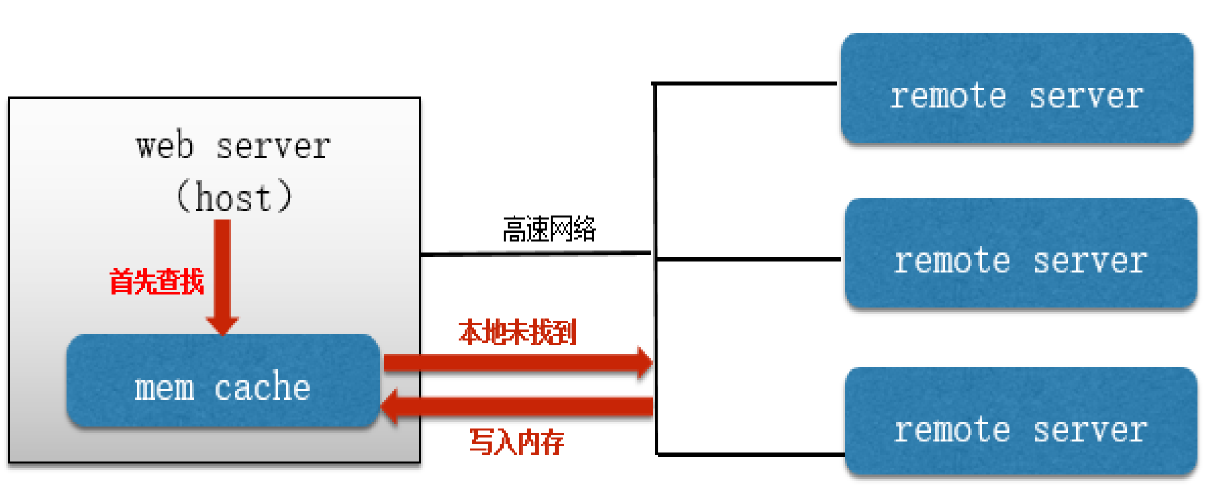
Cuckoo hash

软件学院《数据结构》讲义
内部使用

引例：Memcached 内存缓存



数据库中的商品信息

id (主键)	颜色	尺寸	价格

将一件商品以键值对(key, value)的形式存储在内存里的cuckoo hash中，key和数据库中的主键id一致，能够唯一表示一件商品，value集合了颜色、尺寸、样式这些基本属性。

Cuckoo hash基本思想

Cuckoo hash的基本组成是2个hash 函数和一个hash table，并且两个hash函数会确保将某个键映射至table中的不同位置，也就是说对于任意键k，h1(k)≠h2(k)。一个键仅可能出现在table 中的h1(k)位置或h2(k)位置，这两个位置中的唯一一个。

对比：　线性hash，顺序遍历探测序列

链式hash，需要遍历一次链表

Cuckoo

cuckoo意为布谷鸟，布谷鸟会偷偷的在其它鸟的巢穴中产蛋，当布谷鸟幼崽孵化出来后，这些幼崽便会将其它幼鸟踢出巢穴，以获得更大的生存空间。

Cuckoo hash基本操作

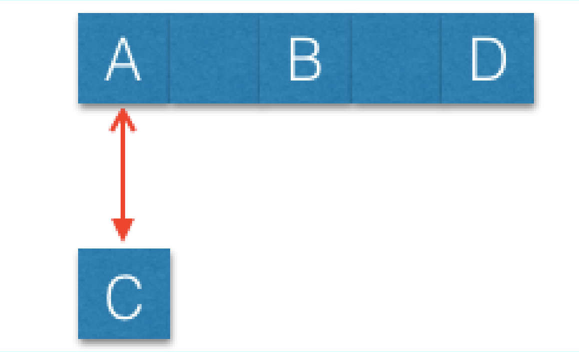
查找(get)操作：在cuckoo hash中，因为一个键仅可能出现在table中的h1(k)位置或者h2(k)位置，所以查找时仅需要探测这两个位置。

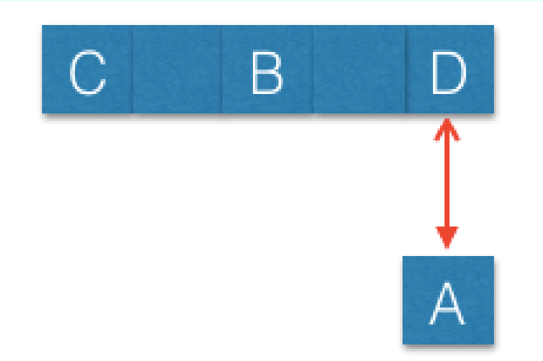
插入(put)操作：和其它hash 方法一样，cuckoo hash避免不了插入时的冲突。对于某个键k，如果h1(k)位置发生冲突，则查看h2(k)位置，为空则将k插入至h2(k)位置，但如果h2(k)非空呢?

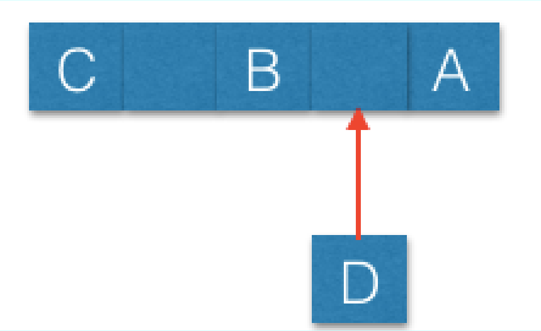
Cuckoo hash基本操作

插入(put)操作：和其它hash 方法一样，cuckoo hash避免不了插入时的冲突。

对于某个键k，如果h1(k)位置发生冲突，则查看h2(k)位置，为空则将k插入至h2(k)位置，但如果h2(k)非空呢？







问题： 最后一个被踢出的元素永远无法找到一个空位置，这样整个踢出过程便无法终止。无法终止的踢出过程都会形成一个环。

键值分离存储

hash table中每一项存储的是<key, address>， address记录了值所在的地址。

```
typedef int KeyType;
class Cuckoo{
protected:
    std::mutex mtx;
    KeyType T[SIZE];
    // hash key by hash func 1
    int hash1(const KeyType &key);
    // hash key by hash func 2
    int hash2(const KeyType &key);
    // find key by hash func 1 in T, exist return key otherwise 0
    KeyType get1(const KeyType &key);
    // find key by hash func 2 in T, exist return key otherwise 0
   KeyType get2(const KeyType &key);
   void bt_evict(const KeyType &key, int which, int pre_pos);
public:
    Cuckoo();
    //~Cuckoo();
    KeyType get(const KeyType &key);
    void put(const KeyType &key);
 };
```

串行Get

```
Cuckoo::Cuckoo(){
    memset(T, 0, sizeof(KeyType) * SIZE);
}
//~Cuckoo();
int Cuckoo::hash1(const KeyType &key){
    assert(SIZE != 0);
    int half_siz = SIZE / 2;
    return key%half_siz;
}
int Cuckoo::hash2(const KeyType &key){
    assert(SIZE != 0);
    int half_siz = SIZE / 2;
    return key/half_siz%half_siz + half_siz;
}
```

```
// find key by hash func 1 in T, exist return key otherwise 0
KeyType Cuckoo::get1(const KeyType &key){
    return (T[hash1(key)] == key)?key:0;
}
// find key by hash func 2 in T, exist return key otherwise 0
KeyType Cuckoo::get2(const KeyType &key){
    return (T[hash2(key)] == key)?key:0;
}
KeyType Cuckoo::get(const KeyType &key){
    // 0 is reserved for null, invalid input
    if(key == 0){
        printf(″invalid key\n″);
        return 0;
    }
    KeyType result = get1(key);
    if(result == 0){
        result = get2(key);
    }
    return result;
}
```

并行Get

共享变量为何没有锁保护?

```
static const int TOTAL = 10;
int main(int argc, char* argv[]){
    Cuckoo test;
    // single-thread to put [1, TOTAL]
    for(int i = 1; i <= TOTAL; ++i){
        test.put(i);
    }
    // create multiple threads to get in parallel
    std::vector<std::thread> threads;
    threads.clear();
    for(int i = 1; i <= TOTAL; ++i){
        threads.emplace_back([&](int thread_id){
                printf("thread: %d get %d\n", thread_id, test.get(thread_id));
        }, i);
    }
    for(int i = 0; i < TOTAL; ++i){
        threads[i].join();
    }
    return 0;
}
```

Put

```
template <typename T>
inline void swap(T* a, T* b){
    assert(a != NULL && b != NULL);
    T tmp = *a;
    *a = *b;
    *b = tmp;
}
void Cuckoo::put(const KeyType &key){
    if(key == 0){
        printf("invalid key\n");
        return;
    }
    if(get(key) != 0){
        printf("duplicate key, put fail\n");
        return;
    }
    // basic way
    if(T[hash1(key)] == 0){
        T[hash1(key)] = key;
    }else if(T[hash2(key)] == 0){
        T[hash2(key)] = key;
    }else{ // two place for one certain key has been occupied, need evict others
        // basic way
        KeyType evicted = key;
        // determine which pos hash1 or hash2 to put key
        // 0 is hash1, 1 is hash2
        int which = 0;
        // first evict key in hash1
        int idx = hash1(evicted);
        // != 0 means place has been occupied
        // if there is a cycle, maybe cannot terminate
        int pre_pos = -1;
        while(T[idx] != 0){
            printf("evicted key %d from %d to %d\n", evicted, pre_pos, idx);
            swap(&evicted, &T[idx]);
            pre_pos = idx;
            which = 1 - which;
            idx = (which == 0)?hash1(evicted):hash2(evicted);
        }
        printf("evicted key %d from %d to %d\n", evicted, pre_pos, idx);
        T[idx] = evicted;
    }
}
```

基于回溯的实现

当n!=1时，程序会不断的向下调用，形成一个没有分叉的递归调用树。

当n==1时，程序从调用树的叶子节点返回计算结果，并且每一层都会向调用层返回自己这一层的计算结果，到达根节点时便会得到最终结果

```
int fac(int n) {
    if(n==1)
        return n;
    else
        return n * fac(n-1);
}
```

基于回溯的实现

假定产生的踢出序列为A->B->C->D->nil

先踢出D，依次向上直到A，我们便可以发现：在保证将某个键插入到指定位置的操作是原子的前提下，就可以确保这些元素始终在hash table里

```
void Cuckoo::bt_evict(const KeyType &key, int which, int pre_pos){
    int idx = (which == 0)?hash1(key):hash2(key);
    // basic case: find a empty pos for the last evicted element
    if(T[idx] == 0){
        printf(″evicted key %d from %d to %d\n″, key, pre_pos, idx);
        T[idx] = key;
        return;
    }
    printf(″evicted key %d from %d to %d\n″, key, pre_pos, idx);
    KeyType cur = T[idx];
    // first evict latter elements
    bt_evict(cur, 1 - which, idx);
    T[idx] = key;
}
```

基于回溯的实现

```
void Cuckoo::put(const KeyType &key){
    if(key == 0){
        printf(″invalid key\n″);
        return;
    }
    if(get(key) != 0){
        printf(″duplicate key, put fail\n″);
        return;
    }
    // basic way
    if(T[hash1(key)] == 0){
        T[hash1(key)] = key;
    }else if(T[hash2(key)] == 0){
        T[hash2(key)] = key;
    }else{ // two place for one certain key has been occupied, need evict others
        // backtrace way
        bt_evict(key, 0, -1);
    }
}
```

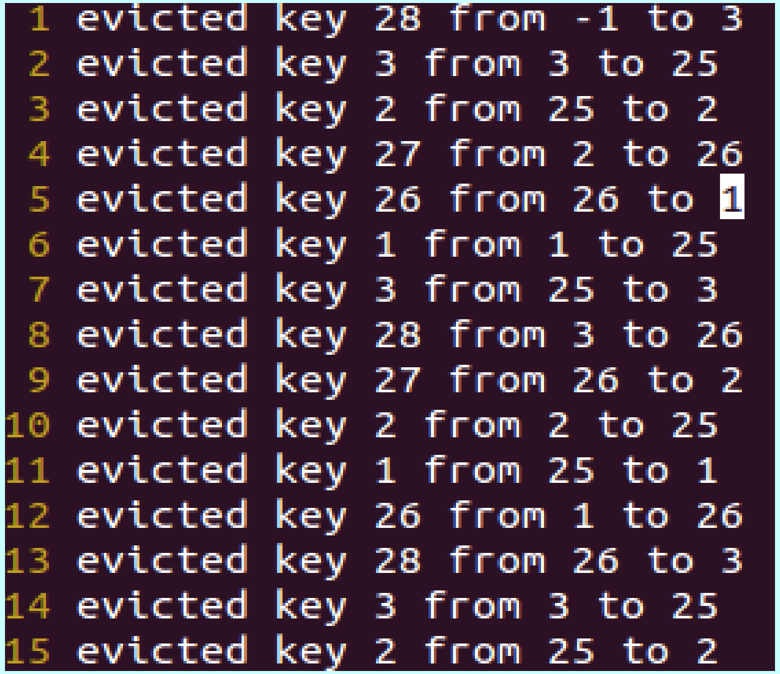
并行Put

```
void Cuckoo::put(const KeyType &key){
    if(key == 0){
        printf("invalid key\n");
        return;
    }
    if(get(key) != 0){
        printf("duplicate key, put fail\n");
        return;
    }
    // basic way
    if(T[hash1(key)] == 0){
        T[hash1(key)] = key;
    }else if(T[hash2(key)] == 0){
        T[hash2(key)] = key;
```

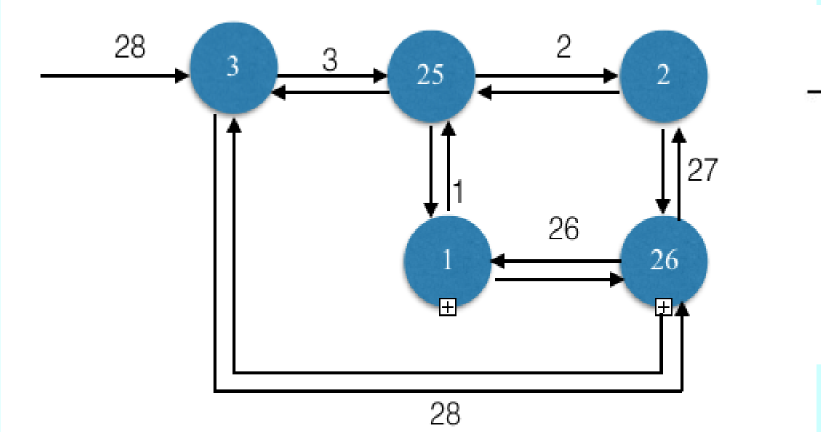
```
}else{ // two place for one certain key has been occupied, need evict others
        // lock way
        // need lock for write-operations
        std::unique_lock<std::mutex> lck(mtx);

        KeyType evicted = key;
        // determine which pos hash1 or hash2 to put key
        // 0 is hash1, 1 is hash2
        int which = 0;
        // first evict key in hash1
        int idx = hash1(evicted);
        // != 0 means place has been occupied
        // if there is a cycle, maybe cannot terminate
        int pre_pos = -1;
        while(T[idx] != 0){
            printf("evicted key %d from %d to %d\n", evicted, pre_pos, idx);
            swap(&evicted, &T[idx]);
            pre_pos = idx;
            which = 1 - which;
            idx = (which == 0)?hash1(evicted):hash2(evicted);
        }
        printf("evicted key %d from %d to %d\n", evicted, pre_pos, idx);
        T[idx] = evicted;
      }
}
```

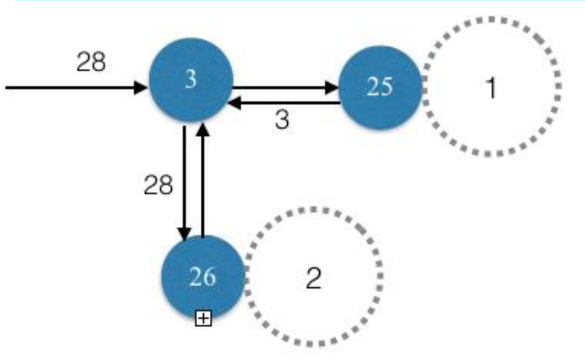
死循环

```
#include <vector>
#include "cuckoo.cpp"
using namespace cuckoo;
static const int TOTAL = 28;
int main(int argc, char* argv[]){
    Cuckoo test;
    // single-thread to put [1, TOTAL]
    for(int i = 1; i <= TOTAL; ++i){
        test.put(i);
    }
    return 0;
}
```

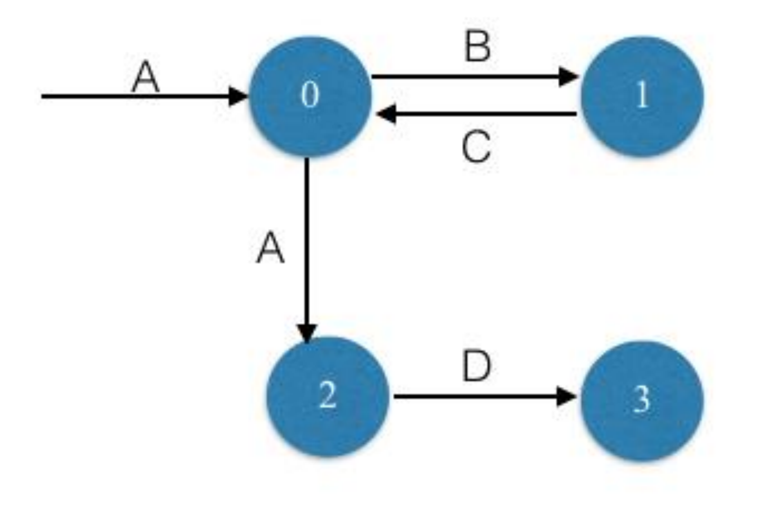


```
evicted key 28 from -1 to 3
evicted key 3 from 3 to 25
evicted key 2 from 25 to 2
evicted key 27 from 2 to 26
evicted key 26 from 26 to 1
evicted key 1 from 1 to 25
evicted key 3 from 25 to 3
evicted key 28 from 3 to 26
evicted key 27 from 26 to 2
evicted key 2 from 2 to 25
evicted key 1 from 25 to 1
evicted key 26 from 1 to 26
evicted key 28 from 26 to 3
evicted key 3 from 3 to 25
evicted key 2 from 25 to 2
```





可以终止的环



只有当一个键的两个可选位置都各自形成一个环结构时，才会导致整个过程无法终止

检测循环路径的方法也比较简单，可以预先设定一个阈值(threshold)，当循环次数或者递归调用次数超过阈值时，就可以认为产生了循环路径。一旦发生循环路径之后，常规方法就是进行rehash操作

New hash functions are chosen, and the whole data structure is **rebuilt** (“rehashed”)

性能分析

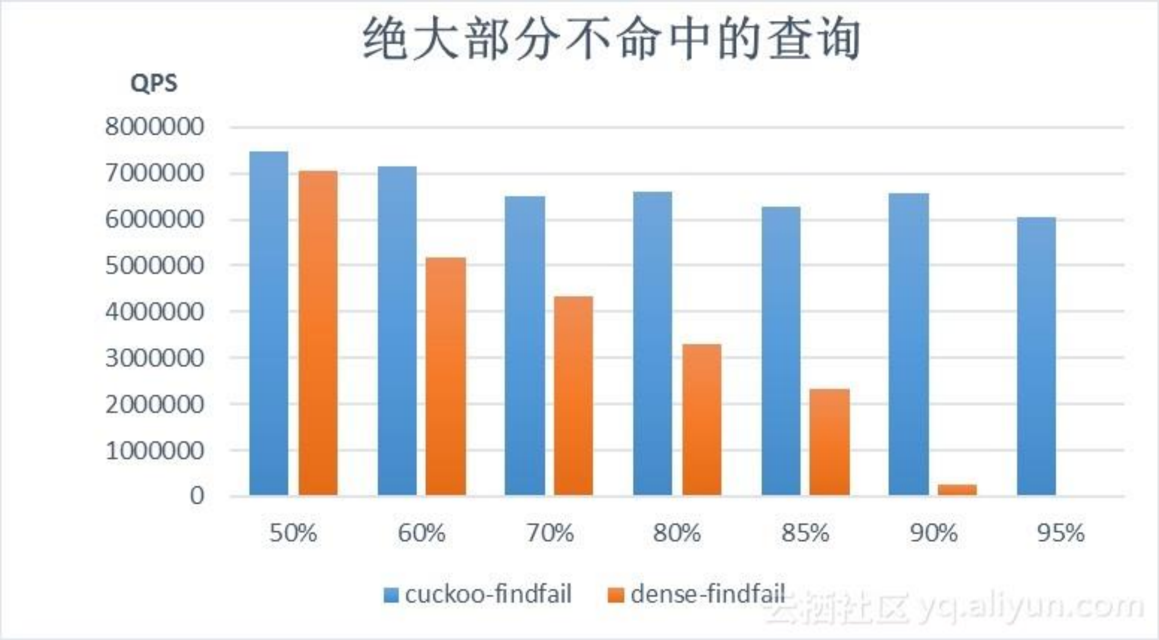
Cuckoo hash的总容量限制为500个键

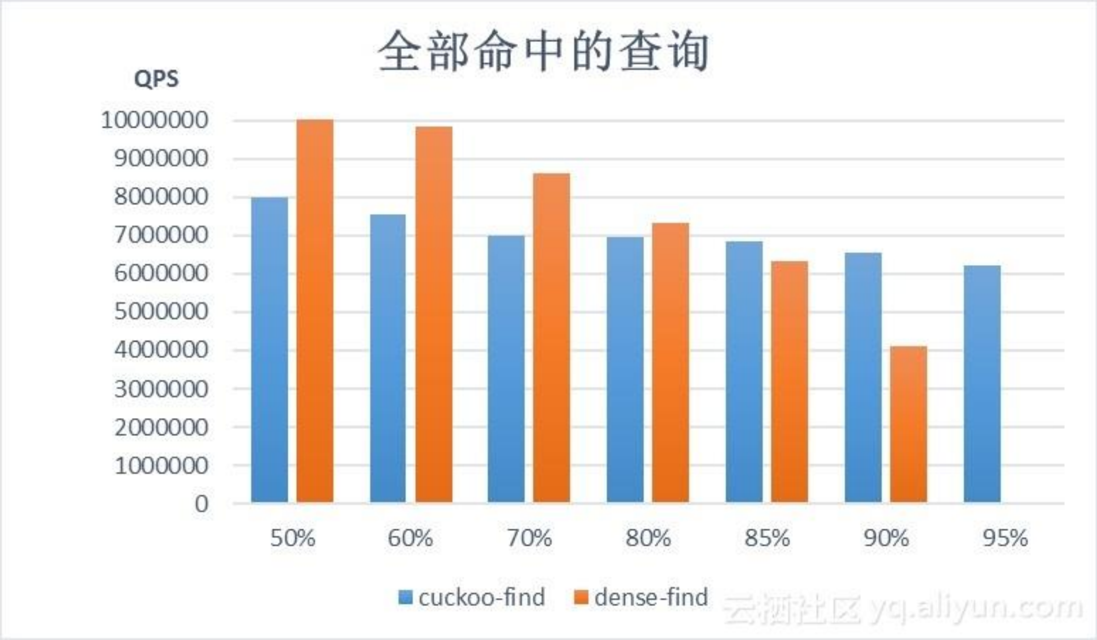
查找键总数 Hash 方法	50	250	375	500
Cuckoo hash	1	1	1.33	1.5
链式hash	1	3	4.67	5.5

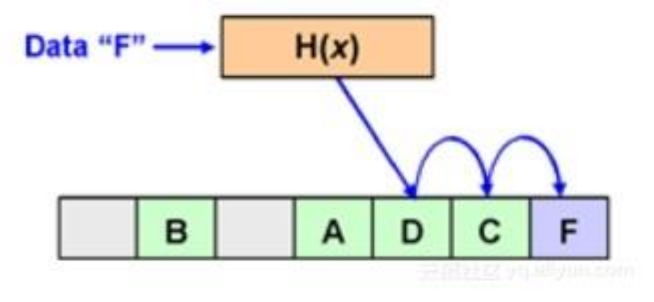
最多只会访问两个位置的键，所以每次的比较次数不会超过2，平均比较次数当然就在2以内。从表中也可以发现，当cuckoo hash的负载因子分别为0.10、0.50、0.75、1.00时，平均比较次数都维持在2以内

$$\text{负载因子} = \frac{\text{键的总数}}{\text{总容量}}$$

性能分析

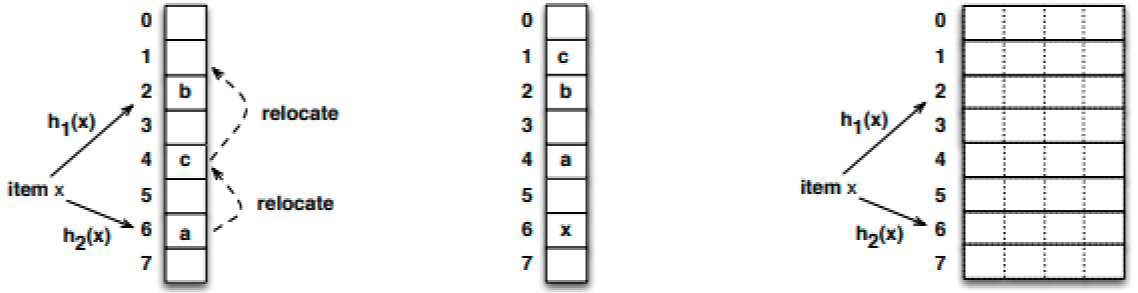






Source: https://developer.aliyun.com/article/563053

扩展：Cuckoo Filter



(a) before inserting item *x*

(b) after item *x* inserted

(c) A cuckoo filter, two hash per item and functions and four entries per bucket

Figure 1: Illustration of cuckoo hashing

Cuckoo Filter: Practically Better Than Bloom

Bin Fan, David G. Andersen, Michael Kaminsky[†], Michael D. Mitzenmacher[‡]
Carnegie Mellon University, [†]Intel Labs, [‡]Harvard University
{binfan,dga}@cs.cmu.edu, michael.e.kaminsky@intel.com, michaelm@eecs.harvard.edu

Bin Fan, David G. Andersen, Michael Kaminsky, Michael Mitzenmacher: Cuckoo Filter: Practically Better Than Bloom. CoNEXT 2014: 75-88

Next

- 并行最小生成树
- 数据结构讲义