

数据结构与算法

Data Structures and
Algorithms

苗东菁



海量数据计算研究中心



哈工大计算机科学与技术学院

第5章 查找结构





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**适合的结构和方法解决实际问题。





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⁺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⁺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⁺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：**
- 数据集合 → 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？
- 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：假设待查的记录是查找表中第 i 个记录，其为第 i 个位置的概率为 p_i , $\sum p_i = 1$, 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**





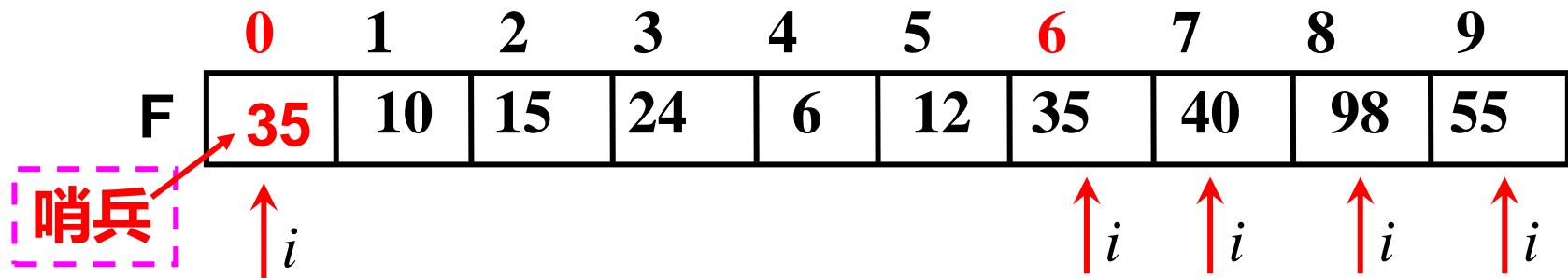
5.2 线性查找(Cont.)

→ 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 基本概念和术语(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 )
```

```
        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

```
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 ;  
}
```

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





5.3 折半查找(二分查找)

→ 折半查找 (也称二分查找) 的要求(适用条件):

- 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

- $F[1].key \leq F[2].key \leq F[3].key \leq \dots \leq F[last].key$
- 或 $F[1].key \geq F[2].key \geq F[3].key \geq \dots \geq F[last].key$
- 注意: 折半查找只适合于静态查找!

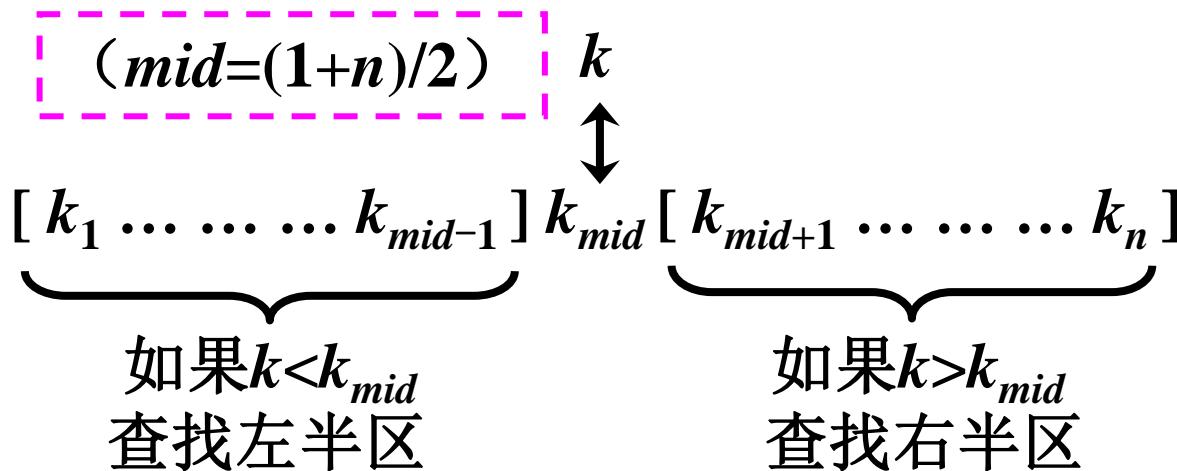




5.3 折半查找(Cont.)

→ 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。

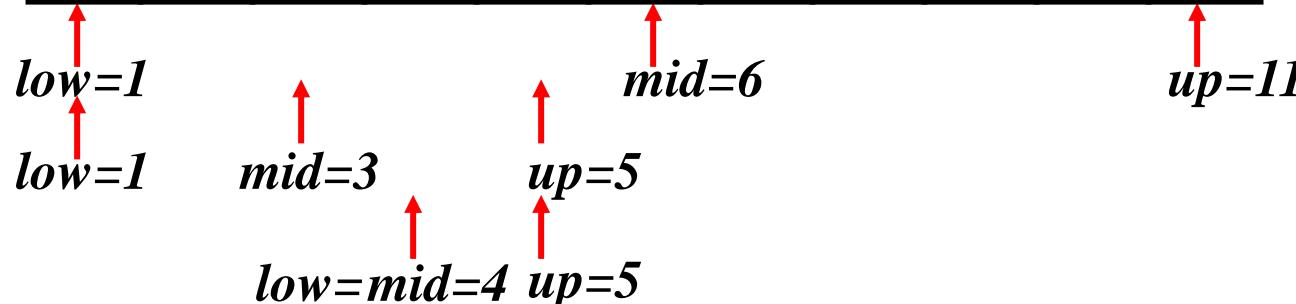




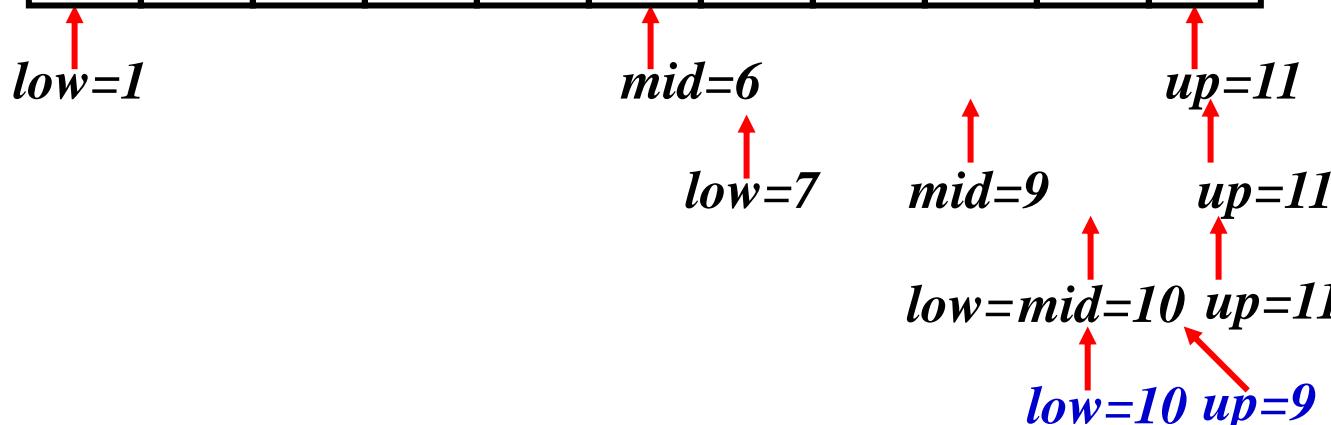
5.3 折半查找(Cont.)

◆ 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 low, up 分别表示查找范围的上、下界，初始时 $low = 1, up = last$ ；
- 2. 折半：令 $mid = (low+up)/2$ ，取查找范围中间位置元素下标；
- 3. 比较： k 与 $F[mid].key$
 - 3.1 若 $F[mid].key == k$ ，查找成功，返回 mid ；
 - 3.2 若 $F[mid].key > k$ ， low 不变，调整 $up = mid - 1$ ，查找范围缩小一半；
 - 3.3 若 $F[mid].key < k$ ，调整 $low = mid + 1$ ， up 不变，查找范围缩小一半；
- 4. 重复 2 ~ 3 步。当 $low > up$ 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. **初始化**: 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 : $O(\log_2 n)$ */





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。

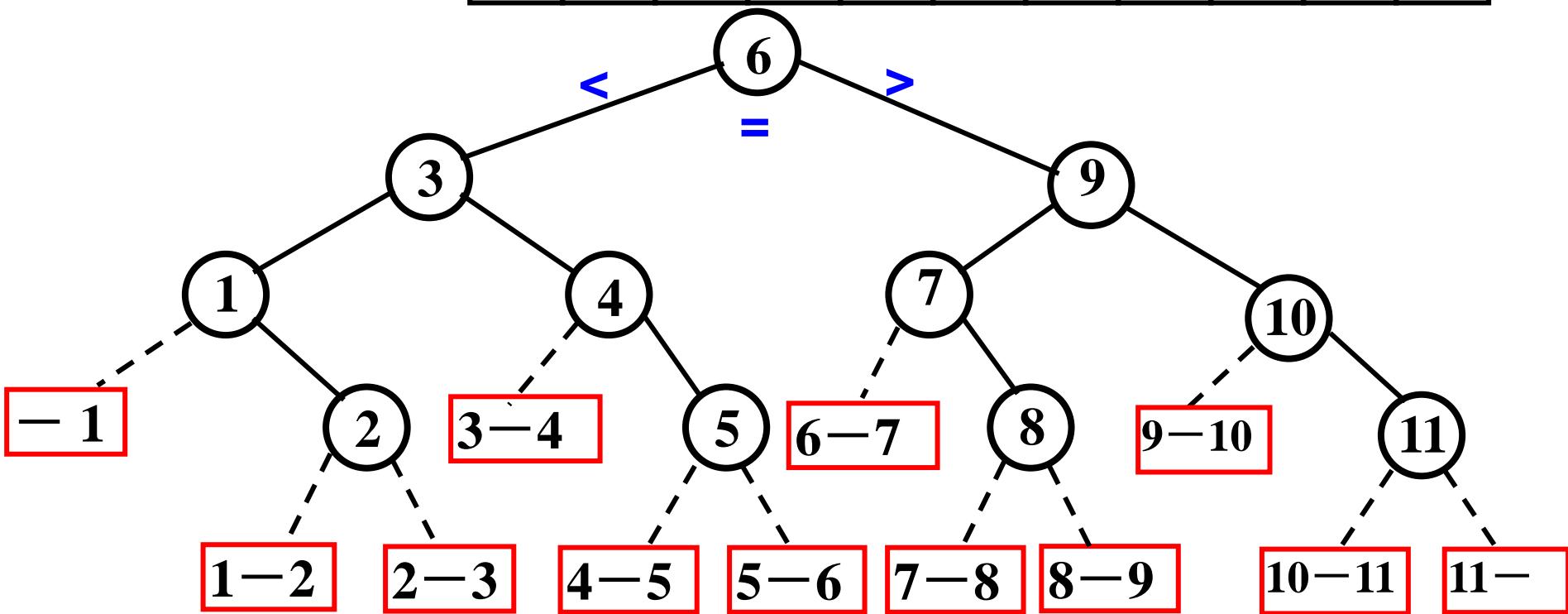




5.3 折半查找(Cont.)

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

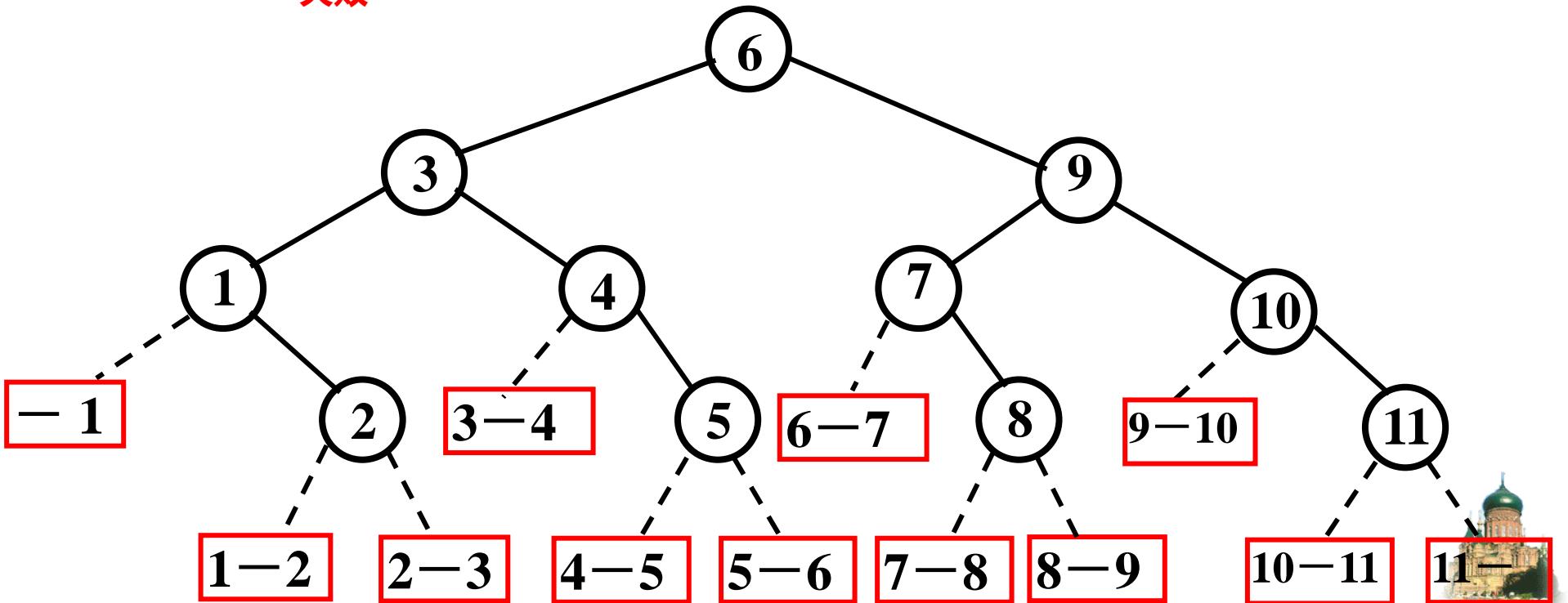




5.3 折半查找(Cont.)

◆ 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
- $ASL_{\text{成功}} = (1*1 + 2*2 + 3*4 + 4*4) / 11 = 33/11 = 3$
- $ASL_{\text{失败}} = (3*4 + 4*8) / 12 = 44/12$

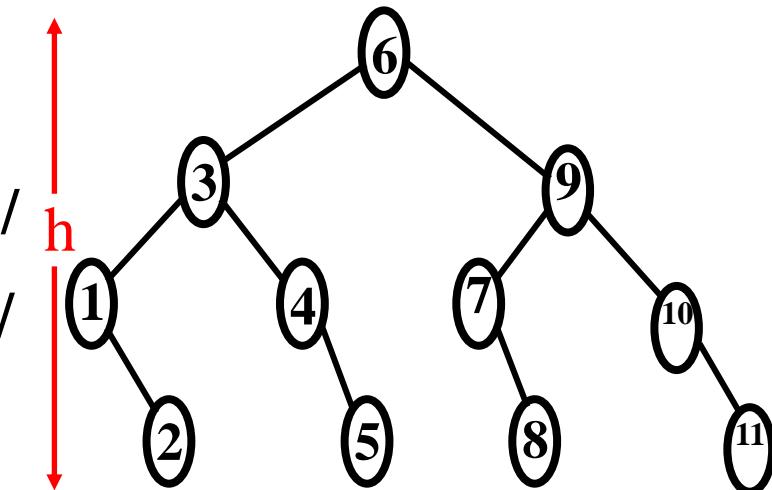




5.3 折半查找(Cont.)

折半查找的判定树高度

$$\begin{aligned} ASL_{bs} &= \sum_{i=1}^n p_i \cdot c_i \quad /* p_i = 1/n */ \\ &= 1/n \cdot \sum_{j=1}^h j \cdot 2^{j-1} \quad /* S=2S - S */ \\ &= (n+1)/n[\log_2(n+1)-1] \end{aligned}$$



- 当n很大时, $ASL_{bs} \approx \log_2(n+1)-1$ 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。





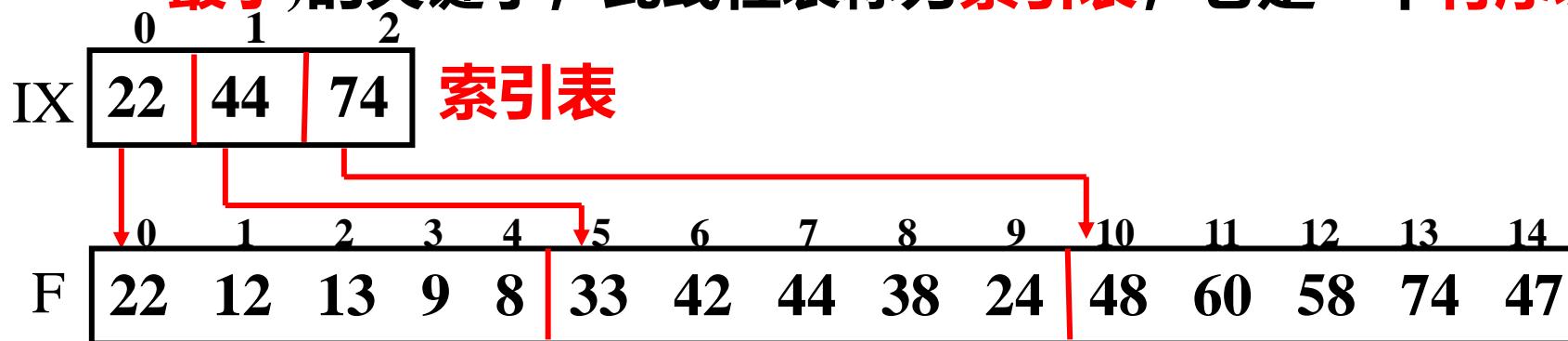
5.4 分块查找----线性查找+折半查找

分块查找的基本思想

■ **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。





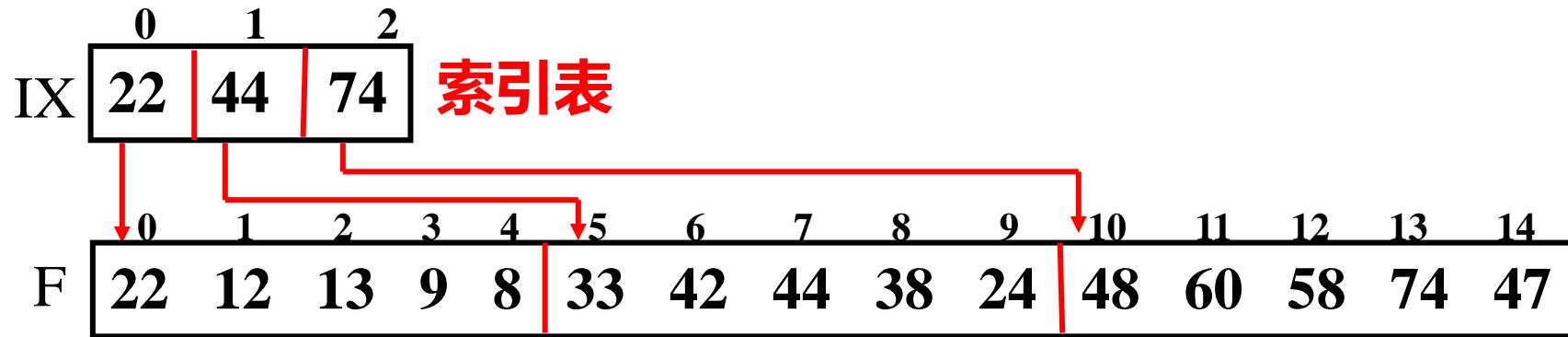
5.4 分块查找(Cont.)

- 分块查找算法的要点：假设在带索引的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，

$$\text{■ 索引表的 } ASL_{ix} = \sum_{i=1}^b p_i \cdot c_i = \frac{1}{b} \sum_{i=1}^b i$$

■ 块内的平均查找长度： $ASL_{blk} = \sum_{j=1}^L p_j \cdot c_j = \frac{1}{L} \sum_{j=1}^L j$

■ 所以分块查找平均长度为：

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

令 $ASL'(L)=0$ ，知当 $L=\sqrt{n}$ 时， $ASL(L) = \sqrt{n} + 1$ (最小值)

■ 另外，索引表为有序表，可采用折半查找





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。

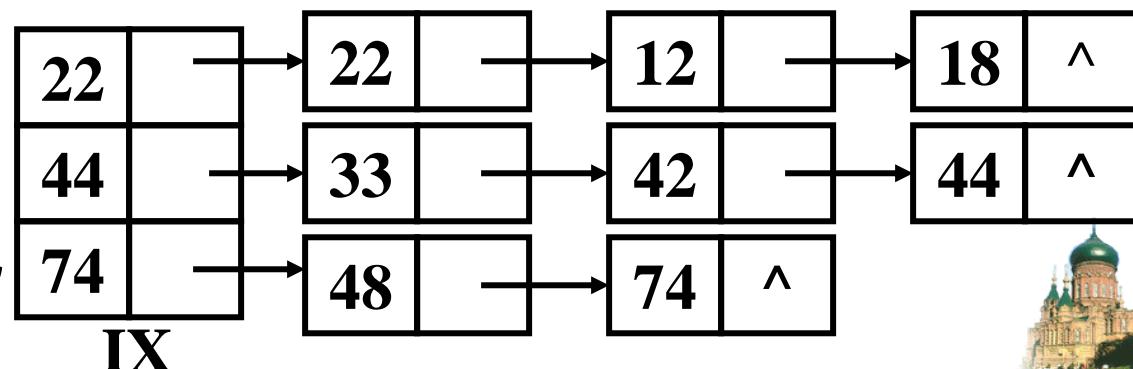
→ 动态环境的分块查找

- ## ■ 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



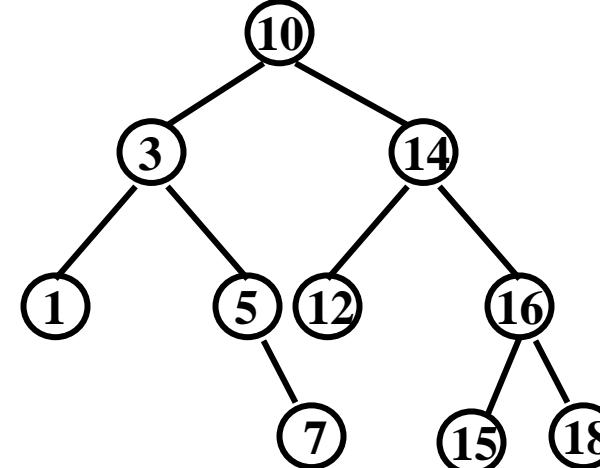
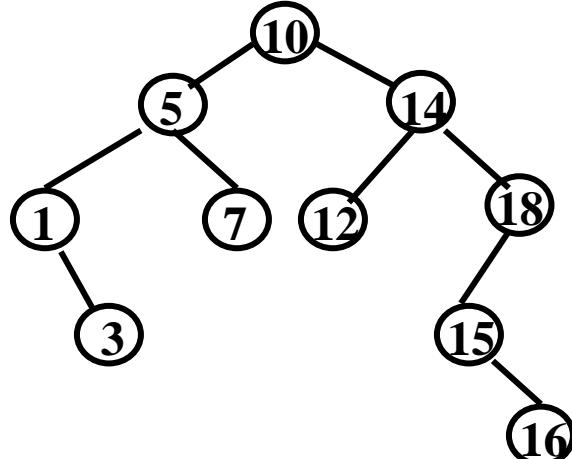


5.5 二叉查找树BST

→ 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。

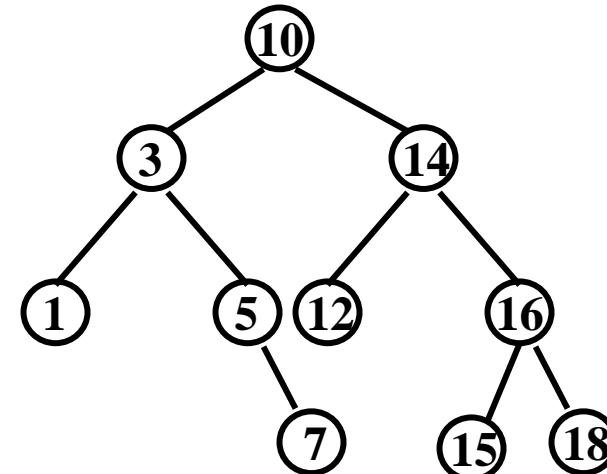
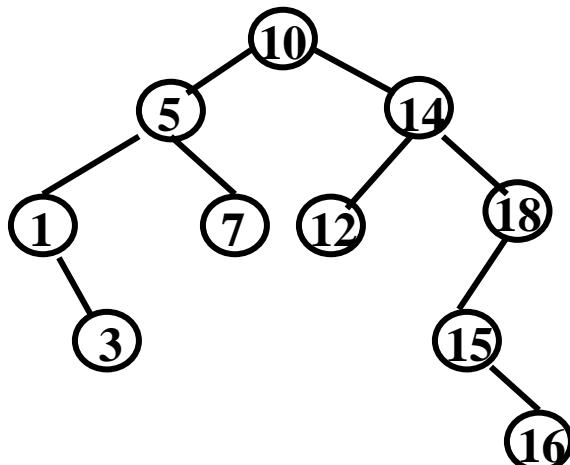




5.5 二叉查找树BST (Cont.)

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。





5.5 二叉查找树BST (Cont.)

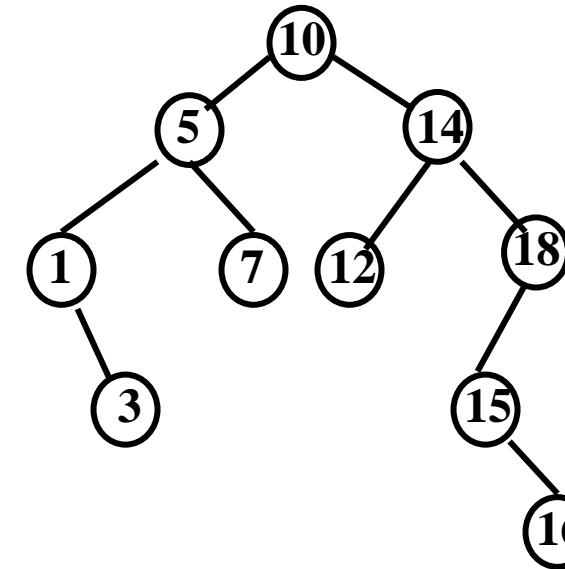
→ 二叉查找树的结构特点：

■ 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 ;
```



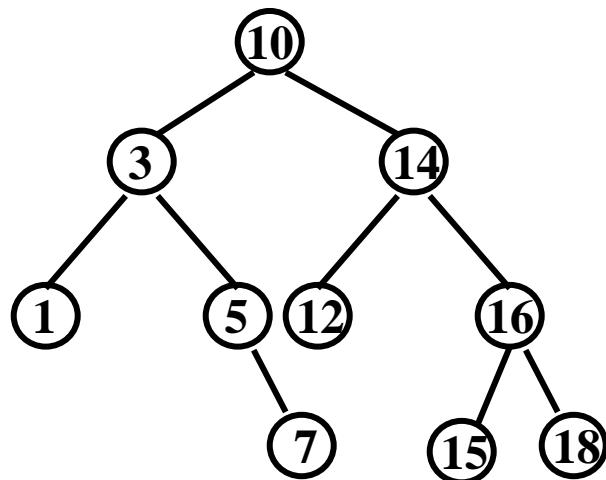


5.5 二叉查找树BST (Cont.)

→ 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



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？

1. **随机输入各结点**

2. **在建立、插入和删除各结点过程中平衡相关结点的左、右子树高度，**



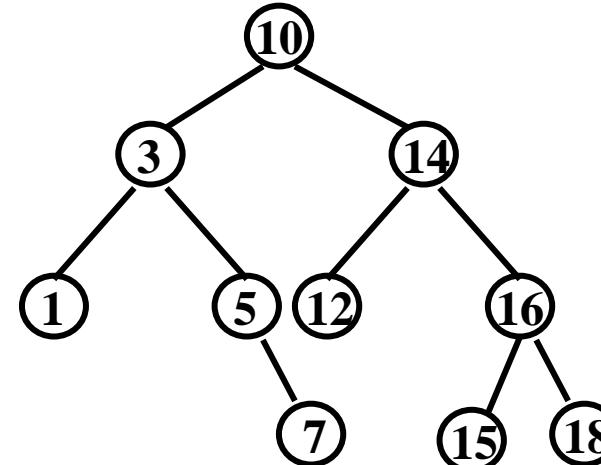
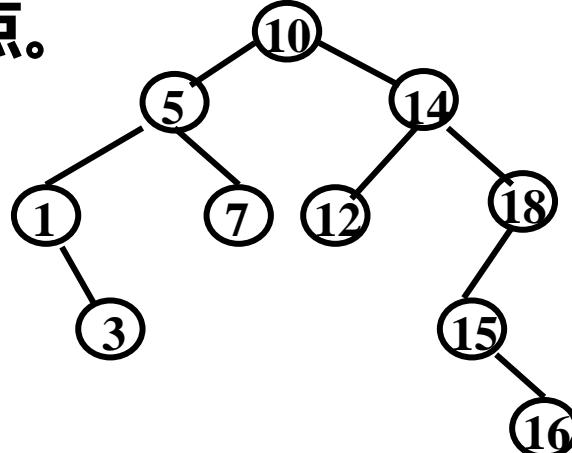


5.5 二叉查找树BST(Cont.)

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。





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，则只需重接p的左子树；
若结点p只有右子树，则只需重接p的右子树；
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 若结点p的右孩子无左子树，
则将s的右子树接到par的右子树上；
否则，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)
/*删除F的最小结点*/
{ records tmp ; BST p ;
  if (F->lchild==NULL){//是最小元
    p = F ;
    tmp = F->data ;
    F = F->rchild ; //右链继承
    delete p ;
    return tmp ;
  }
  else//左子树不空， 最小结点在左子树上
  return//在左子树上递归地删除
  (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 // ( k==F->data.key
    if ( F->rchild == NULL )
      F = F->lchild;//右链继承,包括叶
    else if( F->lchild == NULL )
      F = F->rchild;//右链继承,包括叶
    else //有两棵子树
      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

→ AVL树(Balanced Binary Tree or Height-Balanced Tree)

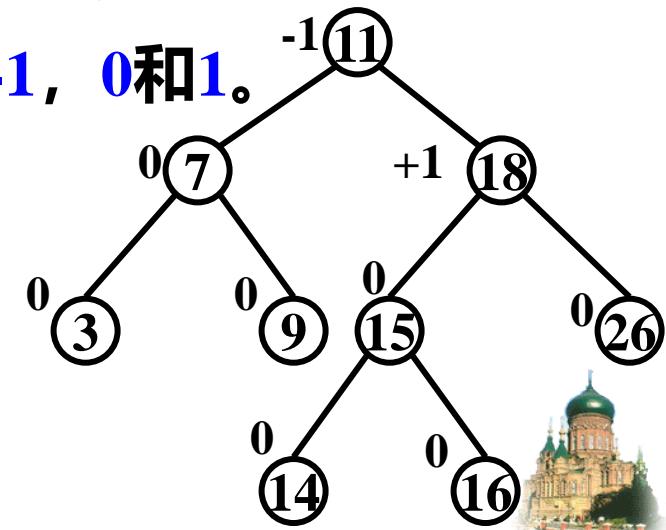
- AVL树或者是空二叉树，或者是具有如下性质的BST：
 - 根结点的左、右子树高度之差的绝对值不超过1；
 - 且根结点左子树和右子树仍然是AVL树。

→ 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

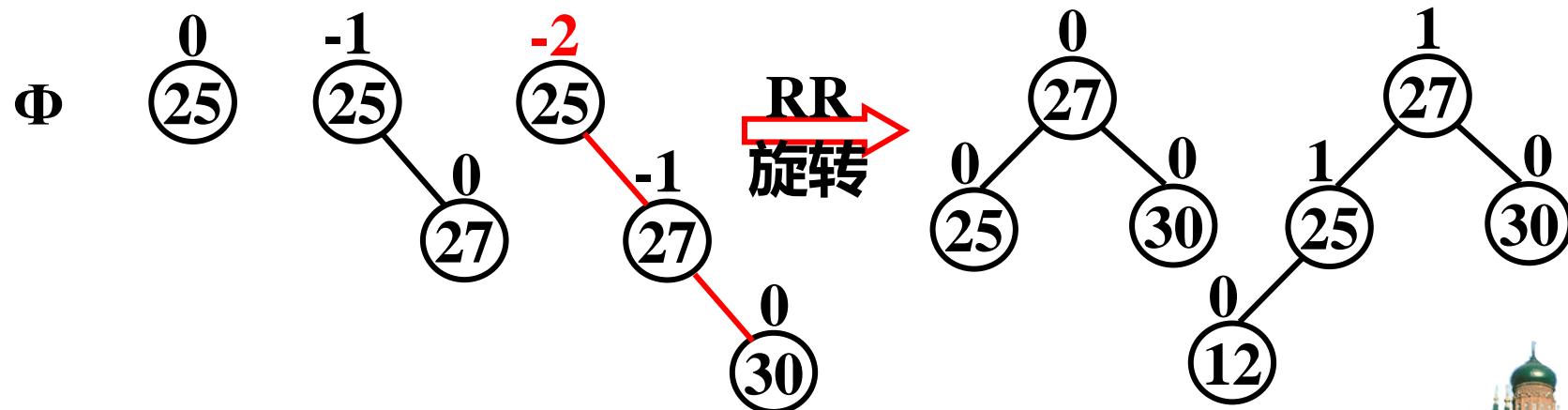




5.6 AVL树(Cont.)

→ 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
- 我们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。

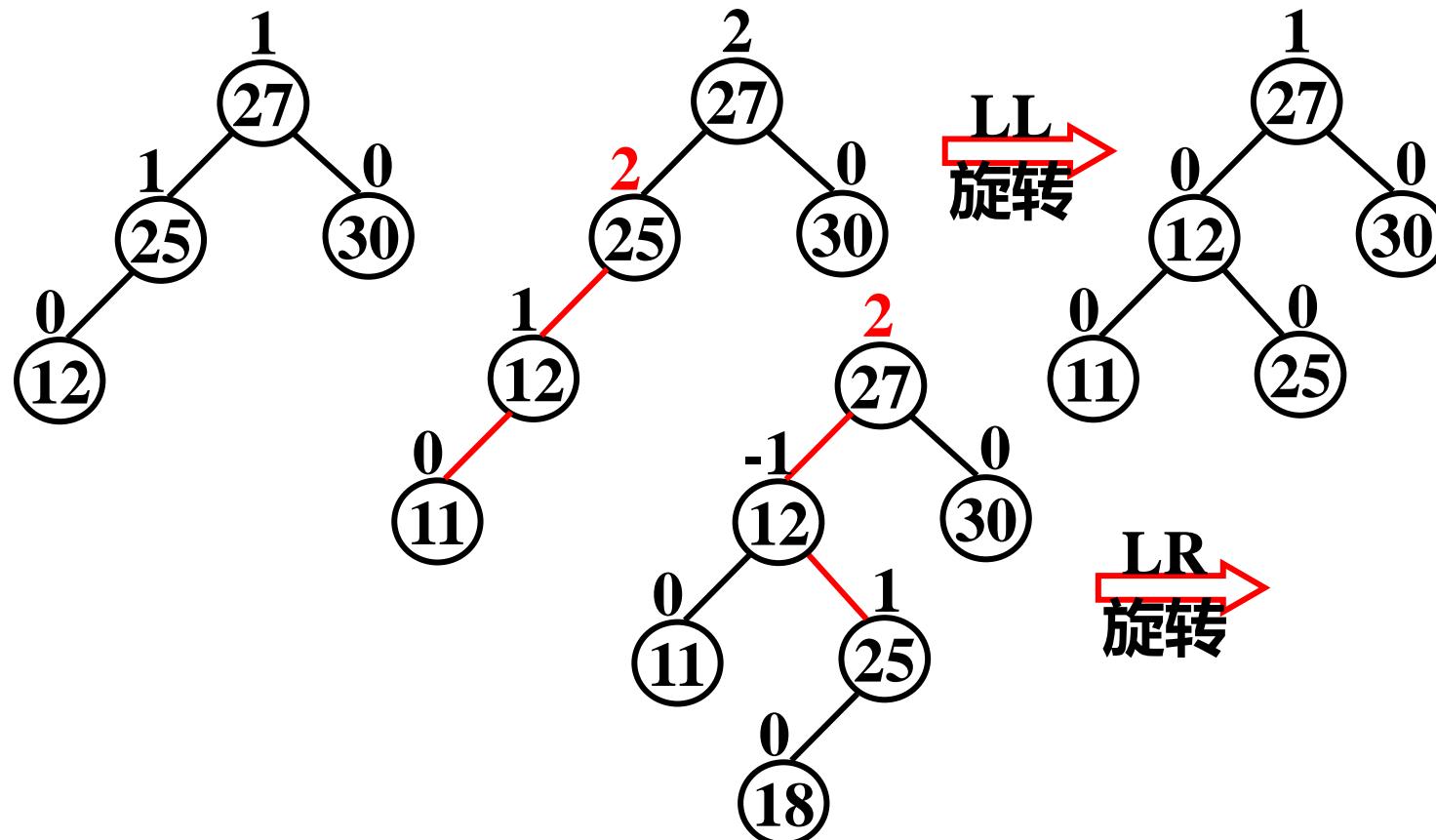




5.6 AVL树(Cont.)

→ 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。

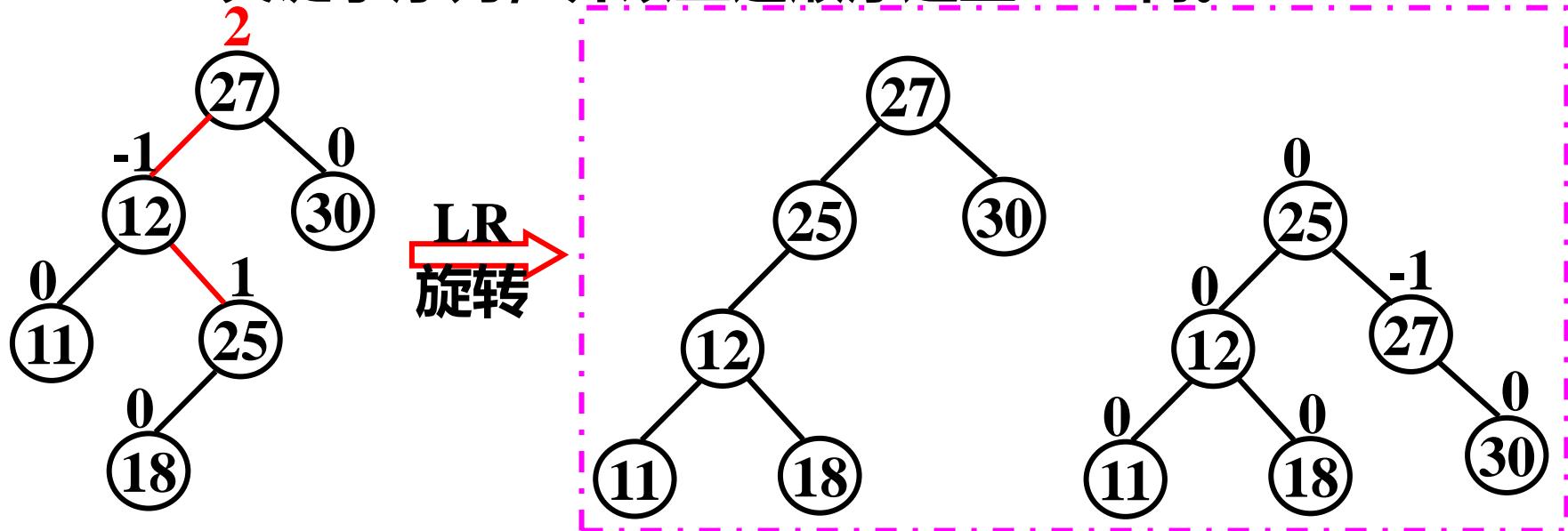




5.6 AVL树(Cont.)

→ 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。

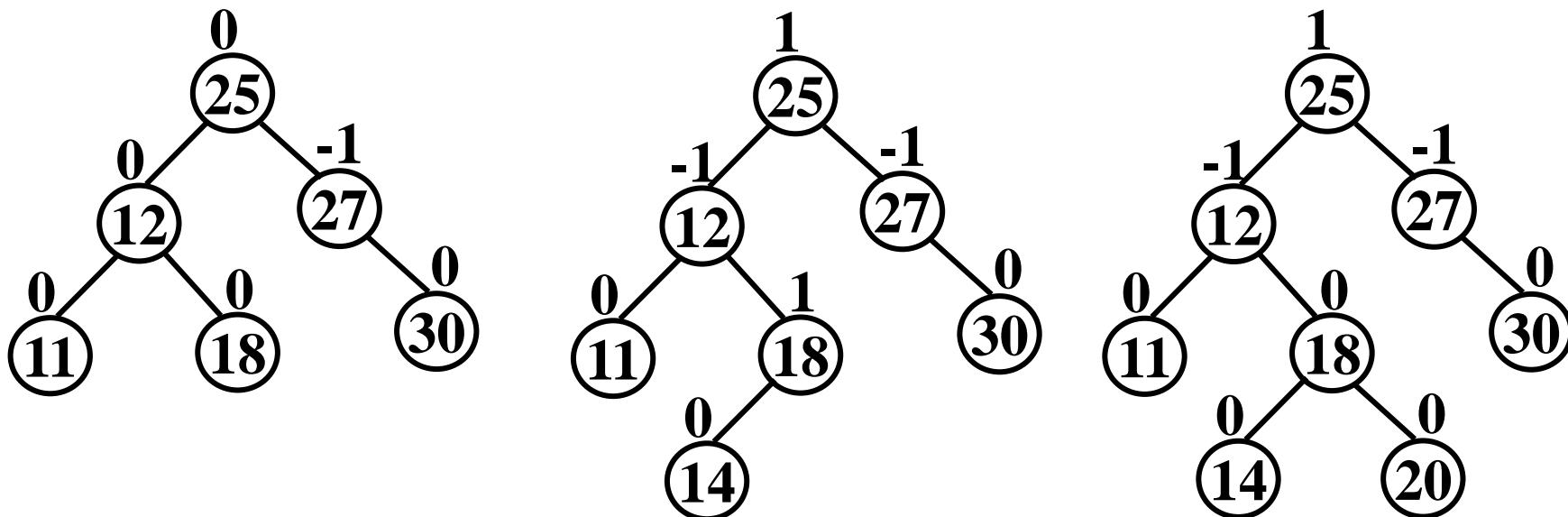




5.6 AVL树(Cont.)

→ 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。

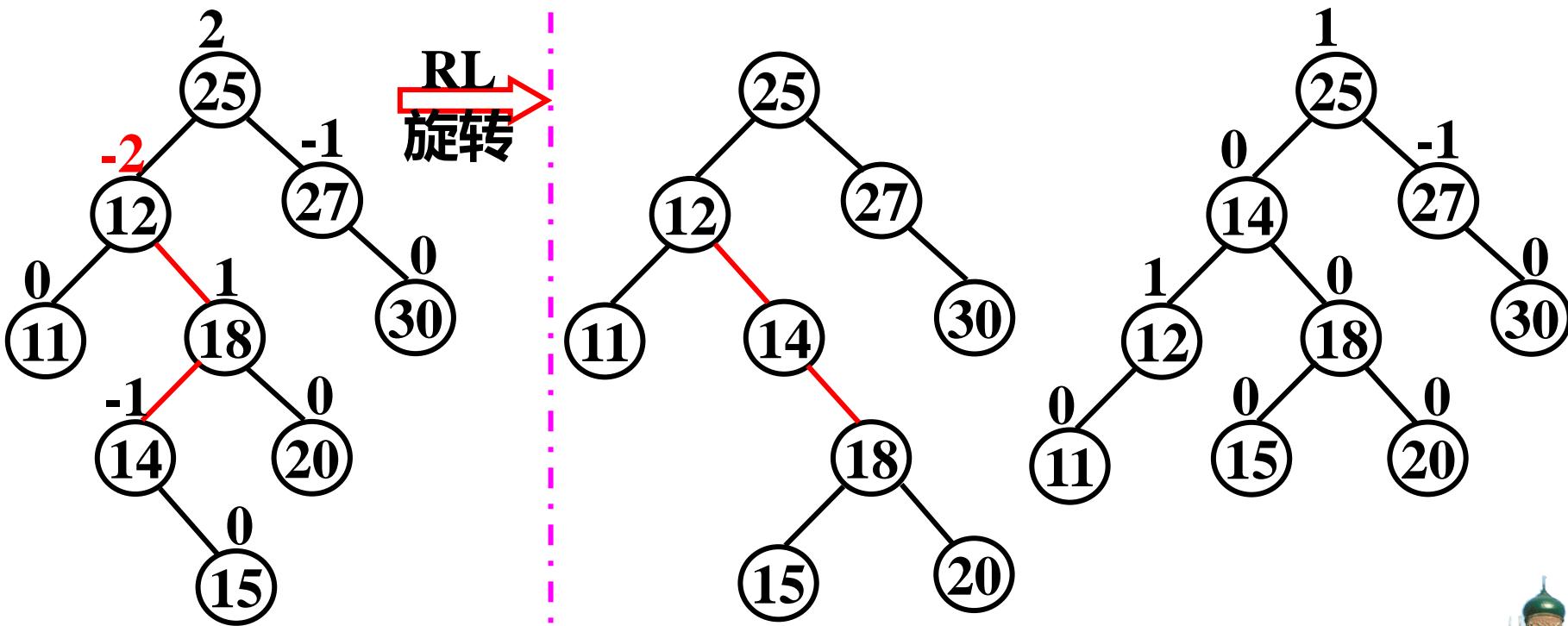




5.6 AVL树(Cont.)

→ 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。

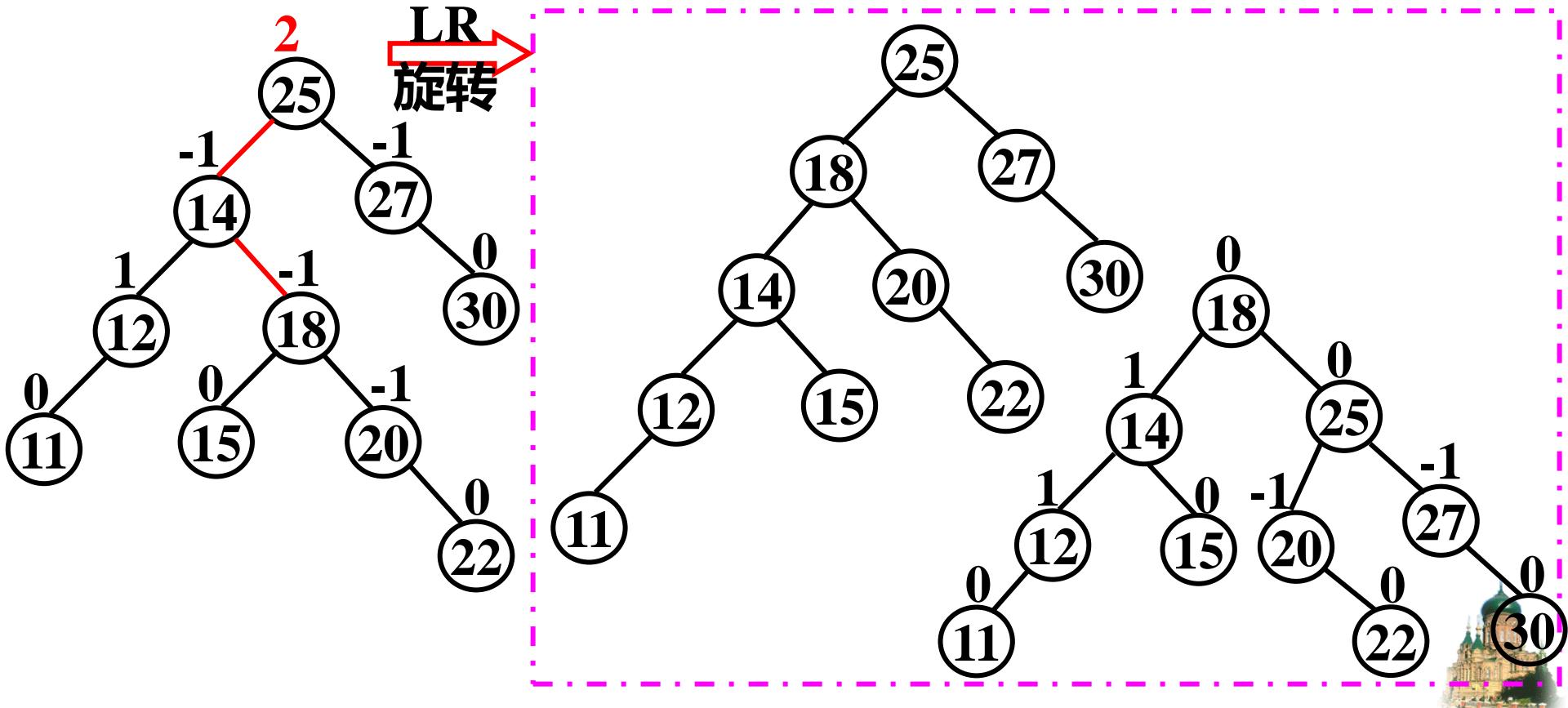




5.6 AVL树(Cont.)

→ 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。





5.6 AVL树(Cont.)

→ AVL树的平衡化处理

- 在一棵AVL树上插入结点可能会破坏树的平衡性，需要平衡化处理恢复平衡，且保持BST的结构性质。
- 若用Y表示新插入的结点，A表示离新插入结点Y最近的，且平衡因子变为 ± 2 的祖先结点。
- 可以用4种旋转进行平衡化处理：
 - ① **LL型**：新结点Y被插入到A的左子树的左子树上（顺）
 - ② **RR型**：新结点Y被插入到A的右子树的右子树上（逆）
 - ③ **LR型**：新结点Y被插入到A的左子树的右子树上（逆、顺）
 - ④ **RL型**：新结点Y被插入到A的右子树的左子树上（顺、逆）

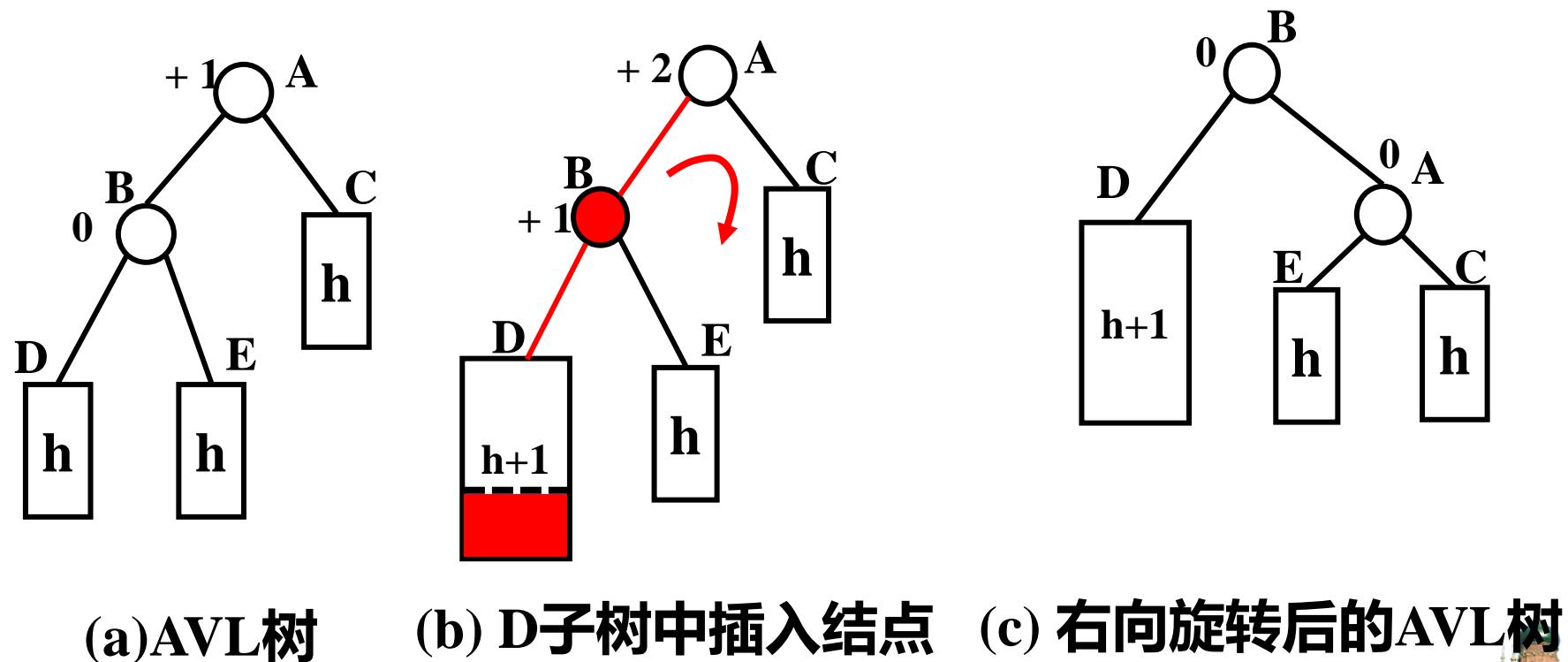




5.6 AVL树(Cont.)

AVL树的平衡化处理

■ LL型：新结点Y被插入到 A 的左子树的左子树上（顺）

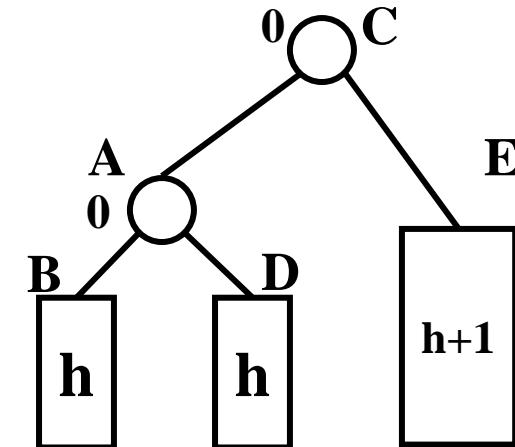
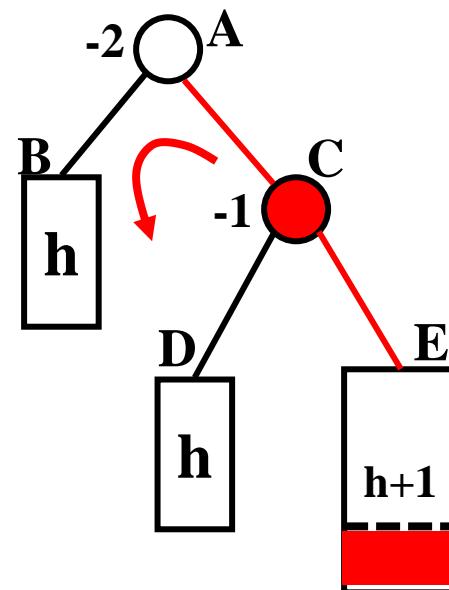
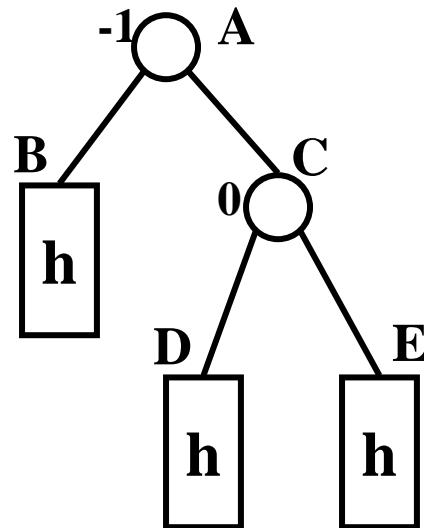




5.6 AVL树(Cont.)

→ AVL树的平衡化处理

■ RR型：新结点Y被插入到A的右子树的右子树上（逆）



(a)AVL树

(b)E子树中插入结点

(c)左向旋转后的AVL树

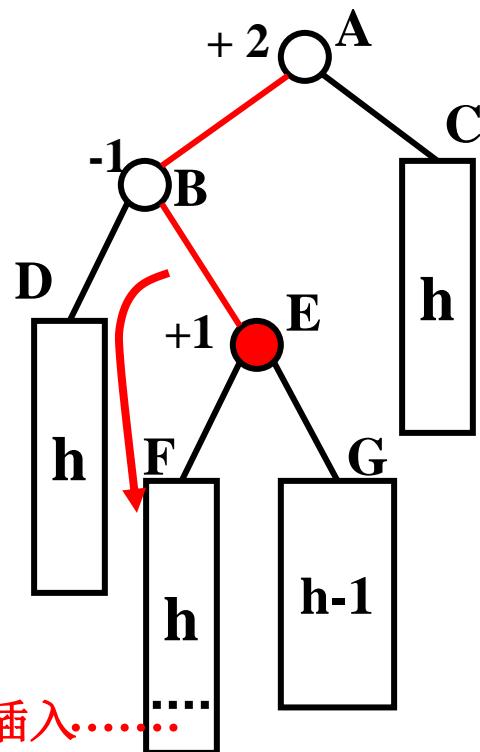




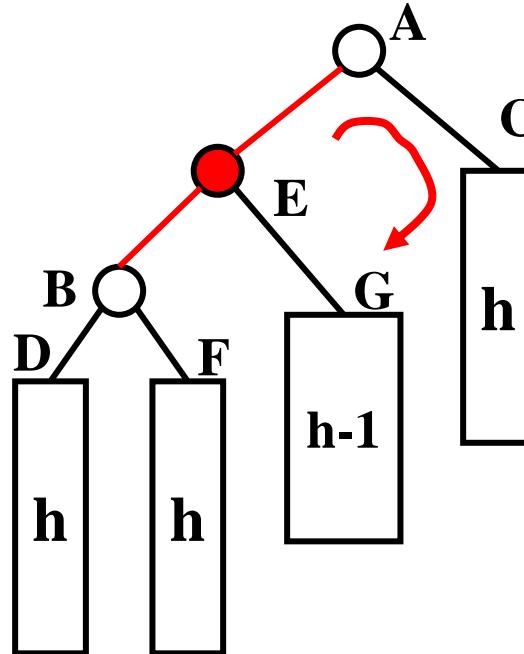
5.6 AVL树(Cont.)

AVL树的平衡化处理

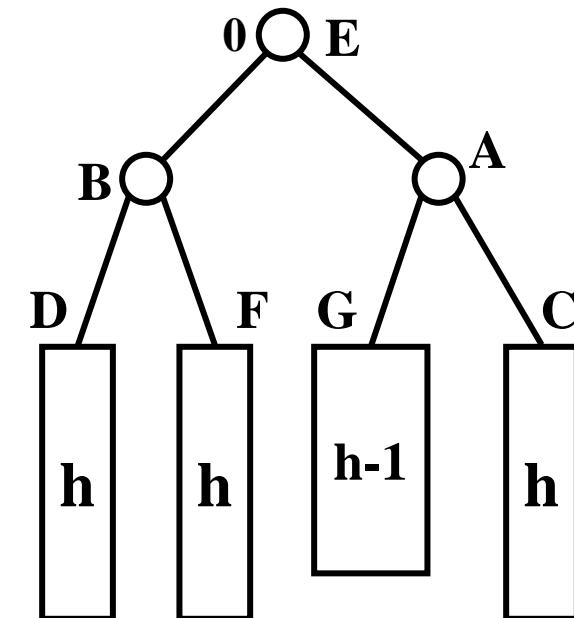
■ LR型：新结点Y被插入到A的左子树的右子树上(逆,顺)



(a) F子树插入结点
高度变为h



(b) 绕E，将B
逆时针转后



(c) 绕E，将A
顺时针转后

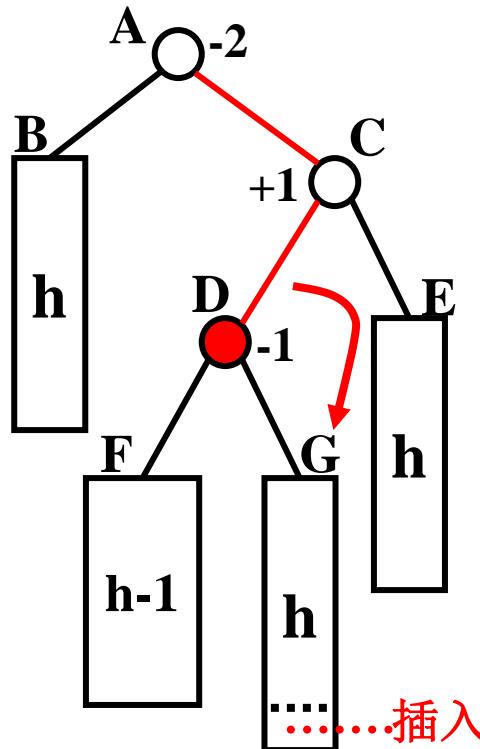




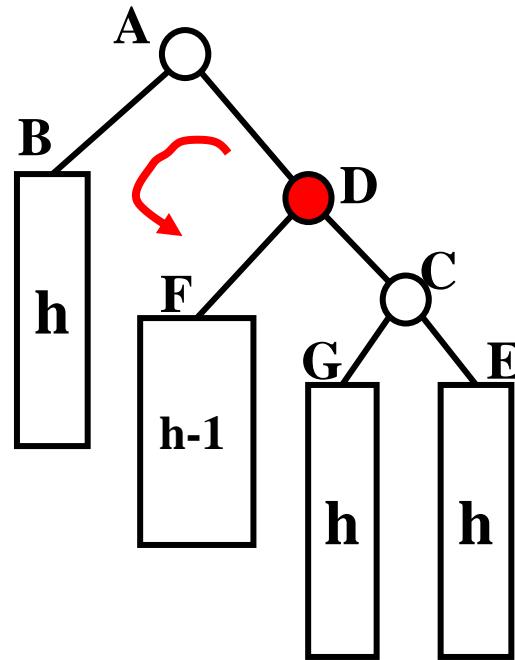
5.6 AVL树(Cont.)

AVL树的平衡化处理

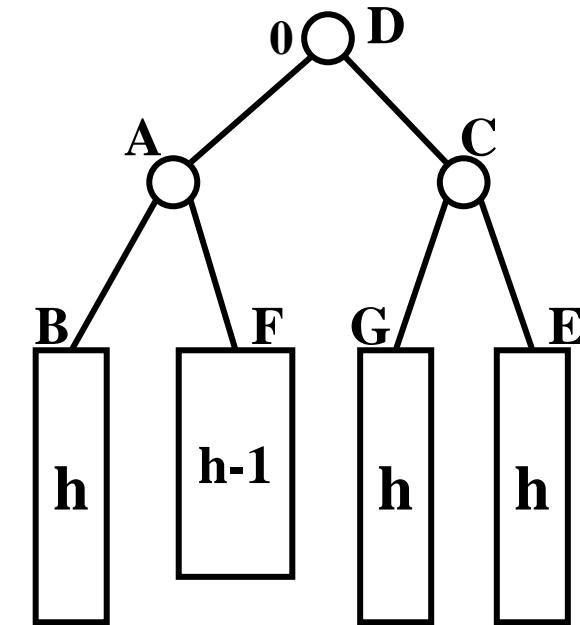
■ RL型：新结点Y被插入到A的右子树的左子树上(顺, 逆)



(a) G子树插入结点
高度变为h



(b) 绕D, C顺时
针转之后



(c) 绕D, A逆时
针转之后





5.6 AVL树(Cont.)

→ AVL树的插入操作与建立

- 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

→ 小结：插入→判断平衡→判断失衡类型→旋转调整





5.6 AVL树(Cont.)

→ AVL树的删除操作

- **删除与插入操作是对称的 (镜像, 互逆的) :**
 - **删除右子树结点导致失衡时, 相当于在左子树插入导致失衡, 即LL或LR;**
 - **删除左子树结点导致失衡时, 相当于在右子树插入导致失衡, 即RR或RL;**
- **删除操作可能需要多次平衡化处理**
 - **因为平衡化不会增加子树的高度, 但可能会减少子树的高度。**
 - **在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;**
 - **而在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。**
 - **因此, 删除操作可能需要多次平衡化处理。**

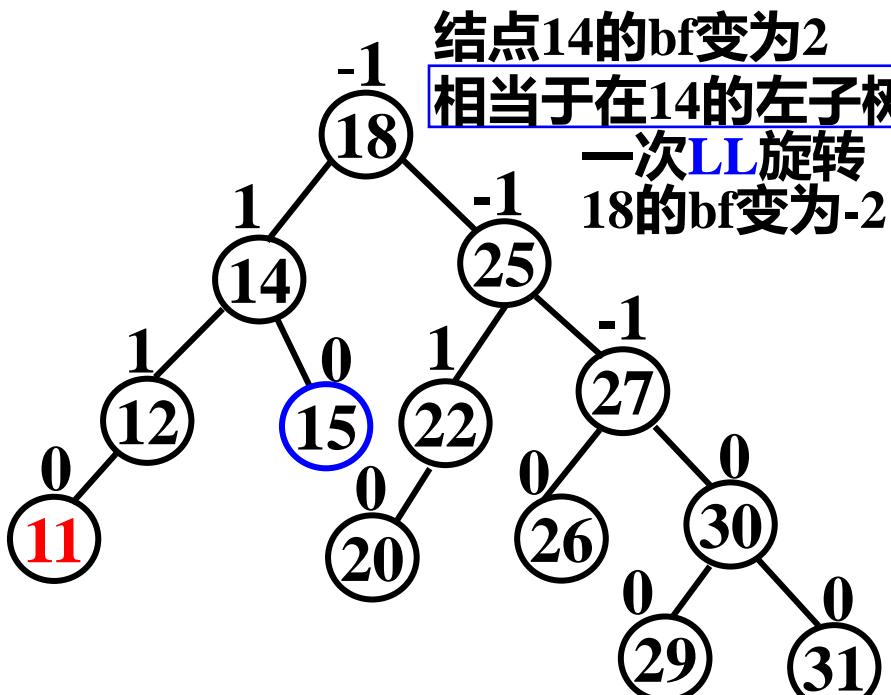




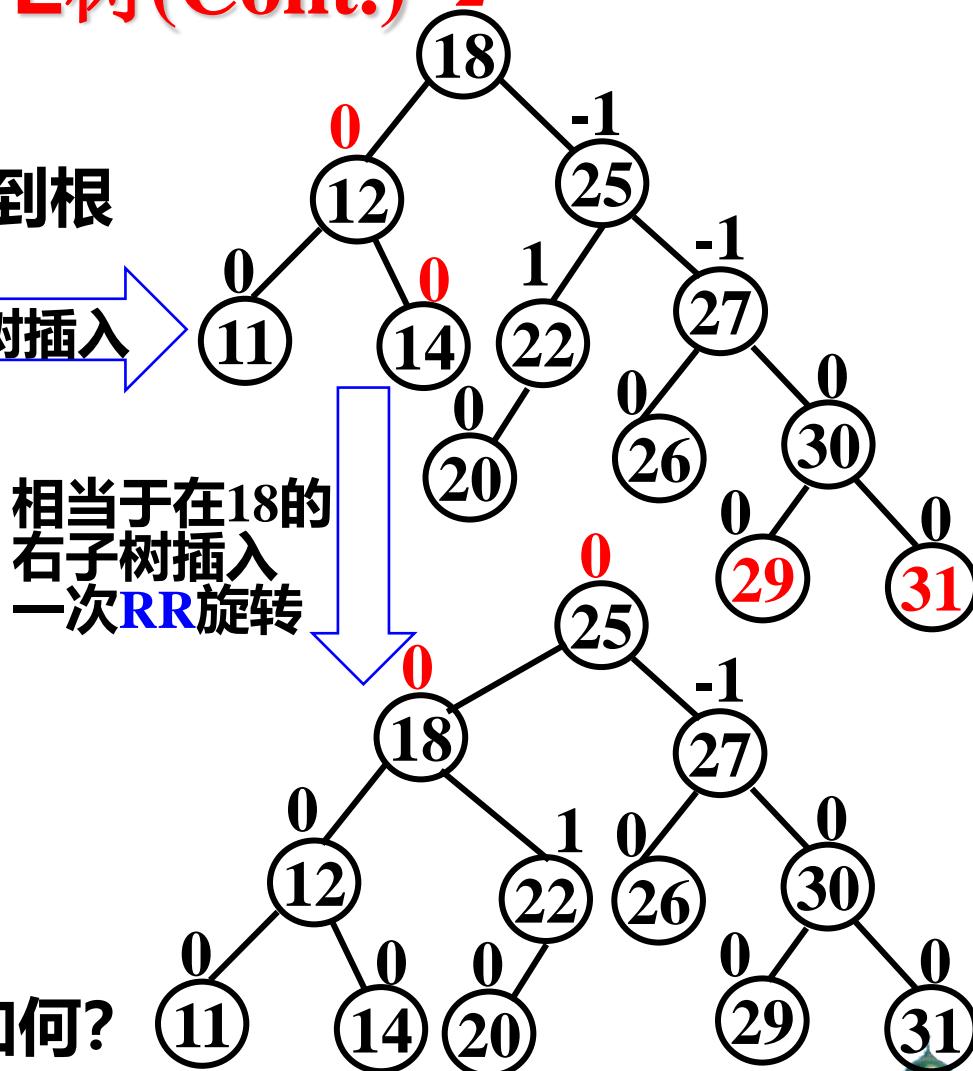
5.6 AVL树(Cont.) -2

AVL树的删除操作

- 示例：删除15，2次旋转到根



结点14的bf变为2
相当于在14的左子树插入
一次LL旋转
18的bf变为-2



- 若12只有右儿子13，会如何？

● LR旋转





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 $N_0=0$, $N_1=1$, $N_2=2$, $N_3=4$
- 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 $\log_{\varphi}(n+1) - 2$
- 因此，对于包含 n 个结点的AVL树，其最坏情况下的查找、插入和删除操作时间复杂度均为 $O(\log n)$





5.7 B-树和B⁺树

- 在AVL树在结点高度上采用**相对平衡**的策略，使其平均性能接近于BST的最好情况的性能。
- 当符号表的大小超过内存容量时，由于必须从**磁盘**等辅助存储设备上去读取这些查找树结构中的结点，**每次只能根据需要读取一个结点**，因此，AVL树性能就不是很高（每个结点只有一个关键字，**宽度**[结点包含的记录数量]**太小**）。
- 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（**增加宽度**），能否获得很好的查找性能呢？
- 基于这样的想法，人们设计了许多在**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⁺树(Cont.)

→ m -路(叉)查找树:

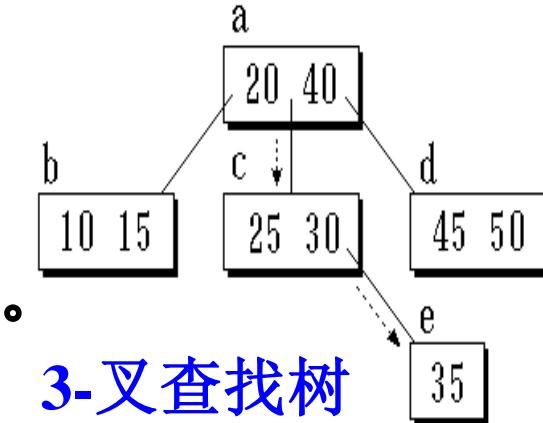
一棵 m -路查找树或者是一棵空树, 或者是满足如下性质的树:

- 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, $0 \leq i \leq n$ 。



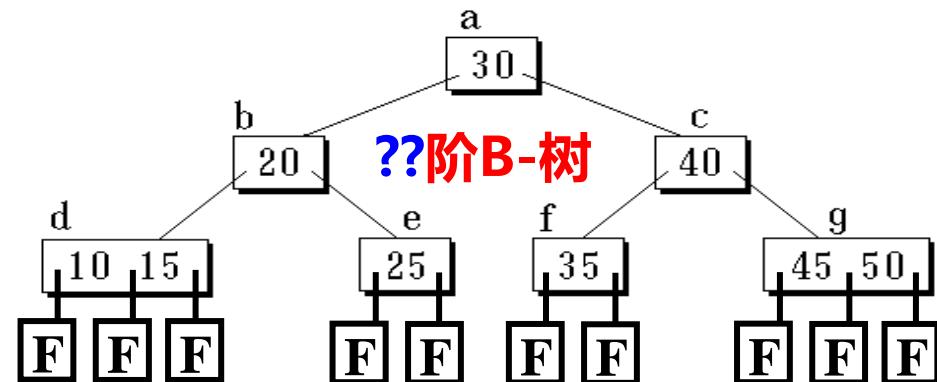
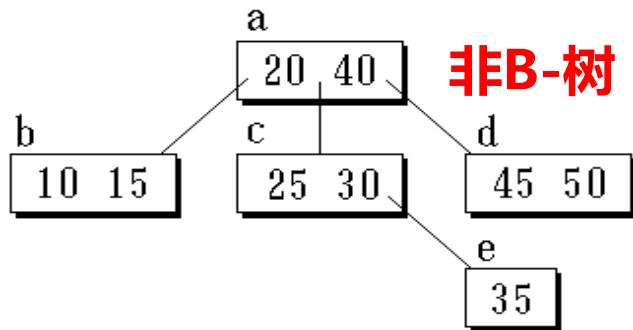
3-叉查找树



5.7 B-树和B⁺树(Cont.)

→ B-树：一棵 m 阶B-树是一棵 m -路查找树，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； $\lceil m/2 \rceil \sim m$
- 所有的终端结点和失败结点都位于同一层。



→ 高为 h 的 B-树最多的结点数和关键字数？

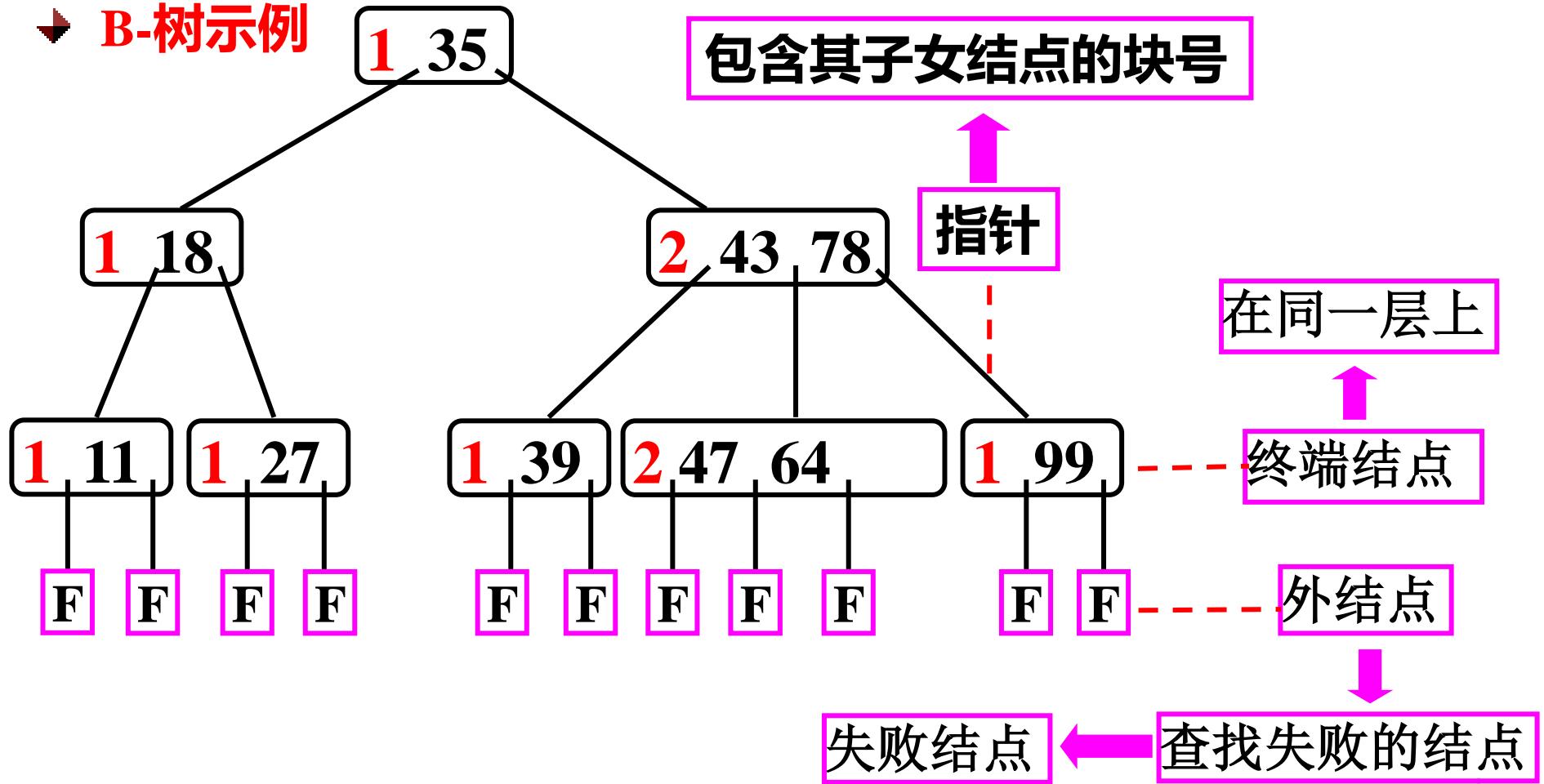
$$\sum_{i=1}^h m^{i-1} = (m^h - 1)/(m - 1)$$





5.7 B-树和B⁺树(Cont.)

→ B-树示例





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? (两种解法)

■ 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 $2 \cdot m/2$ 个结点
- 4层 至少 $2 \cdot m/2^2$ 个结点
- 如此类推,
- h 层 至少有 $2 \cdot m/2^{h-2}$ 个结点。
- 上述所有结点都不是失败结点。

高为 h 的 m 阶B-树最少有 N 个关键字

层号	结点数 \geq	关键字数 \geq
1	1	1
2	2	$2(m/2 - 1)$
3	$2 \cdot m/2$	$2 \cdot m/2 \cdot (m/2 - 1)$
4	$2 \cdot m/2^2$	$2 \cdot m/2^2 \cdot (m/2 - 1)$
...
h	$2 \cdot m/2^{h-2}$	$2 \cdot m/2^{h-2} \cdot (m/2 - 1)$
$h+1$	$2 \cdot m/2^{h-1}$	列和: $2 \cdot m/2^{h-1}$





5.7 B-树和B⁺树(Cont.)

■ 设 m 若树中关键字有 N 个，则失败结点数为 $N + 1$ ，即

$N + 1 = \text{失败结点数}$

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

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

■ 反之，如果在一棵 m 阶B-树中有 N 个关键字，则

$h \leq 1 + \log_{\lceil m/2 \rceil}((N + 1) / 2)$ (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

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

■ 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：
 - 在结点中查找 x 所用时间
 - 从磁盘中读出/写入结点所用时间





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（可采用折半查找）交替进行的过程。
- 因此，B-树的查找时间与
 - B-树的阶数 m （横向查找）
 - 和
 - B-树的高度 h （纵向查找）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 $[m/2 - 1, m-1]$ 之间（**根结点**： $[1, m-1]$ ）。
- **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没有超出上界 $m-1$ ，**直接插入，写盘**；否则($=m$)，结点需要“**分裂**”。
- 实现结点“**分裂**”的原则：设插入前**终端**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

- 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 : $(m/2 - 1, A_0, K_1, A_1, \dots, K_{\lceil m/2 \rceil - 1}, A_{\lceil m/2 \rceil - 1})$
 - 结点 q : $(m - m/2, A_{\lceil m/2 \rceil}, K_{m/2 + 1}, A_{m/2 + 1}, \dots, K_m, A_m)$
 - 位于中间的关键字 $K_{m/2}$ 与指向新结点 q 的指针形成一个二元组 $(K_{m/2}, q)$, 插入到这两个结点的双亲结点中去, 没有增加树的高度。
 - 在插入该二元组之前, 先将结点 p 和 q 写到磁盘上。
 - 若双亲结点也“满”了呢?
 - 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, 树增高1

分裂与提示



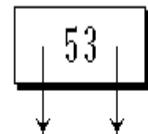


5.7 B-树和B⁺树(Cont.)

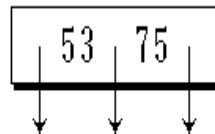
→ B-树的插入操作与建立

■ 例：从空树开始逐个加入关键字建立3阶B-树

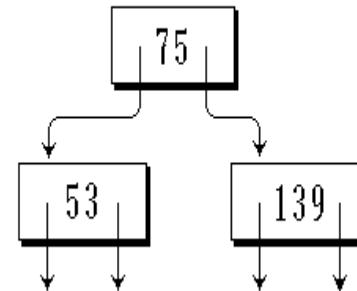
$n=1$ 加入 53



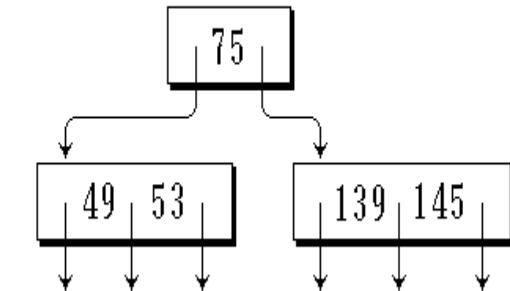
$n=2$ 加入 75



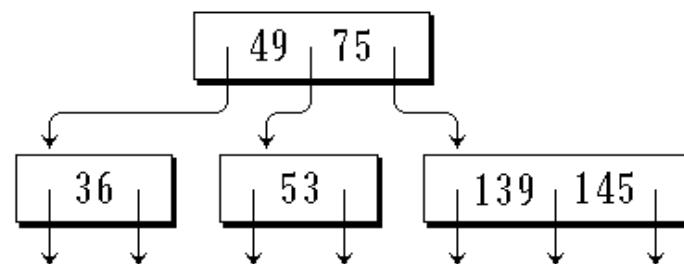
$n=3$ 加入 139



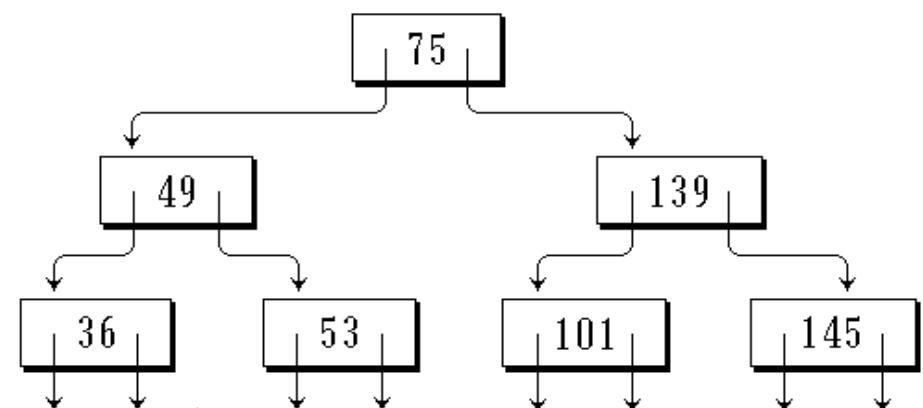
$n=5$ 加入 49, 145



$n=6$ 加入 36



$n=7$ 加入 101



为什么根结点可以只有2个子树（而不必至少半满 [$m/2$]) ?

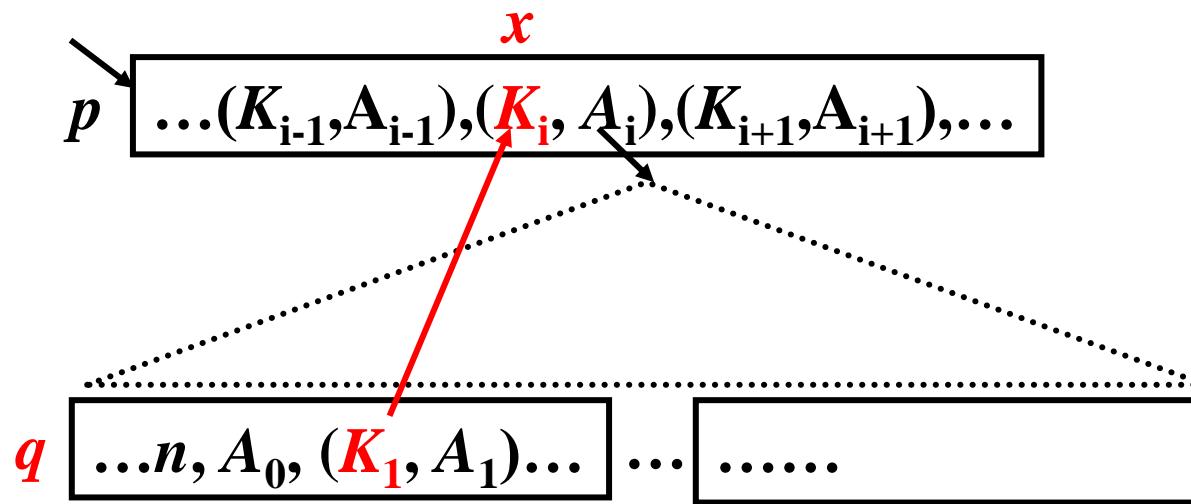




5.7 B-树和B⁺树(Cont.)

→ B-树的删除操作：转换成在终端结点上的删除

- 首先，找到被删除关键字 x 所在结点 p 。若该结点不是终端结点，且 $x = K_i$ ($1 \leq i \leq n$)，则以该结点 p 中 A_i 所指示子树中的最小关键字（如在结点 q ）或者 A_{i-1} 所指示子树中的最大关键字来代替被删关键字 K_i （类似 BST 或 AVL）；
- 然后在 q 所在的终端结点中删除 x 。把删除操作转换为终端结点上的删除操作。





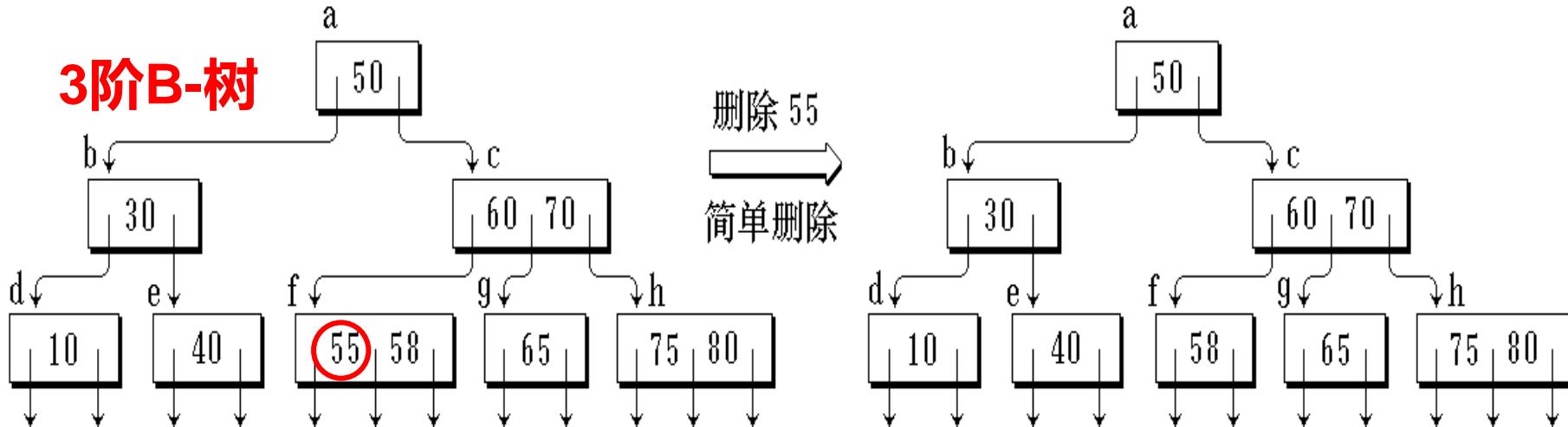
5.7 B-树和B⁺树(Cont.)

→ B-树的删除操作：在终端结点上的删除分 4 种情况：

- (1) 被删关键字所在**终端结点** p 同时又是**根结点**，若删除后该结点中至少有一个关键字，则直接删去该关键字并将**修改后的结点写回磁盘**。否则，删除以后，B-树为空。

对于以下**3**种情况， p 都不是**根结点**：

- (2) 被删关键字所在**终端结点** p 不是**根结点**，且删除前该结点中关键字个数 $n \geq m/2$ ，则直接删去该关键字并将**修改后的结点写回磁盘**，删除结束。





5.7 B-树和B⁺树(Cont.)

→ **B-树的删除操作：**在终端结点上的删除分 4 种情况：

■ (3) 被删关键字 x 所在的终端结点 p **删除前关键字个数 $n = m/2 - 1$** , 若此时其**右兄弟** (或左兄弟) q 的关键字个数 $n \geq m/2$, 则可按以下步骤**调整结点 p 、右兄弟** (或左兄弟) q 和**其双亲结点 r** , 以达到新的平衡(**保持结构性质不变**)。

- 将双亲结点 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)关键字上移到双亲结点 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。

兄弟够借，向兄弟借；调整

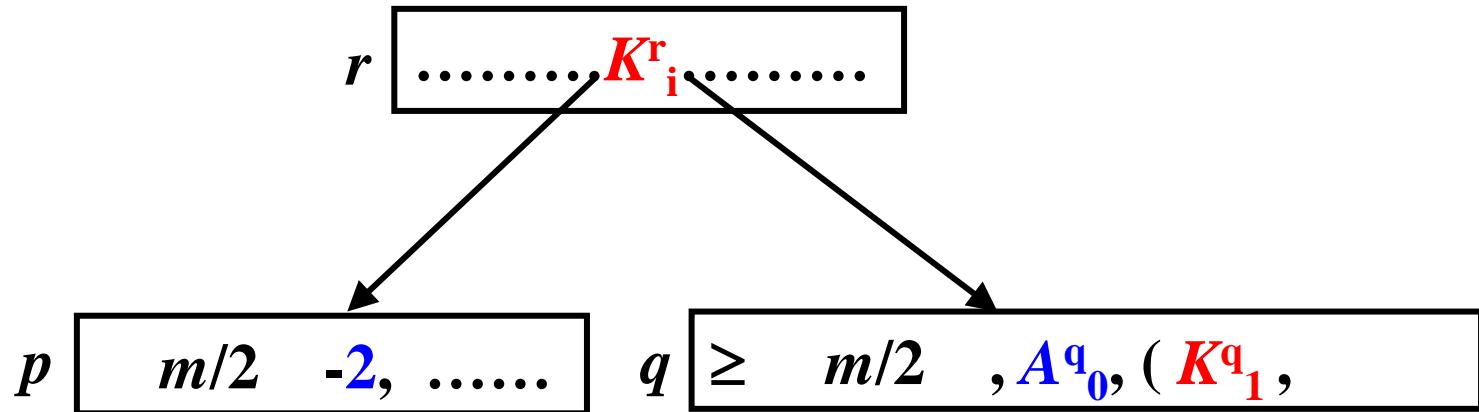




5.7 B-树和B⁺树(Cont.)

→ B-树Kri的删除操作：在终端结点上的删除分4种情况：

兄弟够借，向兄弟借；调整



$p++$: K^r_i 下移至 p 成为其最右关键字

$q--$: K^q_1 上移至 r 代替 K^r_i



q 的最左指针成为 p 的最右指针





5.7 B-树和B⁺树(Cont.)

→ **B-树的删除操作：在终端结点上的删除分 4 种情况：**

- (4) 被删关键字 x 所在的终端结点 p 删除前关键字个数 $n = m/2 - 1$ ，若此时其右兄弟（和左兄弟）结点 q 的关键字个数 $n = m/2 - 1$ ，则按以下步骤合并这两个结点。
 - 将双亲结点 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-1} 与 A_i 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。

兄弟不够借，合并





5.7 B-树和B⁺树(Cont.)

→ B-树的删除操作：在终端结点上的删除分 4 种情况：

$r \dots (K_{i-1}^r, A_{i-1}^r), (K_i^r, A_i^r), (K_{i+1}^r, A_{i+1}^r) \dots$

与插入时的分裂对称

$p \quad m/2 -1, \dots$

$q \quad m/2 -1, A_0^q, (K_1^q,$

$A_1^q), \dots$

- $(m/2 -2) + (m/2 -1) + 1 = 2(m/2 -2) \leq m - 1$

合并
↓

$r \dots (K_{i-1}^r, A_{i-1}^r), (K_{i+1}^r, A_{i+1}^r) \dots$

若 r 中的关键字数也少于 $[m/2]-1$ 或 1，怎么办？

$p \leq m - 1, \dots, (K_i^r, A_0^q), (K_1^q, A_1^q), \dots$

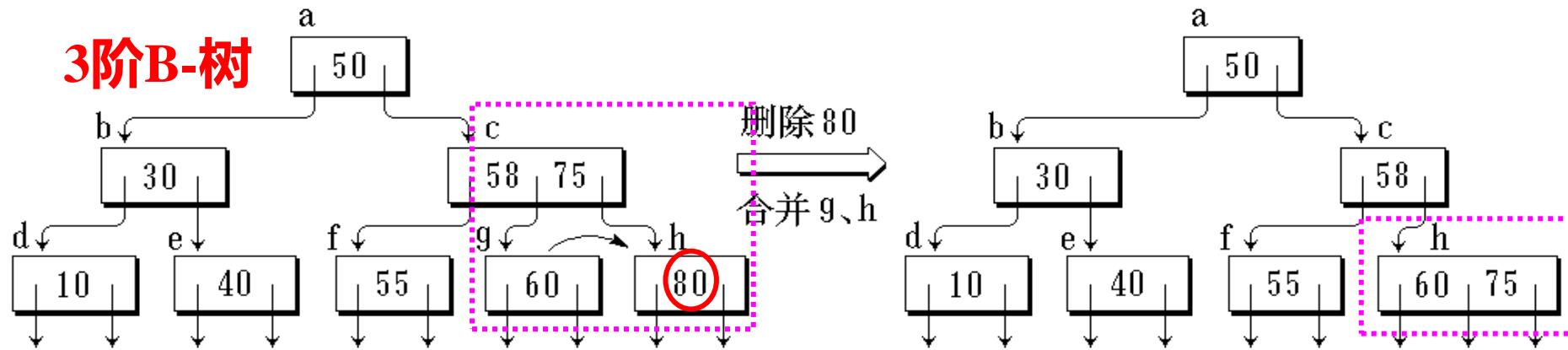




5.7 B-树和B⁺树(Cont.)

→ **B-树的删除操作：在终端结点上的删除分 4 种情况：**

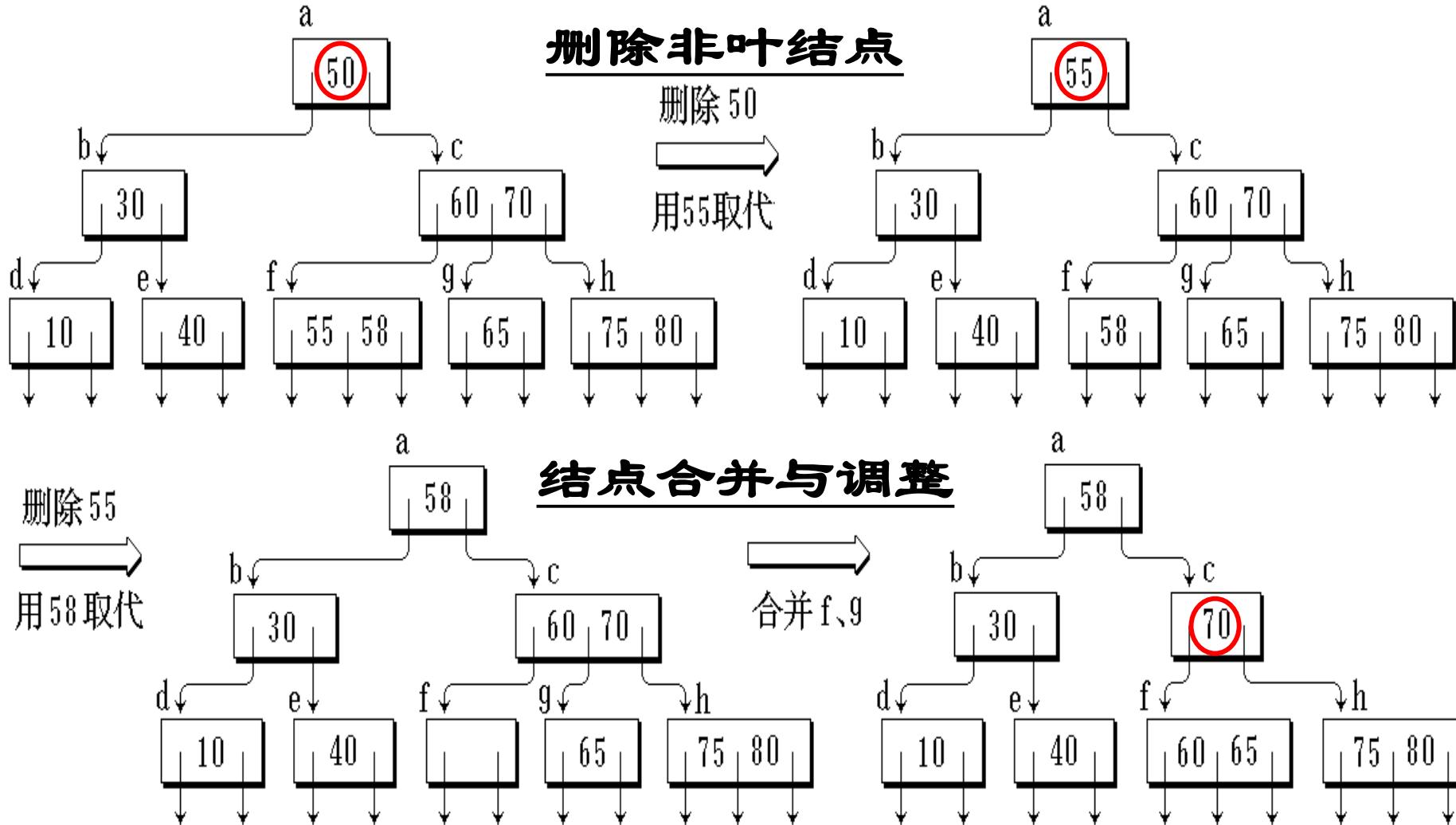
- 在**合并结点的过程中**，双亲结点中的关键字个数减少了。
- 若双亲结点是**根结点且结点关键字个数减到 0**，则该双亲结点应从树上删去，**合并后保留的结点成为新的根结点（树高减1）**；否则（**根结点关键字个数未减到 0**）将双亲结点与**合并后保留的结点都写回磁盘**，删除处理结束。
- 若双亲结点**不是根结点，且关键字个数减到 $m/2 - 2$** ，则又要**与它自己的兄弟结点合并**，重复上面的**合并步骤**。**最坏情况下这种结点合并处理要自下向上直到根结点。**





5.7 B-树和B⁺树(Cont.)

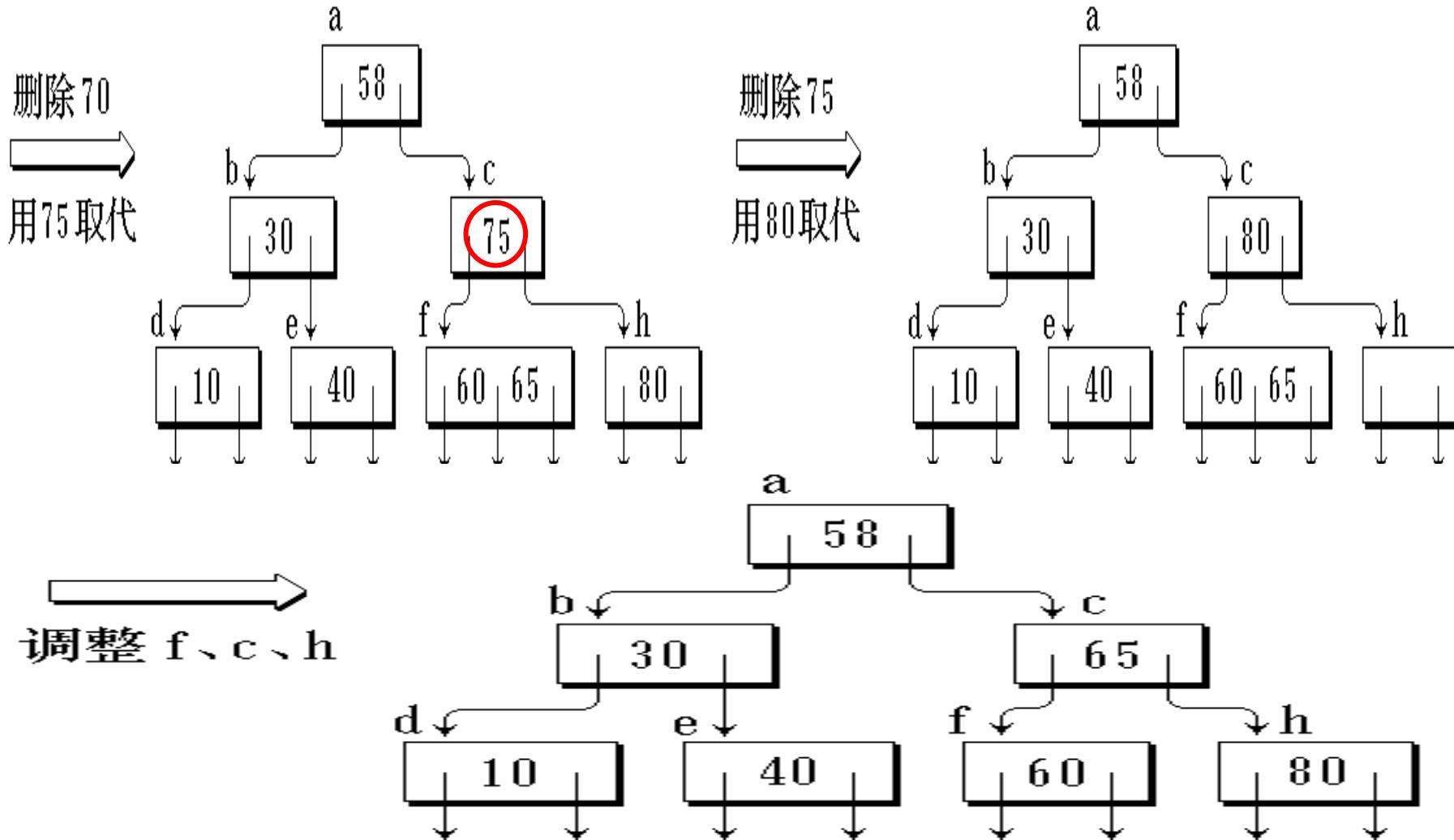
→ B-树的删除操作：在终端结点上的删除分 4 种情况：





5.7 B-树和B⁺树(Cont.)

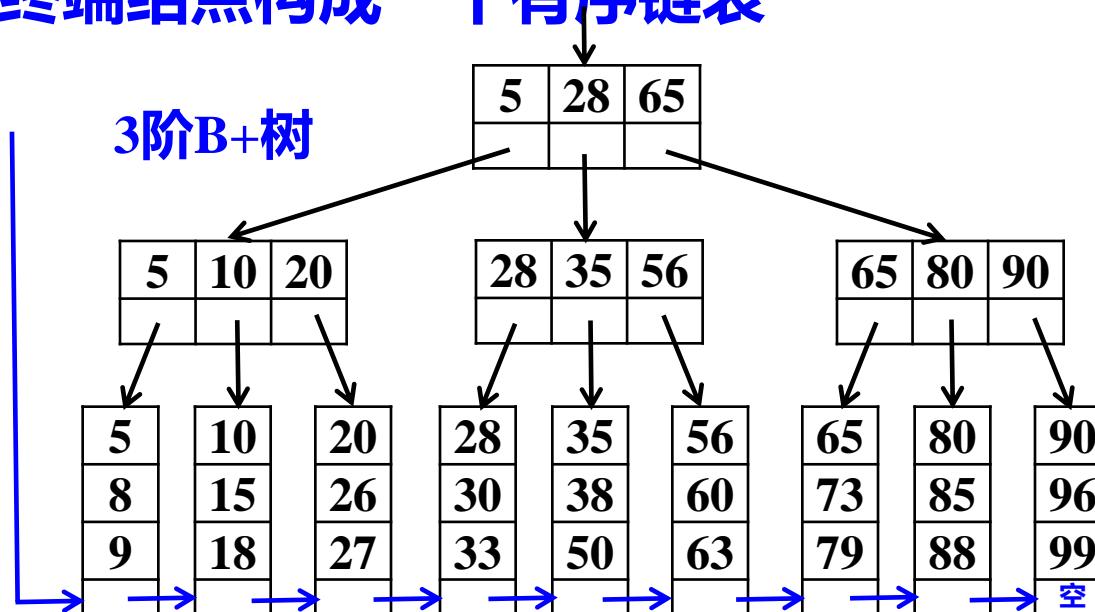
→ B-树的删除操作：在终端结点上的删除分 4 种情况：





5.7 B-树和B⁺树(Cont.)

- **B⁺树**: 是B-树的一种变体，作为**文件索引**比B-树更普遍
- **与B-树的差异在于：**
 - (1)非终端结点的**子树个数与关键字个数相同**
 - (2)**与记录有关的信息均存放在终端结点中，尽管非终端结点具有与之相同的关键字，非终端结点仅具有索引作用**
 - (3)**终端结点附加一个指向下一个终端结点的指针，把所有终端结点构成一个有序链表**





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。

- 终端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终端结点。可对B⁺树进行两种查找操作：一种是沿终端结点链顺序查找，另一种是从根结点开始，进行自顶向下，直至终端结点的随机查找。

→ 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终端结点中，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终端结点层的关键码。
- 由于B⁺树的所有终端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⁺树非常适合范围检索。





5.7 B-树和B⁺树(Cont.)

- B⁺树的插入与B-树的插入过程类似。
 - 不同的是B⁺树在终端结点上进行；
 - 如果终端结点中的关键码个数超过m，就必须分裂成关键码数目大致相同的两个结点，并保证上层结点中有这两个结点的最大关键码。
- B⁺树中的关键码在终端结点层删除后，其在上层的副本可以保留，作为一个“分解关键字”存在，如果因为删除而造成结点中关键码数小于 $m/2$ ，其处理过程与B-树的处理一样。
 -
- B*树：
 - 是B⁺树的变体，非根和非终端结点增加指向兄弟的指针
 - 非终端结点关键字个数至少为 $2m/3$ ，即块的最低利用率为 $2/3$ (B_树是 $1/2$)





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 $\Omega(\log n)$

- 要想**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？**

- 在**关键字值和存储位置之间建立一个确定的对应关系**

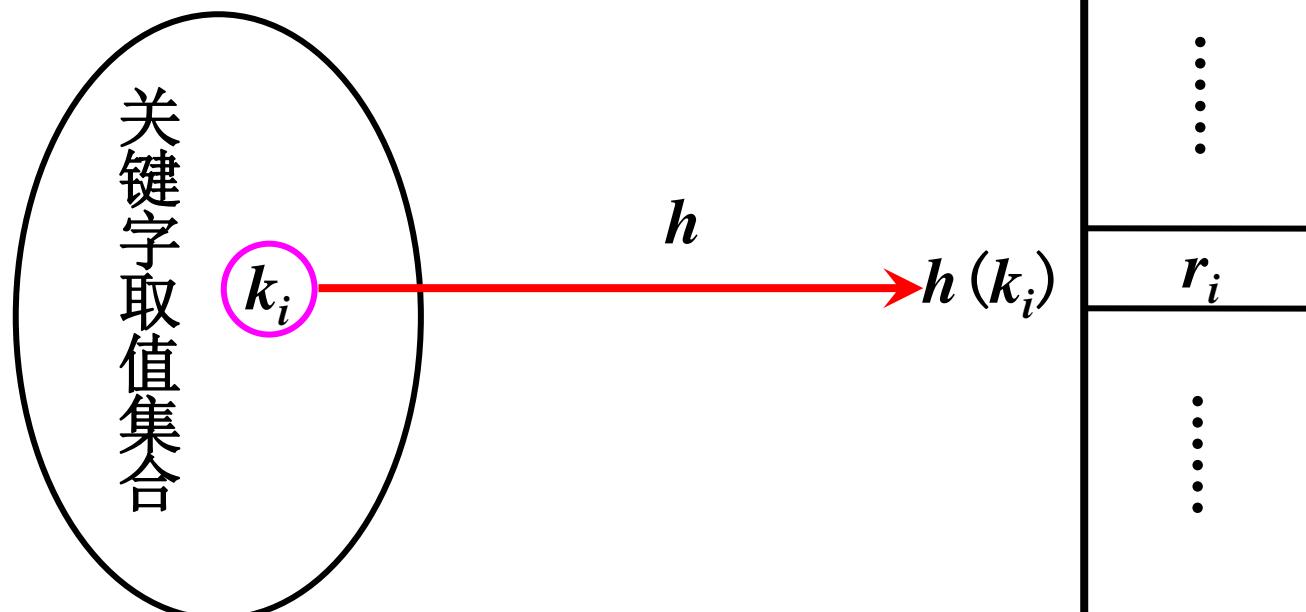




5.8 散列技术(Cont.)

◆ 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 $O(1)$ 。



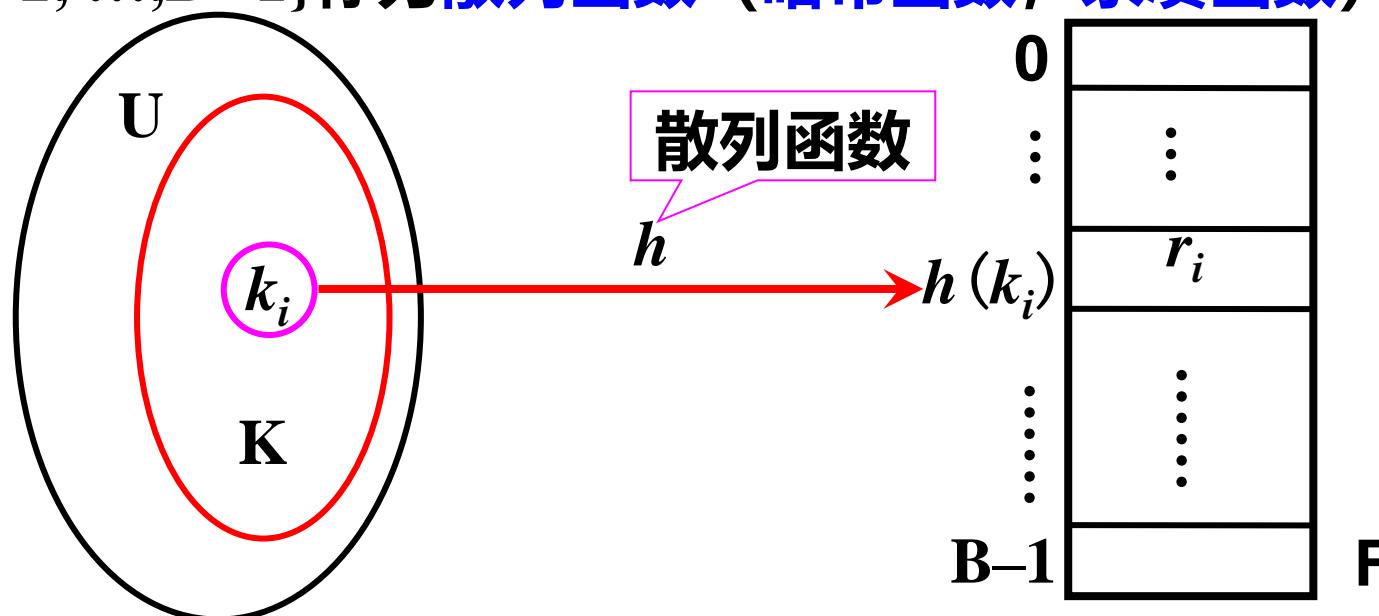


5.8 散列技术(Cont.)

◆ 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）

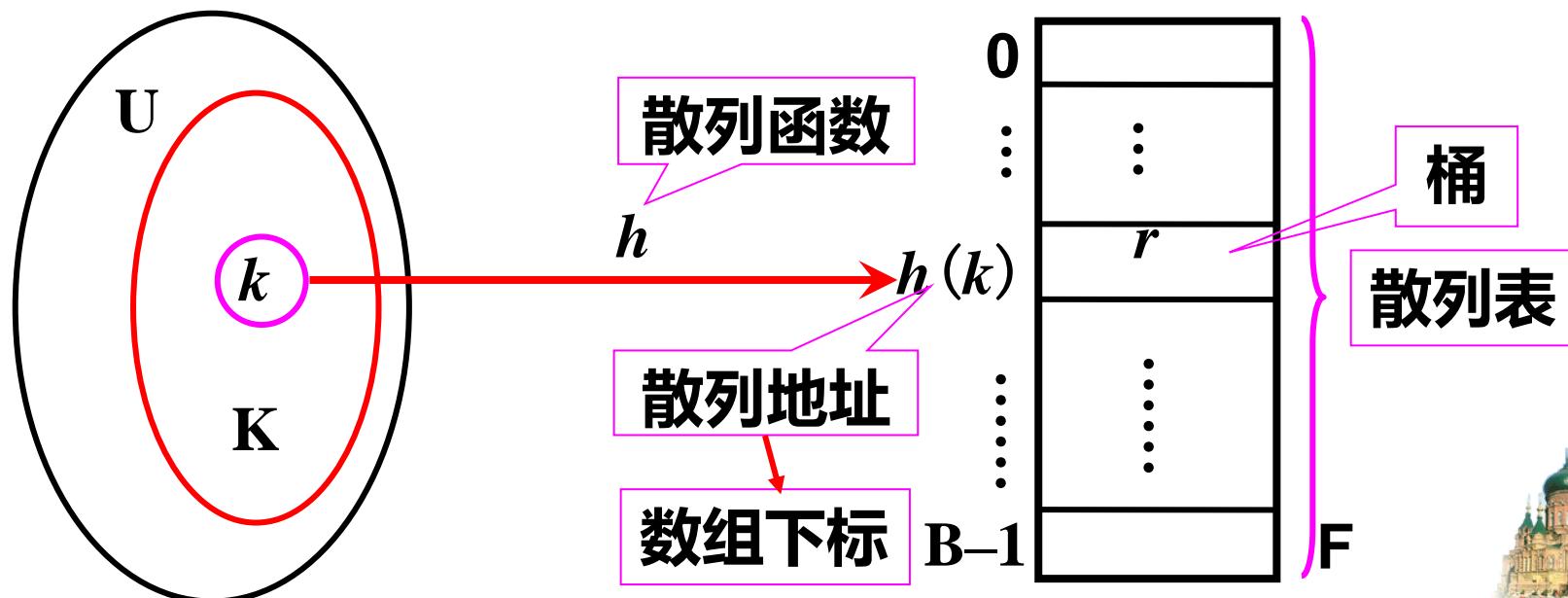




5.8 散列技术(Cont.)

◆ 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)

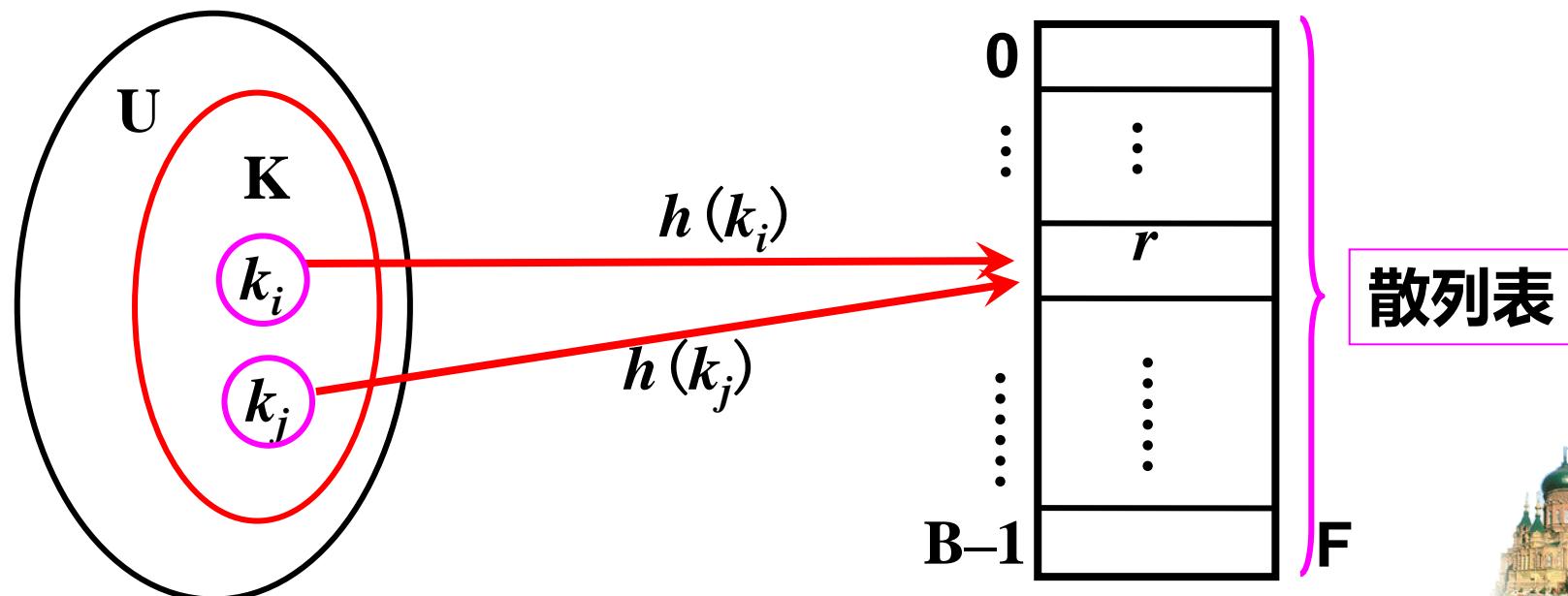




5.8 散列技术(Cont.)

◆ 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**。

→ **散列技术不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**，如在散列表中，找**最大或最小关键字**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。**





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

→ 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。





5.8 散列技术(Cont.)

散列函数的构造

→ 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

→ 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。





5.8 散列技术(Cont.)

散列函数的构造

→ 散列函数的构造方法----平方取中法

- 取 key^2 的中间的几位数作为散列地址
- 扩大相近数的差别，然后根据表长取中间几位作为散列值，使地址值与关键字的每一位都相关。
- 散列地址的位数要根据B来确定，有时要设一个比例因子，限制地址越界。
- 适用情况：事先不知道关键码的分布且关键码的位数不是很大

记录	key	key ²	Hash
A	0100	0 010 000	010
I	1100	1 210 000	210
J	1200	1 440 000	440
I0	1160	1 370 400	370
P1	2061	4 310 541	310
P2	2062	4 314 704	314
Q1	2161	4 734 741	734
Q2	2162	4 741 304	741
Q3	2163	4 745 651	745





5.8 散列技术(Cont.)

散列函数的构造

→ 散列函数的构造方法----折叠法

- 若关键位数较多，可根据B的位数将关键字分割成位数相同的若干段（最后一段位数可以不同），然后将各段叠加和(舍去进位)作为散列地址。
- 示例：图书编号: 0-442-20586-4

5864	5864	左: 移位叠加
4220	0224	$\text{Hash}(key) = 0088$
+ 04	+ 04	右: 间界叠加
<u>10088</u>	<u>6092</u>	$\text{Hash}(key) = 6092$

- 适用情况：关键码位数很多，事先不知道关键码的分布。





5.8 散列技术(Cont.)

散列函数的构造

→ 散列函数的构造方法----数字分析法

- 根据关键字值在各个位上的分布情况，选取分布比较均匀的若干位组成散列地址。

- 示例：

假设散列表长为 100_{10} ，可取中间**四**位中的**两位**为散列地址。

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

- **适用情况：**若事先知道关键字集合，且关键字的位数比散列表的地址位数多





5.8 散列技术(Cont.)

散列函数的构造

→ 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
- 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	

22

3

3

3

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。





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

■ Insert算法的实现

■ Delete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.)

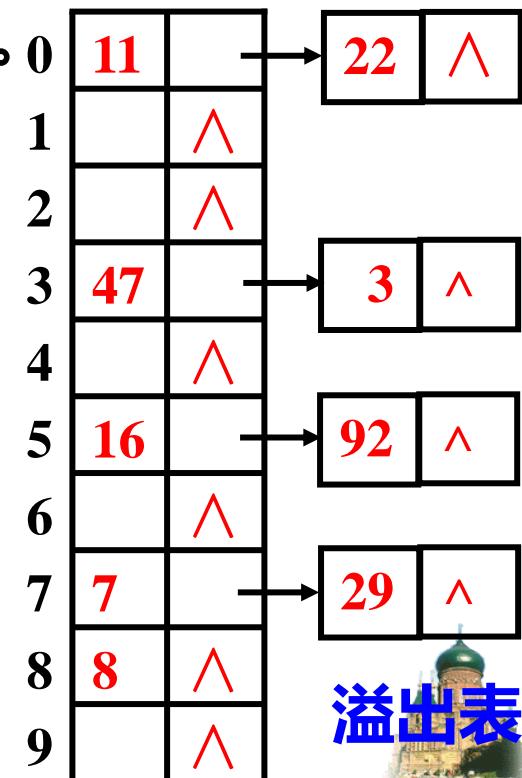
冲突处理的方法----带溢出表的内散列法

- **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

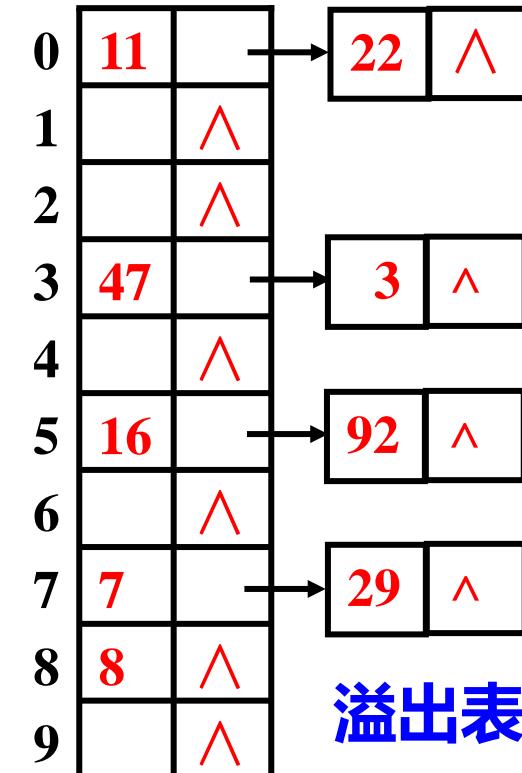
→ 查找算法的实现：

■ 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，删除首结点。



溢出表

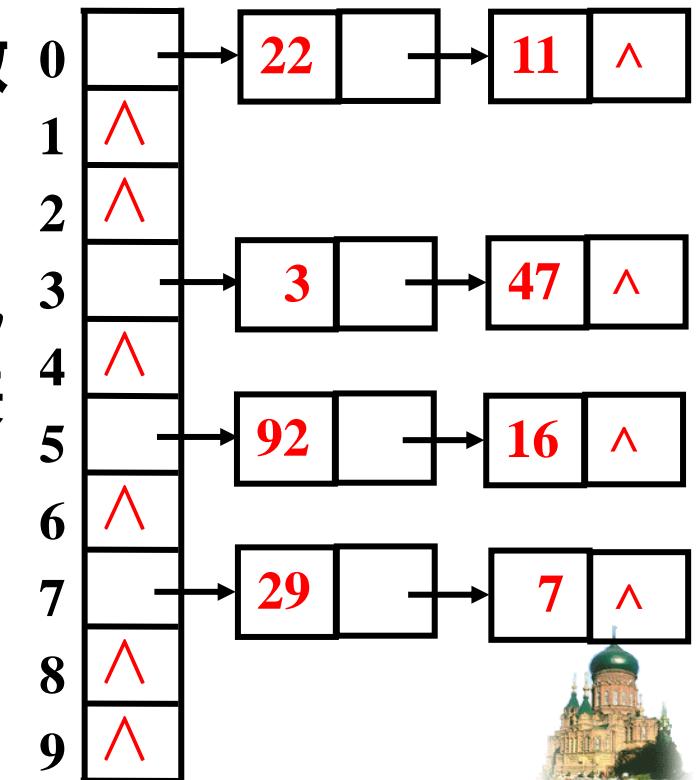




5.8 散列技术(Cont.)

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（桶）。
- **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 。





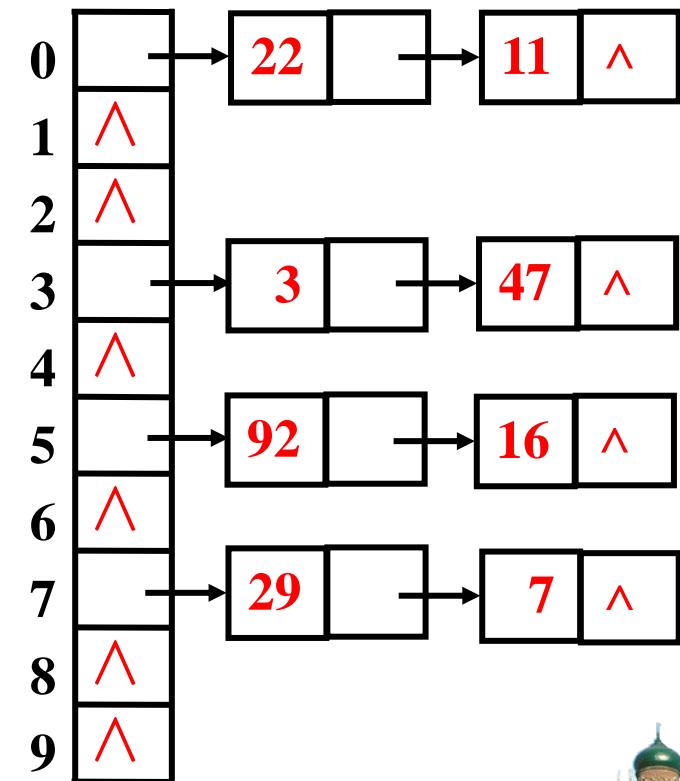
5.8 散列技术(Cont.)

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];
```





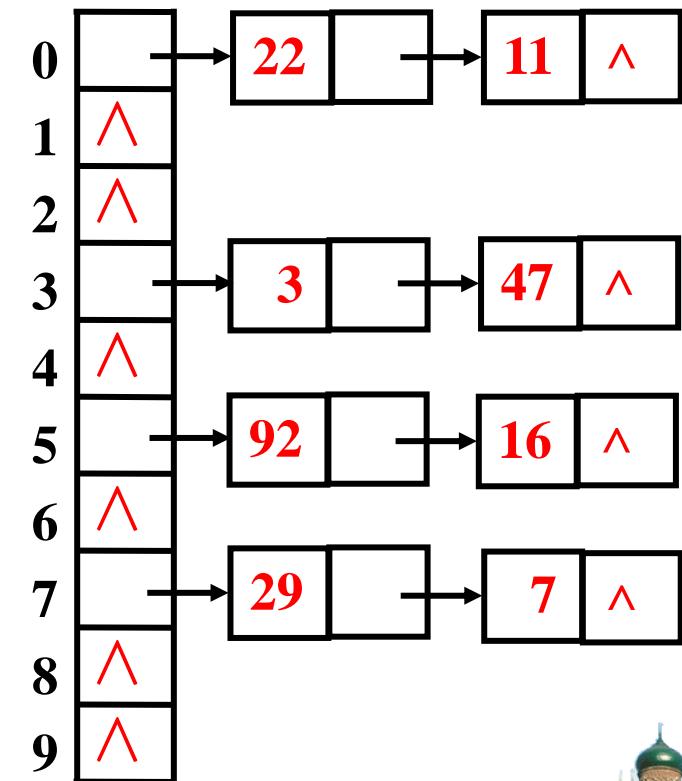
5.8 散列技术(Cont.)

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*/
```





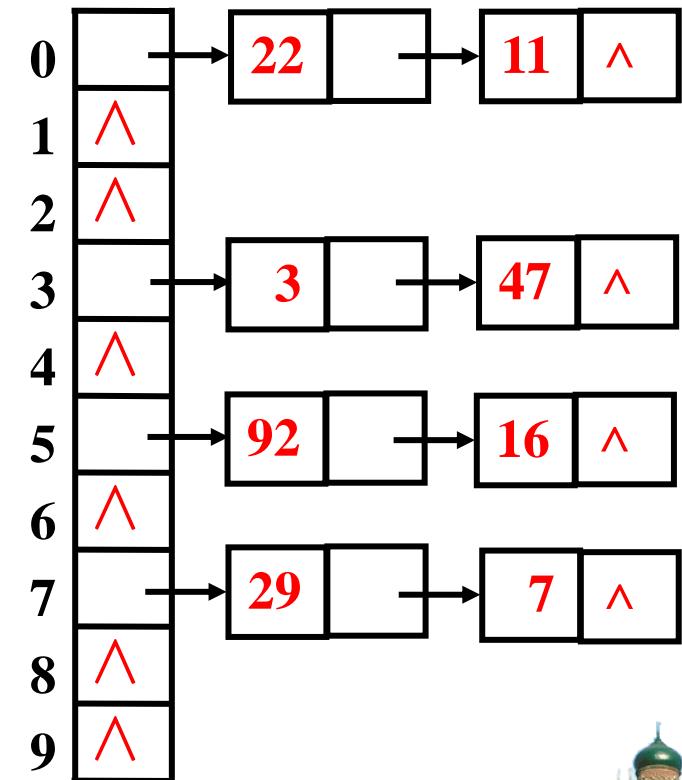
5.8 散列技术(Cont.)

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
```



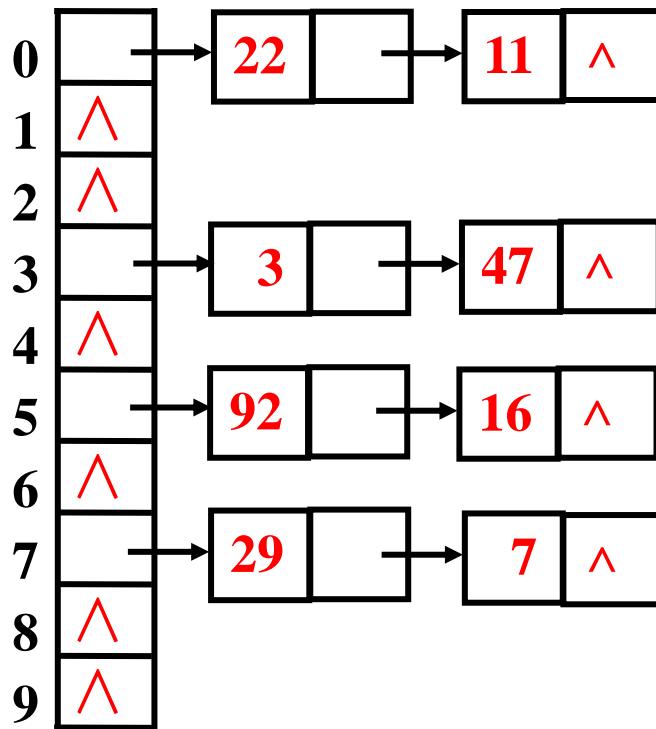


5.8 散列技术(Cont.)

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	$\frac{1}{2}(1 + \frac{1}{(1 - \alpha)^2})$	$\frac{1}{2}(1 + \frac{1}{1 - \alpha^2})$
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





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成功的平均长度





例题(Cont.)

- $(7, 8, 30, 11, 18, 9, 14)$ $H(key) = (\text{key} * 3) \% 7$
- 解：(1) 由于序列长度为7，装填因子为0.7，所以，表长为10，下标为0至9。所构造散列表如下：

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	
成功的比较次数	1	2		1		1	1	3	3	
不成功比较次数	3	2	1	2	1	5	4	--	--	--

- (2) $ASL_{\text{成功}} = (1+2+1+1+1+3+3) / 7 = 12/7$
- $ASL_{\text{不成功}} = (3+2+1+2+1+5+4) / 7 = 18/7$





本章小结

→ 查找结构

■ 线性结构

- 线性无序表
- 线性有序表
- 有序表+无序表

■ 树型结构

- BST 和 AVL
- B-树和B+树

■ 散列结构

- 内散列结构
- 外散列结构
- 带溢出表的散列结构

→ 查找方法

- 基于关键字比较的查找
- 基于关键字存储位置的查找

→ 查找操作

- Search
- Insert(Create)
- Delete

→ 判定树

→ 查找的性能指标

- 查找成功的ASL
- 查找不成功的ASL

