第十章　检索

张铭　主讲

采用教材：张铭，王腾蛟，赵海燕 编写
高等教育出版社，2008. 6 （“十二五”国家级规划教材）

http://jpk.pku.edu.cn/course/sjjg/
https://www.icourse163.org/course/PKU-1002534001

第十章　检索

- 检索的基本概念
- 10.1 线性表的检索
- 10.2 集合的检索
- 10.3 散列表的检索
- 总结

学习目标

- 检索的地位
  - 排序、索引、检索
  - 面向用户
- 检索的度量：**平均检索长度ASL**
  - 成功检索
  - 失败检索
- **散列检索**
  - 散列函数的选择
  - 冲突的解决方案（开散列、闭散列）

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、散列检索
- 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

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

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

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

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

- 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

检索算法评估的其他问题

- 衡量一个检索算法还需要考虑
  - 算法所需的存储量
  - 算法的繁杂性
  - …

思考

- 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是多少？（即平均需要多少此次比较给定值与表中关键码值才能找到待查元素）

第十章　检索

- 检索的基本概念
- 10.1 线性表的检索
- 10.2 集合的检索
- 10.3 散列表的检索
- 总结

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

“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>
class Item {
private:
    Type  key;                                  // 关键码域
                                                // 其它域
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
};
vector<Item<Type>*>  dataList;
template  <class Type>  int SeqSearch(vector<Item<Type>*>& dataList, int
length, Type k) {
    int i=length;
    dataList[0]->setKey (k);                    // 将第0个元素设为待检索值，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等概率 $P_i = 1/n$

$$\sum_{i=0}^{n-1} Pi \cdot (n-i) = \frac{1}{n} \sum_{i=0}^{n-1} (n-i)$$

$$= \sum_{i=1}^{n} i = \frac{n+1}{2}$$

- 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 $q=(1-p)$

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

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

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 $\Theta(n)$

二分检索法

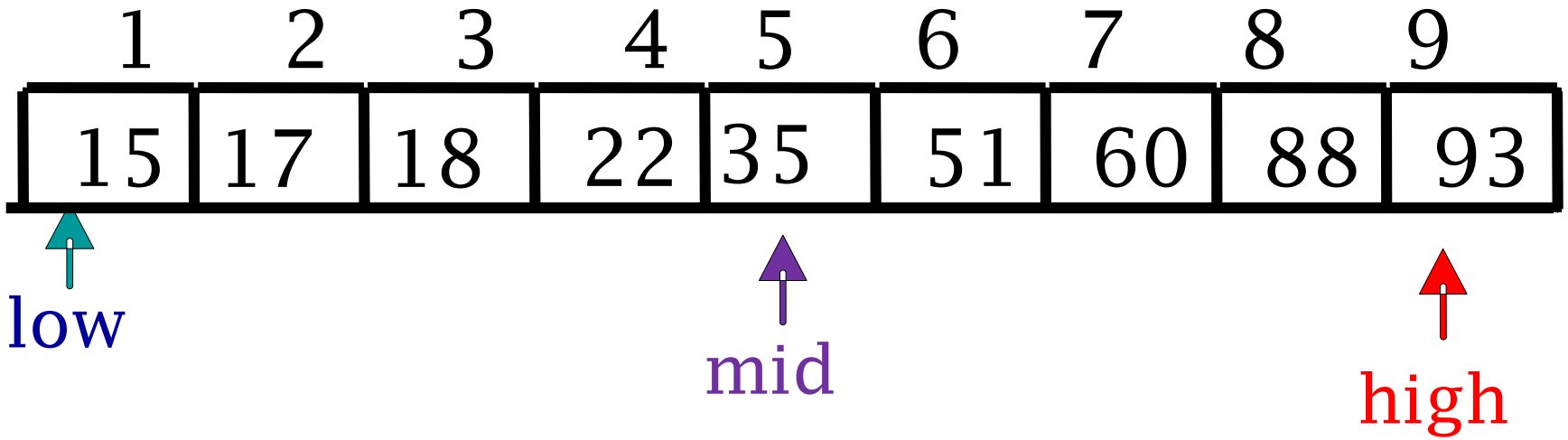
- 将任一元素 dataList[i] .Key 与给定值 K 比较，三种情况：
  - （1）Key = K，检索成功，返回 dataList[i]
  - （2）Key > K，若有则一定排在 dataList[i]前
  - （3）Key < K，若右则一定排在 dataList[i]后

- 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)  {                          // 算法结束条件
        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
```

关键码18 low=1 high=9



第一次：l=1, h=9, mid=5; array[5]=35>18

第二次：l=1, h=4, mid=2; array[2]=17<18

第三次：l=3, h=4, mid=3; array[3]=18 = 18

二分法检索性能分析

- 最大检索长度为 （完全二叉树的高度）

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

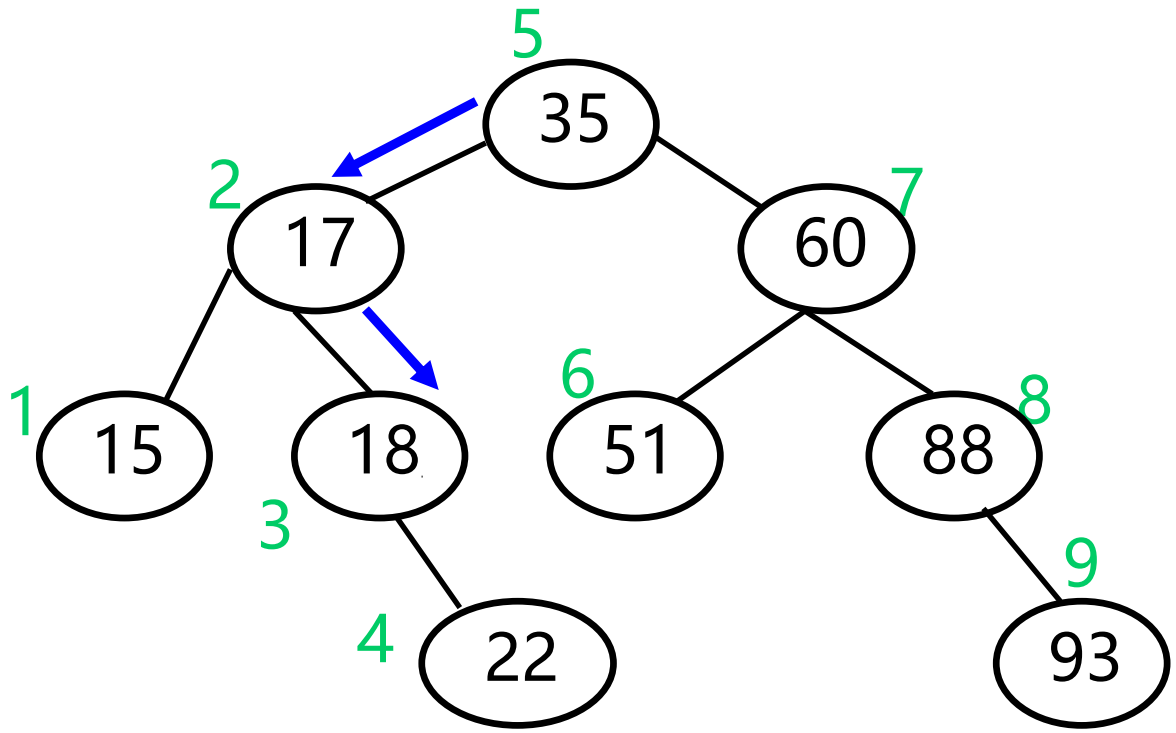
- 失败的检索长度是

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

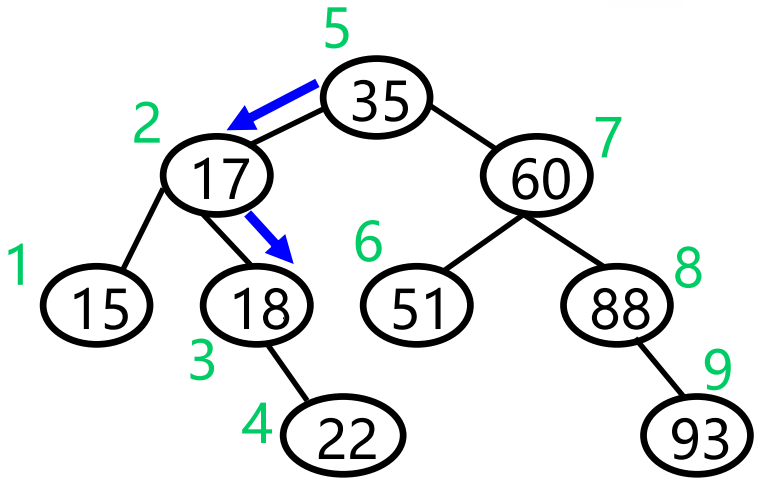
或

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

- 在算法复杂性分析中
  - log n 是以2为底的对数
  - 以其他数值为底，算法量级不变



二分法检索性能分析（续）



- 成功的平均检索长度为：

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

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

$$\approx \log_2(n+1) - 1 \qquad (n > 50)$$

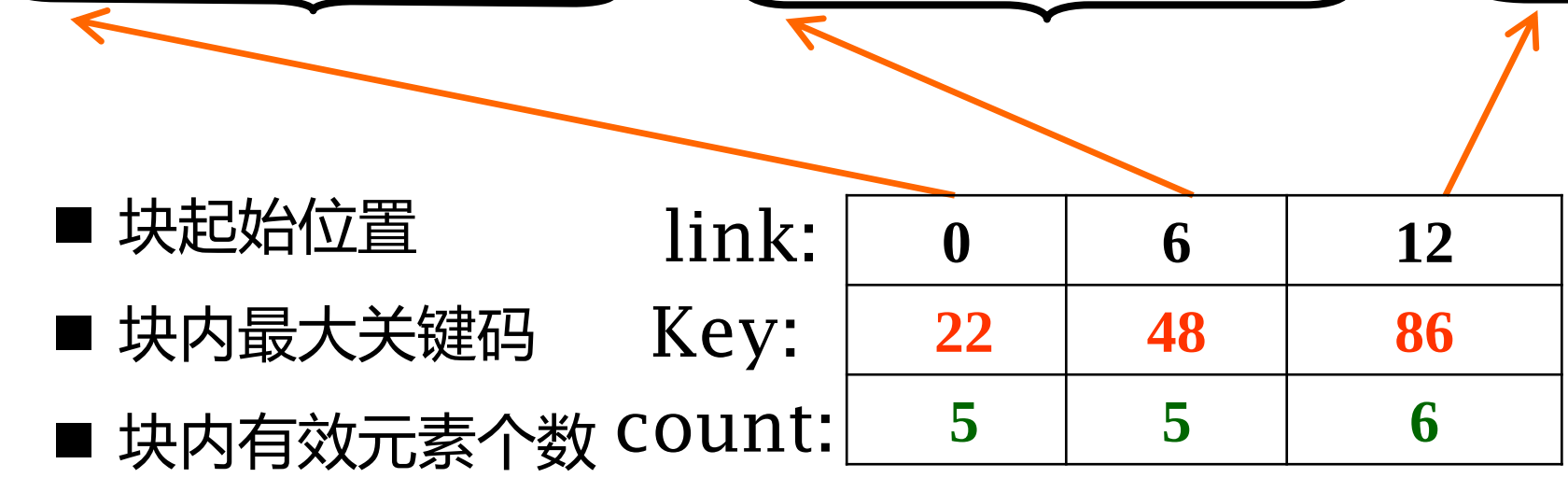
- 优缺点
  - 优点：平均与最大检索长度相近，检索速度快
  - 缺点：要排序、顺序存储，不易更新(插/删)

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
  - 前一块最大关键码必须小于后一块最小关键码
  - 每一块中的关键码不一定有序
- 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——索引顺序结构

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



- 块起始位置
- 块内最大关键码
- 块内有效元素个数

link:	0	6	12
Key:	22	48	86
count:	5	5	6

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，ASL_b
  - 然后在块内检索待查的元素，ASL_w

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

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

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

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

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

- 当 s $=\sqrt{n}$ 时，ASL 取最小值

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

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

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

思考

- 试比较顺序检索、二分检索和分块检索的优缺点。

- 这几种检索方法适合的应用场景分别是什么?

第十章　检索

- 检索的基本概念
- 10.1 线性表的检索
- 10.2 集合的检索
- 10.3 散列表的检索
- 总结

集合

- 集合（set）：由若干个确定的、相异的对象（element）构成的整体

- 集合的检索：确定一个值是不是某个集合的元素

10.2 集合的检索

	运算名称	数学运算符号	计算机运算符号
算术运算	并	∪	+、|、*OR*
	交	∩	*、&、*AND*
	差	-	-
	相等	=	==
	不等	≠	!=
逻辑运算	包含于	⊆	<=
	包含	⊇	>=
	真包含于	⊂	<
	真包含	⊃	>
	属于	∈	IN、at

集合的抽象数据类型

```
template<size_t N>                          // N为集合的全集元素个数
class mySet {
public:
        mySet() ;                           // 构造函数
        mySet(ulong X);
        mySet<N>& set();                    // 设置元素属性
        mySet<N>& set(size_t P, bool X = true);
        mySet<N>& reset();                  // 把集合设置为空
        mySet<N>& reset(size_t P);          // 删除元素P
        bool at(size_t P) const;            // 属于运算
        size_t count() const;               // 集合中元素个数
        bool none() const;                  // 判断是否空集
```

集合的抽象数据类型

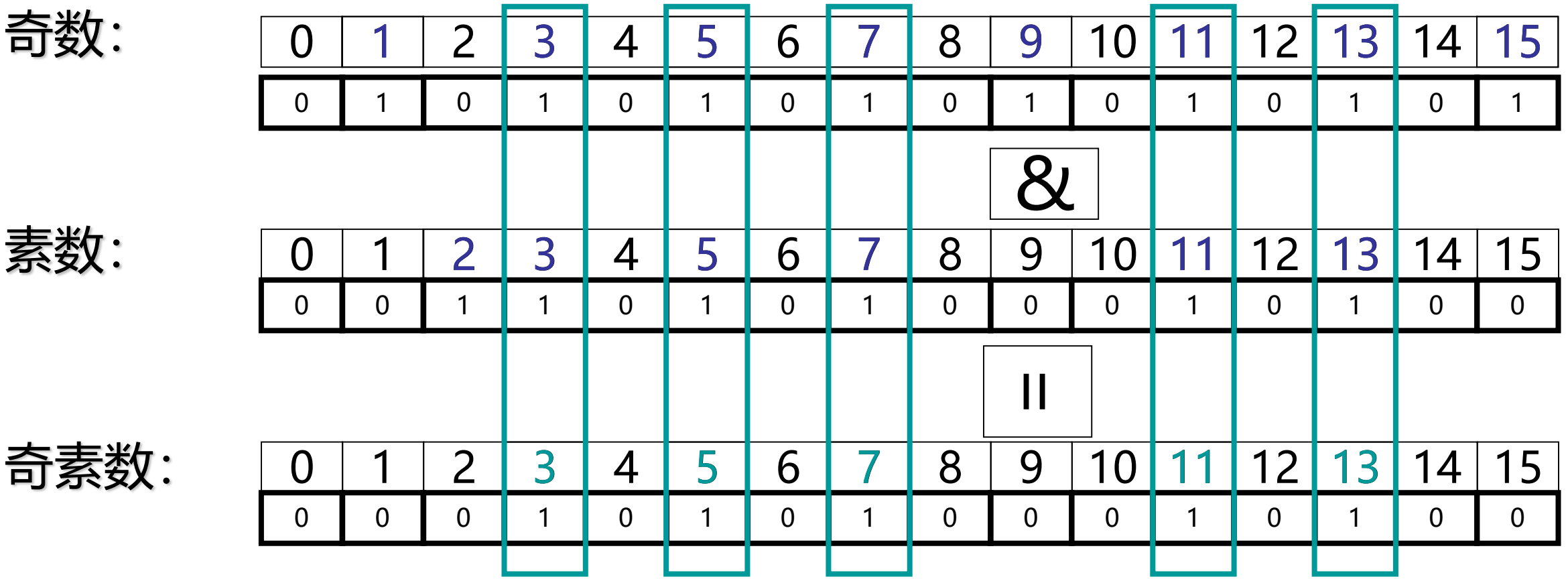
```
bool operator==(const mySet<N>& R) const;          // 等于
bool operator!=(const mySet<N>& R) const;          // 不等
bool operator<=(const mySet<N>& R) const;          // 包含于
bool operator< (const mySet<N>& R) const;          // 真包含于
bool operator>=(const mySet<N>& R) const;          // 包含
bool operator>(const mySet<N>& R) const;           // 真包含

friend mySet<N> operator&(const mySet<N>& L, const mySet<N>& R);  // 交
friend mySet<N> operator|(const mySet<N>& L, const mySet<N>& R);  // 并
friend mySet<N> operator-(const mySet<N>& L, const mySet<N>& R);  // 差
friend mySet<N> operator^(const mySet<N>& L, const mySet<N>& R);  // 异或
};
```

集合的检索

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

- 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）

例：计算0到15之间所有的奇素数



例：集合的无符号数表示

- 全集元素个数 N=40 的集合
- 集合{35, 9, 7, 5, 3, 1} 用 2 个 ulong 表示

0000 0000 0000 0000 0000 0000 0000 1000
0000 0000 0000 0000 0000 0010 1010 1010

不够一个 ulong , 左补 0

```
typedef unsigned long ulong;
enum {
    // unsigned long数据类型的位的数目
    NB = 8 * sizeof (ulong),
    // 数组最后一个元素的下标
    LI = N == 0 ? 0 : (N - 1) / NB
};
    // 存放位向量的数组
ulong A[LI + 1];
```

设置集合元素

```
template<size_t N>
mySet<N>& mySet<N>::set(size_t P, bool X) {
        if (X)              // X为真，位向量中相应值设为1
            A[P / NB] |= (ulong)1 << (P % NB);
                            // P对应的元素进行按位或运算
        else    A[P / NB] &= ~((ulong)1 << (P % NB));
                            // X为假，位向量中相应值设为0
        return (*this);
}
```

集合的交运算“&”

```
template<size_t N>
mySet<N>& mySet<N>::operator&=(const mySet<N>& R)
{  // 赋值交
    for (int i = LI; i >= 0; i--)       // 从低位到高位
        A[i] &= R.A[i];                 // 以ulong元素为单位按位交
    return (*this);
}
template<size_t N>
mySet<N> operator&(const mySet<N>& L, const mySet<N>& R)
{   //交
    return (mySet<N>(L) &= R);
}
```

思考

- 集合还可以用哪些技术来实现?
- 调研 STL 中集合的各种实现方法。

第十章　检索

- 检索的基本概念
- 10.1 线性表的检索
- 10.2 集合的检索
- 10.3 散列表的检索
- 总结

散列检索

- **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 闭散列表的算法实现
- 10.3.5 散列方法的效率分析
- 10.3.6 散列方法的应用

散列函数 [编辑]

维基百科，自由的百科全书

"**散列**"重定向至此。关于其他用法，请见"**散列 (消歧义)**"。

散列函数（英语：Hash function）又称**散列算法**、**哈希函数**，是
一个短的随机字母和数字组成的字符串来代表。[1]好的散列函数

如今，散列算法也被用来加密存在数据库中的密码（password）

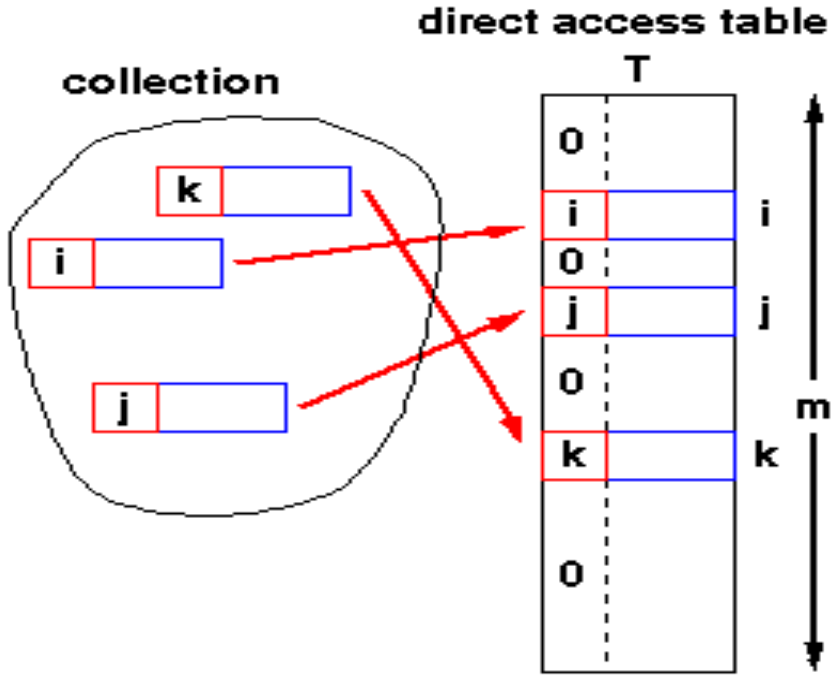
目录 [隐藏]

基于关键码检索的局限

- 基于**关键码比较**的检索
  - 顺序检索，==, !=
  - 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散列

- 例如，读取指定下标的数组元素
  - 受此启发，计算机科学家发明了散列方法 (Hash, 有人称“哈希”或“杂凑”)
- 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}

可设散列表为：

char HT2[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

```
// 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(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（或叫**碰撞**）
  - 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

M 不用幂

$x \bmod 2^8$ 选择最右边 8 位

01100101110**00011010**

- 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

$$hash(123456) =$$
$$= \lfloor 10000*(0.6180339*123456 \ \% \ 1) \rfloor =$$
$$= \lfloor 10000 * (76300.0041151... \ \% \ 1) \rfloor =$$
$$= \lfloor 10000 * 0.0041151... \rfloor = 41$$

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 - ⌊ A * key ⌋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**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）

- 各位数字中符号分布的均匀度 λ_k

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

  - 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，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）

- 若散列表地址范围有 3 位数字, 取各关键码的 ④ ⑤ ⑥ 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

①	②	③	④	⑤	⑥
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， $\lambda_6 = 5.60$

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
  - 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

- 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10.6] 如果一本书的编号为 04-42-20586-4

5 8 6 4	0 4 4 2 2 0 5 8 6 4
4 2 2 0	0 2 2 4 4 0
+ 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**

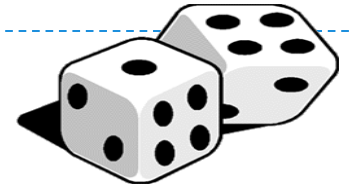
7. ELFhash字符串散列函数

- 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**每个字符都参与运算**，有同样的作用
- 对于散列表中的位置不可能产生不平均的分布



散列函数的应用

- Java 的 hashCode()方法
  - 所以数据类型，返回32位整数的散列值
- 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

思考

- 采用散列技术时考虑：
  - （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?

- 散列表本身的组织方法

散列检索

- 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 闭散列表的算法实现
- 10.3.5 散列方法的效率分析
- 10.3.6 散列方法的应用

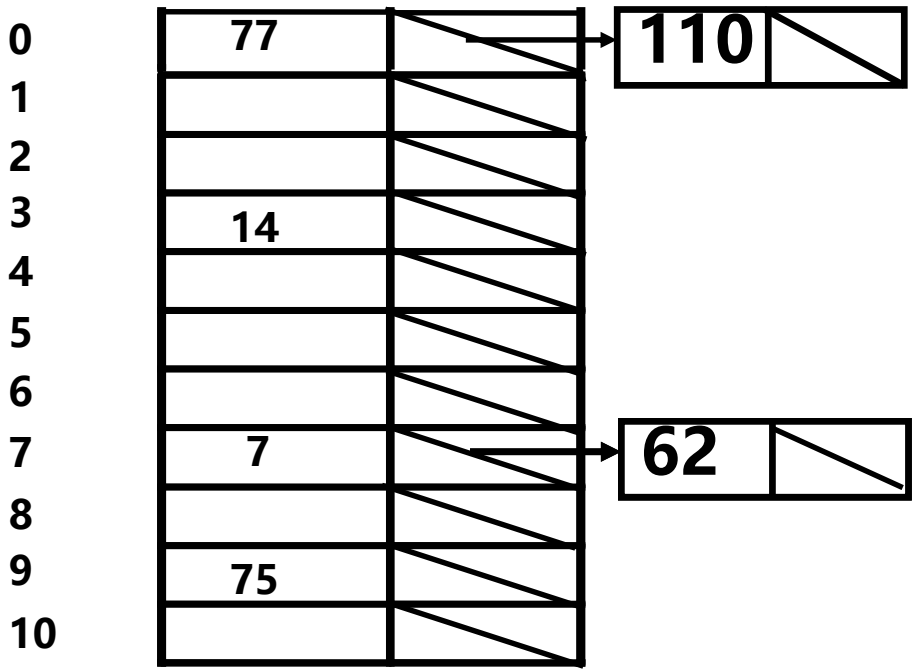
常用散列函数选取方法

- 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

开散列法

{77、14、75、 7、110、62 、95 }

■h(Key) = Key % 11



- 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

开散列性能分析

- 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，拉链开散列方法的**平均代价就是 Θ(1)**

10.3.3 闭散列方法

- d_0=h（K）称为 K 的**基地址**
- 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_1，d_2，… d_i，…，d_{m-1}
  - 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
- 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

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

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，那么就在表中下移，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，……，M-1，0，1，……，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

散列表示例

- M = 15，h(key) = 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	11	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

    $(\mathrm{h}(K) + ic) \bmod M$
  - 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 $\mathbf{p(K, i) = i*c}$
  - *必须使常数 c 与 M 互素*

例：改进线性探查

- 例，$c = 2$，要插入关键码 k_1 和 k_2，$\mathrm{h}(k_1) = 3$，$\mathrm{h}(k_2) = 5$
- 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
  - 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，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

- 消除基本聚集
  - 基地址不同的关键码，其探查序列有所重叠
  - 伪随机探查和二次探查可以消除
- 二级聚集 (secondary clustering)
  - 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，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

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

  - 探查序列函数

$$d = h_1(key)$$

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

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，则再计算 $h_2(key)$，得到的探查序列为：

  $(d+h_2(key)) \% M$，$(d+2h_2(key)) \% M$，

  $(d+3h_2(key)) \% M$，…
- **$h_2(key)$ 尽量与 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
- 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 $1 \le h2(K) \le M-1$ 范围之间
- 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，$h_1(K) = K \% M$
  - $h_2(K) = K \% (M-2) + 1$
  - 或者 $h_2(K) = [K / M] \% (M-2) + 1$
- 方法4： 若 M 是任意数，$h_1(K) = K \% p$，（p 是小于 M 的最大素数）
  - $h_2(K) = K \% q + 1$（q 是小于 p 的最大素数）

思考

- 插入同义词时，如何对同义词链进行组织?
- 双散列函数 $h_2(key)$ 与 $h_1(key)$ 有什么关系?

散列检索

- 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 闭散列表的算法实现**
- 10.3.5 散列方法的效率分析
- 10.3.6 散列方法的应用

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插入和查找

```
bool hashInsert(const Elem&);                  // insert(key, value)
bool hashSearch(const Key&, Elem&) const; // lookup(key)
```

散列字典ADT（属性）

```
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 h(int x) const ;         // 散列函数
    int h(char* x)const ;        // 字符串散列函数
    int p(Key K, int i)          // 探查函数
```

散列字典ADT（方法）

```
public:
hashdict(int sz, Elem e) {                    // 构造函数
    M=sz; EMPTY=e;
    currcnt=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 currcnt; }                // 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

散列表插入算法代码

```
// 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;
}
```

检索算法

- 与插入过程类似
  - 采用的探查序列也相同
- 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
    - 走到探测序列尾部还没找到，检索失败

10.3 散列检索

```
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，因为检索算法 “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		

- 例，长度 $M = 13$ 的散列表，假定关键码 $k1$ 和 $k2$，$h(k1) = 2$，$h(k2) = 6$。
- 二次探查序列
  - $k1$ 的二次探查序列是 2、3、1、6、11、11、6、5、12、...
  - $k2$ 的二次探查序列是 6、7、5、10、2、2、10、9、3、...
- 删除位置 6，用$k2$ 序列的最后位置 2 的元素替换之，位置 2 设为空
- 检索 $k1$ ，查不到（事实上还可能存放在位置 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
- 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

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版

```
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;
}
```

散列检索

- 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 闭散列表的算法实现
- **10.3.5 散列方法的效率分析**
- 10.3.6 散列方法的应用

散列方法的效率分析

- 衡量标准：插入、删除和检索操作 所需要的记录访问次数
- 散列表的插入和删除操作 都基于检索进行
  - 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

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 α= 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（1）

- 基地址被占用的可能性是 α
- 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

$$(N/M)^i$$

- 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，即插入代价：

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

散列表算法分析（2）

- 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)：

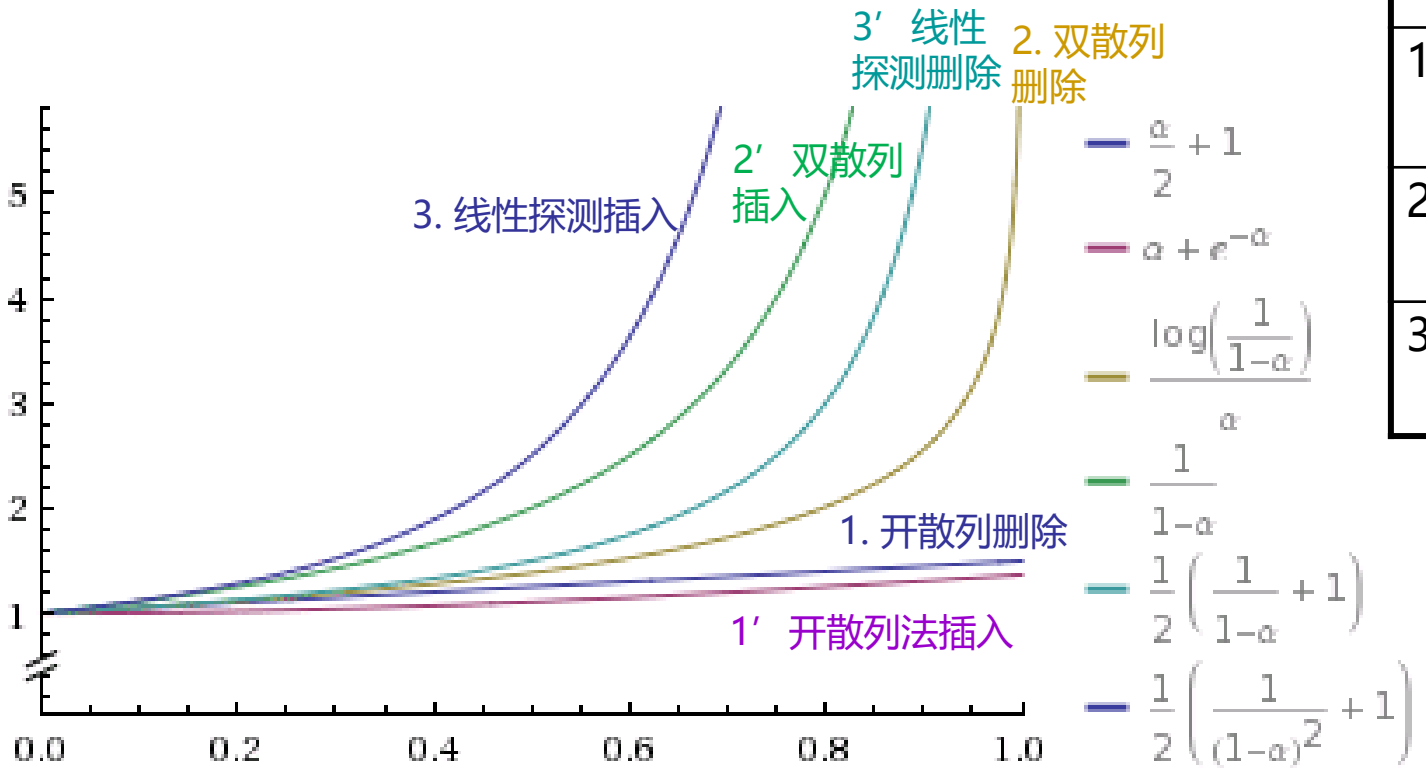
$$\frac{1}{a}\int_0^a \frac{1}{1-x}dx = \frac{1}{a}\ln\frac{1}{1-a}$$

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）

- 用几种不同方法解决碰撞时散列表的平均检索长度



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，即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
- 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**

思考

- 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？
- 调研除散列以外字典的其他实现方法

散列方法的应用

- 1. Bloom Filter
- 2. Karp Rabin 模式匹配
- 2. MD5 消息摘要
- 3. SHA 数字签名

数据结构与算法

谢谢倾听

国家精品课“数据结构与算法”

http://jpk.pku.edu.cn/course/sjjg/

https://www.icourse163.org/course/PKU-1002534001

张铭，王腾蛟，赵海燕

高等教育出版社，2008.6。“十二五”国家级规划教材