Rodger Lee

Com Sci 32

Professor Smallberg

Lecture 2, Discussion 2E

UID:204914719

Project 2 Report

Design Description

This project requires us to implement a doubly-linked list that holds a struct of objects: a pair of key and value. The linked list is not circular and does not contain a dummy node, Instead of a dummy node, it contains a head pointer that points to the first node of the list. If the list is empty, the head pointer points to nullptr. The nodes are not sorted in any specific way, and each new node is inserted or stacked on the top or front. Each node contains a KeyType variable, ValueType variable, next pointer, and previous pointer.

A close up of text on a white background

Description automatically generated

Pseudocode

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

{

If map contains key

Return false;

If empty

{

Create new node;

Update node values and pointers

Change head pointer to current new node;

Increase total size counter;

Return true;

}

If not empty

{

Increase total size counter;

Create new node;

Update node values;

Change next pointer to head, and prev pointer to null;

Change head’s previous pointer to current new node;

Change head pointer to current new node;

Return true;

}

}

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

{

If map contains key return false;

if erasing first node

{

create node that points to head and change pointer to point to next;

update next’s previous pointer to null and delete current node;

decrement size;

return true;

}

Otherwise

{

Create new node equal to head

Increment until and stop at the node before our deserved node

Create node for erasure and point to the next node;

Update previous and next nodes’ pointers;

Delete node;

Decrement size;

Return true;

}

}

**void** Map::swap(Map& other)

{

Check for aliasing

Create temp pointers and swap head pointers

Swap sizes

}

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

{

Create temp map and temp KeyType and ValueType values

Increment through first map m1 and copy into tempmap

Increment through second map m2 and check if values already exist in temp map. If the value of the existing key equals the key we’re checking

Copy that value into tempmap, otherwise return false and delete that existing key in tempmap

Set result equal to tempmap and return true;

}

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

{

Create temp map and temp Valuetype since we’re only changing values of

The keyvalue pairs

Increment through map nodes and get each key and the next nodes value and Insert into tempmap.

When count reaches last node, copy the value of the first node into the last node of tempmap.

Set result equal to tempmap;

}

Test Cases

#include "Map.h"

#include <string>

#include <iostream>

#include <cassert>

**using** **namespace** std;

**int** main()

{

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

// default constructor

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.update("Tom", 17.2)); //nothing to update

assert(!m.contains("Tom")); //doesn't contain anything

string s = "unchanged";

**double** d = 10;

assert(!m.get("Ricky", d)); //key isn't equal to any key in the map

assert(!m.get(0, s, d)); // i isn't less than size so should return false and leave variables unchanged

assert(s == "unchanged" && d == 10); //both gets should leave the values unchanged

//copy constructor

Map toBeCopied;

toBeCopied.insert("first", 1);

toBeCopied.insert("second", 2);

toBeCopied.insert("third", 3);

toBeCopied.insert("fourth", 4);

Map copy(toBeCopied);

assert(toBeCopied.size() == copy.size());

assert(copy.contains("first"));

assert(copy.contains("fourth"));

//assignment operator

Map assigned;

assigned = toBeCopied;

assert(toBeCopied.size() == assigned.size());

assert(assigned.contains("first"));

assert(assigned.contains("fourth"));

//insert function (for keys not already in the list)

assert(m.insert("A", 10));

assert(m.insert("B", 20));

assert(m.insert("C", 30));

assert(m.insert("D", 40));

assert(m.insert("E", 50));

assert(m.insert("F", 60));

assert(m.size() == 6); // test size

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

//insert function (for keys matching first item)

assert(!m.insert("A", 10)); //test insert (when already in list/same value)

assert(!m.insert("A", 12)); //test insert (when already in list/different value)

//insert function (for keys matching a middle item)

assert(!m.insert("D", 10)); //test insert (when already in list/same value)

assert(!m.insert("D", 12)); //test insert (when already in list/different value)

//insert function (for keys matching the last item)

assert(!m.insert("F", 10)); //test insert (when already in list/same value)

assert(!m.insert("F", 12)); //test insert (when already in list/different value)

//update function (for first item)

assert(m.update("A", 11)); //test update (when key exists)

assert(m.get("A", d)); //test get when key exists

assert(d == 11); //test that the update function changed the key

//update function (for a middle item)

assert(m.update("C", 31)); //test update (when key exists)

assert(m.get("C", d)); //test get when key exists

assert(d == 31); //test that the update function changed the key

//update function (for the last item)

assert(m.update("F", 61)); //test update (when key exists)

assert(m.get("F", d)); //test get when key exists

assert(d == 61); //test that the update function changed the key

//update function (for a key not equal to any in the map)

assert(!m.update("H", 61)); //testing update when key doesn't exist

//insertOrUpdate function (for first item)

assert(m.insertOrUpdate("A", 12)); //should always return true

assert(m.get("A", d)); //test get when key exists

assert(d == 12); //test that insertOrUpdate changed the key

//insertOrUpdate function (for middle item)

assert(m.insertOrUpdate("C", 32)); //should always return true

assert(m.get("C", d)); //test get when key exists

assert(d == 32); //test that insertOrUpdate changed the key

//insertOrUpdate function (for first item)

assert(m.insertOrUpdate("F", 62)); //should always return true

assert(m.get("F", d)); //test get when key exists

assert(d == 62); //test that insertOrUpdate changed the key

//insertOrUpdate function (for a key not equal to any in the map)

assert(m.insertOrUpdate("G", 70)); //should always true

assert(m.size() == 7); //test updated size

//erase function (for a key not equal to any in the map)

assert(!m.erase("L"));

//erase function (for the first element)

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

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

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

//erase function (for a middle element)

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

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

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

//erase function (for the last element)

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

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

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

//contains function (for a key currently in the map)

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

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

//contains function (for a key not currently in the map)

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

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

//get function with 2 parameters (for a key currently in the map)

string testString = "test";

**double** testDouble = 9.6;

assert(m.get("E", testDouble));

assert(testDouble == 50);

//get function with 2 parameters (for a key not currently in the map)

assert(!m.get("A", testDouble));

assert(testDouble == 50); //should remain unchanged

//get function with 3 parameters (for an invalid i)

assert(!m.get(-1, testString, testDouble)); // i < 0

assert(!m.get(4, testString, testDouble)); // i = size()

assert(!m.get(5, testString, testDouble)); // i > size()

assert(testString == "test" && testDouble == 50); //should remain unchanged

//get function with 3 parameters (for first item)

assert(m.get(0, testString, testDouble));

assert(testString == "F" && testDouble == 62);

//get function with 3 parameters (for a middle item)

assert(m.get(2, testString, testDouble));

assert(testString == "D" && testDouble == 40);

//get function with 3 parameters (for the last item)

assert(m.get(3, testString, testDouble));

assert(testString == "B" && testDouble == 20);

Map a;

a.insertOrUpdate("a", 1);

a.insertOrUpdate("b", 2);

a.insertOrUpdate("c", 3);

a.insertOrUpdate("d", 4);

a.insertOrUpdate("e", 5);

a.insertOrUpdate("f", 6);

a.insertOrUpdate("g", 7);

a.insertOrUpdate("h", 8);

Map b;

b.insertOrUpdate("w", 23);

b.insertOrUpdate("x", 24);

b.insertOrUpdate("y", 25);

b.insertOrUpdate("z", 26);

//swap function (for the same Map (aliasing))

a.swap(a); //a should remain unchanged

assert(a.size() == 8);

assert(a.contains("a"));

assert(a.contains("d"));

assert(a.contains("h"));

//swap function (for two different maps)

a.swap(b);

//check a to see if it contains what b previously did

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

assert(a.contains("w"));

assert(a.contains("x"));

assert(a.contains("y"));

assert(a.contains("z"));

//check b to see if it contains what a previously did

assert(b.size() == 8);

assert(b.contains("a"));

assert(b.contains("b"));

assert(b.contains("c"));

assert(b.contains("d"));

assert(b.contains("e"));

assert(b.contains("f"));

assert(b.contains("g"));

//swap function (when one map goes out of scope)

{

Map c;

c.insertOrUpdate("i", 9);

c.insertOrUpdate("j", 10);

c.insertOrUpdate("k", 11);

c.insertOrUpdate("l", 12);

c.insertOrUpdate("m", 13);

c.swap(a);

//check c to see if it contains what a previously did

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

assert(c.contains("w"));

assert(c.contains("x"));

assert(c.contains("y"));

assert(c.contains("z"));

}

//check a to see if it contains what c previously did

//it should still work even though c is now deleted since it's out of scope

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

assert(a.contains("i"));

assert(a.contains("j"));

assert(a.contains("k"));

assert(a.contains("l"));

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

//copy constructor after swapping with something out of scope

Map n(a);

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

//assignment operator after swapping with something out of scope

Map p;

p = a;

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

//testing non-member functions

Map one;

one.insert("A", 5);

one.insert("B", 10);

one.insert("C", 15);

one.insert("D", 20);

one.insert("E", 25);

one.insert("F", 30);

Map two;

two.insert("G", 35);

two.insert("H", 40);

two.insert("I", 45);

two.insert("J", 50);

two.insert("K", 55);

Map result;

//combine function (with 2 different maps and an empty result)

assert(combine(one, two, result));

assert(result.size() == one.size() + two.size());

assert(result.contains("A"));

assert(result.contains("F"));

assert(result.contains("G"));

assert(result.contains("K"));

//combine function (with 2 different maps and a non-empty result)

assert(combine(one, two, result));

assert(result.size() == one.size() + two.size());

assert(result.contains("A"));

assert(result.contains("F"));

assert(result.contains("G"));

assert(result.contains("K"));

//combine function with m1 = m2 and an empty result

Map result2;

assert(combine(one, one, result2));

assert(result2.size() == one.size());

assert(result2.contains("A"));

assert(result2.contains("F"));

//combine function with m1 = m2 and a non-empty result

assert(combine(one, one, result2));

assert(result2.size() == one.size());

assert(result2.contains("A"));

assert(result2.contains("F"));

//combine function with m1 = result

Map originalOne(one);

assert(combine(one, two, one)); //one is indirectly modified

assert(one.size() == originalOne.size() + two.size());

assert(one.contains("A"));

assert(one.contains("F"));

assert(one.contains("G"));

assert(one.contains("K"));

//combine function with m2 = result

Map originalTwo(two);

assert(combine(originalOne, two, two));

assert(two.size() == originalOne.size() + originalTwo.size());

//combine function with m1 and m2 having similar keys and values (with empty result)

Map result3;

Map dups;

dups.insert("A", 5);

dups.insert("D", 20);

dups.insert("F", 30);

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

assert(combine(originalOne, dups, result3));

assert(result3.size() == 6);

assert(result3.contains("A"));

assert(result3.contains("F"));

//combine function with m1 and m2 having similar keys and values (with non-empty result)

assert(combine(originalOne, dups, result3));

assert(result3.size() == 6);

assert(result3.contains("A"));

assert(result3.contains("F"));

//combine function with m1 and m2 having a similar key with conflicting values (with 2 pairs remaining)

Map result4;

assert(dups.update("A", 6));

assert(originalOne.erase("E"));

assert(originalOne.erase("B"));

assert(originalOne.erase("C"));

assert(!combine(originalOne, dups, result4)); //pair with key"A" should not appear in result

assert(result4.size() == 2);

assert(!result4.contains("A"));

//test reassign

Map dance;

Map dance2;

cout << "originalTwo" << endl;

print(originalTwo); //check prints to see if values have shifted

reassign(originalTwo, dance);

cout << "dance" << endl;

print(dance);

reassign(dance, dance2);

cout << "dance2" << endl;

print(dance2);

cout << "Passed all tests" << endl;

}