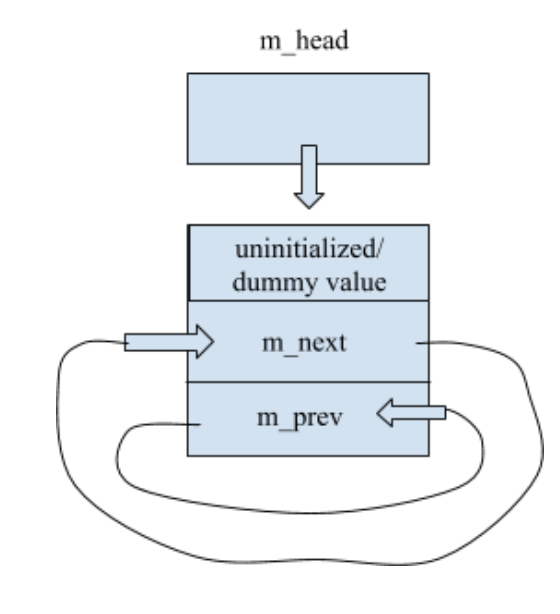
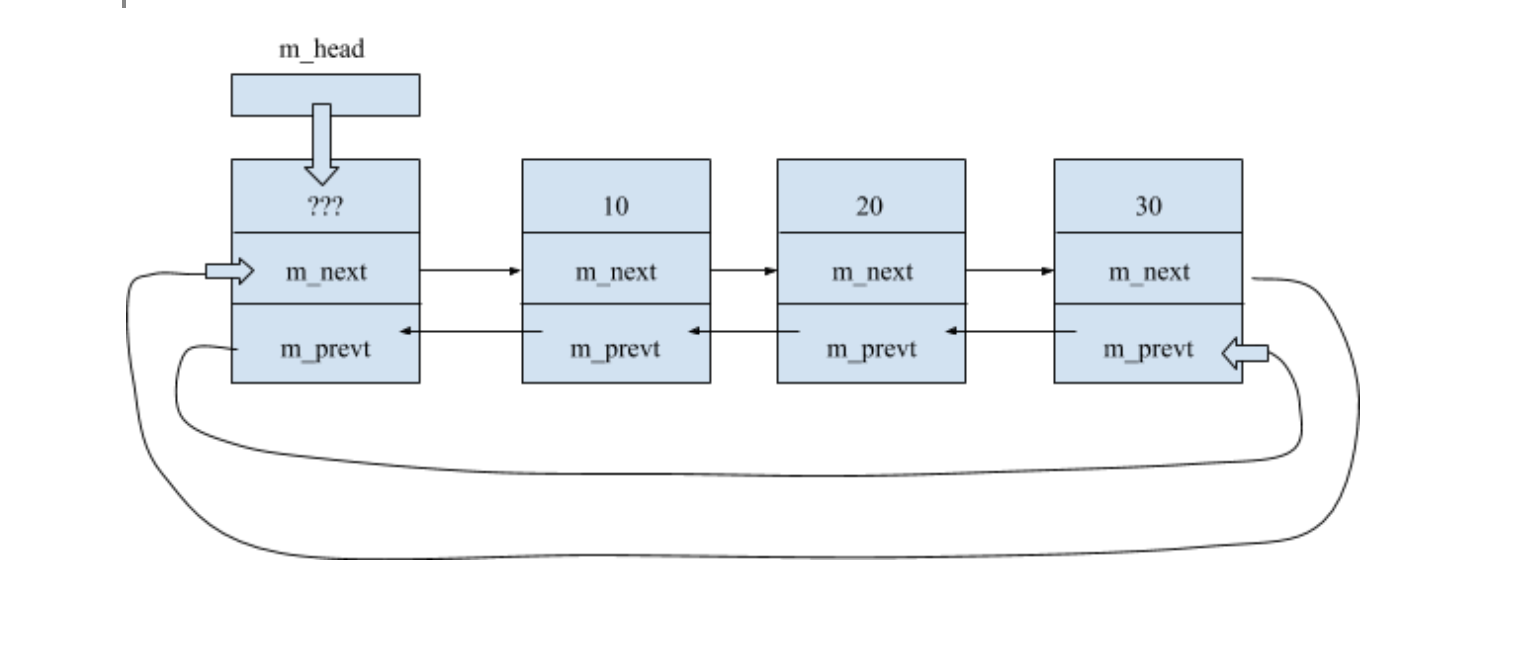
1. A description of my implementation and design:

I chose to implement my Sequence as a circular, doubly- linked list in order to eliminate several special cases involving nullptr and simplify the implementation process. Every node in my list is always pointing to another physical node, providing easy access to its predecessor and successor. An empty linked list is represented by a “dummy” (head) node, one with no initialized value that points both forward and backwards to itself to maintain the circular nature of the list. 

The diagram to the left is a representation of an empty list, as implemented by my code.

An example of a non-empty linked list is shown below (where 10, 20, 30 are values stored in variable m\_value):



Each node contains a pointer (m\_next) to its successor node in the list and a pointer (m\_prev) to its predecessor node, as well as a variable (m\_value) to hold its value.

2) Pseudocode

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

If position exceeds size or is negative

Return false

If position is equal to size

Insert at the end of the list..

Create pointer to last element in list

Create new node with proper value

Adjust new node’s prev and next pointers

Adjust surrounding nodes’ pointers to appropriately point to new node

Increment size

Return true

Else position in middle of list

Traverse through list until we reach the node at specified position

Create new node with proper value

Adjust new node’s prev and next pointers

Adjust surrounding nodes’ pointers to appropriately point to new node

Increment size

Return true

**int Sequence::insert(const ItemType& value)**

Set default position for insertion to be end of list

If list is empty

Insert at beginning of list..

Create new node with proper value

Adjust new node’s prev and next pointers

Adjust surrounding nodes’ pointers to appropriately point to new node

Increment size

Return position at which node was inserted

If list not empty

Traverse through list and compare node values to the value of new node

If current node’s value is greater than/ equal to new node’s value

Mark current index as position for insertion

Break

Insert value at proper position

Return position

**bool Sequence::erase(int pos)**

If position is non negative and does not exceed size

Traverse through nodes in list while keeping track of element index

If index equals the position at which a node should be erased

Remove item at position..

Create a temporary pointer to node being removed

Adjust the pointers of surrounding nodes to point to temp node’s next and prev

Delete temporary pointer

Decrement size and return true

Otherwise position is invalid

Return false

**int Sequence::remove(const ItemType& value)**

If list is empty

Return 0

If list is not empty

While not yet at end of list

If current node’s value equals value to be deleted

Advance to next node

Erase node with matching value

Increment number of removed items

Decrement index by one to account for lost element

Else if value not equal to item to be deleted

Advance to next node

Return number of removed items

**void Sequence::swap(Sequence& other)**

Create temporary pointer to head of linked list

Make m\_head point to other’s head

Make other’s head point to temporary head

Create temporary variable to store size of linked list

Set m\_size to other’s size

Set other’s size to temporary size

**Sequence::~Sequence()**

While not empty list

Create a pointer to first element of list

Set current node’s previous to point to node after current node

Set current node’s next to point to node preceding current node

Delete current node

**Sequence::Sequence(const Sequence& other)**

Initialize new Sequence by creating new head node

Set head to point forward and backwards to itself (empty list)

Start with a node pointer to head

Traverse over each node in existing Sequence

For each, create a new node with proper value

Adjust pointers of new node

Set new node’s next pointer to be head as in a circular linked list

Set head’s previous to be new node, so new node is last in sequence

Advance node pointer to the new node in sequence

**Sequence& Sequence::operator=(const Sequence& other)**

If other Sequence equals Sequence object

Do nothing, Return Sequence object

While not empty list

Delete all nodes in sequence

Traverse over each node in Sequence to be copied

For each, create a new node with proper value

Adjust pointers of new node

Set new node’s next pointer to be head as in a circular linked list

Set head’s previous to be new node

Advance node pointer to the new node in sequence

Set m\_size of Sequence object to that of other Sequence

Return Sequence object

**int subsequence(const Sequence& seq1, const Sequence& seq2)**

If sequence two is empty or is larger than seq 1

No valid subsequence, return -1

Iterate over the sequences

Get the current node’s value in seq1

Get the corresponding value in seq2

If the values in seq1 and seq2 are equal

Set current index to be possible starting position of subsequence

If possible starting position is last element and size of subsequence > 1

No valid subsequence, return -1

Start with possible starting position and iterate over sequences

If at the end of sequence two

All values match, return starting position of valid subsequence

Get value at current index in sequence one

Get value at current index in sequence two

If seq1 value does not equal seq2 value

Reset index of sequence two to 0 to restart iteration

Set k to be negative one

Break

Return k, starting position of subsequence or -1 if no subsequence found

**void interleave(const Sequence& seq1, const Sequence& seq2, Sequence& result)**

Create temporary Sequence object from result

Clear contents of temporary Sequence object to avoid aliasing

Set as maximum size to length of longer sequence

Iterate through sequences until reaching maximum size

Attempt to get value at node of current index for seq1

If successful (index was valid)

Insert seq1 value into proper index of temporary result sequence

Increment position of insertion in resulting index

Attempt to get value at node of current index for seq2

If successful (index was valid)

Insert seq2 value into proper index of temporary result sequence

Increment position of insertion in resulting index

Set result Sequence equal to temporary Sequence

3) Test Cases: tests were performed on a sequence of **unsigned long** (i.e, ItemType was a type alias for unsigned long)

#include <iostream>

#include <cassert>

#include "Sequence.h"

using namespace std;

int main() {

// default constructor

Sequence s;

// tests for empty list

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

assert(s.empty() == 1); // test empty

assert(s.remove(10) == 0); // nothing to remove

/\* test cases for insert() functions \*/

s.insert(0,10); // insert value 10 at position 0

// CASE: pos equals size, empty list

s.dump(); // 10, size of 1

s.insert(1,30); // insert value 20 at position 1

// CASE: pos equals size

s.dump(); // 10 30, size of 2

s.insert(1,20); // insert value 20 at position 1

// CASE: insertion in middle of list (pos != size())

s.dump(); // 10 20 30, size of 3

s.insert(15); // insert value 15

// CASE: list not empty, insert in ascending order

s.dump(); // 10 15 20 30, size of 4

/\* test case for erase() functions \*/

assert(s.erase(1) == 1); // erase value 15

s.dump(); // 10 20 30, size of 3

/\* test cases for get() function \*/

ItemType x;

// get values at specified positions

assert(s.get(0, x) && x == 10);

assert(s.get(1, x) && x == 20);

assert(s.get(2, x) && x == 30);

/\* test cases for remove() function \*/

s.insert(15); s.insert(25); s.insert (25);

s.dump(); // 10 15 20 25 25 30, size of 6

assert(s.remove(25) == 2); // remove all instances of value 25 (consecutive)

s.dump(); // 10 15 20 30, size of 4

assert(s.size() == 4);

assert(s.insert(3,15) == 1);

assert(s.insert(0,15) == 1);

s.dump(); // 15 10 15 20 15 30, size of 6

assert(s.remove(15) == 3); // case: instances of value 15 are not consecutive in list

s.dump(); // 10 20 30, size of 3

assert(s.insert(40) == 3);

assert(s.insert(50) == 4);

assert(s.insert(5,60) == 1);

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

s.dump(); // 10 20 30 40 50 60

/\* test cases for set() function \*/

Sequence s1;

assert(s1.set(12,12) == 0); // invalid position

assert(s.set(2,20) == 1);

s.dump(); // 10 20 20 40 50 60

assert(s.set(0,80) == 1);

s.dump(); // 80 20 20 40 50 60

assert(s.set(6,70) == 0); // invalid position

/\* test cases for find() function \*/

s.dump(); // 80 20 20 40 50 60

assert(s.find(80) == 0);

assert(s.find(20) == 1);

assert(s.find(100) == -1);

Sequence s2;

assert(s2.empty());

s2.dump();

assert(s2.find(0) == -1);

/\* test cases for swap() function \*/

Sequence mySeq;

for(int i = 1; i <= 10; i++)

mySeq.insert(i);

mySeq.dump(); // 1 2 3 4 5 6 7 8 9 10

assert(mySeq.size() == 10);

Sequence otherSeq;

for(int i = 1; i <= 5; i++)

otherSeq.insert(i);

otherSeq.dump(); // 1 2 3 4 5

assert(otherSeq.size() == 5);

cerr << "before swap: " << endl;

cerr << "mySeq: ";

mySeq.dump();

cerr << "otherSeq: ";

otherSeq.dump();

cerr << endl;

mySeq.swap(otherSeq);

cerr << "after swap: " << endl;

cerr << "mySeq: ";

mySeq.dump();

cerr << "otherSeq: ";

otherSeq.dump();

/\* test case for destructor \*/

Sequence banana;

for (int i = 1; i <= 5; i++)

banana.insert(i);

banana.dump();

banana.~Sequence();

/\* test case for copy constructor \*/

Sequence obj1;

for (int i = 1; i <= 5; i++)

obj1.insert(i);

obj1.dump();

Sequence obj2(obj1);

obj2.dump(); // obj2 now same as obj1 in terms of elements and size

/\* test case for assignment operator \*/

Sequence lhs;

for (int i=1; i<=5; i++)

lhs.insert(i);

lhs.dump();

Sequence rhs;

for (int i=1; i<=10; i++)

rhs.insert(i);

rhs.dump();

lhs = rhs;

lhs.dump(); // lhs and rhs now equal in terms of elements and size

/\* test cases for subsequence() function \*/

// TEST CASE ONE: basic case with one matching subsequence

Sequence seqOne;

seqOne.insert(0,30); seqOne.insert(1,21); seqOne.insert(2,63); seqOne.insert(3,42);

seqOne.insert(4,17); seqOne.insert(5,63); seqOne.insert(6,17); seqOne.insert(7,29); seqOne.insert(8,8); seqOne.insert(9,32);

Sequence seqTwo;

seqTwo.insert(0,63); seqTwo.insert(1,17); seqTwo.insert(2,29);

seqOne.dump(); // 30 21 63 42 17 63 17 29 8 32

seqTwo.dump(); // 63 17 29

assert (subsequence(seqOne,seqTwo) == 5); // valid subsequence starts at position 5

// TEST CASE TWO: possible starting element is last element of seq one

Sequence one;

one.insert(0,10); one.insert(1,20); one.insert(2,30); one.insert(3,40);

Sequence two;

two.insert(0,40); two.insert(1,50); two.insert(2,60);

one.dump(); // 10 20 30 40

two.dump(); // 40 50 60

assert (subsequence(one,two) == -1); // no valid subsequence

// TEST CASE THREE: end of seq 1 reached before full subsequence is completed

Sequence uno;

uno.insert(0,10); uno.insert(1,20); uno.insert(2,30); uno.insert(3,40); uno.insert(4,50);

Sequence dos;

dos.insert(0,40); dos.insert(1,50); dos.insert(2,60);

uno.dump(); // 10 20 30 40 50

dos.dump(); // 40 50 60

assert (subsequence(uno,dos) == -1); // no valid subsequence

// TEST CASE FOUR: subsequence exists at very end of list

Sequence uno1;

uno1.insert(0,10); uno1.insert(1,20); uno1.insert(2,30); uno1.insert(3,40); uno1.insert(4,50); uno1.insert(5,60);

Sequence dos2;

dos2.insert(0,40); dos2.insert(1,50); dos2.insert(2,60);

uno1.dump(); // 10 20 30 40 50 60

dos2.dump(); // 40 50 60

assert (subsequence(uno1,dos2) == 3); // valid subsequence starts at position 3

// TEST CASE FOUR: no matching starting value at all

Sequence noMatch1;

noMatch1.insert(2); noMatch1.insert(4); noMatch1.insert(6); noMatch1.insert(8);

Sequence noMatch2;

noMatch2.insert(80); noMatch2.insert(81); noMatch2.insert(86); noMatch2.insert(80);

noMatch1.dump(); // 2 4 6 8

noMatch2.dump(); // 80 81 86 80

assert (subsequence(noMatch1,noMatch2) == -1); // no valid subsequence

// TEST CASE FIVE: multiple subsequences, should return starting value of first one

Sequence multiple;

multiple.insert(0,1); multiple.insert(1,2); multiple.insert(2,3); multiple.insert(3,4); multiple.insert(4,7); multiple.insert(5,11); multiple.insert(6,2); multiple.insert(7,3); multiple.insert(8,4); multiple.insert(9,8);

Sequence subSequence;

subSequence.insert(2); subSequence.insert(3); subSequence.insert(4);

multiple.dump(); // 1 2 3 4 7 11 2 3 4 8

subSequence.dump(); // 2 3 4

assert(subsequence(multiple,subSequence) == 1); // returns starting value of first valid subsequence, at position 1

// TEST CASE SIX: sequence one is empty list

Sequence empty;

Sequence notEmpty;

notEmpty.insert(0, 10);

assert(subsequence(empty,notEmpty) == -1); // empty list, no valid subsequence

// TEST CASE SIX 1/2: sequence two is empty list

Sequence imNotEmpty;

Sequence imEmpty;

imNotEmpty.insert(0, 10);

assert(subsequence(imNotEmpty,imEmpty) == -1); // empty list, no valid subsequence

// TEST CASE SEVEN: one element list

Sequence oneEl;

Sequence El;

oneEl.insert(0,10);

El.insert(0,10);

assert(subsequence(oneEl, El) == 0); // one element lists match,

// subsequence starts at position 0

// TEST CASE SEVEN: subsequence is one element only

Sequence one\_El;

Sequence two\_El;

one\_El.insert(0,10); // 10

two\_El.insert(0,10); two\_El.insert(1,20); // 10 20

assert(subsequence(two\_El, one\_El) == 0); // subsequence is one element only, starts at index 0

// TEST CASE EIGHT: seq2 larger than seq1

Sequence small;

Sequence big;

small.insert(1); // 1

big.insert(10); big.insert(15); big.insert(17); // 10 15 17

assert(subsequence(small,big) == -1); // no valid subsequence

/\* test cases for interleave() function \*/

// case 1: sequences of same size

Sequence a;

for (int i = 0; i <= 10; i++)

a.insert(i);

Sequence b;

for (int i = 10; i <= 20; i++)

b.insert(i);

a.dump(); // 0 1 2 3 4 5 6 7 8 9 10, size of 11

b.dump(); // 10 11 12 13 14 15 16 17 18 19 20, size of 11

Sequence result;

interleave(a,b,result);

result.dump(); // 0 10 1 11 2 12 3 13 4 14 5 15 6 16 7 17 8 18 9 19 10 20

// size of list: 22

// case 2: seq 1 larger than seq 2

Sequence c;

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

c.insert(i);

Sequence d;

for (int i = 10; i <= 12; i++)

d.insert(i);

c.dump(); // 0 1 2 3 4 5 , size of 6

d.dump(); // 10 11 12 , size of 3

Sequence result2;

interleave(c,d,result2);

result2.dump(); // 0 10 1 11 2 12 3 4 5

// size of list: 9

// case 3: seq 2 larger than seq 1

Sequence e;

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

e.insert(i);

Sequence f;

for (int i = 10; i <= 20; i++)

f.insert(i);

e.dump(); // 0 1 2 3 4 5 , size of 6

f.dump(); // 10 11 12 13 14 15 16 17 18 19 20, size of 11

Sequence result3;

interleave(e,f,result3);

result3.dump(); // 0 10 1 11 2 12 3 13 4 14 5 15 16 17 18 19 20

// size of list: 17

// case 4: seq 1 is empty

Sequence g;

Sequence h;

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

h.insert(i);

g.dump(); // empty , size of 0

h.dump(); // 0 1 2 3 4 5 , size of 6

Sequence result4;

interleave(g,h,result4);

result4.dump(); // 0 1 2 3 4 5 , size of 6

// case 5: seq 2 is empty

Sequence i;

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

i.insert(k);

Sequence j;

i.dump(); // 0 1 2 3 4 5 , size of 6

j.dump(); // empty , size of 0

Sequence result5;

interleave(i,j,result5);

result5.dump(); // 0 1 2 3 4 5 , size of 6

// case 6: aliasing; seq1 and result refer to the same sequence

Sequence yo;

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

yo.insert(k);

yo.dump();

i.dump();

interleave(yo,i,yo);

yo.dump(); // 0 0 1 1 2 2 3 3 4 4 5 5

// size of list: 12

return 0;

}