**Report**

**Doubly Linked List:**

For this project, we used a doubly linked list, where the class Sequence had a private struct known as Node. Inside of the Node struct, we had members of value, which stored the value at each node, a pointer to Node named prev and a pointer to Node called next. Finally, Sequence had a private pointer to Node known as head which stored the first node. The linked list was not circular. If the sequence was empty, then head would be a nullptr; however, if there were items in the sequence, then head would have nodes coming after it and its previous node would be a nullptr.

**Pseudocode:**

*Sequence::Sequence(Sequence &old)* {

*New node for head*

*Assign head values*

*Repeatedly*

*Insert new node*

*Set new nodes value to corresponding old node value*

*Increment*

*Set sizes equal   
}*

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

*Aliasing check*

*If (head is null)*

*Insert head node*

*If (LHS sequence is greater)*

*Set head values equal*

*Repeatedly*

*Put in values of old sequence*

*Increment*

*Repeatedly*

*Delete excess values*

*If (RHS sequence is greater)*

*Set head values equal*

*Repeatedly*

*Replace values with those of new sequence*

*Increment*

*Repeatedly*

*Insert new nodes for excess values*

*If (sizes are equal)*

*Set head values equal*

*Repeatedly*

*Set values equal to each other*

*Increment*

*}*

*Sequence::~Sequence() {*

*Repeatedly*

*Delete node*

*Increment*

*}*

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

*if (value found at head)*

*if (next node not null)*

*Delete head node*

*Replace with one after*

*Increment counter*

*Else*

*Delete head node*

*Increment counter*

*Repeatedly*

*If node above found*

*Replace node with node after*

*Increment counter*

*Return counter*

*}*

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

*if (pos is first node)*

*delete head*

*Move head->next to head*

*Return true*

*Repeatedly*

*If (you hit node above pos)*

*If (pos is not null)*

*Replace the value with the one after*

*Else*

*Delete the pos position as it is the last node*

*Increment counter*

*Increment loop*

*Return false if nothing happens*

*}*

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

*if (no values in sequence)*

*Insert as head node*

*Increment size*

*Return true*

*If (pos is first value )*

*Move head node up one*

*Set value as head node*

*Increment*

*Return true*

*Repeatedly*

*If (node above pos)*

*Break*

*Increment*

*Insert a new node*

*Move all nodes above it up one*

*Set new node to value in question*

*Increment size*

*Return true*

*}*

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

*if (empty or head >= value)*

*Insert a value at head*

*Move other values up one*

*Repeatedly*

*If (node above pos found)*

*Insert a new node*

*Set all nodes above it up one*

*Increment size*

*Return counter*

*If no value inserted after loop*

*Insert value at last node*

*Return the size of sequence*

*}*

*int Sequence::find(const ItemType &value) {*

*Repeatedly*

*If (value found)*

*Return counter*

*Increment counter*

*Increment loop*

*}*

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

*Repeatedly*

*If pos found*

*Set value passed in to value at pos*

*Return true*

*Increment counter*

*Return false if nothing found*

*}*

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

*Repeatedly*

*If pos found*

*Set value at pos to that passed in*

*Return true*

*Increment counter*

*Return false if nothing happens*

*}*

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

*switch the heads of each sequence*

*switch the sizes of each sequence*

*}*

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

*if (preconditions not met)*

*return -1;*

*Repeatedly*

*If (equal values found and if size will not be passed)*

*Set temp values*

*Repeatedly*

*Replace value1 and value2 with get function*

*If (size hit)*

*Return temp*

*Increment*

*Return -1 if it didn’t work*

*}*

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

*aliasing precondition*

*assign size to the greater size of the two*

*empty out result*

*Repeatedly*

*If (size1 > size2 and x passed second sequence size)*

*Add values only from first sequence*

*If (size2 > size1 and x passed first sequence size)*

*Add values only from second sequence*

*Else*

*Add values from both sequence*

*}*

***Test Cases:***

Sequence q;

q.insert(0, "Nice");

assert(q.find("Nice") == 0);

q.insert(1, "Nicer");

//check if find works properly

assert(q.find("Nicer") == 1);

//Check if insert insert works

ItemType value;

q.get(0, value);

assert(value == "Nice");

//Check if get changes the value

q.insert("value");

assert(q.find("value") == 2);

//Check if insert places value based on greater than or equal to

Sequence r;

r.insert("Swap");

r.insert(1, "The");

q.swap(r);

assert(!(q.find("Nicer") == 1));

assert(q.find("Swap") == 0);

assert(r.find("Nicer") == 1);

//Check if swap executes properly

assert(r.remove("Nicer") == 1);

assert(r.find("Nicer") == -1);

//Check if remove will take out the values

Sequence t;

t.insert(0, "Hello");

assert(t.remove("Hello") == 1);

assert(t.find("Hello") == -1);

//Checks if remove works for the first value

t.insert("Some");

t.set(0, "New");

assert(t.find("New") == 0);

//Check if the set function works on the first value

Sequence s1;

Sequence s2;

s1.insert("1");

s2.insert("1");

s1.insert(1, "2");

s2.insert(1, "3");

s1.insert(2, "3");

s1.insert(3, "1");

s1.insert(4, "3");

//s1 = {1, 2, 3, 1, 3};

//s2 = {1, 3}

assert(subsequence(s1, s2) == 3);

//Checks if subsequence executes properly

Sequence s3;

Sequence s4;

Sequence s5;

s3.insert(0, "0");

s4.insert(0, "1");

s3.insert(1, "2");

s4.insert(1, "3");

s3.insert(2, "4");

s4.insert(2, "5");

s4.insert(3, "6");

s4.insert(4, "7");

s4.insert(5, "8");

interleave(s3, s4, s5);

assert(s5.find("0") == 0);

assert(s5.find("4") == 4);

assert(s5.find("8") == 8);

//Checks if interleave executed properly

assert(s5.insert(6, "9") == true);

//Checks if insert is possible after interleaving

Sequence s7;

s7.insert(0, "copy");

s7.insert(1, "cat");

s7.insert(2, "values");

Sequence s8 = s7;

assert(s8.find("copy") == 0);

assert(s8.find("cat") == 1);

assert(s8.find("values") == 2);

assert(s8.size() == 3);

//Checks the execution for the copy constructor

Sequence s9;

s9.insert("1");

s9.insert("2");

s9.insert("3");

s9.insert("4");

s9 = s7;

assert(s9.find("copy") == 0);

assert(s9.find("cat") == 1);

assert(s9.find("values") == 2);

assert(s9.size() == 3);

//Checks execution of assignment operator with size > other.size

Sequence s10;

s10.insert("It's");

s10.insert("Okay");

s10.insert("This");

s10.insert("Will");

s10.insert(4, "Get Copied Over");

s10 = s9;

assert(s10.find("copy") == 0);

assert(s10.find("cat") == 1);

assert(s10.find("values") == 2);

//Checks assignment operator

assert(s10.find("This") == -1);

//Checks if original values are gone

assert(s10.size() == 3);

Sequence s11;

s11.insert(0, "Hello");

s11.insert("Zebra");

assert(s11.find("Zebra") == 1);

//Checks if insert works with placing values greater than others at the end

assert(s11.size() == 2);

Sequence s12;

assert(s12.size() == 0);

s12.insert("Value");

assert(s12.size() == 1);

assert(!(s12.empty()));

//Checks the insert and empty functions

s12.insert("cello");

assert(s12.size() == 2);

assert(s12.insert(1, "cool"));

assert(s12.find("cool") == 1);

assert(s12.insert(-1, "not inserted") == false);

assert(s12.insert(4, "not inserted") == false);

//Checks if the preconditions for position on insert work

Sequence s13;

assert(s13.insert("someething") == 0);

assert(s13.insert("a") == 0);

//Check if insert with no pos will work if it is meant to be the first value

assert(s13.find("someething") == 1);

assert(s13.size() == 2);

assert(s13.find("Someething") == -1);

//Check case sensitivity

Sequence s14;

s14.insert("Somevalue");

s14.erase(0);

assert(s14.size() == 0);

//Check if erase executes properly on first element

s14.insert(0, "this");

s14.insert(1, "value");

s14.insert(2, "will");

s14.insert(3, "be");

s14.insert(4, "deleted");

s14.erase(2);

assert(s14.find("be") == 2);

assert(s14.find("deleted") == 3);

assert(s14.size() == 4);

//Check if erase works properly on middle element. Moves all elements down one

s14.erase(3);

assert(s14.size() == 3);

//Check if erae works properly on last element

Sequence s15;

Sequence s16;

Sequence s17;

s15.insert("Hello");

s16.insert(0, "Goodbye");

s17.insert("Hopefully");

s17.insert(1, "This");

s17.insert(2, "Gets");

s17.insert(3, "Overwritten");

interleave(s15, s16, s15);

assert(s15.find("Hello") == 0);

assert(s15.find("Goodbye") == -1);

//Check if aliasing precondition works

interleave(s15, s16, s17);

assert(s17.find("Overwritten") == -1);

//Checks if original values are erased in interleave

std::cerr << "Passed";