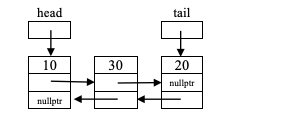
Madeleine Murphy

004991129

CS 32 Spring 2019

Project 2

My Sequence is a doubly linked list. It has one head pointer that points to the first item in the list, and one tail pointer that points to the last item in the list. Each node in the list has an m\_next and an m\_prev. If you are at the first item in the list, you cannot go to a previous, so m\_prev is nullptr. If you are the last item of the list, you cannot go to a next item, so m\_next is nullptr.

 Picture from PDF.

My design is based on this doubly linked list system.

Pseudocode

Sequence::Sequence()

Create an empty linked list

int Sequence::size()

bool Sequence::empty()

Sequence::~Sequence()

Create a pointer to a node p

Increment pointer down linked list until reach end

Delete each node

Sequence::Sequence(const Sequence& other)

If other is empty, make sequence empty

Else

Make a new itemtype p

Repeatedly

Put other data into p

Add p to rear of sequence with rear function

Make sequence size = other size

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

Check that rhs is not an alias for sequence

Call the copy constructor to copy rhs into temp sequence

Swap temp sequence with sequence

Return pointer to sequence

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

if pos out of bounds, return -1

if pos is 0, add value to front function

if pos is = size, add value to back

otherwise

add item at given position

return position

int Sequence::insert(const ItemType& value)

if empty list, add to front

if value is less than first item

add to front function

return position 0

otherwise

make a pointer to a node p

repeatedly

increment counter until find position one above desired

==position where value <= item at next position

if we find desired position

add item & return position

otherwise return -1

bool Sequence::erase(int pos)

if pos out of bounds, return false

if empty list, return false

return bool helper function deleteItem if deleteItem deletes

int Sequence::remove(const ItemType& value)

if empty list return 0

make a pointer to a node p

repeatedly

increment pointer through list until find desired position

item in sequence == value

call erase function on item at that position

move position down one since we delete

increment p

increment pos

return number of items removed

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

if position out of bounds return false

if empty list return false

create a pointer to a node p

repeatedly

increment through list by pointer until locating position

set value to data at position

return true

return false otherwise

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

if position is out of bounds return false

if list is empty return false

create a pointer to a node p

repeatedly

increment through list by pointer until locating position

if found, break

set data at position to value

return true

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

create a pointer to a node p

repeatedly

increment through list by pointer until locating position where its data == value

return that position

return -1

void Sequence::swap(Sequence& other)

make a temporary pointer to the sequence head and tail

swap the head with other head

swap the tail with other tail

make temporary size

swap the sizes

void Sequence::addItem(int pos, const ItemType& value)

if empty sequence

add to front function

if position is 0

add to front function

if position is last

add to rear function

otherwise

make a pointer p to first node

loop through p

find node above desired position

if it is the last element in the list

add to rear function

otherwise

make a new node

put value in node

new node gets the node before it’s next

new node gets the node after it’s prev

update next pointer from node before

update prev pointer from node after

increment the size

void Sequence::addToFront(const ItemType& value)

make a pointer to a new node p

put value in p

if empty list

set next to what head points (will be nullptr)

set previous to null pointer

head points to this node

tail points to this node

otherwise

set next to what head points to

set previous to null pointer

next item’s previous becomes this node

head points to this node

increment the size

void Sequence::addToRear(const ItemType& value)

if empty list

addToFront

otherwise

make a new pointer to a node n

put value in n

set next to nullptr

set previous to what tail pointed to

set tail’s next to n

tail points to this node

increment the size

bool Sequence::deleteItem(int pos)

if we are deleting the first node

make a new node killMe that points to head’s object

set head to next item

first item now has prev = null

delete killMe

decrement the size

return true

otherwise

make a pointer to a node p that starts at the first item

repeatedly

search for position one above desired

if found, break

increment counters on position and list

if it is not the last element

make a pointer to node kill me that points to the next object after p

object before desired node points to object after desired

object after desired node points to object before desired

delete desired node

decrement size

return true

if it is the last element in the list

make a pointer to node kill me that points to the next object after p

object before desired node next gets nullptr, last elem

tail now points to p

delete desired node

decrement size

return true

return false

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

if both or one sequences are empty

return -1

if seq1 is smaller than seq2

return -1

if they are equal in size

create two itemtypes

repeatedly

use get function to put into itemtypes

check if itemptypes at each position match

if any element does not match

return -1

otherwise return 0, they are the same starting at pos 0

otherwise

repeatedly

create two itemtypes

if an item in seq1 corresponds to first item in seq2

repeatedly

loop to end of seq2

find if they match consecutive items in seq 1

if they don’t match

break to outer loop again

if they match

return initial position

return -1

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

make an empty temp sequence

if seq1 is empty

repeatedly

get value at position in seq2

insert it into temp

if seq2 is empty

repeatedly

get value at position in seq1

insert it into temp

if neither are empty

if they’re equal in size

get, add seq1 item to temp

increment temp place

get, add seq2 item into temp

increment temp place

if seq1 less than seq2

repeatedly until reach end of seq 1

get, add seq1 item to temp

increment temp place

get, add seq2 item into temp

increment temp place

repeatedly until reach end seq2

get, add rest items of seq2

if seq2 less than seq1

repeatedly until reach end of seq 2

get, add seq1 item to temp

increment temp place

get, add seq2 item into temp

increment temp place

repeatedly until reach end seq1

get, add rest items of seq2

swap temp with result

**Test Cases**

#include "Sequence.h"

#include <cassert>

using namespace std;

int main(){

    Sequence ss;  // ItemType is std::string

    ss.insert(0, "aaa");

    ss.insert(1, "bbb");

    ss.insert(2, "ccc");

    ItemType x = "xxx";

    assert(!ss.get(3, x)  &&  x == "xxx");  // x is unchanged

    assert(ss.get(1, x)  &&  x == "bbb");

    Sequence s;

    assert(s.insert(0, "lavash") == 0);

    assert(s.insert(0, "tortilla") == 0);

    assert(s.size() == 2);

    ItemType p = "injera";

    assert(s.get(0, p)  &&  p == "tortilla");

    assert(s.get(1, p)  &&  p == "lavash");

    Sequence q;

    q.insert(0, "hi");

    q.insert(1, "bye");

    Sequence m = q;         //testing copy constructor

    assert(q.remove("hi") == 1);

}

int main(){

  //for ItemType = unsigned long

    Sequence p;                 //check insert, set up for other checks

    p.insert(0, 1);

    p.insert(1, 2);

    p.insert(2, 4);

    p.insert(3, 5);

    p.insert(4, 6);

    p.insert(0, 7);

    p.insert(3, 8);

    Sequence q;

    q.insert(0, 8);

    q.insert(1, 4);

    q.insert(2, 5);

    Sequence r;

    r.insert(0, 7);

    r.insert(1, 1);

    r.insert(2, 2);

    r.insert(3, 4);

    r.insert(4, 5);

    Sequence m;

    m.insert(0, 7);

    m.insert(1, 1);

    m.insert(2, 2);

    m.insert(3, 8);

    m.insert(4, 4);

    Sequence cool;

    cool.insert(0, 1);

    cool.insert(1, 2);

    cool.insert(2, 4);

    cool.insert(3, 5);

    cool.insert(4, 6);

    Sequence my;

    my.insert(0, 1);

    my.insert(1, 2);

    my.insert(2, 4);

    my.insert(3, 5);

    my.insert(4, 6);

    assert(subsequence(p, q) == 3);             //check subsequence

    assert(subsequence(p, r) == -1);

    assert(subsequence(p, m) == 0);

    assert(m.insert(8) == 3);           //check insert with 1

    assert(cool.insert(3) == 2);

    assert(cool.insert(0) == 0);

    assert(m.insert(9) == 6);

    assert(my.insert(6) == 4 && my.size() == 6);

    assert(r.erase(0) && r.size() == 4);            //check erase

    ItemType boo;

    assert(r.get(0, boo) && boo == 1);

    assert(r.erase(3) && r.size() == 3);

    assert(r.get(2, boo) && boo == 4);

    assert(!r.get(3, boo));

    assert(r.erase(1) && r.size() == 2);

    assert(r.get(0, boo) && boo == 1);

    assert(r.get(1, boo) && boo == 4);

    Sequence seq1;

    seq1.insert(0, 30);

    seq1.insert(1, 21);

    seq1.insert(2, 63);

    seq1.insert(3, 42);

    seq1.insert(4, 17);

    seq1.insert(5, 63);

    Sequence seq2;

    seq2.insert(0, 42);

    seq2.insert(1, 63);

    seq2.insert(2, 84);

    seq2.insert(3, 19);

    Sequence result;            //check interleave, with diff sizes, first bigger

    interleave(seq1, seq2, result);

    ItemType k;

    assert(result.get(0, k)  &&  k == 30);

    assert(result.get(1, k)  &&  k == 42);

    assert(result.get(2, k)  &&  k == 21);

    assert(result.get(3, k)  &&  k == 63);

    assert(result.get(4, k)  &&  k == 63);

    assert(result.get(5, k)  &&  k == 84);

    assert(result.get(6, k)  &&  k == 42);

    assert(result.get(7, k)  &&  k == 19);

    assert(result.get(8, k)  &&  k == 17);

    assert(result.get(9, k)  &&  k == 63);

    Sequence a;

    a.insert(0, 1);

    a.insert(1, 2);

    Sequence b;

    b.insert(0, 3);

    b.insert(1, 4);

    b.insert(2, 5);

    b.insert(3, 6);

    Sequence result2;

    interleave(a, b, result2);      //check interleave  with diff sizes, second bigger

    ItemType blah;

    assert(result2.get(0, blah)  &&  blah == 1);

    assert(result2.get(1, blah)  &&  blah == 3);

    assert(result2.get(2, blah)  &&  blah == 2);

    assert(result2.get(3, blah)  &&  blah == 4);

    assert(result2.get(4, blah)  &&  blah == 5);

    assert(result2.get(5, blah)  &&  blah == 6);

    Sequence c;

    c.insert(0, 1);

    c.insert(1, 2);

    Sequence d;

    d.insert(0, 3);

    d.insert(1, 4);

    Sequence result3;           //check interleave

    interleave(c, d, result3);          //with same size

    ItemType woo;

    assert(result3.get(0, woo)  &&  woo == 1);

    assert(result3.get(1, woo)  &&  woo == 3);

    assert(result3.get(2, woo)  &&  woo == 2);

    assert(result3.get(3, woo)  &&  woo == 4);

    Sequence e;

    //empty

    Sequence f;

    f.insert(0, 1);

    f.insert(1, 2);

    Sequence result4;       //check interleave

    interleave(e, f, result4);          //with first empty

    ItemType ya;

    assert(result4.get(0, ya)  &&  ya == 1);

    assert(result4.get(1, ya)  &&  ya == 2);

    Sequence result5;

    interleave(f, e, result5);          //with second empty

    ItemType hi;

    assert(result5.get(0, hi)  &&  hi == 1);

    assert(result5.get(1, hi)  &&  hi == 2);

    Sequence h;

    Sequence resultempty;           //with two empty

    interleave(e, h, resultempty);

    ItemType empty;

    assert(!resultempty.get(0, empty));

    assert(resultempty.empty());

    Sequence x;

    x.insert(0, 0);

    x.insert(1, 1);

    Sequence x2 = x;         //testing copy constructor

    ItemType ab;

    assert(x2.get(0, ab) && ab == 0);

    assert(x2.get(1, ab) && ab == 1);

    Sequence o;             //testing assignment operator

    o = x2;

    ItemType ba;

    assert(o.get(0, ba) && ba == 0);

    assert(o.get(1, ba) && ba == 1);

    x = x2;

    ItemType cd;

    assert(x.get(0, cd) && cd == 0);

    assert(x.get(1, cd) && cd == 1);

    Sequence yz;

    yz.insert(0, 0);

    yz.insert(1, 1);

    yz.insert(2, 2);

    yz.insert(3, 3);

    yz.insert(4, 4);

    yz.insert(5, 5);

    ItemType z;

    assert(yz.remove(1) == 1 && yz.size() == 5);        //check remove

    assert(yz.get(1, z) && z == 2);

    assert(yz.remove(5) == 1 && yz.size() ==4);     //remove last element

    assert(yz.get(3, z) && z == 4);

    assert(yz.remove(0) == 1 && yz.size() == 3);    //remove first element

    assert(yz.get(0, z) && z == 2);

    Sequence op;

    op.insert(0, 0);

    op.insert(1, 1);

    op.insert(2, 2);

    op.insert(3, 3);

    op.insert(4, 4);

    op.insert(5, 5);

    assert(op.find(0) == 0);        //check find

    assert(op.find(1) == 1);

    assert(op.find(2) == 2);

    assert(op.find(3) == 3);

    assert(op.find(4) == 4);

    assert(op.find(5) == 5);

    assert(op.find(100) == -1);

    Sequence pi;

    pi.insert(0, 10);

    pi.insert(1, 11);

    pi.insert(2, 12);

    pi.insert(3, 13);

    pi.insert(4, 14);

    pi.insert(5, 15);

    op.swap(pi);            //check swap, same size

    assert(pi.get(0, z) && z == 0);

    assert(pi.get(1, z) && z == 1);

    assert(pi.get(2, z) && z == 2);

    assert(pi.get(3, z) && z == 3);

    assert(pi.get(4, z) && z == 4);

    assert(pi.get(5, z) && z == 5);

    assert(op.get(0, z) && z == 10);

    assert(op.get(1, z) && z == 11);

    assert(op.get(2, z) && z == 12);

    assert(op.get(3, z) && z == 13);

    assert(op.get(4, z) && z == 14);

    assert(op.get(5, z) && z == 15);

    Sequence ii;

    ii.insert(0, 20);

    ii.insert(1, 21);

    ii.insert(2, 22);

    ii.insert(3, 23);

    op.swap(ii);            //check swap, diff sizes

    assert(ii.get(0, z) && z == 10);

    assert(ii.get(1, z) && z == 11);

    assert(ii.get(2, z) && z == 12);

    assert(ii.get(3, z) && z == 13);

    assert(ii.get(4, z) && z == 14);

    assert(ii.get(5, z) && z == 15);

    assert(op.get(0, z) && z == 20);

    assert(op.get(1, z) && z == 21);

    assert(op.get(2, z) && z == 22);

    assert(op.get(3, z) && z == 23);

    op.swap(op);            //check alias

    cout << "Passed all tests" << endl;

}