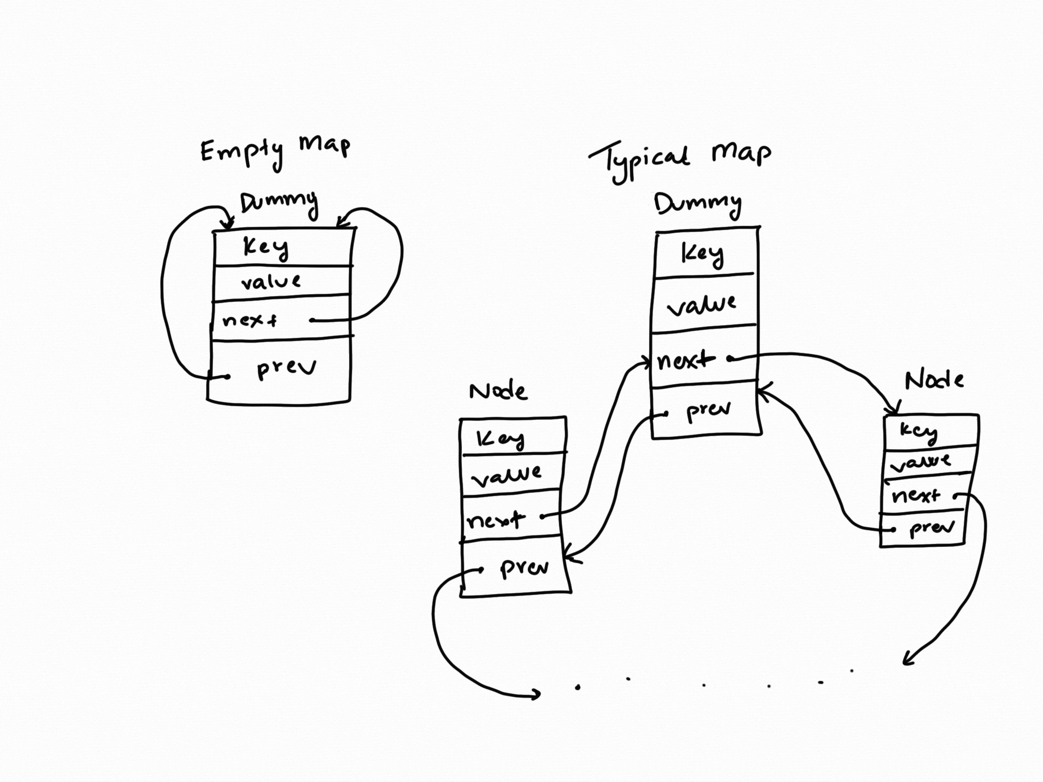
Structure of the linked-list – it has a dummy node with next and prev pointers that point to the next and previous nodes of the list respectively, the list is circular in nature – the next pointer of the last node points to the dummy node, the prev pointer of the first node points to the dummy node, an empty list’s dummy node’s next and prev pointer points back to itself, the nodes are arranged in ascending order of keys



Pseudocode of non-trivial algorithms

Erase(key) {

Find the pointer to the node with the given key

Alter the pointers of the previous and next element to point to each other

Delete the pointer found

}

Insert(key, value) {

If list already contains key 🡪 exit with false

If list is empty

Create a new Node with key/value pair

Make the dummy pointers point to it

Make its pointers point to the dummy

Exit with true

Iterate through the list and find the first element whose key is greater than the given key

Create a new Node with key/value pair

Insert the new Node before the pointer found

Exit with true

If no such key found

Insert new Node with key/value pair at the end of the list

Exit with true

}

Get(i, key, value) {

Find the (i+1)th node in the list

Return its key/value with true

}

Swap(other) {

Swap the pointers of the dummies of the two maps

Swap the pointers of the first and last elements of the two maps that are supposed to point to the dummies

}

Merge(m1, m2, result) {

Erase all elements of the result list

Create a flag variable

Iterate through m1 using the get method

If key received is present in m2

If the values of m1 and m2 of the key do not match

Drop the flag and do not add the key/value pair to result

Add the key value pair to result

Iterate through m2

If key received is not already present in result and not present in m1

Add the key/value pair to result

}

Reassign(m, result) {

Erase all elements of the result list

Store the value of the first node in m

Iterate through m

Add the key of the ith and the value of the (i+1)th node of m to as key/value pair result

Add the last key and the stored first value of m as key/value pair to result

}

List of all test cases (maps of strings to doubles)

**void** testForSizeAndEmpty() {

Map m; // maps strings to doubles

assert(m.empty());

ValueType v = -1234.5;

assert( !m.get("abc", v) && v == -1234.5); // v unchanged by get failure

m.insert("xyz", 9876.5);

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

KeyType k = "hello";

assert(m.get(0, k, v) && k == "xyz" && v == 9876.5);

cout << "Passed all tests" << endl;

}

**void** testForInsert() {

Map mm;

mm.insert("Little Ricky", 3.206);

mm.insert("Ethel", 3.538);

mm.insert("Ricky", 3.350);

mm.insert("Lucy", 2.956);

mm.insert("Ethel", 3.539);

mm.insert("Fred", 2.956);

mm.insert("Lucy", 2.956);

assert(mm.size() == 5); // duplicate "Ethel" and "Lucy" were not added

string x;

**double** y;

mm.get(0, x, y);

assert(x == "Ethel"); // "Ethel" is greater than exactly 0 items in mm

mm.get(4, x, y);

assert(x == "Ricky"); // "Ricky" is greater than exactly 4 items in mm

mm.get(2, x, y);

assert(x == "Little Ricky"); // "Little Ricky" is greater than exactly 2 items in mm

assert(y == 3.206); // the value corresponding to the key "Little Ricky"

cout << "Passed all tests" << endl;

}

**void** testForSwap() {

Map m1;

m1.insert("Fred", 2.956);

assert(m1.size() == 1);

Map m2;

m2.insert("Ethel", 3.538);

m2.insert("Lucy", 2.956);

m1.swap(m2);

assert(m1.size() == 2 && m1.contains("Ethel") && m1.contains("Lucy") &&

m2.size() == 1 && m2.contains("Fred"));

cout << "Passed all tests" << endl;

}

**void** emptyStringTest() {

Map gpas;

gpas.insert("Fred", 2.956);

assert(!gpas.contains(""));

gpas.insert("Ethel", 3.538);

gpas.insert("", 4.000);

gpas.insert("Lucy", 2.956);

assert(gpas.contains(""));

gpas.erase("Fred");

assert(gpas.size() == 3 && gpas.contains("Lucy") && gpas.contains("Ethel") &&

gpas.contains(""));

string k;

**double** v;

assert(gpas.get(1, k, v) && k == "Ethel");

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

cout << "Passed all tests" << endl;

}

**void** test()

{

Map m;

assert(m.insert("Fred", 2.956));

assert(m.insert("Ethel", 3.538));

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

ValueType v = 42;

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

assert(m.get("Fred", v) && v == 2.956);

v = 42;

KeyType x = "Lucy";

assert(m.get(0, x, v) &&

((x == "Fred" && v == 2.956) || (x == "Ethel" && v == 3.538)));

KeyType x2 = "Ricky";

assert(m.get(1, x2, v) &&

((x2 == "Fred" && v == 2.956) || (x2 == "Ethel" && v == 3.538)) &&

x != x2);

cout << "Passed all tests" << endl;

}

**void** basicInsert() {

Map m;

m.insert("Hi", 10);

m.insert("Me", 5);

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

cout << "Passed all tests" << endl;

}

**void** testContains() {

Map m;

m.insert("Hi", 10);

assert(m.contains("Hi") == **true**);

assert(m.contains("Devansh") == **false**);

cout << "Passed all tests" << endl;

}

**void** testEmpty() {

Map m;

assert(m.empty() == **true**);

m.insert("Hi", 10);

assert(m.empty() == **false**);

cout << "Passed all tests" << endl;

}

**void** testErase() {

Map m;

m.insert("Hi", 10);

assert(m.size() == 1 && m.contains("Hi"));

m.erase("Hi");

assert(m.empty());

cout << "Passed all tests" << endl;

}

**void** testMerge() {

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;

merge(m1, m2, result);

assert(result.contains("Fred") && result.contains("Ricky") && result.contains("Lucy") && result.contains("Ethel") && result.size() == 4);

result.insert("Little Ricky", 456);

merge(m1, m2, result);

assert(result.contains("Fred") && result.contains("Ricky") && result.contains("Lucy") && result.contains("Ethel") && result.size() == 4);

m2.update("Lucy", 654);

merge(m1, m2, result);

assert(result.contains("Fred") && result.contains("Ricky") && !result.contains("Lucy") && result.contains("Ethel") && result.size() == 3);

cout << "Passed all tests" << endl;

}

**void** testReassign() {

Map m;

Map result;

m.insert("Fred", 123);

m.insert("Ethel", 456);

m.insert("Lucy", 789);

m.insert("Ricky", 321);

reassign(m, result);

ValueType v1; ValueType v2; ValueType v3; ValueType v4;

result.get("Fred", v1);

result.get("Ethel", v2);

result.get("Lucy", v3);

result.get("Ricky", v4);

assert(result.contains("Fred") && v1 != 123 && result.contains("Ethel") && v2 != 456 && result.contains("Lucy") && v3 != 789 && result.contains("Ricky") && v4 != 321 && result.size() == 4);

result.insert("Little Ricky", 234);

reassign(m, result);

assert(result.contains("Fred") && v1 != 123 && result.contains("Ethel") && v2 != 456 && result.contains("Lucy") && v3 != 789 && result.contains("Ricky") && v4 != 321 && result.size() == 4);

cout << "Passed all tests" << endl;

}

**void** testOperator() {

Map m1;

m1.insert("Fred", 10);

Map m2;

m2.insert("Ethel", 10);

m2.insert("Ricky", 20);

m1 = m2;

assert(m1.contains("Ethel") && m1.contains("Ricky") && m1.size() == 2);

cout << "Passed all tests" << endl;

}