**Description**

My doubly-linked list uses a dummy node and a head pointer. The dummy node allows me to insert and erase items without having to worry about the edge case of the head node. My node struct contains a key, a value, and pointers to the next and previous elements in the linked list. When items are inserted, they are sorted in increasing order by the keys.

**Pseudocode**

**copy constructor(other)**

set dummy to a new node

set size to the other map’s size

if other is empty:

return

loop over all the nodes in other map:

create a new node in current map with the same data as the node in the other map

return current map

**deconstructor**

loop over all the nodes:

delete the pointer to the current the node

**assignment operator(other):**

if the current map points to same location as other:

return current map

loop over all the nodes:

delete the pointer to the current the node

set dummy to a new node

set size to the other map’s size

if other is empty:

return current map

loop over all the nodes in other map:

create a new node in current map with the same data as the node in the other map

return current map

**insert(key, value)**

call and return insertOrUpdate with insert true and update false

**update(key, value)**

call and return insertOrUpdate with insert false and update true

**insertOrUpdate(key, value)**

call insertOrUpdate with insert true and update true

return true

**insertOrUpdate(key, value, insert, update)**

loop over all the nodes either until the end or until we are just before where key needs to be inserted:

if the key is already inserted but we are allowed to update it:

update the value

return true

if the key is already inserted but we aren’t allowed to update it

return false

if we break from the loop and we are supposed to update but not insert:

return false

create a new temporary node with our desired key/value and insert it into the appropriate location in the linked list

**erase(key)**

loop over all the nodes either until the end or until we are just before the node we need to delete

if the current node is the end of the list:

return false

delete the next node after our current one and shift the linked list over

**contains(key)**

create a temporary value variable

return the output of the 2 parameter get function

**get(key, value)**

loop over all the nodes until the end:

if the current node has the desired key:  
 set value to the nodes value

return true

return false

**get(i, key, value)**

if i is out of range:

return false

create a counter variable

loop over all the nodes either until the end or until the counter variable equals our desired i value:

increment the counter

set key/value to the key/value of the found node

**swap(m, other)**

swap the current map and other using a temporary variable and the copy constructor/assignment operator

**merge(m1, m2, result)**

create a return variable to return true by default

set result to an empty map

loop over the pairs of m1:

use the 3 parameter get function to get the ith pair of m1

insert the ith pair of m1 into result

loop over the pairs of m2:

use the 3 parameter get function to get the ith pair of m2

try to insert the ith pair of m2 into result

if the insertion wasn’t successful and if the value associated with the current key isn’t the same in both result and m2:

set the return variable false

remove the pair containing the current key from result

return the return variable

**reassign(m, result)**

set left to 0

set right to the index of m

if m has 0 or 1 elements:

set result to m using the assignment operator

return

set result to an empty map

while left is less than right:

get the pair at left-th element of m

get the pair at right-th element of m

insert into result a pair containing the left-th element’s key and right-th element’s value

insert into result a pair containing the right-th element’s key and left-th element’s value

increment left

decrement right

if left and right are equal:

get the pair at 0thth element of m

get the pair at middle element of m

insert into result a pair containing the middle element’s key and 0th element’s value

update the value of the pair containing the 0th element’s key with the middle element’s key

**Test Cases**

bool testReassign(Map m, Map result)

{

*// ensure sizes are the same*

bool valid = m.size() == result.size();

*// loop over m*

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

{

KeyType k;

ValueType v;

m.get(i, k, v);

ValueType v\_result;

result.get(k, v\_result);

valid = valid && v != v\_result; *// ensures values are different in m and result*

}

*return* valid;

}

*// default constructor*

Map m0;

*// EMPTY MAP:*

assert(m0.empty()); *// test map is empty*

assert(m0.size() == 0); *// test size is 0*

assert(!m0.erase("pop")); *// test erase*

assert(!m0.update("pop", -1)); *// test update*

*// test insert (all keys are valid)*

KeyType k = "pop";

ValueType v = -999.9;

assert(m0.insert(k, v));

assert(m0.insert("abc", 0));

assert(m0.insert("bcd", 2));

assert(m0.insert("zzz", 6));

assert(m0.insert("cde", 3));

assert(m0.insert("aabc", 1));

assert(m0.insert("z", 5));

assert(m0.insert("xyz", 4));

*// ensure duplicates aren't inserted*

assert(!m0.insert("abc", -1));

assert(!m0.insert("xyz", -2));

assert(!m0.insert("zzz", -3));

KeyType k1 = "NULL";

ValueType v1 = -999.9;

assert(!m0.get("test", v1) && v1 == -999.9); *// test get with non existing key*

assert(!m0.get(-1, k1, v1) && !m0.get(m0.size(), k1, v1) && k1 == "NULL" && v1 == -999.9); *// test get with invalid i parameter*

assert(m0.get("abc", v1) && v1 == 0); *// test get with existing key*

*// test get with valid indices*

m0.get(0, k1, v1);

assert(m0.get(0, k1, v1) && k1 == "aabc" && v1 == 1);

assert(m0.get(3, k1, v1) && k1 == "cde" && v1 == 3);

assert(m0.get(7, k1, v1) && k1 == "zzz" && v1 == 6);

assert(!m0.update("test", -1)); *// test update with non existing key*

*// test update with existing keys*

assert(m0.update(k, 14) && m0.get(k, v1) && v1 == 14);

assert(m0.update("xyz", -4) && m0.get("xyz", v1) && v1 == -4);

assert(m0.insertOrUpdate("xyz", 4) && m0.get("xyz", v1) && v1 == 4 && m0.size() == 8); *// test insertOrUpdate with existing key*

assert(m0.insertOrUpdate("frog", 17) && m0.get("frog", v1) && v1 == 17 && m0.size() == 9); *// test insertOrUpdate with new key*

assert(m0.erase("frog") && !m0.contains("frog") && m0.size() == 8); *// test erase with existing key*

*// test contains with existing keys*

assert(m0.contains("cde"));

assert(m0.contains("xyz"));

assert(!m0.empty() && m0.size() == 8); *// test empty and size of non empty array*

assert(m0.contains(k) && m0.erase(k) && !m0.contains(k) && m0.size() == 7); *// test erase with existing key*

assert(!m0.erase(k)); *// test erase with non existing key*

*// test erase with more existing keys*

assert(m0.erase("cde") && m0.erase("xyz"));

assert(!m0.contains("cde") && !m0.contains("xyz") && m0.size() == 5);

Map m1;

bool inserts = true;

int m1\_size = 300;

*// insert 300 values into m1*

*for* (int i = 0; i < m1\_size; i++)

{

inserts = inserts && m1.insert("data" + std::to\_string(i + 100), i + 100);

}

*// assure all inserts were successful*

assert(inserts);

assert(m1.size() == m1\_size);

Map m2;

m2.insert("a", 0);

m2.insert("b", 1);

m2.insert("c", 2);

int m1\_init\_size = m1.size();

int m2\_init\_size = m2.size();

m2.swap(m1);

assert(m2.size() == m1\_init\_size && m1.size() == m2\_init\_size); *// test swap sizes*

KeyType k2;

ValueType v2;

bool m2\_values = true;

*// loop over m2 to ensure its pairs match that of m1*

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

{

m2\_values = m2\_values && m2.get(i, k2, v2) && k2 == "data" + std::to\_string(i + 100) && v2 == i + 100;

}

assert(m2\_values); *// assert the m2 pairs are correct*

*// assert the m1 pairs are correct*

assert(m1.get(0, k2, v2) && k2 == "a" && v2 == 0);

assert(m1.get(1, k2, v2) && k2 == "b" && v2 == 1);

assert(m1.get(2, k2, v2) && k2 == "c" && v2 == 2);

Map \*m3 = new Map(m0); *// set m3 to m0 using copy constructor*

KeyType k3a;

ValueType v3a;

KeyType k3b;

ValueType v3b;

bool m3\_values = true;

*// TEST COPY CONSTRUCTOR*

*// loop over m0 to ensure its pairs match that of m3*

*for* (int i = 0; i < m0.size(); i++)

{

m3\_values = m3\_values && m0.get(i, k3a, v3a) && m3->get(i, k3b, v3b);

m3\_values = m3\_values && k3a == k3b && v3a == v3b;

}

assert(m3\_values); *// assert m3 pairs are correct*

*// TEST ASSIGNMENT OPERATOR*

\*m3 = m0; *// set m3 to m0 using assignment operator*

*// loop over m0 to ensure its pairs match that of m3*

*for* (int i = 0; i < m0.size(); i++)

{

m3\_values = m3\_values && m0.get(i, k3a, v3a) && m3->get(i, k3b, v3b);

m3\_values = m3\_values && k3a == k3b && v3a == v3b;

}

assert(m3\_values); *// assert m3 pairs are correct*

*// TEST DECONSTRUCTOR*

delete m3; *// delete m3*

*// loop over m0 to ensure the deletion of m3 had no effect on m0*

*for* (int i = 0; i < m0.size(); i++)

{

m3\_values = m3\_values && m0.get(i, k3a, v3a);

}

assert(m3\_values); *// assert m0 pairs still exist*

Map m5;

int m5\_size = 9;

*// insert 9 values into m1*

*for* (int i = 0; i < m5\_size; i++)

{

inserts = inserts && m5.insert("data" + std::to\_string(i), i);

}

Map m6;

reassign(m5, m6); *// reassigns m5 into m6*

assert(testReassign(m5, m6)); *// test reassign*

Map m7;

m7.insert("dog", 0);

m7.insert("cat", 1);

m7.insert("frog", 2);

m7.insert("worm", 3);

Map m8;

m8.insert("bird", 4);

m8.insert("bear", 5);

m8.insert("ant", 6);

m8.insert("toad", 7);

Map m9;

assert(merge(m7, m8, m9) && m9.size() == m7.size() + m8.size()); *// test merge with no overlaps*

*// test merge with overlap that has same value in both m7 and m8*

m7.insert("bird", 4);

assert(merge(m7, m8, m9) && m9.size() == m7.size() + m8.size() - 1);

*// test merge with overlap that has different values in m7 and m8*

m7.insert("bear", -999);

assert(!merge(m7, m8, m9) && m9.size() == m7.size() + m8.size() - 3);