**Project 2 Report**

**~ Utkarsh Lal**

**Description of the design of your doubly-linked list implementation. (A couple of sentences will probably suffice, perhaps with a picture of a typical Map and an empty Map. Is the list circular? Does it have a dummy node? What's in your list nodes? Are they in any particular order?)**

I am using a doubly linked list without a dummy node. My logic does not use circular implementation. I have two nodes Head and Tail which point to the first element and the last element respectively.   
My node struct which is included in the private has key (Keytype), value (ValueType), next and previous (both pointer to a node). My project sorts the data whenever a new element in inserted in the map which makes easier to use the get function.

*Diagram, engineering drawing

Description automatically generated*

------------------------------------------------------------------------------------------------------------------------------------------

**pseudocode for non-trivial algorithms (e.g., Map::erase and reassign).**

*Copy Constructor of Map*

set size of map to 0

if other’s size is zero, set head and tail to nullptr

else

looping i as int from 0 to less than size of other

we get the other’s key/value pair using the second get function

then we insert that pair into “this” map

*Assignment Operator*

if this is not equal to the other map

copy contruct the other map into a temp map

swap temp with this

return pointer to this map

*Destructor*

if head is nullptr, exit

make a node pointer to head(p)

make a node pointer next\_del

looping till p reaches nullptr

next\_del is assigned to next of p

**delete p,** p is assigned to next\_del

*Map::insert*

if the key is already present in the map, **return false**

make a newNode which is a new Node and match its key and value to those in the parameter

if size of map equals 0

match newNode’s key and value to the parameters

newNode’s next and previous will be nullptr

head and tail both point to newNode since it’s the first element

increment size of map by 1, **return true**

else presort the map

looping a node pointer p across all elements of the map

if p(element of map)’s key is greater than newnode’s key

newnode’s next is assigned to p, head now points to newnode, newnode’s previous is nullptr and p’s previous is newNode (this newNode acquires the first position in the map)

increment map size by 1, **return true**

if p’s key is less than newNode’s key

check if p’s next key is greater than newNode’s key

if yes, newNode’s next is same as p’s next, p’s next position is now taken by NewNode, NewNode’s previous is p, newNode’s next’s previous is newNode

increment size by 1, **return true**

else if p’s next is nullptr (which means we reached the last position of map)

p’s next is newNode, newNode’s previous is p, newNode’s next is nullptr, tail now points to newNode

increment map size by 1, **return true**

*Map::update*

looping pointer to node p across all nodes of map

if p is not nullptr and p’s key equals the key in the parameter

change p’s value to the value in parameter 2 of function, **return true**

if key not found in the map, **return false**

*Map::insertOrUpdate*

check if the size of map is zero, if yes then apply the first element insertion approach in **insert function**

now traverse through the entire map and try finding the key in the parameter. if found, use the **update function**

if key not found in the map, insert according to the approach in the **insert function.**

*Map::erase*

if the map does not contain the key to be deleted, **return false**

if head points to nullptr, **return false**

if key of first element matches the key to be deleted, create a pointer to node called killme

killme points to the first element

head now points to second element

delete killme, decrement the size of map then **return true**

loop through the map through a pointer p initially pointing to head

if p’s key matches key to be deleted break

if p is the last element of map (means p’s next is nullptr)

p’s previous’ next now points to nullptr

tail now points to p’s previous

delete p, decrement the size of map **return true**

if p is somewhere in middle of map

make a new pointer to node killme point to the node p is pointing to

p’s previous’ next is now p’s next and p’s next’s previous is now p’s previous

delete killme, decrement the size of map **return true**

*Map::contains*

make a new pointer node p

loop p from head till its nullptr

if p’s key matched the key in the parameter, **return true**

**return false** if loop never terminates because of the return statement

*Map::get(key,value)*

make a new node pointer p

loop p from head till its nullptr

if p’s key matched the key in the parameter, set the value parameter to p’s value, **return true**

if loop doesn’t terminate because of the return statement, **return false**

*Map::get(i,key,value)*

if i Is less than zero or more than size of map, **return false**

counter is set to 0

looping in the map using a pointer to a node p

if counter equals i

key equals p’s key and value equals p’s value, **return true**

incrementing counter by 1

*Map::swap*

exchange the map sizes using a temp map size

exchange the head pointers using a temp head

exchange the tail pointers using a temp tail

*merge function*

set returning value to true

m1 is copied into result

looping int i from 0 to less than map size

assigning keytype and valuetype variables k1 and v1

use the get function in m2 and store the key and value in k1,v1 using looping int i as the first parameter

if the key is already present in the map

make a variable v2 of valuetype

use get function to get the value of corresponding key already present in the map

if the already present one and the m2’s don’t match

delete the entire node from result

set return value to false

finally, **return “return value”.**

*Map::reassign*

set result equal to m

make k\_initial keytype and v\_initial valuetype

use the get function to store the first key/value pair in the variables defined above

make a new valuetype variable called assignment which stores the value of first pair’s value

looping integer i from 1 to less than size of result

make new keytype and valuetype variable k1, v1

use the get function (i as first parameter) to store the key/value pair in k1,v1

use the update function in result to change k1’s value to assignment variable

set assignment to v1

finally update the k\_initial’s valuetype to the last stored value in assignment variable.

(This is basically shifting all the values one step to the left)

------------------------------------------------------------------------------------------------------------------------------------------

Testcases

|  |  |  |
| --- | --- | --- |
| Sr. No | Test Case | Explanation |
| 1 | Map msd; | **Creates a map** |
| 2 | assert(msd.empty()); | **Checks if the map is empty** |
| 3 | assert(msd.size() == 0); | **Confirms the size is 0** |
| 4 | assert(msd.update("Lucy", 79) == 0); | **Tries to update in an empty list** |
| 5 | msd.insert("Fred", 123); | **Inserting the first pair to the map** |
| 6 | msd.insert("Ethel", 456); | **Inserting second pair which would presort to become first pair** |
| 7 | assert(msd.get(0, k0, v0) == 1);  assert(k0 == "Ethel" && v0 == 456); | **Checks if the first element is correct according to the sorting** |
| 8 | msd.insert("Lucy", 789); | **Adds an element to the last of map** |
| 9 | assert(msd.size() == 3); | **Checks if correct map size is correct** |
| 10 | Map msd2;  msd2.insert("Lucy", 444);  msd2.insert("Ricky", 321); | **New map created with two elements inserted** |
| 11 | Map result;  result.insert("IP", 66); | **Another new map** |
| 12 | merge(msd, msd2, result); | **Returns false because Lucy in msd and msd2 have different values** |
| 13 | assert(msd2.erase("LuCy") == 0); | **LuCy does not exists in msd2 so incorrect** |
| 14 | assert(msd2.erase("Lucy")); | **Erases successfully** |
| 15 | assert(msd2.insert("Lucy", 789) == 1); | **Adds a new lucy with same value as in msd** |
| 16 | merge(msd, msd2, result); | **This returns true with 4 elements** |
| 17 | Map result2;  reassign(msd, result2); | **All the values of msd shift to next key in result2** |
| 18 | assert(result2.size() == 3); | **Since result2 is a copy of msd, it also has 3 elements** |
| 19 | Map result = msd2;  cout << merge(msd, msd2, result) << endl; | **Preassigns result to another map but still manages to correctly change according to requirement** |
| 20 | Map result = msd;  cout << merge(msd, msd2, result) << endl; | **Points to msd but still works according to plan** |
| 21 | Map msd; // KeyType is std::string, ValueType is double  assert(msd.insert("Lucy", 79) == 1);  assert(msd.insert("Juicy", 79) == 1);  assert(msd.insert("JuicY", 80) == 1);  assert(msd.insert("Mickey", 880) == 1);  assert(msd.insert("Orra", 90) == 1); | **Starting with a new map** |
| 22 | assert(msd.contains("Mickey")); | **Checking correct working of contains for the last element of map (after sorting)** |
| 23 | KeyType k0;  ValueType v0;  assert(msd.get(0, k0, v0) == 1 && v0 == 80); | **Checks the first elements of the map(JuicY > Juicy)** |
| 24 | assert(msd.get(k0, v1) == 1 && v1 == 80); | **Again checking if both get functions work correspondingly** |
| 25 | assert(msd.update("Juice", 90) == 0); | **Juice doesn’t exist in the map** |
| 26 | Map result = msd; | **Starting test of copy constructor** |
| 27 | assert(result.contains("Lucy") == 1);  assert(result.contains("Juicy") == 1); assert(result.contains("JuicY") == 1); assert(result.contains("Mickey") == 1); assert(result.contains("Orra") == 1); | **Checks for all the elements successfully transferred to result** |
| 28 | assert(msd.size() == result.size()); | **Check if size is correct** |
| 29 | KeyType k0;  ValueType v0;  assert(msd.get(0, k0, v0) == 1 && v0 == 80); | **Same as test case but this time in result** |
| 30 | ValueType v1;  assert(result.get(k0, v1) == 1 && v1 == 80); | **Same as test case but in result** |
| 31 | Map n;  assert(n.insertOrUpdate("Jordan", 789) == 1);  assert(n.insertOrUpdate("Mike", 455) == 1);  assert(n.size() == 2); | **Using the insertOrUpdate function and checking the size in new map** |
| 32 | msd.swap(n);  assert(msd.size() == 2 && n.size() == 5); | **Swap function. Checks for the correct change in size** |
| 33 | assert(!msd.contains("Lucy") && n.contains("Lucy")); | **Checks for a particular element if swapped** |
| 34 | Map result = n;  reassign(msd, result);  result.dump(); | **Checks for reassigning when only 2 elements are present** |
| 35 | msd.erase("Lucy"); msd.insert("Nike", 900); reassign(msd, result); | **This testcase found a bug for me that pointed to map size not correctly working in erase function that made result return wrong map** |
| 36 | Map rsu;  rsu.insert("Pep", 666);  merge(msd, result, rsu); | **After reassigning msd into result, when we use merge function, it should make rsu a empty map** |
| 37 | reassign(msd, msd);  msd.dump(); | **Checks aliasing in reassign** |
| 38 | Map result;  merge(msd, msd, result); | **Aliasing in the merge function** |
|  |  |  |