Hash Table

Introduction

`Hash Table` is a data structure which organizes data using `hash functions` in order to support quick insertion and search.

There are two different kinds of hash tables: hash set and hash map.

- The `hash set` is one of the implementations of a `set` data structure to store `no repeated values`.
- The `hash map` is one of the implementations of a `map` data structure to store `(key, value)` pairs.

It is `easy to use` a hash table with the help of `standard template libraries`. Most common languages such as Java, C++ and Python support both hash set and hash map.

By choosing a proper hash function, the hash table can achieve `wonderful performance` in both insertion and search.

In this card, we will answer the following questions:

1. What is the `principle` of a hash table?
2. How to `design` a hash table?
3. How to use `hash set` to solve duplicates related problems?
4. How to use `hash map` to aggregate information by key?
5. How to `design a proper key` when using a hash table?

Design a Hash Table

- The Principle of Hash Table
- Keys to Design a Hash Table
- Design HashSet
- Design HashMap
- Design Hash Table - Solution
- Complexity Analysis - Hash Table

Practical Application - Hash Set

- Hash Set - Usage
- Find Duplicates By Hash Set
- Contains Duplicate
- Single Number
- Intersection of Two Arrays
- Happy Number

Practical Application - Hash Map

- Hash Map - Usage
- Scenario I - Provide More Information
- Two Sum
- Isomorphic Strings
- Minimum Index Sum of Two Lists
- Scenario II - Aggregate by Key
- First Unique Character in a String
- Intersection of Two Arrays II
- Contains Duplicate II
- Logger Rate Limiter

Practical Application - Design the Key

- Design the Key
- Group Anagrams
- Group Shifted Strings
- Valid Sudoku
- Find Duplicate Subtrees
- Design the Key - Summary

Conclusion

- Jewels and Stones
- Longest Substring Without Repeating...
- Two Sum III - Data structure design
- 4Sum II
- Top K Frequent Elements
- Unique Word Abbreviation
- Insert Delete GetRandom O(1)

Design a Hash Table

In this chapter, we will discuss the underlying principle of the hash table.

After completing this chapter, you should be able to answer the following questions:

1. What is the `principle` of hash table?
2. Which factors will influence the choice of `hash function` and `collision resolution strategy`?
3. Understand the difference between a `hash set` and a `hash map`.
4. How to design a simple version of `hash set` and a `hash map` as in a typical `standard template library`.
5. What is the `complexity` of insertion and lookup operations?

Practical Application - Hash Set

In the previous chapter, we talked about how to design a hash table and the great performance of insertion and search in a hash table.

From this chapter on, we will focus on the `practical applications`.

In this chapter, we are going to talk about how to use the `hash set` with the help of standard template libraries and when we should use a hash set.

- [x] **Hash Set - Usage**
  Read this article if you are not familiar with the usage of the hash set.
- [x] **Find Duplicates By Hash Set**
  When to use the hash set?
- [x] **Contains Duplicate**
- [x] **Single Number**
- [x] **Intersection of Two Arrays**
- [x] **Happy Number**

Practical Application - Hash Map

We have known that the `hash set` is able to store only values. On the other hand, the `hash map` is an implementation of map which is able to store `(key, value)` pairs.

With the ability to store more information, the hash map can help us to solve more complicated problems. For example, we can `aggregate all the information by key` using a hash map and look up for the information related to a specific key in average constant time.

In this chapter, we will go through different scenarios to make better use of the hash map.

Practical Application - Design the Key

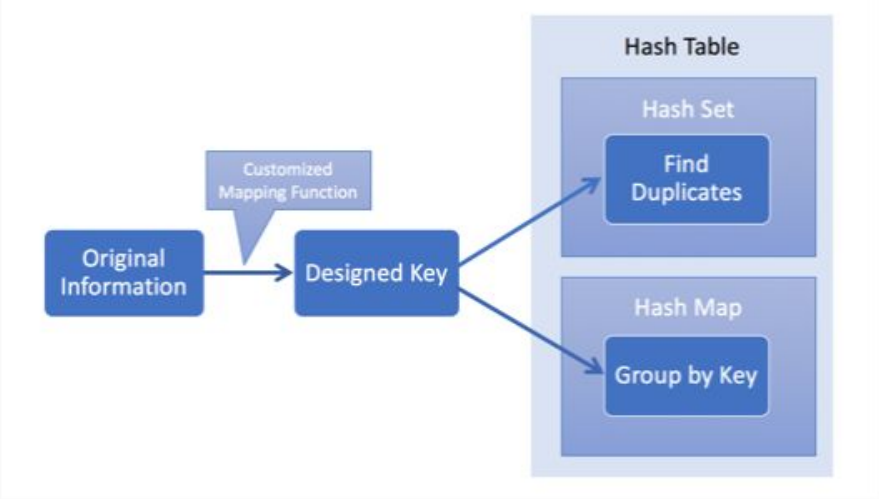
Another problem you might encounter is that when you meet some problems which you thought can be solved by a hash table, you are not able to `figure out a proper key`.

In this chapter, we are going to solve this problem. We will discuss with some examples, provide some exercise and finally, come to a conclusion to give you some tips to solve some classical problems.

Conclusion

We are now more familiar with `the principle and usage` of the hash table.

We have also talked about `how to apply hash table` from three respects in previous chapters. Here we combine them together and come up with a typical thinking process to solve problems by hash table flexibly.



What's more, we will meet more complicated problems sometimes. We might need to:

- use several hash tables together
- combine the hash table with other data structure
- combine the hash table with other algorithms
- ...

We provide some exercise in this chapter. After finishing this chapter, you will be more confident with hash table related problems.

- Jewels and Stones
- Longest Substring Without Repeating Characters
- Two Sum III - Data structure design
- 4Sum II
- Top K Frequent Elements
  Try to solve this problem using two hash maps.
- Unique Word Abbreviation
  Combining DFS with HashTable is the key to solve this problem.
- Insert Delete GetRandom O(1)

The Principle of Hash Table

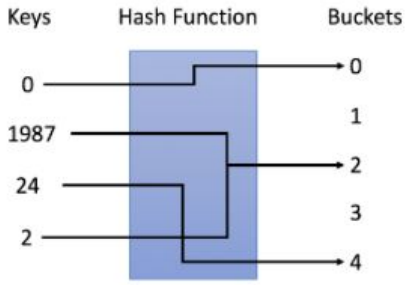
As we mentioned in the introduction, `Hash Table` is a data structure which organizes data using `hash functions` in order to support `quick insertion and search`. In this article, we will take a look at the principle of the hash table.

The Principle of Hash Table

The key idea of Hash Table is to use a hash function to `map keys to buckets`. To be more specific,

1. When we insert a new key, the hash function will decide which bucket the key should be assigned and the key will be stored in the corresponding bucket;
2. When we want to search for a key, the hash table will use the `same` hash function to find the corresponding bucket and search only in the specific bucket.

An Example



In the example, we use `y = x % 5` as our hash function. Let's go through the insertion and search strategies using this example:

1. Insertion: we parse the keys through the hash function to map them into the corresponding bucket.
    - e.g. 1987 is assigned to bucket 2 while 24 is assigned to bucket 4.
2. Search: we parse the keys through the same hash function and search only in the specific bucket.
    - e.g. if we search for 1987, we will use the same hash function to map 1987 to 2. So we search in bucket 2 and we successfully find out 1987 in that bucket.
    - e.g. if we search for 23, will map 23 to 3 and search in bucket 3. And We find out that 23 is not in bucket 3 which means 23 is not in the hash table.

A Keys to Design a Hash Table

There are two essential factors that you should pay attention to when you are going to design a hash table.

1. Hash Function

The hash function is the most important component of a hash table which is used to map the key to a specific bucket. In the example in previous article, we use $y = x \% 5$ as a hash function, where x is the key value and y is the index of the assigned bucket.

The hash function will depend on `the range of key values` and `the number of buckets`.

Here are some examples of hash functions:

Key Type	Key Range	Number of Buckets	Hash Function Example
integer	0 to 100,000	1000	y = x % 1000
char	'a' to 'z'	26	y = x – 'a'
array of integers	size < 10, each number ∈ [0,1]	1024	$y = x_0 * 2^0 + x_1 * 2^1 + ... + x_9 * 2^9$
array of integers	size < 10, each number ∈ [0,3]	1024	$y = x_0 * 4^0 + x_1 * 4^1 + ... + x_4 * 4^4$

* In the table above, we use x to represent the key and y to represent our hash result.

It is an open problem to design a hash function. The idea is to try to assign the key to the bucket as `uniform as you can`. Ideally, a perfect hash function will be a one-one mapping between the key and the bucket. However, in most cases a hash function is not perfect and it is a tradeoff between `the amount of buckets` and `the capacity of a bucket`.

2. Collision Resolution

Ideally, if our hash function is a perfect `one-one mapping`, we will not need to handle collisions. Unfortunately, in most cases, collisions are almost `inevitable`. For instance, in our previous hash function ($y = x \% 5$), both 1987 and 2 are assigned to bucket 2. That is a `collision`.

A collision resolution algorithm should solve the following questions:

1. How to organize the values in the same bucket?
2. What if too many values are assigned to the same bucket?
3. How to search a target value in a specific bucket?

These questions are related to `the capacity of the bucket` and `the number of keys` which might be mapped into `the same bucket` according to our hash function.

Let's assume that the bucket, which holds the maximum number of keys, has `N` keys.

Typically, if N is constant and small, we can simply use an `array` to store keys in the same bucket. If N is variable or large, we might need to use `height-balanced binary search tree` instead.

Exercise

By now, you should be able to implement a basic hash table. We provide the exercise for you to implement a hash set and a hash map. `Read the requirement, determine your hash function and solve the collision if needed.`

> *If you are not familiar with the concepts of hash set and hash map, you can go back to the introduction part to find out the answer.*

Design HashSet

Go to Discuss

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

To be specific, your design should include these functions:

- `add(value)` : Insert a value into the HashSet.
- `contains(value)` : Return whether the value exists in the HashSet or not.
- `remove(value)` : Remove a value in the HashSet. If the value does not exist in the HashSet, do nothing.

Example:

```
MyHashSet hashSet = new MyHashSet();
hashSet.add(1);
hashSet.add(2);
hashSet.contains(1);    // returns true
hashSet.contains(3);    // returns false (not found)
hashSet.add(2);
hashSet.contains(2);    // returns true
hashSet.remove(2);
hashSet.contains(2);    // returns false (already removed)
```

Note:

- All 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 HashSet library.

Python

```python
class MyHashSet(object):

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.hashset = set()

    def add(self, key):
        """
        :type key: int
        :rtype: void
        """
        self.hashset.add(key)

    def remove(self, key):
        """
        :type key: int
        :rtype: void
        """
        if key in self.hashset:
            self.hashset.remove(key)

    def contains(self, key):
        """
        Returns true if this set contains the specified element
        :type key: int
        :rtype: bool
        """
        if key in self.hashset:
            return True
        else:
            return False
```

Design HashMap

Go to Discuss

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.

```python
class MyHashMap(object):

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.table = [-1] * 1000001

    def put(self, key, value):
        """
        value will always be non-negative.
        :type key: int
        :type value: int
        :rtype: void
        """
        self.table[key] = value


    def get(self, key):
        """
        Returns the value to which the specified key is mapped, or -1 if this map contains no mapping for the key
        :type key: int
        :rtype: int
        """
        return self.table[key]


    def remove(self, key):
        """
        Removes the mapping of the specified value key if this map contains a mapping for the key
        :type key: int
        :rtype: void
        """
        self.table[key] = -1

```

A Design Hash Table - Solution

Here are C++ and Java solutions for your reference. In our solution, we use an `array` to represent the hash set. Each element in the array is a bucket. And in each bucket, we use the `array list` (or `vector` in C++) to store all the values.

More

Let's take a look at the operation `"remove"`. After we find out the position of the element, we need to `remove the element from the array list`.

Let's assume that we are going to remove the `ith` element and the size of the array list is `n`.

The strategy used in the built-in function is to move all the elements after ith element one position forward. That is to say, you have to move `n - i` times. So the time complexity to remove an element from an array list will be `O(n)`.

> Consider different value of i. In average, we will move ((n - 1) + (n - 2) + ... + 1 + 0) / n = (n - 1) / 2 times.

Hopefully, there are two solutions to reduce the time complexity from `O(n)` to `O(1)`.

1. Swap

There is a tricky strategy we can use. First, `swap the element` which we want to remove with the last element in the bucket. Then remove the last element. By this way, we successfully remove the element in `O(1)` time complexity.

2. Linked List

Another way to achieve this goal is to use a `linked list` instead of an array list. By this way, we can remove the element in O(1) time complexity `without modifying the order` in the list.

A Complexity Analysis - Hash Table

In this article, we are going to discuss the performance of hash table.

Complexity Analysis

If there are `M` keys in total, we can achieve the space complexity of `O(M)` easily when using a hash table.

However, you might have noticed that the time complexity of hash table has a strong relationship with the design.

Most of us might have used `an array` in each bucket to store values in the same bucket. Ideally, the bucket size is small enough to be regarded as a `constant`. The time complexity of both insertion and search will be `O(1)`.

But in the worst case, the maximum bucket size will be `N`. And the time complexity will be `O(1)` for insertion but `O(N)` for search.

> `Insertion` and `search` are two basic operations in a hash table.
>
> Besides, there are operations which are based on these two operations. For example, when we `remove an element`, we will first search the element and then remove the element from the corresponding position if the element exists.

The Principle of Built-in Hash Table

The typical design of built-in hash table is:

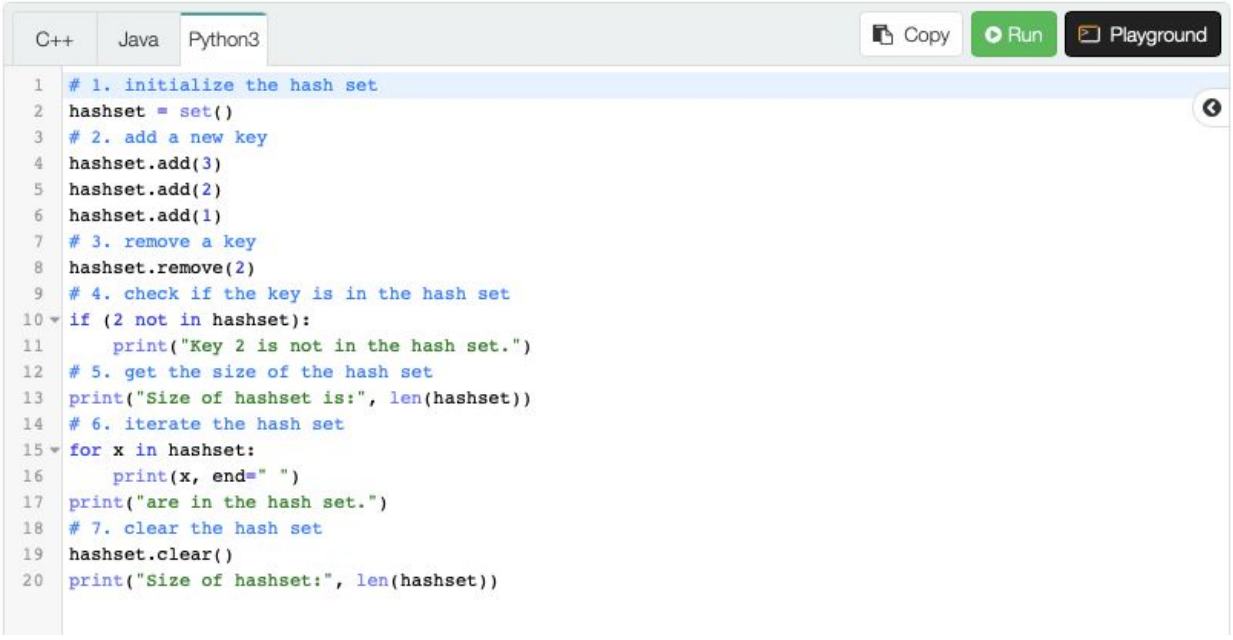
1. The key value can be any `hashable` type. And a value which belongs to a hashable type will have a `hashcode`. This code will be used in the mapping function to get the bucket index.
2. Each bucket contains `an array` to store all the values in the same bucket initially.
3. If there are too many values in the same bucket, these values will be maintained in a `height-balanced binary search tree` instead.

The average time complexity of both insertion and search is still `O(1)`. And the time complexity in the worst case is `O(logN)` for both insertion and search by using height-balanced BST. It is a trade-off between insertion and search.

A Hash Set - Usage

The `hash set` is one of the implementations of a `set` which is a data structure to store `no repeated values`.

We provide an example of using the hash set in Java, C++ and Python. If you are not familiar with the usage of the hash set, it will be helpful to go through the example.



A Find Duplicates By Hash Set

As we know, it is easy and effective to insert a new value and check if a value is in a hash set or not.

Therefore, typically, a hash set is used to `check if a value has ever appeared or not`.

An Example

Let's look at an example:

> Given an array of integers, find if the array contains any duplicates.

This is a typical problem which can be solved by a hash set.

You can simply iterate each value and insert the value into the set. If a value has already been in the hash set, there is a duplicate.

Template

Here we provide a template for you to solve this kind of problems:

C++ Java Copy

```java
/*
 * Template for using hash set to find duplicates.
 */
boolean findDuplicates(List<Type>& keys) {
    // Replace Type with actual type of your key
    Set<Type> hashset = new HashSet<>();
    for (Type key : keys) {
        if (hashset.contains(key)) {
            return true;
        }
        hashset.insert(key);
    }
    return false;
}

```

Contains Duplicate

Go to Discuss

Given an array of integers, find if the array contains any duplicates.

Your function should return true if any value appears at least twice in the array, and it should return false if every element is distinct.

Example 1:

```
Input: [1,2,3,1]
Output: true
```

Example 2:

```
Input: [1,2,3,4]
Output: false
```

Example 3:

```
Input: [1,1,1,3,3,4,3,2,4,2]
Output: true
```

Python

```python
class Solution(object):
    def containsDuplicate(self, nums):
        """
        :type nums: List[int]
        :rtype: bool
        """
        def containsDuplicate(self, nums):
            return len(nums) > len(set(nums))
```

Intersection of Two Arrays

Go to Discuss

Given two arrays, write a function to compute their intersection.

Example 1:

```
Input: nums1 = [1,2,2,1], nums2 = [2,2]
Output: [2]
```

Example 2:

```
Input: nums1 = [4,9,5], nums2 = [9,4,9,8,4]
Output: [9,4]
```

Note:

- Each element in the result must be unique.
- The result can be in any order.

Python

```python
class Solution(object):
    def intersection(self, nums1, nums2):
        """
        :type nums1: List[int]
        :type nums2: List[int]
        :rtype: List[int]
        """
        nums1_set = set(nums1)
        result_set = set([])
        for x in nums2:
            if x in nums1_set and x not in result_set:
                result_set.add(x)
        return [x for x in result_set]
```

Custom Testcase (Contribute)

Run Code | Submit Solution

Happy Number

Go to Discuss

Write an algorithm to determine if a number is "happy".

A happy number is a number defined by the following process: Starting with any positive integer, replace the number by the sum of the squares of its digits, and repeat the process until the number equals 1 (where it will stay), or it loops endlessly in a cycle which does not include 1. Those numbers for which this process ends in 1 are happy numbers.

Example:

Input: 19
Output: true
Explanation:
$1^2 + 9^2 = 82$
$8^2 + 2^2 = 68$
$6^2 + 8^2 = 100$
$1^2 + 0^2 + 0^2 = 1$

Python

```python
class Solution(object):
    def recur(self, n):
        if n == 1:
            return True

        string = str(n)
        add = 0
        for digit in string:
            add += int(digit) ** 2

        if add in self.Map:
            return False
        else:
            self.Map[add] = 0
        return self.recur(add)

    def isHappy(self, n):
        """
        :type n: int
        :rtype: bool
        """
        self.Map = {}
        return self.recur(n)
```

Custom Testcase (Contribute)

Run Code Submit Solution

Hash Map - Usage

The `hash map` is one of the implementations of a `map` which is used to store `(key, value)` pairs.

We provide an example of using the hash map in Java, C++ and Python. If you are not familiar with the usage of the hash map, it will be helpful to go through the example.

C++ Java Python3 Copy Run Playground

```python
# 1. initialize a hash map
hashmap = {0 : 0, 2 : 3}
# 2. insert a new (key, value) pair or update the value of existed key
hashmap[1] = 1
hashmap[1] = 2
# 3. get the value of a key
print("The value of key 1 is: " + str(hashmap[1]))
# 4. delete a key
del hashmap[2]
# 5. check if a key is in the hash map
if 2 not in hashmap:
    print("Key 2 is not in the hash map.")
# 6. both key and value can have different type in a hash map
hashmap["pi"] = 3.1415
# 7. get the size of the hash map
print("The size of hash map is: " + str(len(hashmap)))
# 8. iterate the hash map
for key in hashmap:
    print("(" + str(key) + "," + str(hashmap[key]) + ")", end=" ")
print("are in the hash map.")
# 9. get all keys in hash map
print(hashmap.keys())
# 10. clear the hash map
hashmap.clear();
print("The size of hash map is: " + str(len(hashmap)))

```

A Scenario I - Provide More Information

The first scenario to use a hash map is that we `need more information` rather than only the key. Then we can `build a mapping relationship between key and information` by hash map.

An Example

Let's look at an example:

> Given an array of integers, return **indices** of the two numbers such that they add up to a specific target.

In this example, if we only want to return true if there is a solution, we can use a hash set to store all the values when we iterate the array and check if `target - current_value` is in the hash set or not.

However, we are asked to `return more information` which means we not only care about the value but also care about the index. We need to store not only the number as the key but also the index as the value. Therefore, we should use a hash map rather than a hash set.

What's More

In some cases, we need more information not just to return more information but also to `help us with our decisions`.

In the previous examples, when we meet a duplicated key, we will return the corresponding information immediately. But sometimes, we might want to check if the value of the key is acceptable first.

Template

Here we provide a template for you to solve this kind of problems:

C++ | Java

Copy

```java
/*
 * Template for using hash map to find duplicates.
 * Replace ReturnType with the actual type of your return value.
 */
ReturnType aggregateByKey_hashmap(List<Type>& keys) {
    // Replace Type and InfoType with actual type of your key and value
    Map<Type, InfoType> hashmap = new HashMap<>();
    for (Type key : keys) {
        if (hashmap.containsKey(key)) {
            if (hashmap.get(key) satisfies the requirement) {
                return needed_information;
            }
        }
        // Value can be any information you needed (e.g. index)
        hashmap.put(key, value);
    }
    return needed_information;
}
```

Single Number

Go to Discuss

Given a **non-empty** array of integers, every element appears *twice* except for one. Find that single one.

Note:

Your algorithm should have a linear runtime complexity. Could you implement it without using extra memory?

Example 1:

```
Input: [2,2,1]
Output: 1
```

Example 2:

```
Input: [4,1,2,1,2]
Output: 4
```

Python

```python
class Solution(object):
    def singleNumber(self, nums):
        d = set()
        for num in nums:
            if num in d:
                d.remove(num)
            else:
                d.add(num)
        return d.pop()
```

Two Sum

Go to Discuss

Given an array of integers, return **indices** of the two numbers such that they add up to a specific target.

You may assume that each input would have ***exactly*** one solution, and you may not use the *same* element twice.

Example:

```
Given nums = [2, 7, 11, 15], target = 9,

Because nums[0] + nums[1] = 2 + 7 = 9,
return [0, 1].
```

Python

```python
class Solution(object):
    def twoSum(self,nums, target):
        map ={}
        for i in range(len(nums)):
            if nums[i] not in map:
                map[target -nums[i]] = i
            else:
                return map[nums[i]], i
        return -1,-1

```

Isomorphic Strings

Go to Discuss

Given two strings ***s*** and ***t***, determine if they are isomorphic.

Two strings are isomorphic if the characters in ***s*** can be replaced to get ***t***.

All occurrences of a character must be replaced with another character while preserving the order of characters. No two characters may map to the same character but a character may map to itself.

Example 1:

```
Input: s = "egg", t = "add"
Output: true
```

Example 2:

```
Input: s = "foo", t = "bar"
Output: false
```

Example 3:

```
Input: s = "paper", t = "title"
Output: true
```

Note:

You may assume both ***s*** and ***t*** have the same length.

Python

```python
class Solution(object):
    def isIsomorphic(self, s, t):
        """
        :type s: str
        :type t: str
        :rtype: bool
        """
        return [s.find(i) for i in s] == [t.find(j) for j in t]
```

Minimum Index Sum of Two Lists

Go to Discuss

Suppose Andy and Doris want to choose a restaurant for dinner, and they both have a list of favorite restaurants represented by strings.

You need to help them find out their **common interest** with the **least list index sum**. If there is a choice tie between answers, output all of them with no order requirement. You could assume there always exists an answer.

Example 1:

```
Input:
["Shogun", "Tapioca Express", "Burger King", "KFC"]
["Piatti", "The Grill at Torrey Pines", "Hungry Hunter Steakhouse", "Shogun"]
Output: ["Shogun"]
Explanation: The only restaurant they both like is "Shogun".
```

Example 2:

```
Input:
["Shogun", "Tapioca Express", "Burger King", "KFC"]
["KFC", "Shogun", "Burger King"]
Output: ["Shogun"]
Explanation: The restaurant they both like and have the least index sum is "Shogun" with index sum 1 (0+1).
```

Note:

1. The length of both lists will be in the range of [1, 1000].
2. The length of strings in both lists will be in the range of [1, 30].
3. The index is starting from 0 to the list length minus 1.
4. No duplicates in both lists.

Python

```python
class Solution(object):
    def findRestaurant(self, list1, list2):
        """
        :type list1: List[str]
        :type list2: List[str]
        :rtype: List[str]
        """
        dic1, dic2 = {}, {}
        res_sum = len(list1)+len(list2)
        for i, s in enumerate(list1):
            dic1[s] = i
        for i, s in enumerate(list2):
            if s in dic1:
                tmp_sum = dic1[s] + i
                dic2[tmp_sum] = dic2.get(tmp_sum, []) + [s]
                res_sum = min(tmp_sum, res_sum)
        return dic2[res_sum]
```

Custom Testcase (Contribute)

Run Code | Submit Solution

A Scenario II - Aggregate by Key

Another frequent scenario is to `aggregate all the information by key`. We can also use a hash map to achieve this goal.

An Example

Here is an example:

> Given a string, find the first non-repeating character in it and return it's index. If it doesn't exist, return -1.

A simple way to solve this problem is to `count the occurrence` of each character first. And then go through the results to find out the first unique character.

Therefore, we can maintain a hashmap whose key is the character while the value is a counter for the corresponding character. Each time when we iterate a character, we just add the corresponding value by 1.

What's more

The key to solving this kind of problem is to `decide your strategy when you encounter an existing key`.

In the example above, our strategy is to count the occurrence. Sometimes, we might sum all the values up. And sometimes, we might replace the original value with the newest one. The strategy depends on the problem and practice will help you make a right decision.

Template

Here we provide a template for you to solve this kind of problems:

C++ Java Copy

```java
/*
 * Template for using hash map to find duplicates.
 * Replace ReturnType with the actual type of your return value.
 */
ReturnType aggregateByKey_hashmap(List<Type>& keys) {
    // Replace Type and InfoType with actual type of your key and value
    Map<Type, InfoType> hashmap = new HashMap<>();
    for (Type key : keys) {
        if (hashmap.containsKey(key)) {
            hashmap.put(key, updated_information);
        }
        // Value can be any information you needed (e.g. index)
        hashmap.put(key, value);
    }
    return needed_information;
}
```

First Unique Character in a String

Go to Discuss

Given a string, find the first non-repeating character in it and return it's index. If it doesn't exist, return -1.

Examples:

```
s = "leetcode"
return 0.

s = "loveleetcode",
return 2.
```

Note: You may assume the string contain only lowercase letters.

Python

```python
class Solution(object):
    def firstUniqChar(self, s):
        """
        :type s: str
        :rtype: int
        """
        for i in range(len(s)):
            c = s[i]
            if s.count(c) == 1:
                return i
        return -1
```

Intersection of Two Arrays II

Go to Discuss

Given two arrays, write a function to compute their intersection.

Example 1:

```
Input: nums1 = [1,2,2,1], nums2 = [2,2]
Output: [2,2]
```

Example 2:

```
Input: nums1 = [4,9,5], nums2 = [9,4,9,8,4]
Output: [4,9]
```

Note:

- Each element in the result should appear as many times as it shows in both arrays.
- The result can be in any order.

Follow up:

- What if the given array is already sorted? How would you optimize your algorithm?
- What if *nums1*'s size is small compared to *nums2*'s size? Which algorithm is better?
- What if elements of *nums2* are stored on disk, and the memory is limited such that you cannot load all elements into the memory at once?

Python

```python
class Solution(object):
    def intersect(self, nums1, nums2):
        """
        :type nums1: List[int]
        :type nums2: List[int]
        :rtype: List[int]
        """
        # res =[]
        # while nums1:
        #     curNum = nums1.pop()
        #     if curNum in nums2:
        #         res.append(curNum)
        #         nums2.remove(curNum)
        # return res
        C1 = collections.Counter(nums1)
        C2 = collections.Counter(nums2)
        res =[]
        for i in C1:
            if i in C2:
                res += [i] * min([C1[i], C2[i]])
        return res

```

Custom Testcase (Contribute)

Run Code | Submit Solution

Contains Duplicate II

Go to Discuss

Given an array of integers and an integer *k*, find out whether there are two distinct indices *i* and *j* in the array such that **nums[i] = nums[j]** and the **absolute** difference between *i* and *j* is at most *k*.

Example 1:

```
Input: nums = [1,2,3,1], k = 3
Output: true
```

Example 2:

```
Input: nums = [1,0,1,1], k = 1
Output: true
```

Example 3:

```
Input: nums = [1,2,3,1,2,3], k = 2
Output: false
```

Python

```python
class Solution(object):
    def containsNearbyDuplicate(self, nums, k):
        dic = {}
        for i, v in enumerate(nums):
            if v in dic and i - dic[v] <= k:
                return True
            dic[v] = i
        return False
```

Logger Rate Limiter

Go to Discuss

Design a logger system that receive stream of messages along with its timestamps, each message should be printed if and only if it is **not printed in the last 10 seconds**.

Given a message and a timestamp (in seconds granularity), return true if the message should be printed in the given timestamp, otherwise returns false.

It is possible that several messages arrive roughly at the same time.

Example:

```
Logger logger = new Logger();

// logging string "foo" at timestamp 1
logger.shouldPrintMessage(1, "foo"); returns true;

// logging string "bar" at timestamp 2
logger.shouldPrintMessage(2,"bar"); returns true;

// logging string "foo" at timestamp 3
logger.shouldPrintMessage(3,"foo"); returns false;

// logging string "bar" at timestamp 8
logger.shouldPrintMessage(8,"bar"); returns false;

// logging string "foo" at timestamp 10
logger.shouldPrintMessage(10,"foo"); returns false;

// logging string "foo" at timestamp 11
logger.shouldPrintMessage(11,"foo"); returns true;
```

Python

```python
class Logger(object):
    def __init__(self):
        self.myDict = {}

    def shouldPrintMessage(self, timestamp, message):
        #if happened before
        if message in self.myDict:
            #check the time stamp
            if timestamp - self.myDict.get(message) < 10:
                #don't print
                return False
            else:
                self.myDict.update( {message: timestamp})
                return True
        #happened for the first time
        else:
            self.myDict.update( {message: timestamp})
            return True


# Your Logger object will be instantiated and called as such:
# obj = Logger()
# param_1 = obj.shouldPrintMessage(timestamp,message)
```

Custom Testcase (Contribute)

Run Code | Submit Solution

A Design the Key

In the previous problems, the choice of key is comparatively straightforward. Unfortunately, sometimes you have to think it over to `design a suitable key` when using a hash table.

An Example

Let's look at an example:

> Given an array of strings, group anagrams together.

As we know, a hash map can perform really well in grouping information by key. But we cannot use the original string as key directly. We have to design a proper key to present the type of anagrams. For instance, there are two strings "eat" and "ate" which should be in the same group. While "eat" and "act" should not be grouped together.

Solution

Actually, `designing a key` is to `build a mapping relationship by yourself` between the original information and the actual key used by hash map. When you design a key, you need to guarantee that:

> 1. All values belong to the same group will be mapped in the same group.
> 2. Values which needed to be separated into different groups will not be mapped into the same group.

This process is similar to design a hash function, but here is an essential difference. `A hash function satisfies the first rule but might not satisfy the second one.` But your mapping function should satisfy both of them.

In the example above, our mapping strategy can be: sort the string and use the sorted string as the key. That is to say, both "eat" and "ate" will be mapped to "aet".

The mapping strategy can be really `tricky` sometimes. We provide some exercise for you in this chapter and will give a summary after that.

Group Anagrams

Go to Discuss

Given an array of strings, group anagrams together.

Example:

```
Input: ["eat", "tea", "tan", "ate", "nat", "bat"],
Output:
[
  ["ate","eat","tea"],
  ["nat","tan"],
  ["bat"]
]
```

Note:

- All inputs will be in lowercase.
- The order of your output does not matter.

Python

```python
class Solution(object):
    def groupAnagrams(self, strs):
        d = {}
        for w in sorted(strs):
            key = tuple(sorted(w))
            d[key] = d.get(key, []) + [w]
        return d.values()
```

Group Shifted Strings

Go to Discuss

Given a string, we can "shift" each of its letter to its successive letter, for example: `"abc" -> "bcd"`. We can keep "shifting" which forms the sequence:

```
"abc" -> "bcd" -> ... -> "xyz"
```

Given a list of strings which contains only lowercase alphabets, group all strings that belong to the same shifting sequence.

Example:

```
Input: ["abc", "bcd", "acef", "xyz", "az", "ba", "a", "z"],
Output:
[
  ["abc","bcd","xyz"],
  ["az","ba"],
  ["acef"],
  ["a","z"]
]
```

Python

```python
class Solution:
    def groupStrings(self, strings):
        """
        :type strings: List[str]
        :rtype: List[List[str]]
        """
        dic = {}
        for string in strings:
            key = tuple((ord(char) - ord(string[0])) % 26 for char in string)
            if key in dic:
                dic[key].append(string)
            else:
                dic[key] = [string]
        return [value for _, value in dic.items()]
```

Valid Sudoku

Go to Discuss

Determine if a 9x9 Sudoku board is valid. Only the filled cells need to be validated **according to the following rules**:

1. Each row must contain the digits `1-9` without repetition.
2. Each column must contain the digits `1-9` without repetition.
3. Each of the 9 `3x3` sub-boxes of the grid must contain the digits `1-9` without repetition.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

A partially filled sudoku which is valid.

The Sudoku board could be partially filled, where empty cells are filled with the character `'.'`.

Example 1:

```
Input:
[
  ["5","3",".",".","7",".",".",".","."],
  ["6",".",".","1","9","5",".",".","."],
  [".","9","8",".",".",".",".","6","."],
  ["8",".",".",".","6",".",".",".","3"],
  ["4",".",".","8",".","3",".",".","1"],
  ["7",".",".",".","2",".",".",".","6"],
  [".","6",".",".",".",".","2","8","."],
  [".",".",".","4","1","9",".",".","5"],
  [".",".",".",".","8",".",".","7","9"]
]
Output: true
```

Example 2:

```
Input:
[
  ["8","3",".",".","7",".",".",".","."],
  ["6",".",".","1","9","5",".",".","."],
  [".","9","8",".",".",".",".","6","."],
  ["8",".",".",".","6",".",".",".","3"],
  ["4",".",".","8",".","3",".",".","1"],
  ["7",".",".",".","2",".",".",".","6"],
  [".","6",".",".",".",".","2","8","."],
  [".",".",".","4","1","9",".",".","5"],
  [".",".",".",".","8",".",".","7","9"]
]
Output: false
Explanation: Same as Example 1, except with the 5 in the top left corner being
    modified to 8. Since there are two 8's in the top left 3x3 sub-box, it is invalid.
```

Note:

Python

```python
class Solution(object):
    def isValidSudoku(self, board):
        return 1 == max(collections.Counter(
        x
        for i, row in enumerate(board)
        for j, c in enumerate(row)
        if c != '.'
        for x in ((c, i), (j, c), (i/3, j/3, c))
    ).values() + [1])
```

Find Duplicate Subtrees

Go to Discuss

Given a binary tree, return all duplicate subtrees. For each kind of duplicate subtrees, you only need to return the root node of any **one** of them.

Two trees are duplicate if they have the same structure with same node values.

Example 1:

```
        1
      / \
     2   3
    /   / \
   4   2   4
      /
     4
```

The following are two duplicate subtrees:

```
     2
    /
   4
```

and

```
   4
```

Therefore, you need to return above trees' root in the form of a list.

Python

```python
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    def findDuplicateSubtrees(self, root):
        """
        :type root: TreeNode
        :rtype: List[TreeNode]
        """
        dic = dict()
        res = set()
        self.helper(root, dic, res)
        return list(res)

    def helper(self, root, dic, res):
        if not root:
            return None

        left = self.helper(root.left, dic, res)
        right = self.helper(root.right, dic, res)
        tree = (left, root.val, right)
        if tree not in dic:
            dic[tree] = root
        else:
            res.add(dic[tree])
        return dic[tree]
```

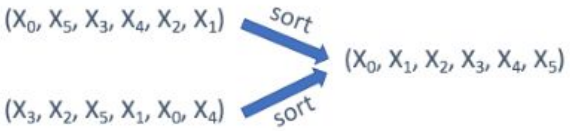
Custom Testcase (Contribute)

Run Code | Submit Solution

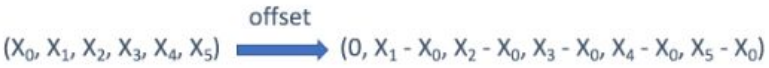
A Design the Key - Summary

Here are some takeaways about how to design the key for you.

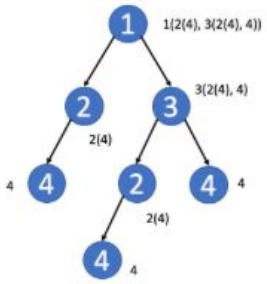
1. When the order of each element in the string/array doesn't matter, you can use the `sorted string/array` as the key.



2. If you only care about the offset of each value, usually the offset from the first value, you can use the `offset` as the key.

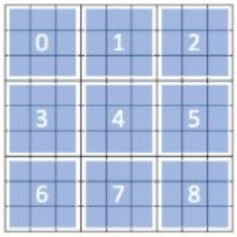


3. In a tree, you might want to directly use the `TreeNode` as key sometimes. But in most cases, the `serialization of the subtree` might be a better idea.



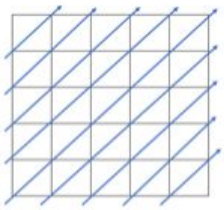
4. In a matrix, you might want to use `the row index` or `the column index` as key.
5. In a Sudoku, you can combine the row index and the column index to identify which `block` this element belongs to.

5. In a Sudoku, you can combine the row index and the column index to identify which `block` this element belongs to.

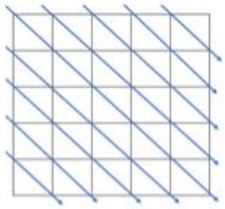


(i, j) ➔ (i / 3) * 3+ j / 3

6. Sometimes, in a matrix, you might want to aggregate the values in the same `diagonal line`.



Anti-Diagonal Order
(i, j) ➔ i + j



Diagonal Order
(i, j) ➔ i – j

Jewels and Stones

Go to Discuss

You're given strings `J` representing the types of stones that are jewels, and `S` representing the stones you have. Each character in `S` is a type of stone you have. You want to know how many of the stones you have are also jewels.

The letters in `J` are guaranteed distinct, and all characters in `J` and `S` are letters. Letters are case sensitive, so `"a"` is considered a different type of stone from `"A"`.

Example 1:

```
Input: J = "aA", S = "aAAbbbb"
Output: 3
```

Example 2:

```
Input: J = "z", S = "ZZ"
Output: 0
```

Note:

- `S` and `J` will consist of letters and have length at most 50.
- The characters in `J` are distinct.

Python

```python
class Solution(object):
    def numJewelsInStones(self, J, S):
        """
        :type J: str
        :type S: str
        :rtype: int
        """
        total =0
        for c in J:
            total = total + S.count(c)
        return total

```

Longest Substring Without Repeating Characters

Go to Discuss

Given a string, find the length of the **longest substring** without repeating characters.

Example 1:

```
Input: "abcabcbb"
Output: 3
Explanation: The answer is "abc", with the length of 3.
```

Example 2:

```
Input: "bbbbb"
Output: 1
Explanation: The answer is "b", with the length of 1.
```

Example 3:

```
Input: "pwwkew"
Output: 3
Explanation: The answer is "wke", with the length of 3.
             Note that the answer must be a substring, "pwke" is a subsequence and not a substring.
```

Python

```python
class Solution(object):
    def lengthOfLongestSubstring(self, s):
        """
        :type s: str
        :rtype: int
        """
        seen = {}
        max_len = 0
        start = 0
        for i, c in enumerate(s):
            if c in seen and start <= seen[c]:
                max_len = max(i - start, max_len)
                start = seen[c] + 1
            seen[c] = i
        max_len = max(max_len, len(s)-start)

        return max_len
    def lenthOfLongestSubstring(self,s):
        seen ={}
        max_len = 0
        start = 0
```

Two Sum III - Data structure design

Go to Discuss

Design and implement a TwoSum class. It should support the following operations: `add` and `find`.

`add` - Add the number to an internal data structure.

`find` - Find if there exists any pair of numbers which sum is equal to the value.

Example 1:

```
add(1); add(3); add(5);
find(4) -> true
find(7) -> false
```

Example 2:

```
add(3); add(1); add(2);
find(3) -> true
find(6) -> false
```

```python
class TwoSum(object):

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.numbers = {}
        self.min_num = float('inf')
        self.max_num = float('-inf')

    def add(self, number):
        """
        Add the number to an internal data structure..
        :type number: int
        :rtype: void
        """
        self.numbers[number] = self.numbers.get(number, None) is not None
        self.min_num = min(self.min_num, number)
        self.max_num = max(self.max_num, number)

    def find(self, value):
        """
        Find if there exists any pair of numbers which sum is equal to the value.
        :type value: int
        :rtype: bool
        """
        if value < self.min_num * 2 or value > self.max_num * 2:
            return False

        return any(value - number in self.numbers and (value - number != number or self.numbers[number])
                   for number in self.numbers)
```

Custom Testcase (Contribute)

Run Code | Submit Solution

4Sum II

Go to Discuss

Given four lists A, B, C, D of integer values, compute how many tuples `(i, j, k, l)` there are such that `A[i] + B[j] + C[k] + D[l]` is zero.

To make problem a bit easier, all A, B, C, D have same length of N where $0 \le N \le 500$. All integers are in the range of -2^{28} to $2^{28} - 1$ and the result is guaranteed to be at most $2^{31} - 1$.

Example:

```
Input:
A = [ 1, 2]
B = [-2,-1]
C = [-1, 2]
D = [ 0, 2]

Output:
2

Explanation:
The two tuples are:
1. (0, 0, 0, 1) -> A[0] + B[0] + C[0] + D[1] = 1 + (-2) + (-1) + 2 = 0
2. (1, 1, 0, 0) -> A[1] + B[1] + C[0] + D[0] = 2 + (-1) + (-1) + 0 = 0
```

Python

```python
class Solution(object):
    def fourSumCount(self, A, B, C, D):
        """
        :type A: List[int]
        :type B: List[int]
        :type C: List[int]
        :type D: List[int]
        :rtype: int
        """
        m = dict()
        for a in A:
            for b in B:
                t = a + b
                if t not in m:
                    m[t] = 0
                m[t] += 1

        ret = 0
        for c in C:
            for d in D:
                t = 0 - (c + d)
                if t in m:
                    ret += m[t]

        return ret
```

Custom Testcase (Contribute)

Run Code | Submit Solution

Top K Frequent Elements

Go to Discuss

Given a non-empty array of integers, return the **k** most frequent elements.

Example 1:

```
Input: nums = [1,1,1,2,2,3], k = 2
Output: [1,2]
```

Example 2:

```
Input: nums = [1], k = 1
Output: [1]
```

Note:

- You may assume *k* is always valid, 1 ≤ *k* ≤ number of unique elements.
- Your algorithm's time complexity **must be** better than O(*n* log *n*), where *n* is the array's size.

Python

```python
class Solution(object):
    def topKFrequent(self, nums, k):
        """
        :type nums: List[int]
        :type k: int
        :rtype: List[int]
        """
        count = collections.Counter(nums)
        return sorted(count.keys(), key = count.get, reverse=True)[:k]
```

Unique Word Abbreviation

Go to Discuss

An abbreviation of a word follows the form <first letter><number><last letter>. Below are some examples of word abbreviations:

```
a) it                      --> it    (no abbreviation)

     1
     ↓
b) d|o|g                   --> d1g

              1    1  1
     1---5----0----5--8
     ↓   ↓    ↓    ↓  ↓
c) i|nternationalizatio|n  --> i18n

              1
     1---5----0
     ↓   ↓    ↓
d) l|ocalizatio|n          --> l10n
```

Assume you have a dictionary and given a word, find whether its abbreviation is unique in the dictionary. A word's abbreviation is unique if no *other* word from the dictionary has the same abbreviation.

Example:

```
Given dictionary = [ "deer", "door", "cake", "card" ]

isUnique("dear") -> false
isUnique("cart") -> true
isUnique("cane") -> false
isUnique("make") -> true
```

Python

```python
class ValidWordAbbr(object):

    def __init__(self, dictionary):
        """
        :type dictionary: List[str]
        """
        self.hash_table = {}
        for d in dictionary:
            if len(d) <= 2:
                continue
            else:
                key = (d[0], len(d) - 2, d[-1])
            if key not in self.hash_table:
                self.hash_table[key] = d
            else:
                self.hash_table[key] = None

    def isUnique(self, word):
        """
        :type word: str
        :rtype: bool
        """
        if len(word) <= 2:
            return True
        else:
            key = (word[0], len(word) - 2, word[-1])
        if key in self.hash_table:
            return self.hash_table[key] == word
        else:
            return True

```

Insert Delete GetRandom O(1)

Go to Discuss

Design a data structure that supports all following operations in *average* **O(1)** time.

1. `insert(val)` : Inserts an item val to the set if not already present.
2. `remove(val)` : Removes an item val from the set if present.
3. `getRandom` : Returns a random element from current set of elements. Each element must have the **same probability** of being returned.

Example:

```
// Init an empty set.
RandomizedSet randomSet = new RandomizedSet();

// Inserts 1 to the set. Returns true as 1 was inserted successfully.
randomSet.insert(1);

// Returns false as 2 does not exist in the set.
randomSet.remove(2);

// Inserts 2 to the set, returns true. Set now contains [1,2].
randomSet.insert(2);

// getRandom should return either 1 or 2 randomly.
randomSet.getRandom();

// Removes 1 from the set, returns true. Set now contains [2].
randomSet.remove(1);

// 2 was already in the set, so return false.
randomSet.insert(2);

// Since 2 is the only number in the set, getRandom always return 2.
randomSet.getRandom();
```

```python
import random
class RandomizedSet(object):

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.hm = {}
        self.l = []

    def insert(self, val):
        """
        Inserts a value to the set. Returns true if the set did not already contain the specified element.
        :type val: int
        :rtype: bool
        """
        if val not in self.hm:
            self.l.append(val)
            self.hm[val] = len(self.l) - 1
            return True
        else:
            return False

    def remove(self, val):
        """
        Removes a value from the set. Returns true if the set contained the specified element.
        :type val: int
        :rtype: bool
        """
        if val not in self.hm:
            return False
        else:
            i = self.hm[val]  # get position of element to be deleted
            self.hm[self.l[-1]] = i
            self.l[i] = self.l[-1]
            self.l.pop()
            self.hm.pop(val, None)
            return True
```

```python
    def getRandom(self):
        """
        Get a random element from the set.
        :rtype: int
        """
        if self.l:
            return random.choice(self.l)
        else:
            return -1
```