假设你们班级100个同学每个人的学号是由院系-年级-班级和编号组成，例如学号为01100168表示是1系，10级1班的68号。为了快速查找到68号的成绩信息，可以建立一张表，但是不能用学号作为下标，学号的数值实在太大。因此将学号除以1100100取余，即得到编号作为该表的下标，那么，要查找学号为01100168的成绩的时候，只要直接访问表下标为68的数据即可。这就能够在O(1)时间复杂度内完成成绩查找。实际上这里就用到了散列的思想。

# 原理

理想散列表（哈希表）是一个包含关键字的具有固定大小的数组，它能够以**常数时间执行插入，删除和查找操作**。

·每个关键字被映射到0到数组大小N-1范围，并且放到合适的位置，这个映射规则就叫散列函数

·理想情况下，两个不同的关键字映射到不同的单元，然而由于数组单元有限，关键字范围可能远超数组单元，因此就会出现两个关键字散列到同一个值得时候，这就是散列冲突

# 实例演示

通过前面的描述，我们已经了解了一些基本概念，现在来看一个实例。

假设有一个大小为7的表，现在，要将13，18，19，50，20散列到表中。

·选择散列函数，例如使用hash(x)=x%7作为散列函数

·计算数据散列值，并放到合适的位置

计算13 % 7得到6，因此将13放到下标为6的位置：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|  |  |  |  |  |  | 13 |

计算18 % 7得到4，因此将18放到下标为4的位置：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|  |  |  |  | 18 |  | 13 |

计算19 % 7得到5，因此将19放到下标为5的位置：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|  |  |  |  | 18 | 19 | 13 |

计算50 % 7得到1，因此将50放到下标为1的位置：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|  | 50 |  |  |  |  | 13 |

计算20 % 7得到6，因此将20放到下标为6的位置，但是此时6的位置已经被占用了，因此就产生了散列冲突。

将数据散列之后，如何从表中查找呢？例如，查找数值为50的数据位置，只需要计算50 % 7，得到下标1，访问下标1的位置即可。但是如果考虑散列冲突，就没有那么简单了。

通过这个实例，了解了以下几个概念：

·散列函数，散列函数的选择非常重要

·散列冲突，涉及散列表时，因尽量避免散列冲突，对于冲突也要有好的解决方案

·快速从散列表中查找数据

# 冲突解决

解决散列冲突通常有以下几种方法：

拉链法

开放定址法

再散列

…

## 拉链法

分离链接法的做法是将同一个值的关键字保存在同一个表中。例如，对于前面：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|  | 50 |  |  | 18 | 19 | 13 |

如果再要插入元素20，则在下标为6的位置存储表头，而表的内容是13和20。

这种方法的特点是需要另外分配新的单元来存储散列到同一个位置的数据。

查找的时候，除了根据计算出来的散列值找到对应位置外，还需要在链表上进行搜索。而在单链表上的查找速度是很慢的。另外散列函数如果设计得好，冲突的概率其实也会很小。

## 开放定址法

开放定址法的思想是，如果冲突发生，就选择另外一个可用的位置。

开放定址法中也有常见的几种策略。

### 线性探测法

还是以前面的为例：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|  | 50 |  |  | 18 | 19 | 13 |

如果此时再要插入20，则20 % 7 = 6，但是6的位置已有元素，因此探测下一个位置（6+1）%7，在这里就是下标为0的位置。因此20的存储位置如下：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 20 | 50 |  |  | 18 | 19 | 13 |

但这种方式的一个问题是，可能造成一次聚集，因为一旦冲突发生，为了处理冲突就会占用下一个位置，而如果冲突较多时，就会出现数据都聚集在一块区域。这样就会导致任何关键字都需要多次尝试才可能解决冲突。

### 平方探测法

顾名思义，如果说前面的探测函数是F（i）= i % 7，那么平方探测法就是F（i）= (i^2 )% 7。但是这也同样会产生二次聚集问题。

### 双散列

为了避免聚集，在探测时选择跳跃式的探测，即再使用一个散列函数，用来计算探测的位置。假设前面的散列函数为hash1(X)，用于探测的散列函数为hash2(X)，那么一种流行的选择是F(i) = i \* hash2(X)，即第一次冲突时探测hash1(X)+hash2(X)的位置，第二次探测hash1(X)+2hash2(X)的位置。

可以看到，无论是哪种开放定址法，它都要求表足够大。

## 再散列

散列表可以认为是具有固定大小的数组，那么如果插入新的数据时散列表已满，或者散列表所剩容量不多该怎么办？这个时候就需要再散列，常见做法是，建立一个是原来两倍大小的散列表，将原来表中的关键字重新散列到新表中。

# 应用

散列表应用很广泛。例如做文件校验或数字签名。当然还有快速查询功能的实现。例如，redis中的字典结构就使用了散列表，使用MurmurHash算法来计算字符串的hash值，并采用拉链法处理冲突，，当散列表的装载因子（关键字个数与散列表大小的比）接近某个大小时，进行再散列。

下面主要介绍哈希表在数据库中的应用：

## 哈希查找

哈希查找，由于哈希算法能够很好的把数据做比较均匀的分布，所以哈希查找的速度要比B+ Tree查找的速度要快。哈希查找的时间复杂度是O(1)，而B+ Tree的时间复杂度是O(h)。当然，B+ Tree的优势是在范围查找，不然也不会成为关系型数据库的基本算法了。

SQL Server在做联接(JOIN)的时候，如果两个表都不是很小，而且没有在关联列上排序，就很可能使用哈希联接。哈希联接使用的就是哈希查找，它的性能并不差，但是要注意的一点是，哈希联接需要先构造一个哈希表，而哈希表需要消耗不小的内存空间，如果数据库服务器的内存不足的话，SQL Server就只好使用“优雅的哈希联接”(Grace HASH JOIN)或者递归哈希联接(Recursive Hash Join)，这样性能就会受到影响了。在设计数据库的时候，我们应该注意建立/更新适当的索引和统计信息(STATISTICS)，以便SQL Server可以准确的估计联接的输入大小，以便选择正确的算法。

在使用哈希联接的时候，SQL Server的查询优化器会选择算法，通常我们并不需要做任何的指令，编写任何代码。但是在其它一些需要用到哈希算法的时候，就需要认真的选择算法，甚至可能需要自己写哈希算法的代码。

在业务系统中，可能生成的业务键（Business key）会比较长，例如某电商网站的订单号会类似这样：202210782169AC7G。这样的业务键如果用作主键的话，会占据16字节，显得有点浪费。如果只是单一系统使用，可以考虑用自动增量的数字作为主键。但是如果在多个系统中使用，比如在一个数据仓库系统中，使用自动增量作为代理键(Surrogate key)，就必须在处理事实表的ETL过程中，用新的自动增量替换掉订单表和详单表中的订单号，这会带来两个大表的JOIN，是一个相当耗时的操作。在这种情况下，我们就可以考虑使用哈希算法来生成代理键，只需要在订单表和详单表都使用同样的哈希算法，就可以保证得到的代理键是可以正确联接的。

## 一致性哈希

# 总结

一个设计良好的散列表能够几乎在O(1)时间复杂度内完成插入，删除和查找，但前提是散列函数设计得足够优雅，以及有着合适散列冲突解决方案。常见冲突解决方案有：

拉链法

开放地址检测法

其中拉链法在实际中是很常见的一种解决方案。

参考：

《数据结构与算法分析》

《redis设计与实现》

https://en.wikipedia.org/wiki/Hash\_table

# 试题

## 最小包含字符窗口

题目要求：

给定一个字符串S（源字符串）和一个字符串T（目标字符串），使用O(n)的时间复杂度在S中找到一个最小的窗口大小，使得该窗口包含T中的所有字符。