随机算法课程

实验报告

实验三：BloomFilter 的抽象数据类型实现

姓名：袁野

学号：1190200122

班级：1903102

评分表：（由老师填写）

最终得分：	
对实验题目的理解是否透彻：	
实验步骤是否完整、可信 ：	
代码质量 ：	
实验报告是否规范 ：	
趣味性、难度加分 ：	
特 色：	1
	2
	3

一、实验题目概述

BloomFilter 是一种随机化的数据结构，可以查找某元素是否在集合内，可以随时向集合内添加元素,而升级版的BloomFilter 可以实现元素的删除操作，而通过设置合适的参数可以使得其出错的概率极低。

本次实验要求实现一个 BloomFilter，通过观察其效果及参数的设置来观察其性能，并通过与数组实现的方式比较来观察其优势及可靠程度。

二、对实验步骤的详细阐述

1、首先实现一个具有插入、查询、删除的 BloomFilter，基础的 BloomFilter 是不支持删除操作的，因此我实现了 CountBloomFilter，也就是每一维记录出现的次数。我采用的是 C++里的 bitset 实现，每四个 01 位视作一个 count，即 BloomFilter 的每一位可以进行不超过 15 的计数，无极特殊情况，这个计数数量是够用的。在使用是需要传入预计的最大元素量 n 和期望误报率 p，而 BloomFilter 的位数 m 和哈希函数个数 k 通过经验公式计算获得最优值。

2、哈希函数：首先生成两个与 n 互质的数字 a_1 和 a_2 ,以及两个随机数 b_1 和 b_2 ，这样会得到两个哈希函数 $y = a_1x + b_1 \, mod \, n$ 和 $y = a_2x + b_2 \, mod \, n$，通过这两个函数的哈希值 y_1 和 y_2 线性叠加，即第 i 个哈希结果的值为 $y_1 + i \times y_2 \, mod \, m$ 得到。

3、查询操作：首先通过哈希函数得到元素 x 对应的 k 个哈希值，然后一次一次查看 BloomFilter 中对应位数是否都大于 0，如果是，则我们认为该元素在这个集合里，否则认为该元素不在这个集合里。

4、插入操作：首先通过查找操作查找 x 是否在集合里，如果已经存在，则返回 bool 值表示插入失败，否则对于元素 x ，借助刚刚查询操作得到的其 k 个哈希函数的值，在对应的 BloomFilter 位上执行加一操作。

5、删除操作：首先通过查找操作查找 x 是否在集合里，如果不存在，则返回 bool 值表示删除失败，否则对于元素 x ，借助刚刚查询操作得到的其 k 个哈希函数的值，在对应的 BloomFilter 位上执行减一操作。

6、设计一个以数组实现的与 BloomFilter 功能一致的类，查找操作即为遍历数组查看是否有该元素且对应 val 为 1，添加操作为在空位置上插入该值并将 val 设置为 1，删除操作即为，先查找该元素是否在数组中，如果在，则将其 val 值设置为 0。同时借用堆来实现删除后的空余位置回收，保证每次插入元素尽可能靠前。

三、实验数据

1. 实验设置

实验环境：

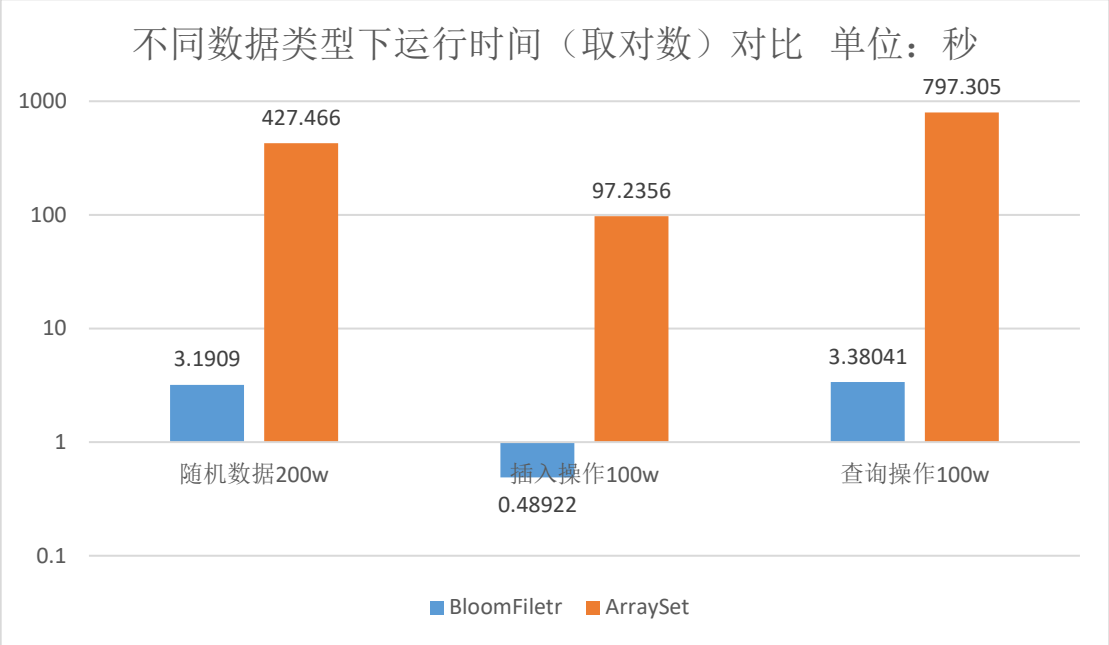
Ubuntu20.04.4 LTS(GNU/Linux 5.10.102.1-microsoft-standard-WSL2 x86_64)

数据：

数据分为两种，第一种我们称之为随机数据，共有 2000000 个具体操作，包含插入、删除、查询，前一百个操作均为插入操作，剩余操作按照 3:2:1 的比例随机生成插入、查询、删除操作，更具体地，每种操作内包含已经合法元素和不合法元素比例为5:1。这组数据可以很好的体现在实际使用上的性能和效率优劣。第二种我们称之为极端数据，也一共包含 2000000 个具体操作，其中前 1000000 个操作为插入操作，剩下的操作为查询操作，合法和不合法元素比例也为 5:1，

这样的数据可以更好展现极限情况下 BloomFilter 的效率和正确率。以上两种数据，包含的预计元素个数为 1000000，假阳率为 0.000742(电脑内存允许下的最小值)。

2. 实验结果



	随机数据	极端数据
总询问数	666506	1000000
假阳率为 0.000742 的期望错误数	494.547	742
实际错误数	2	129
实际错误数/期望错误数	0.4044%	17.385%

四、对实验结果的理解和分析

1、 时间效率上，从实际效果来看，BloomFilter 的时间效率是远远胜过数组的，由于我们维护的是一个集合，因此无论是哪种方式，在插入或者删除之前都需要判断元素是否已经存在于集合中，也就是说都需要执行一次查找操作，因此，数组每次的操作时间复杂度都是 $O(n)$ 的，而 BloomFilter 是 $O(k)$ 的，这就会导致其效率极高。

2、 观察时间表格我们会发现，由于每一次插入之前都会进行查询，等量的查询和插入操作理应插入操作需要的时间长，然而实际上无论是数组还是 BloomFilter，插入操作都比查询操作快不少，这与我们之前的分析截然相反。实际上这是合理的，由于在这一部分实验中，我先插入元素，再查询元素，那么会有以下情况：

a) 对于数组来说，由于开始数组内没有元素或者元素数量不多，而查询时数组内元素个数已经达到上界，因此平均下来插入操作所需要遍历的元素的平均个数要更少。

b) 对于 BloomFilter 来说，我们观察我的查询部分代码：

```
bool find(LL item) {
    hash_youngsc(item);
    bool hasInsterted = true;
    for (int i=0; i<hashNumber && hasInsterted; ++i) {
        if (!count(hashRes[i])) hasInsterted = false;
    }
    return hasInsterted;
}
```

图 1 BloomFilter 查询部分代码截取

我们发现，当得到哈希值之后，我们遍历这些哈希值，当碰到对应位置 count 为 0 时，则认为该值不存在，后边的哈希值便不再查询。而在超过前一半的插入过程中，都能早早地找到一个 count 为 0 的哈希值，因此实际上插入过程中查找所花费的时间很少很少。

因此，虽然二者都是插入时间小于查找时间，但是导致这个现象的原因有所不同，不过归根到本质都是由于我生成数据的方式，导致了实际运行的常数更小。

3、 有关查询正确率的分析，虽然在设置参数是将期望的假阳率设置为 0.000742，但是实际上在随机数据中的假阳率远远低于预期值，虽然由于数据的随机性，导致查询时集合内已存在元素的平均个数要小于 1000000，但是在极端数据中并没有上述的特殊情况，但错误数仍然仅仅只有预计个数的 1/6，因此 FloomFilter 在经验公式的引导下设置参数会达到一个很优秀的水平。

五、实验过程中最值得说起的几个方面

1、 有关m和k值的选取，首先我们需要知道误判率 p 、期望元素个数 n 与他们的关系。我们假设$k \times n < m$ 且哈希函数之间时相互独立的，哈希函数散列的 bit 数组中的位置时完全随机的。

a) 一个长度为 m 的 bit 数组，元素在插入时经过一次哈希散列后 bit 数组的某个位置的值没有被置为 1 的概率为

$$1-\frac{1}{m}$$

经过 k 个哈希函数散列后，还未被置为 1 的概率为

$$\left(1-\frac{1}{m}\right)^{k}$$

插入 n 个元素之后某个位置还未被置为 1 的概率为

$$\left(1-\frac{1}{m}\right)^{nk}$$

所以其被置为 1 的概率为

$$1-\left(1-\frac{1}{m}\right)^{nk}$$

所以一个元素不在集合中，经过 k 个函数散列到 k 个 bit 数组的不同位置且所有这些位置的值为 1 的概率（误判率）为

$$[1-\left(1-\frac{1}{m}\right)^{nk}]^{k}$$

由极限$\lim_{x\to\infty}(1-\frac{1}{x})^{-x}=e$，可得

$$[1-\left(1-\frac{1}{m}\right)^{nk}]^k \approx (1-e^{-\frac{kn}{m}})^k$$

b) 最优哈希函数个数的计算：由(a)中最后得到的公式可得

$$p = exp(kln(1-e^{-\frac{kn}{m}}))$$

令

$$g = kln(1-e^{-\frac{kn}{m}})$$

则当 g 取最小值时，p 取最小值，由于$p = e^{-\frac{kn}{m}}$，我们可以将 g 改写为

$$g = -\frac{m}{n}ln(p)ln(1-p)$$

显然当 $p=\frac{1}{2}$ 时，g 取最小值，此时

$$k = \frac{m}{n}ln2$$

c) 将(a)和(b)结果式子联立，有

$$p = (1-e^{-(\frac{m}{n}ln2)\frac{n}{m}})^{\frac{m}{n}ln2}$$

化简为

$$ln\,p = -\frac{m}{n}ln2^2$$

得

$$m = -\frac{n\,ln\,p}{ln2^2}$$

通过这些公式可以很快地求出最好情况下比特位数和哈希函数个数，通过实验发现，如果比特位数过少，会有比特位计数器溢出得情况发生，而且误判率急剧上升，如果哈希函数过少，同样误判率会上升，哈希函数过多，BloomFilter 的效率会有轻微下降，同时由于置 1 的情况太多，准确率也会下降。

2、有关可删除操作的实现：我在查阅资料后发现两种可以实现带删除的 BloomFilter 的方式，除了我这次试验实现的这种外，还有一种是带缓存的 BloomFilter，实现方式比较简单，建立两个 BloomFilter，其中一个称作 DeleteBloomFilter。当删除元素时，直接将该元素插入 DeleteBloomFilter 即可，查询时，如果查询的元素不在 BloomFilter 中，或者在 BloomFilter 中但是不在 DeleteBloomFilter 中时，我们便认为这个元素不在这个集合中。这种方法简单易行，但是有明显缺点。在实际使用中，我们很难保证一个元素被删除之后再次进入这个集合是一个极低概率的事，相反，这种可能性时存在的，但是带缓存的 BloomFilter 并不支持这种情况判断，当一个元素第一次被删除之后，它将无法再一次被插入，因此我并未采用这种形式的可删除 BloomFilter。

3、关于 countBloomFilter 的实现，将每一位由最先的一个 byte 修改为一个可计数的单位，然而 BloomFilter 的实质是在可接受的空间复杂度下记录元素的特征值，如果用 short 或 int 来实现计数的话显然是无法接受的，因此我们采用压位的方式，然而究竟需要多大的计数值。我们设每一个计数器的最大计数值为 i ，有

$$p(max(c) \geq i) \leq m\binom{nk}{i}\frac{1}{m^i} \leq m(\frac{enk}{im})^i$$

带入最优情况下 $k=\frac{m}{n}ln2$ 有

$$p(max(c) \geq i) \leq m(\frac{eln\,2}{i})^i$$

当$i = 16$时，$p(max(c) \geq 16) \leq 1.37 * 10^{-15} * m$，这是一个足够小的概率，因此，为每一个计数位分配 4 个 byte 就足够使用了，而且在实验过程中，的确没有发现任何一次溢出的情况，如果分配位数过少，首先溢出这件事便会造成加入删除操作后正确率的下降，其次溢出说明了当前映射向这个技术位的哈希函数太多，这本身就是一个在 BloomFilter 中严重影响正确率的现象。