Implementation:

I used a circular doubly-linked list so that the consideration for special cases would be unnecessary. This implied using a dummy node to ensure there was always one node in the linked list as well as to loop through the linked list.

Diagram:

…

m\_value

m\_next

m\_previous

m\_value

m\_next

m\_previous

undefined

m\_next

m\_previous

m\_value

m\_next

m\_previous

head

0

1

m\_size -1

Pseudocode:

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

*if seq2 is greater than seq1 return -1*

*if seq2 is empty return -1*

*loop through seq1*

*compare values at a point*

*if values are equal, mark that point*

*loop through seq2*

*if all values of seq2 are same as those that are looped through seq1 exit the loop*

*return the marked point*

*return -1 otherwise*

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

*Create a temporary sequence to store result*

*Erase all items in that sequence*

*Check if any of the sequences are empty*

*Loop through both sequences simultaneously*

*If the iterator for seq1 is less than its size*

*Store the value at that point in the temporary sequence*

*Increment iterators for the size of the temporary sequence and seq1*

*If the iterator for seq2 is less than its size*

*Store the value at that point in the temporary sequence*

*Increment iterators for the size of the temporary sequence and seq2*

*Make the result the same as temp result*

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

*Create a temporary pointer that points towards other’s head*

*Make other point towards the this pointer’s head*

*Make the this pointer’s head point towards the temporary pointer*

*Make a variable having the same size as that of other*

*Make the size of other same as that of mSize*

*Make mSize the same as that of the variable*

**4)** Sequence::~Sequence()

*loop through the sequence*

*create another pointer*

*place in the value of the current node into the pointer*

*make it point towards head*

*make it also point towards the last node*

*previous node points towards the pointer*

*head's previous points towards the pointer*

*move on to the next node*

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

*If position is out of bounds*

*Return -1*

*if the position is same as the size*

*create a new node that points towards the last node*

*create another node*

*set the value of temp*

*new Node (temp) points to head*

*new Node points to p*

*head points to new Node*

*node before new Node (p) points to temp*

*If the position is between the size and >= 0*

*loop through the sequence until the desired position is reached*

*set the value of temp*

*new Node (temp) points to p*

*new Node points to node before p*

*node before p points to temp*

*p points to new Node*

*return the position*

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

*If the Sequence empty*

*Create a new node*

*Link it to the head*

*Link the head to the new node*

*Increment the size*

*Else loop through the sequence until the value is found*

*If the value at a node is greater than that of the parameter*

*Change the position to where that node is in the sequence*

*If there is no such value at any node, set the position to the size*

*Insert the required value at the acquired position*

*Return the position*

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

*Check if the position is valid*

*Loop through the sequence*

*Find the required position*

*‘unhook’ the node by removing the pointers to it from the nodes previous and next to it*

*Return true*

*Return false*

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

*Check if the position is within the given bounds*

*Loop through the sequence*

*If the value is found*

*Delete that node by ‘unhooking’ it*

*Decrease the size*

*Increase the number of items removed*

*Return the number of items removed*

Test Cases: (Using ItemType = unsigned long)

For interleave and subsequence:

Sequence a;

a.insert(1);

a.insert(1);

a.insert(1);

Sequence b;

b.insert(2);

b.insert(2);

b.insert(2);

Sequence c;

interleave(a, b, c);

ItemType value;

assert(c.get(0, value) && value == 1); // does c start with seq1

assert(c.get(1, value) && value == 2); // is the next element of seq2

assert(c.size() == 6); // checking for the size

Sequence d;

d.insert(3);

d.insert(3);

d.insert(3);

Sequence e;

e.insert(4);

e.insert(4);

e.insert(5);

e.insert(6);

Sequence f;

interleave(d, e, f);

ItemType value2;

assert(f.get(4, value2) && value2 == 3); //checking if element belongs to seq1

assert(f.get(5, value2) && value2 == 5); // is the next one of seq2

assert(f.get(6, value2) && value2 == 6); // is the last one of seq2

assert(f.size() == 7); // have all the elements been included

Sequence i;

i.insert(3);

i.insert(4);

i.insert(4);

i.insert(3);

i.insert(4);

i.insert(4);

i.insert(5);

Sequence k;

k.insert(4);

k.insert(4);

k.insert(5);

assert(subsequence(i, k) == 4); // does subsequence work

For insert, empty, get and set:

Sequence a;

assert(a.insert(0, 8) == 0); //insertion with an empty list

assert(a.size() == 1); // size should change

assert(a.empty() == **false**); // since size > 0, empty should return false

assert(a.insert(8) == 0);

assert(a.insert(12) == 2); // to check ordering of insertion

assert(a.set(0, 100) == **true**); // check for changing an element

ItemType value;

assert(a.get(0, value) == **true** && value == 100); //check for getting an element

assert(a.set(30, 100) == **false**); //cannot work out of bounds

assert(a.get(50, value) == **false**); //cannot work outside range

Sequence b;

assert(b.insert(8) == 0); //insertion works for empty list

For find, remove and erase:

Sequence a;

assert(a.erase(0) == **false**); //make sure erase works with empty list

assert(a.remove(1) == 0); // does not remove anything

a.insert(32);

a.insert(33);

a.insert(33);

a.insert(35);

a.insert(64);

a.insert(100);

assert(a.erase(5) == **true**); // makes sure erase works

assert(a.size() == 5); // checks size

assert(a.remove(33) == 2); //makes sure all instances of the given value are removed

assert(a.size() == 3); // checks size

assert(a.find(40) == -1); // returns -1 if cannot be found

assert(a.find(35) == 1); // makes sure find returns the first instance in which the value is observed

For copy constructor, assignment operator and swap:

Sequence a;

a.insert(4);

a.insert(4);

Sequence b(a);

ItemType value;

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

assert(b.get(0, value) == **true** && value == 4);

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

assert(a.get(0, value) == **true** && value == 4);

Sequence Copy;

Copy = a;

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

assert(Copy.get(0, value) == **true** && value == 4);

Sequence d;

d.insert(16);

a.swap(d);

assert(a.get(0, value) == **true** && value == 16);

assert(d.get(0, value) == **true** && value == 4);