Urvi Shah

Project 2 Report

1. Each element in my list has a pointer to the previous element and to the next element. The first element of my linked list is a dummy node, with head’s previous and next pointer pointing to itself when the linked list is empty. Every node contains a key and a value of type KeyType and ValueType, which are default string and double, respectively.

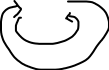
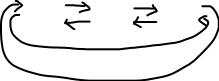
empty Map: typical Map:

second element

first element

head

head



1. Pseudocode

**Map();**

Create Map of size 0

Make next and previous pointers in dummy node point to itself

**Map(const Map &src);**

Create Map of size 0

Make next and previous pointers in dummy node point to itself

Traverse every node in the parameter:

Add each node to the current Map

**Map &operator=(const Map &src);**

Traverse every node in current Map:

Delete each node

Create new Map of size 0

Traverse every node in the Map parameter:

Add each node to the current Map

**~Map();**

Traverse every node in current Map:

Delete each node

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

Exits function if key already exists inside Map

Create a new node with the values passed as parameters

Insert node into current Map

Adjust surrounding nodes’ pointers to point at current node

Adjust the current node’s pointer to point at surrounding nodes

**bool update(const KeyType& key, const ValueType& value);**

Traverse through every node in the current Map:

If key is found in the Map:

Change value of node and exit function

**bool insertOrUpdate(const KeyType& key, const ValueType& value);**

Traverse through every node in the current Map:

If key is found in the Map:

Change value of node and exit function

If key already exists inside Map, create a new node with the values passed as parameters

insert node into current Map

Adjust surrounding nodes’ pointers to point at current node

Adjust the current node’s pointer to point at surrounding nodes

**bool erase(const KeyType& key);**

Traverse through every node in the current Map:

If key is found in the Map:

Node is removed from Map

**bool contains(const KeyType& key) const;**

Traverse through every node in the current Map:

If key is found in the Map:

Exit function and return true

Otherwise exit function and return false

**bool get(const KeyType& key, ValueType& value) const;**

Traverse through every node in the current Map:

If key is found in the Map:

Set value equal to the value corresponding with the current key

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

Traverse through *i* nodes in the Map if i is less than the size and greater than 0:

Set value and key equal to the values corresponding with the current node

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

Switch head pointers for each Map

Switch sizes for each Map

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

Traverse through every node in Map, m1:

Add each node to Map, result

Traverse through every node in Map, m2:

If Map, result has the same key as the node but a different value:

Remove key from Map, result

If Map, result has the same key and value as the node:

Leave key inside Map, result

Otherwise, add node from Map, m2, to Map, result

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

Exit function if size is 0 or 1

Traverse through every node in Map, m:

Insert each node into Map, result, with the key of the current node and the value of the

node after

Insert last node with the value from the first node from Map, m, into Map, result

Traverse through every node in Map, result:

Remove any node that is not also present in Map, m

1. Test Cases

//tests Map’s default constructor

Map m1;

assert(m1.size() == 0); //test size

assert(m1.empty()); //test empty

assert(!m1.contains("test1")); //test contains

KeyType k = "test1";

ValueType v = 2.3;

//tests insert and contains function for elements at beginning

assert(m1.insert(k,v) && !m1.insert(k,v) && m1.size()==1);

assert(m1.contains("test1"));

//tests insert and contains for elements in the middle

assert(!m1.empty() && m1.insert("test2",3.4) && m1.size() == 2);

assert (m1.contains("test1") && m1.contains("test2"));

assert(!m1.empty()); //tests empty when Map isn’t empty

assert (!m1.contains("test3")); //tests contains when it’s false

//tests insertOrUpdate and contains for elements at the end of the list

assert(m1.insertOrUpdate("test3",4.5));

assert(m1.contains("test2") && m1.contains("test3"));

Map m2;

//tests that the get function fails for elements that don't exist

assert(!m2.get("test4",v));

//tests insertOrUpdate and get function for elements in beginning

assert(m2.insertOrUpdate("test4",4.3));

assert(m2.get("test4",v) && v == 4.3);

assert(m2.insertOrUpdate("test4",8.7));

assert(m2.get("test4",v) && v == 8.7);

//tests insertOrUpdate and get function for elements in middle

assert(m2.insertOrUpdate("test5",3.5) && m2.insertOrUpdate("test6",7.1));

//tests insertOrUpdate and get function for elements in end

assert(m2.insertOrUpdate("test6",2.3) && m2.get("test6",v) && v==2.3);

assert(!m2.get("test20",v) && m2.get("test5",v) && v == 3.5);

Map m3(m2); //checks if copy constructor works in a normal case

assert(m3.size()==3);

//tests if get function works for elements at the end, beginning, and middle

assert(m3.get(2,k,v) && k == "test6" && v == 2.3);

assert(m3.get(1,k,v) && k == "test5" && v == 3.5);

assert(m3.get(0,k,v) && k == "test4" && v == 8.7);

//tests that get function returns false if number passed is too big or small

assert(!m3.get(-1,k,v) && !m3.get(3,k,v) && !m3.get(24,k,v));

//tests that erase function works with elements at beginning, end, and middle

assert(m3.erase("test5") && m3.size()==2 && m3.erase("test6"));

assert(m3.get(0,k,v) && k == "test4" && v == 8.7);

assert(!m3.erase("test7"));

assert(m3.erase("test4") && m3.size()==0);

assert(!m3.erase("test4"));

Map m4;

Map m5(m4); //tests that the copy constructor works with empty lists

assert(m5.size()==0 && !m5.get(0,k,v) && !m5.get(1,k,v));

assert(!m5.update("test3",2.0));

assert(m5.insert("test8",42.3) && m5.update("test8", 32.1) && m5.get("test8",v) && v == 32.1); //tests that update works with one element in list

m4 = m2; //tests that assignment operator works when assigning to an empty list

assert(m4.size() == m2.size()); //tests size

assert(m4.contains("test4") && m4.contains("test5") && m4.contains("test6"));

//tests that update works in the beginning, middle, and end

assert(m2.update("test4", 7.8) && m2.get("test4",v) && v == 7.8);

assert(m2.update("test5", 34.2) && m2.get("test5",v) && v == 34.2);

assert(m2.update("test6", 21.5) && m2.get("test6",v) && v == 21.5);

assert(!m2.update("test27", 42.3));

m2.swap(m1); //checks if swap function works with two lists of same size

assert(m2.contains("test1") && m2.contains("test2") && m2.contains("test3"));

assert(m1.contains("test4") && m1.contains("test5") && m1.contains("test6"));

//checks if swap function works with lists of two different sizes

assert(m1.erase("test6"));

m1.swap(m2);

assert(m1.size()==3 && m2.size()==2);

assert(m1.contains("test1") && m1.contains("test2") && m1.contains("test3"));

assert(m2.contains("test4") && m2.contains("test5"));

//checks if swap function works with lists that are empty and lists that have one element

assert(m2.erase("test4") && m2.erase("test5") && m1.erase("test2") && m1.erase("test3"));

m1.swap(m2);

assert(m1.empty());

assert(m2.contains("test1") && !m2.empty());

//checks if assignment operator and copy constructor with lists of one element

m3 = m2;

assert(m3.contains("test1") && m3.size()==1);

Map m6(m2);

assert(m6.contains("test1") && m6.size()==1);

//checks that combine works in a normal case where there are no repeating keys and the result Map is empty

Map m7;

assert(m7.insert("test1",2.4) && m7.insert("test2",2.1) && m7.insert("test3",2.7));

Map m8;

assert(m8.insert("test4",2.1) && m8.insert("test5",3.4) && m8.insert("test6",3.7));

Map result;

assert(combine(m7,m8,result) && result.size()==6);

assert(result.contains("test1") && result.contains("test2") && result.contains("test3") && result.contains("test4") && result.contains("test5") && result.contains("test6"));

//checks if combine works with repeating keys of same and different values and with a result Map that is not empty

assert(m8.insert("test1",2.4) && m8.insert("test2",2.3));

assert(!combine(m7,m8,result));

assert(result.get("test1",v) && v == 2.4 && !result.contains("test2") && result.contains("test3") && result.contains("test4") && result.contains("test5") && result.contains("test6"));

//checks if combine works when the first two Maps are empty

assert(combine(m1,m1,result) && result.size()==0);

//checks if combine works when m1, m2, and result are passed the same Maps

assert(combine(result,m8,result));

assert(result.contains("test2") && result.contains("test1") && result.contains("test6") && result.contains("test5") && result.contains("test4") && result.size()==5);

assert(!combine(m7,result,result));

assert(result.get("test3",v) && v == 2.7 && result.get("test1",v) && v==2.4 && result.contains("test4") && result.contains("test5") && result.contains("test6"));

assert(combine(result,result,result));

assert(result.size() == 5 && result.contains("test3") && result.contains("test1") && result.contains("test6") && result.contains("test5") && result.contains("test4"));

//checks if reassign works with odd and even numbered maps

//and if reassign works with result Maps that are empty and not empty

Map m9;

Map result2;

assert(m9.insert("test1", 2.0) && m9.insert("test2",3.0) && m9.insert("test3",4.0));

m9.dump();

cout << endl;

reassign(m9,result2);

result2.dump();

cout << endl;

assert(m9.insert("test4",5.0));

m9.dump();

cout << endl;

reassign(m9,result2);

result2.dump();

cout <<endl;

//checks if reassign works when both passed values are the same Map

result.dump();

cout<<endl;

reassign(result,result);

result.dump();

cout<< endl;

//checks if reassign works when there is only one element in the Map

assert(m9.erase("test4") && m9.erase("test3") && m9.erase("test2"));

m9.dump();

cout<<endl;

reassign(m9,result);

result.dump();

cout << endl;

//checks if reassign works when there are no elements in the Map

assert(m9.erase("test1"));

m9.dump();

cout<<endl;

reassign(m9,result);

result.dump();

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