Danning Yu, 305087992

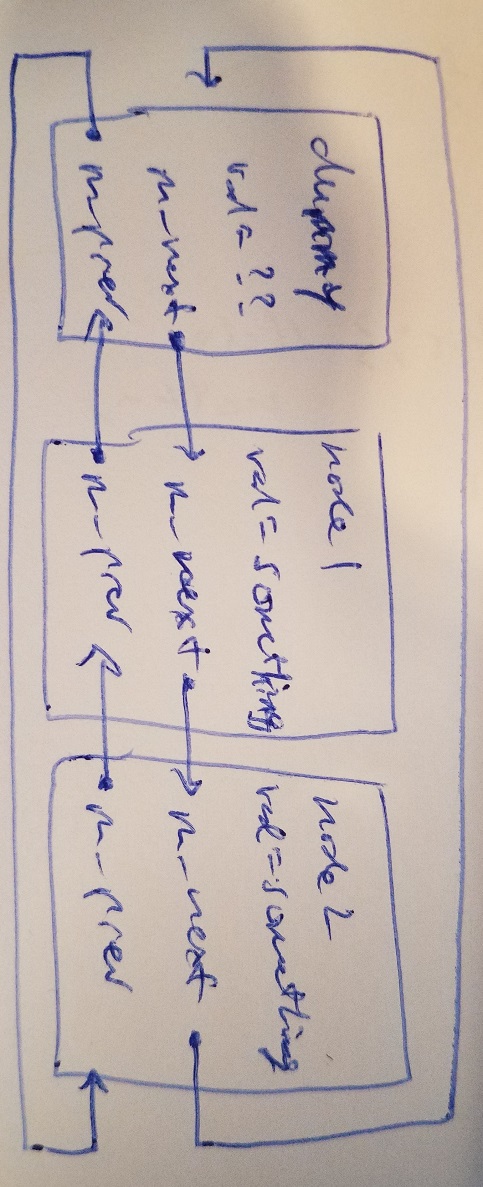
CS32 Discussion 1E

Project 2 Report

4/23/19

Implementation:

My doubly linked list implementation consisted of a circularly doubly linked list with a dummy node. There is no header node, as the dummy node serves as the header node. In each list node, there’s a data member for its value (of type ItemType), as well as pointers to the previous and next node. There are no pointers that have value nullptr, as the linked list is circular. This is shown in the picture below.



Psuedocode:

Sequence::~Sequence():

for as long as items can be erased from the sequence:

erase the first element of the sequence using the erase function

delete the dummy node

Sequence::Sequence(const Sequence& other):

set the size of the sequence to zero

create a new dummy node and direct its pointers to itself (circularly linked)

loop through the other list (that you’re copying from):

get the value of each node in the other list

insert that value into the current list using the insert function

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

if you aren’t assigning the sequence to itself (no aliasing):

delete all the nodes of the lvalue

create a temporary sequence that is a copy of the rvalue (using copy constructor)

swap the current sequence (lvalue) with the temporary sequence)

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

if the insertion position is valid (not negative, not larger than size of sequence):

make a new node to accommodate the value being inserted

loop through sequence until a temporary pointer points to the node at insertion positon

set the new node’s prev and next pointers to point to the right nodes

set the previous and next nodes’ pointers to correctly point to the new item

update the size of the sequence

return the position where it was inserted, or that no insertion occurred

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

if the insertion position is valid (not negative, not larger than size of sequence):

find the first node in the sequence where insertion value is smaller than node’s value

use the two parameter insert function to insert the value at that position

return the position where it was inserted, or that no insertion occurred

bool Sequence::erase(int pos);

if the position is valid (not negative, not larger than size of sequence):

loop through sequence until a temporary pointer points to pos where erasing will occur

redirect pointers of the previous and next nodes as if the node was already erased

delete the node

update the size of the sequence

return if a node was erased or not

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

loop through the sequence:

if the value is found, erase the node at that position using the erase function

increment the counter of how many nodes have been erased

return how many times a node was erased

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

if the position is valid (not negative, not larger than size of sequence):

loop through sequence until a temporary pointer points to desired pos

set value equal to the value held by that node

return if value was successfully set or not

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

if the position is valid (not negative, not larger than size of sequence):

loop through sequence until a temporary pointer points to desired pos

set value of the node equal to the value paramter

return if value was successfully set or not

void Sequence::swap(Sequence& other);

swap the sizes of the sequences

swap the pointers of the dummy nodes of the sequences

swap the previous pointer of the first item of the sequences

swap the next pointer of the last item of the sequences

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

if seq2 is not empty and seq2 is smaller than seq1:

check the first value of seq2 with each value of seq1, if equality is found:

check the next value of seq2 with the next value of seq1:

if inequality is found, stop checking, go back to checking the first value

if no inequality is found and all the values of seq2 match seq1 in order,

return where the checking began

return either where the checking began or that no subsequence was found

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

create a temporary sequence (temp) to handle aliasing

if both seq1 and seq2 are empty, set temp to an empty sequence

if seq1 or seq2 is empty, set temp to the nonempty sequence

do the following (interleave) until either sequence has had all its nodes copied over to result:

insert an item from seq1 into temp

insert an item from seq2 into temp

repeat the above 2 statements; note that they alternate to produce a interwoven sequence

for whichever sequence had leftover elements that weren’t copied: (the “longer” sequence)

copy those leftover elements into temp

assign the items of temp to result

Test Cases:

The following assertions were performed my code using unsigned longs.

Sequence a, b; //make 2 sequences

assert(a.empty() == true); //sequences should be empty upon creation

assert(b.empty() == true);

assert(a.size() == 0); //sequences should have 0 size upon creation

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

a.insert(0, 0); //insert an element into a, test the two parameter insert function

//0 – this is listing what’s currently in sequence a (or any other relevant sequence)

assert(a.size() == 1); //a should now have a nonzero size

assert(a.empty() == false); //a should no longer be empty

a.insert(0, 10); //insert more items into the start

//10, 0

ItemType temp; //temporary holder for an ItemType variable

assert(a.get(0, temp) && temp == 10); //1st element should be 10, and test get function

assert(a.get(1, temp) && temp == 0); //2nd element should be 0

assert(a.get(2, temp) == false);

assert(a.insert(5) == 0); //test the one parameter insert function

//5, 10, 0

assert(a.get(0, temp) && temp == 5); //5<10, so 5 should be first element

assert(a.get(1, temp) && temp == 10); //2nd element should be 10

assert(a.get(2, temp) && temp == 0); //last element should be 0

assert(a.get(3, temp) == false); //out of bounds

a.insert(1, 15); //test two parameter insert function

//5, 15, 10, 0

assert(a.get(0, temp) && temp == 5);

assert(a.get(1, temp) && temp == 15);

assert(a.get(2, temp) && temp == 10);

assert(a.get(3, temp) && temp == 0); //everything should get shifted

assert(a.size() == 4); //m\_size should be updated

assert(a.empty() == false); //a shouldn't be empty

assert(a.insert(4, 20) == 4); //test insert for when inserting at size()

//5, 15, 10, 0, 20

assert(a.insert(6, 20) == -1); //test insert for inserting at pos > size()

assert(a.insert(-1, 10) == -1); //test insert for inserting at pos <0

assert(a.erase(1) == true); //erase element 1, test it worked

//5, 10, 0, 20

assert(a.get(0, temp) && temp == 5);

assert(a.get(1, temp) && temp == 10);

assert(a.get(2, temp) && temp == 0);

assert(a.get(3, temp) && temp == 20); //everything should get shifted

assert(a.insert(2, 0)); //insert another 0 into the sequence

//5, 10, 0, 0, 20

assert(a.erase(5) == false); //can’t erase when pos >= size()

assert(a.erase(-1) == false); //can’t erase when pos < 0

assert(a.insert(5, 15) == 5); //add another element

//5, 10, 0, 0, 20, 15

assert(a.insert(5, 0) == 5); //add another element

//5, 10, 0, 0, 20, 0, 15

assert(a.size() == 7); //test size was updated

assert(a.remove(0) == 3); //test the remove function, should erase the three 0's

//5, 10, 20, 15

assert(a.get(0, temp) && temp == 5); //element in front of 0's shouldn't be affected

assert(a.get(1, temp) && temp == 10);

assert(a.get(2, temp) && temp == 20);

assert(a.get(3, temp) && temp == 15);

assert(a.size() == 4); //size() should have been updated too

assert(a.set(1, 25) == true); //test the set function

//5, 25, 20, 15

assert(a.set(-1, 25) == false); //cannot set value when pos <0

assert(a.set(4, 25) == false); //cannot access out of bounds value

assert(a.get(1, temp) && temp == 25); //test if the set function worked

assert(a.set(3, 25));

//5, 25, 20, 25

assert(a.find(25) == 1); //get function should return the first occurrence

assert(a.find(5) == 0); //test the get function

assert(a.find(20) == 2);

assert(a.remove(25) == 2); //should remove the two occurrences of 25

//5, 20

assert(a.find(30) == -1); //should return -1 for a value not in the sequence

assert(a.size() == 2); //make sure a's size is correct

assert(a.insert(1, 25));//insert the removed items back in

assert(a.insert(3, 25));

assert(b.insert(7)==0); //insert some items into sequence b

assert(b.insert(9)==1);

assert(b.insert(11)==2);

//7, 9, 11

assert(b.find(7) == 0); //the numbers should be in order

assert(b.find(9) == 1);

assert(b.find(11) == 2);

assert(b.empty() == false); //b should no longer be empty

assert(b.size() == 3); //b's size should be updated to 3

a.swap(b);

//a: 7, 9, 11 b: 5, 25, 20, 25

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

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

assert(a.get(0, temp) && temp == 7); //the elements of a and b should be switched

assert(a.get(1, temp) && temp == 9);

assert(a.get(2, temp) && temp == 11);

assert(b.get(0, temp) && temp == 5); //the elements of a and b should be switched

assert(b.get(1, temp) && temp == 25);

assert(b.get(2, temp) && temp == 20);

assert(b.get(3, temp) && temp == 25);

b.swap(b); //test swapping with itself

assert(b.get(0, temp) && temp == 5); //nothing should happen to b

assert(b.get(1, temp) && temp == 25);

assert(b.get(2, temp) && temp == 20);

assert(b.get(3, temp) && temp == 25);

Sequence c = a; //test the copy constructor

//a: 7, 9, 11 c: 7, 9, 11

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

assert(c.get(0, temp) && temp == 7); //c should have the elements of a

assert(c.get(1, temp) && temp == 9);

assert(c.get(2, temp) && temp == 11);

assert(c.size() == 3); Sequence d;

assert(d.size() == 0); //d should be empty

d = c; //test the assignment operator

Sequence d1 = d;

d1 = c; //test the assignment operator for assigning to preinitialized value

d1 = d1; //test the assignment operator

assert(d.size() == 3); //test if assignment operator worked

assert(d.get(0, temp) && temp == 7);

assert(d.get(1, temp) && temp == 9);

assert(d.get(2, temp) && temp == 11);

d = d; //if assigning the same thing, nothing should happen

assert(d.size() == 3); //d shouldn't be affected by getting assigned the same value

assert(d.get(0, temp) && temp == 7); //d shouldn't be affected

assert(d.get(1, temp) && temp == 9);

assert(d.get(2, temp) && temp == 11);

Sequence e, f, g; //test the interleave function

e.insert(0, 63); //insert some values into e

e.insert(0, 17);

e.insert(0, 42);

e.insert(0, 63);

e.insert(0, 21);

e.insert(0, 30);

f.insert(0, 19); //insert some values into f

f.insert(0, 84);

f.insert(0, 63);

f.insert(0, 42);

interleave(e, f, g);

assert(g.get(0, temp) && temp == 30);

assert(g.get(1, temp) && temp == 42);

assert(g.get(2, temp) && temp == 21);

assert(g.get(3, temp) && temp == 63);

assert(g.get(4, temp) && temp == 63);

assert(g.get(5, temp) && temp == 84);

assert(g.get(6, temp) && temp == 42);

assert(g.get(7, temp) && temp == 19);

assert(g.get(8, temp) && temp == 17);

assert(g.get(9, temp) && temp == 63);

interleave(e, e1, g); //test interleaving two identical sequences

assert(g.get(0, temp) && temp == 30);

assert(g.get(1, temp) && temp == 30);

assert(g.get(2, temp) && temp == 21);

assert(g.get(3, temp) && temp == 21);

assert(g.get(4, temp) && temp == 63);

assert(g.get(5, temp) && temp == 63);

assert(g.get(6, temp) && temp == 42);

assert(g.get(7, temp) && temp == 42);

assert(g.get(8, temp) && temp == 17);

assert(g.get(9, temp) && temp == 17);

assert(g.get(10, temp) && temp == 63);

assert(g.get(11, temp) && temp == 63);

interleave(f, e, g); //switch the order around for interleaving

assert(g.get(0, temp) && temp == 42);

assert(g.get(1, temp) && temp == 30);

assert(g.get(2, temp) && temp == 63);

assert(g.get(3, temp) && temp == 21);

assert(g.get(4, temp) && temp == 84);

assert(g.get(5, temp) && temp == 63);

assert(g.get(6, temp) && temp == 19);

assert(g.get(7, temp) && temp == 42);

assert(g.get(8, temp) && temp == 17);

assert(g.get(9, temp) && temp == 63);

Sequence h, i; //test subsequence

h.insert(0, 32); //insert some items into h

h.insert(0, 8);

h.insert(0, 29);

h.insert(0, 17);

h.insert(0, 63);

h.insert(0, 17);

h.insert(0, 42);

h.insert(0, 63);

h.insert(0, 21);

h.insert(0, 30);

i.insert(0, 29); //insert some items into i

i.insert(0, 17);

i.insert(0, 63);

assert(subsequence(h, i) == 5); //first occurrence occurs at the 5th index

assert(subsequence(d, h) == -1); //no subsequence so it should return 0

assert(subsequence(d, i) == -1); //no subsequence so it should return 0

Sequence j, k;

assert(subsequence(j, k) == -1); //test subsequence for empty sequences

assert(subsequence(k, j) == -1);

assert(subsequence(j, h) == -1);

Sequence l;

interleave(j, k, l); //test interleave for empty sequences

assert(l.empty() == true); //l should still be empty

interleave(j, j, j); //interleave should work if all 3 parameters are the same

assert(j.empty() == true); //j should still be empty

interleave(j, l, k);

assert(k.empty() == true);

interleave(k, l, j);

assert(j.empty() == true);

j.insert(10);

k.insert(10);

assert(subsequence(j, k) == 0); //test subsequence for two 1 element functions

l.insert(15);

assert(subsequence(j, l) == -1); //test subsequence for two 1 element functions

assert(subsequence(k, l) == -1);