1. My doubly-linked list simply gives each node a prev and next node pointer. The first node’s prev pointer points to nullptr and the final node’s next pointer points to nullptr. All node’s in between have a prev pointer to the one node before it and a next pointer to the node after it. However, the list is not circular. The head pointer points to the first node and the tail pointer points to the last node in order to access either side as fast as possible.
2. Pseudocode
   1. insert(key, value)

{

If key is already in List

Return false

If not add node to the end of Linked List

Create a new node

Set new node key and value

Set next pointer to nullptr

Set prev pointer to tail

If List is empty

Set head and tail to node

if not empty

Set tail node’s next to new node

Set tail to new node

}

* 1. update(key, value)

{

Create a node pointer that starts at head

Traverse the List

If current node’s key is the target key

Change the current node’s value to target value and exit

}

* 1. erase(key)

{

If List is empty, exit

If key is first node and only node

Delete the node and set head and tail to nullptr, exit

If key is first node

Create a pointer to a node that points to head

Set next node’s prev pointer to nullptr

Set head to next node

Delete the current node

If key is last node

Create a pointer to a node that points to tail

Set prev node’s next pointer to nullptr

Set tail to prev node

Delete the current node

Create a node pointer that starts at head

Traverse the List

stop at the node above the node key

if pointing at a valid node

create a node pointer to the target node

set prev node’s next pointer to targeted node’s next

set the next node’s preb pointer to targeted node’s prev

delete the current node

}

* 1. contains(key)

{

Create a node pointer that starts at head

Traverse the List and return true if current node’s key is the target key

}

* 1. get(key, value)

{

Create a node pointer that starts at head

Traverse the List

If current node’s key is target key

Set value to current node’s value

* 1. get( i, key, value)

{

Exit if I is negative or larger than List size

Create a node pointer that starts at head

Traverse the List to the ith node

Store ith node’s key and value into passed in key and value

}

* 1. combine(m1, m2, result)

{

Put m1 nodes into result

Create a Boolean variable and set to true

Create a key variable and two value variables

Traverse the m2 list

Store m2 current node key and value into variables

Store m1 same key into a value variable

If result doesn’t contain m2 node

Add the m2 node into result

If it does contain but different values

Delete the node from result and change Boolean

}

* 1. subtract(m1, m2, result)

{

Put m1 nodes into result

create a key variable

traverse the m2 list

store m2 current node key into key

if result contains the key

erase the node from result

1. Test cases

//deafult constructor

Map m1;

KeyType name;

ValueType GPA;

assert(m1.size() == 0); //check if map is empty

assert(m1.empty()); //check if map is empty

assert(!m1.erase("Sahil")); //map shouldn't erase anything

assert(!m1.get(1, name, GPA)); //map shouldn't return false because index is larger than list size

assert(!m1.get(-10, name, GPA)); //map shouldn't return false because index is negative

assert(m1.insert("Louis", 4.21)); //check if inserted

assert(!m1.update("Niall", 3.87)); //check if can't update

assert(m1.insert("Niall", 3.98)); //check if inserted

assert(m1.update("Louis", 4.25)); //check if updated

assert(m1.size() == 2); //size should be two

assert(!m1.empty()); //map shouldn't be empty

assert(m1.get(0, name, GPA)); //check if node was read

assert(!m1.get(2, name, GPA)); //check if node can't be read

assert(name == "Louis" && GPA == 4.25); //check if 1st node was used

Map m2 = m1; //copy constructs m2 from m1

assert(m2.size() == 2); //check if size is same as m1

assert(m2.insertOrUpdate("Harry", 2.23)); //check if inserted

assert(m2.insertOrUpdate("Niall", 3.87)); //check if updated

assert(m2.get("Niall", GPA)); //check if node is read

assert(name == "Louis" && GPA == 3.87); //check if GPA is changed but name isn't

m2 = m1; //copies m1 nodes into m2

assert(m2.size() == 2); //check if size is same as m1

assert(m2.insert("Liam", 1.89)); //check if inserted

assert(m2.insert("Harry", 2.11)); //check if inserted

assert(m2.insert("Zayn", 3.56)); //check if inserted

assert(m2.erase("Niall")); //check if erased

assert(m2.erase("Louis")); //check if erased

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

Map result;

assert(result.insertOrUpdate("Ed", 5.00)); //check if inserted

assert(combine(m1, m2, result)); //check if m1 and m2 were combined into result

assert(result.size() == 5); //check if size is now sum of both

assert(result.get(0, name, GPA)); //check if node is read

assert(name == "Louis" && GPA == 4.25); //check if first node was read

assert(result.get(2, name, GPA)); //check if node is read

assert(name == "Liam" && GPA == 1.89); //check if 3rd node was read

assert(result.get(4, name, GPA)); //check if final node is read

assert(name == "Zayn" && GPA == 3.56); //check if final node was read

assert(m2.erase("Liam")); //checks if erased

assert(m2.erase("Harry")); //checks if erased

assert(m2.size() == 1); //checks if List contains only Zayn

subtract(result, m2, result); //subtracts m2 from result and puts the resulting list into result

assert(result.size() == 4); //check if result only has four nodes

assert(!result.contains("Zayn")); //checks if Zayn is no longer part of One Direction

m1.swap(result); //swaps result and m1

assert(m1.size() == 4); //check if m1 is now the size of result

assert(result.size() == 2); //check if result is now the size of m1