

**2020** 级

《大数据存储与管理》课程

**实 验 报 告**

**姓 名 吴昊磊**

**学 号 U202013777**

**班 号 计算机2011班**

**日 期 2023.05.29**

[一、选题 3](#_Toc101339037)

[二、基本介绍 3](#_Toc101339038)

[2.1 技术背景 3](#_Toc101339039)

[2.2 Cuckoo Filter 3](#_Toc101339040)

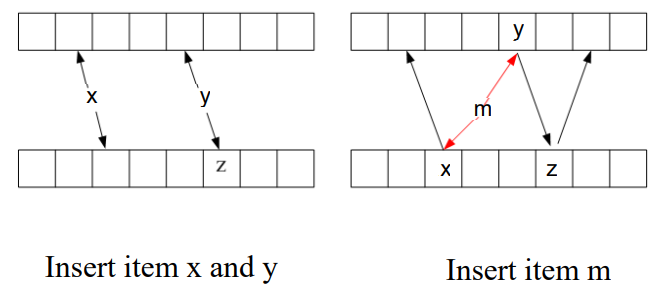
[三、Cuckoo Filter设计与实现 5](#_Toc101339041)

[四 实验心得 9](#_Toc101339042)

# 一、选题

选题3： Cuckoo-driven Way

如何确定循环，减少cuckoo操作中的无限循环的概率和有效存储。



# 二、基本介绍

## 2.1 技术背景

对于海量数据处理业务，我们通常需要一个索引数据结构用来帮助查询，快速判断数据记录是否存在，这种数据结构通常又叫过滤器(filter)。

索引的存储又分为有序和无序，前者使用关联式容器，比如B树，后者使用哈希算法。这两类算法各有优劣：关联式容器时间复杂度稳定O(logN)，且支持范围查询；又比如哈希算法的查询、增删都比较快O(1)，但这是在理想状态下的情形，遇到碰撞严重的情况，哈希算法的时间复杂度会退化到O(n)。因此，选择一个好的哈希算法是很重要的。

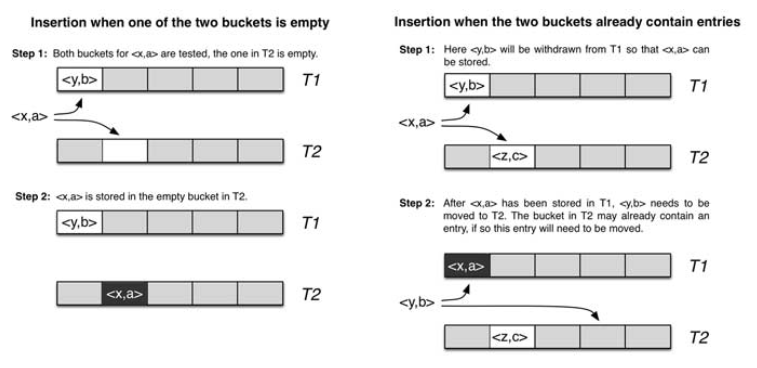
bloom filter的位图模式存在两个问题：一个是误报，在查询时能提供“一定不存在”，但只能提供“可能存在”，因为存在其它元素被映射到部分相同bit位上，导致该位置1，那么一个不存在的元素可能会被误报成存在；另一个是漏报，如果删除了某个元素，导致该映射bit位被置0，那么本来存在的元素会被漏报成不存在。由于后者问题严重得多，所以bloom filter必须确保“definitely no”从而容忍“probably yes”，不允许元素的删除。

为了解决这一问题，引入了一种新的哈希算法——cuckoo filter，它既可以确保元素存在的必然性，又可以在不违背此前提下删除任意元素，仅仅比bitmap牺牲了微量空间效率。

## 2.2 Cuckoo Filter

Cuckoo Hash（布谷鸟散列）是为了解决哈希冲突问题而提出，利用较少的计算换取较大的空间。

它的哈希函数是成对的（具体的实现可以根据需求设计），每一个元素都是两个，分别映射到两个位置，一个是记录的位置，另一个是备用位置。这个备用位置是处理碰撞时用的，这就要说到cuckoo这个名词的典故了，中文名叫布谷鸟，这种鸟有一种即狡猾又贪婪的习性，它不肯自己筑巢，而是把蛋下到别的鸟巢里，而且它的幼鸟又会比别的鸟早出生，布谷幼鸟天生有一种残忍的动作，幼鸟会拼命把未出生的其它鸟蛋挤出窝巢，今后以便独享“养父母”的食物。借助生物学上这一典故，cuckoo hashing处理碰撞的方法，就是把原来占用位置的这个元素踢走，不过被踢出去的元素还要比鸟蛋幸运，因为它还有一个备用位置可以安置，如果备用位置上还有人，再把它踢走，如此往复。直到被踢的次数达到一个上限，才确认哈希表已满，并执行rehash操作。如下图所示：



优化方式：

①将一维改成多维，使用桶（bucket）的4路槽位（slot）；

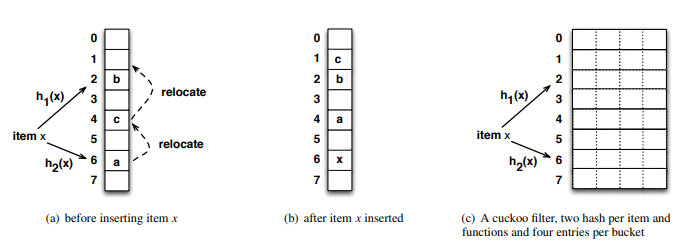
②一个key对应多个value；

③增加哈希函数，从两个增加到多个；

④增加哈希表。

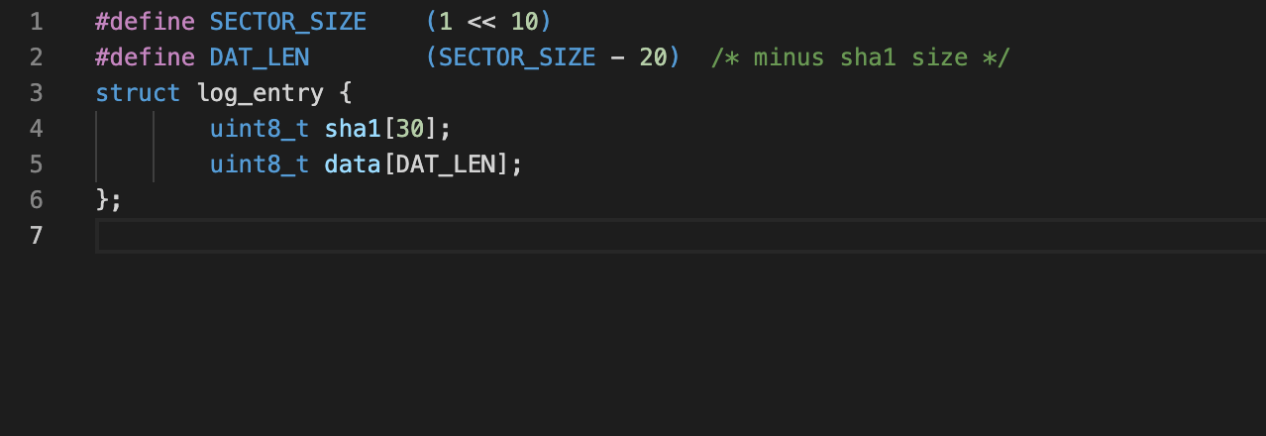
在发生哈希碰撞之前，一维数组的哈希表跟其它哈希函数没什么区别，空间利用率差不多为50%。

一个改进的哈希表如下图所示，每个桶（bucket）有4路槽位（slot）。当哈希函数映射到同一个bucket中，在其它三路slot未被填满之前，是不会有元素被踢的，这大大缓冲了碰撞的几率。采用二维哈希表（4路slot）大约80%的占用率（CMU论文数据据说达到90%以上，应该是扩大了slot关联数目所致）。

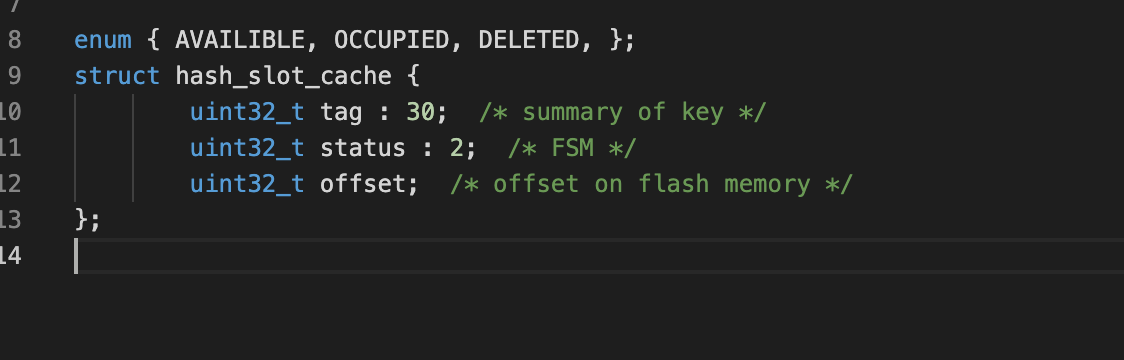


# 三、Cuckoo Filter设计与实现

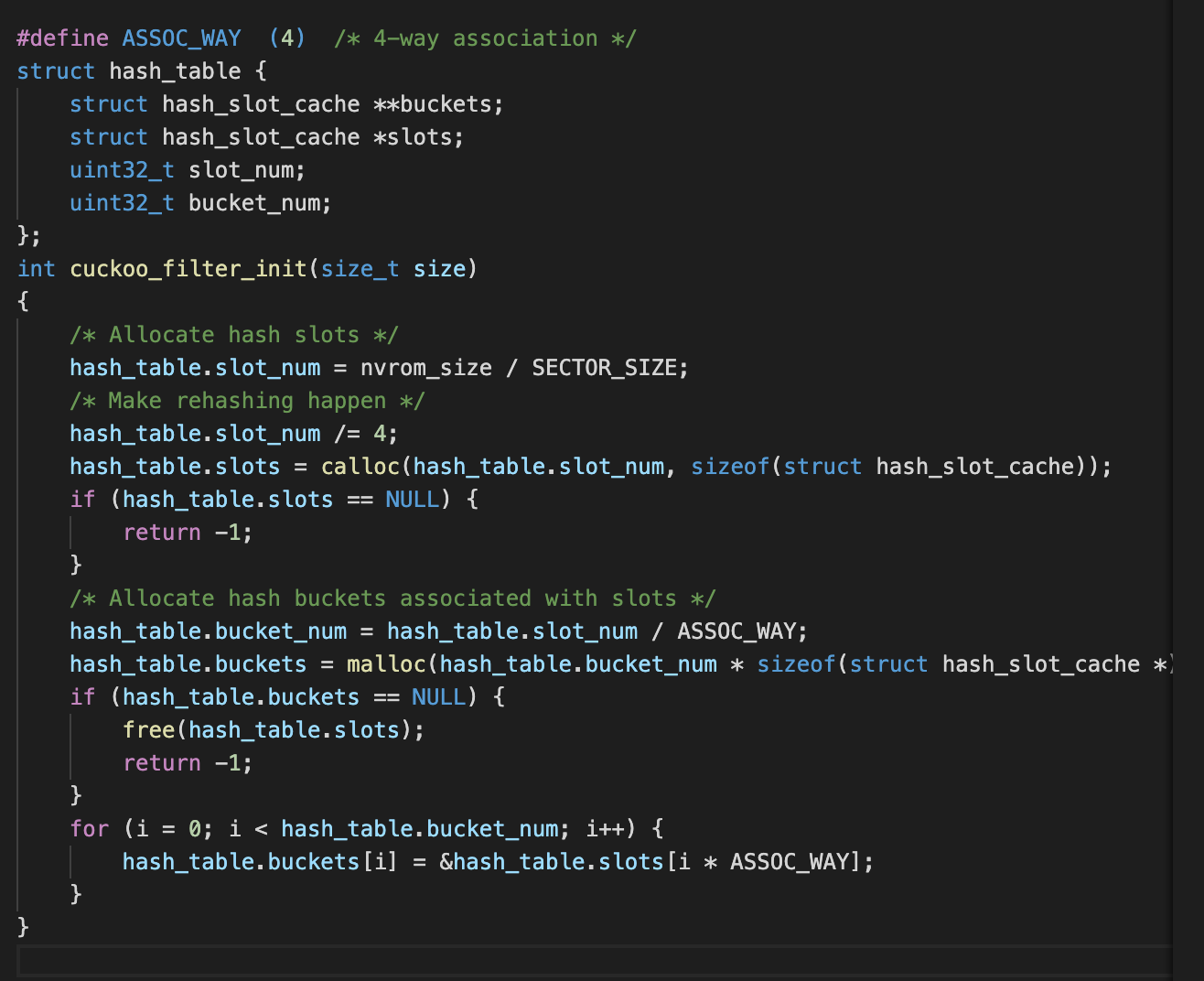
通过cuckoo filter把一段文本数据导入到一个虚拟的flash中，再把它导出到另一个文本文件中。flash存储的单元页面是一个log\_entry，里面包含了一对key/value，value就是文本数据，key就是这段大小的数据的SHA1值。



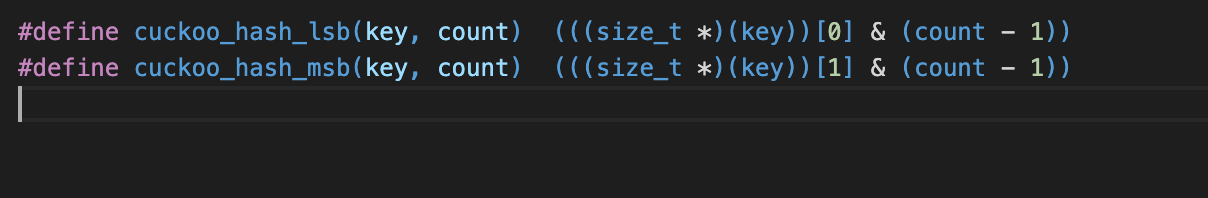
哈希表里的slot有三个成员tag、status和offset，分别是哈希值、状态值和在flash的偏移位置。其中status有三个枚举值：AVAILIBLE、OCCUPIED、DELETED，分别表示这个slot是空闲的，占用的还是被删除的。对于tag，因其中一个哈希值已经对应于bucket的位置上了，所以只要保存另一个备用bucket的位置就行，这样万一被踢，只要用这个tag就可以找到它的另一个的位置。



buckets是一个二级指针，每个bucket指向4个slot大小的缓存，即4路slot，那么bucket\_num也就是slot\_num的1/4。这里把slot\_num调小了点，为的是测试rehash的发生。



下面是哈希函数的设计，这里有两个，前面提到既然key是20字节的SHA1值，我们就可以分别是对key的低32位和高32位进行位运算，只要bucket\_num满足2的幂次方，我们就可以将key的一部分同bucket\_num – 1相与，就可以定位到相应的bucket位置上，注意bucket\_num随着rehash而增大，哈希函数简单的好处是求哈希值很快。

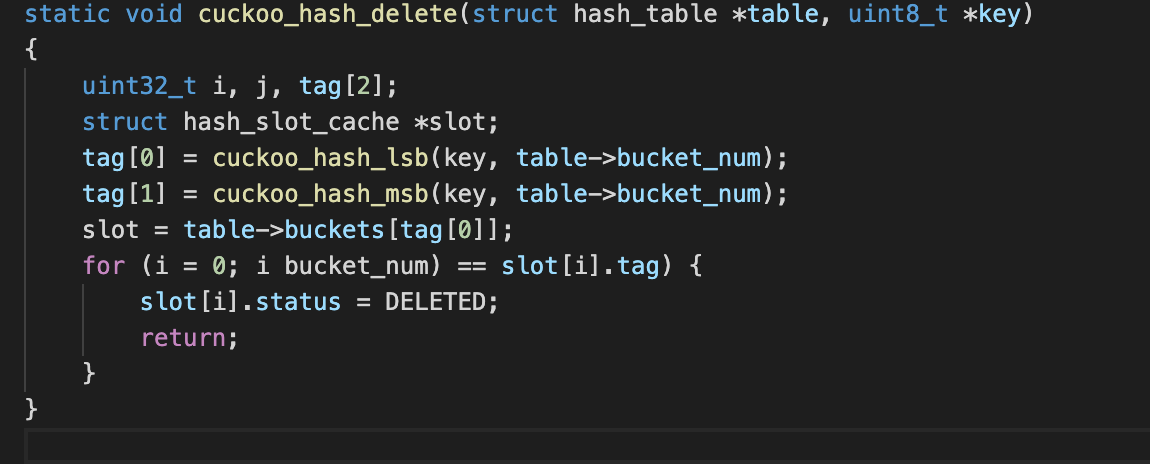


cuckoo filter最重要的三个操作——查询、插入还有删除。

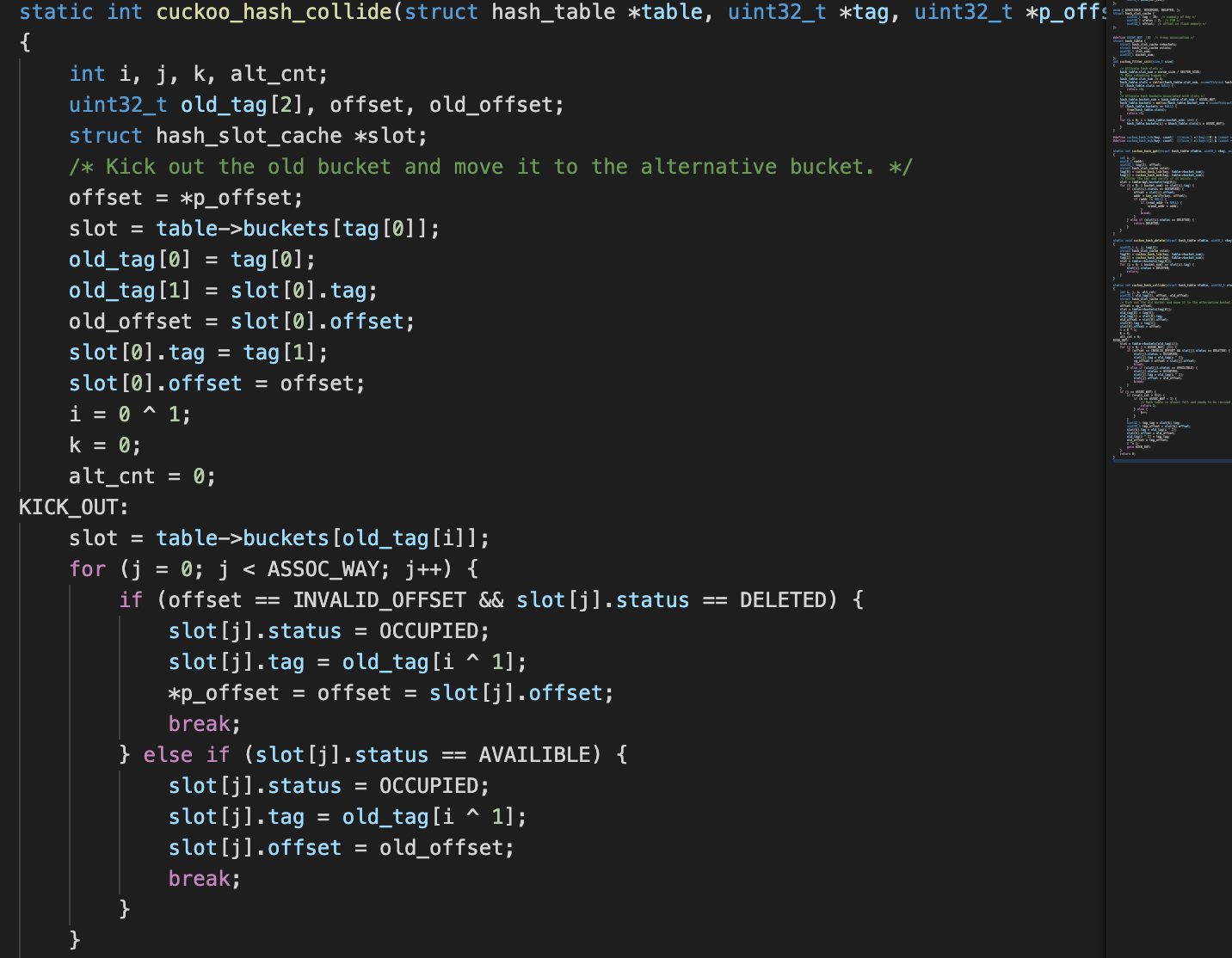
查询操作对传进来的参数key进行两次哈希求值tag[0]和tag[1]，并先用tag[0]定位到bucket的位置，从4路slot中再去对比tag[1]。只有比中了tag后，由于只是key的一部分，再去从flash中验证完整的key，并把数据在flash中的偏移值read\_addr输出返回。相应的，如果bucket[tag[0]]的4路slot都没有比中，再去bucket[tag[1]]中比对，如果还比不中，可以肯定这个key不存在。这种设计的好处就是减少了不必要的flash读操作，每次比对的是内存中的tag而不需要完整的key。

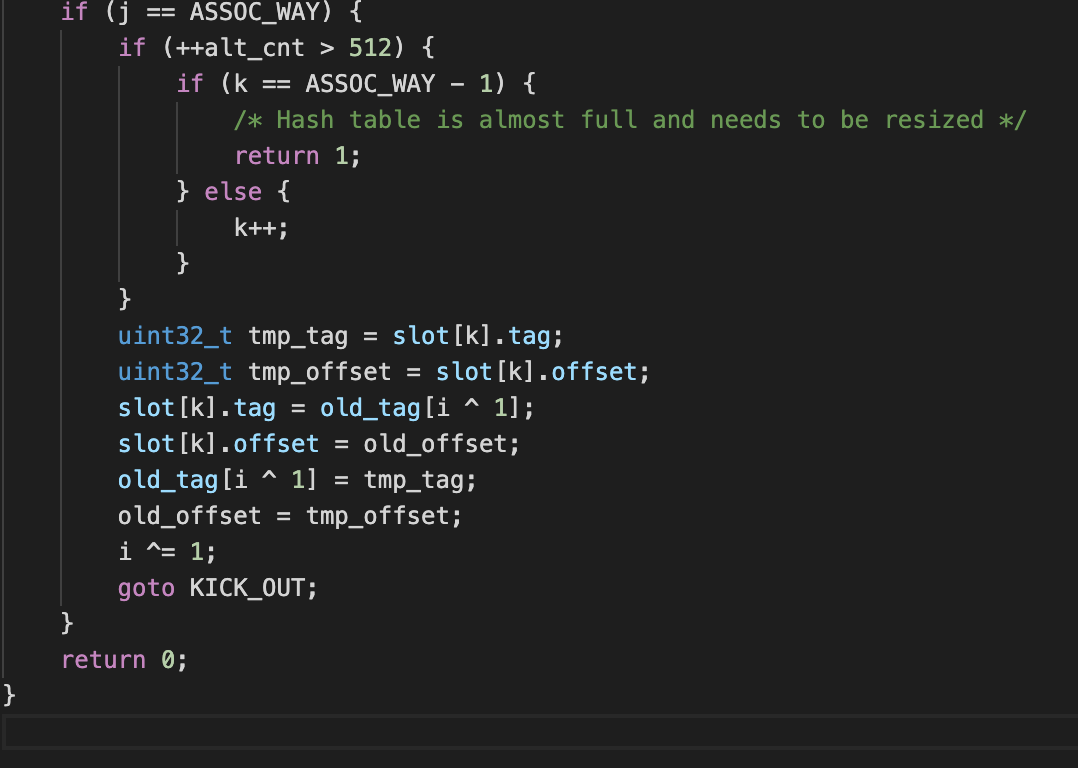


删除操作中的delete仅将相应slot的状态值设置一下，并不会真正的到flash中擦出数据，以免增加设备损耗。

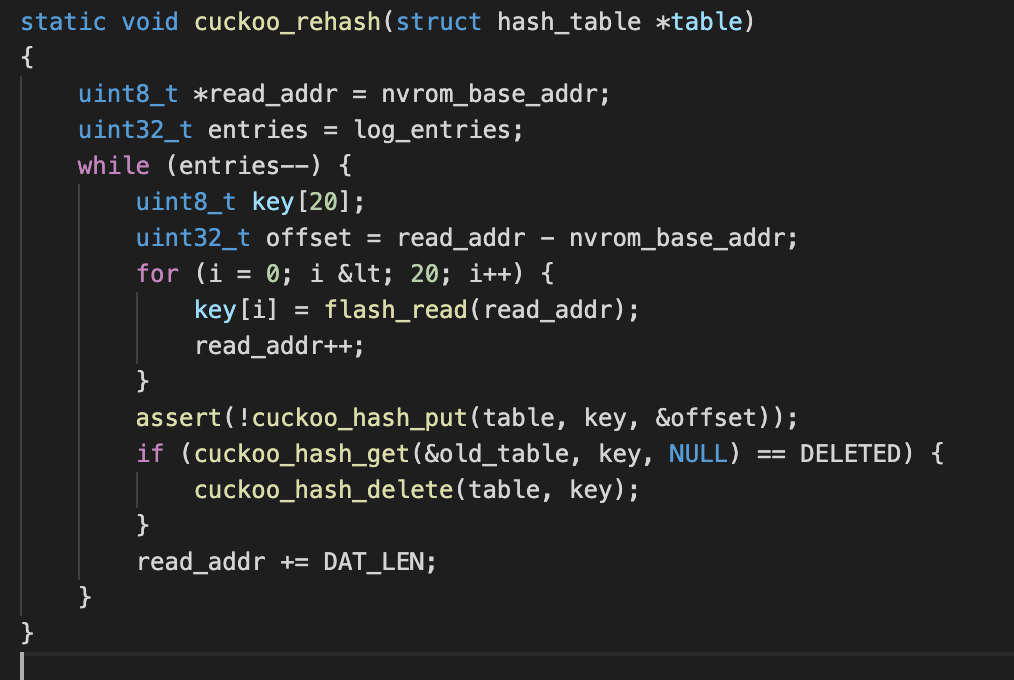


哈希表层面的插入逻辑其实跟查询差不多，不过多说明。这里主要说明如何判断并处理碰撞，用old\_tag和old\_offset保存临时变量，以便一个元素被踢出去之后还能找到备用的位置。这里会有一个判断，每次踢人都会计数，当alt\_cnt大于512时候表示哈希表真的快满了，这时候需要rehash了。





Rehash为将buckets和slots重新realloc，空间扩展一倍，然后再从flash中的key重新插入到新的哈希表里去。需要注意的是，不能有相同的key。



# 四 实验心得

通过这次任务我了解到了什么是cuckoo filter以及他相对于bloom filter的优势，对课程的知识有了更深入的了解，了解了cuckoo filter哈希表中的具体内容，tag bucket和tag的精妙设计支撑起了cuckoo filter。

总的来说，cuckoo filter，它既可以确保该元素存在的必然性，又可以在不违背此前提下删除任意元素，仅仅比bitmap牺牲了微量空间效率，有很大的应用空间

* R. Pagh and F. Rodler, “Cuckoo hashing,” Proc. ESA, pp. 121–133, 2001.
* Yu Hua, Hong Jiang, Dan Feng, "FAST: Near Real-time Searchable Data

Analytics for the Cloud", Proceedings of the International Conference for

High Performance Computing, Networking, Storage and Analysis (SC),

November 2014, Pages: 754-765.

* Yu Hua, Bin Xiao, Xue Liu, "NEST: Locality-aware Approximate Query

Service for Cloud Computing", Proceedings of the 32nd IEEE International

Conference on Computer Communications (INFOCOM), April 2013,

pages: 1327-1335.

* Qiuyu Li, Yu Hua, Wenbo He, Dan Feng, Zhenhua Nie, Yuanyuan Sun,

"Necklace: An Efficient Cuckoo Hashing Scheme for Cloud Storage

Services", Proceedings of IEEE/ACM International Symposium on Quality

of Service (IWQoS), 2014.

* B. Fan, D. G. Andersen, and M. Kaminsky, “MemC3: Compact and

concurrent memcache with dumber caching and smarter hashing,” Proc.

USENIX NSDI, 2013.