Colleen Lemak

Professor Bowers

CPSC 223

1 October 2021

HW2 Unit Test Descriptions

EmptyListSize: This test checks if list is empty by calling the empty and size functions, reporting back the appropriate pass or fail result.

EmptyListContains: Reports back if the empty list contains the number 10 or not, which should always be false in an empty list.

EmptyListMemberAccess: If invalid indexes are being passed into the lists, this test checks for an out-of-range throw and tells the user if they need to include one.

AddAndCheckSize: Tests to make sure you are reporting back the correct size by inserting the integer size number, incremented from 1 to 5, while also checking if the list is empty, in which case it returns false because the list is empty.

AddAndCheckContains: Inserts characters into a list and tests if each letter is in the list using the contains function.

OutOfBoundsInsertIndexes: Creates a list of doubles where three doubles are inserted, and the test checks if there are invalid indexes entered for the called insert function such as -1. If so, the test says you must thow an out-of-range because the index was not valid.

EraseAndCheckSize: Tests the size is correct with a list of strings throughout, erases a value at a specified index and tests if the value still exists after being deleted (tests 4 cases of indexing).

EraseAndCheckContains: Checks to make sure list contains what has been previously inserted, and then erases that value and checks it is erased, then calls contains again to make sure erase worked properly. Double checks that the list is empty after deleting all values.

OutOfBoundsEraseIndexes: Makes sure that if you enter valid indexes with the erase function, there is not a throw of out-of-range in effect, but that if the index is out of range, that there is a throw warning if not implemented.

DestructorNoThrowChecksWithNew: Deletes new list made and says there should not be a throw because that is a valid line. Tests other cases with insert, checking that there are not throws occurring after creating a new list, inserting characters, and deleting the list.

CopyConstructorChecks: Creates two lists and checks their sizes, as one calls copy for the other. Checks the same list after copy and checks different lists after deleting an item from one and not the other.

MoveConsructorChecks: Uses multiple lists to test if it was a successful move for an empty list and one element list. Makes sure the value was inputted correctly using contains, checking it against the original list.

CopyAssignmentOpChecks: Add characters to list1, where it checks if size and the contents are correct. Then, we check if assigning a list to itself changes anything, which it shouldn't, and end by adding items to list2, and assigning it to list1. This will check if they are now the same in size and contents. Then it checks if changing one list changes the other list, making sure they are not the same list, or entangled together.

// ignore CheckRValueAccess

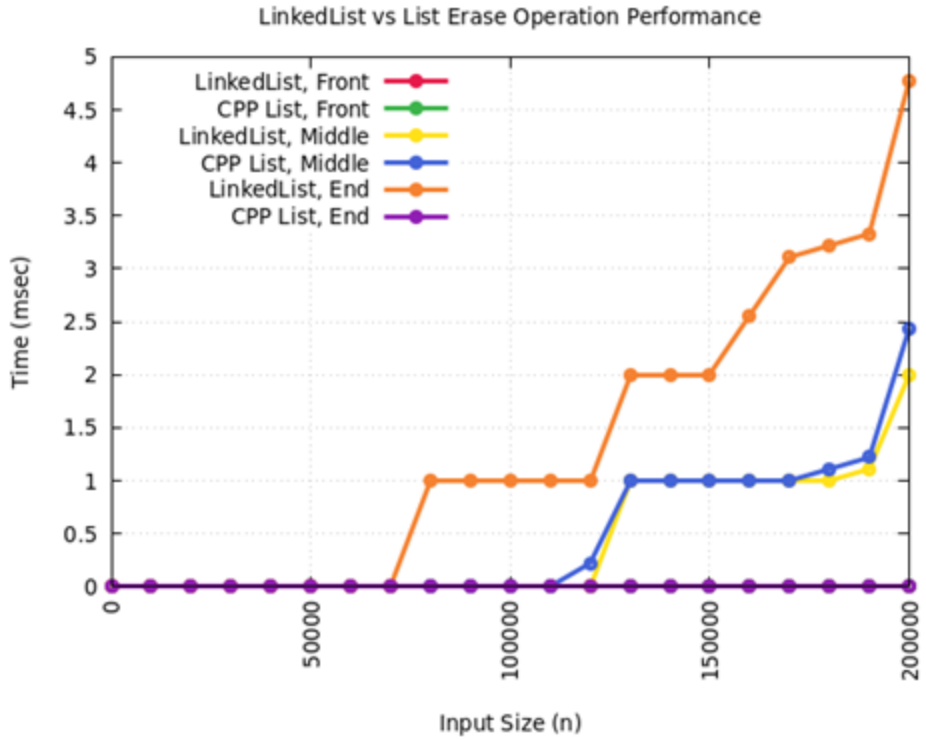
CheckLValueAccess: After assigning a list values, checks for the correct insertions at the proper index with corresponding values.

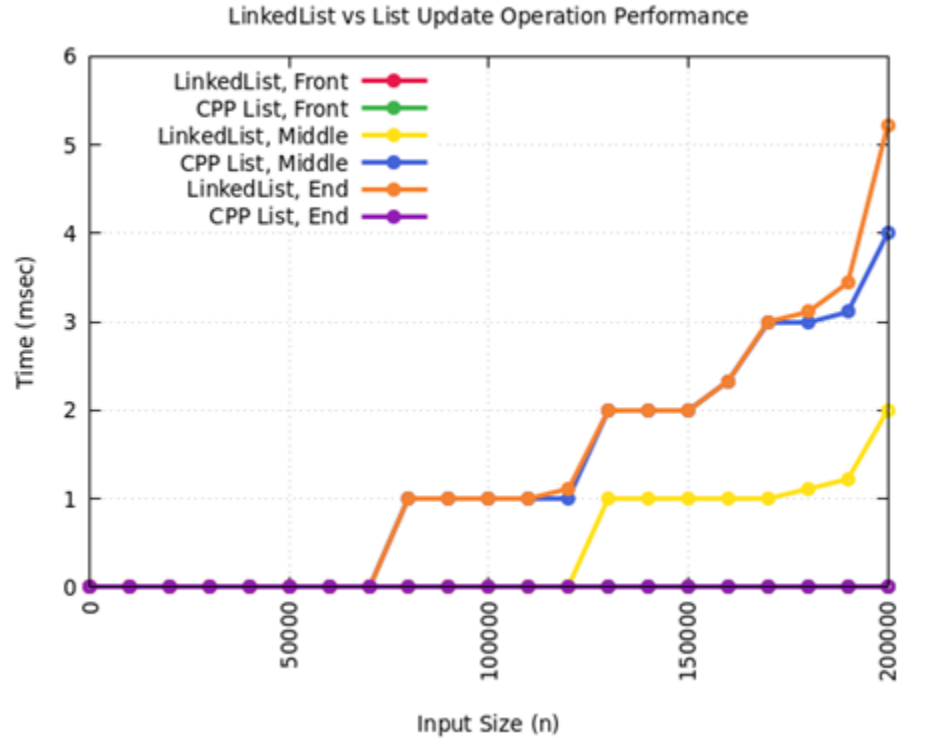
CheckConstRValueAccess: Inserts values and calls the copy constructor and checks if everything copied over properly.

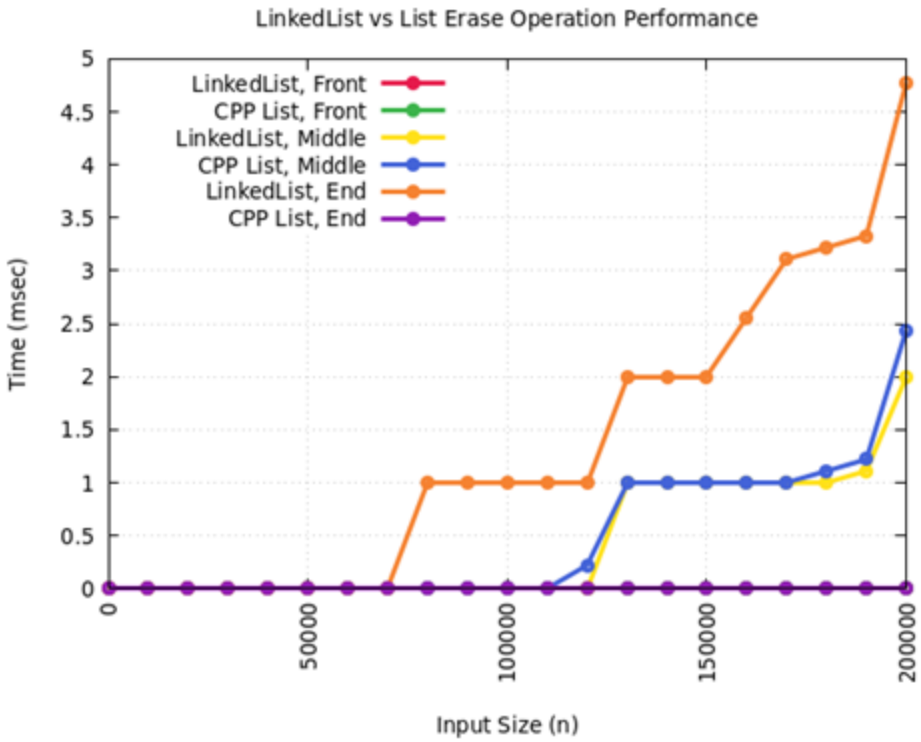
OutOfBoundsLValueAccess: Checks to make sure there is a throw out-of-range call when there is an invalid index passed in for the left hand side after creating a list of characters.

OutOfBoundsRValueAccess: Checks to make sure there is a throw out-of-range call when there is an invalid index passed in for the right hand side after creating a list of characters. Uses tmp as a space to check the throw occurs.

StringInsertionChecks: Checks to make sure the output is desired. Expects no spaces after a one integer is entered, but a ", " when more than one number is being dealt with.


LinkedList vs List Erase Operation Performance
LinkedList, Front
CPP List, Front
LinkedList, Middle
CPP List, Middle
LinkedList, End
CPP List, End
Time (msec)
Input Size (n)
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
0
50000
100000
150000
200000


LinkedList vs List Update Operation Performance
LinkedList, Front
CPP List, Front
LinkedList, Middle
CPP List, Middle
LinkedList, End
CPP List, End
Time (msec)
Input Size (n)
0
1
2
3
4
5
6
0
50000
100000
150000
200000



I ran into some code-based issues as I usually know conceptually what to do, but implementing it into code is more difficult. Office hours are helpful for me as I can understand things in a better setting.

From my graphs, there are mostly spikes in time taken, specifically when inserting in the middle, and when erasing at the middle of the Linked List. I used the tail pointer to minimize the time taken to traverse through the list, and with the purple line, it takes virtually no time to insert or erase at that position, but the LinkedList spikes in time for this test. For deleting and inserting, it takes more time in the middle of a list than the start or end because you have to check every position in order to get the right index to insert in. I used two node pointers to keep track of the positions, which helped the functions run properly. Overall, my data passed the tests, but it is harder to interpret without what seems like a fully filled performance graph.