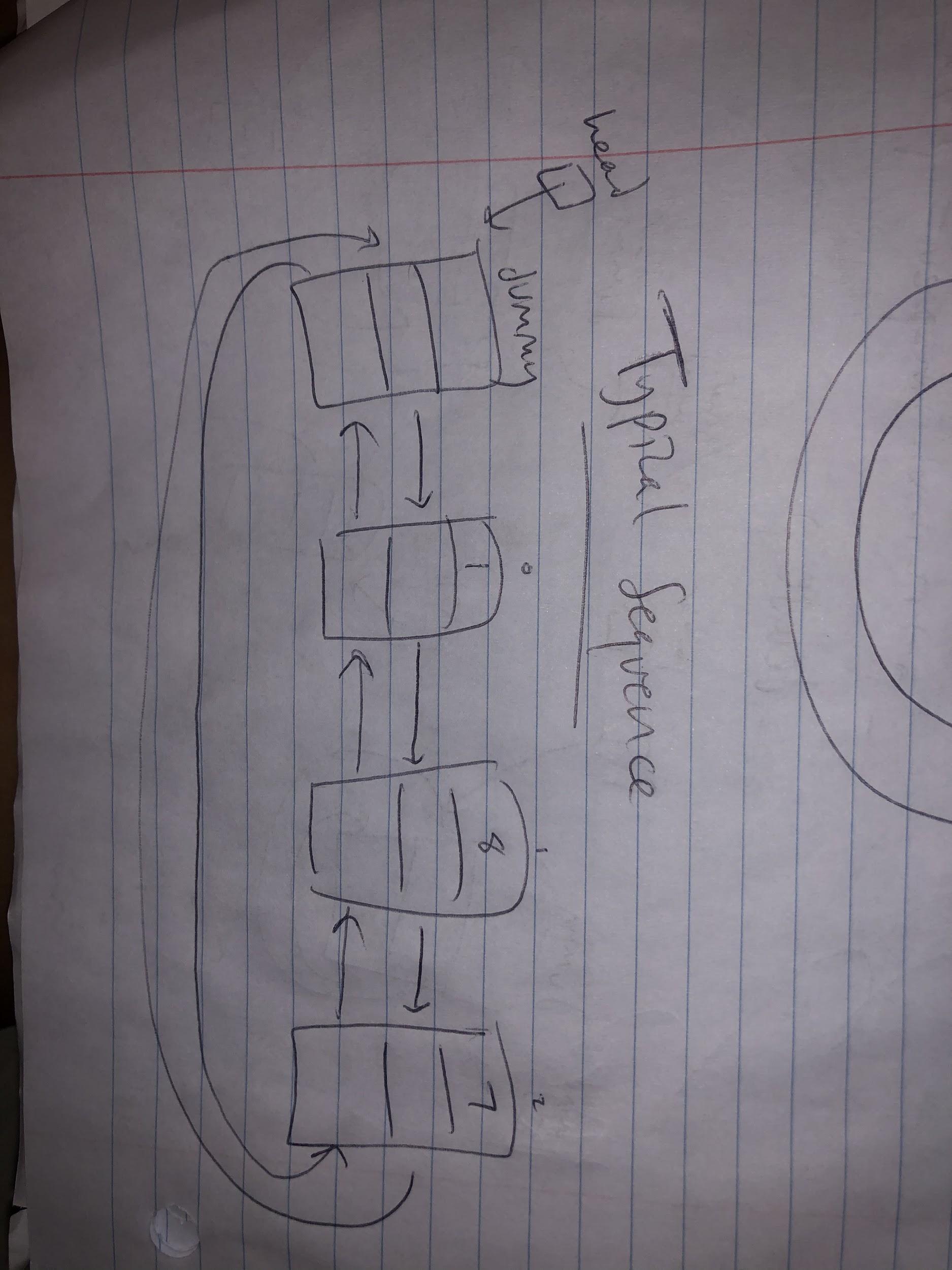
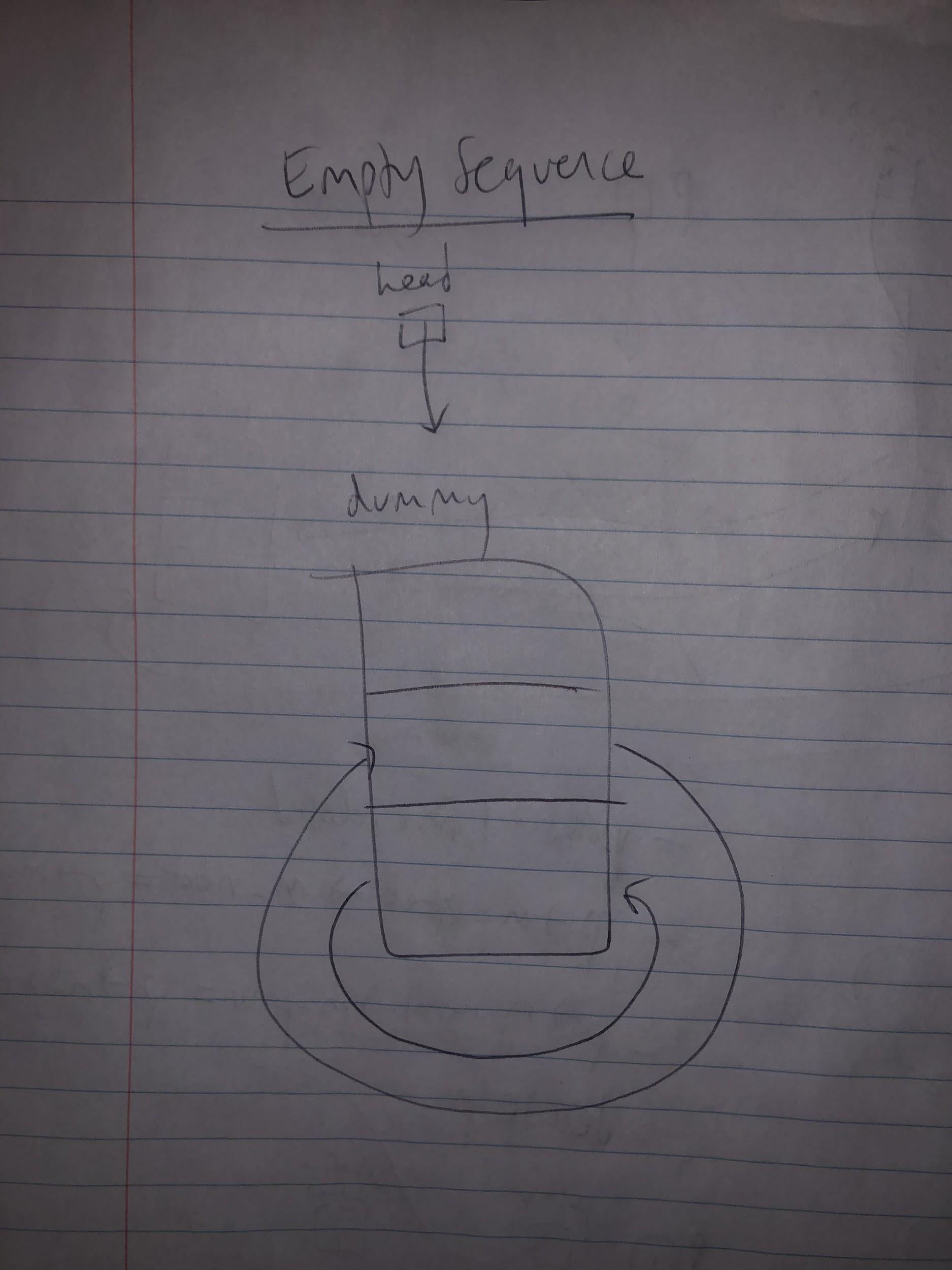
1. My doubly-linked circular list implementation included a dummy node and a head pointer pointing to the dummy node. Each of my list nodes included a value of ItemType type, and two pointers: one m\_next pointing to the next node in the linked list, and one m\_prev pointing to the previous node in the linked list. Because it’s a circular list, the last node’s m\_next points to the dummy node, and the dummy node’s m\_prev points to the last node in the list. An empty Sequence just includes a head pointer to a dummy node with an uninitialized value and m\_next / m\_prev pointers pointing to itself (see pictures attached). 
2. **pseudocode:**

**Destructor:**

Create pointer to first interesting node

Repeatedly:

Break out of loop when only dummy node left

Create a temp ptr pointing to subsequent node

Delete pointer’s node and adjust Sequence size

Assign p to next subsequent node

Delete dummy node

**Copy Constructor:**

Copy size of other over

Create new dummy node and have it point to itself

Create new pointer pointing to head

Create new pointer pointing to other’s head

Iterating through the other Sequence:

Create a temporary pointer pointing to new node

Assign new node other’s value

Assign new node’s pointers to next and previous nodes

Update previous and subsequent item’s pointers to the new node

Update pointer to point to newly allocated node

**int insert(int pos, const ItemType& value);**

Check that pos is within Sequence bounds

Return -1 if not

Iterate through Sequence with ptr until ptr points at node right before the specified position

Insert node and initialize value

Assign new node’s pointers to next and previous nodes

Update previous and subsequent item’s pointers to the new node

Increment Sequence size

Return pos

**int insert(const ItemType& value);**

Create new pointer pointing to node after head

Iterate through Sequence with ptr until ptr points to the node before first node that equals value

Insert node and initialize value

Assign new node’s pointers to next and previous nodes

Update previous and subsequent item’s pointers to the new node

Increment Sequence size

Return pos

**bool erase(int pos);**

Check that pos is within Sequence bounds

Return false if not

Create new pointer pointing to node after head

Iterate through Sequence with ptr until ptr points at node at the specified position

Update previous and subsequent item’s pointers to point to each other

Delete node

Decrement Sequence size

**int remove(const ItemType& value);**

Create variable to keep track of number of items removed

Iterate through Sequence with ptr

If node equals value:

Create temp ptr to previous node

Delete node and update surrounding pointers accordingly

Decrement Sequence size and update count

Reassign ptr to previous node

Return count

**int find(const ItemType& value) const;**

Iterate through Sequence with ptr until ptr points at node with specified value

If no item exists

Return -1

Else

Return pos of node with specified value

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

Swap the nodes the head pointers are pointing to with temporary pointer

Swap the integer values for size with temporary variable

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

If Seq 2 empty:

return -1

If seq1 and seq2 are aliases:

return 0

Create boolean to track if subsequence present

Iterating through seq1 elements:  
 Check for a match with first element of seq2

If match:

Update boolean

Iterate through seq2 checking for continued match

If at any point no match between required elements:

Update boolean to false

If seq2 is completely present in seq1

Return pos in seq1

If no match:

Return -1

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

Create a temporary Sequence

Find sizes of seq1 and seq2

Repeatedly:

if seq1 is bigger:

During portion where both seq1 and seq2 have elements at pos

Insert seq1 elements at even positions in temp Sequence

Insert seq2 elements at odd positions

When only seq1 has elements leftover to add

Insert seq1 remaining elements

if seq2 is bigger:

During portion where both seq1 and seq2 have elements at pos

Insert seq1 elements at even positions in temp Sequence

Insert seq2 elements at odd positions

When only seq2 has elements leftover to add

Insert seq2 remaining elements

Else if seq1 and seq2 are same size:

Insert seq1 elements at even positions in temp Sequence

Insert seq2 elements at odd positions

Swap temp Sequence into result

1. **TEST CASES:**

The tests were performed on a sequence of unsigned longs (i.e., ItemType was a type alias for unsigned long).

#include "Sequence.h"

#include <iostream>

#include <cassert>

using namespace std;

int main(){

//default constructor

Sequence a;

assert(a.insert(0, 10) == 0); //test insert

assert(a.insert(0, 20) == 0); //test insert before first obj

assert(a.size() == 2); //test size w/ objects

ItemType x = 999;

assert(a.get(0, x) && x == 20);

assert(a.get(1, x) && x == 10);

assert(a.find(10) == 1); //find at pos 1 (end)

assert(a.find(20) == 0); //find at pos 0

Sequence s; //default constructor

// For an empty sequence:

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

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

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

assert(s.insert(3) == 0); //insert value into empty seq

assert(s.insert(4) == 1); //insert value at end

assert(s.insert(4) == 1); //insert same value properly

assert(s.erase(0) && s.get(0, x) && x == 4); //erase pos 0

assert(s.erase(2) == false); //invalid pos for erase

assert(s.find(4) == 0); //find returns smallest pos

Sequence e(s); //copy constructor

s=a; //assinment operator

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

Sequence b(a); //copy constructor

assert(b.size() == 2 && b.get(0, x) && x == 20 && b.empty() == false); //test not empty

assert(b.insert(2, 3) == 2 && b.get(2, x) && x == 3); //insert at size()

assert(b.insert(4, 1) == -1); //insert w/ invalid pos

assert(b.insert(3) == 0); //insert value at beginning (pos 0)

assert(b.erase(0) && b.find(3) == 2); //erase item at pos 0

assert(b.remove(20) == 1); //remove one element

assert(b.insert(20) == 2); //setting up for next remove test

assert(b.insert(20) == 2); //setting up for next remove test

assert(b.remove(20) == 2); //remove two consecutive elements

assert(b.get(0, x) && x == 10); //get pos 0

assert(b.get(b.size(), x) == false && x == 10); //invalid pos for get

assert(b.set(0, 20) && b.get(0, x) && x == 20); //set at pos 0

assert(b.set(b.size(), 20) == false && b.get(0, x) && x == 20); //invalid pos for set

assert(b.find(22) == -1); //can't find item

Sequence d;

d.swap(e); //swap function

assert(d.size() == 2 && e.size() == 0 && d.find(4) == 0); //check if swap function worked

//setting up tests for subsequence and interleave

Sequence g;

g.insert(0, 30);

g.insert(1, 21);

g.insert(2, 63);

g.insert(3, 17);

g.insert(4, 17);

g.insert(5, 63);

g.insert(6, 17);

g.insert(7, 29);

g.insert(8, 8);

g.insert(9, 32);

Sequence h;

h.insert(0, 63);

h.insert(1, 17);

h.insert(2, 29);

Sequence i;

i.insert(0, 17);

i.insert(1, 63);

i.insert(2, 29);

Sequence j;

j.insert(0, 29);

j.insert(1, 8);

j.insert(2, 32);

Sequence k;

k.insert(0, 32);

k.insert(1, 2);

Sequence f;

Sequence m;

Sequence l;

l.insert(17);

Sequence n;

n.insert(18);

Sequence o;

o.insert(0, 63);

o.insert(1, 17);

o.insert(2, 29);

o.insert(3, 7);

Sequence p;

Sequence q;

//subsequence tests

assert(subsequence(g, h) == 5); //subsequence present

assert(subsequence(g, g) == 0); //aliasing test

assert(subsequence(g, i) == -1); //subsequence doesn't exist consecutively

assert(subsequence(g, j) == 7); //subsequence exists at end

assert(subsequence(g, k) == -1); //k doesn't exist within g (part of k does at end)

assert(subsequence(g, f) == -1); //empty seq2

assert(subsequence(g, l) == 3); //only earliest pos is returned

assert(subsequence(f, l) == -1); //seq1 is empty

//interleave tests

interleave(f, m, l);

assert(l.size() == 0); //empty seq1 and seq2

interleave(n, n, l);

assert(l.size() == 2 && l.remove(18) == 2); //aliasing for seq1 and seq2

interleave(n, k, n);

assert(n.find(18) == 0 && n.find(32) == 1 && n.find(2) == 2 && n.size() == 3); //aliasing for seq1 and result AND seq2>seq1

n.erase(2);

n.erase(1);

interleave(k, n, n); //aliasing for seq2 and result AND seq1>seq2

assert(n.find(32) == 0 && n.find(18) == 1 && n.find(2) == 2 && n.size() == 3);

interleave(i, h, m); //seq1 and seq2 same size

ItemType y;

ItemType z;

assert(m.find(17) == 0 && m.find(63) == 1 && m.get(2, x) && x==63 && m.get(3, y) && y==17 && m.find(29) == 4 && m.get(5, z) && z==29 && m.size() == 6);

interleave(o, k, p); //seq1>seq2

assert(p.find(63) == 0 && p.find(32) == 1 && p.find(17) == 2 && p.find(2) == 3 && p.find(29) == 4 && p.find(7) == 5 && p.size() == 6);

interleave(k, o, p); //seq2>seq1

assert(p.find(32) == 0 && p.find(63) == 1 && p.find(2) == 2 && p.find(17) == 3 && p.find(29) == 4 && p.find(7) == 5 && p.size() == 6);

interleave(q, o, p); //seq1 empty

assert(p.find(63) == 0 && p.find(17) == 1 && p.find(29) == 2 && p.find(7) == 3 && p.size() == 4);

interleave(o, q, p); //seq2 empty

assert(p.find(63) == 0 && p.find(17) == 1 && p.find(29) == 2 && p.find(7) == 3 && p.size() == 4);

interleave(o, q, q); //seq2 empty AND aliasing for seq2+result

assert(q.find(63) == 0 && q.find(17) == 1 && q.find(29) == 2 && q.find(7) == 3 && q.size() == 4);

interleave(q, q, q); //aliasing for all 3

ItemType w;

assert(q.find(63) == 0 && q.get(1, x) && x==63 && q.find(17) == 2 && q.get(3, y) && y==17 && q.find(29) == 4 && q.get(5, z) && z==29 && q.find(7) == 6 && q.get(7, w) && w==7 && q.size() == 8);

cout << "Passed all tests" << endl;

}