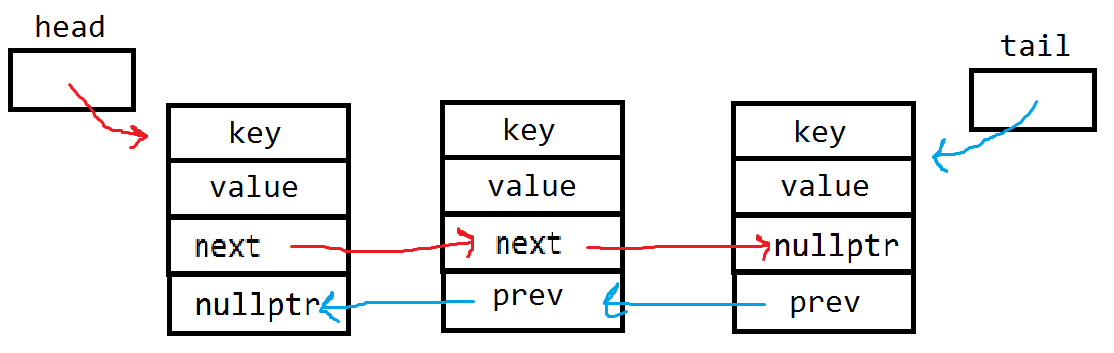
Elena Escalas 7045606967

Project 2 Report

Design description:

My list has next and previous pointers as elements of each node, as well as a head and tail pointer pointing to the first and last elements of the list. Whenever a new node is added, it is added to the front of the list, therefore the order of the nodes within the map are based on the order of insertion.

Here is a drawing describing the organization of my nodes within my linked list:



Pseudocode:

Map::Map(const Map &old) // copy constructor

repeatedly:

get values from the old Map's node key/value pairs

and insertOrUpdate them into "this" Map's node key/value pairs

Map &Map::operator= (const Map &src) // assignment operator

make a copy of src titled old (so you can use same loop as copy constructor)

repeatedly:

get value from "this'" Map to erase

repeatedly:

get values from the src Map's node key/value pairs

and insertOrUpdate them into "this" Map's node key/value pairs

Map::insert

if the map is empty:

create a new node with the reference key and value

set the previous pointer of the new node to nullptr

set the next pointer of the new node to head

return true

else:

create a new node with the reference key and value

set the next pointer of the new node to head

set the previous pointer of the new node to nullptr

set the node head points to's previous pointer to point to new node

set the head to point to the new node

return true

Map::update

traverse through the nodes until the desired key is found

once found, change the value of the node to the referenced value

Map::erase

if the head equals nullptr (the map is empty):

return false

if the head points to the node with the desired value for the node to delete

delete the first node in the map

return true

traverse through the nodes:

if you have not reached the end of the linked list and the next node has the correct value, break out of the loop:

if traversal reached the end of the linked list and hasn't found the desired key: return false

else:

create a temporary killMe node pointing to the node to erase

set node to erase's next and prev pointers to killMe's next and prev

pointers

delete killMe and return true

bool combine

create a boolean NoDuplicate representing whether or not there is a duplicate key/value pair in both m1 and m2

create a copy of m1 and m2 maps

compare each key in m1 and m2 to see if they are equal:

if they are equal and their values are equal:

insert the key as a new key/value pair,

if it already exists from a different loop iteration, update it

else if they are equal and their values are NOT equal:

erase the key's node from the list

make NoDuplicate false

else if you've reached the end of the loop and the key has no match:

insert or update the key/value pair

compare each key in m1 and m2 to see if they are equal to get rid of extras:

if they are equal and their values are NOT equal:

erase the key's node from the list

make NoDuplicate false

return NoDuplicate;

void subtract

create a copy of m1 and m2

delete the contents of result

create a new empty result map

repeatedly:

insert or update m1's key/value pairs into result

until all of m1's key/value pairs are added once

compare each key in m1 and m2 to see if they are equal:

if they are equal, erase that key from result

Test Cases:

KeyType name, her;

ValueType herTime, time;

Map m, m2; // test default constructor

// for an empty map

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

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

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

// filling empty map

m.insert("Elena", 103.12);

m.insert("Ella", 102.46);

m.insert("Alex", 101.1);

m.insert("Allie", 100.3); // test insert

assert(m.contains("Allie")); // test contains

assert(!m.contains("Suzanna"));

assert(m.get("Alex", herTime)); // test get

assert(herTime == 101.1);

assert(m.insertOrUpdate("Julia Jane", 101.4)); // test update of insertOrUpdate

assert(m.get("Julia Jane", herTime));

assert(herTime == 101.4);

assert(m.insertOrUpdate("Suzanna", 103.3)); // test insert of insertOrUpdate

assert(m.contains("Suzanna"));

assert(m.update("Allie", 59.99)); // test update

// comparing and swapping maps

m2.insert("Rafa", 1452);

assert(m2.size() == 1);

Map m3 = m; // test copy constructor

assert(m3.size() == 6);

assert(m3.get("Alex", herTime));

assert(herTime == 101.1);

assert(m3.contains("Ella"));

assert(m3.erase("Elena")); // test erase

assert(!m3.contains("Elena"));

assert(!m3.erase("Suzanne"));

assert(m3.erase("Suzanna"));

Map m4 = m3;

assert(m3.size() == 4);

m3.get(1, name, herTime); // test integer get function

m4.get(1, her, time);

assert(name == her && herTime == time);

Map oneNode; // test erasing a one node map

assert(oneNode.insert("Marina", 110));

assert(oneNode.erase("Marina"));

assert(!oneNode.size() > 0);

assert(oneNode.empty());

m2 = m3; // test assignment operator

assert(m2.size() == 4);

assert(m2.get("Alex", herTime));

assert(m2.contains("Ella"));

m2.swap(m); // test the swap function

assert(m.size() == 4);

assert(m2.size() == 6);

assert(m2.get("Alex", herTime));

assert(m2.contains("Ella"));

assert(m2.erase("Allie"));

assert(m2.size() == 5);

m2.swap(m); // test the swap function

assert(m.size() == 4);

assert(m2.size() == 6);

assert(m2.get("Alex", herTime));

assert(m2.contains("Ella"));

assert(m2.erase("Allie"));

assert(m2.size() == 5);

Map result; // test combine function

assert(combine(m, m2, result));

assert(result.size() == 6); // result contains each unique element only once

assert(m2.insertOrUpdate("Ella", 100.5));

assert(!combine(m, m2, result));

assert(!result.contains("Ella")); // does not contain the key

// in which the values are not equal

assert(result.size() == 5); // result contains each unique element only once

assert(result.contains("Allie")); // result contains element only from m

assert(result.contains("Julia Jane")); // result contains element only from m2

assert(!combine(m, m2, m)); // make sure this unique case still compiles and properly sets m to result

assert(m.size() == 5); // m should be the same size as result now

assert(m.contains("Julia Jane")); // m now contains element from m2, because it is the result of the combination of the two

subtract(m, m3, result); // test subtract function

assert(!result.contains("Ella")); // the subtraction does not include the element in m3 that is not in m

assert(result.contains("Elena") && result.contains("Suzanna"));

// the subtraction successfully eliminated all the values m3 contains that are in m

assert(result.size() == 2);

subtract(m, m2, m); // make sure this unique case compiles and properly sets m to result

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

assert(m.contains("Allie"));