27 第十一讲 散列查找

笔记本:	浙江大学《数据结构》
创建时间:	2025/5/26 20:50	**更新时间:**	2025/5/27 20:02
作者:	panhengye@163.com
URL:	https://www.doubao.com/chat/7107353504077570

散列表

散列（Hashing）的基本思想
①以关键字key为自变量，通过一个确定的函数h(散列函数），计算出对应的函数值h(key)，作为数据对象的存储地址
②可能不同的关键字会映射到同一散列地址上，称为“冲突”(Collision) —— 需要某种冲突解决策略

引入

编译处理时，涉及变量及属性的管理：

- 插入：新变量定义
- 查找：变量的引用

而编译中是一个动态查找问题

> 查找的本质：已知对象找位置

- 有序安排对象
    - 全序：用二分查找
    - 半序：查找树
- 直接“算出”对象位置：散列

散列查找的两项基本工作

- 计算位置：**构造散列函数**确定关键词存储位置
- 解决冲突：应用某种策略解决多个关键词**位置相同**的问题

> 时间复杂度几乎是常量

散列表（哈希表）[Hash Table]

基本特征

- 类型名称：符号表（Symbol Table）
- 数据对象集：名字（Name）- 属性（Attribute）对的集合
- 操作集
    - 创建一个长度为TableSize的符号表
    - 查找特定Name是否在表中
    - Find:获取指定Name对应的属性
    - 将指定Name的属性修改为Attr
    - Insert:向Table中插入一个新名字及其属性
    - Delete:从Table中删除一个名字及其属性

[例] 有n = 11个数据对象的集合{18，23，11，20，2，7，27，30，42，15，34}。

符号表的大小用TableSize = 17，选取散列函数h如下：

h(key) = key mod TableSize （求余）

地址	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
关键词	34	18	2	20			23	7	42		27	11		30		15	

- 存放：

  h(18)=1， h(23)=6， h(11)=11，h(20)=3，h(2)=2, …….

  如果新插入**35**， **h(35)=1,** 该位置已有对象！冲突！！

- 查找:

  - **key = 22，h(22)= 5，该地址空，不在表中**
  - **key = 30，h(30)= 13，该地址存放是30，找到！**

装填因子（Loading Factor）：
设散列表空间大小为m，填入表中元素个数是n，则称α = n / m为散列表的装填因子

散列函数的构造方法

一个好的散列函数一般应考虑以下两个因素

- 计算简单，以便提高转换速度
- 关键词对应的**地址空间分布均匀**，以便减少冲突

数字关键词的散列函数

1. 直接定址法
   1. 取关键词的某个线性函数值为散列地址
   2. h(key) = a * key + b
   3. 例子:

地址h(key)	出生年份(key)	人数(attribute)
0	1990	1285万
1	1991	1281万
2	1992	1280万
…	……	……
10	2000	1250万
…	……	……
21	2011	1180万

h(key)=key-1990

2. 除留余数
   1. h(key) = key mod p
   2. p一般是表的大小，同时为了均匀，一般取**素数**
   3. 例子:

例： h(key) = key % 17

地址 h(key)	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
关键词 key	34	18	2	20			23	7	42		27	11		30		15	

3. 数字分析法
   1. 分析数字关键字在各位上的变化情况，取比较随机的位作为散列地址
   2. 例子

如果关键词 key 是18位的身份证号码：

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	3	0	1	0	6	1	9	9	0	1	0	0	8	0	4	1	9
省		市		区（县）下属辖区编号		（出生）年份				月份		日期		该辖区中的序号			校验

$$h_1(\text{key}) = (\text{key}[6]-'0')\times 10^4 + (\text{key}[10]-'0')\times 10^3 + (\text{key}[14]-'0')\times 10^2 + (\text{key}[16]-'0')\times 10 + (\text{key}[17]-'0')$$

$h(\text{key}) = h_1(\text{key})\times 10 + 10$ （当 key[18] = 'x'时）

或 $= h_1(\text{key})\times 10 + \text{key}[18]-'0'$ （当 key[18] 为'0'~'9'时）

4. 折叠法
    1. 把关键词分割成位数相同的几个部分，然后叠加
    2. 例子：

如： 56793542

```
  542
  793
+ 056
-----
 1391
```

h(56793542) = 391

5. 平方取中法
    1. 例子

如： 56793542

```
  56793542
x 56793542
----------
3225506412905764
```

h(56793542) = 641

字符关键词的散列函数

1. ASCII码加和法
2. 简单的改进——前3个字符移位法
3. 好的散列函数——移位法（公式如下所示）

$$h(\text{key}) = \left(\sum_{i=0}^{n-1} \text{key}[n-i-1] \times 32^i\right) \bmod \text{TableSize}$$

思考一下：为什么先要把哈希值做成一个很大数，再用求余操作获得一个很小的数?
①将不同的 key 尽可能均匀地映射到更广泛的取值空间——减少冲突
②用求余操作将大的哈希值映射到有限大小（TableSize ）的哈希表槽位中——便于存放

伪代码描述

```
Index Hash ( const char *Key, int TableSize ){     unsigned int h = 0;     /* 散列函数值，初始化为0 */     while ( *Key !=  '\0' )  /* 位移映射 */          h = ( h << 5 ) + *Key++;     return   h % TableSize;}
```

冲突处理方法

常见思路:

- 换个地址：开放地址法
- 同一位置的冲突对象组织在一起：链地址法

开放地址法（Open Addressing）

一旦产生了冲突（该地址已有其它元素），就按某 种规则去寻 找另一空地址

> 如果发生冲突，则增加一个偏移量di

d_i决定了不同的解决冲突方案：线性探测、平方探测、双散列

线性探测（Linear Probing）

线性探测法：以增量序列 1，2，……，（TableSize -1） 循环试探下一个存储地址

操作 \ 地址	0	1	2	3	4	5	6	7	8	9	10	11	12	说明
插入47				47										无冲突
插入7				47				7						无冲突
插入29				47				7	29					$d_1 = 1$
插入11	11			47				7	29					无冲突
插入9	11			47				7	29	9				无冲突
插入84	11			47				7	29	9	84			$d_3 = 3$
插入54	11			47				7	29	9	84	54		$d_1 = 1$
插入20	11			47				7	29	9	84	54	20	$d_3 = 3$
插入30	11	30		47				7	29	9	84	54	20	$d_6 = 6$

线性探测存在聚集现象

散列表查找性能分析

- 成功查找长度（ASLs）
- 不成功查找长度（ASLu）

【分析】

ASLs：查找表中关键词的平均查找比较次数（其冲突次数加1）

ASL s= （1+7+1+1+2+1+4+2+4）/ 9 = 23/9 ≈ 2.56

ASLu：不在散列表中的关键词的平均查找次数（不成功）

一般方法：将不在散列表中的关键词分若干类。

如：根据H(key)值分类

ASL u= （3+2+1+2+1+1+1+9+8+7+6）/ 11 = 41/11 ≈ 3.73

平方探测法（Quadratic Probing）

❖ 平方探测法：以增量序列1^2，-1^2，2^2，-2^2，……，q^2，$-q^2$ 且$q \le \lfloor TableSize/2 \rfloor$循环试探下一个存储地址。

平方探测的一个缺陷是，有可能出现频繁跳跃但就是找不到空位的情况

有定理显示：如果散列表长度 TableSize 是某个 4k+3 (k是正整数) 形式的**素数**时，平方探测法就可以探查到整个散列表空间

在开放地址散列表中，删除操作要很小心。通常只能 "**懒惰删除**"，即需要增加一个 "删除标记(Deleted)"，而并不是真正删除它。以便查找时不会 "断链"。其空间可以在下次插入时重用。

双散列探测法（Double Hashing）

探测序列还应该保证所有的散列存储单元都应该能够被探测到。

选择以下形式有良好的效果：

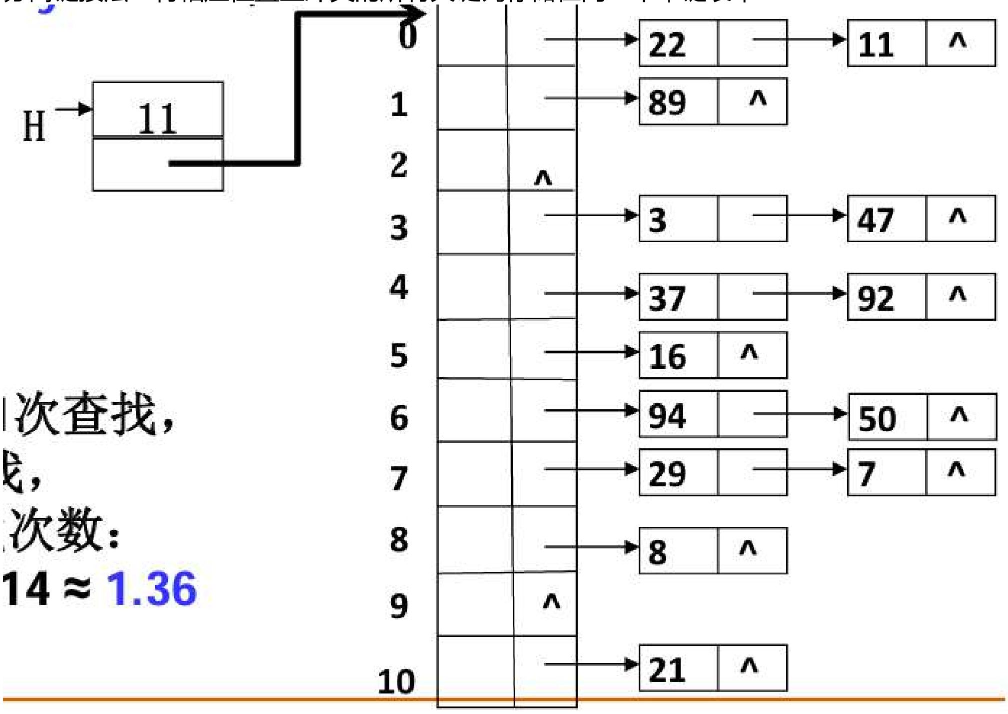
$$h_2(key) = p - (key \bmod p)$$

再散列（Rehashing）

当散列表元素太多（即装填因子 α太大）时，查找效率会下降，实用最大装填因子一般取 0.5 <= α <= 0.8
当装填因子过大时，解决的方法是加倍扩大散列表，这个过程叫 做“再散列”（Rehashing）
散列表扩大时，原有元素需要重新计算放置到新表中

分离链接法（Separate Chaining）

分离链接法：将相应位置上冲突的所有关键词存储在同一个单链表中



散列表的性能分析

关于散列表的一些认识：
①选择合适的 h(key)，散列法的查找效率期望是关键字的空间的大小n无关，适合于关键字直接比较计算量大的问题
②以较小的α为前提：**以空间换时间**
③散列方法的存储对关键字是**随机**的，不便于**顺序查找**关键字， 也不适合于**范围查找**，或**最大值最小值**查找

- 平均查找长度（ASL）用来度量散列表查找效率：成功、不成功
- 关键词的比较次数，取决于产生冲突的多少，影响产生冲突多少有以下三个因素：
  - 散列函数是否均匀
  - 处理冲突的方法
  - 散列表的装填因子α

线性探查法的查找性能

可以证明，线性探测法的期望探测次数 满足下列公式：

$$
p=\begin{cases} \dfrac{1}{2}\left[1+\dfrac{1}{(1-\alpha)^2}\right] & （对插入和不成功查找而言） \\ \dfrac{1}{2}\left(1+\dfrac{1}{1-\alpha}\right) & （对成功查找而言） \end{cases}
$$

对于线性探测，如果当前装填因子值为0.654321，此时不成功情况下的期望探测次数小于成功情况下的期望探测次数。

- A. ✔
- B. ✕

针对上图小测验的计算：

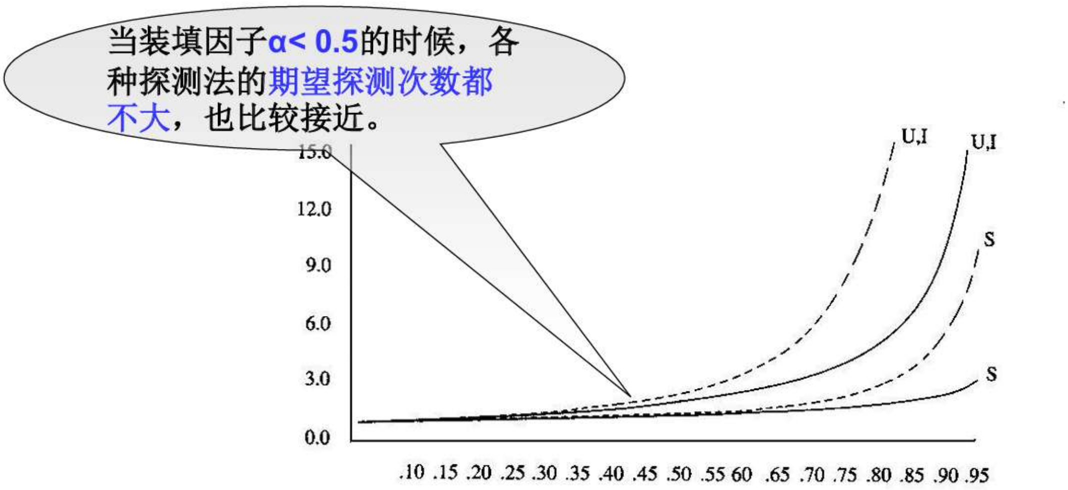
```
def calculate_p(alpha):     # 不成功的期望     p1 = 0.5 * (1 + 1 / ((1 - alpha)
** 2))     # 成功期望     p2 = 0.5 * (1 + 1 / (1 - alpha))     return p1, p2 def
main():     alpha = 0.654321     p1, p2 = calculate_p(alpha)     print(p1 - p2 <
0) if __name__ == '__main__':     main()
```

输出结果为：False

平方探测法和双散列探测法

可以证明，平方探测法和双散列探测法探测次数 满足下列公式：

$$
p=\begin{cases} \dfrac{1}{1-\alpha} & （对插入和不成功查找而言） \\ -\dfrac{1}{\alpha}\ln(1-\alpha) & （对成功查找而言） \end{cases}
$$



线性探测法（虚线）、双散列探测法（实线）
U表示不成功查找，I表示插入，S表示成功查找

分离链接法

所有地址链表的平均长度定义成装填因子α，α有可能超过1。
不难证明：其期望探测次数 p为：

$$p=\begin{cases}\alpha+e^{-\alpha} & \text{（对插入和不成功查找而言）}\\ 1+\dfrac{\alpha}{2} & \text{（对成功查找而言）}\end{cases}$$

开放地址法和分离链接法的对比

比较维度	开放地址法	分离链接法
存储结构	散列表是一个数组	散列表是顺序存储和链式存储的结合
优点	存储效率高，支持随机查找	关键字删除不需要 “懒惰删除” 法，无存储 “垃圾” 问题
缺点	存在 “聚集” 现象	链表部分存储和查找效率较低；α 值不合理时，可能空间浪费或时间代价高；链表长度不均匀会严重降低时间效率