1. A 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?)
   1. I created a linked list with a head and tail pointer and three elements in the Node struct, prevNode, value of Pair, and nextNode. The prevNode points to the node linked before it and nextNode points to the node linked after it. The list is not circular nor does it have a dummy node. 
   2. Empty map has nothing in it, no Nodes
2. P[seudocode](http://web.cs.ucla.edu/classes/winter24/cs32/pseudocode.html) for non-trivial algorithms (e.g., Map::erase and reassign).

//Constructor

Map::Map(){

Set size equal to 0

Point head and tail to nullptr since it is empty

}

//Destructor

Map::~Map(){

Check if head is a nullptr

If so, return

Create nodes to clear memory

Repeatedly, as long as copyHead is not nullptr

Make next the copyHead’s next node

Delete copyHead

Set copyHead equal to next

}

//Copy Constructor

Map::Map(const Map& other){

Set size of member function equal to the size of other (passed in Map)

Check if other if empty

Set head and tail to nullptr and return

If other is not empty

Repeatedly, as long as copyHead is not nullptr

Set newNode’s value and key to other’s value and key

Check for if we need to make head the newNode

Increment

}

//Assignment Operator

Map& Map::operator=(const Map& rhs){

Make sure left hand side and right hand side aren’t equal

Create a temp Map since rhs is a constant

Perform swap

Return pointer to this Map

}

//Contains

Map::contains(const KeyType& key) const{

start at the head of the linked list

while traverse is not nullptr:

if traverse's key matches the target key:

return true

move to the next node in the linked list

// If the loop completes without finding the key

return false

}

//Insert

Map::insert(const KeyType& key, const ValueType& value){

If key is in linked list

Return false

Start at the head of the linked list

Create a newNode to insert

Fill it with the key and value passed in

//Nullptr Case

If the linked list is empty

Make head and tail point to newNode

Set head and tail next and prev nodes to nullptr

Increase size

Return true

//Add to head case

If key is less than traverse’s key

Add newNode to the list

Set next node to traverse/head

Set previous

Increase size

Return true

Traverse through the keys greater than yours

//Add to tail case

Set tail to newNode

Set prevNode to traverse and set nextNode to nullptr

Increase size

Return true

//Add to middle

Connect the linked lists by setting nextNode and prevNode in its respective places

}

//Update

Map::update(const KeyType& key, const ValueType& value){

// Start traversal from the head

traverse = head

// Check if the linked list is not empty

if traverse is not nullptr:

// Iterate through the linked list

while traverse.nextNode is not nullptr:

// Check if the key repeats

if key == traverse.value.m\_key:

// Update the value for the matching key

traverse.value.m\_value = value

return true

// Move to the next node

traverse = traverse.nextNode

// The key was not found in the linked list

return false

}

//Insert Or Update

Map::insertOrUpdate(const KeyType& key, const ValueType& value){

// Start traversal from the head

// Iterate through the linked list

while traverse is not nullptr:

// Check if the key repeats

if key == traverse.value.m\_key:

// Update the value for the matching key

// Move to the next node

// Insert the key-value pair if it doesn't exist

// The function was successful

return true

}

//Erase

Map::erase(const KeyType& key){

// If the linked list is empty, return false

if head is nullptr:

return false

// Check if the key matches the head node

// Remove the head node

if head is not nullptr:

Decrease size

delete temp

return true

// Check if the key matches the tail node

// Remove the tail node

Decrease size

delete temp

return true

// Traverse the linked list to find and remove the node with the specified key

temp = head

while temp.nextNode is not nullptr:

temp = temp.nextNode

if temp.value.m\_key == key:

// Remove the node

if next is not nullptr:

next.prevNode = prev

Decrease size

delete temp

return true

// The key was not found in the linked list

return false

}

//Get

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

// Start traversal from the head

// Iterate through the linked list while the key is less than or equal to the specified key

while traverse is not nullptr and traverse.value.m\_key is less than or equal to key:

// Check if the key matches the current node

// Retrieve the corresponding value

return true

// Move to the next node

// The key was not found in the linked list

return false

}

//Get

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

// Check if the index is out of bounds

if i is less than 0 or i is greater than or equal to m\_size:

return false

// Start traversal from the head

// Iterate through the linked list until reaching the specified index for k from 0 to i-1:

// Check if the next node is nullptr (reached the end of the list)

return false

// Move to the next node

// Retrieve the key and value at the specified index

// The function was successful

return true

}

//Swap

Map::swap(Map& other){

Swap sizes

Swap head pointers

Swap tail pointers

}

//Merge

Map::merge(const Map& m1, const Map& m2, Map& result){

// Check if result has values and erase

while result is not empty:

Erase key

// Add m1 into result

for each element i in m1:

Insert key1 and value1

// Add m2 and handle different cases into result

for each element i in m2:

M2 get

if result contains key2:

if value2 is not equal to valueR:

Erase key2

Set returnVar to false

else:

Result insert

return returnVar

}

//Reassign

Map::reassign(const Map& m, Map& result){

Variables to keep track of values

Assume this is already the value of the previous node

Variables for the current key-value pair

Clear the result map

while result is not empty:

retrieve the first key-value pair from result

remove that pair from result

for each subsequent node in 'm':

retrieve the key and value of the current node

Update previous value for the next iteration

set 'previousValue' to 'temporaryValue'

Handle the first node separately

insert 'firstKey' and 'previousValue' into result

}

* A list of test cases that would thoroughly test the functions. Be sure to indicate the purpose of the tests. For example, here's the beginning of a presentation in the form of code:

//Test Constructor

**void** testCon()

{

Map m;

//Check if size of map is 0

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

}

//Test Destructor

**void** testCopyCon()

{

//Check size if lhs = rhs size

Map m;

KeyType key1 = "alpha";

ValueType value1 = 1.0;

KeyType key2 = "beta";

ValueType value2 = 3.0;

KeyType key3 = "delta";

ValueType value3 = 2.0;

m.insert(key1, value1);

m.insert(key2, value2);

m.insert(key3, value3);

Map n(m);

assert(n.size() == m.size());

//Loop through every node and check if they are the same

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

m.get(i, key1, value1);

n.get(i, key2, value2);

assert(key1 == key2 && value1 == value2);

}

}

//Test Assignment Operator

**void** testAssignmentOp()

{

Map m;

Map n;

KeyType key1 = "alpha";

ValueType value1 = 1.0;

KeyType key2 = "delta";

ValueType value2 = 2.0;

KeyType key3 = "beta";

ValueType value3 = 3.0;

KeyType key4 = "gamma";

ValueType value4 = 4.0;

KeyType key5 = "zeta";

ValueType value5 = 5.0;

m.insert(key1, value1);

m.insert(key2, value2);

m.insert(key3, value3);

n.insert(key4, value4);

n.insert(key5, value5);

m = n;

//Check if size of m is equal to n

assert(m.size() == n.size());

assert(m.size() == 2);

//Loop through to see if m = n now

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

m.get(i, key1, value1);

n.get(i, key2, value2);

assert(key1 == key2 && value1 == value2);

}

}

//Test contains

**void** testContains() {

Map m;

KeyType key1 = "alpha";

ValueType value1 = 1.0;

KeyType key2 = "delta";

ValueType value2 = 2.0;

KeyType key3 = "beta";

ValueType value3 = 3.0;

KeyType key4 = "gamma";

m.insert(key1, value1);

m.insert(key2, value2);

m.insert(key3, value3);

//Contains

assert(m.contains(key1));

assert(m.contains(key2));

assert(m.contains(key3));

//Doesn't contain

assert(!m.contains(key4));

}

//Test insert

**void** testInsert() {

Map m;

KeyType key1 = "alpha";

ValueType value1 = 1.0;

ValueType valuex = 2.0;

KeyType key2 = "delta";

ValueType value2 = 2.0;

KeyType key3 = "beta";

ValueType value3 = 3.0;

KeyType key4 = "aaa";

ValueType value4 = 11.0;

//Same key, should not be inserting alpha more than once and should return false

assert(m.insert(key1, value1)); //Adding to null node

assert(!m.insert(key1, value1));

assert(!m.insert(key1, valuex));

//Should insert and return true

assert(m.insert(key2, value2)); //Adding to tail

assert(m.insert(key3, value3)); //Adding to middle

assert(m.insert(key4, value4)); //Adding to head

assert(m.size() == 4);

//Check if list is sorted correctly

m.get(0, key1, value1);

assert(key1 == "aaa" && value1 == 11);

m.get(1, key1, value1);

assert(key1 == "alpha" && value1 == 1);

m.get(2, key1, value1);

assert(key1 == "beta" && value1 == 3);

m.get(3, key1, value1);

assert(key1 == "delta" && value1 == 2);

}

//Test update

**void** testUpdate() {

Map m;

KeyType key1 = "alpha";

ValueType value1 = 1;

ValueType valuex = 2;

KeyType key2 = "beta";

ValueType value2 = 2;

m.insert(key1, value1);

m.insert(key2, value2);

assert(m.update(key1, value1));

//Same key, can update

assert(m.update(key1, value1)); //Same value

assert(m.update(key1, valuex)); //Different Value

//Different key, can't update, should return false

assert(!m.update(key2, value2));

//Check if value was updated

m.get(0, key1, value1);

assert(key1 == "alpha" && value1 == 2);

}

//Test insertOrUpdate

**void** testInsertOrUpdate() {

Map m;

KeyType key1 = "alpha";

ValueType value1;

m.insert(key1, 1);

//Same key, can update, not insert

assert(m.insertOrUpdate(key1, 1)); //Same value

assert(m.insertOrUpdate(key1, 2)); //Different Value

//Different key, can insert, should return true

KeyType key2 = "beta";

assert(m.insertOrUpdate(key2, 2));

//Check if value was updated

m.get(0, key1, value1);

assert(key1 == "alpha" && value1 == 2);

//Check if key was added

m.get(1, key1, value1);

assert(key1 == "beta" && value1 == 2);

}

//Test erase

**void** testErase(){

Map m;

KeyType key1 = "alpha";

ValueType value1 = 1.0;

KeyType key2 = "beta";

ValueType value2 = 2.0;

KeyType key3 = "delta";

ValueType value3 = 3.0;

//If empty

//Should be false because it's empty

assert(!m.erase(key1));

//If head if equal to key

m.insert(key1, value1);

m.get(key1, value1);

assert(m.erase(key1));

assert(!m.get(key1, value1));

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

//If tail is equal to key

m.insert(key2, value2);

m.insert(key3, value3);

assert(m.erase(key3));

assert(!m.get(key3, value3));

assert(m.get(key2, value2) && key2 == "beta");

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

//If key is in the middle

key1 = "alpha";

value1 = 1.0;

key2 = "delta";

value2 = 3;

m.insert(key1, value1);

m.insert(key2, value2);

key3 = "beta";

assert(m.erase(key3));

assert(!m.get(key3, value1));

assert(m.get(key1, value1) && key1 == "alpha");

assert(m.get(key2, value1) && key2 == "delta");

assert(m.size() == 2);

}

//Test get1

**void** testGet1(){

Map m;

KeyType key1 = "alpha";

ValueType value1 = 1.0;

KeyType key2 = "beta";

ValueType value2 = 0.0;

KeyType key3 = "delta";

ValueType value3 = 0.0;

KeyType key4 = "gamma";

ValueType value4 = 0.0;

m.insert(key1, value1);

m.insert(key2, value2);

m.insert(key3, value3);

m.insert(key4, value4);

assert(m.get(key1, value1) && key1 == "alpha");

assert(m.get(key2, value2) && key2 == "beta");

assert(m.get(key3, value3) && key3 == "delta");

assert(m.get(key4, value4) && key4 == "gamma");

}

//Test get2

**void** testGet2(){

Map m;

KeyType key1 = "alpha";

ValueType value1 = 1.0;

KeyType key2 = "beta";

ValueType value2 = 0.0;

KeyType key3 = "delta";

ValueType value3 = 0.0;

KeyType key4 = "gamma";

ValueType value4 = 0.0;

m.insert(key2, value2);

m.insert(key1, value1);

m.insert(key4, value4);

m.insert(key3, value3);

assert(m.get(0, key1, value1) && key1 == "alpha");

assert(m.get(1, key2, value2) && key2 == "beta");

assert(m.get(2, key3, value3) && key3 == "delta");

assert(m.get(3, key4, value4) && key4 == "gamma");

}

//Test swap

**void** testSwap(){

Map m;

KeyType key1 = "alpha";

ValueType value1 = 1.0;

KeyType key2 = "beta";

ValueType value2 = 0.0;

KeyType key3 = "delta";

ValueType value3 = 0.0;

KeyType key4 = "gamma";

ValueType value4 = 0.0;

m.insert(key2, value2);

m.insert(key1, value1);

m.insert(key4, value4);

m.insert(key3, value3);

Map n;

KeyType key5 = "chi";

ValueType value5 = 1.0;

KeyType key6 = "omega";

ValueType value6 = 0.0;

KeyType key7 = "zeta";

ValueType value7 = 0.0;

KeyType key8 = "theta";

ValueType value8 = 0.0;

n.insert(key6, value6);

n.insert(key5, value5);

n.insert(key8, value8);

n.insert(key7, value7);

m.swap(n);

assert(m.get(0, key1, value1) && key1 == "chi");

assert(m.get(1, key2, value2) && key2 == "omega");

assert(m.get(2, key3, value3) && key3 == "theta");

assert(m.get(3, key4, value4) && key4 == "zeta");

assert(n.get(0, key5, value5) && key5 == "alpha");

assert(n.get(1, key6, value6) && key6 == "beta");

assert(n.get(2, key7, value7) && key7 == "delta");

assert(n.get(3, key8, value8) && key8 == "gamma");

}

//Test merge

**void** testMerge(){

Map m1, m2, m3, m4, m5, m6, result;

KeyType key = "Lucy";

KeyType key1 = "Fred";

KeyType key2 = "Ethel";

KeyType key3 = "Ricky";

ValueType value = 789;

ValueType value1 = 123;

ValueType value2 = 456;

ValueType value3 = 321;

//Empty Case

assert(merge(m1, m2, result));

// Case 1: No common keys

m1.insert(key1, value1);

m1.insert(key2, value2);

m2.insert(key, value);

m2.insert(key3, value3);

assert(merge(m1, m2, result));

assert(result.size() == 4);

// Case 2: Common keys with the same values

m3.insert(key, value1);

m3.insert(key2, value2);

m4.insert(key, value1);

m4.insert(key3, value3);

assert(merge(m3, m4, result));

assert(result.contains(key));

assert(result.size() == 3);

// Case 3: Common keys with different values

m5.insert(key, value);

m5.insert(key2, value2);

m6.insert(key, value1);

m6.insert(key3, value3);

assert(!merge(m5, m6, result));

assert(!result.contains(key));

assert(result.size() == 2);

}

// Test for reassign function

**void** testReassign() {

Map m, result;

// Case 1: Empty map

reassign(m, result);

//assert(result.size() == 0);

// Case 2: Single pair in the map

m.insert("Fred", 123);

reassign(m, result);

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

assert(mapsEqual(result, m)); // result should be equal to m

// Case 3: Multiple pairs in the map

m.insert("Ethel", 456);

m.insert("Lucy", 789);

m.insert("Ricky", 321);

reassign(m, result);

assert(result.size() == 4);

// result should contain pairs from m, with values reassigned

// make sure each value in result corresponds to a different key in m

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

KeyType key1, key2;

ValueType value1, value2;

m.get(i, key1, value1);

result.get(i, key2, value2);

assert(value1 != value2);

}

// Add more test cases as needed

}

**void** test()

{

Map m;

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

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

assert(m.size() == 2);

ValueType v = 42;

assert(!m.get("Lucy", v) && v == 42);

assert(m.get("Fred", v) && v == 123);

v = 42;

KeyType x = "Lucy";

assert(m.get(0, x, v) && x == "Ethel" && v == 456);

KeyType x2 = "Ricky";

assert(m.get(1, x2, v) && x2 == "Fred" && v == 123);

}

**int** main() {

// Run test functions

testCon();

testCopyCon();

testAssignmentOp();

testContains();

testInsert();

testUpdate();

testInsertOrUpdate();

testErase();

testGet1();

testGet2();

testSwap();

testMerge();

testReassign();

test();

// If everything passes, print success

std::cout << "All tests passed successfully!" << std::endl;

**return** 0;

}