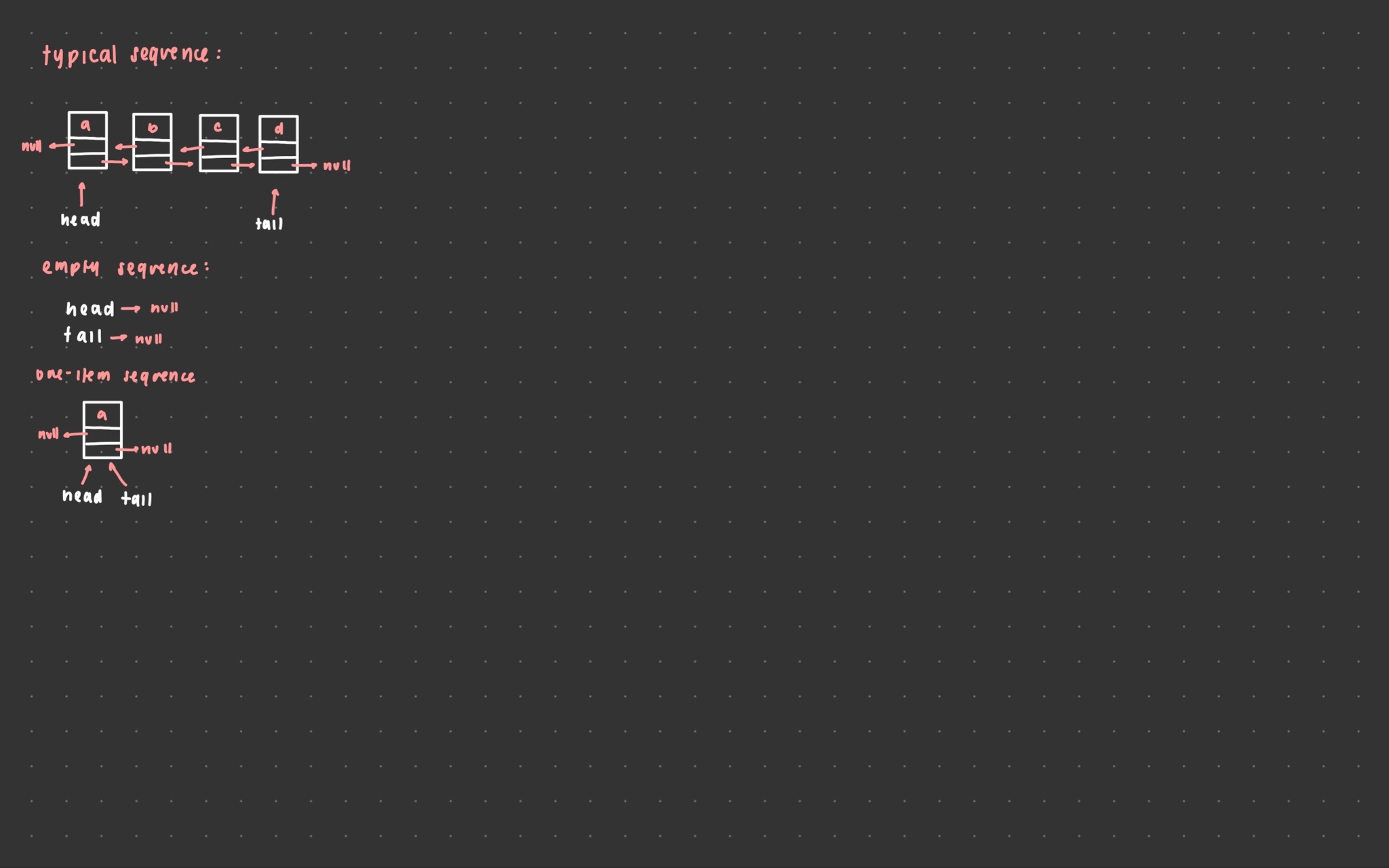
Ying Chou

905754513

CS 32: Project 2

**Design of Doubly-Linked List:**

Each node will contain a pointer to the next node, a pointer to the previous node, and hold a value. I did not use a dummy node and my list is not circular, so the very first node of the list is the head and the last node is the tail. The head’s previous pointer is nullptr, and the tail’s next pointer is nullptr. In a single-item sequence, that item is both the head and tail, and both its previous pointer and next pointer is nullptr. In an empty sequence, both the head and tail points to nullptrs.



**Pseudocode:**

Sequence Constructor:

Sets the size of the sequence to 0

Points both the head and the tail to nullptr

Sequence Copy Constructor:  
 If the other sequence is empty:

Set the size of the sequence to 0

Points both the head and the tail to nullptr

Otherwise:

Create a new node for the head

Set the head’s value to the other head’s value

Create two pointers to traverse other and this (current this node and current other node)

Traversing every node of other:

Make a new node to add to the list

Set the value to the respective node from other

Connect the new node to the previous node

Set the tail to the very last node created

Sequence Operator =:

Check if the sequence isn’t at the same address as the one it’s being set to

Create a copy of rhs (what needs to be assigned) into a temp variable

Swaps this sequence with the temp variable

Return this

Sequence Destructor:  
 Check that the sequence is not empty

Traverse through each node of the list:

Delete the current node

Change the current node to the next node

Sequence Insert (two argument):

If the index passed is not valid

Returns -1

Otherwise:

If the set is empty:

Create a new node and set its value as passed

Set the previous and next pointers to nullptr

Set the head and the tail to this new node

Increment the sequence size

Return position

If adding at the beginning of the list:

Create a new node and set its value as passed

Set the previous pointer to null and next pointer to head

Set the head to the new node

Increment the sequence size

Return position

If adding at the end of the list:

Create a new node and set its value as passed

Set the previous pointer to tail and next pointer to null

Set the tail to the new node

Increment the sequence size

Return position

If adding in the middle of the list:

Traverse the list to find the index passed

Create a new node and set its value as passed

Connect the new node to the previous and next node

Increment the sequence size

Return position

Sequence Insert (one argument):

If sequence is empty:

Create a new node, set its values, set its pointers to null

Reassign head and tail

Increment the sequence size

Return position

Otherwise:

Traverse the list to find the position

If adding at the beginning of the list:

Create a node, set its value, reassign pointers

Reassign head

Increment size

Return position

If adding at the end of the list:

Create a node, set its value, reassign pointers

Reassign tail

Increment size

Return position

If adding in the middle:

Create a node, set its value, reassign pointers

Increment size

Return position

Sequence erase:

If the index passed is not valid:

Return false

Otherwise:

Traverse the list to find the index passed

If erasing in a one item sequence:

Reassign head and tail to null

Decrement the size

Delete the node

Return true

If erasing the first node:  
 Assign head to the next node, set its previous to null

Decrement the size

Delete the node

Return true

If erasing the last node:

Assign tail to the previous node, sets its next to null

Decrement the size

Delete the node

Return true

If erasing in the middle:

Reassign pointers, excluding the to be deleted node

Decrement the size

Delete the node

Return true

Sequence remove:

If the list is empty:

Nothing to remove (return 0)

If sequence has one item:

Erase at index

Increment removed counter and return

Loop through list to find passed value:

If found:

Erase at that index

Increment removed counter

If removing at the end of list:

erase at index

Increment removed counter

Return removed counter

Sequence get:

If the index passed is not valid:

Return false

Loop through list to find the index passed

Set value to the value of the node found

Return true

Sequence set:

If the index passed is not valid:

Return false

Loop through list to find the index passed

Set value of the node found to value

return true

Sequence find:

If the sequence is empty:

Return -1

If one item in the list:

Return position (0)

Loop through list to find the value passed

If found:

Pass position it was found at

If needed node is at the end of the sequence:

Pass the last position of the list

If not found, return -1

Sequence swap:

Swap the sizes of this and other

Swap the head and tail of this and other

Subsequence:

If the sequences passed are not valid:

Return -1

Otherwise:

Loop through seq1:

Access each value in the nodes

Loop through seq2

Access each value in the nodes

If the values of seq1 and seq2 don’t match:

Reset the position of seq2 to 0

break

If loop reaches the end of seq2

Return starting position in seq1

If subsequence not found, return -1

Interleave:

If seq1 and seq2 are empty:

Erase the elements already in result

If seq1 or seq2 are empty:

Set result to the not empty one

If the addresses of seq1 or seq2 match result:

Create a copy of seq1/seq2 to use instead of seq1/seq2

` run the same code as below

Otherwise:

If seq1 and seq2 are the same size:

Loop through result to add items:

Access each value in seq1 and seq2

Insert seq1 values into even result indexes

Insert seq2 values into odd result indexes

Delete the items result started with

If seq2 size is greater than seq1:

Loop through result to add items (stopping at size of seq1):

Access each value in seq1 and seq2

Insert seq1 values into even result indexes

Insert seq2 values into odd result indexes

Loop through the rest of seq2 to add the items

Delete the items result started with

If seq1 size is greater than seq2:

Loop through result to add items (stopping at size of seq2):

Access each value in seq1 and seq2

Insert seq1 values into even result indexes

Insert seq2 values into odd result indexes

Loop through the rest of seq1 to add the items

Delete the items result started with

**Test Cases:**

// testing with ItemType as std::string

// testing with empty sequence

Sequence empty;

std::string x;

assert(empty.empty() == true); // test empty

assert(empty.size() == 0); // test size on an empty sequence

assert(empty.erase(0) == false); // test erase on an empty sequence

assert(empty.erase(3) == false);

assert(empty.remove("heyo") == 0); // test remove on an empty sequence

assert(empty.get(0, x) == false); // test get on an empty sequence

assert(empty.get(3, x) == false);

assert(empty.set(0, "heyo") == false); // test set on an empty sequence

assert(empty.set(3, "heyo") == false);

assert(empty.find("heyo") == -1); // test find on an empty sequence

// testing insertions, erase, and remove

Sequence ss;

assert(ss.insert(0, "bbb") == 0); // test insert (2 arguments) on an empty sequence

assert(ss.size() == 1);

assert(ss.insert(1, "eee") == 1); // test insert (2 arguments), adding at end of sequence

assert(ss.insert(1, "ccc") == 1); // test insert (2 arguments), adding in the middle of seqeunce

assert(ss.insert(0, "aaa") == 0); // test insert (2 arguments), adding at the beginning of sequence

assert(empty.insert("heyo") == 0); //test insert (1 argument) on an empty sequence

assert(ss.insert("ddd") == 3); // test insert (1 argument), adding in the middle of sequence

assert(ss.insert("fff") == 5); // test insert (1 argument), adding at the end of sequence

assert(ss.insert("aaa") == 0); //test insert (1 argument), adding at the beginning of sequence

assert(ss.erase(0) == true); // test erase on first node in sequence

assert(ss.erase(5) == true); // test erase on last node in sequence

assert(ss.erase(2) == true); // test erase on a node in the middle of sequence

assert(empty.erase(0) == true); // test erase for a one item sequence

assert(ss.size() == 4); // to see if erase worked

assert(empty.size() == 0 && empty.empty() == true);

assert(ss.erase(10) == false); // test erase with nonvalid index argument

assert(ss.insert("aaa") == 0); // adding "aaa" to be removed

assert(ss.insert("aaa") == 0);

assert(ss.insert("aaa") == 0);

assert(ss.remove("aaa") == 4); // test remove with removeable argument (multiple) + at the beginning of the sequence

assert(ss.remove("eee") == 1); // test remove with removeable argument (one) + at the end of the sequence

assert(ss.remove("zzz") == 0); // test remove with non-removeable argument

assert(empty.insert("heyo") == 0); // adding to be removed

assert(empty.remove("heyo") == 1); // test remove on one item sequence

assert(ss.size() == 2); // to see if erase worked

assert(empty.size() == 0 && empty.empty() == true);

// testing get, set, and find

assert(ss.get(0, x) == true && x == "bbb"); // test get on the first node in sequence

assert(ss.get(1, x) == true && x == "ddd"); // test get on the last node in sequence

assert(ss.insert("ccc") == 1);

assert(ss.get(1, x) == true && x == "ccc"); // test get on node in the middle of sequence

assert(empty.insert("heyo") == 0);

assert(empty.get(0, x) == true && x == "heyo"); // test get on a one item sequence

assert(ss.get(3, x) == false && x == "heyo"); // test get with invalid index argument

assert(ss.set(0, "zzz") == true); // test set on the first node in sequence

assert(ss.set(2, "xxx") == true); // test set on the last node in sequence

assert(ss.set(1, "yyy") == true); // test set on node in the middle of sequence

assert(empty.set(0, "byebe") == true); // test set on a one item sequence

assert(ss.set(3, "www") == false); // test set with invalid index argument

assert(ss.find("zzz") == 0); // test find on the first node in sequence

assert(ss.find("xxx") == 2); // test find on the last node in sequence

assert(ss.find("yyy") == 1); // test find on node in the middle of sequence

assert(ss.insert("xxx") == 0);

assert(ss.insert("xxx") == 0);

assert(ss.find("xxx") == 0); // test find with multiple nodes with the same value

assert(empty.find("byebe") == 0); // test find on a one item sequence

assert(ss.find("aaa") == -1); // test find with argument not in sequence

// testing copy constructor, assignment operator, and swap

Sequence copy(ss); // test copy constructor

assert(copy.size() == 5); // check to see if it was copied over correctly

assert(copy.get(0, x) == true && x == "xxx");

assert(copy.get(1, x) == true && x == "xxx");

assert(copy.get(2, x) == true && x == "zzz");

assert(copy.get(3, x) == true && x == "yyy");

assert(copy.get(4, x) == true && x == "xxx");

assert(copy.insert(5, "haha") == 5);

assert(copy.size() == 6 && ss.size() == 5); // check to make sure ss and copy are not the same list

assert(ss.get(5, x) == false && x == "xxx");

empty = copy; // test assignment operator

assert(empty.size() == 6); // check to see if the assignment worked

assert(empty.get(0, x) == true && x == "xxx");

assert(empty.get(1, x) == true && x == "xxx");

assert(empty.get(2, x) == true && x == "zzz");

assert(empty.get(3, x) == true && x == "yyy");

assert(empty.get(4, x) == true && x == "xxx");

assert(empty.get(5, x) == true && x == "haha");

assert(empty.remove("haha") == 1);

assert(empty.size() == 5 && copy.size() == 6); // check to make sure copy and empty are not the same list

assert(copy.get(5, x) == true && x == "haha");

assert(empty.get(5, x) == false && x == "haha");

assert(copy.remove("xxx") == 3);

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

assert(empty.size() == 5);

copy.swap(empty); // testing swap

assert(copy.size() == 5);

assert(copy.get(0, x) == true && x == "xxx");

assert(copy.get(1, x) == true && x == "xxx");

assert(copy.get(2, x) == true && x == "zzz");

assert(copy.get(3, x) == true && x == "yyy");

assert(copy.get(4, x) == true && x == "xxx");

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

assert(empty.get(0, x) == true && x == "zzz");

assert(empty.get(1, x) == true && x == "yyy");

assert(empty.get(2, x) == true && x == "haha");

Sequence sad;

empty.swap(sad); // testing swap with an empty sequence

assert(sad.size() == 3 && empty.size() == 0);

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

assert(sad.get(0, x) == true && x == "zzz");

assert(sad.get(1, x) == true && x == "yyy");

assert(sad.get(2, x) == true && x == "haha");

// testing subsequence

Sequence one;

Sequence two;

Sequence three;

assert(subsequence(one, two) == -1); // test subsequence with two empty sequences

one.insert(0, "a");

one.insert(1, "b");

one.insert(2, "c");

one.insert(3, "d");

one.insert(4, "e");

assert(subsequence(one, one) == 0); // test subsequence works with aliasing, passing the same sequence for seq1 and seq2

assert(subsequence(one, two) == -1); // test subsequence with empty seq2

three.insert("v");

three.insert("w");

three.insert("x");

three.insert("y");

three.insert("z");

assert(subsequence(three, two) == -1); // test subsequence where seq2 is not in seq1 at all

two.insert("a");

assert(subsequence(one, two) == 0); // test subsequence with one item seq2 + k at the beginning of seq1

two.set(0, "b");

assert(subsequence(one, two) == 1); // k in the middle of seq1

two.set(0, "c");

assert(subsequence(one, two) == 2); // k at the end of seq1

two.set(0, "a");

two.insert("c");

two.insert("e");

assert(subsequence(one, two) == -1); // test subsequence with non consequtive subsequence

two.set(1, "b");

two.set(2, "c");

assert(subsequence(one, two) == 0); // test subsequence with consecutive subsequence, starts at beginning

two.set(0, "b");

two.set(1, "c");

two.set(2, "d");

assert(subsequence(one, two) == 1); // test subsequence with consecutive subsequence, starts in middle

one.insert(3, "a");

one.insert(4, "b");

one.insert(7, "b");

one.insert(8, "c");

one.insert(9, "d");

assert(subsequence(one, two) == 7); // test subsequence with parts of seq2 in seq1

// testing interleave

Sequence result;

// when result is empty

Sequence m;

Sequence n;

interleave(m, n, result); // test interleave with empty seq1 and seq2

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

m.insert(0, "a");

m.insert(1, "b");

m.insert(2, "c");

interleave(m, n, result); // test interleave with empty seq2

assert(result.size() == m.size());

result.erase(2);

result.erase(1);

result.erase(0);

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

interleave(n, m, result); // test interleave with empty seq1

assert(result.size() == m.size());

result.erase(2);

result.erase(1);

result.erase(0);

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

n.insert(0, "x");

n.insert(1, "y");

n.insert(2, "z");

interleave(m, n, result); // test interleave with same size seq1 and seq2

assert(result.size() == m.size() + n.size());

result.erase(5);

result.erase(4);

result.erase(3);

result.erase(2);

result.erase(1);

result.erase(0);

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

m.insert(3, "d");

m.insert(4, "e");

m.insert(5, "f");

interleave(m, n, result); // test interleave with seq1 > seq2

assert(result.size() == m.size() + n.size());

result.erase(8);

result.erase(7);

result.erase(6);

result.erase(5);

result.erase(4);

result.erase(3);

result.erase(2);

result.erase(1);

result.erase(0);

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

interleave(n, m, result); // test interleave with seq2 > seq1

assert(result.size() == m.size() + n.size());

// when result is not empty

Sequence p;

Sequence k;

interleave(p, k, result); // test interleave with empty seq1 and seq2

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

p.insert(0, "p");

p.insert(1, "o");

p.insert(2, "o");

p.insert(3, "p");

interleave(p, k, result); // test interleave with empty seq2

assert(result.size() == p.size());

interleave(k, p, result); // test interleave with empty seq1

assert(result.size() == p.size());

k.insert(0, "f");

k.insert(1, "a");

k.insert(2, "r");

k.insert(3, "t");

interleave(p, k, result); // test interleave with same size seq1 and seq2

assert(result.size() == p.size() + k.size());

p.insert(4, "e");

p.insert(5, "e");

interleave(p, k, result); // test interleave with seq1 > seq2

assert(result.size() == p.size() + k.size());

interleave(k, p, result); // test interleave with seq2 > seq1

assert(result.size() == p.size() + k.size());

interleave(p, p, result); // test interleave with aliasing, same seq1 and seq2

assert(result.size() == p.size() + p.size());

interleave(p, k, p); // test interleave with aliasing same seq1 and result

assert(p.size() == 10);

interleave(p, k, k); // test interleave with aliasing same seq2 and result

assert(p.size() == 10);