# Question

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

To be specific, your design should include these functions:

* put(key, value) : Insert a (key, value) pair into the HashMap. If the value already exists in the HashMap, update the value.
* get(key): Returns the value to which the specified key is mapped, or -1 if this map contains no mapping for the key.
* remove(key) : Remove the mapping for the value key if this map contains the mapping for the key.

**Example:**

MyHashMap hashMap = new MyHashMap();

hashMap.put(1, 1);

hashMap.put(2, 2);

hashMap.get(1);            // returns 1

hashMap.get(3);            // returns -1 (not found)

hashMap.put(2, 1);          // update the existing value

hashMap.get(2);            // returns 1

hashMap.remove(2);          // remove the mapping for 2

hashMap.get(2);            // returns -1 (not found)

**Note:**

* All keys and values will be in the range of [0, 1000000].
* The number of operations will be in the range of [1, 10000].
* Please do not use the built-in HashMap library.

# Solution

#### **Intuition**

Hashmap is a common data structure that is implemented in various forms in different programming languages, e.g. dict in Python and HashMap in Java. The most distinguish characteristic about hashmap is that it provides a fast access to a **value** that is associated with a given **key**.

There are two main issues that we should tackle, in order to design an efficient hashmap data structure: 1). hash function design and 2). collision handling.

* **1). hash function design**: the purpose of hash function is to map a key value to an address in the storage space, similarly to the system that we assign a postcode to each mail address. As one can image, for a good hash function, it should map different keys **evenly** across the storage space, so that we don't end up with the case that the majority of the keys are concentrated in a few spaces.
* **2). collision handling**: essentially the hash function reduces the vast key space into a limited address space. As a result, there could be the case where two different keys are mapped to the same address, which is what we call 'collision'. Since the collision is inevitable, it is important that we have a strategy to handle the collision.

Depending on how we deal with each of the above two issues, we could have various implementation of hashmap data structure.

#### **Approach 1: Modulo + Array**

**Intuition**

As one of the most intuitive implementations, we could adopt the modulo operator as the hash function, since the key value is of integer type. In addition, in order to minimize the potential collisions, it is advisable to use a prime number as the base of modulo, e.g. 2069.

We organize the storage space as an **array** where each element is indexed with the output value of the hash function.

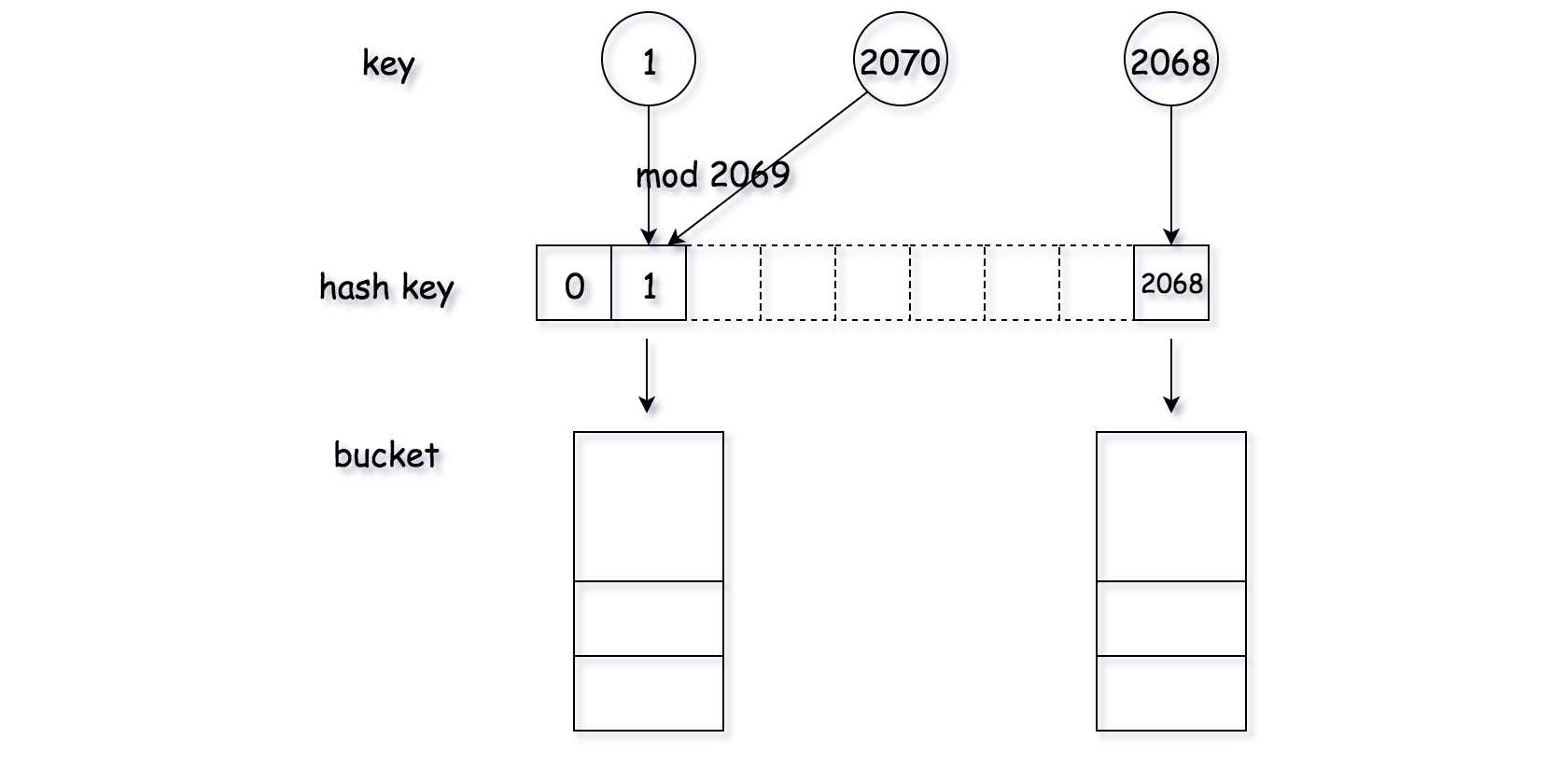
In case of collision, where two different keys are mapped to the same address, we use a **bucket** to hold all the values. The bucket is a container that hold all the values that are assigned by the hash function. We could use either a LinkedList or an Array to implement the bucket data structure.

**Algorithm**

For each of the methods in hashmap data structure, namely get(), put() and remove(), it all boils down to the method to locate the value that is stored in hashmap, given the key.

This localization process can be done in two steps:

* For a given key value, first we apply the hash function to generate a hash key, which corresponds to the address in our main storage. With this hash key, we would find the bucket where the value should be stored.
* Now that we found the bucket, we simply iterate through the bucket to check if the desired <key, value> pair does exist.



|  |
| --- |
| class Pair<U, V> {  public U first;  public V second;  public Pair(U first, V second) {  this.first = first;  this.second = second;  }  }  class Bucket {  private List<Pair<Integer, Integer>> bucket;  public Bucket() {  this.bucket = new LinkedList<Pair<Integer, Integer>>();  }  public Integer get(Integer key) {  for (Pair<Integer, Integer> pair : this.bucket) {  if (pair.first.equals(key))  return pair.second;  }  return -1;  }  public void update(Integer key, Integer value) {  boolean found = false;  for (Pair<Integer, Integer> pair : this.bucket) {  if (pair.first.equals(key)) {  pair.second = value;  found = true;  }  }  if (!found)  this.bucket.add(new Pair<Integer, Integer>(key, value));  }  public void remove(Integer key) {  for (Pair<Integer, Integer> pair : this.bucket) {  if (pair.first.equals(key)) {  this.bucket.remove(pair);  break;  }  }  }  }  class MyHashMap {  private int key\_space;  private List<Bucket> hash\_table;  /\*\* Initialize your data structure here. \*/  public MyHashMap() {  this.key\_space = 2069;  this.hash\_table = new ArrayList<Bucket>();  for (int i = 0; i < this.key\_space; ++i) {  this.hash\_table.add(new Bucket());  }  }  /\*\* value will always be non-negative. \*/  public void put(int key, int value) {  int hash\_key = key % this.key\_space;  this.hash\_table.get(hash\_key).update(key, value);  }  /\*\*  \* 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) {  int hash\_key = key % this.key\_space;  return this.hash\_table.get(hash\_key).get(key);  }  /\*\* Removes the mapping of the specified value key if this map contains a mapping for the key \*/  public void remove(int key) {  int hash\_key = key % this.key\_space;  this.hash\_table.get(hash\_key).remove(key);  }  }  /\*\*  \* 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);  \*/ |

Note that in the above implementations, we use Array to implement the bucket in Python, while we use LinkedList in Java.

**Complexity Analysis**

* Time Complexity: for each of the methods, the time complexity is O(N/K) where *N* is the number of all possible keys and *K* is the number of predefined buckets in the hashmap, which is 2069 in our case.
  + In the ideal case, the keys are evenly distributed in all buckets. As a result, on average, we could consider the size of the bucket is N/K.
  + Since in the worst case we need to iterate through a bucket to find the desire value, the time complexity of each method is O(*K/N*​).
* Space Complexity: \mathcal{O}(K+M)O(*K*+*M*) where K*K* is the number of predefined buckets in the hashmap and M*M* is the number of unique keys that have been inserted into the hashmap.