我来讲Hash

今日目标：

1：能够说出散列表，散列函数的概念，作用

2：能够说出几种散列函数的设计方法

3：能够说出散列冲突的几种解决方案

4：能够分析出散列表的复杂度

5：能够列举出几个散列表在工程中的应用示例

6：完成面试实战题目

1、散列表（Hash Table）

1.1、概述

散列表(Hash Table)又名**哈希表**/Hash表，是**根据键（Key）直接访问**在内存存储位置**值（Value）**的数据结构，它是**由数组演化而来**的，利用了数组支持按照下标进行随机访问数据的特性；举例说明散列的思想及原理如下：

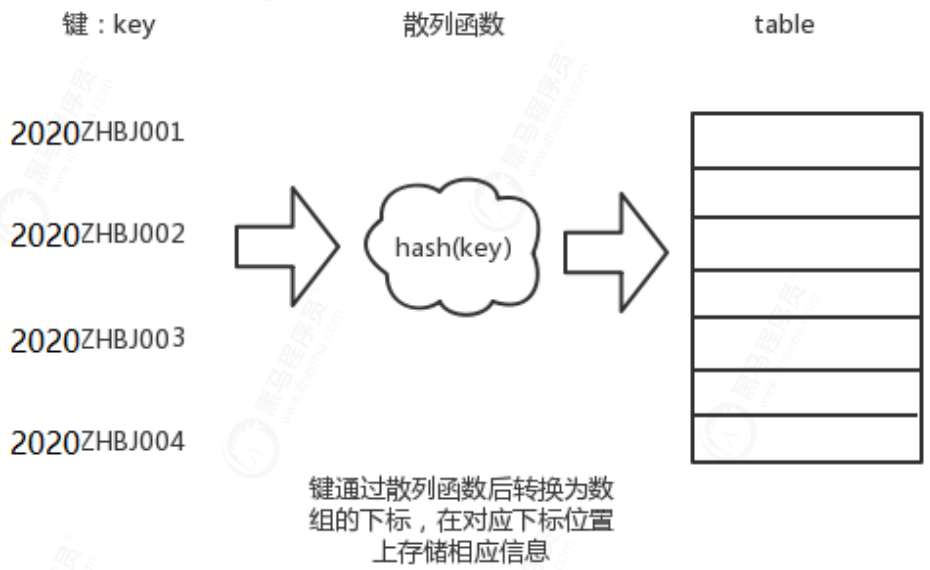
> 场景描述：
>
> 全国每年都会举行的马拉松比赛，举办方会给每一个参赛选手分配一个参赛编号，最终进行成绩记录的时候根据参赛编号来记录，然后需要根据参赛编号查询参赛选手的信息；现假设有100个人参加马拉松，编号是1-100，如果要编程实现根据选手的编号迅速找到选手信息？
>
> 解决方案：
>
> 将这100位参赛选手的信息存入到一个数组中，编号为1的选手信息存放在数组下标为1的位置，编号为2的选手信息存放在数组下标为2的位置.....依次同理编号为K的选手信息存放在数组下标为K的位置。因为参赛选手的编号跟数组的下标对应，所以当我们需要查询编号为X的选手信息我们只需要根据数组的下标X取出数组中的元素数据即可。我们知道在数组中查询数据的时间复杂度是O(1)，这个查询的效率非常的高。

这个例子中就已经用到了散列的思想，***选手编号和数组下标一一对应***，我们可以用O(1)的时间复杂度来获取选手信息，但该例中蕴含的散列思想还不够明显，下面进行场景升级

> 场景升级：
>
> 假设马拉松大赛主办方对每个选手不采用1-100的自然数对选手进行编号，编号有一定的规则比如：2020ZHBJ001，其中2020代表年份，ZH代表中国，BJ代表北京，001代表原来的编号，那此时的编号2020ZHBJ001不能直接作为数组的下标，此时应该如何实现呢？
>
> 解决方案：
>
> 思路跟之前的还是一样，虽然2020ZHBJ001不能直接作为数组的下标，但是我们可以通过某种方式将编号2020ZHBJ001转化为数组的下标，然后在对应位置上存储选手信息，获取的时候也是如此，比如：
>
> 2020ZHBJ001----------转换为数组下标---------->1

> 2020ZHBJ002----------转换为数组下标---------->2

这就是一个非常典型的散列的思想，我们可以将选手编号作为：**键(key)**用以标识每一个参赛选手，把编号转换为数组下标的方法叫做**散列函数**(或者**哈希函数**)，通过散列函数对key进行运算得到的值就叫做**散列值**(或**哈希值**)如下图所示：



散列表利用了数组按照下标访问数组元素时间复杂度是O(1)的特性，通过散列函数将键(key)映射为数组下标，然后将数据存储到对应下标的位置上，当我们根据key再次查询数据的时候，同样根据散列函数将key映射为下标，根据下标从数组对应位置上获取数据。

1.2、散列函数

将键(key)映射为数组下标的函数叫做散列函数。可以表示为：hashValue = **hash**(key)

> 上面的列子中：选手编号2020ZHBJ001就是key，转换后得到的数组下标就是hashValue，中间的转换过程就是散列函数hash

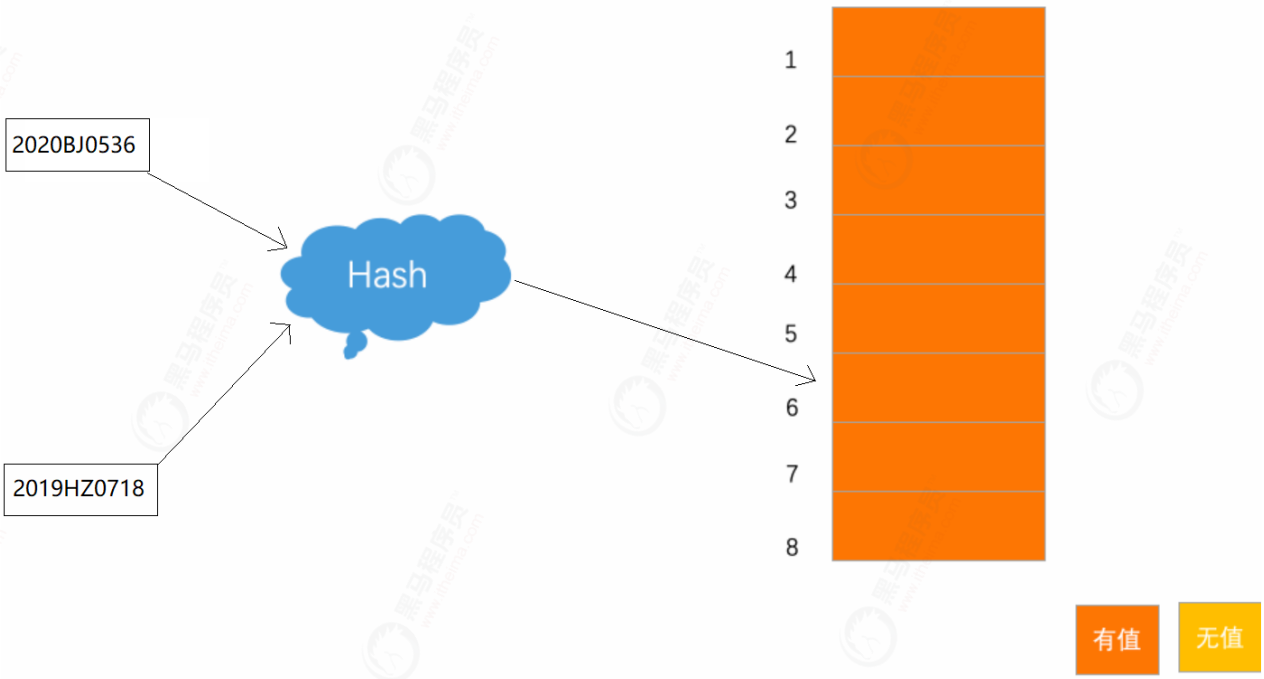
散列函数的基本要求：

1：散列函数计算得到的散列值必须是大于等于0的正整数，因为hashValue需要作为数组的下标。

2：如果key1==key2，那么经过hash后得到的哈希值也必相同即：hash(key1) == hash(key2)

3：如果key1 != key2，那么经过hash后得到的哈希值也必不相同即：hash(key1) != hash(key2)

前两个要求我们很容易理解，第三个要求看起来没有任何问题，但是在实际的情况下想找一个散列函数能够做到对于不同的key计算得到的散列值都不同几乎是不肯能的，即便像著名的MD5,SHA等哈希算法也无法避免这一情况，这也就是我们即将要说到的**散列冲突(或者哈希冲突，哈希碰撞，就是指多个key映射到同一个数组下标位置)**



另外数组的存储空间是有限的，当数组快要存满了的时候散列冲突的概率会增加。

如何设计一个企业级的散列函数呢?

散列函数设计的好与坏直接决定了散列冲突发生的概率，也直接决定了散列表的性能，那好的散列函数应该满足一些什么特点呢?

1：散列函数不能太复杂，因为太复杂度势必要消耗很多的时间在计算哈希值上，也会间接影响散列表性能。

2：散列函数计算得出的哈希值尽可能的能随机并且均匀的分布，这样能够将散列冲突最小化。

散列函数设计方法

实际工作中，需要综合考虑各种因素。这些因素有关键字的长度、特点、分布、还有散列表的大小等。散列函数各式各样，在此举几个常用的、简单的散列函数的设计方法。

- **1：直接寻址法：**

比如我们现在要对0-100岁的人口数字统计表，那么我们对年龄这个关键字key就可以直接用年龄的数字作为地址。此时hash(key) = key。这个时候，我们可以得出这么个哈希函数：hash(0) = 0，hash(1) = 1，……，hash(20) = 20。

地址	年龄	人数
00	0	500万
01	1	600万
02	2	450万
…	…	…
20	20	1500万
…	…	…

如果我们现在要统计的是1980年后出生年份的人口数，那么我们对出生年份这个关键字可以用年份减去1980来作为地址。此时hash(key) = key-1980

地址	出生年份	人数
00	1980	1500万
01	1981	1600万
02	1982	1300万
…	…	…
20	2000	500万
…	…	…

也就是说，我们可以**取关键字key的某个线性函数值为散列地址，即：**

> **hash(key) = a x key + b，其中a,b为常量**

这样的散列函数优点就是简单、均匀，也不会产生冲突，但问题是这需要事先知道关键字key的分布情况，适合查找表较小且连续的情况。由于这样的限制，在现实应用中，直接寻址法虽然简单，但却并不常用。

- **2：除留余数法**

除留余数法此方法为最常用的构造散列函数方法。**对于散列表长为m的散列函数公式为：**

> **hash(key) = key mod p (p ≤ m)**

本方法的关键就在于选择合适的p, p如果选得不好，就可能会容易产生哈希冲突，比如：有12个关键字key，现在我们要针对它设计一个散列表。如果采用除留余数法，那么可以先尝试将散列函数设计为hash(key) = key mod 12的方法。比如29 mod 12 = 5，所以它存储在下标为5的位置。

下标	0	1	2	3	4	5	6	7	8	9	10	11
关键字	12	25	38	15	16	29	78	67	56	21	22	47

不过这也是存在冲突的可能的，因为12 = 2×6 = 3×4。如果关键字中有像18(3×6)、30(5×6)、42(7×6)等数字，它们的余数都为6，这就和78所对应的下标位置冲突了。

此时如果我们不选用p=12而是选用p=11则结果如下：

下标	1	2	3	4	5	6	7	8	9	10	0	1
关键字	12	24	36	48	60	72	84	96	108	120	132	144

使用除留余数法的一个经验是：**当P取小于哈希表长的最大质数时，产生的哈希函数较好。**

- **3：数字分析法：**

数字分析法并没有特定的的公式可以寻，比如我们想要把手机号码作为键(key)进行哈希处理，手机号码的前几位重复的可能性非常大，但是一般后四位就比较随机，所以我们可以选取手机号的后四位作为哈希值，这种设计散列函数的方法我们一般叫做数字分析法，数字分析法的核心思想就是从关键字key中提取数字分布比较均匀的若干位作为哈希值，即当关键字的位数很多时，可以通过对关键字的各位进行分析，丢掉分布不均匀的位，作为哈希值它只适合于所有关键字值已知的情况。通过分析分布情况把关键字取值区间转化为一个较小的关键字取值区间。

4：平方取中法：

这是一种常用的哈希函数构造方法。这个方法是先取关键字的平方，然后根据可使用空间的大小，选取平方数是中间几位为哈希地址。

> **hash(key) = key平方的中间几位**

这种方法的原理是通过取平方扩大差别，平方值的中间几位和这个数的每一位都相关，则对不同的关键字得到的哈希函数值不易产生冲突，由此产生的哈希地址也较为均匀。

关键字	关键字的平方	哈希函数值
1234	1522756	227
2143	4592449	924
4132	17073424	734
3214	10329796	297

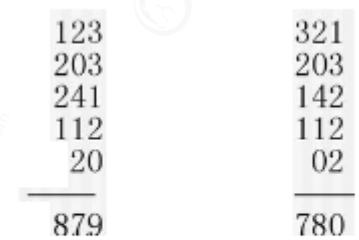
5：折叠法：

有时关键码所含的位数很多，采用平方取中法计算太复杂，则可将关键码分割成位数相同的几部分（最后一部分的位数可以不同），然后取这几部分的叠加和（舍去进位）作为散列地址，这方法称为折叠法，折叠法可分为两种：

移位叠加：将分割后的几部分低位对齐相加。

边界叠加:从一端沿分割界来回折叠，然后对齐相加。

比如关键字为:12320324111220，分为5段，123，203，241，112，20，两种方式如下：



移位叠加　　边界叠加

当然了散列函数的设计方法不仅仅只有这些方法，对于这些方法我们无需全部掌握也不需要死记硬背，我们理解其设计原理即可。

1.3、散列冲突

前面讲到即使再好的散列函数我们也无法避免不了散列冲突(哈希冲突，哈希碰撞)，那如果真的出现了散列冲突我们应该如何来解决散列冲突呢？

两类方法解决散列冲突：开放寻址法，链表法

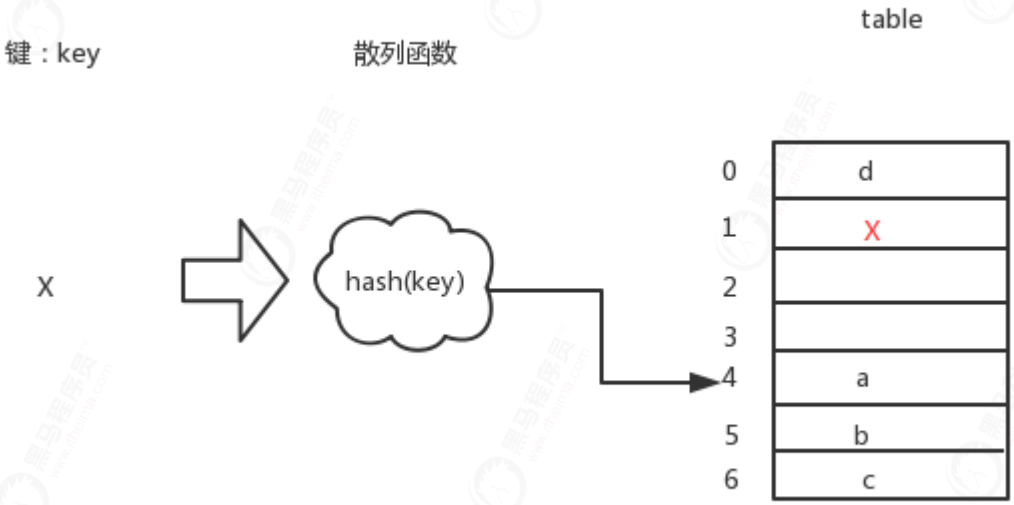
开放寻址法

开放寻址法的核心思想是：一旦出现了散列冲突，我们就重新去寻址一个空的散列地址，只要散列表足够大，空的散列地址总能找到，一旦找到就将记录存入该地址。那如何重新寻址一个空的散列地址呢？我们由浅入深依次介绍几种方法。

线性检测

我们往散列表中插入数据时，如果某个数据经过散列函数散列之后，存储位置已经被占用了，我们就从当前位置开始，依次往后查找，看是否有空闲位置，直到找到为止。

静态图示说明如下：



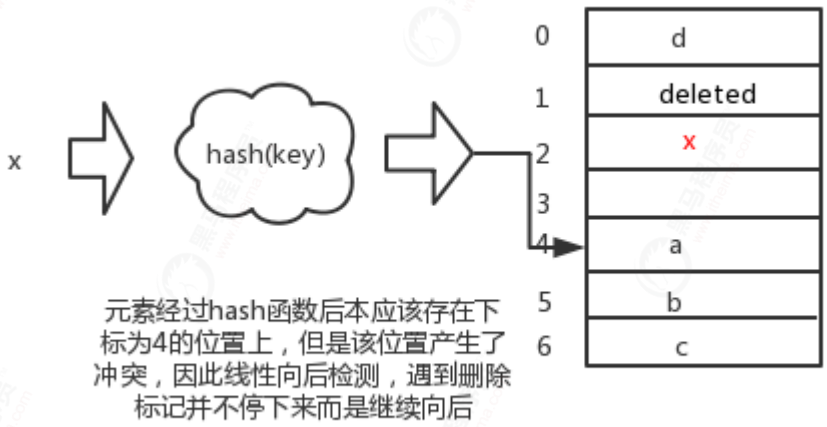
> 散列表的大小为7，在元素X插入之前已经有a，b，c，d四个元素插入到散列表中了，元素X经过hash(X)计算之后得到的哈希值为4，但是4这个位置已经有数据了，所以产生了冲突，于是我们需要按照顺序依次向后查找，一直查找到数组的尾部都没有空闲位置了，所以再从头开始查找，直到找到空闲位置下标为1的位置，至此将元素X插入下标为1的位置。

如果要**从散列表中查找是否存在某个元素**，这个过程跟插入类似，先根据散列函数求出要查找元素的key的散列值，然后比较数组中下标为其散列值的元素和要查找的元素，如果相等则表明该元素就是我们想要的元素，如果不等还要继续向后寻找遍历，如果遍历到数组中的空闲位置还没有找到则说明我们要找的元素并不在散列表中。

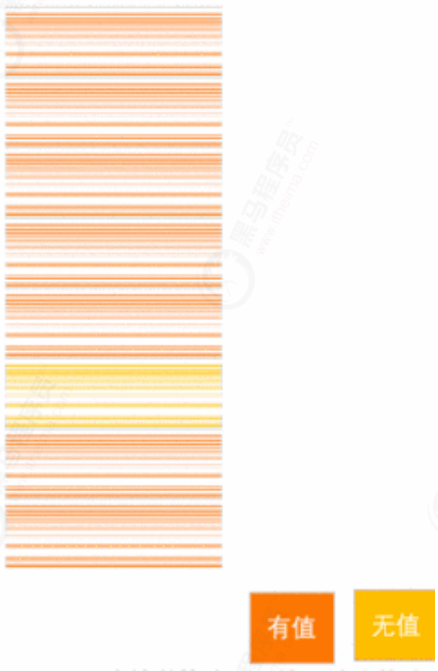
散列表跟数组一样，不仅支持插入、查找操作，还**支持删除操作。**其中删除操作稍微有点特殊，删除操作不能简单的将要删除的位置设置为空，为什么呢?

从散列表中查找是否存在某个元素一旦在对应hash值下标下的元素不是我们想要的就会继续在散列表中向后遍历，直到找到数组中的空闲位置，但如果这个空闲位置是我们刚刚删除的，那就会中断向后查找的过程，那这样的话查找的算法就会失效，本来应该认定为存在的元素会被认定为不存在，那删除的问题如何解决呢?

我们可以将删除的元素特殊标记为deleted，当线性检测遇到标记deleted的时候并不停下来而是继续向后检测，如下图所示:



不过可能大家也发现了**使用线性检测的方式存在很大的问题，那就是当散列表中的数据越来越多的时候，散列冲突发生的可能性就越来越大，空闲的位置越来越少，那线性检测的时间就会越来越长，在极端情况下我们可能需要遍历整个数组，所以最坏的情况下时间复杂度为O(n)。**

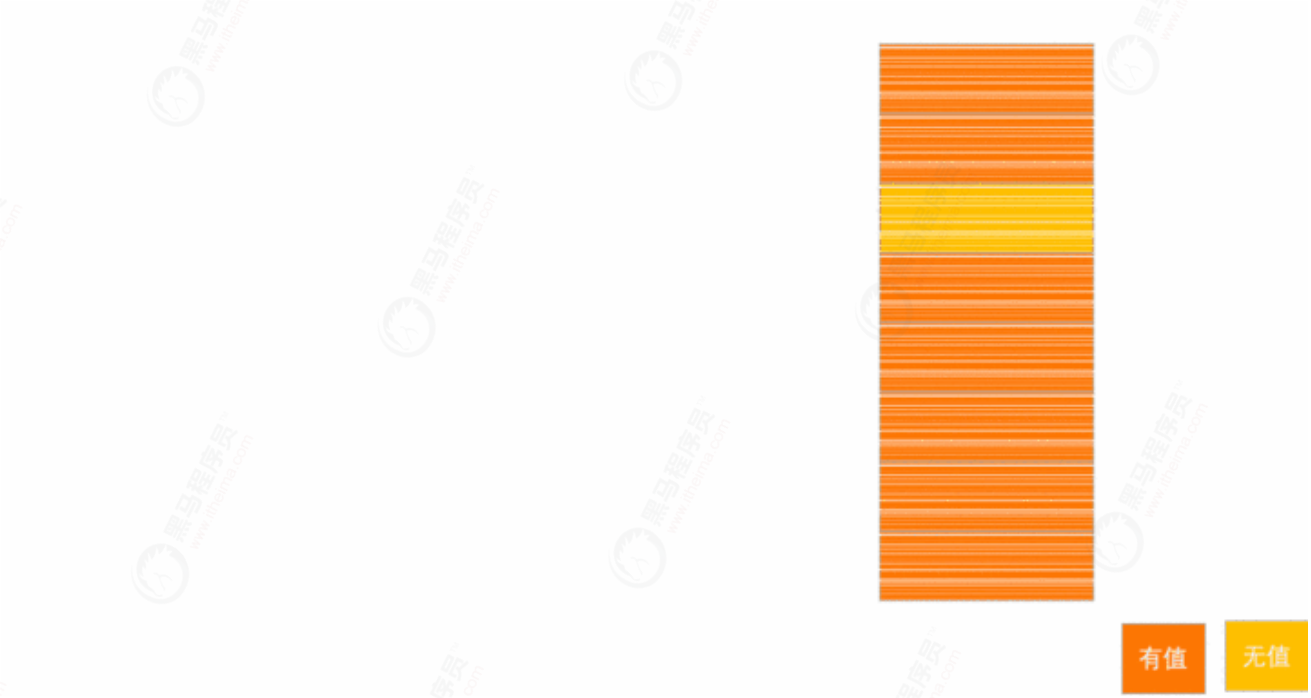


因此对于开放寻址解决冲突还有另外两种比较经典的的检测方式：**二次检测，双重散列**

二次检测

所谓的二次检测跟线性检测的原理一样，只不过线性检测每次检测的步长是1，每次检测的下标依次是：hash(key)+0，hash(kjey)+1，hash(key)+2，hash(key)+3.....，所谓的二次检测指的是每次检测的步长变为原来的二次方，即每次检测的下标为

$$hash(key)+0, hash(key)+1^2, hash(key)+2^2, hash(key)+3^2, ...$$



双重散列

所谓的双重散列，意思就是不仅要使用一个散列函数。我们使用一组散列函数 hash1(key)，hash2(key)，hash3(key)……我们先用第一个散列函数，如果计算得到的存储位置已经被占用，再用第二个散列函数，依次类推，直到找到空闲的存储位置。

总之不管采用哪种探测方法，当散列表中空闲位置不多的时候，散列冲突的概率就会大大提高。

为了尽可能保证散列表的操作效率，一般情况下，我们会尽可能保证散列表中有一定比例的空闲位置。我们用**装载因子(load factor)**来表示空位的多少。散列表装载因子的计算公式为：

> **装载因子 = 散列表中元素的个数 / 散列表的长度**

装载因子越大，说明空闲位置越少，冲突越多，散列表的性能会下降。那**如果装载因子过大了怎么办**？装载因子过大不仅插入的过程中要多次寻址，查找的过程也会变得很慢。

对于没有频繁插入和删除的静态数据集合来说，我们很容易根据数据的特点、分布等，设计出极少冲突的散列函数，因为毕竟之前数据都是已知的。对于动态散列表来说，数据集合是频繁变动的，我们事先无法预估将要加入的数据个数，所以我们也无法事先申请一个足够大的散列表。随着数据慢慢加入，装载因子就会慢慢大。当装载因子大到一定程度之后，散列冲突就会变得不可接受。这个时候，我们该如何处理呢？

此时我们需要针对散列表，**当装载因子过大时**，进行**动态扩容**，重新申请一个更大的散列表，将数据搬移到这个新散列表中。假设每次扩容我们都申请一个原来散列表大小两倍的空间。如果原来散列表的装载因子是 0.8，那经过扩容之后，新散列表的装载因子就下降为原来的一半，变成了 0.4。针对数组的扩容，数据搬移操作比较简单。但是，**针对散列表的扩容**，数据搬移操作要复杂很多。因为散列表的大小变了，数据的存储位置也变了，所以我们**需要通过散列函数重新计算每个数据的存储位置。**

插入一个数据，最好情况下，不需要扩容，最好时间复杂度是 O(1)。最坏情况下，散列表装载因子过高，启动扩容，我们需要重新申请内存空间，重新计算哈希位置，并且搬移数据，所以时间复杂度是 O(n)。但是这个动态扩容的过程在n次操作中会遇见一次，因此平均下来时间复杂度接近最好情况，就是 O(1)。

当散列表的装载因子超过某个阈值时，就需要进行扩容。装载因子阈值需要选择得当。如果太大，会导致冲突过多；如果太小，会导致内存浪费严重。装载因子阈值的设置要权衡时间、空间复杂度。如果内存空间不紧张，对执行效率要求很高，可以降低负载因子的阈值；相反，如果内存空间紧张，对执行效率要求又不高，可以增加负载因子的值，甚至可以大于 1。

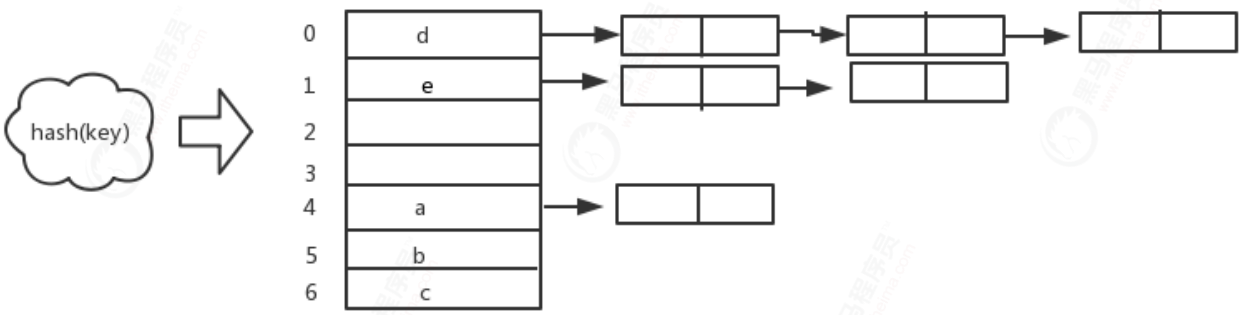
总结一下，当数据量比较小、装载因子小的时候，适合采用开放寻址法。这也是 Java 中的ThreadLocalMap使用开放寻址法解决散列冲突的原因

解决散列冲突除了使用这类开放寻址法外还可以使用链表法，这是一种比开放寻址法更加常用的解决散列冲突的方案，接下来我们就来学习使用链表法解决散列冲突。

链表法（拉链）

相比开放寻址法，它要简单很多。我们来看这个图，在散列表中，数组的每个下标位置我们可以称之为“桶（bucket）”或者“槽（slot）”，每个桶(槽)会对应一条链表，所有散列值相同的元素我们都放到相同槽位对应的链表中。

散列表，每个下标位置就是一个桶或者槽，所有散列值相同的元素存储在一个桶内



当**插入**的时候，我们只需要通过散列函数计算出对应的散列槽位，将其插入到对应链表中即可，所以插入的时间复杂度是 O(1)。

当**查找、删除**一个元素时，我们同样通过散列函数计算出对应的槽，然后遍历链表查找或者删除。那查找或删除操作的时间复杂度是多少呢？

实际上，这两个操作的时间复杂度跟链表的长度 k 成正比，也就是 O(k)。对于散列比较均匀的散列函数来说，理论上讲，k=n/m，其中 n 表示散列中数据的个数，m 表示散列表中“槽”的个数，但是基本上或者说平均情况下基于链表法解决冲突时查询的时间复杂度是O(1)。

当然了散列表的查询效率并不能笼统地说成是 O(1)。它跟散列函数、装载因子、散列冲突等都有关系。如果散列函数设计得不好，或者装载因子过高，都可能导致散列冲突发生的概率升高，查询效率下降。在极端情况下，有些恶意的攻击者，还有可能通过精心构造的数据，使得所有的数据经过散列函数之后，都散列到同一个槽里。如果我们使用的是基于链表的冲突解决方法，那这个时候，散列表就会退化为链表，查询的时间复杂度就从 O(1) 急剧退化为 O(n)，这也就是散列表碰撞攻击的基本原理。

链表法对内存的利用率比开放寻址法要高。因为链表结点可以在需要的时候再创建，并不需要像开放寻址法那样事先申请好。实际上，这一点也是我们前面讲过的链表优于数组的地方。链表法比起开放寻址法，对大装载因子的容忍度更高。开放寻址法只能适用装载因子小于 1 的情况。接近 1 时，就可能会有大量的散列冲突，导致大量的探测、再散列等，性能会下降很多。但是对于链表法来说，只要散列函数的值随机均匀，即便装载因子变成 100，也就是链表的长度变长了而已，虽然查找效率有所下降，但是比起顺序查找还是快很多。

我们之前学习链表的时候讲过链表因为要存储指针，所以对于比较小的对象的存储，是比较消耗内存的，还有可能会让内存的消耗翻倍，如果我们存储的是大对象，也就是说要存储的对象的大小远远大于一个指针的大小（4 个字节或者 8 个字节），那链表中指针的内存消耗在大对象面前就可以忽略了。

实际上，我们对链表法稍加改造，可以实现一个更加高效的散列表。那就是，我们将链表法中的链表改造为其他高效的动态数据结构，比如跳表、红黑树。这样，即便出现散列冲突，极端情况下，所有的数据都散列到同一个桶内，那最终退化成的散列表的查找时间也只不过是 O(logn)。这样也就有效避免了前面讲到的散列碰撞攻击。

总结一下，基于链表的散列冲突处理方法比较适合存储大对象、大数据量的散列表，而且，比起开放寻址法，它更加灵活，支持更多的优化策略，比如用红黑树代替链表。

1.4、复杂度分析

Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

图片来源：https://www.bigocheatsheet.com/

https://www.bigocheatsheet.com/

1.5、工程应用

在实际的工程应用中，高级编程语言都有在哈希表的基础上抽象出具体的实现，并且都已内置给开发者直接使用，在java中使用的最多的就是map和set

- Map：key-value键值对，key不重复

接口定义：https://docs.oracle.com/javase/8/docs/api/java/util/Map.html

API	说明
put(K key, V value)	在map中将指定key和指定value关联，根据key存储value。
get(Object key)	返回指定键映射到的值，如果该映射不包含该键的映射，则返回null。
containsKey(Object key)	如果此映射包含指定键的映射，则返回true。
remove(Object key)	如果key存在，则从该映射中删除该key的映射。
size()	返回此映射中键-值映射的数目。
isEmpty()	如果该映射不包含键-值映射，则返回true。
clear()	从该映射中删除所有映射。

Map的企业级实现：ConcurrentHashMap, ConcurrentSkipListMap，HashMap, Hashtable,Properties,TreeMap,

> 课后思考：HashTable和HashMap的区别，各自是怎么实现的?
>
> 扩展：其他语言的map有什么操作?

- Set：不重复元素的集合

接口定义：https://docs.oracle.com/javase/8/docs/api/java/util/Set.html

API	说明
add(E e)	如果指定的元素不存在，则将其添加到该集合中。
contains(Object o)	如果此集合包含指定的元素，则返回true。
remove(Object o)	如果指定的元素存在，则从该集合中删除它。
iterator()	在此集合的元素上返回迭代器。
size()	返回此集合中的元素数(其基数)。
isEmpty()	如果此集合不包含任何元素，则返回true。
clear()	从这个集合中删除所有元素。

Map的企业级实现： HashSet, LinkedHashSet, TreeSet

> 课后思考：HashSet是如何实现的?
>
> 扩展：其他语言的set有什么操作?

2、面试实战

2.1、242. 有效的字母异位词

亚马逊，微软最近面试题，242. 有效的字母异位词

字母异位词：指的是字母相同，但排列不同的字符串

构造哈希表计数器

```java
class Solution {
    public boolean isAnagram(String s, String t) {
        //特殊判断
        if (s == null || t == null || s.length() != t.length()) {
            return false;
        }
        //构造哈希表
        int[] hashtable = new int[26];
        for (int i=0;i<s.length();i++) {
            hashtable[hash(s.charAt(i))]++;
        }
        for (int i=0;i<t.length();i++) {
            hashtable[hash(t.charAt(i))]--;
            if (hashtable[hash(t.charAt(i))] < 0 ) {
                return false;
            }
        }
        return true;
    }

    //根据key获取数组下标
    public int hash(char key){
        return key - 'a';
    }
}
```

> 散列函数的设计原理：

ASCII 值	控制字符	ASCII 值	控制字符	ASCII 值	控制字符	ASCII 值	控制字符
0	NUT	32	(space)	64	@	96	、
1	SOH	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	,	71	G	103	g
8	BS	40	(	72	H	104	h
9	HT	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DCI	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	X	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	TB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[	123	{
28	FS	60	<	92	/	124	|
29	GS	61	=	93	]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	—	127	DEL

转义字符	含义	ASCII 码（16/10 进制）
\o	空字符(NULL)	00H/0
\n	换行符(LF)	0AH/10
\r	回车符(CR)	0DH/13
\t	水平制表符(HT)	09H/9

时间复杂度：O(n)，空间复杂度O(1)

扩展：

如果输入字符串包含 unicode 字符怎么办？你能否调整你的解法来应对这种情况？

2.2、49. 字母异位词分组

腾讯，高盛集团最近面试题，49. 字母异位词分组

使用map来存储同组的异位词，设法让异位词对应相同的key，value部分用一个集合来存储异位词

```java
class Solution {
    public List<List<String>> groupAnagrams(String[] strs) {
        //特殊判断
        if (strs==null || strs.length < 1) {
            return new ArrayList();
        }
        //使用map
        Map<String,List<String>> map = new HashMap();
        for (String str:strs) {
            //针对每个str构造key,确保字母异位词对应相同的key
            String key = genKey(str);
            if (!map.containsKey(key)) {
                map.put(key,new ArrayList());
            }
            map.get(key).add(str);
        }
        return new ArrayList(map.values());
    }
    //生成key,确保字母异位词对应相同的key
    public String genKey(String str){
        int[] table = new int[26];
        for (int i=0;i<str.length();i++) {
            table[str.charAt(i) - 'a']++;
        }
        StringBuilder sb = new StringBuilder();
        for (int i=0;i<table.length;i++) {
            sb.append("-").append(table[i]);
        }
        return sb.toString();
    }
}
```

3、知识回顾

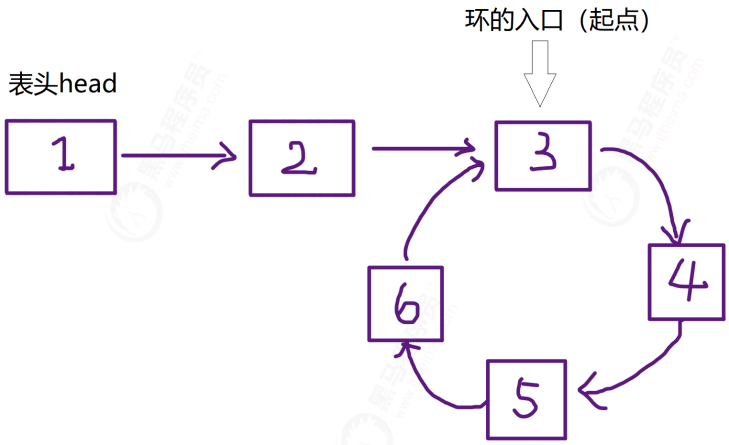
快慢指针解决链表问题

- **1：链表是否有环**

```java
boolean hasCycle(ListNode head) {
    ListNode fast, slow;
    fast = slow = head;
    while (fast != null && fast.next != null) {
        fast = fast.next.next;
        slow = slow.next;

        if (fast == slow) return true;
    }
    return false;
}
```

- **2：已知链表中含有环，返回这个环的入口**




```java
ListNode detectCycle(ListNode head) {
    ListNode fast, slow;
    fast = slow = head;
    while (fast != null && fast.next != null) {
        fast = fast.next.next;
        slow = slow.next;
        if (fast == slow) break;
    }
    // 上面的代码类似 hasCycle 函数
    slow = head;
    while (slow != fast) {
        fast = fast.next;
        slow = slow.next;
    }
    return slow;
}
```

可以看到，当快慢指针相遇时，让其中任一个指针指向头节点，然后让它俩以相同速度前进，再次相遇时所在的节点位置就是环开始的位置

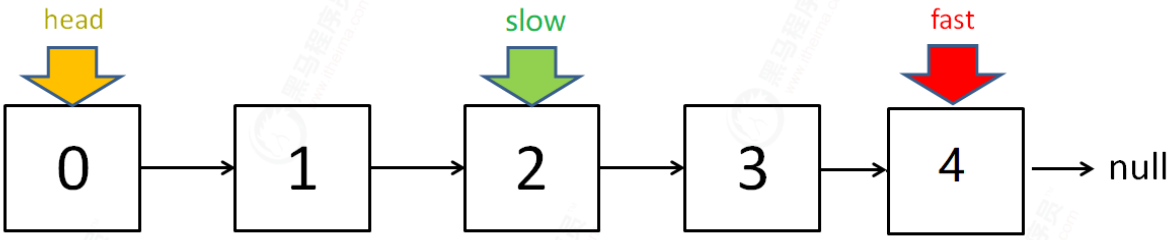
- **3：寻找链表的中点**

876. 链表的中间结点

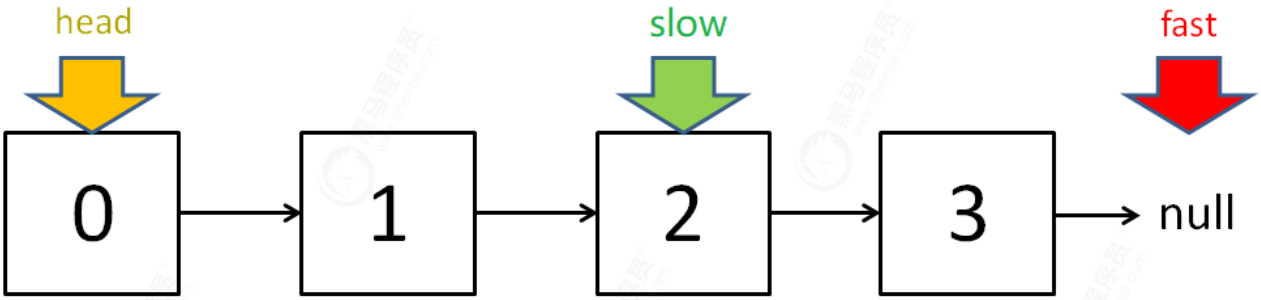
类似上面的思路，我们还可以让快指针一次前进两步，慢指针一次前进一步，当快指针到达链表尽头时，慢指针就处于链表的中间位置。

```
ListNode fast,slow;
fast = slow = head;
while (fast != null && fast.next != null) {
    fast = fast.next.next;
    slow = slow.next;
}
// slow 就在中间位置
return slow;
```

当链表的长度是奇数时，slow 恰巧停在中点位置：



如果长度是偶数，slow 最终的位置是中间偏右：



- **4：寻找链表的倒数第 k 个元素**

剑指 Offer 22. 链表中倒数第k个节点

思路还是使用快慢指针，让快指针先走 k 步，然后快慢指针开始同速前进。这样当快指针走到链表末尾 null 时，慢指针所在的位置就是倒数第 k 个链表节点（为了简化，假设 k 不会超过链表长度）

```
ListNode slow, fast;
slow = fast = head;
while (k-- > 0)
    fast = fast.next;

while (fast != null) {
    slow = slow.next;
    fast = fast.next;
}
return slow;
```

实战题目:234. 回文链表

1：**快慢指针+反转链表**

```
class Solution {
    //快慢指针
    public boolean isPalindrome(ListNode head) {

        //定义快慢指针
        ListNode fast,slow;
        fast = slow = head;
        //反转前半部分链表,定义当前要反转的节点以及前面的节点
        ListNode curr,pre;
        curr = null;
        pre = null;
        //快指针走完，慢指针正好在链表中点,并且反转slow指针走过的节点
        while (fast != null && fast.next != null) {

            curr = slow;

            fast = fast.next.next;
            slow = slow.next;

            curr.next = pre;
            pre = curr;

        }
        //如果链表节点数为奇数
        if (fast != null) {
            slow = slow.next;
        }

        //slow向后走,curr向前走依次比较
        while (slow != null && curr != null) {
            if (slow.val != curr.val) {
                return false;
            }
            slow = slow.next;
            curr = curr.next;
```

```
        }
        return true;
    }
}
```

2：**巧用“栈”**

```
class Solution {
    //使用栈
    public boolean isPalindrome(ListNode head) {
        if (head == null || head.next == null) {
            return true;
        }
        Deque<Integer> stack = new ArrayDeque();
        ListNode curr = head;
        while (head != null) {
            stack.push(head.val);
            head = head.next;
        }
        while (curr != null) {
            if (curr.val != stack.pop()) {
                return false;
            }
            curr = curr.next;
        }
        return true;
    }
}
```