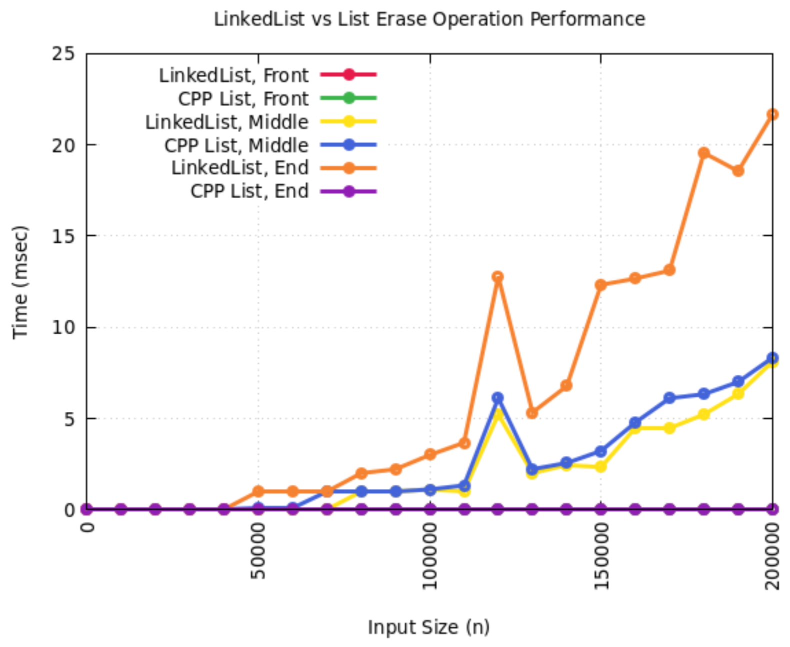
NAME: Ben Puryear

FILE: HW-2_WriteUp.pdf

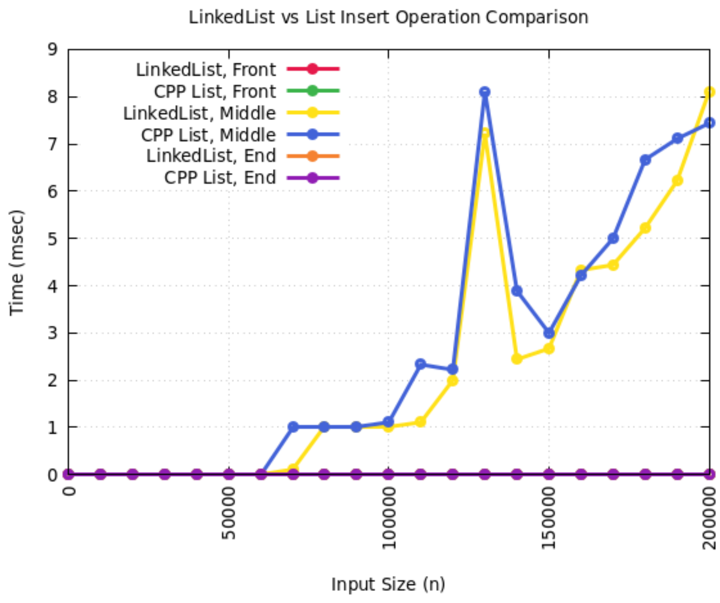
DATE: Fall 2021

DESC: This pdf goes over the basics accomplished throughout all the HW-2 related files.

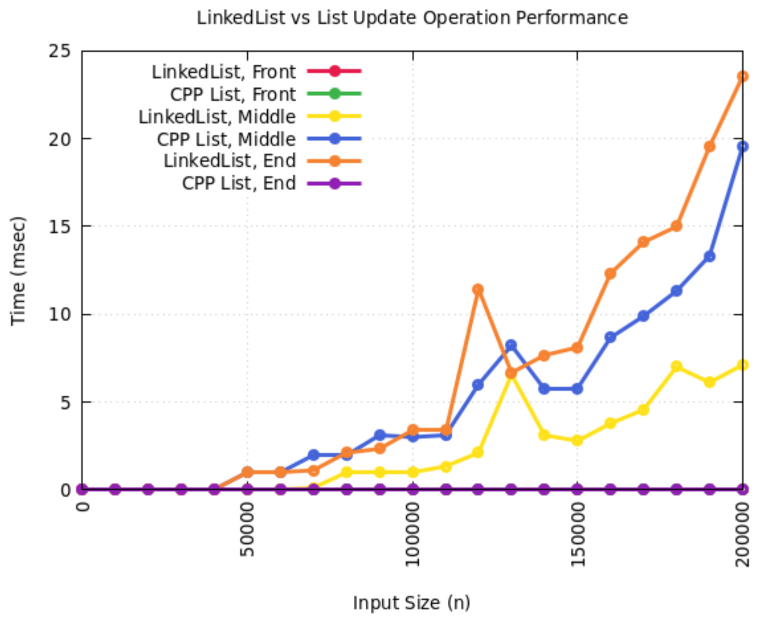
Graphs / Explanations



The first graph is the Erase graph. This graph tests to see on average how long it takes for an erase() to be executed. The graph shows how the LinkedList does not fair well to using erase() on the end of the list. This contrasts with the CPP list, which takes computationally no time to execute. A theory for why this is the case is that the way that my erase() function works is that it traverses the list instead of using the tail pointer. Apart from that, it seems that my linkedlist preforms just the same as the CPP list. The erase() when used on the front is a constant, and when used in the middle, it follows the same structure of movement as the CPP list.



This is the insertion graph. This graph is very similar to the erase graph in the ways where my LinkedList performs just as well as the CPP list, with the only exception being when the action being tested is the last index. The reasoning behind it is also similar to erase(), because my theory on why this is the case is that I never used tail in insert(). Looking back that was a clear mistake for performance, and that definitely affected the amount of time it took to run hw2_perf.



This is the update operation graph. This graph is very similar to the previous two graphs as well. With the tail not being utilized, the LinkedList end statistic is quite high. Something that confuses me is just how well my LinkedList performs in comparison to the CPP list in the middle.

Issues I faced:

The only issue that I faced was the extremely long time it took for the hw2_perf to run. In total it took about an hour to fully complete and log the performance. This was because my first draft did not include the usage of tail during insert. This has now been fixed and it takes around 10 seconds to do a full hw2_perf.

Explanation of all the unit tests:

```
TEST(BasicLinkedSeqTests, EmptyListSize)
{
    LinkedSeq<int> list;
    ASSERT_EQ(true, list.empty());
    ASSERT_EQ(0, list.size());
}
```

This test, EmptyListSize, will test to see if when I call empty() on a new list, it returns 0.

```
TEST(BasicLinkedSeqTests, EmptyListContains)
{
    LinkedSeq<int> list;
    ASSERT_EQ(false, list.contains(10));
}
```

This test, EmptyListContains, will test to see if after creating a new list, it does not contain the number 10. Not only will this test to see that the list does not contain any numbers on creation, it also tests to see if there will be an error when running contains() on an empty list.

```
TEST(BasicLinkedSeqTests, EmptyListMemberAccess)
{
    LinkedSeq<int> list1;
    EXPECT_THROW(list1.erase(0), std::out_of_range);
    EXPECT_THROW(list1[0] = 10, std::out_of_range);
    const LinkedSeq<int> list2;
    EXPECT_THROW(auto x = list2[0], std::out_of_range);
}
```

This test, EmptyListMemberAccess, will test to see if when using erase(0) or trying to use the assignment [] on an empty list, the out of range error is thrown. Following this it also tests the copy [] function to see if that also will throw an out of range error.

```
TEST(BasicLinkedSeqTests, AddAndCheckSize)
{
    LinkedSeq<int> list;
    list.insert(10, 0);
    ASSERT_EQ(1, list.size());
    ASSERT_EQ(false, list.empty());
    list.insert(20, 1);
    ASSERT_EQ(2, list.size());
    ASSERT_EQ(false, list.empty());
    list.insert(30, 2);
    ASSERT_EQ(3, list.size());
    ASSERT_EQ(false, list.empty());
    list.insert(5, 0);
    ASSERT_EQ(4, list.size());
    ASSERT_EQ(false, list.empty());
    list.insert(15, 2);
    ASSERT_EQ(5, list.size());
    ASSERT_EQ(false, list.empty());
}
```

This test, AddAndCheckSize, repeatedly uses insert() on a new list, following each time will check to see if empty() ever returns true.

```
TEST(BasicLinkedSeqTests, AddAndCheckContains)
{
    LinkedSeq<char> list;
    list.insert('b', 0);
    list.insert('d', 1);
    list.insert('c', 1);
    list.insert('a', 0);
    list.insert('f', 4);
    list.insert('e', 4);
    ASSERT_EQ(true, list.contains('a'));
    ASSERT_EQ(true, list.contains('b'));
    ASSERT_EQ(true, list.contains('c'));
    ASSERT_EQ(true, list.contains('d'));
    ASSERT_EQ(true, list.contains('e'));
    ASSERT_EQ(true, list.contains('f'));
    ASSERT_EQ(false, list.contains('g'));
}
```

This test, AddAndCheckContains, will repeatedly add certain characters using insert(). Following that it will test to see if contains() with a character inserted will result in true. Afterwards, the test will use contains() with a character that was not inserted to the list. The test will want the result to be false.

```
TEST(BasicLinkedSeqTests, OutOfBoundsInsertIndexes)
{
    LinkedSeq<double> list;
    list.insert(4.4, 0);
    list.insert(3.3, 0);
    list.insert(2.2, 0);
    EXPECT_THROW(list.insert(1.1, -1), std::out_of_range);
    EXPECT_THROW(list.insert(6.6, 4), std::out_of_range);
}
```

This test, OutOfBOundsInsertIndexes, tests to see if when you use insert() with the index either negative, or >= the size of the list, it throws the out of range error.

```
TEST(BasicLinkedSeqTests, EraseAndCheckSize)
{
    LinkedSeq<string> list;
    list.insert("A", 0);
    list.insert("B", 1);
    list.insert("C", 2);
    list.insert("D", 3);
    ASSERT_EQ(4, list.size());
    ASSERT_EQ(false, list.empty());
    // remove internal
    list.erase(2);
    ASSERT_EQ(3, list.size());
    ASSERT_EQ(false, list.contains("C"));
    // remove head
    list.erase(0);
    ASSERT_EQ(2, list.size());
    ASSERT_EQ(false, list.contains("A"));
    // remove tail
    list.erase(1);
    ASSERT_EQ(1, list.size());
    ASSERT_EQ(false, list.contains("D"));
    // remove last elem
    list.erase(0);
    ASSERT_EQ(0, list.size());
    ASSERT_EQ(true, list.empty());
    ASSERT_EQ(false, list.contains("B"));
}
```

This test, EraseAndCheckSize, starts out by inserting multiple characters into a new list. Then it proceeds to erase specific indexes, afterwards checking to see if size() returns the correct value, as well as testing to see if contains() with the parameter of the character in the erased index returns false.

```
TEST(BasicLinkedSeqTests, EraseAndCheckContains)
{
    LinkedSeq<char> list;
    list.insert('b', 0);
    list.insert('d', 1);
    list.insert('c', 1);
    list.insert('a', 0);
    list.insert('f', 4);
    list.insert('e', 4);
    // erase 'c'
    ASSERT_EQ(true, list.contains('c'));
    list.erase(2);
    ASSERT_EQ(false, list.contains('c'));
    // erase 'f'
    ASSERT_EQ(true, list.contains('f'));
    list.erase(4);
    ASSERT_EQ(false, list.contains('f'));
    // erase 'a'
    ASSERT_EQ(true, list.contains('a'));
    list.erase(0);
    ASSERT_EQ(false, list.contains('a'));
    // erase 'd'
    ASSERT_EQ(true, list.contains('d'));
    list.erase(1);
    ASSERT_EQ(false, list.contains('d'));
    // erase 'b'
    ASSERT_EQ(true, list.contains('b'));
    list.erase(0);
    ASSERT_EQ(false, list.contains('b'));
    // erase 'e'
    ASSERT_EQ(true, list.contains('e'));
    list.erase(0);
    ASSERT_EQ(false, list.contains('e'));
    // ensure empty
    ASSERT_EQ(true, list.empty());
}
```

This test, EraseAndCheckContains, is very similar to the previous test, but instead instead of checking if size() decreases when erase() is used, it also checks to see if contains() with the parameter of the character in the erase() index returns true before erase() is called. After erase() is called, it will also check to see if contains() returns false.

```
TEST(BasicLinkedSeqTests, OutOfBoundsEraseIndexes)
{
    LinkedSeq<double> list;
    list.insert(3.2, 0);
    list.insert(2.4, 0);
    list.insert(1.6, 0);
    EXPECT_THROW(list.erase(-1), std::out_of_range);
    EXPECT_THROW(list.erase(3), std::out_of_range);
    EXPECT_NO_THROW(list.erase(2));
    EXPECT_NO_THROW(list.erase(1));
    EXPECT_NO_THROW(list.erase(0));
}
```

This test, OutOfBoundsEraseIndexes, checks to see that when using erase() with a parameter of an invalid index, the out of range exception is thrown. It also checks to see if when using a valid parameter, erase() does not throw an out of range exception.

```
TEST(BasicLinkedSeqTests, DestructorNoThrowChecksWithNew)
{
    Sequence<char> *list = new LinkedSeq<char>;
    EXPECT_NO_THROW(delete list);
    list = new LinkedSeq<char>;
    list->insert('a', 0);
    EXPECT_NO_THROW(delete list);
    list = new LinkedSeq<char>;
    list->insert('a', 0);
    list->insert('b', 1);
    EXPECT_NO_THROW(delete list);
}
```

This test, DestructorNoThrowChecksWithNew, checks to see that when using delete on a new list, there is no exception thrown. The same is checked on a list that contains 1 value, as well as 2 values.

```
TEST(BasicLinkedSeqTests, CopyConstructorChecks)
{
    // copy empty list
    LinkedSeq<int> list1;
    LinkedSeq<int> list2(list1);
    ASSERT_EQ(list1.size(), list2.size());
    // copy one-element list
    list1.insert(10, 0);
    ASSERT_NE(list1.size(), list2.size());
    list2.insert(10, 0);
    list2.insert(20, 0);
    list2.insert(30, 0);
    LinkedSeq<int> list3(list2);
    ASSERT_EQ(list2.size(), list3.size());
    // remove item and check two lists are different
    list3.erase(1);
    ASSERT_NE(list2.size(), list3.size());
    ASSERT_EQ(true, list2.contains(20));
    ASSERT_EQ(false, list3.contains(20));
}
```

This test, CopyConstructorChecks, checks to see if when using the copy constructor, the two lists are equal. Following that, items are added and removed from each list, and the test checks to see that they are not equal.

```cpp
TEST(BasicLinkedSeqTests, MoveConstructorChecks)
{
    // move empty list
    LinkedSeq<int> list1;
    LinkedSeq<int> list2(move(list1));
    ASSERT_EQ(0, list1.size());
    ASSERT_EQ(0, list2.size());
    // move one-element list
    LinkedSeq<char> list3;
    list3.insert('a', 0);
    ASSERT_EQ(1, list3.size());
    ASSERT_EQ(true, list3.contains('a'));
    LinkedSeq<char> list4(move(list3));
    ASSERT_EQ(0, list3.size());
    ASSERT_EQ(1, list4.size());
    ASSERT_EQ(true, list4.contains('a'));
    // add item to new list (to ensure correct move)
    list4.insert('b', 1);
    ASSERT_EQ(0, list3.size());
    ASSERT_EQ(2, list4.size());
    ASSERT_EQ(true, list4.contains('a') and list4.contains('b'));
}
```

This test, MoveConstructoChecks, is very similar to the previous test, but instead of using the copy constructor, it is using the move constructor.

```cpp
TEST(BasicLinkedSeqTests, CopyAssignmentOpChecks)
{
    LinkedSeq<char> list1;
    LinkedSeq<char> list2;
    // both empty
    list2 = list1;
    ASSERT_EQ(0, list1.size());
    ASSERT_EQ(0, list2.size());
    // add to list1 (shouldn't change list2)
    list1.insert('a', 0);
    list1.insert('b', 1);
    ASSERT_EQ(2, list1.size());
    ASSERT_EQ(0, list2.size());
    ASSERT_EQ(true, list1.contains('a') and list1.contains('b'));
    // assign list 1 to itself (shouldn't change anything)
    list1 = list1;
    ASSERT_EQ(2, list1.size());
    ASSERT_EQ(true, list1.contains('a') and list1.contains('b'));
    // add items to list 2 and assign list2 to list
    list2.insert('c', 0);
    list2.insert('d', 1);
    list2.insert('e', 2);
    list1 = list2;
    ASSERT_EQ(3, list2.size());
    ASSERT_EQ(3, list1.size());
    ASSERT_EQ(true, list1.contains('c') and list2.contains('c'));
    ASSERT_EQ(true, list1.contains('d') and list2.contains('d'));
    ASSERT_EQ(true, list1.contains('e') and list2.contains('e'));
    // add to list1 (shouldn't change list2)
    list1.insert('f', 3);
    ASSERT_EQ(4, list1.size());
    ASSERT_EQ(3, list2.size());
    ASSERT_EQ(true, not list2.contains('f') and list1.contains('f'));
}
```

This test, CopyAssignmentOpChecks, does the same thing as the previous 2 tests, but instead of using the move or copy constructor, it uses the copy assignment operator.

```
TEST(BasicLinkedSeqTests, CheckRValueAccess)
{
    LinkedSeq<int> list;
    list.insert(10, 0);
    list.insert(20, 1);
    list.insert(30, 2);
    ASSERT_EQ(10, list[0]);
    ASSERT_EQ(20, list[1]);
    ASSERT_EQ(30, list[2]);
}
```

This test, CheckRValueAccess, checks to see that the access operator works on values after being inserted into a new list.

```
TEST(BasicLinkedSeqTests, CheckLValueAccess)
{
    LinkedSeq<int> list;
    list.insert(10, 0);
    list.insert(20, 1);
    list.insert(30, 2);
    list[0] = 15;
    list[1] = 25;
    list[2] = 35;
    ASSERT_EQ(15, list[0]);
    ASSERT_EQ(25, list[1]);
    ASSERT_EQ(35, list[2]);
}
```

This test, CheckLValueAccess, does the same thing as the previous test, but before checking the values, it changes them with an LValue assignment. The test then checks to see if the list has the new assigned values in the correct position.

```
TEST(BasicLinkedSeqTests, CheckConstRValueAccess)
{
    LinkedSeq<int> list1;
    list1.insert(10, 0);
    list1.insert(20, 1);
    list1.insert(30, 2);
    const LinkedSeq<int> list2 = list1; // calls copy constructor
    ASSERT_EQ(10, list2[0]);
    ASSERT_EQ(20, list2[1]);
    ASSERT_EQ(30, list2[2]);
}
```

This test, CheckConstRValueAccess, does what the CheckRValueAccess test did, but uses a const list instead.

```
TEST(BasicLinkedSeqTests, OutOfBoundsLValueAccess)
{
    LinkedSeq<char> list;
    list.insert('b', 0);
    list.insert('c', 1);
    list.insert('d', 2);
    EXPECT_THROW(list[-1] = 'a', std::out_of_range);
    EXPECT_THROW(list[3] = 'e', std::out_of_range);
}
```

This test, OutOfBoundsLValueAccess, checks to see that when using the access operator on an invalid index, the out of range exception is thrown.

```
TEST(BasicLinkedSeqTests, OutOfBoundsRValueAccess)
{
    LinkedSeq<char> list;
    list.insert('b', 0);
    list.insert('c', 1);
    list.insert('d', 2);
    char tmp = ' ';
    EXPECT_THROW(tmp = list[-1], std::out_of_range);
    EXPECT_THROW(tmp = list[3], std::out_of_range);
}
```

This test, OutOfBoundsRValueAccess, is the same as the previous test, but instead of using a left side access, it uses a right side access.

```
TEST(BasicLinkedSeqTests, StringInsertionChecks)
{
    LinkedSeq<int> list;
    stringstream strstrm1;
    // check empty list
    strstrm1 << list;
    ASSERT_EQ("", strstrm1.str());
    // check one-item list
    list.insert(10, 0);
    stringstream strstrm2;
    strstrm2 << list;
    ASSERT_EQ("10", strstrm2.str());
    // check two-item list
    list.insert(20, 1);
    stringstream strstrm3;
    strstrm3 << list;
    ASSERT_EQ("10, 20", strstrm3.str());
    // check three-item list
    list.insert(30, 2);
    stringstream strstrm4;
    strstrm4 << list;
    ASSERT_EQ("10, 20, 30", strstrm4.str());
    stringstream strstrm5;
    strstrm5 << list << endl;
    ASSERT_EQ("10, 20, 30\n", strstrm5.str());
    stringstream strstrm6;
    strstrm6 << list << "; " << list;
    ASSERT_EQ("10, 20, 30; 10, 20, 30", strstrm6.str());
}
```

This test, StringInsertionChecks, checks to see if the << steam operator works correctly. This is done with multiple checks with different lists as well as different data types.