

位运算

(Bitwise operations)

本节内容

1. 位运算符
2. 算数移位与逻辑移位
3. 位运算的应用

什么是位运算

程序中的所有数在计算机内存中都是以二进制的形式储存的。位运算说穿了，就是直接对整数在内存中的二进制位进行操作。比如，and运算本来是一个逻辑运算符，但整数与整数之间也可以进行and运算。举个例子，6的二进制是110，11的二进制是1011，那么6 and 11的结果就是2，它是二进制对应位进行逻辑运算的结果（0表示False，1表示True，空位都当0处理）：

```
110 AND 1011 --> 0010(b) --> 2(d)
```

由于位运算直接对内存数据进行操作，不需要转成十进制，因此处理速度非常快。当然有人会说，这个快了有什么用，计算6 and 11没有什么实际意义啊。本文就将告诉你，位运算到底可以干什么，有些什么经典应用，以及如何用位运算优化你的程序。

符号	描述	运算规则
&	与	两个位都为1时，结果才为1
	或	两个位都为0时，结果才为0
^	异或	两个位相同为0，相异为1
~	取反	0变1，1变0
<<	左移	各二进制位全部左移若干位，高位丢弃，低位补0
>>	右移	各二进制位全部右移若干位，对无符号数，高位补0，有符号数，各编译器处理方法不一样，有的补符号位（算术右移），有的补0（逻辑右移）

XOR – 异或

异或：相同为0，不同为1。也可用「不进位加法」来理解。

异或操作的一些特点：

$$x \oplus 0 = x$$

$$x \oplus 1s = \sim x \quad // \quad 1s = \sim 0$$

$$x \oplus (\sim x) = 1s$$

$$x \oplus x = 0 \quad // \text{ interesting and important!}$$

$$a \oplus b = c \Rightarrow a \oplus c = b, \quad b \oplus c = a \quad // \text{ swap}$$

$$a \oplus b \oplus c = a \oplus (b \oplus c) = (a \oplus b) \oplus c \quad // \text{ associative}$$

1. 将 x 最右边的 n 位清零 - $x \& (\sim 0 \ll n)$
2. 获取 x 的第 n 位值(0或者1) - $(x \gg n) \& 1$
3. 获取 x 的第 n 位的幂值 - $x \& (1 \ll (n - 1))$
4. 仅将第 n 位置为 1 - $x \mid (1 \ll n)$
5. 仅将第 n 位置为 0 - $x \& (\sim(1 \ll n))$
6. 将 x 最高位至第 n 位(含)清零 - $x \& ((1 \ll n) - 1)$
7. 将第 n 位至第0位(含)清零 - $x \& (\sim((1 \ll (n + 1)) - 1))$

要点

$X \& 1 == 1$ or $== 0$

$X = X \& (X-1)$ 清零最低位的1

$X \& -X \Rightarrow$ 得到最低位的1

$X \& \sim X \Rightarrow 0$

预习题目（上课考察）

<https://leetcode.com/problems/number-of-1-bits/>

<https://leetcode.com/problems/power-of-two/>

<https://leetcode.com/problems/reverse-bits/>

<https://leetcode.com/problems/n-queens-ii/description/>

实战题目

<https://leetcode.com/problems/n-queens-ii/description/>

```
def totalNQueens(self, n):
    if n < 1: return []
    self.count = 0
    self.DFS(n, 0, 0, 0, 0)
    return self.count

def DFS(self, n, row, cols, pie, na):
    # recursion terminator
    if row >= n:
        self.count += 1
        return

    bits = (~(cols | pie | na)) & ((1 << n) - 1) # 得到当前所有的空位

    while bits:
        p = bits & -bits # 取到最低位的1
        self.DFS(n, row + 1, cols | p, (pie | p) << 1, (na | p) >> 1)
        bits = bits & (bits - 1) # 去掉最低位的1
```

LeetCode 338:

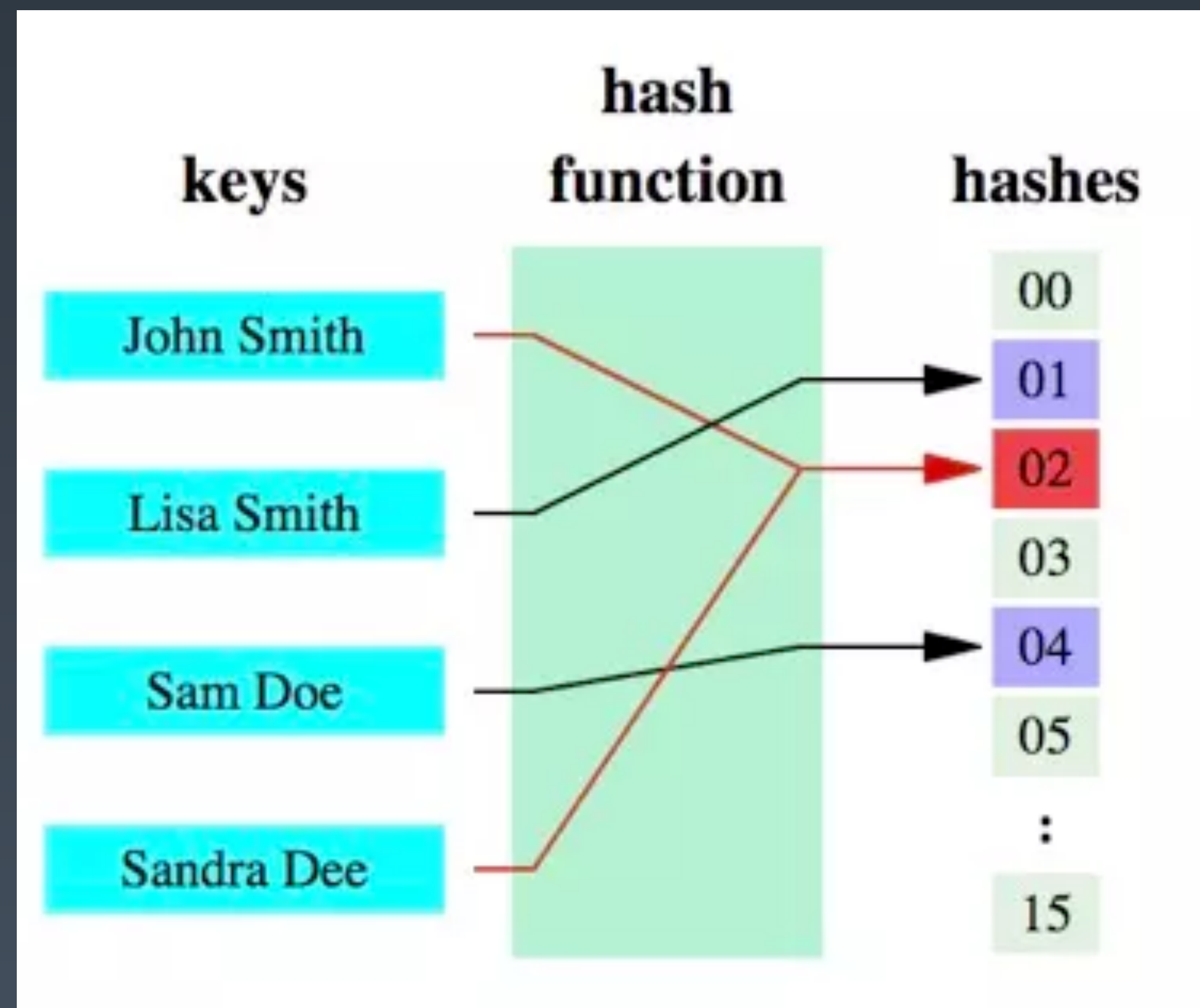
<https://leetcode.com/problems/counting-bits/description/>

```
vector<int> countBits(int num) {  
    vector<int> bits(num+1, 0);  
    for (int i = 1; i <= num; i++) {  
        bits[i] += bits[i & (i - 1)] + 1;  
    }  
    return bits;  
}
```

Bloom Filter

布隆过滤器

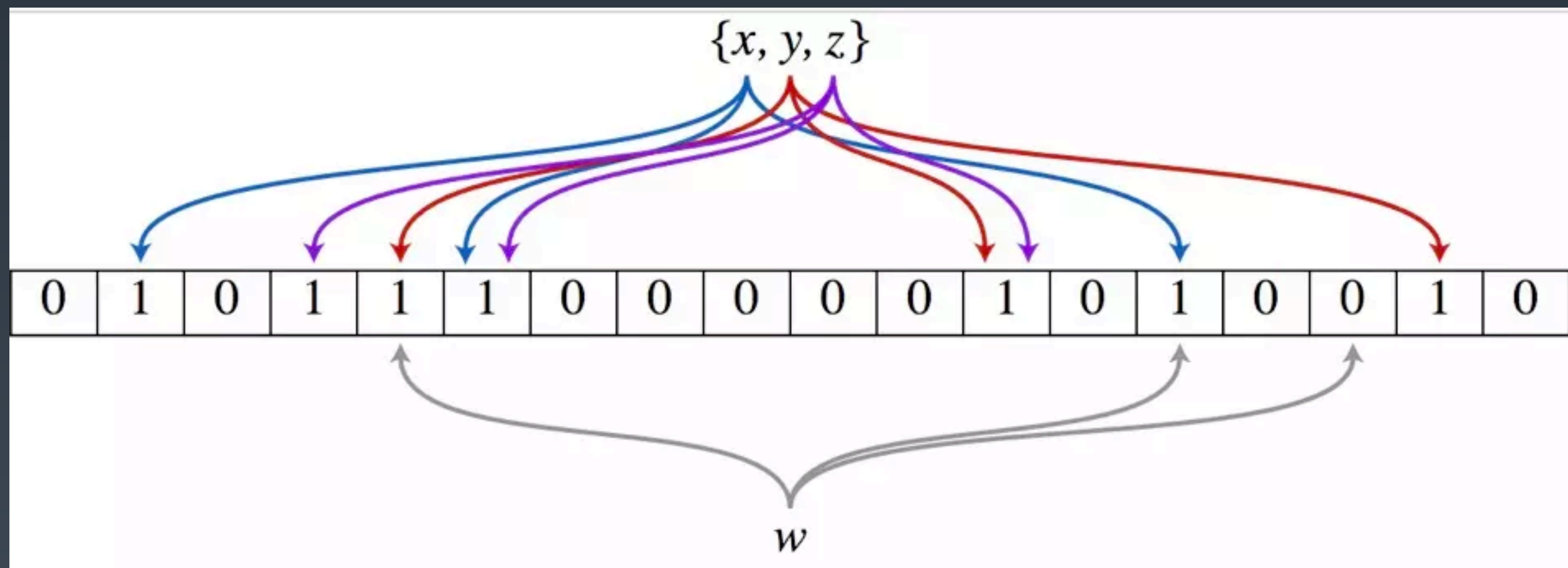
哈希函数

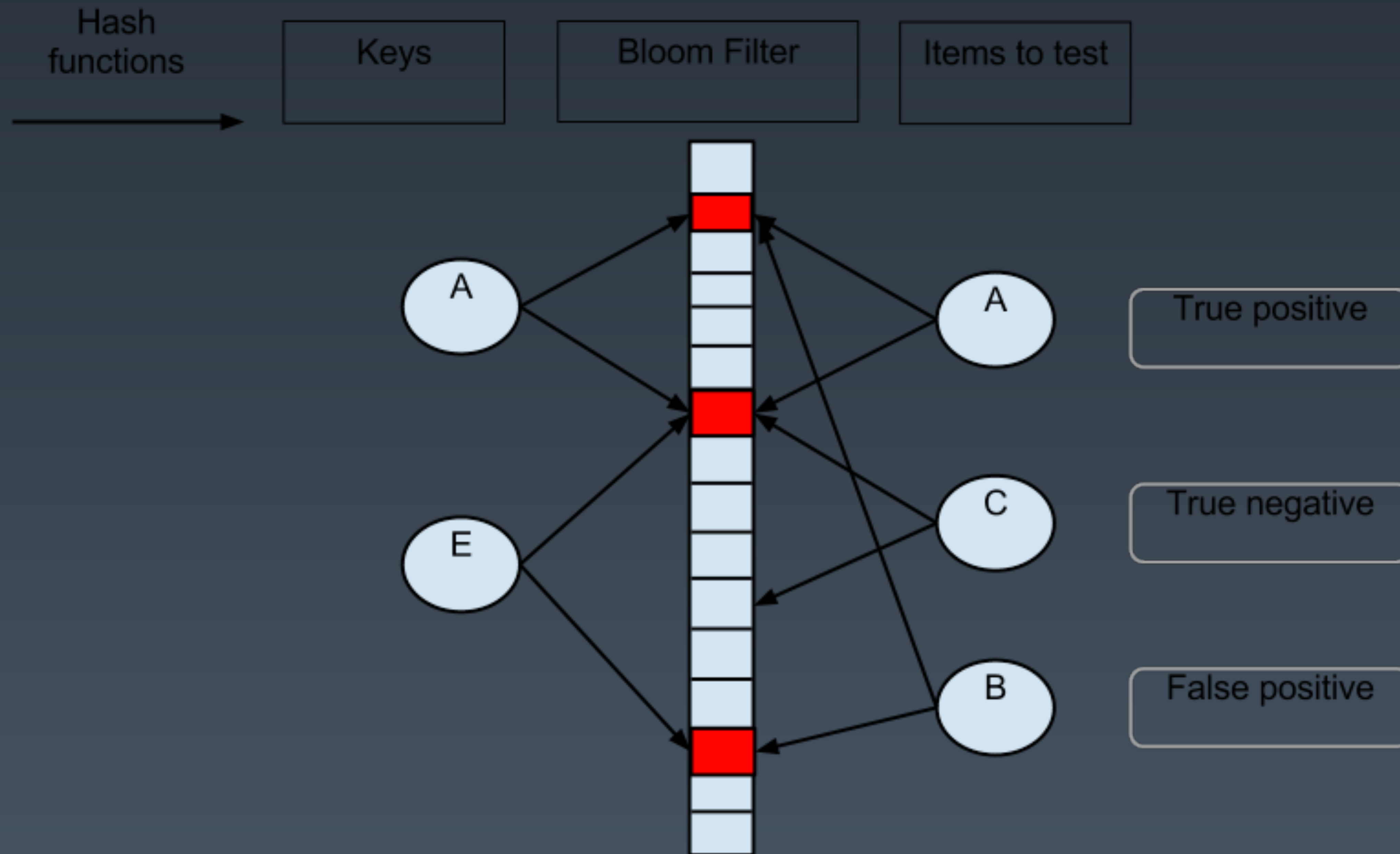


Bloom Filter vs HashTable

一个很长的**二进制**向量和一系列随机**映射函数**。布隆过滤器可以用于检索一个元素是否在一个集合中。

它的优点是空间效率和查询时间都远远超过一般的算法，缺点是有一定的误识别率和删除困难。





案例

1. 比特币网络
2. 分布式系统（Map-Reduce） — Hadoop、search engine

```
1  from bitarray import bitarray
2  import mmh3
3
4  class BloomFilter:
5      def __init__(self, size, hash_num):
6          self.size = size
7          self.hash_num = hash_num
8          self.bit_array = bitarray(size)
9          self.bit_array.setall(0)
10
11     def add(self, s):
12         for seed in range(self.hash_num):
13             result = mmh3.hash(s, seed) % self.size
14             self.bit_array[result] = 1
15
16     def lookup(self, s):
17         for seed in range(self.hash_num):
18             result = mmh3.hash(s, seed) % self.size
19             if self.bit_array[result] == 0:
20                 return "Nope"
21         return "Probably"
22
23 bf = BloomFilter(500000, 7)
24 bf.add("dantezhao")
25 print (bf.lookup("dantezhao"))
26 print (bf.lookup("yyj"))
```

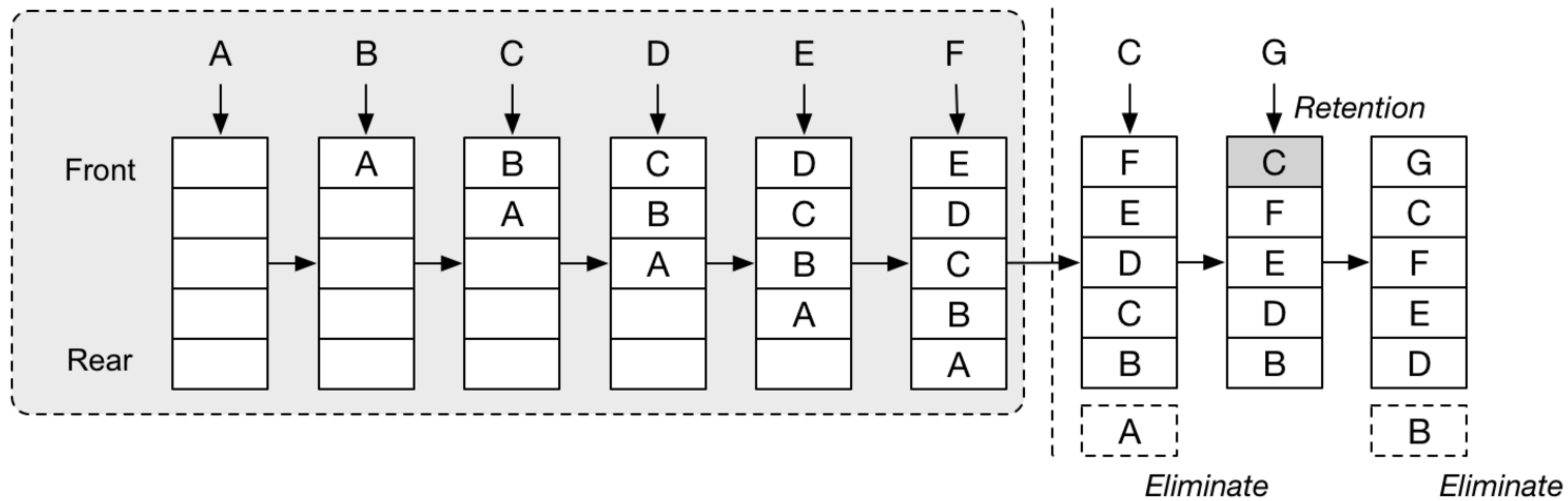
LRU Cache

LRU Cache

Least recently used

Hash Table + Double LinkedList

$O(1)$ get and $O(1)$ set



LFU cache

LFU - least frequently used

LRU - least recently used

etc: https://en.wikipedia.org/wiki/Cache_replacement_policies

练习题

<https://leetcode.com/problems/lru-cache/#/>

```
class LRUCache(object):

    def __init__(self, capacity):
        self.dic = collections.OrderedDict()
        self.remain = capacity

    def get(self, key):
        if key not in self.dic:
            return -1
        v = self.dic.pop(key)
        self.dic[key] = v    # set key as the newest one
        return v

    def put(self, key, value):
        if key in self.dic:
            self.dic.pop(key)
        else:
            if self.remain > 0:
                self.remain -= 1
            else:    # self.dic is full
                self.dic.popitem(last=False)
        self.dic[key] = value
```

Disjoint Set

并查集

适用场景

组团、配对问题

Group or not ?

练习题目

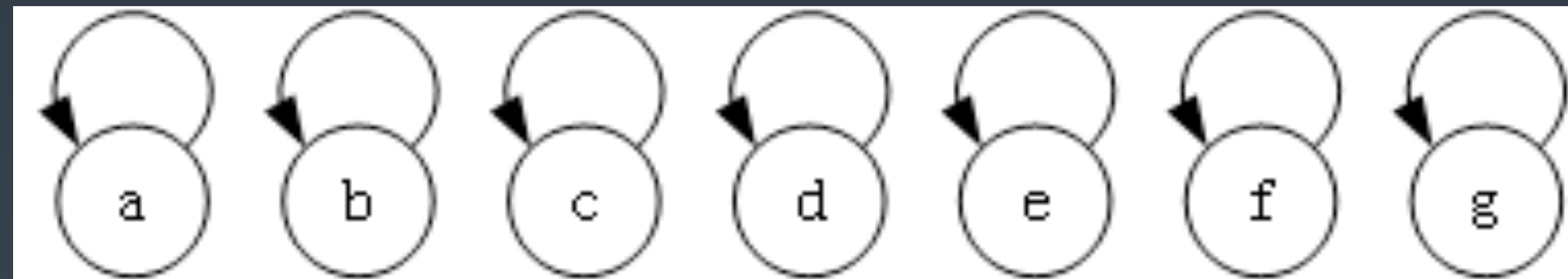
<https://leetcode.com/problems/friend-circles/#/description>

<https://leetcode.com/problems/surrounded-regions/#/description>

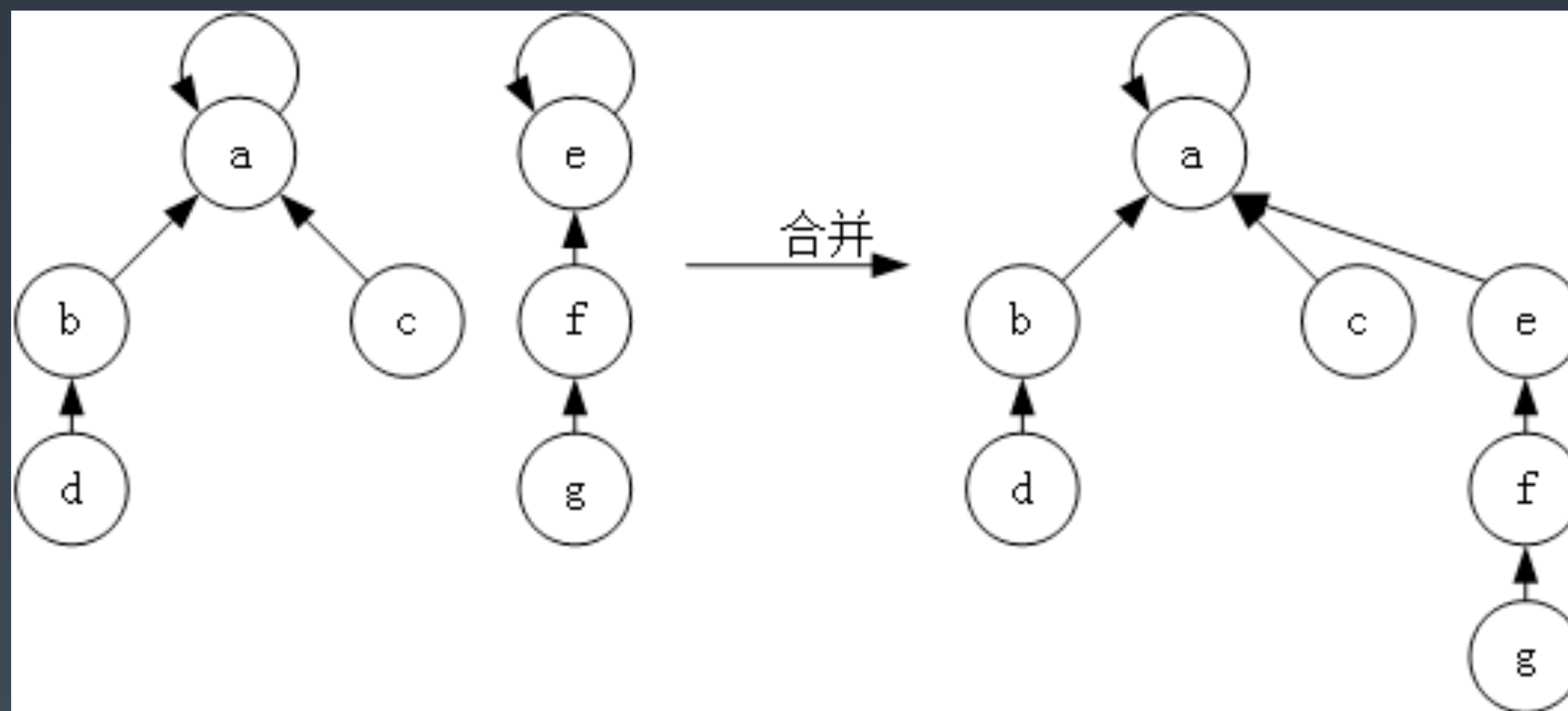
基本操作

- `makeSet(s)`: 建立一个新的并查集, 其中包含 s 个单元素集合。
- `unionSet(x, y)`: 把元素 x 和元素 y 所在的集合合并, 要求 x 和 y 所在的集合不相交, 如果相交则不合并。
- `find(x)`: 找到元素 x 所在的集合的代表, 该操作也可以用于判断两个元素是否位于同一个集合, 只要将它们各自的代表比较一下就可以了。

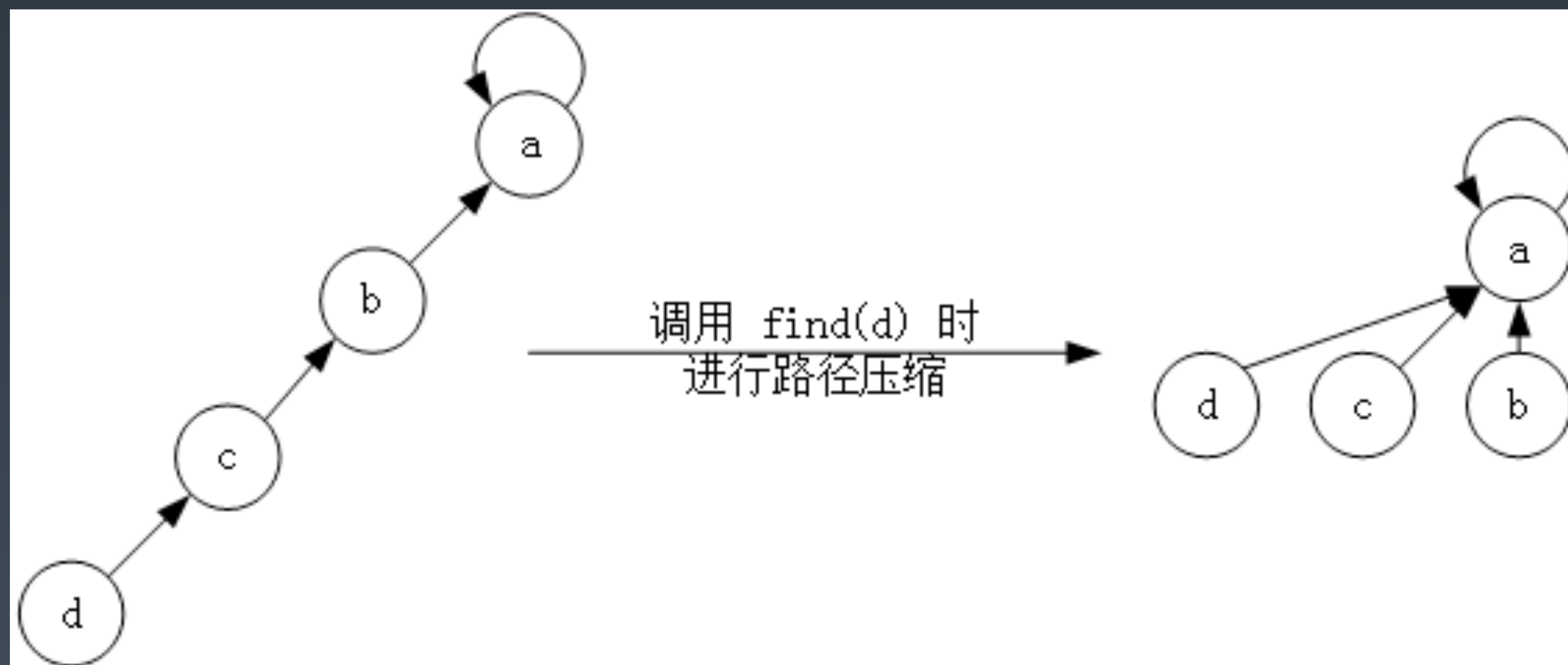
初始化



查询、合并



路径压缩



练习题目

<https://leetcode.com/problems/number-of-islands/>

<https://leetcode.com/problems/surrounded-regions/>

THANKS! |  极客大学