My doubly-linked list consists of various nodes each containing A Keytype, a Valuetype, a pointer to the next node, and a pointer to the previous node. Also my doubly-linked list contains a private int that keeps track of the size of the list, a pointer to the “head” or first node in the list, and a pointer to the “tail” or the last node in the list. Also, it contains all the various public functions including but not limited to 2 constructors, a destructor, one that returns whether the list is empty or not, one that returns the size, and functions to insert values or update them or erase them.

When empty, my doubly-linked list consists of no nodes, two pointers (head and tail) that are nullptrs, and an int (m\_size) equal to 0. A typical map will contain multiple nodes each containing data about Keytypes, Valuetypes, and two pointers pointing to the next and last node. The first node’s prev pointer will be a nullptr, and the last node’s next pointer will also be a nullptr. m\_size will be the amount of nodes the doubly-linked list has, and head will point to the first node as tail points to the last.

My list is not circular, nor does it have a dummy node. The order is in the order that people put values in.

**Deconstructor:**

Map::~Map {

if it’s not empty:

set a temp pointer to the head

create a second (temp2) pointer not pointing to anything

while the first temp pointer isn’t null:

set temp2 equal to the one right after the head

delete temp

make temp equal to temp2

set the size equal to 0

**Copy constructor:**

Map::Map(const Map& m) {

set size to 0, make head and tail nullptrs

make a Keytype and a Valuetype

for each of Map m’s nodes:

insert map m’s Keytype and Valuetype

**Insert**

bool Map::insert(const KeyType& key, const ValueType& value)

if the key is already in there:

return false

create new node n with Keytype key and Valuetype value

set n’s pointer to the next node to null

if its empty:

set head equal to n and tail equal to n

make n’s pointer to the previous node to null

otherwise:

set n’s pointer to the previous node to the last node in the list

set the tail’s next pointer to n

set tail equal to n

add one to size

return true

**Update**

bool Map::update(const KeyType& key, const ValueType& value)

if empty:

return false

if it doesn’t contain the key:

return false

find the node containing the key

set the node’s value equal to value and return true

**Insert or Update**

bool Map::insertOrUpdate(const KeyType& key, const ValueType& value)

determine if its in there or not

use insert or update accordingly

return true

**Erase**

bool Map::erase(const KeyType& key) {

if the key isn’t in there:

return false

find the node with the key

if the list’s size is one:

set head and tail to null

if the node we find is at the head:

set head to the next one

set the next one’s previous pointer to null

if it’s at the tail:

set the tail to the previous node

set the tail’s next pointer to null

otherwise:

set the next node’s previous pointer to the previous node

set the previous node’s next pointer to the next node

set the pointer to null and delete the node

increment the size by one

return true

**Contains**

bool Map::contains(const KeyType& key) const {

if the list is empty:

return false

create a temp node pointing to the head

while the node isn’t null:

if the temp node’s key is equal to what we’re looking for:

return true

set temp to the next node

return false if it gets to this point

**Get without int i**

bool Map::get(const KeyType& key, ValueType& value) const {

if it doesn’t contain key:

return false

find the node with key

set it’s value to value

return true

**Get with i**

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

if i is out of bounds:

return false

if the list is empty:

return false

find the “i”’th node

set their parameters equal to the node’s values

return true

**Swap**

void Map::swap(Map& other) {

set their heads and tails equal to the other ones’

set their sizes equal to each other’s

**Combine**

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

copy m1 and m2 into new lists

make a bool true

erase all the data currently inside result

go through m1’s copy:

if m4 contains the same key:

if the value’s aren’t the same

make the bool false

otherwise:

insert the key value pair into result

if m4 doesn’t contain the same key:

insert the key value pair into result

do all the above to m2’s copy

return the bool

**Subtract**

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

create copies of m1 and m2

clear result

go through m1:

if m4 doesn’t contain the key:

insert the key and value pair into result

**Test Cases**

The tests were performed on a map from strings to doubles:

//Default Constructors

Map a;

Map b;

KeyType k[6] = {"brad", "ray", "nathan", "andrew", "yif", "david"};

ValueType v = 7.1;

//Empty, insert, and size

assert(a.empty()); //It should be empty.

assert(a.size()==0); //And size should be zero.

for (int n = 0; n < 5; n++)

assert(b.insert(k[n], v)); //Since b is empty, every insert should return true.

assert(!b.insert(k[4], v)); //Since “yif” is already in there, it should return false.

assert(b.size()==5); //Should have 5 things in there.

assert(!b.empty()); //Should return false since it’s not empty.

//Swap

a.swap(b); //Swaps them

assert(a.size()==5); //B used to have size 5, so now a does

assert(b.size()==0); //Makes sure a and b swapped sizes

assert(!a.insert(k[4], v)); //a should already have it so it should return false

assert(b.insert(k[4], v)); //b shouldn’t have it anymore; returns true

a.swap(b); //swaps them back (now both have stuff)

assert(a.size()==1); //a should now return size 1

assert(b.size()==5); //b should now return size 5

assert(!a.insert(k[4], v)); //a should already have it; returns false

assert(a.insert(k[3], v)); //should insert and return true

assert(!b.insert(k[4], v)); //should already have it; return false

//Copy constructor

Map c(a); //Test copy constructor

assert(!c.empty()); //c shouldn’t be empty anymore

assert(c.size()==2); //c should have gotten a’s size

assert(!c.insert(k[4], v)); //c should have it, returns false

assert(!c.insert(k[3], v)); //see above

//Equal operation

b = c; //Tests equal operator

assert(b.size()==c.size()); //They should have same size

assert(!b.insert(k[4], v)); //Shouldn’t be able to insert it

Map d;

b = d; //tests equal operator with an empty list

assert(b.empty() && b.size()==0); //makes sure it made b empty

assert(b.insert(k[4], v)); //b should now be able to insert it, returns true

//Update and InsertorUpdate

assert(!b.update(k[3], v)); //can’t update because it’s already there

assert(a.update(k[3], v)); //can update cause it’s there, returns true

assert(b.insertOrUpdate(k[4], v)); //always returns true

assert(a.insertOrUpdate(k[3], v)); //see above

//Erase

assert(a.erase(k[3])); //Since it contains it, erasing it should be true

assert(!a.erase(k[3])); //doesn’t have it anymore, returns false

assert(!d.erase(k[1])); //tests erase on an empty list

assert(d.insert(k[2], v)); //tests insert on an empty list

assert(b.erase(k[4])); //erases something in b emptying it

for (int n = 0; n < 5; n++)

assert(b.insert(k[n], v)); //refills b

//Contains

assert(!a.contains(k[3])); //tests contains, a doesn’t have it; returns false

assert(b.contains(k[1])); //b has it, returns true

//Get

ValueType v1[6] = {1.1,2.2,3.3,4.4,5.5,6.6};

KeyType k1 = k[1];

assert(b.get(k1, v) && v == 7.1); //should return true, and set v to 7.1

v = 5;

assert(!d.get(k1, v) && v == 5); //should return false and not touch v

for (int n = 0; n < 5;n ++)

assert(b.insertOrUpdate(k[n], v1[n])); //resets first 5 stuff of b

//Get

for (int i = 0; i < 5; i++) {

assert(b.get(i, k[i], v) && v == v1[i]); //tests that get with number works

}

assert(!d.get(3, k1, v)); //d doesn’t have 3 stuff, so it should return false

Map\* e = new Map;

for (int i = 0; i < 5; i++) {

assert(e->insert(k[i], v1[i])); //sets e to what i want

}

//Destructor

delete e; //destructs e

//Combine

Map r;

Map one;

Map two;

for (int i = 0; i < 5; i++) {

assert(r.insert(k[i], v1[i])); //sets r to what i want, making sure get can handle stuff //already in there

}

KeyType w[] = {"Fred", "Ethel", "Lucy"};

ValueType x[] = {123, 456, 789};

KeyType y[] = {"Lucy", "Ricky"};

ValueType z[] = {789, 321};

for (int i = 0; i < 3; i++) {

assert(one.insert(w[i], x[i])); //puts stuff in one

}

for (int i = 0; i < 2; i++) {

assert(two.insert(y[i], z[i])); //puts stuff in two

}

assert(combine(one, two, r)); //tests combine, returns true

assert(r.size()==4); //r should have 4 things

KeyType ans[] = {"Fred", "Ricky", "Ethel", "Lucy"};

for (int i = 0; i < 4; i++) {

assert(r.erase(ans[i])); //should return true cause r should have those 4 keys

}

assert(two.update("Lucy", 654)); //changes the value of Lucy

assert(!combine(one, two, r)); //return should now return false

assert(r.size()==3); //return should be of size 3

for (int i = 0; i < 3; i++) {

assert(r.erase(ans[i])); //making sure r has the correct answer

}

//Subtract

assert(two.update("Lucy", 789)); //sets two to what i want

assert(two.insert("Ethel", 654)); //see above

subtract(one, two, r); //tests subtract

assert(r.size()==1); //makes sure r is the correct size

assert(r.erase("Fred")); //makes sure r is what i want it to be

subtract(one, one, r); //tests it on two lists that are identical

assert(r.size()==0); //should make r not have anything

}