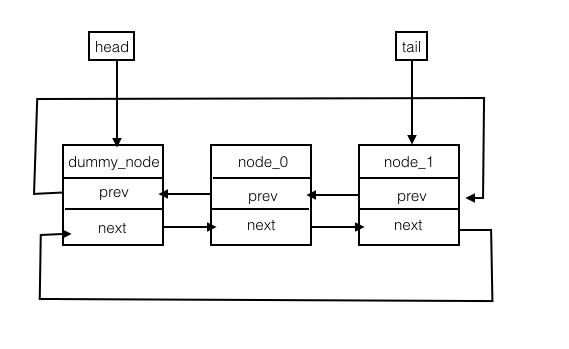
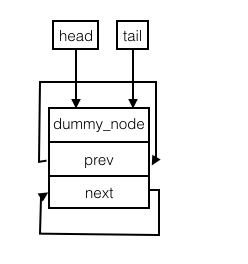
**Description:**

I used a doubly-linked circular list with a dummy node to implement this project. The node has member data: pair(containing key and value), pointer to next node, pointer to previous node. If the list is empty, the only node is dummy node with prev(next) pointer pointing to itself. Also there is a head(tail) node which is pointing to the first(last) node. If the list is empty, they are pointing to the dummy node.



Normal Case



Empty Case

**Pseudocode:**

Map::~Map()

loop over the map nodes:

get the key in current node;

set current pointer to previous one;

erase the node containing the key;

delete dummy\_node

Map::erase(const KeyType& key)

repeatedly:

if current node’s key equals key given by the argument, do:

break the double links between current node and prev/next node;

if current node is the tail node

change tail pointer to point to previous node;

delete current node;

decrease list length by one;

return true

return false

Combine(const Map& m1, const Map& m2, Map& result)

set combine\_flag to true;

copy m1 to tmp\_result;

loop over m2 nodes:

get key and value in current m2 node;

if m1 contains the current key:

if value in m1’s node is equal to m2’s:

continue;

else:

delete the node containing the key intmp\_result

set combine\_flag to false

else:

insert the pair in current m2 node to tmp\_result

assign tmp\_result to result;

return combine\_flag;

Subtract(const Map& m1, const Map& m2, Map& result)

create an empty Map tmp\_result;

loop through m1 list:

get current node’s key;

if m2 contains current key:

insert corresponding value to tmp\_result;

assign tmp\_result to result;

**Test Cases:**

// Empty Map Test

Map m;

KeyType key;

ValueType value;

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

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

assert(!m.erase("Fred")); // test erase()

assert(!m.contains("Fred")); // test contains()

assert(!m.get(0, key, value)); // test get()

assert(!m.get(key, value)); // test get()

assert(m.insert("Fred", 222)); // test insert()

// Normal Map Test

m.get("Fred", value); // test get()

assert(value== 222);

//assert(m.insertOrUpdate("Fred", 334));

assert(m.contains("Fred")); // test contains()

assert(m.update("Fred", 334)); // test update()

m.get("Fred", value); // test get()

assert(value== 334);

assert(m.insertOrUpdate("Fred", 434.6)); // test insertOrUpdate()

m.get("Fred", value);

assert(value== 434.6);

m.erase("Fred"); // test erase()

assert(m.empty());

// Swap;

Map m1;

m1.insert("Ethel", 456);

m1.insert("Lucy", 789);

m1.insert("Fred", 123);

m.swap(m1);

assert(m1.empty());

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

assert(m.contains("Ethel"));

assert(m.contains("Lucy"));

assert(m.contains("Fred"));

m.swap(m1);

// Combine

Map m2;

Map m3;

assert(m2.empty());

m2.insert("Lucy", 789);

m2.insert("Ricky", 321);

m3.insert("Wenliwen", -1111);

std::cerr << m1.size() << " = m1.size()" << std::endl;

assert(combine(m1, m2, m3)); // test return true

assert(!m3.contains("Wenliwen"));

assert(m3.contains("Ricky"));

assert(m3.contains("Ethel"));

assert(m3.contains("Fred"));

assert(m3.contains("Lucy"));

assert(m3.size() == 4);

Map m4;

m2.update("Lucy", 9999);

assert(!combine(m1, m2, m4)); // test return false

assert(m4.size() == 3); // test false case for combine

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

std::cerr << "\*\*\*\*\*\*\*\*" << std::endl;

m1.dump();

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

//Subtract

Map m5;

Map m6 = m4; // Ricky 321, Ethel 456, Fred 123

std::cerr << "m6==============" << std::endl;

m6.dump();

m5.insert("Lucy", 789);

m5.insert("Ricky", 321);

m5.insert("Ethel", 456);

Map m7;

m7.insert("Pan", 873);

subtract(m5, m6, m7); // m7: Pan 873, Lucy 789

assert(m7.size() == 1);

assert(!m7.contains("Pan"));

assert(m7.contains("Lucy"));

m7.dump();

subtract(m5, m6, m5);

assert(m5.size() == 1);

assert(m5.contains("Lucy"));

assert(!m5.contains("Ricky"));

assert(!m5.contains("Ethel"));

//Destructor

for(int i = 0; i < 5; i++){

Map m\_temp;

m\_temp.insert("test", 0.1);

}

//Copy Constructor

Map m\_origin;

m\_origin.insert("origin0", 0);

m\_origin.insert("origin1", 1);

m\_origin.insert("origin2", 2);

Map m\_copy(m\_origin);

m\_origin.erase("origin2");

assert(m\_copy.size() == 3);

assert(m\_copy.contains("origin0"));

assert(m\_copy.contains("origin1"));

assert(m\_copy.contains("origin2"));

ValueType value\_copy;

m\_copy.get("origin0", value\_copy);

assert(value\_copy == 0);

m\_copy.get("origin1", value\_copy);

assert(value\_copy == 1);

m\_copy.get("origin2", value\_copy);

assert(value\_copy == 2);

//Assign Operator

Map m\_assign = m\_origin;

m\_origin.insert("origin2", 2);

assert(m\_assign.size() == 2);

assert(m\_assign.contains("origin0"));

assert(m\_assign.contains("origin1"));

assert(!m\_assign.contains("origin2"));

ValueType value\_assign;

m\_assign.get("origin0", value\_assign);

assert(value\_assign == 0);

m\_assign.get("origin1", value\_assign);

assert(value\_assign == 1);