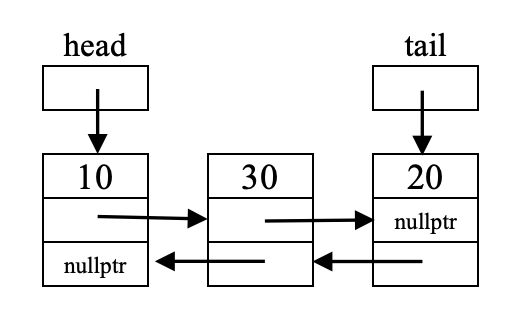
Report

**Description**

I structured my Sequence to be a doubly-linked list including a private struct called Node which contains variables ItemType value, Node \*next, and Node \*prev. Sequence also contains private member variables Node \*head, Node \*tail, and int length.

This is what a typical Sequence looks like:



And this is what an empty Sequence looks like:pasted-image.tiff

My implementation of the list is not circular nor does it have a dummy node.

**Psuedocode**

**Sequence::~Sequence()**

// start at the head

// iterate through the list

// create a pointer to the current node

// current node moves to the next node

// delete the temp (previously current) node

// reset variables

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

// initialize member variables

// iterate through the list

// create a new node

// if first node

// set head and tail to the new node

// update the new node’s pointers

// if not first node

// update the new node’s pointers

// set previous node’s next to this node

// update tail to the new node

// copy value from the original node to the new node

// move to the next node in the original list

// increase length

**Sequence& Sequence::operator = (const Sequence &rhs)**

// if they’re not equal

// create a temp sequence with what you want to copy

// swap the current sequence with the temp sequence

// return the current sequence

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

// return -1 if out of bounds

// create a new node with the passed in value

// check if inserting in front

// update the new node’s pointers

// if not inserting into an empty list

// assign the head node’s prev to the new node

// assign the head to the new node

// if inserting into an empty list

// assign the tail to the new node

// check if inserting in back

// assign the new node’s next to a nullptr and the new node’s prev to the current tail

// if not inserting into an empty list

// assign the current tail’s next to the new node

// assign tail to the new node

// if inserting into an empty list

// assign the head to the new node

// check if inserting in middle

// loop through the list until the temporary node is pointing at the node in the desired position

// assign all the pointers

// increase the length

// return the position

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

// create a new node with the passed in value

// iterate through the list until you find a value in the list that’s larger than the passed in value

// check if inserting in front

// update the new node’s pointers

// if not inserting into an empty list

// assign the head node’s prev to the new node

// assign the head to the new node

// if inserting into an empty list

// assign the tail to the new node

// check if inserting in back

// update the new node’s pointers

// if not inserting into an empty list

// assign the current tail’s next to the new node

// assign tail to the new node

// if inserting into an empty list

// assign the head to the new node

// check if inserting in middle

// assign all the pointers

// increase the length

// return the position

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

// return false if out of bounds

// iterate through the list until the pointer points to the node at pos

// if removing the head

// if it’s not a single item

// make the node after this one the head

// make the head’s prev be a nullptr

// if it’s a single item

// make the head and tail a nullptr

// delete the node

// if removing the tail

// if it’s not a single item

// make the node before this one the tail

// make the node after the tail a nullptr

// if it’s a single item

// make the head and tail a nullptr

// delete the node

// if removing something in the middle

// adjust the pointers of the nodes next to it

// delete the node

// decrease the length

// return true

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

// iterate through the entire list

// store the next node before deleting

// if the values match

// erase the node

// increase the number of items deleted

// decrease the loop counter

// look at the next item of the list

// return the number of items deleted

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

// return false if out of bounds

// iterate through the entire list until you reach the desired position

// set value to the node’s value

// return true

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

// return false if out of bounds

// iterate through the entire list until you reach the desired position

// set the node’s value to value

// return true

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

// iterate through the entire list

// if the node’s value is equal to the value you’re looking for

// return the position

// increment the position counter

// look at the next item of the list

// return -1 if no position is found

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

// swap the heads

// swap the tails

// swap the lengths

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

// return -1 if the subsequence is larger or if either sequences are empty

// loop through seq1

// if the value of seq1 is equal to the start of seq2, place a marker

// loop through the length of seq2

// check each value is equal

// if each value is equal, return the position marker

// return -1 if nothing is found

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

// create a new sequence that copies the result sequence

// clear the copy of result

// flip seq1 and add it to the result copy sequence

// flip seq2 and add it to the result copy sequence

// assign result with whatever’s in the result copy sequence

**Test Cases**

// constructor

Sequence a;

assert(a.empty() == 1); // empty = true

assert(a.size() == 0); // size = 0

// copy constructor

cerr << "COPY CONSTRUCTOR" << endl;

Sequence a2 = a; // copying empty list

a2.dump(); // [empty]

// insert(value)

cerr << endl << "INSERT(VALUE)" << endl;

a.insert("i"); // insert into empty sequence

a.insert("love"); // insert at back, one-element list

a.insert("cats"); // insert at front

a.insert("so"); // insert at back

assert(a.insert("much") == 3); // insert in middle

a.dump(); // cats i love much so

assert(a.empty() == 0); // empty = false

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

// copy constructor pt.2

cerr << endl << "COPY CONSTRUCTOR PT.2" << endl;

Sequence a3 = a;

a3.dump(); // cats i love much so

// get

string str;

a.get(3, str);

assert(str == "much");

a.get(4, str);

assert(str == "so");

// insert(pos, value)

cerr << endl << "INSERT(POS, VALUE)" << endl;

Sequence b;

b.insert(0, "are"); // insert into empty sequence

b.insert(1, "than"); // insert at back, one-element list

b.get(1, str);

assert(str == "than");

b.insert(1, "better"); // insert middle

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

b.get(1, str);

assert(str == "better");

assert(b.insert(3, "dogs") == 3); // insert at back

assert(b.insert(0, "cats") == 0); // insert at front

assert(b.insert(-1, "meow") == -1); // out of bounds

assert(b.insert(6, "meow") == -1); // out of bounds

b.dump(); // cats are better than dogs

// find

assert(b.find("better") == 2); // middle

assert(b.find("cats") == 0); // beginning

assert(b.find("dogs") == 4); // end

// assignment operator

cerr << endl << "ASSIGNMENT OPERATOR" << endl;

Sequence b2;

b2.operator=(b);

b2.dump(); // cats are better than dogs

b2.set(4, "everything");

b.dump(); // cats are better than dogs

b2.dump(); // cats are better than everything

Sequence c;

c.insert("meow");

c.insert("meow");

c.insert("meow");

// find pt.2

assert(c.find("meow") == 0); // multiple instances of the same word

assert(c.find("woof") == -1); // nonexistent

// swap

cerr << endl << "SWAP" << endl;

c.swap(b);

b.dump(); // meow meow meow

c.dump(); // cats are better than dogs

// erase

cerr << endl << "ERASE" << endl;

assert(c.erase(-1) == 0); // out of bounds

c.erase(0); // erase at front

c.dump(); // are better than dogs

c.erase(3); // erase at back

c.dump(); // are better than

c.erase(1); // erase middle

c.dump(); // are than

c.erase(0);

c.dump(); // than

c.erase(0); // erase single item

c.dump(); // [empty]

assert(c.erase(0) == 0); // empty list

// remove

cerr << endl << "REMOVE" << endl;

b.insert(2, "woof"); // removing with a word in between

b.dump(); // meow meow woof meow

assert(b.remove("meow") == 3);

b.dump(); // woof

b.remove("woof");

b.insert("meow"); // removing a single word

assert(b.remove("meow") == 1);

b.dump(); // [empty]

assert(b.remove("meow") == 0); // removing an empty list

b.dump(); // [empty]

// swap pt.2

cerr << endl << "SWAP PT.2" << endl;

// swapping empty sequences

b.swap(c);

b.dump(); // [empty]

c.dump(); // [empty]

// swapping one non-empty sequence with an empty sequence

b.insert("meow");

b.swap(c);

b.dump(); // [empty]

c.dump(); // meow

assert(b.find("meow") == -1); // finding in an empty sequence

cerr << endl << "SET" << endl;

Sequence d;

d.insert(0, "cats");

d.insert(1, "could");

d.insert(2, "rule");

d.insert(3, "the");

d.insert(4, "world");

d.dump(); // cats could rule the world

// set

assert(d.set(5, "meow") == 0); // out of bounds

assert(d.set(-1, "meow") == 0); // out of bounds

assert(d.set(0, "bow") == 1); // within bounds

d.dump(); // bow could rule the world

assert(d.set(1, "before") == 1);

d.dump(); // bow before rule the world

assert(d.set(2, "the") == 1);

d.dump(); // bow before the the world

assert(d.set(3, "cats") == 1);

d.dump(); // bow before the cats world

assert(d.set(4, "now") == 1);

d.dump(); // bow before the cats now

d.insert(2, "bow");

d.insert(3, "before");

d.dump(); // bow before bow before the cats now

// subsequence

cerr << endl << "SUBSEQUENCE" << endl;

Sequence e;

e.insert(0, "bow");

e.insert(1, "before");

e.dump(); // bow before

assert(subsequence(d, e) == 0);

e.insert(2, "the");

assert(subsequence(d, e) == 2);

e.insert(3, "felines");

assert(subsequence(d, e) == -1);

Sequence f; // empty sequence

assert(subsequence(d, f) == -1); // empty subsequence

assert(subsequence(e, d) == -1); // longer subsequence

// concatReverse

cerr << endl << "CONCATREVERSE" << endl;

Sequence h; // empty sequence

Sequence g; // result sequence 1

Sequence i; // result sequence 2

e.dump(); // bow before the felines

concatReverse(f, e, g); // seq1 is empty

g.dump(); // felines the before bow

concatReverse(e, f, i); // seq2 is empty

i.dump(); // felines the before bow

concatReverse(f, h, g); // both sequences are empty, result is empty

g.dump(); // [empty]

concatReverse(i, h, g); // i = "felines before the bow", h = empty, g = "felines the before bow"

g.dump(); // bow before the felines

concatReverse(f, h, g); // both sequences are empty, result is not empty

g.dump(); // [empty]

concatReverse(e, e, e); // same sequence for all three

e.dump(); // felines the before bow felines the before bow

concatReverse(e, i, e); // seq1 and result are the same sequence

e.dump(); // bow before the felines bow before the felines bow before the felines

cout << endl << "ALL TESTS PASSED" << endl;