Jan 23, 2016 **CS 32 Project 2 Report** Karen Li

**Doubly Linked List Design:**

My doubly linked list implementation did not incorporate a dummy node nor is it circular. Rather, it is similar to a typical singly linked list except for the fact that each list node contains both pointers to the previous node and the next node. As well, each node holds the key and the value for the key value pairs in the Map. The Map itself contains a head and a tail pointer to the list as well as the number of nodes in the list. The nodes are in the order that were inserted into the Map, such that the head pointer points to the first node inserted to the Map and the tail pointer points to the most recently inserted node.

A typical map:

head

KEY1

VALUE1

NEXT

PREV

PREV

NEXT

VALUE2

PREV

NEXT

VALUE3

KEY3

KEY2

m\_numNodes = 3

nullptr

tail

nullptr

An empty map: head 🡪 nullptr, tail 🡪 nullptr, m\_numNodes = 0;

**Pseudocode:**

bool Map::insert(const KeyType& key, const KeyType &value)

{

if the key is already in the map, return false

if the map is empty

Initialize a new node, update map’s member variables, and return true

else

Initialize a new node, attach to current last node, and return true

}

bool Map::update(const KeyType& Key, const ValueType& value)

{

iterate through the list

if the node containing the key is found, change the value and return true

return false (since the key wasn’t found)

}

bool Map::insertOrUpdate(const KeyType& key, const ValueType& value)

{

call update() with key and value (assuming the key is in the map)

call insert() with key and value (since the key is not in the map)

return true

}

bool Map::erase(const KeyType& key)

{

if the key isn’t found in the map, return false;

if the map only has one node, delete the node, and return true

if erasing the first node in the list

delete the first node, set the second node as head, and return true

if erasing the last node in the list

delete the last node, set the second last node as tail, and return true

else, iterate through the list to find the node we want to delete

link the two nodes above and below the node

delete the node and return true

}

bool Map::contains(const KeyType& key) const

{

iterate through the nodes in the list

return true if a node containing the key is found

return false (if the key isn’t found)

}

bool Map::get(const KeyType& key, ValueType& value) const

{

iterate through the list

if the node containing the key is found, set the value and return true

return false (if the key isn’t found)

}

bool Map::get(int i, KeyType& key, ValueType& value) const

{

return false if i is invalid

iterate i times through the list so iterator points to the (i+1)th node

get the data from the node and use them to initialize the last two parameters

return true

}

void Map::swap(Map& other)

{

exchange the number of nodes in the maps

exchange the head and tail pointers to the maps

}

Map::~Map()

{

iterate through all of the nodes

delete the node

}

Map::Map(const Map& source)

{

in the initialization list, call Map’s default constructor to make an empty map

loop through the source Map

get key value pairs from the source node

insert them into the map being constructed

}

Map& Map::operator=(const Map& source)

{

if trying to assign a map to itself, return \*this

iterate through all of the left-hand Map and delete the nodes one by one

reinitialize member variables in left-hand Map so its empty

loop through all of the nodes in the source Map

get key value pairs from the source nodes

insert them into the left-hand Map

return \*this

}

bool combine(const Map& m1, const Map& m2, Map& result)

{

create a new empty map

use a Boolean variable to record whether all key value pairs are valid (starts true)

loop through m1 and get the keys and values

if the key is contained in m2

if the key has the same value in m1 and m2, insert

else, don’t insert and set the Boolean variable to false

else, insert into the new map

loop through m2

insert only keys that are unique to m2 into the new map

assign the new map to result and return the Boolean variable

}

void subtract(const Map& m1, const Map& m2, Map& result)

{

create a new empty map

loop through m1 and get the key and value from m1

continue if the key is also contained in m2

insert the key value pair into the new map

assign the new map to result

}

**Test Cases:**

All of the following test cases were performed on a map from strings to doubles.

// default constructor

Map KarensWorld;

// test to see that map is created empty

assert(KarensWorld.empty());

cout << KarensWorld.size() << endl; // should print 0

// test update() on an empty map

assert( ! KarensWorld.update("Mark Watney", 206));

// test insert()

if (KarensWorld.insert("Carter", 18))

cout << "Carter is Karen's friend!" << endl;

cout << KarensWorld.size() << endl; // should print 1

if (KarensWorld.insert("Tanya", 18))

cout << "Tanya is Karen's friend!" << endl;

cout << KarensWorld.size() << endl; // should print 2

assert(KarensWorld.contains("Carter") && KarensWorld.contains("Tanya"));

assert( ! KarensWorld.contains("Waldo"));

// test update(), insertOrUpdate(), and get() with two parameters

assert(KarensWorld.update("Carter", 19));

double testDouble;

if (KarensWorld.get("Carter", testDouble))

cout << testDouble << endl; // should print 19

cout << KarensWorld.size() << endl; // should print 2

assert(!KarensWorld.get(“Stale Sandbech”, 22));

assert(KarensWorld.insertOrUpdate("Kylie", 18));

assert(KarensWorld.insertOrUpdate("Jessica", 18));

if (KarensWorld.get("Jessica", testDouble))

cout << testDouble << endl; // should print 18

assert(KarensWorld.insertOrUpdate("Jessica", 19));

if (KarensWorld.get("Jessica", testDouble))

cout << testDouble << endl; // should print 19

cout << KarensWorld.size() << endl; // should print 4

// test get() with three parameters

string testString;

for (int i = 0; i < KarensWorld.size(); i++) // should print Carter 19, Tanya 18, Kylie 18, Jessica 19

{

KarensWorld.get(i, testString, testDouble);

cout << "Key: " << testString << endl;

cout << "Value: " << testDouble << endl;

}

cout << "Karen's World finished printing." << endl;

// test the copy constructor

Map KarensImagination(KarensWorld);

assert( ! KarensImagination.empty());

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

for (int i = 0; i < KarensImagination.size(); i++) // should print Carter 19, Tanya 18, Kylie 18, Jessica 19

{

KarensImagination.get(i, testString, testDouble);

cout << "Key: " << testString << endl;

cout << "Value: " << testDouble << endl;

}

cout << "Karen's Imagination finished printing." << endl;

assert(KarensImagination.update("Carter", 22)); // make some changes to the map created w/ copy constructor

assert(KarensImagination.insertOrUpdate("Santa", 60));

for (int i = 0; i < KarensWorld.size(); i++) // should print Carter 19, Tanya 18, Kylie 18, Jessica 19 since it is the original map

{

KarensWorld.get(i, testString, testDouble);

cout << "Key: " << testString << endl;

cout << "Value: " << testDouble << endl;

}

cout << "Karen's World finished printing." << endl;

for (int i = 0; i < KarensImagination.size(); i++) // should print Carter 22, Tanya 18, Kylie 18, Jessica 19, Santa 60

{

KarensImagination.get(i, testString, testDouble);

cout << "Key: " << testString << endl;

cout << "Value: " << testDouble << endl;

}

cout << "Karen's Imagination finished printing." << endl;

// test the assignment operator

Map KarensLife;

KarensLife.insert("Shawn Mendes", 18);

assert(KarensLife.size() == 1 && !KarensLife.empty());

KarensLife = KarensImagination;

assert(KarensLife.size() == 5 && KarensLife.contains("Santa"));

for (int i = 0; i < KarensLife.size(); i++) // should print Carter 22, Tanya 18, Kylie 18, Jessica 19, Santa 60

{

KarensLife.get(i, testString, testDouble);

cout << "Key: " << testString << endl;

cout << "Value: " << testDouble << endl;

}

cout << "Karen's Life finished printing." << endl;

KarensLife = KarensLife; // test for self assignment! should just return itself

assert(KarensLife.size() == 5 && KarensLife.contains(“Santa"));

for (int i = 0; i < KarensLife.size(); i++) // should print Carter 22, Tanya 18, Kylie 18, Jessica 19, Santa 60

{

KarensLife.get(i, testString, testDouble);

cout << "Key: " << testString << endl;

cout << "Value: " << testDouble << endl;

}

cout << "Karen's Life finished printing, even after being assigned to itself!" << endl;

// test insertOrUpdate() on an empty map

Map KarensFeels;

KarensFeels.insertOrUpdate("Happy", 100);

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

KarensFeels.insertOrUpdate("Sad", -100);

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

KarensFeels.insertOrUpdate("Happy", 101);

assert(KarensFeels.size() == 2 && KarensFeels.contains("Happy") && KarensFeels.contains("Sad") && !KarensFeels.contains("Mad"));

// test the erase() on a totally new map

Map KarensTunes;

assert(KarensTunes.size() == 0);

KarensTunes.erase("Sunday Morning"); // trying to erase something in an empty map (CASE 0)

KarensTunes.insert("This Love", 3); // only one key in the map

KarensTunes.erase("Blank Space"); // trying to erase a key that isn't there (CASE 0)

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

KarensTunes.erase("This Love"); // delete the only key in the map (CASE 1)

assert(KarensTunes.size() == 0 && KarensTunes.empty());

KarensTunes.insert("Drops of Jupiter", 6);

KarensTunes.insertOrUpdate("Locked Away", 7);

KarensTunes.insert("Tenerife Sea", 10);

KarensTunes.insertOrUpdate("Love On Top", 20);

KarensTunes.insert("Stitches", 4);

assert(KarensTunes.contains("Tenerife Sea") && KarensTunes.size() == 5);

KarensTunes.erase("Drops of Jupiter"); // delete the first key in the map (CASE 2)

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

for (int i = 0; i < KarensTunes.size(); i++) // should print Locked Away 7, Tenerife Sea 10, Love On Top 20, Stitches 4

{

KarensTunes.get(i, testString, testDouble);

cout << "Key: " << testString << endl;

cout << "Value: " << testDouble << endl;

}

cout << "Karen's Tunes have finished printing." << endl;

KarensTunes.erase("Stitches"); // delete the last key in the map (CASE 3)

assert(KarensTunes.size() == 3 && !KarensTunes.contains("Stitches") && !KarensTunes.contains("Drops of Jupiter"));

KarensTunes.update("Locked Away", 14);

for (int i = 0; i < KarensTunes.size(); i++) // should print Locked Away 14, Tenerife Sea 10, Love On Top 20

{

KarensTunes.get(i, testString, testDouble);

cout << "Key: " << testString << endl;

cout << "Value: " << testDouble << endl;

}

cout << "Karen's Tunes have finished printing." << endl;

KarensTunes.erase("Tenerife Sea"); // delete a key in the middle of the map (CASE 4)

assert(KarensTunes.size() == 2 && KarensTunes.contains("Locked Away") && !KarensTunes.contains("Tenerife Sea"));

for (int i = 0; i < KarensTunes.size(); i++) // should print Locked Away 14, Love On Top 20

{

KarensTunes.get(i, testString, testDouble);

cout << "Key: " << testString << endl;

cout << "Value: " << testDouble << endl;

}

cout << "Karen's Tunes have finished printing." << endl;

// test subtract() using the spec’s example

Map m1;

m1.insert("Fred", 123);

m1.insert("Ethel", 456);

m1.insert("Lucy", 789);

Map m2;

m2.insert("Rick", 321);

m2.insert("Ethel", 654);

m2.insert("Lucy", 789);

Map m3;

m3.insert("Joey", 420); // the parameter for result is not empty

subtract(m1, m2, m3);

for (int i = 0; i < m3.size(); i++) // should print Fred 123

{

m3.get(i, testString, testDouble);

cout << "Key: " << testString << endl;

cout << "Value: " << testDouble << endl;

}

cout << "m3 has finished printing." << endl;

subtract(m1, m2, m1); // use aliasing (m1 and result both refer to m1)

for (int i = 0; i < m1.size(); i++) // should print Fred 123

{

m1.get(i, testString, testDouble);

cout << "Key: " << testString << endl;

cout << "Value: " << testDouble << endl;

}

cout << "m1 has finished printing." << endl;

subtract(m2, m1, m2); // use aliasing (m1 and result both refer to m2)

for (int i = 0; i < m2.size(); i++) // should print Rick 321, Ethel 654, Lucy 789

{

m2.get(i, testString, testDouble);

cout << "Key: " << testString << endl;

cout << "Value: " << testDouble << endl;

}

cout << "m2 has finished printing." << endl;

m1.insert("Rick", 420);

m1.insert("Jonny", 456);

// test the swap() function

m1.swap(m2);

for (int i = 0; i < m2.size(); i++) // should print Fred 123, Rick 420, Jonny 456

{

m2.get(i, testString, testDouble);

cout << "Key: " << testString << endl;

cout << "Value: " << testDouble << endl;

}

cout << "Swapped m2 has finished printing." << endl;

// continue testing the subtract() function post swap

subtract(m2, m1, m2); // use aliasing (m1 and result both refer to m2)

for (int i = 0; i < m2.size(); i++) // should print Fred 123, Jonny 456

{

m2.get(i, testString, testDouble);

cout << "Key: " << testString << endl;

cout << "Value: " << testDouble << endl;

}

cout << "m2 has finished printing." << endl;

// test combine()

Map n1;

n1.insert("Fred", 123);

n1.insert("Ethel", 456);

n1.insert("Lucy", 789);

Map n2;

n2.insertOrUpdate("Lucy", 789);

n2.insertOrUpdate("Ricky", 321);

Map n3;

n3.insert("Kramer", 27); // n3 isn't empty before we pass it into combine()

assert(combine(n1, n2, n3) == true);

for (int i = 0; i < n3.size(); i++) // should print Fred 123, Ethel 456, Lucy 789, Ricky 321

{

n3.get(i, testString, testDouble);

cout << "Key: " << testString << endl;

cout << "Value: " << testDouble << endl;

}

cout << "n3 has been combined and printed." << endl;

Map n4;

n4.insert("Lucy", 654);

n4.insert("Ricky", 321);

n2.swap(n4);

assert(!combine(n1, n2, n3));

for (int i = 0; i < n3.size(); i++) // should print Fred 123, Ethel 456, Ricky 321

{

n3.get(i, testString, testDouble);

cout << "Key: " << testString << endl;

cout << "Value: " << testDouble << endl;

}

cout << "n3 has been combined and printed." << endl;

// test combine() with aliasing

Map n5;

n5.insert("Daniel", 234);

n5.insert("Fred", 123);

assert(combine(n1, n5, n1));

for (int i = 0; i < n1.size(); i++) // should print Fred 123, Ethel 456, Lucy 789, Daniel 234

{

n1.get(i, testString, testDouble);

cout << "Key: " << testString << endl;

cout << "Value: " << testDouble << endl;

}

cout << "n1 has been combined and printed." << endl;

n5.update("Daniel", 466);

n5.insertOrUpdate("Ethel", 498);

n5.insertOrUpdate("Spongebob", 123);

assert(!combine(n5, n1, n5));

for (int i = 0; i < n5.size(); i++) // should print Fred 123, Spongebob 123, Lucy 789

{

n5.get(i, testString, testDouble);

cout << "Key: " << testString << endl;

cout << "Value: " << testDouble << endl;

}

cout << "n5 has been combined and printed." << endl;

assert(combine(n5, n5, n5));

for (int i = 0; i < n5.size(); i++) // should print Fred 123, Spongebob 123, Lucy 789

{

n5.get(i, testString, testDouble);

cout << "Key: " << testString << endl;

cout << "Value: " << testDouble << endl;

}

cout << "n5 has been recombined and printed." << endl;

// test ~Map() by creating both a constructor and a destructor for a Node that print out a message whenever a node is created or destroyed and then counting to see if the number of messages printed by both are equal

struct Node {

…

Node() {

std::cerr << “CONSTRUCTOR!” << std::endl;

}

~Node() {

std::cerr << “DESTRUCTOR!” << std::endl;

}