散列表

本章目标

- 散列表概念
- 散列函数构造方法
- 冲突处理方法
- 散列法性能分析

基本概念

- 散列表(hash table)，也叫哈希表，是根据关键码而直接进行访问的数据结构。也就是说，它通过把关键码映射到表中一个位置来访问记录，以加快查找的速度。这个映射函数叫做散列函数，存放记录的数组叫做散列表。
- 若结构中存在关键码为x的记录，则必定在hash(x)的存储位置上。由此，不需比较便可直接取得所查记录。称这个对应关系hash为散列函数(hash function)，按这个思想建立的表为散列表。

基本概念

电话号码	客户名称、借出日期、借出光碟等

- 影碟出租店维护一张表格，以电话号码作为关键码，为了提高查找速度，可以用选择哈希表进行存储
- 假设影碟出租店有一万张光碟，每天借出与归还均不超出500人次。因此哈希表维护500条记录即可。
- 我们发现真正要存储的记录比关键码总数要少得多。

基本概念

- 散列函数是一个压缩映象函数。关键码集合比散列表地址集合大得多。因此有可能经过散列函数的计算，把不同的关键码映射到同一个散列地址上，这就产生了冲突 (Collision)。即key1≠key2，而hash(key1)=hash(key2)，这种现象称冲突。我们将key1与key2称做同义词。
- 由于关键码集合比地址集合大得多，冲突很难避免。所以对于散列方法，需要讨论以下两个问题：
  - 对于给定的一个关键码集合，选择一个计算简单且地址分布比较均匀的散列函数，避免或尽量减少冲突；
  - 拟订解决冲突的方案。

散列函数选取原则

- 散列函数的选择有两条标准：简单和均匀
- 简单指散列函数的计算简单快速，能在较短时间内计算出结果。
- 均匀指散列函数计算出来的地址能均匀分布在整个地址空间。若key是从关键字码集合中随机抽取的一个关键码，散列函数能以等概率均匀地分布在表的地址集{0，1，…，m-1}上，以使冲突最小化。

散列函数构造方法

- 直接定址法
- 数字分析法
- 平方取中法
- 折叠法
- 随机数法
- 除留余数法
- 乘余取整法

直接定址法

- 此类函数取关键码的某个线性函数值作为散列地址：

  hash (key) = a * key + b　　{ a, b为常数 }

- 这类散列函数是一对一的映射，一般不会产生冲突。但是，它要求散列地址空间的大小与关键码集合的大小相同。

数字分析法

- 构造：对关键字进行分析，取关键字的若干位或其组合作哈希地址。
- 适于关键字位数比哈希地址位数大，且可能出现的关键字事先知道的情况。

例 有80个记录，关键字为8位十进制数，哈希地址为2位十进制数

①② ③ ④⑤⑥ ⑦⑧

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

分析：①只取8
②只取1
③只取3、4
⑧只取2、7、5
④⑤⑥⑦数字分布近乎随机

所以：取④⑤⑥⑦任意两位或两位与另两位的叠加作哈希地址

平方取中法

- 取关键字平方后中间几位作哈希地址。适于关键字位数不定情况。
- 具体方法：先通过求关键字的平方值扩大相近数的差别，然后根据表长度取中间的几位数作为散列函数值。又因为一个乘积的中间几位数和乘数的每一位都相关，所以由此产生的散列地址较为均匀。

例：

标识符	内码	内码的平方	散列地址
A	01	01	001
*A*1	0134	20420	042
*A*9	0144	23420	342
B	02	4	004
DMAX	04150130	21526443617100	443
*DMAX*1	0415013034	52644735222151420	352
AMAX	01150130	1354236617100	236
*AMAX*1	0115013034	34542465222151420	652

标识符的八进制内码表示及其平方

折叠法

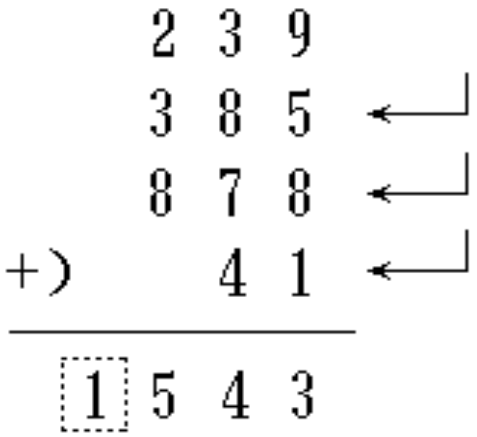
- 此方法把关键码自左到右分成位数相等的几部分，每一部分的位数应与散列表地址位数相同，只有最后一部分的位数可以短一些。
- 把这些部分的数据叠加起来，就可以得到具有该关键码的记录的散列地址。
- 有两种叠加方法：
  - 移位法 — 把各部分的最后一位对齐相加；
  - 分界法 — 各部分不折断，沿各部分的分界来回折叠，然后对齐相加，将相加的结果当做散列地址。
- 一般当关键码的位数很多，而且关键码每一位上数字的分布大致比较均匀时，可用这种方法得到散列地址。

折叠法

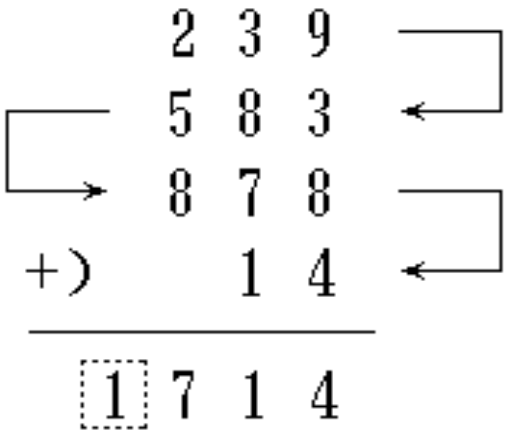
- 示例：设给定的关键码为 key = 23938587841，若存储空间限定 3 位, 则划分结果为每段 3 位. 上述关键码可划分为 4段：

  239　　385　　878　　41

移位法



分界法



- 把超出地址位数的最高位删去, 仅保留最低的3位，做为可用的散列地址。

随机数法

- 选择一个随机函数，取关键字的随机函数值为它的散列地址，即

  hash(key) = random(key)

  其中random为伪随机函数，但要保证函数值是在0到m-1之间。

除留余数法

- 设散列表中允许的地址数为 m, 散列函数为：

  hash (key) = key % p　　*p* ≤ *m*

- 若p取100，则关键字159、259、359互为同义词。我们选取的p值应尽可能使散列函数计算得到的散列地址与各位相关，根据经验，p最好取一个不大于 m，但最接近于或等于 m 的质数 p，或选取一 个不小于 20 的质因数的合数作为除数
- 示例：有一个关键码 key = 962148，散列表大小 m = 25，即 HT[25]。取质数 p= 23。散列函数 hash (key) = key % p。则散列地址为

  hash (962148) = 962148 % 23 = 12

- 可以按计算出的地址存放记录。需要注意的是，使用上面的散列函数计算出来的地址范围是 0到 22，因此，从23到24这几个散列地址实际上在一开始是不可能用散列函数计算出来的，只可能在处理溢出时达到这些地址。

乘余取整法

- 使用此方法时，先让关键码 *key* 乘上一个常数 $A\ (0 < A < 1)$，提取乘积的小数部分。然后，再用整数 n 乘以这个值，对结果向下取整，把它做为散列的地址。散列函数为：

$$hash(key) = \lfloor n * (A * key \% 1) \rfloor$$

其中，“$A * key \% 1$”表示取 $A * key$ 小数部分：

$$A * key \% 1 = A * key - \lfloor A * key \rfloor$$

Donald Ervin Knuth（高德纳）建议A取 $(\sqrt{5}-1)/2$

示例：设关键码 $key = 123456$，$n = 10000$，且取 $A = (\sqrt{5}-1)/2 = 0.6180339$，则 $hash(key) = \lfloor 10000*(0.6180339*key \% 1) \rfloor$

因此有：

$$hash(123456) =$$
$$= \lfloor 10000*(0.6180339*123456 \% 1) \rfloor =$$
$$= \lfloor 10000 * (76300.0041151\ldots \% 1) \rfloor =$$
$$= \lfloor 10000 * 0041151\ldots \rfloor = 41$$

此方法的优点是对 n 的选择不很关键。

常见字符串哈希函数

```
unsigned int SDBMHash(char *str)
{
    unsigned int hash = 0;

    while (*str)
    {
        // equivalent to: hash = 65599*hash + (*str++);
        hash = (*str++) + (hash << 6) + (hash << 16) - hash;
    }

    return (hash & 0x7FFFFFFF);
}
```

常见字符串哈希函数

```
unsigned int RSHash(char *str)
{
    unsigned int b = 378551;
    unsigned int a = 63689;
    unsigned int hash = 0;

    while (*str)
    {
        hash = hash * a + (*str++);
        a *= b;
    }

    return (hash & 0x7FFFFFFF);
}
```

常见字符串哈希函数

```
unsigned int JSHash(char *str)
{
    unsigned int hash = 1315423911;

    while (*str)
    {
        hash ^= ((hash << 5) + (*str++) + (hash >> 2));
    }

    return (hash & 0x7FFFFFFF);
}
```

常见字符串哈希函数

```
unsigned int PJWHash(char *str)
{
    unsigned int BitsInUnignedInt = (unsigned int)(sizeof(unsigned int) * 8);
    unsigned int ThreeQuarters    = (unsigned int)((BitsInUnignedInt  * 3) / 4);
    unsigned int OneEighth        = (unsigned int)(BitsInUnignedInt / 8);
    unsigned int HighBits         = (unsigned int)(0xFFFFFFFF) << (BitsInUnignedInt - OneEighth);
    unsigned int hash             = 0;
    unsigned int test             = 0;

    while (*str)
    {
        hash = (hash << OneEighth) + (*str++);
        if ((test = hash & HighBits) != 0)
        {
            hash = ((hash ^ (test >> ThreeQuarters)) & (~HighBits));
        }
    }

    return (hash & 0x7FFFFFFF);
}
```

常见字符串哈希函数

```
unsigned int ELFHash(char *str)
{
    unsigned int hash = 0;
    unsigned int x    = 0;

    while (*str)
    {
        hash = (hash << 4) + (*str++);
        if ((x = hash & 0xF0000000L) != 0)
        {
            hash ^= (x >> 24);
            hash &= ~x;
        }
    }

    return (hash & 0x7FFFFFFF);
}
```

常见字符串哈希函数

```
unsigned int BKDRHash(char *str)
{
    unsigned int seed = 131; // 31 131 1313 13131 131313 etc..
    unsigned int hash = 0;

    while (*str)
    {
        hash = hash * seed + (*str++);
    }

    return (hash & 0x7FFFFFFF);
}
```

常见字符串哈希函数

```
unsigned int DJBHash(char *str)
{
    unsigned int hash = 5381;

    while (*str)
    {
        hash += (hash << 5) + (*str++);
    }

    return (hash & 0x7FFFFFFF);
}
```

常见字符串哈希函数

```
unsigned int APHash(char *str)
{
    unsigned int hash = 0;
    int i;

    for (i=0; *str; i++)
    {
        if ((i & 1) == 0)
        {
            hash ^= ((hash << 7) ^ (*str++) ^ (hash >> 3));
        }
        else
        {
            hash ^= (~((hash << 11) ^ (*str++) ^ (hash >> 5)));
        }
    }

    return (hash & 0x7FFFFFFF);
}
```

冲突处理的方法

- 链地址法
- 开地址法
- 建立公共溢出区

链地址法

- 这种基本思想：将所有哈希地址为i的元素构成一个称为同义词链的链表，并将链表的头指针存在哈希表的第i个单元中，因而查找、插入和删除主要在同义词链中进行。
- 开散列方法首先对关键码集合用某一个散列函数计算它们的存放位置。
- 若设散列表地址空间的所有位置是从0到m-1，则关键码集合中的所有关键码被划分为m个子集，具有相同地址的关键码归于同一子集。我们称同一子集中的关键码互为同义词。每一个子集称为一个桶。
- 通常各个桶中的表项通过一个链表链接起来，称之为同义词子表。所有桶号相同的表项都链接在同一个同义词子表中，各链表的表头结点组成一个向量。

链地址法

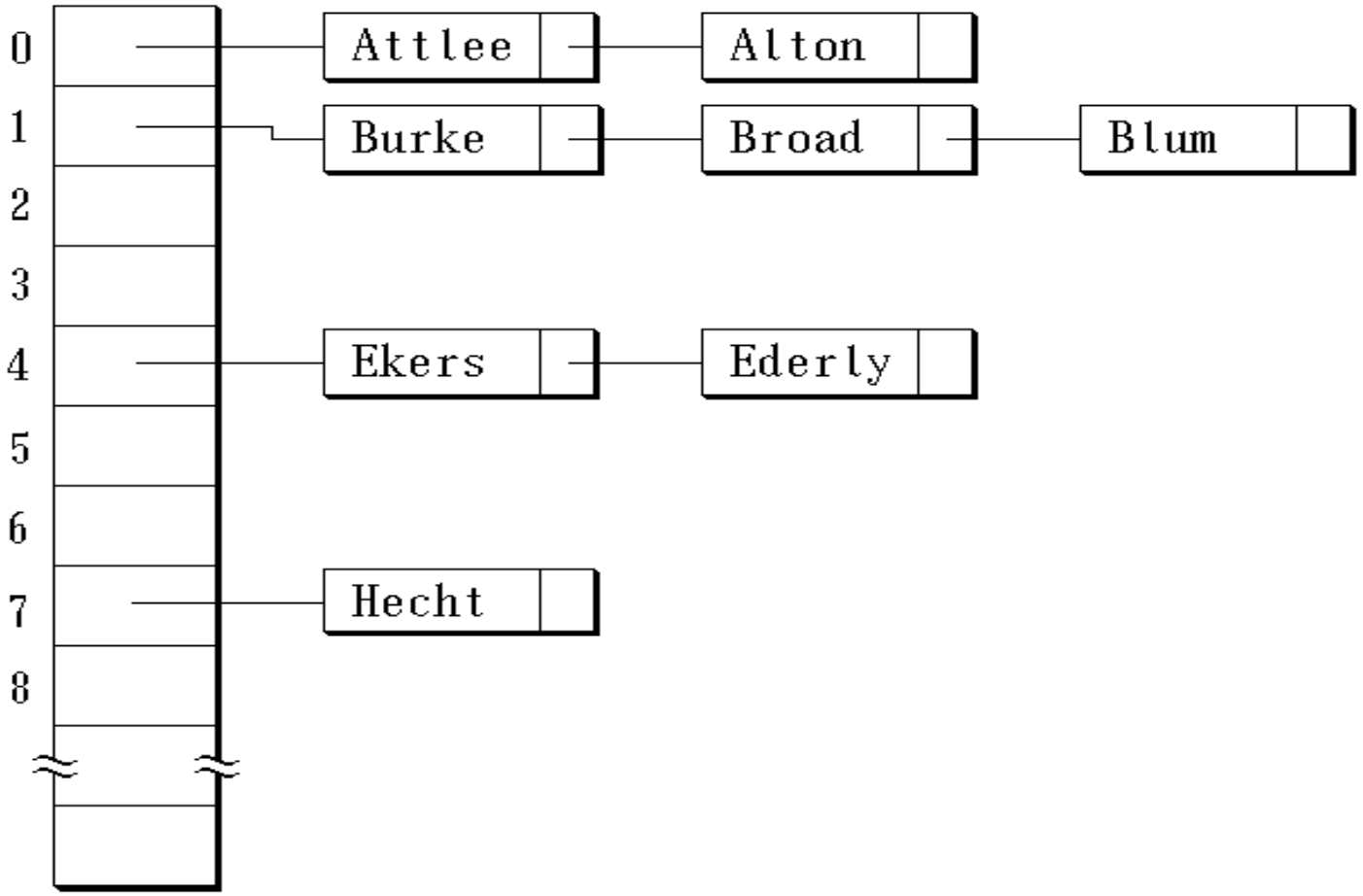
- 假设给出一组表项，它们的关键码为 Burke, Ekers, Broad, Blum, Attlee, Alton, Hecht, Ederly。采用的散列函数是：取其第一个字母在字母表中的位置。

  ***hash* (*x*) = *ord* (*x*) - *ord* (‘*A*’)**

- 这样，可得

  hash (Burke) = 1　　*hash* (Ekers) = 4
  hash (Broad) = 1　　*hash* (Blum) = 1
  hash (Attlee) = 0　　*hash* (Hecht) = 7
  hash (Alton) = 0　　*hash* (Ederly) = 4

链地址法


0
1
2
3
4
5
6
7
8
Attlee
Alton
Burke
Broad
Blum
Ekers
Ederly
Hecht

链地址法

- 通常，每个桶中的同义词子表都很短，设有n个关键码通过某一个散列函数，存放到散列表中的 m 个桶中。那么每一个桶中的同义词子表的平均长度为 n / m。这样，以搜索平均长度为 n / m 的同义词子表代替了搜索长度为 n 的顺序表，搜索速度快得多。
- 应用链地址法处理溢出，需要增设链接指针，似乎增加了存储开销。事实上，由于开地址法必须保持大量的空闲空间以确保搜索效率，如二次探查法要求装填因子$\alpha \leq 0.5$，而表项所占空间又比指针大得多，所以使用链地址法反而比开地址法节省存储空间。

开地址法

- 基本思想：当关键码key的哈希地址$H_0 = hash(key)$出现冲突时，以H_0为基础，产生另一个哈希地址H_1，如果H_1仍然冲突，再以H_0为基础，产生另一个哈希地址H_2，…，直到找出一个不冲突的哈希地址H_i，将相应元素存入其中。这种方法有一个通用的再散列函数形式：

  $H_i = (H_0 + d_i) \% m$，　$i = 1, 2, \ldots, m-1$

- 其中H_0为$hash(key)$，m为表长，d_i称为增量序列。增量序列的取值方式不同，相应的再散列方式也不同。主要有以下四种：
  - 线性探测再散列
  - 二次探测再散列
  - 伪随机探测再散列
  - 双散列法

线性探测再散列

- 需要搜索或加入一个表项时，使用散列函数计算桶号：

$$H_0 = hash(key)$$

- 一旦发生冲突，在表中顺次向后寻找“下一个”空桶 H_i 的公式为

$$H_i = (H_{i-1} + 1) \% m, \quad i = 1, 2, \ldots, m-1$$

- 即用以下的线性探查序列在表中寻找“下一个”空桶的桶号

$$H_0+1, H_0+2, \ldots, m-1, 0, 1, 2, \ldots, H_0-1$$

- 亦可写成

$$H_i = (H_0 + i) \% m, \quad i = 1, 2, \ldots, m-1$$

线性探测再散列

- 假设给出一组表项，它们的关键码为 Burke, Ekers, Broad, Blum, Attlee, Alton, Hecht, Ederly。采用的散列函数是：取其第一个字母在字母表中的位置。

  ***hash* (*x*) = *ord* (*x*) - *ord* ('*A*')**

- 这样，可得

  hash (Burke) = 1　　*hash* (Ekers) = 4
  hash (Broad) = 1　　*hash* (Blum) = 1
  hash (Attlee) = 0　　*hash* (Hecht) = 7
  hash (Alton) = 0　　*hash* (Ederly) = 4

线性探测再散列

- 又设散列表为*HT*[26]，$m = 26$。采用线性探查法处理溢出，则上述关键码在散列表中散列位置如图所示。括号内的数字表示找到空桶时的探测次数。

0	1	2	3	4
Attlee	Burke	Broad	Blum	Ekers
(1)	(1)	(2)	(3)	(1)

5	6	7	8	9
Alton	Ederly	Hecht		
(6)	(3)	(1)		

堆积

- 散列地址不同的结点争夺同一个后继散列地址的现象称为堆积(Clustering)。
- 这将造成不是同义词的结点也处在同一个探测序列中，从而增加了探测序列长度，即增加了查找时间。
- 若散列函数不好、或装填因子过大，都会使堆积现象加剧。

二次探测再散列

- 为改善“堆积”问题，减少为完成搜索所需的平均探查次数，可使用二次探测法。
- 通过某一个散列函数对表项的关键码 x 进行计算，得到桶号，它是一个非负整数。

$$H_0 = hash(x)$$

- 二次探测法在表中寻找“下一个”空桶的公式为：

$$H_i = (H_0 + i^2) \% m,$$

$$H_i = (H_0 - i^2) \% m, \quad i = 1, 2, \ldots, (m-1)/2$$

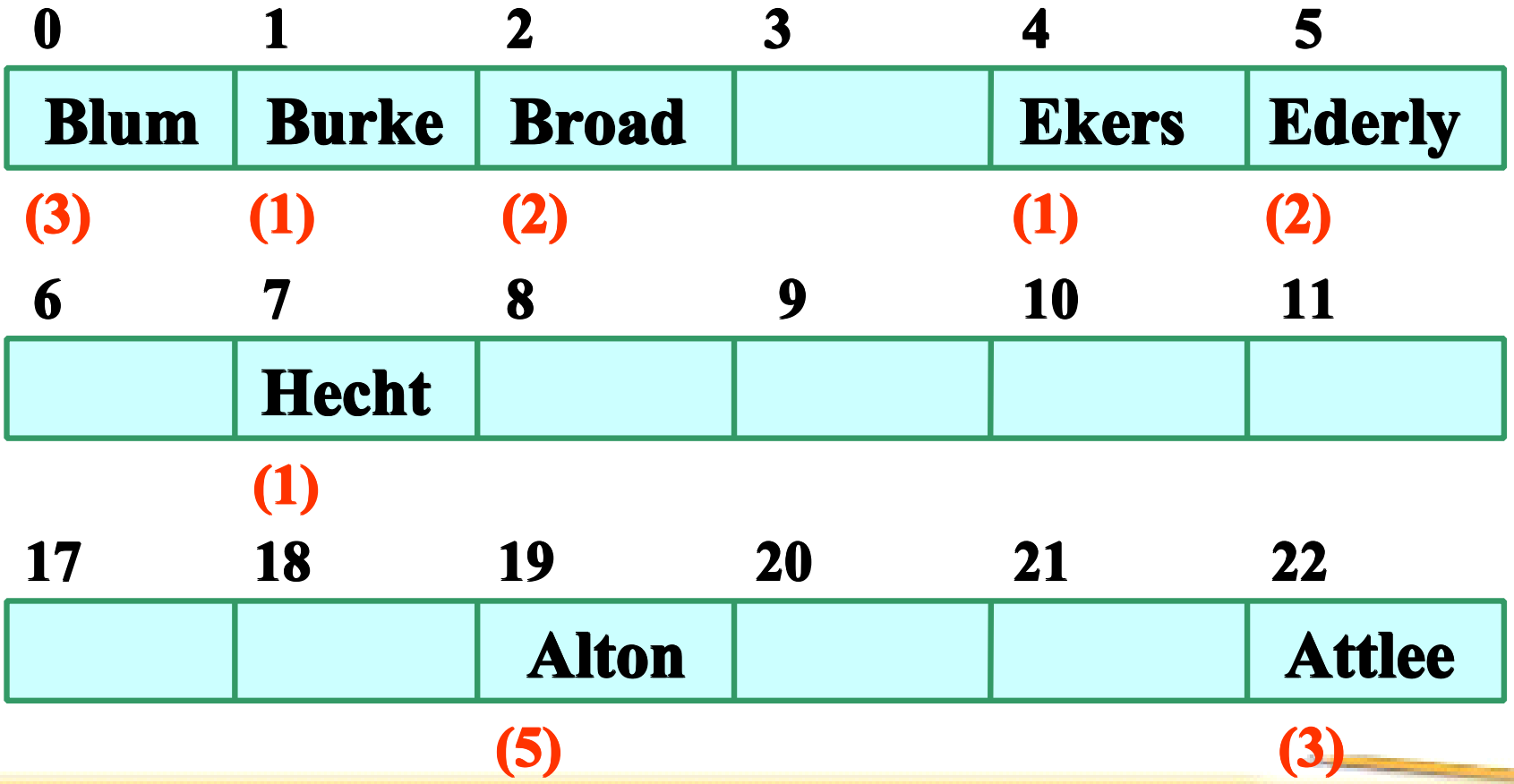
$$1^2, -1^2, 2^2, -2^2, \ldots$$

- 式中的 m 是表的大小，它应是一个值为 $4k+3$ 的质数，其中 k 是一个整数。这样的质数如 3, 7, 11, 19, 23, 31, 43, 59, 127, 251, 503, 1019, …。
- 探测序列形如 H_0, H_0+1, H_0−1, H_0+4, H_0−4, …。
- 在做 $(H_0 - i^2) \% m$ 的运算时，当 $H_0 - i^2 < 0$ 时，运算结果也是负数。实际算式可改为

```
if ( (j = (H0 - i*i) % m) < 0) j += m
```

二次探测再散列

- 若设表的长度为*TableSize*＝23，则利用二次探查法所得到的散列结果如图所示。



二次探测再散列

$$H_{i-1}^{(0)} = (H_0 + (i-1)^2) \% m,$$

$$H_{i-1}^{(1)} = (H_0 - (i-1)^2) \% m.$$

$$H_i^{(0)} = (H_0 + i^2) \% m,$$

$$H_i^{(1)} = (H_0 - i^2) \% m.$$

相减，可以得到：

$$H_i^{(0)} - H_{i-1}^{(0)} = (2 * i - 1) \% m,$$

$$H_i^{(1)} - H_{i-1}^{(1)} = (-2 * i + 1) \% m.$$

从而

$$H_i^{(0)} = (H_{i-1}^{(0)} + 2 * i - 1) \% m,$$

$$H_i^{(1)} = (H_{i-1}^{(1)} - 2 * i + 1) \% m.$$

伪随机探测再散列

- 伪随机探测再散列d_i取随机数

$$H_i = (hash(\text{key}) + d_i) \% m, \quad i = 1, 2, \ldots, m-1$$

双散列法

- 二次探测法的缺点是不易探测到整个散列空间。更好的方法是使用双散列法。
- 使用双散列方法时，需要两个散列函数。
- 第一个散列函数 *Hash*( ) 按表项的关键码 *key* 计算表项所在的桶号 $H_0 = Hash(key)$。
- 一旦冲突，利用第二个散列函数 ***ReHash*( )** 计算该表项到达“下一个”桶的移位量。它的取值与 *key* 的值有关，要求它的取值应当是小于地址空间大小 *TableSize*，且与 *TableSize* 互质的正整数。
- 若设表的长度为 *m* = *TableSize*，则在表中寻找“下一个”桶的公式为：

  $j = H_0 = Hash(key)$，　$p = ReHash(key)$;

  $j = (j + p) \% m$;

  *p*是小于*m*且与*m*互质的整数（使发生冲突的同义词地址均匀分布在整个表中，否则可能造成同义词地址的循环计算）
- 利用双散列法，按一定的距离，跳跃式地寻找“下一个”桶，减少了“堆积”的机会。

双散列法

- 双散列法的探查法公式：

$$H_i = (H_{i-1} + ReHash(key)) \% m \qquad i=1, 2, \ldots, m-1$$

- 双散列法的探查序法公式也可写成：

$$H_i = (H_0 + i * ReHash(key)) \% m \qquad i=1, 2, \ldots, m-1$$

- 最多经过$m-1$次探查，它会遍历表中所有位置，回到H_0位置。
- *Rehash*( )的取法很多。例如，当m是质数时，可定义
  - $ReHash(key) = key \% (m-2) + 1$
  - $ReHash(key) = \lfloor key / m \rfloor \% (m-2) + 1$
- 当 m 是 2 的方幂时，$ReHash(key)$ 可取从 0 到$m-1$ 中的任意一个奇数。

双散列法

- 示例：给出一组表项关键码{ 22, 41, 53, 46, 30, 13, 01, 67 }。散列函数为：$Hash(x)=(3x)\ \%\ 11$。
- 散列表为 $HT[0..10]$，$m=11$。因此，再散列函数为 $ReHash(x)=(7x)\ \%\ 10+1$

$$H_i=(H_{i-1}+(7x)\ \%\ 10+1)\ \%\ 11,\quad i=1,2,\ldots$$

- $H_0(22)=0$　　$H_0(41)=2$　　$H_0(53)=5$

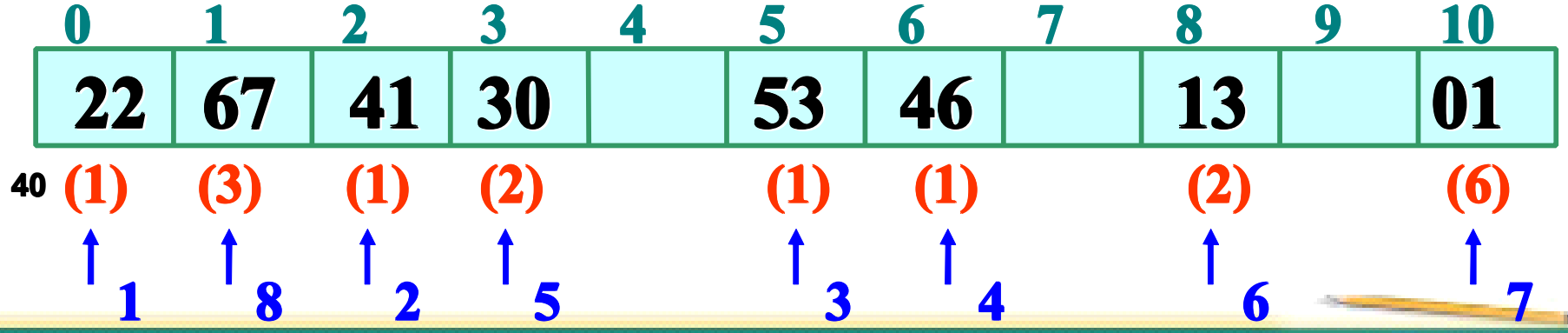
  $H_0(46)=6$　　$H_0(30)=2$ 冲突 $H_1=(2+1)\ \%\ 11=3$

  $H_0(13)=6$ 冲突 $H_1=(6+2)\ \%\ 11=8$

  $H_0(01)=3$ 冲突 $H_1=(3+8)\ \%\ 11=0$　$H_2=(0+8)\ \%\ 11=8$

  $H_3=(8+8)=5$　$H_4=(5+8)\ \%\ 11=2$　$H_5=(2+8)\ \%11=10$

  $H_0(67)=3$ 冲突 $H_1=(3+10)\ \%11=2$　$H_2=(2+10)\ \%11=1$



建立公共溢出区

- 这种方法的基本思想是：将哈希表分为基本表和溢出表两部分，凡是和基本表发生冲突的元素，一律填入溢出表。

散列法性能分析

- 由于冲突的存在，散列法在进行搜索的时候，实际上也要进行关键码的比较。
- 用平均搜索长度*ASL* (*Averagy Search Length*)衡量散列方法的搜索性能。
- 根据搜索成功与否，它又有搜索成功的平均搜索长度*ASLsucc*和搜索不成功的平均搜索长度*ASLunsucc*之分。
- 搜索成功的平均搜索长度 *ASLsucc* 是指能搜索到待查表项的平均探查次数。它是找到表中各个已有表项的探查次数的平均值。
- 搜索不成功的平均搜索长度 *ASLunsucc* 是指在表中搜索不到待查的表项，但找到插入位置的平均探查次数。它是表中所有可能散列到的位置上要插入新元素时为找到空桶的探查次数的平均值。

散列法性能分析

- 已知一组关键码为(26,36,41,38,44,15,68,12,06,51)，用除留余数法构造散列函数。
- 用线性探测法解决冲突。取表长m=13。
- 散列函数为 hash(key) = key % 13

$$ASLsucc = (1\times 6 + 2\times 2 + 3\times 1 + 9\times 1)/10 = 2.2$$

$$ASLunsucc = (9+8+7+6+5+4+3+2+1+1+2+1+10)/13 \approx 4.54$$

- 用链地址法解决冲突。取表长m=13。

$$ASLsucc = (1\times 7 + 2\times 2 + 3\times 1)/10 = 1.4$$

$$ASLunsucc = (1+0+2+1+0+1+1+0+0+0+1+0+3)/13 \approx 0.77$$

哈希法性能分析

- 哈希查找过程仍是一个给定值与关键字进行比较的过程。
- 评价哈希查找效率仍要用ASL
- 哈希法中影响关键字比较次数的因素有三个：
- 哈希函数、处理冲突的方法以及哈希表的装填因子。哈希表的装填因子 α 的定义如下：

$$\alpha = \frac{\text{哈希表中元素个数}}{\text{哈希表的长度}}$$

- 散列表的装填因子 α 表明了表中的装满程度。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。

哈希法性能分析

- 当装填因子 α 较高时，选择的散列函数不同，散列表的搜索性能差别很大。在一般情况下多选用除留余数法，其中的除数在实用上应选择不含有20以下的质因数的质数。
- 对散列表技术进行的实验评估表明，它具有很好的平均性能，优于一些传统的技术，如平衡树。
- 但散列表在最坏情况下性能很不好。如果对一 个有 n 个关键码的散列表执行一次搜索或插入操作，最坏情况下需要 $O(n)$ 的时间。
- Knuth对不同的溢出处理方法进行了概率分析。

哈希法性能分析

- Donald Ervin Knuth（高德纳）对不同的冲突处理方法进行了概率分析，如图下所示：

处理溢出的方法		平均查找长度 *ASL*	
		成功的查找 ***Sn***	不成功的查找 ***Un***
开地址法	线性探查法		
	伪随机探查法 二次探查法 双散列法		
链地址法			

哈希法性能分析

- 下图给出一些实验结果，列出在采用不同的散列函数和不同的处理溢出的方法时，搜索关键码所需的对桶访问的平均次数。实验数据为{ 33575, 24050, 4909, 3072, 2241, 930, 762, 500 }
- 从图中可以看出，链地址法优于开地址法；在散列函数中，用除留余数法作散列函数优于其它类型的散列函数，最差的是折叠法。

哈希法性能分析

$\alpha = n / m$	0.50		0.75		0.90		0.95	
散列函数种类	链地址法	开地址法	链地址法	开地址法	链地址法	开地址法	链地址法	开地址法
平方取中	1.26	1.73	1.40	9.75	1.45	310.14	1.47	310.53
除留余数	1.19	4.52	1.31	10.20	1.38	22.42	1.41	25.79
移位折叠	1.33	21.75	1.48	65.10	1.40	710.01	1.51	118.57
分界折叠	1.39	22.97	1.57	48.70	1.55	69.63	1.51	910.56
数字分析	1.35	4.55	1.49	30.62	1.52	89.20	1.52	125.59
理论值	1.25	1.50	1.37	2.50	1.45	5.50	1.48	10.50

搜索关键码时所需对桶的平均访问次数

几种查找方法对比

- 顺序查找算法复杂度　　$O(N)$
- 二分查找算法复杂度　　$O(\log_2 N)$
- 二叉树查找算法复杂度　　$O(\log_2 N)$
- 哈希查找算法复杂度　　$O(1)$

- 后三种效率都比较快，哈希查找是最快的。二分查找要求数据是已经排序过的，二叉树查找与哈希查找没有此问题
- 二叉树查找效率虽不及哈希查找，但是它在进行遍历的时候能将顺便将数据排好序。而哈希查找很难找到一种方法按顺序遍历表空间。

总结

- 散列表概念
- 散列函数构造方法
- 冲突处理方法
- 散列法性能分析