Project 2

# Description of my Doubly Linked List

Text

Description automatically generated For the doubly linked list used by my map, the pointer to the previous node of the head node is nullptr and the next node of the tail node is also nullptr. I do not have a dummy node.

The head and tail pointer stored in the map would both be nullptr. The map is not linked currently. Each node contains information on its own data and pointers to the previous and next nodes. The map does contain information on the head and tail nodes. As well as the map size.

# Pseudocode

Map constructor:

* + Set the number of items to zero
  + Point the head and tail to null because there are currently no nodes

Map destructor:

* + If there are no items in the list:
    - There is no allocated memory, so no need to do anything

* + - Otherwise:
    - Go through each node:
      * Delete this node
      * Go to the next node

Map copy constructor:

* + Set the number of items to other’s number of items
  + Traversing through each node in other:
    - Make a new node in this map
    - Set its data to the data from the respective node in other
    - Connect this node to the previous and next node

Map equal operator override:

* + Make sure this object isn’t being set to itself:
    - If not, copy the right hand side to a temp variable
    - Swap the left hand side with that temp variable
  + Return \*this, aka itself

Map insert:

* + Check if this map contains the key to be inserted:
    - If it does, exit and return that this function failed
  + If this map is empty:
    - Create a new node
    - Fill its data with the given key and value
    - Connect it to the null pointer on either side because it’s the only node
    - Point the map’s head and tail to it
    - Increment number of items
  + If this map isn’t empty:
    - Create a new node
    - Fill its data with the given value
    - Attach it to the end:
      * Connect its previous pointer to the map’s current tail
      * Connect its next pointer to the null pointer
      * Point the map’s tail to this new node
    - Increment number of items
  + Return true

Map erase:

* + Check if this map contains the value to be erased:
    - If it doesn’t, exit and return that this function failed
  + Find the node that contains the value
  + Point the previous node to the next node and next to the previous
  + Delete the node containing the value

Map get:

* + Check if given index is key:
    - If it isn’t, return that this function failed
  + Create a temporary copy of this map:
  + Delete the minimum key from the copy i number of times
  + Set key to the minimum key remaining in the copy, which will be the desired key

Map swap:

* + Swap the number of items in this with other
  + Swap the head in this with other
  + Swap the tail in this with other

Merge:

* Copy the first map with the result map
* Go through each node in the second map and insert into result
* If duplicate
  + - But Values are not the same, do not include in result
  + Return whether values matched up or if there were duplicates who did not

Reassign:

* Create a temporary map
* Obtain key and value pairs of current and next nodes
* Pair the current key with next’s value
* Then Pair next’s key with current’s value
* For the next one, obtain original value of current
  + - Use that value to pair it with next’s key
* Set result map to equal the temporary map

# Test Cases

#include "Map.h"

#include <iostream>

#include <cassert>

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

**int** main() {

Map my\_map;

assert(my\_map.empty() == **true**);

my\_map.insert("hello", 45.0);

assert(!my\_map.empty() && my\_map.size()== 1 && !my\_map.contains("Hello"));

my\_map.insert("what", 43.0);

assert(!my\_map.empty() && my\_map.size()== 2);

my\_map.insert("yellow", 23.0);

assert(!my\_map.empty() && my\_map.size()== 3);

my\_map.update("yellow", 0.0);

my\_map.update("hello", 2.0);

assert(my\_map.size()== 3);

assert(my\_map.contains("hello") && !my\_map.contains("Michelle"));

my\_map.insertOrUpdate("yellow", 9);

assert(!my\_map.contains("Teresa"));

my\_map.insertOrUpdate("Teresa", 17);

assert(my\_map.contains("Teresa") && my\_map.size()== 4);

my\_map.erase("hello");

assert(my\_map.size()== 3 && !my\_map.contains("hello"));

my\_map.insert("Camila", 8.70);

ValueType test;

assert(my\_map.get("Camila", test) && test==8.70);

assert(!my\_map.get("hello", test) && test==8.70);

assert(my\_map.get("yellow", test) && test==9);

Map m1;

m1.insert("yolo", 55.0);

my\_map.swap(m1);

assert(m1.size() != 0 && my\_map.size() == 1);

my\_map.swap(my\_map);

assert(my\_map.size() == 1);

ValueType test1;

assert(m1.get("Camila", test1) && test1==8.70);

//check copy constructor

Map testMe = m1;

ValueType test2;

assert(testMe.get("Camila", test2) && test2==8.70);

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

//check assignment operator

Map your\_map;

your\_map = my\_map;

ValueType your\_val;

your\_map.get("yolo", your\_val);

assert(your\_map.size() == 1 && your\_val == 55.0);

your\_map.erase("yolo");

assert(!your\_map.contains("yolo"));

//check merge function

Map my\_result;

my\_result.insert("no", 0.0);

Map mm1; Map mm2;

mm1.insert("a", 1.0);

mm1.insert("c", 3.0);

mm2.insert("b", 2.0);

mm2.insert("d", 4.0);

assert(merge(mm1, mm2, my\_result)&& my\_result.size()==4);

mm2.insert("c", 4.5);

assert(!merge(mm1, mm2, my\_result) && my\_result.size()==3);

Map cs1; Map cs2; Map cs\_result;

cs\_result.insert("no", 3.0);

cs1.insert("Fred", 123);

cs1.insert("Ethel", 456);

cs1.insert("Lucy", 789);

cs2.insert("Lucy", 789);

cs2.insert("Ricky", 321);

assert(merge(cs1, cs2, cs\_result) && cs\_result.size()==4

&& !cs\_result.contains("no"));

cs2.update("Lucy", 788);

assert(!merge(cs1, cs2, cs\_result) && cs\_result.size()==3

&& !cs\_result.contains("Lucy"));

//check reassign function

//refer to same map? reassign same map!

Map redo;

redo.insert("Fred", 123);

redo.insert("Ethel", 456);

redo.insert("Lucy", 789);

redo.insert("Ricky", 321);

reassign(redo, redo);

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

//must have: "Fred" 789 "Ethel" 123 "Lucy" 321 "Ricky" 456

ValueType fred, ethel, lucy, ricky;

redo.get("Fred", fred);

redo.get("Ethel", ethel);

redo.get("Lucy", lucy);

redo.get("Ricky", ricky);

assert(fred == 789 && ethel == 123 && lucy == 321 && ricky == 456);

Map redo2; Map redo3;

redo2.insert("Fred", 123);

redo2.insert("Ethel", 456);

redo2.insert("Lucy", 456);

reassign(redo2, redo3);

redo3.get("Fred", fred);

redo3.get("Ethel", ethel);

redo3.get("Lucy", lucy);

// must have: "Fred" 456 "Ethel" 123 "Lucy" 456

assert(fred == 456 && ethel == 123 && lucy == 456);

redo3.erase("Lucy");

assert(!redo3.contains("Lucy") && !redo3.get("Lucy", lucy));

cout << "All tests passed.\n";

}