Eideen Mozaffari

UID: 105988436

Programming Assignment 2 Report

**Design**

I implemented a circular, doubly-linked list with a “Dummy Node” and head pointer to a Node. I implemented a Node to be a type of struct, where the Node holds a value of ItemType, a pointer to the next Node in the list, and a pointer to the previous Node in the list. So, an empty sequence would look like the following:

Diagram

Description automatically generated

The Dummy Node’s value should be left uninitialized, as we are never going to access it, the next-pointer points to the Dummy Node, and the previous-pointer also points to the Dummy Node, and the head-pointer points to the Dummy Node. When a Sequence is created, all these pointers are initialized as described above, and the size of the sequence is initialized to 0. Note, the Dummy Node does not count as part of the Sequence, as it is only included to make implementation easier.

A populated Sequence would look like the following:

Diagram, schematic

Description automatically generated

With this picture, we can see the circularity of the design. The last object in the Sequence next-pointer points the Dummy Node, and the Dummy Node’s previous-pointer points to the last object in the Sequence. Note, the size of the Sequence is 3, not 4, as we do not count the Dummy Node as part of the Sequence. All in all, this is the implementation I used for my linked list.

**Pseudocode for Non-Trivial Algorithms**

Destructor:

Use pointer to iterate through list

Repeat until traversal pointer’s next object is head:

Update pointer to next object, delete the previous object

Current pointer at last element, delete current pointer

Insert(2 parameters):

If Valid position passed

Save pointer to current node at position

Create new node, initialize value and pointers

Change pointers for nodes before and after insertion spot

Previous node’s next points to new node, initial node previous points to new node

Increment size

Otherwise

Do nothing

Remove:

Use a pointer to traverse list

Until reach the end of list:

Look for value in list

If value found

Move traversal pointer back one node

Erase element at position where value found

Move position in the list back

Increment number of items removed

Return number of items removed

getNode:

While position of node is not reached:

Traverse list

Once position is reached

Return pointer to node at that position

Subsequence:

Check if seq2 could satisfy definition of subsequence:

Sequences not empty, seq2 must be same size or smaller than seq1

If sequences are aliases:

Since a sequence is a subsequence of itself, seq2 is subsequence

Otherwise

Store value (\*\*\*call it *head*\*\*\* ) of first object of seq2

If head not in seq1

Seq2 not a subsequence

Otherwise compare seq1 consecutive values with seq2 consecutive values, starting at head’s position in seq1 and head’s position in seq 2

If consecutive values not equal, look for another occurrence of head in seq1

If no occurrence, seq2 not a subsequence

Otherwise, compare consecutive values from each sequence again

If reach end of seq1 and not end of seq2, seq2:

not subsequence

If there are consecutive values after head in each sequence and reach end of seq2:

seq2 is subsequence

Return position of head in seq1

concatReverse:

If no aliasing is occurring with result and either seq or seq2:

Algorithm:

Erase all objects in result

Insert seq1 objects from back to front into result

Insert seq2 objects from back to front into result

If aliasing is occurring:

Result and seq1 are aliases, seq2 is distinct:

Make copy of seq1

Erase all objects in result

Insert seq1 copy’s objects from back to front into result

Insert seq2 objects from back to front into result

Result and seq2 are aliases, seq1 is distinct:

Make copy of seq2

Erase all objects in result

Insert seq1 objects from back to front into result

Insert seq2 copy’s objects from back to front into result

Result, seq1, and seq2 are all aliases:

Make copy of seq1 and copy of seq2

Erase all objects in result

Insert seq1 copy’s objects from back to front into result

Insert seq2 copy’s objects from back to front into result

**Test Cases**

The following tests were performed on a sequence of unsigned longs:

**Given by Professor Smallberg:**

// default constructor

Sequence s;

// For an empty sequence:

assert(s.size() == 0); // test size

assert(s.empty()); // test empty

assert(s.find(10) == -1); // test find

assert(s.remove(0) == 0); // nothing to remove

**Start of my own test cases:**

// Default Constructor, insert single parameter, insert double parameter

Sequence s1;

assert(s1.insert(0, 1) == 0); // test insert on empty list

assert(s1.insert(1, 3) == 1); // end of list, one element list

assert(s1.insert(1, 2) == 1); // middle of list

assert(s1.insert(4) == 3); // test insert with 1 parameter

assert(s1.insert(10,5) == -1); // out of bounds insert

assert(s1.insert(-1,5) == -1); // out of bounds insert

// Copy Constructor, insert, find, size

Sequence s3 = s1;

for (int i = 0, k = 1; i < 4; i++, k++) { // test copy constructor

assert(s3.find(k) == i);

}

for (int i = 0; i < 4; i++) { // test erase

assert(s3.erase(0) == true);

}

assert(s1.size() == 4); // s1 is not affected

assert(s3.empty() == true); // s3 is empty now

// Assignment Operator/Copy Constructor

Sequence s2 = s1; // Copy Constructor

s3 = s2; // Assignment operator

for (int i = 0, k = 1; i < 4; i++, k++) { // test assignment op

assert(s3.find(k) == i);

}

for (int i = 0; i < 4; i++) { // test erase

assert(s3.erase(0) == true);

}

assert(s1.size() == 4); // s1 is not affected

assert(s2.size() == 4); // s2 is not affected

assert(s3.empty() == true); // s3 is empty now

// Remove

Sequence s4;

for (int i = 0; i < 5; i++)

s4.insert(7);

assert(s4.remove(7) == 5); // Test remove

assert(s4.remove(10) == 0); // Test element not in list

assert(s4.empty() == true); // All 5 elements removed

// Get and Set

Sequence s5;

s5.insert(0, 10);

s5.insert(0, 20);

assert(s5.size() == 2);

ItemType x = 999;

assert(s5.get(0, x) && x == 20); // x is 20

assert(s5.get(1, x) && x == 10); // x is 10

assert(s5.get(2, x) == false); // out of bounds get

assert(s5.get(-1, x) == false); // out of bounds get

assert(s5.set(0, 30) && s5.find(30) == 0); // Element 1 has val 30

assert(s5.set(1, 40) && s5.find(40) == 1); // Element 2 has val 40

assert(s5.set(2, 50) == false); // out of bounds set

assert(s5.set(-1, 60) == false); // out of bounds set

// Swap, insert, erase, remove

Sequence s6;

Sequence s7;

assert(s6.empty() == true);

assert(s6.insert(1, 5) == -1); // out of bounds insert

assert(s6.insert(0, 1) == 0);

assert(s6.insert(1, 2) == 1);

assert(s6.insert(2, 3) == 2);

assert(s6.insert(3) == 2);

assert(s6.insert(0) == 0);

assert(s6.insert(4) == 5);

assert(s6.erase(0) == true); // erase first element

assert(s6.erase(4) == true); // erase last element

assert(s6.erase(2) == true);

assert(s6.remove(3) == 1);

assert(s6.remove(1) == 1);

assert(s6.remove(2) == 1); // s6 is empty now

for (int i = 0; i < 5; i++) // s6 is {0,1,2,3,4}

assert(s6.insert(i) == i);

for (int i = 0; i < 6; i++)

assert(s7.insert(i, i+5) == i); // s7 is {5,6,7,8,9,10}

assert(s6.remove(5) == 0); // no element exists

assert(s7.remove(4) == 0); // no element exists

// Swap lists

s6.swap(s7);

assert(s6.size() == 6 && s6.find(5) == 0 && s7.size() == 5 &&

s7.find(0) == 0 && s7.find(1) == 1);

// Swap lists back

s6.swap(s7);

assert(s6.size() == 5 && s6.find(0) == 0 && s7.size() == 6 &&

s7.find(5) == 0 && s7.find(6) == 1);

// s6 is {0,1,2,3,4} and s7 is {5,6,7,8,9,10} after double swap, both back to original sequences

// Subsequence

Sequence s8;

s8.insert(0,30);

s8.insert(1,21);

s8.insert(2,63);

s8.insert(3,42);

s8.insert(4,17);

s8.insert(5, 63);

s8.insert(6, 17);

s8.insert(7, 29);

s8.insert(8, 8);

s8.insert(9, 32);

Sequence s9;

s9.insert(0, 63);

s9.insert(1, 17);

s9.insert(2, 29);

assert(subsequence(s8, s9) == 5); // Example from spec, tests for

// multiple occurrences of s9’s

// first element. We need to keep iterating through s8 to produce the right behavior

Sequence s10, s11, s12;

assert(subsequence(s10, s11) == -1); // two empty sequences

s10.insert(0, 63);

s10.insert(1, 17);

s10.insert(2, 29);

assert(subsequence(s10, s11) == -1); // tests empty sequence, which is !subsequence of another sequence

assert(subsequence(s12, s10) == -1); // tests two empty sequences

assert(subsequence(s10, s10) == 0); // tests same sequence

s11.insert(0, 63);

s11.insert(1, 17);

s11.insert(2, 29);

s11.insert(3, 30);

assert(subsequence(s10, s11) == -1); // tests when the “subsequence” still has more characters but the other sequence has no more left, even if they are identical up until the end of the first sequence

// concatReverse

Sequence s13;

s13.insert(0, 3);

s13.insert(1, 5);

s13.insert(2, 7);

s13.insert(1, 4);

s13.insert(10);

s13.erase(4); // s13 is going to get changed with concatReverse

Sequence a;

a.insert(1);

a.insert(2);

a.insert(3);

Sequence b;

b.insert(4);

b.insert(5);

b.insert(6);

concatReverse(a, b, s13); // s13 is {3,2,1,6,5,4}

assert(s13.find(3) == 0 && s13.find(2) == 1 && s13.find(1) == 2 && s13.find(6) == 3 && s13.find(5) == 4 && s13.find(4) == 5);

// concatReverse and Assignment operator

Sequence c = a;

Sequence d = a;

concatReverse(c, d, a);

assert(a.find(3) == 0 && a.find(2) == 1 && a.find(1) == 2);

assert(a.size() == 6);

assert(c.find(1) == 0 && c.find(2) == 1 && c.find(3) == 2);

assert(c.size() == 3);

assert(d.find(1) == 0 && d.find(2) == 1 && d.find(3) == 2);

assert(d.size() == 3);

// concatReverse and Aliasing

Sequence e;

e.insert(1);

e.insert(2);

e.insert(3);

Sequence f;

f.insert(4);

f.insert(5);

f.insert(6);

Sequence g = f;

// seq1 and result are aliases

concatReverse(e, f, e);

assert(e.find(3) == 0 && e.find(2) == 1 && e.find(1) == 2 && e.find(6) == 3 && e.find(5) == 4 && e.find(4) == 5);

assert(e.size() == 6);

// seq 2 and result are aliases

concatReverse(e, f, f);

assert(f.find(4) == 0 && f.find(5) == 1 && f.find(6) == 2 && f.find(1) == 3 && f.find(2) == 4 && f.find(3) == 5);

assert(f.size() == 9);

// seq1, seq2, and result are all aliases

concatReverse(g, g, g);

assert(g.find(6) == 0 && g.find(5) == 1 && g.find(4) == 2);

assert(g.size() == 6);

// Note for concatReverse, using the dump function helps illustrate every single element in the lists, but it’s hard to use the find function to illustrate everything is in the right spot (as it only finds the first occurrence), so I would recommend the dump function I wrote, which prints every element in the list in order and prints the size of the list!

// However, these are the type of test cases I used to test the program in the face of aliasing to make sure I got correct behavior.