#### 布隆过滤器

#### 引言

在日常生活中，包括在设计计算机软件时，我们经常要判断一个元素是否在一个集合中。比如在字处理软件中，需要检查一个英语单词是否拼写正确（也就是要判断 它是否在已知的字典中）；在 FBI，一个嫌疑人的名字是否已经在嫌疑名单上；在网络爬虫里，一个网址是否被访问过等等。最直接的方法就是将集合中全部的元素存在计算机中，遇到一个新 元素时，将它和集合中的元素直接比较即可。一般来讲，计算机中的集合是用哈希表（hash table）来存储的。它的好处是快速准确，缺点是费存储空间。当集合比较小时，这个问题不显著，但是当集合巨大时，哈希表存储效率低的问题就显现出来 了。

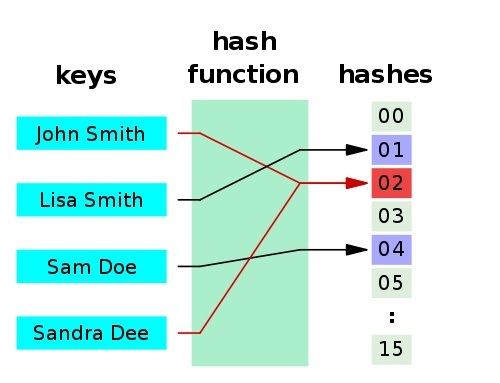
## 哈希 hash

### 原理

Hash （哈希，或者散列）函数在计算机领域，尤其是数据快速查找领域，加密领域用的极广。

**其作用是将一个大的数据集映射到一个小的数据集上面（这些小的数据集叫做哈希值，或者散列值）**。

一个应用是Hash table（散列表，也叫哈希表），是根据哈希值 (Key value) 而直接进行访问的数据结构。也就是说，它通过把哈希值映射到表中一个位置来访问记录，以加快查找的速度。下面是一个典型的 hash 函数 / 表示意图：



**哈希函数有以下两个特点：**

* 如果两个散列值是不相同的（根据同一函数），那么这两个散列值的原始输入也是不相同的。
* 散列函数的输入和输出不是唯一对应关系的，如果两个散列值相同，两个输入值很可能是相同的。但也可能不同，这种情况称为 “散列碰撞”（或者 “散列冲突”）。

### 哈希表的空间效率还是不够高。如果用哈希表存储一亿个垃圾邮件地址，每个email地址 对应 8bytes, 而哈希表的存储效率一般只有50%，因此一个email地址需要占用16bytes. 因此一亿个email地址占用1.6GB，如果存储几十亿个email address则需要上百GB的内存。除非是超级计算机，一般的服务器是无法存储的。

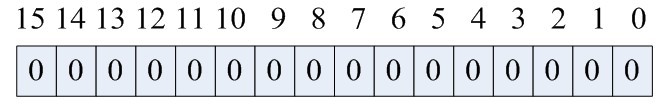
**Bitmap数据结构**

     因为布隆过滤器是位图+哈希表构成的，所以他的基础知识一定要知道，位图是什么？

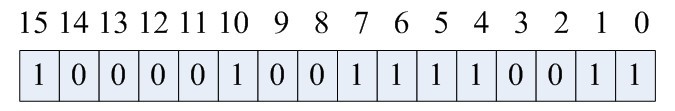
      位图就是bitmap的缩写。所谓bitmap，就是用每一位来存放某种状态，适用于大规模数据，但数据状态又不是很多的情况。通常是用来判断某个数据存不存在的。

 unsigned int bit[N];

在这个数组里面，可以存储 N \* sizeof(int)个数据，但是最大的数只能是N \* sizeof(int) - 1。假如，我们要存储的数据范围为0-15，则我们只需要使得N=1



数据为【5，1，7，15，0，4，6，10】，则存入这个结构中的情况为



【面试题】给40亿个不重复的无符号整数，没排过序。给一个无符号整数，如何快速判断一个数是否在这40亿个数中。

  在看到这个题后最先想到的方法是遍历这40亿个数，依次进行判断，但此做法需要的内存很大，大约为15G（4000000000 \* 4 ÷（1024\*1024\*1024）），可见此算法不可取。

  如果内存够的话，我们可以通过位图实现，位图一个数组每个数据的每个二进制位表示一个数据，每一位用0，1表示当前这个位置上是否存有值，同样是利用哈希存储的方法。此做法可以大大减少内存，对于此题是一个int类型就可以编程32个位，需要的内存空间从15G降到500M。

### Bloom Filter（布隆过滤器）：

### 原理

如果想判断一个元素是不是在一个集合里，一般想到的是将集合中所有元素保存起来，然后通过比较确定。链表、树、散列表（又叫哈希表，Hash table）等等数据结构都是这种思路。但是随着集合中元素的增加，我们需要的存储空间越来越大。同时检索速度也越来越慢。

（Bloom Filter）是由布隆（Burton Howard Bloom）在1970年提出的。它实际上是由一个很长的二进制向量和一系列随机映射函数组成，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（假正例False positives，即Bloom Filter报告某一元素存在于某集合中，但是实际上该元素并不在集合中）和删除困难，但是没有识别错误的情形（即假反例False negatives，如果某个元素确实没有在该集合中，那么Bloom Filter 是不会报告该元素存在于集合中的，所以不会漏报）。

原理要点：一是位数组， 而是k个独立hash函数。

**1）位数组：**

        假设Bloom Filter使用一个m比特的数组来保存信息，初始状态时，Bloom Filter是一个包含m位的位数组，每一位都置为0，即BF整个数组的元素都设置为0。

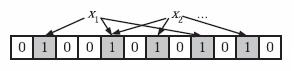
http://my.csdn.net/uploads/201208/12/1344737647_1631.jpg

**2）添加元素，k个独立hash函数**

       为了表达S={x1, x2,…,xn}这样一个n个元素的集合，Bloom Filter使用k个相互独立的哈希函数（Hash Function），它们分别将集合中的每个元素映射到{1,…,m}的范围中。

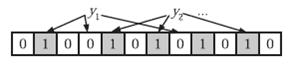
         当我们往Bloom Filter中增加任意一个元素x时候，我们使用k个哈希函数得到k个哈希值，然后将数组中对应的比特位设置为1。即第i个哈希函数映射的位置hashi(x)就会被置为1（1≤i≤k）。

 注意，如果一个位置多次被置为1，那么只有第一次会起作用，后面几次将没有任何效果。在下图中，k=3，且有两个哈希函数选中同一个位置（从左边数第五位，即第二个“1“处）。



**3）判断元素是否存在集合**

    在判断y是否属于这个集合时，我们只需要对y使用k个哈希函数得到k个哈希值，如果所有hashi(y)的位置都是1（1≤i≤k），即k个位置都被设置为1了，那么我们就认为y是集合中的元素，否则就认为y不是集合中的元素。下图中y1就不是集合中的元素（因为y1有一处指向了“0”位）。y2或者属于这个集合，或者刚好是一个false positive。



      显然这 个判断并不保证查找的结果是100%正确的。

例如我们有一个简单的Bloom Filter结构如下：(所有位都是0)

[ 0 0 0 0 0 0 0 0 0 0 ]

第一次插入a用两个哈希函数，映射到1 4位置上，变为1.

[ 1 0 0 1 0 0 0 0 0 0 ]

第二次插入b同样的hash函数，映射到1 8位置上, 变为1.

[ 1 0 0 1 0 0 0 1 0 0 ]

这样就存放了a b两个元素，当我们查找a是否在的时候，两次hash找到1 4位置，发现同时为 1。则表明a存在。

但是假如我们查找的d哈希后映射到4 8位置，发现也同时为 1. 认为存在，这就出错了，因为现在里面只存放了a b没有d。

Bloom Filter 是一种空间效率很高的随机数据结构，Bloom filter 可以看做是对 bit-map 的扩展, 它的原理是：

当一个元素被加入集合时，通过 K 个 Hash 函数 将这个元素映射成一个 位阵列（Bit array）中的 K 个点 ，把它们置为 1 。检索时，我们只要看看这些点是不是都是 1 就（大约）知道集合中有没有它了：

* 如果这些点有任何一个 0，则被检索元素 **一定不在** ；
* 如果都是 1，则被检索元素 **很可能** 在。

### 优点

它的优点是 空间效率 和 查询时间 都远远超过一般的算法，布隆过滤器存储空间和插入 / 查询时间都是常数 O(k) 。另外, 散列函数相互之间没有关系，方便由硬件并行实现。布隆过滤器不需要存储元素本身，在某些对保密要求非常严格的场合有优势。

### 缺点

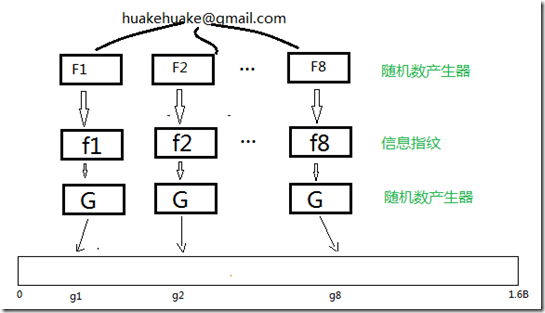
但是布隆过滤器的缺点和优点一样明显。误算率是其中之一。随着存入的元素数量增加， 误算率 随之增加。但是如果元素数量太少，则使用散列表足矣。

另外，一般情况下不能从布隆过滤器中 删除 元素. 我们很容易想到把位数组变成整数数组，每插入一个元素相应的计数器加 1, 这样删除元素时将计数器减掉就可以了。然而要保证安全地删除元素并非如此简单。首先我们必须保证删除的元素的确在布隆过滤器里面. 这一点单凭这个过滤器是无法保证的。另外计数器回绕也会造成问题。

### 例子

**1）检测垃圾邮件**

假定我们存储一亿个电子邮件地址，我们先建立一个十六亿二进制（比特），即两亿字节的向量，然后将这十六亿个二进制全部设置为零。对于每一个电子邮件地址 X，我们用八个不同的随机数产生器（F1,F2, ...,F8） 产生八个信息指纹（f1, f2, ..., f8）。再用一个随机数产生器 G 把这八个信息指纹映射到 1 到十六亿中的八个自然数 g1, g2, ...,g8。现在我们把这八个位置的二进制全部设置为一。当我们对这一亿个 email 地址都进行这样的处理后。一个针对这些 email 地址的布隆过滤器就建成了。



现在，让我们看看如何用布隆过滤器来检测一个可疑的电子邮件地址 Y 是否在黑名单中。我们用相同的八个随机数产生器（F1, F2, ..., F8）对这个地址产生八个信息指纹 s1,s2,...,s8，然后将这八个指纹对应到布隆过滤器的八个二进制位，分别是 t1,t2,...,t8。如果 Y 在黑名单中，显然，t1,t2,..,t8 对应的八个二进制一定是一。这样在遇到任何在黑名单中的电子邮件地址，我们都能准确地发现。

**2）A,B 两个文件，各存放 50 亿条 URL，每条 URL 占用 64 字节，内存限制是 4G，让你找出 A,B 文件共同的 URL。**

分析 ：如果允许有一定的错误率，可以使用 Bloom filter，4G 内存大概可以表示 340 亿 bit。将其中一个文件中的 url 使用 Bloom filter 映射为这 340 亿 bit，然后挨个读取另外一个文件的 url，检查是否与 Bloom filter，如果是，那么该 url 应该是共同的 url（注意会有一定的错误率）。

**3）布隆过滤器既然有错误率，为什么还能应用在key-value系统中？难道key-value系统允许出错？**

bloom filter的特点是会出现误报，但不会漏报，也就是说对于bloom filter验证的一个数据文件，可能不包含你查找的数据项，但是包含你查找的数据项的数据文件它一定是会返回的，[key-value](https://www.baidu.com/s?wd=key-value&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d9uWK-uhNhuyFWnWf1uhRd0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3En101PWb4nj6srjn3nWR1nHmY)系统中bloom filter返回的数据文件还是需要查看里面的内容才能知道是否存在所需的数据的，这就保证了执行结果的正确性和完整性。因此[key-value](https://www.baidu.com/s?wd=key-value&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d9uWK-uhNhuyFWnWf1uhRd0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3En101PWb4nj6srjn3nWR1nHmY)系统不会因此而出错的，只是多访问一些数据文件而已。在数据量很大[key-value](https://www.baidu.com/s?wd=key-value&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d9uWK-uhNhuyFWnWf1uhRd0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3En101PWb4nj6srjn3nWR1nHmY)系统中，建立统一的B+树索引的代价是非常大的，维护成本也很高，因此综合起来bloom filter的性能是最好的

**4)移动公司需要对已经发放的所有139段的号码进行统计排序，已经发放的139号码段的文件都存放在一个文本文件中（原题是放在两个文件中），一个号码一行，现在需要将文件里的所有号码进行排序，并写入到一个新的文件中；号码可能会有很多，最多可能有一亿个不同的号码（所有的139段号码），存入文本文件中大概要占1.2G的空间；JVM最大的内存在300以内，程序要考虑程序的可执行性及效率；只能使用Java标准库，不得使用第三方工具。**

用位向量存储电话号码，1个号码占1个bit，1亿个电话号码也只需要大概12M的空间；算法大概如下：

1、初始化bits[capacity]；

2、顺序所有读入电话号码，并转换为int类型，修改位向量值：bits[phoneNum]=1；

3、遍历bits数组，如果bits[index]=1，转换index为电话号码输出。

**5) 情景1：对无重复的数据进行排序**

给定数据（2，4，1，12，9，7，6）如何对它排序？

     方法1：基本的排序方法包括冒泡，快排等。

     方法2：使用BitMap算法

     方法1就不介绍了，方法2中所谓的BitMap是一个位数组，跟平时使用的数组的唯一差别在于操作的是位。

首先是开辟2个字节大小的位数组，长度为16（该长度由上述数据中最大的数字12决定的）如图

http://dl.iteye.com/upload/attachment/0067/2186/dacb3480-a9ac-3fe1-8d95-7d83832da843.png  
   
      然后，读取数据，2存放在位数组中下标为1的地方，值从0改为1，4存放在下标为3的地方，值从0改为1....结果如图

http://dl.iteye.com/upload/attachment/0067/2190/989ff8b1-5e95-3eb7-8dab-6376a14c08f5.png

      最后，读取该位数组，得到排好序的数据是：（1，2，4，6，7，9，12）

      比较方法1和方法2的差别：方法2中，排序需要的时间复杂度和空间复杂度很依赖与数据中最大的数字比如12，因此空间上讲需要开2个字节大小的内存，时间上需要遍历完整个数组。当数据类似（1，1000，10万）只有3个数据的时候，显然用方法2，时间复杂度和空间复杂度相当大，但是当数据比较密集时该方法就会显示出来优势。

**情景2：对有重复的数据进行判重**

   数据（2，4，1，12，2，9，7，6，1，4）如何找出重复出现的数字？

       首先是开辟2个字节大小的位数组，长度为16（该长度由上述数据中最大的数字12决定的）如图

http://dl.iteye.com/upload/attachment/0067/2186/dacb3480-a9ac-3fe1-8d95-7d83832da843.png

      当读取完12后，数组中的数据如下图：

http://dl.iteye.com/upload/attachment/0067/2197/818d3221-2acb-3a39-bec9-a455452135a8.png

      当读取2的时候，发现数组中的值是1，则判断出2是重复出现的。

**6）假设要你写一个网络蜘蛛（web crawler）。由于网络间的链接错综复杂，蜘蛛在网络间爬行很可能会形成“环”。为了避免形成“环”，就需要知道蜘蛛已经访问过那些URL。给一个URL，怎样知道蜘蛛是否已经访问过呢？稍微想想，就会有如下几种方案：**  
  
　　1. 将访问过的URL保存到数据库。  
  
　　2. 用HashSet将访问过的URL保存起来。那只需接近O(1)的代价就可以查到一个URL是否被访问过了。  
  
　　3. URL经过MD5或SHA-1等单向哈希后再保存到HashSet或数据库。  
  
　　4. Bit-Map方法。建立一个BitSet，将每个URL经过一个哈希函数映射到某一位。  
  
　　方法1~3都是将访问过的URL完整保存，方法4则只标记URL的一个映射位。  
  
   
  
　　以上方法在数据量较小的情况下都能完美解决问题，但是当数据量变得非常庞大时问题就来了。  
  
　　方法1的缺点：数据量变得非常庞大后关系型数据库查询的效率会变得很低。而且每来一个URL就启动一次数据库查询是不是太小题大做了？  
  
　　方法2的缺点：太消耗内存。随着URL的增多，占用的内存会越来越多。就算只有1亿个URL，每个URL只算50个字符，就需要5GB内存。  
  
　　方法3：由于字符串经过MD5处理后的信息摘要长度只有128Bit，SHA-1处理后也只有160Bit，因此方法3比方法2节省了好几倍的内存。  
  
　　方法4消耗内存是相对较少的，但缺点是单一哈希函数发生冲突的概率太高。还记得数据结构课上学过的Hash表冲突的各种解决方法么？若要降低冲突发生的概率到1%，就要将BitSet的长度设置为URL个数的100倍。  
  
　　实质上上面的算法都忽略了一个重要的隐含条件：允许小概率的出错，不一定要100%准确！也就是说少量url实际上没有没网络蜘蛛访问，而将它们错判为已访问的代价是很小的——大不了少抓几个网页呗。其实上面方法4的思想已经很接近Bloom Filter了。方法四的致命缺点是冲突概率高，为了降低冲突的概念，Bloom Filter使用了多个哈希函数，而不是一个。

**布隆过滤器的简易版本实现：**

#include<iostream>

#include<bitset>

#include<string>

#define MAX 2<<24

using namespace std;

bitset<MAX> bloomSet; //简化了由n和p生成m的过程

int seeds[7]={3, 7, 11, 13, 31, 37, 61}; //使用7个hash函数

int getHashValue(string str,int n) //计算Hash值

{

int result=0;

int i;

for(i=0;i<str.size();i++)

{

result=seeds[n]\*result+(int)str[i];

if(result > 2<<24)

result%=2<<24;

}

return result;

}

bool isInBloomSet(string str) //判断是否在布隆过滤器中

{

int i;

for(i=0;i<7;i++)

{

int hash=getHashValue(str,i);

if(bloomSet[hash]==0)

return false;

}

return true;

}

void addToBloomSet(string str) //添加元素到布隆过滤器

{

int i;

for(i=0;i<7;i++)

{

int hash=getHashValue(str,i);

bloomSet.set(hash,1);

}

}

void initBloomSet() //初始化布隆过滤器

{

addToBloomSet("http://www.baidu.com");

addToBloomSet("http://www.cnblogs.com");

addToBloomSet("http://www.google.com");

}

int main(int argc, char \*argv[])

{

int n;

initBloomSet();

while(scanf("%d",&n)==1)

{

string str;

while(n--)

{

cin>>str;

if(isInBloomSet(str))

cout<<"yes"<<endl;

else

cout<<"no"<<endl;

}

}

return 0;

}