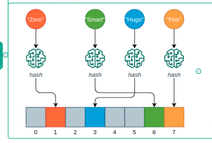
散列表

定义

散列表基于数组实现，并使用哈希算法计算元素在数组中的下标，是一种以空间换时间的数据结构

在二分查找中，我们查找某一个元素的平均时间复杂度为O(logn)，并且要求序列必须有序。而精心设计的散列表，其查找效率可以达到O(1)



组件

- 数组
  - 数组用来保存实际的数据，并且利用数组杀手级的特性: 根据下标寻找数组元素的时间复杂度为O(1)
- 哈希函数
  - 哈希函数用于映射，将一个值通过某种计算方式映射成一个整数，将此结果对数组的容量进行取余，那么得到的结果就是该元素在当前数组的下标
  - 在查询元素时，按照同样的方式来计算该元素在数组的位置，采用下标的方式访问数组元素，时间复杂度为O(1)

以空间换时间

散列表元素获取的时间复杂度固然可达到O(1)，但是这是建立在额外的内存空间之上的

前面提到的哈希函数，需要将自然界中的所有值都映射在一小范围的集合内，那么在映射时必然存在不同的值所映射的结果相同，这种现象称之为哈希冲突

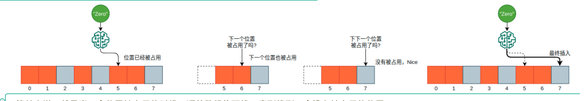
哈希冲突越多，那么在处理冲突时必然要花费更多的资源，哈希表的效率也就会越低。所以，为了尽可能的避免哈希冲突的出现，数组本身的容量会超过数组中实际存储的元素数量

但是浪费一部分空间的代价是值得的，相较于暴力查询，哈希表的元素查找时间复杂度可达O(1)

处理哈希冲突的方式

开放寻址法

开放寻址法的核心思想是，如果出现了散列冲突，我们就重新探测一个空闲位置，将其插入。一个比较简单的方式就是线性探测

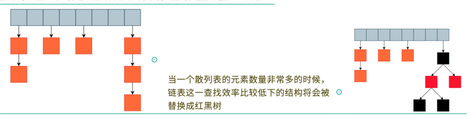


线性探测

- 基本原理
  - 简单来说，就是当一个位置被占用的时候，顺着数组往下找，直到找到一个没有被占用的位置，插入该元素
- 元素的查找
  - 在查找元素时，通过计算得到了元素位置以后，若数组当前位置元素并不是目标元素，需要顺着数组继续往下找，直到找到元素，或者是碰到了未存储过元素的位置，那么此时表示当前值并不存在于哈希表中
- 元素的删除
  - 在开放地址寻址法实现的哈希冲突中，删除元素时不能直接将数组某位置设置为空，因为元素的查找依赖于值为空(NULL)的位置来作为查找的终止
  - 所以，通常会使用一个特殊的标志位，例如一个布尔值，来表示数组某位置元素是否被删除，如此一来不会影响到元素的查找

拉链法

拉链法的核心思想是，数组中每个位置都对应着一个链表，当出现哈希冲突时，往链表里面追加一个元素



基于链表的拉链法

- 基本原理
  - 当一个散列表的元素数量非常多的时候，链表这一查找效率比较低下的结构将会被替换成红黑树
- 元素的查找
  - 查找元素首先还是计算key所在的索引位置，完成后再去对应"槽"中的链表进行查找
- 元素的删除
  - 直接移除掉链表中的元素即可

散列表的扩容

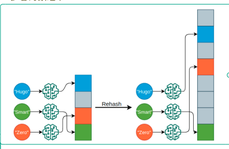
负载因子与扩容时机

- 在散列表中，有一个非常重要的属性: 负载因子。负载因子 = 散列表实际元素数量 / 数组容量
  - 若负载因子大于1，说明至少有一个槽中的元素数量超过1。但是，负载因子小于1并不能保证槽中的元素数量一定是1
- Java语言中HashSet的默认负载因子为0.75，当实际负载因子超过0.75时，底层数组直接扩容至2倍

更快速的扩容

散列表的扩容不等同于数组的扩容，散列表在扩容时需要将原有的元素重新散列至扩容的数组中

index = hash(key) % capicity，capicity为数组容量



- 由于数组容量发生了改变，导致原本的映射关系失效，所以需要重新的对整个散列表重新散列
- 理论上来说，在rehash的过程中散列表的查询、修改以及删除都应被阻塞，直到操作完成

当散列表具有十万甚至百万的元素时，rehash的过程将会非常耗时，并且再次期间客户端无法进行任何操作，必须对其进行优化

使用双写加速rehash过程

基本原理

- 当散列表需要扩容时，首先只执行申请更大数组的操作，不进行数据的迁移，并且两个散列表同时存在
- 元素访问
  - 在访问元素时，首先访问原有散列表
    - 若数据存在则直接返回，并将该元素迁移至新的散列表中
    - 若数据不存在则在新的散列表中进行查询
- 元素添加
  - 添加元素时，直接将元素添加至新的散列表中
- 元素删除、修改
  - 元素的删除与修改需要同时在两个散列表中进行，并且对于修改操作，原有散列表修改完毕后，需将该修改同步至新的散列表中

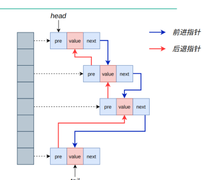
效率

- 使用双写的迁移方式将散列表的rehash成本均摊到了每一个增删改查上，避免了一次性rehash消耗过多的时间，导致客户端长时间的阻塞

散列表+链表

index = hash(key) % capicity，capicity为数组容量 从这个最简单的索引计算公式中可以看出，散列表中的元素顺序是不定的，对散列表进行遍历时，得到的元素顺序和元素的添加顺序可能完全不同

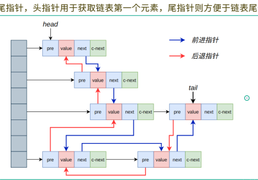
但是，在有些应用场景中，我们希望散列表中的顺序就是元素添加时的顺序，并且此时仍然具有时间复杂为O(1)的元素获取。此时，就需要链表的登场了



有序散列表

在原有的散列表基础之上，再维护一个双向链表即可

- 双向链表中必须添加头指针以及尾指针，头指针用于获取链表第一个元素，尾指针则方便于链表尾部的相关操作
- 散列表在处理冲突可能并不使用拉链法处理，故为虚线指向



- 如果使用拉链法来处理哈希冲突的话，链表元素除了前驱指针和后驱指针以外，还需要存在一个额外的指针，用于链接出现哈希冲突的元素
- 情形稍微复杂一些，不过也只是多了一个指针而已

在遍历散列表时，只需要从双向链表的头指针开始，沿着前进指针的方向前进即可