Name: Michael Beaver  
Other students in group work: None

Course: CS 355

Semester: Fall 2012

Assignment Number: 3

Assignment Type: Homework 3

Assignment Description: Modify the LinkedList class as specified and provide appropriate test cases.

Assignment Due Date: Tuesday, September 4, 2012 (precisely at 12:30 p.m.)

To Be Included in Portfolio: YES

Total Grade:   
 Coding Requirements Grade (60 possible), Test cases Grade (20 possible), Analysis Grade (20 possible)

Implement the Linked List class as specified in class. Create test cases to ensure the Linked List works. After completing the work, answer the Analysis questions.

Coding Requirements:

1. \_\_\_\_\_Node class complete
   1. \_\_\_\_3 Node Constructors
2. \_\_\_\_\_Linked List class complete
   1. \_\_\_\_Linked List constructor
   2. \_\_\_\_Insert at front
   3. \_\_\_\_Insert at end
   4. \_\_\_\_Generic insert
   5. \_\_\_\_Remove from front
   6. \_\_\_\_Remove from end
   7. \_\_\_\_Generic remove
   8. \_\_\_\_insertion operator
3. \_\_\_\_\_Create a driver program to thoroughly test class.

Test Case Requirements Met:  
\_\_\_\_ test cases I required are met  
\_\_\_\_ new test cases are added to show accuracy

Name: Michael Beaver

Course: CS 355

Semester: Fall 2012

Assignment Number: 3

Assignment Type: Homework 3 – Test Cases

Assignment Description: Complete the Test Cases below. If a test case fails, modify code and test again. Copy and paste table for subsequent tests until the code is correct. Add more test cases to ensure multiple possibilities are checked. Also, check Remove routines and constructors.

Assignment Due Date: Tuesday, September 4, 2012 (precisely at 12:30 p.m.)

To Be Included in Portfolio: YES

I had all the kinks worked out before class ended Thursday. Unfortunately, I did not record my test case I was using. Fortunately, it was somewhat similar to all the test cases below, insofar as it inserted values in random orders and removed values and tried to remove nonexistent values. The only problem I ran into was a loss of order when inserting values in the middle of the list. I fixed that problem relatively quickly, and there seemed to be no other issues, even with new, numerous test cases.

Test Case 1

|  |  |  |  |
| --- | --- | --- | --- |
| Date/Time: | Expected Result | Actual Result | Action needed (Yes/No) |
| Insert (13) | 13 | 13 | No |
| Insert (20) | 13 20 | 13 20 | No |
| Insert (18) | 13 18 20 | 13 18 20 | No |
| Insert (40) | 13 18 20 40 | 13 18 20 40 | No |

Test Case 2

|  |  |  |  |
| --- | --- | --- | --- |
| Date/Time: | Expected Result | Actual Result | Action needed (Yes/No) |
| Insert (50) | 50 | 50 | No |
| Insert (3) | 3 50 | 3 50 | No |
| Insert (7) | 3 7 50 | 3 7 50 | No |
| Insert (5) | 3 5 7 50 | 3 5 7 50 | No |

Test Case 3

|  |  |  |  |
| --- | --- | --- | --- |
| Date/Time: | Expected Result | Actual Result | Action needed (Yes/No) |
| Insert (2) | 2 | 2 | No |
| Insert (3) | 2 3 | 2 3 | No |
| Insert (4) | 2 3 4 | 2 3 4 | No |
| Insert (1) | 1 2 3 4 | 1 2 3 4 | No |

Test Case 4

|  |  |  |  |
| --- | --- | --- | --- |
| Date/Time: | Expected Result | Actual Result | Action needed (Yes/No) |
| Insert (10) | 10 | 10 | No |
| Insert (12) | 10 12 | 10 12 | No |
| Insert (14) | 10 12 14 | 10 12 14 | No |
| Insert (13) | 10 12 13 14 | 10 12 13 14 | No |

Test Case 5 – Testing duplicate values, which are allowed (<= test)

|  |  |  |  |
| --- | --- | --- | --- |
| Date/Time: | Expected Result | Actual Result | Action needed (Yes/No) |
| Insert (15) | 15 | 15 | No |
| Insert (10) | 10 15 | 10 15 | No |
| Insert (17) | 10 15 17 | 10 15 17 | No |
| Insert (15) | 10 15 15 17 | 10 15 15 17 | No |

Test Case 6

|  |  |  |  |
| --- | --- | --- | --- |
| Date/Time: | Expected Result | Actual Result | Action needed (Yes/No) |
| Insert (13) | 13 | 13 | No |
| Insert (20) | 13 20 | 13 20 | No |
| Insert (18) | 13 18 20 | 13 18 20 | No |
| Insert (40) | 13 18 20 40 | 13 18 20 40 | No |
| Remove (18) | 13 20 40 | 13 20 40 | No |
| Remove (13) | 20 40 | 20 40 | No |
| Remove (42) | <failure message> | “Failed to remove 42!” | No |

Test Case 7

|  |  |  |  |
| --- | --- | --- | --- |
| Date/Time: | Expected Result | Actual Result | Action needed (Yes/No) |
| Insert (20) | 20 | 20 | No |
| Insert (25) | 20 25 | 20 25 | No |
| Insert (23) | 20 23 25 | 20 23 25 | No |
| Insert (21) | 20 21 23 25 | 20 21 23 25 | No |
| Remove (23) | 20 21 25 | 20 21 25 | No |
| Remove (25) | 20 21 | 20 21 | No |
| Remove (25) | <failure message> | “Failed to remove 25!” | No |

Test Case 8 – Removing values from an empty list

|  |  |  |  |
| --- | --- | --- | --- |
| Date/Time: | Expected Result | Actual Result | Action needed (Yes/No) |
| Insert (17) | 17 | 17 | No |
| Remove (23) | <failure message> | “Failed to remove 23!” | No |
| Remove (17) | (empty list) | (empty) | No |
| Remove (25) | <failure message> | “Failed to remove 25!” | No |

Test Case 9

|  |  |  |  |
| --- | --- | --- | --- |
| Date/Time: | Expected Result | Actual Result | Action needed (Yes/No) |
| Insert (10) | 10 | 10 | No |
| Insert (20) | 10 20 | 10 20 | No |
| Insert (5) | 5 10 20 | 5 10 20 | No |
| Insert (15) | 5 10 15 20 | 5 10 15 20 | No |
| Remove (23) | <failure message> | “Failed to remove 23!” | No |
| Remove (20) | 5 10 15 | 5 10 15 | No |
| Remove (5) | 10 15 | 10 15 | No |

Test Case 10

|  |  |  |  |
| --- | --- | --- | --- |
| Date/Time: | Expected Result | Actual Result | Action needed (Yes/No) |
| Insert (5) | 5 | 5 | No |
| Insert (10) | 5 10 | 5 10 | No |
| Insert (15) | 5 10 15 | 5 10 15 | No |
| Insert (13) | 5 10 13 15 | 5 10 13 15 | No |
| Remove (15) | 5 10 13 | 5 10 13 | No |
| Remove (5) | 10 13 | 10 13 | No |
| Remove (13) | 10 | 10 | No |

Test Case 11

|  |  |  |  |
| --- | --- | --- | --- |
| Date/Time: | Expected Result | Actual Result | Action needed (Yes/No) |
| Insert (10) | 10 | 10 | No |
| Insert (20) | 10 20 | 10 20 | No |
| Insert (15) | 10 15 20 | 10 15 20 | No |
| Insert (25) | 10 15 20 25 | 10 15 20 25 | No |
| Remove (25) | 10 15 20 | 10 15 20 | No |
| Remove (15) | 10 20 | 10 20 | No |
| Remove (20) | 10 | 10 | No |
| Remove (13) | <failure message> | “Failed to remove 13!” | No |
| Remove ( 10) | (empty list) | (empty) | No |
| Remove (25) | <failure message> | “Failed to remove 25!” | No |

Name: Michael Beaver

Course: CS 355

Semester: Fall 2012

Assignment Number: 2

Assignment Type: Homework 2 - Analysis

Assignment Description: Carefully answer the questions below. Be sure you answer in complete sentences and with correct grammar.

Assignment Due Date: Thursday, August 30, 2012 (precisely at 12:30 p.m.)

To Be Included in Portfolio: YES

Question 1: What are the implications for not overloading the assignment operator in this class?

The implicitly defined assignment operator knows how to handle basic data types, such as ints, but it does not know how to handle dynamic memory, such as the pointer “next” in the Node class. Hence, if using the implicitly defined assignment operator, the “data” values for each Node may be reassigned, but the reassigning of the pointers will be flubbed. If ListA already has its own set of Nodes and it is assigned to ListB (which also has its own Nodes), then ListA’s original set of Nodes will be lost. This memory leak is unfortunate, costly, and totally avoidable.

To prevent such leaks, the class’s writer should explicitly define an overloaded assignment operator. In the overloaded assignment operator, the class’s writer can handle the dynamic memory and the basic data types. If the dynamic memory is properly handled, then ListA’s original set of Nodes should be destroyed, and its “head” pointer should redirect to the start of ListB. As a costlier alternative (with regard to memory allocation), the assignment operator could overloaded to clear ListA and insert new Nodes into ListA. These new Nodes would be exact duplicates of the Nodes from ListB. Regardless of the implementation, the goal is to avoid immediate memory leaks and latent memory leaks that may occur later in a user’s program.

Question 2: Based upon Question 1, what other methods should be written for this type of class and why?

Most importantly, this class needs a copy constructor. The implicitly defined copy constructor cannot effectively handle dynamic memory pointers (e.g., head). Having only an overloaded assignment operator is not sufficient. Therefore, it is essential that a copy constructor is explicitly defined for the class. An explicitly defined copy constructor would allow the user to initialize a LinkedList at instantiation. It is important to note that instantiation is not assignment; hence, a copy constructor is valuable. The copy constructor is useful when passing data by value, returning a function result by value, or returning an object. Thus, for the LinkedList class to be more robust, a copy constructor should be implemented.

For improved functionality, a proper search method should be implemented. Whether inserting a value or merely checking if a Node contains a value, searching can be beneficial. Regardless of the searching algorithm, a search method could increase code reuse; rather than having the same code ten times, a method could be called ten times. If a user wanted to restrict data to being unique (i.e., no duplicates), then a search method could determine if a value already exists in the list. If a data value is contained in the list, then the method would return true, and vice versa.

Also, rather than having separate insertion methods, such as insertFront(int) and insertBack(int), a comprehensive insertion method would be convenient and reduce the amount of code. The user could explicitly specify the location, or the location could be based on some user-defined sorting criteria. Furthermore, a comprehensive insertion method would allow for insertion anywhere in the list, not just the front and the back of the list.

Finally, it is possible that other operators could be overloaded. A programmer may wish to append two lists, compare two lists, use the extraction operator, etc. To accompany such a user, the class’s writer would need to overload these operators. The overloaded addition operator would be particularly useful for appending one list to the end of another list. The equality operator could be overloaded to determine if two LinkedLists are equivalent or not. The beauty of overloading operators is that they may be defined however the writer wishes to define them. Hence, the overloaded addition operator could be overloaded to add the data values from one list to the data values of another list, and so on.

Question 3: Other than methods implemented for the purpose of the clean copying and removal of dynamic data, what other methods might you implement for this class? Give the prototypes for two and briefly describe their purpose and how they would work.

Note: Method prototypes are written as they would be written in the class header/declaration.

bool append(const LinkedList& list2); // if (myList.append(myList2)) . . .

LinkedList operator +(const LinkedList& list) const; // cout << (myList + myList2) << endl;

These methods could be written to append one LinkedList to the end of a source LinkedList. For example, the append method could be written so that the next pointer of the final Node in the source list actually points to the Node that the head pointer of list2 points to. However, this implementation could result in costly memory leaks. It would be better to add Nodes onto the end of the source list (i.e., copies of the Nodes from list2), rather than redirecting the last Node to point to the head of list2. Yet, this solution also implies costly overhead.

Even so, it would be more user friendly to implement an overloaded version of the addition operator. The addition operator could call the append method, or it could use the implementation of the append method. Most users are not going to instinctively use the append method, so the overloaded addition operator might prove to be a user friendly solution. Though, it is important to remember that users often have different definitions of what it means to “add” two LinkedLists, or any programmer-defined data type.

bool linearSearch(const int value) const; // if (myList.linearSearch(6)) . . .

bool linearSearch(const int value, int& position) const; // if (myList.linearSearch(6, pos)) . . .

As state beforehand, a search method could prove valuable with regard to streamlining code and code reuse. A search method would eliminate the need to implement or copy and paste a search routine into each function that requires one. One centralized search method would allow for any function to call it without unnecessarily copying the same code. What is more, a search method would prove invaluable if the LinkedList’s data values have restricting criteria, such as data values must be unique.

As the method name suggests, this method would perform a linear search for a data value in a LinkedList. Of course, it is somewhat redundant to refer to the search as “linear” since a singly LinkedList is linear by definition (i.e., one Node points to the next). Regardless, if the data value is contained in a Node in the LinkedList, then the method would return true. If the data value is not found, then the method would return false. The method could be overloaded to take in an integer value “position” as a pass-by-reference value. The “position” value would be the number Node that the data value is found in. For example, a search for the value 6 in the LinkedList myList with Node data values [1, 3, 6, 9, 0], would return true. The same search with the overloaded version of the method would return true as well, and the “position” value would be assigned the value 2 or 3 if the LinkedList begins at 0 or 1, respectively.