Eduardo Ramirez Torres

Winter 2020

CS32

**Project 2 Report**

Linked List Implementation

My linear linked list initializes each new list with a header pointer pointing to null. When a new node is allocated, it is added at the front of the linked list by having the next pointer point to the head node, and then having the head pointer point to the new node. The previous pointer is only used when necessary (i.e. in the functions where we need to insert or delete a node in the middle of a list, in which case the loop will initialize each previous pointer to the previous node). No tail pointer is included since I personally prefer to work using only the header, and because the tail pointer would be useless due to the way I implemented the functions of this project.

Pseudocode

Map& Map::operator=(const Map &other){

If this map has the same address as other

Return \*this

Else, copy construct holder map using other

Use swap function to swap holder and this

Return \*this

}

Bool Map::insert(const KeyType &key, const ValueType &value){

If map contains key

Return false

Else, initialize pointer to new node

Point to node’s value and key, set to arguments key and value

New node’s next links to current node head points to (the top)

Head points to new node

Increment number of nodes in list

Return true

}

Bool Map::erase(const KeyType &key){

Node pointer points to head node

Traverse through list

If key is in list

If the node is at the start (previous = null)

Head points to next

If the node is at the end (next = null)

Other Node pointer points to previous node

Set Other node’s next pointer to null

Else,

Other Node pointer points to previous node

Set Other node’s next pointer to Node’s next

Delete Node, decrement list size counter, return

Else if not found yet

Holder Node pointer points to current node

Current node moves to next

Next node’s previous pointer points to Holder

Return false

}

Void Map::swap(Map &other){

Node pointer points to head node

Temporary int holds list size

Head pointer points to other map head node

Other map head pointer points to Node Pointer node

List size now equals other list size

Other list size now equals temporary holder size

}

Bool combine(const Map &m1, const Map &m2, Map &result){

Put m1 in a temp map

Swap temp map and result; m1 is now in result

Create flag to check repeats in keys, set true

Traverse through m2

Get value and key at each iteration point

If result contains a key in m2

If result value not equal m2 value

Set flag to false

Erase the key in result

Insert value into result map

Return flag

}

Void reassign(const Map &m, Map &result){

Construct an empty Map

Swap empty map with result, ensure result is empty

If m size is 0 or 1

Set result equal to m, return

Create two Map groups

If the size of m is even

Traverse through one half of m

Get and insert values into group1

Traverse through the other half of m

Get and insert values into group2

Traverse through the size of group1

Get the values and keys of each node

Update each node by swapping value1 and value2

Combine group1 and group2, return

Else

Traverse through one half of m

Get and insert values into group1

Traverse through the other half of m -1 node

Get and insert values into group2

Traverse through the size of group1

Get the values and keys of each node

Update each node by swapping value1 and value2

Combine group1 and group2 into a temp map

Get and insert the last node of m into group3

Get the value and key of the first node

Update by swapping value3 and firstNodeValue

Combine group 3 and temp into result, return

}

Test Cases

All default test cases on website were utilized //Test if program works

KeyType keyTest;

ValueType valueTest;

Map m1;

assert(m1.empty()); //checks that empty function works

m1.insert("alex",1);

assert(!m1.empty());

assert((m1.size())==1); //checks size function

assert(!(m1.insert("alex",1))); //checks insert function when key already exists

assert((m1.size())==1);

m1.insert("davidsmallberg",2); //checks insert function

assert((m1.size())==2);

m1.insert("carey",2); //checks erase function

m1.erase("carey");

assert((m1.size())==2 && !(m1.contains("carey")));

assert(m1.contains("davidsmallberg")); //checks contains function

m1.insertOrUpdate("alex",3); //checks insertOrUpdate function

m1.get("alex",valueTest); //checks get function

assert(valueTest==3);

m1.insertOrUpdate("eggert",2);

assert((m1.size())==3);

m1.get(0,keyTest,valueTest); //check get function with iteration argument

assert(keyTest=="eggert" && valueTest==2);

assert(!(m1.get(4,keyTest,valueTest))); //check when getting values out of bounds

Map m2;

m2.insert("c", 1);

m2.insert("a", 2);

m2.insert("t", 3);

m2.insert("s", 3);

Map m3=m2; //checks copy constructor

assert(m1.get(0,keyTest,valueTest)==m2.get(0,keyTest,valueTest));

m3=m1; //check assignment operator

assert(m1.get(0,keyTest,valueTest)==m3.get(0,keyTest,valueTest));

combine(m1, m2, m3); //check combine function

for (int i=0; i<m1.size(); i++)

{

m1.get(i,keyTest,valueTest);

cout<<keyTest<<" "<<valueTest<<endl;

}

cout<<endl;

for (int i=0; i<m2.size(); i++)

{

m2.get(i,keyTest,valueTest);

cout<<keyTest<<" "<<valueTest<<endl;

}

cout<<endl;

for (int i=0; i<m3.size(); i++)

{

m3.get(i,keyTest,valueTest);

cout<<keyTest<<" "<<valueTest<<endl;

}

cout<<endl;

m2.insertOrUpdate("s", 2);

combine(m1, m2, m3); //checks that differing value keys don’t get copied over

for (int i=0; i<m3.size(); i++)

{

m3.get(i,keyTest,valueTest);

cout<<keyTest<<" "<<valueTest<<endl;

}

Map m4;

m4.insert(“sally”,6);

m4.insert(“craig”,9);

m4.insert(“diana”,2);

m4.insert(“jack”,4);

for (int i=0; i<m4.size(); i++)

{

m4.get(i,keyTest,valueTest);

cout<<keyTest<<” “<<valueTest<<endl; //cout all values

}

Map m\_result;

reassign(m4,m\_result); //checks reassign function

for (int i=0; i<m\_result.size(); i++)

{

m\_result.get(i,keyTest,valueTest);

cout<<keyTest<<” “<<valueTest<<endl; //cout all values to compare to m4

}