Report: Project 2

**Implementation:**

In this project, I created a doubly-linked list with access to both the head and tail. In my inserting implementations, I approached the challenge by adding the new Nodes to the back of the linked list. The tail was mostly accessed by a previous pointer that kept track of the previous node when it needed to delete or access already-observed parts of the linked list. The list is not circularly created, and looks like as follows:

previous

previous

previous

next

next

next

head

With data:

tail

Pair

Key,value

Pair

Key,value

Pair

Key,value

Note: every node contains a next and previous pointer with a pair

head

p = NULL

next = NULL

previous = NULL

tail

p = NULL

next = NULL

previous = NULL

NULL

Without data:

**Pseudocode for non-trivial algorithms:**

**Erase:**

Set a temp node for counting

//most functions required a temp pointer

find out the position (if one) in the list

if the key to be erased is not found, return false

repeatedly:

search through the list for the key in the map

if there is one element,

set it to an empty linked list

if it is the head,

set the head to the next node, remove data at current node

if it is the tail

set the tail to the previous node, remove data at current node

otherwise,

connect the previous and next node together, remove data at current node

**Swap:**

Store values of the first and second map’s head and tail

Use the temp variables to set the opposite map’s head and tails with the other map’s head and tail.

Then, switch the numeric sizes of each map.

**Insert/Update (insert, update, doInsertOrUpdate):**

Create a new node for searching (iterator)

Look to see if it is in the list

If it is (and it wants to update),

Update it with the value the iterator is pointing to

Otherwise (if it wants to be inserted),

Insert a node into the list

If the first value,

Point everything to the node

If another value,

Add to the back of the list, increase size

**Combine:**

Make a temporary array to edit the values of the second map

Repeatedly:

Check to see if a node from the first map is in the second map

If it is and it does NOT match the key AND value,

Note the false combining and continue and continue combining

If it is and it matches the key AND value,

Insert it only once

At the end, set the result array as the temp array edited

**Subtract:**

Make a temporary array to edit the values of the first map

Repeatedly:

Check to see if a key from the second map is in the map

If it is present,

Erase the node

At the end, set the result array as the temp array edited

**Test Cases:**

~Map();

“Forget” to set certain pointers to NULL //error

Node \*previous as a private member of Node vs. a private member of Map //error

Map&operator=(const Map &mp)

Map=n;

...

//insert things into n

Map=o;

O=n;

assert(n.size() == o.size()) //should be true

//start from beginning and do n=o; should become an empty list

assert(n.size() == 0)

Map(const Map &mp)

Map q;

assert(q.size() == 0); //make sure there’s nothing

assert(q.empty()); //should return yes, it is empty (true)

assert(!q.erase(“Lucy”)); //can’t erase nothing

**Insert/Update (insert, update, doInsertOrUpdate)**

**(PERFORM WITH EMPTY AND NONEMPTY LISTS)**

assert(m.insert("Lucy", 789));

assert(!m.insert("Lucy", 789)); //won’t add duplicates

assert(!m.insert("Lucy", 799)); //key already exists //(should use update instead!)

//try to insert 251 items, will not work

bool erase(const KeyType& key)

**(PERFORM WITH EMPTY AND NONEMPTY LISTS)**

Map n;

assert(!n.erase("lol”); //there’s nothing in there!

assert(n.insert("e", 5));

//(before moving on, try deleting this single node)

assert(n.insert("f", 6));

assert(n.insert("g", 7));

//for trying deletion of middle node, f

assert(n.erase("g”); //try deleting tail

assert(!n.erase("g”); //should not be able to erase

assert(!n.erase("h”); //doesn’t exist

bool contains(const KeyType& key) const

assert(!m.contains("Bret")); //not in the list

assert(!m.contains("")); //not there, only when a new //node is initialized

bool get(const KeyType& key, ValueType& value) const

assert(!m.get("Lucy", v) && v == 42);

//make sure value changes

assert(m.get("Fred", v) && v == 123);

assert(m.get(0, x, v) &&

((x == "Fred" && v == 123) || (x == "Ethel" && v == 456)));

KeyType x2 = "Ricky";

assert(m.get(1, x2, v) &&

((x2 == "Fred" && v == 123) || (x2 == "Ethel" && v == 456)) && x != x2);

//also checking keys that are not in the map

bool get(int i, KeyType& key, ValueType& value) const

//replace i with the m\_size or a negative, not accessible

//inadvertently tested numerous times with other functions (i.e many of the functions would not work properly without this working)

//could use contains(m1) to make sure keys and values were actually transferred

void swap(Map& other)

m.swap(n) when: (? = some amount of data)

m=empty, n=? 🡪 should produce m=?,n=empty

m=empty, m=empty -> stays the same

m=?, m=? -> stays the same

//checking commutative properties

bool combine(const Map& m1, const Map& m2, Map& result)

Map m1 has: "Fred" 123,"Ethel" 456,"Lucy" 789

Map m2 has: "Lucy" 789, "Ricky" 321

Result: "Fred" 123,"Ricky" 321,"Lucy" 789,"Ethel" 456

Return value: true //general case, check for order

//Check for value discrepencies

Map m1 has: "Fred" 123,"Ethel" 456,"Lucy" 789

Map m2 has: "Lucy" 654, "Ricky" 321

Result: "Fred" 123,"Ricky" 321, “Ethel" 456

Return value: false

Map m1 has: "Fred" 123,"Ethel" 456,"Lucy" 789

Map m2 has: "Fred" 123,"Ethel" 456,"Lucy" 789

Result: "Fred" 123,"Ethel" 456,"Lucy" 789

//Check for duplicates

Map m1 has: "Fred" 123,"Ethel" 456,"Lucy" 789

Map m2 has: "Fred" 111,"Ethel" 111,"Lucy" 111

Result: Empty

//conflicting values, doesn’t know which is real one

void subtract(const Map& m1, const Map& m2, Map& result)

subtract(m1, m2, result)

Map m1 has: "Fred" 123, "Ethel" 456, "Lucy" 789

Map m2 has: "Lucy" 789,"Ricky" 321,"Ethel" 654

Result: "Fred" 123 //normal case

subtract(m2, m1, result)

Result: "Ricky" 321

subtract(m1, m1, result) //check for alias case

Result: empty list

subtract(m1, m2, result) //where m1 is empty

Result: empty list

subtract(m1, m2, result) //where m2 is empty

Result: m1

//check that it is not commutative