Mudith Mallajosyula -- ID# 404937201 -- Discussion 2D

Project 2

Report

1. **Doubly Linked List Implementation**

I modified the Pair struct to include two more data members, a pointer to the previous

pair and a pointer to the next. I also changed the data member for the array to a pointer

to the first element of the linked list. This defaulted to a null pointer.

Typical Map setup:

m\_size = 4;

m\_head points to 800;

Linked List of Pairs:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Address: | 800 (m\_head) | 1000 | 1200 | 1400 |
| prev | null | 800 | 1000 | 1200 |
| m\_key | “Hello” | “Hi” | “Hey” | “Hiya” |
| m\_value | 143 | 253 | 263 | 654 |
| next | 1000 | 1200 | 1400 | null |

Empty Map:

m\_size = 0;

m\_head = nullptr;

**2. Non-Trivial Algorithms**

**copy constructor**

**Map& Map::Map(const Map& mp)**

if size is neg

head is nullptr

return

repeatedly

set next to null

set prev to previous

set key to other key

set value to other value

if next not null

next is new pair

set current to next

**destructor**

**Map::~Map()**

create pointer next to head

create temp pointer

repeatedly

set temp to next

set next to next->next

delete temp

**bool Map::erase()**

if head contains to key

set next to head

delete head

decrement size

return true

repeatedly

if current contains key

set next of previous to next

set previous of next to previous

delete current

decrement size

return true

return false

**bool Map::doInsertOrUpdate(const KeyType& key, const ValueType& value, bool mayInsert, bool mayUpdate)**

if head is nullptr

create new pair

set value to argument

set key to argument

set next to nullptr

increment size

set head to new pair

return true

if key is in list

if mayUpdate is true

repeatedly

find next link if key is not key

update key

update value

return mayUpdate

if mayInsert is false

return false

repeatedly

if curr pair is not nullptr

go to next pair

next is new pair

previous of pair is curr

key of pair is key

value of pair is value

next of curr is null

increment size

return true

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

if m1 is m2

set result to m1

return true

repeatedly

get key and value of m1

if key of m1 is in m2

if value in m1 is value in m2

add key and value to result

else

set output to false

else

add key and value to result

return output

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

if m1 is m2

set result to empty map

return;

repeatedly

get key and value of m1

if key not in m2

add key and value to result

**3. Test Cases**

All tests were performed on a map from strings to doubles.

/\* For an empty map: \*/

// default constructor

Map m;

// member functions

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

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

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

assert(m.insert("hello", 1)); // test insert

assert(m.erase("hello")); // test erase

assert( ! m.update("hello", 4));// test update

ValueType value;

assert( ! m.get("hello", value)); // test get

assert(m.insertOrUpdate("hello", 56)); // test insertOrUpdate

assert(m.insertOrUpdate("hello", 57)); // test insertOrUpdate

assert(m.get("hello", value) && value == 57.0); // test insertOrUpdate

assert(m.erase("hello")); // test erase

// Copy and Assignment

Map mp = m;

assert(mp.size() == m.size()); // test copy constructor

m.insert("Test", 55);

int sz = m.size();

mp.swap(m); // test swap function

assert(mp.size() == sz); // test swap function

assert(mp.get("Test", value) && value == 55); // test swap function

m = mp;

assert(mp.size() == m.size()); // test assignment operator

assert(m.size() == sz); // test assignment operator

assert(m.erase("Test"));

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

// Algorithm Functions

Map out;

assert(combine(m, mp, out)); // test combine

assert(out.size() == m.size() + mp.size()); // test subtract

subtract(m, mp, out);

assert(out.size() == m.size()); // test subtract

// aliasing

assert(combine(m, m, m)); // test combine w/ aliasing

subtract(mp, mp, mp);

assert(mp.size() == 0); // test subtract w/ aliasing

cerr << "Empty Map tests passed" << endl;

Map d; // emptying lists

m = d;

/\* For an one pair map: \*/

// member functions

assert(m.insert("a", 1));

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

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

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

assert(m.insert("hello", 2)); // test insert

assert(m.erase("hello")); // test erase

assert( ! m.update("hello", 4));// test update

assert( ! m.get("hello", value)); // test get

assert(m.insertOrUpdate("hello", 56)); // test insertOrUpdate

assert(m.insertOrUpdate("hello", 57)); // test insertOrUpdate

assert(m.get("hello", value) && value == 57.0); // test insertOrUpdate

assert(m.erase("hello")); // test erase

// Copy and Assignment

mp = m;

assert(mp.size() == m.size()); // test copy constructor

m.insert("Test", 55);

sz = m.size();

mp.swap(m); // test swap function

assert(mp.size() == sz); // test swap function

assert(mp.get("Test", value) && value == 55); // test swap function

m = mp;

assert(mp.size() == m.size()); // test assignment operator

assert(m.size() == sz); // test assignment operator

assert(m.erase("Test"));

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

// Algorithm Functions

m = d; // emptying lists

mp = d;

m.insert("thing1", 345);

m.insert("thing2", 346);

m.insert("thing3", 347);

mp.insert("thing4", 348);

mp.insert("thing5", 349);

mp.insert("thing6", 350);

assert(combine(m, mp, out)); // test combine

assert(out.size() == m.size() + mp.size()); // test subtract

subtract(m, mp, out);

assert(out.size() == m.size()); // test subtract

// aliasing

assert(combine(m, m, m)); // test combine w/ aliasing

subtract(mp, mp, mp);

assert(mp.size() == 0); // test subtract w/ aliasing

cerr << "Single Pair Map tests passed" << endl;

/\* For a multiple pair map: \*/

m = d; // emptying lists

mp = d;

// member functions

assert(m.insert("a", 1));

assert(m.insert("b", 2));

assert(m.insert("c", 3));

assert(m.size() == 3); // test size

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

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

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

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

assert(m.insert("hello", 2)); // test insert

assert(m.erase("hello")); // test erase

assert( ! m.update("hello", 4));// test update

assert( ! m.get("hello", value)); // test get

assert(m.insertOrUpdate("hello", 56)); // test insertOrUpdate

assert(m.insertOrUpdate("hello", 57)); // test insertOrUpdate

assert(m.get("hello", value) && value == 57.0); // test insertOrUpdate

assert(m.erase("hello")); // test erase

// Copy and Assignment

mp = m;

assert(mp.size() == m.size()); // test copy constructor

m.insert("Test", 55);

sz = m.size();

mp.swap(m); // test swap function

assert(mp.size() == sz); // test swap function

assert(mp.get("Test", value) && value == 55); // test swap function

m = mp;

assert(mp.size() == m.size()); // test assignment operator

assert(m.size() == sz); // test assignment operator

assert(m.erase("Test"));

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

// Algorithm Functions

m = d;

mp = d;

m.insert("thing1", 345);

m.insert("thing2", 346);

m.insert("thing3", 347);

mp.insert("thing4", 348);

mp.insert("thing5", 349);

mp.insert("thing6", 350);

assert(combine(m, mp, out)); // test combine

assert(out.size() == m.size() + mp.size()); // test subtract

subtract(m, mp, out);

assert(out.size() == m.size()); // test subtract

// aliasing

assert(combine(m, m, m)); // test combine w/ aliasing

subtract(mp, mp, mp);

assert(mp.size() == 0); // test subtract w/ aliasing

cerr << "Multiple Pair Map tests passed" << endl;