**Bonnie Liu** (005300989)

CS 32 Discussion 1F

Project 2: Double Trouble Report

Due Date: January 28, 2020

**Description of Design**

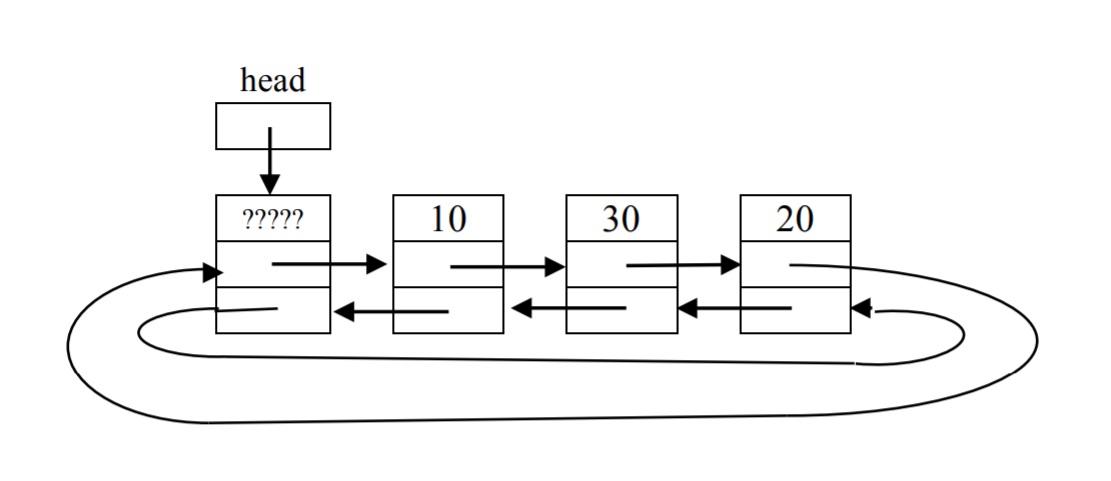
My general approach to this project was using a circular doubly-linked list with a dummy node. I declared a struct called Pair in my Map.h file. Each Pair holds a key, value, pointer to previous node, and pointer to next node. Additional nodes are added to the end of the linked list (away from the header).

This is what an empty linked list looks like under this implementation:

A drawing of a person

Description automatically generated

This is what a typical linked list looks like under this implementation:



Note: All picture credits go to Professor David A. Smallberg.

**Pseudocode for Non-Trivial Algorithms**

Map::Map() // constructor

{

Create head pointer and dummy node

Set dummy node’s next and previous pointers to itself

}

Map::Map(const Map& other) // copy constructor

{

For each “interesting” Pair in other,

Insert a copy of that Pair into this Map

}

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

{

If right-hand side and left-hand side are the same Map,

Return Map;

Clear this Map so that it only has dummy node

For each “interesting” Pair in rhs,

Insert a copy of that Pair into this Map

Return this Map

}

Map::~Map()

{

Loop through linked list and delete all the “interesting” nodes

Delete dummy node

}

int Map::size() const

{

Set count to 0

For every “interesting” element of the linked list,

Add 1 to count

Return count

}

bool Map::erase(const KeyType& key)

{

Find location of node with key in linked list

If key is not in linked list,

Return false

Update next and previous pointers for affected nodes (since key is in our list)

Delete Pair with KeyType key

Return true

}

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

{

Find location of node with key in linked list

If key is not in linked list,

Return false

Update value parameter otherwise

Return true

}

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

{

If i is negative or i is greater than or equal to size

Return false and leave key and value parameters unchanged

Loop to find the ith element of linked list

Copy desired Pair’s key and value into key and value parameters

Return true

}

void Map::swap(Map& other)

{

Swap head pointers using temporary pointer to Pair  
}

bool Map::doInsertOrUpdate(const KeyType& key, const ValueType& value, bool mayInsert, bool mayUpdate) // called in insert(), update(), and insertOrUpdate() with appropriate boolean parameters

{

Find location of node with key in linked list

If key has been found,

{

If we can update value parameter, do so and return true

Else return false

}

If we can’t insert either,

Return false

Create new node using key and value parameters

Connect new node to linked list and reconnect affected nodes

Return true

}

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

{

If result is not the same Map as either m1 or m2,

Clear result

For each “interesting” m1 and m2 Pair,

Try to insert the Pair into the result Map

If the Pair has a key that is already in result AND the corresponding Pair in result has a different value than the Pair in m1,

Be prepared to return false;

Return whether the instance mentioned above occurred

}

void reassign(const Map& m, Map& result)

{

If m and result are not the same Map,

Clear result

Store key of last node in temporary variable

Store value of first node in temporary variable

For each “interesting” node in m,

Shift the value of the nth node to the (n+1)th node

Update value of last node so that it stores the original value of the first node

}

**List of Test Cases**

The tests were performed on a Map from strings to doubles:

// declare variables to use later in get functions

KeyType key;

ValueType value;

// default constructor

Map a;

// for an empty Map

assert(a.size() == 0); // test size

assert(a.empty()); // test empty

assert(!a.get("Annie", value)); // test get function on empty Map

assert(!a.get(1, key, value)); // test get function on empty Map

assert(!a.contains("Annie")); // test contains

assert(a.insert("Annie", 1)); // insert a Pair

assert(a.get("Annie", value) && value == 1); // test if insertion succeeded

assert(a.get(0, key, value) && key == "Annie" && value == 1); // other way to test insertion

assert(a.size() == 1); // check size after insertion

assert(!a.contains("Bonnie")); // check contains

assert(!a.empty()); // check Map not empty

assert(!a.insert("Annie", 3)); // check cannot insert same key

assert(!a.insert("Annie", 1)); // check cannot insert same key

assert(a.update("Annie", 25)); // test update

assert(!a.get(1, key, value)); // check cannot go out of bounds

assert(!a.get("Bonnie", value)); // check get fails when it’s supposed to

assert(a.insertOrUpdate("Bonnie", 22)); // test insertOrUpdate (insert case)

assert(a.get("Bonnie", value) && value == 22); // check insertOrUpdate worked

assert(a.get(1, key, value) && key == "Bonnie" && value == 22); // check again

assert(!a.update("Tony", 0)); // check update fails when key not found

assert(a.size() == 2); // check size

assert(a.contains("Annie")); // test contains

assert(a.contains("Bonnie")); // test contains

assert(!a.contains("Tony")); // check nothing was added when it wasn’t supposed to

assert(a.size() == 2); // check size again

// create another empty Map and end some elements to it

Map parents;

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

assert(parents.empty());

assert(!parents.update("Sunny", 55));

assert(parents.insertOrUpdate("Sunny", 53));

assert(!parents.update("Tony", 54));

assert(parents.update("Sunny", 55));

assert(parents.insertOrUpdate("Tony", 54));

// test copy constructor

Map b(a);

assert(b.size() == 2); // check size was copied correctly

assert(!b.empty()); // check new Map is not empty

// check if member functions work properly on a Map that was copy constructed

assert(b.insertOrUpdate("Annie", 35));

assert(!b.insert("Annie", 98));

assert(b.update("Bonnie", 18));

assert(b.insert("Conny", 13));

assert(b.get("Conny", value) && value == 13);

assert(b.size() == 3);

assert(!b.empty());

// test assignment operator

Map c = a;

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

assert(!c.empty());

assert(!c.contains("Conny"));

assert(!c.insert("Annie", 12));

assert(!c.insert("Bonnie", 12));

assert(!c.update("Conny", 12));

// check if member functions work properly on a Map that used assignment operator

assert(c.insertOrUpdate("Bonnie", 19));

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

assert(c.insertOrUpdate("Tony", 54));

assert(c.insertOrUpdate("Sunny", 55));

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

assert(!c.empty());

// create empty Map

Map empty;

assert(empty.empty());

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

// see if copy constructor works when copying an empty Map

Map emptyCopy(empty);

assert(emptyCopy.empty());

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

// try out other member functions on Map that was copy constructed from empty Map

emptyCopy.insertOrUpdate("Sara", 18);

assert(!emptyCopy.empty());

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

// test assignment operator on Map that was constructed using copy constructor

emptyCopy = empty;

assert(emptyCopy.empty());

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

assert(!emptyCopy.contains("Sara"));

// create another empty Map

Map single;

assert(single.empty());

assert(single.insert("key", 1));

// use non-empty Map to copy construct something

Map anotherSingle(single);

assert(anotherSingle.contains("key"));

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

assert(anotherSingle.insertOrUpdate("door", 0));

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

// use non-empty Map for assignment operator

anotherSingle = single;

assert(!anotherSingle.empty());

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

assert(anotherSingle.contains("key"));

assert(!anotherSingle.contains("door"));

assert(!anotherSingle.erase("door"));

assert(anotherSingle.erase("key"));

assert(anotherSingle.empty());

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

// test erase function

assert(!empty.erase("clap"));

assert(c.insert("Conny", 5));

assert(c.size() == 5);

assert(c.erase("Conny"));

assert(c.erase("Bonnie"));

assert(!c.get("Bonnie", value));

assert(!c.get(3, key, value));

assert(c.size() == 3);

// test swap function

swap(c, parents);

assert(parents.size() == 3 && c.size() == 2);

assert(c.get("Sunny", value) && value == 55);

assert(c.get(0, key, value) && key == "Sunny" && value == 55);

assert(c.get(c.size() - 1, key, value) && key == "Tony" && value == 54);

assert(parents.get(0, key, value) && key == "Annie" && value == 25);

assert(parents.get(parents.size() - 1, key, value) && key == "Sunny" && value == 55);

assert(parents.get(parents.size() - 2, key, value) && key == "Tony" && value == 54);

// test combine function using Professor Smallberg’s test cases in the spec

Map m1, m2;

// set up m1 and m2

m1.insertOrUpdate("Fred", 123);

m1.insertOrUpdate("Ethel", 456);

m1.insertOrUpdate("Lucy", 789);

m2.insertOrUpdate("Lucy", 789);

m2.insertOrUpdate("Ricky", 321);

assert(combine(m1, m2, empty));

assert(empty.size() == 4); // check size after combine

assert(empty.contains("Lucy")); // check that Lucy is there

// check each node of linked list

assert(empty.get(0, key, value) && key == "Fred" && value == 123);

assert(empty.get(1, key, value) && key == "Ethel" && value == 456);

assert(empty.get(2, key, value) && key == "Lucy" && value == 789);

assert(empty.get(3, key, value) && key == "Ricky" && value == 321);

assert(!empty.get(4, key, value));

// update m2 to test second scenario: Lucy shouldn’t be there after combine

m2.update("Lucy", 654);

assert(!combine(m1, m2, empty));

assert(empty.size() == 3);

assert(!empty.contains("Lucy")); // Lucy shouldn’t be here

// check each node of linked list

assert(empty.get(0, key, value) && key == "Fred" && value == 123);

assert(empty.get(1, key, value) && key == "Ethel" && value == 456);

assert(empty.get(2, key, value) && key == "Ricky" && value == 321);

// check combine if we try to combine the same Map

assert(combine(m1, m1, empty));

// empty Map should just be a copy of m1

assert(empty.size() == m1.size());

assert(!empty.contains("Ricky"));

assert(empty.get(0, key, value) && key == "Fred" && value == 123);

assert(empty.get(1, key, value) && key == "Ethel" && value == 456);

assert(empty.get(2, key, value) && key == "Lucy" && value == 789);

// test combine function where it should return false

assert(!combine(m2, empty, empty));

assert(empty.size() == 3);

assert(empty.get(0, key, value) && key == "Fred" && value == 123);

assert(empty.get(1, key, value) && key == "Ethel" && value == 456);

assert(empty.get(2, key, value) && key == "Ricky" && value == 321);

assert(!empty.contains("Lucy")); // Lucy should not be here (different values)

// reset test cases to original

Map cleared;

m1 = m2 = cleared;

m1.insertOrUpdate("Fred", 123);

m1.insertOrUpdate("Ethel", 456);

m1.insertOrUpdate("Lucy", 789);

m2.insertOrUpdate("Lucy", 789);

m2.insertOrUpdate("Ricky", 321);

// test combine if result and the first Map are the same

assert(combine(m1, m2, m1));

assert(m1.size() == 4);

assert(m1.get(0, key, value) && key == "Fred" && value == 123);

assert(m1.get(1, key, value) && key == "Ethel" && value == 456);

assert(m1.get(2, key, value) && key == "Lucy" && value == 789);

assert(m1.get(3, key, value) && key == "Ricky" && value == 321);

// reset test cases to option 2 (where Lucy has different values in m1 and m2)

m1 = m2 = cleared;

m1.insertOrUpdate("Fred", 123);

m1.insertOrUpdate("Ethel", 456);

m1.insertOrUpdate("Lucy", 789);

m2.insertOrUpdate("Lucy", 654);

m2.insertOrUpdate("Ricky", 321);

// test combine if Lucy holds different values in m1 and m2

assert(!combine(m1, m2, m1));

assert(m1.size() == 3);

assert(m1.get(0, key, value) && key == "Fred" && value == 123);

assert(m1.get(1, key, value) && key == "Ethel" && value == 456);

assert(m1.get(2, key, value) && key == "Ricky" && value == 321);

// set up tests for reassign function

// create empty Maps

Map getReassigned, reassigned;

assert(getReassigned.empty() && reassigned.empty());

// test reassign function for empty Map

reassign(getReassigned, reassigned);

assert(getReassigned.empty() && reassigned.empty());

// set up getReassigned Map for later test

getReassigned.insert("Annie", 25);

getReassigned.insert("Bonnie", 14);

getReassigned.insert("Tony", 31);

getReassigned.insert("Sunny", 21);

// test reassign function for Map with 4 interesting elements

reassign(getReassigned, reassigned);

assert(reassigned.size() == getReassigned.size());

assert(reassigned.get(0, key, value) && key == "Annie" && value == 14);

assert(reassigned.get(1, key, value) && key == "Bonnie" && value == 31);

assert(reassigned.get(2, key, value) && key == "Tony" && value == 21);

assert(reassigned.get(3, key, value) && key == "Sunny" && value == 25);

// empty out getReassigned by using the assignment operator

getReassigned = cleared;

getReassigned.insert("Annie", 4);

// test reassign function for Map with 1 “interesting” node

reassign(getReassigned, reassigned);

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

assert(reassigned.get(0, key, value) && key == "Annie" && value == 4);

assert(!reassigned.get(-1, key, value));

assert(!reassigned.get(reassigned.size(), key, value));

assert(!reassigned.get(1, key, value));

// test reassign function if two parameters are the same Map with 1 element

reassign(getReassigned, getReassigned);

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

assert(getReassigned.get(0, key, value) && key == "Annie" && value == 4);

assert(!getReassigned.get(-1, key, value));

assert(!getReassigned.get(reassigned.size(), key, value));

assert(!getReassigned.get(1, key, value));

// reset getReassigned

getReassigned = cleared;

getReassigned.insert("Annie", 25);

getReassigned.insert("Bonnie", 14);

getReassigned.insert("Tony", 31);

getReassigned.insert("Sunny", 21);

// test reassign function if two parameters are the same Map with 4 elements

reassign(getReassigned, getReassigned);

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

assert(getReassigned.get(0, key, value) && key == "Annie" && value == 14);

assert(getReassigned.get(1, key, value) && key == "Bonnie" && value == 31);

assert(getReassigned.get(2, key, value) && key == "Tony" && value == 21);

assert(getReassigned.get(3, key, value) && key == "Sunny" && value == 25);

getReassigned = cleared;

// test reassign function if two parameters are the same Map with 0 elements

reassign(getReassigned, getReassigned);

assert(getReassigned.empty());

}