Colleen Lemak

Professor Bowers

CPSC 223

7 October 2021

HW #3 Write Up

EmptySeqSize: Tests to make sure it is an empty list

EmptySeqContains: Tests if the empty list contains 10, should be false

EmptySeqMemberAccess: Tests out of range, makes sure you cannot erase, index, or assign anything if the sequence is empty

AddAndCheckSize: Inserts integers into the sequence and tests to make sure size is correct and not empty

AddAndCheckContains: inserts characters and makes sure they are correctly detected if supposed to be in sequence

OutOfBoundsInsertIndexes: Checks that it will throw out of range if you insert at a value that is not valid like too high of an index or a negative one

EraseAndCheckSize: Inserts various characters and erases them and checks size after performing. Tests first, last, and middle case for insert, contains, and erase.

EraseAndCheckContains: Erases a character in sequence and checks before and after if the sequence contains that character as a check if it was erased or not.

OutOfBoundsEraseIndexes: Checks you cannot erase an invalid index's element, but that you can erase a valid one

DestructorNoThrowChecksWithNew: Creates a new sequence and tests cases and makes sure you can delete the sequence without calling a throw

CopyConstructorChecks: Makes sure the constructor copies the sequence into an empty sequence, and that you can make changes to one without also changing the other. Uses size and contains to check.

MoveConstructorChecks: Uses characters to make sure that you correctly moved one sequence into another and that the one you moved is now empty. Insert a character into seq4 and then check that the sequences correctly contain their characters added.

CopyAssignmentOpChecks: Assigns various sequences to other sequences, making sure if set to itself, nothing will change, and that if it is set to another sequence, it now has all of its contents, tested using contains.

CheckRValueAccess: inserts values into sequence and makes sure they were correctly placed.

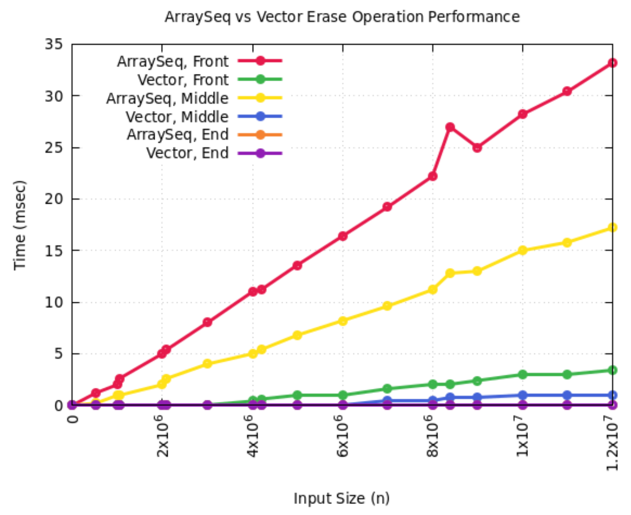
CheckLValueAccess: checks if the index's element was correctly placed.

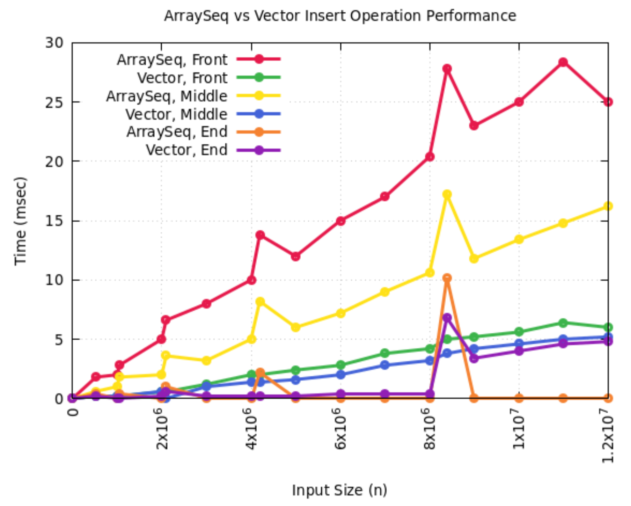
OutOfBoundsLVlaueAccess: checks you are trying to access a valid index for the left hand side, expects throw if not valid

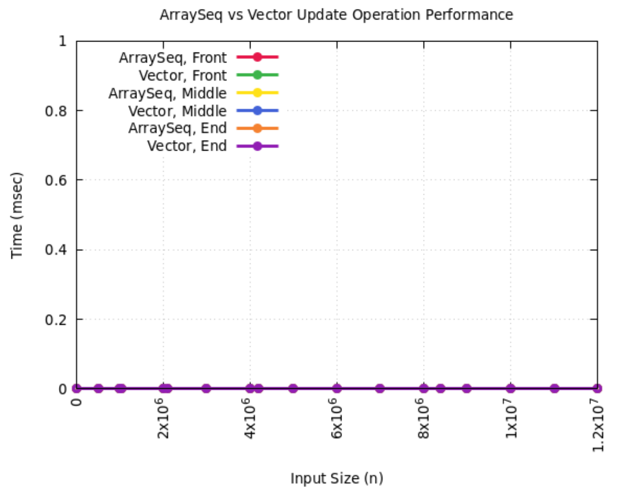
OutOfBoundsRValueAccess: checks you are trying to access a valid index for the right hand side, expects throw if not valid

StringInsertionChecks: checks formatting of string insertion and makes sure it is nothing if empty, just the number if only one, and a comma and a space if there are multiple numbers

ResizeCheck: forces a resize on the sequence and inserts values to check that it is inserted correctly and the size has been updated accordingly.


ArraySeq vs Vector Erase Operation Performance
Time (msec)
Input Size (n)
ArraySeq, Front
Vector, Front
ArraySeq, Middle
Vector, Middle
ArraySeq, End
Vector, End
0
5
10
15
20
25
30
35
0
2x10^6
4x10^6
6x10^6
8x10^6
1x10^7
1.2x10^7


ArraySeq vs Vector Insert Operation Performance
Time (msec)
Input Size (n)
ArraySeq, Front
Vector, Front
ArraySeq, Middle
Vector, Middle
ArraySeq, End
Vector, End
0
5
10
15
20
25
30
0
2x10^6
4x10^6
6x10^6
8x10^6
1x10^7
1.2x10^7


ArraySeq vs Vector Update Operation Performance
Time (msec)
Input Size (n)
ArraySeq, Front
Vector, Front
ArraySeq, Middle
Vector, Middle
ArraySeq, End
Vector, End
0
0.2
0.4
0.6
0.8
1
0
2x10^6
4x10^6
6x10^6
8x10^6
1x10^7
1.2x10^7

The Erase graph had relatively consistent lines when testing ArraySeq over different large input sizes. The vector functions took fewer msec to run, and were typically under five msec. ArraySeq at the beginning of the list took the longest, and spiked around 8x10^6 input size. In Insert, the time spiked around the same time as Erase's ArraySeq at the front. This is likely because in my insert(), I begin from the end of the list and traverse backwards so it takes more time to get to the first item to insert there. Inserting at the middle index takes a little over half the time as for front, which makes sense as it should not change the items that are before the index insert position. Most of the vector functions ran quickly again, and the most consistently expensive run was the vector at the front in green. The update function just details how the vector function at the end did not take much time to run, regardless of input size.

I ran into a few segmentation faults and memory leaks when coding, and I had to run valgrind and gdb; they ended up being because of my move and copy functions.