**CS 32 Project 2 Report**

1. Design of doubly linked list

**Typical Map**

nullptr

nullptr

Head

**next**

**next**

**next**

**prev**

**prev**

**prev**

**Key**

**Value**

**Key**

**Value**

**Key**

**Value**

**Empty Map**

Head

nullptr

Every node of the linked list contains pointers to the next and previous nodes. The head pointer points to the first node of the linked last. There is no dummy node. Every node contains a Key and Value variable. The doubly-linked linked list is not circular. The previous pointer of the first node is the nullptr while the last node’s next pointer is the nullptr. An empty list is denoted by the head pointer being the nullptr. The nodes are arranged in the order in which they were inserted. The last node is the most recent node inserted whereas the first node is the oldest node inserted in the list.

1. Pseudocode for non-trivial algorithms

**Map::erase**  
*If the map is empty return false  
Check if Map contains given key and do the following if true*

*If there is only one node In the list and its key value is the same as the given key delete the node and make head the nullptr. Return true  
Otherwise if the first node’s key value is equal to the key parameter create temporary pointers to the head node and the following node. The second node’s previous pointer must become the nullptr and head must become the second node. The first node is deleted. Return true.  
Otherwise,  
Traverse through Map and repeat following steps until the Map ends*

*If the node is not the last node and if the next node’s key value is the same as the key parameter create a temporary pointer to the node to be erased. The current node’s next pointer is made to point to the node following the node to be erased. Erase the node. If the node following the erased node is not the nullptr link it to the preceding node using the previous pointer. Return true.*

*End traversal*

*If Map does not contain the given key, return false.*

**Map::insert**

*If the map does not contain the key passed as an argument*

*If the list is empty*

*Create a new node and assign its key and value as per parameters*

*Make head point to this new node and initialize its next and previous pointers to nullptr*

*Return true*

*Otherwise, for every other case*

*Create a new node and assign its key and value as per parameters*

*Traverse through the list until it reaches the last node*

*Make the last node’s next pointer point to the new node*

*Make the new node’s previous pointer point to the last node*

*Make the new node’s next pointer the nullptr*

*Return true*

*Otherwise if the map contains the key passed as an argument, return false*

**Map::get** (with 3 parameters)

*If the value ‘i’ specified is greater than or equal to 0 and less than the size of the Map*

*Create an integer variable to keep track of which node we are on (like array indexing)*

*Traverse through list and repeat the following step until you reach the node specified by the function parameter or you go past the end of the list*

*Decrement the integer variable by 1*

*If you are at the node specified, change the key and value parameters to the corresponding values of this node and return true*

*Return false*

**Subtract**

*Create a temporary map*

*Traverse through map m1 and repeat the following steps until the map ends*

*Get the key and value of each node*

*If m2 does not contain the current key, then insert the current key/value pair to the temporary Map*

*Assign the contents of the temporary Map to the result Map parameter*

**Combine**

*Create a temporary Map*

*Create a Boolean variable to tell if there exists a key that appears in both m1 and m2 but with different corresponding values.*

*Traverse through every node of the list m1 and repeat the following steps until the map ends*

*Get the key and value of each node*

*If the temporary map doesn’t already contain the current key, insert node*

*Otherwise if the resultant map already contains the current key*

*Get the value being mapped to by the current key*

*If this value is the same as value mapped to by already existing key skip the node without inserting it.  
Otherwise if the values of the nodes with the same key are different, delete old node with key being referred to and skip without inserting current node. Set r to false.*

*Traverse through every node of the list m2 and repeat the following steps until the map ends*

*Get the key and value of each node*

*If the temporary map doesn’t already contain the current key, insert node*

*Otherwise if the resultant map already contains the current key*

*Get the value being mapped to by the current key*

*If this value is the same as value mapped to by already existing key skip the node without inserting it.  
Otherwise if the values of the nodes with the same key are different, delete old node with key being referred to and skip without inserting current node. Set r to false.*

*Assign contents of the temporary Map to the result Map parameter*

*Return value in r*

1. List of test cases

Map m1;

assert(m1.size()==0) //test size

assert(m1.empty()) //test empty

assert(!m1.erase(“Goal”); //Nothing to erase

ValueType v = -1234.5;

KeyType testing = “Car”;

assert(!m1.get("abc", v) && v == -1234.5); //Nothing to get as Map is empty

assert(!m1.get(3,testing,v) && v == -1234.5 && testing==”Car”); //Map is empty

assert(!m1.update(testing,33)); //Map is empty so it doesn’t contain key given

assert(!m1.contains(“lmn”); //Map is empty so it doesn’t contain key given

assert(m1.insert(“xyz”,9876.5); //Insert node to empty map

assert(m1.size()==1); //test size

assert(m1.erase(“xyz”)); //erase only node in map

assert(m1.insert(“xyz”,9876.5);

KeyType k=”hello”;

assert(m1.get(0, k, v) && k == "xyz" && v == 9876.5); //test get

Map m2;

m2.insert("Cool",987.65);

m2.insert("Boring",432.10);

assert(!m2.insert("Cool",532.1)); //Cannot insert key that already exists

ValueType test=12.3;

KeyType test2="Check";

m2.insert("Gas",test);

assert(m2.contains("Gas")); //test contains

assert(m2.update("Gas",121.21) && m2.get("Gas",test) && test==121.21); //test update

m2.swap(m1);

assert(m1.size()==3 && m2.size()==1 && m1.contains("Cool") && m1.contains("Gas") && m1.contains("Boring") && m2.contains("xyz")); //test swap

assert(m2.insertOrUpdate("Flatulan",32.32) && m2.get("Flatulan",test) && test==32.32); //test insertOrUpdate for case when it must insert

assert(m2.insertOrUpdate("xyz",44.44) && m2.get(0,test2,test) && test2=="xyz" && test==44.44); //test insertOrUpdate for case when it must update

assert(m2.get(1,test2,test) && test2=="Flatulan" && test==32.32); //test get when first parameter is not 0

assert(!m2.erase("Updated")); //cannot erase key that does not exist

assert(m2.erase("xyz") && m2.size()==1 && !m2.contains("xyz") && m2.get(0,test2,test) && test2=="Flatulan" && test == 44.44); //test size, contains, and get after erasing first node

Map m3;

Map m4;

Map r;

r.insert(“Cable”,123);

assert(combine(m3,m4,r) && r.empty()) ; //combining 2 empty maps

subtract(m3,m4,r) ; //subtracting 2 empty maps

assert(r.empty());

m3.insert("Fred",123);

m3.insert("Ethel",456);

m3.insert("Lucy",789);

m4.insert("Lucy",789);

m4.insert("Ricky",321);

m4.insert("Ethel",654);

assert(!combine(m3, m4, r) && r.contains(“Fred”) && !r.contains(“Ethel”) && r.contains(“Lucy”) && r.contains(“Ricky”)); //test combine

subtract(m3,m4,r);

assert(r.contains(“Fred”) && !r.contains(“Lucy”) && !r.contains(“Ethel”)); //test subtract

r=m3;

assert(!combine(m3,m4,r) && r.contains(“Fred”) && !r.contains(“Ethel”) && r.contains(“Lucy”) && r.contains(“Ricky”)); //test combine in case of aliasing

subtract(m4, m4, r);

assert(r.empty()); //test subtract in case of aliasing