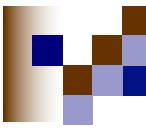


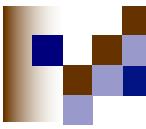
第7章 集合与符号表

- 以集合为基础的抽象数据类型
 - 集合的定义和记号
 - 定义在集合上的基本运算
- 用位向量实现集合
- 用链表实现集合
- 实现符号表的简单方法
- 用散列表实现符号表
 - 开散列
 - 闭散列
 - 散列函数及其效率
 - 闭散列的重新散列技术
- 应用举例



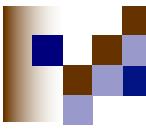
■ 7.1 以集合为基础的抽象数据类型

- 集合是表示事物的最有效的数学工具之一，生活中也随处可见集合的例子。
- 如：银行中所有储户帐号的集合；图书馆中所有藏书的集合；一个程序中所有标识符的集合；**2003级34班全体同学的集合等。**
- 在数据结构和算法的设计中，集合是许多重要抽象数据类型的基础。



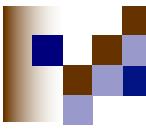
→ 7.1.1 集合的定义

- 集合：集合是由元素(成员)组成的一个类。集合的成员可以是一个集合，也可以是一个原子。
- 原子：不可再分的元素。
- 多重集合：允许同一元素在集合中多次出现的集合。
- 有序集：当由原子组成的集合具有线性序关系“ $<$ ”时，称该集合为有序集。“ $<$ ”是集合的一个线性序，它满足：
 - (1)若 a 、 b 是集合中任意2个原子，则 $a < b$, $a = b$ 和 $b < a$ 三者必居其一；
 - (2) a , b 和 c 是集合中的原子，且 $a < b$, $b < c$ 则 $a < c$ (传递性)。



→ 7.1.1 集合的定义

- 记录：集合中的元素通常是记录。
- 项(域)：元素的属性。元素通常有多个项。
- 键(值)：能唯一地标识元素的值。它也是元素的属性。
有序集通常以键(值)的序为序。
- 为方便叙述，常将一个元素当作一个键来处理，但要记住键只是元素记录中许多域中的一个域。



→ 7.1.2 集合的记号

- 枚举式：把集合的元素枚举在一对花括号中。

例如 $\{1, 4\}$ 表示由1和4两个元素组成的集合。集合中元素的枚举顺序是任意的。例如 $\{1, 4\}$ 和 $\{4, 1\}$ 表示同一集合。

这种方式只对有穷集可行。

- 解析式：把集合的元素应满足的条件解析地表达在一对花括号中。

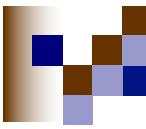
例如 $\{x \mid \text{存在整数 } y, \text{ 使 } x = y^2\}$ 表示由全体完全平方数组成的集合。

这种方式既可表示有穷集，又可表示无穷集。

→ 7.1.3 定义在集合上的基本运算

约定：其中大写字母表示一个集合，小写字母表示集合中的一个元素。

- **SetUnion(A,B)**: 并集运算
- **SetIntersection(A,B)**: 交集运算
- **SetDifference(A,B)** : 差集运算
- **SetAssign(A,B)** : 赋值运算(将B的值赋给A)
- **SetEqual(A,B)** : 判等运算
- **SetMember(x,S)** : 成员运算(x是否属于S)
- **SetInsert(x,S)** : 插入运算
- **SetDelete(x,S)** : 删除运算

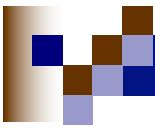


■ 7.2 用位向量实现集合

- 位向量是一种每个元素都是二进制位（即0/1值）的数据组。
- 当所讨论的集合都是全集合 $\{1, 2, \dots, n\}$ 的子集，而且n是一个不大的固定整数时，可以用位向量来实现集合。此时，对于任何一个集合 $A \subseteq \{1, 2, \dots, N\}$ ，可以定义它的特征函数为：

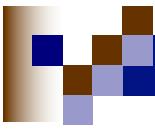
$$\delta_A(x) = \begin{cases} 1 & x \in A \\ 0 & x \notin A \end{cases}$$

- 用一个n位的向量v来存储集合A的特征函数值 $v[i] = \delta_A(i)$ ， $i=1, 2, \dots, n$ ，可以唯一地表示集合A。位向量v的第i位为1当且仅当i属于集合A。



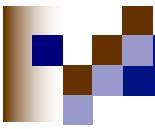
■ 7.2 用位向量实现集合

- 这种表示法的主要优点是**SetMember**, **SetInsert**和**SetDelete**运算都可以在常数时间内完成，只要访问相应的位即可。
- 在这种集合表示法下，执行并集运算、交集运算和差集运算所需的时间正比于全集合的大小n。



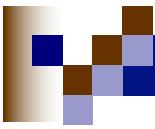
■ 位向量实现的集合结构**BitSet**的定义

```
typedef struct bitset *Set  
typedef struct bitset{  
    int setsize;      //集合大小  
    int arraysize;    //位数组大小  
    unsigned short *v; //位数组  
}Bitset;
```



- 创建一个用位向量实现，可存储集合大小为**size**的空集

```
Set SetInit(int size)
{
    int i;
    Set S=malloc(sizeof *S);
    S->setsize=size;
    S->arraysize=(size+15)>>4;
    S->v=malloc(arraysize*sizeof(unsigned short));
    for (i=0; i<arraysize; i++)  S->v[i]=0;
    return S;
}
```



- 通过复制表示集合的位向量实现赋值运算

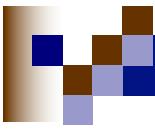
```
void SetAssign(Set A, Set B)
{
    int i;
    if (A->setsize !=B->setsize)
        Error("Sets are not the same size");
    for(i=0; i<A->arraysize; i++)
        A->v[i]=B->v[i];
}
```

- 通过检测元素在表示集合的位向量中相应位，来判定成员属性

```
int SetMemeber(int x, Set S)
{
    if (x<0 || x>=S->setsize)
        Error("Invalid member reference");
    return S->v[ArrayIndex(x)] & BitMask(x);
}
```

```
int ArrayIndex(int x)
{
    return x>>4;
}
```

```
unsigned short BitMask(int x)
{
    return 1<<(x & 15);
}
```



- 通过检测集合**A**和**B**的位向量来判定**A**和**B**是否相等

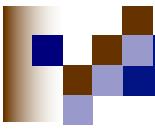
```
int SetEqual(Set A, Set B)
{
    int i,retval=1;
    if (A->setsize !=B->setsize)
        Error(".....");
    for(i=0; i<A->arraysize; i++)
        if(A->v[i] !=B->v[i])
            { retval=0; break;}
    return retval;
}
```

- 通过集合**A**和**B**的位向量按位或来实现并集运算

```
Set SetUnion(Set A, Set B)
{
    int i;
    Set tmp=SetInit(A->setszie);
    for(i=0; i<A->arraysize; i++)
        tmp->v[i]=A->v[i] | B->v[i];
    return tmp;
}
```

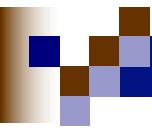
- 通过集合**A**和**B**的位向量按位与来实现交集运算

```
Set SetIntersection(Set A, Set B)
{
    int i;
    Set tmp=SetInit(A->setszie);
    for(i=0; i<A->arraysize; i++)
        tmp->v[i]=A->v[i] & B->v[i];
    return tmp;
}
```



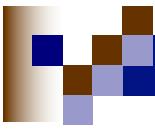
- 通过集合**A**和**B**的位向量按位与和按位异或来实现差集运算

```
Set SetDifference(Set A, Set B)
{
    int i;
    Set tmp=SetInit(A->setszie);
    for(i=0; i<A->arraysize; i++)
        tmp->v[i]=A->v[i] ^ (B->v[i] & A->v[i]);
    return tmp;
}
```



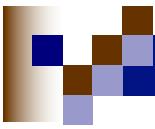
- 通过将集合**S**的位向量相应位置**1**来实现元素插入运算

```
void SetInsert(int x, Set S)
{
    if (x<0 || x>=S->setsize)
        Error(".....");
    S->v[ArrayIndex(x)] |=BitMask(x);
}
```



- 通过清除集合S的位向量相应位来实现元素删除运算

```
void SetDelete(int x, Set S)
{
    if (x<0 || x>=S->setsize)
        Error(".....");
    S->v[ArrayIndex(x)] &= ~BitMask(x);
}
```

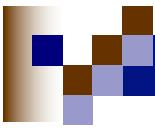


符号表

- 在算法设计中用到的集合，往往不做集合的并、交、差运算，而常要判定某个元素是否在给定的集合中，并且要不断地对这个集合进行元素的插入和删除操作。
- 以集合为基础，并支持**SetMember**、**SetInsert**和**SetDelete**三种运算的抽象数据类型叫做**符号表**。
- ADT符号表的应用
 - 公司的**VIP**客户列表
 - 一个地区的固定/移动电话号码列表
 - 软件开发中的数据字典
 - 网上的在线字典
 - 互联网/企业网/局域网网管中的**IP**地址列表等等

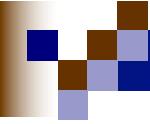
- 7.3 实现符号表的简单方法——用固定数组
- 用数组实现符号表的结构定义如下：

```
typedef struct atab *Table;  
typedef struct atab{  
    int arraysize;  
    int last;  
    SetItem*data;  
}Atab;
```



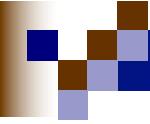
- 创建一个定长数组大小为**size**的空符号表

```
Table TableInit(int size)
{
    Table T=malloc(sizeof *T);
    T->arraysize=size;
    T->last=0;
    T->data=malloc(size*sizeof(SetItem));
    return T;
}
```



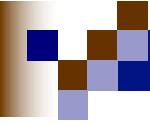
■ 符号表的成员查询运算

```
int TableMember(SetItem x, Table T)
{
    int i;
    for(i=0;i<T->last;i++)
        if(T->data[i]==x) return 1;
    return 0;
}
```



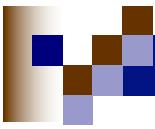
■ 符号表的元素插入运算

```
void TableInsert(SetItem x,Table T)
{
    if(! TableMember(x,T) && T->last<T->arraysize)
        T->data[T->last++]=x;
}
```



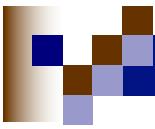
■ 符号表的元素删除运算

```
void TableDelete(SetItem x, Table T)
{
    int i=0;
    if (T->last>0){
        while(T->data[i]!=x && i<T->last) i++;
        if(i<T->last && T->data[i]==x)
            T->data[i]=T->data[--T->last];
    }
}
```



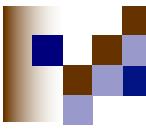
■ 7.3 实现符号表的简单方法——用固定数组

- 优点：结构简单，实现操作的逻辑简单。
- 缺点：
 - 所表示的集合的大小受到数组大小的限制。
 - 三个基本操作在最坏情况下都需要 $O(n)$ 。
 - 通常集合元素并不占满整个数组，因此，空间没有得到充分利用。



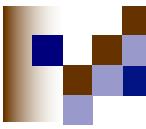
■ 7.3 用散列表实现符号表

- 实现符号表的另一个重要技巧是散列技术。
- 用散列来实现符号表可以使符号表的每个运算所需的平均时间是一个常数值，在最坏情况下每个运算所需的时间正比于集合的大小。
- 散列有两种形式：
 - 开散列：它将符号表元素放在一个潜无穷的空间里，能处理任意大小的集合。
 - 闭散列：它使用一个固定大小的存储空间，所能处理的集合大小不能超过其存储空间大小。



→ 7.3.1 开散列

- 基本思想：
 - 把符号表中的所有元素散列(**hashing**)，即映射到一个桶数组(散列表)的桶中。当有多个不同元素被散列到同一个桶时，这些元素用链在一个表里。期望散列能均匀，使得当桶数组的规模与符号表的规模同阶时，桶数组的每一个桶中大致有常数个元素，从而对符号表的三个基本操作都只需要常数时间。
 - 这里的问题是如何散列即如何构造散列(映射)函数去达到设想的期望？



→ 7.3.1 开散列

■ 开散列的数据要素

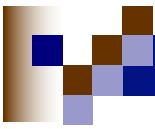
- 按照设想，开散列首先需要拥有一个桶数组，桶数组的规模与符号表的规模同阶，桶数组中的每一个桶有一个指针各指向一个链表。其次需要拥有一个散列(映射)函数，它把符号表中的元素分别散列(映射，分散)到各桶所对应的链表中。

→ 7.3.1 开散列

■ 散列函数的例子

- 假设符号表的元素是字符串x，桶数组的规模为101，那么，下面是散列函数的一个具体例子：

```
int hash1(char* x)
{
    int len=strlen(x), hashval = 0;
    for(int i=0;i< len ;i++)
        hashval += x[i];
    return hashval % 101;
}
```

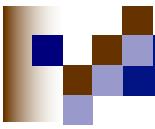


■ 用开散列实现的符号表结构**OpenHashTable**的定义

```
typedef struct open *OpenHashTable;  
typedef struct open{  
    int size; //桶数组的大小  
    int (*hf)(SetItem x); //散列函数  
    List *ht; //桶数组  
}Open;
```

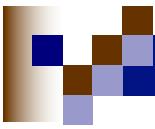
- 创建一个空的开散列表， 其桶数组的大小为nbuckets，
散列函数为hashf(x)

```
OpenHashTable HTInit(int nbuckets,int(*hashf)(SetItem x))  
{  
    int i;  
    OpenHashTable H=malloc(sizeof *H);  
    H->size=nbuckets;  
    H->hf=hashf;  
    H->ht=malloc(H->size*sizeof(List));  
    for(i=0;i<H->size;i++)  
        H->ht[i]=ListInit();  
    return H;  
}
```



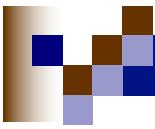
- 开散列表的成员查询函数**HTMember(x,H)**，根据元素x的散列函数值确定存储该元素的桶号，然后调用相应的表定位函数返回查询结果。

```
int HTMember(SetItem x,OpenHashTable H)
{
    int i=(*H->hf)(x) % H->size;
    return (ListLocate(x,H->ht[i])>0);
}
```



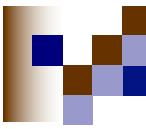
- 开散列表的元素插入运算**HTInsert(x,H)**，根据元素x的散列函数值确定存储该元素的桶号，然后在该桶的表首插入元素x。

```
void HTInsert(SetItem x,OpenHashTable H)
{
    int i;
    if (HTMember(x,H)) Error("Bad Input");
    i=(*H->hf)(x) % H->size;
    ListInsert(0,x,H->ht[i]);
}
```



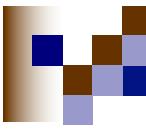
- 开散列表的元素删除运算**HTDelete(x,H)**，根据元素x的散列函数值确定存储该元素的桶号，然后调用相应的表元素删除函数删除元素x。

```
void HTDelete(SetItem x,OpenHashTable H)
{
    int i;
    i=(*H->hf)(x) % H->size;
    if (k=ListLocate(x,H->ht[i]))
        ListDelete(k,H->ht[i]);
}
```



→ 7.3.2 闭散列

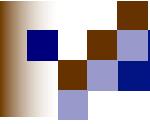
- 与开散列的相同点和区别
 - 相同点：把符号表中的所有元素散列(**hashing**)即映射到一个桶数组(散列表)的桶中。
 - 区别：（闭散列）桶数组(散列表)的桶直接用来存放符号表元素，而且一个桶只存放一个元素。出现多个元素散列到同一个桶时(这叫冲突)，需要按照确定的规则在桶数组中进行探测，找还没有存放元素的桶(这叫空桶)，然后将发生冲突的后到元素放入空桶，解决冲突，实现散列。



→ 7.3.2 闭散列

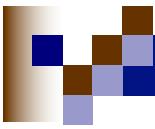
■ 闭散列引发的问题

- (1) 需要一个探测函数 $c(i)$, $i=0,1,2,\dots,\text{size}-1$:
 $c(0)=0$, 而且对于任意的 $0 \leq j \leq \text{size}-1$, $\{(j+c(0)) \% \text{size}, (j+c(1)) \% \text{size}, \dots, (j+c(\text{size}-1)) \% \text{size}\}$ 是
 $\{0,1,2,\dots,\text{size}-1\}$ 的重排, 保证不会漏测。
- (2) 需要对 $ht[]$ 的每一个桶的状态加以标记:
 - $state[k]=0$ 表示 $ht[k]$ 桶存放着元素。
 - $state[k]=1$ 表示 $ht[k]$ 桶一直是空桶。
 - $state[k]=2$ 表示 $ht[k]$ 桶现在是空桶但曾经存放过元素



■ 用闭散列实现的符号表结构**HashTable**的定义

```
typedef struct hashtable *HashTable;  
typedef struct hashtable{  
    int size; //桶数组大小  
    int (*hf)(SetItem x); //散列函数  
    SetItem *ht; //桶数组  
    int *state; //占用状态数组  
}Hashtable;
```

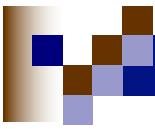


- 初始化桶数组**ht**和**state**，将每个桶都设置为空桶。
创建一个空的散列表

```
HashTable HTInit(int divisor, int (*hashf)(SetItem x))
{
    int i;
    HashTable H=malloc(sizeof *H);
    H->size=divisor;
    H->hf = hashf;
    H->ht=malloc(H->size*sizeof(SetItem));
    H->state=malloc(H->size*sizeof(int));
    for (i=0;i<H->size;i++)
        H->state[i]=1;
    return H;
}
```

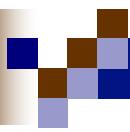
- 在散列表H的桶数组中查找元素x，并返回它在桶数组中的位置。

```
int FindMatch(SetItem x,HashTable H)
{
    int l,j,k;
    j=(*H->hf)(x);
    for (i=0;i<H->size;i++){
        k=(j+HashProb(i))%H->size;
        if (H->state[k]==1) break;
        if (!H->state[k] && H->ht[k]==x) return k;
    }
    return H->size;
}
int HashProb(int i)
{
    return i;
}
```



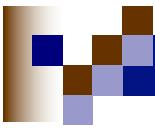
- 返回散列表H的桶数组中可存储元素x的未占用桶单元位置k。

```
int Unoccupied(SetItem x,HashTable H)
{
    int l,j,k;
    j=(*H->hf)(x);
    for (i=0;i<H->size;i++){
        k=(j+HashProb(i)) % H->size;
        if (H->state[k]) return k;
    }
    return H->size;
}
```



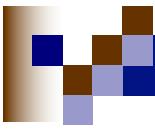
- 闭散列表通过调用函数**FindMatch**实现成员查询

```
int HTMember(SetItem x,HashTable H)
{
    int i=FindMatch(x,H);
    if (i<H->size && H->ht[i]==x) return 1;
    return 0;
}
```



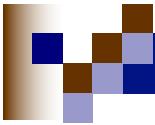
■ 闭散列表的元素插入运算

```
int HTInsert(SetItem x,HashTable H)
{
    int i;
    if(HTMember(x,H)) Error("Bad Input");
    i=Unoccupied(x,H);
    if(i<H->size){
        H->state[i]=0;
        H->ht[i]=x;
    }
    else Error("table full");
}
```

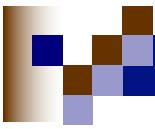


- 闭散列表的元素删除运算

```
int HTDelete(SetItem x,HashTable H)
{
    int i=FindMatch(x,H);
    if (i<H->size && H->ht[i]==x)
        H->state[i]=2;
}
```



- 上述删除算法**HTDelete**的缺点：
 - 在执行了大量元素删除运算后，大量的桶的状态标记为**2**，即大量的桶的状态标记既非**1**也非**0**，使得运算**FindMatch** 中的循环次数大大增加，从而导致运算**FindMatch**的速度大大减慢。因此人们提出**HTDelete**的一种改进版本**HTDelete1**。



■ 改进HTDelete的基本思想：

- 动机是希望腾出的空桶(记为 $ht[i]$)不仅仅可供新元素插入，而且能为提高还在桶数组中的元素(比如 $y = ht[j]$)的查找速度作出贡献：减少查找 y 的探测次数。
- 容易理解，如果不作任何改进，查找 y 的探测次数会等于插入 y 时的探测次数。如果插入 y 时 x 已在桶 $ht[i]$ 中且与 x 发生冲突而增加了插入的探测次数，那么，现在 x 不存在了，只要将 x 腾出的桶 $ht[i]$ 让 y 顶替，就可以使得将来查找 y 时减少探测次数。
- 因此，改进HTDelete的途径是在当前的桶数组中找能顶替 x 的 y 。如果找不到，就按HTDelete的原版处理；如果找得到，在用 y 顶替 x 之后还可以引起连锁反应。

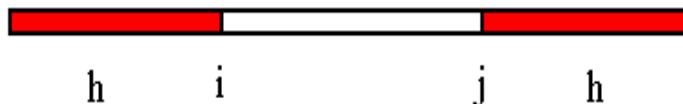
■ 改进HTDelete的细节——找能顶替x的y

■ 假设被删除元素x位于桶单元 $ht[i]$ 。现考察一个非空单元 $ht[j]$ 中的元素y，其散列函数值设为 $h=hf(y)$ ，则按从h出发的线性探测，只要i比j离h近即可使得在顶替后找y的探测次数减少。

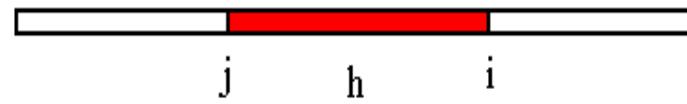
①当 $i < j$ 时，若 $i < h \leq j$ ，则不可用元素 $ht[j]$ 顶替 $ht[i]$ ；若 $h \leq i < j$ 或 $i < j < h$ ，则可用元素 $ht[j]$ 顶替 $ht[i]$ 。如下图(a)。

②当 $j < i$ 时，若 $j < h \leq i$ ，则可用元素 $ht[j]$ 顶替 $ht[i]$ ，如下图(b)；否则不可。

■ 这里以线性探测为前题，以顶替后减少探测次数为目标。



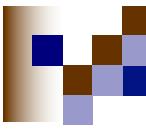
(a)



(b)

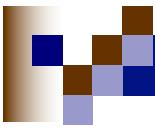
■ 改进HTDelete的 函数——HTDelete1

```
Void HTDelete1SetItem x,HashTable H)
{ int i=FindMatch(x);
  if (i<H->size) // &&H->ht[i]==x可以去掉
    for (;;) { //找ht[i]的顶替元素
      H->state[i]=2; //先按无顶替元素处理
      int j; //从(i+1)%size开始线性搜索可顶替元素
      for (j=(i+1)%H->size; !state[j]; j=(j+1)%H->size)
        { int h=(*H->hf)(H->ht[j]);
          if ((h<=i&&i<j)|| (i<j&&j<h)|| (j<h&&h<=i)) break;
            //ht[j]可顶替ht[i],跳出内for循环,做顶替工作
            // ht[j]不可顶替ht[i],继续线性搜索可顶替元素
        }
      if (H->state[j]) break; //跳出外for循环
      H->ht[i]=ht[j]; H->state[i]=state[j]; //做顶替工作
      i=j; //为构成循环找ht[j]的顶替元素
    }
}
```



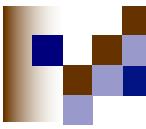
→ 7.3.3 散列函数及其效率

- 若能选择一个好的散列函数，将集合中的 n 个元素均匀地散列到 B 个桶中，那么每个桶中平均有 n/B 个元素。使得在开散列表中，**HTInsert**, **HTDelete**和**HTMember**运算都只要 $O(n/B)$ 的平均时间。当 n/B 为一常数时，每一个符号表运算都可在常数时间内完成。
- 因此：对于开散列表，关键在于选择一个好的散列函数。



→ 7.3.3 散列函数及其效率

- 几种计算简单且效果较好的散列函数构造方法：
 - (1)除余法： $h(k)=k \% m$
 - (2)数乘法： $h(k)=\lfloor B(ka - \lfloor ka \rfloor) \rfloor$
 - (3)平方取中法： $h(k)=\lfloor k^2 / c \rfloor \% B$
 - (4)基数转换法
 - (5)随机数法： $h(k)=\text{random}(k)$



→ 7.3.4 闭散列的重新散列技术

■ (1) 二次重新散列技术

它选取的探查桶序列为: $h(x), h_1(x), h_2(x), \dots, h_{2i}(x), h_{2i+1}(x)$,

其中, $h_{2i}(x) = (h(x) - i^2) \% B$ $h_{2i+1}(x) = (h(x) + i^2) \% B$

■ (2) 随机重新散列技术

它选取的探查桶序列为: $h_i(x) = (h(x) + d_i) \% B$, $i=1, 2, \dots, B-1$ 。

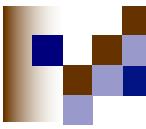
其中, d_1, d_2, \dots, d_{B-1} 是 $1, 2, \dots, B-1$ 的一个随机排列。

■ (3) 双重散列技术

■ 这种方法使用2个散列函数 h 和 h' 来产生探索序列:

$$h_i(x) = (h(x) + ih'(x)) \% B$$

其中 $i=1, 2, \dots, B-1$ 。要求 $h'(x)$ 的值和 B 互素。



实例

设有一组关键字{10,100,32,45,58,126,3,29,200,400,0}，散列表大小hashSize=13，表项下标从0到12，散列函数h(x)采用先将关键码各位数字折叠相加,再用%hashSize将相加的结果映像到表中的办法，采用二次散列技术

$$h_{2i-1}(x)=|h(x)+i^2|\%hashSize, \quad h_{2i}(x)=|h(x)-i^2|\%hashSize$$

解决冲突，画出相应的闭散列表，并指出每一个产生冲突的关键码分别产生了多少次冲突。

- 给出详细求解过程
- 等概率情况下，查找成功和查找不到时的平均查找长度