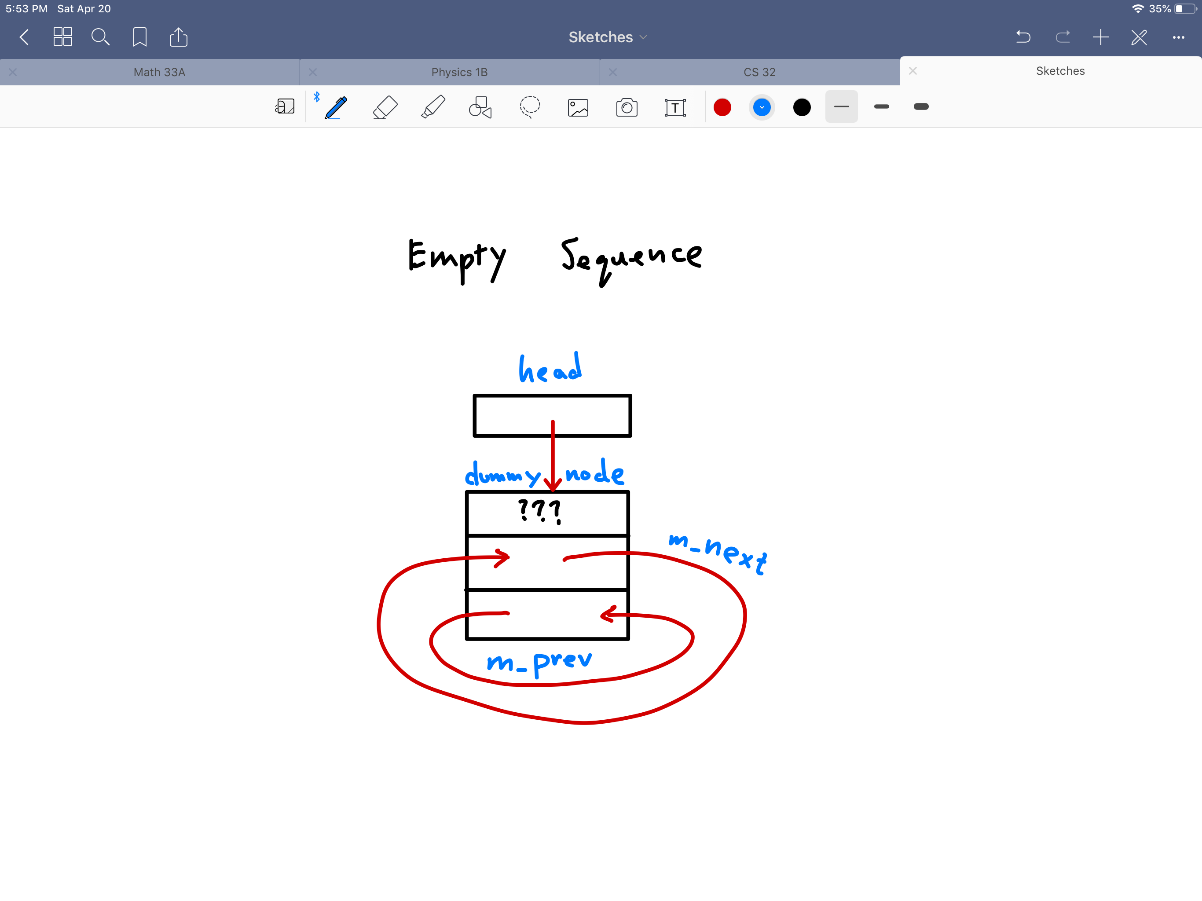
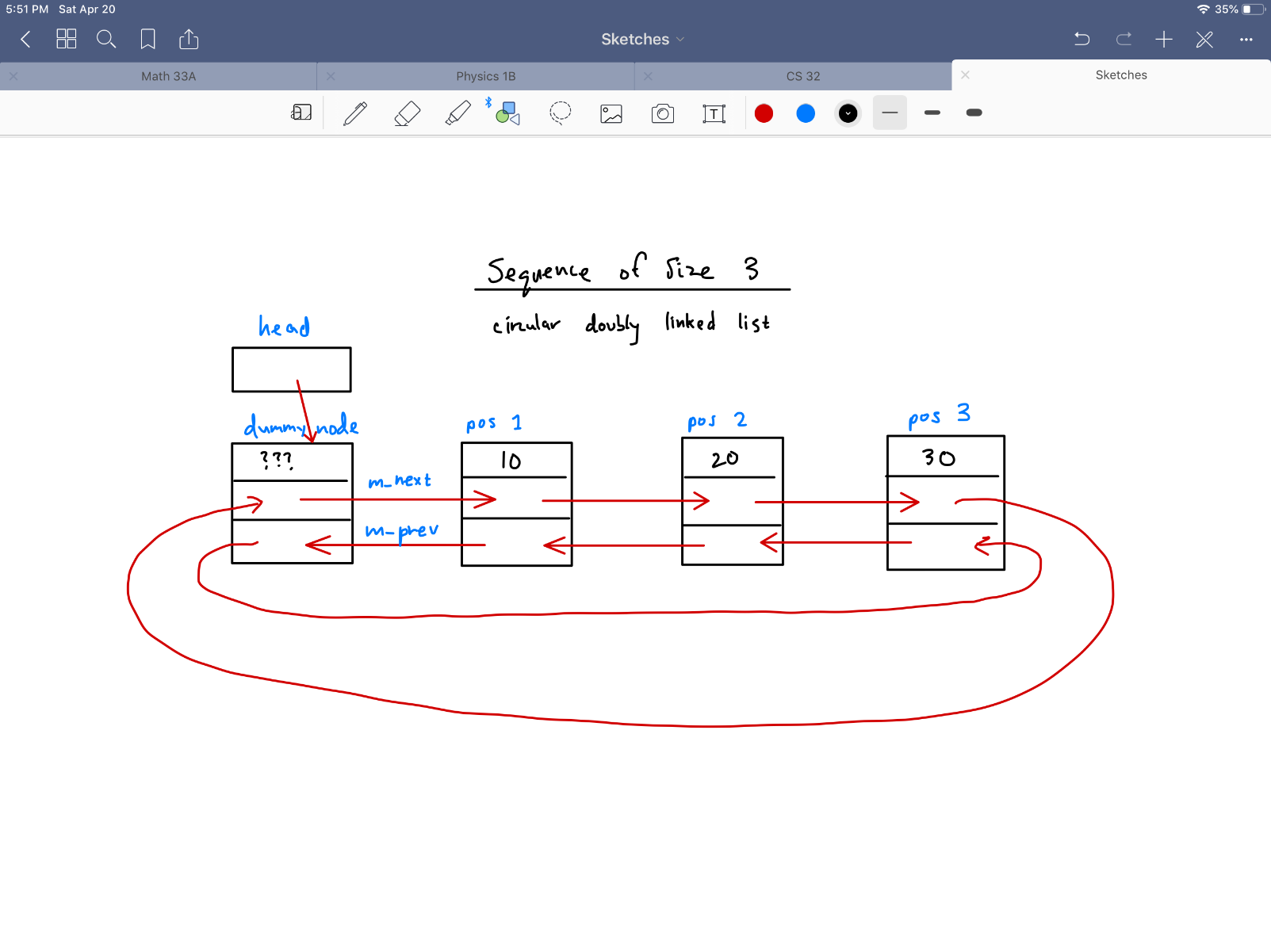
Joshua Liu

105136031

April 20, 2019

CS 32

Description of Design

My design of the program consists of a circular doubly linked list with a dummy node at the front. This design allows easy access to all the nodes, making it simpler to insert and delete elements from the list. The empty Sequence consists of a dummy node with a head pointer pointing to the dummy node. Because the list is circular, the elements will all point to the next element, and the last one will point back to the first node, which is the dummy node. There is one data holder in the list, which is m\_data, which holds the data for the Sequence. Aside from that, each node has two more member variables, which are the pointes m\_next and m\_prev. The m\_next pointer points to the next element, while the m\_prev pointer points to the previous element. All nodes are dynamically allocated. For the functions, a new pointer is created that points to head, which also points to the dummy node. We move this new pointer in order to access different nodes by going through the m\_next and m\_prev pointers. Below is a picture that visually shows what my linked list looks like.

Pseudocode

**Sequence::Sequence()**

*set sequence length to 0*

*make head point to dummy*

*make the dummy a circular list my making the next and previous pointer point to itself*

**Sequence::~Sequence()**

*create pointer p, point to a node, make it point at the last item of the list*

*loop through list starting from the back, stopping when we get back to dummy node*

*point p at previous node*

*delete the next node*

*delete last node(dummy node)*

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

*set head to dummy node*

*make dummy node a circular list by changing next and previous pointers*

*loop through body the amount of Sequence's size*

*create new ItemType*

*set this value to the value of a node in other's Sequence*

*set own Sequence's corresponding node to this value*

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

*if current object isn't other object*

*create temp Sequence*

*swap other with current Sequence*

*return current Sequence*

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

*if position given fits into the Sequence(between 0 and size, inclusive)*

*create a pointer that points to head(dummy node)*

*move pointer to given position*

*create a new Node and set its data*

*change the pointers to fit new Node into Sequence*

*increment size*

*return position*

*else return -1*

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

*create pointer pointing to head(dummy node)*

*move to first position(if none, then point to dummy node)*

*look through Nodes starting from position 0 and compare their values to given value*

*if given value is less than or equal to Node value*

*insert at this position and return position*

*else check next Node*

*if none are larger, insert at end and return position*

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

*if size is zero, return false*

*if position lies within Sequence*

*create pointer pointing to head*

*move pointer to this position*

*change the pointers to remove the Node*

*delete the Node*

*decrease size by 1*

*return true*

*else return false*

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

*create pointer p pointing to the first element*

*create a counter to count how many items are deleted*

*loop thorugh list using its size*

*if value of a Node is equal to value given*

*create a temporary pointer to p's previous Node*

*erase Node that p points to*

*increment counter*

*set p to temp*

*else check next Node*

*return counter*

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

*if position is valid in Sequence*

*create pointer p pointing to first element*

*make p point to given position*

*get value and put it into function's value*

*return true*

*else return false*

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

*if position is valid in Sequence*

*create pointer p pointing to first element*

*move p to position given*

*set the value of p's Node to value given*

*return true*

*else return false*

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

*create pointer point pointing to first position*

*loop through list*

*if Node data is equal to given value*

*return that position*

*else check next Node*

*else return -1*

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

*(copy and swap)*

*create temporary for all data member*

*set other data members to current data members*

*set current data members to temporary data members*

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

*if sequence 1 less than sequence 2*

*return -1*

*create sequence 1 counter get sequence 1 size*

*loop using sequence 1's size*

*create two ItemType variables*

*get sequence 1's value at sequence 1 counter's position and 2's values from position 0*

*compare values and if they are equal*

*check the following Nodes to see if they equal as well*

*if they are, return counter, which is the position of the equal substrings*

*else increment sequence 1 and check next Node*

*else, this means nothing matches so return -1*

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

*create a Sequence to store the results*

*if both sequences empty*

*set result to empty string and return*

*sequence 1 or 2 are empty*

*set result to correct sequence and return*

*create integer variables for the lengths and find the larger length*

*create a counter for inserting and positions of sequence 1 and 2, starting at 0*

*loop through max length*

*if sequence 1 position is less than its length*

*get the value at the position*

*put it in a temporary result at position of the insert counter*

*increment sequence 1 counter and insert counter*

*if sequence 2 position is less than its length*

*get the value at the position*

*put it in a temporary result at position of the insert counter*

*increment sequence 2 counter and insert counter*

*set results equal to temporary results*

*return*

Data to Test Program

Below is main code used to test my program. After each assert statement is the reason for the test. All tests cases run correctly.

// Test empty sequence

Sequence emptyOne;

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

assert(emptyOne.empty() == 1); // test empty

assert(emptyOne.erase(0) == 0); // test erase == 0

assert(emptyOne.remove(0) == 0); // test remove == 0

// Test insert and erase functions

Sequence a;

// test insert(position, value\_to\_be\_inserted)

assert(a.insert(0, 4) == 0); // insert at beginning

assert(a.insert(1, 10) == 1);

assert(a.insert(1, 9) == 1);

assert(a.insert(1, 8) == 1);

assert(a.insert(1, 7) == 1);

assert(a.insert(1, 6) == 1);

assert(a.insert(1, 5) == 1);

assert(a.insert(7, 100) == 7); // insert at end

// 4 5 6 7 8 9 10 100

assert(a.erase(2) == 1); // test erase from position 2

// 4 5 7 8 9 10 100

// test insert(value\_to\_be\_inserted)

assert(a.insert(6) == 2);

assert(a.insert(0) == 0); // insert at front

assert(a.insert(10) == 7);

// 0 4 5 6 7 8 9 10 10 100

assert(a.insert(100) == 9); // insert at back

// test insert and remove functions

Sequence c;

// insert values

assert(c.insert(0, 1) == 0);

assert(c.insert(1, 3) == 1);

assert(c.insert(2, 5) == 2);

// 1 3 5

assert(c.insert(6) == 3);

assert(c.insert(6) == 3);

assert(c.insert(7) == 5);

// 1 3 5 6 6 7

assert(c.remove(6) == 2); // remove more than 1 value

assert(c.insert(7) == 3);

assert(c.remove(7) == 2); // remove from back

assert(c.remove(1) == 1); // remove from front

assert(c.insert(0, 1) == 0);

// 1 3 5

ItemType get; // used to get values to test get function

assert(c.get(1, get) == 1 && get == 3); // test get

assert(c.get(0, get) == 1 && get == 1); // test get from front

assert(c.get(2, get) == 1 && get == 5); // test get from back

assert(c.get(-1, get) == 0); // test invalid position

assert(c.get(3, get) == 0); // test invalid position

// test remove same values in a sequence of 1 value

Sequence allSame;

assert(allSame.insert(5) == 0);

assert(allSame.insert(5) == 0);

assert(allSame.insert(5) == 0);

assert(allSame.insert(5) == 0);

assert(allSame.insert(5) == 0);

assert(allSame.remove(5) == 5); // remove all 5's

ItemType x\_allSame = 2; // test get function on empty Sequence

assert(allSame.get(0, x\_allSame) == 0 && x\_allSame == 2); // ensure value of ItemType variable does not change

// test set nd find functions

Sequence set;

// insert values

assert(set.insert(5) == 0);

assert(set.insert(4) == 0);

assert(set.insert(3) == 0);

assert(set.insert(2) == 0);

assert(set.insert(1) == 0);

// set values for all Nodes

assert(set.set(0, 99) == 1);

assert(set.set(1, 98) == 1);

assert(set.set(2, 97) == 1);

assert(set.set(3, 96) == 1);

assert(set.set(4, 95) == 1);

assert(set.set(-1, 7) == 0); // test for invalid position

assert(set.set(5, 7) == 0); // test for invalid position

// 99 98 97 96 95

assert(set.find(50) == -1); // test for no values in Sequence

assert(set.find(99) == 0);

assert(set.find(98) == 1);

assert(set.find(97) == 2);

assert(set.find(96) == 3);

assert(set.find(95) == 4);

// test swap

Sequence one; // has 4 values

assert(one.insert(30) == 0);

assert(one.insert(20) == 0);

assert(one.insert(10) == 0);

assert(one.insert(5) == 0);

Sequence two; // has 3 values

assert(two.insert(300) == 0);

assert(two.insert(200) == 0);

assert(two.insert(100) == 0);

one.swap(two); // swap

assert(one.size() == 3); // test sizes after swap

assert(two.size() == 4); // test sizes after swap

// test individual sequences

assert(one.insert(55) == 0); // make sure insert works if it has correct size

assert(two.insert(55) == 4); // make sure insert works if it has correct size

// test copy constructor

Sequence original; // insert values

assert(original.insert(36) == 0);

assert(original.insert(31) == 0);

assert(original.insert(19) == 0);

Sequence copy(original);

assert(copy.find(36) == 2); // check to see correct value

assert(copy.find(31) == 1); // check to see correct value

assert(copy.find(19) == 0); // check to see correct value

assert(copy.size() == 3); // check size

// test assignment operator

Sequence equal1; // insert values

assert(equal1.insert(87) == 0);

assert(equal1.insert(86) == 0);

assert(equal1.insert(85) == 0);

Sequence equal2; // insert values

assert(equal2.insert(123) == 0);

assert(equal2.insert(122) == 0);

assert(equal2.insert(121) == 0);

equal2 = equal1; // use assignment operator, equal2 Sequence should be deleted

assert(equal2.find(87) == 2); // test value

assert(equal2.find(86) == 1); // test value

assert(equal2.find(85) == 0); // test value

assert(equal2.size() == 3); // test size

std::cerr << "Sequence Class tests passed" << std::endl;

// test subsequence function

Sequence testingSS1; // insert values

testingSS1.insert(0, 13);

testingSS1.insert(1, 11);

testingSS1.insert(2, 12);

testingSS1.insert(3, 13);

testingSS1.insert(4, 12);

testingSS1.insert(5, 11);

testingSS1.insert(6, 12);

testingSS1.insert(7, 11);

testingSS1.insert(8, 14);

testingSS1.insert(9, 67);

Sequence testingSS2; // insert values, pattern to be found in testingSS1

testingSS2.insert(0, 11);

testingSS2.insert(1, 12);

testingSS2.insert(2, 11);

assert(subsequence(testingSS1, testingSS2) == 5); // test functionality

assert(subsequence(testingSS1, testingSS1) == 0); // make sure same string works

assert(subsequence(testingSS2, testingSS1) == -1); // first sequence smaller than second

// test interleave

Sequence il1; // first sequence

il1.insert(0, 1);

il1.insert(1, 3);

il1.insert(2, 5);

il1.insert(3, 7);

il1.insert(4, 8);

Sequence il2; // second sequence

il2.insert(0, 2);

il2.insert(1, 4);

il2.insert(2, 6);

Sequence ilResult; // result

ilResult.insert(0, 76); // put values in result

ilResult.insert(0, 24); // should not appear in result after function

interleave(il1, il2, ilResult);

// check values

assert(ilResult.find(1) == 0);

assert(ilResult.find(2) == 1);

assert(ilResult.find(3) == 2);

assert(ilResult.find(4) == 3);

assert(ilResult.find(5) == 4);

assert(ilResult.find(6) == 5);

assert(ilResult.find(7) == 6);

assert(ilResult.find(8) == 7);

assert(ilResult.size() == 8);

// test interleave with seq1 as result

interleave(il1, il2, il1);

assert(il1.find(1) == 0);

assert(il1.find(2) == 1);

assert(il1.find(3) == 2);

assert(il1.find(4) == 3);

assert(il1.find(5) == 4);

assert(il1.find(6) == 5);

assert(il1.find(7) == 6);

assert(il1.find(8) == 7);

assert(il1.size() == 8);

// test interleave with seq2 empty

Sequence aaa;

aaa.insert(10);

aaa.insert(8);

aaa.insert(6);

aaa.insert(4);

Sequence bbb; // empty sequence

Sequence res1;

interleave(aaa, bbb, res1);

assert(res1.find(4) == 0);

assert(res1.find(6) == 1);

assert(res1.find(8) == 2);

assert(res1.find(10) == 3);

// test interleave with seq1 empty

interleave(bbb, aaa, res1);

assert(res1.find(4) == 0);

assert(res1.find(6) == 1);

assert(res1.find(8) == 2);

assert(res1.find(10) == 3);

// test interleave with all empty

Sequence emptyinterleave;

interleave(emptyinterleave, emptyinterleave, emptyinterleave);

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

assert(emptyinterleave.empty() == 1);

std::cerr << "Functions tests passed" << std::endl;