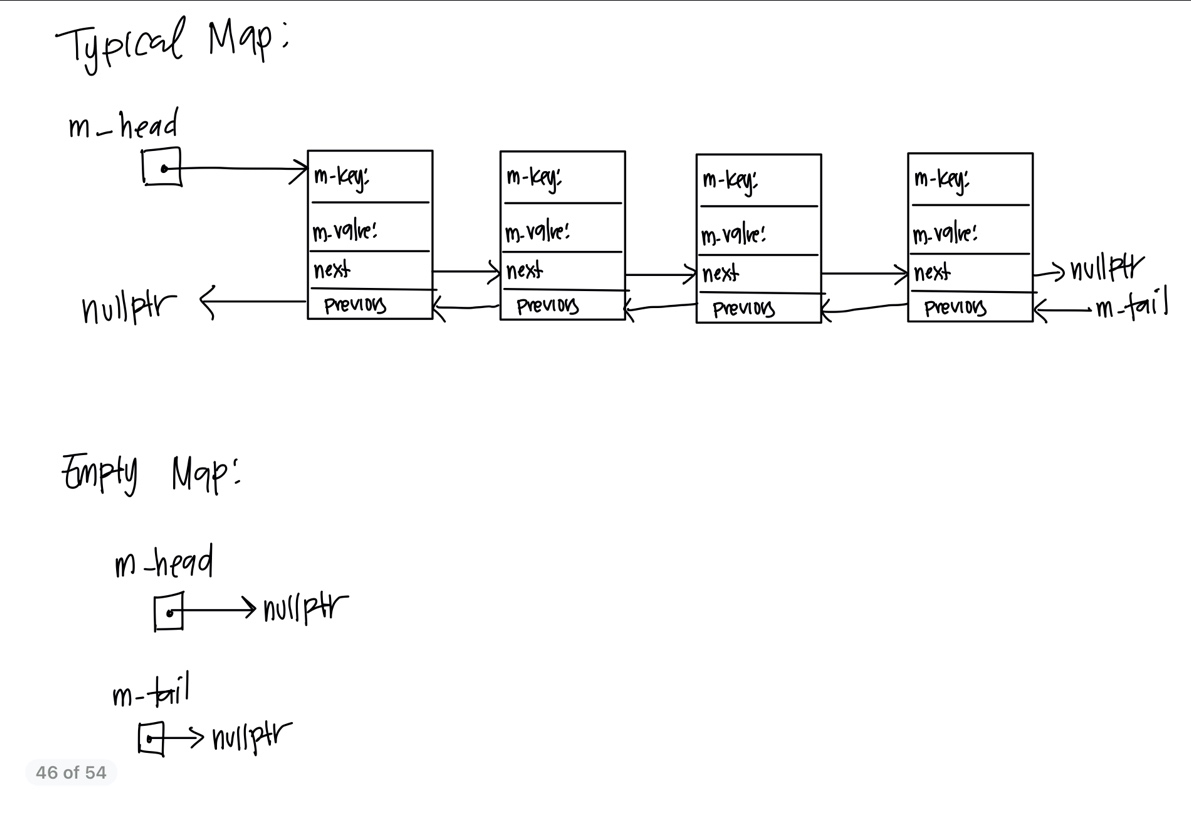
**UID: 005738216**

**Description:**

My doubly-linked list utilises two pointers, one pointing to the head of the list and one pointing to the tail of the list. To create my Map, I created a new struct, Node. Each node has a KeyType, a ValueType, a next pointer, and a previous pointer. The purpose of the next pointer is to point to the next Node in the list. The very last Node has a next pointer pointing to null. Similarly, the previous pointer points to the previous Node in the list. The first Node has a previous pointer pointing to null. If the list is empty, the head and tail pointers both point to null. The Nodes are organised in a sorted order with the insert function inserting each Node in the respective position rather than at the end of the list. I also created a variable, m\_size, to keep track of how many Nodes are in the Map. Below is an illustration of a typical Map and an empty one:



Pseudocode:

Map():

set head and tail pointers to null

set size to 0

~Map():

create temporary pointer to point to object to delete

create temporary pointer to point to next object

repeatedly until end of Map:

delete current object

use pointer to iterate to next object

Map(const Map& m):

set size to m’s size

if size is 0

set head and tail pointers to null

else

create first Node of Map

create temporary pointer to iterate m

Repeatedly until end of m:

create new Node and assign key and value

add newly created Node to Map

set tail pointer to last element of Map

operator=(const Map& rhs)

swap *this* Map with a copy of rhs

return pointer to *this*

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

if key already exists

return false

create new Node using key and value

if Map is empty

insert Node as first element

else if Node has smaller key than first Node

insert Node at beginning

set head pointer to point to Node

else if Node has larger key than last Node

insert Node as end of Map

set tail pointer to point to Node

else

iterate until next Node has larger key value than Node

insert Node at position reached during iteration

increment size

return true

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

iterate through Map end of Map or until Node with key found

if end of Map reached

return false

else

set value of Node with key to value

return true

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

update Map with key and value or insert key and value

erase(const KeyType& key):

if size is 1

delete object

set head and tail pointers to null

else if key is at head

set head pointer to Node following head

delete object

else if key is at tail

set tail pointer to Node before tail

delete object

else

iterate till end of Map or Node with key found

return false if key not found

update pointers of Nodes before and after Node with key

delete object

decrement size

return true

contains(const KeyType& key):

iterate until end of Map or Node with key found

return whether key is found

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

iterate until end of Map or Node with key found

return false if key not found

set value to value held by Node with key

return true

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

return false if i not within range of size

iterate through Map i times

set key to key stored by ith Node

set value to value stored by ith Node

return true

swap(Map& other):

swap sizes of Maps

swap head pointers

swap tail pointers

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

create temporary map

create variable that indicates whether all same keys have same values

Repeatedly size of the smaller map times:

set variable to false if both maps have same key with different value

insert Node into temporary map if Node is in both maps

Repeatedly i times where i is size of larger map – size of smaller map

Insert Node into temporary map if not contained in smaller map

swap temporary map with result

return result

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

create a copy of m as temporary Map

repeatedly size of copy – 1 times

call 3 parameter get function with current iteration #

call 3 parameter get function with next iteration #

update temp with key of first call to get and value of second call to get

update temp with key of second call to get and value of first call to get

swap result with temporary Map

Test Cases:

//default constructor

Map m1;

//test size and test empty

assert(m1.size() == 0 && m1.empty());

//test erase on empty Map

assert(!m1.erase(“Hi”));

//test inserting elements into Map

m1.insert("Banana", 2.2);

m1.insert("Daikon", 4.4);

m1.insert("Apple", 1.1);

m1.insert("Carrot", 3.3);

//test size

assert(m1.size() == 4);

//test contains on all inserted elements

assert(m1.contains("Apple") && m1.contains("Banana") && m1.contains("Carrot") && m1.contains("Daikon"));

//test contains on nonexistent element

assert(!m1.contains("Elderberry"));

//test first get function and make sure correct value is assigned to v

ValueType v = 111.1;

assert(m1.get("Apple", v) && v == 1.1);

assert(m1.get("Banana", v) && v == 2.2);

assert(m1.get("Carrot", v) && v == 3.3);

assert(m1.get("Daikon", v) && v == 4.4);

//test insertOrUpdate properly inserted key-value pair into Map

m1.insertOrUpdate("Elderberry", 5.5);

assert(m1.contains("Elderberry"));

//test insertOrUpdate properly updates value of existent element

m1.insertOrUpdate("Daikon", 44.4);

assert(m1.get("Daikon", v) && v == 44.4);

//test second get function

KeyType k = "";

assert(m1.get(0, k, v) && k == "Apple" && v == 1.1);

assert(m1.get(1, k, v) && k == "Banana" && v == 2.2);

assert(m1.get(2, k, v) && k == "Carrot" && v == 3.3);

assert(m1.get(3, k, v) && k == "Daikon" && v == 44.4);

assert(m1.get(4, k, v) && k == "Elderberry" && v == 5.5);

//test that second get function returns false when i >= # elements in Map

assert(!m1.get(5, k, v));

//test that second get function returns false when i < 0

assert(!m1.get(-3, k, v));

Map m2;

m2.insert("Anteater", 111.11);

m2.insert("coyote", 555.55);

m2.insert("Beluga", 222.22);

m2.insert("dolphin", 666.66);

assert(m2.get(0, k, v) && k == "Anteater");

m2.insert("Elephant", 333.33);

m2.insert("Fox", 444.44);

assert(m2.contains("Fox"));

assert(!m2.empty() && m2.size() == 6);

assert(m2.update("Fox", 44.4));

assert(m2.get("Fox", v) && v == 44.4);

//test swap

m1.swap(m2);

//check that sizes have been swapped

assert(m1.size() == 6 && m2.size() == 5);

//check that m1 now contains all elements originally stored in m2

assert(m1.get(0, k, v) && k == "Anteater" && v == 111.11);

assert(m1.get(1, k, v) && k == "Beluga" && v == 222.22);

assert(m1.get(2, k, v) && k == "Elephant" && v == 333.33);

assert(m1.get(3, k, v) && k == "Fox" && v == 44.4);

assert(m1.get(4, k, v) && k == "coyote" && v == 555.55);

assert(m1.get(5, k, v) && k == "dolphin" && v == 666.66);

//check that m2 now contains all elements originally stored in m1

assert(m2.get(0, k, v) && k == "Apple" && v == 1.1);

assert(m2.get(1, k, v) && k == "Banana" && v == 2.2);

assert(m2.get(2, k, v) && k == "Carrot" && v == 3.3);

assert(m2.get(3, k, v) && k == "Daikon" && v == 44.4);

assert(m2.get(4, k, v) && k == "Elderberry" && v == 5.5);

//test erase

assert(m1.erase("Anteater") && m1.erase("coyote") && m2.erase("Elderberry"));

//check that erased key-value pairs no longer exist

assert(!m1.contains("Anteater") && !m1.contains("coyote") && !m2.contains("Elderberry"));

//check that size of Maps has been changed accordingly

assert(m1.size() == 4 && m2.size() == 4);

//check that first element of Map has been changed

assert(m1.get(0, k, v) && k == "Beluga" && v == 222.22);

//erase all elements in m2

assert(m2.erase("Apple") && m2.erase("Banana") && m2.erase("Carrot") && m2.erase("Daikon"));

//check that m2 is now empty

assert(m2.empty());

//test merge function

Map m3;

Map m4;

Map m5;

m3.insert("A", 1.1);

m3.insert("B", 2.2);

m3.insert("C", 3.3);

m3.insert("b", 4.4);

//check that inserting a key that already exists fails

assert(!m3.insert("A", 8.8));

assert(m3.size() == 4);

m4.insert("A", 1.1);

m4.insert("D", 4.4);

m4.insert("e", 5.5);

//test merge

assert(merge(m3, m4, m5));

//check that m5 now contains merged values

assert(m5.contains("A") && m5.contains("B") && m5.contains("C") && m5.contains("b") && m5.contains("D") && m5.contains("e"));

assert(m5.size() == 6);

//test merge where both parameters refer to the same map

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

assert(m3.contains("A") && m3.contains("B") && m3.contains("C") && m3.contains("b") && m3.contains("D") && m3.contains("e"));

assert(m3.size() == 6);

m4.update("A", 101.1);

//check that merge returns false when same key has different value in two different maps

assert(!merge(m3, m4, m5));

assert(m5.size() == 5);

//test merge when result is not an empty Map

Map m6;

Map m7;

m6.insert("A", 4.4);

m6.insert("G", 8.8);

m7.insert("B", 2.2);

m7.insert("D", 4.4);

m7.insert("E", 5.5);

m7.insert("G", 9.9);

assert(!merge(m6, m7, m5) && !m5.contains("G"));

//test reassign function with even number of elements

Map m8;

Map m9;

m8.insert("A", 123);

m8.insert("B", 456);

m8.insert("C", 789);

m8.insert("D", 321);

reassign(m8, m9);

for(int i = 0; i < 4; i++)

{

ValueType v;

KeyType k;

m9.get(i, k, v);

cout << k << " " << v << '\t'; //check that values have been properly reassigned

}

cout << endl;

//test reassign function with odd number of elements

Map m10;

Map m11;

m10.insert("A", 123);

m10.insert("B", 456);

m10.insert("C", 456);

reassign(m10, m11);

for(int i = 0; i < 3; i++)

{

ValueType v;

KeyType k;

m11.get(i, k, v);

cout << k << " " << v << '\t'; //check that values have been properly reassigned

}

cout << endl;

//test reassign function where both parameters refer to same Map

reassign(m8, m8);

for(int i = 0; i < 4; i++)

{

ValueType v;

KeyType k;

m8.get(i, k, v);

cout << k << " " << v << '\t'; //check that values have been properly reassigned

}

cout << endl;

//test reassign function where second parameter is not an empty Map

reassign(m8, m10);

for(int i = 0; i < 4; i++)

{

ValueType v;

KeyType k;

m10.get(i, k, v);

cout < <k << " " << v << '\t';

}

cout << endl;

//test copy constructor

Map m12 = m7;

assert(m12.get(0, k, v) && k == "B" && v == 2.2);

assert(m12.get(1, k, v) && k == "D" && v == 4.4);

assert(m12.get(2, k, v) && k == "E" && v == 5.5);

assert(m12.get(3, k, v) && k == "G" && v == 9.9);

//check that m7 and m12 aren't pointing to the same Map

m7.insert("F", 6.6);

assert(!m12.contains("F"));

//test assignment operator

//assign empty Map to empty Map

Map m13;

Map m14;

m14 = m13;

//check that both m13 and m14 have size 0 and are empty

assert(m13.size() ==0 && m14.size() == 0);

assert(m13.empty() && m14.empty());

//assign empty Map to nonempty Map

m14.insert("A", 1.1);

m14.insert("B", 2.2);

m14.insert("C", 3.3);

m14 = m13;

//check that m14, nonempty before, is now empty and has size 0

assert(m14.size() == 0 && m14.empty());

//assign nonempty Map to empty Map;

m13.insert("Z", 10.10);

m13.insert("Y", 9.9);

m13.insert("X", 8.8);

//check that m14, empty before, is not not empty and has size 3

m14 = m13;

assert(!m14.empty() && m14.size() == 3);

//check that m14 contains all the correct values

assert(m14.get(0, k, v) && k == "X" && v == 8.8);

assert(m14.get(1, k, v) && k == "Y" && v == 9.9);

assert(m14.get(2, k, v) && k == "Z" && v == 10.10);

//assign nonempty map to nonempty map (both maps are same size)

Map m15;

m15.insert("A", 23.3);

m15.insert("B", 34.3);

m15.insert("C", 52.9);

m15 = m13;

assert(m15.size() == 3);

//check that m15 now has all values that are in m13

assert(m15.get(0, k, v) && k == "X" && v == 8.8);

assert(m15.get(1, k, v) && k == "Y" && v == 9.9);

assert(m15.get(2, k, v) && k == "Z" && v == 10.10);

//assign nonempty map to nonempty map (Maps are different sizes)

Map m16;

m16.insert("E", 234.5);

m16.insert("R", 34.5);

m16.insert("C", 745.2);

m16.insert("X", 81.5);

m13 = m16;

//check that m13 now has size 4 instead of size 3

assert(m13.size() == 4);

//check that m13 has all the correct values

assert(m13.get(0, k, v) && k == "C" && v == 745.2);

assert(m13.get(1, k, v) && k == "E" && v == 234.5);

assert(m13.get(2, k, v) && k == "R" && v == 34.5);

assert(m13.get(3, k, v) && k == "X" && v == 81.5);

**Repeat same test cases for when KeyType = int and ValueType = std::string**

Map m1;

assert(m1.size() == 0 && m1.empty());

m1.insert(2, "BB");

m1.insert(4, "DD");

m1.insert(1, "AA");

m1.insert(3, "CC");

assert(m1.size() == 4);

assert(m1.contains(1) && m1.contains(2) && m1.contains(3) && m1.contains(4)); //test contains on all inserted elements

assert(!m1.contains(5));

ValueType v = "";

assert(m1.get(1, v) && v == "AA");

assert(m1.get(2, v) && v == "BB");

assert(m1.get(3, v) && v == "CC");

assert(m1.get(4, v) && v == "DD");

m1.insertOrUpdate(5, "EE");

assert(m1.contains(5));

m1.insertOrUpdate(4, "D");

assert(m1.get(4, v) && v == "D");

KeyType k = -1;

assert(m1.get(0, k, v) && k == 1 && v == "AA");

assert(m1.get(1, k, v) && k == 2 && v == "BB");

assert(m1.get(2, k, v) && k == 3 && v == "CC");

assert(m1.get(3, k, v) && k == 4 && v == "D");

assert(m1.get(4, k, v) && k == 5 && v == "EE");

assert(!m1.get(5, k, v));

assert(!m1.get(-3, k, v));

Map m2;

m2.insert(10, "Z");

m2.insert(11, "Y");

m2.insert(12, "X");

m2.insert(13, "W");

assert(m2.get(0, k, v) && k == 10);

m2.insert(14, "V");

m2.insert(15, "U");

assert(!m2.empty() && m2.size() == 6);

assert(m2.update(15, "R"));

assert(m2.get(15, v) && v == "R");

m1.swap(m2);//test swap

assert(m1.size() == 6 && m2.size() == 5);

assert(m1.get(0, k, v) && k == 10 && v == "Z");

assert(m1.get(1, k, v) && k == 11 && v == "Y");

assert(m1.get(2, k, v) && k == 12 && v == "X");

assert(m1.get(3, k, v) && k == 13 && v == "W");

assert(m1.get(4, k, v) && k == 14 && v == "V");

assert(m1.get(5, k, v) && k == 15 && v == "R");

assert(m2.get(0, k, v) && k == 1 && v == "AA");

assert(m2.get(1, k, v) && k == 2 && v == "BB");

assert(m2.get(2, k, v) && k == 3 && v == "CC");

assert(m2.get(3, k, v) && k == 4 && v == "D");

assert(m2.get(4, k, v) && k == 5 && v == "EE");

assert(m1.erase(10) && m1.erase(13) && m2.erase(4));

assert(!m1.contains(10) && !m1.contains(13) && !m2.contains(4));

assert(m1.size() == 4 && m2.size() == 4);

assert(m1.get(0, k, v) && k ==11 && v == "Y"); //check that first element of Map has been changed

assert(m2.erase(5) && m2.erase(1) && m2.erase(2) && m2.erase(3));

assert(m2.empty());

Map m3;

Map m4;

Map m5;

m3.insert(1, "A");

m3.insert(2, "B");

m3.insert(3, "C");

m3.insert(4, "D");

assert(!m3.insert(1, "E"));

assert(m3.size() == 4);

m4.insert(1, "A");

m4.insert(6, "F");

m4.insert(7, "G");

assert(merge(m3, m4, m5));

assert(m5.contains(1) && m5.contains(2) && m5.contains(3) && m5.contains(4) && m5.contains(7) && m5.contains(6));

assert(m5.size() == 6);

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

assert(m3.contains(1) && m3.contains(2) && m3.contains(3) && m3.contains(4) && m3.contains(7) && m3.contains(6));

assert(m3.size() == 6);

m4.update(1, "Z");

assert(!merge(m3, m4, m5)); value in two different maps

assert(m5.size() == 5);

Map m6;

Map m7;

m6.insert(1, "A");

m6.insert(7, "G");

m7.insert(2, "B");

m7.insert(3, "C");

m7.insert(4, "D");

m7.insert(7, "E");

assert(!merge(m6, m7, m5) && !m5.contains(7));

Map m8;

Map m9;

m8.insert(1, "A");

m8.insert(2, "B");

m8.insert(3, "C");

m8.insert(4, "D");

reassign(m8, m9);

for(int i = 0; i < 4; i++)

{

ValueType v;

KeyType k;

m9.get(i, k, v);

cout << k << " " << v << '\t';

}

cout << endl;

Map m10;

Map m11;

m10.insert(1,"A");

m10.insert(2, "B");

m10.insert(3, "C");

reassign(m10, m11);

for(int i = 0; i < 3; i++)

{

ValueType v;

KeyType k;

m11.get(i, k, v);

cout << k << " " << v << '\t';

}

cout << endl;

reassign(m8, m8);

for(int i = 0; i < 4; i++)

{

ValueType v;

KeyType k;

m8.get(i, k, v);

cout << k << " " << v << '\t';

}

cout << endl;

reassign(m8, m10);

for(int i = 0; i < 4; i++)

{

ValueType v;

KeyType k;

m10.get(i, k, v);

cout << k << " " << v << '\t';

}

cout << endl;

}

Map m12 = m7;

assert(m12.get(0, k, v) && k == 2 && v == "B");

assert(m12.get(1, k, v) && k == 3 && v == "C");

assert(m12.get(2, k, v) && k == 4 && v == "D");

assert(m12.get(3, k, v) && k == 7 && v == "E");

m7.insert(12, "test");

assert(!m12.contains(12));

Map m13;

Map m14;

m14 = m13;

assert(m13.size() ==0 && m14.size() ==0);

assert(m13.empty() && m14.empty());

m14.insert(1, "A");

m14.insert(2, "B");

m14.insert(3, "C");

m14 = m13;

assert(m14.size() == 0 && m14.empty());

m13.insert(10, "X");

m13.insert(9, "Y");

m13.insert(8, "Z");

m14 = m13;

assert(!m14.empty() && m14.size() == 3);

assert(m14.get(0, k, v) && k == 8 && v == "Z");

assert(m14.get(1, k, v) && k == 9 && v == "Y");

assert(m14.get(2, k, v) && k == 10 && v == "X");

Map m15;

m15.insert(4, "G");

m15.insert(5, "D");

m15.insert(6, "E");

m15 = m13;

assert(m15.size() == 3);

assert(m15.get(0, k, v) && k == 8 && v == "Z");

assert(m15.get(1, k, v) && k == 9 && v == "Y");

assert(m15.get(2, k, v) && k == 10 && v == "X");

Map m16;

m16.insert(11, "B");

m16.insert(12, "V");

m16.insert(13, "C");

m16.insert(14, "X");

m13 = m16;

assert(m13.size() == 4);

assert(m13.get(0, k, v) && k == 11 && v == "B");

assert(m13.get(1, k, v) && k == 12 && v == "V");

assert(m13.get(2, k, v) && k == 13 && v == "C");

assert(m13.get(3, k, v) && k == 14 && v == "X");