned and the node after the current node would be deleted.

Inside DoublyLinkedList I implemented a copy constructor, assignment operator, and output operator. These three functions had a worst case O(n).

The *copy constructor* had to provide an interface to initialize a DoublyLinkedList object with a previously defined DoublyLinkedList. To do this, I had to increment through the “passed in” linked list, and create a new linked list, with new nodes being created in the same order, with the same values, as the passed in linked list. This involves iterating through the entire list, and for that reason is O(n).

The *assignment operator* was also O(n) because it required iterating through the “passed in” list in the same way that the copy constructor did. The only difference is that before creating a new linked list, the assignment operator had to iterate through *this*’s previous linked list and delete every node. These processes are both linear, therefore the total runtime is still linear.

The *output operator* involved iterating through the entire linked list, and for each node, getting the value, and sending it to the output stream. This process also involves traversing the container, and for that reason is linear in runtime.

In Part II of the assignment I implemented a templated DoublyLinkedList and built a less than operator function and insert_orderly function.

The *less than operator* function would compare two records and determine which was was less than the other. First, the two records would be compared by last name. If the last names were different, the comparison would be completed, but if they were the same, the function would then compare first names. If first names were different, the function would return based on the first name values. If first names were the same, the function would then return true or false (according to which was less than the other) based on UIN values.

The *insert_orderly* function has a linear runtime O(n). To *insert_orderly* the function would first test to see if the linked list was empty, if so, a new node would be created with the passed in value. If the linked list already existed (and therefore wasn’t empty) the function would iterate through the container, comparing the current node’s value with the next node’s value. Once the function reached a spot where *current’s value < new node’s value < next node’s value,* the function would essentially “insert_after”, assigning pointers accordingly to insert the new node in it’s correct order. If the function reached the end of the list without finding a correct place, the element would be inserted at the end. If the first node’s element in the linked list was greater than the new node’s element, the new node would be insetted at the beginning of the list. This all requires traversing the container of the linked list, and for this reason, the *insert_orderly* function is linear, O(n).

Program Organization and Description of Classes

Class definitions were placed in header files (except for the templated linked list due to limitations with C++), and implementation files were put in cpp files.

In DoublyLinkedList template, the main class was DoublyLinkedList. It provided functions to add nodes, as well as helpful features to tell when a list was empty, or assign, reassign, or delete lists. To do this, a node class had to be created (and templated) to hold an object, and pointers for previous and next nodes. There was one exception class, which would be used

when trying to run functions, other than insertions, on an empty linked list. I also built a Records class, which would store the first name, last name, UIN, and phone number of a student.

Instructions to Compile and Run Program

Navigate to the desired subdirectory (ex: for templated doubly linked list, look in the folder called “TemplatedDoublyLinkedList”). Next run “make clean”. Then run “make”.
For SimpleDoublyLinkedList, to run type “./SimpleDoublyLinkedList”
For DoublyLinkedList and TemplateDoublyLinkedList, to run type “./Main”

Input and Output Specifications

The program will ask for specific input formatting. For example, when searching for a student in the phonebook, you must type their last name with an upper case letter. Same when searching by first name. The program will loop the search function as long as the user desires, so that they may run multiple searches without exiting the program. To continue the search loop the program looks explicitly for a “y” or “n” input character when prompted. The same sort of functionality is implementing when asking whether or not to output phonebook data. The program is also very particular with its input data text file. The formatting must be precise, otherwise unforeseen errors will occur when loading data.

Logical Exceptions

The program should run without issues. The only foreseeable problems may occur when the input data file is not formatted correctly.

C++ Object Oriented or Generic Programming Features

This assignment relied heavily on templated classes, which in this scenario allows the DoublyLinkedList class to be used as a container for any list of objects. Without this tempting a DoublyLinkedList class would have to be hard coded for each desired object type.

Test Cases

SimpleDoublyLinkedList

```
Create a new list
list:

Insert 10 nodes at back with value 10,20,30,..,100
list: 10, 20, 30, 40, 50, 60, 70, 80, 90, 100,

Insert 10 nodes at front with value 100,90,80,..,10
list: 100, 90, 80, 70, 60, 50, 40, 30, 20, 10, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100,

Delete the last 10 nodes
list: 100, 90, 80, 70, 60, 50, 40, 30, 20, 10,

Delete the first 10 nodes
list:
```

DoublyLinkedList

```
Create a new list
list:

Insert 10 nodes at back with value 10,20,30,..,100
list: 10,20,30,40,50,60,70,80,90,100

Insert 10 nodes at front with value 10,20,30,..,100
list: 100,90,80,70,60,50,40,30,20,10,10,20,30,40,50,60,70,80,90,100

Copy to a new list
list2: 100,90,80,70,60,50,40,30,20,10,10,20,30,40,50,60,70,80,90,100

Assign to another new list
list3: 100,90,80,70,60,50,40,30,20,10,10,20,30,40,50,60,70,80,90,100

Delete the last 10 nodes
list: 100,90,80,70,60,50,40,30,20,10

Delete the first 10 nodes
list:

Make sure the other two lists are not affected.
list2: 100,90,80,70,60,50,40,30,20,10,10,20,30,40,50,60,70,80,90,100
list3: 100,90,80,70,60,50,40,30,20,10,10,20,30,40,50,60,70,80,90,100
```

TemplatedDoublyLinkedList<string>

```
Create a new list
list:

Insert 10 nodes at back with value 10,20,30,..,100
list: 10,20,30,40,50,60,70,80,90,100

Insert 10 nodes at front with value 10,20,30,..,100
list: 100,90,80,70,60,50,40,30,20,10,10,20,30,40,50,60,70,80,90,100

Copy to a new list
list2: 100,90,80,70,60,50,40,30,20,10,10,20,30,40,50,60,70,80,90,100

Assign to another new list
list3: 100,90,80,70,60,50,40,30,20,10,10,20,30,40,50,60,70,80,90,100

Delete the last 10 nodes
list: 100,90,80,70,60,50,40,30,20,10

Delete the first 10 nodes
list:

Make sure the other two lists are not affected.
list2: 100,90,80,70,60,50,40,30,20,10,10,20,30,40,50,60,70,80,90,100
list3: 100,90,80,70,60,50,40,30,20,10,10,20,30,40,50,60,70,80,90,100
```

TemplatedDoublyLinkedList<Record> —Phonebook program

searching for a person:

```
Search:
Enter Last Name, with a capital first letter
Arenas
Found!
Edward Arenas
UIN:   239924731
Phone: 2525976612
```

searching for a person who shares a last name and first name with another person:

```
Search:
Enter Last Name, with a capital first letter
Wiseman
Enter First Name
Mary
Enter UIN
464996747
Found!
Mary Wiseman
UIN:   464996747
Phone: 3253740973
```

searching for a person who isn't in the phonebook:

```
Search:
Enter Last Name, with a capital first letter
Kyle
Not Found
```

giving bad input:

```
Search:
Enter Last Name, with a capital first letter
f
INPUT ERROR
```

dumping phonebook:

```
Dump phonebook?(y/n) y
Dumping phonebook
-----------------------------------
Edna Andrews
UIN:   528320876
Phone: 4352514822

Edward Arenas
UIN:   239924731
Phone: 2525976612

Richard Autry
UIN:   527646269
Phone: 6028236739

Mary Latham
UIN:   174485583
Phone: 2156902061

Adelle Lawrence
UIN:   606880340
Phone: 4154068779

Fred Leblanc
UIN:   253174074
Phone: 6782391146

Rolando Lee
UIN:   680385098
Phone: 7752476489

Mozell Lester
UIN:   173503264
Phone: 5706145308

Francis Lile
UIN:   237818062
Phone: 7044943520

Felecia Lopez
UIN:   524085043
Phone: 3036105461

Andrew Weaver
UIN:   460497442
Phone: 2817601558

Floyd Whitney
UIN:   367196315
Phone: 2313697993

Bernice Wilham
UIN:   8641309
Phone: 8027701514

Kristi Wiseman
UIN:   311708896
Phone: 8122398910

Mary Wiseman
UIN:   150521209
Phone: 7324463703

Mary Wiseman
UIN:   224376947
Phone: 7037213439

Mary Wiseman
UIN:   464996747
Phone: 3253740973

Paul Wiseman
UIN:   347075568
Phone: 2179340158
```