1. The Map class contains two pointers to a Node, m\_head and m\_tail. These point to the first and last Nodes in the doubly linked list. They point to NULL if the list is empty. The Nodes in Map each hold a KeyType item and a ValueType item as well as a pointer to the next Node and a pointer to the previous Node. The Nodes in the doubly linked list are not in any particular order; new Nodes are inserted at the end of the list.
2. Map::Map(const Map& other)

for every Node in other Map

dynamically create new Node

copy key and value from other Map’s Node

place new Node at end of new list

link with previous Node

Map::~Map()

for each Node starting from m\_head

create temporary pointer to next Node

delete current Node

point to next Node

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

delete each Node originally in current Map

for every Node in other Map

dynamically create new Node

copy key and value from other Map’s Node

place new Node at end of new list

link with previous Node

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

visit each Node in Map

if Node already contains current key

return false

dynamically create new Node with key and value

point to previous Node

previous Node points to new Node

m\_tail points to new Node

if list is empty

m\_head points to new Node

increment size by on

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

visit each Node in Map

break if matching key is found

return false if no keys match

set current value of key to new value

bool Map::erase(const KeyType& key)

visit each Node in Map

break if matching key is found

return false if no keys match

check for first Node or last Node case

set new pointers for adjacent Nodes

delete current Node

decrease size by one

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

visit each Node in Map

break if matching key is found

return false if no keys match

copy value of current Node to value in parameter

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

check if int parameter is valid

follow the m\_next pointer through the list as many times as int parameter

copy key and value of Node into parameter key and value

void Map::swap(Map& other)

create temporary head pointer

swap m\_head pointers of both Maps

create temporary tail pointer

swap m\_tail pointers of both Maps

swap the m\_size

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

set a bool to true

create temporary key and two values

copy m1 to result using assignment operator

for each Node in m2

if key matches a key in result

{

if the values are different

set bool to false

erase the Node from result

}

else

insert Node into result

return the bool

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

create temporary key and value

copy m1 to result using assignment operator

for each Node in m2

if the key matches a key in result

erase the Node from result

1. Map m; // default constructor

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

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

assert(!m.erase("Ricky")); // nothing to erase

assert(m.insert(“Ricky”,10)); // insert node

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

assert(m.update(“Ricky”,20)); //update value

assert(!m.update(“Ben”, 10)); //update non-existing node

assert(m.size()==1); //test size

assert(m.insert(“Ben”,10));

assert(m.insert(“Tom”,30));

// list should now contain in this order

// m\_head->(“Ricky”,20)<-->(“Ben”,10)<-->(“Tom”,30)<-m\_tail

assert(m.contains(“Ben”)); // test contains

assert(!m.contains(“Kevin”));

assert(m.insertOrUpdate(“Ben”,40)); // update the node

assert(m.insertOrUpdate(“Kevin”,50)); // insert new node

// list should now contain in this order

// m\_head🡪(“Ricky”,20)🡨🡪(“Ben”,40)🡨🡪(“Tom”,30)🡨🡪(“Kevin”,50)🡨m\_tail

assert(!m.erase(“John”)); // test erase

assert(m.erase(“Tom”));

KeyType tempKey = “Tom”;

ValueType tempValue;

assert(!m.get(tempKey,tempValue)); // test get

tempKey = “Ben”;

assert(m.get(tempKey,tempValue));

assert(tempValue == 40);

assert(m.get(0,tempKey,tempValue)); // test 2nd get

assert(tempKey == “Ricky” && tempValue == 20);

assert(!m.get(5,tempKey,tempValue));

// m should contain //m\_head🡪(“Ricky”,20)🡨🡪(“Ben”,40)🡨🡪(“Kevin”,50)🡨m\_tail

// tests with multiple Maps

Map m2(m); // copy construction

assert(m2.size() == 3); // check if successful

assert(m2.contains(“Ricky”) && m2.contains(“Ben”) && m2.contains(“Kevin”));

Map m3;

m3 = m; // assignment operator

assert(m3.size() == 3); // check if successful

assert(m3.contains(“Ricky”) && m3.contains(“Ben”) && m3.contains(“Kevin”));

m3.insert(“Fred”,100); // test swap

m.swap(m3);

assert(m.contains(“Fred”));

assert(!m3.contains(“Fred”));

// m3 should contain //m\_head🡪(“Ricky”,20)🡨🡪(“Ben”,40)🡨🡪(“Kevin”,50)🡨m\_tail

Map m4; // testing combine

m4.insert(“Fred”,100);

m4.insert(“Ricky”,20);

assert(combine(m3,m4,m)); // combine m3 and m4 into m

assert(m.contains(“Ricky”) && m.contains(“Ben”) && m.contains(“Kevin”) && m.contains(“Fred”) );

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

Map m5;

m5.insert(“Fred”,100);

m5.insert(“Ricky”,30);

assert(!combine(m3,m5,m)); //combine m3 and m5 into m

assert(!m.contains(“Ricky”) && m.contains(“Ben”) && m.contains(“Kevin”) && m.contains(“Fred”));

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

// m3 should contain //m\_head🡪(“Ricky”,20)🡨🡪(“Ben”,40)🡨🡪(“Kevin”,50)🡨m\_tail

// m5 should contain //m\_head🡪(“Fred”,100)🡨🡪(“Ricky”,30)🡨m\_tail

//testing subtract

subtract(m3,m5,m); //subtract m5 from m3 into m

assert(!m.contains(“Ricky”) && m.contains(“Ben”) && m.contains(“Kevin”) && !m.contains(“Fred”));

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