11.3 冲突处理方法

❖ 处理冲突的方法

常用处理冲突的思路：

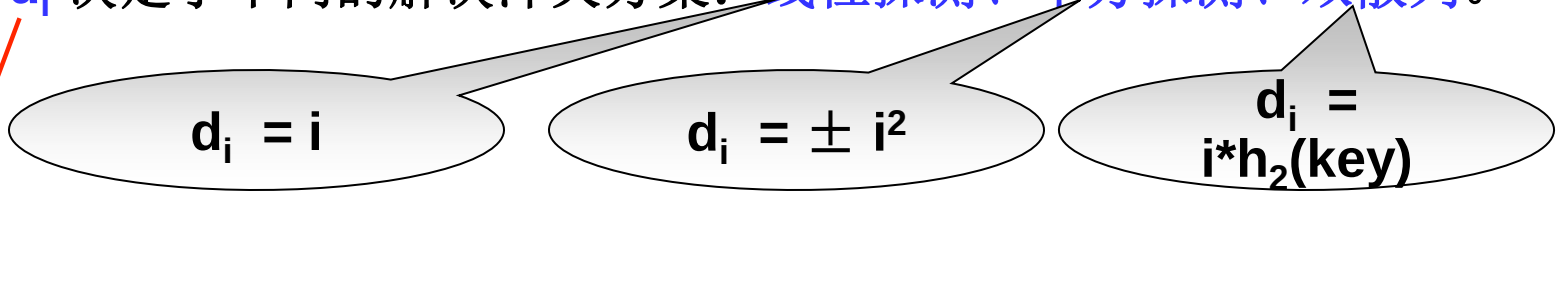
- 换个位置：开放地址法
- 同一位置的冲突对象组织在一起：链地址法

❖ 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 种规则去寻找另一空地址

❖ 开放定址法（Open Addressing）

- 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：
  $h_i(key) = (h(key)+d_i) \bmod TableSize \quad (1 \le i < TableSize)$
- 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。



1. 线性探测法（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。

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

- 散列表表长TableSize =13 （装填因子 α = 9/13 ≈ 0.69）；
- 散列函数为：h(key) = key mod 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(key)	3	7	7	0	9	7	10	9	8

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

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	$d_6 = 6$

注意”聚集”现象

散列表查找性能分析

- 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）

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。

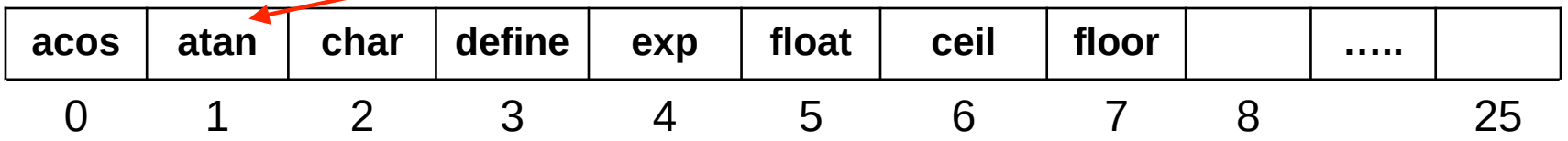
如：根据H(key)值分类

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

比如22/33，开始找位置0，不符合；再找位置1，不符合；再找位置2，为空，说明不在里面

[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.



【分析】

ASLs：表中关键词的平均查找比较次数

ASL s= （1+1+1+1+1+2+5+3）/ 8 = 15/8 ≈ 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 以a,b,c等开头

ASL u= （9+8+7+6+5+4+3+2+1*18）/ 26 = 62/26 ≈ 2.38

18种情况只需要查找一次

2. 平方探测法（Quadratic Probing）--- 二次探测

❖ 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。

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

- 散列表表长TableSize = 11，
- 散列函数为：h(key) = key mod 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(key)	3	7	7	0	9	7	10	9	8

关键词 key	47	7	29	11	9	84	54	20	30
散列地址h(key)	3	7	7	0	9	7	10	9	8
冲突次数	0	0	1	0	0	2	0	3	3

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	+2*2	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（Quadratic Probing）

是否有空间，平方探测(二次探测)就能找得到？

h(k)= k mod 5

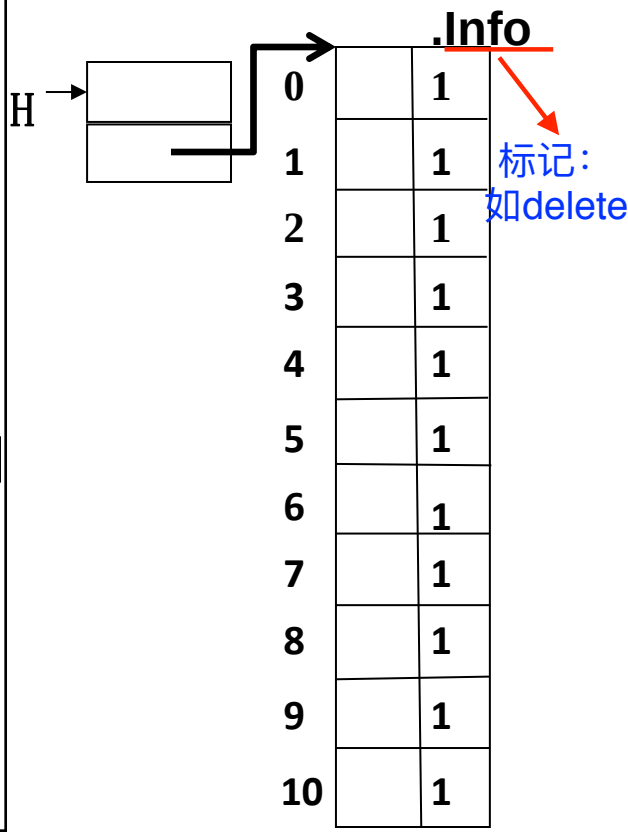
插入 **11**， h(11)=1

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。

```
HashTable  InitializeTable( int TableSize )
{
    HashTable H;
    int i;
    if ( TableSize < MinTableSize ){
        Error( "散列表太小" );
        return NULL;
    }
        /* 分配散列表 */
    H = (HashTable)malloc( sizeof( struct HashTbl ) );
    if ( H == NULL )
        FatalError( "空间溢出!!!" );
    H->TableSize = NextPrime( TableSize );
        /* 分配散列表 Cells */
    H->TheCells=(Cell *)malloc(sizeof( Cell )*H->TableSize);
    if( H->TheCells == NULL )
        FatalError( "空间溢出!!!" );
    for( i = 0; i < H->TableSize; i++ )
        H->TheCells[ i ].Info = Empty;
    return H;
}
```

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



```
Position  Find( ElementType Key, HashTable H )    /*平方探测*/
{    Position CurrentPos, NewPos;
     int CNum;       /*  记录冲突次数  */
     CNum = 0;
     NewPos = CurrentPos = Hash( Key, H->TableSize );
     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!! */
          if(++CNum % 2){ /*  判断冲突的奇偶次  */
               NewPos = CurrentPos + (CNum+1)/2*(CNum+1)/2;
               while( NewPos >= H->TableSize )
                    NewPos -= H->TableSize;
          } else {
               NewPos = CurrentPos - CNum/2 * CNum/2;
               while( NewPos < 0 )
                    NewPos += H->TableSize;
          }
     }
     return NewPos;
}
```

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

奇数时加1除2，偶数时除2

```
void  Insert( ElementType Key, HashTable H )
{     /* 插入操作 */
    Position Pos;
    Pos = Find( Key, H );
     if( H->TheCells[ Pos ].Info != Legitimate ) {
        /* 确认在此插入 */                    占用
        H->TheCells[ Pos ].Info = Legitimate;
        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。

重用:直接替换新值就好

3. 双散列探测法（Double Hashing）

双散列探测法：d_i 为$i*h_2(key)$，$h_2(key)$是另一个散列函数

探测序列成：$h_2(key)$，$2h_2(key)$，$3h_2(key)$，……

☞ 对任意的key，$h_2(key) \neq 0$ ！

☞ 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：

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

其中：$p < TableSize$，p、TableSize都是素数。

4. 再散列（Rehashing）

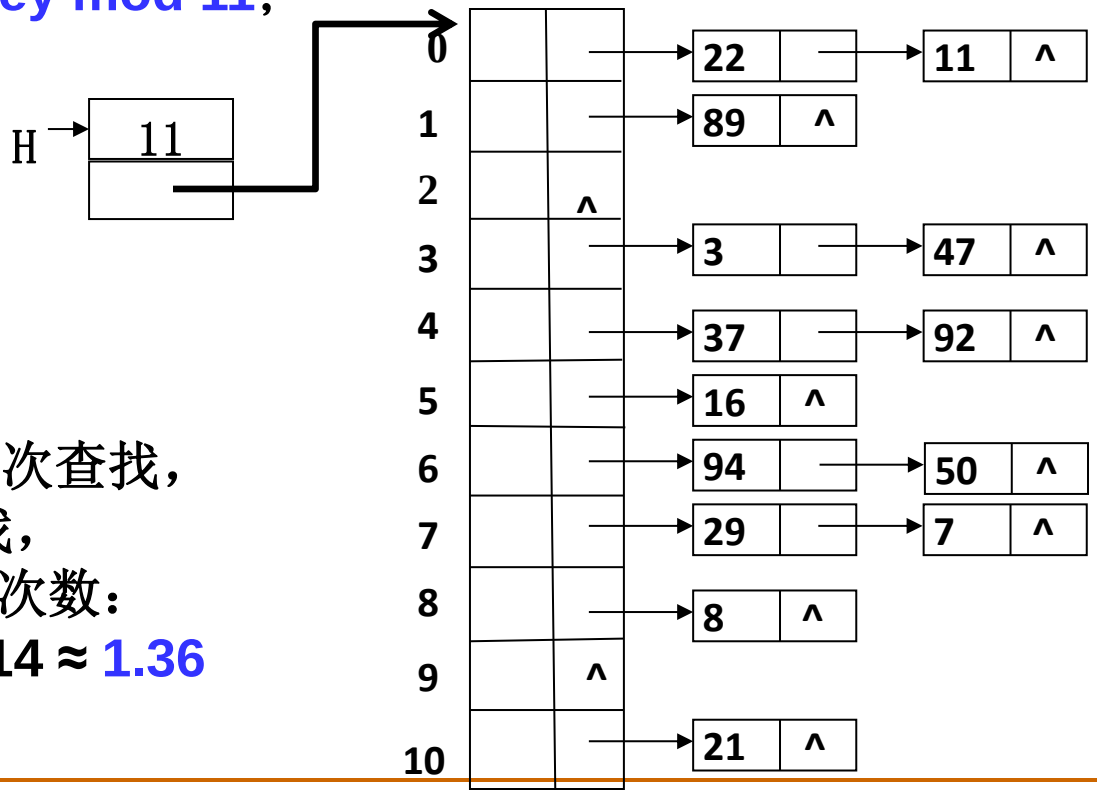
- 当散列表元素太多（即装填因子 α太大）时，查找效率会下降；
  - 实用最大装填因子一般取 **0.5 <= α<= 0.85**
- 当装填因子过大时，解决的方法是加倍扩大散列表，这个过程叫做“再散列（Rehashing）”
  需要对原来所有的元素重新计算

❖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(key) = key mod 11；
用分离链接法处理冲突。

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



- 表中有9个结点只需1次查找，
- 5个结点需要2次查找，
- 查找成功的平均查找次数：
  ASL s=（9+5*2）/ 14 ≈ 1.36

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

```
Position  Find( ElementType Key, HashTable H )
{   Position P;
    int Pos;

    Pos = Hash( Key, H->TableSize );  /*初始散列位置*/
    P = H->TheLists[Pos]. Next;          /*获得链表头*/
    while( P != NULL && strcmp(P->Element, Key) )
        P = P->Next; 典型遍历链表的方式
    return P;
}
```