数据结构与算法

第10章　检索

主讲：赵海燕
北京大学信息科学技术学院
“数据结构与算法”教学组

国家精品课“数据结构与算法”
http://www.jpk.pku.edu.cn/pkujpk/course/sjjg/

张铭，王腾蛟，赵海燕
高等教育出版社，2008. 6, “十一五”国家级规划教材

基本内容

- 基本概念
- 线性表的检索
- 集合的检索
- 散列表的检索
- 总结

基本概念

- 检索
  - 计算机中使用最频繁的任务之一
  - 在一组记录集合中找到关键码/属性值 等于或符合 给定值/条件 的某个/组记录，或者找到关键码/属性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

- 在数据结构中查找满足某种条件的结点

  假定一个具有n个记录集合 T 的形式如下：

  $$(k_1, R_1)，(k_2, R_2)，\ldots，(k_n, R_n)$$

  其中$k_1, k_2 \ldots, k_n$是互不相同的关键码，R_j 是与关键码k_j 相关的信息，$1 \le j \le n$

  - 按关键码检索
    - 给定一个特定的关键码值 k，检索问题是在 T 中确定记录(k_j, R_j)，使得 $k_j = k$
  - 按属性检索
    - 给定一个属性的值，在 T 中找出某属性值等于指定值的结点

基本概念

- 检索结果
  - 成功
    - 在数据集中找到至少一个的记录，使得满足检索条件
  - 失败
    - 在数据集中找不到满足检索条件的记录

检索的分类

- **精确匹配查询（exact-matching query）**
  - 检索匹配某个特定值的记录
- **范围查询（range query）**
  - 检索给定属性的取值在某个指定值范围内的所有记录

检索算法分类

- 根据被检索数据的组织方式，检索算法可分为：
  - 线性表方法
    - 顺序检索、二分法检索、分块检索
  - 散列法（根据关键码值直接访问法）
  - 树索引法
    - 二叉排序树、字符树、 ISAM文件、B-树
  - 基于属性的检索
    - 倒排表、多重表

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
  不适合进行范围查询
  一般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，可以选择B树方法

平均检索长度（ASL）

- 检索运算的**基本操作**：元素的比较
- **平均检索长度**(Average Search Length：ASL)
  - 检索过程中对元素的**平均比较次数**
  - 衡量检索算法优劣的时间标准

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

- ✓ *ASL*: 存储结构中对象规模 n 的函数
- ✓ P_i 为检索第 i 个元素的概率
- ✓ C_i：找到第 i 个元素所需的比较次数

平均检索长度示例

- 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
    0.4×1+0.1×2+0.5×3 = 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

- 衡量一个检索算法还需考虑
  - 算法所需的**存储代价**
  - 算法的**复杂性**
  - …

基于线性表的检索

- 顺序检索
- 二分检索
- 分块检索

顺序检索

- 针对线性表里的所有记录，按**某种方式**逐个比较元素的特定域与**给定值**
  - 若某个记录的特定域和给定值相等，则检索成功；
  - 否则检索失败（找遍了仍找不到）
- 特殊要求
  - 存储：可以顺序、链接
  - 排序要求：无

带“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是等概率的：$P_i = 1/n$

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

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

- 检索失败

  假设检索失败时需要 n+1 次比较（设置了一个**监视哨**）

顺序检索的ASL

- 假设 检索成功的概率为 *p*，则检索失败的概率为 *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
  - 检索时间长　$\Theta(n)$

对检索频率不等的线性表如何处理?

检索频率不等

- Zipfian's 分布
  - 出现概率遵循Zipfian'law 的分布
    - Zipfian's law: $p_n \sim 1/n^a$, where p_n is the frequency of occurrence of the nth ranked item and a is close to 1
  - 主要应用:
    - 自然语言中单词使用的频率
    - 城市中人口数量的分布
- 80/20 rules
  - 80%的访问针对20%的数据

顺序检索的改进

1. 计数方法
   - 为每个记录保存一个访问计数，且一直维护计数
   - 组织成**按频率排序**的线性表
   - 缺点：保存访问计数需要空间，时间推移访问频率改变
2. 向前移动方法
   - 一个记录被找到时就将其移至线性表的头部，其他记录顺退一个位置
   - 适宜于**链表实现**，对访问频率的**局部变化**能够很好地响应

顺序检索的改进

3. 转置方法
   - 把找到的记录与其在线性表中的**前驱记录交换**位置
   - 随着时间的推移，**最常使用**的记录将移动到**线性表头部**；曾经被频繁访问但是以后不再使用的记录将会慢慢地落在后面
   - 适宜于顺序表和链表
   - 例外的**极端**情况

二分检索法

- 将任一元素 dataList[i].Key 与给定值 K 比较
  - 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，若有，则一定排在dataList[i]后

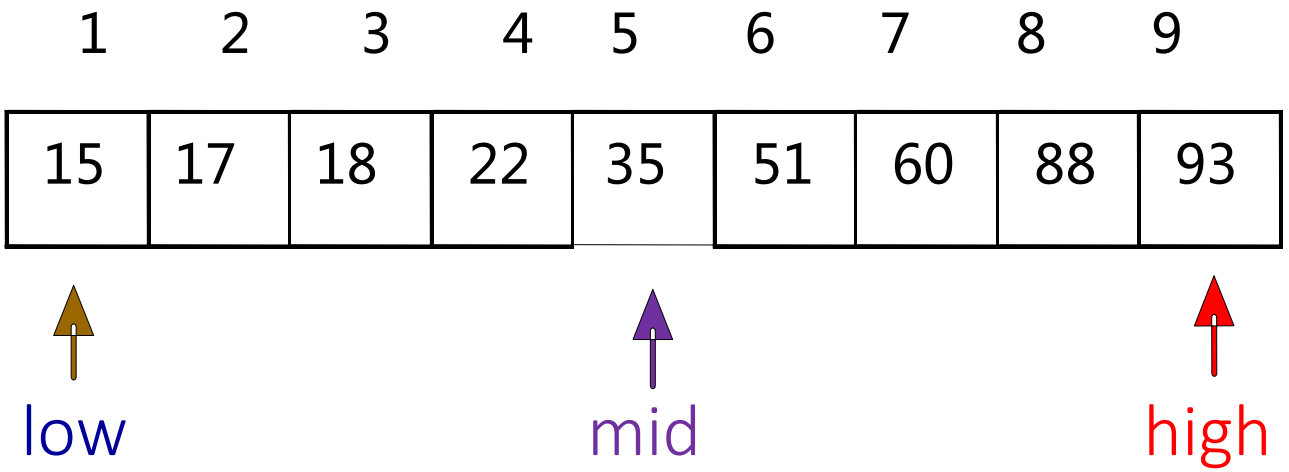
- 缩小进一步检索的区间

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
}
```

二分检索示例

关键码18； low = 1 high = 9



第1次： l=1, h=9, mid=5; array[5]=35>18

第2次： l=1, h=4, mid=2; array[2]=17<18

第3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

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

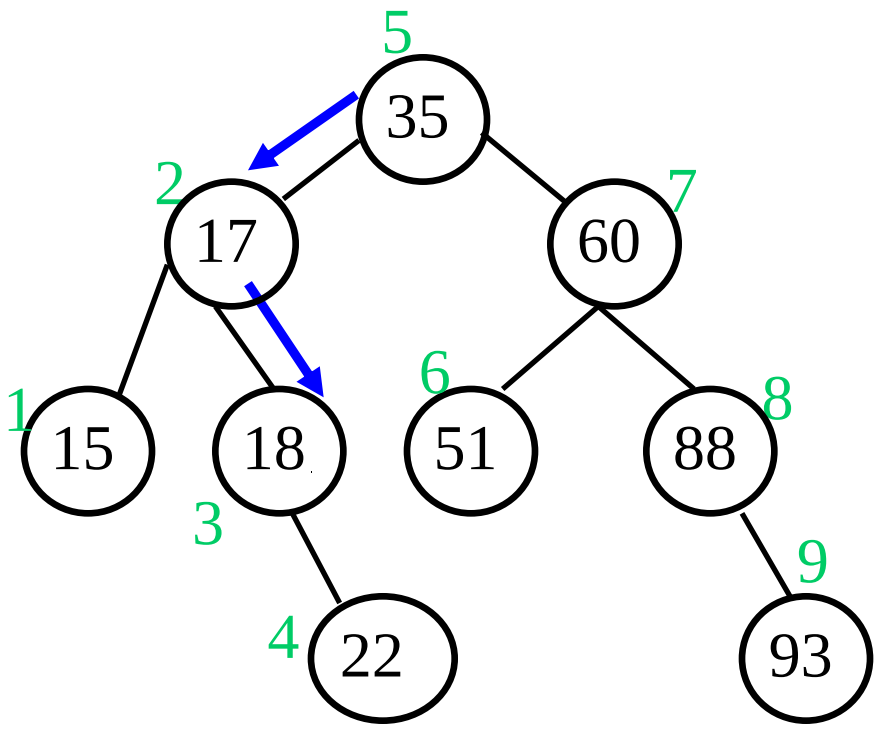
- 失败的检索长度是

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

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
  - 其他底，算法量级不变



二分法检索性能分析

- 成功的平均检索长度为

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

- 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)

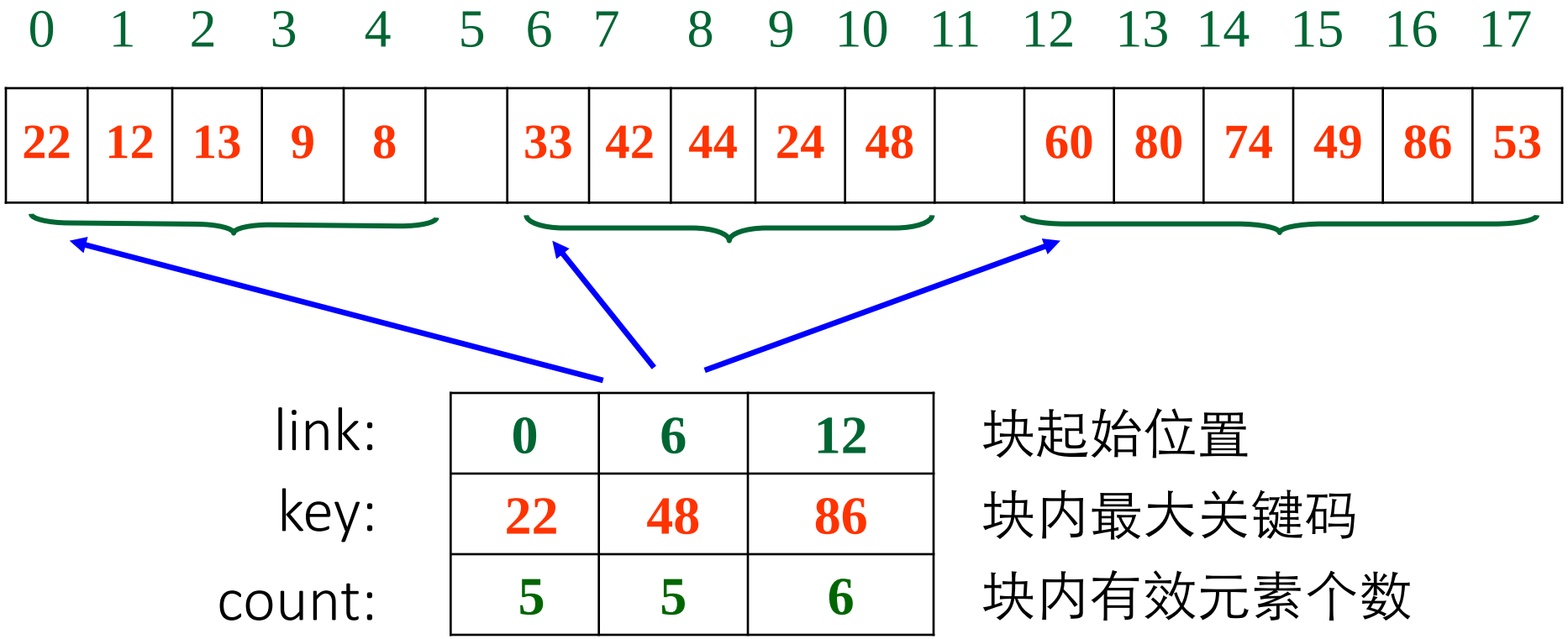
分块检索

- 顺序与二分法的**折衷**
  - 既有 较快 的检索，又有 较灵活 的更改
- 基本思想：按块有序
  - 将线性表分成若干数据块，每一块中结点的存放是任意的，但**块与块间有序**
- 建立一个**索引表**(辅助数组)，每个数据块最大的关键码按块的顺序存放在索引表中
- 检索时，先在索引中查找关键码所在的**块号**，再到相应数据块中查找关键码所在的**记录**

索引表

- 一个递增有序表
  - 表间分块有序
- 存放内容
  - 各块中的最大关键码
  - 各块起始位置
  - 可能还需要块中元素个数（每一块可能不满）

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
link: 0 6 12 块起始位置
key: 22 48 86 块内最大关键码
count: 5 5 6 块内有效元素个数

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
  - 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

有

$$ASL(n) = ASL_b + ASL_w$$

分块检索性能分析

- 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

- 例如，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
- 若数据块（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

- 若采用二分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

$$\begin{aligned} \mathrm{ASL} &= \mathrm{ASL}_{\mathrm{b}} + \mathrm{ASL}_{\mathrm{w}} \\ &= \log(b+1) - 1 + \frac{s+1}{2} \\ &= \log(1+\frac{n}{s}) + \frac{s}{2} \end{aligned}$$

分块检索的优劣

- 优点
  - 插入、删除相对较易，只要找到其应属的块，因为块内结点任意存放，故没有大量记录移动
- 缺点
  - 增加一个索引空间
  - 初始线性表的分块运算
  - 当大量插入/删除时，或结点分布不均匀时，速度下降

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
  - 确定一个值是否某个集合的元素（集合归属）

集合运算

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

集合的抽象数据类型

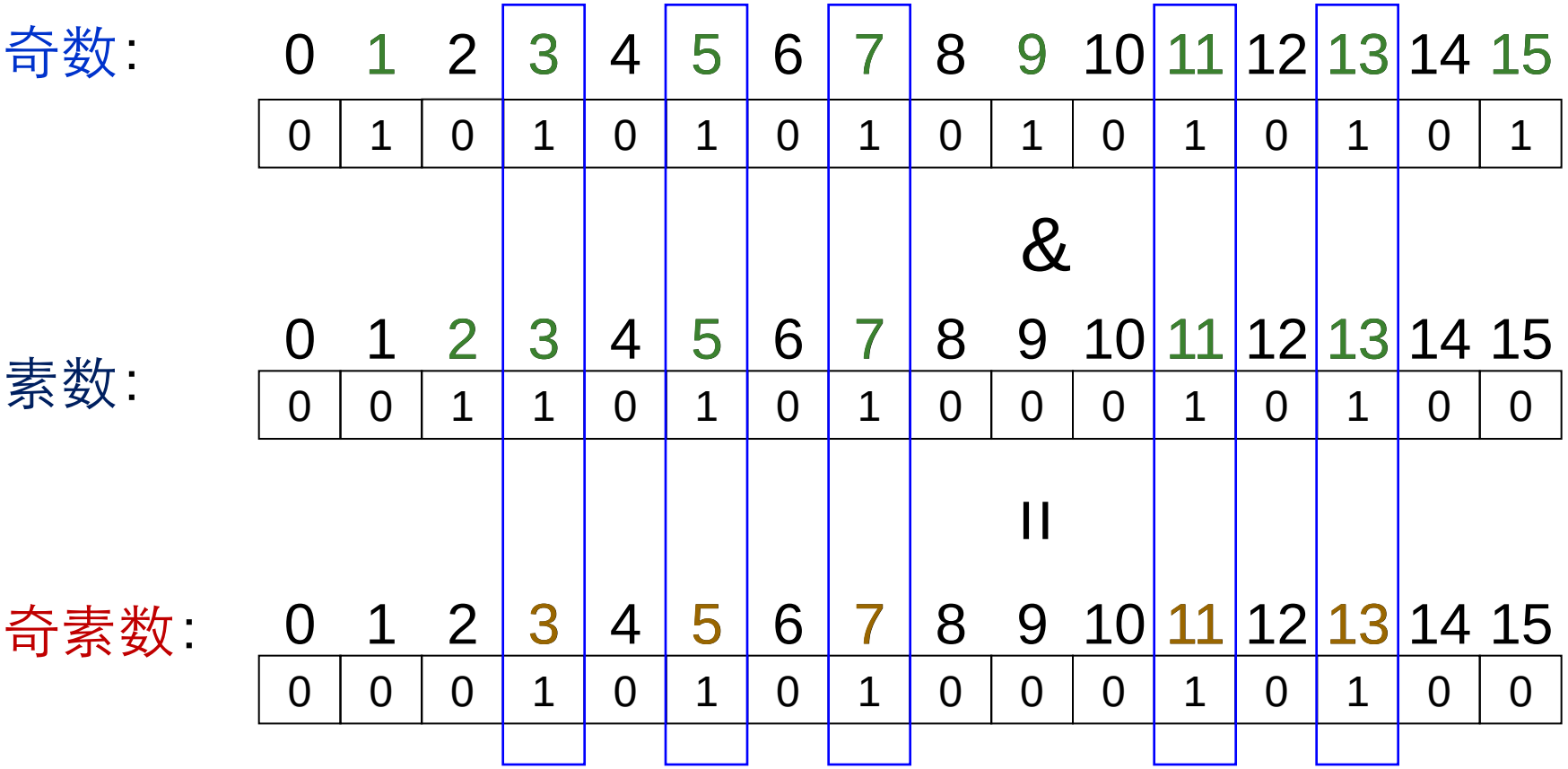
```
    bool operator<=(const mySet<N>& R) const;       // 包含于
    bool operator< (const mySet<N>& R) const;       // 真包含于
    bool operator>=(const mySet<N>& R) const;       // 包含
    bool operator>(const mySet<N>& R) const;        // 真包含

    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

- 用位向量来表示集合
  - 适用于**密集型**集合（数据范围小，而集合中有效元素个数较多）
  - C++中提供位操作
  - STL中 vector<bool>

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

示例：计算0到15之间所有的奇素数



示例：集合的无符号数表示

- 一个全集元素个数N=40 的集合，其中的一个子集合 {35, 9, 7, 5, 3, 1} 表示为

两个无符号长整数 ulong

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

不够一个ulong, 左补0

集合的检索

```
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

```
// 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);
}
```

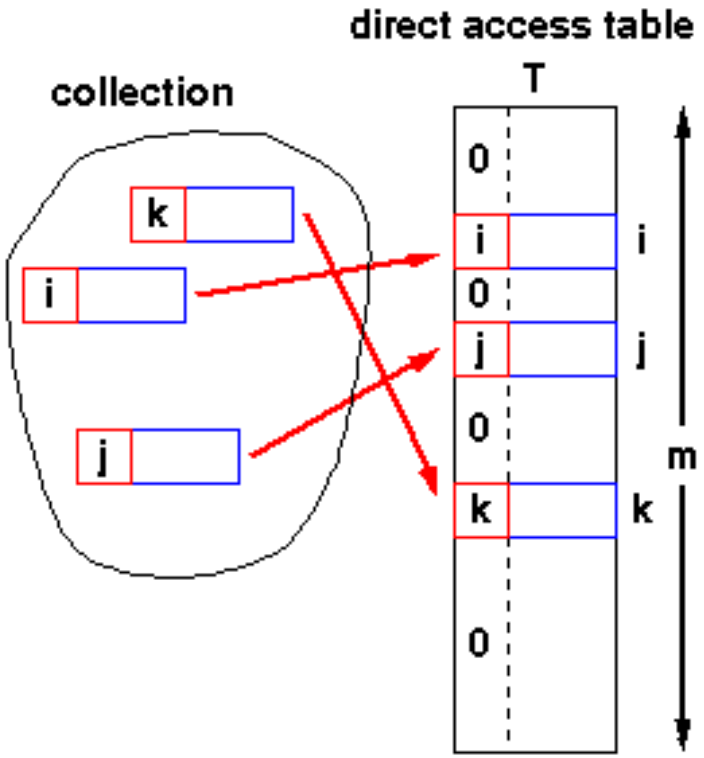
思考

- 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

检索的基本问题

- **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
  - 当问题规模很大时，基于比较的检索的时间效率可能使得用户无法忍受
- 理想情况
  - 根据记录取值（关键码），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

数组直接寻址的启示



- 例如，读取指定下标的数组元素
  - 根据数组的起始存储地址、以及下标值而直接计算其地址，时间代价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”）
  - 一种重要的存储方法
  - 也是一种常见的检索方法

散列的基本思想

- 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

- 检索时根据此函数计算其存储位置
  - 通常散列表的存储空间组织为一维数组
  - 散列地址是数组的下标

散列检索及技术

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

散列示例

已知关键码集合为：

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

可设散列表为：

char HT2[26][8];

若将散列函数 h(key)定义为关键码key中的首字母在字母表{a, b, c, …, z}中的序号，即：

h(k) = k[0] – ‘a’

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	

散列示例

若在集合S中增加 4个 关键码 形成集合

S1 = S + { else, array, with, up}

且将散列函数修改为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 )
}
```

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	

散列示例

- 冲突及其处理

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	

散列涉及的重要概念

- 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/碰撞（collision）
  - 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

碰撞示例

- 不同的关键码值，通过散列函数得到了相同的散列值，即，存在

  $k_1 != k_2$，但 $h(k_1) = h(k_2)$

  - 例，一组表项，其关键码分别为

    12361, 07251, 03309, 30976

    采用的散列函数为

    $h(x) = x \% 73$

    则有

    $h(12361) = h(07251) = h(03309) = h(30976)$

散列技术的两个要素

- 涉及散列的基本问题/首要问题可分成两类
  1. 如何构造（选择）使结点“**分布均匀**”的散列函数?
     - 散列函数需具备怎样的特性（properties）？怎样才能获得或设计一个具有这些特性的散列函数?
  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

逐一来考虑 ……

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

$$Address = h(key)$$

- 散列函数的选取原则
  - 运算尽可能简单（简单且快速）
  - 函数的值域必须在表长的范围内
  - 计算出来的地址应能均匀分布在整个地址空间中
    - 即，若 *k* 为关键码集合中一个随机抽取的关键码，散列函数应能以同等概率取 *0* 到 *M-1* 之间的每一个值

散列函数的选择

- 须考虑的因素
  - 计算时间（运算尽可能简单）
  - 关键码长度
  - 散列表大小
  - 关键码的分布情况
  - 记录的查找频率
  - ...

常用散列函数选取方法

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

散列函数也称杂凑函数： 各种拼凑方法

1. 除余法

- 除余法：用关键码 x 除以M(往往取与散列表长度有关的数)，取其余数作为散列地址：

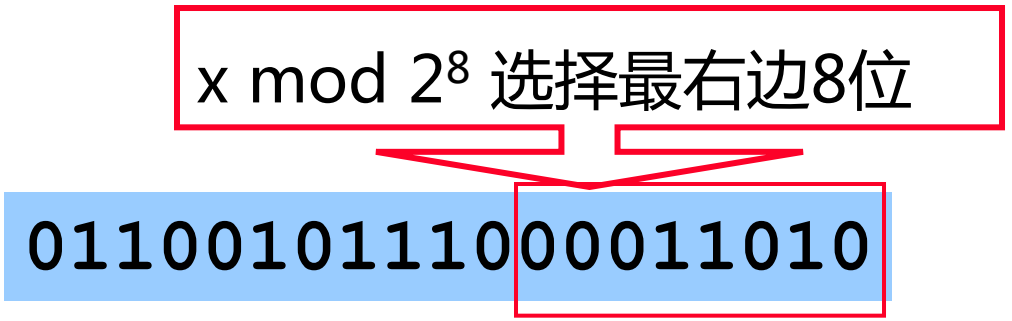
$$h(x) = x \bmod M$$

- 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非偶原则

- 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
  - 若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非幂原则



- 若把M设置为2的幂
  - 则有，h(*x*) = *x* mod 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
  - 则，h(*x*) = *x* mod 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
- 即使保证连续的关键码不发生冲突，但意味着要占据连续的散列单元
- 在某些冲突解决策略下，可能导致散列性能的降低

2. 乘余取整法

1. 先让关键码 x 乘上一个常数 $A(0<A<1)$，提取乘积的小数部分
2. 再用整数 n 乘以这个值，对结果向下取整，将其作为散列地址

亦即，散列函数为：

$$h(x) = \lfloor n * (A * x \% 1) \rfloor$$

“$A * x \% 1$”表示取 $A * x$ 小数部分：

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

乘余取整法示例

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

  h (123456) =

  = ⌊10000*(0.6180339*123456 % 1)⌋

  = ⌊10000 * (76300.0041151… % 1)⌋

  = ⌊10000 * 0.0041151…⌋

  = 41

乘余取整法的参数取值

- 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

    $A * x \% 1 = A * x - \lfloor A * x \rfloor$ 值的小数点后最左p位（bit）值
  - 优点：对 *n* 的选择无关紧要

- Knuth认为： A取值与待排序的数据特征有关，理论上可任意值。一般情况下取黄金分割最理想

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，且 r 种不同的符号在各位上出现的**频率不等**
  - 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

- 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

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
- 也可把第①②③和第⑤位相加，舍去进位，变成一位数，与第④⑥位合起来作为散列地址。
- 还可根据情况参用其它方法

数字分析法

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

- 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

- 基本思想
  - 把关键码看成是**另一进制**上的数
  - 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

基数转换法示例

- 例如，给定一个十进制数的关键码是$(210485)_{10}$，将其看作以13为基数的十三进制数$(210485)_{13}$，再转换为十进制

$$(210485)_{13} = 2\times13^5 + 1\times13^4 + 4\times13^2 + 8\times13 + 5$$
$$= (771932)_{10}$$

- 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

- 若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**成位数相同的若干部分（最后一部分的位数可以不同）
  - 然后取这些部分的**叠加和**（舍去进位）作为散列地址

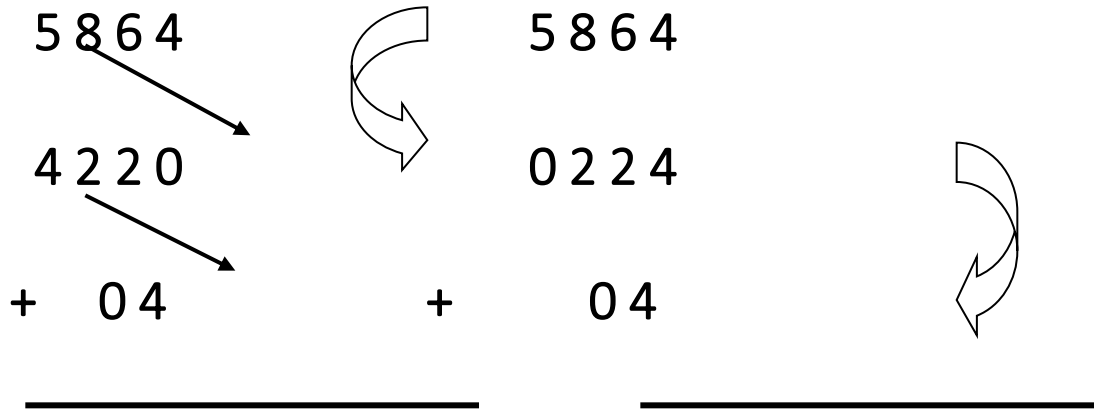
两种折叠方法

- 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，沿其分界来回折叠，然后对齐相加

将相加的结果当做散列地址

折叠法示例

- 若一本书的编号为04-42-20586-4



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均有效
- 字符串中每个字符起到相同的作用
- 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
- 曾有人用“轮盘赌”的统计分析方法对上述散列函数进行模拟分析，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

- 散列表本身的组织方法?