Ethan Wong

CS32

Professor Smallberg

Project 2 Report

Linked List Implementation

My nodes are structs that are comprised of a KeyType KEY, ValueType VALUE, Node\* prev, and Node\* next. The pointers point to the nodes before and after the current node, respectively.

My doubly-linked list is a standard linear linked list. It uses a head pointer that points at the first node in the list, as well as “next” and “previous” pointers within each node that is typical of a doubly-linked list. I included a variable “amount” in my program as well to keep track of the amount of nodes were in the linked list at any given time. This was useful for implementing the size() function. My last node’s “next” points to nullptr” in order to signify that it is the end of the list.

Pseudocode

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

If the key already exists in the map, return false

Create a new node, set KEY and VALUE to key and value, respectively

New node’s “prev” to point at nullptr

New node’s “next” to point at head

Head now points at newly added node

Add 1 to amount if inserted successfully

Return true

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

Create another pointer that points at head

Traverse through map and see if there is a node with a KEY that corresponds with key

If a node with corresponding KEY is found, update its VALUE so it equals value

Return true

If update could not be done successfully, return false

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

While not at the end of the list, check each node if it has a KEY that matches key:

If that node happens to be the first node:

Have head point to the second element of the map

Delete the node, return true

If that node happens to be the last node:

Make second-to-last node point to nullptr (this makes it the last node)

Delete the node, return true

If the node happens to be in the middle of the list (not first or last node):

Set the previous node’s next to deleted node’s next

Set the succeeding node’s prev to the deleted node’s prev

Delete the node, return true

Return false if the node with the correct key was not found

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

While not at the end of the list, check each node if it has a KEY that matches key:

If a node has a KEY that matches key

Return true

Else

Return false

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

If key is contained within the map

Traverse through map until finding node with KEY that matches with key

When that node is found, change its VALUE to equal value

Return true

If the node with the specified key cannot be found

Return false

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

If “i” is in bounds

Iterate through the map “i” times

Once done iterating, set value to equal VALUE and key to equal KEY of that node

Return true if this was done successfully

If “i” was not in bounds, do nothing and return false

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

Swap the head pointers of the two maps

Swap the “amount” of the two maps

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

Create a boolean (outcome) and set it to true

Make result contain all the same information as m1

Iterate through and all the nodes in m2

Get the information about the keys and values in m2

If result contains key

Get its value

If values aren’t equal

Delete node and set “outcome” to false

Else

Add the node to result

After loop is finished, return “outcome”

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

If there is only one key/value pair

Put that in the result and return

Make m and result identical by assignment operator

Begin a loop to iterate through elements

If at last element

Swap value from last element with value from first element

Loop through and exchange value with value from element succeeding the current one

Update result

Return

Test Cases

Map t;

assert(t.size() == 0); // testing size function

assert(t.empty()); // testing empty function

assert(t.insert("Joe", 10)); //testing insert function

assert(t.insert("Mama", 11));

assert(t.insert("aye", 91));

assert(t.size() == 2); // testing size function

// testing insert and update function

assert(t.update("Joe", 96) && !t.update("cakeman", 2) && t.insert("please", 1854) && !t.insert("please", 558));

//tests the insertOrUpdate function

t.insertOrUpdate("Holy", 239);

t.insertOrUpdate("aye", 45);

assert(t.contains("Holy") && !t.contains("Mamba"));

// test that erase works

assert(!t.erase("Whoop") && t.erase("Mama") && !t.contains("lifetime"));

double value;

t.get("Joe", value);

assert(value == 96);

//Another similar test to above page

Map cry;

assert(cry.empty()); //check if newly created map is empty

std::string animals[10] =

{“Cow”, “Horse”, “Hippo”, “Lion”, “Doge”, “Cat”, “Monkey”, “Rhino”, “Parrot”, “Mule”};

for (int i = 0; i < 10; i++)

{

cry.insert(animals[i], i \* i); //inserting more elements into the map

}

for (int i = 0; i < 10; i++)

{

std::string key;

double value;

yay.get(i, key, value); //testing the three parameter get function

assert(key == animals[10 - i] && value == (10 - i)\*(10 - i));

}

Map m;

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

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

assert(m.insert("Pizza", 123)); //test insert function

assert(m.insert("Tortilla", 234));

assert(m.insert("Cow", 345));

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

assert(m.erase("Pizza")); //test erase function

(!m.erase("Pizza"));

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

//testing swap function

int cry = cry.size();

int t = t.size();

cry.swap(t);

assert(t.size() == cry && cry.size() == t); //check if sizes changed properly

for (int i = 0; i < 10; i++) {

std::string key;

double value;

t.get(i, key, value); //want to check if the values got swapped

assert(key == animals[10 - i] && value == (10 - i) \* (10 - i));

assert(cry.contains("Joe")); //check on the value to see if it swapped correctly

t.swap(cry);

for (int i = 0; i < 10; i++) {

std::string key;

double value;

cry.get(i, key, value); //checking if swap can swap things back to normal

assert(key == animals[10 - i] && value == (10 - i) \* (10 - i));

}

t.swap(t); //testing if function can swap with itself

Map lame;

t.swap(lame); //final check for swap with a blank map

assert(!m.size() == 10 && !m.contains("Ronald Reagan"));

//testing combine

Map one;

Map two;

Map result;

one.insert("Pikachu", 777);

two.insert("ayyyy", 635);

one.insert("Charmander", 555);

two.insert("Squirtle", 222);

one.insert("Roselia", 888);

assert(combine(one, two, result)); //test if the combination worked as expected

assert(result.contains("Pikachu") && result.contains("Roselia") && !result.contains(“ty”));

//testing for if there are two of the same key in the different maps

Map one;

Map two;

Map result;

one.insert("Pikachu", 777);

two.insert("Pikachu", 635);

one.insert("Charmander", 555);

two.insert("Squirtle", 222);

one.insert("Roselia", 888);

assert(combine(one, two, result)); //test if the combination worked as expected

assert(result.contains("Pikachu") && result.contains("Roselia") && !result.contains(“ty”));

//testing if “one” is empty

Map one;

Map two;

Map result;

two.insert("Pikachu", 635);

two.insert("Squirtle", 222);

assert(combine(one, two, result)); //test if the combination worked as expected

assert(result.contains("Pikachu") && result.contains("Roselia") && !result.contains(“ty”));

//testing reassign

Map one;

Map result;

one.insert("Fred", 123);

one.insert("Ethel", 456);

one.insert("Lucy", 789);

one.insert("Ricky", 321);

reassign(one, result); //this is to test if the reassign works properly

for (int i = 0; i < result.size(); i++)

{

ValueType value;

KeyType key;

result.get(i, key, value);

cout << key “->” value << endl; //print out values to see if they’re correct

}

(I commented out some/all of the insert functions and tested multiple times to see if it worked properly. I won’t type out the same thing that many times because it’s practically the same as what’s shown above.)