<https://leetcode.com/problems/design-hashmap/>

Design a HashMap without using any built-in hash table libraries.

Implement the MyHashMap class:

* MyHashMap() initializes the object with an empty map.
* void put(int key, int value) inserts a (key, value) pair into the HashMap. If the key already exists in the map, update the corresponding value.
* int get(int key) returns the value to which the specified key is mapped, or -1 if this map contains no mapping for the key.
* void remove(key) removes the key and its corresponding value if the map contains the mapping for the key

**Example 1:**

Input

["MyHashMap", "put", "put", "get", "get", "put", "get", "remove", "get"]

[[], [1, 1], [2, 2], [1], [3], [2, 1], [2], [2], [2]]

Output

[null, null, null, 1, -1, null, 1, null, -1]

Explanation

MyHashMap myHashMap = new MyHashMap();

myHashMap.put(1, 1); // The map is now [[1,1]]

myHashMap.put(2, 2); // The map is now [[1,1], [2,2]]

myHashMap.get(1); // return 1, The map is now [[1,1], [2,2]]

myHashMap.get(3); // return -1 (i.e., not found), The map is now [[1,1], [2,2]]

myHashMap.put(2, 1); // The map is now [[1,1], [2,1]] (i.e., update the existing value)

myHashMap.get(2); // return 1, The map is now [[1,1], [2,1]]

myHashMap.remove(2); // remove the mapping for 2, The map is now [[1,1]]

myHashMap.get(2); // return -1 (i.e., not found), The map is now [[1,1]]

**Constraints:**

* 0 <= key, value <= 106
* At most 104 calls will be made to put, get, and remove.

**Attempt 1: 2023-06-08**

**Solution 1:  Traditional Hash-Table Implementation - Using Array of LinkedList (30 min)**

class ListNode {

int key;

int val;

ListNode next;

public ListNode(int key, int val, ListNode next) {

this.key = key;

this.val = val;

this.next = next;

}

}

class MyHashMap {

ListNode[] list;

static final int size = 19997;

static final int multiplier = 12582917;

public MyHashMap() {

list = new ListNode[size];

}

// Simple multiplicative hashing function utilizing

// a large prime multiplier and then modulo the result

// to the desired size (also a prime) of our hashmap array

private int hash(int key) {

// return Integer.hashCode(key) % size;

return (int)((long) key \* multiplier % size);

}

public void put(int key, int value) {

remove(key);

int index = hash(key);

ListNode newNode = new ListNode(key, value, list[index]);

list[index] = newNode;

}

public int get(int key) {

int index = hash(key);

ListNode node = list[index];

while(node != null) {

if(node.key == key) {

return node.val;

}

node = node.next;

}

return -1;

}

public void remove(int key) {

int index = hash(key);

ListNode node = list[index];

if(node == null) {

return;

}

// For the 1st node's key directly equal to given key,

// we have to remove the 1st node from the node list

// stored on array bucket at index

// e.g

// [index] = (1st node) -> (2nd node) ... -> (xth node)

// => 1st node to remove

// the wrong way is node = node.next; only change the pointer

// representation for 'list[index]' as 'node' from 1st node

// to 2nd node, 'list[index]' itself doens't change, still

// point to original 1st node

if(node.key == key) {

// node = node.next; --> wrong way !

list[index] = node.next;

} else {

while(node.next != null) {

if(node.next.key == key) {

node.next = node.next.next;

return;

}

node = node.next;

}

}

}

}

/\*\*

\* Your MyHashMap object will be instantiated and called as such:

\* MyHashMap obj = new MyHashMap();

\* obj.put(key,value);

\* int param\_2 = obj.get(key);

\* obj.remove(key);

\*/

**Refer to**

<https://leetcode.com/problems/design-hashmap/solutions/227081/java-solutions/>

**Solution 1: Traditional Hash-Table Implementation - Using Array of LinkedList - Accepted in 75 ms**

* The general implementation of HashMap uses bucket which is basically a **chain of linked lists** and each node containing <key, value> pair.
* So if we have duplicate nodes, that doesn't matter - it will still replicate each key with it's value in linked list node.
* When we insert the pair (10, 20) and then (10, 30), there is technically no collision involved. We are just replacing the old value with the new value for a given key 10, since in both cases, 10 is equal to 10 and also the hash code for 10 is always 10.
* Collision happens when multiple keys hash to the same bucket. In that case, we need to make sure that we can distinguish between those keys. **Chaining collision resolution** is one of those techniques which is used for this.
* Just for the information. In JDK 8, HashMap has been tweaked so that if keys can be compared for ordering, then any densely-populated bucket is implemented as a tree, so that even if there are lots of entries with the same hash-code, the complexity is O(log n).

class MyHashMap

{

ListNode[] nodes = new ListNode[10000];

public int get(int key)

{

int index = getIndex(key);

ListNode prev = findElement(index, key);

return prev.next == null ? -1 : prev.next.val;

}

public void put(int key, int value)

{

int index = getIndex(key);

ListNode prev = findElement(index, key);

if (prev.next == null)

prev.next = new ListNode(key, value);

else

prev.next.val = value;

}

public void remove(int key)

{

int index = getIndex(key);

ListNode prev = findElement(index, key);

if(prev.next != null)

prev.next = prev.next.next;

}

private int getIndex(int key)

{

return Integer.hashCode(key) % nodes.length;

}

private ListNode findElement(int index, int key)

{

if(nodes[index] == null)

return nodes[index] = new ListNode(-1, -1);

ListNode prev = nodes[index];

while(prev.next != null && prev.next.key != key)

{

prev = prev.next;

}

return prev;

}

private static class ListNode

{

int key, val;

ListNode next;

ListNode(int key, int val)

{

this.key = key;

this.val = val;

}

}

}

Time complexity: O(1) average and O(n) worst case - for all get(), put() and remove() methods.

Space complexity: O(n) - where n is the number of entries in HashMap

**Solution 2: Using Large Sized Array - Accepted in 100 ms**

class MyHashMap

{

int[] map;

public MyHashMap()

{

map = new int[1000001];

Arrays.fill(map,-1);

}

public int get(int key)

{

return map[key];

}

public void put(int key, int value)

{

map[key] = value;

}

public void remove(int key)

{

map[key] = -1;

}

}

For 2nd solution, Array.Fill() can be avoided

/\*\* value will always be non-negative. \*/

public void Put(int key, int value) {

array[key] = value +1; // making value in range 1 - 10000001

}

/\*\* Returns the value to which the specified key is mapped, or -1 if this map contains no mapping for the key \*/

public int Get(int key) {

return array[key] -1;

}

/\*\* Removes the mapping of the specified value key if this map contains a mapping for the key \*/

public void Remove(int key) {

array[key] = 0;

}

Time complexity: O(1) - for all get(), put() and remove() methods.

Space complexity: O(n) - where n is the maximum possible value for the key.

**Refer to**

<https://leetcode.com/problems/design-hashmap/solutions/1097755/js-python-java-c-updated-hash-array-solutions-w-explanation/>

#### ***Idea:***

The "easy" solution for this problem is simply to create an **array** large enough to accommodate the entire range of **keys**. This would seem to be the intended first solution, since the range of allowed keys is **non-negative** and constrained to **10^6**, which is not unmanageable.

(*Note: The* ***Code*** *section has examples of each language's Array approach down below. The remainder of the* ***Idea*** *section discusses the Hash approach.*)

While this is probably the correct answer for an easy problem, it doesn't really get to the heart of what a **hashmap** really is. Hashmaps were created to speed up the lookup time of data to a hopefully **O(1) time**. **Arrays** do this naturally with index lookups, but it becomes more complicated if you're trying to look up an entry by some other non-index value instead.

We can see from the basic solution here that its easy enough to mimic a **key** lookup if the keys themselves are integers that are constrained enough to act as their own indexes. But what if they're not? Or what if they're some other data type, like strings?

Surprisingly, the idea in that case is somewhat similar. We can still use an Array to store the data, but we must first find a way to transform the key into an index. For that, we look to a **hashing function**. Hashing functions exist to convert data into a randomized, but reproduceable, integer version of themselves.

In this case, we can use a hashing function to convert the key into an integer within the bounds of our hashmap array's index range. In an ideal situation, that would allow us to reduce the size of the hashmap array to the maximum number of entries, which is **10^4**. Unfortunately, however, it's always possible for **collisions** to exist when two keys devolve to the same integer via the hashing function.

To deal with collisions, we can just make each of our hashmap array's elements a **linked list**. This will allow us to treat them like a simple stack, where we look first at the most recently added **node** and then move to the next until we find the correct key.

Since navigating a linked list will drop our lookup time past **O(1)**, the goal of a good hashing function is to randomize the keys' hashes enough to limit collisions as much as possible for a given hashmap array size, thus keeping down the **average lookup time complexity**.

Therefore, the size of our hashmap array should probably be *at least* equal to the number of entries. Increasing the size of the hashmap array will naturally reduce collisions (and therefore time complexity) at the expense of **space complexity**, and vice versa. The tradeoff is highly dependent on the quality of the hashing function.

There are many, many hashing functions out there, but for this problem we'll use a very simple **multiplicative hashing function** utilizing a **large prime multiplier** and then **modulo** the result to the desired size (also a prime) of our hashmap array. This should hopefully result in an even distribution of the entries throughout the hashmap array.

The **get()** method is fairly easy, then. We just **hash()** our **key**, access the corresponding bucket in our hashmap array (**data**), and navigate through the linked list (if necessary) and return the correct **val**, or **-1** if the **key** is not found.

For the **put()** method, we should first **remove()** any earlier instance of that **key** to avoid chaining multiple versions of a **key** definition in our linked list. Then we simply form a **new ListNode** at the head of the proper hashmap bucket, pushing any others back.

The **remove()** method will be similar to the **get()** method, except that we need to find and stitch together the nodes on either side of the **node** that matches the **key**, removing it from the linked list entirely.

#### ***Implementation:***

Python has **deque** and Java has **LinkedList** that can do the work of the linked list management, but it comes at the cost of efficiency. In this case, it's not really worth using over a custom **ListNode** class implementation.

#### ***Java Code w/ Array:***

class MyHashMap {

int[] data;

public MyHashMap() {

data = new int[1000001];

Arrays.fill(data, -1);

}

public void put(int key, int val) {

data[key] = val;

}

public int get(int key) {

return data[key];

}

public void remove(int key) {

data[key] = -1;

}

}

#### ***Java Code w/ Hash:***

class ListNode {

int key, val;

ListNode next;

public ListNode(int key, int val, ListNode next) {

this.key = key;

this.val = val;

this.next = next;

}

}

class MyHashMap {

static final int size = 19997;

static final int mult = 12582917;

ListNode[] data;

public MyHashMap() {

this.data = new ListNode[size];

}

private int hash(int key) {

return (int)((long)key \* mult % size);

}

public void put(int key, int val) {

remove(key);

int h = hash(key);

ListNode node = new ListNode(key, val, data[h]);

data[h] = node;

}

public int get(int key) {

int h = hash(key);

ListNode node = data[h];

for (; node != null; node = node.next)

if (node.key == key) return node.val;

return -1;

}

public void remove(int key) {

int h = hash(key);

ListNode node = data[h];

if (node == null) return;

if (node.key == key) data[h] = node.next;

else for (; node.next != null; node = node.next)

if (node.next.key == key) {

node.next = node.next.next;

return;

}

}

}