HashMap是一个链表数组，也就是一个数组，只不过内部元素为链表。可以简单的理解为:

// 链表

class Node {

Object key;

Object value;

Node next;

}

// 数组

class HashMap {

Node[] table;

}

当HashMap中的元素超过8的时候，链表会进化为一个红黑树，可以大致理解为一个平衡二叉树，左节点都比父节点小，右节点都比父节点大，所以查找的效率跟二分查找是一致的，都是O(logn)。

put(key,value)操作

* 1 根据key获取hashcode，如果key是null，则hashcode是0，否则hashcode为: **自身的hashcode 异或 自身hashcode的无符号右移16位**

static final int hash(Object key) {

int h;

return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);

}

 2 得到了int类型的hashcode之后，就计算key在数组里面的下标，计算方法为: hash&(size-1)，其实也就是: hash%size，它俩的结果是一样的。

 3 得到了下标index后，就直接在table数组中找到了要插入的链表，接下来就遍历该链表，先寻找是否有跟hashcode相同的元素，如果相同，再判断key是否相同，如果key也相同，则直接将该node元素的value值替换为传入的value，否则就插入到链表头部，伪代码:

// 获取hashcode

int hashcode = hash(key);

// 获取下标

int index = hashcode&(size-1);

// 获取元素节点

Node node = table[index];

while(node!=null) {

// 如果hashcode不同，就不用比较了

if (hashcode != hash(node)) continue;

// hashcode相同，再比较key是否相同

if (key.equals(node.key)) {

// 先保存旧值

Object old = node.value;

// 替换为新值

node.value = value;

// 返回旧值

return old;

}

node = node.next;

}

// 没有相同的key，插入到表头

Node node = new Node(key,value);

node.next = header.next;

header.next = node;

说白了就是: 1 计算hash 2 计算下标 3 比较并插入

get(key)操作

* 1 根据key获取hashcode，同上
* 2 根据hashcode计算下标，同上
* 3 根据下标获取node并比较，先比较hashcode，相同再比较key，同上；存在都相同的元素则返回value，否则返回null

扩容resize()操作

HashMap有个loadFactor变量，叫做负载因子，当数据达到size \* loadFactor后，就会触发扩容机制，也就是会调用resize()函数。

* 1 将原来数组长度乘以2(HashMap保证容量始终为2的n次幂)，得到新数组长度，并创建一个新数组
* 2 将原来数组的数据转移到新数组中，这里的转移跟ArrayList不同，因为数组长度变了，所以下标可能也变了，所以要重新遍历并计算下标
* 1 HashMap为何要将size设定为2的n次幂

因为如果任意一个数p，如果p是2的n次幂，那么对于任意的整数a，有: a % p = a & (p-1); 我们知道，HashMap求数组的下标的方法为: hashcode % size；而且取模运算效率很低，所以如果size是2的n次幂，那么可以直接变为: hashcode & (size-1)，这是非常高效的。

* 2 证明: 当p是2的n次幂时，a%p=a&(p-1)

上面我们知道HashMap求下标的简便算法为hashcode&(size-1)，怎么证明呢？我们采用分类讨论思想，如下:

声明: 因为p是2的n次幂，所以p除了最高位为1，其余全部是0，p-1除了最高位是0，其余全部是1；所以有:

结论1: 任意的a<p，有a&(p-1)=a

结论2: p&(p-1)=0，且任意的t，tp&(p-1)=0

1 当a<p: 左边a%p=a，右边a%(p-1)=a，左边=右边，成立。

2 当a=p: 左边a%p=0，右边a&(p-1)=p&(p-1)=0，左边=右边，成立。

3 当a>p: 假设左边a%p=b，那么有a/p=t余b，也就是a=tp+b，其中t>=1，且b属于[0,p-1]；那么右边a&(p-1)=(tp+b)&(p-1)，我们知道p是2的n次幂，除了最高位其余全是0，那么tp除了最高位其余也全是0，而b<=p-1，也就是b在tp的低位0上面，

那么(tp+b)&(p-1) = tp&(p-1)+b&(p-1) = 0(结论2)+b(结论1) = b; 左边=右边，成立；

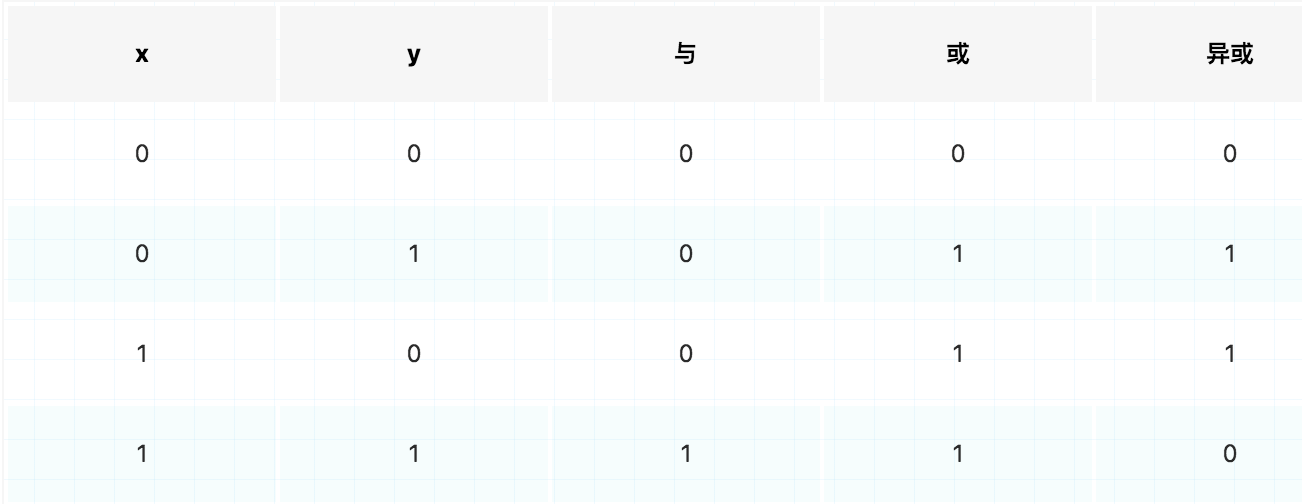
综上可知: 如果p是2的n次幂，对于任意的a，有a%p=a&(p-1)。



我们看到，tp就是在高位添加1或0，不影响低位，也就是低位永远是0。那么tp(p-1)永远是0，并且tp+b(b<=p-1)中，n永远只加在p的低位，也就是p-1中为1的位置，所以(tp+b)&(p-1)永远等于b。

* 3 为什么hashcode的计算是h^(h>>>16)

我们知道，计算下标index是h&(size-1)，一般来说，size很小，很难超过16位，也就是说，此时hashcode只有低位起作用，那么如果两个数据只有高位不同，低位相同，那么它们的index很可能相同，那么碰撞的几率就会大大增加，怎么办呢？ 我们让高位也参与index的计算，也就是size的计算方式变为: (h^(h>>>16))&(size-1)，先让h和高位异或，充分混淆，然后再计算index的值，那么为什么要用异或，不用"与"或者"或"呢，我们来看:



"与"操作1和0的比例为1:3，"或"操作1和0的比例为3:1，都是不公平的，只有"异或"操作中1和0的比例为1:1，是公平的，所以我们采取异或操作，从而保证公平，保证均匀。

* 4 为什么要用红黑树

我们知道，链表不支持随机存取，只能单向遍历，效率很低，如果冲突比较严重，同一个index上的节点很多，那么链表就会很长，此时查找效率就会很低，而使用红黑树，可以将查找效率由原来的线性时间变为对数时间，也就是O(n)变为O(logn)，所以为了效率问题， 这里直接使用了红黑树，也就是二分的思想。冲突越严重，红黑树的效果就越明显，比如链表长度为1024时，采用链表的效率就是1024，而红黑树就是log(1024)=10，差了100倍!