数据结构与算法（11）：哈希表

一、哈希表的基本概念

哈希表（Hash Table）是一种特殊的数据结构，它最大的特点就是可以快速实现查找、插入和删除。因为它独有的特点，Hash表经常被用来解决大数据问题，也因此被广大的程序员所青睐。

我们知道，数组的最大特点就是：寻址容易，插入和删除困难；而链表正好相反，寻址困难，而插入和删除操作容易。那么如果能够结合两者的优点，做出一种寻址、插入和删除操作同样快速容易的数据结构，那该有多好。这就是哈希表创建的基本思想，而实际上哈希表也实现了这样的一个“夙愿”，哈希表就是这样一个集查找、插入和删除操作于一身的数据结构。

哈希表（Hash Table）：也叫散列表，是根据关键码值（key-value）而直接进行访问的数据结构，也就是我们常用到的map。

哈希函数：也称为是散列函数，是Hash表的映射函数，它可以把任意长度的输入变换成固定长度的输出，该输出就是哈希值。哈希函数能使对一个数据序列的访问过程变得更加迅速有效，通过哈希函数，数据元素能够被很快的进行定位。

哈希表和哈希函数的标准定义：若关键字为k，则其值存放在h(k)的存储位置上。由此，不需比较便可直接取得所查记录。称这个对应关系f为哈希函数，按这个思想建立的表为哈希表。

设所有可能出现的关键字集合记为U(简称全集)。实际发生(即实际存储)的关键字集合记为K（|K|比|U|小得多）。

散列方法是使用函数h将U映射到表T[0..m-1]的下标上（m=O(|U|)）。这样以U中关键字为自变量，以h为函数的运算结果就是相应结点的存储地址。从而达到在O(1)时间内就可完成查找。
其中：

1. h：U→{0，1，2，…，m-1}，通常称h为哈希函数(Hash Function)。哈希函数h的作用是压缩待处理的下标范围，使待处理的|U|个值减少到m个值，从而降低空间开销。
2. T为哈希表(Hash Table)。
3. $h(K_i)(K_i \in U)$是关键字为Ki结点存储地址(亦称散列值或散列地址)。
4. 将结点按其关键字的哈希地址存储到哈希表中的过程称为散列(Hashing)

设计出一个简单、均匀、存储利用率高的散列函数是散列技术中最关键的问题。
但是，一般散列函数都面临着冲突的问题。两个不同的关键字，由于散列函数值相同，因而被映射到同一表位置上。该现象称为冲突(Collision)或碰撞。发生冲突的两个关键字称为该散列函数的同义词(Synonym)。

最理想的解决冲突的方法是安全避免冲突。要做到这一点必须满足两个条件：其一是$|U| \leq m$，其二是选择合适的散列函数。这只适用于|U|较小，且关键字均事先已知的情况，此时经过精心设计散列函数h有可能完全避免冲突。

但通常情况下，h是一个压缩映像。虽然$|K| \leq m$，但$|U| > m$，故无论怎样设计h，也不可能完全避免冲突。因此，只能在设计h时尽可能使冲突最少。同时还需要确定解决冲突的方法，使发生冲突的同义词能够存储到表中。

冲突的频繁程度除了与h相关外，还与表的填满程度相关。设m和n分别表示表长和表中填入的结点数，则将α=n/m定义为散列表的装填因子(Load Factor)。α越大，表越满，冲突的机会也越大。通常取α≤1。

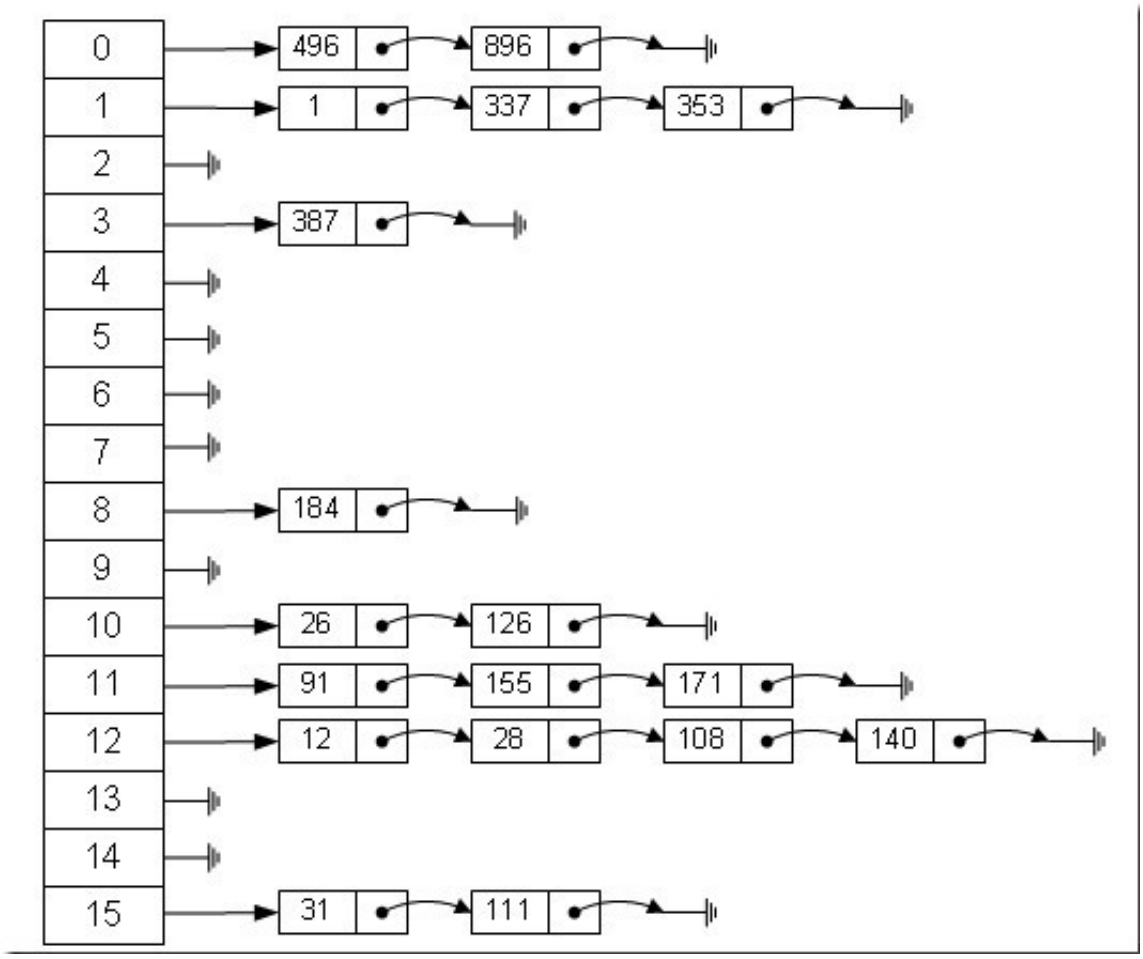
二、哈希表的实现方法

我们之前说了，哈希表是一个集查找、插入和删除操作于一身的数据结构。那这么完美的数据结构到底是怎么实现的呢？哈希表有很多种不同的实现方法，为了实现哈希表的创建，这些所有的方法都离不开两个问题——“定址”和“解决冲突”。

在这里，我们通过详细地介绍哈希表最常用的方法——取余法（定值）+拉链法（解决冲突），来一起窥探一下哈希表强大的优点。

取余法大家一定不会感觉陌生，就是我们经常说的取余数的操作。

拉链法是什么，“拉链”说白了就是“链表数组”。我这么一解释，大家更晕了，啥是“链表数组”啊？为了更好地解释“链表数组”，我们用下图进行解释：图中的主干部分是一个顺序存储结构数组，但是有的数组元素为空，有的对应一个值，有的对应的是一个链表，这就是“链表数组”。比如数组0的位置对应一个链表，链表有两个元素“496”和“896”，这说明元素“496”和“896”有着同样的Hash地址，这就是我们上边介绍的“冲突”或者“碰撞”。但是“链表数组”的存储方式很好地解决了Hash表中的冲突问题，发生冲突的元素会被存在一个对应Hash地址指向的链表中。实际上，“链表数组”就是一个指针数组，每一个指针指向一个链表的头结点，链表可能为空，也可能不为空。



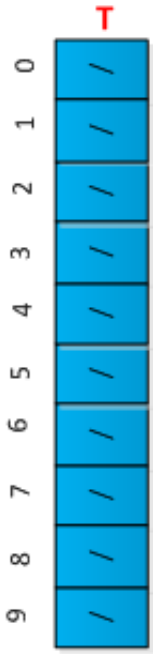
说完这些，大家肯定已经理解了“链表数组”的概念，那我们就一起看看Hash表是如何根据“取余法+拉链法”构建的吧。

将所有关键字为同义词的结点链接在同一个链表中。若选定的散列表长度为m，则可将散列表定义为一个由m个头指针组成的指针数组$T[0..m-1]$。凡是散列地址为i的结点，均插入到以$T[i]$为头指针的单链表中。T中各分量的初值均应为空指针。在拉链法中，装填因子α可以大于1，但一般均取$\alpha \leq 1$。

举例说明拉链法的执行过程，设有一组关键字为(26，36，41，38，44，15，68，12，6，51)，用取余法构造散列函数，初始情况如下图所示：

(26，36，41，38，44，15，68，12，6，51)

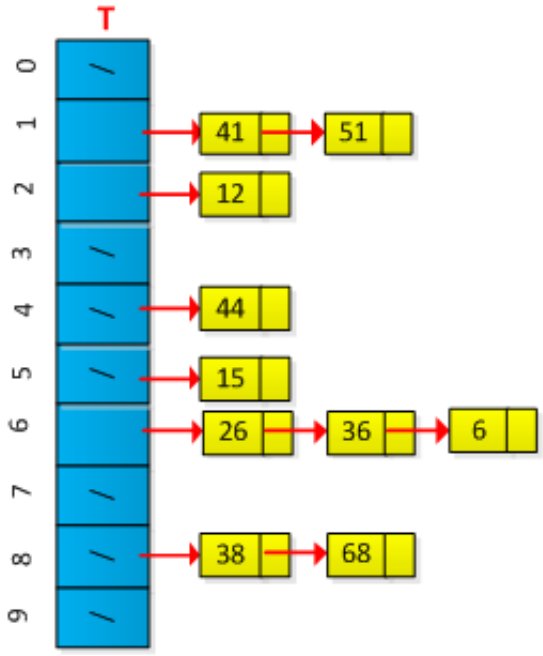
h(k) = k%10



最终结果如下图所示：

(26，36，41，38，44，15，68，12，6，51)

h(k) = k%10



理解了Hash表的创建，那根据建立的Hash表进行查找就很容易被理解了。

查找操作，如果理解了插入和删除，查找操作就比较简单了，令待查找的关键字是x，也可分为几种情况：

- 1）x所属的Hash地址未被占用，即不存在与x相同的Hash地址关键字，当然也不存在x了；
- 2）x所属的Hash地址被占用了，但它所存的关键不属于这个Hash地址，与1）相同，不存在与x相同Hash地址的关键字；
- 3）x所属的Hash地址被占用了，且它所存的关键属于这个Hash地址，即存在与x相同sHash地址的关键字，只是不知这个关键字是不是x，需要进一步查找。

由此可见，Hash表插入、删除和插入操作的效率都相当的高。

思考一个问题：如果关键字是字符串怎么办？我们怎么根据字符串建立Hash表?

通常都是将元素的key转换为数字进行散列，如果key本身就是整数，那么散列函数可以采用keymod tablesize（要保证tablesize是质数）。而在实际工作中经常用字符串作为关键字，例如身姓名、职位等等。这个时候需要设计一个好的散列函数进程处理关键字为字符串的元素。参考《数据结构与算法分析》第5章，有以下几种处理方法：

- 方法1：将字符串的所有的字符的ASCII码值进行相加，将所得和作为元素的关键字。此方法的缺点是不能有效的分布元素，例如假设关键字是有8个字母构成的字符串，散列表的长度为10007。字母最大的ASCII码为127，按照方法1可得到关键字对应的最大数值为127×8=1016，也就是说通过散列函数映射时只能映射到散列表的槽0-1016之间，这样导致大部分槽没有用到，分布不均匀，从而效率低下。
- 方法2：假设关键字至少有三个字母构成，散列函数只是取前三个字母进行散列。该方法只是取字符串的前三个字符的ASCII码进行散列，最大的得到的数值是2851，如果散列的长度为10007，那么只有28%的空间被用到，大部分空间没有用到。因此如果散列表太大，就不太适用。
- 方法3：借助Horner's 规则，构造一个质数（通常是37）的多项式，（非常的巧妙，不知道为何是37）。计算公式为:$key[keysize-i-1]*37^i, 0 \leq i < keysize$求和。该方法存在的问题是如果字符串关键字比较长，散列函数的计算过程就变长，有可能导致计算的hashVal溢出。针对这种情况可以采取字符串的部分字符进行计算，例如计算偶数或者奇数位的字符。

三、哈希表定址与解决冲突

3.1 哈希表“定址的方法”

其实常用的“定址”的手法有“五种”：

1. 直接定址法：很容易理解，key=Value+C；这个“C"是常量。Value+C其实就是一个简单的哈希函数。
2. 除法取余法：key=value%C
3. 数字分析法：这种蛮有意思，比如有一组value1=112233，value2=112633，value3=119033，针对这样的数我们分析数中间两个数比较波动，其他数不变。那么我们取key的值就可以是key1=22,key2=26,key3=90。
4. 平方取中法
5. 折叠法：举个例子，比如value=135790，要求key是2位数的散列值。那么我们将value变为13+57+90=160，然后去掉高位“1”,此时key=60，哈哈，这就是他们的哈希关系，这样做的目的就是key与每一位value都相关，来做到“散列地址”尽可能分散的目地。

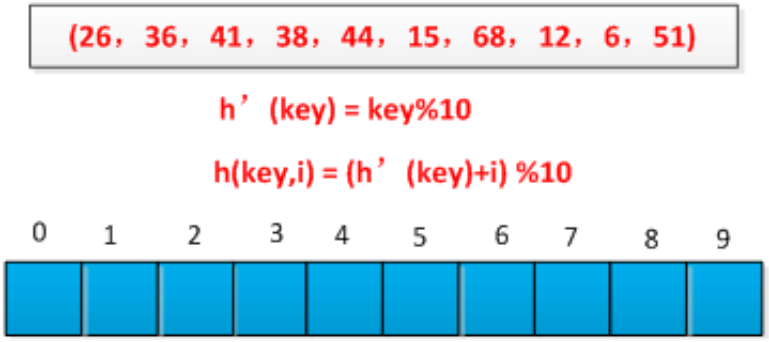
影响哈希查找效率的一个重要因素是哈希函数本身。当两个不同的数据元素的哈希值相同时，就会发生冲突。为减少发生冲突的可能性，哈希函数应该将数据尽可能分散地映射到哈希表的每一个表项中。

3.2 哈希表“解决冲突”的方法

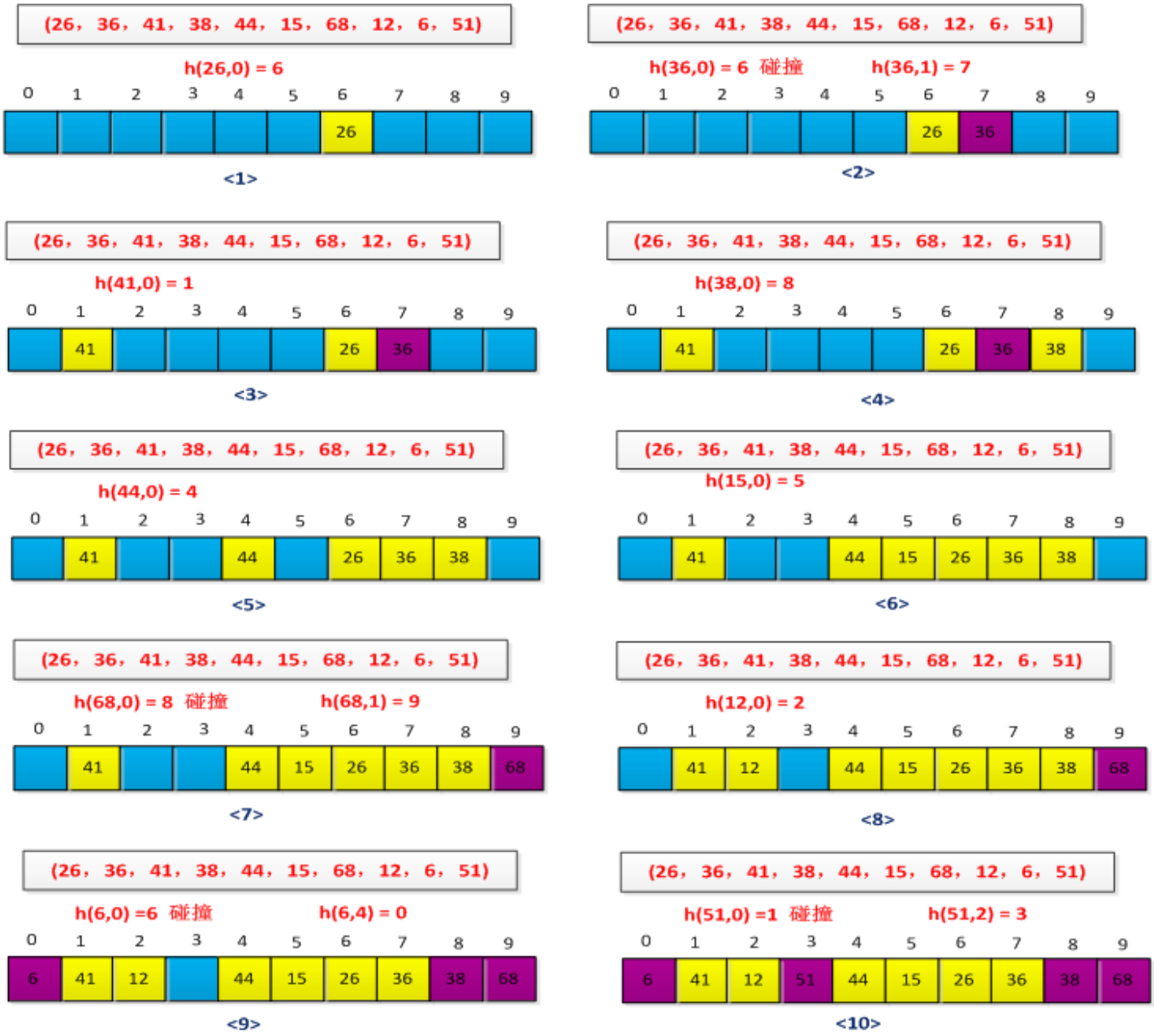
Hash表解决冲突的方法主要有以下几种：

链接地址法：
将哈希值相同的数据元素存放在一个链表中，在查找哈希表的过程中，当查找到这个链表时，必须采用线性查找方法。

开放定址法：
如果两个数据元素的哈希值相同，则在哈希表中为后插入的数据元素另外选择一个表项。当程序查找哈希表时，如果没有在第一个对应的哈希表项中找到符合查找要求的数据元素，程序就会继续往后查找，直到找到一个符合查找要求的数据元素，或者遇到一个空的表项。线性探测带来的最大问题就是冲突的堆积，你把别人预定的坑占了，别人也就要像你一样去找坑。改进的办法有二次方探测法和随机数探测法。开放地址法包括线性探测、二次探测以及双重散列等方法。其中线性探测法示意图如下：



散列过程如下图所示：



再散列函数法：
发生冲突时就换一个散列函数计算，总会有一个可以把冲突解决掉，它能够使得关键字不产生聚集，但相应地增加了计算的时间。

公共溢出区法：
其实就是为所有的冲突，额外开辟一块存储空间。如果相对基本表而言，冲突的数据很少的时候，使用这种方法比较合适。

3.3 哈希表“定址”和“解决冲突”之间的权衡

虽然哈希表是在关键字和存储位置之间建立了对应关系，但是由于冲突的发生，哈希表的查找仍然是一个和关键字比较的过程，不过哈希表平均查找长度比顺序查找要小得多，比二分查找也小。

查找过程中需和给定值进行比较的关键字个数取决于下列三个因素：哈希函数、处理冲突的方法和哈希表的装填因子。

哈希函数的"好坏"首先影响出现冲突的频繁程度，但如果哈希函数是均匀的，则一般不考虑它对平均查找长度的影响。

对同一组关键字，设定相同的哈希函数，但使用不同的冲突处理方法，会得到不同的哈希表，它们的平均查找长度也不同。

一般情况下，处理冲突方法相同的哈希表，其平均查找长度依赖于哈希表的装填因子α。显然，α越小，产生冲突的机会就越大；但α过小，空间的浪费就过多。通过选择一个合适的装填因子α，可以将平均查找长度限定在一个范围内。

总而言之，哈希表“定址”和“解决冲突”之间的权衡决定了哈希表的性能。

四、实现Hashmap

假设我们要设计的是一个用来保存某大学所有在校学生个人信息的数据表。因为在校学生数量也不是特别巨大(8W?)，每个学生的学号是唯一的,因此，我们可以简单的应用直接定址法，声明一个10W大小的数组，每个学生的学号作为主键。然后每次要添加或者查找学生，只需要根据需要去操作即可。

但是，显然这样做是很脑残的。这样做系统的可拓展性和复用性就非常差了，比如有一天人数超过10W了？如果是用来保存别的数据呢？或者我只需要保存20条记录呢？声明大小为10W的数组显然是太浪费了的。

如果我们是用来保存大数据量（比如银行的用户数，4大的用户数都应该有3-5亿了吧？），这时候我们计算出来的HashCode就很可能会有冲突了， 我们的系统应该有“处理冲突”的能力，此处我们通过挂链法“处理冲突”。

如果我们的数据量非常巨大，并且还持续在增加，如果我们仅仅只是通过挂链法来处理冲突，可能我们的链上挂了上万个数据后，这个时候再通过静态搜索来查找链表，显然性能也是非常低的。所以我们的系统应该还能实现自动扩容，当容量达到某比例后，即自动扩容，使装载因子保存在一个固定的水平上。

综上所述，我们对这个Hash容器的基本要求应该有如下几点：

- 满足Hash表的查找要求
- 能支持从小数据量到大数据量的自动转变（自动扩容）
- 使用挂链法解决冲突

代码详见hashmap源码剖析