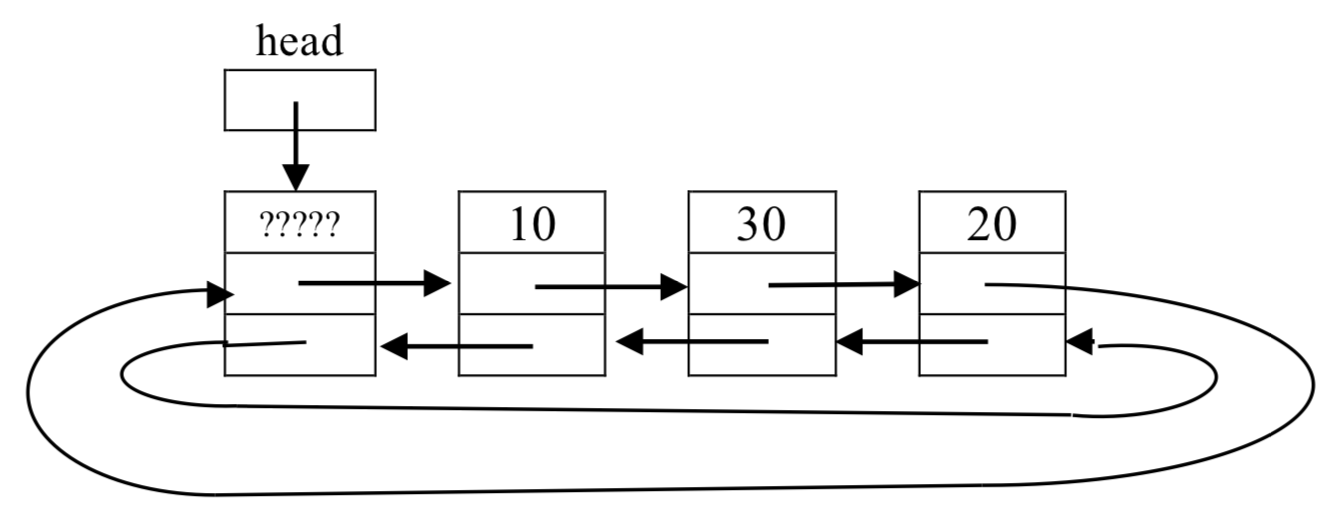
Joseph Picchi

Project 2 Report

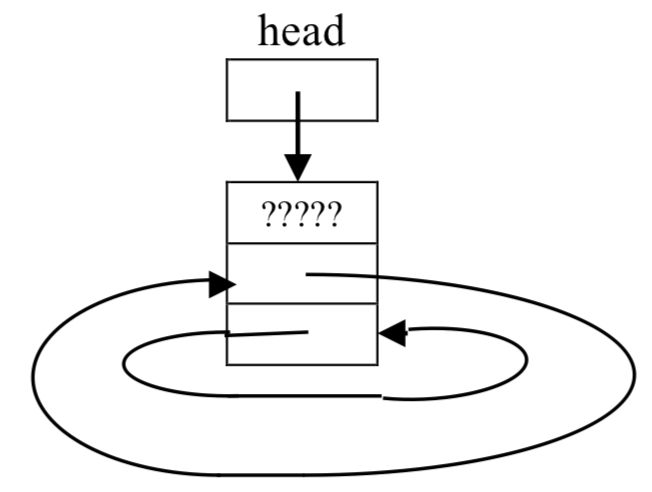
**DESIGN OF THE DOUBLY-LINKED LIST:**

The design of the doubly-linked list that I implemented is a circular, doubly-linked list with a dummy node and a head pointer that always points the the dummy node. The nodes of the list each contain a data member of type ItemType, a next pointer, and a previous pointer.

A typical sequence containing multiple nodes is illustrated as follows, with each of the member items containing an integer data member and the dummy node containing “?????” to indicate that the data member of its dummy node is uninitialized:



An empty sequence (consisting of just the dummy node), implemented according to my linked list design, is illustrated below:



(NOTE: these illustrations are credited to the author of “Supplement to Linked List Lecture” on the CS32 course website. I reproduced them in this report because my implementation of the doubly-linked list was identical to that described in this particular section of the supplemental article.)

**PSEUDOCODE FOR THE ALGORITHMS:**

Sequence::Sequence()

set sequence size to 0

create a new node

point the new node’s pointers to itself

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

allocate a new node

link the new node to the previous and following nodes

link the following node to the new node

link the previous node to the new node

increment sequence size by 1

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

if position is valid

add an item at the position (using uncheckedInsert)

return position

if position is invalid, return -1

Sequence::insert(const ItemType& value)

loop through the items in the sequence:

if item at that position <= value argument

record the index

break

if an index was found

insert an item at that index (using uncheckedInsert)

otherwise

insert an item at the end (using uncheckedInsert)

return the the position where the item was inserted

Sequence::erase(int pos)

if the position is valid

link the following node to the previous node

link the previous node to the following node

delete the node at position

decrement the size

return true

otherwise

return false

Sequence::remove(const ItemType& value)

loop through every item in the sequence

if an item equals the value argument

erase the item (using erase)

increment the count

return the count

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

if the position is valid

set value equal to the item at position

return true

otherwise

return false

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

if the position is invalid

return false

set the item at position equal to value

return true

Sequence::find(const ItemType& value) const

loop through the sequence

if an item matches the value argument

record the index

break

if a match was found

return the index

otherwise

return -1

Sequence::swap(sequence& other)

exchange the positions where the two objects’ heads point

exchange the objects’ sizes

Sequence::~Sequence()

loop through the sequence

delete the item

delete the dummy node

Sequence::createAndFillNodes(const Sequence& other)

for a number of times equivalent to other’s size, repeatedly:

create a new node

link the new node to the previous and following node

link the previous node to the new node

link following node to the new node

loop through the calling object’s sequence

set the item equal to the item at the same position in other

Sequence::Sequence(const Sequence& other)

set the calling object’s size to other’s size

create a dummy node for the calling object

create the necessary number of nodes and fill them (using createAndFillNodes)

Sequence::operator = (const Sequence& other)

in the case of aliasing

return a reference

create a copy of other

swap the sequences of the copy and calling object (using swap)

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

if either sequence is empty

return -1

loop through seq1

if an item of seq1 matches the first element of seq2

record the index

loop through seq2

if seq1 contains all elements of seq2

return the recorded index

otherwise

break

if seq1 doesn’t contain seq2

return -1

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

make copies of seq1 and seq2

remove all elements from result

loop as many times as the size of the larger array

if seq1’s copy is not empty

insert the next element of seq1 at the end of result

if seq2’s copy is not empty

insert next element of seq2 at the end of result

**TEST CASES:**

Sequence s1; // create a new sequence

assert(s1.empty()); // new sequences should be empty

assert(s1.insert(1, "hi") == -1); // invalid position

// should return -1

assert(s1.empty()); // invalid insert should not affect

// the sequence

assert(s1.insert(0, "hi") == 0); // valid insert should

// return pos

assert(s1.insert(1, "yep") == 1); // assertions allowed at

// the end of an existing

// sequence

assert(s1.insert(0, "yee") == 0); // insertions allowed at

// the beginning of an existing sequence

assert(s1.insert(1, "works") == 1); // insertions allowed in

// the middle of an

// existing sequence

assert(!s1.empty()); // a sequence with items is not empty

assert(s1.size() == 4); // s1 should have 4 items

Sequence s2;

assert(s2.insert("apple") == 0); // "insert" returns the

// index where "value"

// was inserted

assert(s2.insert("pear") == 1); // value is not less than

// the items in the list,

// so it should be

// inserted at the end

assert(s2.insert("banana") == 1); // value is less than the

// item at index 1, so it

// should be placed there

assert(!s2.erase(3)); // invalid position should return

// false

assert(s2.erase(1)); // valid position should erase the

// item at pos and return true

assert(s2.size() == 2); // size should change when an item is

// removed

assert(s2.erase(0)); // allowed to erase items at the

// beginning of the sequence

assert(s2.erase(0)); // allowed to erase items from a

// single-item sequence

assert(s2.empty());

assert(!s2.erase(0)); // not allowed to erase from an

// empty sequence

assert(s2.find("cat") == -1); // can never find a value

// in an empty sequence

assert(s2.insert("cat") == 0);

assert(s2.find("cat") == 0); // find works for

// one-item sequences

assert(s2.insert("dog") == 1);

assert(s2.insert(2, "cat") == 2);

assert(s2.find("cat") == 0); // find should return the

// index of the first

// found item

assert(s2.insert(3, "zoo") == 3);

assert(s2.find("zoo") == 3); // find works for the

// last item in the

// sequence

assert(s2.remove("yack") == 0); // no matches should

// return 0

assert(s2.remove("dog") == 1); // "remove" works for

// items in the middle of

// a list

assert(s2.remove("cat") == 2); // works when removing

// two consecutive items

assert(s2.remove("zoo") == 1); // works for single-item

// sequences

assert(s2.empty()); // size is altered when items are

// removed

ItemType value = "initial";

assert(!s2.get(0, value) && value == "initial"); // returns // false for

// empty

// sequences

// and value

// is

// unchanged

assert(s2.insert("zoo") == 0);

assert(s2.insert("yack") == 0);

assert(s2.insert("mouse") == 0);

assert(!s2.get(3, value) && value == "initial"); // returns false

// for a

// position out

// of range and

// value

// remains

// unchanged

assert(s2.get(0, value) && value == "mouse"); // returns true

// for valid

// positions and

// works

// for items at

// the beginning

// of the

//sequence

assert(s2.get(1, value) && value == "yack"); // works for

//items in the

//middle of the

//sequence

assert(s2.get(2, value) && value == "zoo"); // works for

//items at the

//end of the

//sequence

Sequence s3;

ItemType placeholder = "placeholder";

assert(!s3.set(0, "yee")); // always returns false

//for an empty sequence

assert(s3.insert("zelda") == 0);

assert(s3.insert("legend") == 0);

assert(s3.insert("beta") == 0);

assert(!s3.set(4, placeholder)); // returns false for

//invalid positions

assert(s3.set(0, placeholder) && s3.get(0, value) && value == placeholder); // "set" works for the first item in a sequence

assert(s3.set(1, placeholder) && s3.get(1, value) && value == placeholder); // "set" works for items in the middle of a

// sequence

assert(s3.set(2, placeholder) && s3.get(2, value) && value == placeholder); // "set" works for the last item in a sequence

Sequence s4;

Sequence s5;

assert(s4.insert("zelda") == 0);

assert(s4.insert("legend") == 0);

assert(s4.insert("beta") == 0);

assert(s5.insert("banana") == 0);

assert(s5.insert("apple") == 0);

s5.swap(s4);

assert(s4.size() == 2 && s5.size() == 3); // sizes should

// be swapped

assert(s4.get(0, value) && value == "apple" && s4.get(1, value) && value == "banana"); // s4 should have the contents of s5

// before the swap

assert(s5.get(0, value) && value == "beta" && s5.get(1, value) && value == "legend" && s5.get(2, value) && value == "zelda"); // s5 should have the contents of s4 before the swap

Sequence s6, s7, s24;

assert(s6.insert("banana") == 0);

assert(s6.insert("apple") == 0);

assert(s7.insert("zoology") == 0);

// test the copy constructor:

ItemType res;

Sequence s22(s6);

assert(s22.get(0, res) && res == "apple" && s22.get(1, res) && res == "banana"); // copy constructor can copy sequences with

// more than one item

Sequence s23(s7);

assert(s23.get(0, res) && res == "zoology"); //copy

//constructor can

//copy

//single-item

//sequences

Sequence s25(s24);

assert(!s25.get(0, res) && s25.empty()); // copy

//constructor can

//copy empty

// sequences

Sequence s26, s27;

s26 = s6;

assert(s26.get(0, res) && res == "apple" && s26.get(1, res) && res == "banana"); // assignment operator works for a calling

// object that is initially empty

s26 = s7;

assert(s26.get(0, res) && res == "zoology" && s26.size() == 1); // assignment operator works for a calling object that is not

//empty

s26 = s27;

assert(s26.empty()); // assignment operator works for an

//empty argument

s26 = s26;

assert(s26.empty()); // assignment operator works for

//aliasing when the sequence is empty

s26 = s6;

s26 = s26;

assert(s26.get(0, res) && res == "apple" && s26.get(1, res) && res == "banana"); // assignment operator works for aliasing

// when the sequence is not empty

Sequence s10, s11, s12, s13, s14;

assert(s10.insert(0, "1") == 0);

assert(s10.insert(1, "2") == 1);

assert(s10.insert(2, "3") == 2);

assert(s10.insert(3, "4") == 3);

assert(s10.insert(4, "5") == 4);

assert(s10.insert(5, "1") == 5);

assert(s10.insert(6, "3") == 6);

assert(s10.insert(7, "4") == 7);

assert(s10.insert(8, "5") == 8);

assert(s11.insert(0, "3") == 0);

assert(s11.insert(1, "4") == 1);

assert(s11.insert(2, "5") == 2);

assert(s12.insert(0, "5") == 0);

assert(s13.insert(0, "5") == 0);

assert(s13.insert(1, "6") == 1);

// Elements of s10: 1, 2, 3, 4, 5, 1, 3, 4, 5

// Elements of s11: 3, 4, 5

// Elements of s12: 5

// Elements of s13: 5, 6

// s14 is empty

int index = 10000;

index = subsequence(s10, s11);

assert(index == 2); // subsequence returns

//the first occurrence of

//seq2 in seq 1

index = subsequence(s12, s11);

assert(index == -1); // returns -1 if seq2 is

//larger than seq1

index = subsequence(s10, s12);

assert(index == 4); // works for

//singe-element

//subsequences

index = subsequence(s10, s13);

assert(index == -1); // returns -1 if seq1

//doesn't contain seq2

index = subsequence(s10, s14);

assert(index == -1); //returns -1 if seq2 is

//empty

index = subsequence(s10, s10);

assert(index == 0); // returns index 0 in

//the case of aliasing

Sequence s15, s16, s17, s18, s19, s20, s21;

assert(s15.insert(0, "1") == 0);

assert(s15.insert(1, "2") == 1);

assert(s15.insert(2, "3") == 2);

assert(s15.insert(3, "4") == 3);

assert(s16.insert(0, "5") == 0);

assert(s16.insert(1, "6") == 1);

assert(s16.insert(2, "7") == 2);

assert(s19.insert(0, "5") == 0);

assert(s19.insert(1, "6") == 1);

assert(s19.insert(2, "7") == 2);

assert(s20.insert(0, "1") == 0);

assert(s20.insert(1, "2") == 1);

assert(s21.insert(0, "1") == 0);

assert(s21.insert(1, "2") == 1);

// elements of s15: 1,2,3,4

// elements of s16: 5,6,7

// s17 is empty

// s18 is empty

// elements of s19: 5,6,7

// elements of s20: 1,2

// elements of s21: 1,2

interleave(s15, s16, s17);

ItemType result;

assert(s17.get(0, result) && result == "1" && s17.get(1, result) && result == "5" && s17.get(2, result) && result == "2" && s17.get(3, result) && result == "6" && s17.get(4, result) && result == "3" && s17.get(5, result) && result == "7" && s17.get(6, result) && result == "4");

// interweave works for non-empty seq1 and seq2 and empty result

interleave(s16, s15, s17);

assert(s17.get(0, result) && result == "5" && s17.get(1, result) && result == "1" && s17.get(2, result) && result == "6" && s17.get(3, result) && result == "2" && s17.get(4, result) && result == "7" && s17.get(5, result) && result == "3" && s17.get(6, result) && result == "4");

// interweave works for a non-empty result

interleave(s15, s16, s16);

assert(s16.get(0, result) && result == "1" && s16.get(1, result) && result == "5" && s16.get(2, result) && result == "2" && s16.get(3, result) && result == "6" && s16.get(4, result) && result == "3" && s16.get(5, result) && result == "7" && s16.get(6, result) && result == "4");

// works in the case of aliasing when seq2 and result refer to

//the same sequence

interleave(s19, s15, s19);

assert(s19.get(0, result) && result == "5" && s19.get(1, result) && result == "1" && s19.get(2, result) && result == "6" && s19.get(3, result) && result == "2" && s19.get(4, result) && result == "7" && s19.get(5, result) && result == "3" && s19.get(6, result) && result == "4");

// works in the case of aliasing when seq1 and result refer to

// the same sequence

interleave(s15, s21, s21);

assert(s21.get(0, result) && result == "1" && s21.get(1, result) && result == "1" && s21.get(2, result) && result == "2" && s21.get(3, result) && result == "2" && s21.get(4, result) && result == "3" && s21.get(5, result) && result == "4");

// works in the case of aliasing when seq2 and result refer to

// the same sequence

interleave(s20, s20, s20);

assert(s20.get(0, result) && result == "1" && s20.get(1, result) && result == "1" && s20.get(2, result) && result == "2" && s20.get(3, result) && result == "2");

// works in the case of aliasing when every sequence is the same

interleave(s15, s18, s17);

assert(s17.get(0, result) && result == "1" && s17.get(1, result) && result == "2" && s17.get(2, result) && result == "3" && s17.get(3, result) && result == "4");

// result equals seq1 if seq2 is empty

interleave(s18, s15, s17);

assert(s17.get(0, result) && result == "1" && s17.get(1, result) && result == "2" && s17.get(2, result) && result == "3" && s17.get(3, result) && result == "4");

// result equals seq2 if seq1 is empty