Valerie So UID 505970340

**Design Description**

This Map is implemented as a doubly-linked list consisting of Pair nodes, Pair is a privately defined struct consisting of: a KeyType key, a Va;ueType value, a pointer to the previous node and a pointer to the next node. I did not implement the list as circular. The nodes are in the order you add them, and when the nodes are added, they will be added to the beginning. There is no implemented dummy node.

**Pseudocode**

bool insert(const KeyType& key, const ValueType& value) {

if key in map:

return false

create pointer p to new Pair

assign key and value

set previous pointer to null and next pointer to current head

if head is not null, set head->prev to p

move head

increase size

return true

}

bool update(const KeyType& key, const ValueType& value) {

set pointer p to head

if p is null:

return false

if key not in map:

return false

loop through list until key is found

set p->value to updated value

return true

}

bool insertOrUpdate(const KeyType& key, const ValueType& value) {

if cannot update: // call already implemented update function

insert // call already implemented insert function

return true

}

bool erase(const KeyType& key) {

if key not in map:

return false

set pointer p to head

loop through pairs to find key

if p->prev is not null:

set temporary pointer to p->prev

set next of temp pointer to p->next

otherwise:

set head to next

if next is not null

set temp pointer to next

set temp->prev to p->prev

delete Pair

decrease size

}

bool contains(const KeyType& key) const {

set pointer p to head

loop through pairs until you reach the end or you find key

return (p != null)

}

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

set pointer p to head

if key not in map:

return false

loop through map until key found

set value to correct value

return true

}

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

check if i in bounds

set pointer p to head

loop through map:

if isGreaterThanX(p->key) is correct value:

set key and value appropriately

break

return true

}

void swap(Map& other) {

set temporary pointer to head

swap head with other.head

set temp to size

swap size with other size

}

Map(const Map& m) {

create pointers p, next, and prev

if m head is null:

just set head to null

else:

set head to new Pair

set head key and value

head prev = null

set p to head

move to next Pair

repeatedly:

set p->next to new Pair

initialize variables properly

set prev to p

set next to next node

if p is not null:

set next to null

set the sizes equal

}

Map& operator= (const Map& m) {

if it is already equal:

return \*this

eraseAll()

call copy constructor for temp map

swap with temp

return \*this

}

~Map() {

call eraseAll function

}

int isGreaterThanX(const KeyType& key) const {

int count = 0

set pointer p to head

loop through pairs:

if key > p->key:

increase count

return count

}

void eraseAll() {

set temp pointer to head

loop through pairs:

set temp2 to next pointer

delete temp

set temp to temp2

}

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

declare KeyType key and ValueType value, value2

create copy of m1 in temp

declare bool returnvalue as true

loop through m2:

get key/values from m2

if key not in temp:

insert

else:

get key, value2 from temp

if value is the same as value2:

erase key from temp

set return value to false

swap return with temp

return returnvalue

}

void reassign(const Map& m, Map& result) {

create copy of m

declare KeyType key, key2 and ValueType value, value2

loop from 0 to 1 less than m size:

get the key and value from the get(i, key, value) function

do the same in key2 value2 using i+1

update the key to pair with value2

get the first value

update the last key to value

swap result with temp

}

**Test Cases**

Map m;

// For an empty map:

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

assert(m.empty()); // test empty

assert(!m.erase("Ricky")); // nothing to erase

assert(m.insert("Ricky", 5)); // insert a node Ricky

assert(m.erase("Ricky")); // can now erase Ricky

m.insert("Jennifer", 69); // insert more nodes

m.insert("Gertrude", 420);

m.insert("Janice", 19);

assert(!m.insert("Gertrude", 666)); // cannot insert with same key

assert(!m.update("Francesca", 3.14)); // cannot update when the key is not already there

assert(m.update("Gertrude", 101)); // update succeeds

assert(m.insertOrUpdate("Victoria", 2.718)); // insertOrUpdate succeeds if key not there

assert(m.insertOrUpdate("Janice", 20)); // insertOrUpdate succeeds if key exists already

assert(m.contains("Janice")); // check if contains works

assert(!m.contains("Judy")); // check if contains returns false when key is not there

KeyType key;

ValueType value;

assert(m.get("Janice", value) && value == 20); // check to make sure get returns correct value if key exists

assert(!m.get("Jisoo", value)); // check to make sure get returns false if key not there

assert(!m.get(-1, key, value)); // get should return false if i is negative

assert(!m.get(69, key, value)); // get should return false if i >= size()

assert(m.get(2, key, value)); // get should return the correct value

Map m2; // make a second map

m2.insert("Lucifer", 666); // add some nodes to the second map

m2.insert("Euler", 2.718);

m2.insert("Avogadro", 6.022);

m.swap(m2); // swap the two maps

assert(m2.size() == 4); // check the size of m2

assert(m.size() == 3); // check the size of m

assert(m.contains("Euler")); // make sure m contains original m2 nodes

assert(!m.contains("Janice")); // make sure m does not contain its original nodes

assert(m2.contains("Janice")); // make sure m2 contains nodes from m

Map result;

assert(merge(m, m2, result)); // merge should return true because there are no identical keys with different values

assert(result.contains("Janice")); // result should have values from m2

assert(result.contains("Euler")); // result should have values from m

assert(merge(m, m2, m)); // merge should still work if the passed in parameters are the same

assert(m.contains("Janice")); // m should have values from m2

assert(m.contains("Euler")); // m should also have values from m before

assert(merge(m, m, m2)); // test if the two merged Maps are the same it still works

assert(m2.contains("Euler")); // test if m2 has values from m

m2.update("Jennifer", 777); // Jennifer won the lottery

assert(!merge(m, m2, result)); // this should return false because m and m2 both have Jennifer but with different values

reassign(m, result); // reassign values of m

assert(result.get("Janice", value) && value != 20); // check to make sure the value is different

reassign(m, m); // reassign when result Map is the same as the original map

assert(m.get("Janice", value) && value != 20); // make sure it is changed