**1. Description**

My implementation of the doubly-linked list is not circular. The “previous” pointer in the first node and the “next” pointer in the last node are both null pointers, indicating the beginning and end of the linked list. My implementation consists of a Map class that contains a private struct object called mapNode which contains the data pair values (KeyType key and ValueType value) as well as the next and previous pointers. The other private members of Map are the head pointer and tail pointer and an integer variable containing the size of the map (# of nodes). My implementation does not contain a dummy node and the nodes are added to the end of the linked list.

**2. Pseudocode**

Map::~Map()

{

Call destructor only if nodes exist

If there is only one node

Delete it

Otherwise, if there are more than one node

Iterate through the nodes

If we are not on the last node, delete the nodes

Delete the last node

{

Map::Map(const Map &other)

{

Initialize new map’s size to 0

Iterate through other map’s nodes

Insert other map’s nodes into new map

}

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

{

Check that the left hand value does not equal right hand value

While this map is not empty

Iterate through this map’s nodes and delete them

Set this map’s head and tail pointers to null

Set this map’s size to 0

Iterate through other map

Insert each of other map’s nodes into this map

Return this map

}

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

{

If map is empty

Return false

If i is greater than/equal to zero and less than map’s size

Move a temporary pointer to the ith node

Set parameter key and value to the node’s key and value

Return true when finished

Otherwise, return false b/c i was negative or greater than/equal to map’s size

}

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

{

If map is empty

Increment map’s size

Dynamically allocate new node, initialize its values, and have head point to it

Set node’s next and prev pointers and tail pointer accordingly

Return true

If map is not empty and key matches existing key

Change existing key’s value to the parameter value

Return true

If map is not empty and no matching key

Increment map’s size

Add the new node after tail and initialize its values

Set tail pointer and the next/prev pointers accordingly

Return true

}

bool Map::erase(const KeyType& key)

{

If map contains key

If map only contains one node

Delete the node

Set the head and tail pointer accordingly

Decrement map’s size and return true

Otherwise, iterate through map

If erasing the first node

Set all pointers accordingly and delete the node

Decrement map’s size and return true

If erasing last node

Set all pointers accordingly and delete the node

Decrement map’s size and return true

If erasing any node in between

Set all pointers accordingly and delete the node

Decrement map’s size and return true

}

void Map::swap(Map& other)

{

Create a temporary variable to hold this map’s size

Create two temporary mapNode pointers

Point them to what this map’s head and tail pointers point to

Set this map’s head and tail pointers to other map’s head and tail pointers

Set this map’s size to other map’s size

Set other map’s head and tail pointers to what the temporary mapNode pointers point to

Set other map’s size to this map’s size from temporary variable

}

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

{

Create a copy of map 1 and map 2 to prevent aliasing issues

If result map is not empty

Erase its nodes

Create bool variable to true indicating return value

Iterate through first map’s nodes

Get each node’s key/value pairs

If the second map does not contain a matching key

Insert the key/value pair into result

Otherwise, if second map contains a matching key

Compare the first map key’s value to second map key’s value

If they are equal, insert the key/value pair into result

Otherwise, do nothing and set bool variable to false

Iterate through second map’s nodes

Get each node’s kay/value pair

If the first map does not contain a matching key

Insert the key/value pair into result

Return the value from bool variable

}

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

Create copies of map 1 and map 2 to prevent aliasing issues

If result map is not empty

Erase its nodes

Iterate through first map’s nodes

Get each node’s key/value pair

If the second map does not contain a matching key

Insert the key/value pair into result

}

**3. Test Cases**

void test1() // Test constructor, accessors, insert/update/insertOrUpdate/erase

{

Map m1;

assert(m1.empty() && m1.size() == 0); // Test that constructor initialized m\_size to 0 and

empty() and size() functions work

assert(! m1.contains("")); // Test that contains returns false when no matches

assert(m1.insert("Fred", 123)); // Test insert function which calls insertOrUpdate and creates first node

assert(m1.insert("Ethel", 456)); // Test insert creates second node

ValueType t;

assert(! m1.insert("Ethel", 789) && m1.get("Ethel", t) && t == 456); //Test that insert does nothing and returns false when key already exists

assert(m1.size() == 2); // Test that two nodes were created

ValueType v = 42;

assert(! m1.get("Lucy", v) && v == 42); // Test that get returns false and does nothing when no match

assert(m1.get("Fred", v) && v == 123); // Test that get returns true and changes the ValueType passed in to the value of key/value pair

v = 42;

KeyType x = "Lucy";

assert(m1.get(0, x, v) && ((x == "Fred" && v == 123))); // Test that get changed parameter key/value to node's key/value

KeyType x2 = "Ricky";

assert(m1.get(1, x2, v) && (x2 == "Ethel" && v == 456) && x != x2); // Test that get does the correct things

assert(m1.update("Fred", 321) && m1.get("Fred", v) && v == 321); // Test that update works correctly

assert(m1.insertOrUpdate("Fred", 111) && m1.get("Fred", v) && v == 111); // Test insertOrUpdate updates value of existing key

assert(m1.insertOrUpdate("Lucy", 789) && m1.get(2, x, v) && v == 789 && x == "Lucy"); // Test insertOrUpdate inserts the key/value pair

assert(m1.erase("Fred") && (! m1.get("Fred", v)) && v == 789); // Test that erase correctly erases first node

assert(m1.erase("Lucy") && (! m1.get("Lucy", v)) && v == 789); // Test that erase correctly erases last node

assert(m1.get(0, x, v) && x == "Ethel" && v == 456); // Test that the only node is the one with Ethel/456 pair

assert(! m1.erase("Ricky") && m1.get(0, x, v) && x == "Ethel" && v == 456); // Test that erase does not do anything w/ non-matching key

assert(m1.erase("Ethel") && m1.empty()); // Test that erase erases last node

}

void test2() // Test copy constructor/ assignment operator/ swap/ combined/ and subtract

{

Map m1;

m1.insert("A1", 123);

m1.insert("B2", 456);

m1.insert("B3", 789);

Map m2;

m2.insert("B1", 123);

m2.insert("B2", 456);

m2.insert("B3", 987);

Map m3;

m3.insert("C1", 123);

m3.insert("C2", 456);

m3.insert("C3", 789);

Map result;

assert(result.empty() && result.size() == 0); // Check that result is empty

m1.swap(result);

assert(m1.empty() && ! result.empty()); // Test swap function

m1.swap(result);

assert(! m1.empty() && result.empty()); // Test swap function

KeyType k;

ValueType v;

Map m4 = m3;

Map m5 = result;

assert(! m3.empty() && m4.get(0, k, v) && k == "C1" && v == 123); // Test copy constructor

assert(m5.empty());

m5 = m3; // Test assignment operator

assert(! m5.empty() && m5.get(0, k, v) && k == "C1" && v == 123);

m5 = result; // Test assignment operator works on empty maps

assert(m5.empty());

assert(! combine(m1, m2, result) && result.size() == 3); // Test combine function w/ same key diff value pair

assert(! combine(m1, m2, result) && result.size() == 3); // Test combine on non empty result map

assert(combine(m3, m4, result) && result.size() == 3); // Test combine on same key/value pairs

subtract(m1, m2, result); // Test subtract function

assert(result.size() == 1 && result.get(0,k,v) && k == "A1" && v == 123);

subtract(m3, m4, m5); // Test subtract function with all matching key/value pairs

assert(m5.size() == 0);

}

int main()

{

test1();

cout << "Passed test 1" << endl << endl;

test2();

cout << "Passed test 2" << endl << endl;

}