# Doubly-Linked List Implementation

The implementation is similar to that of a simple doubly-linked list with no circularity or dummy nodes. Each node has four public data members:

* *m\_key*: a key of type KeyType
* *m\_value*: a value of type ValueType
* *m\_next*: a pointer to the next node (equal to *nullptr* for the last node)
* *m\_prev*: a pointer to the previous node (equal to *nullptr* for the first node)

The Map object itself has a pointers *m\_head* and *m\_tail* pointing to the first and last nodes of the list respectively. In the case of an empty Map, both are null pointers and there aren’t any nodes.

A Map with three nodes *n1*, *n2*, and *n3* can be represented as follows:

|  |  |  |
| --- | --- | --- |
| ***n1*** | ***n2*** | ***n3*** |
| *m\_key* | *m\_key* | *m\_key* |
| *m\_value* | *m\_value* | *m\_value* |
| *m\_next* | *m\_next* | *m\_next* |
| *m\_prev* | *m\_prev* | *m\_prev* |

*m\_head m\_tail  
  
   
  
  
  
 nullptr  
nullptr*

The Map nodes are ordered in the same order they are inserted in the list, since the implementation of the *doInsertOrUpdate* function is such that it always inserts (key, value) pairs at the end of the doubly-linked list.

# Pseudocode Descriptions of Implementations

## Map destructor implementation

The destructor *Map::~Map()* relies on the *erase* function to delete all (key, value) pairs from the map as follows:

*while Map is not empty:*

*erase/delete the first Pair in the map*

## Map copy constructor implementation

The copy constructor *Map::Map(const Map& other)* overwrites data members of the Map with data members of *other* as follows:

*initialise m\_size to 0, and m\_head, m\_tail to null pointers  
for each element in other:  
 create an identical element in the Map using insert()*

*(note: m\_size is increased correspondingly, via the )*

## Map *operator=* implementation

The = operator *Map& Map::operator=(const Map& other)* is implemented using the copy constructor as follows:

*if this map is not the same as the other map:*

*initialise a new Map temp with other using the copy constructor*

*swap information in temp and this Map  
return this Map*

**Note**: since *temp* is a local variable, it’s deleted when we exit the *if* statement and there is therefore no memory leak.

## *erase* implementation

The erase function *Map::erase(const KeyType& key)* deletes a (key, value) pair with the specified key from the Map as follows:

*find pointer to the right pair using find*

*if the pair doesn’t exist,* ***return false****; continue if pair exists*

*if node is the first in the Map:*

*make m\_head point to the next node (if any) or null pointer (if no nodes)*

*else:*

*make m\_next of the previous node point to the next node (if any) or null pointer (if no nodes)*

*if node is the last in the Map:*

*make m\_tail point to the previous node (if any) or null pointer (if no nodes)*

*else:*

*make m\_prev of the next node point to the previous node (if any) or null pointer (if no nodes)*

*delete node and decrease m\_size*

*return true*

## Two-member *get* implementation

The two-member get function *Map::get(const KeyType& key, ValueType& value)* changes *value* to the value associated with *key* (if key in list) as follows:

*find pointer to the right pair using find*

*if the pair doesn’t exist,* ***return false***

*if pair exists, change value to value associated with the key in the pair*

## Three-member *get* implementation

The three-member *get* function *Map::get(int i, KeyType& key, ValueType& value)* changes *key* and *value* to the key and value associated with some pair in the map as follows:

*if i < 0 or i >= size of the map, return false*

*initialise a Pair pointer to m\_head (the zero’th pair in the map)*

*repeatedly, if we haven’t found the i'th pair in Map:*

*update the pointer to point to the next pair (or null pointer, if we’ve reached end of list)*

*update key and value to pair’s key and value*

*return true*

## *Swap* implementation

The *swap* function Map::swap(Map& other) swaps data members from the Map *other* and the map it’s called on as follows:

*swap m\_head pointers  
swap m\_tail pointers  
swap m\_size*

## *find* implementation

The private *find* function *Map::Pair\* Map::find(const KeyType& key)* finds a pointer to the (key, value) pair with the desired key as follows:

*initialise a Pair pointer to m\_head*

*repeatedly, if the pointer isn’t the null pointer and we haven’t yet found the right pair:*

*update the pointer to point to the next pair (or null pointer, if we’ve reached end of list)*

*return pointer*

**Note**: This function returns a pointer to the desired Pair object if such an object exists, and a null pointer otherwise.

## doInsertOrUpdate implementation

The private function *Map::doInsertOrUpdate(const KeyType& key, const ValueType& value, bool mayInsert, bool mayUpdate)* inserts or updates a pair in the map per instructions codified in *mayInsert* and *mayUpdate* as follows:

*find pointer to the pair with key key (if it exists) using find*

*if pair found:*

*if allowed to update (per mayUpdate):*

*update m\_value of pair to value  
 return mayUpdate (true if updated, false if not)*

*if pair not found and not allowed to insert new pair:*

*return false*

*// at this point, we must have not found a pair and be allowed to insert one; adding new pair at the // end of the linked array*

*create new pair with m\_key key, m\_value value, m\_next null pointer*

*if list is currently empty:*

*update m\_head to point to new pair*

*set m\_prev of new pair to null pointer*

*else:*

*set m\_prev of new pair to m\_tail (pointer of previously last pair)*

*set m\_next of m\_tail (previously last pair) to a pointer to the new pair*

*update m\_tail to point to new pair and increment m\_size*

*return true;*

## *combine* implementation

The non-Map *combine* function *combine(const Map& m1, const Map& m2, Map& result)* takes the union of pairs with unique keys from maps *m1* and *m2* and stores it into *result* as follows:

*overwrite result with m1 using result = m1*

*for each pair of m2:*

*if m1 doesn’t have the key:*

*insert pair in result (since pair is only in m2)*

*else if m1 also has the key:*

*if the values associated with the key in m1 and m2 aren’t equal:*

*remove the pair from result*

*if the values are the same:*

*do nothing, since pair already in result (which was initialised to m1)*

*if a key was found in both maps with different values:*

*return false*

*else*

*return true*

## *reassign* implementation

The non-Map *reassign* function *reassign(const Map& m, Map& result)* changes *result* to a map containing, for each pair *p1* in *m*, a pair with *p1*’s key mapping to a value copied from a different pair *p2* in *m* as follows:

*create a new map (say, result\_map) to store the new mappings*

*for each pair of m (as returned by get):*

*find the key and value in that pair using get*

*find the key and value of the next pair using get*

*(note: if the current pair is the last pair returned by get, the “next” pair is the first pair)*

*in result\_map, insert a new pair with the key of the current pair and the value of the next pair*

*overwrite result with result\_map using the = operator*

Taking the approach where we overwrite *result* with our locally-created result\_map has two advantages:

1. We no longer have to worry about the case where *result* is a non-empty map, since it’ll be overwritten
2. We implicitly deal with the case where *m* and *result* refer to the same maps, since we don’t modify *result* until we have already created the desired map

# Test Cases

**Note**: Some of the test cases mentioned may use a function *map\_info(const Map& m);* which returns a string with the format “key\_1 value\_1 key\_2 value\_2 … key\_n value\_n ” if Map *m* has n (key, value) pairs. By using assert statements that call this function, we’re directly testing the presence of expected (key, value) pairs in the map.

Also note that the assert statements using *map\_info* test its returned string based on the implementation of my three-member *get* function – these test cases might not work on alternate implementations of *get*, since the order in which *map\_info* returns (key, value) pairs may be different than that written in the assert statements.

All tests assume KeyType is *std::string* and ValueType is *double*.

***// testing default constructor***

*Map test;*

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

***// testing copy constructor***

*Map test; test.insert("Fred", 3.29); test.insert("Ethel", 3.93); // creating a map*

*Map test\_copy = test; // creating a new map test\_copy with copy constructor*

*assert(test\_copy.size() == 2 && map\_info(test\_copy) == "Fred 3.29 Ethel 3.93 ");*

*// checking whether test\_copy was initialised correctly*

***// testing = operator***

*Map test; test.insert("Fred", 3.29); test.insert("Ethel", 3.93); // creating a map*

*test\_equal = test;*

*assert(test\_equal.size() == 2 && map\_info(test\_equal) == "Fred 3.29 Ethel 3.93 ");*

*// testing whether test was copied into test\_equal correctly*

*test\_equal = test\_equal;*

*assert(test\_equal.size() == 2 && map\_info(test\_equal) == "Fred 3.29 Ethel 3.93 ");  
 // testing aliasing; test\_equal should remain the same*

***// testing empty and size functions***

*Map test;*

*assert(test.empty() && test.size() == 0); // m is empty, has size 0*

*assert(test.insert("abc", 1.5)); // preliminary test of insert; should return true*

*assert(!test.empty() && test.size() == 1); // m is no longer empty, has size 1*

***// testing insert function***

*Map test;*

*for (int i = 0; i != 10; i++)*

*assert(test.insert(to\_string(i), i));*

*// creates a map of first ten integer strings and corresponding values*

*assert(test.erase("0")); // preliminary test of erase; should return true*

*assert(!test.insert("1", int('1'))); // “1” already in insert\_test; should return false*

***// testing update function***

*Map test;*

*test.insert("test key", 1.21);*

*assert(test.update("test key", 85)); // updates value associated with test key and returns true*

*assert(!test.update("some other key", 85)); // "some other key" not in map, returns false*

***// testing insertOrUpdate function***

*Map test;*

*for (int i = 0; i != 10; i++)*

*test.insertOrUpdate(to\_string(i), i);*

*// inserts first ten number strings and corresponding values*

*double updated\_value;*

*assert(test.insertOrUpdate("1", -1) && test.get("1", updated\_value) && updated\_value == -1);*

*// first test updates "1" value to -1 and returns true via insertOrUpdate*

*// second test changes updated\_value to new value associated with "1" and returns true*

*// third test checks whether updated value is -1*

*assert(test.insertOrUpdate("abc", 1));*

*// "abc" not in list, and therefore can be added -> returns true*

***// testing erase function***

*Map test;*

*test.insert("abc", 1);*

*assert(!test.erase("xyz")); // should return false, since "xyz" not in map*

*assert(test.erase("abc") && test.size() == 0 && !test.contains("abc"));*

*// erase "abc" -> size of test is 0, and test no longer contains "abc"*

***// testing contains function***

*Map test;*

*test.insert("abc", 1);*

*assert(test.contains("abc") && !test.contains("xyz")); // "abc" in map, "xyz" not*

***// testing two-member get function***

*Map test;*

*test.insert("Fred", 3.29); test.insert("Ethel", 3.93); test.insert("Jack", 3.12);*

*double value;*

*assert(test.get("Fred", value) && value == 3.29); // value modified to "Fred" value 3.29*

*assert(test.get("Ethel", value) && value == 3.93); // value modified to "Ethel" value to 3.92*

*assert(!test.get("Lucy", value)); // "Lucy" not in map, get returns false*

***// testing three argument get function***

*Map test;*

*string key = "this should not be changed"; double val = -1;*

*for (int i = -1; i != 10; i++)*

*assert(!test.get(i, key, val));*

*// since the map is empty, key and val should be unchanged by the get function*

*test.insert("Fred", 3.29); test.insert("Ethel", 3.93); test.insert("Lucy", 3.88);  
assert(map\_info(test) == "Fred 3.29 Ethel 3.93 Lucy 3.88 ");*

*// checking whether get returned each key and value*

*assert(!test.get(3, key, val); 3 out of bounds, so get returns false*

***// testing swap function***

*Map test1; test1.insert("Fred", 2.956);*

*Map test2; test2.insert("Ethel", 3.538); test2.insert("Lucy", 2.956);*

*test1.swap(test2); // swap test1 and test2*

*assert(test1.size() == 2 && map\_info(test1) == "Ethel 3.538 Lucy 2.956 " &&*

*test2.size() == 1 && map\_info(test2) == "Fred 2.956 ");*

*// testing whether maps were swapped successfully*

*// testing whether swap takes care of aliasing*

*test1.swap(test1);*

*assert(map\_info(test1) == “Ethel 3.538 Lucy 2.956 “);*

*// test1 should remain unchanged*

***// testing combine function***

*Map m1; m1.insert("Fred", 123); m1.insert("Ethel", 456); m1.insert("Lucy", 789);*

*Map m2; m2.insert("Lucy", 789); m2.insert("Ricky", 321);*

*Map result; result.insert("Jack", 129);*

*// testing combine with a non-empty result map*

*assert(combine(m1, m2, result)); // no pairs with conflicting values -> combine returns true*

*assert(result.size() == 4 && map\_info(result) == "Fred 123 Ethel 456 Lucy 789 Ricky 321 ");*

*// testing case where m1 and m2 have conflicting (key, value) pairs*

*m2.update("Lucy", 654); // updating value associated with "Lucy" so it's no longer 789*

*// values associated with Lucy are now different in m1 and m2*

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

*assert(result.size() == 3 && map\_info(result) == "Fred 123 Ethel 456 Ricky 321 ");  
 // “Lucy” not in result*

***// testing reassign function***

*Map result; result.insert("Jack", 84); // non-empty result map*

*Map m;*

*// calling reassign on empty map m and non-empty map result*

*reassign(m, result);*

*assert(result.size() == 0); // result should be empty*

*// calling reassign on map m with single entry and non-empty map result*

*m.insert("Fred", 123);*

*reassign(m, result);*

*assert(result.size== 1 && map\_info(result) == "Fred 123 ");*

*// calling reassign on map m with two entries and non-empty map result*

*m.insert("Ethel", 456);*

*reassign(m, result); // exchange Fred and Ethel's values*

*assert(result.size== 2 && map\_info(result) == "Fred 456 Ethel 123 ");*

*// calling reassign on map m with four entries and non-empty map result*

*m.insert("Lucy", 789); m.insert("Ricky", 321);*

*reassign(m, result);*

*assert(resultsize() == 4 && map\_info(result) == "Fred 456 Ethel 789 Lucy 321 Ricky 123 ");*

*// calling reassign on map m with four entries (where two entries have same value) and non-empty map result*

*m.update("Lucy", 456);*

*reassign(m, result);*

*assert(result.size() == 4 && map\_info(result) == "Fred 456 Ethel 456 Lucy 321 Ricky 123 ");*

*// calling reassign on map m with itsel to check whether the result map (in this case, m itself) is modified correctly*

*reassign(m, m);*

*assert(result.size() == 4 && map\_info(result) == "Fred 456 Ethel 456 Lucy 321 Ricky 123 ");*