Cody Do

105140467

3C

Doubly-Linked List Description

My doubly-linked list is quite standard in design. It makes use of multiple node structs, each one possessing a KeyType *key* variable, a ValueType *value* variable, a *next* pointer that points to the next item on the list, and a *prev* pointer that points to the previous item on the list. It also has a head and tail pointer, which points to the first and last items of the list respectively. The list, however, is not circular but rather linear in nature. No dummy nodes are used in the implementation. The list is also built from the top-down, meaning new items are connected to the last added item.



Pseudocode

**Copy Constructor:**

*set target size equal to source size*

*repeatedly for size of map:*

*pull keys and values from source*

*place keys and values to target*

**Assignment Operator:**

*clear target map’s nodes*

*set target size equal to source size*

*pull keys and values from source*

*place keys and values to target*

**findItem (Private Class member function):**

*create node pointer*

*scan through the items in the list*

*if item matches, return*

**insertOrUpdate:**

*check if key already exists (findItem function)*

*if it does, update it*

*if key doesn’t exist*

*add the key/value pair*

**Erase:**

*check if the key already exists (findItem function)*

*if it exists, delete item*

*reduce map size and return true*

**Combine:**

*check if one map is empty*

*if one is, return non-empty map*

*create copy of second map and boolean to return*

*repeatedly for size of first map:*

*pull key/value pair from the first map*

*check if key/value pair matches any in second map*

*if key matches but values don’t, remove pair from copy of second map*

*if no matches found, add to copy of second map*

*replace result parameter with the modified copy of second map*

**Reassign:**

*check if map has less than or equal to one pair*

*if yes, return the map as is*

*create a copy of map*

*pull key/value of first node*

*pull key/value of last node*

*update first node’s value with last node*

*repeatedly for size of map, starting at second position:*

*pull key/value of current position*

*update value of current position with first node’s value*

*replace first node’s value with old value of current node*

*replace result with the copy of the map*

Test Code

Map m; **//Default constructor**

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

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

assert(!m.erase("Ricky"));  **// nothing to erase**

if (m.empty()) {

cerr << "Empty Works" << endl; **//Tests empty**

cerr << "There are/is " << m.size() << " item(s) in the map." << endl; **//Tests m.size**

}

m.insert("Pineapple", 10); **//Tests insert function**

if (!m.empty()) {

cerr << "Empty still works" << endl; **//Tests empty again**

cerr << "There are/is " << m.size() << " item(s) in the map." << endl; **//Tests m.size again**

}

m.insert("Koi", 123);

m.insert("Polka Dots", 101);

if (m.update("Koi", 456))

cerr << "Update properly returns true" ; **//Tests if update returns true if key exists**

if (!m.update("Strawberry", 99)) **//Tests if update properly returns false if key doesn't exist**

cerr << "Update properly returns false" << endl;

cerr << "There are "<< m.size() << " items in the map." << endl;

m.insertOrUpdate("Bread", 350); **//Tests insert aspect of insertorUpdate**

cerr << "There are "<< m.size() << " items in the map." << endl;

if (m.insertOrUpdate("Bread", 200)) **//Tests update aspect of insertorUpdate**

cerr << "insertOrUpdate works" << endl;

if (!m.erase("Carrots"))

cerr << "Erase works" << endl; **//Tests erase's ability to find matching key**

if (m.erase("Pineapple"))

cerr << "Erase works again" << endl; **//Tests erase's ability to properly remove an item**

**from the map**

ValueType x = 444;

if (!m.get("Water", x)) **//Tests if get properly returns false if key not found**

cerr << "get1 works -- no 'Water' key found. Value of x is " << x << endl;

if (m.get("Koi", x)) **//Tests if get returns true/alters x if key is found**

cerr << "get1 works -- Value of x is " << x << endl;

cerr << "There are "<< m.size() << " items in the map." << endl;

KeyType test1 = "blank";

ValueType test2 = 0;

if (!m.get(4, test1, test2))

cerr << "get2 test 1 passed." << " Key is " << test1 << " value is " << test2 << endl; **//Tests if**

**get returns false if the number is > m\_mapSize**

if (!m.get(-1, test1, test2))

cerr << "get2 test 2 passed" << " Key is " << test1 << " value is " << test2 << endl; **//Tests if**

**get returns false if the number is < m\_mapSize**

if (m.get(2, test1, test2))

cerr << "get2 test 3 passed" << " Key is " << test1 << " value is " << test2 << endl; **//Tests if**

**get returns true if the number is within range**

m.dump();

cerr << "---------" << endl << "Below is the new (copied) map: " << endl;

Map a = m; **//Tests the copy constructor**

a.dump();

a.insert("Orchid", 9009);

a.swap(m); **//Tests the swap function, and due to the swap function's implementation, the**

**assignment operator as well**

cerr << "Below is m, which should have Orchid: " << endl;

m.dump();

cerr << endl << "Below is a, which shouldn't have Orchid: " << endl;

a.dump();

Map result;

if (combine(m, a, result)){ **//Tests map's return true function by adding a non-repeating pair**

**in the result list.**

cerr << endl << "Map's return true function works" << endl;

result.dump();

}

a.update("Bread", 555); **//Alters value for combine test**

if (!combine(m, a, result)){ **//Tests map's return false function by having a**

**same-key-different-value match between both maps**

cerr << endl << "Map's return false function works" << endl;

result.dump();

}

Map empty;

if (combine(a, empty, result)) { **//Checks if function recognizes empty map**

cerr << endl << "Combined empty check works" << endl;

result.dump();

}

cerr << endl << endl;

Map switcheroo;

cerr << "Original Map below (3 items): " << endl;

a.dump();

reassign(a, switcheroo); **//Tests reassign with 3 items**

cerr << endl << "Switched Map below (3 items): " << endl;

switcheroo.dump();

a.insert("Bananas", 12345);

cerr << endl << "Original Map below (4 items): " << endl;

a.dump();

reassign(a, switcheroo); **//Tests reassign with 3 items**

cerr << endl << "Switched Map below (4 items): " << endl;

switcheroo.dump();

reassign(empty, switcheroo); **//Tests reassign with no items**

cerr << endl << "Switched Map below (0 items): " << endl;

switcheroo.dump();

empty.insert("notEmpty", 0);

cerr << endl << "Original Map below (1 item): " << endl;

empty.dump();

reassign(empty, switcheroo); **//Tests reassign with 1 item**

cerr << endl << "Switched Map below (1 item): " << endl;

switcheroo.dump();

empty.insert("Lunar New Year", 14);

cerr << endl << "Original Map below (2 items): " << endl;

empty.dump();

reassign(empty, switcheroo); **//Tests reassign with 2 items**

cerr << endl << "Switched Map below (2 items): " << endl;

switcheroo.dump();