第7章 索引结构与散列技术

- 本章的主要内容

  1、索引结构

  - 线性索引
  - 倒排表
  - 多级索引

  2、散列技术

  - 基本概念
  - 构造散列函数
  - 散列表的查找及分析

四种常用的存储方法

- 顺序存储
- 链式存储
- 索引结构（查找运算）
- 散列技术 （查找运算）

索引结构基本概念

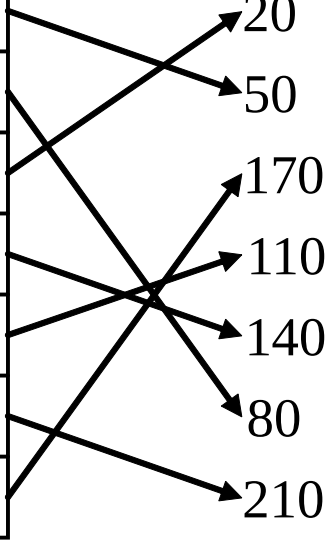
- 索引结构包括两部分：索引表和数据表。指明结点与其存储位置之间的对应关系的表就叫做索引表；数据表是存储结点信息的，索引结构中常用的数据表是线性表。
- 索引表中的每一项称作索引项，索引项的一般形式是：（关键字，地址）。其中关键字是能唯一标识一个结点的那些数据项。
- 如果数据表中的记录按关键字顺序排列，这时的索引结构称索引顺序结构。反之，若数据表中的数据未按关键字顺序排列时，则称索引非顺序结构。

线性索引——稠密索引

- 对于索引非顺序结构，由于数据表中的记录是无序的，则必须为每个记录建立一个索引项，这种一个索引项对应数据表中一个对象的索引结构称为稠密索引。

索引表

关键字	地址
21	50
29	80
30	20
34	140
38	110
42	210
53	170



学生数据表

学号	姓名	性别	籍贯
30	张　祺	女	陕西
21	王　军	男	山东
53	江　丽	女	北京
38	周　强	男	北京
34	刘晓飞	男	陕西
29	王江平	男	浙江
42	陈　娟	女	陕西

两种线性索引的比较

- 对于索引顺序结构，由于数据表中的记录按关键字有序，则可对一组记录建立一个索引项，这种索引称为稀疏索引。
- 在稠密索引中，索引项中的地址将指出结点所在的存储位置，
- 而在稀疏索引中，索引项中的地址指出的则是一组结点的起始存储位置。

无论是稠密索引还是稀疏索引，都属于线性索引。

索引结构上的检索

- 查找索引表，若索引表上存在该记录，则根据索引项的指示在数据表中读取数据；否则说明数据表中不存在该记录，也就不需要访问数据表。
- 由于索引表是有序的，则索引表的查找既可顺序查找，又可使用折半查找。

索引结构查找方法—分块查找

- 分块查找性能介于顺序查找和二分查找之间的查找方法，又称索引顺序查找。构造方法：

1、将数据表R[n]均分成b块，前b-1块中记录个数为S=⌈n/b⌉，第b块的记录数小于等于S；

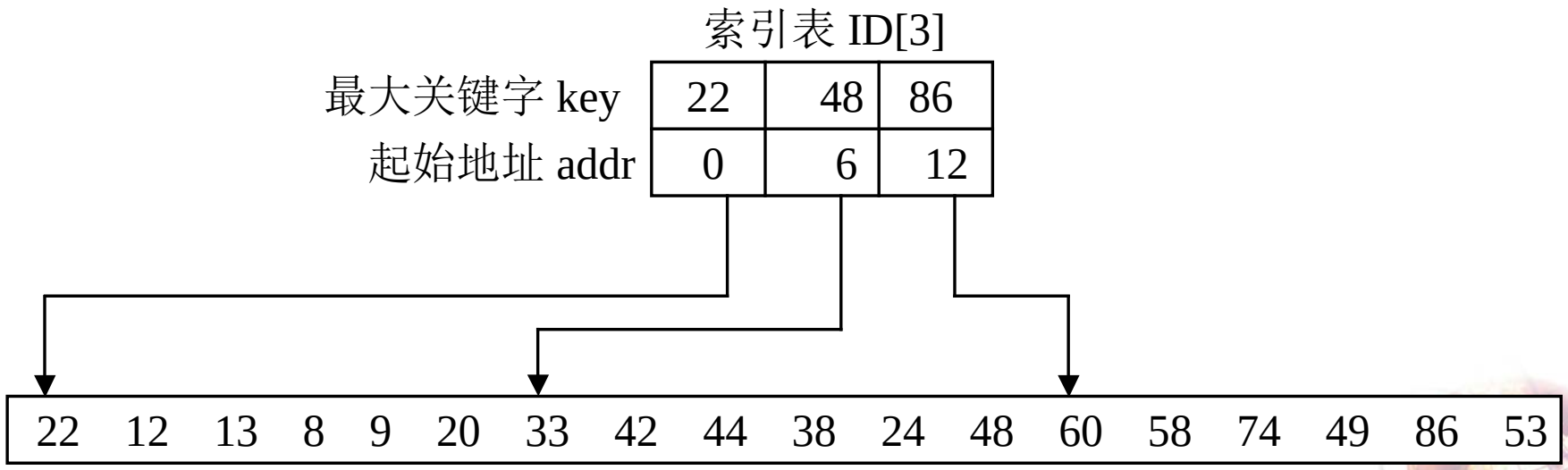
2、前一块中的最大关键字必须小于后一块中的最小关键字，即要求表是“分块有序”的；

3、抽取各块中的最大关键字及其起始位置构成一个索引表ID[i]，即ID[i](0≤i＜b)中存放着第i块的最大关键字及该块在表R中的起始位置。

由于表是分块有序的，所以索引表是递增有序表。

分块查找的存储结构示例

- 下图表R中只有18个记录，被分成3块，每块中有6个记录，第一块中最大关键字22小于第二块中最小关键字24，第二块中最大关键字48小于第三块中最小关键字49。



分块查找的基本思想

1、查找索引表，因为索引表是有序表，故可采用顺序查找或折半查找，以确定待查找的记录在哪一块；

2、在已确定的那一块中进行顺序查找。

说明：由于分块查找实际上是两次查找过程，故分块查找的算法即为这两种算法的简单合成，而整个算法的平均查找长度，即是两次查找的平均查找长度之和。

```
typedef struct          /*  索引表的结点类型 */
{ keytype  key;
  int addr;
}Idtable;
Idtable ID[b];    /*  索引表  */
```

散列表的查找

- 问题：
  - 能否构造一种查找方法与比较次数无关或关系较小

散列表的查找

- 问题：
  - 能否构造一种查找方法与比较次数无关或关系较小
- 散列表查找可达到这一要求：不用比较而直接计算出记录所在地址，从而可直接进行存取的方法。
- 散列表查找的基础是散列存储结构。

散列表的概念

- 以结点的关键字k为自变量，通过一个确定的函数关系f，计算出对应的函数值，把这个值解释为结点的存储地址，将结点存入f(k)所指的存储位置上。该方法称为散列法；又称关键字——地址转换法。
- 用散列法存储的线性表叫散列表（Hush Table）或哈希表。
- 函数f()称为散列函数或哈希函数，f(k)的值则称为散列地址或哈希地址。
- 通常散列表的存储空间是一个一维数组，散列地址是数组的下标，在不致于混淆之处，我们将这个一维数组空间就简称为散列表。

散列表实例

- 例1 假设要建立一张全国30个地区的各民族人口统计表，每个地区为一个记录，记录的各数据项为：

编 号	地 区	总人口	汉族	回族	……

显然数组R[30] 来存放这张表，其中R[i]是编号为i的地区的人口情况。编号i便为记录的关键字，由它惟一确定记录的存储位置R[i]。

- 例2 已知一个含有70个结点的线性表，其关键字都由两位十进制数字组成，则可将此线性表存储在如下说明的散列表中。

```
datatype  HT1[100];
```

其中，HT1[i]存放关键字为i的结点，即散列函数为

$H_1(key)=key$

散列表实例

- 例3 已知线性表的关键字集合为：
  S={and,begin,do,end,for,go,if,repeat,then,until,while}
  则可设散列表为：
  char HT2[26][8];
  散列函数H_2(key)的值，取为关键字key中第一个字母在字母表{a,b,Λ,z}中的序号（序号范围是0至25），即
  H_2(key)=key[0]–′a′
  其中，key的类型是长度为8的字符数组，利用H_2构造的散列表如右表所示。

表 7-1 关键字集合 S 对应的散列表

散列地址	关键字	其它数据项
0	and	
1	begin	
2		
3	do	
4	end	
5	for	
6	go	
7		
8	if	
•	•	•
•	•	•
•	•	•
17	repeat	
18		
19	then	
20	until	
21		
22	while	
•	•	•

散列表讨论

- 讨论：

  1 散列函数是一个一对一的函数。

  2 装填因子：散列表空间大小为m，填入表中的结点数是n，则称α=n/m为散列表的装填因子。实用时，常在区间[0.65,0.9]上取α的适当值。

  3 散列函数的选取原则是：运算应尽可能简单；函数的值域必须在表长的范围之内；尽可能使得关键字不同时，其散列函数值亦不相同。

  4 冲突：若某个散列函数H对于不相等的关键字key_1和key_2得到相同的散列地址（即$H(key_1)=H(key_2)$），则将该现象称为冲突，而发生冲突的这两个关键字则称为该散列函数H的同义词。

- 散列法查找必须解决下面两个主要问题：
  - (1)选择一个计算简单且冲突尽量少的“均匀”的散列函数；
  - (2)确定一个解决冲突的方法，即寻求一种方法存储产生冲突的同义词。

散列函数的构造

- 4. 除留余数法
  - 基本思想：
    选择适当的正整数p，用p去除关键字，取所得余数作为散列地址，即：

    H(key)=key % p

    一般地选p为小于或等于散列表长度m的某个最大素数比较好。
  - 例如：

    m=8, 16, 32, 64, 128, 256, 512, 1024

    p=7, 13, 31, 61, 127, 251, 503, 1019

解决冲突的方法

- 1. 开放地址法
  - 基本思想

    开放地址法解决冲突的做法是：当发生冲突时，使用某种方法在散列表中形成一个探查序列，沿着此序列逐个单元进行查找，直到找到一个空的单元时将新结点放入。

解决冲突的方法

- 1. 开放地址法
  - 基本思想

    开放地址法解决冲突的做法是：当发生冲突时，使用某种方法在散列表中形成一个探查序列，沿着此序列逐个单元进行查找，直到找到一个空的单元时将新结点放入。
- 2. 拉链法
  - 基本思想

    拉链法解决冲突的方法是：将所有关键字为同义词的结点链接到同一个单链表中。若选定的散列函数的值域为0到m–1，则可将散列表定义为一个由m个头指针组成的指针数组HTP[m]，凡是散列地址为i的结点，均插入到以HTP[i]为头指针的单链表中。

开放地址法

- 当发生冲突时，使用某种方法在散列表中形成一个探查序列，沿着此序列逐个单元进行查找，直到找到一个空的单元时将新结点放入。

开放地址法

- 当发生冲突时，使用某种方法在散列表中形成一个探查序列，沿着此序列逐个单元进行查找，直到找到一个空的单元时将新结点放入。
- 待解决的问题是——探查序列

开放地址法

- 当发生冲突时，使用某种方法在散列表中形成一个探查序列，沿着此序列逐个单元进行查找，直到找到一个空的单元时将新结点放入。
- 待解决的问题是——探查序列
- 探查序列包括：
  - 线性探查法
  - 二次探测法
  - 随机探测法

线性探查法

- 基本思想：将散列表看成是一个环形表，若发生冲突的单元地址为d，则依次探d+1,d+2,…,m–1,0,1,…,d–1，直到找到一个空单元为止。

线性探查法

- 基本思想：将散列表看成是一个环形表，若发生冲突的单元地址为d，则依次探d+1,d+2,…,m–1,0,1,…,d–1，直到找到一个空单元为止。

- 开放地址公式： $d_i = (d+i) \% m (1 \leqslant i \leqslant m-1)$ $d=H(key)$

线性探查法

- 基本思想：将散列表看成是一个环形表，若发生冲突的单元地址为d，则依次探d+1,d+2,…,m–1,0,1,…,d–1，直到找到一个空单元为止。

- 开放地址公式： $d_i = (d+i) \% m (1 \leqslant i \leqslant m-1)$　　d=H (key)

- 已知一组关键字集(26,36,41,38,44,15,68,12,06,51,25)，用线性探查法解决冲突，试构造这组关键字的散列表。取α=0.75，m= $\lceil n/\alpha \rceil$ =15，散列表为HT[15]。

线性探查法

- 基本思想：将散列表看成是一个环形表，若发生冲突的单元地址为d，则依次探d+1,d+2,…,m–1,0,1,…,d–1，直到找到一个空单元为止。

- 开放地址公式： $d_i = (d+i) \% m (1 \leqslant i \leqslant m-1)$ $d=H(key)$

- 已知一组关键字集(26,36,41,38,44,15,68,12,06,51,25)，用线性探查法解决冲突，试构造这组关键字的散列表。取$\alpha=0.75$，$m=\lfloor n/\alpha \rfloor=15$，散列表为HT[15]。

  散列函数为： $H(key)=key\%13$ (15)

线性探查法

- 基本思想：将散列表看成是一个环形表，若发生冲突的单元地址为d，则依次探d+1,d+2,…,m–1,0,1,…,d–1，直到找到一个空单元为止。

- 开放地址公式： $d_i = (d+i) \% m (1 \leqslant i \leqslant m-1)$　　d=H (key)

- 已知一组关键字集(26,36,41,38,44,15,68,12,06,51,25)，用线性探查法解决冲突，试构造这组关键字的散列表。取α=0.75，m= ⌈n/α⌉=15，散列表为HT[15]。

  散列函数为：　H(key)=key%13　(15)

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
关键字	26	25	41	15	68	44	06				36		38	12	51
比较次数	1	5	1	2	2	1	1				1		1	2	3
关键字	15	25					36	6	8	68	51	26	41	12	44
比较次数	1	7					1	2	1	2	2	1	2	2	1

线性探查法

- 基本思想：将散列表看成是一个环形表，若发生冲突的单元地址为d，则依次探d+1,d+2,…,m–1,0,1,…,d–1，直到找到一个空单元为止。

- 开放地址公式： $d_i = (d+i) \% m (1 \leqslant i \leqslant m-1)$ $d=H(key)$

- 已知一组关键字集(26,36,41,38,44,15,68,12,06,51,25)，用线性探查法解决冲突，试构造这组关键字的散列表。取α=0.75，m= ⌈n/α⌉=15，散列表为HT[15]。

  散列函数为： H(key)=key%13 (15)

堆积现象

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
关键字	26	25	41	15	68	44	06				36		38	12	51
比较次数	1	5	1	2	2	1	1				1		1	2	3
关键字	15	25					36	6	8	68	51	26	41	12	44
比较次数	1	7					1	2	1	2	2	1	2	2	1

二次探查法

- 探查序列依次：$1^2,-1^2,2^2,-2^2,\ldots$等
- 发生冲突时，将同义词来回散列在第一个地址d=H(key)的两端。

二次探查法

- 探查序列依次：$1^2,-1^2,2^2,-2^2,\ldots$等
- 发生冲突时，将同义词来回散列在第一个地址d=H(key)的两端。
- 发生冲突时，求下一个开放地址的公式为：

$$d_{2i-1}=(d+i^2)\ \%\ m$$

$$d_{2i}=(d-i^2)\ \%\ m \quad (1\leqslant i\leqslant (m-1)/2)$$

随机探查法

- 采用一个随机数作为地址位移计算下一个单元地址

随机探查法

- 采用一个随机数作为地址位移计算下一个单元地址
- 求下一个开放地址的公式为：

$$d_i=(d+R_i)\ \%\ m \quad (1\leqslant i\leqslant m-1)$$

其中：$d=H(key)$，R_1,R_2，…,R_{m-1}是1,2,…,m　1的一个随机排列。

随机探查法

- 采用一个随机数作为地址位移计算下一个单元地址
- 求下一个开放地址的公式为：

$$d_i=(d+R_i)\ \%\ m \quad (1\leqslant i\leqslant m-1)$$

其中：$d=H(key)$，R_1,R_2，…，R_{m-1}是1,2,…,m　1的一个随机排列。

- 实际应用中，常用移位寄存器序列代替随机数序列。

拉链法

- 将所有关键字为同义词的结点链接到同一个单链表中。
- 若选定的散列函数的值域为0到m–1，则可将散列表定义为一个由m个头指针组成的指针数组HTP[m]，凡是散列地址为i的结点，均插入到以HTP[i]为头指针的单链表中。

拉链法

- 将所有关键字为同义词的结点链接到同一个单链表中。
- 若选定的散列函数的值域为0到m–1，则可将散列表定义为一个由m个头指针组成的指针数组HTP[m]，凡是散列地址为i的结点，均插入到以HTP[i]为头指针的单链表中。
- 例：

  已知一组关键字集(26,36,41,38,44,15,68,12,06,51,25)，用拉链法解决冲突，试构造这组关键字的散列表。

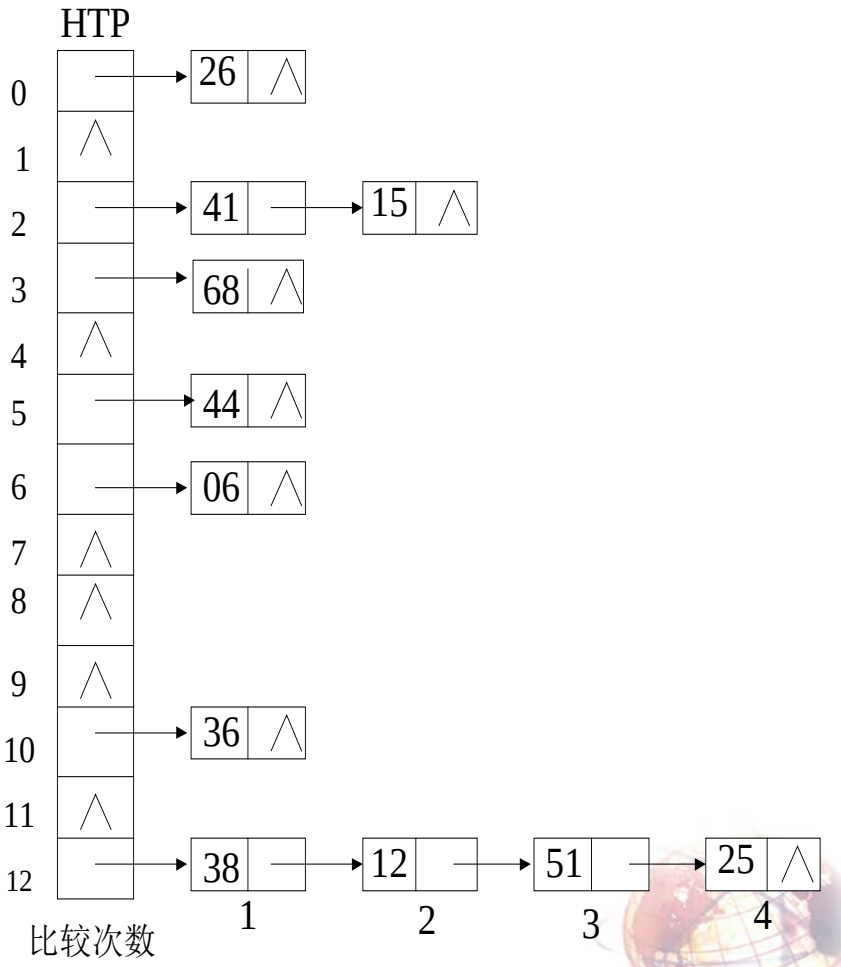
  散列表为HTP[13]。散列函数为：H(key)=key%13

拉链法

- 将所有关键字为同义词的结点链接到同一个单链表中。
- 若选定的散列函数的值域为0到m–1，则可将散列表定义为一个由m个头指针组成的指针数组HTP[m]，凡是散列地址为i的结点，均插入到以HTP[i]为头指针的单链表中。
- 例：

  已知一组关键字集(26,36,41,38,44,15,68,12,06,51,25)，用拉链法解决冲突，试构造这组关键字的散列表。

  散列表为HTP[13]。散列函数为：H(key)=key%13



拉链法构造散列表的优点

- 拉链法不会产生堆积现象，因而平均查找长度较短；

拉链法构造散列表的优点

- 拉链法不会产生堆积现象，因而平均查找长度较短；
- 由于拉链法中各单链表的结点是动态申请的，故它更适合于造表前无法确定表长的情况；

拉链法构造散列表的优点

- 拉链法不会产生堆积现象，因而平均查找长度较短；
- 由于拉链法中各单链表的结点是动态申请的，故它更适合于造表前无法确定表长的情况；
- 在用拉链法构造的散列表中，删除结点的操作易于实现，只要简单地删去链表上相应的结点即可；

拉链法构造散列表的优点

- 拉链法不会产生堆积现象，因而平均查找长度较短；
- 由于拉链法中各单链表的结点是动态申请的，故它更适合于造表前无法确定表长的情况；
- 在用拉链法构造的散列表中，删除结点的操作易于实现，只要简单地删去链表上相应的结点即可；
- 当装填因子α较大时，拉链法所用的空间比开放地址法多，但是α越大，开放地址法所需的探查次数越多，所以，拉链法所增加的空间开销是合算的。

散列表的查找算法-1

- 线性探查法解决冲突的查找和插入算法：

```
# define nil  0      /* nil为空结点标记 */
# define m 18      /* 这时假设表长m为18 */
typedef struct      /* 散列表结点结构 */
{ keytype key;
  datatype other;
} hashtable;
hashtable HT[m];       /* 散列表 */
```

散列表的查找算法-1

- 线性探查法解决冲突的查找和插入算法：

```
# define nil  0      /* nil为空结点标记 */
# define m 18      /* 这时假设表长m为18 */
typedef struct      /* 散列表结点结构 */
{ keytype key;
  datatype other;
} hashtable;
hashtable HT[m];       /* 散列表 */
```

散列地址	0	1	2	3	4	5	6	7	8
关键字	26	25	41	15	68	44	06		
比较次数	1	5	1	2	2	1	1		

散列表的查找算法-1

- 线性探查法解决冲突的查找和插入算法：

```
# define nil  0      /* nil为空结点标记 */
# define m 18      /* 这时假设表长m为18 */
typedef struct      /* 散列表结点结构 */
{ keytype key;
  datatype other;
} hashtable;
hashtable HT[m];       /* 散列表 */
```

散列地址	0	1	2	3	4	5	6	7	8
关键字	26	25	41	15	68	44	06		
比较次数	1	5	1	2	2	1	1		

- int LINSRCH(HT,k)/*在散列表HT[m]中查找关键字为k的结点 */

```
hashtable HT[ ];   keytype k;
{ int d,i=0;       /* i为冲突时的地址增量 */
  d=H(k);          /* d为散列地址 */
 while  ((i<m) && (HT[d].key ! =k)  && (HT[d].key ! =nil))
      { i++;d=(d+i) % m}
 return d;         /* 若HT[d].key =k查找成功，否则失败 */
}     /* LINSRCH */
```

散列表的查找算法-1

- 线性探查法解决冲突的查找和插入算法：

```
# define nil  0      /* nil为空结点标记 */
# define m 18      /* 这时假设表长m为18 */
typedef struct      /* 散列表结点结构 */
{ keytype key;
   datatype other;
} hashtable;
hashtable HT[m];       /* 散列表 */
```

散列地址	0	1	2	3	4	5	6	7	8
关键字	26	25	41	15	68	44	06		
比较次数	1	5	1	2	2	1	1		

```
int LINSRCH(HT,k)/*在散列表HT[m]中查找关键字为k的结点 */
hashtable HT[ ];   keytype k;
{ int d,i=0;       /* i为冲突时的地址增量 */
   d=H(k);         /* d为散列地址 */
 while  ((i<m) && (HT[d].key ! =k)  && (HT[d].key ! =nil))
      { i++;d=(d+i) % m}
 return d;        /* 若HT[d].key =k查找成功，否则失败 */
}     /* LINSRCH */
```

```
void LINSERT(HT,s)       /* 将结点s插入散列表HT[m]中 */
hashtable s,HT[ ];
{ int d;
   d=LINSRCH(HT[ ], s.key)       /* 查找s的插入位置 */
   if (HT[d].key= =nil)   HT[d]=s;  /* d为开放地址，插入s */
   else  printf("ERROR");       /* 结点存在或表满 */
}      /* LINSERT */
```

散列表的查找算法-2

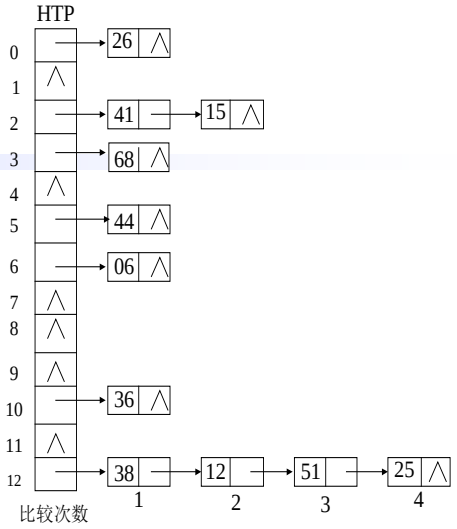
- 拉链法解决冲突的查找和插入算法：

```
typedef struct nodetype
{ keytype key;
   datatype other;
   struct nodetype  *next;
} chainhash;
chainhash *HTC[m];
```

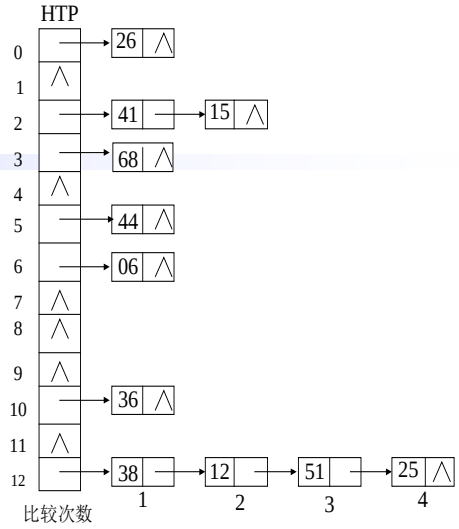
散列表的查找算法-2

- 拉链法解决冲突的查找和插入算法：

```
typedef struct nodetype
{ keytype key;
   datatype other;
   struct nodetype  *next;
} chainhash;
chainhash *HTC[m];
```



散列表的查找算法-2



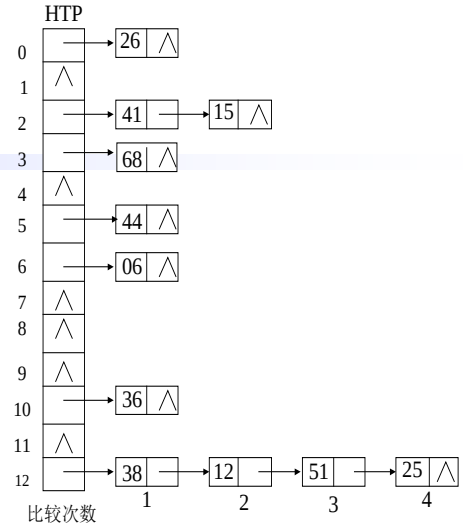
- 拉链法解决冲突的查找和插入算法：

```
typedef struct nodetype
{ keytype key;
   datatype other;
   struct nodetype  *next;
} chainhash;
chainhash *HTC[m];
```

- chainhash *CHNSRCH(HTC,k)/*在散列表HTC[m]中查找关键字为k的结点 */

```
chainhash *HTC[ ];  keytype k;
{ chainhash  *p; p==NULL;
   p=HTC[H(k)];     /* 取k所在链表的头指针 */
   while (p && (p→key ! =k))  p=p→next;      /* 顺序查找 */
   return p;      /* 查找成功，返回结点指针，否则返回空指针 */
}    / *CHNSRCH */
```

散列表的查找算法-2



- 拉链法解决冲突的查找和插入算法：

```
typedef struct nodetype
{ keytype key;
  datatype other;
  struct nodetype  *next;
} chainhash;
chainhash *HTC[m];
```

- chainhash *CHNSRCH(HTC,k)/*在散列表HTC[m]中查找关键字为k的结点 */

```
chainhash *HTC[ ];  keytype k;
{ chainhash  *p; p==NULL;
  p=HTC[H(k)];     /* 取k所在链表的头指针 */
  while (p && (p→key ! =k))  p=p→next;      /* 顺序查找 */
  return p;      /* 查找成功，返回结点指针，否则返回空指针 */
}   / *CHNSRCH */
```

- CINSERT(HTC,s)   /* 将结点*s插入散列表HTC[m]中 */

```
chainhash *s, *HTC[ ];
{ int d ;  chsinhash *p;
  p=CHNSRCH(HTC,s→key);     /* 查看表中有无待插结点 */
  if (p)  printf("ERROR");      /* 表中已有该结点 */
  else { d=H[s→key];  s→next=HTC[d];  HTC[d]=s; } /* 插入*s */
}    /* CINSERT * /
```

查找成功的平均查找长度比较

- 线性探查法(参见表7-3)：

ASL=(1+5+1+2+2+1+1+1+1+2+3)/11=20/11≈1.82

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
关键字	26	25	41	15	68	44	06				36		38	12	51
比较次数	1	5	1	2	2	1	1				1		1	2	3

查找成功的平均查找长度比较

- 线性探查法(参见表7-3)：

ASL=(1+5+1+2+2+1+1+1+1+2+3)/11=20/11≈1.82

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
关键字	26	25	41	15	68	44	06				36		38	12	51
比较次数	1	5	1	2	2	1	1				1		1	2	3

- 拉链法(参见图7-5)：

ASL=(1*7+2*2+3*1+4*1)/11=18/11≈1.64

查找成功的平均查找长度比较

- 线性探查法(参见表7-3)：

ASL=(1+5+1+2+2+1+1+1+1+2+3)/11=20/11≈1.82

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
关键字	26	25	41	15	68	44	06				36		38	12	51
比较次数	1	5	1	2	2	1	1				1		1	2	3

- 拉链法(参见图7-5)：

ASL=(1*7+2*2+3*1+4*1)/11=18/11≈1.64

- 当n=11，顺序查找和折半查找的平均查找长度：

$ASL_{sq}(11)=(11+1)/2=6$

$ASL_{bn}(11)=(1*1+2*2+3*4+4*4)/11=33/11=3$

散列表不成功查找分析

- 顺序查找和折半查找所需进行的关键字比较次数仅取决于表长；

散列表不成功查找分析

- 顺序查找和折半查找所需进行的关键字比较次数仅取决于表长；
- 散列查找需进行的关键字比较次数和待查结点有关，等概率情况下，散列表查找不成功的平均查找长度定义为对关键字需要执行的平均比较次数。

散列表不成功查找分析

- 顺序查找和折半查找所需进行的关键字比较次数仅取决于表长；

- 散列查找需进行的关键字比较次数和待查结点有关，等概率情况下，散列表查找不成功的平均查找长度定义为对关键字需要执行的平均比较次数。

1）线性探查法：

$ASK_{unsucc}=(8+7+6+5+4+3+2+1+1+1+2+1+11)/13$

$=52/13=4$

散列表不成功查找分析

- 顺序查找和折半查找所需进行的关键字比较次数仅取决于表长；

- 散列查找需进行的关键字比较次数和待查结点有关，等概率情况下，散列表查找不成功的平均查找长度定义为对关键字需要执行的平均比较次数。

1）线性探查法：

$ASK_{unsucc}=(8+7+6+5+4+3+2+1+1+1+2+1+11)/13$

$=52/13=4$

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
关键字	26	25	41	15	68	44	06				36		38	12	51
比较次数	1	5	1	2	2	1	1				1		1	2	3
关键字	15	25					36	6	8	68	51	26	41	12	44
比较次数	1	7					1	2	1	2	2	1	2	2	1

散列表不成功查找分析

2）链地址法：

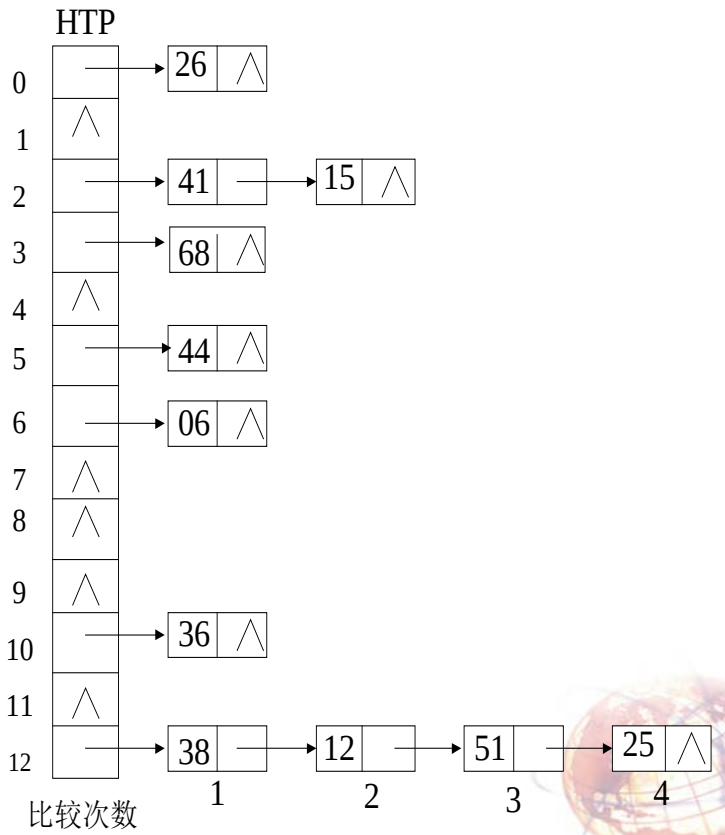
$ASL_{unsucc}=(1+0+2+1+0+1+1+0+0+0+1+0+4)/13$

$=11/13\approx0.85$

散列表不成功查找分析

2）链地址法：

$$ASL_{unsucc}=(1+0+2+1+0+1+1+0+0+0+1+0+4)/13$$
$$=11/13\approx 0.85$$



散列表不成功查找分析

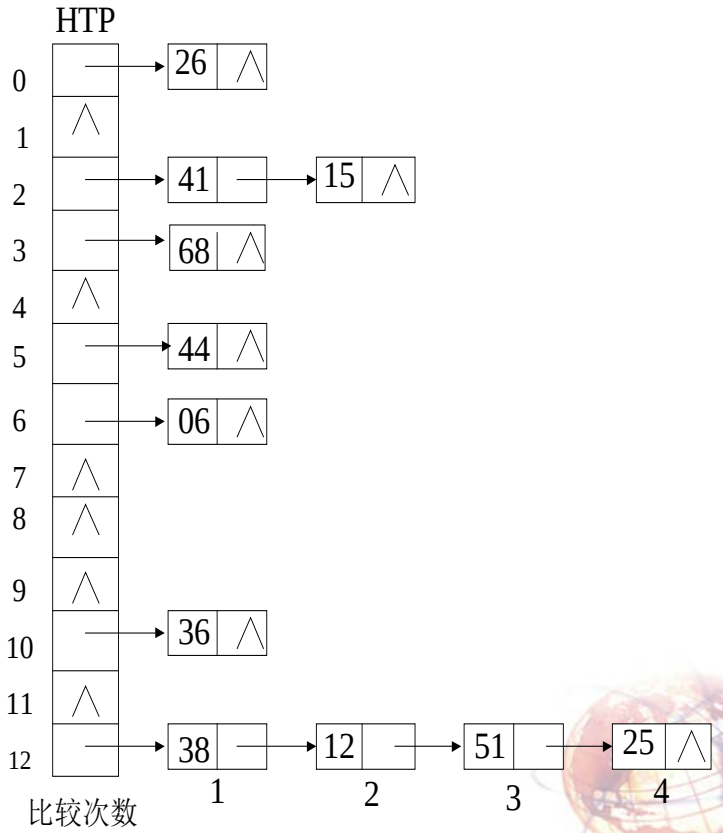
2）链地址法：

$ASL_{unsucc}=(1+0+2+1+0+1+1+0+0+0+1+0+4)/13$

$=11/13\approx0.85$

同一个散列函数、不同解决冲突方法构成的散列表，平均查找长度是不相同。

散列表技术具有很好的平均性能。



散列表的查找说明

几种不同方法解决冲突时散列表的平均查找长度

解决冲突的方法	平均查找长度	
	成功的查找	不成功的查找
线性探查法	$(1+1/(1-\alpha))/2$	$(1+1/(1-\alpha)^2)/2$
二次探查,随机探查或双散列函数探查法	$-\ln(1-\alpha)/\alpha$	$1/(1-\alpha)$
拉链法	$1+\alpha/2$	$\alpha+\exp(-\alpha)$

散列表的查找说明

几种不同方法解决冲突时散列表的平均查找长度

解决冲突的方法	平均查找长度	
	成功的查找	不成功的查找
线性探查法	$(1+1/(1-\alpha))/2$	$(1+1/(1-\alpha)^2)/2$
二次探查,随机探查或双散列函数探查法	$-\ln(1-\alpha)/\alpha$	$1/(1-\alpha)$
拉链法	$1+\alpha/2$	$\alpha+\exp(-\alpha)$

1）散列表的平均查找长度不是结点个数n或表长m的函数，而是装填因子α的函数。

散列表的查找说明

几种不同方法解决冲突时散列表的平均查找长度

解决冲突的方法	平均查找长度	
	成功的查找	不成功的查找
线性探查法	$(1+1/(1-\alpha))/2$	$(1+1/(1-\alpha)^2)/2$
二次探查,随机探查或双散列函数探查法	$-\ln(1-\alpha)/\alpha$	$1/(1-\alpha)$
拉链法	$1+\alpha/2$	$\alpha+\exp(-\alpha)$

1）散列表的平均查找长度不是结点个数n或表长m的函数，而是装填因子α的函数。

2）α越大，说明表越满，再插入新元素时发生冲突的可能性就越大，但α过小，空间的浪费就会过多。

散列表的查找说明

几种不同方法解决冲突时散列表的平均查找长度

解决冲突的方法	平均查找长度	
	成功的查找	不成功的查找
线性探查法	$(1+1/(1-\alpha))/2$	$(1+1/(1-\alpha)^2)/2$
二次探查,随机探查或双散列函数探查法	$-\ln(1-\alpha)/\alpha$	$1/(1-\alpha)$
拉链法	$1+\alpha/2$	$\alpha+\exp(-\alpha)$

1）散列表的平均查找长度不是结点个数n或表长m的函数，而是装填因子α的函数。

2）α越大，说明表越满，再插入新元素时发生冲突的可能性就越大，但α过小，空间的浪费就会过多。

选择合适的装填因子，把平均查找长度限制在一定范围内。

小结和习题

- 习 题
  - 12，14

第6章习题

16.判别邻接表表示的有向图中是否存在v_i到v_j的路径

深度优先搜索

```
int flag=0;
DFSL ( vexnode *ga, int m, int n )
{    edgenode  *p;
     if(m==n) { flag=1;
                    return;
                  }
      visited[m]=1;
      p=ga[m].link;
      while( p!=NULL )
           {  if (visited[p→adjvex]= =0)     DFSL(ga, p→adjvex, n);
              p=p→next;
            }
       return;
}
```

广度优先搜索

```
int flag=0;
BFSL(vexnode *ga, int m, int n )
{   int i;   edgenode  *p;
    SETNULL(Q);              /* 置空队 */
    if(m==n) { flag=1;   return;  }
    visited[m]=1;
    ENQUEUE(Q, m);
    while( !EMPTY(Q) )
          {   i=DEQUEUE(Q);
              p=ga[i]. link;
              while( p!=NULL)
                    {  if (visited[p→adjvex]!=1)
                          {   if(p→adjvex ==n)  { flag=1;  return; }
                              visited[p->adjvex]=1;
                               ENQUEUE(Q, p→adjvex);
                          }
                       p=p→next;
                    }
          }
}
```