Greyson Beck

805010132

Project 2 Report

**Design:**

My project was implemented without using a circularly linked list, thus no need for a dummy node. While I suppose this made some aspects of the design challenging, particularly wherever it involved looping through the list to insert, delete, etc., it saved me from a certain level of complexity. As hinted upon, I did run into some special cases, so my code may not be as efficient as I would have liked it to be, what with all the null pointers scattered throughout! The overall approach, for most of the function implementations, was to deal with the special case(s) first and then proceed to deal with the more general case(s).



|  |
| --- |
| Head |



|  |
| --- |
| m\_value |
| m\_next |
| m\_previous |



|  |
| --- |
| m\_value |
| m\_next |
| m\_previous |

|  |
| --- |
| m\_value |
| m\_next |
| m\_previous |



\*\*\*Node 1’s m\_next points to Node 2, etc.

\*\*\* Node 3’s m\_previous points to Node 2, etc.

**Pseudocode:**

Sequence::insert(int pos, const ItemType& value)

If (pos < 0 or pos > length)

Return false

Else if (empty())

If (pos = 0)

Create new Node

Set head equal to new Node

New Node value = value passed in

Previous and next pointers of newNode = NULL

Length plus 1

Return true

Else

Return false

Else if (pos in between 0 and length, not inclusive)

Create Node pointer (ie. current), set equal to head

While ( i is not pos)

Current goes to next

i plus one

now at pos:

create dynamically allocated new node

new node’s value = value passed in

set new node’s next to current

new Node’s previous = current’s previous

current’s previous’s next = new Node

current’s previous is new Node

length plus 1

return true

else

pos must equal size():

same process as previous, loop with Node pointer until the next is NULL

then create new Node dynamically allocated

new Node’s value set

current’s next = new Node

new Node’s previous = current

new Node’s next = NULL

length plus 1

return true

//End of function

Sequnce::insert(const ItemType& value)

If size = 0

New Node dynamically allocated

New Node’s value is set

Head equals new Node now

New Node’s previous equals NULL

Length plus 1

Return 0

Else if (value < head’s value)

Belongs in front of non-empty list:

Dynamically allocated new Node

New Node’s value = value

New Node’s next = head

Head’s previous = new Node

New Node’s previous = NULL

Head NOW = newNode

Length plus 1

Return 0

Else

Value either in middle or end:

Node pointer set = head

If length = 1

New Node dynamically allocated

Placed at end, next’s value = NULL

Previous = head

Head’s next = newNode

Length plus 1

Return 1

int I = 0

while (current’s next != NULL)

if (value equals current’s value)

call insert(I, value)

return I

else

current goes to next

I plus 1

If out of loop:

If ( value equals current’s value)

New Node inserted where last item was

If (value is greater than current’s value)

New Node inserted at the end, making sure it’s next pointer = NULL

and it’s previous equals current

// end of function

Sequence::interleave(seq 1, seq2, result)

Copy result into temporary result sequence

While (result isn’t empty)

Erase position 0

If (seq1 is empty)

Result = seq2

Else if (seq2 is empty)

Result = seq1

Else

int i(0), j(0), count(0)

ItemType value1, value2

While (I < seq1’s size && j < seq2’s size)

Get item at pos i in seq1, copy into value1

Insert value1 into result at pos count

Get item at pos j in seq2, copy into value2

Inset value2 into result at pos (count+1)

Loop breaks when one/both seq’s have been iterated through:

If (I equals seq1’s size && j equals seq2’s size)

do nothing

else

if ( I equals seq1’s size)

while (j < seq2’s size)

get item at pos j in seq2, copy into value2;

insert value2 into result at pos count

j plus 1

count plus 1

else if (j equals seq2’s size)

while (I < seq1’s size)

get item at pos I in seq1, copy into value1;

insert value1 into result at pos count

I plus 1

Count plus 1

Set the temp result used in the function equal to actual result Sequence (passed in)

// end of function

**Test Cases:**

These are tested using ItemType = unsigned long (though I know it’s turned in with std::string):

Sequence a;

Sequence b;

assert(a.size() == 0);

assert(b.size() == 0);

assert(a.empty());

assert(b.empty());

a.insert(0, 20);

a.insert(1, 30);

a.insert(2, 15);

assert(a.insert(4, 30) == false);

a.insert(3, 30);

a.insert(4, 45);

a.insert(4, 40);

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

assert(a.empty() == false);

a.insert(a.size(), 44);

a.insert(a.size(), 55);

a.insert(a.size(), 30);

for (int i = 0; i < 50; i++) {// Make sure values are put in ascending order…insert works correctly

b.insert(i);

}

Sequence c;

assert(subsequence(a, b, result) == -1); // Makes sure subsequence returns -1 when seq2 > seq1

assert(subsequence(b, c) == -1); // when seq2 is empty, return seq

Sequence a;

Sequence b;

assert(a.size() == 0); //Check if default size is 0

assert(b.size() == 0); // Check if default size is 0

assert(a.empty()); // Make sure empty() and size() return same values

assert(b.empty());

// Insert items with given pos

a.insert(0, 20);

a.insert(1, 30);

a.insert(2, 15);

assert(a.size() == 3); // Check if size() works after inserting items

assert(a.erase(2)); // Erasing last element in list

assert(a.size() == 2); // size() should change accordingly

assert(a.insert(4, 30) == false); // Can’t insert past length of list

a.insert(2, 30); // inserting at end

a.insert(3, 45); //inserting at end again

a.insert(3, 40); // making sure insert allows value to replace

assert(a.find(40) == 3); // checking above statement

assert(a.size() == 5); // Keep checking to see if size() responds to inserts

assert(a.empty() == false); // empty should be false, list has items

a.insert(a.size(), 44); // test larger values, inserting at end:

a.insert(a.size(), 55);

a.insert(a.size(), 30);

for (int i = 0; i < 50; i++) {// Make sure values are put in ascending order…insert works correctly

b.insert(i);

}

assert(b.size() == 50); // size() responds accordingly

assert(b.insert(50) == 50); // can insert at end

Sequence c;

assert(subsequence(a, b) == -1); // Makes sure subsequence returns -1 when seq2 > seq1

assert(subsequence(b, c) == -1); // when seq2 is empty, return -1

c.insert(0, 44); // insert values into c

c.insert(1, 55);

c.insert(2, 30);

assert(b.insert(44) == 0); // should be in front of list

assert(b.insert(55) == 2); // goes after both 44 elements

assert(b.insert(30) == 0); // should be in front of list, since smaller than 44

assert(subsequence(a, c) == 5); // seq2 is a subsequence of seq1, starting middle of list (position 5)

assert(subsequence(b, c) == -1); // seq2 is almost subsequence of seq1, still returns -1

Sequence d;

assert(d.empty()); // insert various items, in random order to make sure insert will move items around accordingly:

assert(d.insert(1) == 0);

assert(d.insert(4) == 1);

assert(d.insert(3) == 1);

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

assert(d.insert(44) == 3);

assert(d.insert(55) == 4);

assert(d.insert(5, 30)); // use forced insert to make sure it works in the middle of list

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

assert(d.insert(4) == 2); // make sure 4 will take spot of previous 4, other 4 moves up

assert(d.size() == 7); // check size() again (why not)

assert(d.find(100) == -1); // value is not in list, should return -1

assert(subsequence(d, c) == 4); // subsequence still works after many values have been moved around…start position has changed

// Checking remove will erase value x number of times:

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

assert(d.remove(0) == 0);

assert(d.remove(30) == 1);

assert(d.remove(44) == 1);

assert(d.remove(4) == 2); // still removes all occurrences when there’s more than one of them

assert(d.size() == 2); // make sure remove actually decremented size() of list

Sequence e;

Sequence f;

// create new sequence with ascending values:

assert(e.insert(0, 1));

assert(e.insert(1, 5));

assert(e.insert(2, 10));

assert(e.insert(3, 15));

assert(e.insert(4, 20));

// create other new sequence with ascending values

assert(f.insert(1) == 0);

assert(f.insert(2) == 1);

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

assert(f.insert(10) == 3);

Sequence result;

interleave(e, f, result); // Makes sure values input correctly when m > n

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

assert(result.find(2) == 3); // Makes sure value 2 is at position 3

assert(result.find(20) == 8);

assert(f.insert(12) == 4); // Making m = n

assert(f.find(10) == 3);

interleave(e, f, result); // Makes sure a non-empty result passed in will be erased and new values added

assert(result.find(12) == 9); // Last item in result should be from seq2, should be 12

assert(result.find(20) == 8); // Earlier values in result should be at same position as before

assert(result.find(2) == 3);

assert(f.insert(2, 5) == true); // Making n > m, with value inserted into middle, not end

assert(f.find(10) == 4);

interleave(e, f, result); // Again makes sure result will be erased to 0 items, and refilled

assert(result.size() == 11);

assert(result.find(12) == 10); // Last item in result from seq2 again, but one index higher...with same value

assert(result.find(5) == 2); // new item inserted previously into seq2 should be in position 5 in result

assert(result.find(3) == 7); // this item should be pushed up in seq2, 2 items later in result