


# 课前签到

数据结构与算法2023秋季



 微信扫一扫，使用小程序

1. 微信扫码+**实名**

2. 点击今日签到

签到时间：

9:45~10:15

签到地方：

珠海校区-教学

大楼-C407

人脸识别；智能定位

# 上课纪律



手机静音

上课期间手机静音：

1. 关闭手机

2. 飞行模式



# 数据结构与算法 散列

余建兴

中山大学人工智能学院

## 提纲

- 1 散列表
- 2 散列函数构造
- 3 冲突处理
- 4 应用实例

## 5.1 散列表

### 散列表意义



# 散列表

C语言中变量名必须：  
先定义（或者声明）后再使用

```
int n;                int *p
...                   ...
n=n+2;               p=p*2;
...                   ...
```

编译处理时，涉及变量及属性（如：变量类型）的管理  
插入：新变量定义  
查找：变量的引用

编译处理中对变量管理： 利用查找树（搜索树）进行变量管理？  
动态查找问题 两个变量名（字符串）比较效率不高

是否可以把字符串转换为数字，再处理？

## 已知的查找方法

顺序查找  $O(N)$

二分查找（静态查找）  $O(\log_2 N)$

二叉搜索树  $O(h)$   $h$  为二叉查找树的高度

$O(\log_2 N)$  已经是相当不错的时间复杂度！

到底还有没有其他**适应性广**  
而**速度又快**的查找方法呢？

# 已知的查找方法

**[例]** 在登录QQ的时候，QQ服务器如何核对你的身份，以确定你就是该号码的主人？

**[分析]** 看看是否可以用二分法查找。

- 十亿 ( $10^9 \approx 2^{30}$ ) 有效用户，用二分查找30次。 ✓
- 十亿 ( $10^9 \approx 2^{30}$ )  $\times$  1K  $\approx$  1024G，1T连续空间。 ✓
- 按有效QQ号大小有序存储：在连续存储空间中，插入和删除一个新QQ号码将需要移动大量数据。 ✗

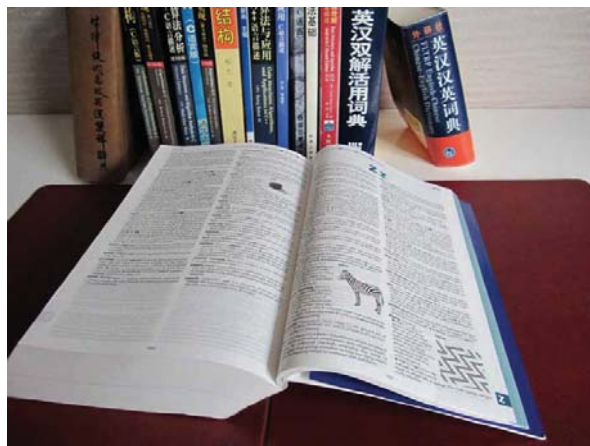
用不了二分查找，  
我们该怎么办？



# 已知的查找方法

**[例]** 查英文字典的过程——查询英文单词 “zoo”，你为什么不用二分法，而直接从字典的后面找？

- 我们已经根据要查找的关键词 “zoo” 在脑子里经过了 “计算”，得出该关键词所在的大致位置，这样就能更快地找到它。这个 “计算” 过程非常类似于本章将要介绍的散列查找中的 “散列函数计算”。
- 查字典的过程结合了散列查找（用于初步定位）、二分查找（一般不是准确二分）和顺序查找（当很接近关键词的时候）等几种查找方法。



# 已知的查找方法

**[例]** 网上搜索。搜索引擎是如何如此神速地把我们需要的有关信息找到的？

**[分析]** 主要数据结构是“倒排索引” -- “单词到文档”的映射关系。



Docs	文档 <sub>1</sub>	文档 <sub>2</sub>	文档 <sub>3</sub>	文档 <sub>4</sub>	文档 <sub>m</sub>
关键词 <sub>1</sub>	3: 1,12,20	0	.....	2: 1, 22	3: 9,40,52
关键词 <sub>2</sub>	0	2: 11,22	.....	4: 9,20,32,65	5: 5,9,10,32,35
.....	.....	.....	.....	.....	.....
关键词 <sub>n-1</sub>	0	0	.....	5: 3,9,10,32,56	10: 5,6,19, ..,44
关键词 <sub>n</sub>	5: 1,9,20,22,55	0	.....	0	1: 7

# 已知的查找方法

**[问题]** 如何快速搜索到需要的关键词？如果关键词不方便比较怎么办？

- **查找的本质: 已知对象找位置**
  - 有序安排对象：全序、半序
  - 直接“算出”对象位置：散列
- **散列查找法的两项基本工作：**
  - **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： $O(1)$ ，即查找时间与问题规模无关！**

# 散列表Demo

## 概念

散列表（哈希表）：通过给定的关键字的值直接访问到具体对应的值的一个数据结构

作用：用关键字的值直接访问记录，以加快访问速度。



## 散列表（哈希）

- **类型名称**：符号表 (SymbolTable)
- **数据对象集**：符号表是“名字(Name)-属性(Attribute)”对的集合。
- **操作集**：对于一个符号表  $Table \in \text{SymbolTable}$ ，一个给定名字  $Name \in \text{NameType}$ ，属性  $Attr \in \text{AttributeType}$ ，以及正整数  $TableSize$ ，符号表的基本操作主要有：
  - 1、**SymbolTable InitializeTable( int TableSize )**：创建一个长度为  $TableSize$  的符号表；
  - 2、**Boolean IsIn( SymbolTable Table, NameType Name )**：查找特定的名字  $Name$  是否在符号表  $Table$  中；
  - 3、**AttributeType Find( SymbolTable Table, NameType Name )**：获取  $Table$  中指定名字  $Name$  对应的属性；
  - 4、**SymbolTable Modify( SymbolTable Table, NameType Name, AttributeType Attr )**：将  $Table$  中指定名字  $Name$  的属性修改为  $Attr$ ；
  - 5、**SymbolTable Insert( SymbolTable Table, NameType Name, AttributeType Attr )**：向  $Table$  中插入一个新名字  $Name$  及其属性  $Attr$ ；
  - 6、**SymbolTable Delete( SymbolTable Table, NameType Name )**：从  $Table$  中删除一个名字  $Name$  及其属性。



# 散列表（哈希）

**[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 $\text{TableSize} = 17$ ，选取散列函数 $h$ 如下：  
 $h(\text{key}) = \text{key} \bmod \text{TableSize}$ （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：

$h(18)=1, h(23)=6, h(11)=11, h(20)=3, h(2)=2, \dots\dots$

如果新插入35， $h(35)=1$ ，该位置已有对象！**冲突！！**

- 查找：

- $\text{key} = 22, h(22) = 5$ ，该地址空，不在表中

- $\text{key} = 30, h(30) = 13$ ，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 $\alpha = n / m$ 为散列表的装填因子

$\alpha = 11 / 17 \approx 0.65$ 。

# 散列表（哈希）

**[例]** 将给定的10个C语言中的关键词(保留字或标准函数名)顺次存入一张散列表。这10个关键词为：**acos**、**define**、**float**、**exp**、**char**、**atan**、**ceil**、**floor**、**clock**、**ctime**。散列表设计为一个二维数组 $\text{Table}[26][2]$ ，2列分别代表2个槽。

如何设计**散列函数**，使得发生**冲突的概率**尽可能小？

装填因子  $\alpha = ?$

为了把字母  $a \sim z$  映射到  $0 \sim 25$ ，如何设计散列函数  $h(\text{key}) = ?$   $h(\text{key}) = \text{key}[0] - 'a'$

当**冲突或溢出**不可避免时，**如何**  
**处理**使得表中没有空单元被浪费，  
同时**插入**、**删除**、**查找**等操作都  
能正确完成？

	槽 0	槽 1
0	acos	atan
1		
2	char	ceil
3	define	
4	exp	
5	float	floor
6		
.....		
25		

如果没有溢出，

$T_{\text{查询}} = T_{\text{插入}} = T_{\text{删除}} = O(1)$



# 散列表（哈希）

---

- “散列（Hashing）”的基本思想是：
  - 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。
  - 可能不同的关键字会映射到同一个散列地址上，即  $h(key_i) = h(key_j)$  (当  $key_i \neq key_j$ )，称为“冲突 (Collision)”。----需要某种冲突解决策略

## 休息时间

---



第13节休息

## 5.2 散列函数的构造方法

### 散列函数的构造方法

- 一个“好”的散列函数一般应考虑下列两个因素：
  - 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，以尽量减少冲突。
- 数字关键词的散列函数构造
  - 1. 直接定址法

取关键词的某个线性函数值为散列地址，即

$$h(\text{key}) = a \times \text{key} + b \quad (a、b \text{ 为常数})$$

$$h(\text{key}) = \text{key} - 1990$$

地址 $h(\text{key})$	出生年份 $(\text{key})$	人数 $(\text{attribute})$
0	1990	1285万
1	1991	1281万
2	1992	1280万
...	.....	.....
10	2000	1250万
...	.....	.....
21	2011	1180万

# 散列函数的构造方法

## 2. 除留余数法

散列函数为： $h(key) = key \bmod p$

例5.4:  $h(key) = key \% 17$

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	

- 这里： $p = \text{TableSize} = 17$ 。也可以采用  $p \neq \text{TableSize}$ ；
- $\text{TableSize} = n/\alpha$ ； $n$ --key集合的大小， $\alpha$ 装填因子上限；
- $p \leq \text{TableSize}$ 的某个最大素数。

TableSize	8	16	32	64	128	256	512	1024
p	7	13	31	61	127	251	503	1019

# 散列函数的构造方法

## 3. 数字分析法

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

比如：取11位手机号码key的后4位作为地址：

散列函数为： $h(key) = \text{atoi}(key+7)$  (char \*key)

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	

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

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

或  $h(key) = h_1(key) \times 10 + key[18] - '0'$  (当  $key[18]$  为  $'0' \sim '9'$  时)

# 散列函数的构造方法

- 4. 折叠法

把关键词分割成位数相同的几个部分，然后叠加

如：56793542

542  
793  
+ 056  
-----  
1391

$h(56793542) = 391$

- 5. 平方取中法

如：56793542

56793542  
x 56793542  
-----

3225506412905764

$h(56793542) = 641$

## 字符关键词的散列函数构造

- 1. 一个简单的散列函数——ASCII码加和法

对字符型关键词key定义散列函数如下：

$$h(\text{key}) = (\sum \text{key}[i]) \bmod \text{TableSize}$$

冲突严重：a3、b2、  
c1；eat、tea；

- 2. 简单的改进——前3个字符移位法

改造如下：

$$h(\text{key}) = (\text{key}[0] + \text{key}[1] \times 27 + \text{key}[2] \times 27^2) \bmod \text{TableSize}$$

- 3. 好的散列函数——移位法

涉及关键词的所有n个字符，并且

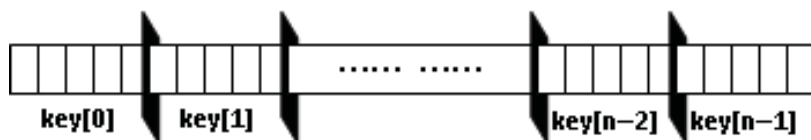
仍然冲突：string、street、  
strong、structure等等；  
空间浪费： $3000/26^3 \approx 30\%$

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

# 如何快速计算

$h("abcde") = 'a' * 32^4 + 'b' * 32^3 + 'c' * 32^2 + 'd' * 32 + 'e'$

- 每位字符占5位二进制（即 $2^5 = 32$ ）。实现时不需要做乘法运算，只需一次左移5位来完成。这也是为什么选用32来代替27的原因。



```
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;
}
```

对32位字长的无符号整数，只有7个字符起作用： $32 \leq 7 \times 5$ 。不过在实际应用中已经不错了。

## 5.3 散列冲突处理方法

# 处理冲突的办法

- 常用处理冲突的思路：
  - 换个位置：开放地址法
  - 同一位置的冲突对象组织在一起：链地址法
- 开放定址法 ( 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找另一空地址

## 开放定址法

**[定义]** 所谓开放定址法，就是一旦产生了冲突，即该地址已经存放了其它数据元素，就去寻找另一个空的散列地址。

- 若发生了第  $i$  次冲突，试探的下一个地址将增加  $d_i$ ，基本公式是：

$$h_i(\text{key}) = (h(\text{key}) + d_i) \bmod \text{TableSize} \quad (1 \leq i < \text{TableSize})$$

在没有装满的散列表中，  
空的散列地址是否总能找到？

- $d_i$  决定了不同的解决冲突方案：线性探测、二次探测、双散列。

$$d_i = i$$

$$d_i = \pm i^2$$

$$d_i = i * h_2(\text{key})$$

# 1. 线性探测法

- 即线性探测法以增量序列  $1, 2, \dots, (\text{TableSize} - 1)$  循环试探下一个存储地址。

[例] 设关键词序列为  $\{47, 7, 29, 11, 9, 84, 54, 20, 30\}$ ,

- 散列表表长  $\text{TableSize} = 13$ ,
- 装填因子  $\alpha = 9/13 \approx 0.69$ ;
- 散列函数为： $h(\text{key}) = \text{key} \bmod 11$ 。
- 用线性探测法处理冲突，列出依次插入后的散列表，
- 并估算查找性能。

关键词 (key)	47	7	29	11	9	84	54	20	30
散列地址 $h(\text{key})$	3	7	7	0	9	7	10	9	8

# 1. 线性探测法

关键词 (key)	47	7	29	11	9	84	54	20	30
散列地址 $h(\text{key})$	3	7	7	0	9	7	10	9	8
冲突次数	0	0	1	0	0	3	1	3	6

问题：如何删除关键字“7”？  
“一次聚集 (Primary Clustering)”现象：需要经过很多次冲突才找到空位置。

地址	0	1	2	3	4	5	6	7	8	9	10	11	12	说明
操作														
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	$d_6 = 6$

注意“聚集”现象



# 1. 线性探测法

- 成功平均查找长度 (ASLs)
- 不成功平均查找长度 (ASLu)

散列表：

H(key)	0	1	2	3	4	5	6	7	8	9	10	11	12
key	11	30		47				7	29	9	84	54	20
冲突次数	0	6		0				0	1	0	3	1	3

[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）

$$ASLs = (1 + 7 + 1 + 1 + 2 + 1 + 4 + 2 + 4) / 9 = 23/9 \approx 2.56$$

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

$$ASLu = (3 + 2 + 1 + 2 + 1 + 1 + 1 + 9 + 8 + 7 + 6) / 11 \\ = 41/11 \approx 3.73$$

# 1. 线性探测法

[例] 将acos、define、float、exp、char、atan、ceil、floor

顺次存入一张大小为26的散列表中。

$H(key) = key[0] - 'a'$ ，采用线性探测 $d_i = i$ 。

acos	atan	char	define	exp	float	ceil	floor		.....	
0	1	2	3	4	5	6	7	8		25

[分析]

ASLs：表中关键词的平均查找比较次数

$$ASLs = (1 + 1 + 1 + 1 + 1 + 2 + 5 + 3) / 8 = 15/8 \approx 1.87$$

ASLu：不在散列表中的关键词的平均查找次数（不成功） 根据

H(key)值分为26种情况：H值为0,1,2,...,25

$$ASLu = (9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 * 18) / 26 \\ = 62/26 \approx 2.38$$

## 2. 平方探测法

- 即平方探测法以增量序列  $1^2, -1^2, 2^2, -2^2, \dots, q^2, -q^2$  且  $q \leq \lfloor \text{TableSize}/2 \rfloor$  循环试探下一个存储地址。

[例] 设关键词序列为 {47, 7, 29, 11, 9, 84, 54, 20, 30}

- 散列表表长  $\text{TableSize} = 11$  (即满足  $4 \times 2 + 3$  形式的素数)
- 装填因子  $\alpha = 9/11 \approx 0.82$
- 散列函数为:  $h(\text{key}) = \text{key} \bmod 11$
- 用平方探测法处理冲突, 列出依次插入后的散列表
- 并估算ASLs

关键词 key	47	7	29	11	9	84	54	20	30
散列地址 $h(\text{key})$	3	7	7	0	9	7	10	9	8

## 2. 平方探测法

关键词 key	47	7	29	11	9	84	54	20	30
散列地址 $h(\text{key})$	3	7	7	0	9	7	10	9	8

“二次聚集 (Secondary Clustering)” 现象: 散列到同一地址的那些数据对象将探测相同的备选单元。

$$\text{ASL}_s = (1 + 1 + 2 + 1 + 1 + 3 + 1 + 4 + 4) / 9 = 18/9 = 2$$

地址 操作	0	1	2	3	4	5	6	7	8	9	10	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			84	7	29	9		$d_2 = -1$
插入54	11			47			84	7	29	9	54	无冲突
插入20	11		20	47			84	7	29	9	54	$d_3 = 4$
插入30	11	30	20	47			84	7	29	9	54	$d_3 = 4$

## 2. 平方探测法

是否有空间，平方探测(二次探测)就能找得到？

5	6	7		
0	1	2	3	4

$$h(k) = k \bmod 5$$

插入 11,  $h(11)=1$

探测序列:  $1 + 1 = 2$ ,  $1 - 1 = 0$ ,  $(1 + 2^2) \bmod 5 = 0$ ,  
 $(1 - 2^2) \bmod 5 = 2$ ,  $(1 + 3^2) \bmod 5 = 0$ ,  $(1 - 3^2) \bmod 5 = 2$ ,  
 $(1 + 4^2) \bmod 5 = 2$ , ...

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。

## 2. 平方探测法

```
HashTable InitializeTable( int TableSize ) {
    HashTable H;
    int i;
    /* 1*/ if ( TableSize < MinTableSize ){
    /* 2*/     Error( "散列表太小" );
    /* 3*/     return NULL;
    }
    /* 4*/ /* 分配散列表 */
    /* 5*/ H = malloc( sizeof( struct HashTbl ) );
    /* 6*/ if ( H == NULL )
    /* 7*/     FatalError( "空间溢出!!!" );
    /* 8*/ H->TableSize = NextPrime( TableSize );
    /* 9*/ /* 分配散列表 Cells */
    /*10*/ H->TheCells = malloc( sizeof( Cell ) * H->TableSize );
    /*11*/ if( H->TheCells == NULL )
    /*12*/     FatalError( "空间溢出!!!" );
    /*13*/ for( i = 0; i < H->TableSize; i++ )
    /*14*/     H->TheCells[ i ].Info = Empty;
    return H;
}
```

**typedef struct HashTbl \*HashTable;**  
**struct HashTbl{**  
    **int TableSize;**  
    **Cell TheCells;**  
**};**

H →

	0	
	1	1
	2	
	3	1
	4	
	5	1
	6	1
	7	1
	8	
	9	1
	10	1

**.Info**

## 2. 平方探测法

```
Position Find( ElementType Key, HashTable H )           /*平方探测*/
{
    Position CurrentPos, NewPos;
    int CNum;      /* 记录冲突次数 */
/* 1*/   CNum = 0;
/* 2*/   NewPos = CurrentPos = Hash( Key, H->TableSize );
/* 3*/   while( H->TheCells[ NewPos ].Info != Empty &&
              H->TheCells[ NewPos ].Element != Key ) {
              /* 字符串类型的关键词需要 strcmp 函数!! */
/* 4*/       if(++CNum % 2){ /* 判断冲突的奇偶次 */
/* 5*/           NewPos = CurrentPos + (CNum+1)/2*(CNum+1)/2;
/* 6*/           while( NewPos >= H->TableSize )
/* 7*/               NewPos -= H->TableSize;
              } else {
/* 8*/           NewPos = CurrentPos - CNum/2 * CNum/2;
/* 9*/           while( NewPos < 0 )
/* 10*/               NewPos += H->TableSize;
              }
/* 11*/   return NewPos;
}
```

$d_i$		$+1^2$	$-1^2$	$+2^2$	$-2^2$	$+3^2$	$-3^2$	....
Cnum	1	2	3	4	5	6		

## 2. 平方探测法

```
void Insert( ElementType Key, HashTable H )
{
    /* 插入操作 */
    Position Pos;
/* 1*/   Pos = Find( Key, H );
/* 2*/   if( H->TheCells[ Pos ].Info != Legitimate ) {
        /* 确认在此插入 */
/* 3*/       H->TheCells[ Pos ].Info = Legitimate;
/* 4*/       H->TheCells[ Pos ].Element = Key;
        /*字符串类型的关键词需要 strcpy 函数!!
    */
    }
}
```

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。

# 休息时间

---



第14节休息

## 3. 双散列探测法

---

- $d_i$  选为  $i * h_2(\text{key})$  , 其中  $h_2(\text{key})$  是另一个散列函数。我们把它叫做双散列探测法。由此, 探测序列成了：  
 $h_2(\text{key})$  ,  $2h_2(\text{key})$  ,  $3h_2(\text{key})$  , ...
- 对任意的  $\text{key}$  ,  $h_2(\text{key}) \neq 0$  !
- 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：

$$h_2(\text{key}) = p - (\text{key} \bmod p)$$

其中： $p < \text{TableSize}$  ,  $p$ 、 $\text{TableSize}$  都是素数。

## 4. 再散列

- 当散列表元素太多 (即装填因子 $\alpha$ 太大) 时, 查找效率会下降;
  - 实用最大装填因子一般取  $0.5 \leq \alpha \leq 0.85$
- 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 ) ”

## 4. 再散列

### ● 分离链接法 ( Separate Chaining )

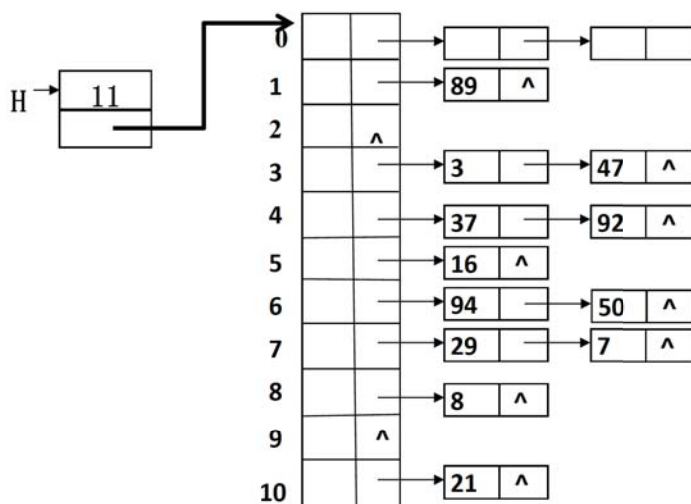
分离链接法：将相应位置上冲突的所有关键词存储在**同一个单链表中**

[例] 设关键字序列为 47, 7, 29, 11, 16, 92, 22, 8, 3, 50, 37, 89, 94, 21;

- 散列函数取为： $h(\text{key}) = \text{key} \bmod 11$
- 用分离链接法处理冲突

```
struct HashTbl
{
    int TableSize;
    List TheLists;
}H;
```

- 该表中有9个结点只需1次查找
- 5个结点需要2次查找
- 因此查找成功的平均查找次数为：
  - $ASL_s = (9 + 5 \times 2) / 14 \approx 1.36$
  - $ASL_u$ 估算比较复杂, 见后



## 4. 再散列

```
struct ListNode;  
typedef struct ListNode *Position, *List;  
struct HashTbl;  
typedef struct HashTbl *HashTable;  
struct ListNode  
{  
    ElementType Element;  
    Position Next;  
};
```

```
Position Find( ElementType Key, HashTable H )  
{  
    Position P;  
    List L;  
    /* 1*/ L = &( H->TheLists[ Hash( Key, H->TableSize ) ] );  
    /* 2*/ P = L->Next;  
    /* 3*/ while( P != NULL && strcmp(P->Element, Key) )  
    /* 4*/     P = P->Next;  
    /* 5*/ return P;  
}
```

## 开放定址法构建散列表演示

bobo 开放定址法建立散列表

26 36 41 38 44 15 68 12 06 51

线性探查法的探查序列为:

$$h_i = (h(\text{key}) + i) \% m \quad 0 \leq i \leq m-1$$

在这里:  $m=13$

$$h_i = ( \quad + i ) \% 13 = \quad \quad 0 \leq i \leq 12$$

下标	0	1	2	3	4	5	6	7	8	9	10	11	12
T[0..12]													
探查次数													

开始建表



# 拉链法构建散列表演示

## 拉链法创建散列表

bobo

26 36 41 38 44 15 68 12 06 51

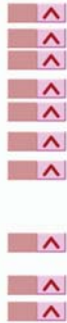
存储池

$$h_i = (h(\text{key}) + i) \% m \quad 0 \leq i \leq m-1$$

在这里:  $m=13$

$$h = \%13 = 0 \leq i \leq 12$$

开始创建



0 1 2 3 4 5 6 7 8 9 10 11 12  
比较次数

## 4.4 散列表的性能分析

# 散列表的性能分析

- 平均查找长度 ( ASL ) 用来度量散列表查找效率。
- 另一方面，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 $\alpha$ 。
- 分析：不同冲突处理方法、装填因子对效率的影响

## 1. 线性探测法的查找性能

可以证明，线性探测法的期望探测次数满足下列公式：

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

- 当 $\alpha = 0.5$ 时，  
插入操作和不成功查找的期望  $ASL_u = 0.5 * (1 + 1 / (1 - 0.5)^2) = 2.5$  次，  
成功查找的期望  $ASL_s = 0.5 * (1 + 1 / (1 - 0.5)) = 1.5$  次
- 例子5.6的 $\alpha = 0.69$ ，于是  
期望  $ASL_u = 0.5 * (1 + 1 / (1 - 0.69)^2) = 5.70$  次  
期望  $ASL_s = 0.5 * (1 + 1 / (1 - 0.69)) = 2.11$  次 (例中  $ASL_s = 2.56$ )。

## 2. 平方探测法和双散列探测的查找性能

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：

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

- 当 $\alpha = 0.5$ 时，  
插入操作和不成功查找的期望  $ASL_u = 1/(1-0.5) = 2$  次，  
成功查找的期望  $ASL_s = -1/0.5 * \ln(1-0.5) \approx 1.39$  次。
- 例子5.7的 $\alpha = 0.82$ ，于是  
期望  $ASL_u = 1/(1-0.82) \approx 5.56$  次  
期望  $ASL_s = -1/0.5 * \ln(1-0.5) \approx 2.09$  次（例中 $ASL_s = 2$ ）。

## 2. 平方探测法和双散列探测的查找性能

### 期望探测次数与装填因子 $\alpha$ 的关系

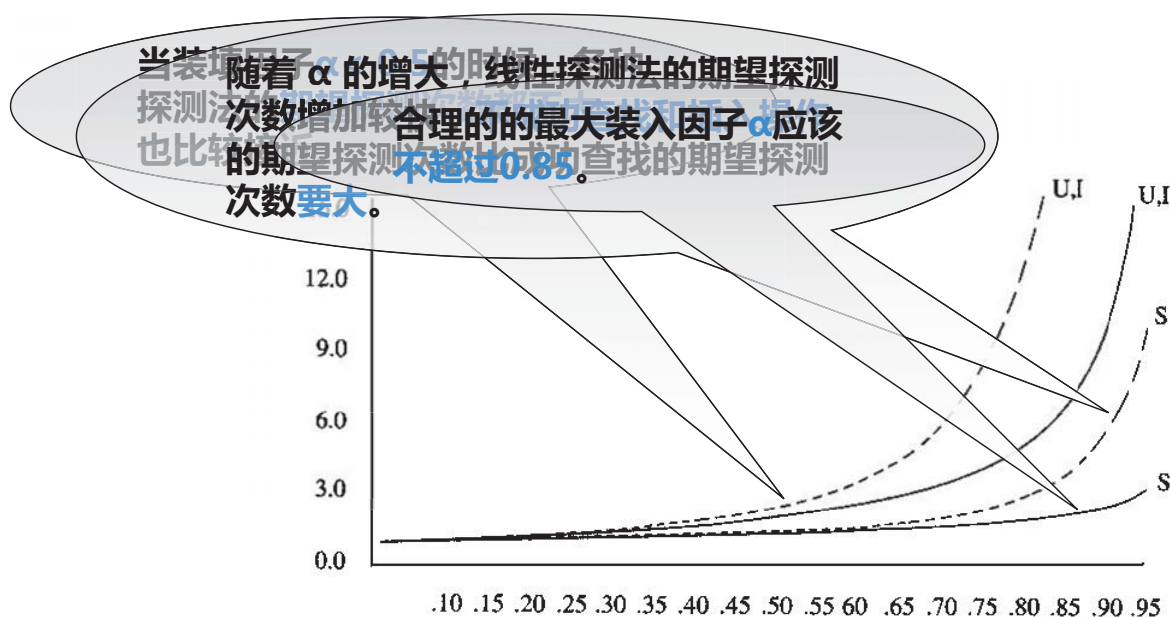


图5.8 线性探测法（虚线）、双散列探测法（实线）  
U表示不成功查找，I表示插入，S表示成功查找

### 3. 分离链表法的查找性能

所有地址链表的平均长度定义成装填因子 $\alpha$ ， $\alpha$ 有可能超过1。

不难证明：其期望探测次数  $p$  为：

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

随着 $\alpha$ 增大，ASLu增加很慢；  
而ASLs线性增长。

- 当 $\alpha = 1$ 时，  
插入操作和不成功查找的期望  $ASLu = 1 + e^{-1} = 1.37$  次，  
成功查找的期望  $ASLs = 1 + 1 / 2 = 1.5$  次。
- 例子5.8的14个元素分布在11个单链表中，所以 $\alpha = 14/11 \approx 1.27$ ，  
期望  $ASLu = 1.27 + e^{-1.27} \approx 1.55$  次  
期望  $ASLs = 1 + 1.27 / 2 \approx 1.64$  次（例中 $ASLs = 1.36$ ）

### 3. 分离链表法的查找性能

- 选择合适的  $h(\text{key})$ ，散列法的查找效率期望是常数  $O(1)$ ，它几乎与关键字的空间的大小 $n$ 无关！
- 它是以较小的 $\alpha$ 为前提。因此，散列方法是一个以空间换时间的成功范例。
- 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。

### 3. 查找：开放地址法

---

- 开放地址法：

- 散列表是一个数组，存储效率高，随机查找。
- 散列表有“聚集”现象，再散列时有“停顿”现象。

### 3. 查找：分离链法

---

- 分离链法：

- 散列表是顺序存储和链式存储的结合，链表部分的存储效率和查找效率都比较低。
- 关键字删除不需要“懒惰删除”法，从而没有存储“垃圾”。
- 太小的 $\alpha$ 可能导致空间浪费，大的 $\alpha$ 又将付出更多的时间代价。不均匀的链表长度导致时间效率的严重下降。

## 5.4 应用实例

### 应用1：文件中单词词频统计

**[例]** 给定一个英文文本文件，统计文件中所有单词出现的频率，并输出词频最大的前10%的单词及其词频。

为简单起见，假设单词字符定义为**大小写字母、数字和下划线**，其他字符均认为是单词分隔符，不予考虑。

**[分析]** 关键是不断对**新读入的单词在已有单词表中查找**，如果已经存在，则将该单词的词频加1，如果不存在，则插入该单词并记词频为1。

**核心问题：**

如何设计该单词表的**数据结构**才可以进行快速地**查找和插入**？



散列表！

# 应用：文件中单词词频统计

```
int main() {
/* 1 */ int TableSize = 10000 ; /* 散列表的估计大小 */
    int wordcount = 0, length;
    HashTable H;
    ElementType word;
    FILE *fp;
/* 2 */ char document[30] = "HarryPotter.txt "; /* 要被统计词频的文件名 */
/* 3 */ H = InitializeTable( TableSize ); /* 建立散列表 */
/* 4 */ if( ( fp = fopen( document, "r" ) ) == NULL ) FatalError( "无法打开文件!\n" );
    while( !feof( fp ) ){
/* 5 */         length = GetAWord( fp, word ); /* 从文件中读入单词 */
/* 6 */         if( length > 3 ){ /* 只考虑适当长度的单词 */
/* 7 */             wordcount++;
/* 8 */             InsertAndCount( word, H );
        }
    }
    fclose( fp );
/* 9 */ printf( "该文档共出现 %d 个有效单词\n", wordcount );
/* 10 */ Show( H, 10.0/100 ); /* 显示词频前10%的所有单词 */
/* 11 */ DestroyTable( H ); /* 销毁散列表 */
    return 0;
}
```

查找散列表，若存在，  
则将该单词的词频加1，  
并  
根据散列表，输出最频繁出现的给定百分比  
( 10% ) 的单词。

## 应用2：电话聊天狂人

输入样例:

4		
13005711862	13005711862	1
13588625832	13588625832	3
13505711862	13505711862	1
13088625832	13088625832	1
13588625832	18087925832	1
18087925832	15005713862	1
15005713862	15005713862	1
13588625832		

输出样例：

13588625832

3



## 解法1：排序

---

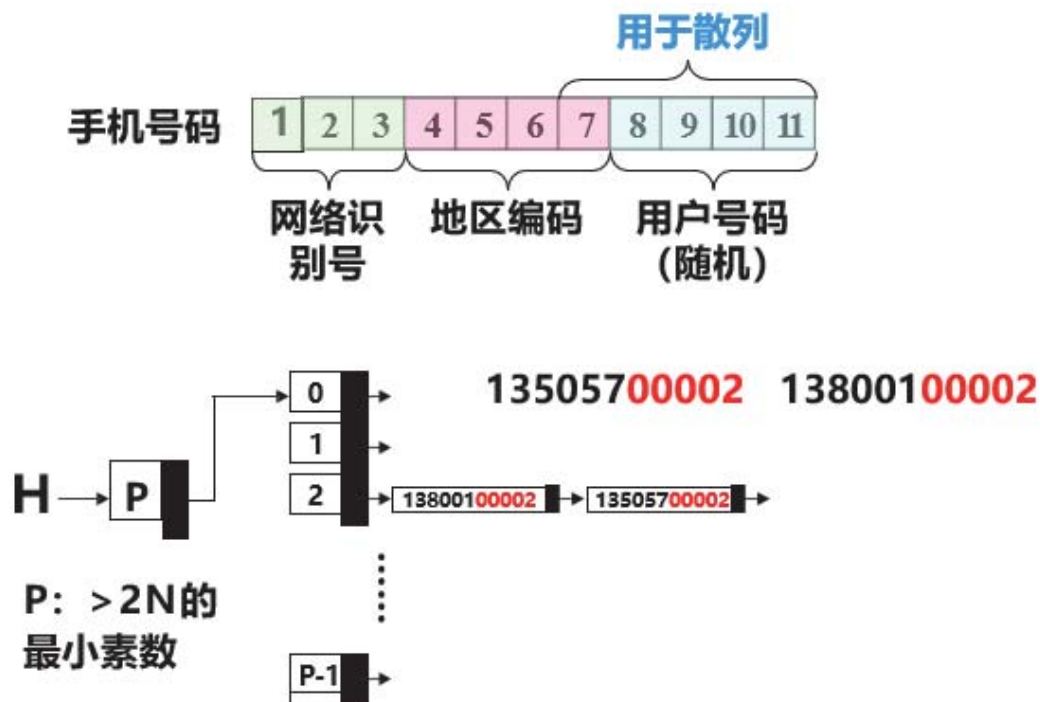
- 第1步：读入最多 $2 \times 10^5$ 个电话号码，每个号码存为长度为11的字符串
- 第2步：按字符串非递减顺序排序
- 第3步：扫描有序数组，累计同号码出现的次数，并且更新最大次数
  - 编程简单快捷
  - 无法拓展解决动态插入的问题

## 解法2：直接映射

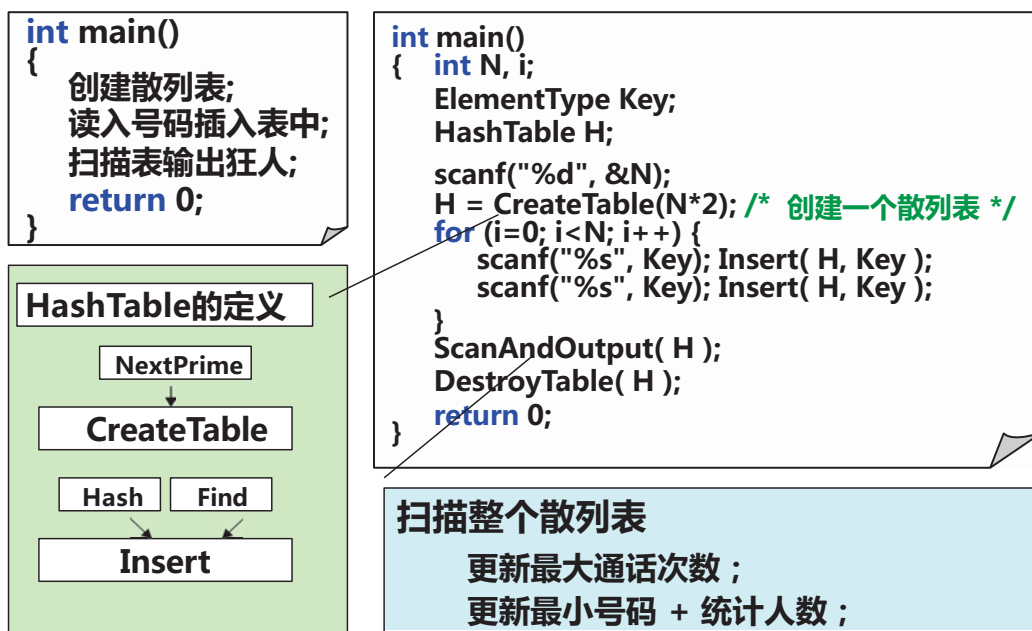
---

- 第1步：创建有 $2 \times 10^{10}$ 个单元的整数数组，保证每个电话号码对应唯一的单元下标；数组初始化为0
- 第2步：对读入的每个电话号码，找到以之为下标的单元，数值累计1次
- 第3步：顺序扫描数组，找出累计次数最多的单元
  - 编程简单快捷，动态插入快
  - 下标超过了unsigned long
    - 需要  $2 \times 10^{10} \times 2 \text{ bytes} \approx 37\text{GB}$
    - 为了  $2 \times 10^5$  个号码扫描  $2 \times 10^{10}$  个单元

## 解法3：带智商的散列



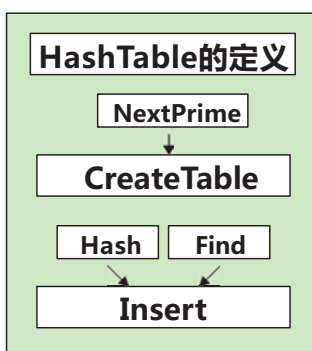
## 程序框架搭建



# 输出狂人

```
void ScanAndOutput( HashTable H )
{
    int i, MaxCnt = PCnt = 0;
    ElementType MinPhone;
    List Ptr;
    MinPhone[0] = '\0';
    for (i=0; i<H->TableSize; i++) { /* 扫描链表 */
        Ptr = H->Heads[i].Next;
        while (Ptr) {
            if (Ptr->Count > MaxCnt) { /* 更新最大通话次数 */
                MaxCnt = Ptr->Count;
                strcpy(MinPhone, Ptr->Data); PCnt = 1;
            }
            else if (Ptr->Count == MaxCnt) {
                PCnt++; /* 狂人计数 */
                if ( strcmp(MinPhone, Ptr->Data)>0 )
                    strcpy(MinPhone, Ptr->Data); /* 更新狂人的最小手机号码 */
            }
            Ptr = Ptr->Next;
        }
    }
    printf("%s %d", MinPhone, MaxCnt);
    if ( PCnt > 1 ) printf(" %d", PCnt);
    printf("\n");
}
```

## 模块的引用和裁剪



```
#define KEYLENGTH 11 /* 关键词字符串的最大长度 */
/* 关键词类型用字符串 */
```

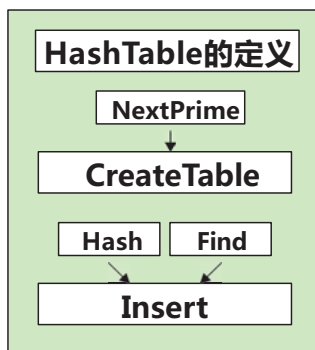
```
typedef char ElementType[KEYLENGTH+1];
typedef int Index; /* 散列地址类型 */
```

```
typedef struct LNode *PtrToLNode;
struct LNode {
    ElementType Data;
    PtrToLNode Next; int
    Count;
};
```

```
typedef PtrToLNode Position;
typedef PtrToLNode List;
```

```
typedef struct TblNode *HashTable;
struct TblNode { /* 散列表结点定义 */
    int TableSize; /* 表的最大长度 */
    List Heads; /* 指向链表头结点的数组 */
};
```

# 模块的引用和裁剪



```
int Hash ( int Key, int P )
{ /* 除留余数法散列函数 */
  return Key%P;
}
```

```
#define MAXTABLESIZE 1000000
int NextPrime( int N )
{ /* 返回大于N且不超过MAXTABLESIZE的最小素数 */
  int i, p = (N%2)? N+2 : N+1; /* 从大于N的下一个奇数开始 */
  while( p <= MAXTABLESIZE ) {
    for( i=(int)sqrt(p); i>2; i-- )
      if ( !(p%i) ) break; /* p不是素数 */
    if ( i==2 ) break; /* for正常结束, 说明p是素数 */
    else p += 2; /* 否则试探下一个奇数 */
  }
  return p;
}
```

```
HashTable CreateTable( int TableSize )
{
  HashTable H; int i;
  H = (HashTable)malloc(sizeof(struct TblNode));
  H->TableSize = NextPrime(TableSize);
  H->Heads = (List)malloc(H->TableSize*sizeof(struct LNode));
  for( i=0; i<H->TableSize; i++ ) {
    H->Heads[i].Data[0] = '\0';
    H->Heads[i].Next = NULL;
    H->Heads[i].Count = 0;
  }
  return H;
}
```

# 模块的引用和裁剪

Position Find( HashTable H, ElementType Key )

```
{ Position P; Index
  Pos;
```

```
/* 初始散列位置 */
```

```
Pos = Hash(atoi(Key+KEYLENGTH-MAXD), H->TableSize);
```

```
P = H->Heads[Pos].Next; /* 从该链表的第1个结点开始 */
```

```
/* 当未到表尾, 并且Key未找到时 */
```

```
while( P && strcmp(P->Data, Key) )
```

```
  P = P->Next;
```

```
return P; /* 此时P或者指向找到的结点, 或者为NULL */
```

```
}
```

# 模块的引用和裁剪

```
bool Insert( HashTable H, ElementType Key )
{ Position P, NewCell; Index Pos;
  P = Find( H, Key );
  if ( !P ) { /* 关键词未找到，可以插入 */
    NewCell = (Position)malloc(sizeof(struct LNode));
    strcpy(NewCell->Data, Key);
    /* 初始散列位置 */
    Pos = Hash( Key, H->TableSize );
    NewCell->Count = 1;
    Pos = Hash(atoi(Key+KEYLENGTH-MAXD), H->TableSize);
    /* 将NewCell插入为H->Heads[Pos]链表的第1个结点 */
    NewCell->Next = H->Heads[Pos].Next;
    H->Heads[Pos].Next = NewCell; return
    true;
  }
  else { /* 关键词已存在 */
    printf("键值已存在");
    P->Count++;
    return false;
  }
}
```

## 休息时间



第15节休息