Reema Kshetramade

SID: 205328402

Project #2 Report

1. The list has a dummy node which head points to which is present even when the list has 0 relevant items. The list is also circularly linked, so the dummy node’s prev pointer points to the last item in the list and the last item in the list’s next point points to the dummy node. Each node in the list contains the data members of the key and value pair, as well as a next pointer which points to the node next in the list and a prev pointer which points to the node previous in the list. The nodes are in the order of the first inserted at the start of the list to the last inserted at the end of the list.

Map::Map():

Create the dummy node and store its address in the head pointer

Set the next and prev pointers for this node also equal to head (because there are no other items in the list)

Set the size data member equal to 0

Map::~Map():

Traverse through the list, visiting each item once

Store information about the pointer to the next node in the list (which would otherwise be lost when the current

node was deleted)

Delete the current node and move to the next node in the list

Delete the dummy node

Map::Map(const Map &other):

Create a new head pointer and dummy node and set the size of the list equal to zero

While you have not reached the start of other

Insert a new node into the list with the values of the corresponding node

Map& Map::Map operator=(const Map& other):

Delete all the current nodes in the list

Create a new head pointer and dummy node and reset the size of list to zero

While you have not reached the start of other

Insert a new node into the node with the corresponding values

Return the current map

bool Map::insert(const KeyType key, const ValueType value):

Return false if the list already contains key

Otherwise, create a new node and set the data members equal to key and value

Set the next pointer of the new node equal to head because the new node is added at the end of the list

Set the prev pointer of the new node equal to the dummy nodes’ prev pointer which previously pointed to the last

item in the list

Set the dummy node’s prev pointer to the address of the new node

Increment the size data member and return true

bool Map::update(const KeyType key, const ValueType value):

While the start of the list has not been reached, traverse the list

If the key value of the current node is equivalent to the key parameter, update the value and return true

Otherwise move to the next item in the list

Return false because the item was not found in the list

bool Map::insertOrUpdate(const KeyType key, const ValueType value):

If possible, update the key in the list with the value parameter and return true

Otherwise, insert the key into the list and return the result of that insertion (which is always true)

bool Map::erase(const KeyType key):

Traverse the list and if a node is found with that key value, break out of the loop

If the key was found

Set the prev pointer of the next value in the list equal to the previous value to the node to be deleted

Set the next pointer of the previous value equal to the next value in the list after the node to be deleted

Decrement the size and return true

Return false which means the key was not found

bool Map::contains(const KeyType key):

Traverse each value of the list

If any node in the list has a key data member equal to the parameter return true

Return false if no node has a matching key data member

bool Map::get(const KeyType key, ValueType& value):

Traverse the list

If the node has a matching key data member

Set the value parameter equal to the value data member of this node and return true

Return false if no node has a matching key data member

bool Map::get(int i, KeyType& key, ValueType& value):

If the parameter i is less than zero or greater than or equal to the size of the list return false

Otherwise, start at the first relevant item in the list,

Repeat i number of times

Use the next pointer to go the next node in the list

Set the key and value parameters equal to the key and value data members of this current node

Return true

void Map::swap(Map& other):

Exchange the head pointers of the two lists

Exchange the sizes of the two lists

bool combine(const Map& m1, const Map& m2, Map& result)

Create a new map to hold the combinations of m1 and m2 and a variable to hold the return value

Loop through each item in m1 and add it to the local result map variable

Loop through each item in m2

If the key has a duplicate key in m1 and they do not have the same values, do not add it to the result map and set

the return value variable to false

Otherwise, add the key value pair to the local map variable

Set the result map parameter equal to the local map variable

Return the return value variable

void reassign(const Map& m, Map& result)

Store the key and value data of the first item in the map

For each item in the list, except the last

Find the value of the next item in the list and insert into result the key and new value combination

For the last item in the list, insert that key and the value of the first item into the result map

Check if there are any items in result that are not in m and remove them if there are

3. Test Cases:

//Create a map using default constructor

Map m;

//-------Test functions on an empty map:-----

//Check that empty and size functions work on empty map

assert(m.empty() && m.size() == 0);

//Check update works on an empty map

assert(!m.update("Fred", 2.5));

//Check contains works on an empty map

assert(!m.contains("Fred"));

//Check that erase works on an empty map (returns false and does not erase anything)

assert(!m.erase("Fred"));

assert(m.empty() && m.size() == 0);

KeyType k;

ValueType v = 0;

//Check that get works on an empty map (returns false and does not change the value paramter)

assert(!m.get("Fred", v) && v == 0);

//Check that the second implementation of get works when called on an empty map (returns false and does

//not change the map or the key and value parameters)

assert(!m.get(0, k, v) && k == "" && v == 0);

Map ma;

//Check that swap works on an empty map (does nothing and returns false);

m.swap(ma);

assert(m.empty() && ma.empty());

//------Test insert and insertOrUpdate functions on map that is empty:-----

//Insert in an empty map, check empty and size on a map that is not empty

assert(m.insert("Sam", 4.0) && m.size() == 1 && m.contains("Sam"));

assert(m.erase("Sam"));

assert(m.size() == 0 && !m.contains("Sam"));

//Check that insertOrUpdate works on an empty map and the size is incremented correctly

assert(m.insertOrUpdate("Drew", 3.5) && m.size() == 1);

//-------Test functions on maps that have one or more items:-------

//Check that erase works when the key is present

assert(m.erase("Drew") && m.size() == 0);

assert(m.insert("Fred", 2.5) && !m.empty() && m.size() == 1 && m.contains("Fred"));

//Check that erase works when the key is not in the map (returns false and does not change the map)

assert(!m.erase("Drew") && m.size() == 1);

//Check that you cannot insert something with a duplicate key, and size does not increase with this

//failed insertion

assert(!m.insert("Fred", 3.0) && m.size() == 1);

//Check that insertion works on a map which already has items, check that size is incrementing correctly

//and check that contains works for a map with multiple values

assert(m.insert("Mary", 3.5) && m.size() == 2 && m.contains("Mary"));

assert(m.insert("Joe", 4.0) && m.size() == 3 && m.contains("Joe"));

//Check that you cannot insert values with duplicate keys in a map with multiple elements

assert(!m.insert("Joe", 2.0) && !m.insert("Mary", 3.9));

assert(m.insert("George", 3.7) && m.size() == 4 && m.contains("George"));

//Check that the map with multiple elements contains the correct values for the keys of the elements

assert(m.contains("Fred") && m.contains("Mary") && m.contains("Joe") && m.contains("George"));

//Check that update works for a map with multiple items and a key that is in the map

//and the size remains the same

assert(m.update("Fred", 3.0) && m.size() == 4);

assert(m.update("Joe", 2.5) && m.size() == 4);

//Check that update works when the key is not in the map (returns false and does not change the map)

assert(!m.update("Sally", 3.0) && m.size() == 4);

//Check that erase returns false for input value that is not in the map

assert(!m.erase("Hannah"));

Map m2;

//Insert a value into the map and erase a value

assert(m2.insert("Cody", 5.0) && m2.size() == 1 && !m2.empty() && m2.contains("Cody"));

assert(m2.erase("Cody") && m2.size() == 0 && !m2.contains("Cody"));

//Check that get returns false when given a value that has been erased previously

assert(!m2.get("Cody", v) && v == 0);

//Insert a value into the map

assert(m2.insert("Cody", 4.7) && m2.size() == 1);

//Check that get works on a map with one element with an input that is in the map

assert(m2.get("Cody", v) && v == 4.7 && m2.size() == 1);

//Check that get works on a map with one element with an input that is not in the map (returns false

//and does not change the map or the value parameter passed to the function)

assert(!m2.get("Fred", v) && v == 4.7 && m2.size() == 1);

//Insert another value into the map

assert(m2.insert("Noel", 4.6) && m2.size() == 2 && m2.contains("Noel"));

//Check that get works on a map with multiple elements and correctly updates the value parameter

assert(m2.get("Noel", v) && v == 4.6);

assert(m2.insert("Codel", 2.0) && m2.size() == 3 && m2.contains("Codel"));

assert(m2.get("Codel", v) && v == 2.0);

assert(m2.insertOrUpdate("Codel", 2.5));

//Check that get changes the value parameter to the correct value after the item has been updated

assert(m2.get("Codel", v) && v == 2.5);

v = 0;

k = "";

**for** (**int** i = 0; i < m2.size(); i++)

{

KeyType k1;

ValueType v1;

assert(m2.get(i, k1, v1));

k += k1;

v += v1;

}

assert(v == 4.7 + 4.6 + 2.5);

assert(k == "CodyNoelCodel" || k == "NoelCodyCodel" || k == "CodelNoelCody" || k == "CodelCodyNoel" || k == "NoelCodelCody" || k == "CodyCodelNoel");

assert(m2.insert("Talia", 3.5) && m2.size() == 4);

//--------Test variations of swap function:---------

//Create a map using the default constructor

Map m5;

Map m6;

//Check that swap works on two empty maps

m5.swap(m6);

assert(m5.empty() && m6.empty());

//Check that map works on two empty maps that have already been swapped

m6.swap(m5);

assert(m5.empty() && m6.empty());

ValueType v2;

ValueType v3;

//Insert value into one map

assert(m5.insert("Nick", 3.5) && m5.size() == 1);

//Check that swap works with one empty map and one map with size 1

m5.swap(m6);

assert(m6.contains("Nick") && m6.size() == 1 && m5.empty() && !m5.contains("Nick"));

//Insert value into second map

assert(m5.insert("Jack", 2.0) && m5.size() == 1);

//Check that swap works for two maps of size 1

m5.swap(m6);

assert(m6.contains("Jack") && m6.size() == 1 && m5.contains("Nick") && m5.size() == 1);

//Insert more values into the arrays

assert(m5.insert("Joe", 3.0) && m5.size() == 2);

assert(m5.insert("Kevin", 4.0) && m5.size() == 3);

//Check that swap works for maps with multiple values

m5.swap(m2);

//Check that after the swap the maps have the correct sizes and correct key value pairs

assert(m5.size() == 4 && m2.size() == 3);

assert(m5.get("Cody", v2) && v2 == 4.7);

assert(m5.get("Noel", v2) && v2 == 4.6);

assert(m5.get("Codel", v2) && v2 == 2.5);

assert(m5.get("Talia", v2) && v2 == 3.5);

//Check that the other map has the correct key value pairs after the swap

assert(m2.get("Nick", v3) && v3 == 3.5);

assert(m2.get("Joe", v3) && v3 == 3.0);

assert(m2.get("Kevin", v3) && v3 == 4.0);

assert(!m2.get("Bob", v3) && v3 == 4.0);

//------Test variations of using copy constructor:------

//Create a map using the default constructor

Map m7;

//Create a map using the copy constructor and check that the copy constructor works with an empty map

//passed as the parameter

Map m8(m7);

//Check that the map created using the copy constructor is empty

assert(m8.empty());

//Check that values are correctly inserted into a map created using the copy constructor

assert(m8.insert("Bear", 2.5) && m8.size() == 1);

assert(m8.insert("Taylor", 3.5) && m8.size() == 2);

Map m9;

//Insert a value into the map

assert(m9.insert("Brad", 3.6) && m9.size() == 1);

//Check that the copy constructor works when a map with values is passed as the parameter

Map m10(m9);

ValueType v9;

assert(m10.get("Brad", v9) && v9 == 3.6);

assert(m10.insert("Chad", 2.0) && m10.size() == 2);

Map m11;

assert(m9.erase("Brad") && m9.size() == 0);

//Check that the assignment operator works for maps that are empty

m11 = m9;

ValueType v11;

assert(m11.size() == m9.size());

//Check that the assignment operator works when assigning a non empty map to an empty map

m11 = m6;

assert(m11.get("Jack", v11) && v11 == 2);

//Check that the assignment operator works when assigning a non empty map to a non empty map

m11 = m10;

assert(m11.get("Brad", v11) && v11 == 3.6);

assert(m11.get("Chad", v11) && v11 == 2.0);

//--------Test combine function-----

Map map;

Map map1;

Map mapComb;

//Check that combine works with empty maps

assert(combine(map, map1, mapComb));

assert(map.empty() && map1.empty() && mapComb.empty());

//Insert values into the maps

assert(map.insert("Andy", 3.5) && map.insert("Michael", 2.0) && map.insert("Jim", 4.0));

assert(map1.insert("Pam", 4.0) && map1.insert("Erin", 2.0) && map1.insert("Holly", 3.5));

ValueType val;

//Check that combine works for maps that are not empty

assert(combine(map, map1, mapComb));

assert(mapComb.get("Jim", val));

assert(val == 4.0);

assert(mapComb.get("Andy", val) && val == 3.5);

assert(mapComb.get("Michael", val) && val == 2.0 && mapComb.get("Pam", val) && val == 4.0);

assert(mapComb.get("Erin", val) && val == 2.0 && mapComb.get("Holly", val) && val == 3.5);

//Check that combine works when a non empty map is passed in the result parameter

assert(combine(m5, m11, mapComb));

assert(mapComb.get("Cody", val) && val == 4.7);

assert(mapComb.get("Codel", val) && val == 2.5);

assert(mapComb.get("Talia", val) && val == 3.5);

assert(mapComb.get("Noel", val) && val == 4.6);

assert(mapComb.get("Brad", val) && val == 3.6);

assert(mapComb.get("Chad", val) && val == 2.0);

assert(mapComb.size() == 6);

//Check that combine does not have any pairs that were not in the original maps passed as parameters

assert(!mapComb.get("Jim", val) && !mapComb.get("Andy", val));

assert(!mapComb.get("Michael", val) && !mapComb.get("Pam", val));

assert(!mapComb.get("Erin", val) && !mapComb.get("Holly", val));

//Check that when there is a duplicate key with different values the result map does not contain that key

//and returns false because there is a nonmatching duplicate and has the other correct pairs

map.update("Jim", 3.5);

map1.insert("Jim", 4.0);

Map mapComb2;

assert(!combine(map, map1, mapComb2));

assert(!mapComb2.get("Jim", val));

assert(mapComb2.get("Andy", val) && val == 3.5);

assert(mapComb2.get("Michael", val) && val == 2.0 && mapComb2.get("Pam", val) && val == 4.0);

assert(mapComb2.get("Erin", val) && val == 2.0 && mapComb2.get("Holly", val) && val == 3.5);

//Check that when there is a duplicate key with matching values, the result map does contain that key and

//returns true because the duplicates are matching and has the other correct pairs

map1.update("Jim", 3.5);

Map mapComb3;

assert(combine(map, map1, mapComb3));

assert(mapComb3.get("Jim", val) && val == 3.5);

assert(mapComb3.get("Andy", val) && val == 3.5);

assert(mapComb3.get("Michael", val) && val == 2.0 && mapComb3.get("Pam", val) && val == 4.0);

assert(mapComb3.get("Erin", val) && val == 2.0 && mapComb3.get("Holly", val) && val == 3.5);

Map map2(map);

//Check that combine works when one of the map parameter and result are the same map

combine(map2, map1, map2);

assert(map2.get("Jim", val) && val == 3.5);

assert(map2.get("Andy", val) && val == 3.5);

assert(map2.get("Michael", val) && val == 2.0 && map2.get("Pam", val) && val == 4.0);

assert(map2.get("Erin", val) && val == 2.0 && map2.get("Holly", val) && val == 3.5);

//Check that combine works when the first map parameter and result parameter refer to the same map and

//there is a nonmatching duplicate between the two parameter maps

Map map3(map);

map3.update("Jim", 3.6);

combine(map3, map1, map3);

assert(!map3.get("Jim", val));

assert(map3.get("Andy", val) && val == 3.5);

assert(map3.get("Michael", val) && val == 2.0 && map3.get("Pam", val) && val == 4.0);

assert(map3.get("Erin", val) && val == 2.0 && map3.get("Holly", val) && val == 3.5);

//-------Test reassign function--------

Map mp;

Map mapResult;

//Check that reassign works for empty maps (does nothing)

reassign(mp, mapResult);

//Insert values into the map

mp.insert("One", 1);

mp.insert("Two", 2);

mp.insert("Three", 3);

mp.insert("Four", 4);

//Check that reassign works when the first map parameter is not empty and the result map is empty

reassign(mp, mapResult);

assert(mapResult.size() == mp.size());

KeyType key;

ValueType va;

KeyType strMp;

**for** (**int** i = 0; i < mp.size(); i++)

{

mp.get(i, key, va);

strMp += key;

strMp += va;

}

KeyType strRes;

**for** (**int** i = 0; i < mapResult.size(); i++)

{

mapResult.get(i, key, va);

strRes += key;

strRes += va;

}

assert(strMp != strRes);

//Check that reassign works when the result map is not empty

reassign(mp, mapResult);

assert(mapResult.size() == mp.size());

**for** (**int** i = 0; i < mp.size(); i++)

{

mp.get(i, key, va);

strMp += key;

strMp += va;

}

**for** (**int** i = 0; i < mapResult.size(); i++)

{

mapResult.get(i, key, va);

strRes += key;

strRes += va;

}

assert(strMp != strRes);

//Check that reassign works for a map with only one value

Map mp1;

mp1.insert("A", 1);

Map mapResult1;

reassign(mp1, mapResult1);

assert(mapResult1.get("A", va) && va == 1);

assert(mapResult1.size() == 1);

//Check that reassign works when the map and result parameter refer to the same map

reassign(mp, mp);

**for** (**int** i = 0; i < mp.size(); i++)

{

mp.get(i, key, va);

strMp += key;

strMp += va;

}

**for** (**int** i = 0; i < mapResult.size(); i++)

{

mapResult.get(i, key, va);

strRes += key;

strRes += va;

}

assert(strMp != strRes);