**CS 32 Project 2 Report**

**Design**

The structure I chose to for the doubly linked list was the most conventional one. Therefore, I did not apply any circular node or dummy node to my linked list. I used a structure containing four variables in it for my nodes: a pointer that points to the previous node (*m\_before*), a pointer that points to the next node (*m\_next*), a variable of *ItemType* that stores the value we want (*m\_value*), and an integer (*m\_position*) to indicate the position of the node. A head pointer pointing to the start of the linked list and a tail pointer pointing to the end of the list are also used to tell where the sequence starts and where it ends. The structure of the design is shown below.

**Struct Node**

|  |
| --- |
| *m\_before* |
| *m\_next* |
| *m\_value* |
| *m\_position* |

**Data Structure of Sequence**

*m\_tail*

*m\_head*

Node1 Node2 Node3 Node4

m\_position

m\_position

m\_position

m\_position

m\_value

m\_value

m\_value

m\_value

m\_next

m\_next

m\_next

m\_next

m\_before

m\_before

m\_before

m\_before

NULL

Value 4

3

NULL

Value 1

0

Value 2

1

Value 3

2

**PSEUDOCODE OF NON-TRAVIAL FUNCTIONS:**

sequence ():

Set the sequence size to 0

Set head and tail pointer to null

~sequence ():

Declare *temp* pointing to the head

For *temp* is not null; after each loop let *temp* point to next element

Advance head to its next pointer

Delete the element pointed by *temp*

sequence (const Sequence & another\_seq):

if another\_seq is empty

set the size of copy to be 0

set the head and tail of the copy to nullptr

else if another\_seq is not empty

set the size of the copy to be the size of another\_seq

set the head of the copy to point to a new Node

initialize the first element’s before pointer to be null

copy the value of another\_seq’s element one

declare *temp* points to this’s head

declare *temp2* points to another\_seq’s second element

for *temp2* is not null; advance *temp2* and *temp* each round of the loop

assign new Node to *temp*’s next pointer

assign *temp*’s next element’s before pointer to *temp*

assign *temp*’s next element’s position to be previous element’s position + 1

copy the value of *temp2* to value of *temp*’s next element

assign *temp*’s next to be null and assign tail to point to *temp*

operator = (const Sequence & another\_seq):

if this is not another\_seq:

Copy construct *temp* to another\_seq

Swap with *temp*

Return reference of this pointer

insert (int pos, const ItemType & value):

Return false if pos is negative or greater than size

If this is empty:

Set head and tail pointer to new node

Increment the size

Return true

If insert at the head

Declare *temp* points to original head

Let the head point to a new node

Build the link to the element *temp* points, set *m\_before* to be null

Increment position for all old elements

Increment size

Return true

If insert in the middle

Build links to the previous element and next element

Increment position for all element behind the new element

Increment the size

Return true

If insert in the tail

Let the original element’s next points to a new Node.

Copy the value to the new element

Set its *m\_before* to the original final item, set its *m\_next* to be null

Increment size and return true

insert (const ItemType & value):

If the list is empty:

Insert at position 0 and return 0

Repeatedly:

look down the linked list

If value is less or equal that m\_item:

Insert at that node’s position and return this position

If no such node is found

Insert value at position = size and return size

erase (int pos):

Return false is pos is negative or greater equal to the size

If there is only one element

Delete the head and tail

Set head and tail to null

Return true

Else if there are more than one element

If pos is 0:

Move head pointer to next element

Delete the old head

Decrement position for all old element

If pos is the last position:

Move tail pointer to the previous element

Delete the old last tail

If pos is in the middle

Delete the node at pos and rebuild the link

Change the position of all element after it

Decrement size

Return true

remove (const ItemType & value):

Initialize count to 0

Apply find function to find the value

If find function cannot find it

Return 0

When the find function find value, repeatedly:

Use erase function to delete the item at position given by find

Use find to find the value again

Increment count

Return count

get (int pos, ItemType & value) const:

Return false if pos is negative or greater or equal to the size

Using next pointer to follow down the linked list until position pos

Assign that node’s *m\_value* to value

Return true

Set (int pos, const ItemType & value):

Return false if pos is negative or greater equal to size

Using next pointer to follow down the linked list until position pos

Assign value to that node’s *m\_value*

Return true

find (const ItemType & value) const:

Repeatedly:

Look down the list until a matched value is find or reach the end of the list

Return -1 if reached the end of the list

Otherwise return the position when a match is found

swap (Sequence & other):

swap the size of this and other

swap the head of this and other

swap the tail of this and other

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

Return -1 if seq1’s size is smaller than seq2’s size or if seq2 is empty

Set ItemType *temp*2 to seq2’s element at position 0

Declare ItemType temp

Declare bool equal

Repeatedly:

Get the next element of seq1 and set it to *temp*

If current value of *temp* equals *temp2*

Set equal = true

Repeatedly:

Check if next element of seq2 matches seq 1

Break if equal is false or reach the end of seq2

If equal is true after the previous loop

Return the current position

Else set *temp2* back the first element of seq2

Return -1

Interleave (const Sequence & seq1, const Sequence & seq2, Sequence & result):

If result is not empty:

Empty the list

Set an int minSize to be the smaller size of seq1 and seq2

Set an int count to be 0

Initialize i to be 0, for i smaller than minSize; increment i each loop

Copy item at position i of seq1 to result at position count

Increment count

Copy item at position i of seq2 to result at position count

Increment count

If seq1 is larger than minSize

Copy the rest of the item in seq1 to result

Else if seq2 is larger than minSize

Copy the rest of the item in seq2 to result

**Test Cases**

**// Constructor & Sequence::empty() test**

Sequence a,b,c;

assert(a.empty() && b.empty() && c.empty());

**// Two Sequence::insert() function and Sequence::size() test**

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

assert(a.insert("b") == 1);

for(int i = 0; i < 10; i++)

assert(b.insert(i, "hello"));

assert(b.size()==10);

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

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

assert(a.find("b") == 1);

assert(b.find("hello") == 0);

assert(a.find("c") == -1 && b.find("wow") == -1);

//set cases of returning false

assert(!a.insert(-1, "wow"));

assert(!a.insert(3, "wow"));

assert(!c.insert(1, "wow"));

**// Copy constructor test**

//outside main

void copytest(Sequence s) {

assert(s.insert(0, "test"));

assert(s.find("test")==0);

}

//inside main, a is not changed

copytest(a);

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

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

assert(a.find("b") == 1);

copytest(c);

assert(c.empty());

**//Assignment Operator Test**

Sequence d,e,f;

d = d;

assert(d.empty());

d = a;

assert(d.size() == a.size() && d.find("a") == a.find("a"));

f=e=d=d;

assert(f.size() == a.size() && e.find("a") == a.find("a") && e.find("b") == f.find("b"));

**//Sequence::erase() and Sequence::remove() Test**

//general case

assert(a.erase(0) && !a.erase(1));

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

assert(a.find("a") == -1);

assert(a.find("b") == 0);

assert(b.remove("hello") == 10);

assert(b.empty());

//empty sequence

assert(c.empty());

assert(!c.erase(0));

assert(c.remove("a") == 0);

assert(c.empty());

//one element sequence

c.insert(0, "wow");

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

assert(c.remove("wo") == 0);

assert(c.remove("wow") == 1);

assert(c.empty());

c.insert(0, "wow");

assert(!c.erase(1));

assert(c.erase(0));

assert(c.empty());

**//Sequence::get() and Sequence::set Test**

ItemType value = "hello world";

//for empty sequence

assert(!c.get(0,value));

assert(!c.set(0,value));

assert(value == "hello world");

//general case

assert(d.get(1, value));

assert(value == "b");

assert(d.set(0, value));

assert(d.get(0,value));

assert(value == "b");

**//Sequence::find() and Sequence::swap Test**

assert(c.find("\0") == -1);

assert(c.find("1") == -1);

assert(d.find("b") == 0);

assert(e.find("b") == 1);

c.swap(d);

assert(d.empty());

assert(c.find("b") == 0);

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

**//Function Subsequence() Test**

string seq1 [5] = {"hello","are","you","ok","?"};

string seq2 [5] = {"hello","you", "are","ok","?"};

string seq3 [3] = {"you","ok","?"};

Sequence test1,test2,test3,test4,test5;

for(int i=0;i<5;i++){

test1.insert(i, seq1[i]);

test2.insert(i, seq2[i]);

}

for(int i=0;i<3;i++){

test3.insert(i, seq3[i]);

}

test4.insert("?");

assert(subsequence(test1, test2) == -1);

assert(subsequence(test1, test3) == 2);

assert(subsequence(test2, test4) == 4);

assert(subsequence(test5, test5) == -1);

assert(subsequence(test1, test5) == -1);

assert(subsequence(test1, test1) == 0);

assert(subsequence(test4, test4) == 0);

**//Function interleave() test**

Sequence samp1,samp2,samp3,samp4,result;

for(int i = 0; i < 10; i++)

assert(samp1.insert(i, "hello"));

for(int i = 0; i < 15; i++)

assert(samp2.insert(i, "Jennifer"));

interleave(samp1, samp2, result);

assert(result.size()==25);

assert(result.find("Jennifer")==1 && result.find("hello")==0);

ItemType element;

result.get(20, element);

assert(element == "Jennifer");

//for empty sequences

interleave(samp3, samp1, result);

assert(result.size() == samp1.size());

assert(result.find("hello") == 0);

for(int i = 0; i<9 ;i++)

assert(result.get(9, element) && element == "hello");

interleave(samp2, samp4, result);

assert(result.size() == samp2.size());

assert(result.find("Jennifer") == 0);

assert(result.get(9, element) && element == "Jennifer");

interleave(result, samp1, result);

assert(result.find("hello") == 1);