

# 第5章 查找





# 学习目标

- ◆ **查找**是指在某种数据结构上找出满足给定条件的数据元素，又称**检索**，是数据处理中常见的**重要操作**。
- ◆ 了解不同数据结构上的查找方法。
- ◆ 掌握各种**查找结构的性质**、**查找算法的设计思想和实现方法**。
- ◆ 掌握各种查找方法的**时间性能**(平均查找长度)的**分析方法**。
- ◆ 能够根据具体情况选择适合的方法解决实际问题。





# 本章主要内容

- ➔ 5.1 基本概念和术语
- ➔ 5.2 线性查找
- ➔ 5.3 折半（二分）查找
- ➔ 5.4 分块查找
- ➔ 5.5 BST——二叉查找树
- ➔ 5.6 AVL树
- ➔ 5.7 B-树与B<sup>+</sup>树
- ➔ 5.8 散列技术
- ➔ 本章小结





## 5.1 基本概念和术语

- ◆ **查找表：**由同一类型的数据元素(或记录)构成的集合(文件)。
- ◆ **关键字：**可以标识一个记录的某个**数据项**或**数据项组合**。
- ◆ **键值：**关键字的取值。
- ◆ **主关键字：**可以唯一地标识一个记录的关键字。
- ◆ **次关键码：**不能唯一地标识一个记录的关键字。
- ◆ **查找：**在查找表中找出（确定）一个关键字值等于给定值的数据元素（或记录）。
- ◆ **查找结果：**若在查找集合中找到了与给定值相匹配的数据元素记录，则称**查找成功**；否则，称**查找失败**。

学号	姓名	性别	年龄	入学成绩
0001	张亮	男	19	625
0002	张亮	女	28	617
0003	刘楠	女	19	623
...	...	...	...	...





## 5.1 基本概念和术语 (Cont.)

### ► 查找的分类:

- 根据查找方法取决于记录的键值还是记录的存储位置?
  - 基于关键字比较的查找
    - ◆ 顺序查找、折半查找、分块查找、BST&AVL、B-树和B<sup>+</sup>树
  - 基于关键字存储位置的查找
    - ◆ 散列法
- 根据被查找的数据集合存储位置
  - 内查找: 整个查找过程都在内存进行;
  - 外查找: 若查找过程中需要访问外存, 如B树和B<sup>+</sup>树





## 5.1 基本概念和术语 (Cont.)

### ► 查找的分类:

■ 根据查找方法是否改变数据集合?

#### ● 静态查找:

- ◆ 查找+提取数据元素属性信息
- ◆ 被查找的数据集合经查找之后**并不改变**, 就是说, 既不插入新的记录, 也不删除原有记录。

#### ● 动态查找:

- ◆ 查找+ (插入或删除元素)
- ◆ 被查找的数据集合经查找之后**可能改变**, 就是说, 可以插入新的记录, 也可以删除原有记录。





# 5.1 基本概念和术语 (Cont.)

## → 查找表的操作

### ■ Search (k , F) :

- 在数据集合(查找表、文件) F 中查找关键字值等于 k 的数据元素(记录)。若查找成功，则返回包含 k 的记录的位置；否则，返回一个特定的值。

### ■ Insert (R, F) :

- 在动态环境下的插入操作。在 F 中查找记录 R，若查找不成功，则插入 R；否则不插入 R。

### ■ Delete(k, F):

- 在动态环境下的删除操作。在 F 中查找关键字值等于 k 的数据元素(记录)。若查找成功，则删除关键字值等于 k 的记录，否则不删除任何记录。





## 5.1 基本概念和术语（Cont.）

### ► 查找（表）结构：

- 面向查找操作的数据结构，即查找所使用的数据结构。
- 查找结构决定查找方法。
- 主要的查找结构：
- 集合 → 线性表、树表、散列表
  - **线性表**：适用于静态查找，主要采用线性（顺序）查找技术、折半查找技术。
  - **树表**：静态和动态查找均适用，主要采用BST、AVL和B树等查找技术。
  - **散列表**：静态和动态查找均适用，主要采用散列技术。





## 5.1 基本概念和术语（Cont.）

- ◆ 查找表结点（数据元素、记录）的类型定义：

```
struct records{  
    keytype key;  
    fields other; };
```

- ◆ 查找的性能

- 查找算法时间性能由关键字的比较次数来度量。
- 同一查找集合、同一查找算法，关键字的比较次数与哪些因素有关呢？

问题规模*n*、位置*k*

- 查找算法的时间复杂度是问题规模*n*和待查关键字在查找集合中的位置*k*的函数，记为***T(n, k)***。





## 5.1 基本概念和术语 (Cont.)

### ◆ 查找的性能

#### ■ 平均查找长度:

● 把给定值与关键字进行比较的次数的期望值称为查找算法在查找成功时的平均查找长度—**ASL(Average Search Length)**。

● 计算公式: 假设查找集合中的记录个数  $p_i$  为查找表中第  $i$  个记录的概率,  $\sum p_i = 1$ ,  $c_i$  为查找第  $i$  个记录所进行的比较次数, 则

$$ASL = \sum_{i=1}^n p_i c_i$$

● 在等概率情况下, 即  $p_i = 1/n$  时,

$$ASL = \frac{1}{n} \sum_{i=1}^n c_i$$





## 5.2 线性查找

### → 线性（顺序）查找基本思想：

- 从线性表的一端开始，**顺序扫描**线性表，依次将扫描到的结点关键字与给定值K相比较。
- 若当前扫描到的结点关键字与k相等，则**查找成功**；
- 若扫描结束后，仍未找到关键字等于k的结点，则**查找失败**。

### → 线性（顺序）查找对存储结构要求

- 既适用于线性表的**顺序存储结构**----适用于**静态查找**
- 也适用于线性表的**链式存储结构**----也适用于**动态查找**





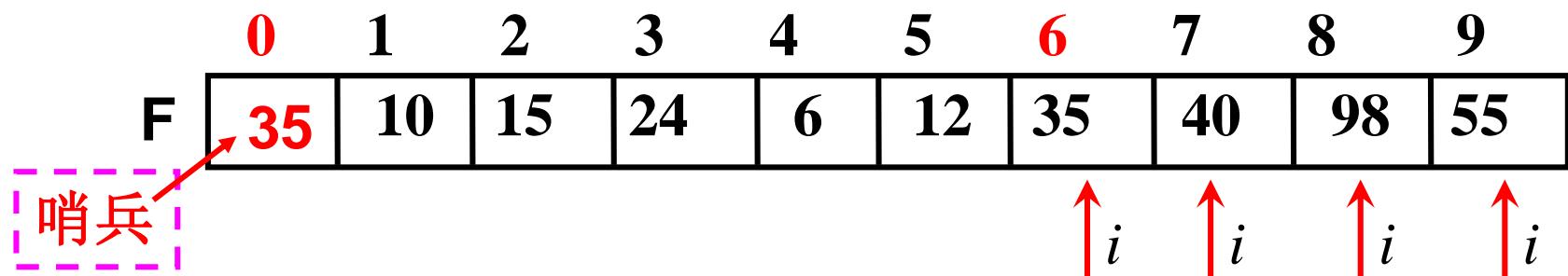
## 5.2 线性查找 (Cont.)

→ 顺序表上的查找——适合于静态查找

■ 顺序表的类型定义

```
typedef records LIST[MaxSize] ;  
LIST F ;
```

■ Search操作的实现: k=35



■ Insert操作的实现  
■ Delete操作的实现 } 不适合顺序表





## 5.2 线性查找 (Cont.)

```
int Search (keytype k, int last, LIST F )  
/* 在F[1]...F[last]中查找关键字为k的记录，若找到，则返回该记录  
所在的下表，否则返回 0 */  
{  int i ;  
    F[0].key = k ; // F[0]为伪记录或哨兵  
    i = last ;  
    while ( F[i].key != k ) //失败需要比较n+1次  
        i = i -1 ;  
    return i ;  
}  
/* 时间复杂度 O( n ) ; ASL成功=(n+1)/2, ASL失败=n+1 */
```





## 5.2 线性查找 (Cont.)

### → 单向表上的查找——也适合于动态查找

#### ■ 单向表的类型定义

```
struct celltype {  
    records data ;  
    celltype * next ;  
};
```

```
typedef celltype *LIST ;
```

#### ■ Insert操作的实现

#### ■ Delete操作的实现

■ 时间复杂度  $O(n)$ ；  $ASL_{\text{成功}} = (n+1)/2$ ,  $ASL_{\text{失败}} = n+1$

```
LIST Search(keytype k, LIST F)  
/*在不带表头的单向链表中查找关  
键字为k 的记录，返回其指针*/  
{   LIST p = F ;  
    while ( p != NULL )  
        if ( p->data.key == k )  
            return p ;  
        else  
            p = p->next ;  
    return p ;  
}
```





## 5.3 折半查找

### ◆ 折半查找（也称二分查找）的要求：

- 查找表（被查找的数据集合）必须采用顺序式存储结构；
- 查找表中的数据元素（记录）必须按关键字有序。

	1	2	3	4	5	6	7	8	9	10	11
F	05	13	19	21	37	56	64	75	80	88	92

- $F[0].key \leq F[1].key \leq F[2].key \leq \dots \leq F[last].key$
- 或  $F[0].key \geq F[1].key \geq F[2].key \geq \dots \geq F[last].key$
- 注意：折半查找只适合于静态查找！

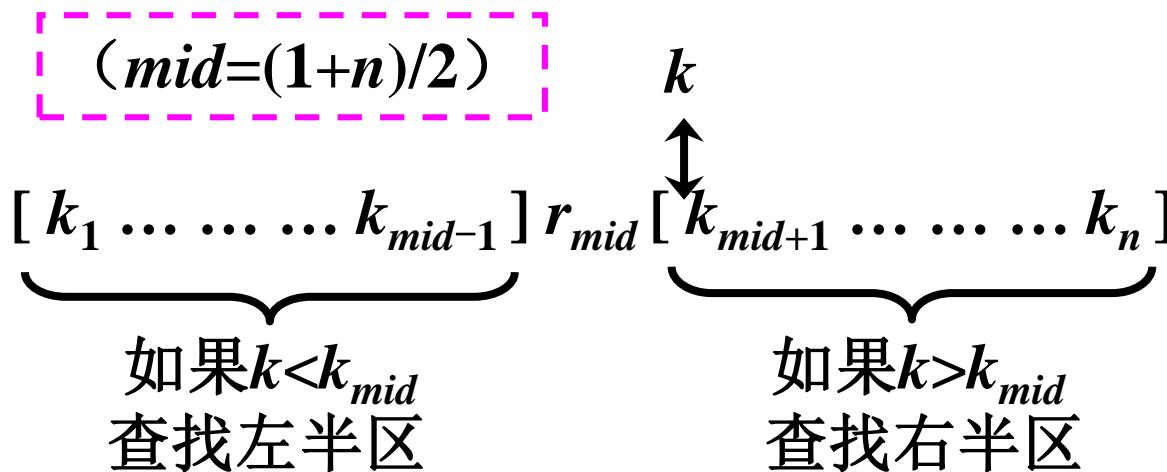




## 5.3 折半查找 (Cont.)

### ◆ 折半查找的基本思想:

- 在有序表中，取中间记录作为比较对象，若给定值与中间记录的关键码相等，则查找成功；若给定值小于中间记录的关键码，则在中间记录的左半区继续查找；若给定值大于中间记录的关键码，则在中间记录的右半区继续查找。不断重复上述过程，直到查找成功，或所查找的区域无记录，查找失败。

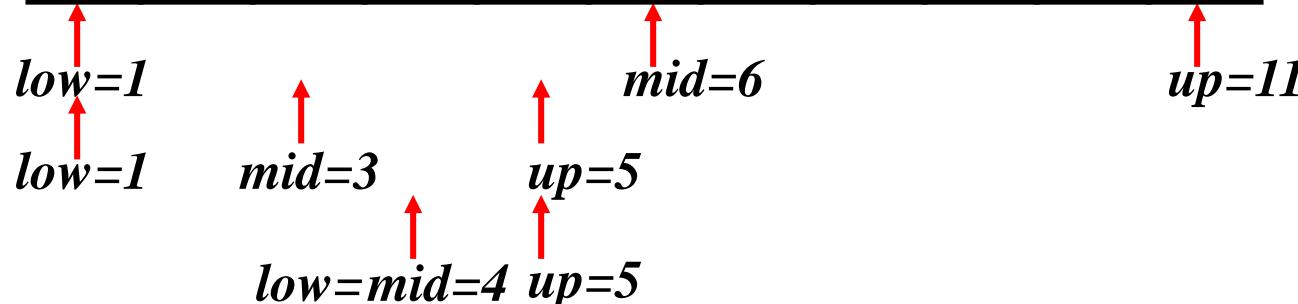




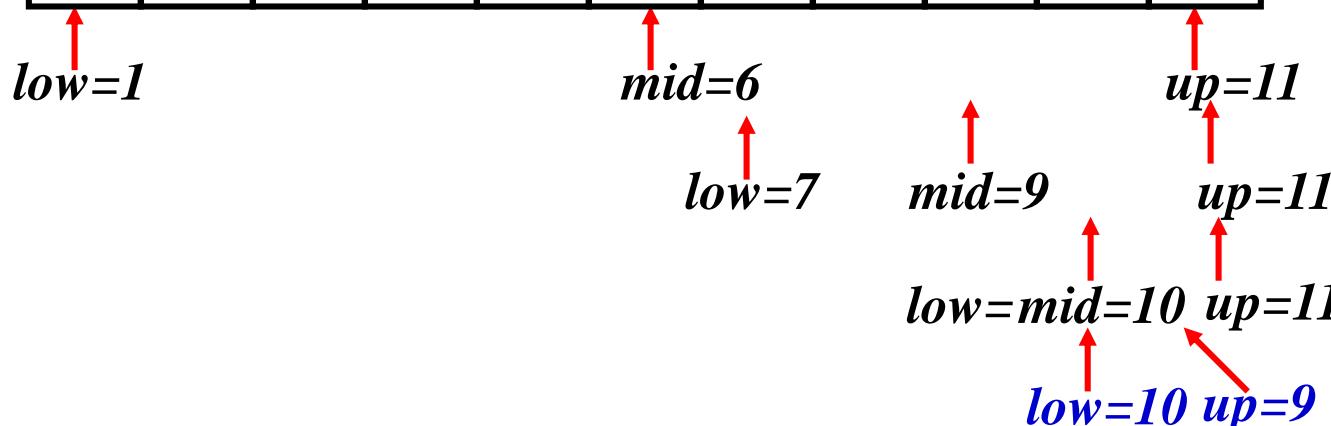
## 5.3 折半查找 (Cont.)

► 折半查找的示例:

$k = 21$	1	2	3	4	5	6	7	8	9	10	11	F
	05	13	19	21	37	56	64	75	80	88	92	



$k = 85$	1	2	3	4	5	6	7	8	9	10	11	F
	05	13	19	21	37	56	64	75	80	88	92	





## 5.3 折半查找（Cont.）

### 折半查找的非递归算法实现步骤

- 1. 初态化：令  $\text{low}$ ,  $\text{up}$  分别表示查找范围的上、下界，初始时  $\text{low} = 0$ ,  $\text{up} = \text{last}$ ;
- 2. 折半：令  $\text{mid} = (\text{low} + \text{up}) / 2$ , 取查找范围中间位置元素下标;
- 3. 比较： $k$  与  $F[\text{mid}].\text{key}$ 
  - 3.1 若  $F[\text{mid}].\text{key} == k$ , 查找成功, 返回  $\text{mid}$ ;
  - 3.2 若  $F[\text{mid}].\text{key} > k$ ,  $\text{low}$  不变, 调整  $\text{up} = \text{mid} - 1$ , 查找范围缩小一半;
  - 3.3 若  $F[\text{mid}].\text{key} < k$ , 调整  $\text{low} = \text{mid} + 1$ ,  $\text{up}$  不变, 查找范围缩小一半;
- 4. 重复 2 ~ 3 步。当  $\text{low} > \text{up}$  时, 查找失败, 返回 0。





## 5.3 折半查找 (Cont.)

### ◆ 折半查找的非递归算法实现

```
int BinSearch1(keytype k, LIST F )
{
    int low , up , mid ;
    low = 1 ; up = last ;
    while ( low <= up ) {
        mid = ( low + up ) / 2 ;
        if ( F[mid].key == k )    return mid ;
        else if (F[mid].key > k)  up = mid - 1 ;
        else                      low = mid + 1 ;
    }
    return 0;
} /* F必须是顺序有序表(此处为增序);时间复杂度 :O(log2 n) */
```





## 5.3 折半查找（Cont.）

### 折半查找的递归算法实现步骤

- 1. 初态化：设置查找范围的上界up和下界low；
- 2. 测试查找范围：如果 $low > up$ ，则查找失败；否则，
- 3. 取查找范围中间位置元素下标令 $mid = (low+up)/2$ ；比较k与 $F[mid].key$ ：
  - 3.1 若 $F[mid].key == k$ ，查找成功，返回 mid；
  - 3.2 若 $F[mid].key > k$ ，递归地在左半部分查找（low不变，调整 $up = mid - 1$ ）；
  - 3.3 若 $F[mid].key < k$ ，递归地在右半部分查找（调整 $low = mid + 1$ ，up不变。）





## 5.3 折半查找（Cont.）

### ◆ 折半查找的递归算法实现

```
int BinSearch2(LIST F, int low, int up, keytype k )
{
    if (low>up) return 0;
    else {
        mid=(low+up)/2;
        if (k < F[mid].key )
            return BinSearch2(F, low, mid-1, k);
        else if (k>F[mid].key)
            return BinSearch2(F, mid+1, up, k);
        else return mid;
    }
} /* F必须是顺序有序表(此处为增序);时间复杂度 :O(log2 n) */
```





## 5.3 折半查找 (Cont.)

需要注意的几个问题：

- (1) 将  $mid = (left + right) / 2$  写成  $mid = left + (right - left) / 2$ , 为了避免  $left + right$  会整型溢出。
- (2) 除以2没必要写成  $>> 1$ , 写成  $/2$  就行, 因为编译器会自动把  $/2$  优化成  $>> 1$ , 写成左移的形式反而会降低代码的可读性。
- (3) 二分查找代码的写法不唯一, 选择一种自己喜欢的就好,

思路很简单, 细节是魔鬼。

细节问题: 1) **mid**加一还是减一

2) **while** 里到底用  $\leq$  还是  $<$ 。

因为初始化  $up$  的赋值是  $last$ , 即最后一个元素的索引,  
[ $low, up$ ] 两端都闭的区间。这个区间其实就是每次进行搜索的  
区间。





## 5.3 折半查找 (Cont.)

有序数组 `nums = [1, 2, 2, 2, 3]`, `target` 为 2, 此算法返回的索引是 2, 没错。如果想得到 `target` 的左侧边界, 即索引 1, 或者想得到 `target` 的右侧边界, 即索引 3, 这样的话此算法是无法处理的。

思考题：修正二分查找算法，可以求出左侧（或右侧）边界





## 5.3 折半查找（Cont.）

### ◆ 折半查找的判定树：

- 折半查找的过程可以用二叉树来描述，树中的每个结点对应有序表中的一个记录，结点的值为该记录在表中的位置。通常称这个描述折半查找过程的二叉树为**折半查找判定树**，简称**判定树**。

### ◆ 折半查找的判定树的构造

- 当 $n=0$ 时，折半查找判定树为空；
- 当 $n>0$ 时，折半查找判定树的根结点是有序表中序号为 $mid=(n+1)/2$ 的记录，根结点的左子树是与有序表 $F[1] \sim F[mid-1]$ 相对应的折半查找判定树，根结点的右子树是与 $F[mid+1] \sim F[n]$ 相对应的折半查找判定树。

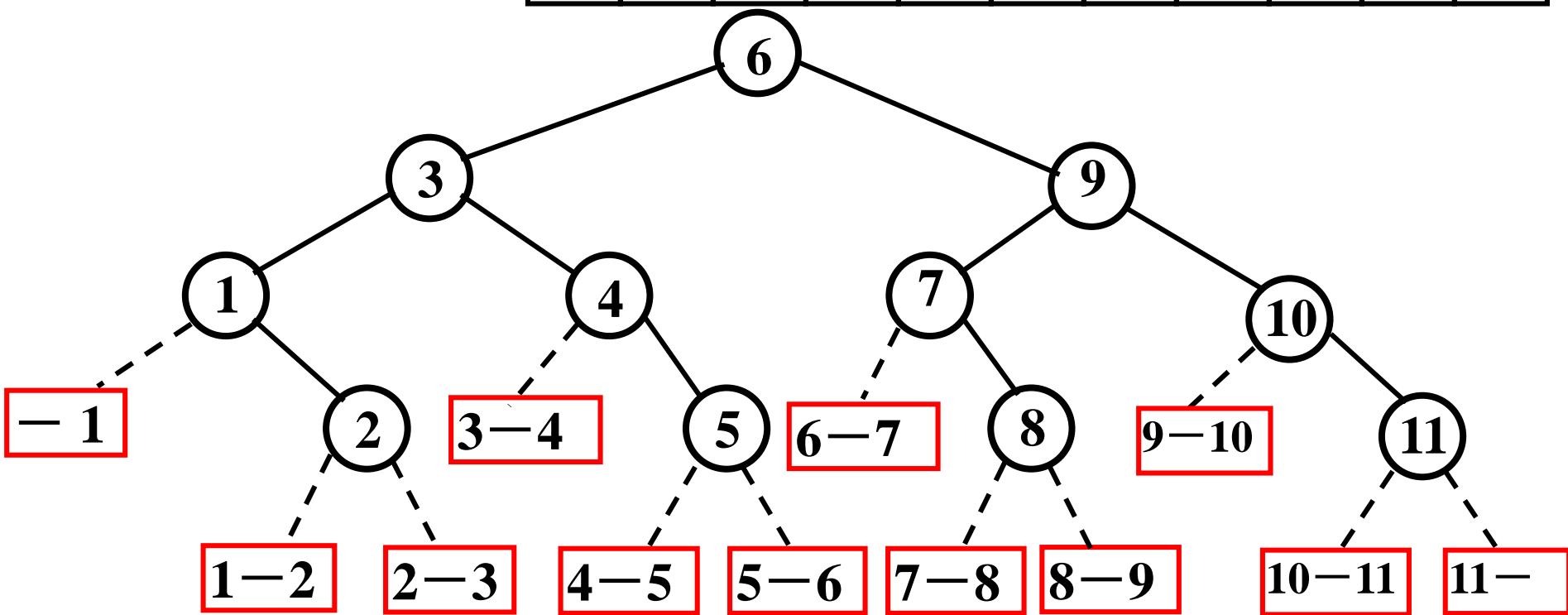




## 5.3 折半查找 (Cont.)

判定树的构造

F	1	2	3	4	5	6	7	8	9	10	11
	05	13	19	21	37	56	64	75	80	88	92



○ 内部结点---查找成功    □ 外部结点---失败结点

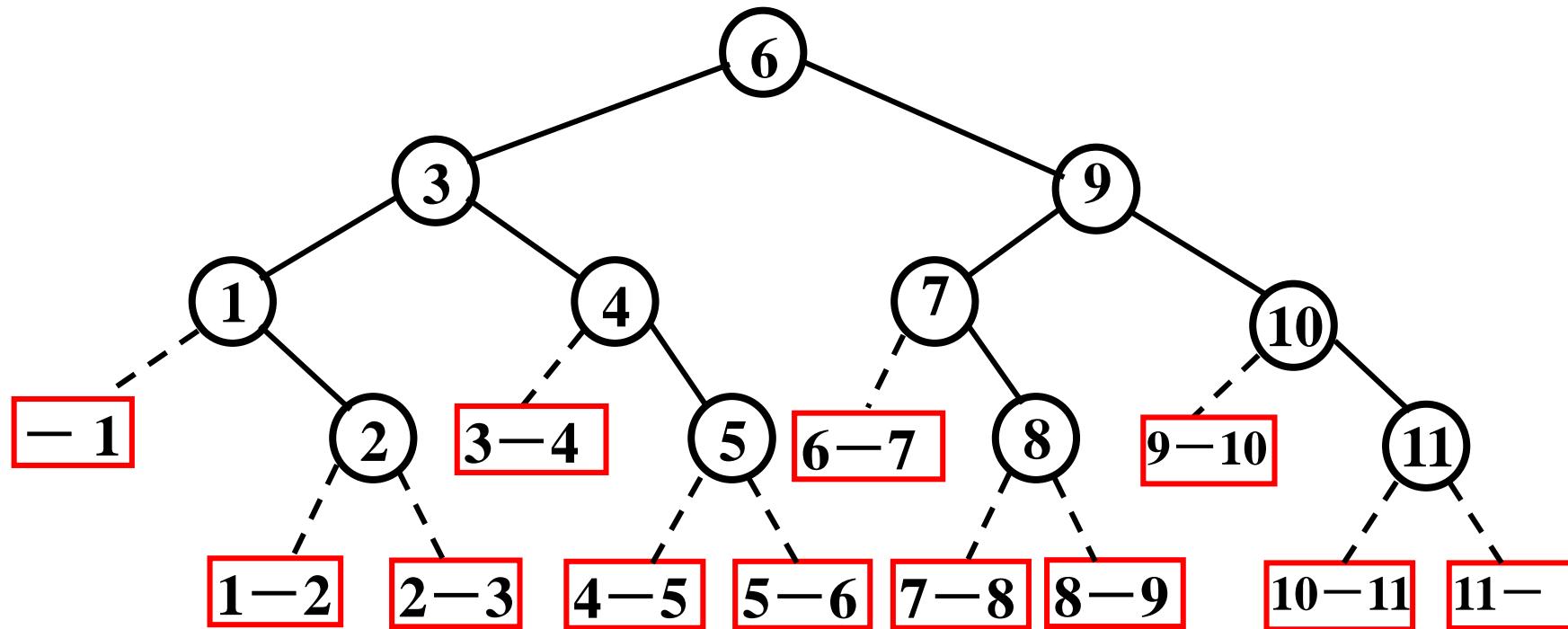




## 5.3 折半查找 (Cont.)

### 折半查找的ASL

- 若有n个关键字，则判定树的失败结点数为n+1个
- $ASL_{\text{成功}} = (1*1 + 2*2 + 3*4 + 4*4) / 11 = 25/11$
- $ASL_{\text{失败}} = (3*4 + 4*8) / 12 = 44/12$





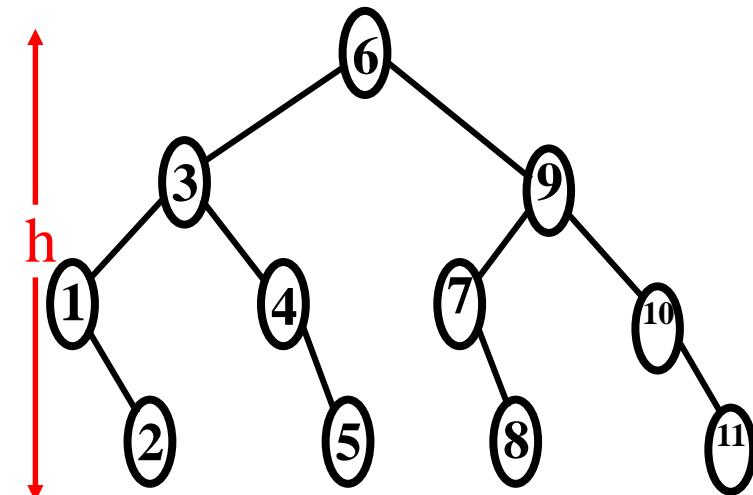
## 5.3 折半查找 (Cont.)

### 折半查找的判定树高度

$$\text{ASL}_{\text{bs}} = \sum_{i=1}^n p_i \cdot c_i \quad /* p_i = 1/n */$$

$$= 1/n \cdot \sum_{j=1}^h j \cdot 2^{j-1}$$

$$= (n+1)/n \log_2(n+1) - 1$$



- 当n很大时,  $\text{ASL}_{\text{bs}} \approx \log_2(n+1)-1$ 作为查找成功时的平均查找长度。
- 在查找不成功和最坏情况下查找成功所需关键字的比较次数都不超过判定树的高度。
- 因为判定树的中度小于2的结点只能出现在下面两层上, 所以n个结点的判定树高度和n个结点的完全二叉树的高度相同, 即 $\lceil \log_2(n+1) \rceil$ 。由此可见, 折半查找的最坏性能与平均性能相当接近。

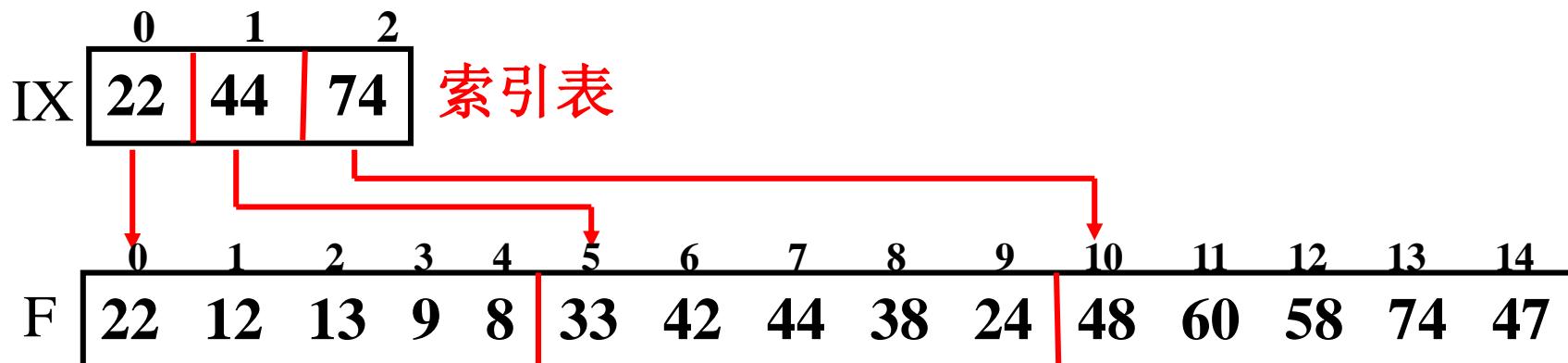




## 5.4 分块查找----线性查找+折半查找

分块查找的基本思想 → 快速排序

- 均匀分块，块间有序，块内无序：首先将表中的元素均匀地分成若干块，每一块中的元素的任意排列，而各块之间要按顺序排列；
  - 若按从小到大的顺序排列，则第一块中的所有元素的关键字都小于第二块中的所有元素的关键字，第二块中的所有元素的关键字都小于第三块中的所有元素的关键字，如此等等。
- 建块索引：然后再建一个线性表，用以存放每块中最大(或最小)的关键字，此线性表称为索引表，它是一个有序表。





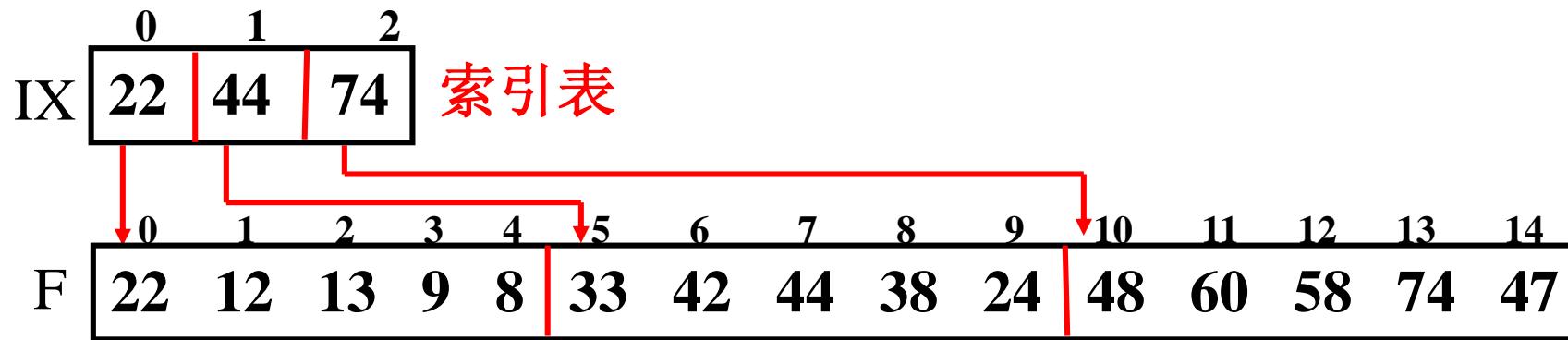
## 5.4 分块查找 (Cont.)

- ◆ 分块查找算法的要点：在线性表中查找已知关键字为k 的记录，则
  - 首先查找索引表，确定k 可能出现的块号；
  - 然后到此块中进行顺序查找。

### ◆ 算法的实现

```
typedef records LIST[maxsize] ;// 线性表—主表
```

```
typedef keytype INDEX[maxblock] ;// 线性表—索引表
```





## 5.4 分块查找 (Cont.)

```
int index_search(keytype k, int last, int blocks, INDEX ix, LIST F, int L )
{ int i =0, j ;
  while (( k > ix[i])&&( i < blocks)) //查索引表,确定k 所在块i
    i++ ;
  if( i<blocks ) {
    j = i*L; // 第i 块的起始下标
    while(( k != F[j].key )&&( j <= (i+1)*L-1 )&&( j < last ))
      j = j + 1 ;
    if ( k == F[ j ].key ) return j ; // 查找成功
  }
  return -1 ; /* 查找失败 */
}
```





## 5.4 分块查找 (Cont.)

### 分块查找性能分析

- 设长度为n 的表分成b 块，每块长度为L，则 $b = \lceil n / L \rceil$ ，又设表中每个元素的查找概率相等，则每块查找的概率为 $1/b$ ，块中每个元素的查找概率为 $1/L$ 。于是，
- 索引表的 $ASL_{ix} = \sum_{i=1}^b p_i \cdot c_i = \frac{1}{b} \sum_{i=1}^b i$

- 块内的平均查找长度： $ASL_{blk} = \sum_{j=1}^L p_i \cdot c_i = \frac{1}{L} \sum_{j=1}^L j$
- 所以分块查找平均长度为：

$$ASL(L) = ASL_{ix} + ASL_{blk} = (b+1)/2 + (L+1)/2 = (n/L + L)/2 + 1$$

可证明，当 $L=\sqrt{n}$ 时， $ASL(L) = \sqrt{n} + 1$ （最小值）。





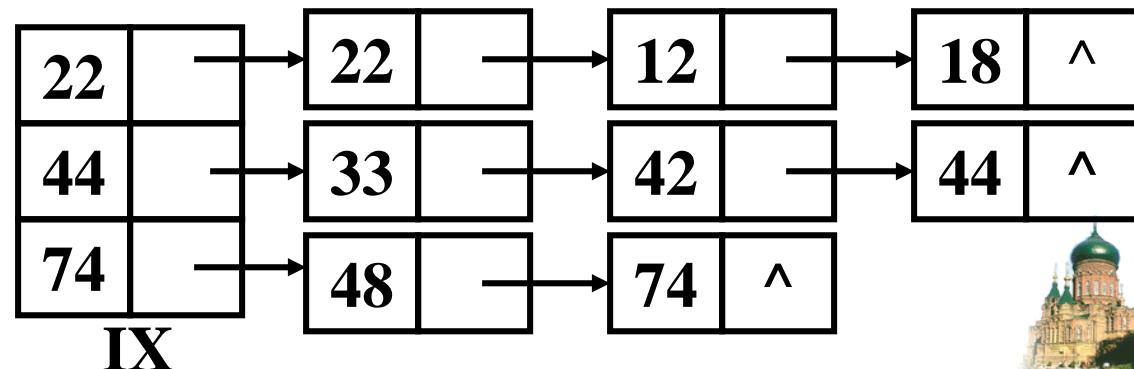
## 5.4 分块查找 (Cont.)

### 分块查找局限性和改进

- 只适合静态查找；
- 改进：
  - 在索引表中保存各块的下标范围，此时不必均匀分块
  - 各块存放在不同的向量（一维数组）中；
  - 把同一块中的元素组织成一个链表。

### 动态环境的分块查找

- 带索引表的链表
- 算法的实现
  - 数据结构定义
  - 三个算法的实现





### 分块查找的优点

①在表中插入或删除一个记录时，只要找到该记录所属的块，就在该块内进行插入和删除运算。

②因块内记录的存放是任意的，所以插入或删除比较容易，无须移动大量记录。

分块查找的主要代价是增加一个辅助数组的存储空间和将初始表分块排序的运算。





## 5.5 二叉查找树BST

判断：①中序遍历有序

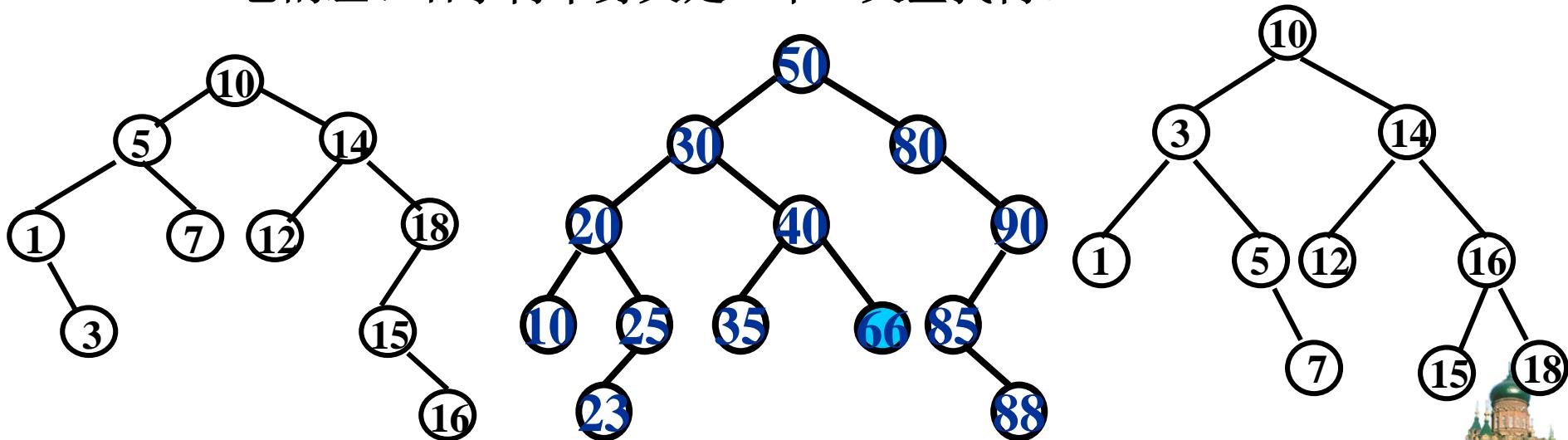
②左边 < 根

右边 > 根

→ 二叉查找树——二叉搜索树、二叉分类（排序）树

■ 二叉查找树或者是空树，或者是满足下列性质的二叉树：

- 若它的左子树不空，则左子树上所有结点的关键字的值都小于根结点关键字的值；
- 若它的右子树不空，则右子树上所有结点的关键字的值都大于根结点关键字的值；
- 它的左、右子树本身又是一个二叉查找树。

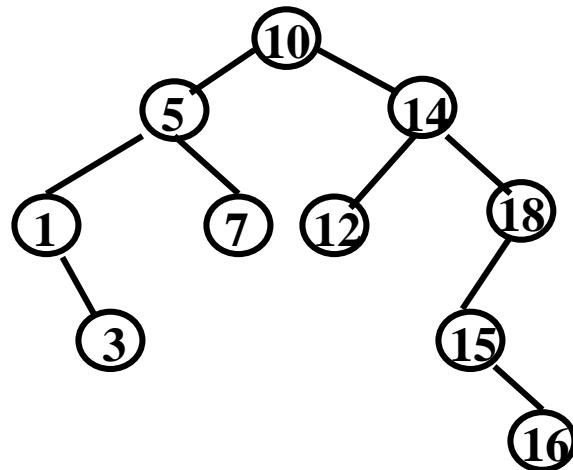




## 5.5 二叉查找树BST (Cont.)

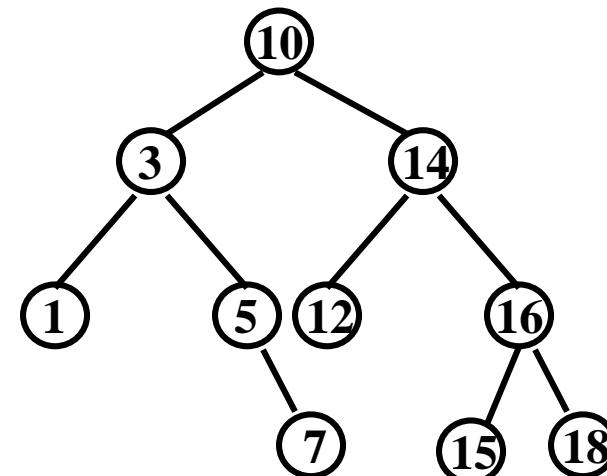
### 二叉查找树的结构特点:

- 任意一个结点的关键字，都大于(小于)其左(右)子树中任意结点的关键字，因此各结点的关键字互不相同
- 按中序遍历二叉查找树所得的中序序列是一个递增的有序序列，因此，二叉查找树可以把无序序列变为有序序列。
- 同一个数据集合，可按关键字表示成不同的二叉查找树，即同一数据集合的二叉查找树不唯一；但中序序列相同。



相同数据：

- ① 加权值  
② 加链  
③ ?? ≤





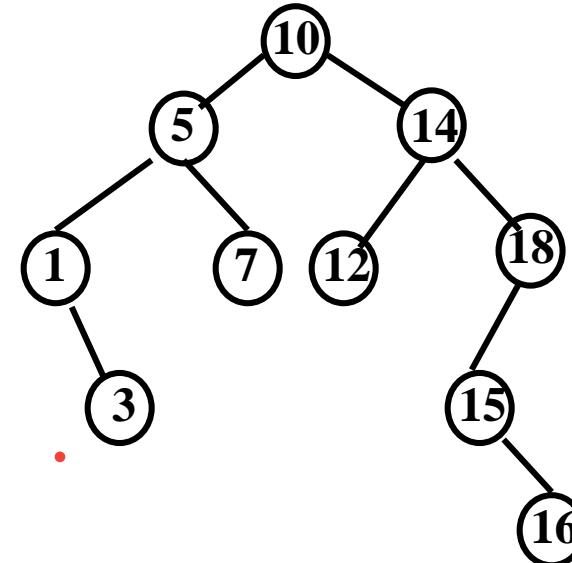
## 5.5 二叉查找树BST (Cont.)

### 二叉查找树的结构特点:

- 每个结点X的右子树的最左结点Y，称为X的继承结点。有如下性质：
  - (1) 在此右子树中，其关键字值最小，但大于X的关键字
  - (2) 最多有一个右子树，即没有左子树。

### 二叉查找树的存储结构:

```
typedef struct celltype {  
    records data ;  
    struct celltype *lchild,*rchild ;  
} BSTNode;  
  
typedef BSTNode * BST ;
```



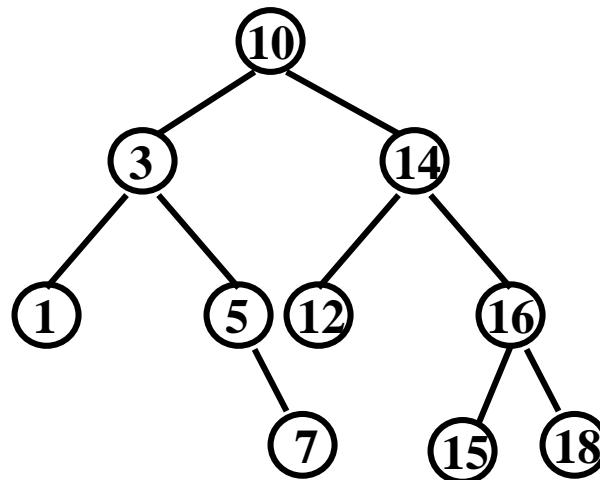


## 5.5 二叉查找树BST (Cont.)

### 二叉查找树的查找操作:

在F 中查找关键字为k 的记录如下:

- 若F = Null, 则查找失败; 否则,
- k ==F->data.key, 则查找成功; 否则,
- k < F->data.key, 则递归地在F 的左子树查找k; 否则
- k > F->data.key, 则递归地在F 的右子树查找k。





## 5.5 二叉查找树BST (Cont.)

➔ 二叉查找树的查找操作:

```
BSTNode * SearchBST( keytype k, BST F )  
{   BSTNode * p = F ;  
    if ( p == Null || k == p->data.key ) // 递归终止条件  
        return p;  
    if ( k < p->data.key )  
        return ( SearchBST ( k, p->lchild ) ); // 查找左子树  
    else  
        return ( SearchBST ( k, p->rchild ) ); // 查找右子树  
}
```





## 5.5 二叉查找树BST (Cont.)

### 二叉查找树的插入操作

- 若二叉排序树为空树，则新插入的结点为根结点；
  - 否则，新插入的结点必为一个新的叶结点。
- 新插入的结点一定是查找不成功时，查找路径上最后一个结点的左儿子或右儿子。

```
void InsertBST(records R, BST &F)
{
    if ( F == Null ) {
        F = new BSTNode ;
        F->data = R ;
        F->lchild = Null ;
        F->rchild = Null ;
    }else if ( R.key < F->data.key )
        InsertBST( R , F->lchild );
    else if ( R.key > F->data.key )
        InsertBST( R , F->rchild );
}
```

//若R.key==F->data.key,则返回





```
void insert_BST(BST &F, records R)
{
    BST p = new celltype;
    p->data = R;
    p->lchild = Null;
    p->rchild = Null;
    while(F!=Null)
    {
        if(R.key < F->data.key)
            if(F->lchild) F =F->lchild;
            else {F->lchild=p; return;}
        else
            if(F->rchild) F =F->rchild;
            else {F->rchild=p; return;}
    }
    if(F == Null)
        F = p;
    return;
}
```





## 5.5 二叉查找树BST (Cont.)

### → 二叉查找树的建立

BST **CreateBST** ( void )

```
{ BST F = Null; //初始时F为空  
keytype key;  
cin>>key>>其他字段;//读入一个记录  
while( key ){ //假设key=0是输入结束标志  
    InsertBST( R , F ); // 插入记录R  
    cin>>key>>其他字段 ;//读入下个记录  
}  
return F;//返回建立的二叉查找树的根  
}
```

注意：在建立二叉查找树时，若按关键字有序顺序输入各记录，则产生退化的二叉查找树—单链表

### 如何防止？

- ✓ 随机输入各结点
- ✓ 在建立、插入和删除各结点过程中平衡相关结点的左、右子树。



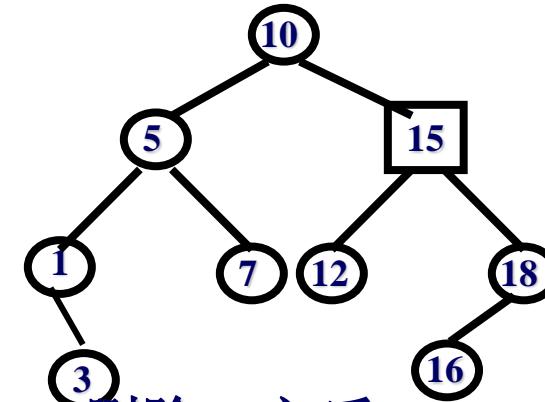
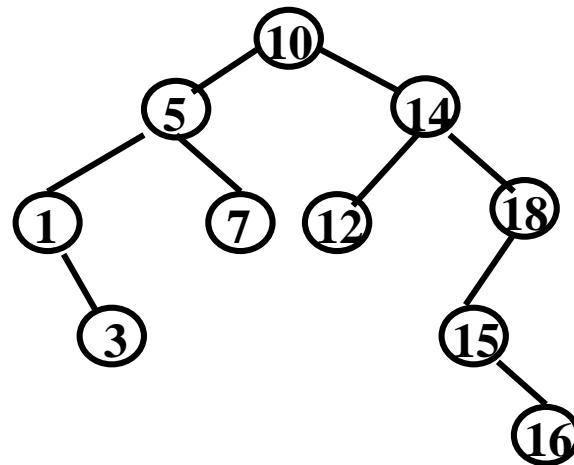


## 5.5 二叉查找树BST (Cont.)

### 二叉查找树的删除操作

删除某结点，并保持二叉排序树特性，分三种情况处理：

- 1) 如果删除的是叶结点，则直接删除；
- 2) 如果删除的结点只有一株左子树或右子树，则直接继承：将该子树移到被删结点位置；
- 3) 如果删除的结点有两株子树，则用继承结点代替被删结点，这相当于删除继承结点——按 1) 或 2) 处理继承结点。



删除14之后

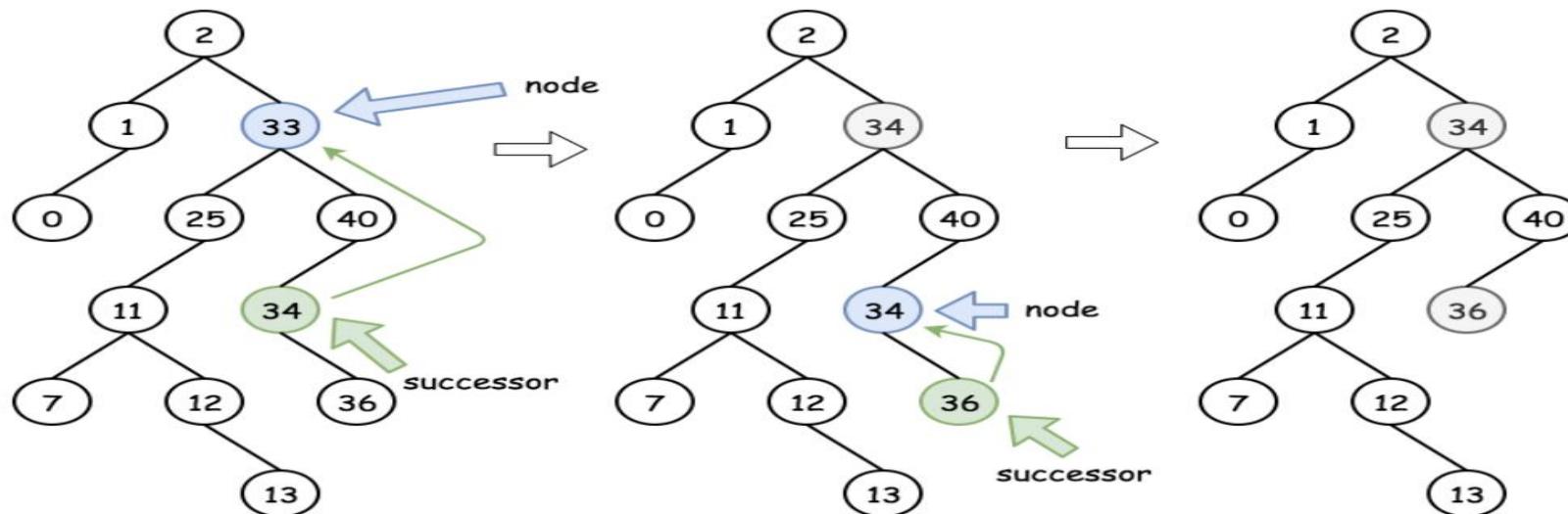




## 5.5 二叉查找树BST (Cont.)

### → 二叉查找树的删除操作的实现步骤

1. 若结点p是叶子，则直接删除结点p；
2. 若结点p只有左子树，则p的左子树继承；若结点p只有右子树，则p的右子树继承；
3. 若结点p的左右子树均不空，则
  - 3.1 查找结点p的右子树上的最左下结点s及其双亲结点par；
  - 3.2 将结点s数据域替换到被删结点p的数据域；
  - 3.3 将s的右子树接到结点par的左子树上；
  - 3.4 删除结点s；



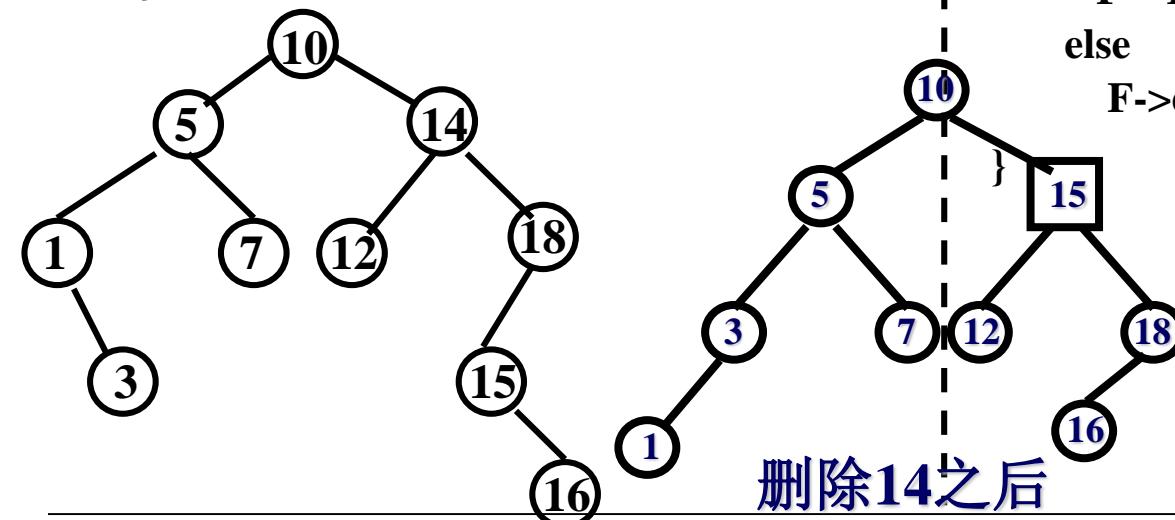


## 5.5 二叉查找树BST (Cont.)

### 二叉查找树的删除操作的实现

```
records deletemin(BST &F)
{ records tmp ; BST p ;
  if ( F->lchild == Null) {
    p = F ;
    tmp = F->data ;
    F = F->rchild ;
    delete p ;
    return tmp ;
  }
  else
    return(deletemin( F->lchild)) ;
}
```

```
void DeleteB( keytype k, BST &F )
{ if ( F != Null)
  if ( k < F->data.key )
    DeleteB( k, f->lchild ) ;
  else if ( k > F->data.key )
    DeleteB( k, f->rchild ) ;
  else
    if ( F->rchild == Null )
      F = F->lchild ;
    else if( F->lchild == Null )
      F = F->rchild ;
    else
      F->data =deletemin(F->rchild);
```





## 5.5 二叉查找树BST (Cont.)

### 二叉查找树的查找性能

- 二叉排序树的查找性能取决于二叉排序树的形态，在 $O(\log_2 n)$  和  $O(n)$  之间。
- 在**最坏情况下**，二叉查找树是通过把有序表的n 个结点依次插入而生成的，此时所得到的二叉查找树退化为一株高度为n 的单支树，它的平均查找长度和单链表上的顺序查找相同， $(n+1)/2$ 。
- 在**最好情况下**，二叉查找树的形态比较均匀，最终得到一株形态与折半查找的判定树相似，此时的平均查找长度约为 $\log_2 n$ 。
- 二叉查找树的平均高度为 $O(\log_2 n)$ 。因此**平均情况下**，三种操作的平均时间复杂性为 $O(\log_2 n)$
- 就**平均性能**而言，二叉查找树上的查找与二分查找差不多
- 就**维护表的有序性**而言，二叉查找树更有效。





## 5.6 AVL树

1962年, Adelson-Velskii和Landis提出的

为了保证树的高度为 $\log n$ , 从而保证二叉查找树上实现的插入、删除和查找等基本操作的平均时间为 $O(\log n)$ , 在树中插入或删除结点时, 要调整树的形态来保持树的平衡。使之既保持BST性质不变又保证树的高度在任何情况下均为 $O(\log n)$ , 从而确保树上的基本操作在最坏情况下的时间均为 $O(\log n)$ 。

### ◆ AVL树 (Balanced Binary Tree or Height-Balanced Tree)

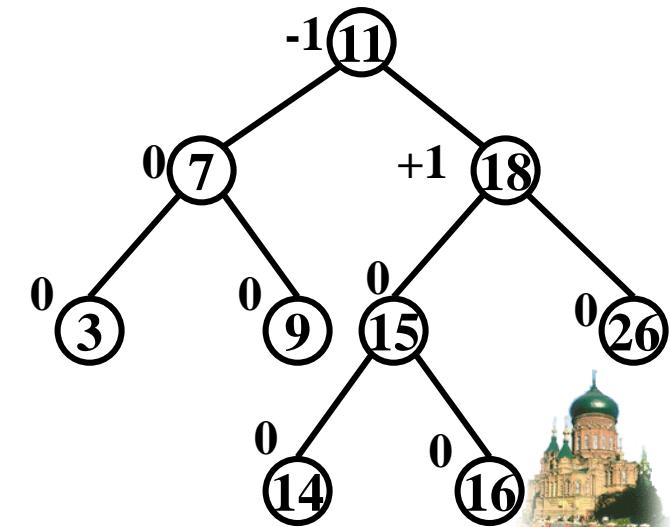
- AVL树或者是空二叉树, 或者是具有如下性质的**BST**:
  - 根结点的左、右子树高度之差的绝对值不超过1;
  - 且根结点左子树和右子树仍然是AVL树。

### ◆ 结点的平衡因子BF (Balanced Factor)

- 一个结点的左子树与右子树的高度之差。
- AVL树中的任意结点的BF只可能是-1, 0和1。
- AVL树的ASL可保持在 $O(\log_2 n)$

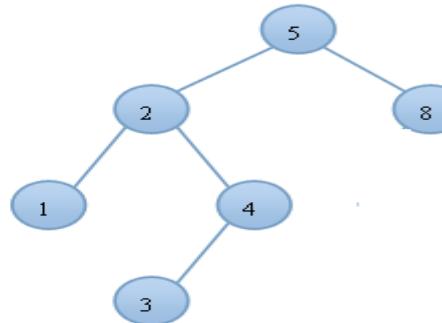
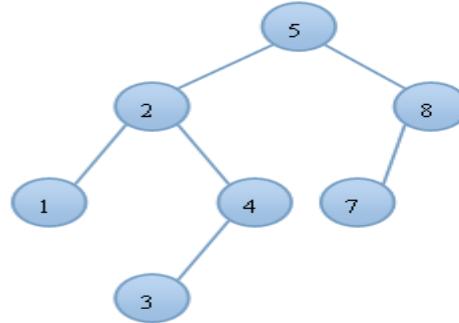
### ◆ AVL树的查找操作

- 与BST的相同



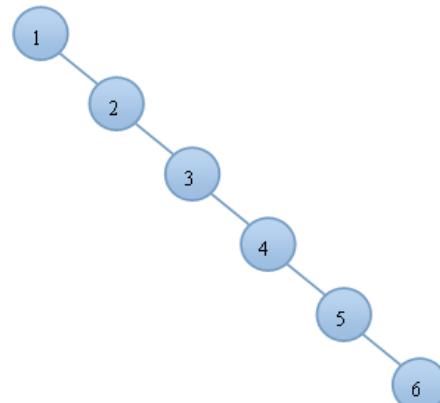


## 5.6 AVL树

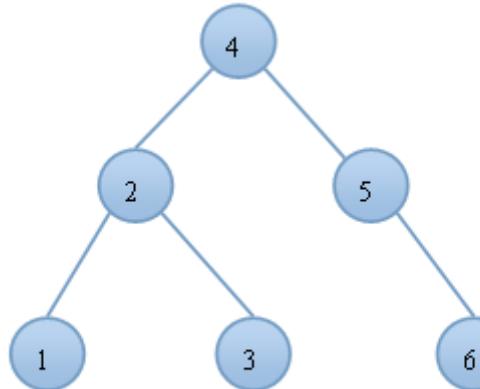


两颗二叉查找树，只有左边的树是AVL树

例如：按顺序将一组数据1, 2, 3, 4, 5, 6分别插入到一颗空二叉查找树和AVL树中，插入的结果如下图：



插入到二叉查找树后的结果



插入到AVL树的结果

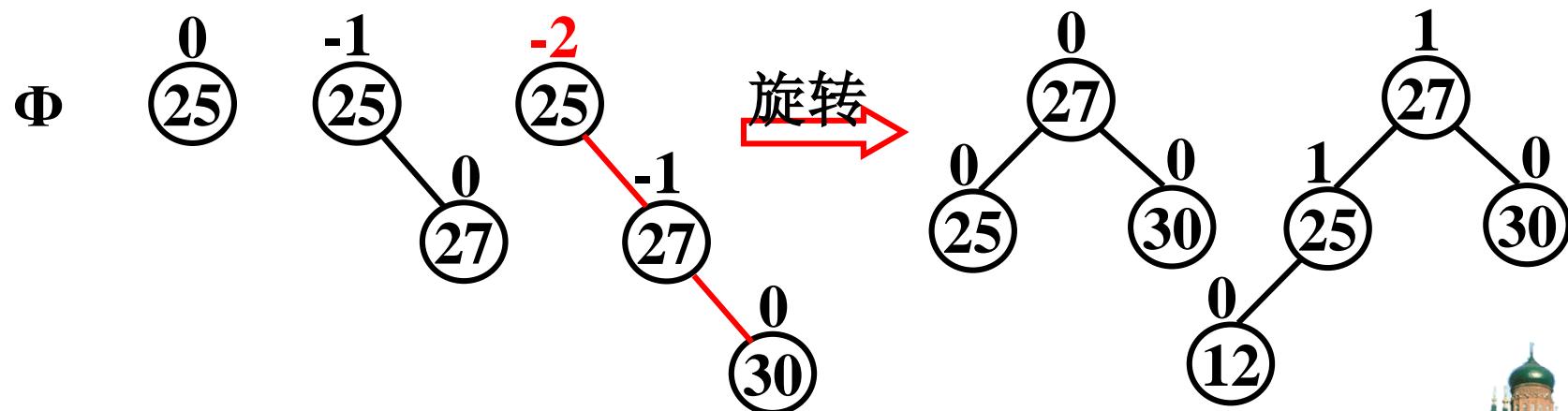




## 5.6 AVL树 (Cont.)

### → AVL树的平衡化处理

- 向AVL树插入结点可能造成不平衡，此时要调整树的结构，使之重新达到平衡
- 希望任何初始序列构成的二叉树都是AVL树
- 示例：假设25, 27, 30, 12, 11, 18, 14, 20, 15, 22是一关键字序列，并以上述顺序建立AVL树。

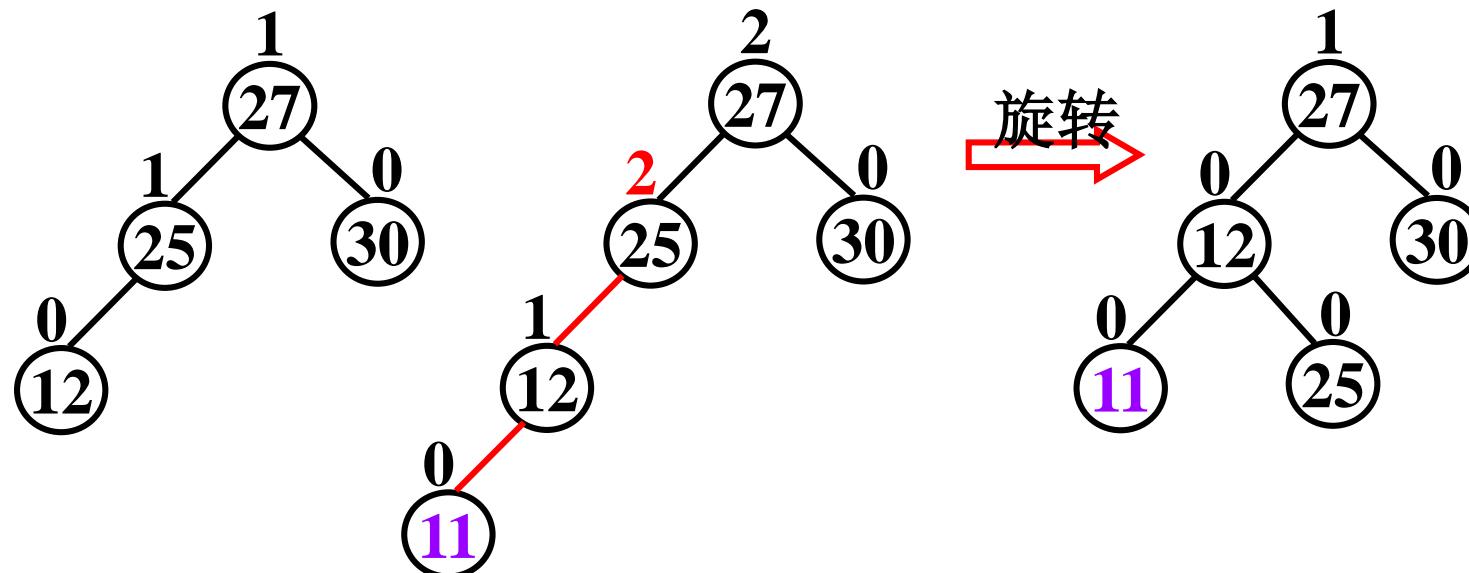




## 5.6 AVL树 (Cont.)

### → AVL树的平衡化处理

- 示例：假设 $25, 27, 30, 12, 11, 18, 14, 20, 15, 22$ 是一关键字序列，并以上述顺序建立AVL树。

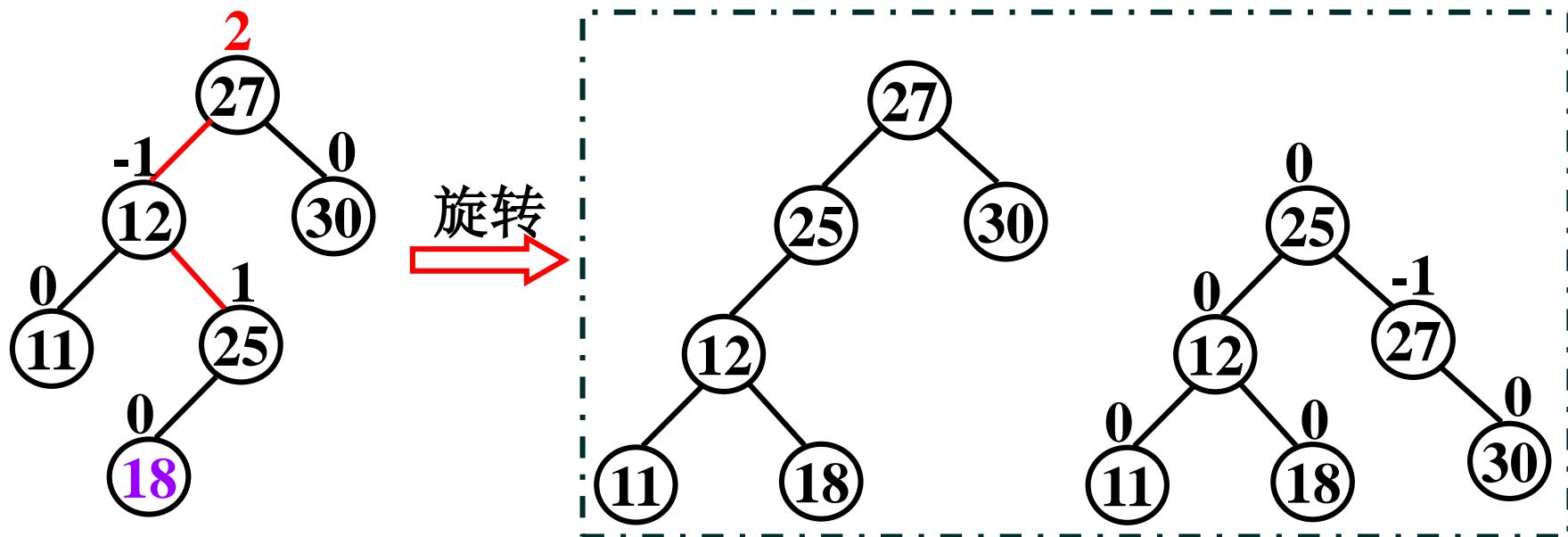




## 5.6 AVL树 (Cont.)

### AVL树的平衡化处理

- 示例：假设25, 27, 30, 12, 11, 18, 14, 20, 15, 22是一关键字序列，并以上述顺序建立AVL树。

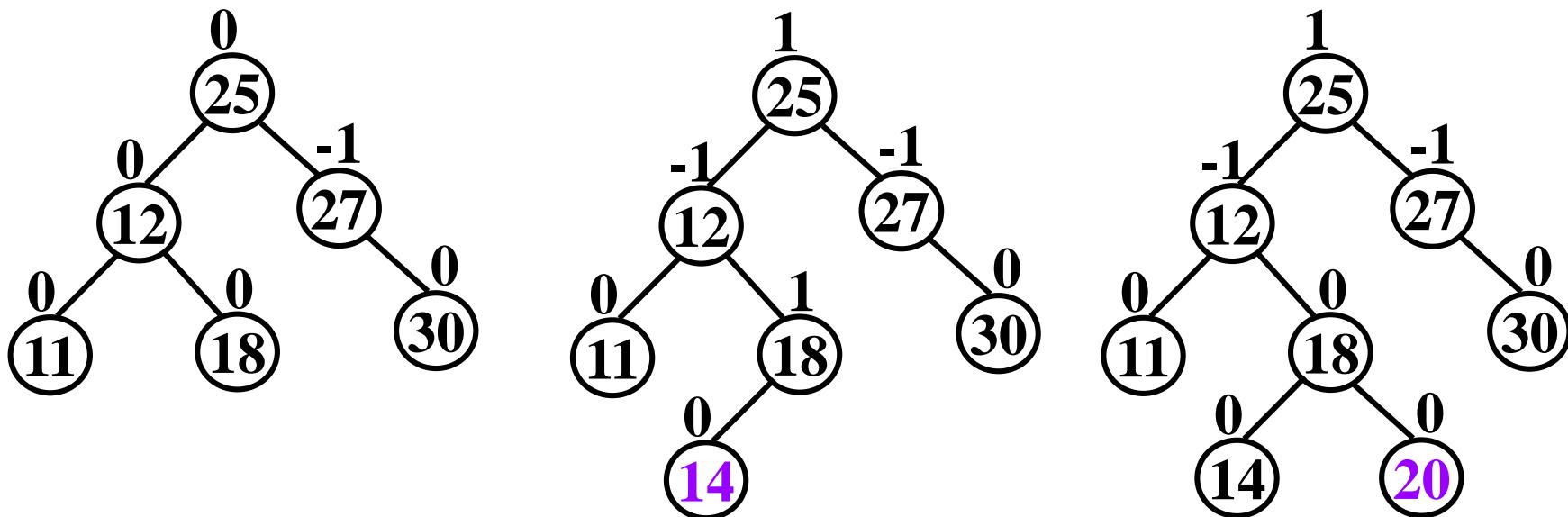




## 5.6 AVL树 (Cont.)

### → AVL树的平衡化处理

- 示例：假设25, 27, 30, 12, 11, 18, 14, 20, 15, 22是一关键字序列，并以上述顺序建立AVL树。

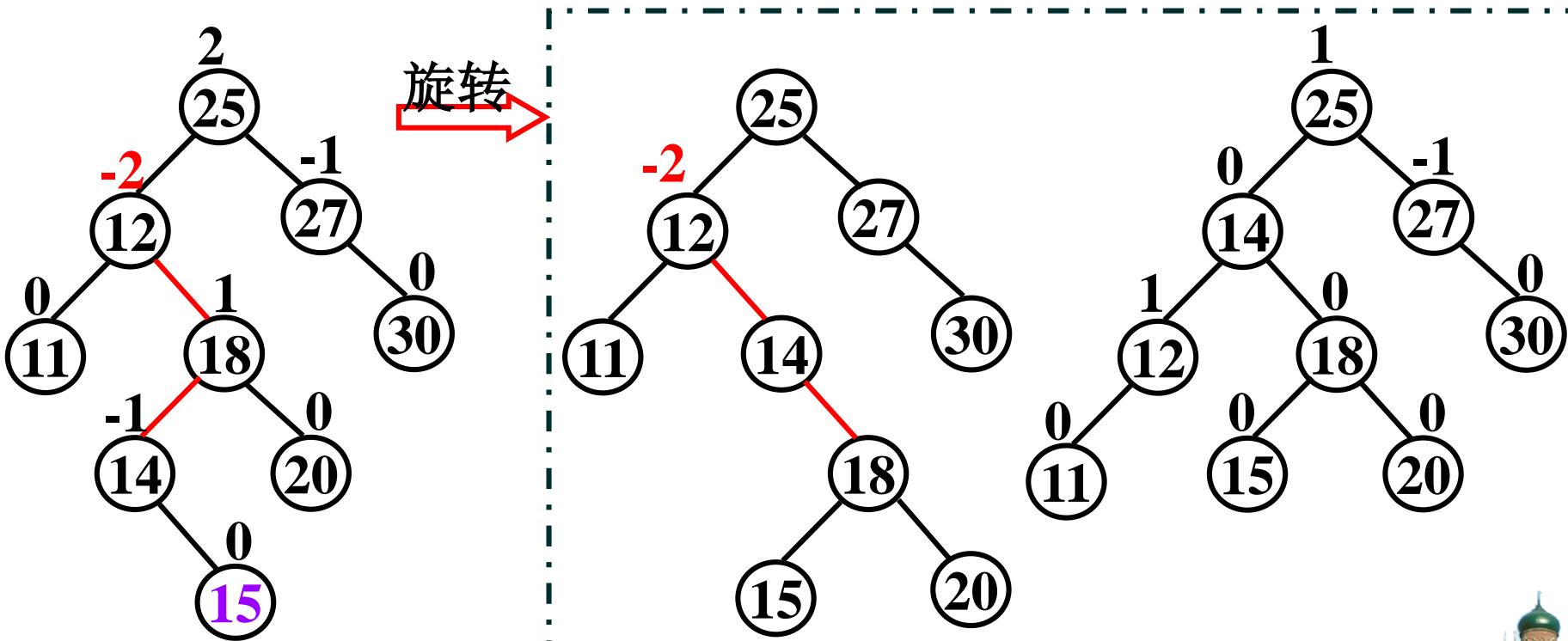




## 5.6 AVL树 (Cont.)

### → AVL树的平衡化处理

- 示例：假设25, 27, 30, 12, 11, 18, 14, 20, 15, 22是一关键字序列，并以上述顺序建立AVL树。

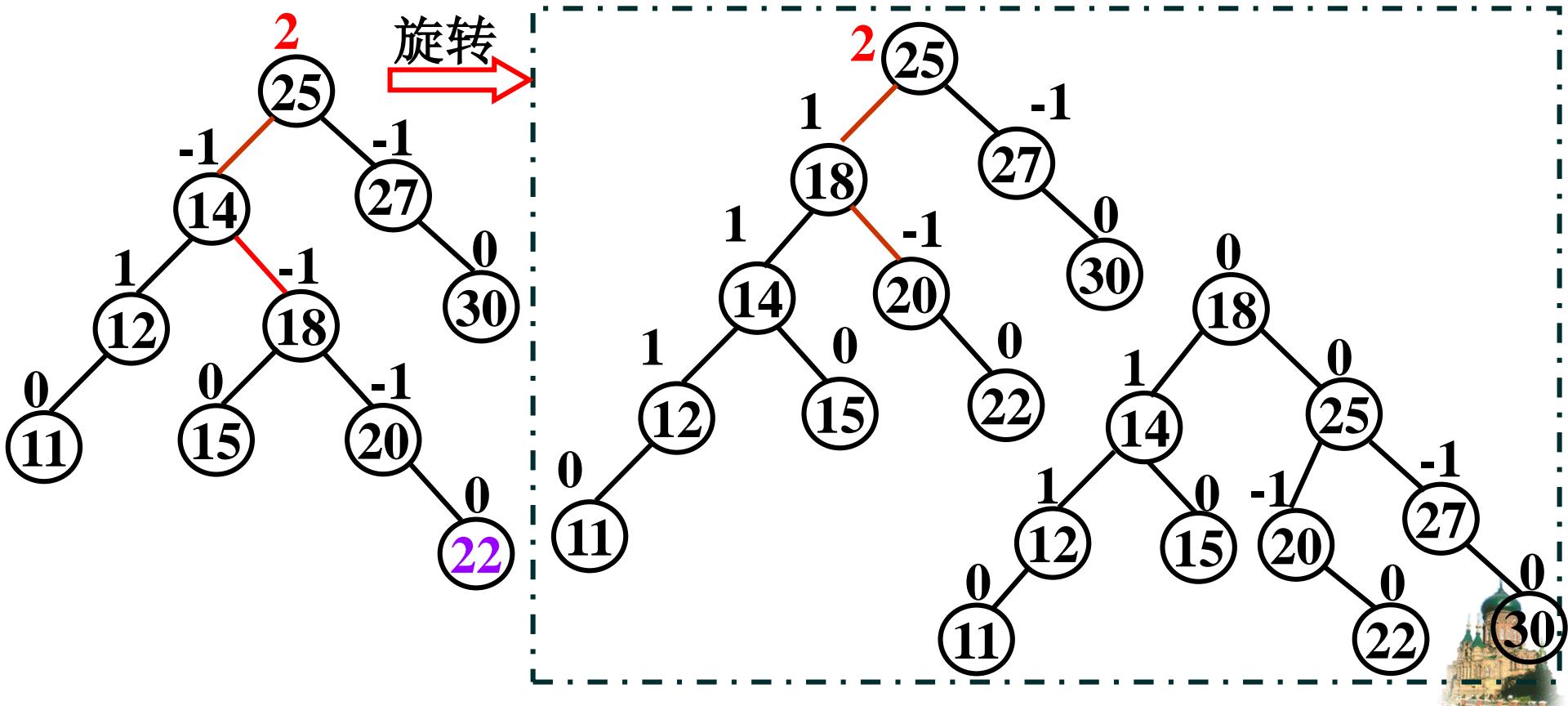




## 5.6 AVL树 (Cont.)

### AVL树的平衡化处理

- 示例：假设25, 27, 30, 12, 11, 18, 14, 20, 15, 22是一关键字序列，并以上述顺序建立AVL树。

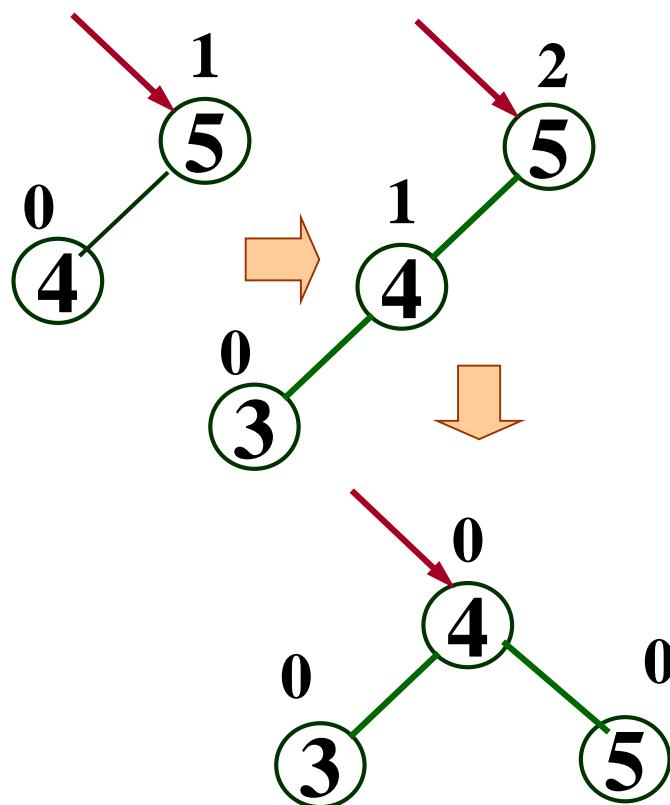




## AVL树的平衡化处理

在插入过程中，采用平衡旋转技术。

例如：依次插入的关键字为 5, 4, 3



(1) 插入结点3使得树不再平衡，而3是插入在失去平衡的最小子树根节点5的左子树4的左子树上。

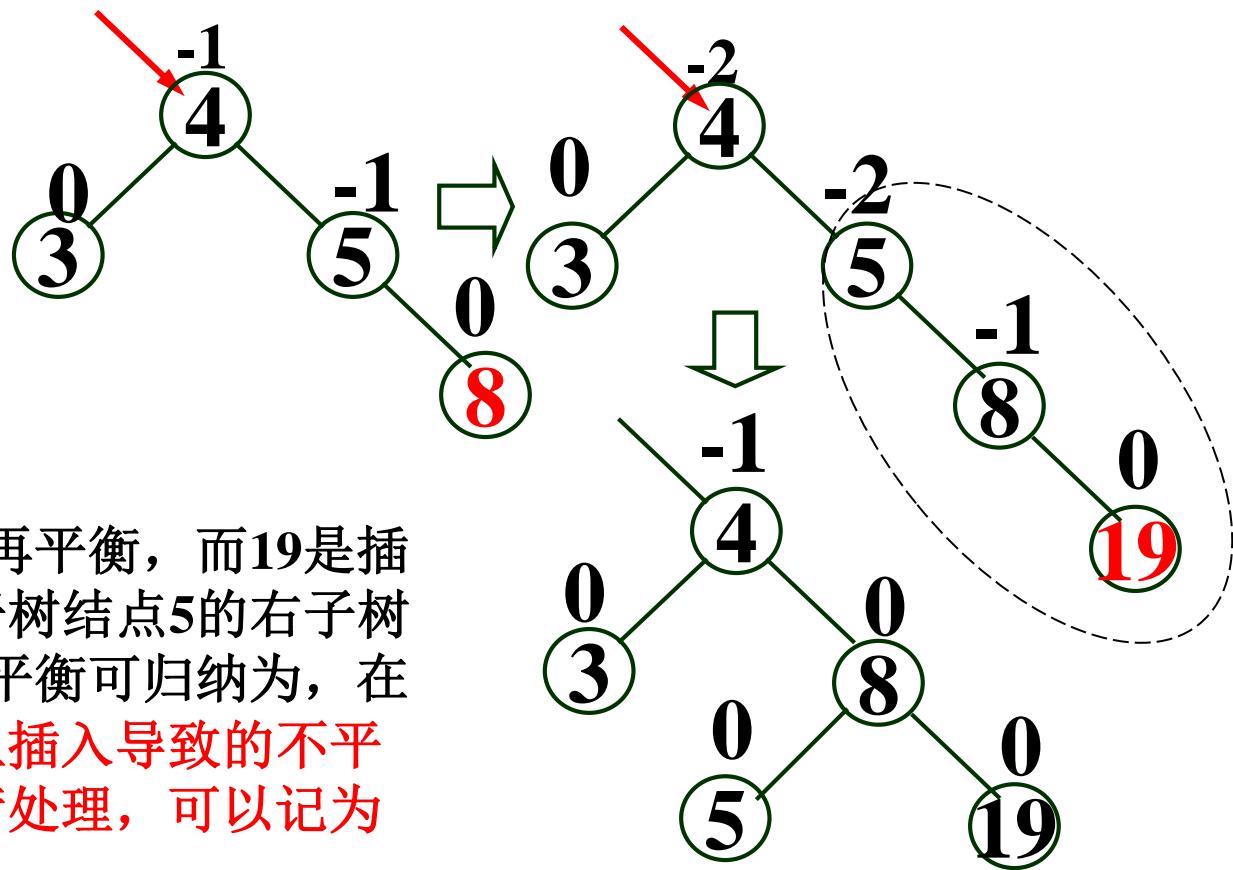
在左子树的左子树上插入导致的不平衡可使用**单向右旋平衡处理**，可以记为左左->右。

① LL型：新结点Y被插入到A的左子树的左子树上（顺）





继续插入的关键字为 5, 4, 3, 8, 19



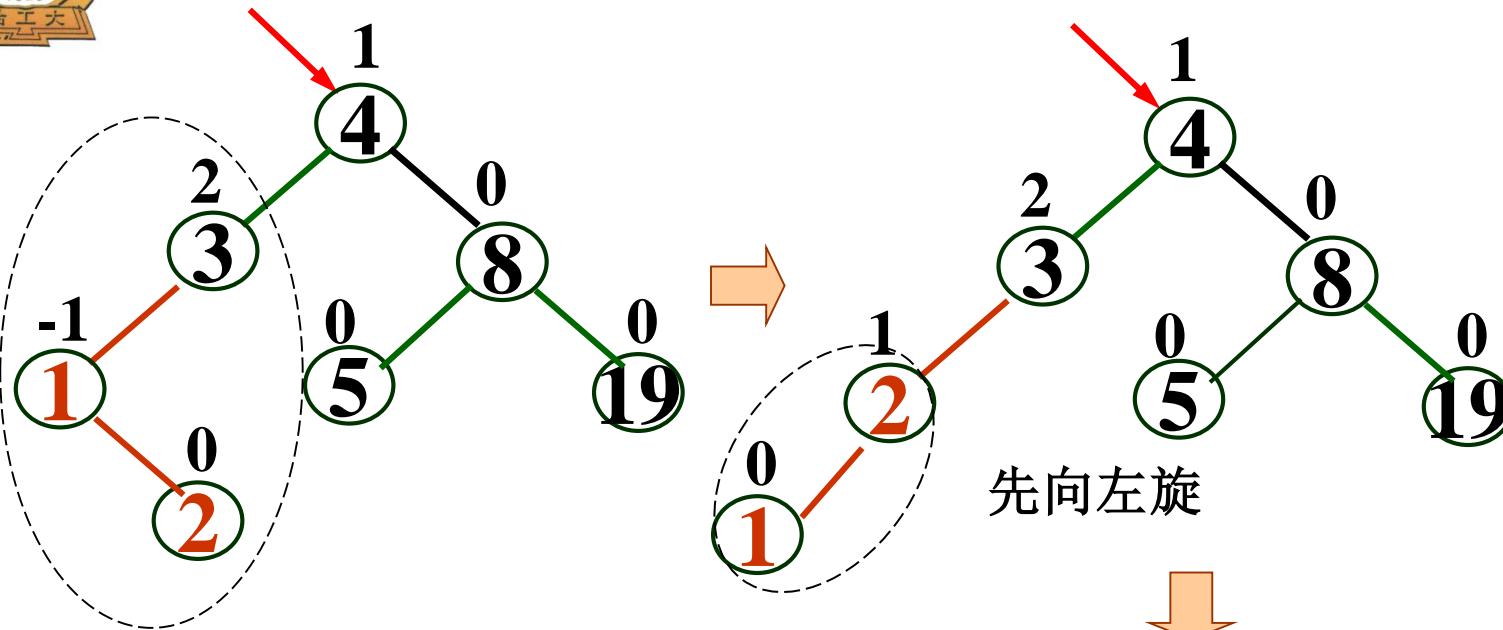
(2) 插入19使得树不再平衡，而19是插入在失去平衡的最小子结点5的右子树8的右子树上。该类不平衡可归纳为，在结点右子树的右子树上插入导致的不平衡可使用单向左旋平衡处理，可以记为右右->左。

② RR型：新结点Y被插入到A的右子树的右子树上（逆）



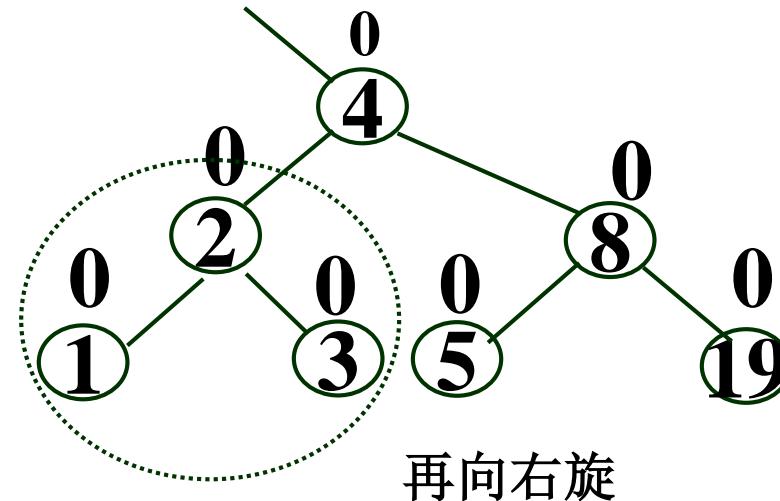


继续插入的关键字为5, 4, 3, 8, 19, 1, 2



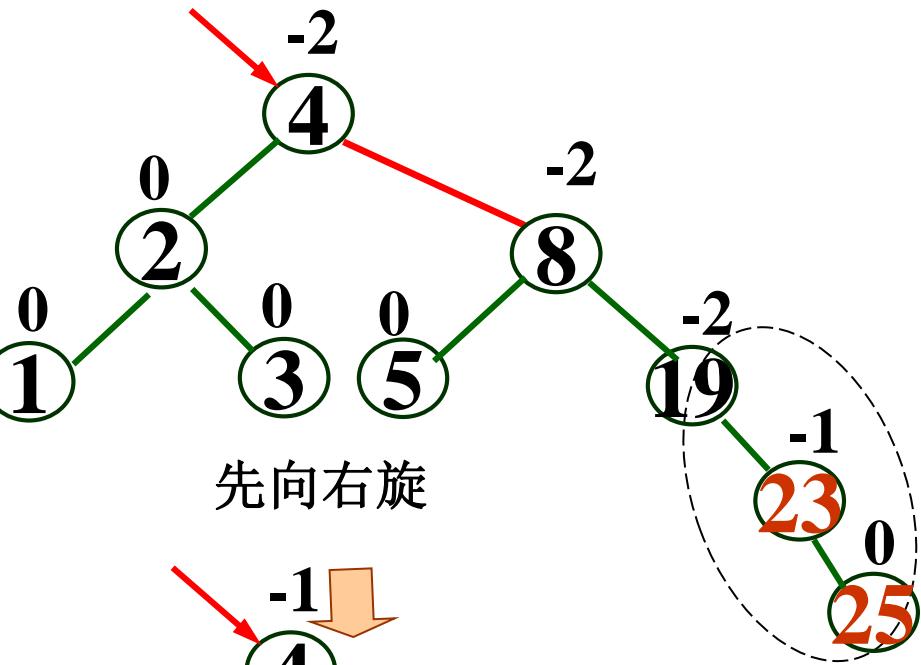
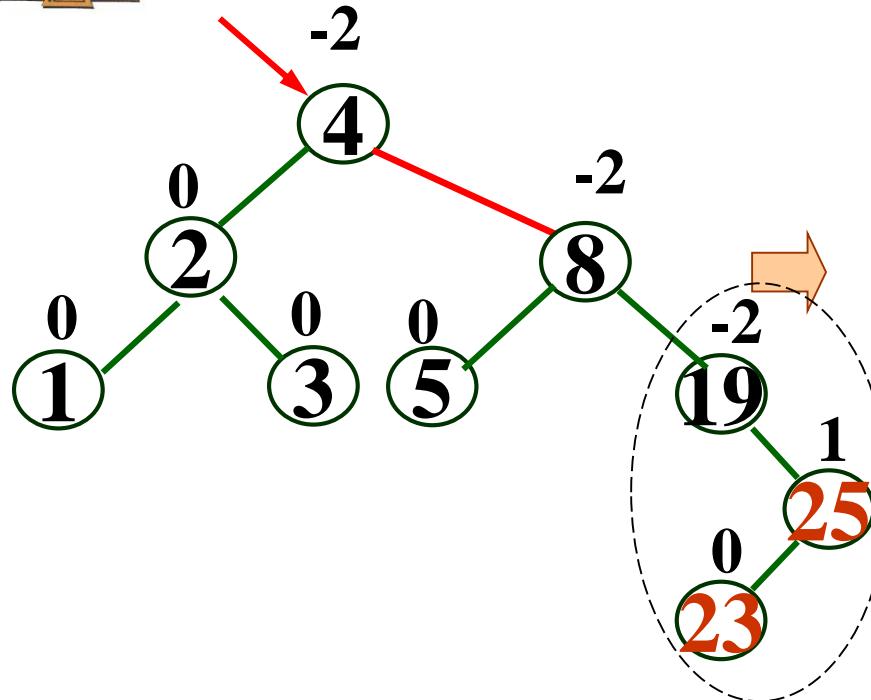
(3) 在3的左子1的右子树上插入2  
导致不平衡，可使用双向旋转：先  
使其子树左旋再右旋，可记为左右  
->左右。

③ LR型：新结点Y 被插入到 A 的左子  
树的右子树上（逆、顺）



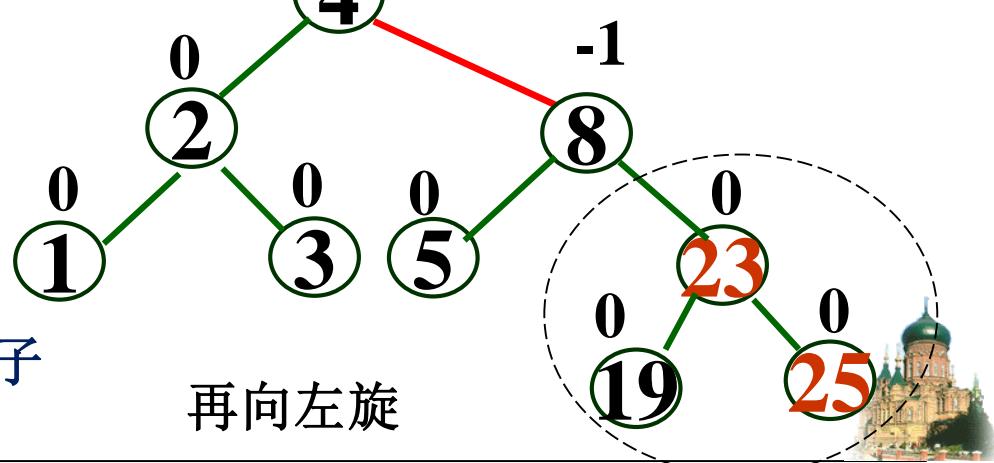


继续插入的关键字为5, 4, 3, 8, 19, 1, 2, 25, 23



(4) 在19的右子树25的左子树上插入23导致不平衡，可使用双向旋转：先使其子树右旋再左旋，可记为右左->右左。

④ RL型：新结点Y被插入到A的右子树的左子树上（顺、逆）





## 5.6 AVL树 (Cont.)

```
void R_Rotate(BSTree &p)
```

{ //对以p为根的二叉排序树作右旋处理,处理之后p指向新的树  
根结点,即旋转处理之前的左子树的根结点.

```
lc=p->lchild;
```

```
p->lchild=lc->rchild;
```

```
lc->rchild=p;
```

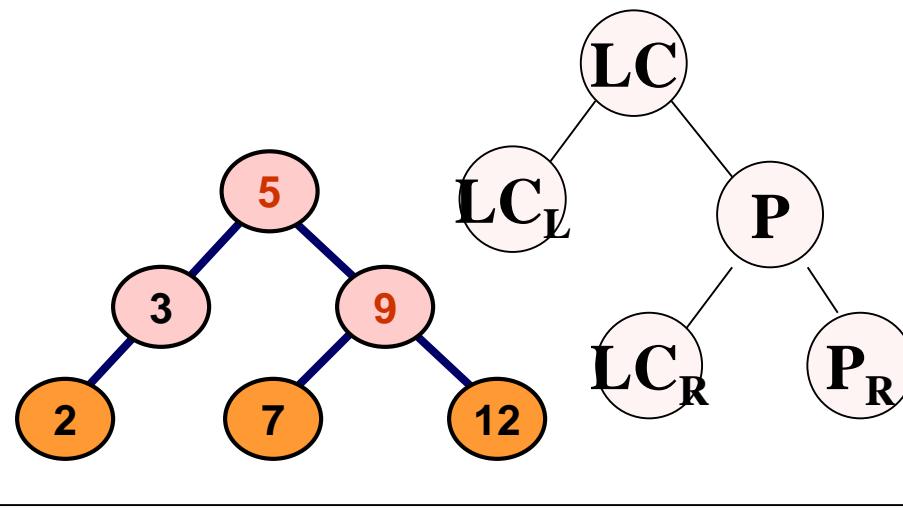
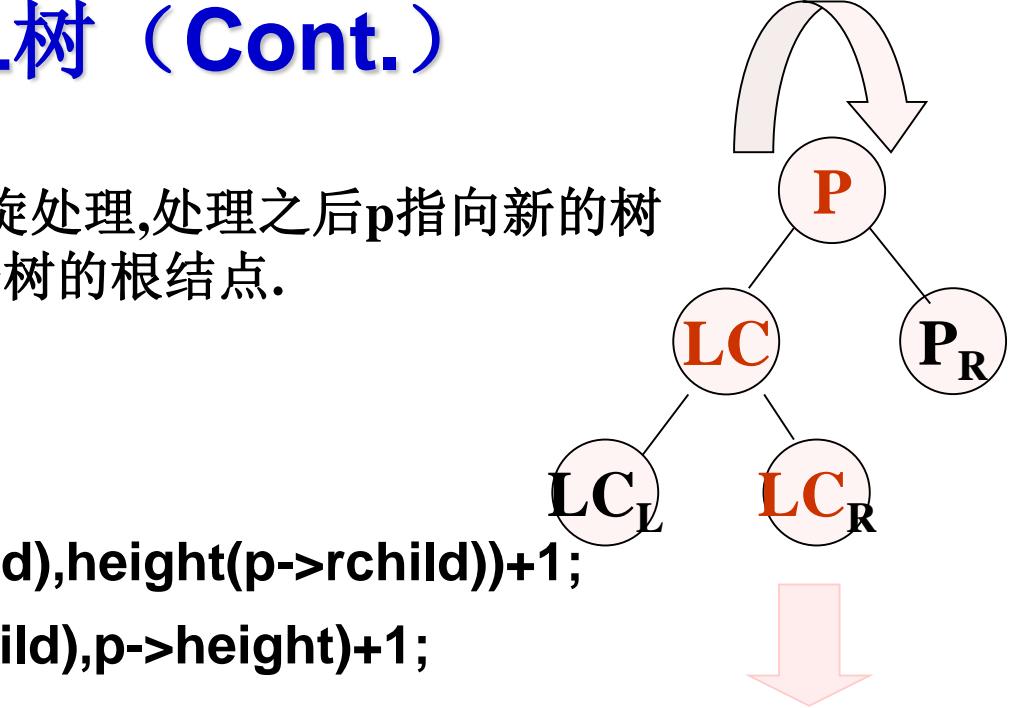
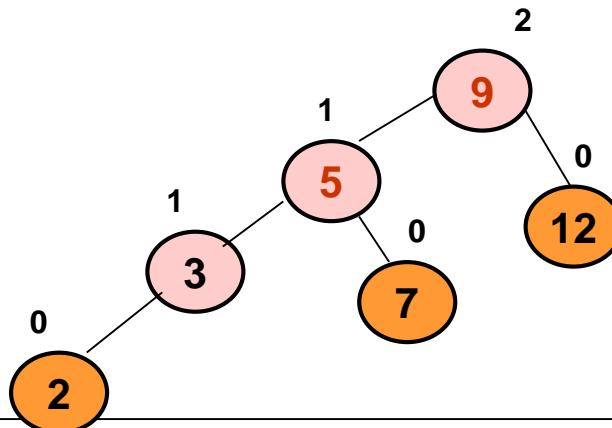
```
p->height=max(height(p->lchild),height(p->rchild))+1;
```

```
lc->height=max(height(lc->lchild),p->height)+1;
```

//重新计算p和lc的高度

```
p=lc;
```

```
}
```



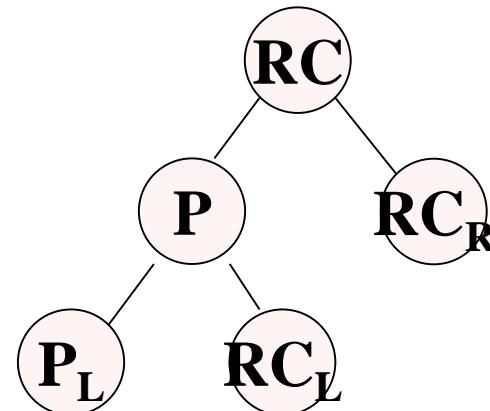
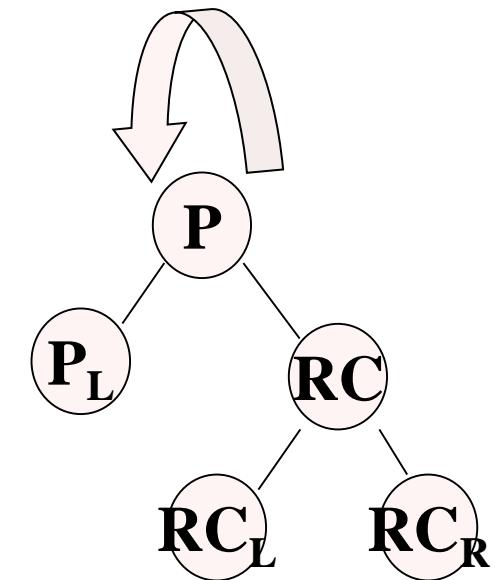


## 5.6 AVL树 (Cont.)

```
void L_Rotate(BSTree &p)
```

```
{ //对以p为根的二叉排序树作左旋处理  
, 处理之后p指向新的树根结点,即旋转处  
理之前的右子树的根结点.
```

```
rc=p->rchild;  
p->rchild=rc->lchild;  
rc->lchild=p;  
p=rc;  
}
```





## 5.6 AVL树 (Cont.)

### ◆ AVL树的建立与插入

对于一组关键字的输入序列，从空开始不断地插入结点，最后构成AVL树

- 每插入一个结点后就应判断从该结点到根的路径上有无结点发生不平衡
- 如有不平衡问题，利用旋转方法进行树的调整，使之平衡化
- 建AVL树过程是不断插入结点和必要时进行平衡化的过程

### ◆ AVL树的删除

删除操作与插入操作是对称的（镜像），可能需要的平衡化次数多。

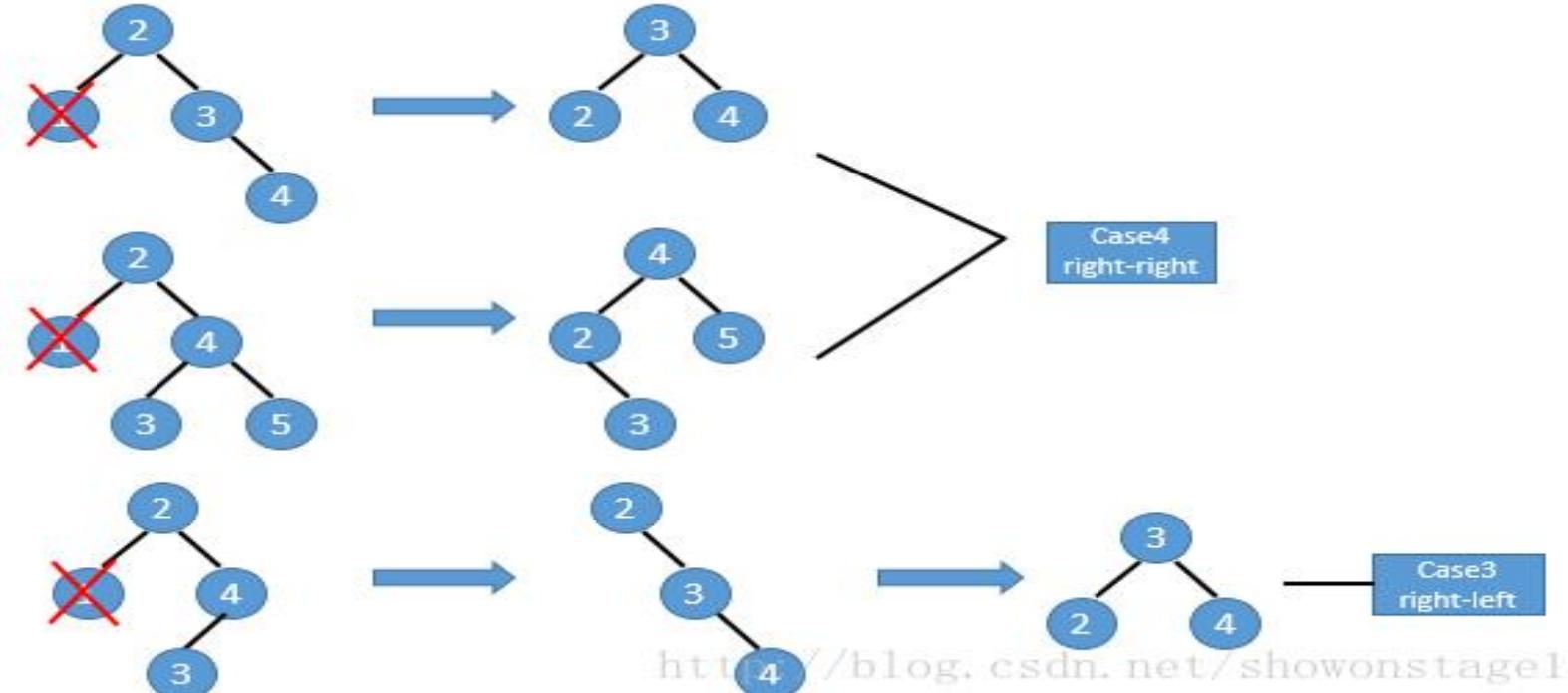
- 平衡化不会增加子树的高度，但可能会减少子树的高度。
- 有可能使树增高的插入操作中，一次平衡化能抵消掉树增高；
- 有可能使树减低的删除操作中，平衡化可能会带来祖先结点的不平衡。





## 5.6 AVL树 (Cont.)

注意问题：AVL树上删除一个结点有可能引起多次平衡。





## 5.6 AVL树 (Cont.)

### → AVL树的性能分析

- 令 $N_h$ 是高为 $h$ 的AVL树中结点个数的最小值，在最稀疏情况下，这棵AVL树的一棵子树的高度为 $h-1$ ，而另一棵子树的高度为 $h-2$ ，这两棵子树也都是AVL树。

因此， $N_h = N_{h-1} + N_{h-2} + 1$ ，其中 $N_0=0$ ， $N_1=1$ ， $N_2=2$ 。

- 可以发现， $N_h$ 的递归定义与Fibonacci数的定义 $F_n = F_{n-1} + F_{n-2}$ （其中 $F_0 = 0$ ， $F_1 = 1$ ）相似，可以用数学归纳法证明， $N_h = F_{h+2} - 1$  ( $h \geq 0$ )
- $F_h \approx \varphi^h / \sqrt{5}$ , 其中 $\varphi = (1 + \sqrt{5})/2$ ，所以， $N_h \approx \varphi^{h+2} / \sqrt{5} - 1$
- 所以，一棵包含 $n$ 个结点的AVL树，其高度 $h$ 至多为 $\log_\varphi(\sqrt{5}(n+1)) - 2$
- 对于包含 $n$ 个结点的AVL树，其最坏情况下的插入时间为 $O(\log n)$





# 5.6 AVL树 (Cont.)

BST的应用：map、multimap、set、multiset等容器其底层都是BST实现的，缺点 $O(n)$ ；

如果需要一种查询高效且有序的数据结构，并且不需要频繁的插入删除时，可以考虑用AVL可以解决这个问题，否则不合适。

问题1. 高效的查询有序序列中元素使用哪种数据结构？

问题2. 频繁插入删除使用哪种数据结构？

如果需要一种查询高效且有序的数据结构，并且需要频繁的插入删除，是否有替代的数据结构，其性能与AVL树相近，实现还容易？

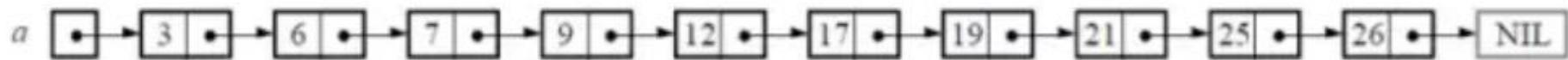




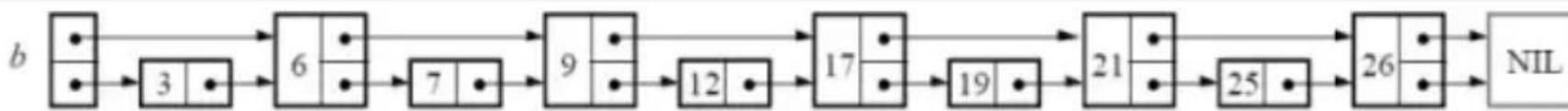
## 5.6 AVL树 (Cont.)

简单的说，要达到以 $\log n$ 的速度查找链表中的元素

跳表是一种随机化的数据结构，目前开源软件Redis 和 LevelDB 都有用到它，它的效率和红黑树以及 AVL 树不相上下



从中抽出一些节点，建立一层索引作用的链表：



课后阅读学习：跳表，红黑树





# 5.7 B-树和B+树

问题1. 数据库索引为什么要采用树结构存储，而不是采用数组或者链表的形式呢？

问题2. 为什么索引不用BST呢？

- ◆ 索引存在磁盘上，每次查询必须从磁盘上去读取一个结点，性能取决于树高，AVL树性能就不是很高。
- ◆ 如果保持查找树在高度上的绝对平衡，而允许查找树结点的子树个数（分支个数）在一定范围内变化，能否获得很好的查找性能呢？
- ◆ 基于这样的想法，1970年，R. Bayer设计了许多在高度上保持绝对平衡，而在宽度上保持相对平衡的查找结构
- ◆ 如B-树及其各种变形结构，这些查找结构不再是二叉结构，而是 $m$ -路查找树 ( $m$ -way search tree)，且以其子树保持等高为其基本性质，在实际中都有着广泛的应用。





## 5.7 B-树和B<sup>+</sup>树 (Cont.)

### ◆ $m$ -路查找树:

一棵 $m$ -路查找树或者是一棵空树, 或者是满足如下性质的树:

- 根结点最多有  $m$  棵子树, 并具有如下的结构:

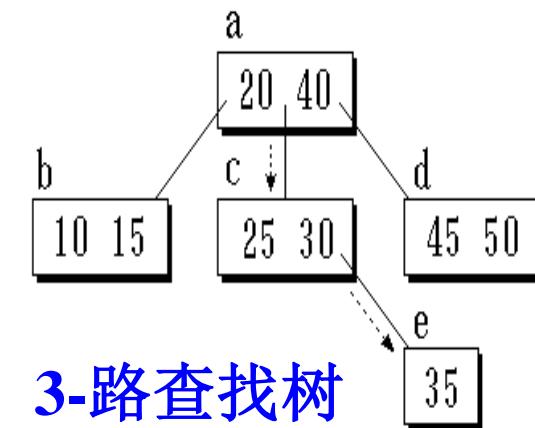
$$n, A_0, (K_1, A_1), (K_2, A_2), \dots, (K_i, A_i), \dots, (K_n, A_n)$$

其中,  $A_i$  是指向子树的指针,  $0 \leq i \leq n < m$ ;  $K_i$  是关键字值,  $1 \leq i \leq n < m$ 。  $K_i < K_{i+1}$ ,  $1 \leq i < n$ 。

- 子树  $A_i$  中所有的关键字值都小于  $K_{i+1}$  而大于  $K_i$ ,  $0 < i < n$ 。
- 子树  $A_n$  中所有的关键字值都大于  $K_n$ ;
- 子树  $A_0$  中的所有关键字值都小于  $K_1$ 。
- 每棵子树  $A_i$  也是  $m$ -路查找树,  $0 \leq i \leq n$ 。

树中可容纳结点数量最大值, 关键字个数最多

$$m^h - 1$$



3-路查找树

- ◆ 对于高度  $h=3$  的二叉树, 关键码最大个数为  $2^3-1=7$ ;
- ◆ 对于高度  $h=3$  的3路查找树, 关键码最大个数为  $3^3-1 = 26$ 。

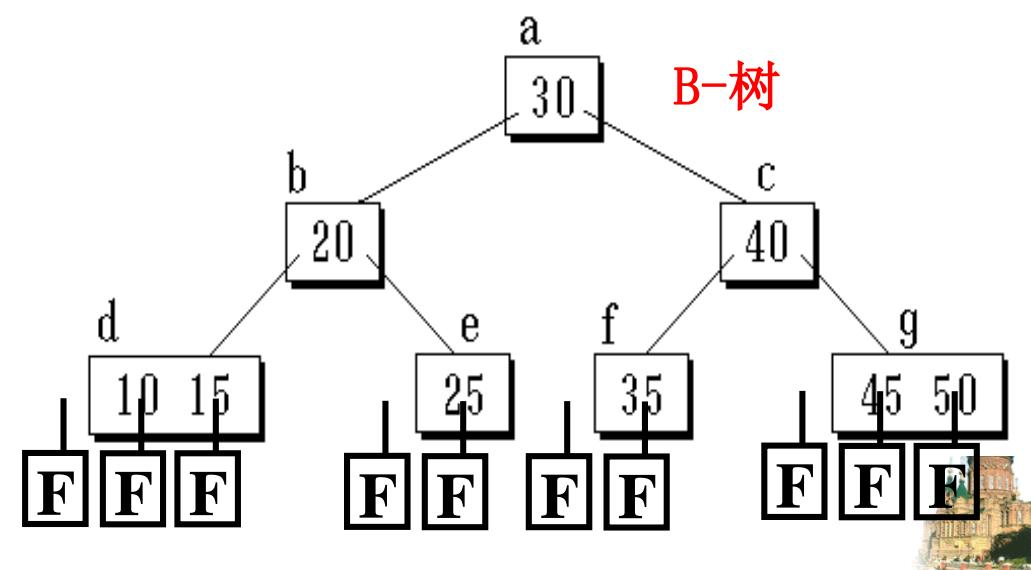
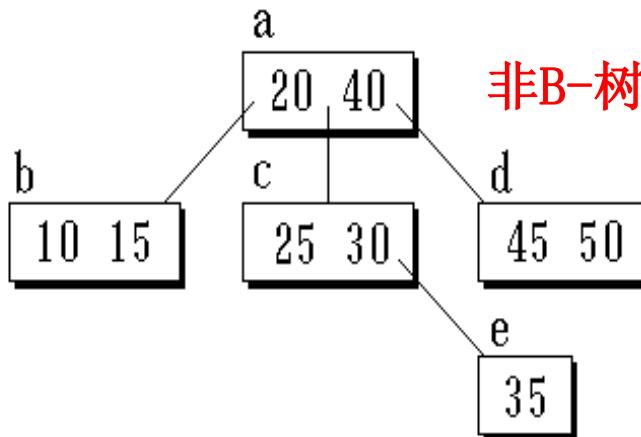




## 5.7 B-树和B<sup>+</sup>树 (Cont.)

→ **B-树 (B-tree , B是Balanced )** : 一棵  $m$  阶B-树是一棵  $m$ -路查找树，它或者是空树，或者是满足下列性质：

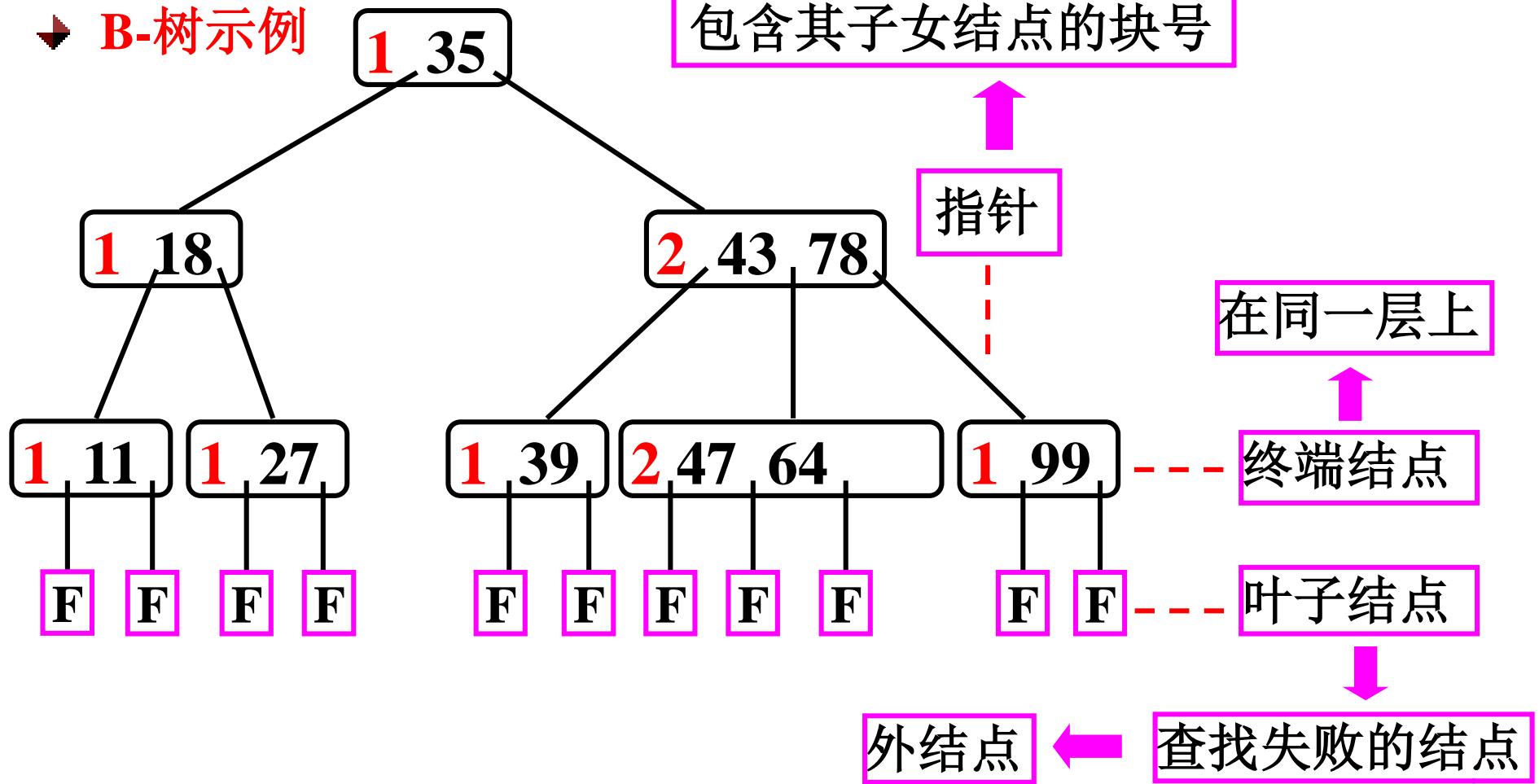
- 树中每个结点至多有  $m$  棵子树；
- 根结点至少有  $2$  棵子树； ( $2 \sim m$ )
- 除根结点和失败结点外，所有结点至少有  $\lceil m/2 \rceil$  棵子树；  $\lceil m/2 \rceil \sim m$
- 所有的终端结点和叶子结点（失败结点）都位于同一层。





## 5.7 B-树和B<sup>+</sup>树 (Cont.)

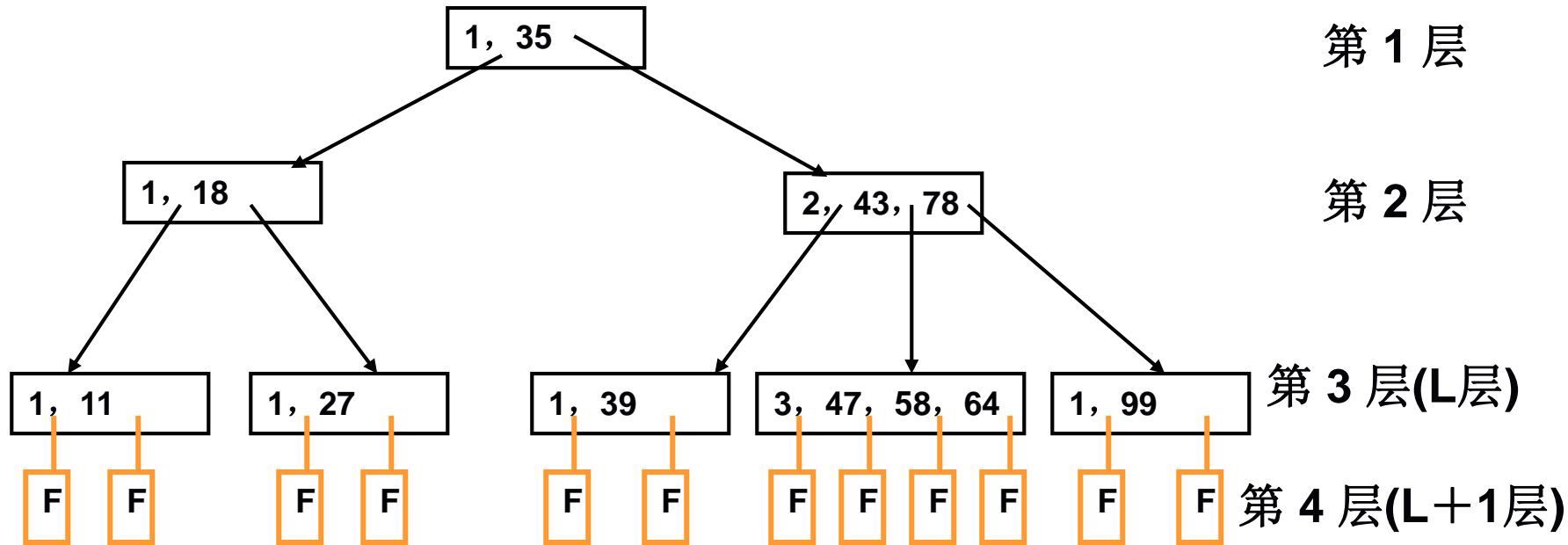
→ B-树示例





## 5.7 B-树和B<sup>+</sup>树 (Cont.)

例：m = 4 阶 B- 树。除根结点和叶子结点之外，每个结点的儿子个数为  $m/2 = 2$  个~4个；结点的关键字个数为 1~3 。该 B- 树的深度为 4。叶子结点都在第 4 层上。





## 5.7 B-树和B+树 (Cont.)

### → B-树高度 $h$ 与关键字个数 $N$ 之间的关系

- 设 $m$ 阶B-树的高为 $h$ , 失败结点位于第 $h+1$ 层, 在这棵B-树中关键字个数 $N$ 最小能达到多少?
- 从B-树的定义知,
  - 1层 1个结点
  - 2层 至少2个结点
  - 3层 至少 $2\lceil m/2 \rceil$ 个结点
  - 4层 至少 $2\lceil m/2 \rceil^2$ 个结点
  - 如此类推, .....
  - $h$ 层 至少有 $2\lceil m/2 \rceil^{h-2}$ 个结点。
  - 上述的所有结点都不是失败结点。





## 5.7 B-树和B+树 (Cont.)

- 设  $m$  阶树中关键字有  $N$  个，则失败结点数为  $N + 1$ ，
  - 设树中的关键字是  $K_1, K_2, \dots, K_N$  且  $K_i < K_{i+1}, 1 \leq i \leq N$ ，失败结点与  $K_i < x < K_{i+1}$  对应， $0 \leq i \leq N$ 。  $N + 1$  个指针的结点的一般形式为  $(N, P_0, K_1, P_1, K_2, P_2, \dots, K_N, P_N)$
  - $N + 1 = \text{失败结点个数}$

$$N + 1 = \text{位于第 } h+1 \text{ 层的结点数} \geq 2 \lceil m / 2 \rceil^{h-1}$$

$$N \geq 2 \lceil m / 2 \rceil^{h-1} - 1 \quad (\text{最少关键字的个数})$$

- 反之，如果在一棵  $m$  阶 B-树中有  $N$  个关键字，则

$$h-1 \leq \log_{\lceil m / 2 \rceil} ((N + 1) / 2) \quad (\text{最大高度})$$

- 例，若 B-树的阶数  $m = 199$ ，关键字总数  $N = 1999999$ ，则 B-树的高度  $h$  不超过

$$\log_{\lceil 199 / 2 \rceil} ((1999999 + 1) / 2) + 1 = \log_{100} 1000000 + 1 = 4$$

- 例，若 B-树的阶数  $m = 3$ ，高度  $h = 4$ ，则关键字总数至少为

$$N = 2 \lceil 3 / 2 \rceil^{4-1} - 1 = 15$$





## 5.7 B-树和B+树 (Cont.)

### → B-树的阶 $m$ 值的选择

- 如果提高B-树的阶数  $m$ ，可以减少树的高度，从而减少读入结点的次数，因而可减少读磁盘的次数。
- 但是， $m$ 受到内存可使用空间的限制。当  $m$ 很大超出内存工作区容量时，结点不能一次读入到内存，增加了读盘次数，也增加了结点内查找的难度。
- **$m$ 值的选择：**应使得在B-树中找到关键字  $x$  的时间总量达到最小
- 这个时间由两部分组成：
  - 从磁盘中读入结点所用时间
  - 在结点中查找  $x$  所用时间





## 5.7 B-树和B+树 (Cont.)

### → B-树的查找操作

- B-树的查找过程是一个顺指针（纵向）查找结点和在结点中（横向）查找交替进行的过程。
- 因此，B-树的查找时间与
  - B-树的阶数 $m$ 和
  - B-树的高度 $h$直接有关，必须加以权衡。
- 在B-树上进行查找，
  - 查找成功所需的时间取决于关键字值所在的层次；
  - 查找不成功所需的时间取决于树的高度。





## 5.7 B-树和B+树 (Cont.)

### → B-树的插入操作与建立

B-树的是从空树起，逐个插入关键字而生成的。

- 在 $m$ 阶B-树中，每个非失败结点的**关键字个数**都在 $\lceil m/2 \rceil - 1, m-1$ 之间。
- 插入操作，首先执行**查找操作**以确定可以插入新关键字的**终端结点  $p$** ；
- 如果在关键字插入后，结点中的关键字个数超出了上界  $m-1$ ，则结点需要“**分裂**”，否则可以**直接插入**。
- 实现结点“**分裂**”的原则是：设结点  $p$  中已经有  $m-1$  个关键字，当再插入一个关键字后结点中的状态为：

$$(m, A_0, K_1, A_1, K_2, A_2, \dots, K_m, A_m)$$

其中  $K_i < K_{i+1}, 1 \leq i < m$





## 5.7 B-树和B+树 (Cont.)

### → B-树的插入操作与建立

分裂与提升

- 这时必须把结点  $p$  分裂成两个结点  $p$  和  $q$ , 它们包含的信息分别为:
- 结点  $p$ :  
 $(\lceil m/2 \rceil - 1, A_0, K_1, A_1, \dots, K_{\lceil m/2 \rceil - 1}, A_{\lceil m/2 \rceil - 1})$
- 结点  $q$ :  
 $(m - \lceil m/2 \rceil, A_{\lceil m/2 \rceil}, K_{\lceil m/2 \rceil + 1}, A_{\lceil m/2 \rceil + 1}, \dots, K_m, A_m)$
- 位于中间的关键字  $K_{\lceil m/2 \rceil}$  与指向新结点  $q$  的指针形成一个二元组  
 $(K_{\lceil m/2 \rceil}, q)$ , 插入到这两个结点的父结点中去。
- 在插入该二元组之前, 先将结点  $p$  和  $q$  写到磁盘上。

例：从空树开始逐个加入关键字建立3阶B-树

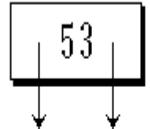




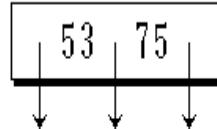
## 5.7 B-树和B+树 (Cont.)

### ◆ 3阶B-树的插入操作与建立

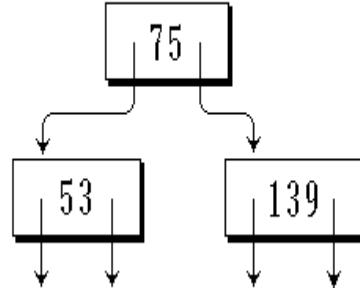
$n=1$  加入 53



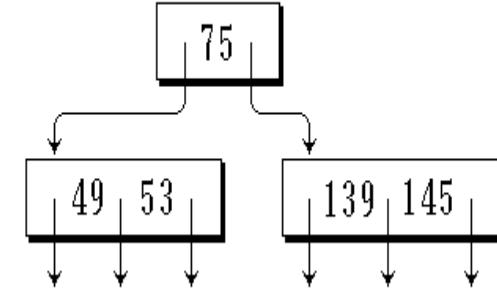
$n=2$  加入 75



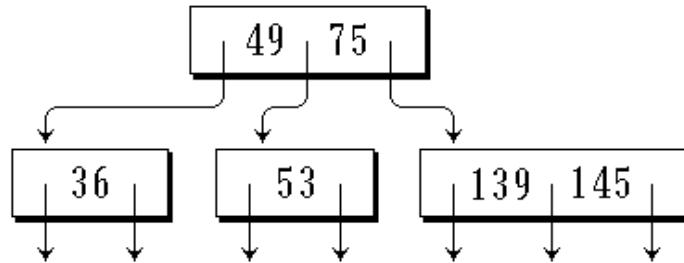
$n=3$  加入 139



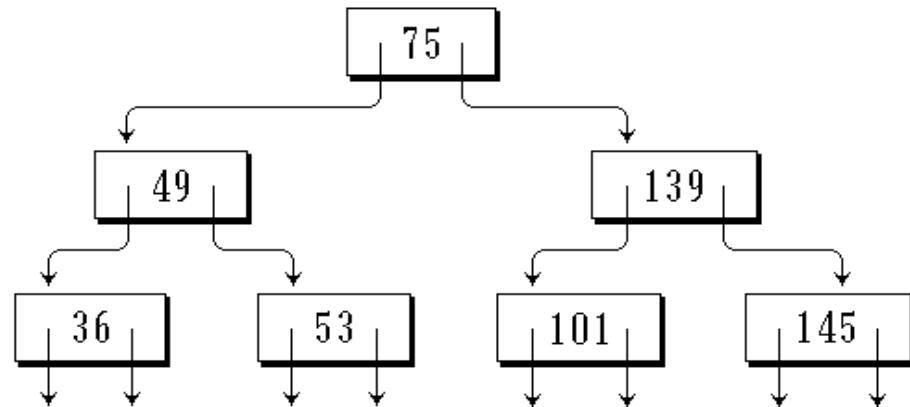
$n=5$  加入 49, 145



$n=6$  加入 36



$n=7$  加入 101



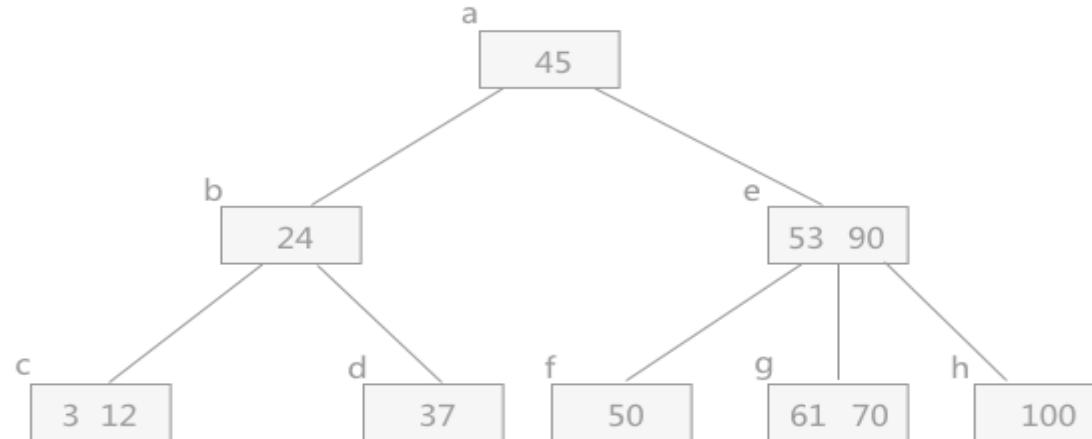
- 在插入新关键字时，需要**自底向上**分裂结点，最坏情况下从被插关键字所在终端结点到根的路径上的所有结点都要分裂



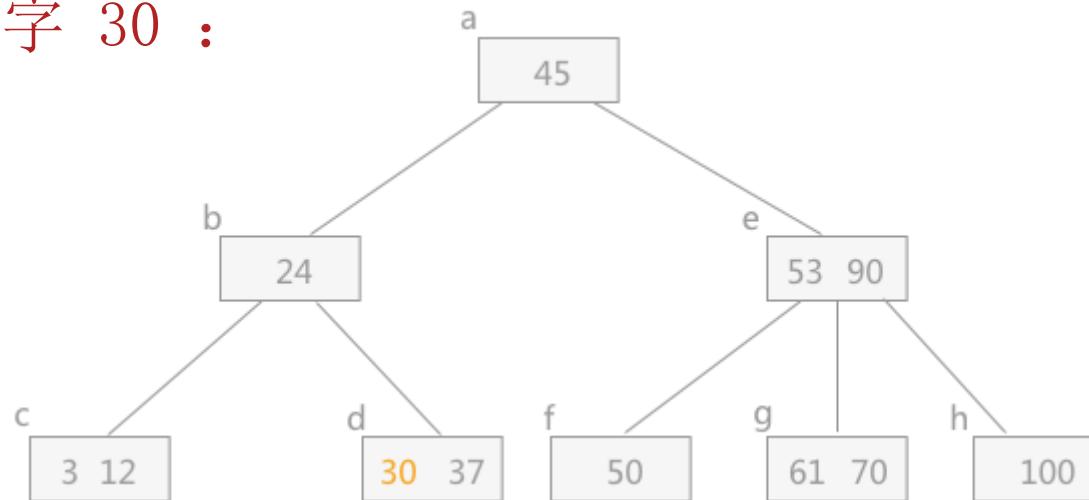


## 5.7 B-树和B+树 (Cont.)

插入 4 个关键字 30、26、85 和 7:



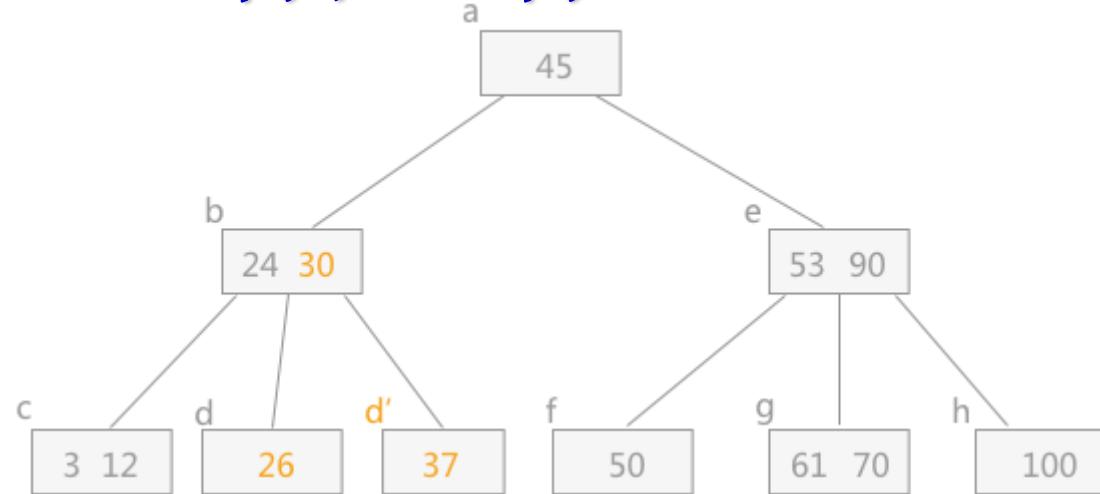
插入关键字 30 :



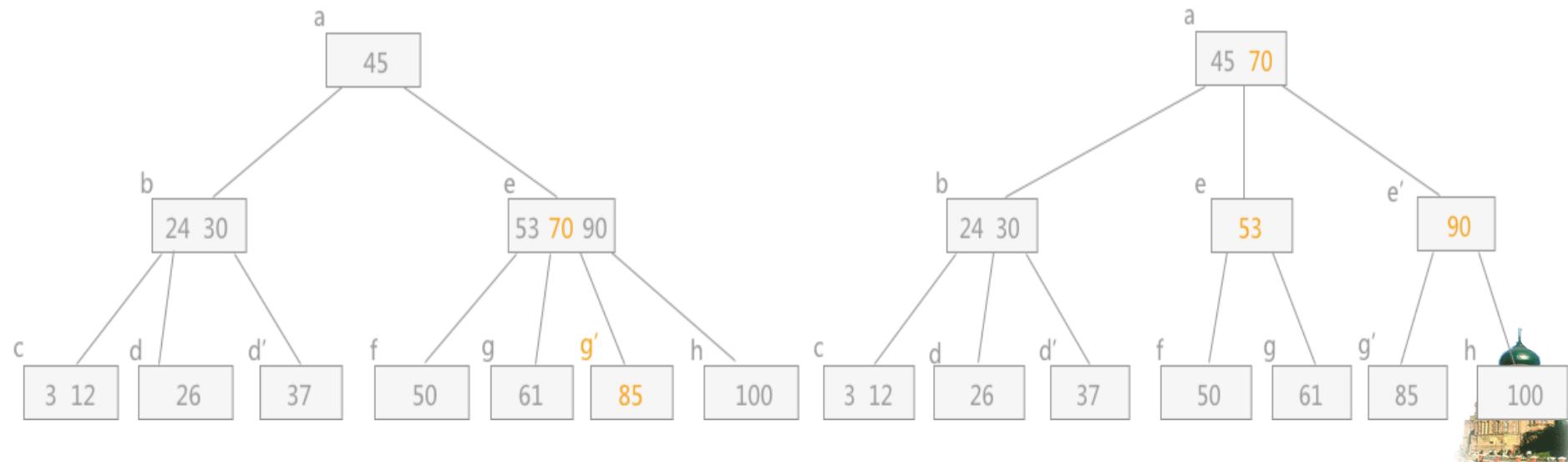


## 5.7 B-树和B+树 (Cont.)

插入关键字 26:



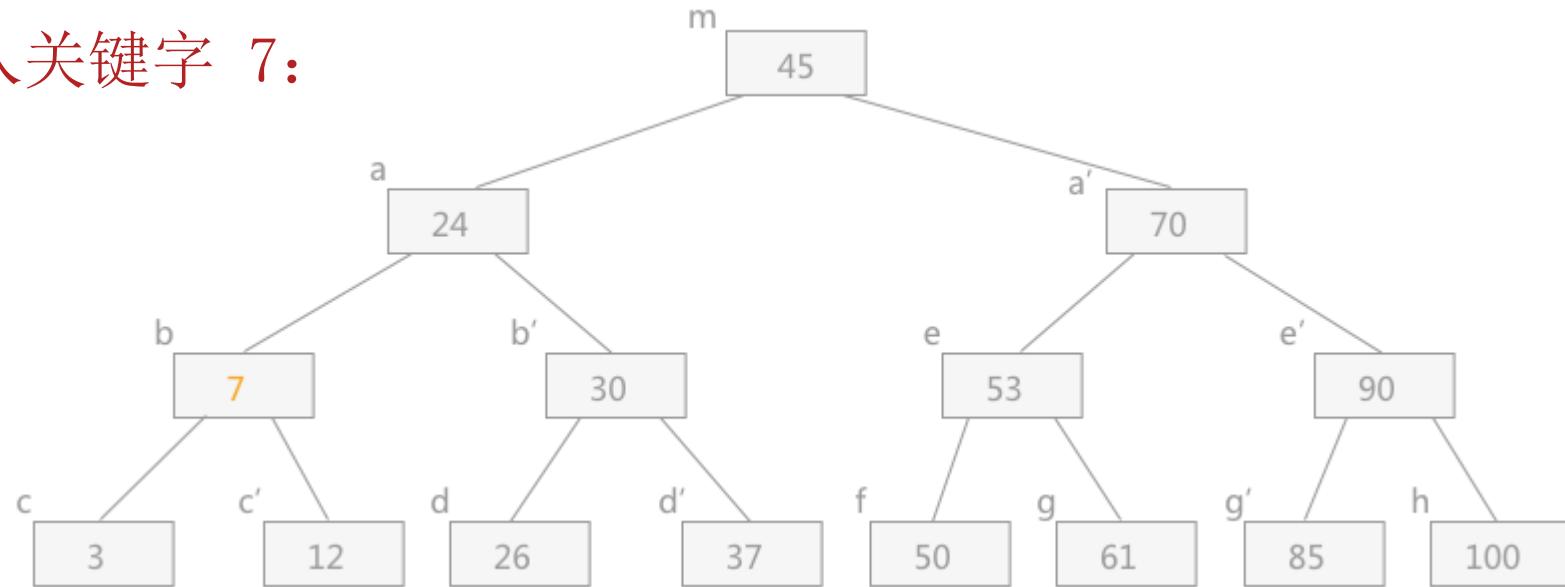
插入关键字 85:





## 5.7 B-树和B+树 (Cont.)

插入关键字 7:



在构建 B-树的过程中，假设  $p$  结点中已经有  $m-1$  个关键字，当再插入一个关键字之后，此结点分裂为两个结点

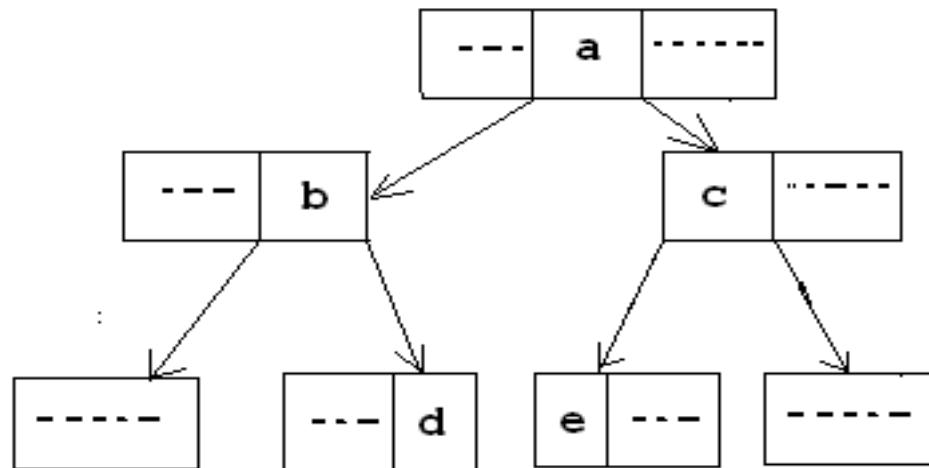




## 5.7 B-树和B+树 (Cont.)

→ **B-树的删除操作：**在B-树上删除一个关键字x时，

- 首先，要找到该关键字  $x$  所在的结点  $p$ ，从  $p$  中删去这个关键字。若该结点不是非叶结点，比如a，找到其子树中的最小关键字（如在结点d）或者其子树中的最大关键字(比如结点e)来代替被删关键字a；
- 然后在终端结点中删除d或e。把删除操作转换为终端结点上的删除操作。

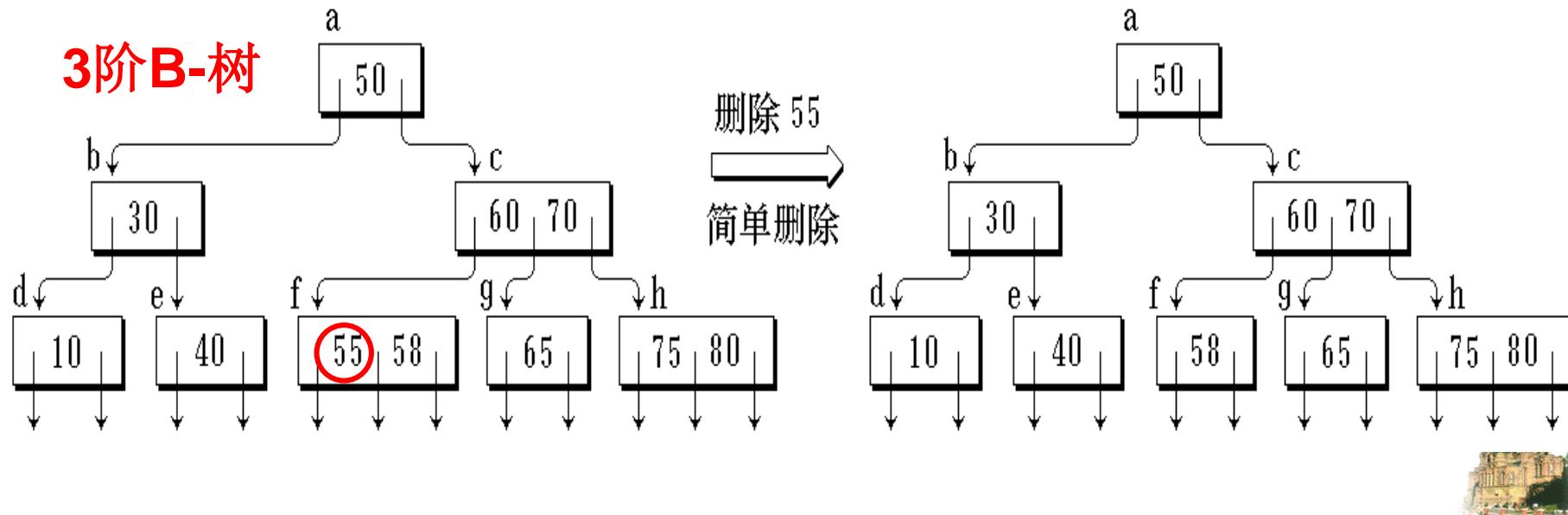




## 5.7 B-树和B+树 (Cont.)

→ B-树的删除操作：在叶结点上的删除分 4 种情况：

- 1) 被删关键字所在叶结点  $p$  同时又是根结点，若该结点有两个关键字及以上，则直接删去该关键字并将修改后的结点写回磁盘。只有一个关键字，删除以后，B-树为空。
- 2) 被删关键字所在叶结点  $p$  不是根结点，且删除前该结点中关键字个数  $n \geq \lceil m/2 \rceil$ ，则直接删去该关键字并将修改后的结点写回磁盘。

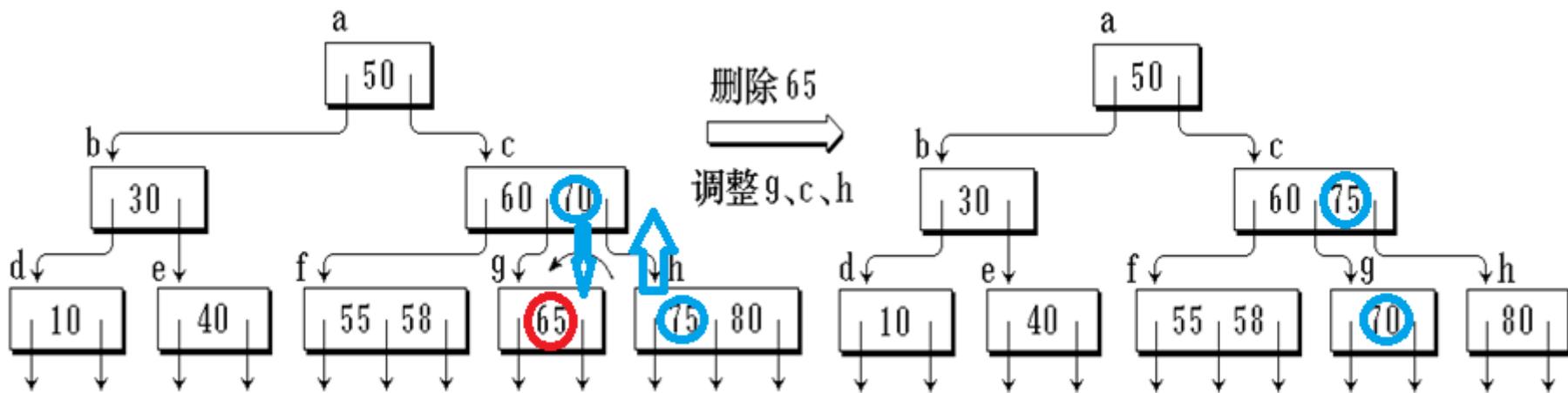




## 5.7 B-树和B+树 (Cont.)

3) 被删关键字  $x$  所在的叶结点  $p$  删除前关键字个数  $n = \lceil m/2 \rceil - 1$ , 若此时其右兄弟 (或左兄弟)  $q$  的关键字个数  $n \geq \lceil m/2 \rceil$ , 则可按以下步骤调整结点  $p$ 、右兄弟 (或左兄弟)  $q$  和其父结点  $r$ , 以达到新的平衡。

- 将父结点  $r$  中大于 (或小于) 被删的关键字的最小 (最大) 关键字  $K_i$  ( $1 \leq i \leq n$ ) 下移至结点  $p$ ;
- 将右兄弟 (或左兄弟) 结点中的最小 (或最大) 关键字上移到父结点  $r$  的  $K_i$  位置;
- 将右兄弟 (或左兄弟) 结点中的最左 (或最右) 子树指针平移到被删关键字所在结点  $p$  中最后 (或最前) 子树指针位置;
- 在右兄弟 (或左兄弟) 结点  $q$  中, 将被移走的关键字和指针位置用剩余的关键字和指针填补、调整。再将结点  $q$  中的关键字个数减1。

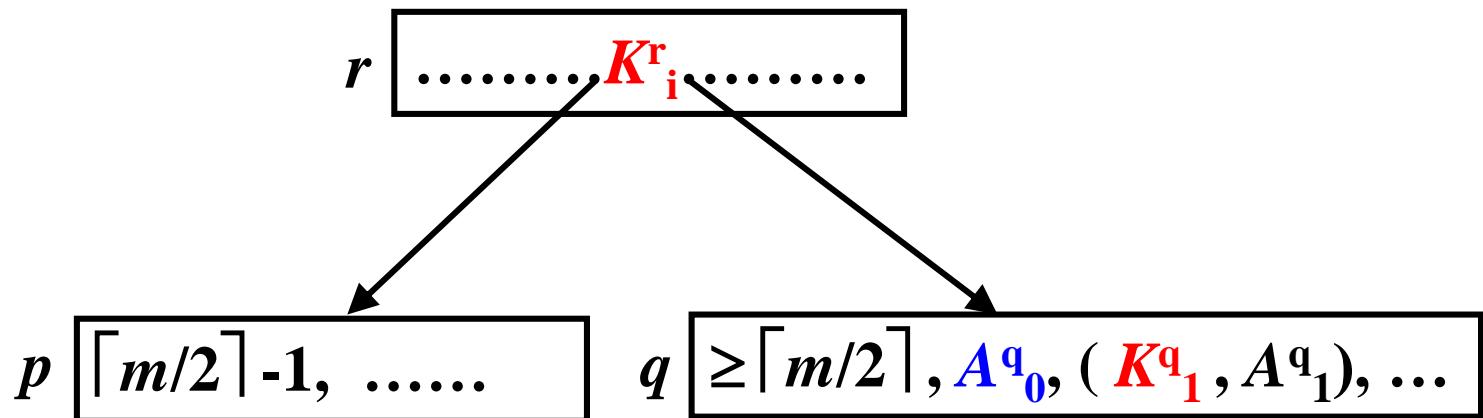




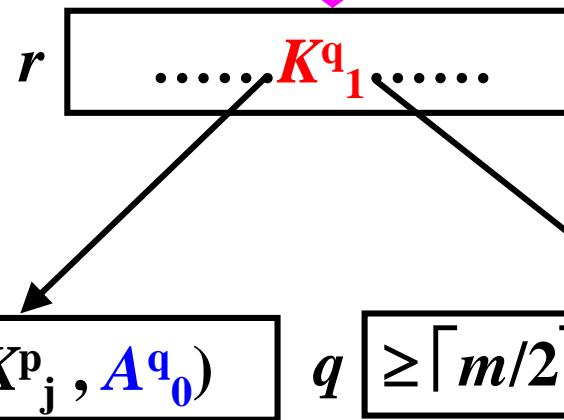
## 5.7 B-树和B+树 (Cont.)

→ B-树的删除操作：在叶结点上的删除分 4 种情况：

兄弟够借，向兄弟借；调整



调整

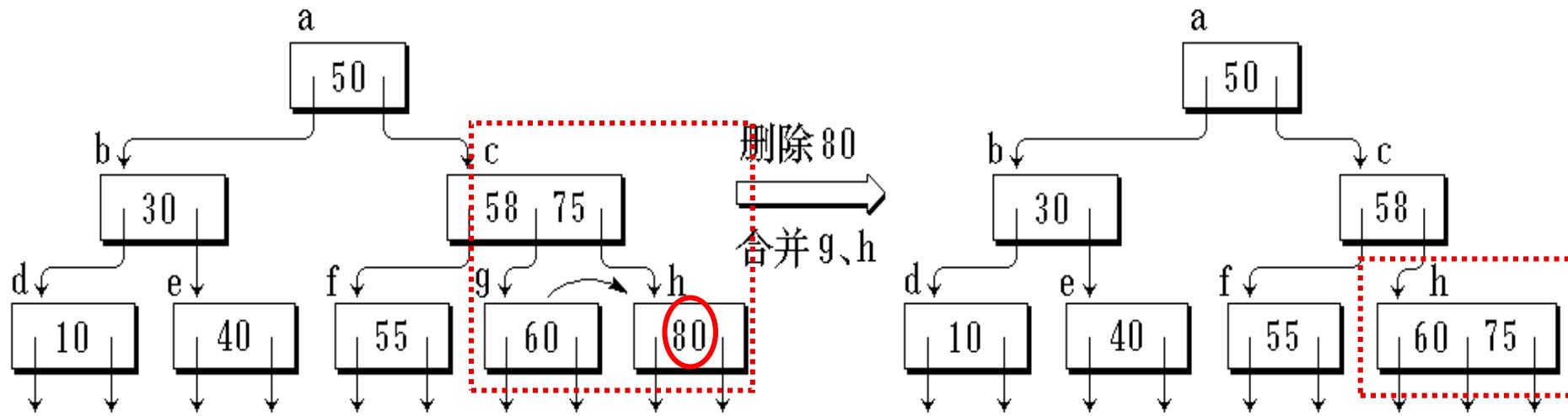




## 5.7 B-树和B+树 (Cont.)

- 4) 被删关键字 $x$ 所在的叶结点 $p$ 删除前关键字个数 $n = \lceil m/2 \rceil - 1$ , 若此时其右兄弟(或左兄弟)结点 $q$ 的关键字个数 $n = \lceil m/2 \rceil - 1$ , 则按以下步骤合并这两个结点。

- 将父结点 $r$ 中相应关键字 $K_i$ 下移到选定保留的结点中。若要合并 $r$ 中的子树指针 $A_i$ 与 $A_{i-1}$ 所指的结点, 且保留 $A_{i-1}$ 所指结点, 则把 $r$ 中的关键字 $K_i$ 下移到 $A_{i-1}$ 所指的结点中。
- 把 $r$ 中子树指针 $A_i$ 所指结点中的全部指针和关键字都拷贝到 $A_{i-1}$ 所指结点的后面。删去 $A_i$ 所指的结点。
- 在结点 $r$ 中用后面剩余的关键字和指针填补关键字 $K_i$ 和指针 $A_i$ 。
- 修改结点 $r$ 和选定保留结点的关键字个数。





## 5.7 B-树和B+树 (Cont.)

→ B-树的删除操作：在叶结点上的删除分 4 种情况：

$r \quad \dots(K^r_{i-1}, A^r_{i-1}), (\textcolor{red}{K^r_i}, A^r_i), (K^r_{i+1}, A^r_{i+1})\dots$

$p \quad [\lceil m/2 \rceil - 1, \dots]$

$q \quad [\lceil m/2 \rceil - 1, A^q_0, (\textcolor{red}{K^q_1}, A^q_1), \dots]$

$$\begin{aligned} & (\lceil m/2 \rceil - 2) + (\lceil m/2 \rceil - 1) + 1 \\ & = 2\lceil m/2 \rceil - 2 \leq m - 1 \end{aligned}$$

合并

$r \quad \dots(K^r_{i-1}, A^r_{i-1}), (K^r_{i+1}, A^r_{i+1})\dots$

$p \quad \leq m - 1, \dots(\textcolor{red}{K^r_i}, A^q_0), (\textcolor{red}{K^q_1}, A^q_1)\dots$

父节点下移到 p 中，然后 p q 合并

兄弟不够借，合并





## 5.7 B-树和B+树 (Cont.)

→ B-树的删除操作：在叶结点上的删除分 4 种情况：

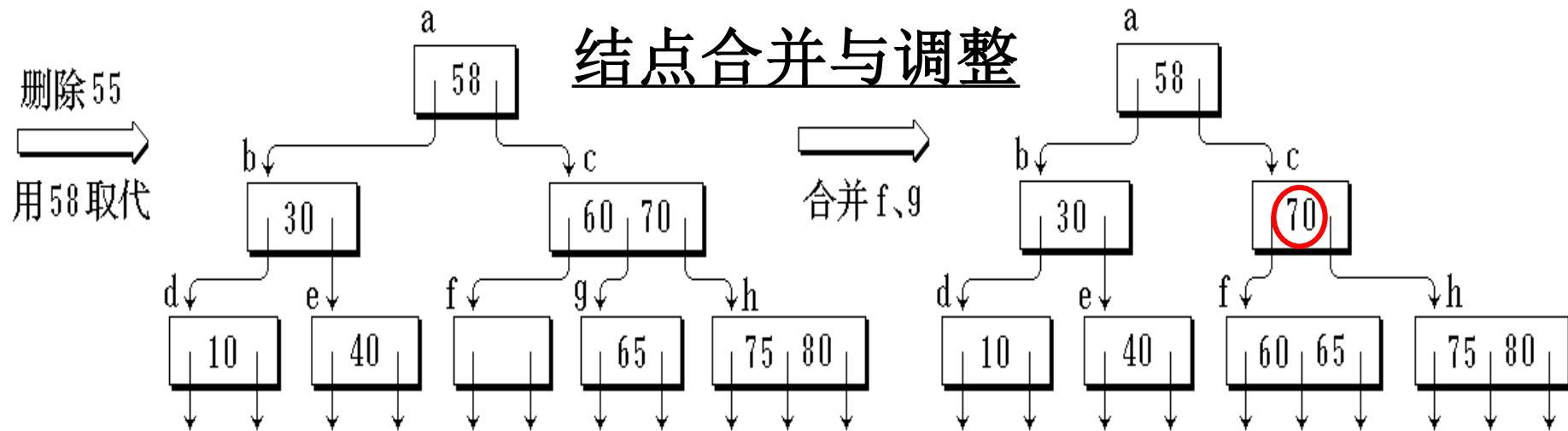
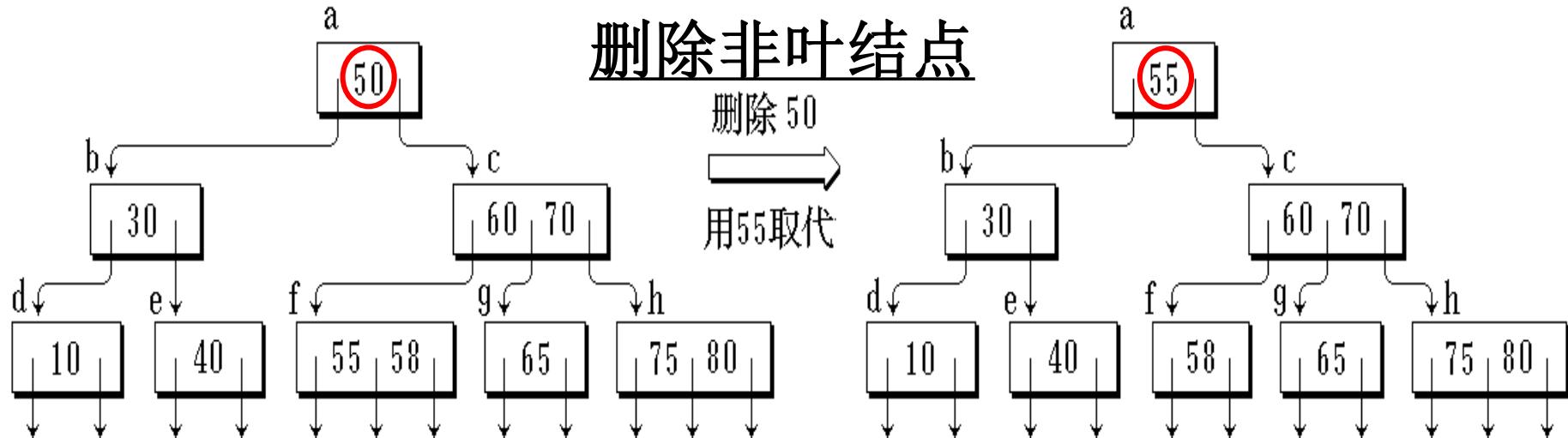
- 在合并结点的过程中，父结点中的关键字个数减少了。
- 若父结点是根结点且关键字个数为 0，则删去，合并后保留的结点成为新的根结点；否则将父结点与合并后保留的结点都写回磁盘。
- 若父结点不是根结点，且关键字个数减到 $\lceil m/2 \rceil - 2$ ，则又要与它自己的兄弟结点合并，重复上面的合并步骤。最坏情况下这种结点合并处理要自下向上直到根结点。





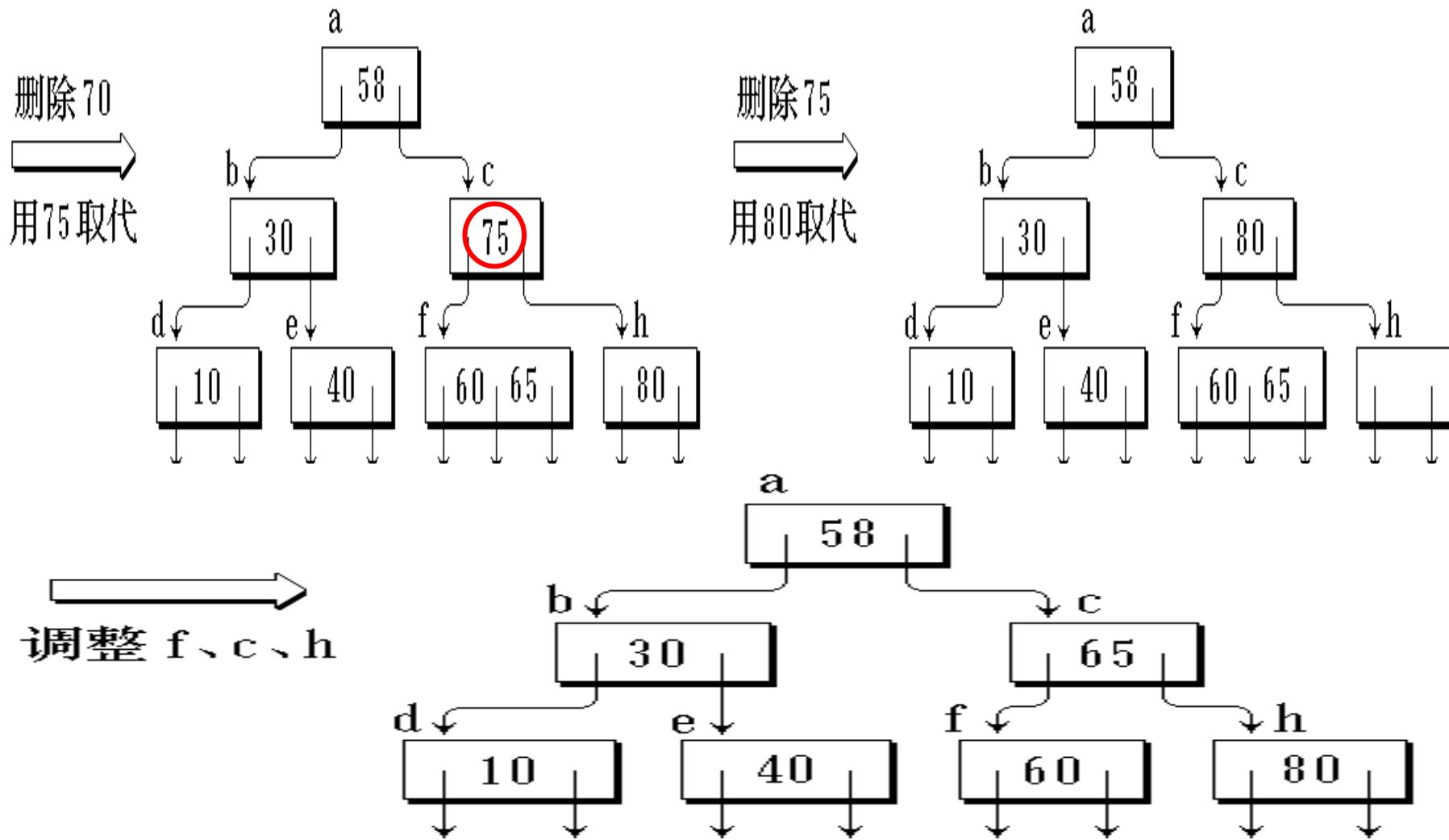
## 5.7 B-树和B+树 (Cont.)

示例：B-树的删除操作：





## 5.7 B-树和B+树 (Cont.)





## 5.7 B-树和B+树 (Cont.)

B-树只有利于单个关键字查找，而在数据库等许多应用中常常也需要范围查询。

为了满足这种需求，需要改进B-树。

B+树就是这种改进的结果，也是现实中最常用的一种B树变种。

B+树与B-树的最大区别是：B+树的元素只存放在叶结点中。

不是叶的结点又称为中间结点。中间结点也存放关键字，但这些关键字只起引导查找的“路标”作用。

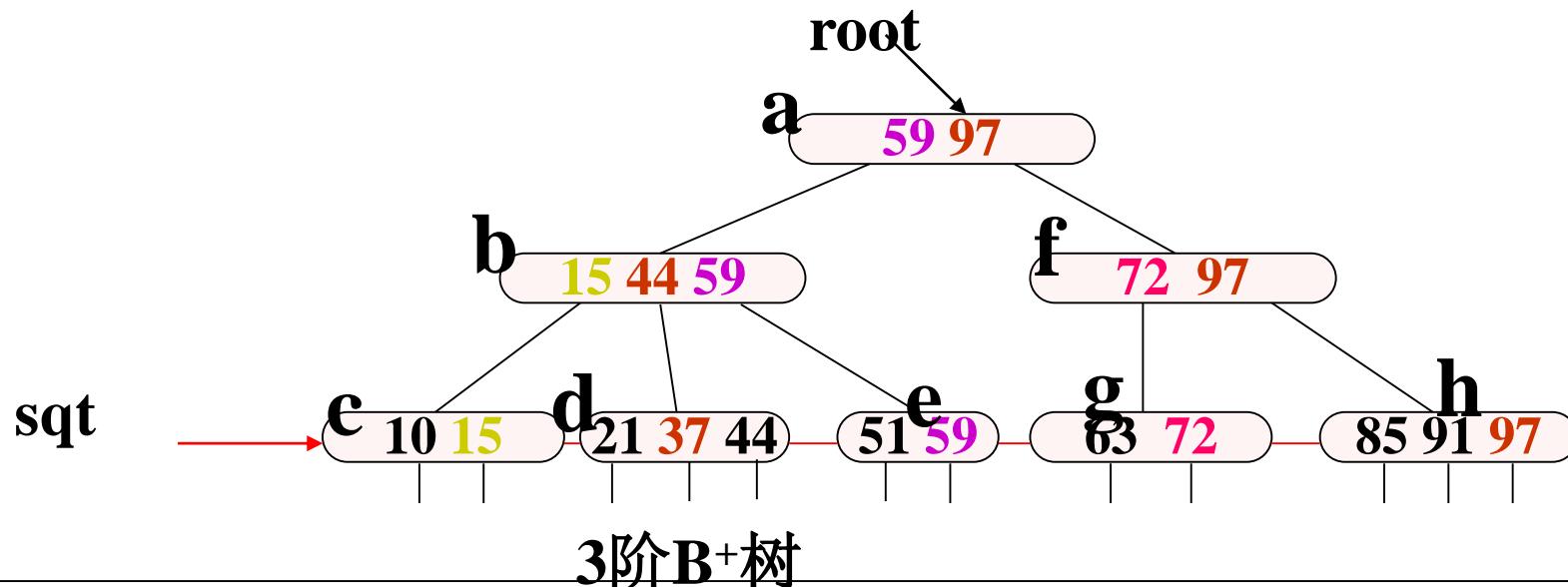




## 5.7 B-树和B+树 (Cont.)

- ◆ **B<sup>+</sup>树**: 可以看作是B-树的一种变形，在实现文件索引结构方面比B-树使用得更普遍。
- ◆ 它与B-树的差异在于：
  - (1) 有k个子结点的结点必然有k个关键字；
  - (2) 所有的叶子结点中包含了全部关键字的信息，及指向含这些关键字记录的指针，且叶子结点本身依关键字的大小自小而大顺序链接。
  - (3) 所有的非终结结点可以看成是索引部分，结点中仅含其子树（根结点）中的最大（或最小）关键字。

通常在B<sup>+</sup>树上有两个头指针，一个指向根结点，一个指向关键字最小的叶子结点。





## 5.7 B-树和B+树 (Cont.)

- ◆ B+树的查找与B-树的查找类似，但是也有不同。

由于与记录有关的信息存放在叶结点中，查找时若在上层已找到待查的关键码，并不停止，而是继续沿指针向下一直查到叶结点层的关键码。

B<sup>+</sup>树的所有叶结点构成一个有序链表，可以按照关键码排序的次序遍历全部记录。上面两种方式结合起来，使得B<sup>+</sup>树非常适合范围检索。

- ◆ B+树的插入与B-树的插入过程类似。
- ◆ B+树中的关键码在叶结点层删除后，其在上层的副本可以保留，作为一个“分界关键字”存在，如果因为删除而造成结点中关键码数小于 $\lceil m/2 \rceil$ ，其处理过程与B-树的处理一样。





## 5.7 B-树和B+树 (Cont.)

- ◆ B-树只适合随机检索，但B+树同时支持随机检索和顺序检索，在实际中应用比较多。
  - 叶结点中存放的是对实际数据对象的索引。
  - 在B+树中有两个头指针：一个指向B+树的根结点，一个指向关键字最小的叶结点。
  - 可对B+树进行两种查找操作：  
一种是沿叶结点链顺序查找（区间查找），  
另一种是从根结点开始，进行自顶向下，直至叶结点的随机查找。

### 问题3. 为什么索引采用B+树而不采用B-树？

#### 1. IO次数更少

因为B+树的中间节点没有卫星数据，只有代表地址的数据，对应的节点就可以省下卫星数据的空间，所以同样大小的磁盘页可以容纳更多的节点元素，代表着数据量相同的情况下B+树比B-树的高度更加低，查询时的IO次数也会更少

#### 2. 查询性能更稳定

因为B-树每个节点都存储着数据，B+树只有叶子结点存储着数据，所以B-树的查询性能并不稳定（最好情况是只查根节点，最坏情况是查到叶子结点），而B+树的每一次查找都是稳定的

#### 3. 范围查询更方便

因为B+树底层是链表结构，所以在连续查询时只需要顺着底部的链表进行查询就可以了，而B-树需要在叶子结点、中间节点上反复横跳





# 几种查找树的应用：

“高度严格”的平衡二叉树  
理想化的概念

**AVL树**: 最早的平衡二叉树之一。应用相对其他数据结构比较少。windows对进程地址空间的管理用到了AVL树。

**红黑树**: 平衡二叉树，广泛用在C++的STL中。如map和set都是用红黑树实现的。

**B/B+树**: 用在磁盘文件组织 数据索引和数据库索引。

**Trie树(字典树)**: 用在统计和排序大量字符串，如自动机。

“大致上”平衡树  
是AVL的具体应用





## 红-黑树 (R-B Tree)

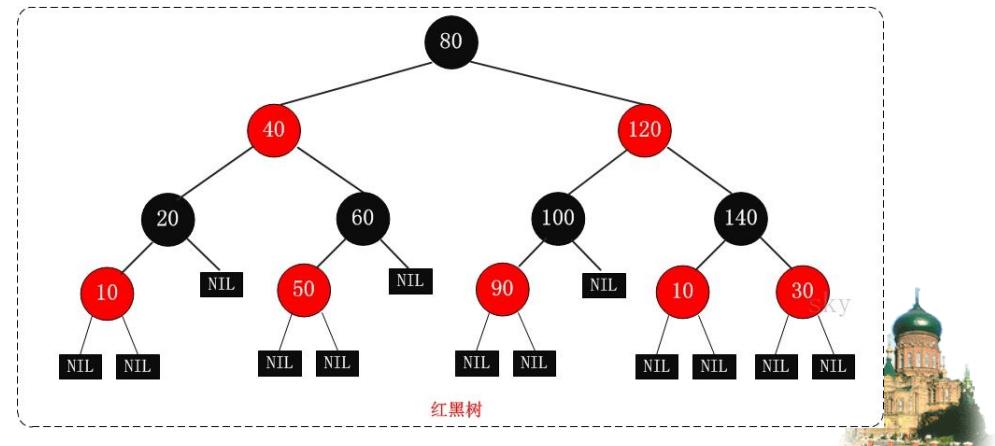
全称是Red-Black Tree，又称为“红黑树”，它一种特殊的二叉查找树。红黑树的每个节点上都有存储位表示节点的颜色，可以是红(Red)或黑(Black)。

红黑树是一种近似平衡的二叉查找树，能够确保任何一个节点的左右子树的高度差不会超过二者中较低那个的一倍。

红黑树的特性：

- (1) 每个节点或者是黑色，或者是红色。
- (2) 根节点是黑色。
- (3) 每个叶子节点 (NIL) 是黑色。
- (4) 如果一个节点是红色的，则它的子节点必须是黑色的。
- (5) 从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。

红黑树的应用比较广泛，主要是用它来存储有序的数据，它的时间复杂度是 $O(\lg n)$ 。





- 二叉树
  - 二叉树
  - T 树
- 自平衡二叉查找树
  - AA 树
  - 树堆
  - AVL 树
  - 节点大小平衡树
- B 树
  - B 树
  - UB 树
  - Dancing tree
  - H 树
  - B+ 树
  - 2-3 树
- Trie
  - 前缀树
  - 后缀树
  - 基数树
- 空间划分树
  - 四叉树
  - R 树
  - M 树
  - 八叉树
  - R\* 树
  - 线段树
  - k-d 树
  - R+ 树
  - 希尔伯特 R 树
  - VP 树
  - X 树
  - 优先 R 树
- 非二叉树
  - Exponential tree
  - Range tree
  - Fusion tree
  - SPQR tree
  - 堆
  - 散列树
  - 区间树
  - Van Emde Boas tree
  - Finger tree
  - Metric tree
- 其他类型
  - Cover tree
  - BK-tree
  - Doubly-chained tree
  - iDistance tree
  - Link-cut tree
  - 树状数组





## 5.8 散列技术

► 查找操作要完成什么任务？

- 对于待查值 $k$ ，通过比较，确定 $k$ 在存储结构中的位置

► 基于关键字比较的查找的时间性能如何？

- 其时间性能为 $O(\log n) \sim O(n)$ 。

- 实际上用判定树可以证明，基于关键字比较的查找的平均和最坏情况下的比较次数的下界是 $\log n + O(1)$ ，即 $\Omega(\log n)$

- 要向突破此下界，就不能仅依赖于基于比较来进行查找。

► 能否不用比较，通过关键字的取值直接确定存储位置？

□ 关键字值和存储位置之间建立一个确定的对应关系

► 什么情况下检索元素的速度最快？

- 如果关键字是顺序表的下标，就可以直接找到元素

- 但是一般而言，关键码可能不是整数（不能作为下标）

- 即使是整数，也可能取值范围太大，不适合直接作为下标

- 例如，身份证号 18 位，取值范围达  $10^{18}$ ，人口  $10^9$ ，数据密度  $1/10^9$

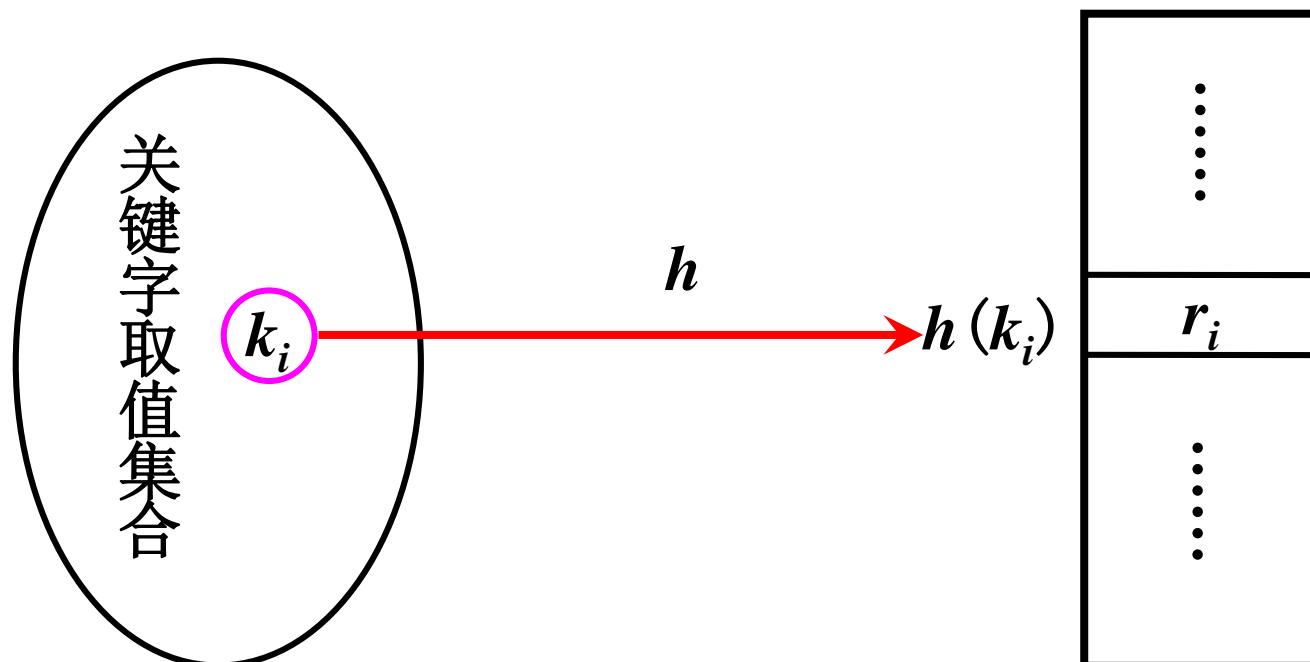




## 5.8 散列技术 (Cont.)

### ◆ 散列技术的基本思想

- 把记录（元素）的存储位置和该记录的关键字的值之间建立一种映射关系。关键字的值在这种映射关系下的像，就是相应记录在表中的存储位置。
- 散列技术在理想情况下，无需任何比较就可以找到待查的关键字，其查找的期望时间为**O(1)**。



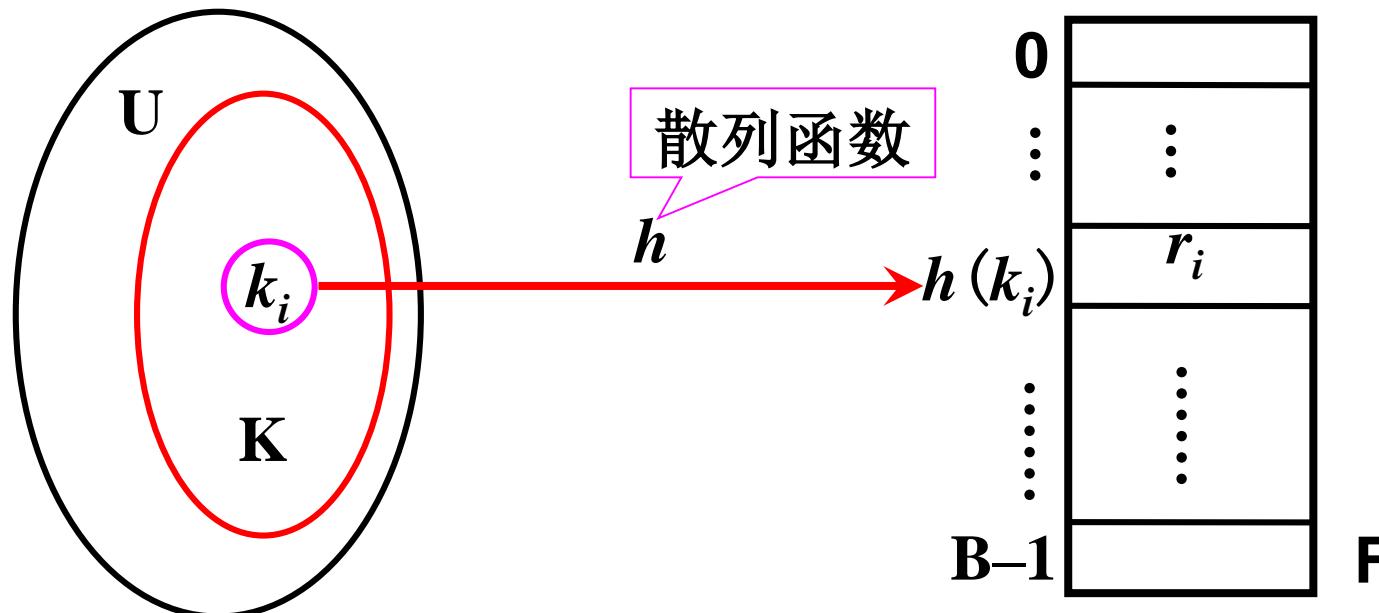


## 5.8 散列技术 (Cont.)

### ◆ 散列技术的相关概念

设  $U$  表示所有可能出现的关键字集合， $K$  表示实际出现（实际存储）的关键字集合，即  $K \subseteq U$ ， $F[B - 1]$  是一个数组，其中  $B = O(|K|)$ 。则，

- 从  $U$  到表  $F[B - 1]$  下标集合上的一个映射  $h$ :  $U \rightarrow \{0, 1, 2, \dots, B - 1\}$  称为散列函数（哈希函数，杂凑函数）

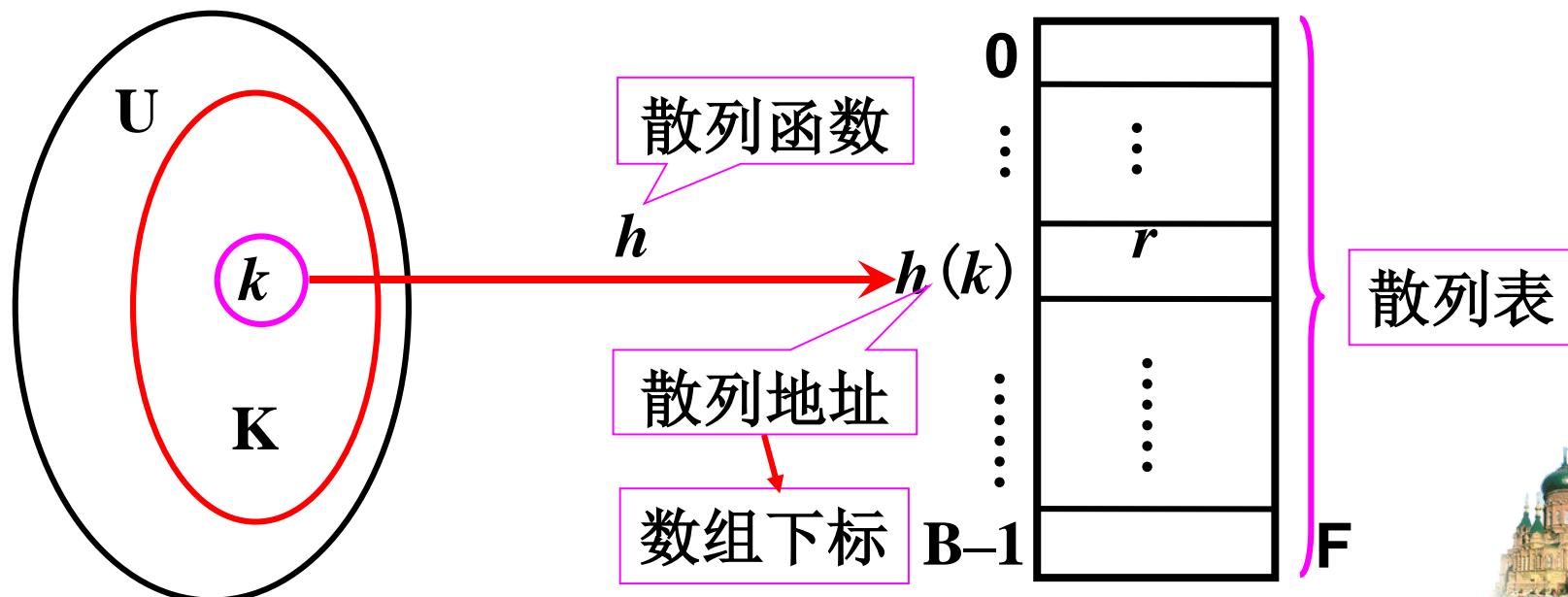




## 5.8 散列技术 (Cont.)

### ◆ 散列技术的相关概念

- 数组  $F$  称为散列表（Hash表，杂凑表）。数组  $F$  中的每个单元称为桶(bucket)。
- 对于任意关键字  $k \in U$ ，函数值  $h(k)$  称为  $k$  的散列地址（Hash地址，散列值，存储地址，桶号）

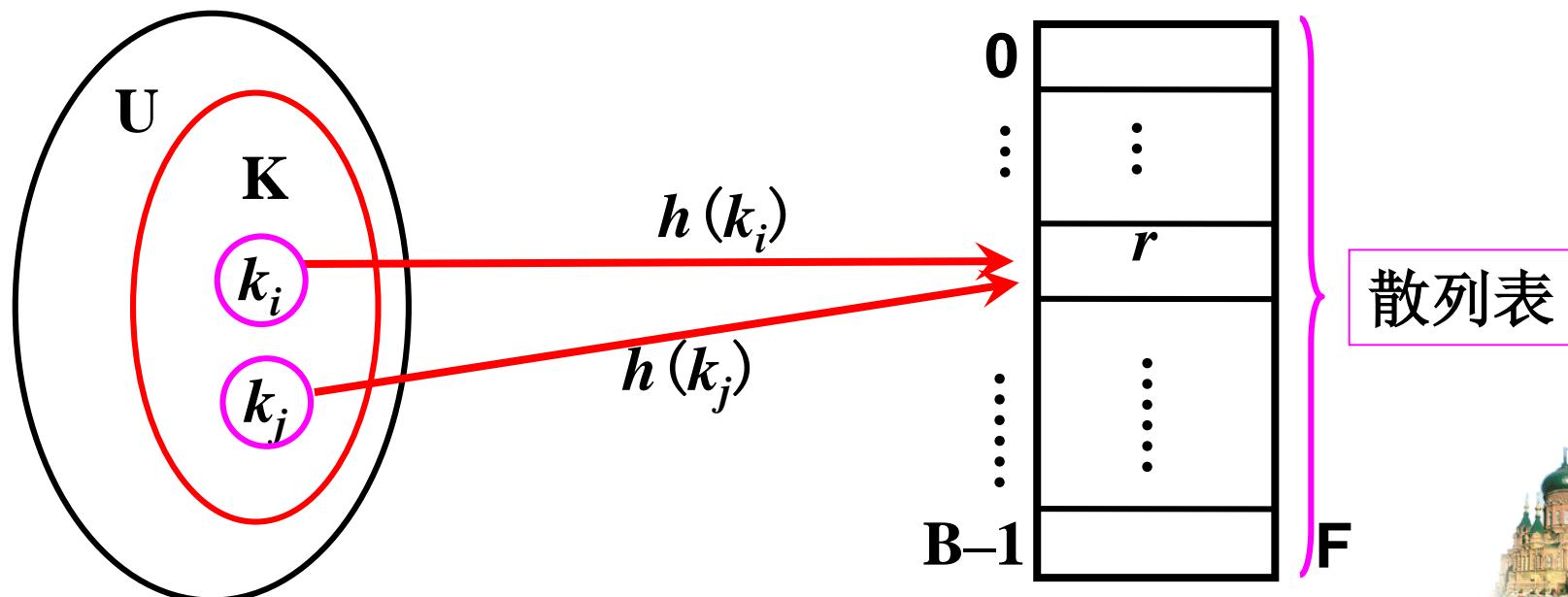




## 5.8 散列技术 (Cont.)

### ◆ 散列技术的相关概念

- 将结点（记录）按其关键字的散列地址存储到散列表中的过程称为散列（collision）。
- 不同的关键字具有相同散列地址的现象称为散列冲突（碰撞）。而发生冲突的两个关键字称为同义词(synonym)。





## 5.8 散列技术 (Cont.)

- ▶ **散列技术仅仅是一种查找技术吗？**
  - 散列既是一种查找技术，也是一种存储技术。
- ▶ **散列是一种完整的存储结构吗？**
  - 散列只是通过记录的关键字的值定位该记录，没有表达记录之间的逻辑关系，所以散列主要是**面向查找**的存储结构
- ▶ **散列技术适用于何种场合？**
  - 通常用于实际出现的关键字的数目远小于关键字所有可能取值的数量。
- ▶ **散列技术适合于哪种类型的查找？**
  - 不适用于允许多个记录有同样关键字值的情况。
  - 也不适用于范围查找，比如找最大或最小关键值的记录，也不可能找到在某一范围内的记录。
- ▶ **散列技术最适合回答的问题是：**如果有的话，哪个记录的关键字的值等于待查值。
- ▶ **散列技术有两种：**一种是内散列法或称闭散列表，基本的数据结构是数组；另一种称为外散列表或称开散列表，基本数据结构是邻接表。





## 5.8 散列技术 (Cont.)

### ◆ 散列技术需解决的关键问题:

- 散列函数的构造。
  - 如何设计一个**简单、均匀**、存储利用率高的散列函数
- 冲突的处理
  - 如何采取合适的处理冲突方法来解决冲突。
- 散列结构上的查找、插入和删除

### ◆ 散列函数的构造的原则:

- 计算**简单**: 散列函数不应该有很大的计算量, 否则会降低查找效率。
  - 分布**均匀**: 散列函数值即散列地址, 要尽量**均匀分布**在地址空间, 这样才能保证存储空间的有效利用并减少冲突。
- ◆ 一些方法依赖于对实际数据集的分析, 实际中很难使用

如果事先知道需要存储的数据及其分布, 有可能设计出一个散列函数取得最佳存储效果, 甚至可能保证不出现冲突

书中介绍了几种散列函数设计, 如**数字分析法, 折叠法, 中平方法等**





## 5.8 散列技术 (Cont.)

### ◆ 散列函数的构造方法----直接定址法

- 散列函数是关键字值的线性函数，即： $h(key) = a \times key + b$  ( $a, b$ 为常数)
- 示例：关键字的取值集合为{10, 30, 50, 70, 80, 90}，选取的散列函数为 $h(key)=key/10$ ，则散列表为：

0	1	2	3	4	5	6	7	8	9
	10		30		50		70	80	90

- 适用情况：事先知道关键字的值，关键字取值集合不是很大且连续性较好。





## 5.8 散列技术 (Cont.)

### ◆ 散列函数的构造方法---质数除余法

- 散列函数为:  $h(key)=key \% m$
- 一般情况下, 选 $m$ 为小于或等于表长B的最大质数。
- 示例: 关键字的取值集合为{14, 21, 28, 35, 42, 49, 56, 63}, 表长B=12。则选取 $m=11$ , 散列函数为 $h(key)=key \% 11$ , 则散列表为:

0	1	2	3	4	5	6	7	8	9	10
	56	35	14		49	28		63	42	21

- **适用情况:** 质数除余法是一种最简单、也是最常用的构造散列函数的方法, 并且不要求事先知道关键码的分布。

Knuth的研究结果表明, M应为质数, 且不能整除 $r^k \pm a$ , 其中, r是字符集的基数(例如, 如果一个字符用6位二进制数表示, 则 $r = 64$ ), k和a是小的整数。

实际应用表明, 只要选择不能被小于20的质数整除的整数作为M就足够了。





## 5.8 散列技术 (Cont.)

### → 散列函数的构造方法----随机数法

- 选择一个随机函数，取关键字的随机函数值作为散列地址，即  
 $\text{Hash}(\text{key}) = \text{random}(\text{key})$

其中random是某个伪随机函数，且函数值在0,...,B-1之间。

- 适用情况：通常，当关键字长度不等时采用此法较恰当

■ 小结：构造Hash函数应注意以下几个问题：

- 计算Hash函数所需时间      **计算简单**
  - 关键字的长度
  - 散列表的大小
  - 关键字的分布情况
  - 记录的查找频率
- }
- 分布均匀





## 5.8 散列技术 (Cont.)

冲突处理的方法——开放定址法

◆ 基本思想：

- 当冲突发生时，使用某些探测技术在散列表中形成一个探测序列，沿此序列逐个单元查找，直到找到给定的关键字或者碰到一个开放地址（即该空的地址单元、空桶）或者既未找到给定的关键字也没碰到一个开放地址为止。

◆ 常用的探测技术——如何寻找下一个空的散列地址？

- 线性探测法
- 线性补偿探测法
- 二次探测法
- 随机探测法

◆ 闭散列表：用开放定址法处理冲突得到的散列表叫闭散列表。





## 5.8 散列技术 (Cont.)

冲突处理的方法----开放定址法----线性探测法( $c=1$ )

- 基本思想：当发生冲突时，从冲突位置的下一个位置起，依次寻找空的散列地址。
- 探测序列：设关键字值 $key$ 的散列地址为 $h(key)$ ，闭散列表的长度为 $B$ ，则发生冲突时，寻找下一个散列地址的公式为：

$$h_i = (h(key) + d_i) \% B \quad (d_i = 1, 2, \dots, m-1)$$

- 示例：关键字取值集合为  $\{47, 7, 29, 11, 16, 92, 22, 8, 3\}$ ，散列函数为  $h(key) = key \% 11$ ，用线性探测法处理冲突，则散列表为：

0	1	2	3	4	5	6	7	8	9
11	22		47	92	16	3	7	29	8

- 堆积现象：在处理冲突的过程中出现的非同义词之间对同一个散列地址争夺的现象。

22                    3            3            3            29            8





## 5.8 散列技术 (Cont.)

冲突处理的方法----开放定址法

- 线性补偿探测法：当发生冲突时，寻找下一个散列地址的公式为：

$$h_i = (h(key) + d_i) \% B \quad (d_i = 1\text{c}, 2\text{c}, \dots)$$

- 二次探测法：当发生冲突时，寻找下一个散列地址的公式为：

$$h_i = (h(key) + d_i) \% B$$

$$(d_i = 1^2, -1^2, 2^2, -2^2, \dots, q^2, -q^2 \text{且 } q \leq B/2)$$

- 随机探测法：当发生冲突时，下一个散列地址的位移量是一个随机数列，即寻找下一个散列地址的公式为：

$$h_i = (h(key) + d_i) \% B$$

(其中， $d_1, d_2, \dots, d_{B-1}$ 是 $1, 2, \dots, B-1$ 的随机序列。)

注意：插入、删除和查找时，要使用同一个随机序列。





## 5.8 散列技术 (Cont.)

冲突处理的方法----开放定址法---线性探测法( $c=1$ )的的实现

### → 查找算法实现

#### ■ 存储结构定义

```
struct records{
```

```
    keytype key;
```

```
    fields other;
```

```
};
```

```
typedef records HASH[B];
```

#### ■ Search算法的实现

```
int Search(keytype k, HASH F)
{
    int locate=first= h(k),rehash=0;
    while((rehash<B)&&
          (F[locate].key!=empty)){
        if(F[locate].key==k)
            return locate;
        else
            rehash=rehash+1;
        locate=(first+rehash)%B
    }
    return -1;
}/*Search*/
```





## 5.8 散列技术 (Cont.)

冲突处理的方法----开放定址法---线性探测法( $c=1$ )的实现

### → 查找算法实现

- Insert算法的实现
- Delete算法的实现

```
void Delete(keytype k, HASH F)
{
    int locate;
    locate = Search(k, F);
    if( locate != -1)
        F[locate].key = deleted;
}/*Delete*/
```

```
void Insert(records R, HASH F)
{
    int locate = first=h(k), rehash= 0;
    while((rehash<B)&&
          (F[locate].key!=R.key)) {
        locate=(first+rehash)%B;
        if((F[locate].key==empty)||

            (F[locate].key==deleted))
            F[locate]=R;
        else
            rehash+=1; }
        if(rehash>=B)
            cout<<"hash table is full!";
    } /*Insert */
```





## 5.8 散列技术 (Cont.)

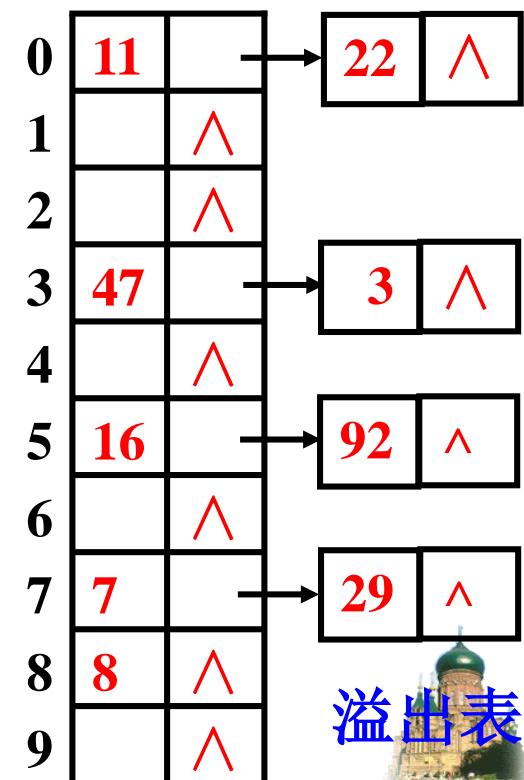
冲突处理的方法----带溢出表的内散列法

- 基本思想：扩充散列表中的每个桶，形成带溢出表的散列表。每个桶包括两部分：一部分是主表元素；另一部分或者为空或者由一个链表组成溢出表，其首结点的指针存入主表的链域。主表元素的类型与溢出表的类型相同。

- 示例：关键字取值集合为 {47, 7, 29, 11, 16, 92, 22, 8, 3}，散列函数为  $h(key)=key \% 11$ ，用带溢出表的内散列法处理冲突，则散列表为：

- 特点：

- 主表及其溢出表元素的散列地址相同。
- 空间利用率不高；





## 5.8 散列技术 (Cont.)

冲突处理的方法----带溢出表的内散列法

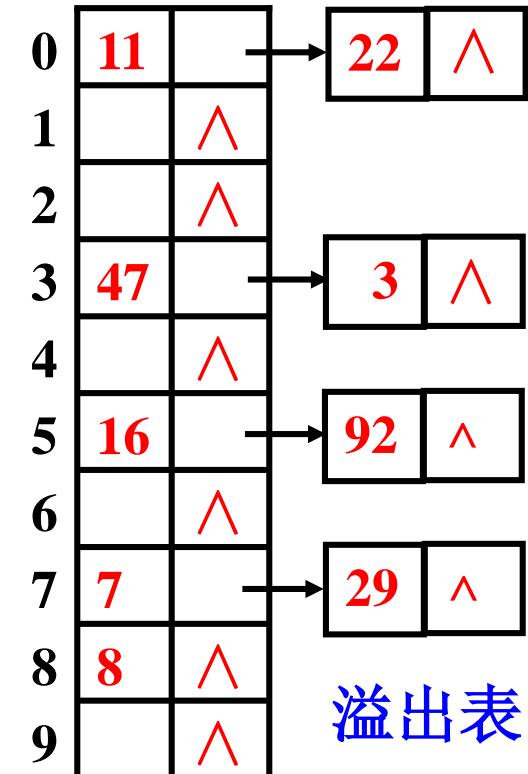
→ 查找算法的实现:

■ 存储结构的定义

```
typedef struct celltype{  
    records data;  
    celltype *next;  
} HASH[ B ];
```

■ 查找算法:

- 插入: 当发生冲突时, 把新元素插入溢出表;
- 查找: 要查看同一散列地址的主表和溢出表;
- 删除: 若被删除的是主表元素, 则要溢出表的首结点移入主表, 删掉首结点。



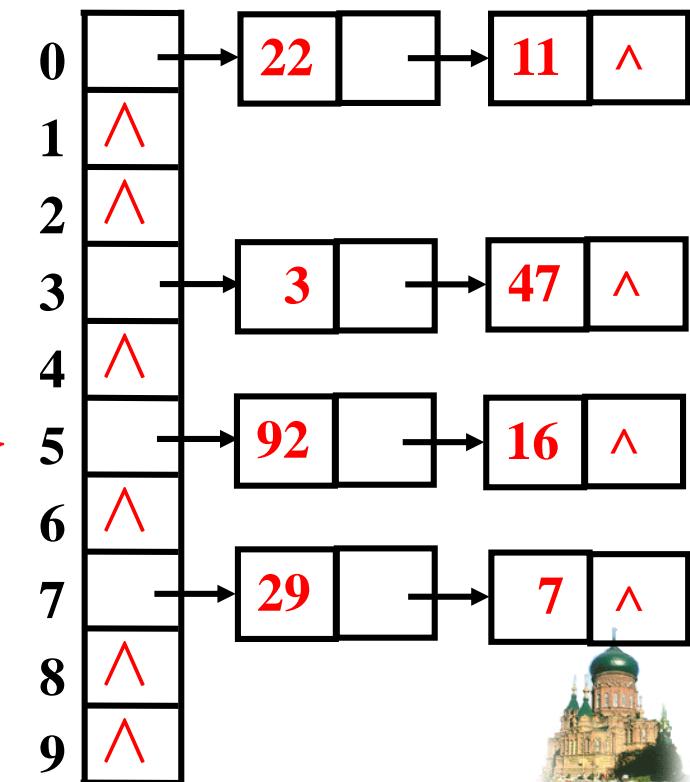


## 5.8 散列技术 (Cont.)

冲突处理的方法----拉链法（链地址法）

- 基本思想：将所有散列地址相同的记录，即所有同义词的记录存储在一个单链表中（称为同义词子表），在散列表中存储的是所有同义词子表的头指针。

- 开散列表：用拉链法处理冲突构造的散列表叫做开散列表。
- 示例：关键字取值集合为 {47, 7, 29, 11, 16, 92, 22, 8, 3}，散列函数为  $h(key)=key \% 11$ ，用链地址法处理冲突，则散列表为：
- 设  $n$  个记录存储在长度为  $B$  的散列表中，则同义词子表的平均长度为  $n / B$ 。





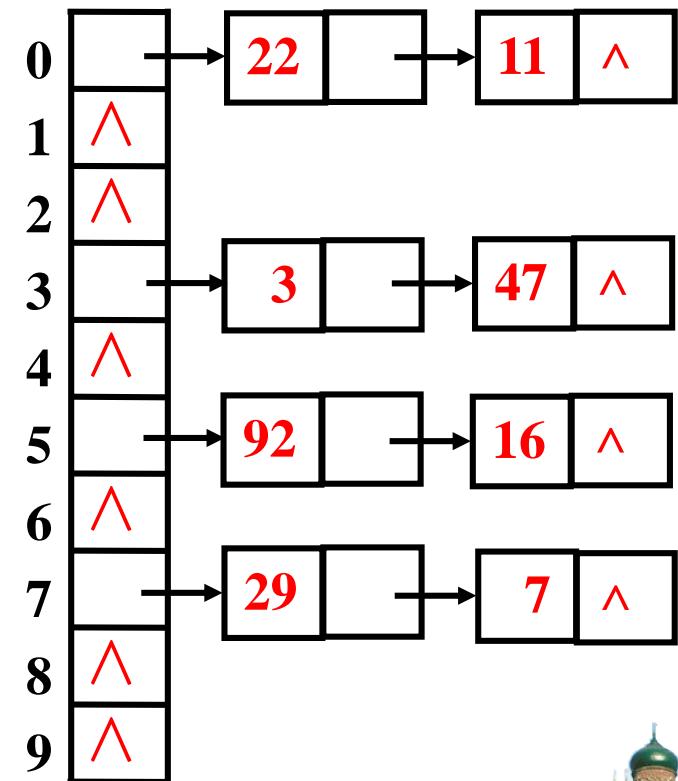
## 5.8 散列技术 (Cont.)

冲突处理的方法----拉链法（链地址法）

### ◆ 开散列表的实现

#### ■ 存储结构定义

```
struct celltype{  
    records data;  
    celltype *next;  
}; //链表结点类型  
  
typedef celltype *cellptr;  
//开散列表类型，B为桶数  
  
typedef cellptr HASH[B];
```





## 5.8 散列技术 (Cont.)

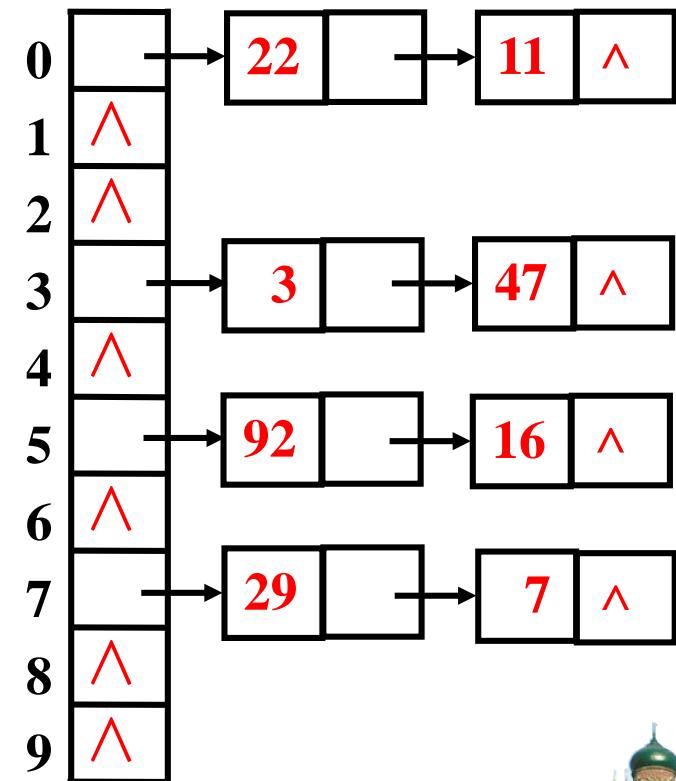
冲突处理的方法----拉链法（链地址法）

### ◆ 开散列表的实现

#### ■ 查找算法

*cellptr Search(keytype k, HASH F)*

```
{   cellptr bptr;  
    bptr=F[h(k)];  
    while(bptr!=Null)  
        if(bptr->data.key==k)  
            return bptr;  
        else  
            bptr=bptr->next;  
    return bptr;//没找到  
}//Search
```





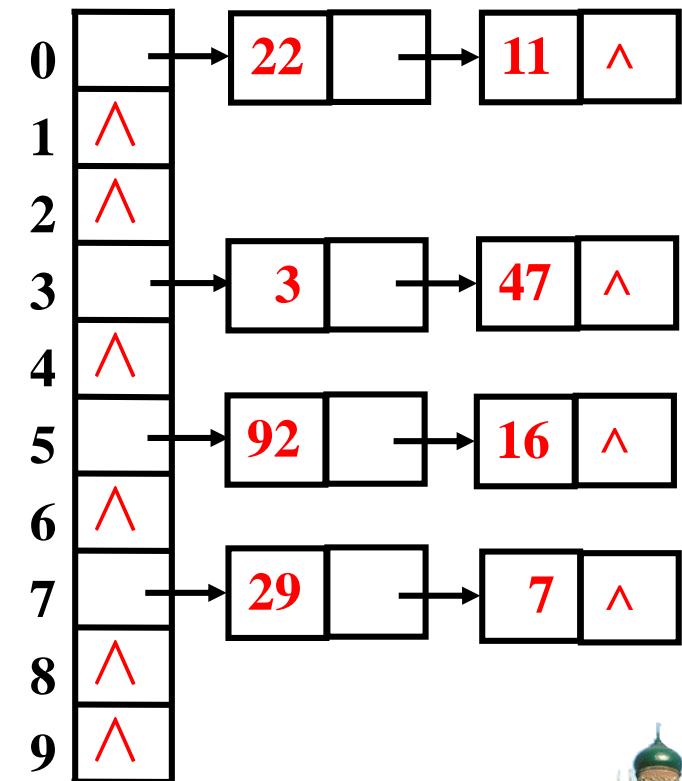
## 5.8 散列技术 (Cont.)

冲突处理的方法----拉链法（链地址法）

### ◆ 开散列表的实现

#### ■ 插入算法

```
void Insert(records R, HASH F)
{
    int bucket;
    cellptr oldheader;
    if( SEARCH(R.key,F)==Null){
        bucket=h(R.key);
        oldheader=F[bucket];
        F[bucket]=new celltype;
        F[bucket]->data=R;
        F[bucket]->next=oldheader;
    }
}//Insert
```



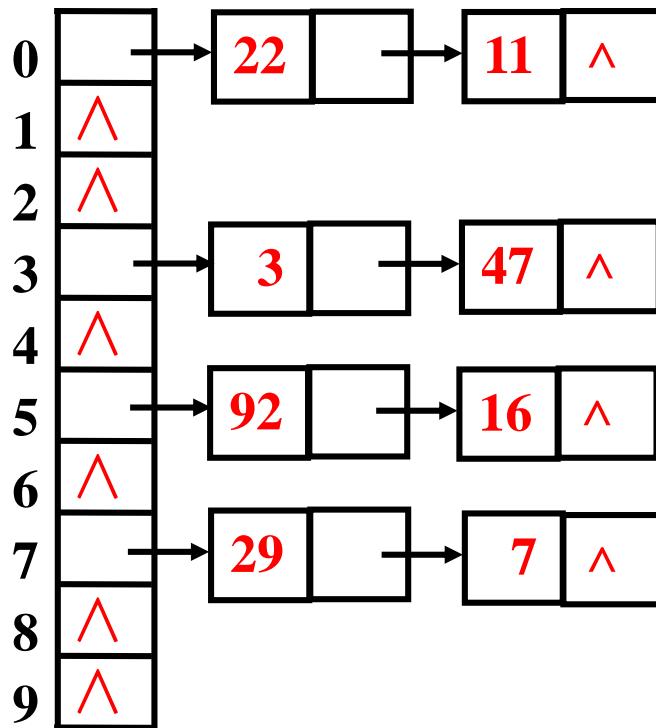


## 5.8 散列技术 (Cont.)

冲突处理的方法----拉链法  
(链地址法)

开散列表的实现

■ 删除算法



```
void Delete(keytype k, HASH F)
{ int bucket=h(k); celltype bptr, p;
  if(F[bucket] != NULL)//可能在表中
    if(F[bucket]->data.key==k){//首元素就是
      bptr= F[bucket];
      F[bucket]=F[bucket]->next;
      free (bptr);
    }else{//可能在中间或不存在
      bptr=F[bucket];
      while(bptr->next!=NULL)
        if(bptr->next->data.key==k){
          p=bptr->next;
          bptr->next=p->next;
          free( p);
        }else
          bptr=bptr->next;
    }
  } // Delete
```





## 5.8 散列技术 (Cont.)

### 散列查找的性能分析

- ▶ 由于冲突的存在，产生冲突后的查找仍然是给定值与关键码进行比较的过程。
- ▶ 在查找过程中，关键码的比较次数取决于产生冲突的概率。而影响冲突产生的因素有：
  - 散列函数是否均匀
  - 处理冲突的方法
  - 散列表的装载因子  
 $\alpha = \text{表中填入的记录数} / \text{表的长度}$





## 5.8 散列技术 (Cont.)

### 散列查找的性能分析

- ◆ 几种不同处理冲突方法的平均查找长度

ASL 处理冲突方法	查找成功时	查找不成功时
线性探测法	$\frac{1}{2} \left(1 + \frac{1}{(1-\alpha)^2}\right)$	$\frac{1}{2} \left(1 + \frac{1}{1-\alpha^2}\right)$
二次探测法	$-\frac{1}{\alpha} \ln(1+\alpha)$	$\frac{1}{1-\alpha}$
拉链法	$1 + \frac{\alpha}{2}$	$\alpha + e^{-\alpha}$





## 5.8 散列技术 (Cont.)

### 散列查找的性能分析

#### ◆ 开散列表与闭散列表的比较

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





# 5.8 散列技术 (Cont.)

一般的说，Hash函数可以简单的划分为如下几类：

1. 加法Hash;
2. 位运算Hash;
3. 乘法Hash;
4. 除法Hash;
5. 查表Hash;
6. 混合Hash;





# 5.8 散列技术 (Cont.)

## 一、加法Hash

输入元素一个一个的加起来构成最后的结果。

标准的加法Hash的构造如下：

```
static int addHash(String key, int prime)
{
    int hash, i;
    for (hash = key.length(), i = 0; i < key.length(); i++)
        hash += key.charAt(i);
    return (hash % prime);
} //prime是任意的质数，结果的值域为[0,prime-1]
```





## 5.8 散列技术（Cont.）

### 二、位运算Hash

通过利用各种位运算（常见的是移位和异或）来充分的混合输入元素。标准的旋转Hash的构造如下：

```
static int rotatingHash(String key, int prime)
{
    int hash, i;
    for (hash = key.length(), i = 0; i < key.length(); i++)
        hash = (hash<<4>>28) ^ key.charAt(i);
    return (hash % prime);
}
```





## 5.8 散列技术 (Cont.)

### 二、位运算Hash

先移位，然后再进行各种位运算是这种类型Hash函数的主要特点。以上的那段计算hash的代码还可以有如下几种变形：

- 1)  $\text{hash} = (\text{hash} \lll 5) \wedge \text{key.charAt}(i);$
- 2)  $\text{hash} += (\text{hash} \ll 10);$
- 3)  $\text{hash} \wedge= (\text{hash} \gg 6);$
- 4)  $\text{if}((i \& 1) == 0)\{\text{hash} \wedge= (\text{hash} \lll 7) \gg 3);\}$   
 $\text{else }\{\text{hash} \wedge= \sim((\text{hash} \lll 11) \gg 5));\}$
- 5)  $\text{hash} += \text{hash} \lll 5 \gg 2$
- 6)  $\text{hash} = \text{key.charAt}(i) + (\text{hash} \lll 6 \gg 16) ? \text{hash};$
- 7)  $\text{hash} \wedge= ((\text{hash} \lll 5) \gg 2));$





## 5.8 散列技术 (Cont.)

### 三 乘法Hash

这种类型的Hash函数利用了乘法的不相关性（乘法的这种性质，最有名的莫过于平方取头尾的随机数生成算法，虽然这种算法效果并不好）。比如，

```
static int bernstein(String key)
{
    int hash = 0;
    int i;
    int a = 63689;
    for(int i = 0; i < key.length(); i++)
        hash = hash * a + key.charAt(i);
    return hash;
}
```





## 5.8 散列技术 (Cont.)

### 四 除法Hash

除法和乘法一样，同样具有表面上看起来的不相关性。不过，因为除法太慢，这种方式几乎找不到真正的应用。需要注意的是，前面看到的hash的结果除以一个prime的目的只是为了保证结果的范围。

### 五 查表Hash

查表Hash最有名的例子莫过于CRC系列算法。虽然CRC系列算法本身并不是查表，但是，查表是它的一种最快的实现方式。

### 六 混合Hash

混合Hash算法利用了以上各种方式。各种常见的Hash算法，比如MD5、Tiger都属于这个范围。它们一般很少在面向查找的Hash函数里面使用。





## 5.8 散列技术（Cont.）

对Hash函数的建议如下：

字符串的Hash。最简单可以使用基本的乘法Hash，当乘数为33时，对于英文单词有很好的散列效果（小于6个的小写形式可以保证没有冲突）。复杂一点可以使用FNV算法（及其改进形式），它对于比较长的字符串，在速度和效果上都不错。





## 例题

- ◆ 将关键字序列 (7, 8, 30, 11, 18, 9, 14) 散列存储到散列表中，散列表的存储空间是一个下标从0开始的一维数组，散列函数为： $H(key) = (key * 3) \% 7$ ，处理冲突采用线性再散列法，要求装填（载）因子为0.7。
- ◆ (1) 请画出所构造的散列表。
- ◆ (2) 分别计算等概率情况下查找成功和查找不到成功的平均长度





(1) 概念装载因子: 装载因子是指所有关键字填充哈希表后饱和的程度, 它等于关键字总数/哈希表的长度。根据题意, 确定哈希表的长度为  $L = 7/0.7 = 10$ ; 构建的哈希表是下标为0~9的一维数组。根据散列函数可以得到如下散列函数值表。

$H(Key) = (key * 3) \text{ MOD } 7$ , 例如key=7时,  $H(7) = (7 \times 3) \% 7 = 21 \% 7 = 0$ , 其他关键字同理。

Key	7	8	30	11	18	9	14
H(Key)	0	3	6	5	5	6	0

采用线性探测再散列法处理冲突, 所构造的散列表为:

地址	0	1	2	3	4	5	6	7	8	9
关键字	7	14		8		11	30	18	9	





## 构造的散列表

地址	0	1	2	3	4	5	6	7	8	9
关键字	7	14		8		11	30	18	9	

(2) 等概率情况下查找成功平均查找长度:

可以根据构造过程求解:

key7一次就填入了表中，因此查找次数为1，同理8， 30， 11查找次数均为1； key18 进行了3次探测，探测位置分别是5， 6， 7， 因此查找次数为3； key9也是3次； key14 进行了两次探测，因此查找次数为2。

所以ASLsuccess=  $(1+1+1+1+3+3+2) / 7 = 12/7$ 。

## 查找成功比较次数

Key	7	8	30	11	18	9	14
Count	1	1	1	1	3	3	2





等概率情况下查找不成功的平均查找长度：在等概率情况下，查找不成功时的平均查找长度，定义为查找不成功时对关键字需要执行的平均比较次数。

计算查找不成功的次数就直接找关键字到第一个空位置的距离即可，但根据哈希函数地址为**MOD7**，因此计算出的位置只可能在**0~6**，只处理**7**个位置即可。

所以 $ASL_{unsuccess} = (3+2+1+2+1+5+4) / 7 = 18/7$ 。

关键字序列（7， 8， 30， 11， 18， 9， 14）

数组下标	0	1	2	3	4	5	6	7	8	9
数组元素	7	14		8		11	30	18	9	
元素插入次序	1	7		2		4	3	5	6	
不成功的比较次数	3	2	1	2	1	5	4	--	--	--





# 不适合散列解决的问题

- ◆ 需要排序的问题
- ◆ 多维数据的问题
- ◆ 包含动态数据的问题
- ◆ 数据不具有唯一关键字的问题

日常开发中最常见的散列算法应用就是通过 `md5` 函数对数据进行加密了，`md5` 就是一个散列函数，结合 `md5` 可以归纳出散列算法的一般特性：

- 从散列值不能反向推导出原始数据（所以散列算法也叫单向算法，不可逆）；
- 对输入数据非常敏感，哪怕原始数据只修改了一个比特位，最后得到的散列值也大不相同；
- 散列冲突的概率要很小，对于不同的原始数据，散列值相同的概率非常小；
- 散列算法的执行效率要尽量高效，针对较长的文本，也能快速地计算出散列值





# 散列算法的应用

## 1、场景一：安全加密

日常用户密码加密通常使用的都是 md5、sha 等散列函数，因为不可逆，而且微小的区别加密之后的结果差距很大，所以安全性更好。

## 2、场景二：唯一标识

比如 URL 字段或者图片字段要求不能重复，这个时候就可以通过对相应字段值做 md5 处理，将数据统一为 32 位长度从数据库索引构建和查询角度效果更好，此外，还可以对文件之类的二进制数据做 md5 处理，作为唯一标识，这样判定重复文件的时候更快捷。

## 3、场景三：数据校验

比如从网上下载的很多文件（尤其是 P2P 站点资源），都会包含一个 MD5 值，用于校验下载数据的完整性，避免数据在中途被劫持篡改。





### 4、场景五：散列函数

前面已经提到，PHP 中的 md5、sha1、hash 等函数都是基于散列算法计算散列值

### 5、场景五：负载均衡

对于同一个客户端上的请求，尤其是已登录用户的请求，需要将其会话请求都路由到同一台机器，以保证数据的一致性，这可以借助散列算法来实现，通过用户 ID 尾号对总机器数取模（取多少位可以根据机器数定），将结果值作为机器编号。

### 6、场景六：分布式缓存

分布式缓存和其他机器或数据库的分布式不一样，因为每台机器存放的缓存数据不一致，每当缓存机器扩容时，需要对缓存存放机器进行重新索引（或者部分重新索引），这里应用到的也是散列算法的思想。





# 例题 (cont.)

百度面试题：

搜索引擎会通过日志文件把用户每次检索使用的所有检索串都记录下来，每个查询串的长度为1-255字节。

假设目前有一千万个记录（这些查询串的重复度比较高，虽然总数是1千万，但如果除去重复后，不超过3百万个。一个查询串的重复度越高，说明查询它的用户越多，也就是越热门。），请你统计最热门的10个查询串，要求使用的内存不能超过1G。

问题分析：

要统计最热门查询，首先就是要统计每个查询串出现的次数，然后根据统计结果，找出前**10**。





### 第一步：查询串统计

查询串统计可以选择：

#### 1、直接排序法

对日志里所有查询串都进行排序，然后遍历排好序的查询串，统计每个查询串出现的次数了。

但是题目中有明确要求，那就是内存不能超过**1G**，一千万条记录，每条记录是**255Byte**，很显然要占据**2.375G**内存，这个条件就不满足要求了。可以用外排序中的归并算法，效率为**O(nlogn)**

排完序之后再对已经有序的查询串文件进行遍历，统计每个查询串出现的次数，再次写入文件中。

综合分析一下，排序的时间复杂度是**O(nlogn)**，而遍历的时间复杂度是**O(n)**，因此该算法的总体时间复杂度就是**O(n+nlogn)=O(nlogn)**。





### 2、Hash 表

在方法1中，采用了排序的办法来统计每个**查询串**出现的次数，时间复杂度是 $O(n \log n)$ ，能否有更好的方法来存储，而时间复杂度更低呢？

题目中说明了，虽然有一千万个查询串，但是由于重复度比较高，因此事实上只有**300万**的查询串，每个查询串**255Byte**，因此可以考虑把他们都放进内存中去，而现在只是需要一个合适的数据结构，**Hash表**绝对是优先的选择，因为**Hash表**的查询速度非常的快，几乎是 $O(1)$ 的时间复杂度。

- 算法：维护一个**Key**为查询串字串，**count**为该查询串出现次数，每次读取一个查询串，如果该字串不在表中，那么加入该字串，并且将**count**值设为1；如果该字串在表中，那么将该字串的计数加一即可。最终在 $O(n)$ 的时间复杂度内完成了对该海量数据的处理。
- 该方法与算法1相比：在时间复杂度上提高了一个数量级，为 $O(n)$ ，需要读取**IO**数据文件一次。





## 第二步：找出前10

### 算法一：普通排序

排序算法的时间复杂度是 $O(n \log n)$ ，在本题目中，三百万条记录，用1G内存是可以存下的。

### 算法二：部分排序

题目要求是求出前10，因此没有必要对所有的查询串都进行排序，只需要维护一个10个大小的数组，初始化放入10个查询串，按照每个查询串的统计次数由大到小排序，然后遍历这300万条记录，每读一条记录就和数组最后一个查询串对比，如果小于这个查询串，那么继续遍历，否则，将数组中最后一条数据淘汰，加入当前的查询串。最后当所有的数据都遍历完毕之后，那么这个数组中的10个查询串便是我们要找的前10了。

算法的最坏时间复杂度是 $n * k$ ，其中 $k$ 是指要取得记录多少。

### 算法三：堆

在算法二中，时间复杂度由 $O(n \log n)$ 优化到 $O(n * k)$ 。借助堆结构，可以在 $\log$ 量级的时间内查找和调整/移动。算法可以改进为这样，维护一个 $k$ (该题目中是10)大小的小根堆，然后遍历300万的查询串，分别和根元素进行对比。采用堆数据结构，算法三，最终的时间复杂度就降到了 $O(n \log k)$ 。





先用**Hash**表统计每个查询串出现的次数， $O(n)$ ；然后  
第二步、采用堆数据结构找出前 10， $O(n' \log k)$ 。时间复杂度是： $O(n) + O(n' \log k)$ 。（n为1000万，n'为300万）。

### 问题：关于**B+Tree**索引和**Hash**索引的优劣以及使用场景的区别

与业务场景有关，如果只选择一个数据，hash更快，但是数据库中经常会选择多条数据，这时候因为B+树索引有序，并且又有链表系总量，查询效率比hash就要快的多了

