第十章 检索

宋国杰

北京大学信息科学技术学院

gjsong@pku.edu.cn

内容

- 基本概念
- 10.1 线性表的检索
- 10.2 散列表的检索

基本概念

- 检索
  - 在记录集合中找到“关键码值=给定值” 的记录
  - 或找到关键码值“符合特定约束条件”的记录集
- 检索效率非常重要
  - 尤其对于大数据量
  - 需要对数据进行特殊的存储处理

平均检索长度（ASL）

- 检索运算的主要操作：关键码的比较
- 平均检索长度(Average Search Length，AVL)
  - 检索过程中对关键码的平均比较次数
  - 衡量检索算法优劣的时间标准

$$ASL = \sum_{i=1}^{n} P_i C_i$$

为检索第 i 个元素的概率

找到第 i 个元素所需的比较次数

平均检索长度的例子

- 假设线性表为(a, b, c)，检索a、b、c的概率分别为0.4、0.1、0.5
  - 顺序检索算法的平均检索长度为

    $0.4\times 1 + 0.1\times 2 + 0.5\times 3 = 2.1$
  - 即平均需要2.1次给定值与表中关键码值的比较才能找到待查元素

提高检索效率的方法

- 预排序
  - 排序算法本身比较费时
  - 只是预处理（在检索之前已经完成）
- 建立索引
  - 检索时充分利用辅助索引信息
  - 牺牲一定的空间，从而提高检索效率
- 散列技术
  - 把数据组织到一个表中
  - 根据关键码的值确定表中记录的位置

检索算法的分类

- 基于线性表的检索
  - 如顺序检索、二分检索
- 根据关键码值的直接访问
  - 如根据数组下标的直接检索
- 树索引的方法
  - 如二叉树检索、B树等
- 基于属性的检索
  - 如倒排表、倒排文件等

10.1 基于线性表的检索

- 10.1.1 顺序检索
- 10.1.2 二分检索
- 10.1.3 分块检索

10.1.1 顺序检索

- 针对线性表里的所有记录，逐个进行关键码和给定值的比较
  - 检索成功：若某个记录的关键码和给定值比较相等；
  - 检索失败：找遍了仍找不到。
- 物理存储：可以顺序、或者链接
- 排序要求：无

“监视哨”顺序检索算法

- 检索成功返回元素位置，检索失败统一返回0;

```
template  <class Type>  int SeqSearch(vector<Item<Type>*>&
    dataList, int n, Type k) {                //查找关键字K是否在序列当中
    int i=n;
    //将第0个元素设为待检索值
    dataList[0] = k;                          //设监视哨
    while(dataList[i] != k)
        i--;
    return i;                                 //返回元素位置
}
```

顺序检索性能分析

- 检索成功
  - 假设检索每个关键码是等概率的：$P_i = 1/n$

$$ASL_S = \sum_{i=1}^{n} P_i * (n-i+1) = \frac{1}{n} * \sum_{i=1}^{n} (n-i+1)$$

$$= \frac{n+1}{2}$$

- 检索失败：设置了一个监视哨

$$ASL_F = n+1$$

顺序检索平均检索长度

- 假设检索成功的概率为p，检索失败的概率为q=(1-p)

$$
\begin{aligned}
\mathrm{ASL} &= p \cdot ASL_S + q \cdot ASL_F \\
&= p \cdot \frac{n+1}{2} + q \cdot (n+1) \\
&= p \cdot \frac{n+1}{2} + (1-p)(n+1) \\
&= (n+1)(1-p/2)
\end{aligned}
$$

- 因此，（n+1)/2 < ASL < (n+1)

顺序检索优缺点

- 优点：插入元素可以直接加在表尾$\Theta(1)$
- 缺点：检索时间太长$\Theta(n)$

10.1.2 二分检索法

- 前提条件：**待检索序列有序！！**
- 将dataList[i] .Key与给定值K比较
  - 三种情况：

（1）K = Key，检索成功，返回dataList[i]

（2）K < Key，若有则一定排在dataList[i]]前

（3）K > Key，若右则一定排在dataList[i]后

- 加快缩小进一步检索的区间

举例：关键码18 low=1 high=9

1	2	3	4	5	6	7	8	9
15	17	18	22	35	51	60	88	93

low mid high

第一次：l=1, h=9, mid=5; 18<array[5]=35

第二次：l=1, h=4, mid=2; 18>array[2]=17

第三次：l=3, h=4, mid=3; 18＝array[3]=18

二分法检索算法

```
template  <class Type> int BinSearch (vector<Item<Type>*>& dataList, int
    length, Type k){
    int low=1, high=length, mid;
    while (low<=high)  {                    //结束条件！！
        mid = (low+high)/2;
        if (k<dataList[mid]->getKey())
            high = mid-1;                   //右缩检索区间
        else if (k>dataList[mid]->getKey())
            low = mid+1;                    //左缩检索区间
        else return mid;                    //成功返回位置
    }
    return 0; //检索失败，返回0
}   //为与顺序检索保持一致，位置0不存放实际元素；
```

二分法检索性能分析

- 最大检索长度（<u>完全二叉树的高度！</u>）

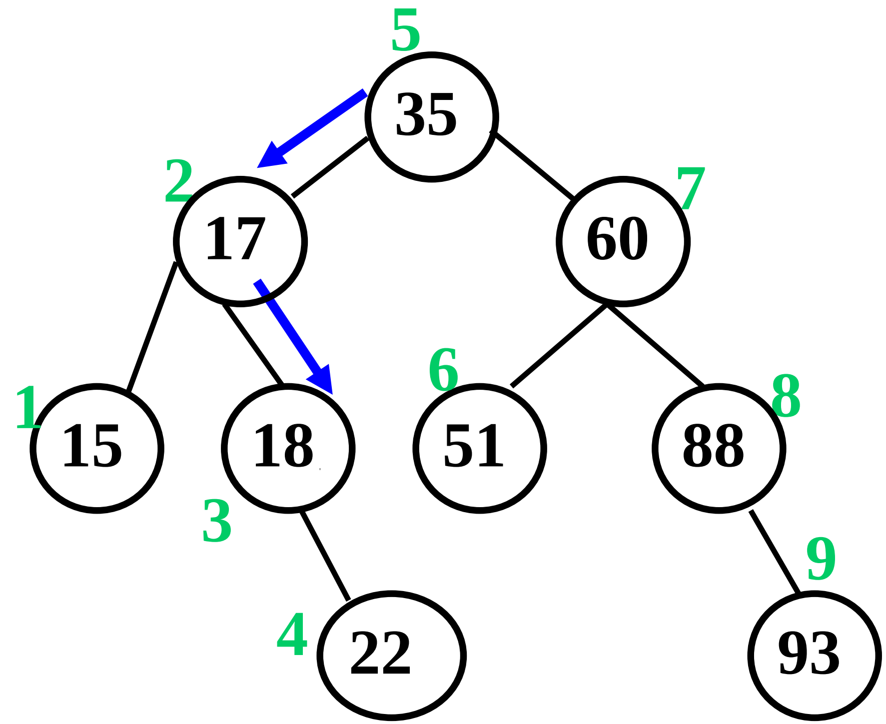
$$\lceil \log_2(n+1) \rceil$$

- 失败的检索长度是

$$\lceil \log_2(n+1) \rceil$$

或

$$\lfloor \log_2(n+1) \rfloor$$



二分法检索性能分析（续）

- 成功的平均检索长度为：

$$\mathrm{ASL} = \frac{1}{n}\left(\sum_{i=1}^{j} i \cdot 2^{i-1}\right)$$

$$= \frac{n+1}{n}\log_2(n+1) - 1$$

$$\approx \log_2(n+1) - 1$$

- 优缺点
  - 优点：平均与最大检索长度相近，检索速度快
  - 缺点：要排序、顺序存储，不易更新(插/删)

10.1.3 分块检索

- 顺序检索与二分检索的折衷
  - 既有较快的检索
  - 又有较灵活的更改

分块检索思想

- “按块有序”
  - 设线性表中共有n个数据元素，将表分成b块
    - ✓不需要有序
    - ✓每一块可能不满
  - 块内无序：每一块中的关键码不一定有序
  - 块间有序
    - ✓前块中最大关键码 < 后块中最小关键码

索引表

- 索引表
  - 各块中的最大关键码
  - 各块起始位置
  - 块中有效元素个数（块可能不满）
- 索引表是一个递增有序表
  - 索引表是分块有序的

分块检索

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
22	12	13	9	8		33	42	44	24	48		60	80	74	49	86	53

起始位置：	**0**	**6**	**12**
最大关键码：	**22**	**48**	**86**
元素个数：	**5**	**5**	**6**

性能分析

- 分块检索为两级检索
  - 索引表检索
    - ✓设在索引表中确定块号的时间开销是ASL_b
  - 块内检索
    - ✓在块中查找记录的时间开销为ASLw
- $\mathrm{ASL}(n) = \mathrm{ASL}_b + \mathrm{ASL}_w$

分块检索性能分析(续2)

- 若在索引表中用顺序检索，在块内也用顺序检索

$$\mathrm{ASL}_b = \frac{b+1}{2} \quad \mathrm{ASL}_w = \frac{s+1}{2}$$

$$\mathrm{ASL} = \frac{b+1}{2} + \frac{s+1}{2} = \frac{b+s}{2} + 1$$

$$= \frac{n+s^2}{2s} + 1$$

- 当$s = \sqrt{n}$时，**ASL取最小值（S为块内元素个数）**

$\mathbf{ASL} = \sqrt{n} + 1 \approx \sqrt{n}$

分块检索性能分析(续3)

- 若采用二分法检索确定记录所在的子表，则检索成功时的平均检索长度为

$$ASL = ASL_b + ASL_w$$

$$\approx \log_2 (b+1) - 1 + (s+1)/2$$

$$\approx \log_2 (1 + n / s) + s/2$$

分块检索的优缺点

- 优点：
  - 插入、删除容易
  - 无大量记录移动
- 缺点：
  - 增加一个辅助索引表
  - 初始线性表分块排序
  - 元素大量插入/删除，或分布不均匀时性能下降

内容

- 基本概念
- 10.1 线性表的检索
- 10.2 散列表的检索

10.3 散列检索

- 10.3.0 散列问题
- 10.3.1 散列函数
- 10.3.2 开散列方法
- 10.3.3 闭散列方法
- 10.3.4 闭散列的实现
- 10.3.5 效率分析

10.3.0 散列中的基本问题

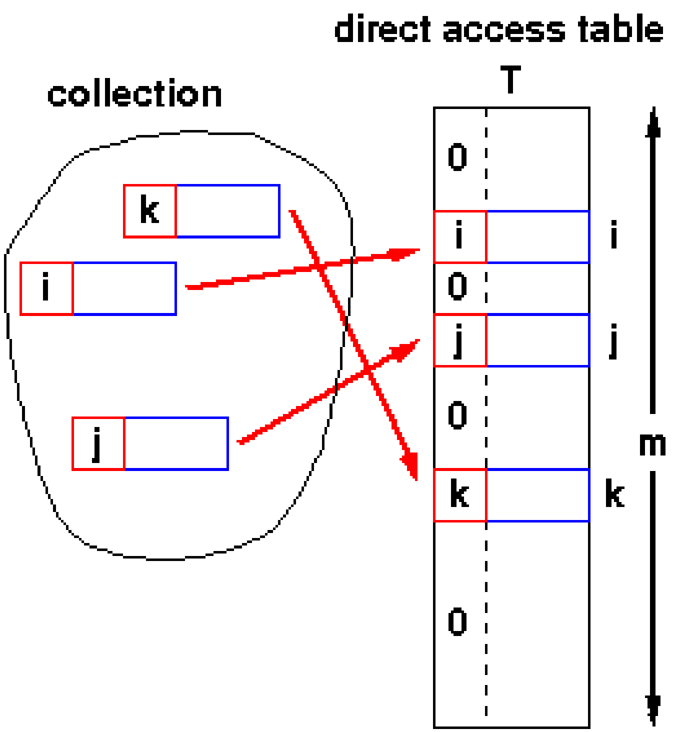
- **基于关键码比较的检索**
  - 顺序检索：==, !=
  - 二分法、树型： >, == , <
  - **复杂性与问题规模n直接相关**
    - ✓ 当问题规模很大时，上述方法检索效率低下！
- **理想情况**
  - 根据关键码值，直接找到记录的存储地址
  - 不需把待查关键码与候选记录集合进行逐个比较

数组直接寻址带来的启示

- **例如，读取指定下标的数组元素**
  - 根据数组的起始存储地址、以及数组下标值而直接计算出来的，所花费的时间是**O(1)**
  - **与数组规模n无关**
- **受此启发，计算机科学家发明了散列方法（Hash, 称“哈希”，或称“杂凑”）**
  - **建立起关键码与存储地址之间的直接映射关系**
  - 一种非常有用的检索方法

散列基本思想

- 一个确定的函数h
- 待检索的关键码K
- 函数值h(K)
- 根据h(K)计算记录存储位置
  - 散列表的存储空间是一维数组
  - 散列地址是数组的下标



例子1

- **例10.1：已知线性表关键码集合为：**

  S = { and, array, begin, do, else, end, for, go, if, repeat, then, until, while, with}

- 设散列表为：

  char HT[26][8];

- 散列函数H(key)的值，取为关键码key中的第一个字母在字母表{a, b, c, …, z}中的序号，即：

  H(key)=key[0] – ‘a’

例子1（续）

散列地址	关键码
0	(and, array)
1	begin
2	
3	do
4	(end, else)
5	for
6	go
7	
8	if
9	
10	
11	
12	

散列地址	关键码
13	
14	
15	
16	
17	repeat
18	
19	then
20	until
21	
22	(while, with)
23	
24	
25	

例子 2

- 修改散列函数：散列函数的值为key中首尾字母在字母表中序号的平均值，即：

```
int H1(char key[])
{
   int i = 0;
   while ((i<8) && (key[i]!='\0'))  i++;
   return((key[0] + key(i-1) – 2*'a') /2 )
}
```

例子2（续）

散列地址	关键码
0	
1	and
2	
3	end
4	else
5	
6	if
7	begin
8	do
9	
10	go
11	for
12	array

散列地址	关键码
13	while
14	with
15	until
16	then
17	
18	repeat
19	
20	
21	
22	
23	
24	
25	

几个重要概念

- **负载（或者装填）因子 $\alpha = n/M$**
  - n：散列表中已有结点数
  - M：散列表空间大小
- **冲突**
  - 将不同的关键码映射到相同的散列地址
  - 实际应用中，不产生冲突的散列函数极少存在
- **同义词**
  - 发生冲突的两个关键码

两个重要问题

I. 散列函数的构造

- 使结点“均匀分布”，尽可能降低“冲突”现象发生的概率

II. 冲突解决的方法

- 发生了冲突，如何解决？

I. 散列函数的构造

- 散列函数：把关键码映射到存储位置的函数，通常用 h 来表示

$$Address = Hash\,(\,key\,)$$

散列函数的选取原则

1. 运算简单
2. 函数值在散列表范围内：**[0, M-1]**
3. 关键码不同时，尽可能使其散列值亦不相同

需要考虑各种因素

- 关键码长度
- 散列表大小
- 关键码分布情况
- 记录的检索频率
- …

常用散列函数选取方法

- 除余法
- 乘余取整法
- 平方取中法
- 数字分析法
- 基数转换法
- 折叠法
- ELFhash字符串散列函数

1. 除余法

- 用关键码除以M(往往取散列表长度)，并取余数作为散列地址。散列函数为：

$$h(x) = x \bmod M$$

- 通常选择**质数**作为M值
  - 函数值依赖于变量x的所有位，而不是某些位，增大了均匀分布的可能性

M不取偶数

- 若把M设置为偶数
  - x是偶数，h(*x*)也是偶数
  - x是奇数，h(*x*)也是奇数
- 缺点：分布不均匀
  - 如果偶数关键码比奇数关键码出现的概率大，那么函数值就不能均匀分布
  - 反之亦然

除余法面临的问题

- 除余法的潜在缺点
  - 连续的关键码映射成连续的散列值

- 虽然能保证连续的关键码不发生冲突
- 但也意味着要占据连续的数组单元
- 可能导致散列性能的降低

2. 乘余取整法

- 散列函数

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

- 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，把它作为散列地址
- “A * key % 1”表示取 A * key 小数部分

乘余取整法示例

- 设关键码 key = 123456, n = 10000且取

  $A = (\sqrt{5}-1)/2 = 0.6180339$,

- 因此有

$$hash(123456) =$$

$$= \lfloor 10000*(0.6180339*123456 \% 1) \rfloor$$

$$= \lfloor 10000 * (76300.0041151\ldots \% 1) \rfloor$$

$$= \lfloor 10000 * 0.0041151\ldots \rfloor = 41$$

3. 平方取中法

- 先通过求关键码的平方来扩大差别，再取其中的几位或其组合作为散列地址
- 例如，
  - 一组二进制关键码：（00000100，00000110，000001010，000001001，000000111）
  - 平方结果为：（00010000，00100100，01100010，01010001，00110001）
  - 若表长为4个二进制位，则可取中间四位作为散列地址：（0100，1001，1000，0100，1100）

4. 数字分析法

- 设有 n 个 d 位数，每一位可能有 r 种不同的符号
- 这 r 种不同的符号在各位上出现的频率不一定相同
  - 可能在某些位上分布均匀些，出现几率均等
  - 在某些位上分布不均，只有某几种符号经常出现
- 可根据散列表的大小，选取其中各种符号**均匀分布**的若干位作为散列地址

数字分析法（续1）

- 计算各位数字中符号分布的均匀度 λ_k 的公式

$$\lambda_k = \sum_{i=1}^{r} (\alpha_i^k - n/r)^2$$

  - 其中，α_i^k 表示第 i 个符号在第 k 位上出现的次数
  - n/r 表示各种符号在 n 个数中均匀出现的期望值

- **λ_k 值越小，第 k 位符号分布越均匀**

数字分析法（续2）

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

①位， $\lambda_1 = 57.60$
②位， $\lambda_2 = 57.60$
③位， $\lambda_3 = 17.60$
④位， $\lambda_4 = 5.60$
⑤位， $\lambda_5 = 5.60$
⑥位， $\lambda_6 = 5.60$

- 若散列表地址范围有 3 位数字, 取各关键码的④⑤⑥位做为记录的散列地址

数字分析法（续3）

- 数字分析法仅适用于事先明确知道表中所有关键码每一位数值的分布情况
  - 它完全依赖于关键码集合
- 如果换一个关键码集合，选择哪几位数据要重新决定

5. 基数转换法

- 把关键码看成是另一进制上的数后
- 再把它转换成原来进制上的数
- 取其中若干位作为散列地址

- 一般取大于原来基数的数作为转换的基数，并且两个基数要互素

例：基数转换法

- 例如，给定一个十进制数的关键码是$(210485)_{10}$，把它看成以13为基数的十三进制数$(210485)_{13}$，再把它转换为十进制

$$(210485)_{13} = 2\times13^5 + 1\times13^4 + 4\times13^2 + 8\times13 + 5$$

$$= (771932)_{10}$$

- 假设散列表长度是10000，则可取低4位1932作为散列地址

6. 折叠法

- 基本思想
  - 将关键码分割成位数相同的几部分
    - ✓ 最后一部分的位数可以不同
  - 然后取这几部分的叠加和（舍去进位）作为散列地址

两种折叠方法

- 两种叠加方法
  - 移位叠加：把各部分的最后一位对齐相加
  - 分界叠加：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

例：折叠法

- 如果一本书的编号为04-4220-5864

(a)	(b)
5 8 6 4	5 8 6 4
4 2 2 0	0 2 2 4
+ 0 4	+ 0 4
[1] 0 0 8 8	6 0 9 2
h(key)=0088	h(key)=6092
(a)移位叠加	(b)分界叠加

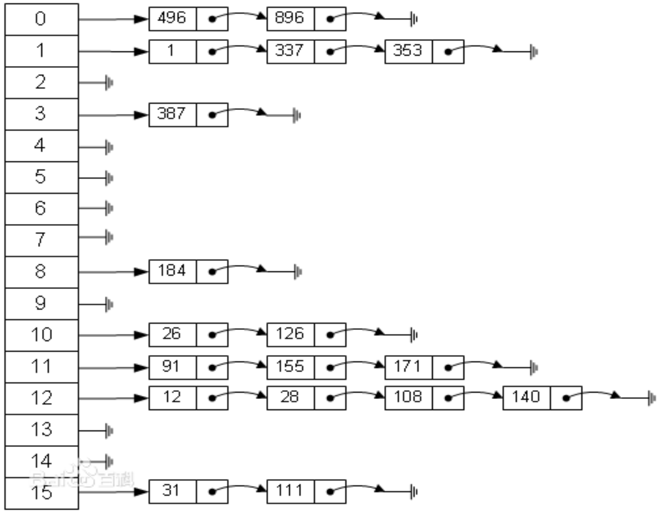
散列函数的应用

- 实际应用中，应根据关键码的特点，选用适当的散列函数
- 统计分析表明，平方取中法最接近于“随机化”
  - 若关键码不是整数而是字符串时，可以把每个字符串转换成整数，再应用平方取中法

II. 冲突的解决方法

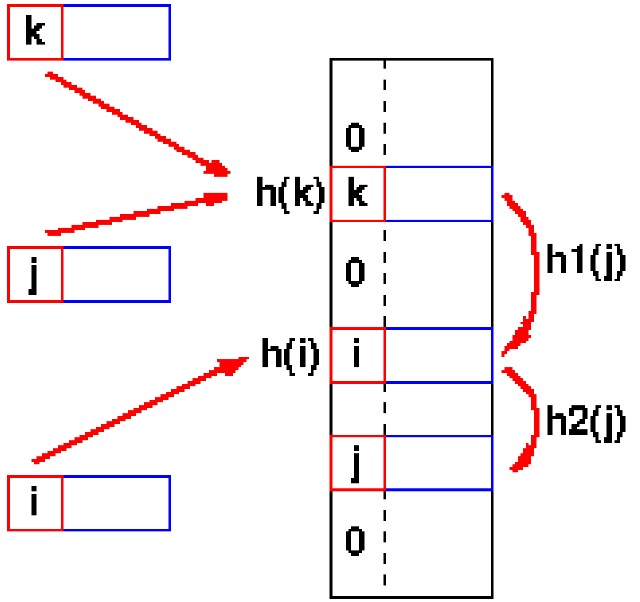
一．开散列方法（也称拉链法）

- 所有同义词链接在同一链表
- α可以大于1，但一般取$\alpha \le 1$



二．闭散列方法（也称开地址法）

- 把发生冲突的关键码存储在散列表中另一个空地址内



一、开散列方法

- 两类方法

  A.　拉链方法（**适用于内存**）

  B.　桶式散列（**适用于外存**）

A. 拉链法

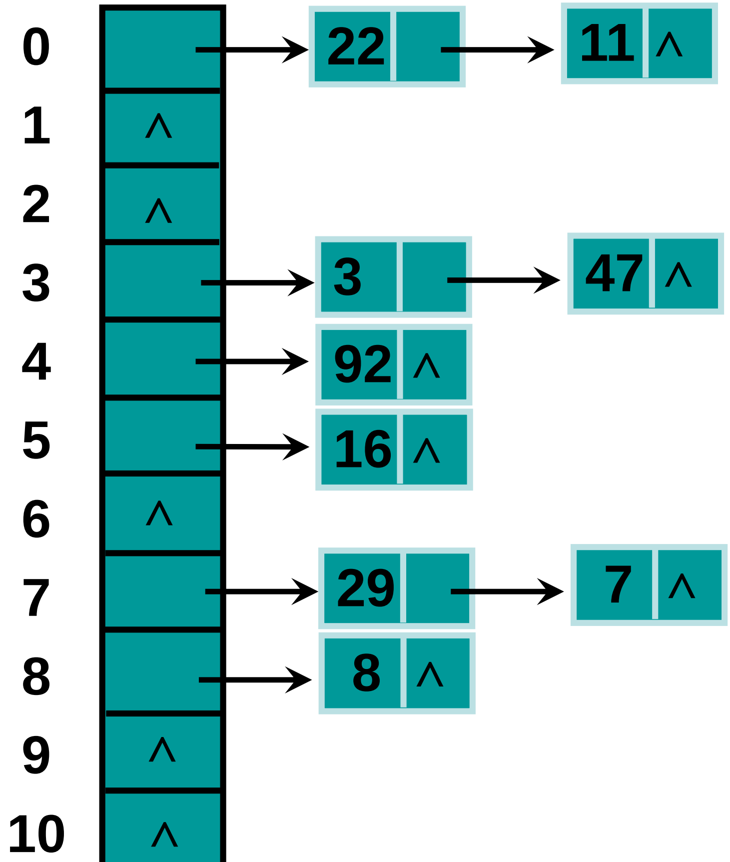
- 把散列表中的每个槽定义为一个链表的表头，散列到特定槽的所有记录都放到这个槽的链表中

例子：关键码集合：

{47, 7, 29, 11, 16, 92, 22, 8, 3}

散列函数：

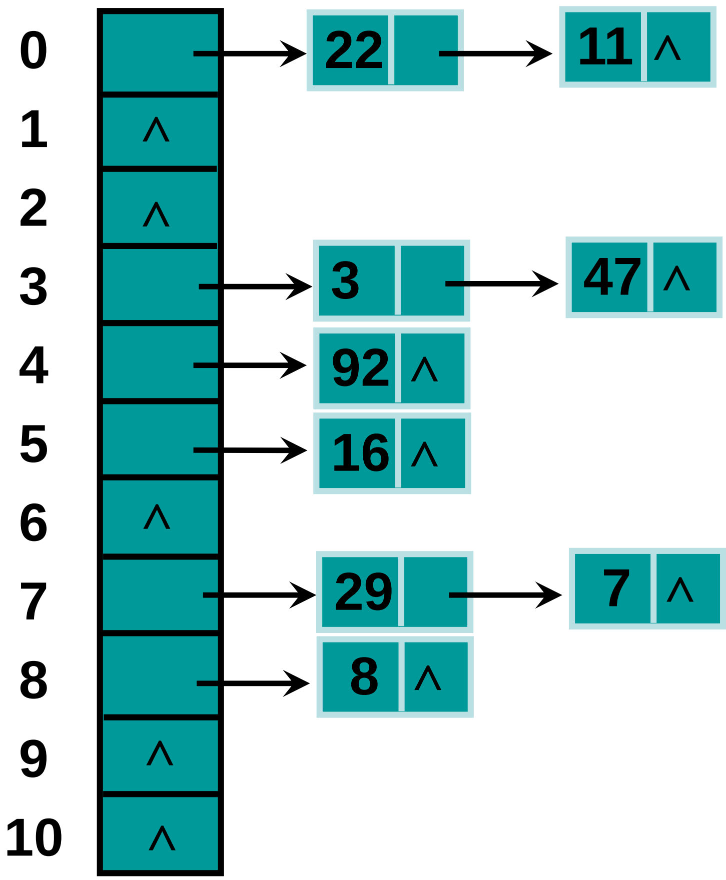
$H(key)=key \bmod 11$



同义词表的组织

- 组织方式
  - 根据输入顺序
  - 根据访问频率的顺序
  - 根据值的顺序
    - ✓适合检索不成功的情况：一旦遇到一个比待检索的关键码大的值，就停止检索。
    - ✓如果记录没排序或者根据访问频率排序，那么一次不成功的检索就需要访问同义词表中的每一个记录。

查找长度分析



- **成功**

$$ASL_{succ} = \frac{1*6+2*3}{9} = \frac{12}{9} = \frac{4}{3}$$

- **失败**

$$ASL_{unsucc} = \frac{1}{11}(3+1+1+3+2+2+1+$$

$$+3+2+1+1) = \frac{20}{11}$$

拉链法的优点

- 处理冲突简单，不同基地址冲突彼此独立，平均查找长度短
- 链表结点动态申请，适合于表长不确定情况
- 拉链法中可取α ≥1，且结点较大时，拉链法中增加的指针域可忽略不计，故节省空间
  - 闭散列为减少冲突要求α较小，故当结点规模较大时会浪费空间
- 用拉链法构造的散列表，删除结点易于实现
  - 只要简单地删去链表上相应的结点即可
  - 闭散列远没有如此简单！

缺点

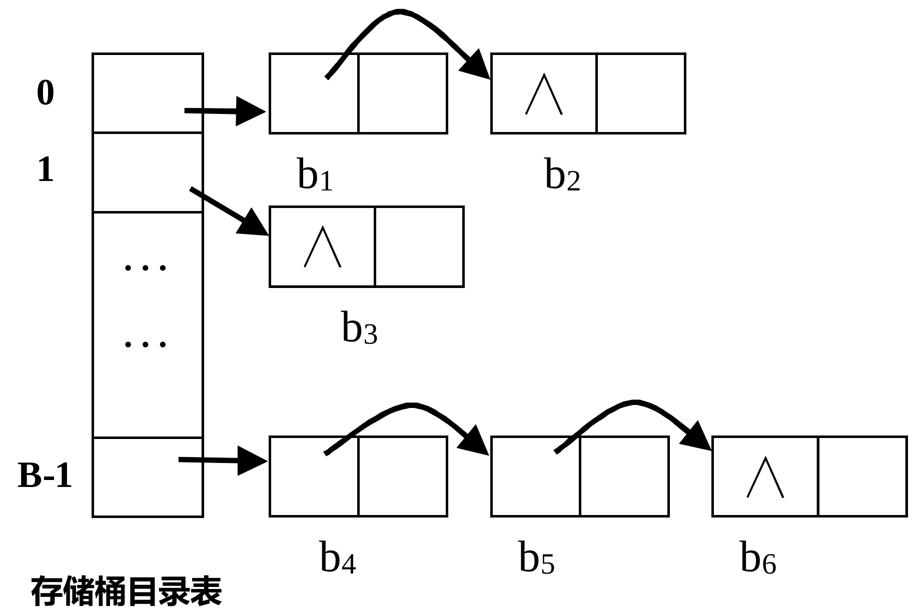
- **如果整个散列表元素存储于内存，拉链法容易实现**
- **如果散列表元素存储在磁盘，用拉链法则不太适用**
  - 同义词表中的元素可能存储在不同的磁盘页中
  - 这就会导致在检索一个特定关键码值时引起多次磁盘访问，从而增加了检索时间
- 引入桶式散列

B. 桶式散列

- **适合存储于磁盘的散列表**
- 基本思想
  - 散列文件记录分为若干桶，每个桶包含若干页块
  - 桶内各页块用指针链接，每个页块包含若干记录
  - 散列函数h(K)表示具有关键码K的记录所在桶号

桶式散列文件组织示例

- 右图表示一个具有B个桶的散列文件
  - 如果B很小，存储桶目录表可放在内存
  - 如果B存放多个页块，则桶目录表存到外存



桶式散列的访问

- **检索访问**
  - 计算H(*i*)的值，然后调桶目录表中包含第*i*个桶目录的页块进入内存，查到第*i*个存储桶的第一个页块的地址，然后根据该地址调入相应页块
- **磁盘访问性能**
  - 调存储桶目录表进入内存（设不在内存）需进行一次访外
  - 逐个检查桶内各页块，则平均访外次数为桶内页块数一半
  - 对于修改、插入等其他运算尚需另1次访外写外存。

二、闭散列方法

- d_0=h(K)称为K的<u>基地址</u>
- 当冲突发生时，使用某种方法为关键码K生成一个候选的散列地址序列，称为<u>探查序列</u>

  d_1，d_2，... d_i，...d_{M-1}

- $d_i = d_0 + p(K，i)$ $(0<i<M)$是后继散列地址，p(K，i)是<u>探查函数</u>

解决冲突的基本思想

- 插入K时，若基地址结点已被占用
  - 则按探查函数生成的探查序列依次查找，将找到的第一个空闲位置d_i作为K的存储位置
- 若所有后继散列地址都不空闲，说明该闭散列表已满，报告溢出

检索过程

- 检索要遵循插入时同样的探查序列
  - 重复冲突解决过程
  - 找出在基位置没有找到的记录
- 插入和检索函数都假定每个关键码的探查序列中<u>至少有一个存储位置是空的</u>
  - 否则可能会进入一个无限循环中
  - 也可以限制探查序列长度

探查方法

1. 线性探查法
2. 二次探查法
3. 伪随机数序列探查法
4. 双散列探查法

1. 线性探查

- 基本思想
  - 如果记录的基位置存储位置被占用，那么就在表中下移，直到找到一个空存储位置
    - ✓探查序列：d+1, d+2, ……, M-1, 0, 1, ……, d-1
  - 用于简单线性探查的探查函数是：<u>p(K，i) = i</u>
- 优点
  - 表中所有的存储位置都可作为插入记录的候选

产生的问题：聚集

- “聚集”（或称“堆积”）
  - 散列地址不同的记录，争夺同一后继散列地址
  - 小的聚集可能汇合成大的聚集
  - 导致很长的探查序列

散列表示例

- 已知一组关键码为（26，36，41，38，44，15，68，12，06，51，25），散列表长度M = 15，用线性探查法解决冲突构造这组关键码的散列表。
  - 利用除余法构造散列函数，选取小于M的最大质数P = 13，则散列函数为：h(K)= K %13。顺序插入各个结点：<u>26: h(26) = 0</u>；<u>36: h(36) = 10</u>；<u>41: h(41) = 2</u>；<u>38: h(38) = 12</u>；<u>44: h(44) = 5</u>
  - 发生冲突：15，68，12，6，51，25

15 68 6

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
26		41			44					36		38		
1	5	1	2	2	1	1				1		1	2	3

聚集示例

- 在理想情况下，表中的每个空槽都应该有相同的机会接收下一个要插入的记录。
  - 下一条记录放在第11个槽中的概率是2/13
  - 放到第7个槽中的概率是9/13

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
26	25	41	15	68	44	6				36		38	12	51
1	5	1	2	2	1	1				1		1	2	3

平均查找性能分析

- **成功查找的ASLsucc**

$$ASL_{succ} = \frac{1}{11}\sum_{i=1}^{11} C_i = \frac{1}{11}(1*6+2+2+2+3+5) = \frac{20}{11}$$

- **失败查找的ASLunsucc**

$$ASL_{unsucc} = \frac{8+7+6+5+4+3+2+1+1+1+2+1+11}{13}$$

$$= \frac{52}{13} = 4$$

改进线性探查

- 每次跳过常数c个而不是1个槽
  - 探查序列中的第i个槽是(h(K) + ic) mod M
  - 基位置相邻的记录就不会进入同一个探查序列了
- 探查函数是p(K，i) = i*c

新的问题

- 例如，设c = 2，要插入关键码k_1和k_2，$h(k_1) = 3$，$h(k_2) = 5$
- 探查序列
  - k_1的探查序列是3、5、7、9、...
  - k_2的探查序列就是5、7、9、...
- k_1和k_2的探查序列还是纠缠在一起，从而导致了聚集

2. 二次探查

- 探查增量序列依次为：1^2，-1^2，2^2，-2^2，…，即地址公式是

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

$$d_{2i} = (d - i^2) \% M$$

- 用于简单线性探查的探查函数是

$$p(K, 2i-1) = i*i$$

$$p(K, 2i) = - i*i$$

例：二次探查

- 例：使用一个大小M = 13的表
  - 假定对于关键码k_1和k_2，$h(k_1)=3$，$h(k_2)=2$
- 探查序列
  - k_1的探查序列是3、4、2、7 、...
  - k_2的探查序列是2、3、1、6 、...
- 尽管k_2会把k_1的基位置作为第2个选择来探查，但是这两个关键码的探查序列此后就立即分开了

3. 伪随机数序列探查

- 探查函数

$$p(K, i) = perm[i - 1]$$

  - 这里perm是一个长度为M - 1的数组
  - 值从“1~M - 1”的随机序列

例：伪随机数序列探查

- 考虑一个大小为M = 13的表，perm[0] = 2，perm[1] = 3，perm[2] = 7。
  - 假定两个关键码k_1和k_2，$h(k_1)=4$，$h(k_2)=2$
- 探查序列
  - k_1的探查序列是4、6、7、11 、...
  - k_2的探查序列是2、4、5、9 、...
- 尽管k_2会把k_1的基位置作为第2个选择来探查，但是这两个关键码的探查序列此后就立即分开了

二级聚集

- **基本聚集**
  - 基地址不同的关键码，其探查序列的某些段重叠在一起
  - 伪随机探查和二次探查**可以消除基本聚集**
- **二级聚集 (secondary clustering)**
  - 如果两个关键码散列到同一个基地址，还是得到同样的探查序列，所产生的聚集
  - 原因
    - ✓ 探查序列只是基地址的函数，与关键码值无关
    - ✓ 例子：伪随机探查和二次探查

4. 双散列探查法

- 避免二级聚集
  - 探查序列是原来关键码值的函数
  - 而不仅仅是基地址的函数
- 双散列探查法
  - 利用第二个散列函数作为常数
  - 每次跳过常数项，做线性探查

双散列探查法的基本思想

- 双散列探查法使用两个散列函数h_1和h_2
- 若在地址$h_1(key)=d$发生冲突，则计算$h_2(key)$，得到的探查序列为：

  $(d+h_2(key))$ % M，

  $(d+2h_2\ (key))$ %M ，

  $(d+3h_2\ (key))$ % M ，

  …

明确两个公式概念

- 双散列函数探查法序列公式：

$$d_i=(d + i* h_2 (key)) \ \% \ M$$

- 探查函数：

$$p(K, i) = i * h_2 (key)$$

双散列函数法特征

- h_2 (key) 必须与M互素
  - 使发生冲突的同义词地址均匀地分布在整个表中
  - 否则可能造成同义词地址的循环计算
- 双散列的优点：不易产生“聚集”
- 缺点：计算量增大

字典(Dictionary)

- 一种特殊的集合，其元素是(关键码，属性值)二元组
  - **关键码必须是互不相同的(在同一个字典之内)**
- 主要操作是依据关键码来存储和查找值
  - insert(key, value)
  - lookup(key)
- 用散列表方法高效实现

散列字典ADT

```
template <class Key， class Elem， class KEComp， class
    EEComp> class hashdict {
  private:
     Elem* HT;                    // 散列表
     int M;                       // 散列表大小
     int currcnt;                 // 现有元素数目
     Elem EMPTY;                  // 空槽
     int p(Key K， int i)         // 探查函数
     int h(int x) const ;         // 散列函数
```

散列字典ADT（续）

```
public:
    hashdict(int sz， Elem e)                        // 构造函数
    ~hashdict() { delete [] HT; }
    bool hashSearch(const Key&， Elem&) const;
    bool hashInsert(const Elem&);
    Elem hashDelete(const Key& K);
    int size() { return currcnt; }                  // 元素数目
};
```

1、散列表的插入

散列函数h，假设给定的值为K

- 若表中基地址空间未被占用，则插入记录
- 若表中基地址的值与K相等，则报告“已有此记录”
  - **不允许重复记录存在！**
- 否则，按设定的处理冲突方法查找探查序列的下一个地址，如此反复下去
  - 直到某个地址空间未被占用（可以插入）
  - 或者关键码比较相等（不需要插入）为止

插入算法代码

```
bool HashInsert(const Elem& e) {
    int home= h(getkey(e));                      //home存储基位置
    int i=0;
    int pos = home;                              //探查序列的初始位置
    while (!eq(EMPTY, HT[pos])) {
        if (eq(e, HT[pos])) return false;        //若插入值e存在
        i++;
        pos = (home+p(getkey(e), i)) % M;        //下一探查地址
    }
    HT[pos] = e;                                 // 插入元素e
    return true;
}
```

2、散列表的检索

假设散列函数h，给定的值为K

- 若基地址空间未被占用，则检索失败
- 否则将该地址中的值与K比较，若相等则检索成功
- 否则，按建表时设定的处理冲突方法查找探查序列的下一个地址，如此反复下去
  - 关键码比较相等，检索成功
  - 地址空间未被占用，检索失败

检索算法代码

```
bool HashSearch(const Key& K, Elem& e) const{
   int i=0, pos= home= h(K);                    // 初始位置
   while (! eq(EMPTY, HT[pos])) {
      if (eq(K, HT[pos])) {                     // 找到
            e = HT[pos];
            return true;
      }
      i++;
      pos = (home + p(K, i)) % M;               // 探查序列中的下一地址
   }
   return false;
}
```

3、散列表的删除

- 删除记录的时候，有两点需要重点考虑：
  - **(1) 删除记录不能影响后续检索**
  - **(2) 释放存储位置能为将来所用**
- 只有开散列方法可以真正删除，空间重用
- 闭散列方法都只能作标记，不能真正删除，空间未再次分配之前不可用

删除带来的问题

0	1	2	3	4	5	6	7	8	9	10	11	12
	K1	K2	K1		K2	K2	K2			K2		

- M = 13的散列表，假定关键码k1和k2，h(k1) = 2，h(k2) = 6
- 二次探查序列
  - k2的二次探查序列是6、7、5、10、2、2、10…
  - k1的二次探查序列是2、3、1、6、11、11、6…
- 删除位置6，用序列最后位置2的元素替换之，位置2设为空
- 影响k1的检索：k1的同义词将查不到
  - 可事实上它们还存放在位置3和1上!

墓碑

- 设置一特殊的标记位，来记录散列表中的单元状态
  - **单元被占用**、**空单元**、**已删除**
- 是否可以把**空单元**、**已删除**这两种状态，用统一的标记，以区别于“单元被占用”状态?
  - 不可以!
  - **检索时遇到已删除标记还需要继续检索下去**
  - 增加了平均检索长度
- 被删除标记值称为**墓碑**(**tombstone**)
  - 标志一个记录曾经占用这个槽，现在已经不再占用了

带墓碑的删除算法

```
Elem hashDelete(const Key& K){
    int i=0,  pos = home= h(K);                    //初始位置
     while (!eq(EMPTY, HT[pos]))  {
         if (eq(K, HT[pos])){
             temp = HT[pos];
             HT[pos] = TOMB;                        //设置墓碑
             Return temp;                           //返回目标
         }
         i++;
         pos = (home + p(K, i)) % M;
     }
    Return EMPTY;
}
```

带墓碑的插入操作

- 在插入时，如果遇到标志为墓碑的槽，可以把新记录存储在该槽中吗?
  - 避免插入两个相同的关键码
  - 检索过程仍然需要沿着探查序列下去，直到找到一个真正的空位置

带墓碑的插入操作改进

```
bool HashInsert(const Elem &e){
    int insplace, i=0, pos=home= h(getkey(e)); bool tomb_pos=false;
    while (!eq(EMPTY, HT[pos])) {
        if (eq(e, HT[pos])) return false;                //出现相同值元素！
        if (eq(TOMB, HT[pos]) && !tomb_pos) {
                insplace=pos;  tomb_pos=true;
        }                                                //记下第1个墓碑！
        pos = (home + p(getkey(e), ++ i)) % M;
    }
    if (!tomb_pos) insplace=pos;                         //没有墓碑
    HT[insplace]=e;    return true;
}
```

10.3.5 散列方法的效率分析

- 衡量标准：插入、删除和检索操作所需的ASL
- 散列表的插入和删除操作都是基于检索进行的
  - 删除：必须先找到该记录
  - 插入：必须找到探查序列的尾部，即对这条记录进行一次不成功的检索
    - ✓不考虑墓碑的情况，是尾部的空槽
    - ✓考虑墓碑的情况，也要找到尾部，才能确定是否有重复记录

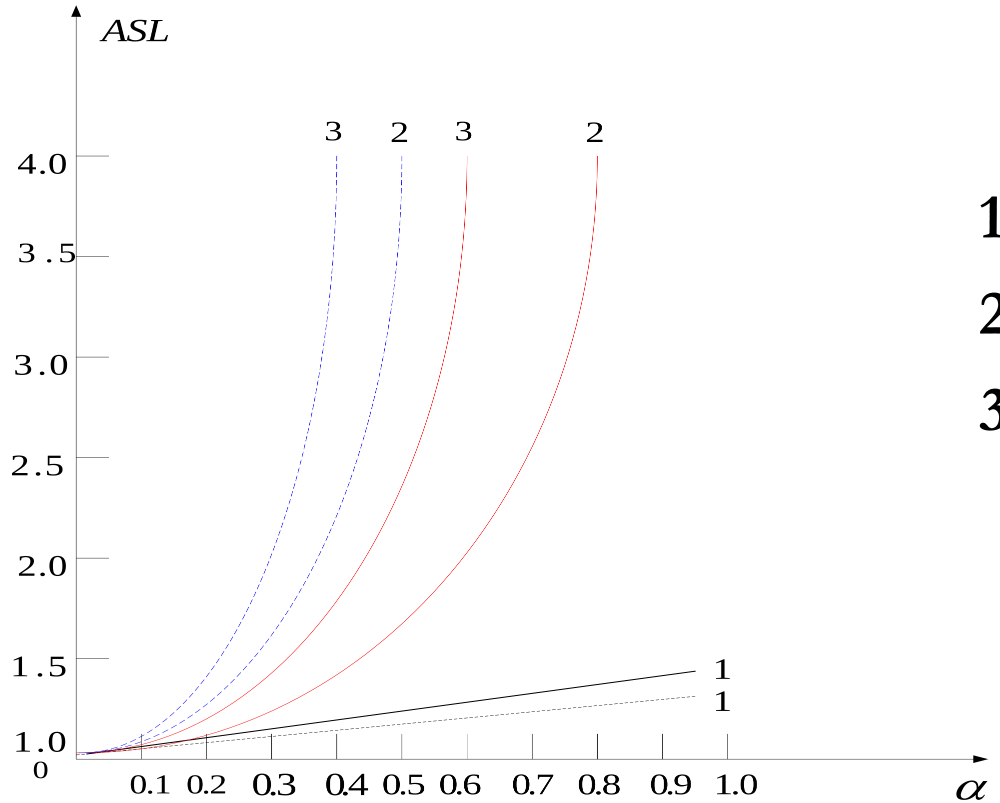
影响检索的效率的重要因素

- 散列效率与负载因子 $\alpha = N/M$ 有关
  - α 较小时，散列表比较空，所插入的记录比较容易插入到其空闲的基地址
  - α 较大时，插入记录很可能要靠冲突解决策略来寻找探查序列中合适的另一个槽
- 随着 α 增加，越来越多的记录有可能放到离其基地址更远的地方

散列表算法分析（表）

编号	冲突解决策略		平均检索长度	
			成功检索（删除）	不成功检索（插入）
1	开散列法		$1+\frac{\alpha}{2}$	$\alpha+e^{-\alpha}$
2	闭散列	双散列法	$\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$	$\frac{1}{1-\alpha}$
3	闭散列	线性探查法	$\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$	$\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$

散列表算法分析（图）



1. 开散列法
2. 双散列探查法
3. 线性探查法

- 图中是不同方法解决碰撞时散列表的平均检索长度。
- 红线是删除或成功检索的时间代价，蓝线是插入或不成功检索情况下的时间代价

开散列表与闭散列表的比较

	堆积现象	结构开销	插入/删除	查找效率	估计容量
开散列	无	有	效率高	效率高	不需要
闭散列	有	没有	效率低	效率低	需要

结论1

- 散列方法代价一般接近于访问一个记录的时间，效率高，比需要log n次记录访问的二分检索好得多
  - 不依赖于n，只依赖于负载因子$\alpha=n/M$
  - 随着α增加，预期的代价也会增加
  - $\alpha \le 0.5$时，大部分操作的分析预期代价都小于2
- 经验表明，负载因子的临界值是0.5（将近半满）
  - 大于这个临界值，性能就会急剧下降

结论2

- 散列表的插入和删除操作如果很频繁，将降低散列表的检索效率
  - 大量的插入操作，将使得负载因子增加
    - ✓从而增加了同义词子表的长度
    - ✓也就是增加了平均检索长度
  - 大量的删除操作，也将增加墓碑的数量
    - ✓这将增加记录本身到其基地址的平均长度

结论3

- 实际应用中，对于插入和删除操作比较频繁的散列表，可以定期对表进行重新散列
  - 把所有记录重新插入到一个新的表中
    - ✓清除墓碑
    - ✓把最频繁访问的记录放到其基地址

再见…

联系信息：

电子邮件：gjsong@pku.edu.cn

电　　话： 62754785

办公地点：理科2号楼2307室