Aidan Shahmaram

Professor Nachenberg

Computer Science 32

30 January 2023

Project 2 Report

1. The class contains a “head” and “tail” node which represent the start and end point of the linked list. Each node contains, alongside its key and value, a pointer to the previous (prev) and next (next) node. When constructed, the head holds tail as its next and a “nullptr” as its prev, while the tail holds head as its prev and a “nullptr” as its next. When a node is added, it is created through a pointer and joins the list through the tail, being placed just before it. When going through the list of nodes, a function will start at the node held by head’s next, and go until a node holds “nullptr” as its next (being the tail). They’re only order is that in which they were added in.

Pictures are at the end.

2.

Destructor

Start at this head’s next node

Repeatedly

Go to the next node

Delete the previous node

Copy constructor:

initiate member variables

start at the node after head

repeatedly

Insert a new node

Go to the next node

Assignment operator:

If it is given itself

Return itself

Start at this head’s next node

repeatedly

delete the current node

Go to the next node

Reset all member variables

Starting at “old” head’s next

Repeatedly

Insert a new node

Go to the next node

Insert:

Start at this head’s next node

Repeatedly

if the key is the same

Return false

Go to the next node

Create a new node

Set its key and value

Set its pointers

Change the “next”pointer of the previous node

Change the “prev” pointer of the tail node

Return true

Update:

Start at this head’s next node

Repeatedly

If the current node’s key is the same as the given input

changes its value to the input

Return true

Go to the next node

Return false

Erase:

Start at this head’s node

Repeatedly

If the current node’s key is the same as the given input

change the “next” pointer of “prev”

change the “prev” pointer of “next”

Delete the current node

Decrease size

Return true

Go to the next node

Return false

Get (with integer input):

Start at this head’s next node

Set counter to 0

Repeatedly

Start again at the head’s next node

Repeatedly

if the first node’s key is greater than the current node’s key

Increase the counter

Go to the next node

If count is equal to the given integer

Set the given variables to the current node’s key and value

Go to the next node

Reset the counter

Return false

Swap:

If it is given itself

Return

Swap where both map’s head’s point “next” to

Change where the first node in each map’s “prev” points to

Swap where both map’s tail’s point “prev” to

Change where the last node in each map’s “next” points to

Switch both size variables

Merge:

Create an output map

Set the return value to true

Copy map 1 into the output

Repeatedly

retrieve the key and value of a node in map 2

Insert a new node with those values into output

If it can’t be inserted

Check if the value variables are the same

If they are not the same

Change the return value

Erase the node from the output

Copy the output map into result

Return the return value

Reassign:

Copy the given map into the result

If the result has one or less nodes

Return the current result

Store the value of the first node

For each node in the given map

Change the corresponding node in return to have the same value as the next node

Set the last node’s value to that of the first node

3.

//test contructor

Map newMap;

//for an empty map

string st = "test";

double ret = 5;

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

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

assert(!newMap.erase("Josh"));//nothing to erase

assert(!newMap.get("Dan", ret));//nothing to get

assert(ret == 5);// get should not have changed "ret"

assert(!newMap.get(0, st, ret));

assert(st == "test" && ret == 5);

//test insert

assert(newMap.insert("first", 1023));

assert(newMap.insert("second", 76));

assert(newMap.insert("third", 908));

assert(!newMap.insert("first", 12));//test insert for already existing key

//test contains

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

assert(newMap.contains("third"));

assert(!newMap.contains("fourth"));

//test size

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

//test get (no integer input)

double a, b, c, g;

assert(newMap.get("first", a));

assert(newMap.get("second", b));

assert(newMap.get("third", c));

assert(!newMap.get("fourth", g));//test for non existant key

assert(a == 1023 && b == 76 && c == 908);//test values changed by get

//test update

assert(newMap.update("third", 45));

assert(newMap.get("third", c));

assert(c == 45);//test value changed by update

//test insertOrUpdate

double d;

assert(newMap.insertOrUpdate("second", 807));//(update)

assert(newMap.insertOrUpdate("fourth", 3400));//(insert)

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

assert(newMap.get("second", b));

assert(newMap.get("fourth", d));

assert(b == 807 && d == 3400);//test values from insertOrUpdate

//test erase

assert(newMap.erase("second"));

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

assert(!newMap.contains("second"));

//test get (with integer input)

string key;

double value;

assert(newMap.get(0, key, value));

assert(key == "first" && value == 1023);

assert(newMap.get(1, key, value));

assert(key == "fourth" && value == 3400);

assert(newMap.get(2, key, value));

assert(key == "third" && value == 45);

assert(!newMap.get(3, key, value));//test for integer with no corresponding pair

assert(key == "third" && value == 45);

//test swap

Map nintendo;

nintendo.insert("Mario", 35);

nintendo.insert("Kirby", 3);

nintendo.insert("Donkey Kong", 27.5);

nintendo.insert("Link", 406);

nintendo.swap(newMap);//two maps containing values;

assert(nintendo.contains("first"));//checkse

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

assert(!nintendo.contains("Mario"));

assert(newMap.contains("Kirby"));

nintendo.swap(newMap);

Map left;

nintendo.swap(left);//swapping with a Map with no values

assert(!nintendo.contains("Mario"));

assert(!nintendo.contains("Kirby"));

assert(left.contains("Mario"));

assert(left.contains("Donkey Kong"));

nintendo.swap(left);

nintendo.swap(nintendo);//swapping a map with itself

assert(nintendo.contains("Link"));

assert(nintendo.contains("Kirby"));

//test merge

Map merged, merged2;

assert(merged.insert("nat", 45.6));

assert(merge(nintendo, newMap, merged));//two maps with different keys

assert(merged.contains("Kirby"));

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

assert(merged.contains("third"));

assert(!merged.contains("nat"));

assert(merge(nintendo, merged, merged2));//two maps that have similar keys with similar values

assert(merged2.contains("Kirby"));

assert(merged2.contains("Mario"));

assert(merged2.contains("fourth"));

assert(merged.update("Mario", 14.87));

assert(!merge(nintendo, merged, merged2));//two maps that have a similar key with a different value

assert(!merged2.contains("Mario"));

//test reassign

Map rearranged;

rearranged.insert("SCREAM", 21);

reassign(nintendo, rearranged);//rearrange a map into another map

string s;

double p;

rearranged.get(0, s, p);

assert(s == "Donkey Kong" && p != 27.5);

rearranged.get(1, s, p);

assert(s == "Kirby" && p != 3);

rearranged.get(2, s, p);

assert(s == "Link" && p != 406);

rearranged.get(3, s, p);

assert(s == "Mario" && p != 35);

assert(!rearranged.contains("SCREAM"));//should not have any nodes outside of “nintendo"

reassign(nintendo, nintendo);//rearrange a map into itself

nintendo.get(0, s, p);

assert(s == "Donkey Kong" && p != 27.5);

nintendo.get(1, s, p);

assert(s == "Kirby" && p != 3);

nintendo.get(2, s, p);

assert(s == "Link" && p != 406);

nintendo.get(3, s, p);

assert(s == "Mario" && p != 35);

Map last;

reassign(last, rearranged);//reassigning an empty map

assert(rearranged.empty());

last.insert("lamp", 0.43);

reassign(last, rearranged);//reassigning a map with only one node

assert(rearranged.contains("lamp"));

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

Head

Next: Tail

Prev: nullptr

Tail

Next: nullptr

Prev: Head

Head

Next: Node1

Prev: nullptr

Node2

Next: Tail

Prev: Node1

Node1

Next: Node2

Prev: Head

Tail

Next: nullptr

Prev: Node2

Empty Map

Typical Map