Name: Michael Beaver  
Other students in group work: None

Course: CS 355

Semester: Fall 2012

Assignment Number: 3

Assignment Type: Homework 3

Assignment Description: Implement LinkedList class as specified, provide appropriate test cases, and answer analysis questions. The LinkedList MUST be singly linked.

Assignment Due Date: Tuesday, September 11, 2012 (precisely at 12:30 p.m.)

To Be Included in Portfolio: YES

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

Implement a complete 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. \_\_\_\_\_ Templated Node class complete
   1. \_\_\_\_3 Node Constructors
2. \_\_\_\_\_Templated Singly-Linked List class complete
   1. \_\_\_\_Linked List constructor
   2. \_\_\_\_Linked List copy constructor
   3. \_\_\_\_Destructor
   4. \_\_\_\_Assignment operator
   5. \_\_\_\_Insert (in order, return true or false, place cursor at new item)
   6. \_\_\_\_Remove (given item to remove, return true or false, place cursor at item following removed item, place at end of list if not found)
   7. \_\_\_\_Search (return address if found from cursor to end of list, return NULL if not found, place cursor at found location or at end of list if not found)
   8. \_\_\_\_Test Print Routine (Print the list separated by tabs, print square brackets around the value at cursor)
   9. \_\_\_\_AtCursor (return data item at the cursor)
   10. \_\_\_\_GoToBeginning (move cursor to beginning of the list, NULL if empty)
   11. \_\_\_\_GoToEnd (move cursor to last item in list, NULL if empty)
   12. \_\_\_\_GoToNext (move cursor to next slot, if on last item, move to first item)
   13. \_\_\_\_GotToPrev (move cursor to the previous slot. If on first item, move to last)
   14. \_\_\_\_ClearList (deallocate space, set head and cursor to NULL, can be called from destructor)
   15. \_\_\_\_EmptyList (return true if empty, false otherwise)
3. \_\_\_\_\_Use the driver provided to test work.
4. \_\_\_\_\_const is used wherever appropriate

Test Case Requirements Met:  
\_\_\_\_ created at least one test case for each method  
\_\_\_\_ test cases showed methods were correct  
Analysis Requirements Met:  
\_\_\_\_ Clear and correct communication  
\_\_\_\_ Reasonable/correct answers and justifications

Name: Michael Beaver

Course: CS 355

Semester: Fall 2012

Assignment Number: 3

Assignment Type: Homework 3 – Test Cases

Assignment Description: Create Test cases for each of the methods. You should show enough test cases to demonstrate the method works. Be sure to think about special cases such as multiple removes from an empty list. Try to break your code.

Assignment Due Date: Tuesday, September 11, 2012 (precisely at 12:30 p.m.)

To Be Included in Portfolio: YES

Test Case 1 – Insertion Test 1

|  |  |  |  |
| --- | --- | --- | --- |
| Date/Time: | Expected Result | Actual Result | Action needed (Yes/No) |
| Insert (10) | [10] | <compile error> | Yes – Change method name |
| Insert (17) | 10 [17] | “ | “ |
| Insert (5) | [5] 10 17 | “ | “ |
| Insert (3) | [3] 5 10 17 | “ | “ |
| Insert (20) | 3 5 10 17 [20] | “ | “ |
| Insert (21) | 3 5 10 17 20 [21] | “ | “ |
| Insert (22) | 3 5 10 17 20 21 [22] | “ | “ |
| Insert (7) | 3 5 [7] 10 17 20 21 22 | “ | “ |

|  |  |  |  |
| --- | --- | --- | --- |
| Date/Time: | Expected Result | Actual Result | Action needed (Yes/No) |
| Insert (10) | [10] | 10 | Yes – Using wrong Print() |
| Insert (17) | 10 [17] | 10 17 | “ |
| Insert (5) | [5] 10 17 | 5 10 17 | “ |
| Insert (3) | [3] 5 10 17 | 3 5 10 17 | “ |
| Insert (20) | 3 5 10 17 [20] | 3 5 10 17 20 | “ |
| Insert (21) | 3 5 10 17 20 [21] | 3 5 10 17 20 21 | “ |
| Insert (22) | 3 5 10 17 20 21 [22] | 3 5 10 17 20 21 22 | “ |
| Insert (7) | 3 5 [7] 10 17 20 21 22 | 3 5 7 10 17 20 21 22 | “ |

|  |  |  |  |
| --- | --- | --- | --- |
| Date/Time: | Expected Result | Actual Result | Action needed (Yes/No) |
| Insert (10) | [10] | [10] | No |
| Insert (17) | 10 [17] | 10 [17] | “ |
| Insert (5) | [5] 10 17 | [5] 10 17 | “ |
| Insert (3) | [3] 5 10 17 | [3] 5 10 17 | “ |
| Insert (20) | 3 5 10 17 [20] | 3 5 10 17 [20] | “ |
| Insert (21) | 3 5 10 17 20 [21] | 3 5 10 17 20 [21] | “ |
| Insert (22) | 3 5 10 17 20 21 [22] | 3 5 10 17 20 21 [22] | “ |
| Insert (7) | 3 5 [7] 10 17 20 21 22 | 3 5 [7] 10 17 20 21 22 | “ |

Test Case 2 – Insertion Test 2

|  |  |  |  |
| --- | --- | --- | --- |
| Date/Time: | Expected Result | Actual Result | Action needed (Yes/No) |
| Insert (1000) | [1000] | [1000] | No |
| Insert (-20) | [-20] 1000 | [-20] 1000 | “ |
| Insert (50) | -20 [50] 1000 | -20 [50] 1000 | “ |
| Insert (1) | -20 [1] 50 1000 | -20 [1] 50 1000 | “ |
| Insert (0) | -20 [0] 1 50 1000 | -20 [0] 1 50 1000 | “ |
| Insert (-300) | [-300] -20 0 1 50 1000 | [-300] -20 0 1 50 1000 | “ |
| Insert (50) | -300 -20 0 1 [50] 50 1000 | -300 -20 0 1 [50] 50 1000 | “ |
| Insert (999) | -300 -20 0 1 50 50 [999] 1000 | -300 -20 0 1 50 50 [999] 1000 | “ |

Test Case 3 – Removal Case 1

|  |  |  |  |
| --- | --- | --- | --- |
| Date/Time: | Expected Result | Actual Result | Action needed (Yes/No) |
| Remove (1) from empty list | (empty) | (empty) | No |
| Insert (0..7) | 0 1 2 3 4 5 6 [7] | 0 1 2 3 4 5 6 [7] | “ |
| GoToBeginning() | [0] 1 2 3 4 5 6 7 | [0] 1 2 3 4 5 6 7 | “ |
| Remove (5) | 0 1 2 3 4 [6] 7 | 0 1 2 3 4 [6] 7 | “ |
| GoToBeginning() | [0] 1 2 3 4 6 7 | [0] 1 2 3 4 6 7 | “ |
| Remove (0) | [1] 2 3 4 6 7 | [1] 2 3 4 6 7 | “ |
| GoToBeginning() | [1] 2 3 4 6 7 | [1] 2 3 4 6 7 | “ |
| Remove(7) | [1] 2 3 4 6 | <nothing> | Yes – Tweak Search() |
| GoToBeginning() | [1] 2 3 4 6 | [1] 2 3 4 6 | No |
| Remove (3) | 1 2 [4] 6 | 1 2 [4] 6 | “ |
| GoToBeginning() | [1] 2 4 6 | [1] 2 4 6 | “ |
| Remove(10) | 1 2 4 [6] | 1 2 4 6 [7] | Yes – Affected by above |

|  |  |  |  |
| --- | --- | --- | --- |
| Date/Time: | Expected Result | Actual Result | Action needed (Yes/No) |
| Remove (1) from empty list | (empty) | (empty) | No |
| Insert (0..7) | 0 1 2 3 4 5 6 [7] | 0 1 2 3 4 5 6 [7] | “ |
| GoToBeginning() | [0] 1 2 3 4 5 6 7 | [0] 1 2 3 4 5 6 7 | “ |
| Remove (5) | 0 1 2 3 4 [6] 7 | 0 1 2 3 4 [6] 7 | “ |
| GoToBeginning() | [0] 1 2 3 4 6 7 | [0] 1 2 3 4 6 7 | “ |
| Remove (0) | [1] 2 3 4 6 7 | [1] 2 3 4 6 7 | “ |
| GoToBeginning() | [1] 2 3 4 6 7 | [1] 2 3 4 6 7 | “ |
| Remove(7) | [1] 2 3 4 6 | [1] 2 3 4 6 | “ |
| GoToBeginning() | [1] 2 3 4 6 | [1] 2 3 4 6 | “ |
| Remove (3) | 1 2 [4] 6 | 1 2 [4] 6 | “ |
| GoToBeginning() | [1] 2 4 6 | [1] 2 4 6 | “ |
| Remove(10) | 1 2 4 [6] | 1 2 4 [6] | “ |

Test Case 4 – Removal Test 2

|  |  |  |  |
| --- | --- | --- | --- |
| Date/Time: | Expected Result | Actual Result | Action needed (Yes/No) |
| Remove (1) from empty list | (empty) | (empty) | No |
| Insert (5) | [5] | [5] | “ |
| Insert (4) | [4] 5 | [4] 5 | “ |
| Insert (3) | [3] 4 5 | [3] 4 5 | “ |
| Insert (2) | [2] 3 4 5 | [2] 3 4 5 | “ |
| Insert (-10) | [-10] 2 3 4 5 | [-10] 2 3 4 5 | “ |
| GoToBeginning() | [-10] 2 3 4 5 | [-10] 2 3 4 5 | “ |
| Remove (4) | -10 2 3 [5] | -10 2 3 [5] | “ |
| GoToBeginning() | [-10] 2 3 5 | [-10] 2 3 5 | “ |
| Remove (2) | -10 [3] 5 | -10 [3] 5 | “ |
| GoToBeginning() | [-10] 3 5 | [-10] 3 5 | “ |
| Remove(-10) | [3] 5 | [3] 5 | “ |
| GoToBeginning() | [3] 5 | [3] 5 | “ |
| Remove (3) | [5] | [5] | “ |
| GoToBeginning() | [5] | [5] | “ |
| Remove(10) | [5] | [5] | “ |

Test Case 5 – Search Test 1

|  |  |  |  |
| --- | --- | --- | --- |
| Date/Time: | Expected Result | Actual Result | Action needed (Yes/No) |
| Insert (10) | [10] | [10] | No |
| Insert (12) | 10 [12] | 10 [12] | “ |
| Insert (14) | 10 12 [14] | 10 12 [14] | “ |
| Insert (8) | [8] 10 12 14 | [8] 10 12 14 | “ |
| Insert (6) | [6] 8 10 12 14 | [6] 8 10 12 14 | “ |
| GoToBeginning() | [6] 8 10 12 14 | [6] 8 10 12 14 | “ |
| Search (12) | &Node(12) | <runtime error> | Yes |
| Search (10) | NULL | “ | “ |
| Search (14) | &Node(14) | “ | “ |
| GoToBeginning() | [6] 8 10 12 14 | “ | “ |
| Search (6) | &Node(6) | “ | “ |
| Search (8) | &Node(8) | “ | “ |

|  |  |  |  |
| --- | --- | --- | --- |
| Date/Time: | Expected Result | Actual Result | Action needed (Yes/No) |
| Insert (10) | [10] | [10] | No |
| Insert (12) | 10 [12] | 10 [12] | “ |
| Insert (14) | 10 12 [14] | 10 12 [14] | “ |
| Insert (8) | [8] 10 12 14 | [8] 10 12 14 | “ |
| Insert (6) | [6] 8 10 12 14 | [6] 8 10 12 14 | “ |
| GoToBeginning() | [6] 8 10 12 14 | [6] 8 10 12 14 | “ |
| Search (12) | . . . [12] . . ., &Node(12) | . . . [12] . . ., 007E4CC0 | “ |
| Search (10) | . . . [10] . . ., NULL | . . . [10] . . ., 00000000 | “ – Starts search at cursor |
| Search (14) | . . . [14], &Node(14) | . . . [14], 007E4D08 | “ |
| GoToBeginning() | [6] 8 10 12 14 | [6] 8 10 12 14 | “ |
| Search (6) | [6] . . ., &Node(6) | [6] . . ., 007E4D98 | “ |
| Search (8) | . . . [8] . . ., &Node(8) | . . . [8] . . ., 007E4D98 | “ |

Test Case 6 – Search Test 2

|  |  |  |  |
| --- | --- | --- | --- |
| Date/Time: | Expected Result | Actual Result | Action needed (Yes/No) |
| Search empty list (1) | NULL | 00000000 | No |
| Insert (10) | [10] | [10] | “ |
| Insert (12) | 10 [12] | 10 [12] | “ |
| Insert (14) | 10 12 [14] | 10 12 [14] | “ |
| Insert (8) | [8] 10 12 14 | [8] 10 12 14 | “ |
| Insert (6) | [6] 8 10 12 14 | [6] 8 10 12 14 | “ |
| GoToBeginning() | [6] 8 10 12 14 | [6] 8 10 12 14 | “ |
| Search (6) | [6] . . ., &Node(6) | [6] . . ., 00194FB8 | “ |
| Search (8) | . . . [8] . . ., &Node(8) | . . . [8] . . ., 00194D50 | “ |
| Search (10) | . . . [10] . . ., &Node(10) | . . . [10] . . ., 00194C78 | “ |
| Search (12) | . . . [12] . . ., &Node(12) | . . . [12] . . ., 00194CC0 | “ |
| Search (14) | . . . [14], &Node(14) | . . . [14], 00194D08 | “ |
| GoToBeginning(); | [6] 8 10 12 14 | [6] 8 10 12 14 | “ |
| Search (16) | . . . [14], NULL | . . . [14], 00000000 | “ |

Test Case 7 – Print Test 1

|  |  |  |  |
| --- | --- | --- | --- |
| Date/Time: | Expected Result | Actual Result | Action needed (Yes/No) |
| Print() | (empty) | (empty) | No |
| Insert (0..7) | 0 1 2 3 4 5 6 [7] | 0 1 2 3 4 5 6 [7] | “ |
| Print() | 0 1 2 3 4 5 6 [7] | 0 1 2 3 4 5 6 [7] | “ |

Test Case 8 – Print Test 2

|  |  |  |  |
| --- | --- | --- | --- |
| Date/Time: | Expected Result | Actual Result | Action needed (Yes/No) |
| << | (empty) | (empty) | No |
| Insert (10..15) | 10 11 12 13 14 [15] | 10 11 12 13 14 [15] | “ |
| << | 10 11 12 13 14 [15] | 10 11 12 13 14 [15] | “ |

Test Case 9 – Cursor Test 1

|  |  |  |  |
| --- | --- | --- | --- |
| Date/Time: | Expected Result | Actual Result | Action needed (Yes/No) |
| Insert (10) | [10] | [10] | No |
| Insert (5) | [5] 10 | [5] 10 | “ |
| Insert (12) | 5 10 [12] | 5 10 [12] | “ |
| Insert (3) | [3] 5 10 12 | [3] 5 10 12 | “ |
| GoToNext() | 3 [5] 10 12 | 3 [5] 10 12 | “ |
| AtCursor() | 5 | 5 | “ |
| GoToNext() | 3 5 [10] 12 | 3 5 [10] 12 | “ |
| AtCursor() | 10 | 10 | “ |
| GoToPrev() | 3 [5] 10 12 | 3 [5] 10 12 | “ |
| AtCursor() | 5 | 5 | “ |
| GoToPrev() | [3] 5 10 12 | [3] 5 10 12 | “ |
| AtCursor() | 3 | 3 | “ |

Test Case 10 – Cursor Test 2

|  |  |  |  |
| --- | --- | --- | --- |
| Date/Time: | Expected Result | Actual Result | Action needed (Yes/No) |
| Insert (10) | [10] | [10] | No |
| Insert (5) | [5] 10 | [5] 10 | “ |
| Insert (12) | 5 10 [12] | 5 10 [12] | “ |
| Insert (3) | [3] 5 10 12 | [3] 5 10 12 | “ |
| GoToPrev() | 3 5 10 [12] | 3 5 10 [12] | “ |
| AtCursor() | 12 | 12 | “ |
| GoToPrev() | 3 5 [10] 12 | 3 5 [10] 12 | “ |
| AtCursor() | 10 | 10 | “ |
| GoToBeginning() | [3] 5 10 12 | [3] 5 10 12 | “ |
| AtCursor() | 3 | 3 | “ |
| GoToEnd() | 3 5 10 [12] | 3 5 10 [12] | “ |
| AtCursor() | 12 | 12 | “ |

Test Case 11 – Cursor Test 3

|  |  |  |  |
| --- | --- | --- | --- |
| Date/Time: | Expected Result | Actual Result | Action needed (Yes/No) |
| Insert (10) | [10] | [10] | No |
| Insert (5) | [5] 10 | [5] 10 | “ |
| Insert (12) | 5 10 [12] | 5 10 [12] | “ |
| Insert (3) | [3] 5 10 12 | [3] 5 10 12 | “ |
| GoToPrev() | 3 5 10 [12] | 3 5 10 [12] | “ |
| AtCursor() | 12 | 12 | “ |
| GoToPrev() | 3 5 [10] 12 | 3 5 [10] 12 | “ |
| AtCursor() | 10 | 10 | “ |
| GoToPrev() | 3 [5] 10 12 | 3 [5] 10 12 | “ |
| AtCursor() | 5 | 5 | “ |
| GoToEnd() | 3 5 10 [12] | 3 5 10 [12] | “ |
| AtCursor() | 12 | 12 | “ |

Test Case 12 – Cursor Test 4

|  |  |  |  |
| --- | --- | --- | --- |
| Date/Time: | Expected Result | Actual Result | Action needed (Yes/No) |
| Insert (10) | [10] | [10] | No |
| Insert (5) | [5] 10 | [5] 10 | “ |
| Insert (12) | 5 10 [12] | 5 10 [12] | “ |
| Insert (3) | [3] 5 10 12 | [3] 5 10 12 | “ |
| GoToNext() | 3 [5] 10 12 | 3 [5] 10 12 | “ |
| AtCursor() | 5 | 5 | “ |
| GoToNext() | 3 5 [10] 12 | 3 5 [10] 12 | “ |
| AtCursor() | 10 | 10 | “ |
| GoToNext() | 3 5 10 [12] | 3 5 10 [12] | “ |
| AtCursor() | 12 | 12 | “ |
| GoToNext() | [3] 5 10 12 | [3] 5 10 12 | “ |
| AtCursor() | 3 | 3 | “ |

Test Case 13 – Clear Test 1

|  |  |  |  |
| --- | --- | --- | --- |
| Date/Time: | Expected Result | Actual Result | Action needed (Yes/No) |
| Insert (5) | [5] | [5] | No |
| Insert (7) | 5 [7] | 5 [7] | “ |
| Insert (1) | [1] 5 7 | [1] 5 7 | “ |
| Insert (3) | 1 [3] 5 7 | 1 [3] 5 7 | “ |
| ClearList() | (empty) | (empty) | “ |

Test Case 14 – Clear Test 2

|  |  |  |  |
| --- | --- | --- | --- |
| Date/Time: | Expected Result | Actual Result | Action needed (Yes/No) |
| Insert (-100) | [-100] | [-100] | No |
| Insert (200) | -100 [200] | -100 [200] | “ |
| Insert (-500) | [-500] -100 200 | [-500] -100 200 | “ |
| ClearList() | (empty) | (empty) | “ |
| Insert (1000) | [1000] | [1000] | “ |
| Insert (-250) | [-250] 1000 | [-250] 1000 | “ |
| Insert(0) | -250 [0] 1000 | -250 [0] 1000 | “ |
| ClearList() | (empty) | (empty) | “ |

Test Case 15 – Empty Test 1

|  |  |  |  |
| --- | --- | --- | --- |
| Date/Time: | Expected Result | Actual Result | Action needed (Yes/No) |
| Empty() | true (1) | true (1) | No |
| Insert (10) | [10] | [10] | “ |
| Insert (20) | 10 [20] | 10 [20] | “ |
| Insert (30) | 10 20 [30] | 10 20 [30] | “ |
| Empty() | false (0) | false (0) | “ |

Test Case 16 – Empty Test 2

|  |  |  |  |
| --- | --- | --- | --- |
| Date/Time: | Expected Result | Actual Result | Action needed (Yes/No) |
| Empty() | true (1) | true (1) | No |
| Insert (10) | [10] | [10] | “ |
| Insert (20) | 10 [20] | 10 [20] | “ |
| Insert (30) | 10 20 [30] | 10 20 [30] | “ |
| Empty() | false (0) | false (0) | “ |
| ClearList() | (empty) | (empty) | “ |
| Empty() | true (1) | true (1) | “ |

Test Case 17 – Full Test 1

|  |  |  |  |
| --- | --- | --- | --- |
| Date/Time: | Expected Result | Actual Result | Action needed (Yes/No) |
| Full() | false (0) | false (0) | No |
| Insert (10) | [10] | [10] | “ |
| Insert (20) | 10 [20] | 10 [20] | “ |
| Insert (30) | 10 20 [30] | 10 20 [30] | “ |
| Full() | false (0) | false (0) | “ |

Test Case 18 – Full Test 2

|  |  |  |  |
| --- | --- | --- | --- |
| Date/Time: | Expected Result | Actual Result | Action needed (Yes/No) |
| Full() | false (0) | false (0) | No |
| Insert (0..9999) | 0 1 2 3 . . . [9999] | 0 1 2 3 . . . [9999] | “ |
| Full() | false (0) | false (0) | “ – Still plenty of memory available, apparently |

Test Case 19 – Assignment Test 1

|  |  |  |  |
| --- | --- | --- | --- |
| Date/Time: | Expected Result | Actual Result | Action needed (Yes/No) |
| testList.Insert (10) | [10] | [10] | No |
| testList.Insert (20) | 10 [20] | 10 [20] | “ |
| << testList | 10 [ 20] | 10 [ 20] | “ |
| assignedList = testList | assignedList = 10 [20] | assignedList = 10 [20] | “ |
| << assignedList | 10 [20] | 10 [20] | “ |
| assignedList.Insert (15) | 10 [15] 20 | 10 [15] 20 | “ |
| << testList | 10 [20] | 10 [20] | “ |
| << assignedList | 10 [15] 20 | 10 [15] 20 | “ |

Test Case 20 – Assignment Test 2

|  |  |  |  |
| --- | --- | --- | --- |
| Date/Time: | Expected Result | Actual Result | Action needed (Yes/No) |
| testList.Insert (10) | [10] | [10] | No |
| testList.Insert (20) | 10 [20] | 10 [20] | “ |
| << testList | 10 [ 20] | 10 [ 20] | “ |
| assignedList = testList | assignedList = 10 [20] | assignedList = 10 [20] | “ |
| << assignedList | 10 [20] | 10 [20] | “ |
| testList.Remove(10) | [20] | 10 [20] | Yes – 10 not found because cursor is at 20 |
| assignedList.Insert (5) | [5] 10 20 | [5] 10 20 | No |
| << testList | [20] | 10 [20] | Yes |
| << assignedList | [5] 10 20 | [5] 10 20 | No |

|  |  |  |  |
| --- | --- | --- | --- |
| Date/Time: | Expected Result | Actual Result | Action needed (Yes/No) |
| testList.Insert (10) | [10] | [10] | No |
| testList.Insert (20) | 10 [20] | 10 [20] | “ |
| << testList | 10 [ 20] | 10 [ 20] | “ |
| assignedList = testList | assignedList = 10 [20] | assignedList = 10 [20] | “ |
| << assignedList | 10 [20] | 10 [20] | “ |
| testList.GoToBeginning() | [10] 20 | [10] 20 | “ |
| testList.Remove(10) | [20] | [20] | “ |
| assignedList.Insert (5) | [5] 10 20 | [5] 10 20 | “ |
| << testList | [20] | [20] | “ |
| << assignedList | [5] 10 20 | [5] 10 20 | “ |

Test Case 21 – Copy Constructor Test 1

|  |  |  |  |
| --- | --- | --- | --- |
| Date/Time: | Expected Result | Actual Result | Action needed (Yes/No) |
| testList.Insert (10) | [10] | [10] | No |
| testList.Insert (20) | 10 [20] | 10 [20] | “ |
| << testList | 10 [ 20] | 10 [ 20] | “ |
| LL<int> copyList = testList | copyList = 10 [20] | copyList = 10 [20] | “ |
| << copyList | 10 [20] | 10 [20] | “ |
| copyList.Insert (15) | 10 [15] 20 | 10 [15] 20 | “ |
| << testList | 10 [20] | 10 [20] | “ |
| << copyList | 10 [15] 20 | 10 [15] 20 | “ |

Test Case 22 – Copy Constructor Test 2

|  |  |  |  |
| --- | --- | --- | --- |
| Date/Time: | Expected Result | Actual Result | Action needed (Yes/No) |
| testList.Insert (10) | [10] | [10] | No |
| testList.Insert (20) | 10 [20] | 10 [20] | “ |
| << testList | 10 [ 20] | 10 [ 20] | “ |
| LL<int> copyList = testList | copyList = 10 [20] | copyList = 10 [20] | “ |
| << copyList | 10 [20] | 10 [20] | “ |
| testList.GoToBeginning() | [10] 20 | [10] 20 | “ |
| testList.Remove(10) | [20] | [20] | “ |
| copyList.Insert (5) | [5] 10 20 | [5] 10 20 | “ |
| << testList | [20] | [20] | “ |
| << copyList | [5] 10 20 | [5] 10 20 | “ |

Name: Michael Beaver

Course: CS 355

Semester: Fall 2012

Assignment Number: 3

Assignment Type: Homework 3 - Analysis

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

Assignment Due Date: Thursday, September 13, 2012 (precisely at 12:30 p.m.)

To Be Included in Portfolio: YES

Question 1: Consider the runtime of GoToNext vs. GoToPrevious routines. State the runtime of each. Follow the statement of each with a justification. Discuss why they are the same or why they are different.

|  |  |
| --- | --- |
| Code | Time Units & Reasons |
| template <class T> |  |
| void LL<T>::GoToNext() { |  |
|  |  |
| // Empty list |  |
| if (this->Empty()) | 2: Comparison in method, returns 1 or 0 |
| this->cursor = NULL; | 1: Assignment |
|  |  |
| // Cursor is at last item |  |
| else if ((this->cursor)->next == NULL) | 1: Equality comparison |
| this->cursor = this->head; | 1: Assignment |
|  |  |
| // Cursor is not at last item |  |
| else |  |
| this->cursor = (this->cursor)->next; | 1: Assignment |
|  |  |
| } |  |
|  | Runtimes: 3, 2, 1 |
|  | Overall: O(3) => O(1) (by constant rule) |

GoToNext is a simple algorithm that relies on three conditional statements, simple operations, and a method call.

The first conditional statement calls the function Empty(), which performs a comparison and returns a Boolean value. Inside Empty(), the first branch of the conditional statement has a runtime of two. The default return of the method has a runtime of one (i.e., returns false). In GoToNext, the executed statement of the first conditional branch is a simple assignment operation; it has a runtime of one. Therefore, the combined runtime of the first conditional statement of GoToNext is three time units.

The second conditional statement performs an equality comparison, so it has a runtime of one. If executed, the statement will also have a runtime of one because it is a simple assignment operation. Therefore, the combined runtime of the second conditional statement is two time units.

The third conditional statement performs no conditional checks. Rather, if executed, the statement will have a runtime of one because it is a simple assignment operation. Therefore, the combined runtime of the last conditional statement is one time unit.

The worst-case scenario is obviously the first conditional statement, which has a runtime of three time units. However, by the rules of Big-O, the constant 3 may be reduced to one. Thus, the worst-case scenario runtime of GoToNext is one time unit. Hence, GoToNext is O(1).

|  |  |
| --- | --- |
| Code | Time Units & Reasons |
| template <class T> |  |
| void LL<T>::GoToPrev() { |  |
|  |  |
| // Empty list |  |
| if (this->Empty()) | 2: Comparison in method, returns 1 or 0 |
| this->cursor = NULL; | 1: Assignment |
|  |  |
| // Temporary Node used to traverse LinkedList |  |
| Node<T> \*current = this->head; | 1: Allocation and instantiation |
|  |  |
| // Cursor is at head |  |
| if (current == this->cursor) | 1: Equality comparison |
| this->placeCursor((this->size)-1); | N+1: placeCursor() is O(N) + 1 subtraction operation |
|  |  |
| // Cursor is not at head |  |
| else { |  |
|  |  |
| // Traverse the list |  |
| for (int i = 0; i < this->size; i++) { | N: for-loop iterates N (this->size) times |
|  |  |
| // Place the cursor at the previous Node |  |
| if (current == this->cursor) { | 1: Equality comparison |
|  |  |
| this->placeCursor(i-1); | N+1: placeCursor() is O(N) + 1 subtraction operation |
| break; |  |
|  |  |
| } |  |
|  |  |
| current = current->next; | 1: Assignment |
|  |  |
| } |  |
|  |  |
| } |  |
|  |  |
| } |  |
|  | Runtimes: 3, 1, N+2, N(N+2), 1 |
|  | Overall: O(N2 + 2N) => O(N2) |

GoToPrev is more complex than GoToNext. Unfortunately, the implementation of GoToPrev is quadratic in its worst-case scenario.

Like GoToNext, GoToPrev has many simple statements that each execute at one time unit. The quadratic nature of the algorithm arises via a nested for-loop concealed by a method call to placeCursor(). The method placeCursor() is defined with several simple statements that each execute at one time unit. However, there is a for-loop in the method that executes N times. Hence, placeCursor is O(N), or linear. Furthermore, an added operation (i.e., subtraction) before the parameter is passed an argument from GoToPrev to placeCursor() raises the combined runtime to N+1. Thus, the conditional statement that checks if the cursor is at the head has a combined runtime of N+2.

The N+2 runtime is irrelevant when the final conditional statement branch is considered. If the conditional statement is entered and executed, a for-loop is executed N-times. Inside the for-loop, there is a conditional statement that performs an equality comparison, which has a runtime of one. If the condition is true, then a method call is executed. The method call is to placeCursor(), which is O(N) in its worst case. The added subtraction operation before calling placeCursor() increases the combined runtime to N+1. (The simple assignment operation outside of the conditional statement may be safely ignored because it is simply constant.) By the rules of complexity, the combined runtime of the for-loop is N(N+2), or N2 + 2N. This is the worst-case scenario. Hence, GoToPrev is O(N2 + N), or simply O(N2). Thus, GoToPrev is a quadratic method.

Obviously GoToNext’s runtime is preferable because O(1) is less complex than O(N2). The methods’ runtimes differ because GoToNext follows a simpler algorithm than GoToPrev. GoToNext has the luxury of following the next pointers in the Node objects. GoToPrev, however, has to loop through the LinkedList to find the previous Node. Hence, more complex operations are needed for GoToPrev’s algorithm. If the LinkedList was not singly-linked (i.e., doubly-linked), then GoToPrev could be as simple as GoToNext. In fact, GoToPrev could just follow the “previous” pointers in the Node objects. In theory, the GoToPrev method of a doubly-linked list would be O(1), like GoToNext. However, the current implementation of GoToPrev is O(N2) because of the extra operations and computations needed to successfully find the previous Node in a singly-linked list.

Indeed, the method call to placeCursor() adds complexity. If the call to placeCursor() was replaced with a statement moving the cursor pointer to the head Node, then a simple for-loop could be used to traverse the LinkedList. In such a case, GoToPrev would be linear in its worst-case scenario because there would only be one for-loop executing N-times, as opposed to a concealed, nested for-loop. Hence, a more optimized version of GoToPrev as described would be O(N), or linear. A linear method is less complex—and while not necessarily ideal, it is preferable—than a quadratic method.

Question 2: Consider the runtime of GoToBeginning vs. GoToEnd routines. State the runtime of each. Follow the statement of each with a justification. Discuss what change(s) could be made to the class to make GoToEnd a more efficient routine. Give a justification for each change.

|  |  |
| --- | --- |
| Code | Time Units & Reasons |
| template <class T> |  |
| void LL<T>::GoToBeginning() { |  |
|  |  |
| // No list, set cursor to NULL |  |
| if (this->Empty()) | 2: Comparison in method, returns 1 or 0 |
| this->cursor = NULL; | 1: Assignment |
|  |  |
| // Move to the beginning list[0] |  |
| else |  |
| this->placeCursor(0); | N: placeCursor() is O(N) |
|  |  |
| } |  |
|  | Runtimes: 3, N |
|  | Overall: O(N) |

GoToBeginning is a seemingly simple method that has two conditional statements, one assignment operation, and two method calls.

The first conditional statement makes a method call to Empty(), which performs at most two simple operations—an equality comparison and returning true or false. If the conditional assessment passes, the assignment statement is executed; this execution has a simple runtime of one. Thus, the combined runtimes of the conditional statement is three time units.

The second conditional statement branch, which has no conditional assessment, has only one executable statement. This statement is a method call to placeCursor(), which has been discussed as being O(N). (The memory operations of pass-by-copy should be considered for spatial complexity’s sake. However, this analysis is only concerned with temporal complexity.) Thus, the second conditional statement has a runtime of N time units, due to the runtime of placeCursor().

The worst-case scenario of GoToBeginning is a runtime of N, which is derived from placeCursor(). Hence, GoToBeginning is O(N). Linearity is not terribly complex, but it is not necessarily ideal. This implementation makes an unnecessary method call to placeCursor(). This algorithm could be simple by replacing the method call to placeCursor() with a single assignment statement (i.e., this->cursor = this->head). If this seemingly minor change is made, then GoToBeginning would instantly assumes a runtime of one; hence, GoToBeginning would be O(1). Therefore, this implementation of GoToBeginning, while accurate and making use of class methods, is not the most efficient implementation. Indeed, this implementation of GoToBeginning is needlessly complex.

|  |  |
| --- | --- |
| Code | Time Units & Reasons |
| template <class T> |  |
| void LL<T>::GoToEnd() { |  |
|  |  |
| // No list, set cursor to NULL |  |
| if (this->Empty()) | 2: Comparison in function, returns 1 or 0 |
| this->cursor = NULL; | 1: Assignment |
|  |  |
| // Move to end, list[size-1] |  |
| else |  |
| this->placeCursor((this->size)-1); | N+1: placeCursor() is O(N) + 1 subtraction operation |
|  |  |
| } |  |
|  | Runtimes: 3, N+1 |
|  | Overall: O(N + 1) => O(N) |

GoToEnd is also a seemingly simple method that has two conditional statements, one assignment operation, and two method calls.

The first conditional statement of GoToEnd is identical to the first conditional statement of GoToBeginning. Also, the executed statement (assignment operation) is identical. Hence, the combined runtimes of the first conditional statement is three time units.

Like GoToBeginning, the second conditional statement branch of GoToEnd has a method call to placeCursor(). However, instead of the method call having a runtime of N, it has a runtime of N+1 because of the added subtraction operation. Hence, the combined runtime of the second conditional statement is N+1 time units.

The second conditional statement’s runtime is the worst-case scenario of GoToEnd. Hence, GoToEnd is O(N+1), or simply O(N). If placeCursor() was removed, then it would be possible to eliminate one source of linearity. The linearity of GoToEnd is needlessly complex. The implementation could be simplified by modifying the class.

Rather than having a strictly singly-linked list, a doubly-linked list would be more convenient to traverse. The availability of next and previous Node pointers in the Node objects would make moving the cursor down the list, depending on the situation, a matter of simple statements. More relevantly, one could replace the method call to placeCursor() in GoToEnd with a simple statement to reassign the cursor pointer to point at the head of the list. Then, another simple reassignment of the cursor pointer to the “previous” Node (i.e., the last Node) would achieve the desired result. These simple statements would combine for a runtime of at least two. Hence, the modified GoToEnd method of the modified LinkedList and Node classes would be O(1). This is a great improvement over the original linear runtime restrictions posed by the limits of a singly-linked list. Therefore, if the LinkedList class and Node class are modified to allow for doubly-linked Nodes (i.e., next and previous Node pointers), the GoToEnd method could be simplified to be O(1). Thus, a doubly-linked list implementation of the LinkedList class is preferable to a singly-linked list implementation of the class, with respect to the cursor methods.

Of course, if a singly-linked list is desired, it would be advantageous to create a data member to point to the last Node in the LinkedList. This pointer, “tail,” would be updated each time a Node is added to the end of the LinkedList and it would point to head by default. If one wanted the LinkedList to be circular, the tail Node’s next pointer could point to the head Node. Of course, the tail Node’s next pointer could also be set to NULL for a linear list. When a new Node is inserted at the end of the LinkedList, the tail pointer would be reassigned to point to the new last Node. More relevantly, in the GoToEnd method, the method call to placeCursor() could be replaced with a simple assignment operation to reassign the cursor to point at the tail Node (i.e., this->cursor = this->tail). This implementation would eliminate the need for a method call to placeCursor(). By replacing the complex method call to placeCursor() with a simple assignment operation to reassign the cursor pointer, the complexity of GoToEnd may be reduced. Indeed, the worst-case scenario of GoToEnd would be reduced from N time units to one time unit. Hence, GoToEnd would be O(1). Therefore, if any changes were to be made to the class, a tail pointer to the last Node would be ideal.

Question 3: When you create a temporary pointer to traverse a list, what is the danger in allocating Node space referred to by the pointer when declaring the pointer?

When allocating space, it is possible that there may be no memory available to the program, which results in a bad allocation. To counter this, a try-catch block should be implemented. In the try-block, a Node pointer or a Node object would be declared and instantiated. If a bad allocation exception is thrown, the catch-block should catch it. At that point, the error could be output to the screen. Alternatively, a method (e.g., Full()) would return true if there was a bad allocation or false if there was no bad allocation. Because memory is presumably unavailable to the program to allocate space for the Node, it would make sense for a method like Full() to return true, indicating that the LinkedList is full. If the try-block does not encounter a bad allocation, it would be reasonable to assume that the allocation of the Node was successful; thus, a method like Full() would return false, which indicates memory is still available for allocation.

Bad allocations aside, it is possible that a Node pointer, when declared, is not given any place to point to (i.e., NULL). If the pointer points nowhere, then it is deadweight memory. To remedy this, the pointer should be given a place to point to at instantiation or as soon as possible. If a user tries to access data from a dereferenced Node pointer that is NULL, there will be an exception thrown. Hence, when traversing a list, it is important to instantiate a Node pointer to actually point at a Node in the LinkedList, such as head or cursor. If the Node pointer is not instantiated to point anywhere, it should be assigned to point to a location. Otherwise, any attempt to access data or the next pointer from that NULL Node pointer will result in exceptios. Thus, any Node pointers used to traverse a list should be instantiated to point to a location or assigned to point to a location as soon as possible.

A NULL Node pointer could potentially wreck an entire algorithm; indeed, it could complicate the debugging process. The inherent difficulty of debugging dynamic memory only complicates matters. If a Node pointer is not instantiated or assigned an object to point at, then the user could experience an exception when trying to dereference the pointer and access the object’s data members. More relevantly, if a temporary Node pointer is declared to traverse a LinkedList, it should be instantiated to point at a Node object in the list. Otherwise, when the user tries to traverse the LinkedList by following the temporary pointer, an exception will be thrown. Thus, temporary Node pointers should be instantiated to point to a Node object in the LinkedList, or they should at least be assigned a Node object to point at.

If the pointer is not instantiated, the user risks an exception being thrown. Exceptions are not user-friendly, and they are the sign of poorly tested code. Moreover, if a user tries to debug a method that traverses a list, if that method uses a NULL Node pointer, then the debugging process could be arduous and frustrating. A NULL Node pointer points to the object at 0x00000000; hence, it points to nothing. If the pointer points to nothing, then there are no data members or methods because there is no object. Hence, data members cannot be accessed when traversing the LinkedList. Therefore, a NULL Node pointer is undesirable when traversing a LinkedList. All temporary Node pointers used to traverse a LinkedList should be instantiated with a Node object to point at, or they should at least be assigned a Node object at which to point.