9.3 哈 希 表

9.3.1什么是哈希表

以上两节讨论的表示查找表的各种**结构**的共同**特点**：记录在表中的**位置**和它的**关键字**之间**不存在**一个确定的关系；

查找的过程为给定值依次和关键字集合中各个关键字进行比较；

查找的效率取决于和给定值进行比较的关键字个数。

用这类方法表示的查找表，其平均查找长度都不为零。

对于频繁使用的查找表，希望 $ASL = 0$。

只有一个办法：预先知道所查关键字在表中的位置。即，要求：记录在表中的位置和其关键字之间存在一种确定的关系。

例如：为每年招收的 1000 名新生建立一张查找表，其关键字为学号，其值的范围为 xx000 ~ xx999 (前两位为年份)。

若以下标为000 ~ 999 的顺序表表示之。

则查找过程可以简单进行：取给定值（学号）的后三位，不需要经过比较便可直接从顺序表中找到待查关键字。

但是，对于**动态查找表**而言，有以下问题：

1）表长不确定；

2）在设计查找表时，只知道关键字所属范围，而不知道确切的关键字。

因此在一般情况下，需在关键字与记录在表中的存储位置之间建立一个函数关系，以 f(key) 作为关键字为 key 的记录在表中的位置，通常称这个函数 f(key) 为哈希函数（散列函数）。

例如：对于如下 9 个关键字

{Zhao, Qian, Sun, Li, Wu, Chen, Han, Ye, Dei}

设 哈希函数 f(key) =

⌊(Ord(第一个字母) -Ord('A')+1)/2⌋

0	1	2	3	4	5	6	7	8	9	10	11	12	13
	Chen	Dei		Han		Li		Qian	Sun		Wu	Ye	Zhao

问题: 1、若添加关键字 **Zhou，怎么办？**

2、能否找到另一个哈希函数？

从这个例子可见：

1）哈希函数是一个**映象**，即：将关键字的集合映射到某个地址集合上，它的设置很灵活，只要这个地址集合的大小不超出允许范围即可

2）由于哈希函数是一个**压缩映象**，因此，在一般情况下，很容易产生“**冲突**”现象，即：key1≠ key2，而 f(key1) = f(key2)。具有相同函数值的关键字对该哈希函数来说称作同义词。

3）很难找到一个不产生冲突的哈希函数。一般情况下，只能选择恰当的哈希函数，使冲突尽可能少地产生。

因此，在构造这种特殊的“查找表”时，除了需要选择一个“好”（尽可能少产生冲突）的哈希函数之外；还需要找到一种“处理冲突”的方法。

哈希表的定义：

根据设定的**哈希函数 H(key)** 和所选中的**处理冲突的方法**，将一组关键字**映象到**一个有限的、地址连续的地址集 (区间) 上，并以关键字在地址集中的“象”作为相应记录在表中的**存储位置**，如此构造所得的查找表称之为“**哈希表**”。

9.3.2 构造哈希函数的方法

对**数字**的关键字可有下列构造方法：

1. 直接定址法

2. 数字分析法

3. 平方取中法

4. 折叠法

5. 除留余数法

6. 随机数法

若是非数字关键字，则需先对其进行数字化处理。

1. 直接定址法

哈希函数为关键字的线性函数

$H(key) = key$　　或者

$H(key) = a \times key + b$

此法仅适合于：

地址集合的大小 = = 关键字集合的大小

2. 数字分析法

假设关键字集合中的每个关键字都是由 s 位数字组成 (u1, u2, …, us)，分析关键字集中的全体， 并从中提取分布均匀的若干位或它们的组合作为地址。

此方法仅适合于：

能预先估计出全体关键字的每一位上各种数字出现的频度。

3．平方取中法

以关键字的平方值的中间几位作为存储地址。求“关键字的平方值” 的目的是“扩大差别” ，同时平方值的中间各位又能受到整个关键字中各位的影响。

此方法适合于:

关键字中的每一位都有某些数字重复出现频度很高的现象。

4. 折 叠 法

将关键字分割成若干部分，然后取它们的叠加和为哈希地址。有两种叠加处理的方法：**移位叠加**和**间界叠加**。

此方法适合于:

关键字的数字位数特别多。

5. 除留余数法

设定哈希函数为: H(key) = key MOD p

其中， p≤m (表长) 并且 p 应为不大于 m 的素数，或是不含 20 以下的质因子。

为什么要对 p 加限制？

例如：

给定一组关键字为：12, 39, 18, 24, 33, 21，若取 p=9, 则他们对应的哈希函数值将为：3, 3, 0, 6, 6, 3

可见，若 p 中含质因子 3， 则所有含质因子 3 的关键字均映射到“3 的倍数”的地址上，从而增加了“冲突”的可能。

6. **随机数法**

设定哈希函数为:　H(key) = Random(key)

其中，Random 为伪随机函数

通常，此方法用于对长度不等的关键字构造哈希函数。

实际造表时，采用何种构造哈希函数的方法取决于建表的关键字集合的情况(包括关键字的范围和形态)，总的原则是使产生冲突的可能性降到尽可能地小。

9.3.3　处理冲突的方法

“**处理冲突**”的实际含义是：为产生冲突的地址**寻找下一个**哈希地址。通常有下列几种方法：

1. 开放定址法
2. 再哈希法
3. 链地址法
4. 建立一个公共溢出区

1. 开放定址法

为产生冲突的地址 H(key) 求得一个**地址序列**：$H_0, H_1, H_2,$ …, H_s　$1 \le s \le m-1$

其中：$H_0 = H(key)$

$H_i = (H(key) + d_i) \text{ MOD } m$

$i=1, 2, \ldots, s$，m为哈希表表长

对增量 d_i 有三种取法：

1）线性探测再散列

$d_i = 1，2，3，…，m-1$

2）二次探测再散列

$d_i = 1^2, -1^2, 2^2, -2^2, …,$

3） 伪随机探测再散列

d_i 是一组伪随机数列　或者

$d_i = i \times H_2(key)$ (又称双散列函数探测)

注意：增量 d_i 应具有“完备性”

即：产生的 H_i 均不相同，且所产生的 **s(m-1) 个** H_i 值能覆盖哈希表中所有地址。

在处理冲突过程中发生的两个第一个哈希地址不同的记录争夺同一个后继哈希地址的现象称作‘**二次聚集**’（**碰撞**）。

例如: 关键字集合　{ 19, 01, 23, 14, 55, 68, 11, 82, 36 }

设定哈希函数 H(key) = key **MOD** 11 (表长=11)

若采用**线性探测再散列**处理冲突

0	1	2	3	4	5	6	7	8	9	10
55	01	23	14	68	11	82	36	19		
1	1	2	1	3	6	2	5	1		

若采用**二次探测再散列**处理冲突

0	1	2	3	4	5	6	7	8	9	10
55	01	23	14	36	82	68		19		11

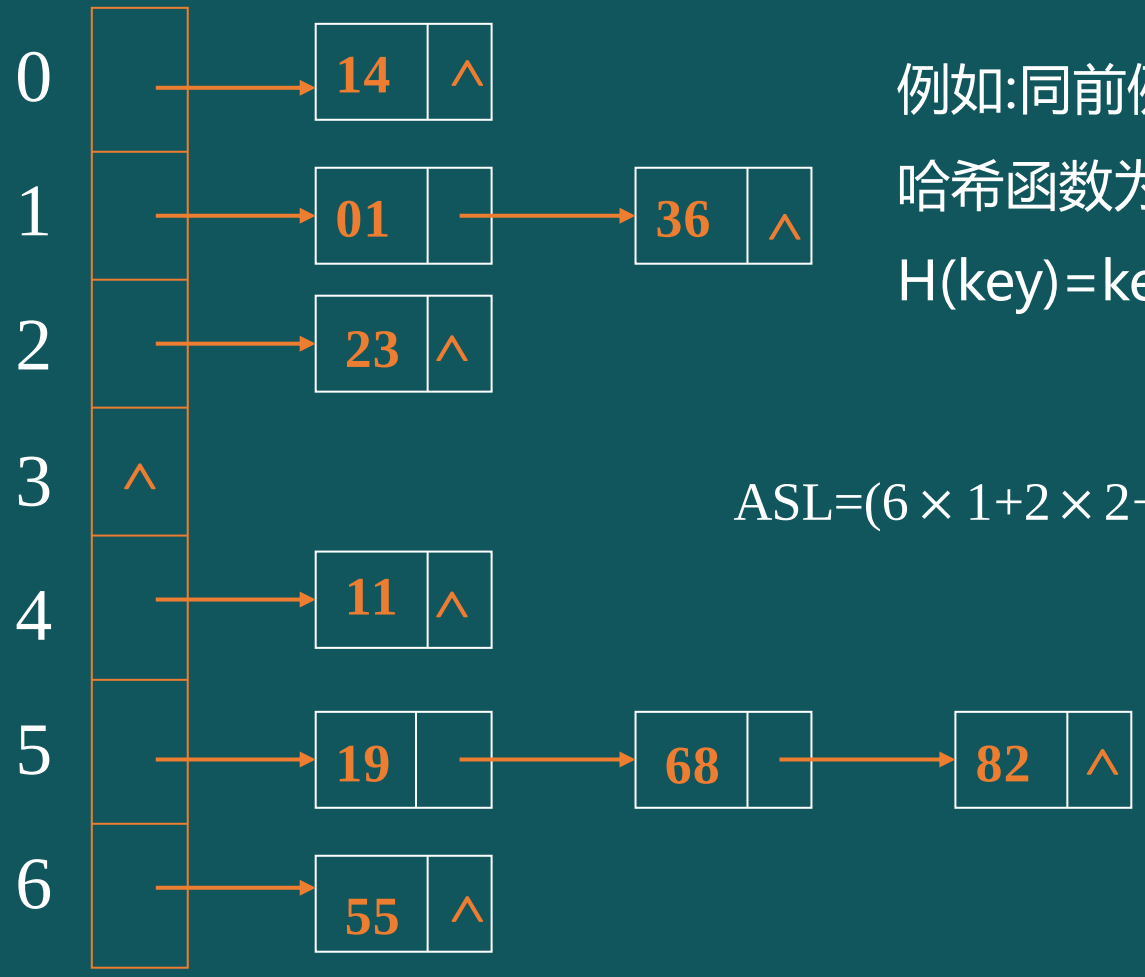
2. 再哈希法

Hi = RHi (key)　　i=1, 2, …, k

RHi 均是不同的哈希函数，即在同义词产生地址冲突时，计算另一个哈希函数地址，直到冲突不再发生。这种方法不易产生“聚集”，但增加了计算的时间。

3. 链地址法

将所有哈希地址相同的记录都链接在同一链表中。



例如:同前例的关键字，

哈希函数为

H(key)=key MOD 7

ASL=(6×1+2×2+3)/9=13/9

4. 建立一个公共溢出区

这也是处理冲突的一种方法。假设哈希函数的值域为[0，m-1]，则设向量HashTable[0，m-1]为基本表，每个分量存放一个记录，另设立向量OverTable[0..v]为溢出表。所有关键字和基本表中关键字为同义词的纪录，不管它们由哈希函数得到的哈希地址是什么，一旦发生冲突，都填入溢出表。

9.3.4 哈希表的查找及其分析

查找过程和造表过程一致。假设采用开放定址处理冲突，则**查找过程**为：

对于给定值 K，计算哈希地址 i = H(K)

若 r[i] = NULL 则查找不成功

若 r[i].key = K 则查找成功

否则 “求下一地址 Hi”（按解决冲突方法），直至 r[Hi] = NULL (查找不成功)

或 r[Hi].key = K (查找成功) 为止。

```
//--- 开放定址哈希表的存储结构 ---

int  hashsize[] = { 997, ... };  //哈希表容量递增表，一
                                 //个合适的素数序列
typedef struct {
  ElemType  *elem; //数据元素存储基址，动态分配数组
  int  count;            // 当前数据元素个数
  int  sizeindex;     // hashsize[sizeindex]为当前容量
} HashTable;
#define  SUCCESS  1
#define  UNSUCCESS  0
#define  DUPLICATE  -1
```

```
Status SearchHash (HashTable H, KeyType K, int &p, int &c) {
 // 在开放定址哈希表H中查找关键码为K的记录，若查找成功，
//以P指示待查数据在表中位置；否则，以P指示插入位置，c用以
//计冲突次数
   p = Hash(K);        // 求得哈希地址
  while ( H.elem[p].key != NULLKEY  && //该位置有记录
                !EQ(K, H.elem[p].key)）// 并且关键字不相等
      collision(p, ++c);       // 求得下一探查地址 p
  if (EQ(K, H.elem[p].key)) return SUCCESS;
          // 查找成功，返回待查数据元素位置 p
  else return UNSUCCESS; //查找失败，P返回的是插入位置
} // SearchHash
```

```
Status InsertHash (HashTable &H, Elemtype e){
//查找不成功时插入数据元素e到开放定址哈希表H中，并返回OK；若冲突次数过大，则重建哈希表
  c = 0;
  if ( HashSearch ( H, e.key, p, c ) == SUCCESS )
      return DUPLICATE;  // 表中已有与 e 有相同关键字的元素
  else  if ( c < hashsize[H.sizeindex]/2 ) {
        // 冲突次数 c 未达到上限，（阀值 c 可调）
           H.elem[p] = e;  ++H.count;  return OK;    // 插入e
         }
  else {RecreateHashTable(H); return UNSUCCESS } // 重建哈希表
} // InsertHash
```

哈希表查找的分析:

从查找过程得知，哈希表查找的平均查找长度**实际上并不等于零。**

决定哈希表查找的ASL的因素：

1）选用的哈希函数；

2）选用的处理冲突的方法；

3）哈希表的饱和程度，装载因子α

α=表中填入的记录数/哈希表的长度

一般情况下，可以认为选用的哈希函数是“均匀”的，则在讨论ASL时，可以不考虑它的因素。

因此，哈希表的ASL是处理冲突方法和装载因子的函数。

例如：前述例子

线性探测处理冲突时，　ASL =22/9

双散列探测处理冲突时，ASL =14/9

链地址法处理冲突时，　ASL =13/9

可以证明：**查找成功**时有下列结果：

线性探测再散列 $S_{nl} \approx \frac{1}{2}(1+\frac{1}{1-\alpha})$

随机探测再散列 $S_{nr} \approx -\frac{1}{\alpha}\ln(1-\alpha)$

链地址法 $S_{nc} \approx 1+\frac{\alpha}{2}$

从以上结果可见：

哈希表的平均查找长度是 α 的函数，而不是 ***n*** 的函数。

这说明，用哈希表构造查找表时，可以选择一个适当的装填因子 α ，使得平均查找长度限定在某个范围内。

—— 这是哈希表所特有的特点。

哈希表的删除操作

从哈希表中删除记录时，要作**特殊处理**，相应地，需要修改查找的算法。

哈希表也可以用来构造静态查找表

并且，对静态查找表，有时可以找到不发生冲突的哈希函数。即，此时的哈希表的 ASL=0，称此类哈希函数为理想（perfect）的哈希函数。