1. My doubly-linked list has both previous and next pointers, as all doubly-linked lists should. It is not circular, so my head prev pointer is a null pointer and the last Node’s next pointer is also a null pointer. No dummy node is present, as I use the lack of a head in order to check if the list is empty in several functions, and each node contains a KeyType, ValueType, and two pointers to nodes. Those pointers are the next and prev pointers. My nodes are in no particular order, and any new node is put at the back of the list, so the most recent data inserted would be the last to be accessed.
2. **Map::Map**

If old map is empty:

Make new map empty

Otherwise

Create a head node for new Map

Insert correct values

Repeatedly:

Create new nodes for Map

And insert correct values from old

If no remaining Nodes

Break;

**Map::~Map**

If Map is not empty

Repeatedly:

Move forward in list

And delete prior Node

If at last Node

Break;

Delete last node

**Map::operator=**

If both maps are the same

Do nothing and return

If map to be modified is not empty

Repeatedly:

Move forward in list

And delete prior Node

If at last node

Break;

Delete last node

If Map to be copied from is empty

Make modified map empty

Otherwise

Create a head node for new Map

Insert correct values

Repeatedly:

Create new nodes for Map

And insert correct values from src

If no remaining Nodes

Break;

Return \*this;

**Map::insert**

If head is empty

Dynamically allocate a new Node for head

And insert key and null pointer values

Then return true;

Repeatedly:

If key at current node is the same as given key

Return false;

Proceed to next node;

If no remaining next nodes

Break;

If last node’s key is NOT same as given key

Dynamically allocate a new Node at end of list

Link it to prior last Node

And insert correct value and pointers

Then return true;

Otherwise return false;

**Map::update**

If Map is empty/ no values to update

Return false;

If key of head node is same as key

Modify it

And return true;

Repeatedly:

If key of current node is equal to key

Modify it

And return true

Proceed to next node

If at last node

Break;

Return false;

**Map::insertOrUpdate**

If Map is empty

Create new node

Input key and value

Return true;

Repeatedly:

If key of current node is equal to key

Modify it

And return true

Proceed to next node

If at last node

Break;

If last key is not same as given key

Create new node at end of list

Input values

Link it appropriately

Return true;

Otherwise

Update last node’s value

Return true;

**Map::erase**

If Map is empty

Return false;

If head node’s key is equal to key

If there is more than one Node

Save position of head

Make head point to next Node

Delete prior head

Modify pointer of new head

Return true;

Otherwise

Delete head Node

Head pointer is null

Return true;

otherwise repeatedly

If current node’s key is equal to key

If current node is last Node

Modify prior Node’s pointer

Otherwise

Modify prior and next node’s pointers

Delete the node

Return true;

Proceed to next Node

If at last node

Break;

Return false;

**Map::contains**

If Map holds no values

Return false;

Otherwise repeatedly

If current node’s key is equal to key

Return true;

Proceed to next Node

If no more nodes

Break;

Return false;

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

If map holds no values

Return false;

Repeatedly:

If current node’s key is equal to key

Set Node’s value to value

And return true;

Proceed to next node

If no more Nodes

Break;

Return false;

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

If i is not within bounds or Map is empty

Return false;

Repeatedly:

Proceed to next Node

Once l reaches value of i

Break;

Change key and value to current Node’s key and value

Return true;

**Map::swap**

Hold pointer to head pointer in temp

Modify head pointer to other’s head pointer

Modify other’s head pointer to temp

**Combine**

Repeatedly:

Get value and key of Node in m1

Repeatedly:

Get value and key of Node in m2

If keys are the same but values aren’t

Set check to false

And break out of current loop

If keys and values are the same

Insert them into result

Break out of current loop

If keys are not the same

Insert key and value of m1 into result

Once i value has reached size of m1

Break;

Repeatedly:

Get value and key of Node in m2

Repeatedly:

Get value and key of Node in m1

If keys are the same but values aren’t

Set check to false

And break out of current loop

If keys and values are the same

Insert them into result

Break out of current loop

If keys are not the same

Insert key and value of m2 node into result

Once i value has reached size of m2

Break;

Return check;

**Reassign**

Make size of result equal to size of m

If size of m is either 0 or 1

Make no change and return

Repeatedly:

If at last Node of m

Give first Node value of last Node

Otherwise

Give next Node value of current Node

1. **Test Cases:**

**Copy constructor:**

Map a; //test if copy constructor works with no pairs

Map b = a;

assert(a.size() == 0 && b.size() == 0); // test size function

Map c; //check to see if correctly copies over one pair

c.insert("scooby", 123); //test insert function

Map d = c;

assert(c.size() == d.size());

KeyType tempk = "scooby";

ValueType tempval;

d.get(tempk, tempval); //test first get function

assert("scooby" == tempk && 123 == tempval);

Map e; //check if works with more than 3 nodes

e.insert("random", 0);

e.insert("yeet", 69);

e.insert("yote", 96);

e.insert("linked", 420);

Map f = e;

ValueType check;

f.get("yote", check);

assert(e.size() == f.size() && check == 96 && f.contains("random") && f.contains("yeet") && f.contains("linked")); //test contains function

**Assignment Operator:**

Map ab; //test with no Nodes, check size and contains functions

Map bb;

bb.insert("scoob", 64);

bb = ab;

assert(bb.size() == 0 && !(bb.contains("scoob")));

Map cb; //test with one Node, check size and contain

Map db;

cb.insert("bees", 987);

db.insert("buffalo", 1988);

db.insert("yahtzee", 26);

db = cb;

assert(db.size() == 1 && db.contains("bees") && !(db.contains("buffalo")) && !(db.contains("yahtzee")));

Map eb; //assignment operator test wth 3+ nodes, check size and contain

Map fb;

eb.insert("how", 90);

eb.insert("bees", 987);

eb.insert("buffalo", 1988);

eb.insert("yahtzee", 26);

fb = eb;

assert(fb.size() == 4 && fb.contains("yahtzee"));

**Empty():**

Map ac; //test an empty Map

assert(ac.empty());

Map bc; //test non empty Map for correct empty classification

bc.insert("wowza", 888);

assert(!bc.empty());

Map cc; //make sure classification changes when filled

assert(cc.empty());

cc.insert("phone", 811);

assert(!cc.empty());

Map dc; //make sure classification changes when emptied, check erase

dc.insert("johnathan", 1);

assert(!dc.empty());

dc.erase("johnathan");

assert(dc.empty());

**Update:**

Map ad;

assert(!ad.update("yote", 99)); //return false for empty Map

ad.insert("touchpad", 220);

assert(ad.update("touchpad", 12)); //returns true

ValueType temp;

ad.get("touchpad", temp);

assert(temp == 12 && ad.contains("touchpad"));

Map bd; //update works for multiple Nodes

bd.insert("dinner", 5);

bd.insert("work", 99999);

bd.insert("taste", 0);

bd.insert("bored", 8008135);

assert(bd.update("taste", 100));

ValueType temp1;

bd.get("taste", temp1);

assert(temp1 == 100);

assert(bd.update("bored", 0));

bd.get("bored", temp1);

assert(temp1 == 0);

**InsertOrUpdate:**

Map ae; //check insert function of insertOrUpdate

assert(ae.insertOrUpdate("like", 777));

assert(ae.size() == 1 && ae.contains("like"));

assert(ae.insertOrUpdate("like", 666));

ValueType checker;

ae.get("like", checker);

assert(checker == 666 && ae.size() == 1);

assert(ae.insertOrUpdate("yowza", 134));

assert(ae.size() == 2 && ae.contains("yowza"));

assert(ae.insertOrUpdate("zoinks", 765));

assert(ae.size() == 3 && ae.contains("zoinks"));

assert(ae.insertOrUpdate("zoinks", 55));

ae.get("zoinks", checker);

assert(ae.size() == 3 && checker == 55);

**Erase:**

Map af;

assert(!af.erase("empty")); //nothing to erase, false

af.insert("empty", 0);

af.insert("you", 456);

af.insert("buzz", 990);

af.insert("yay", 101);

af.erase("yay"); //removal of last node

assert(af.size() == 3 && !(af.contains("yay"))); //make sure erase totally removed the Node

af.erase("you"); //removal of middle node

assert(af.size() == 2 && !(af.contains("you"))); //make sure actually erased

af.erase("empty"); //removal of first Node

assert(af.size() == 1 && !(af.contains("empty"))); //make sure actually erased

**Second Get:**

Map ag; //test second get

KeyType keytemp;

ValueType valtemp;

assert(!ag.get(0, keytemp, valtemp)); //false if there is nothing on Map to get

ag.insert("bruv", 2019);

assert(!ag.get(1, keytemp, valtemp)); //false as 1 is too large, it is out of bounds for what i should be

ag.get(0, keytemp, valtemp);

assert(keytemp == "bruv" && valtemp == 2019); //since only one pair, keytemp and valtemp should be equal

ag.insert("aaah", 2);

assert(ag.get(1, keytemp, valtemp)); //true as 1 is now within bounds

**Swap:**

Map ah, bh; //test swap function

ah.insert("broken", 78);

ah.insert("alone again", 5);

bh.insert("turn around", 10);

ah.swap(bh);

assert(ah.contains("turn around") && bh.contains("broken") && !(ah.contains("broken")) && bh.size() == 2); //make sure swap was successful

**Reassign:**

Map ai, bi; //test reassign function

ai.insert("Fred", 123);

ai.insert("Ethel", 456);

ai.insert("Lucy", 789);

ai.insert("Ricky", 321);

reassign(ai, bi);

ValueType tempval;

bi.get("Fred", tempval); //make sure no pair should be the same as before

assert(tempval != 123);

bi.get("Ethel", tempval);

assert(tempval != 456);

bi.get("Lucy", tempval);

assert(tempval != 789);

bi.get("Ricky", tempval);

assert(tempval != 321);

reassign(ai, ai); //test works if result and m are the same Map

ai.get("Fred", tempval);

assert(tempval != 123);

ai.get("Ethel", tempval);

assert(tempval != 456);

ai.get("Lucy", tempval);

assert(tempval != 789);

ai.get("Ricky", tempval);

assert(tempval != 321);

**Combine:**

Map aj, bj, cj; //test combine function

aj.insert("Fred", 123);

aj.insert("Ethel", 456);

aj.insert("Lucy", 789);

bj.insert("Lucy", 789);

bj.insert("Ricky", 321);

assert(combine(aj, bj, cj));

assert(cj.size() == 4 && cj.contains("Fred") && cj.contains("Ricky"));

assert(combine(aj, aj, aj)); //test if same is being sent into the function

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