Hayley Martinez

Professor Smallberg

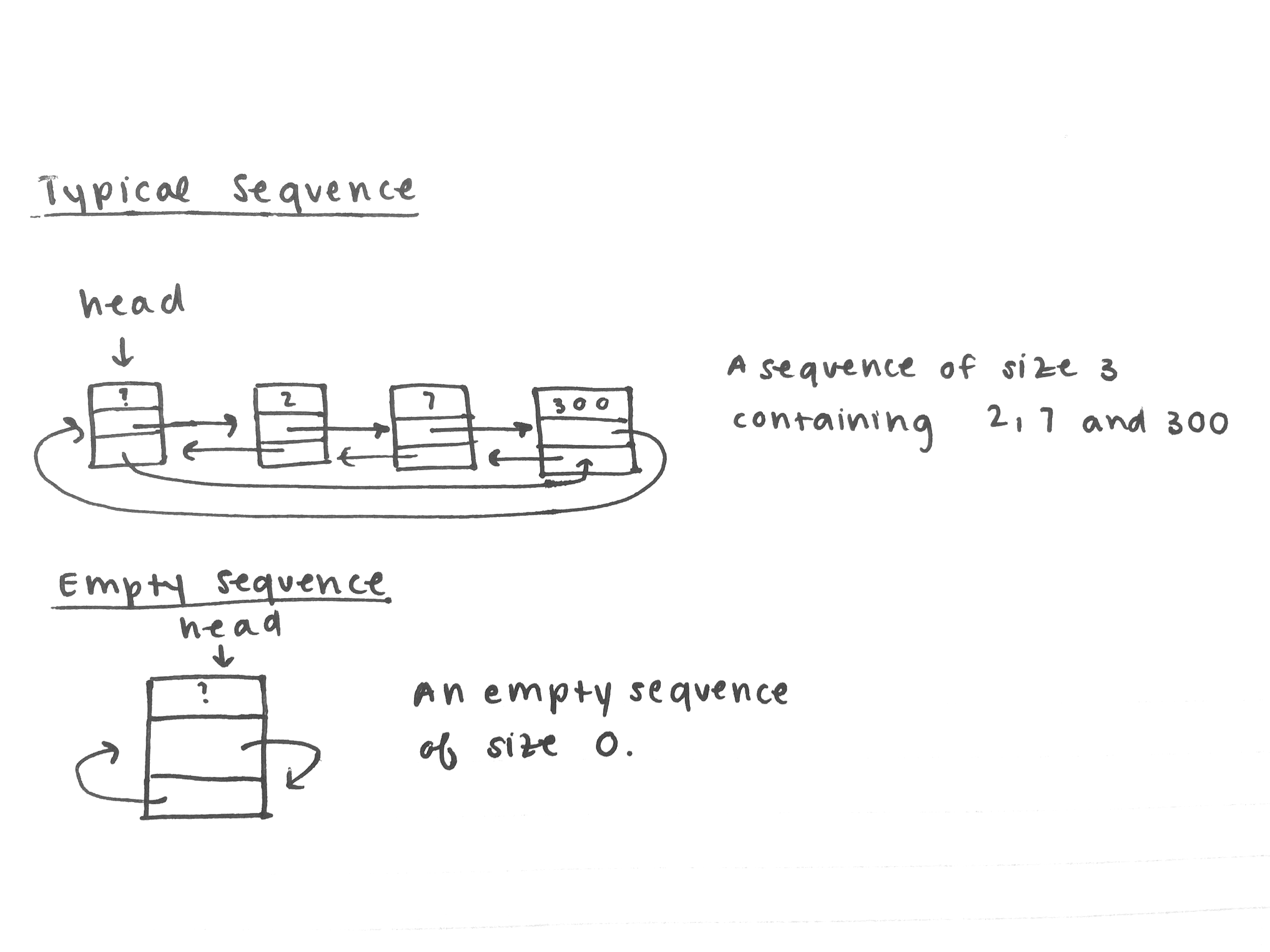
CS 32

23 April 2018

Project 2 Report

# LIST IMPLEMENTATION

The design implemented is a circularly doubly linked list. A Sequence consists of two member variables: a Node pointer head, and an integer m\_size. Each Node consists of three member variables: the stored value, a pointer to the previous Node in the list, and a pointer to the next Node in the list. The last Node’s next pointer will point to head. Similarly, head’s previous pointer will point to the last Node. For an empty Sequence, the next and previous pointers of head point to head. Below is a helpful diagram.



# PSEUDOCODE

*Copy constructor.*

create a new head

create a pointer to first node in source

repeatedly:

create a new node copy of same value as source

link the node to the end of the list

increment size

move to the next source node

*Insert (int).*

make a counter for position

repeatedly:

look through seq values

if the seq value is less than the passed value, break

increment position

insert passed value at the position

return the position

*Erase.*

if the position is out of bounds return false

until you reach the position in seq:

move to the next node in seq

relink the previous and following nodes

delete the current node

decrement size

return true

*Remove.*

make a counter for amount removed

repeatedly:

check if seq value is the same as source value

if it is, erase it

increment counter

otherwise move to the next seq node

return counter

*Get.*

if the position is out of bounds return false

until you reach the position in seq:

move to the next node in seq

assign the source value to the value at the seq node

return true

*Set.*

if the position is out of bounds return false

until you reach the position in seq:

move to the next node in seq

assign the value at the seq node to source value

return true

*Unchecked insert.*

create a new node w/ the specified value

until you reach the position in seq:

move to the next node in seq

relink previous and next nodes

update size of seq

*Subsequence.*

make a position counter

repeatedly:

check if seq1 value at position equals the first value in seq2

if it does

repeatedly:

check if seq1 value equals seq2 value

move to the next value in both

if all of them are the same, return position

else, keep searching

if the subsequence doesn’t exist, return -1

*Interleave.*

make sure there will be no aliasing

make a new sequence

if seq1 is longer than seq2

repeatedly:

alternatingly insert values from seq1 and seq2

starting with seq1 until all values from seq2

have been inserted into the new sequence

insert the rest of seq1 into the new sequence

if seq2 is longer than seq1

repeatedly:

alternatingly insert values from seq1 and seq2

starting with seq1 until all values from seq1

have been inserted into the new sequence

insert the rest of seq2 into the new sequence

set result equal to the new sequence

delete the sequence

# TEST CASES

Sequence bard;

std::string name;

// empty sequence

assert(bard.empty()); // test empty

assert(bard.size() == 0); // test size

assert(!bard.erase(0)); // no such position

assert(bard.remove("William") == 0); // nothing to remove

assert(!bard.get(0, name)); // no such position

assert(!bard.set(0, name)); // no such position

assert(bard.find("Shakespeare") == -1); // not found

// basic checks

Sequence cyprus;

assert(cyprus.insert(0, "Othello")); // test insert on empty

assert(cyprus.size() == 1); // test size

assert(cyprus.insert(1, "Desdemona")); // test insert

assert(cyprus.size() == 2); // test size

assert(cyprus.insert(1, "Cassio")); // test insert middle

assert(cyprus.size() == 3); // test size

assert(!cyprus.empty()); // test empty

assert(cyprus.get(2, name)); // test get

assert(name == "Desdemona"); // check

assert(cyprus.set(2, "Iago")); // test set

assert(cyprus.insert("Emilia") == 0); // test other insert

assert(cyprus.size() == 4); // test size

assert(cyprus.insert("Roderigo") == 4); // test other insert

assert(cyprus.size() == 5); // test size

assert(cyprus.erase(0)); // test erase

assert(cyprus.size() == 4); // test size

assert(cyprus.remove("Othello") == 1); // test remove

cyprus.dump(); // check names

// edge checks

Sequence vienna;

assert(vienna.insert("Angelo") == 0); // populate list

assert(vienna.insert(1, "Isabella"));

assert(vienna.insert(2, "Claudio"));

assert(vienna.insert(3, "Duke"));

assert(vienna.insert("Lucio") == 4);

assert(vienna.insert("Mariana") == 5);

assert(vienna.get(0, name)); // test get first

assert(name == "Angelo"); // check

assert(vienna.get(5, name)); // test get last

assert(name == "Mariana"); // check one

assert(vienna.set(0, "Duke")); // test set first

assert(vienna.set(5, "Duke")); // test set last

assert(vienna.find("Duke") == 0); // test find first & last

assert(vienna.erase(0)); // test erase first

assert(vienna.find("Duke") == 2); // test find in middle

assert(vienna.remove("Duke") == 2); // test remove last

assert(vienna.find("Lucio") == 2); // test find last

vienna.dump();

// copy constructor and operator=

Sequence france;

Sequence england;

assert(england.insert("Henry") == 0); // insert into empty

assert(england.insert(1, "Bardolph")); // populate list

assert(england.insert("Nym") == 2);

assert(england.insert("Montjoy") == 2);

assert(england.insert("Boy") == 0);

france.dump();

england.dump();

Sequence albion(england); // copy constructor

assert(albion.size() == england.size());// check sizes

albion.dump();

france.swap(england); // swap with empty seq

assert(france.size() == 5); // check size

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

france.dump();

england.dump();

assert(albion.set(0, "Lear")); // set new values

assert(albion.set(1, "Cordelia"));

assert(albion.set(2, "Edgar"));

assert(albion.set(3, "Edmund"));

assert(albion.set(4, "Kent"));

assert(albion.insert(5, "Gloucester"));

albion.dump();

england.operator=(albion); // operator= w/ diff size

assert(england.size() == 6); // check size

england.dump();

albion.operator=(bard); // operator= with empty

assert(albion.size() == 0); // check size

albion.dump();

// subsequence

Sequence uni;

assert(uni.insert(0, "Hamlet")); // populate list

assert(uni.insert(1, "Horatio"));

assert(uni.insert(2, "Laertes"));

uni.dump();

Sequence denmark;

assert(subsequence(uni, denmark) == -1);// empty subsequence

assert(subsequence(denmark, uni) == -1);// empty sequence

assert(denmark.insert(0, "Claudius")); // populate list

assert(denmark.insert(1, "Hamlet"));

assert(denmark.insert(2, "Ophelia"));

assert(denmark.insert(3, "Hamlet"));

assert(denmark.insert(4, "Horatio"));

assert(denmark.insert(5, "Laertes"));

assert(subsequence(denmark, uni) == 3); // check subsequence

assert(denmark.set(2, "Horatio")); // check subsequence

assert(denmark.remove("Hamlet") == 2);

denmark.dump();

// interleave

Sequence padua;

assert(padua.insert("Pedro") == 0); // populate list

assert(padua.insert("Benedick") == 0);

assert(padua.insert("John") == 1);

assert(padua.insert("Claudio") == 1);

assert(padua.insert(4, "Boracchio"));

padua.dump();

Sequence messina;

assert(messina.insert("Beatrice") == 0);

assert(messina.insert("Hero") == 1);

assert(messina.insert("Margaret") == 2);

messina.dump();

Sequence leonato;

interleave(messina, padua, leonato); // test when seq2 < seq1

assert(leonato.size() == 8); // test size

leonato.dump();

interleave(padua, messina, leonato); // test when seq1 > seq2

assert(leonato.size() == 8); // test size

leonato.dump();

interleave(bard, albion, albion); // test aliasing

interleave(albion, bard, albion); // test aliasing pt. 2