

数据结构与算法

第九章 检索

任课教员：张 铭

<http://db.pku.edu.cn/mzhang/DS/>

mzhang@db.pku.edu.cn

北京大学信息科学与技术学院

网络与信息系统研究所

©版权所有，转载或翻印必究



第九章 检索

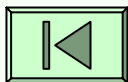
◆ 基本概念

◆ 9.1 线性表的检索

◆ 9.2 集合的检索

◆ 9.3 散列表的检索

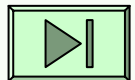
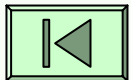
◆ 9.4 总结





基本概念

- 检索：在一组记录集合中找到关键码值等于给定值的某个记录，或者找到关键码值符合特定条件的某些记录的过程
- 检索的效率非常重要
 - 尤其对于大数据量
 - 需要对数据进行特殊的存储处理





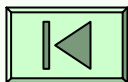
基本概念（续）

■ 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

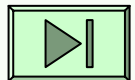
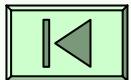
- 检索时充分利用辅助索引信息
- 牺牲一定的空间
- 从而提高检索效率





基本概念（续）

- 散列技术
 - 把数据组织到一个表中
 - 根据关键码的值来确定表中每个记录的位置
 - 缺点：
 - 不适合进行范围查询
 - 一般也不允许出现重复关键码
- 当散列方法不适合于基于磁盘的应用程序时，我们可以选择**B**树方法





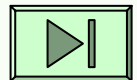
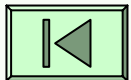
平均检索长度 (ASL)

- 关键码的比较：检索运算的主要操作
- 平均检索长度(Average Search Length)
 - 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

$$A S L = \sum_{i=1}^n P_i C_i$$

P_i 为检索第*i*个元素的概率

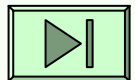
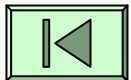
C_i 为找到第*i*个元素所需的关键码值与给定值的比较次数





平均检索长度的例子

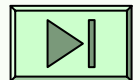
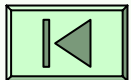
- 假设线性表为 (a, b, c) 检索 a 、 b 、 c 的概率分别为 0.4 、 0.1 、 0.5
 - 顺序检索算法的平均检索长度为
$$0.4 \times 1 + 0.1 \times 2 + 0.5 \times 3 = 2.1$$
 - 即平均需要 2.1 次给定值与表中关键码值的比较才能找到待查元素





检索算法评估的几个问题

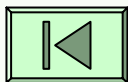
- 衡量一个检索算法还需要考虑
 - 算法所需的存储量
 - 算法的复杂性
 - ...





9.1 基于线性表的检索

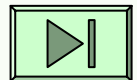
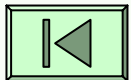
- ◆ 9.1.1 顺序检索
- ◆ 9.1.2 二分检索
- ◆ 9.1.3 分块检索





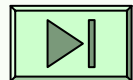
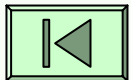
9.1.1 顺序检索

- 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。
 - 若某个记录的关键码和给定值比较相等，则检索成功；
 - 否则检索失败(找遍了仍找不到)。
- 存储：可以顺序、链接
- 排序要求：无



顺序检索算法

```
template <class Type> class Item {  
    public:  
        Item(Type value):key(value) {}  
        Type getKey() {return key;} //取关键码值;  
        void setKey(Type k){ key=k;} //置关键码  
    private:  
        Type key;                //关键码域  
        string other;            //其它域  
};  
vector<Item<Type>*> dataList;
```

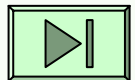
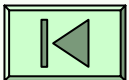




“监视哨”顺序检索算法

- 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 length, Type k) {  
    int i=length;  
    //将第0个元素设为待检索值  
    dataList[0]->setKey (k); //设监视哨  
    while(dataList[i]->getKey()!=k) i--;  
    return i;                //返回元素位置  
}
```

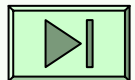
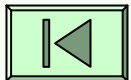


顺序检索性能分析

- 检索成功
假设检索每个关键码是等概率的, $P_i = 1/n$

$$\begin{aligned}\sum_{i=0}^{n-1} P_i \cdot (n-i) &= \frac{1}{n} \sum_{i=0}^{n-1} (n-i) \\ &= \sum_{i=1}^n i = \frac{n+1}{2}\end{aligned}$$

- 检索失败
假设检索失败时都需要比较 $n+1$ 次（设置了一个监视哨）



顺序检索平均检索长度

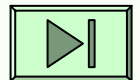
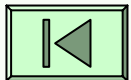
- 假设检索成功的概率为 p ，检索失败的概率为 $q=(1-p)$

$$ASL = p \cdot \frac{n+1}{2} + q \cdot (n+1)$$

$$= p \cdot \frac{n+1}{2} + (1-p)(n+1)$$

$$= (n+1)(1-p/2)$$

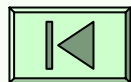
- $(n+1)/2 < ASL < (n+1)$

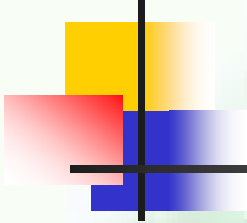




顺序检索优缺点

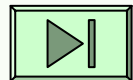
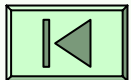
- 优点：插入元素可以直接加在表尾 $\Theta(1)$
- 缺点：检索时间太长 $\Theta(n)$





9.1.2 二分检索法

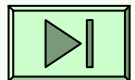
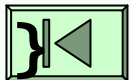
- 将任一元素 **dataList[i].Key** 与给定值 **K** 比较
 - 三种情况：
 - (1) **Key = K**, 检索成功, 返回 **dataList[i]**
 - (2) **Key > K**, 若有则一定排在 **dataList[i]** 前
 - (3) **Key < K**, 若右则一定排在 **dataList[i]** 后
- 缩小进一步检索的区间



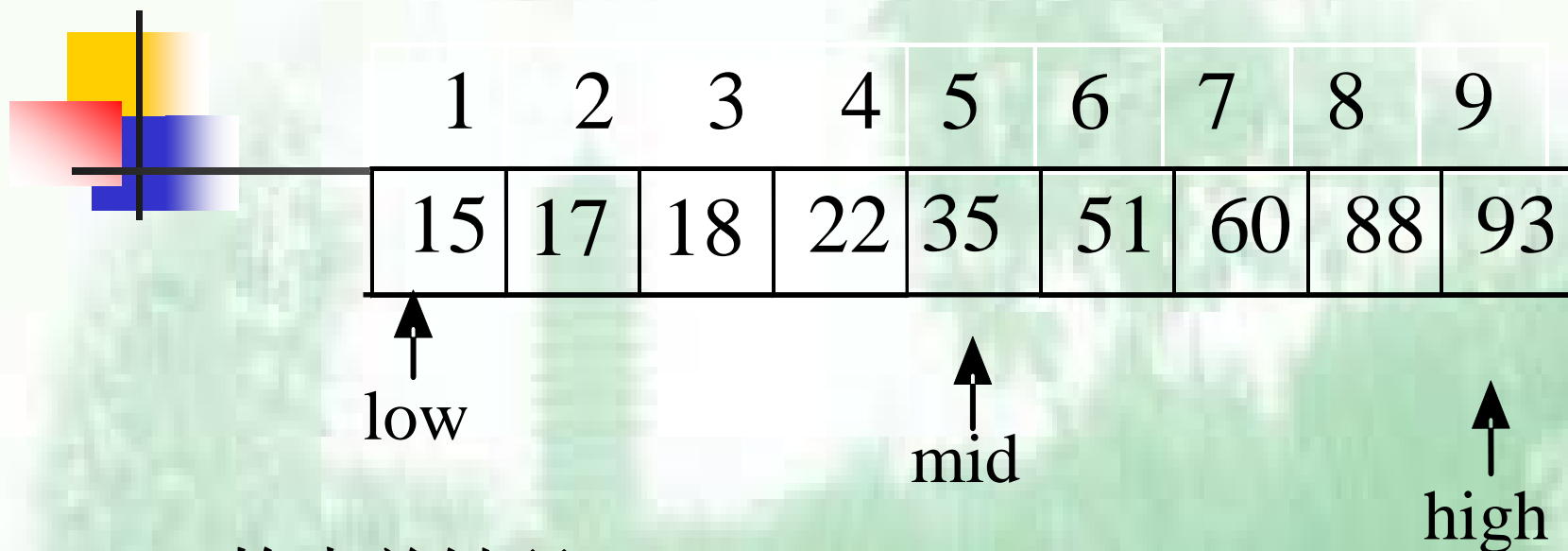


二分法检索算法

```
template <class Type> int BinSearch
(vector<Item<Type>*> & dataList,int length,Type k){
    int low=1, high=length, mid;
    while (low<=high) {
        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
}
```



二分法检索图示



检索关键码18 low=1 high=9 K=18

第一次: mid=5; array[5]=35>18

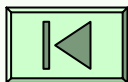
high=4; (low=1)

第二次: mid=2; array[2]=17<18

low=3; (high=4)

第三次: mid=3; array[3]=18=18

mid=3; return 8



二分法检索性能分析

- 最大检索长度为

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

- 失败的检索长度是

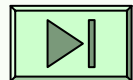
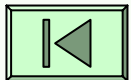
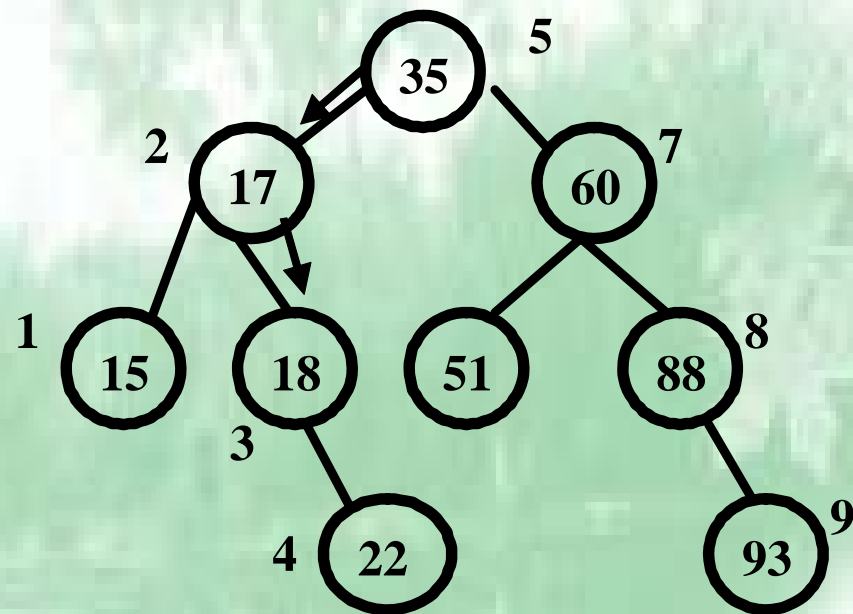
- 停止在内部叶结点

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

或

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

- 停止在外部空结点，则加1



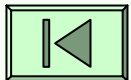
二分法检索性能分析（续）

成功的平均检索长度为：

$$\begin{aligned} ASL &= \frac{1}{n} \left(\sum_{i=1}^j i \cdot 2^{i-1} \right) \\ &= \frac{n+1}{n} \log_2 (n+1) - 1 \\ &\approx \log_2 (n+1) - 1 \quad (n > 50) \end{aligned}$$

■ 优缺点

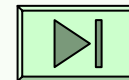
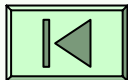
- 优点：平均与最大检索长度相近，检索速度快
- 缺点：要排序、顺序存储，不易更新（插/删）





9.1.3 分块检索

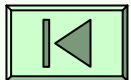
- 顺序与二分法的折衷
 - 既有较快的检索
 - 又有较灵活的更改



分块检索思想

■ “按块有序”

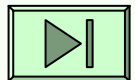
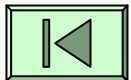
- 设线性表中共有 n 个数据元素，将表分成 b 块
 - 不需要均匀
 - 每一块可能不满
- 每一块中的关键码不一定有序
- 但前一块中的最大关键码必须小于后一块中的最小关键码





索引表

- 索引表
 - 各块中的最大关键码
 - 及各块起始位置
 - 可能还需要块中元素个数（每一块可能不满）
- 索引表是一个递增有序表
 - 由于表是分块有序的

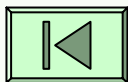




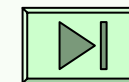
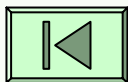
分块检索分两个阶段

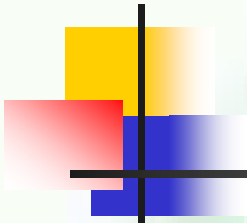
- (1) 确定待查元素所在的块
- (2) 在块内检索待查的元素

分块检索——索引顺序结构



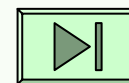
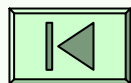
分块检索——索引顺序结构





分块检索性能分析

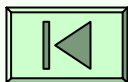
- 分块检索为两级检索
 - 先在索引表中确定待查元素所在的块;
 - 设在索引表中确定块号的时间开销是 ASL_b
 - 然后在块内检索待查的元素。
 - 在块中查找记录的时间开销为 ASL_w
- $ASL(n) = ASL_b + ASL_w$





分块检索性能分析(续1)

- 索引表是按块内最大关键码有序的，且长度也不大，可以二分检索，也可以顺序检索
- 各子表内各个记录不是按记录关键码有序，只能顺序检索



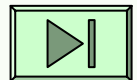
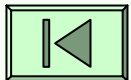
分块检索性能分析(续2)

- 假设在索引表中用顺序检索，在块内也用顺序检索

$$ASL_b = \frac{b+1}{2} \quad ASL_w = \frac{s+1}{2}$$

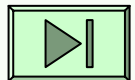
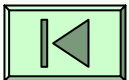
$$\begin{aligned} ASL &= \frac{b+1}{2} + \frac{s+1}{2} = \frac{b+s}{2} + 1 \\ &= \frac{n+s^2}{2s} + 1 \end{aligned}$$

- 当 $s = \sqrt{n}$ 时，ASL取最小值，
 $ASL = \sqrt{n} + 1 \approx \sqrt{n}$



分块检索性能分析(续3)

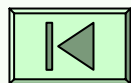
- 当 $n=10,000$ 时
 - 顺序检索5,000次
 - 二分法检索14次
 - 分块检索100次
- 如果数据块（子表）存放在外存时，还会受到页块大小的制约
 - 此时往往以外存一个 I / O 读取的数据（一页）作为一块



分块检索性能分析(续4)

- 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

$$\begin{aligned} ASL &= ASL_b + ASL_w \\ &\approx \log_2(b+1) - 1 + (s+1)/2 \\ &\approx \log_2(1+n/s) + s/2 \end{aligned}$$





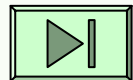
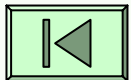
分块检索的优缺点

- 优点:

- 插入、删除相对较易
- 没有大量记录移动

- 缺点:

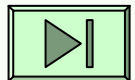
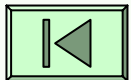
- 增加一个辅助数组的存储空间
- 初始线性表分块排序
- 当大量插入/删除时，或结点分布不均匀时，速度下降

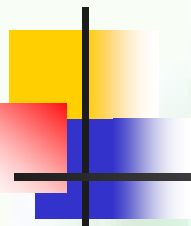


9.2 集合的检索

- 用位向量来表示集合
 - 对于密集型集合（数据范围小，而集合中有效元素个数比较多）

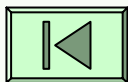
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	1	1	0	1	0	1	0	0	0	1	0	1	0	0

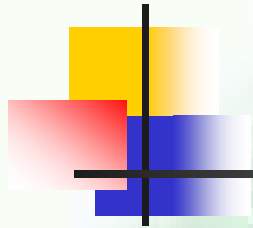




例：计算0到15之间所有的奇素数

奇数：	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
	&															
素数：	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	0	0	1	1	0	1	0	1	0	0	0	1	0	1	0	0
奇素数：	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	0	0	0	1	0	1	0	1	0	0	0	1	0	1	0	0



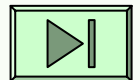
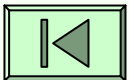


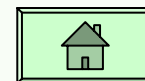
实际实现用无符合长整数数组

- 全集元素个数 **$N=40$** 的集合，集合 **$\{35, 9, 7, 5, 3, 1\}$** 表示为

0000 0000 0000 0000 0000 0000 0000 **1**000
0000 0000 0000 0000 0000 00**1**0 **1**0**1**0 **1**0**1**0

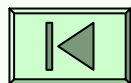
不够一个**unsigned long**，左补0





9.3 散列检索

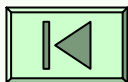
- ◆ 9.3.0 散列中的基本问题
- ◆ 9.3.1 散列函数
- ◆ 碰撞的处理
- ◆ 9.3.2 开散列方法
- ◆ 9.3.3 闭散列方法
- ◆ 9.3.4 闭散列表的算法实现
- ◆ 9.3.5 散列方法的效率分析



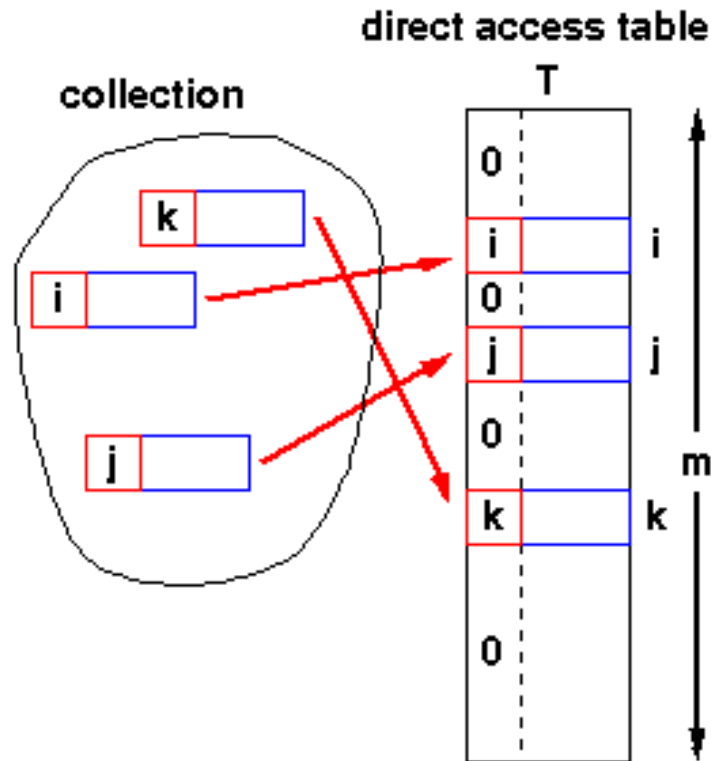


9.3.0 散列中的基本问题

- 基于关键码比较的检索
 - 顺序检索， $=$ ， \neq
 - 二分法、树型 $>$ ， $=$ ， $<$
- 检索是直接面向用户的操作
- 当问题规模 n 很大时，上述检索的时间效率可能使得用户无法忍受
- 最理想的情况
 - 根据关键码值，直接找到记录的存储地址
 - 不需要把待查关键码与候选记录集合的某些记录进行逐个比较



由数组的直接寻址想到的

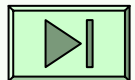
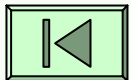


- 例如，读取指定下标的数组元素

- 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 $O(1)$
- 与数组元素个数的规模 n 无关

- 受此启发，计算机科学家发明了散列方法（**Hash**，有人称“哈希”，还有称“杂凑”）散列是一种重要的存储方法

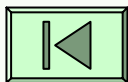
- 也是一种常见的检索方法





散列基本思想

- 一个确定的函数关系 h
- 以结点的关键码 K 为自变量
- 函数值 $h(K)$ 作为结点的存储地址
- 检索时也是根据这个函数计算其存储位置
 - 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





例子1

例9.1: 已知线性表关键码集合为:

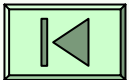
$S = \{ \text{and, begin, do, end, for, go, if, repeat, then, until, while} \}$

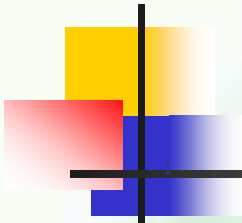
可设散列表为:

char HT2[26][8];

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

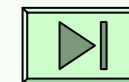
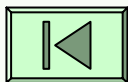
$H(\text{key}) = \text{key}[0] - 'a'$





散 列 地 址	关 键 码
0	a n d (a r r a y)
1	b e g i n
2	
3	d o
4	e n d (e l s e)
5	f o r
6	g o
7	
8	i f
9	
10	
11	
12	

散 列 地 址	关 键 码
13	
14	
15	
16	
17	r e p e a t
18	
19	t h e n
20	u n t i l
21	
22	w h i l e (w i t h)
23	
24	
25	



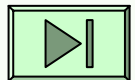
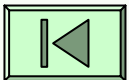


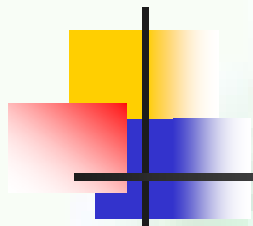
例子2

例9.2: 在集合S中增加4个关键码, 集合 $S1 = S + \{ \text{else, array, with, up} \}$

要修改散列函数: 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:

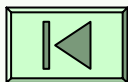
```
int H3(key)
char key[];
int i;
{ i = 0;
  while ((i<8) && (key[i]!='\0')) i++;
  return((key[0] + key(i-1) - 2*'a') / 2 )
}
```

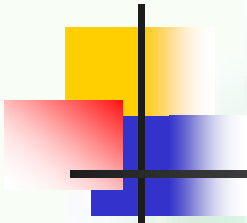




散列地址	关键码
0	
1	and
2	
3	end
4	else
5	
6	if
7	begin
8	do
9	
10	go
11	for
12	array

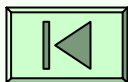
散列地址	关键码
13	while
14	with
15	until
16	then
17	
18	repeat
19	
20	
21	
22	
23	
24	
25	





几个重要概念

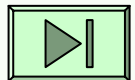
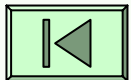
- **负载因子** $\alpha = n/m$
 - 散列表的空间大小为 m
 - 填入表中的结点数为 n
- **冲突**
 - 某个散列函数对于不相等的关键码计算出了相同的散列地址
 - 在实际应用中，不产生冲突的散列函数极少存在
- **同义词**
 - 发生冲突的两个关键码





散列技术的两个重要问题

- 采用散列技术时需要考虑的两个首要问题是：
 - (1) 如何构造(选择)使结点“分布均匀”的散列函数？
 - (2) 一旦发生冲突，用什么方法来解决？
- 还需考虑散列表本身的组织方法

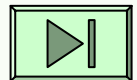
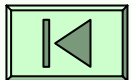




9.3.1 散列函数

- 散列函数：把关键码值映射到存储位置的函数，通常用h来表示

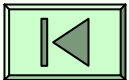
$$Address = Hash(key)$$





散列函数的选取原则

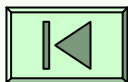
- 运算尽可能简单
- 函数的值域必须在表长的范围内
- 尽可能使得关键码不同时，其散列函数值亦不相同





需要考虑各种因素

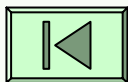
- 关键码长度
- 散列表大小
- 关键码分布情况
- 记录的检索频率
- ...





常用散列函数选取方法

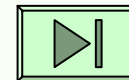
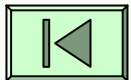
1. 除余法
2. 乘余取整法
3. 平方取中法
4. 数字分析法
5. 基数转换法
6. 折叠法
7. ELFhash字符串散列函数





1. 除余法

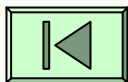
- **除余法**：用关键码**x**除以**M**(往往取散列表长度)，并取余数作为散列地址。散列函数为：
$$h(x) = x \bmod M$$
- 通常选择一个**质数**作为**M**值
 - 函数值依赖于自变量**x**的所有位，而不仅仅是最右边**k**个低位
 - 增大了均匀分布的可能性
 - 例如，**4093**





M为什么不用偶数

- 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
 - 如果偶数密钥比奇数密钥出现的概率大, 那么函数值就不能均匀分布
 - 反之亦然

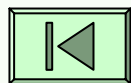


M不用幂

$x \bmod 2^8$ 选择最右边8位

0110010111000011010

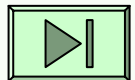
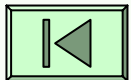
- 若把M设置为2的幂
 - 那么, $h(x) = x \bmod 2^k$ 仅仅是x(用二进制表示)最右边的k个位(bit)
- 若把M设置为10的幂
 - 那么, $h(x) = x \bmod 10^k$ 仅仅是x(用十进制表示)最右边的k个十进制位(digital)
- 缺点: 散列值不依赖于x的全部比特位





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



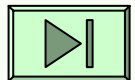
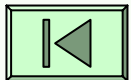
2. 乘余取整法

- 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
- 散列函数为:

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

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

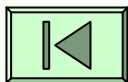
$$A * key \% 1 = A * key - \lfloor A * key \rfloor$$



乘余取整法示例

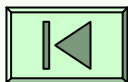
- 设关键码 **key = 123456**, **n = 10000**
且取 **A = $(\sqrt{5} - 1) / 2 = 0.6180339$** ,
- 因此有

$$\begin{aligned} \text{hash}(123456) &= \\ &= \lfloor 10000 * (0.6180339 * 123456 \% 1) \rfloor = \\ &= \lfloor 10000 * (76300.0041151... \% 1) \rfloor = \\ &= \lfloor 10000 * 0.0041151... \rfloor = 41 \end{aligned}$$



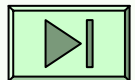
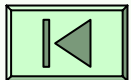
乘余取整法参数取值的考虑

- 若地址空间为 p 位，就取 $n=2^p$
 - 所求出的散列地址正好是计算出来的 $A * key \% 1 = A * key - \lfloor A * key \rfloor$ 值的小数点后最左 p 位 (bit) 值
 - 此方法的优点: 对 n 的选择无关紧要
- Knuth认为: A 可以取任何值, 与待排序的数据特征有关。一般情况下取黄金分割 $(\sqrt{5}-1)/2$ 最理想



3. 平方取中法

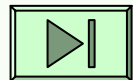
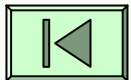
- 此时可采用平方取中法：先通过求关键码的平方来扩大差别，再取其中的几位或其组合作为散列地址
- 例如，
 - 一组二进制关键码：(00000100, 00000110, 000001010, 000001001, 000000111)
 - 平方结果为：(00010000, 00100100, 01100010, 01010001, 00110001)
 - 若表长为4个二进制位，则可取中间四位作为散列地址：(0100, 1001, 1000, 0100, 1100)





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，每种符号出现的几率均等
 - 在某些位上分布不均匀，只有某几种符号经常出现
- 可根据散列表的大小，选取其中各种符号分布均匀的若干位作为散列地址

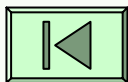


数字分析法（续1）

- 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。
- 计算出的 λ_k 值越小，表明在该位（第 k 位）各种符号分布得越均匀



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, $\lambda_2 = 57.60$

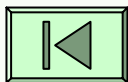
③位, $\lambda_3 = 17.60$

④位, $\lambda_4 = 5.60$

⑤位, $\lambda_5 = 5.60$

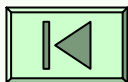
⑥位, $\lambda_6 = 5.60$

- 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
- 也可以把第①, ②, ③和第⑤位相加, 舍去进位, 变成一位数, 与第④, ⑥位合起来作为散列地址。还可以用其它方法



数字分析法（续3）

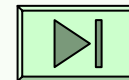
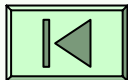
- ①位，仅9出现8次，
 - $\lambda_1 = (8-8/10)^2 \times 1 + (0-8/10)^2 \times 9 = 57.6$
- ②位，仅9出现8次，
 - $\lambda_2 = (8-8/10)^2 \times 1 + (0-8/10)^2 \times 9 = 57.6$
- ③位，0和2各出现两次，1出现4次
 - $\lambda_3 = (2-8/10)^2 \times 2 + (4-8/10)^2 \times 1 + (0-8/10)^2 \times 7 = 17.6$
- ④位，0和5各出现两次，1、2、6、8各出现1次
- ⑤位，0和4各出现两次，2、3、5、6各出现1次
- ⑥位，7和8各出现两次，0、1、5、9各出现1次
 - $\lambda_4 = \lambda_5 = \lambda_6 = (2-8/10)^2 \times 2 + (1-8/10)^2 \times 4 + (0-8/10)^2 \times 4 = 5.6$





数字分析法（续4）

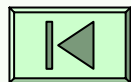
- 数字分析法仅适用于事先明确知道表中所有关键码每一位数值的分布情况
 - 它完全依赖于关键码集合
- 如果换一个关键码集合，选择哪几位数据要重新决定





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



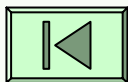


例：基数转换法

- 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

$$(210485)_{13} = 2 \times 13^5 + 1 \times 13^4 + 4 \times 13^2 + 8 \times 13 + 5 \\ = (771932)_{10}$$

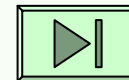
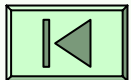
- 假设散列表长度是10000，则可取低4位1932作为散列地址





6. 折叠法

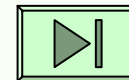
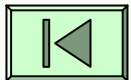
- 关键码所含的位数很多，采用平方取中法计算太复杂
- 折叠法
 - 将关键码分割成位数相同的几部分（最后一部分的位数可以不同）
 - 然后取这几部分的叠加和（舍去进位）作为散列地址





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 — 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



例：折叠法

■ [例9.6] 如果一本书的编号为04-42-20586-4

■ 5 8 6 4

■ 4 2 2 0

■ + 0 4

■ [1] 0 0 8 8

■ $h(\text{key}) = 0088$

■ (a) 移位叠加

5 8 6 4

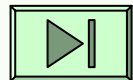
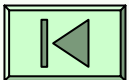
0 2 2 4

+ 0 4

6 0 9 2

$h(\text{key}) = 6092$

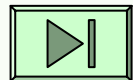
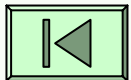
(b) 分界叠加



7. ELFhash字符串散列函数

- 用于UNIX系统V4.0“可执行链接格式”(Executable and Linking Format, 即ELF)

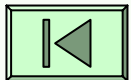
```
int ELFhash(char* key) {  
    unsigned long h = 0;  
    while(*key) {  
        h = (h << 4) + *key++;  
        unsigned long g = h & 0xF0000000L;  
        if (g) h ^= g >> 24;  
        h &= ~g;  
    }  
    return h % M;  
}
```





ELFhash函数特征

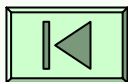
- 长字符串和短字符串都很有效
- 字符串中每个字符都有同样的作用
- 对于散列表中的位置不可能产生不均匀的分布





散列函数的应用

- 在实际应用中应根据关键码的特点，选用适当的散列函数
- 有人曾用“轮盘赌”的统计分析方法对它们进行了模拟分析，结论是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



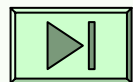
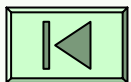
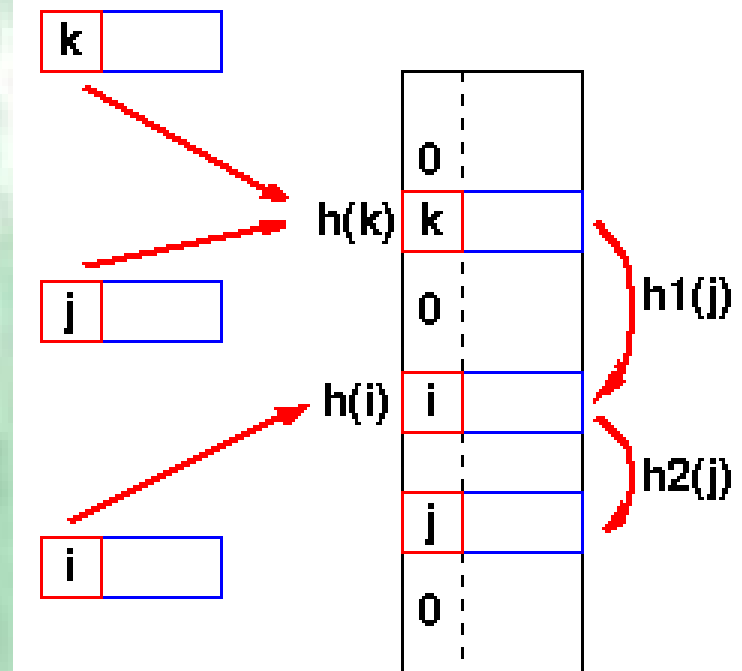
引子：碰撞的处理

开散列方法(open hashing, 也称为拉链法, separate chaining)

- 把发生冲突的关键码存储在散列表主表之外

闭散列方法(closed hashing, 也称为开地址方法, open addressing)

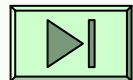
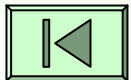
- 把发生冲突的关键码存储在表中另一个槽内



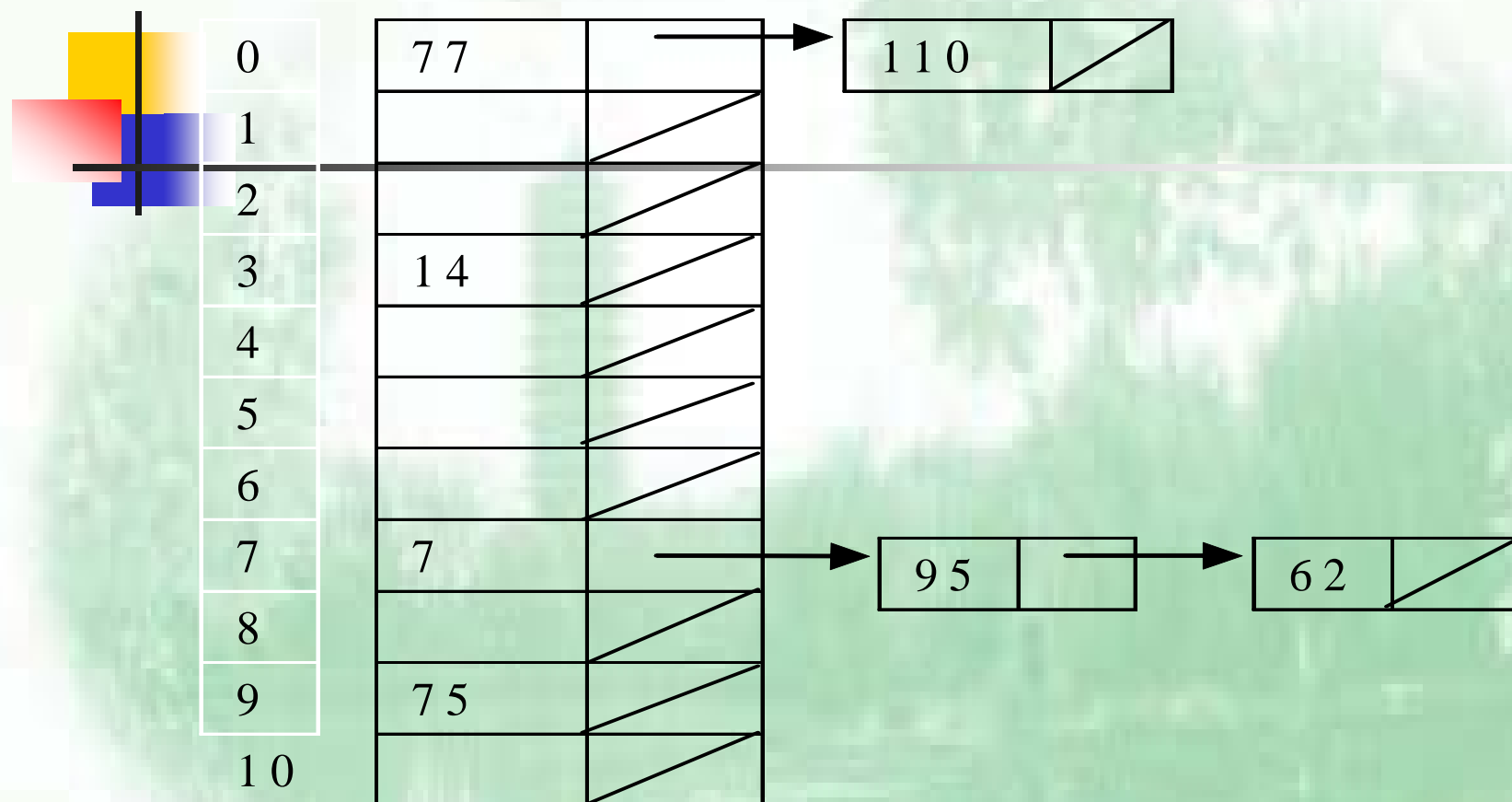
9.3.2 开散列方法

- 当碰撞发生时就拉出一条链，建立一个链表方式的同义词子表
 - 动态申请同义词的空间，适合于内存操作
 - 例：{77、7、110、95、14、75、62 }
$$h(\text{Key}) = \text{Key} \% 11$$

1. 拉链法
2. 桶式散列



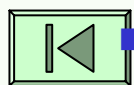
1. 拉链法



- 表中的空单元其实应该有特殊值标记出来，例如-1或INFINITY

- 或者使得散列表中的内容就是指针，空单元则内容为空指针。

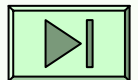
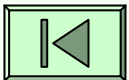
- 插入同义词时，可以对同义词链排序插入





拉链法性能分析

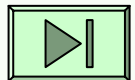
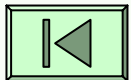
- 给定一个大小为M存储n个记录的表
 - 散列函数(在理想情况下)将把记录在表中M个位置平均放置, 使得平均每一个链表中有 n/M 个记录
 - $M > n$ 时, 散列方法的平均代价就是 $\Theta(1)$



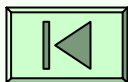
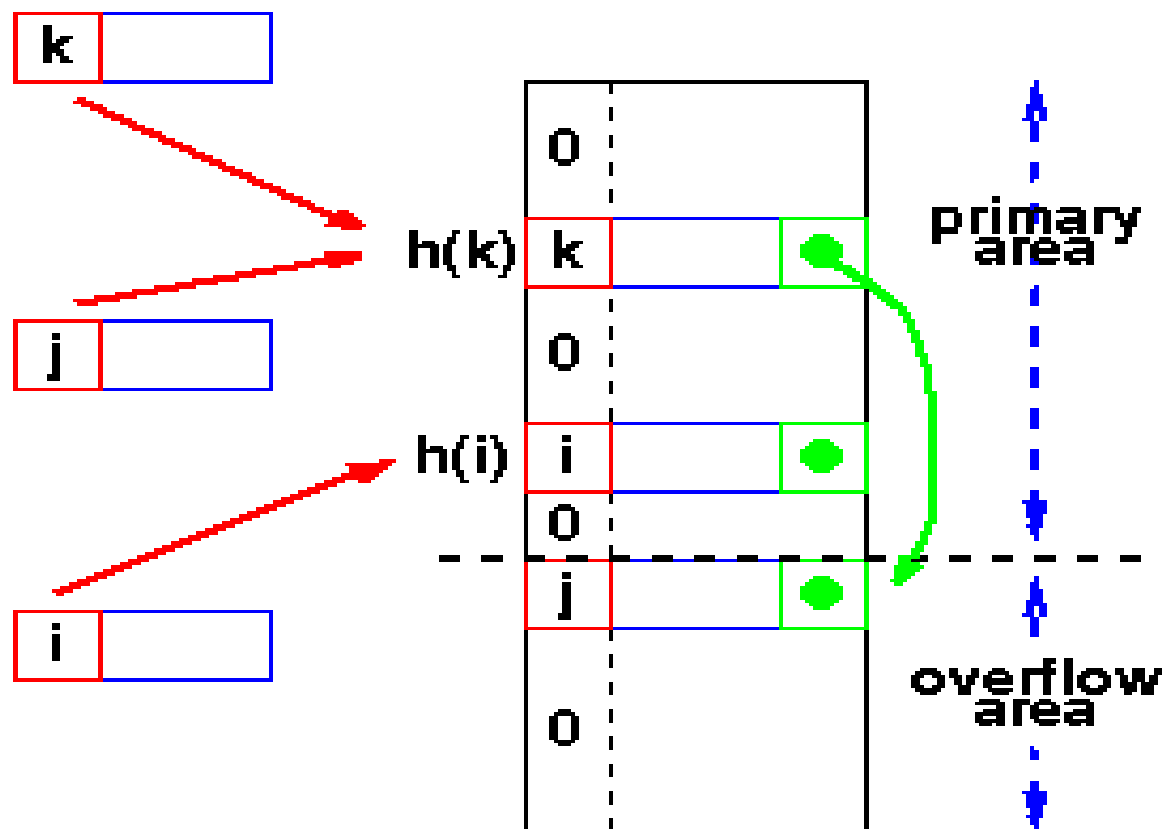


拉链法不适于外存检索

- 如果整个散列表存储在内存中，开散列方法比较容易实现
- 如果散列表存储在磁盘中，用开散列不太合适
 - 一个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
 - 桶式散列



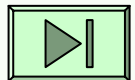
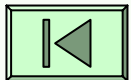
溢出区——静态链





2. 桶式散列

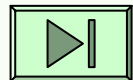
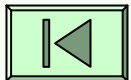
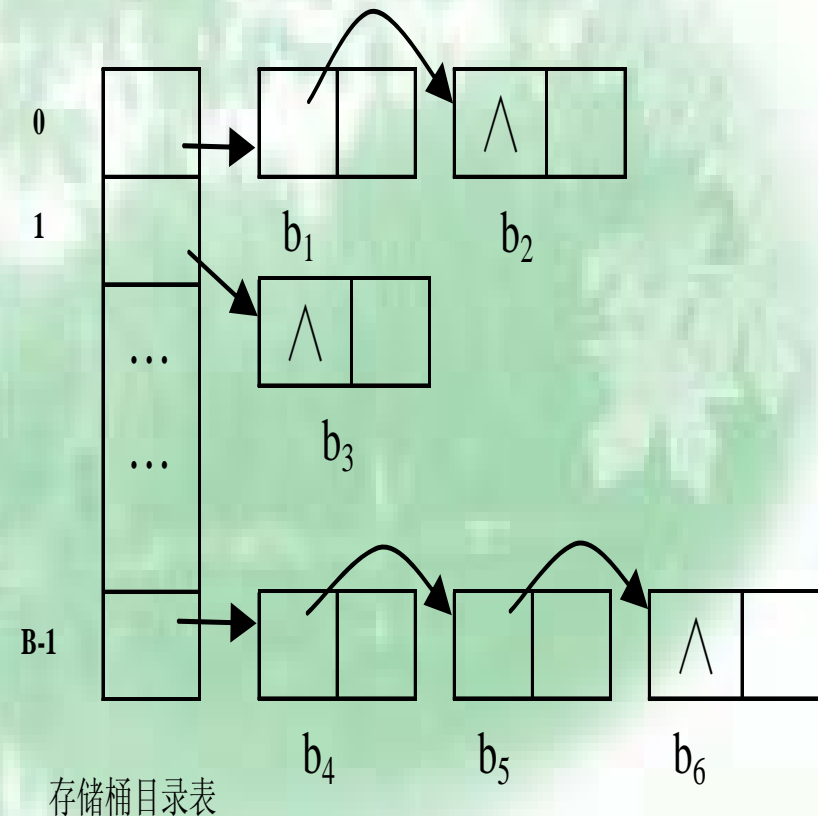
- 适合存储于磁盘的散列表
- 基本思想:
 - 把一个文件的记录分为若干存储桶，每个存储桶包含一个或多个页块
 - 一个存储桶内的各页块用指针连接起来，每个页块包含若干记录
 - 散列函数 $h(K)$ 表示具有关键码值 K 的记录所在的存储桶号



桶式散列文件组织

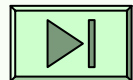
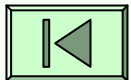
- 右图表示了一个具有**B**个存储桶的散列文件组织

- 如果**B**很小，存储桶目录表可放在内存
- 如果**B**较大，要存放好多页块，则存储桶目录表就放到外存上



桶式散列的磁盘访问性能分析

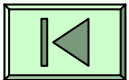
- 一个查找平均访外次数等于桶内页块数 k 的一半
 - 调存储桶目录表进入内存(假定目录表不在内存)
 - 为了寻找要求的记录必须逐个检查一个桶内各页块
 - 实际上是 $(k+1)/2$
- 对于修改、插入、删除等运算尚需另一次访外，用于重新写回外存





桶式散列的磁盘访问性能分析(续)

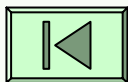
- 最理想状况:
 - 每个桶仅由一个页块组成, 假设目录在外存
 - 这样只需访外二次(对检索)或三次(对其他运算)
- 要求
 - 存储桶的个数大致等于记录存放所需的页块数
 - 散列函数值分布均匀





桶式散列面临的问题

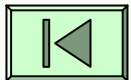
- 理想状况很难实现
 - 尤其当文件不断增长时，桶内的页块数也随之增多
 - 由于分布不均匀，有些桶内页块数可能过多，严重影响检索效率
- 必要时需对文件进行重新组织
 - 改变散列函数
 - 增加存储桶目录表的大小





9.3.3 闭散列方法

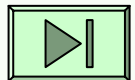
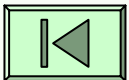
- 把所有记录直接存储在散列表中。
- 每个记录关键码有一个基位置即 $h(\text{key})$ ，即由散列函数计算出来的地址
- 如果要插入一个关键码，而另一个记录已经占据了R的基位置(发生碰撞)，
 - 那么就把R存储在表中的其它地址内，由冲突解决策略确定是哪个地址





闭散列表解决冲突的基本思想

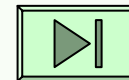
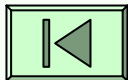
- 当冲突发生时，使用某种方法为关键码**K**生成一个散列地址序列
$$d_0, d_1, d_2, \dots d_i, \dots d_{m-1}$$
 - 其中 $d_0=h(K)$ 称为**K**的基地址地置
 - 所有 $d_i(0 < i < m)$ 是后继散列地址
- 形成探查的方法不同，所得到的解决冲突的方法也不同





闭散列表解决冲突的基本思想（续）

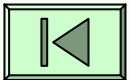
- 当插入**K**时，若基地址上的结点已被别的数据元素占用
 - 则按上述地址序列依次探查，将找到的第一个开放的空闲位置 d_i 作为**K**的存储位置
- 若所有后继散列地址都不空闲，说明该闭散列表已满，报告溢出





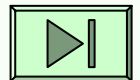
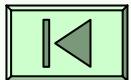
检索过程

- 检索时也要像插入时一样遵循同样的策略
 - 重复冲突解决过程
 - 找出在基位置没有找到的记录



探查序列

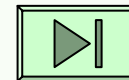
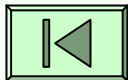
- 插入和检索函数都假定每个关键码的探查序列中至少有一个存储位置是空的
 - 否则可能会进入一个无限循环中
- 也可以限制探查序列长度





几种常见的闭散列方法

- ◆ 1. 线性探查
- ◆ 2. 二次探查
- ◆ 3. 伪随机数序列探查
- ◆ 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, 那么在表中下移, 直到找到一个空存储位置。

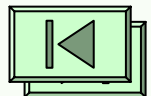
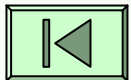
- 依次探查下述地址单元: $d+1, d+2, \dots, M-1, 0, 1, \dots, d-1$

- 用于简单线性探查的探查函数是:

$$p(K, i) = i$$

- 线性探查的优点

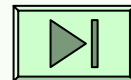
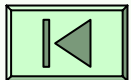
- 表中所有的存储位置都可以作为插入新记录的候选位置





可能产生的问题——聚集

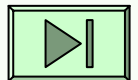
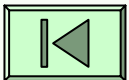
- “聚集”（**clustering**，或称为“堆积”）
 - 散列地址不同的结点，争夺同一后继散列地址
 - 小的聚集可能汇合成大的聚集
 - 导致很长的探查序列



散列表示例

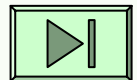
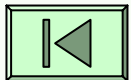
- $M = 15, h(\text{key}) = \text{key} \% 13$
- 在理想情况下，表中的每个空槽都应该有相同的机会接收下一个要插入的记录
 - 下一条记录放在第11个槽中的概率是 $2/15$
 - 放到第7个槽中的概率是 $11/15$

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



改进线性探查

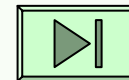
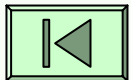
- 每次跳过常数 c 个而不是 1 个槽
 - 探查序列中的第 i 个槽是 $(h(K) + ic) \bmod M$
 - 基位置相邻的记录就不会进入同一个探查序列了
- 探查函数是 $p(K, i) = i * c$
 - 必须使常数 c 与 M 互素





例：改进线性探查

- 例如， $c = 2$ ，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

- 探查序列依次为： 1^2 , -1^2 , 2^2 , -2^2 , ..., 即探查地址是

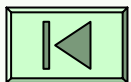
$$d_{2i-1} = (d + i^2) \% M$$

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

- 用于简单线性探查的探查函数是

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

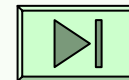
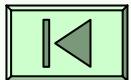
$$p(K, 2i) = -i * i$$





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



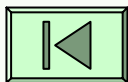


3. 伪随机数序列探查

- 探查函数

$$p(K, i) = \text{perm}[i - 1]$$

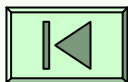
- 这里perm是一个长度为M - 1的数组
- 包含值从1到M - 1的随机序列





产生n个数的伪随机排列

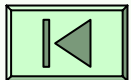
```
■ void permute(int *array, int n) {  
    for (int i = 1; i <= n; i ++)  
        swap(array[i-1], array[Random(i)]);  
}
```





例：伪随机数序列探查

- 例：考虑一个大小为 $M = 13$ 的表， $\text{perm}[0] = 2$ ， $\text{perm}[1] = 3$ ， $\text{perm}[2] = 7$ 。
 - 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





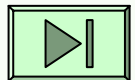
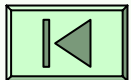
二级聚集

- 能消除基本聚集

- 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

- 二级聚集(**secondary clustering**)

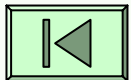
- 如果两个关键码散列到同一个基地址，还是得到同样的探查序列，所产生的聚集
- 原因探查序列只是基地址的函数，而不是原来关键码值的函数
- 例子：伪随机探查和二次探查





4. 双散列探查法

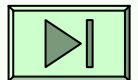
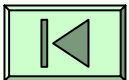
- 避免二级聚集
 - 探查序列是原来关键码值的函数
 - 而不仅仅是基位置的函数
- 双散列探查法
 - 利用第二个散列函数作为常数
 - 每次跳过常数项，做线性探查





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 $h_1(\text{key})=d$ 发生冲突，则再计算 $h_2(\text{key})$ ，得到的探查序列为：
 $(d+h_2(\text{key})) \% M$ ，
 $(d+2h_2(\text{key})) \% M$ ，
 $(d+3h_2(\text{key})) \% M$ ，
...





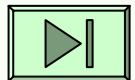
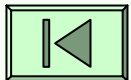
明确两个公式概念

- 双散列函数探查法序列公式:

$$d_i = (d + i h_2(\text{key})) \% M$$

- 双散列函数公式:

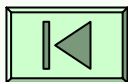
$$p(K, i) = i * h_2(\text{key})$$





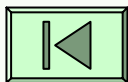
双散列函数法特征

- $h_2(\text{key})$ 尽量与M互素
 - 使发生冲突的同义词地址均匀地分布在表中
 - 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



M和h2(k)选择方法

- 方法1: 选择 M 为一个素数, h_2 返回的值在 $1 \leq h_2(K) \leq M-1$ 范围之内
- 方法2: 设置 $M = 2^m$, 让 h_2 返回一个1到 2^m 之间的奇数值
- 方法3: 若 M 是素数, $h_1(K) = K \bmod M$
 - $h_2(K) = K \bmod (M-2) + 1$
 - 或者 $h_2(K) = [K / M] \bmod (M-2) + 1$
- 方法4: 若 M 是任意数, $h_1(K) = K \bmod p$, (p 是小于 M 的最大素数)
 - $h_2(K) = K \bmod q + 1$ (q 是小于 p 的最大素数)

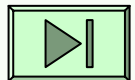
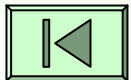




9.3.4 闭散列表的算法实现

字典(dictionary)

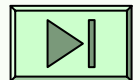
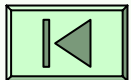
- 一种特殊的集合，其元素是(关键码，属性值)二元组。
 - 关键码必须是互不相同的(在同一个字典之内)
- 主要操作是依据关键码来存储和析取值
 - `insert(key, value)`
 - `lookup(key)`





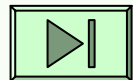
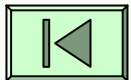
实现方式

- 实现方式
 - 有序线性表
 - 字符树
 - 散列方法



散列字典ADT（属性）

```
template <class Key, class Elem, class
    KEComp, class EECComp> 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
    int h(char* x) const ; // 字符串散列函数
```





散列字典ADT（方法）

public:

```
    hashdict(int sz, Elem e) { // 构造函数  
        M=sz; EMPTY=e;  
        currnt=0; HT=new Elem[sz];  
        for (int i=0; i<M; i++) HT[i]=EMPTY;  
    }
```

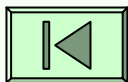
```
    ~hashdict() { delete [] HT; }
```

```
    bool hashSearch(const Key&, Elem&) const;
```

```
    bool hashInsert(const Elem&);
```

```
    Elem hashDelete(const Key& K);
```

```
    int size() { return currnt; } // 元素数目  
};
```

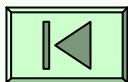




插入算法

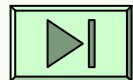
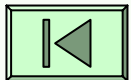
散列函数 h ，假设给定的值为 K

- 若表中该地址对应的空间未被占用，则把待插入记录填入该地址
- 如果该地址中的值与 K 相等，则报告“散列表中已有此记录”
- 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（程序）

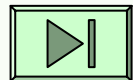
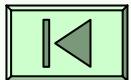
```
// 将数据元素e插入到散列表 HT
template <class Key, class Elem, class KEComp, class
    EEComp>
bool hashdict<Key, Elem, KEComp,
    EEComp>::hashInsert(const Elem& e) {
    int home= h(getkey(e)); //home存储基位置
    int i=0;
    int pos = home;      // 探查序列的初始位置
    while (!EEComp::eq(EMPTY, HT[pos])) {
        if (EEComp::eq(e, HT[pos])) return false;
        i++;
        pos = (home+p(getkey(e), i)) % M; //探查
    }
    HT[pos] = e;          // 插入元素e
    return true;
}
```





散列表的检索

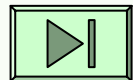
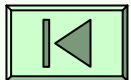
- 与插入过程类似
 - 采用的探查序列也相同



检索算法

假设散列函数 h ，给定的值为 K

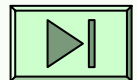
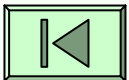
- 若表中该地址对应的空间未被占用，则检索失败
- 否则将该地址中的值与 K 比较，若相等则检索成功
- 否则，按建表时设定的处理冲突方法查找探查序列的下一个地址，如此反复下去
 - 关键码比较相等，检索成功
 - 地址空间未被占用，检索失败





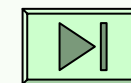
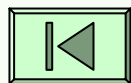
检索算法（程序）

```
template <class Key, class Elem, class KEComp, class
    EEComp>
bool hashdict<Key, Elem, KEComp, EEComp>::
hashSearch(const Key& K, Elem& e) const{
    int i=0, pos= home= h(K); // 初始位置
    while (!EEComp::eq(EMPTY, HT[pos])) {
        if (KEComp::eq(K, HT[pos])) { // 找到
            e = HT[pos];
            return true;
        }
        i++;
        pos = (home + p(K, i)) % M;
    }//while
    return false;
}
```



删除

- 删除记录的时候，有两点需要重点考虑：
 - (1) 删除一个记录一定不能影响后面的检索
 - (2) 释放的存储位置应该能够为将来插入使用
- 只有开散列方法（分离的同义词子表）可以真正删除
- 闭散列方法都只能作标记（墓碑），不能真正删除
 - 若真正删除了探查序列将断掉
 - 检索算法“直到某个地址空间未被占用（检索失败）”
 - 墓碑标记增加了平均检索长度

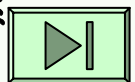
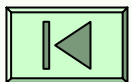


删除，断线索

- $M = 15$
- $h(\text{key}) = \text{key} \% 13$

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

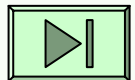
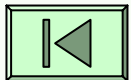
- 删除41，检索15
 - 关键码41和15的基址都是第2个槽，15被线性探查放到第3个
 - 如果从表中删除41，对15的检索必须仍然探查第2个槽，断线索





删除带来的问题？

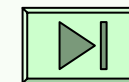
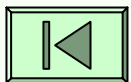
- 另一方面，删除后释放的槽应该能够为将来的插入使用
 - 不想让散列表中的位置由于删除而永远不可用



例：删除带来的问题

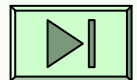
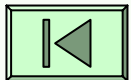
0	1	2	3	4	5	6	7	8	9	10	11	12
	K 1	K 2	K 1		K 2	K 2	K 2			K 2		

- 例如，一个长度 $M = 13$ 的散列表，假定关键码 k_1 和 k_2 ， $h(k_1) = 2$ ， $h(k_2) = 6$ 。
- 二次探查序列
 - k_1 的二次探查序列是 2、3、1、6、11、11、6、5、12、...
 - k_2 的二次探查序列是 6、7、5、10、2、2、10、9、3、... k_2 的同义词占用
- 删除位置 6，用该序列的最后位置 2 的元素替换之，位置 2 设为空
- 检索 k_1 的同义词，查不到
 - 可事实上它们还存放在位置 3 和 1 上！



墓碑

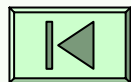
- 设置一个特殊的标记位，来记录散列表中的单元状态
 - 单元被占用
 - 空单元
 - 已删除
- 是否可以把空单元、已删除这两种状态，用特殊的值标记，以区别于“单元被占用”状态？
 - 不可以！
- 被删除标记值称为**墓碑 (tombstone)**
 - 标志一个记录曾经占用这个槽
 - 但是现在已经不再占用了





墓碑对操作的影响

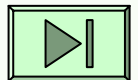
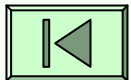
- 加入了删除操作后，闭散列的插入操作需要进行修改
- 检索操作可以不修改
 - 检索时，如果遇到墓碑，检索过程会顺着探查序列继续进行
 - 可以把墓碑看成是不等于任何关键码的特殊值，对于算法没有影响





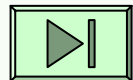
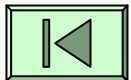
带墓碑的插入操作

- 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



带墓碑的删除算法

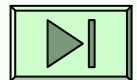
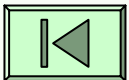
```
template <class Key, class Elem, class KEComp, class
EEComp> Elem
hashdict<Key,Elem,KEComp,EEComp>::hashDelete(const
Key& K)
{ int i=0, pos = home= h(K);    //初始位置
  while (!EEComp::eq(EMPTY, HT[pos])) {
    if (KEComp::eq(K, HT[pos])){
      temp = HT[pos];
      HT[pos] = TOMB; //设置墓碑
      return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





带墓碑的插入算法

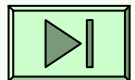
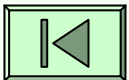
```
template <class Key, class Elem, class KEComp, class
    EEComp>
bool hashdict <Key, Elem, KEComp,
    EEComp>::hashInsert(const Elem &e) {
    int i=0, pos=home= h(getkey(e)); //初始
    while (!EEComp::eq(EMPTY, HT[pos])
        && !EEComp::eq(TOMB, HT[pos])) {
        if (KEComp::eq(getkey(e), HT[pos]))
            return false;    //不允许重复关键码
        i++;
        pos = (home + p(getkey(e), i)) % M;
    }
    HT[pos]=e;                //插入e
    return true;
}
```





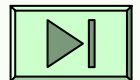
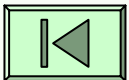
带墓碑插入中仍存在的问题

- 上述插入算法中，**for**循环在遇到第一个**TOMB**即“**HT[pos] = TOMB**”时就跳出，然后在墓碑处插入值
 - 但是难以确定位于墓碑后面的探查序列中是否有与待插入记录关键码相同的记录
- 为避免出现两个相同的关键码，插入过程仍然要沿着探查序列下去，直到找到一个真正的空位置
 - 如下的修改正是基于以上考虑，并将记录插入到第一个**TOMB**处（如果有的话），否则就插入到空位置



带墓碑的插入操作改进

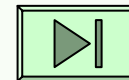
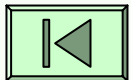
```
template <class Key, class Elem, class KEComp, class
EEComp> bool hashdict<Key, Elem, KEComp,
EEComp>::hashInsert(const Elem &e) {
    int insplace, i=0, pos=home= h(getkey(e)); bool
    tomb_pos=false;
    while (!EEComp::eq(EMPTY, HT[pos])) {
        if (EEComp::eq(e, HT[pos])) return false;
        if (EEComp::eq(TOMB, HT[pos]) && !tomb_pos)
            {insplace=pos; tomb_pos=true;} //第一
        pos = (home + p(getkey(e), ++ i)) % M;
    }
    if (!tomb_pos) insplace=pos; //没有墓碑
    HT[insplace]=e; return true;
}
```





9.3.5 散列方法的效率分析

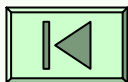
- 衡量标准：插入、删除和检索操作所需的记录访问次数
- 散列表的插入和删除操作都是基于检索进行的
 - 删除：必须先找到该记录
 - 插入：必须找到探查序列的尾部，即对这条记录进行一次不成功的检索
 - 对于不考虑删除的情况，是尾部的空槽
 - 对于考虑删除的情况，也要找到尾部，才能确定是否有重复记录





检索算法分析

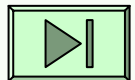
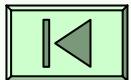
- 平均检索长度
 - 成功的检索
 - 不成功的检索





影响检索的效率的重要因素

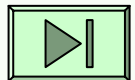
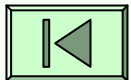
- 散列方法预期的代价与负载因子 $\alpha = N/M$ 有关
 - α 较小时，散列表比较空，所插入的记录比较容易插入到其空闲的基地址
 - α 较大时，插入记录很可能要靠冲突解决策略来寻找探查序列中合适的另一个槽
- 随着 α 增加，越来越多的记录有可能放到离其基地址更远的地方





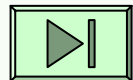
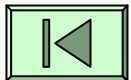
散列表算法分析

- 假定探查序列是散列表中槽的随机排列
- 下面的分析说明插入(或者一次不成功的检索)的预期代价与 α 的关系



散列表算法分析 (1)

- 基地址被占用的可能性是 α
- 基地址和探查序列中下一个槽都被占用的可能性是
$$\frac{N(N-1)}{M(M-1)}$$
- 发生第*i*次冲突的可能性是
$$\frac{N(N-1)\cdots(N-i+1)}{M(M-1)\cdots(M-i+1)}$$
- 如果 N 和 M 都很大, 那么可以近似地表达为 $(N/M)^i$

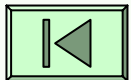




散列表算法分析 (2)

- 探查次数的期望值是1加上每个第*i*次 ($i \geq 1$) 冲突的概率之和, 即:

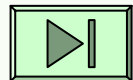
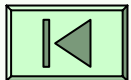
$$1 + \sum_{i=1}^{\infty} (N / M)^i = 1 / (1 - a)$$



散列表算法分析 (3)

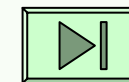
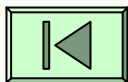
- 一次成功检索(或者一次删除)的代价与当时插入的代价相同
- 由于随着散列表中记录的不断增加, α 值也不断增大
- 我们可以根据从0到 α 的当前值的积分推导出插入操作的平均代价(实质上是所有插入代价的一个平均值):

$$\frac{1}{a} \int_0^a \frac{1}{1-x} dx = \frac{1}{a} \ln \frac{1}{1-a}$$

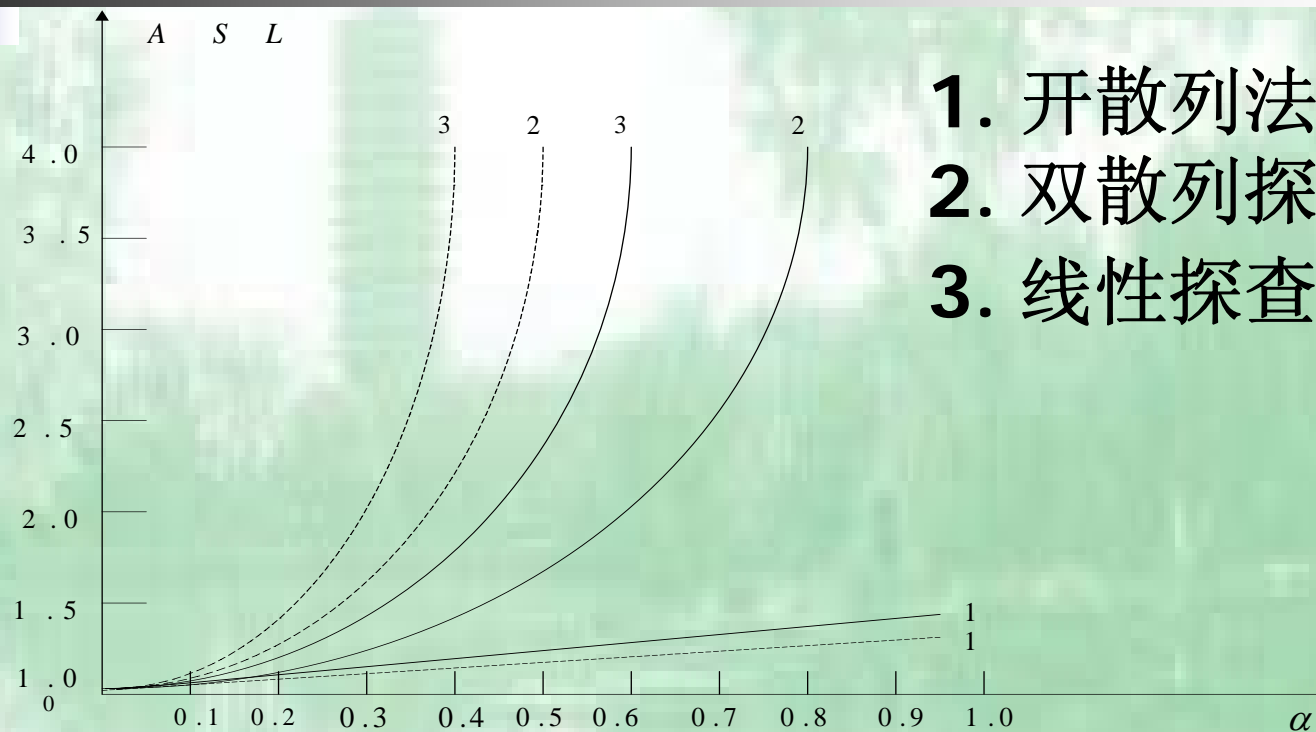


散列表算法分析（表）

编号	冲突解决策略	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	双散列探查法	$\frac{1}{\alpha} \ln \frac{1}{1 - \alpha}$	$\frac{1}{1 - \alpha}$
3	线性探查法	$\frac{1}{2} \left(1 + \frac{1}{1 - \alpha} \right)$	$\frac{1}{2} \left(1 + \frac{1}{(1 - \alpha)^2} \right)$

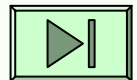
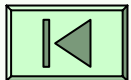


散列表算法分析（图）



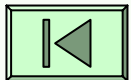
1. 开散列法
2. 双散列探查法
3. 线性探查法

■ 图9-14 用几种不同方法解决碰撞时散列表的平均检索长度。实线显示的是删除或成功检索的时间代价，虚线显示的是插入或不成功检索情况下的时间代价



散列表算法分析结论 (1)

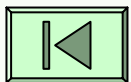
- 散列方法的代价一般接近于访问一个记录的时间，效率非常高，比需要 $\log n$ 次记录访问的二分检索好得多
 - 不依赖于 n ，只依赖于负载因子 $\alpha = n/M$
 - 随着 α 增加，预期的代价也会增加
 - $\alpha \leq 0.5$ 时，大部分操作的分析预期代价都小于2（也有人说1.5）
- 实际经验也表明散列表负载因子的临界值是0.5（将近半满）
 - 大于这个临界值，性能就会急剧下降





散列表算法分析结论（2）

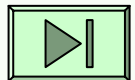
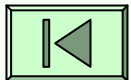
- 散列表的插入和删除操作如果很频繁，将降低散列表的检索效率
 - 大量的插入操作，将使得负载因子增加
 - 从而增加了同义词子表的长度
 - 也就是增加了平均检索长度
 - 大量的删除操作，也将增加墓碑的数量
 - 这将增加记录本身到其基地址的平均长度





散列表算法分析结论（3）

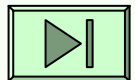
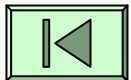
- 实际应用中，对于插入和删除操作比较频繁的散列表，可以定期对表进行重新散列
 - 把所有记录重新插入到一个新的表中
 - 清除墓碑
 - 把最频繁访问的记录放到其基地址





9.4 总结

- 基本概念
- 线性表的检索
- 集合
- 散列表的检索





The End

Thank You!