Kelly Yu

Dr. Smallberg

1/24/22

Monday

1. Doubly Linked List Description

My doubly linked list is a circular linked list that has a dummy Pair. My dummy Pair has pointers to the first and last Pair, and my first and last Pair each have a pointer to the dummy Pair. Each Pair consists of a key, value, pointer to the previous Pair, and pointer to the next Pair. The Pairs in every linked list are in ascending order, determined by each Pair’s key.

Initially, the dummy Pair does not point to any Pairs; its pointers point to null pointer. The circular linked list created after inserting the first Pair, where the dummy Pair points to the first/last Pair, and the first/last Pair points back to the dummy Pair. If all items are erased during the program, both of dummy Pair’s pointers will point back to dummy Pair, rather than null pointers. This won’t be an issue; if we add the first/last item to a fully erased linked list, the circular linked list will still successfully be created the same way as before.

2. Pseudocode

include map header file + iostream

// map’s constructor

set empty list’s size to zero

set dummy Pair’s pointers to null pointers

// map’s copy constructor using an existing map

if existing map is empty

construct new empty map

else

set new map’s size to existing map’s size

start from new map’s dummy Pair,

loop through existing map’s Pairs once

create new Pair

assign new Pair’s data with existing map’s Pair’s data

link new Pair to the last Pair we are at in the new map

link last Pair we are at in the new map to new Pair

move down to next Pair (the new Pair) in the new map

move down to next Pair in existing Map

link the last Pair added to new nap to the new nap’s dummy Pair

link the new map’s dummy Pair to the last Pair added

// map’s assignment operator; assign existing map to another map

if no aliases exist

create copy of the other map

swap existing map with copy of the other map

return the updated existing map

//map’s destructor; deallocate Pairs in map’s linked list

if list is empty

do nothing and return

else

starting from the first Pair

loop through the map’s list once

use a temp pointer to hold onto address of the Pair we delete

move down to the next Pair

delete the previous Pair pointer by temp

//map’s empty function

return true only if size is 0

//map’s size function

return number of Pairs in map

//map’s insert function

if there’s already a Pair with the key we want to insert

don’t insert, return false

otherwise

if it’s the first Pair we are adding to the map

create a new Pair with the data passed into the function

link new Pair to map’s dummy Pair

link map’s dummy Pair to new Pair

increment number of Pairs in map by 1

return true

otherwise find the right space to add new Pair in the middle of the map

loop through the map until we find the Pair with the key greater than our new Pair’s key

add new Pair right before that Pair

link new pair to preceding and seceding Pairs

disconnect preceding and seceding Pairs and link them to new Pair

increment number of Pairs in map by 1

return true

otherwise

add new Pair and link it with to the current last Pair in map

link it with dummy Pair

increment number of Pairs in map by 1

return true

// map’s update function

if list is empty

do nothing, return true

otherwise

starting from our first Pair

traverse through map

if we find a Pair with matching key

update Pair with new value

return true

otherwise

return false

// map’s insert or update function

insert or update & return true

// map’s erase function

if list is empty

return false

otherwise

starting from our first Pair

traverse through map

if we find Pair with matching key

if we don’t find the Pair with matching key

return false

otherwise, if we find the Pair with matching key

link this Pair’s seceding Pair to this Pair’s preceding Pair

link this Pair’s preceding Pair to this Pair’s seceding Pair

delete the Pair

decrement the number of Pairs in map

return true

// map’s contains function

if map is empty

return false

otherwise

traverse through map

if Pair with matching key is found

return true

otherwise

return false

// map’s get function with 2 parameters

if map is empty

return false

otherwise

traverse through map

if Pair with matching key is found

update the value parameter with the value that the Pair’s key maps to

return true

otherwise

return false

// map’s get function with 3 parameters

if map is empty or integer is out of bounds

return false

otherwise

traverse through the list

retrieve pointer to the integer-th Pair

update that Pair’s key and value

return true

// map’s swap function

swap this map’s dummy Pair with the other map’s dummy Pair

if other’s map isn’t empty

link other’s map’s first and last Pair to this map’s new dummy Pair

otherwise

set this map’s new dummy Pair’s pointers to null pointer

if this map isn’t empty

link this’s map’s first and last Pair to other map’s new dummy Pair

otherwise

set other’s map’s new dummy Pair’s pointers to null pointer

swap this map’s size with other map’s size

// merge function

if m1, m2, and result all refer to same map

do nothing, return true

if m1 and m2 refer to same map

result becomes a copy of m1

return true

if m1 and m2 are both empty

result becomes an empty map

return true

if only m1 is empty

result becomes m2

return true

if only m2 is empty

result becomes m1

return true

otherwise

create temporary map to store valid Pairs to merge

create variables to store m1’s Pairs’ data and m2’s Pairs’ data

create a Boolean to track if at least 1 invalid merge is found

loop through each Pair in m1

loop through each Pair in m2 for a Pair with matching keys to each pair in m1

if matching key is found

if value also matches, insert Pair to temporary map

otherwise, an invalid Pair is found, set Boolean to false

break out of loop

if the Pair in m1 is not found in m2

insert the Pair to temporary map

loop through each Pair in m2

loop through each Pair in m2 for a Pair with matching keys to each pair in m1

if matching key is found

skip to find m2’s next Pair in m1

if Pair in m2 is not found in m1

insert the Pair to temporary map

give result the temporary map

return the Boolean (true if no invalid Pairs; false if at least 1 invalid Pair)

// reassign function

if m is empty or only contains 1 Pair

result is the same as m

exit function

otherwise

create variables to store m1’s current Pair’s data and next Pair’s data

create temporary map to store reassigned version of m

traverse through m once, up to the second to last Pair

retrieve each Pair’s data and its next Pair’s data

insert to temp the Pair’s key and the next Pair’s value

insert to temp m’s last Pair’s key and m’s first Pair’s value

give result the temporary map

}

3. Test Cases

All tests are performed on a map that maps strings to doubles. The test code is placed into a main function in a .cpp file with the following headers:

#include "Map.h"

#include <iostream>

#include <cassert>

Map constructor

|  |  |
| --- | --- |
| **Constructing Map with an empty list**  should not be able to update or find any keys in the map | Map z1;  assert(z1.size() == 0 && z1.empty() == **true**);  assert(!z1.update("A", 1) && !(z1.contains("B"))); |

Size and Empty functions

|  |  |
| --- | --- |
| //oo has 1 item  //pp has 2 items  //ss added 1 and erased 1 item == 0 items | Map oo, pp, ss;  assert(oo.insert("A", 1) && pp.insert("AA", 11) && pp.insert("BB", 22));  assert(ss.insert("A", 1) && ss.erase("A"));  assert(oo.size() == 1 && !oo.empty());  assert(pp.size() == 2 && !pp.empty());  assert(ss.size() == 0 && ss.empty()); |

Insert Function

|  |  |
| --- | --- |
| 1. adding to an empty list  2. adding to the rear  3. adding to the front  4. adding to the middle  Output should be:  **A1**  **B2**  **C3**  **D4** | Map m1;  assert(m1.insert("B", 2));  assert(m1.insert("D", 4));  assert(m1.insert("A", 1));  assert(m1.insert("C", 3));  assert(m1.size() == 4);    KeyType k;  ValueType v;  **for**(**int** i=0; i<m1.size(); i++){  m1.get(i, k, v);  std::cerr << k << v << std::endl;  } |
| Adding 1 item to a fully erased list; not originally empty, but became empty during program  There is 1 item.  Output should be:  **D4** | Map m1;  assert(m1.insert("B", 2) && m1.erase("B"));  assert(m1.insert("D", 4));  assert(m1.size() == 1);    KeyType k;  ValueType v;  **for**(**int** i=0; i<m1.size(); i++){  m1.get(i, k, v);  std::cerr << k << v << std::endl;  } |

Update Function

|  |  |
| --- | --- |
| Updating filled list with Pair with matching key and nonmatching key  Output should be:  **A10** | Map m1;  assert(m1.insert("A", 1));  assert(m1.update("A", 10));  assert(!m1.update("E", 10));    KeyType k;  ValueType v;  **for**(**int** i=0; i<m1.size(); i++){  m1.get(i, k, v);  std::cerr << k << v << std::endl;  } |
| Updating empty list  Output: | Map m1;  assert(!m1.update("A", 10));  assert(!m1.update("E", 10));  assert(!m1.update("S", 10));  KeyType k;  ValueType v;  **for**(**int** i=0; i<m1.size(); i++){  m1.get(i, k, v);  std::cerr << k << v << std::endl;  } |

Insert or Update Function

|  |  |
| --- | --- |
| Inserts a Pair since no Pairs in list have key E  Output:  **E5** | Map m1;  assert(m1.insertOrUpdate("E", 5) && m1.size() == 1);  KeyType k;  ValueType v;  **for**(**int** i=0; i<m1.size(); i++){  m1.get(i, k, v);  std::cerr << k << v << std::endl;  } |
| Updates a Pair since there’s a Pair in list with key E  Output:  **E50** | Map m1;  assert(m1.insert ("E", 5) && m1.size() == 1);  assert(m1.insertOrUpdate("E", 50) && m1.size() == 1);  KeyType k;  ValueType v;  **for**(**int** i=0; i<m1.size(); i++){  m1.get(i, k, v);  std::cerr << k << v << std::endl;  } |

Contains Function

|  |  |
| --- | --- |
| List contains Pair with matching key E | Map m1;  assert(m1.insert ("E", 5) && m1.contains("E")); |
| List does not contain any Pairs with matching key A | Map m1;  assert(m1.insert ("E", 5) && !m1.contains("A")); |
| List is empty; no matches | Map m1;  assert(!m1.contains("A")); |

Erase Function

|  |  |
| --- | --- |
| Can’t erase anything from empty lists | Map m1;  assert(!m1.erase(“Z”)); |
| Filled list, but no matches found; nothing is erased  Output:  **F6**  **H8** | Map m1;  assert(m1.insert(“F”, 6) && m1.insert(“H”, 8));  assert(!m1.erase(“Z”) && m1.size() == 2);  KeyType k;  ValueType v;  **for**(**int** i=0; i<m1.size(); i++){  m1.get(i, k, v);  std::cerr << k << v << std::endl;  } |
| Filled list; 1 match found, and Pair is erased from list; list should be 1 size smaller  Output:  **H8** | Map m1;  assert(m1.insert(“F”, 6) && m1.insert(“H”, 8));  assert(!m1.erase(“Z”) && m1.size() == 2);  assert(m1.erase(“F”) && m1.size() == 1);  KeyType k;  ValueType v;  **for**(**int** i=0; i<m1.size(); i++){  m1.get(i, k, v);  std::cerr << k << v << std::endl;  } |
| Erase first Pair  Output:  **G7**  **H8** | Map m1;  assert(m1.insert(“F”, 6) && m1.insert(“H”, 8) && m1.insert(“G”, 7));  assert(m1.erase(“F”));  KeyType k;  ValueType v;  **for**(**int** i=0; i<m1.size(); i++){  m1.get(i, k, v);  std::cerr << k << v << std::endl;  } |
| Erase middle Pair  Output:  **F6**  **H8** | Map m1;  assert(m1.insert(“F”, 6) && m1.insert(“H”, 8) && m1.insert(“G”, 7));  assert(m1.erase(“G”));  KeyType k;  ValueType v;  **for**(**int** i=0; i<m1.size(); i++){  m1.get(i, k, v);  std::cerr << k << v << std::endl;  } |
| Erase last Pair  Output:  **F6**  **G7** | Map m1;  assert(m1.insert(“F”, 6) && m1.insert(“H”, 8) && m1.insert(“G”, 7));  assert(m1.erase(“H”));  KeyType k;  ValueType v;  **for**(**int** i=0; i<m1.size(); i++){  m1.get(I, k, v);  std::cerr << k << v << std::endl;  } |

**bool** get(**const** KeyType& key, ValueType& value) **const**;

|  |  |
| --- | --- |
| Empty list; v stays same, always returns false | Map m2;  KeyType k = "Happy";  ValueType v = 12;    assert(!m2.get(k,v) && v == 12); |
| Filled list; no Pair with matching key; v stays same, returns false | Map m2;  KeyType k = "Happy";  ValueType v = 12;  assert(m2.insert("A", 11) && m2.insert("B", 22));  assert(!m2.get(k,v) && v == 12); |
| Filled list; 1 Pair with matching key, v updated to value that that Pair’s key maps to (11) | Map m2;  KeyType k = "A";  ValueType v = 12;  assert(m2.insert("A", 11) && m2.insert("B", 22));  assert(m2.get(k,v) && v == 11); |

**bool** get(**int** i, KeyType& key, ValueType& value) **const**;

|  |  |
| --- | --- |
| Empty list; k and v are not updated, returns false | Map m3;  KeyType k = "Happy";  ValueType v = 12;  assert(!m3.get(0, k, v) && k == "Happy" && v == 12); |
| Filled list; replaces k and v with data from 1st Pair in the list | Map m3;  KeyType k = "Happy";  ValueType v = 12;  assert(m3.insert("Sad", 90) && m3.insert("IDK", 56) && m3.insert("Meh", 34));  assert(m3.get(0, k, v) && k == "IDK" && v == 56); |
| Filled list; replaces k and v with data from last Pair in the list | Map m3;  KeyType k = "Happy";  ValueType v = 12;  assert(m3.insert("Sad", 90) && m3.insert("IDK", 56) && m3.insert("Meh", 34));  assert(m3.get(2, k, v) && k == "Sad" && v == 90); |
| Filled list; can’t replace k and v with data from 4th Pair in list; list only contains 3 Pairs | Map m3;  KeyType k = "Happy";  ValueType v = 12;  assert(m3.insert("Sad", 90) && m3.insert("IDK", 56) && m3.insert("Meh", 34));  assert(!m3.get(3, k, v) && k == "Happy" && v == 12); |
| Filled list; can’t replace k and v with data from 0th Pair in the list (0th Pair is before the first Pair) | Map m3;  KeyType k = "Happy";  ValueType v = 12;  assert(m3.insert("Sad", 90) && m3.insert("IDK", 56) && m3.insert("Meh", 34));  assert(!m3.get(-1, k, v) && k == "Happy" && v == 12); |

Swap Function

|  |  |
| --- | --- |
| Swapping 2 empty lists | Map f1;  Map f2;  f2.swap(f1);  assert(f1.size() == 0 && f2.size() == 0); |
| Swapping an empty list for a filled list | Map f2, f3;  assert(f3.insert("LOL", 4));  f2.swap(f3);  assert(f2.size()==1 && f3.size()==0);  assert(f2.contains("LOL") && !f3.contains("LOL")); |
| Swapping a filled list for an empty list  Output: | Map f2, f3;  assert(f2.insert("LOL", 4));  f2.swap(f3);  assert(f2.size()==0 && f3.size()==1);    KeyType k;  ValueType v;  **for**(**int** i=0; i<f2.size(); i++){  f2.get(i, k, v);  std::cerr << k << v << std::endl;  } |
| Swapping 2 filled maps of same size  g1 contains 2 items from g2  g2 contains 2 items from g1 | Map g1, g2;  assert(g1.insert("Rawr", 3) && g1.insert("Yawn", 7));  assert(g2.insert("Argh", 1) && g2.insert("Ring", 2));  g2.swap(g1);    assert(g1.size()==2 && g2.size() == 2);  assert(g1.contains("Argh") && g1.contains("Ring"));  assert(g2.contains("Rawr") && g2.contains("Yawn")); |
| Swapping 2 filled maps of different sizes  g1 contains 3 items from g2  g2 contains 2 items from g1 | Map g1, g2;  assert(g1.insert("Rawr", 3) && g1.insert("Yawn", 7) && g1.insert("Zoink", 18));  assert(g2.insert("Argh", 1) && g2.insert("Ring", 2));  g2.swap(g1);    assert(g1.size()==2 && g2.size() == 3);    assert(g1.contains("Argh") && g1.contains("Ring"));  assert(!g1.contains("Rawr") && !g1.contains("Yawn") && !g1.contains("Zoink"));    assert(!g2.contains("Argh") && !g2.contains("Ring"));  assert(g2.contains("Rawr") && g2.contains("Yawn") && g2.contains("Zoink")); |
| Swapped g1 with g2  After the swap, g2’s first Pair (which used to be g1’s first Pair) should be linked to g2’s dummy;  After deleting g2’s first Pair, g2’s new first Pair was what was its second Pair.  And vice versa.  *aka g2’s new dummy and new list should not be linked at all to g1’s list & vice versa* | Map g1, g2;  assert(g1.insert(“Rawr”, 3) && g1.insert(“Yawn”, 7));  assert(g2.insert(“Argh”, 1) && g2.insert(“Ring”, 2));  g2.swap(g1);  KeyType k = “nada”;  ValueType v = 000;  assert(g2.erase(“Rawr”) && g2.get(0, k ,v));  assert(k ==”Yawn” && v == 7);  assert(g1.erase(“Argh”) && g1.get(0, k ,v));  assert(k ==”Ring” && v == 2); |

Copy Constructor

|  |  |
| --- | --- |
| Constructing with a filled list | Map c1;  assert(c1.insert("apples", 2) && c1.insert("bananas", 4) && c1.insert("cats", 3));  Map c2(c1);  assert(c1.size()==3 && c1.contains("apples") && c1.contains("bananas") && c1.contains("cats") ); |
| Constructing with an empty list | Map d1;  Map d2(d1);  assert(d1.size() == 0 && d2.size() == 0); |

Assignment Operator

|  |  |  |
| --- | --- | --- |
| Assigning empty list to empty list | Map j1;  Map j2;  j1 = j2;  assert(j1.size() == 0 && j2.size() == 0); |  |
| Assigning filled list to an empty list (m4)  List that was assigned should be filled | Map m3, m4;  assert(m3.insert("AA", 111) && m3.insert("BB", 222));  assert(m4.size() == 0);  m4 = m3;  assert(m4.size() == 2 && m4.contains("AA") && m4.contains("BB")); |
| Assigning empty list to filled list (m4)  List that was assigned should be empty | Map m3, m4;  assert(m4.insert("WW", 188) && m4.insert("YY", 199) && m4.insert("ZZ", 221));  assert(m4.size() == 3);  m4 = m3;  assert(m4.size() == 0);  assert(!m4.contains("WW") && !m4.contains("YY") && !m4.contains("ZZ")); |
| Assigning filled list of same size to filled list (m4)  List that was assigned should have same size as before | Map m3, m4;  assert(m3.insert("AA", 111) && m3.insert("BB", 222) && m3.insert("CC", 333));  assert(m4.insert("WW", 188) && m4.insert("YY", 199) && m4.insert("ZZ", 221));  m4 = m3;  assert(m4.size() == 3 && m4.contains("AA") && m4.contains("BB") && m4.contains("CC"));  assert(!m4.contains("WW") && !m4.contains("YY") && !m4.contains("ZZ")); |
| Assigning filled list of larger size to filled list (m4)  List (m4) that was assigned should have a larger size | Map m3, m4;  assert(m3.insert("AA", 111) && m3.insert("BB", 222) && m3.insert("CC", 333));  assert(m4.insert("WW", 188) && m4.insert("YY", 199));  m4 = m3;  assert(m4.size() == 3 && m4.contains("AA") && m4.contains("BB") && m4.contains("CC"));  assert(!m4.contains("WW") && !m4.contains("YY")); |
| Assigning filled list of smaller size to filled list (m4)  List (m4) that was assigned should have a smaller size | Map m3, m4;  assert(m3.insert("AA", 111) && m3.insert("BB", 222));  assert(m4.insert("WW", 188) && m4.insert("YY", 199) && m4.insert("ZZ", 221));  m4 = m3;  assert(m4.size() == 2 && m4.contains("AA") && m4.contains("BB"));  assert(!m4.contains("WW") && !m4.contains("YY") && !m4.contains("ZZ")); |

Merge Function

|  |  |
| --- | --- |
| Passing 3 empty maps; result will be empty list | Map a1, a2, a3;  assert(merge(a1, a2, a3));  assert(a3.size() == 0); |
| Passing in 3 different maps,  1 unique in each map, 1 identical, 1 invalid  3 in result; returns false | Map b1, b2, b3;  assert(b1.insert("A", 1) && b1.insert("B", 2) && b1.insert("D", 0));  assert(b2.insert("A", 1) && b2.insert("C", 3) && b2.insert("D", 9));  assert(b3.insert("E", 5) && b3.insert("F", 6));  assert(!merge(b1, b2, b3));  assert(b3.size()==3 && b3.contains("A") && b3.contains("B") && b3.contains("C")); |
| Passing in 3 different Maps  1 unique in each map, 1 identical, 0 invalid  3 in result; Returns true | Map b1, b2, b3;  assert(b1.insert("A", 1) && b1.insert("B", 2));  assert(b2.insert("A", 1) && b2.insert("C", 3));  assert(b3.insert("E", 5) && b3.insert("F", 6));  assert(merge(b1, b2, b3));  assert(b3.size()==3 && b3.contains("A") && b3.contains("B") && b3.contains("C")); |
| Passing in 3 different Maps  1 unique in each map, 0 identical, 1 invalid  2 in result; Returns false | Map b1, b2, b3;  assert(b1.insert("A", 1) && b1.insert("B", 2));  assert(b2.insert("B", 1) && b2.insert("C", 3));  assert(b3.insert("E", 5) && b3.insert("F", 6));  assert(!merge(b1, b2, b3));  assert(b3.size()==2 && b3.contains("A") && b3.contains("C")); |
| Passing in 3 different Maps  1 unique in each map, 0 identical, 0 invalid  2 in result; Returns true | Map b1, b2, b3;  assert(b1.insert("A", 1));  assert(b2.insert("B", 1));  assert(b3.insert("E", 5) && b3.insert("F", 6));  assert(merge(b1, b2, b3));  assert(b3.size()==2 && b3.contains("A") && b3.contains("B")); |
| Passing in 3 different Maps  0 unique in each map, 1 identical, 1 invalid  1 in result; Returns false | Map b1, b2, b3;  assert(b1.insert("A", 1) && b1.insert("B", 5));  assert(b2.insert("A", 1) && b2.insert("B", 0));  assert(b3.insert("E", 5) && b3.insert("F", 6));  assert(!merge(b1, b2, b3));  assert(b3.size()==1 && b3.contains("A")); |
| Passing in 3 different Maps  0 unique in each map, 1 identical, 0 invalid  1 in result; Returns true | Map b1, b2, b3;  assert(b1.insert("A", 1));  assert(b2.insert("A", 1));  assert(b3.insert("E", 5) && b3.insert("F", 6));  assert(merge(b1, b2, b3));  assert(b3.size()==1 && b3.contains("A")); |
| Passing in 3 different maps,  0 unique in each map, 0 identical, 2 invalid  No merges; returns false | Map b1, b2, b3;  assert(b1.insert("A", 1) && b1.insert("B", 2));  assert(b2.insert("A", 2) && b2.insert("B", 1));  assert(b3.insert("E", 5) && b3.insert("F", 6));  assert(!merge(b1, b2, b3));  assert(b3.size()==0); |
| Passing in 3 different maps,  1 unique in each map, 1 identical, 2 invalid  3 in result, returns false | Map b1, b2, b3;  assert(b1.insert("A", 1) && b1.insert("B", 3) && b1.insert("D", 9) && b1.insert("C", 9));  assert(b2.insert("A", 1) && b2.insert("E", 4) && b2.insert("D", 0) && b2.insert("C", 8));  assert(b3.insert("E", 5) && b3.insert("F", 6));  assert(!merge(b1, b2, b3));  assert(b3.size()==3 && b3.contains("A") && b3.contains("B") && b3.contains("E")); |
| Passing in 3 different maps, where the 3rd one is empty  Result is no longer empty; contains all valid merged Pairs | Map b1, b2, b3;  assert(b1.insert("A", 1) && b1.insert("B", 2));  assert(b2.insert("C", 3) && b2.insert("D", 4));  assert(merge(b1, b2, b3));  assert(b3.size()==4 && b3.contains("A") && b3.contains("B") && b3.contains("C") && b3.contains("D")); |
| Passing in 3 different maps, where the 3rd one is originally filled  Result is used to have 4; now it only contains the 3 valid merged Pairs | Map b1, b2, b3;  assert(b1.insert("A", 1) && b1.insert("B", 2));  assert(b2.insert("C", 3) && b2.insert("D", 4));  assert(b3.insert("E", 5) && b3.insert("F", 6));  assert(merge(b1, b2, b3));  assert(b3.size()==4 && b3.contains("A") && b3.contains("B") && b3.contains("C") && b3.contains("D")); |
| **ALIASING** Passing 2 different maps, where the merging 2 are the same, and result is different  Result == copy of the merging map; returns true, every pair used to merge is valid | Map b1, b2;  assert(b1.insert("A", 1) && b1.insert("B", 2));  assert(merge(b1, b1, b2));  assert(b2.size() == b1.size() && b2.contains("A") && b2.contains("B")); |
| **ALIASING**  Passing in 2 different maps, where the 1st of the 2 merging maps is the same as the result map  There’s 2 Pairs in identical maps, and one of them is invalid; so, only 1 Pair is merged to result;  Returns false; | Map b1, b2;  assert(b1.insert("A", 1) && b1.insert("B", 2));  assert(b2.insert("B", 4));  assert(!merge(b1, b2, b1));  assert(b1.size() == 1 && b1.contains("A")); |
| **ALIASING**  Passing in 2 different maps, where the 2nd of the 2 merging maps is the same as the result map  Same thing | Map b1, b2;  assert(b1.insert("A", 1) && b1.insert("B", 2));  assert(b2.insert("B", 4));  assert(!merge(b1, b2, b2));  assert(b2.size() == 1 && b2.contains("A")); |
| **ALIASING**  Passing in 3 identical maps  Result stays identical to “other” maps | Map b1;  assert(b1.insert("A", 1) && b1.insert("B", 2));  assert(merge(b1, b1, b1));  assert(b1.size() == 2 && b1.contains("A") && b1.contains("B")); |
| 1st map == empty; 2nd map == empty  Result == empty | Map b1, b2, b3;  assert(merge(b1, b2, b3));  assert(b3.size() == 0); |
| 1st map == empty; 2nd map == filled  Result == second map | Map b1, b2, b3;  assert(b2.insert("A", 1) && b2.insert("B", 2));  assert(merge(b1, b2, b3));  assert(b3.size() == 2 && b3.contains("A") && b3.contains("B")); |
| 1st map == filled; 2nd map == empty  Result == first map | Map b1, b2, b3;  assert(b1.insert("A", 1) && b1.insert("B", 2));  assert(merge(b1, b2, b3));  assert(b3.size() == 2 && b3.contains("A") && b3.contains("B")); |

Reassign Function

|  |  |
| --- | --- |
| Original list = empty  Result = empty  Result = empty | Map y1, y2;  reassign(y1, y2);  assert(y1.size() == y2.size() && y2.size()==0); |
| Original list = empty  Result = filled  Result = empty | Map y1, y2;  assert(y2.insert("Hey", 200));  reassign(y1, y2);  assert(y1.size() == y2.size() && y2.size()==0); |
| Original list = filled  Result = empty  Result = filled with reassigned copy of original list  Output:  **Ally 2**  **Belly 3**  **Cella 1** | Map y1, y2;  assert(y1.insert("Ally", 1) && y1.insert("Belly", 2) && y1.insert("Cella", 3));  reassign(y1, y2);  assert(y1.size() == y2.size() && y2.size()==3);  KeyType k;  ValueType v;  **for**(**int** i=0; i<y2.size(); i++){  y2.get(i, k, v);  std::cerr << k << " " << v << std::endl;  } |
| Original list = filled  Result = filled  Result = filled with reassigned copy of original list  Output:  **Ally 2**  **Belly 3**  **Cella 1** | Map y1, y2;  assert(y1.insert("Ally", 1) && y1.insert("Belly", 2) && y1.insert("Cella", 3));  assert(y2.insert("Danos", 4));  reassign(y1, y2);  assert(y1.size() == y2.size() && y2.size()==3);  KeyType k;  ValueType v;  **for**(**int** i=0; i<y2.size(); i++){  y2.get(i, k, v);  std::cerr << k << " " << v << std::endl;  } |
| Original list only has 1 Pair; no reassignment  Result = original  Output:  **Ally 1** | Map y1, y2;  assert(y1.insert("Ally", 1));  reassign(y1, y2);  assert(y1.size() == y2.size() && y2.size()==1);  KeyType k;  ValueType v;  **for**(**int** i=0; i<y2.size(); i++){  y2.get(i, k, v);  std::cerr << k << " " << v << std::endl;  } |
| **ALIASING**  Original and result refer to the same map ---- **original has 1 Pair**  Original and result are both reassigned (original, despite const, is altered through result)  Output:  **Ally 1** | Map y1;  assert(y1.insert("Ally", 1));  reassign(y1, y1);  assert(y1.size() == 1);  KeyType k;  ValueType v;  **for**(**int** i=0; i<y1.size(); i++){  y1.get(i, k, v);  std::cerr << k << " " << v << std::endl;  } |
| **ALIASING**  Original and result refer to the same map ---- **original has 2 Pairs**  Original and result are both reassigned (original, despite const, is altered through result)  Output:  **Ally 2**  **Bajo 1** | Map y1;  assert(y1.insert("Ally", 1) && y1.insert("Bajo", 2));  reassign(y1, y1);  assert(y1.size() == 2);  KeyType k;  ValueType v;  **for**(**int** i=0; i<y1.size(); i++){  y1.get(i, k, v);  std::cerr << k << " " << v << std::endl;  } |
| **ALIASING**  Original and result refer to the same map ---- **original is empty**  Original and result are both reassigned (original, despite const, is altered through result) | Map y1;  reassign(y1, y1);  assert(y1.size() == 0); |