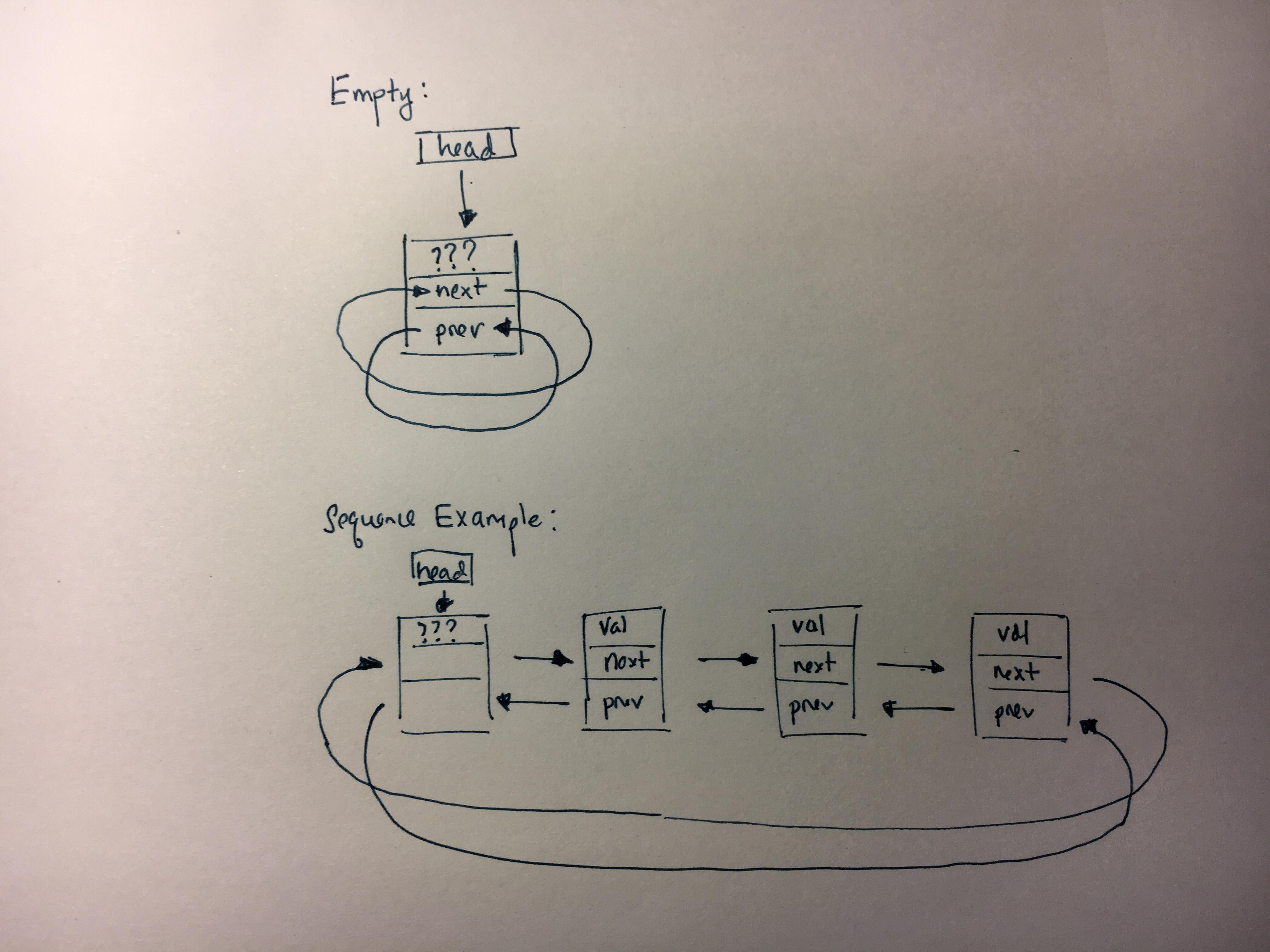
Timothy Yu

UID: 304817925

Project 2 Report

My implementation is a doubly, circularly linked list that uses a dummy node. Each node contains a value of ItemType, a next pointer, and a prev pointer. Therefore, a sequence is represented by a series of linked nodes, and it also contains a private member variable m\_size that refers to the amount of nodes in it. Due to the circular nature of the list, there is no need to deal with special cases where the next or prev pointers of the first and last nodes of the sequence point to nullptr. To illustrate the design of my implementation, the diagram below illustrates what an empty node looks like (default constructed node), as well as a generic sequence with many nodes.



Pseudocode:

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

If insert position is in front:

If sequence is empty:

Create a new insert node and set its value to specified value

Connect the dummy node to the insert node using next and prev pointers

Increment the size of the sequence

Otherwise:

Declare a pointer that points to the first valid node

Create a new insert node and set its value to specified value

Connect the insert node between the dummy and first node using next and prev pointers

Increment the size of the sequence

If insert position is in middle:

Declare a pointer to point to the first valid node and advance the pointer until it is pointing to the node at position specified

Create a new insert node and set its value to specified value

Connect the insert node between the node at position specified and the node 1 position below it using next and prev pointers

Increment the size of the sequence

If the insert position equals the size of the sequence:

Declare a pointer to point to the last node in the sequence

Create a new insert node and set its value to specified value

Connect the insert node to the last node and the dummy node with the next and prev pointers.

Increment the size of the sequence:

If none of these conditions apply

return false, otherwise the function returns true

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

Keep track of where we are in the list with an integer location variable:

Traverse through the list one node at a time:

If value is less than or equal to the value stored in one of the nodes

break

or if we finished traversing the list

break

Perform the insert function at the location where it broke

Return the location

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

If position is 0 or positive and less than size:

Declare a pointer to the first node called kill

Advance the pointer to point to the node at position specified

Connect the nodes before and after the node where kill is pointing

Delete the node where kill is pointing and decrement size of sequence

If kill happened, return true

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

Create a pointer that points to the first node

Repeatedly:

While the value stored in node pointed to is not equal to value, advance the pointer to the next node

If the value stored in node pointed to is equal to value, advance the pointer to the next node first, use erase function on the node that had the equal value, and increment count.

Return the number of counts

1. **bool Sequence::get(int pos, ItemType& value) const**

If the position is positive and less than size:

Create pointer to the first node and advance it to point to the node at position specified

Set the value variable in function to the value stored in the node at position specified and return true

If position was not in that range return false

1. **bool Sequence::set(int pos, const ItemType& value)**

If the position is positive and less than size:

Create pointer to the first node and advance it to point to the node at position specified

Set the value of the node at that position to the value specified and return true.

If position was not in that range return false

1. **int Sequence::find(const ItemType& value) const**

Traversing through the list:

If value is equal to the value stored at a node

Break

Return the location where it broke

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

Declare a temporary head pointer to swap the addresses of the dummy nodes

Declare a temporary size variable to swap the sizes of the two sequences

1. **Sequence::~Sequence()**

Repeatedly:

While the sequence is not empty:

Create a pointer to the last node, connect the node before the last node to the dummy node using next and prev pointers

Delete the last node and its pointers

Keep doing this until the sequence is empty

Delete the dummy node at the end

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

Copy the other sequences size

Initialize a new empty sequence that contains just a dummy node

Traversing through the other sequence:

Use insert function to copy in the value stored in all of other sequence’s nodes

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

Initialize 2 ItemType variables to hold values in sequence 1 and sequence 2

If sequence 2 is larger than sequence 1 or if sequence 2 is empty:

Return -1

Otherwise start traversing through sequence 1

If a value at some positional node in sequence 1 is equal to the first node value in sequence 2:

Record that position in sequence 1

Start looping through the rest of sequence 1 starting at that position and under the condition that each of the subsequent sequence values in sequence 1 and sequence 2 are equal:

If not, then break out of loop and return -1

If it reaches the end of sequence 2 without breaking, then it must be a subsequence and return the position at which the subsequence began

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

Initialize 2 ItemType variables to hold values in sequence 1 and sequence 2

If sequence 1 is empty:

Result is sequence 2

If sequence 2 is empty:

Result is sequence 1

If sequence 1 is smaller than sequence 2:

Loop through the size of sequence 2:

Under the condition that there is a valid value, alternately store in result values from sequence 1 and sequence 2 from beginning to end. Since sequence 1 is smaller, once all of its values are stored, store the rest of sequence 2 behind it.

If sequence 2 is smaller than sequence 1:

Loop through the size of sequence 1:

Under the condition that there is a valid value, alternately store in result values from sequence 1 and sequence 2 from beginning to end. Since sequence 2 is smaller, once all of its values are stored, store the rest of sequence 1 behind it.

Test Cases:

// To ensure that the default constructor works...

    Sequence s;

    // Testing empty function on an empty sequence

    assert(s.empty() == true);

    // Testing that size is 0 for an empty sequence

    assert(s.size() == 0);

    // Testing the insert function for various cases (front (empty), front (not-empty), middle, and back)

    assert(s.insert(0, "work") == true);

    assert(s.insert(0, "my") == true);

    assert(s.insert(0, "will") == true);

    assert(s.insert(1, "please") == true);

    assert(s.insert(3, "copy") == true);

    assert(s.insert(4, "constructor") == true);

    // Testing the remove function

    assert(s.remove("please") == 1);

    assert(s.remove("oio") == 0);

    // Testing that size is 5 for this sequence

    assert(s.size() == 5);

    // Copy constructor called, indeed if dump is called on both sequences they are the same

    Sequence new\_s = s;

    Sequence w;

    assert(w.insert(0, "operator") == true);

    assert(w.insert(0, "yes, ") == true);

    assert(w.insert(1, "assignment") == true);

    assert(w.insert(3, "works!") == true);

    assert(w.insert(-1, "fail") == false);

    //Assignment operator called, indeed if s.dump() is called it now equals w.dump()

    s = w;

    //s.dump();

    //w.dump();

    // Testing of the insert function that only accepts one parameter

    Sequence n;

    assert(n.insert("5") == 0);

    assert(n.insert("6") == 1);

    assert(n.insert("4") == 0);

    assert(n.insert("1") == 0);

    assert(n.insert("2") == 1);

    assert(n.insert("3") == 2);

    // Testing the erase functon

    assert(n.erase(0) == true);

    assert(n.erase(4) == true);

    assert(n.erase(1) == true);

    assert(n.erase(10) == false);

    // Further testing of the inert function that only accepts one parameter

    Sequence a;

    assert(a.insert("5") == 0);

    assert(a.insert("4") == 0);

    assert(a.insert("5") == 1);

    assert(a.insert("4") == 0);

    assert(a.insert("5") == 2);

    assert(a.insert("7") == 5);

    // Testing a remove function that removes multiple nodes

    assert(a.remove("5") == 3);

    // Testing the get, set, and find functions. As well as more remove test cases.

    string value;

    a.get(2, value);

    a.set(2, "DINOSAUR");

    a.set(1, "LEBRON");

    a.set(0, "CELTICS");

    assert(a.find("DINOSAUR") == 2);

    assert(a.find("LEBRON") == 1);

    assert(a.find("CELTICS") == 0);

    assert(a.remove("CELTICS") == 1);

    assert(a.remove("DINOSAUR") == 1);

    string val2;

    a.get(0, val2);

    assert(val2 == "LEBRON");

    // Testing the swap function

    Sequence sw1;

    Sequence sw2;

    sw1.insert(0, "I got");

    sw1.insert(1, "swapped?");

    sw2.insert(0, "Yes");

    sw2.insert(1, "I did");

    sw1.swap(sw2);

    // Testing the two algorithms subsequence and interleave

    Sequence seq1;

    Sequence seq2;

    Sequence seq3;

    // 5 4 3 2 1

    seq1.insert(0, "1");

    seq1.insert(0, "2");

    seq1.insert(0, "3");

    seq1.insert(0, "4");

    seq1.insert(0, "5");

    // 4 3 2

    seq2.insert(0, "2");

    seq2.insert(0, "3");

    seq2.insert(0, "4");

    // Subseq occurs at position 1

    assert(subsequence(seq1, seq2) == 1);

    // Seq 3 is empty so should return -1

    assert(subsequence(seq1, seq3) == -1);

    // Seq 2 is smaller than Seq 1

    assert(subsequence(seq2, seq1) == -1);

    // 2 1

    seq3.insert(0, "2");

    seq3.insert(1, "1");

    // Subseq occurs at position 3

    assert(subsequence(seq1, seq3) == 3);

    Sequence result;

    Sequence result2;

    // result.dump() indeed prints out 5 4 4 3 3 2 2 1

    interleave(seq1, seq2, result);

    // result2.dump() indeed prints out 5 2 4 1 3 2 1

    interleave(seq1, seq3, result2);

}