**CS 32 Project 2 Report**

**Doubly linked list implementation:**

Head and Tail of Sequence are initialized to nullptr. As elements are inserted, the sequence updates so the head points to first element in Sequence and tail points to last element of Sequence. The prev pointer in the first element of Sequence points to nullptr and the next pointer of the last element of Sequence points to nullptr. The next pointers of every other element point to the next element and the prev pointers of all other elements points to the previous element.

This is what my Sequence looks like with 3 item in it:

\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_

head —> | storedVal | | storedVal | | storedVal | <— tail

| \_\_\_\_\_\_\_\_\_\_| |\_\_\_\_\_\_\_\_\_\_\_| |\_\_\_\_\_\_\_\_\_\_\_|

| next——|— |—> next — | —|—> next —|—> nullptr

|\_\_\_\_\_\_\_\_\_\_\_| |\_\_\_\_\_\_\_\_\_\_\_| |\_\_\_\_\_\_\_\_\_\_|

nullptr <—|— prev <—|— |— prev <—|—|— prev |

|\_\_\_\_\_\_\_\_\_\_\_| |\_\_\_\_\_\_\_\_\_\_\_| |\_\_\_\_\_\_\_\_\_\_ |

**Pseudo Code:**

1)

Sequence::Sequence() : Initialize head and tail to nullptr, set size to 0

{

}

2)

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

{

Return -1 if accessing an out of range position

Declare pointer to a new Node

if empty list, set head and tail to point to new Node

Increment size by 1

return 0

if pos is 0

Set new Node’s next pointer to point to first element in list

Set prev pointer of new Node to point to nullptr

Set head to point to new Node

Set prev pointer of element after new Node to point to new Node

Increment size by 1

return 0;

if pos is size of list

Set next pointer of new Node to point to nullptr

Set new Node’s prev pointer to point to last element in list

Set tail to point to new Node

Set next pointer of element before new Node to point to new Node

Increment size by 1

return pos;

if position is between 0 and size

Iterate through Sequence until position is reached

Set next pointer of new Node to point to element which will come after it

Set prev pointer of new Node to point to element which will come before it

Set prev pointer of element after new Node to point to new Node

Set next pointer of element before new Node to point to new Node

if pos is size of list, set next pointer of new Node to point to nullptr

Increment size by 1

return pos;

}

3)

int Sequence::insert(const ItemType&value)

{

Declare pointer to a new Node

if empty list, set head and tail to point to new Node

Increment size by 1

return 0;

if value is smaller than first element in Sequence

Set new Node’s next pointer to point to first element in list

Set prev pointer of new Node to point to nullptr

Set head to point to new Node

Set prev pointer of element after new Node to point to new Node

Increment size by 1

return 0;

Iterate through Sequence until value is less than an element

if value is bigger than all items in Sequence

Set next pointer of new Node to point to nullptr

Set new Node’s prev pointer to point to last element in list

Set tail to point to new Node

Set next pointer of element before new Node to point to new Node

Increment size by 1

return pos;

If value is less than an element

Set next pointer of new Node to point to element which will come after it

Set prev pointer of new Node to point to element which will come before it

Set prev pointer of element after new Node to point to new Node

Set next pointer of element before new Node to point to new Node

Increment size by 1

return pos;

return pos;

}

4)

bool Sequence::erase(int pos)

{

Return -1 if accessing an out of range position

If there is only one item in list

Declare a pointer to only Node

Reassign head and tail to nullptr

Delete node

Decrease size by 1

return true;

If deleting first Node in Sequence

Declare pointer to first Node

Set prev pointer of second Node to point to nullptr

Set head to point to second Node

Decrease size by 1

return true;

If deleting last Node in Sequence

Declare pointer to last Node

Set next pointer of second-last Node to point to nullptr

Set tail to point to second-last Node

Decrease size by 1

return true;

Loop through Sequence until we reach 1 before pos

Declare pointer to Node to be erased

Set next pointer of Node before Node to be erased to point to the Node after

Set prev pointer of Node after Node to be erased to point to the Node before

Decrease size by 1

return true;

return false;

}

5)

Int Sequence::remove(const ItemType& value)

{

Initialize counter for number of items deleted

Initialize counter for current position

Loop through Sequence

if value of item in sequence equals argument value

Call erase function

Increment counter for number of items deleted by 1

Decrease counter for current position by 1

Increment counter for current position by 1

return counter;

}

6)

Int subsequence(const Sequence& seq1, const Sequence& seq2)

{

Return -1 if either sequence is empty or seq2 is larger than seq

Initialize counter for number of commons elements

Initialize position to keep track of first common index

Loop through seq1

If item in seq1 matches first item in seq2

Iterate through seq2

If number of common elements equals size of seq2, return position of first common index;

if consecutive items do not match, reset counter and common index to 0

else increment counter by 1

return -1;

}

7)

void concatReverse(const Sequence& seq1, const Sequence& seq2, const Sequence& result)

{

Loop through seq1 in decreasing order

Get item in position of seq1

Insert item at end of result

Loop through seq2 in decreasing order

Get item in position of seq1

Insert item at end of result

Loop through result

Get item at current position in result

}

**Test Cases:**

// Default constructor

Sequence s;

assert(s.size() == 0); // Ensure sequence has size 0 upon initialization

assert(s.empty()); // Test empty function

assert(s.remove(“paratha” == 0); // Nothing to remove

assert(s.insert(0, “First”) == 1); // Test for inserting element into empty Sequence

assert(s.insert(0, “Me first!”) == 0); // Test for inserting element at position 0

assert(s.insert(s.size(), “Last”) == s.size()); // Test for inserting element at end of Sequence

assert(s.insert(2, “Middle”) == 2); // Test to insert item in middle of Sequence

assert(s.insert(5, “Something) == -1); // Test to insert item out of bounds

// Default constructor

Sequence s1;

assert(s1.insert(“CCC”) == 0); // Test for inserting element into empty Sequence

assert(s1.insert(“AAA”) == 0); // Test for inserting element at position 0

assert(s1.insert(“FFF”) == s.size()-1); // Test for inserting element at end of Sequence

assert(s1.insert(“BBB”) == 1); // Test to insert item in middle of Sequence

assert(s1.insert(“BBB”) == 1); // Edge case where item == value before it

assert(s1.erase(s1.size()) == false); // Test to erase at position out of bounds

assert(s1.erase(0) == true); // Test to erase first item in Sequence

assert(s1.erase(s1.size() - 1) == true); // Test to erase last item in Sequence

assert(s1.erase(2) == true); // Test to erase middle item in Sequence

// Default constructor

Sequence s2;

assert(s2.insert(“CCC”) == 0);

assert(s2.erase(0) == true); // Test to remove single element Sequence

assert(s2.empty()); // Test empty

// Default constructor

Sequence s3;

assert(s3.insert(“aaa”) = 0);

assert(s3.remove(“aaa”) == 1); // Test to remove single element Sequence

assert(s3.empty()); // Test empty

assert(s3.insert(“aaa”) == 0);

assert(s3.insert(“ccc”) == 1);

assert(s3.insert(“bbb”) == 1);

assert(s3.insert(“eee”) == 3);

assert(s3.insert(“ddd”) == 3);

assert(s3.insert(“ddd”) == 3);

assert(s3.insert(“fff”) == 6);

assert(s3.remove(“ggg”) == 0); // Test to remove no items

assert(s3.remove(“fff”) == 1) // Test to single item at end of Sequence

assert(s3.remove(“ddd”) == 2); // Test to remove multiple items

assert(s3.remove(“aaa”) == 1); // Test to remove item at beginning of Sequence

// Default constructor

Sequence s4;

assert(s4.insert(“aaa”) == 0);

assert(s4.insert(“ccc”) == 1);

assert(s4.insert(“bbb”) == 1);

assert(s4.insert(“eee”) == 3);

assert(s4.insert(“ddd”) == 3);

assert(s4.insert(“ddd”) == 3);

assert(s4.insert(“fff”) == 6);

ItemType copyMe;

assert(s4.get(-1, copyMe) == false); // Test to get out of bound item

assert(s4.get(s4.size(), copyMe) == false); // Test to get out of bound item

assert(s4.get(0, copyMe)) == true); // Test to get first item in Sequence

assert(s4.get(s4.size() - 1, copyMe) == true); // Test to get last item in Sequence

assert(s4.get(3, copyMe) == true; // Test to get middle item in Sequence

assert(s4.set(7, “blah”) == false); // Test to set out of bound item

assert(s4.set(0, “Blah”) == true); // Test to set first item in Sequence

assert(s4.set(s.size() - 1, “blorg”) == true); // Test to set last item in Sequence

assert(s4.set(3, “maybe”) == true); // Test to set middle item in Sequence

assert(s4.find(“zzzzzz”) == -1); // Test to find non-existent item

assert(s4.find(“maybe”) == 3); // Test to find middle item in Sequence

assert(s4.find(“blorg”) == s.size() - 1); // Test to find last item in Sequence

assert(s4.find(“Blah”) == 0); // Test to find first item in Sequence

// Default Constructor

Sequence s5;

Sequence s6;

s5.insert(a1);

s5.insert(b1);

s5.insert(c1);

s6.insert(a2);

s6.insert(b2);

s6.insert(c2);

s6.insert(d2);

swap(s5, s6);

assert(s5.size() == 4 && s6.size() == 3); // Test if sizes are swapped

assert(s5.find(“a2”) == 0 && s6.find(“a1”) == 0); // Test if first elements are swapped

assert(s5.find(“d2”) == s5.size()-1 && s6.find(“c1”) == s6.size()-1); // Test if last elements are swapped

// Copy Constructor Tests

Sequence s7;

s7.insert(“ABBA”);

s7.insert(“Pink Floyd”);

s7.insert(1, “baguette”);

Sequence s8(s7);

assert(s8.size() == 3); // Test if size is copied

assert(s8.find(“ABBA”) == 0) // Test if first item in Sequence is copied

assert(s8.find(“Pink Floyd”) == s8.size() - 1); // Test if last item in Sequence is copied

assert(s8.find(“baguette”) == 1); // Test if middle item in Sequence is copied

// Assignment Operator Tests

Sequence s9;

Sequence s10;

s9.insert(“ABBA”);

s9.insert(“Pink Floyd”);

s9.insert(1, “baguette”);

s10 = s9;

assert(s10.find(“ABBA”) == 0); // Test that assignment operator that works

// Subsequence Tests

Sequence s13;

Sequence s14;

assert(subsequence(s13, s14) == -1); // Test for empty sequences

s13.insert(“a”);

s13.insert(“b”);

s13.insert(“c”);

s13.insert(“d”);

s13.insert(“e”);

s14.insert(“c”);

s14.insert(“d”);

s14.insert(“e”);

assert(subsequence(s13, s14) == 2); // Test for subsequence beginning at middle of s13

s13.remove(“e”);

assert(subsequence(s13, s14) == -1); // Test when there is no subsequence

s14.remove(“e”);

s13.remove(“a”);

s13.erase(0);

assert(subsequence(s13, s14) == 0); // Test for subsequence at beginning of s13

// concatReverse Tests

Sequence s11;

Sequence s12;

Sequence reverse;

concatReverse(s11, s12, reverse);

assert(reverse.size() == 2); // Test for size of reverse.size

s11.insert(“a”);

s11.insert(“b”);

s11.insert(“c”);

s11.insert(“d”);

s11.insert(“e”);

s12.insert(“x”);

s12.insert(“y”);

s12.insert(“z”);

assert(reverse.find(“e”) == 0); // Test for item at beginning of reverse

assert(reverse.find(“x”) == reverse.size() - 1); // Test for item at end of reverse

assert(reverse.find(“a”) == 4); // Test for item in middle of reverse