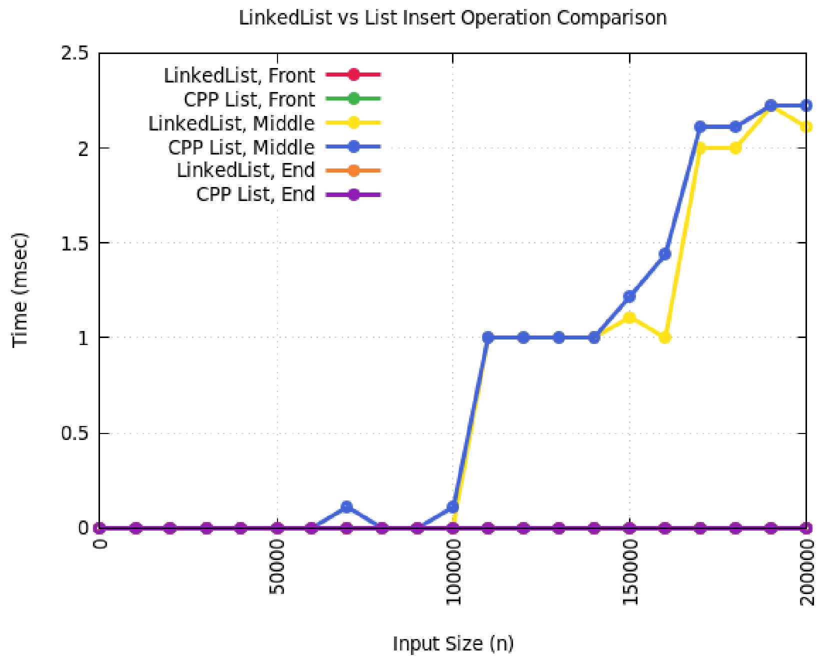
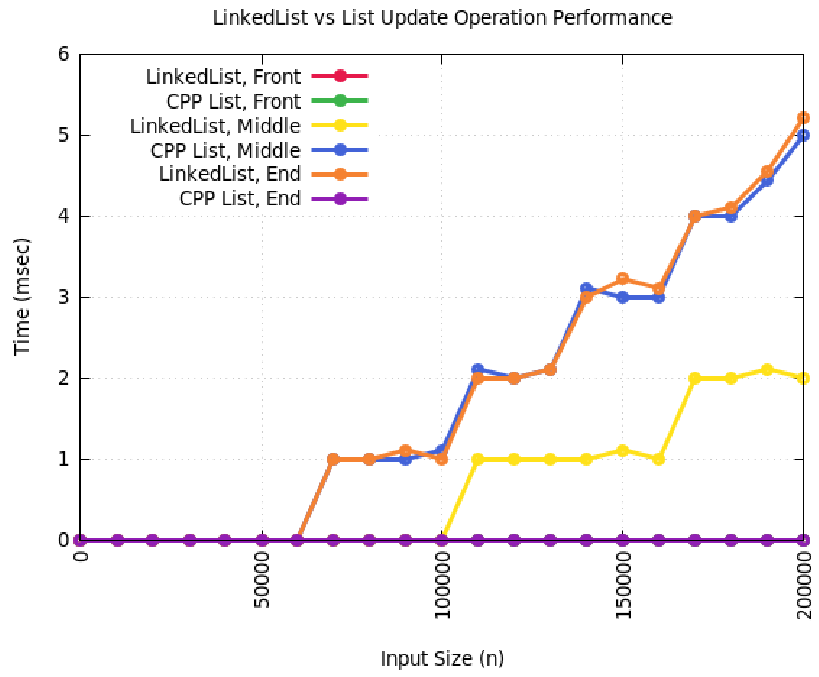
HW2 Writeup (Spring 2022)

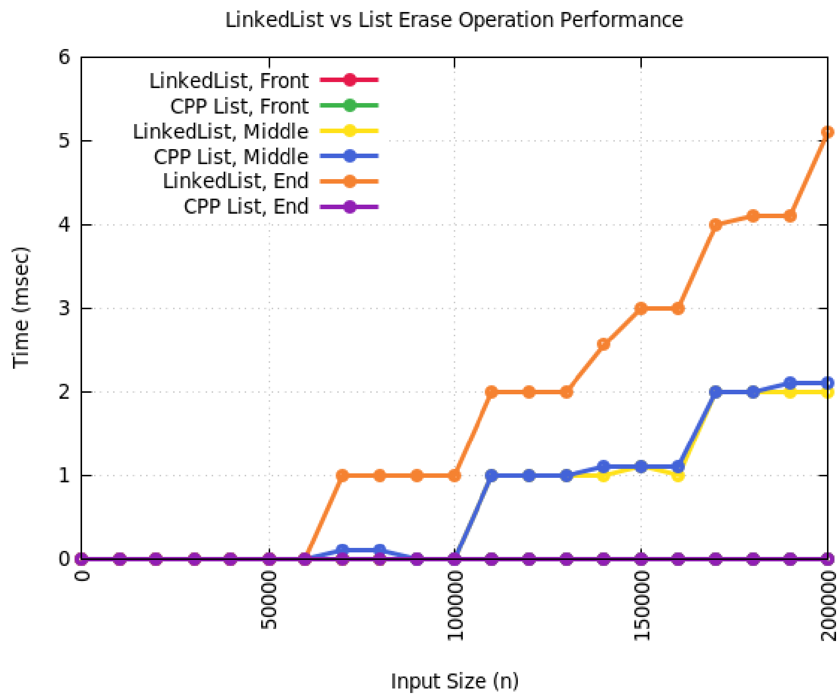
Plots



This first graph shows a comparison between inserting a node into my *LinkedList* implementation at different points. Also, that plot was supplemented with results generated by utilizing the built-in *List* class to perform the same operation at identical points in the list. As my implementation of a *LinkedList* utilizes a head and tail pointer the access each end of the list, it is understandable that inserting at the middle for both structures takes longer than at other points in said lists.



The second graph above illustrates the time complexity of updating nodes at different points in both a *LinkedList* structure and a basic *List* structure. While accessing elements in the middle also takes the most time for updating nodes, updating at the end of the *LinkedList* appears to take a relatively equal amount of time as updating a cpp list in the middle. The end update may take longer due to my implementation of access operators lacking a check for tail index and head index list positions.



Again, erasing from the end takes an excess amount of time. While I have added a condition to check for the end of the *LinkedList*, it seems that it still is taking a greater amount of time to erase a node at the end than the other points in the list. As for the middle erase complexity, this represents a need to iterate to a desired index every time the function is called; an action that adds onto the cost of the erase function when called.

Tests

EmptySeqSize: Checks if the list is truly empty at declaration, then checks if the size, meaning the node count, is set to the correct value too.

EmptySeqContains: Checks to see if a value that should not be present in the list is currently not in said list.

EmptySeqMemberAccess: Calls the overridden array access operator for a left-hand side instance to check that an out-of-range error is thrown if an invalid index is requested.

AddAndCheckSize: Determines if the insert function works correctly and that the empty function returns the correct value of false given a list populated with nodes.

AddAndCheckContains: Inserts character values into the sequence at existing index positions to determine if the new value updates the value currently in the index rather than being added to the list again.

OutOfBoundsInsertIndexes: Checks to see than an error is thrown if a value is called to be placed at an invalid index.

EraseAndCheckSize: Inserts string values into a sequence and checks the size, empty, and contains functions affirming that they return proper values according to the contents of the list.

EraseAndCheckContains: Utilizes the erase function to delete nodes in a list, then checks to make sure the element was deleted.

OutOfBoundsEraseIndexes: Calls upon the erase function to delete a value at some index values that exist and others that do not to determine if an error is thrown.

CheckClear: Inserts character values into a linked sequence, calls clear, and checks to make sure the node count was updated through the size function.

DestructorNoThrowChecksWithNew: Weighs the effectiveness of the LinkedSeq destructor with allocating new sequence objects and giving them values (nodes).

CopyConstructorChecks: Checks that the implemented copy constructor adequately copies the contents of one sequence into a new sequence, inserts new values, then performs another copy and checks the final contents.

MoveConstructorChecks: Transfers the nodes from one sequence into a new sequence multiple times, all the while adding a new node to the sequence and checking the size and contains functions for proper updating of member variables.

CopyAssignmentOpChecks: Uses two sequences to compare the application of adding nodes, copying said nodes, and then changing the contents of one sequence without the other changing too.

CheckRValueAccess: Determine whether referencing the content in any given index of the list is correctly done with the access operator.

CheckLValueAccess: Determines that passing an address by reference is completed such that the value held can be changed.

CheckConstRValueAccess: Check that calling the access operator does not modify the value being access, rather, it copies it or can peek at it.

OutOfBoundsLValueAccess: Inserts values into a sequence and attempts to assign values at index positions that do not exist.

OutOfBoundsRValueAccess: Checks that an error is thrown if attempting to access a value at an index that does not exist due to the size of the list.

String InsertionChecks: Inserts a sequence of integers into the string stream with the insertion operator to determine if this function correctly accomplishes its goal.

Challenges

The largest issue I had with completing this assignment was checking for an index at the end of the list in the access operators and the erase function. Meaning, I had trouble improving the time complexity of the performance tests for updating at the end of the list using the access operator(s) and erasing at the end using my erase function. After pushing my code to the GitHub repository for this assignment, I was able to implement a check for an index where the tail pointer points to within the access operators. But even though the time complexity for update at end improved in performance, I was still having issues with erase and overall time complexity spikes on my computer.