# 第三章 排序

## 符号表

符号表最主要的目的就是将一个键和一个值联系起来。

定义。符号表是一种存储键值对的数据结构，支持两种操作：插入（put），即将一组新的键值对存入表中；查找（get），即根据给定的键得到相应的值。

### API

符号表是一种个典型的抽象数据类型：它代表一组定义清晰的值以及相应的操作，使得我们能够将类型的实现和使用区分开来。

-1 一种简单的泛型符号表API

|  |  |
| --- | --- |
| Public class ST<Key, Value> |  |
| ST() | 创建一张符号表 |
| void put(Key key, Value value) | 将键值对存入表中（若值为空则将键Key从表中删除） |
| Value get(Key key) | 获取键key对应的值（若值不存在则返回null） |
| void delete(Key key) | 从表中删除键key（及其对应的值） |
| boolean contains(Key key) | 键key在表中是否有对应的值 |
| boolean isEmpty() | 表是否为空 |
| int size() | 表中的键值对数量 |
| Iterable<Key> keys() | 表中所有键的集合 |

#### 泛型

#### 重复的键

我们的所有实现都遵循以下规则：

* 每个键只对应一个值（表中不允许存在重复的键）；
* 当用例代码向表中存入的键值对和表中已有的键（及关联的值）冲突时，新的值会替代旧的值。

#### 空（null）值

键不能为空。

#### 删除操作

删除的实现可以有两种方法：延时删除，也就是将对应键的值置为空，然后在某个时候删去所有值为空的键；即时删除，也就是立刻从表中删除制定的键。

put（key，null）是delete（key）的一种简单（延时型）实现。而实现（即时型）delete（）是为了代替这种默认的方案。在put（）实现的开头有这样一句防御性代码：

if（val==null）{delete（key）；return；}

这保证了符号表中任何键的值都不为空。

#### 便捷方法

#### 迭代

#### 键的等价性

## 二叉查找树

二叉查找树由结点组成，结点包含的链接可以为空或者指向其他结点。在二叉查找树中，每个结点只能有一个服结点（只有一个例外，也就是根结点，它没有

## 散列表

如果所有的键都是小整数，我们可以用一个数组来实现无序的符号表，将键作为数组的索引而数组中键i处储存的就是它对应的值。散列表是这种简易方法的扩展并能够处理更加复杂的类型的键。需要用到算数操作符将键转换为数组的索引来访问数组中的键值对。

使用散列表的查找算法分为两步。第一步是用散列函数将被查找的键转化为数组的一个索引。理想情况下，不同的键都能转化为不同的索引值。散列查找的第二步就是一个处理碰撞冲突的过程，两种解决碰撞的方法：拉链法和线性探测法。

散列表是算法在时间和空间上作出权衡的经典例子。使用散列表，可以实现在一般应用中拥有（均摊后）常数级别的查找和插入操作的符号表。这使得它在很多情况下成为实现简单符号表的最佳选择。

### 散列函数

散列函数的计算会将键转化为数组的索引。如果我们有一个能够保存M个键值对的数组，那么我们就需要一个能够将任意键转化为该数组范围的索引（[0,M-1]范围内的整数）的散列函数。我们要找的散列函数应该易于计算并且能够均匀分布所有的键，即对于任意键，0到M-1之间的每个整数都有相等的可能性与之对应（与键无关）。

散列函数和键的类型有关。严格地说，**对于每种类型的键我们都需要一个与之对应的散列函数**。如果键是一个字符串，我们就需要将这个字符串转化为一个数；如果键含有多个部分，我们需要某种方法将这些部分结合起来。

#### 正整数

将帧数散列最常用的方法是**除留余数法。我们选择大小为素数M的数组，对于任意正整数k，计算k除以M的余数。这个函数能够有效地将键散布在0到M-1的范围内。**

#### 浮点数

如果键是0到1之间的实数，我们可以将它乘以M并四舍五入得到一个0至M-1之间的索引值。这个方法是有缺陷的，因为这种情况下高位起的作用更大，最低位对散列的结果没有影响。

修正这个问题的方法是将键表示为二进制数然后在使用除留余数法。

#### 字符串

除留余数法也可以处理较长的键，例如字符串，我们只需要将它当做大整数即可。

#### 组合键

如果键的类型含有多个整形变量，我们可以和String类型一样将它们混合起来。

#### 软缓存

如果散列值得计算很耗时，那么我们或许可以将每个键的散列值缓存起来，即在每个键中更实用一个hash变量来保存它的hashCode()的返回值。第一次调用hashCode()方法时，我们需要计算对象的散列值，但之后对hashCode()方法的调用会直接返回hash变量的值。

总的来说，要为一个数据类型实现一个优秀的散列方法需要满足三个条件：

* **一致性。**等价的键必须产生相等的散列值；
* **高效性。**计算简便；
* **均匀性。**均匀地散列所在的键。

**假设J（均匀散列假设）。**我们使用的散列函数能够均匀并独立地将所有的键散布于0到M-1之间。

讨论。我们在实现散列函数时随意指定了很多参数，这显然无法实现一个能够在数学意义上均匀并独立地散布所有键的散列函数。艰深的理论研究告诉我们想要找到一个计算简单但又拥有一致性和均匀性的散列函数是不大可能的。

### 基于拉链法的散列表

散列算法的第二步是**碰撞处理**，也就是处理两个或多个键的散列值相同的情况。一种直接的办法是将大小为M的数组中的每个元素指向一条链表，链表中的每个结点都存储了散列值为该元素的索引的键值对。这种方法被称为**拉链法**，因为发生冲突的元素都被存储在链表中。这个方法的基本思想就是选择足够大的M，使得所有链表都尽可能短以保证高效的查找。查找分两步：首先根据散列值找到对应的链表，然后沿着链表顺序查找相应的键。

**命题K。**在一张含有M条链表和N个键的散列表中，任意一条链表中的键的数量均在N/M的常数因子范围内的概率无限趋向于1。

**简略的证明。**有了假设J，这个问题就变成了一个经典的概率论问题。由二项分布可知，一条给定的链表正好含有k个键的概率是：

令，上面公式可以写为：

当很小时，二项分布的一个事件发生次数的概率可以用泊松分布的概率来模拟：

由此可知，一条链表链表中含有超过个键的改路不会超过。对于实际应用来说，这个数字非常小。当一定时，最长链表的平均长度的增长速度是logN/LoglogN。

**性质L。**在一张还有M条链表和N个键的散列表中，未命中查找和插入操作所需的比较次数为。

**例证。**大量应用实例令我们确信，在基于拉链法的散列表中使用大小为M的数组能够将查找和插入操作的效率提高M倍。

### 基于线性探测法的散列表

实现散列表的另一种方式就是用大小为M的数组保存N个键值对，其中M>N。我们需要依靠数组中的**空位**解决碰撞冲突。基于这种策略的所有方法被统称为**开放地址**散列表。

开放地址散列表中最简单的方法叫做**线性探测法**：当碰撞发生时（当一个键的散列值已经被另一个不同的键占用），我们直接检查散列表中的下一个位置（将索引值加1）。这样的线性探测可能会产生三种结果：

* 命中，该位置的键和被查找的键相同；
* 未命中，键为空（该位置没有键）；
* 继续查找，该位置的键和被查找的键不同。

我们用散列函数找到键在数组中的索引，检查其中的键和被查找的键是否相同。如果不同则键继续查找（将索引增大，到达数组结尾时折回数组的开头），直到找到该键护着遇到一个空元素。我们习惯将检查一个数组位置是否含有被查找的键的操作叫做**探测**。

开放地址类的散列表的核心思想是与其将内存用作链表，不如将它们作为在散列表的空元素。这些空元素可以作为查找结束的标志。

插入。如果一个新建的散列值是一个空元素，那么就将它保存在那里；如果不是，我们就顺序查找一个空元素来保存它。查找。从它的散列值开始顺序查找，如果找到则命中，如果遇到空元素则未命中。

#### 删除操作

直接将该键所在的位置设为null是不行的，因为这会使得在此位置之后的元素无法被查找。因此，我们需要将簇中被删除键的右侧的所有键重新插入散列表。

和拉链法一样，开放地址类的散列表的性能也依赖于的比值，但意义有所不同。我们将称为散列表的**使用率**。对于基于拉链法的散列表，是每条链表的长度，因此一般大于1；对于基于线性探测的散列表，是表中已被占用的空间的比例，它是不可能大于1的。事实上，在基于线性探测的散列表中，我们不允许达到1（散列表被占满），因为此时未命中的查找会导致无限循环。为了保证性能，我们会动态调整数组的大小来保证使用率在1/8到1/2之间。

#### 键簇

线性探测的平均成本取决于元素在插入数组后聚集成的一组连续的条目，也叫**键簇**。显然，短小的键簇才能保证较高的效率。随着插入的键越来越多，这个要求很难满足，较长的键会越来越多。

#### 线性探测法的性能分析

命题M。在一张大小为M并含有个键的基于线性探测的散列表中，基于假设J，命中和未命中的查找所需的探测次数分别为：

特别是当约为1/2时，查找命中所需要的探测次数为3/2，未命中所需的约为5/2，当趋近与1时，这些估计的精度会下降，但不需要担心这些情况，因为我们会保证散列表的使用率小于1/2。

讨论。

### 调整数组的大小

#### 拉链法

#### 均摊分析

从理论角度来说，当我们动态调整数组大小时，需要找出均摊成本的上限，因为我们知道使散列表长度加倍的插入操作需要大量的探测。

命题M。假设一张散列表能够自己调整数组的大小，初始为空。基于假设J，执行任意顺序的t此查找、插入和删除操作所需的时间和t成正比，所使用的内存量总是在表中的键的总数的常数因子范围内。

证明。对于拉链法和线性探测法，结合命题K和命题M可知，这个命题只是对我们在第1章中第一次讨论过的数组增长的均摊分析的简单重复而已。