I decided that a binary tree for my map to be the best choice. The biggest functions such as: erase, insert, get, and the destructor are well established functions that are known to be solid in terms of functionality. The nodes merely contain the key, value, left and right pointers, simple and solid. There are only two private member variables beside the node, such as the root and a counter to keep track of the size of the map. In other words, I picked a binary search tree because of its speed, simplicity, and sturdy integrity. I also love rm and root in linux and I know it has to do with a tree.

The biggest obstacle I encountered was making the erase function for a BST. Two children deletion isn’t easy and straight forward, but I overcame it by using dynamic memory allocation to save a temporary node and be able to access it between functions. Another notable obstacle was finding duplicates for the combine function. I was able to solve it by making a black and white list to keep track of nodes I should and shouldn’t add. I was able to later reuse the same code for the subtract function.

Pseudo-Code:

Erase function:

If erase helper equals true

Decrement counter

Return true

Erase helpers:

Check if node has no children, 1 child, or two children

Then check if the node is root

If no children delete node and return true

If 1 child reconnect node then delete the child and return true

If 2 children find the greatest node and replace node to delete with that value

Then feed the greatest node back into erase helper to delete

Keep a copy dynamically allocated to access between functions being passed as a pointer

And return true

3 argument get function:

Check i and make sure it is a proper value between 0 and size

Dynamically allocate an integer to keep track of i

Recursively travel through the tree incrementing for each traversal and copying each node into

The argument pointers for key and value

Stop the traversal when i = size

Combine function:

Create black and white list

Iterate through each node in the first list

Check if in each iteration list contains that key

If duplicate keys are found check to see if they have the same or different values

If the values are the same add it to the result list and add to whitelist

If the values are different do not add them and put the key into the blacklist

Then after all the duplicates have been found

Iterate through each list and check if each member isn’t in both the white and black list to

Insert into result and return true if no mismatched duplicates were found

Test Code:

Map(const Map& rhs)//Copy Constructor will tested for accuracy with random nodes

// default constructor

Map test1, test2;

//test1 is empty

//test2 will have some nodes

test2.insert("AAA", 111);

test2.insert("DDD", 11221);

test2.insert("BBB", 11331);

test2.insert("FFF", 11144);

Map test3(test2);

for(int i = 0; i < test3.size(); i++) //Should print out test2

{

KeyType a;

ValueType b;

test3.get(i, a, b);

cout << a << b << endl;

}

Map test4(test1);

for(int i = 0; i < test4.size(); i++) //Should print out test1

{

KeyType a;

ValueType b;

test4.get(i, a, b);

cout << a << b << endl;

}

//Since = assignment operator uses the same code test with similar code

// default constructor

Map test1, test2, test3;

//test1 is empty

//test2 will have some nodes

test2.insert("AAA", 111);

test2.insert("DDD", 11221);

test2.insert("BBB", 11331);

test2.insert("FFF", 11144);

test3 = test2;

for(int i = 0; i < test3.size(); i++) //Should print out test2

{

KeyType a;

ValueType b;

test3.get(i, a, b);

cout << a << b << endl;

}

test3 = test1;

for(int i = 0; i < test3.size(); i++) //Should print out test1

{

KeyType a;

ValueType b;

test3.get(i, a, b);

cout << a << b << endl;

}

bool empty() const; // Return true if the map is empty, otherwise false.

Map test1, test2;

test2.insert("AAA", 111);

if (test2.empty())

cout << "Error\n";

else

cout << "Success!\n";

if (test1.empty())

cout << "Success!\n";

else

cout << "Error!\n";

bool update(const KeyType& key, const ValueType& value);

Map test1, test2; //testing an empty list and to see if the update was successful with a

//Second list

Map test1, test2;

if (test1.update("AAA", 1111))

cout << "Error!\n";

else

cout << "Success!\n";

test2.insert("AAA", 111);

test2.insert("BBB", 111);

if (test2.update("BBB", 1111))

cout << "Success!\n";

else

cout << "Error!\n";

for (int i = 0; i < test2.size(); i++)

{

KeyType a;

ValueType b;

test2.get(i, a, b);

cout << a << b << endl;

}

bool insertOrUpdate(const KeyType& key, const ValueType& value);

Map test1;//This tested to see if the item was inserted into the list

//And updated correctly

if (test1.insertOrUpdate("cow", 123))

cout << "Success!\n";

else

cout << "failure!\n";

if (test1.insertOrUpdate("cow", 666))

cout << "woohoo!\n";

else

cout << "Oh no!\n";

for (int i = 0; i < test1.size(); i++)

{

KeyType a;

ValueType b;

test1.get(i, a, b);

cout << a << b << endl;

}

bool erase(const KeyType& key);

//Test the deletion of a two child node, which in turn tests the single child deletion as well

//And root deletion testing

Map test;

test.insert("Homer", 123);

test.insert("Lisa", 567);

test.insert("Marge", 111);

test.insert("Abe", 12345);

test.erase("Homer");

Map test2;

test2.insert("Hello?", 111);

test2.erase("Hello?");

if (!test2.contains("Hello?"))

cout << "Success!\n";

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

{

KeyType a;

ValueType b;

test.get(i, a, b);

cout << a << b << endl;

}

bool contains(const KeyType& key) const;

//This function has been tested by using it for other tests

bool get(const KeyType& key, ValueType& value) const;

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

//Both of these functions have been used in other tests and have proven their success

void swap(Map& other);

//Test by reading out the contents of the swapped Map

//Includes an empty Map

Map test,test2,test3;

test.insert("Homer", 123);

test.insert("Lisa", 111);

test2.insert("Marge", 222);

test2.insert("Bart", 666);

test2.insert("Maggie", 777);

test.swap(test2);

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

{

KeyType a;

ValueType b;

test.get(i, a, b);

cout << a << b << endl;

}

test.swap(test3);

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

{

KeyType a;

ValueType b;

test.get(i, a, b);

cout << a << b << endl;

}

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

//This test code fuzzes random order duplicates to see if they are picked up by the algorithm for combine()

Map test;

test.insert("aaa", 666);

test.insert("bbb", 555);

test.insert("aaa", 555);

test.insert("ccc", 444);

Map test2;

test2.insert("aaa", 333);

test2.insert("ddd", 111);

test2.insert("eee", 000);

//Second sample. This broke my code.

Map test;

test.insert("aaa", 333);

test.insert("bbb", 666);

test.insert("aaa", 555);

test.insert("ccc", 444);

Map test2;

test2.insert("aaa", 333);

test2.insert("ddd", 111);

test2.insert("aaa", 777);

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

//Used same code as above, but subtract instead of combine

//This test code fuzzes random order duplicates to see if they are picked up by the algorithm for combine()

Map result;

Map test;

test.insert("aaa", 666);

test.insert("bbb", 555);

test.insert("aaa", 555);

test.insert("ccc", 444);

test.insert("eee", 000);

Map test2;

test2.insert("aaa", 333);

test2.insert("ddd", 111);

test2.insert("eee", 000);

subtract(test, test2, result);

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

{

KeyType a;

ValueType b;

result.get(i, a, b);

cout << a << b << endl;

}

//Second sample. This broke my code.

Map result1;

Map test1;

test1.insert("aaa", 333);

test1.insert("bbb", 666);

test1.insert("aaa", 555);

test1.insert("ccc", 444);

Map test3;

test3.insert("aaa", 333);

test3.insert("ddd", 111);

test3.insert("aaa", 777);

subtract(test1, test3, result1);

for (int i = 0; i < result1.size(); i++)

{

KeyType a;

ValueType b;

result1.get(i, a, b);

cout << a << b << endl;

}