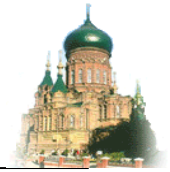
第8章 查找

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。

重点：顺序查找、二分查找、二叉排序树查找以及散列表构造及散列方法实现。

难点：二叉排序树的删除算法和平衡二叉树的构造算法。

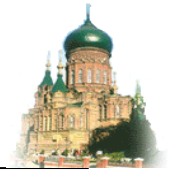


本章主要内容

- 8.1 基本概念和术语
- 8.2 线性查找
- 8.3 折半（二分）查找
- 8.4 分块查找
- 8.5 BST——二叉查找树
- 8.6 AVL树
- 8.7 B-树与B+树
- 8.8 散列技术
- 本章小结

静态查找表

动态查找表

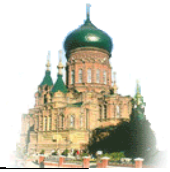


查找表

是由同一类型的数据元素(或记录)构成的集合。

对查找表经常进行的操作：

- 查询某个“特定的”数据元素是否在查找表中；
- 检索（知道存在）某个“特定的”数据元素的各种属性；
- 在查找表中插入一个数据元素；
- 从查找表中删去某个数据元素。



8.1 基本概念和术语

- 关键字：可以标识一个记录的某个数据项的值。
  - 主关键字：可以唯一地标识一个记录的关键字。
  - 次关键字：可以识别若干记录的关键字。
- 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。

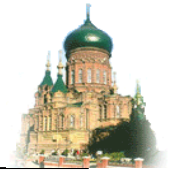
学号	姓名	性别	年龄	入学成绩
0001	张亮	男	19	625
0002	张亮	女	28	617
0003	刘楠	女	19	623
…	…	…	…	…

8.1 基本概念和术语

- 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

8.1 基本概念和术语

- 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。



8.1 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)。

ADT SearchTable {

数据对象D： D是具有相同特性的数据元素的集合，

数据关系R： 数据元素同属一个集合。

基本操作P：

InitST（&F）

Destrory（&F）

Search（k，F）

Insert（R，F ）

Delete(k，F)

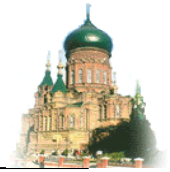
}ADT SearchTable

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。

8.1 基本概念和术语

- 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：为了提高查找的效率，需要在查找表中的元素之间人为地附加某种确定的关系，换句话说，用另外一种结构来表示查找表。
  - 主要的查找结构：
  - 集合 → 线性表、树表、散列表
    - 线性表：主要适用于静态查找，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。



8.1 基本概念和术语

- 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$T(n，k)$。

8.1 基本概念和术语

- 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，

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

8.2 线性查找

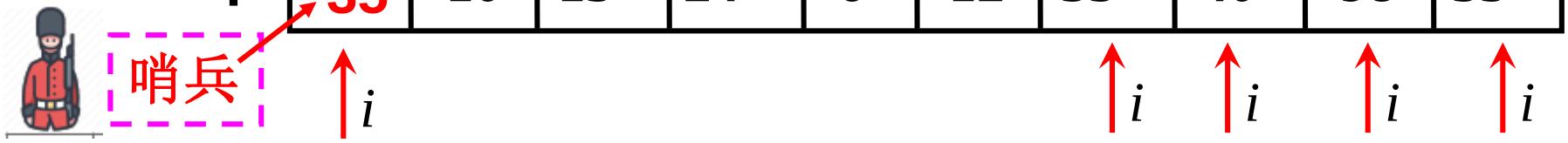
- 线性（顺序）查找基本思想：
  - 从线性表的一端开始，顺序扫描线性表，依次将扫描到的结点关键字与给定值K相比较。
  - 若当前扫描到的结点关键字与k相等，则查找成功；
  - 若扫描结束后，仍未找到关键字等于k的结点，则查找失败。
- 线性（顺序）查找对存储结构要求
  - 既适用于线性表的顺序存储结构----适用于静态查找
  - 也适用于线性表的链式存储结构----也适用于动态查找

8.2 线性查找

- 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

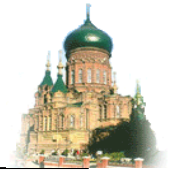


  - Insert操作的实现
  - Delete操作的实现

} 不适合顺序表

8.2 线性查找

```
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 */
```



8.2 线性查找

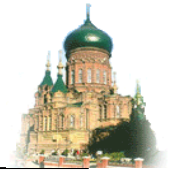
- 单链表上的查找——也适合于动态查找
  - 单链表的类型定义

```
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) ； $ASL_{成功}=(n+1)/2$， $ASL_{失败}=n+1$



8.2 线性查找

线性查找的优化

在不等概率查找的情况下，*ASL*取最小值，当

$$P_n \geqslant P_{n-1} \geqslant \cdots \geqslant P_2 \geqslant P_1$$

根据此性质构造表：
对于出现概率高的排在最后，便于先查找。例如最常用的书放在上面

若查找概率无法事先测定，则查找过程采取的改进办法是，在每次查找之后，将刚刚查找到的记录直接移至表尾的位置上。

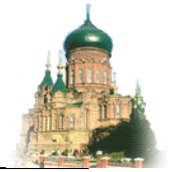
8.3 折半查找

- 折半查找（也称二分查找）：
  - 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 ≤ F[1].key ≤ F[2].key ≤ … ≤ F[last].key
  - 或 F[0].key ≥ F[1].key ≥ F[2].key ≥ …≥ F[last].key

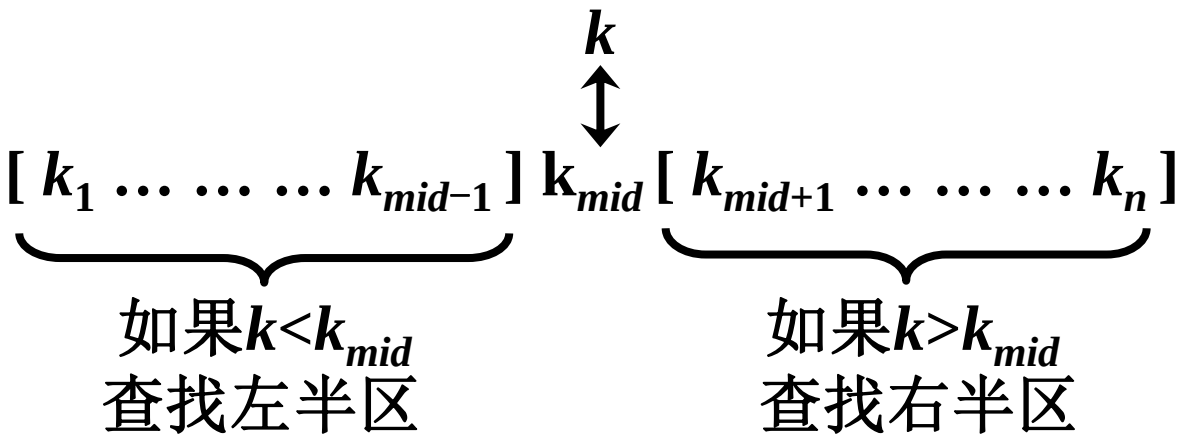
  - *注意：折半查找只适合于静态查找！*



8.3 折半查找

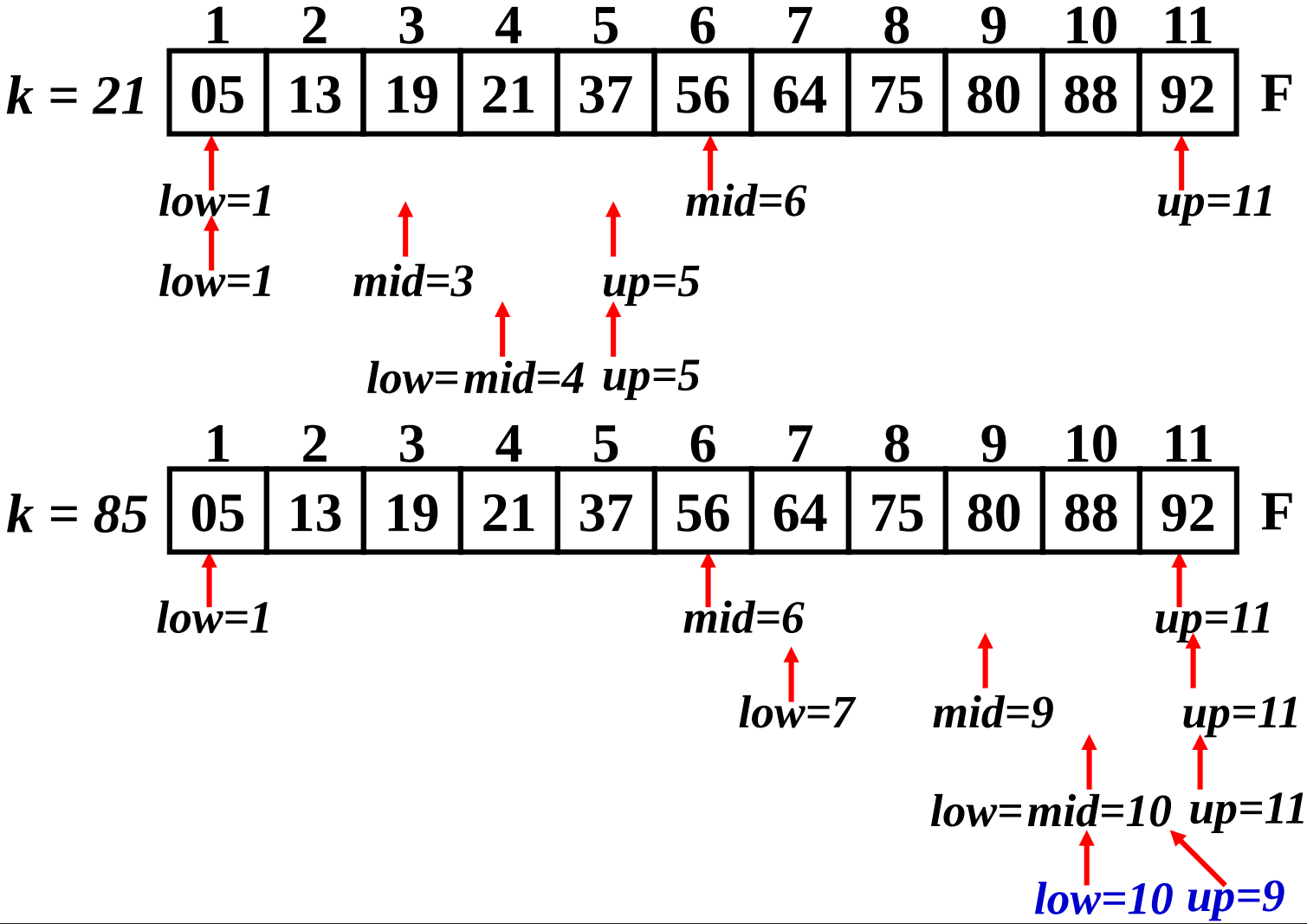
- 折半查找的基本思想：
  - 在有序表中，取中间记录作为比较对象，若给定值与中间记录的关键码相等，则查找成功；若给定值小于中间记录的关键码，则在中间记录的左半区继续查找；若给定值大于中间记录的关键码，则在中间记录的右半区继续查找。不断重复上述过程，直到查找成功，或所查找的区域无记录，查找失败。

（$mid=(1+n)/2$）
向下取整



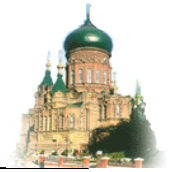
8.3 折半查找

- 折半查找的示例：



8.3 折半查找

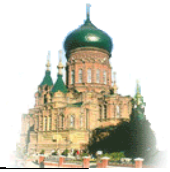
- 折半查找的算法实现步骤
  - 1. 初态化：令low ,up分别表示查找范围的上、下界，初始时low = 1, up = last;
  - 2. 折半：令mid = (low+up)/2，取查找范围中间位置元素下标;
  - 3.比较：k与F[mid].key
    - 3.1 若F[mid].key == k，查找成功，返回 mid;
    - 3.2 若F[mid].key > k，low不变，调整up = mid - 1，查找范围缩小一半;
    - 3.3 若F[mid].key < k，调整low = mid + 1，up不变，查找范围缩小一半;
  - 4. 重复2 ~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



8.3 折半查找

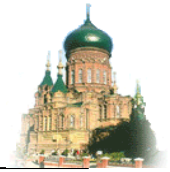
- 折半查找的算法实现

```
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)  */
```



8.3 折半查找

- 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。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]～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]～F[n]相对应的折半查找判定树。

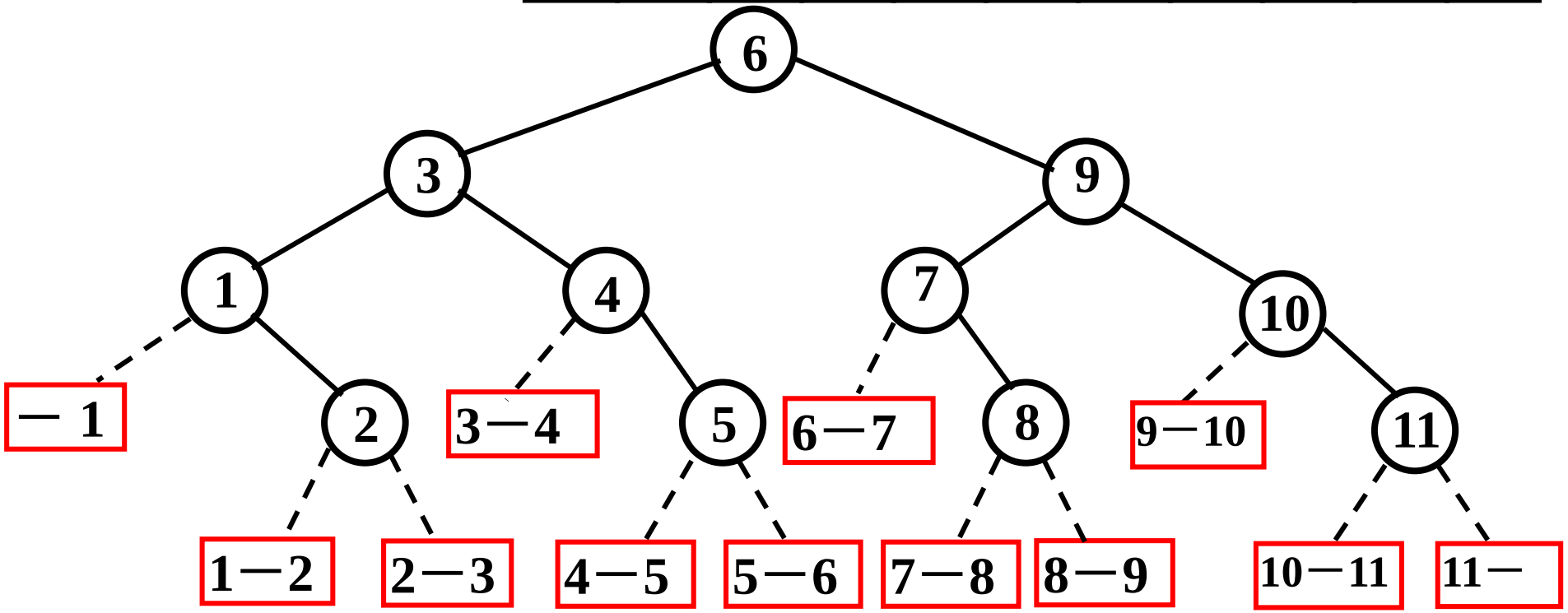


8.3 折半查找

- 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

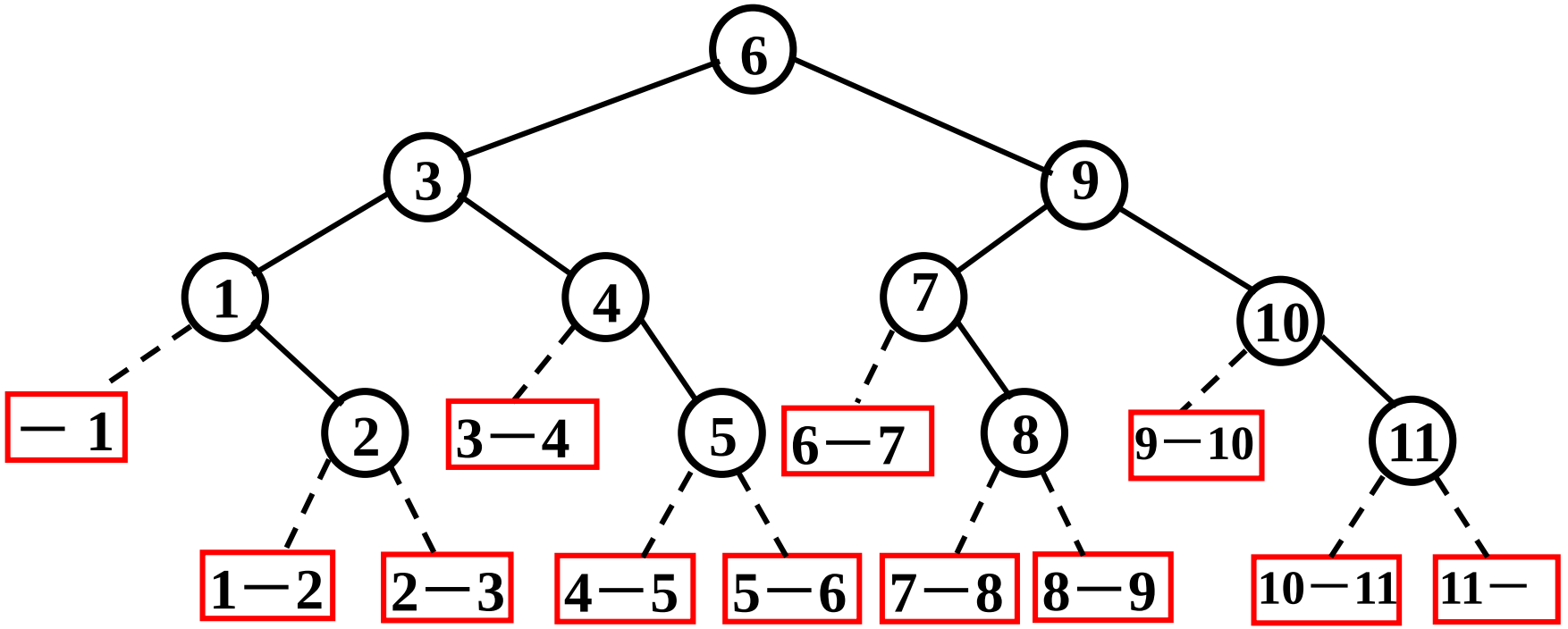


平衡二叉树：右子树节点数-左子树节点数=0或1

○ 内部结点---查找成功　　□ 外部结点---失败结点

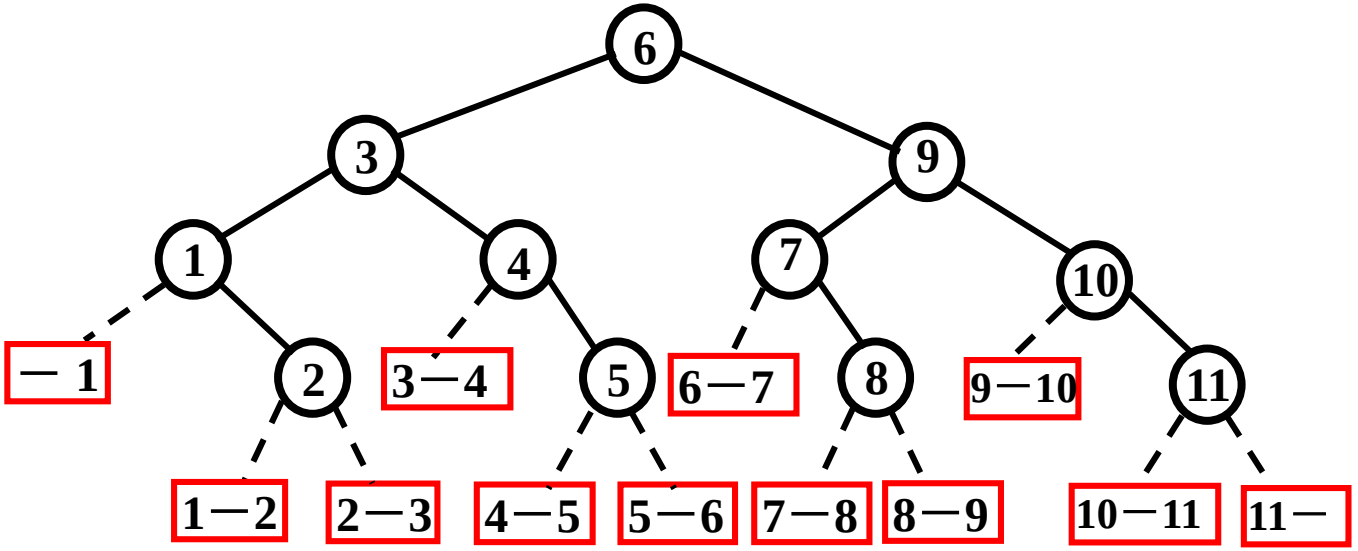
8.3 折半查找

- 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
  - $ASL_{成功}$=（1*1+2*2+3*4+4*4）/11 = 25/11
  - $ASL_{失败}$=（3*4+4*8）/12 = 44/12



8.3 折半查找

- 折半查找的ASL
  - 只有最后一层是不满的，树高h（不含失败节点）同完全二叉树，$h=\lceil \log_2(n+1) \rceil$
  - 查找成功ASL≤h，查找失败ASL≤h，时间复杂度 :$O(\log_2 n)$
  - 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$\lceil \log_2(n+1) \rceil$，折半查找的最坏性能与平均性能相当接近。



8.3 折半查找

- 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。

8.3 折半查找

- 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)  */
```

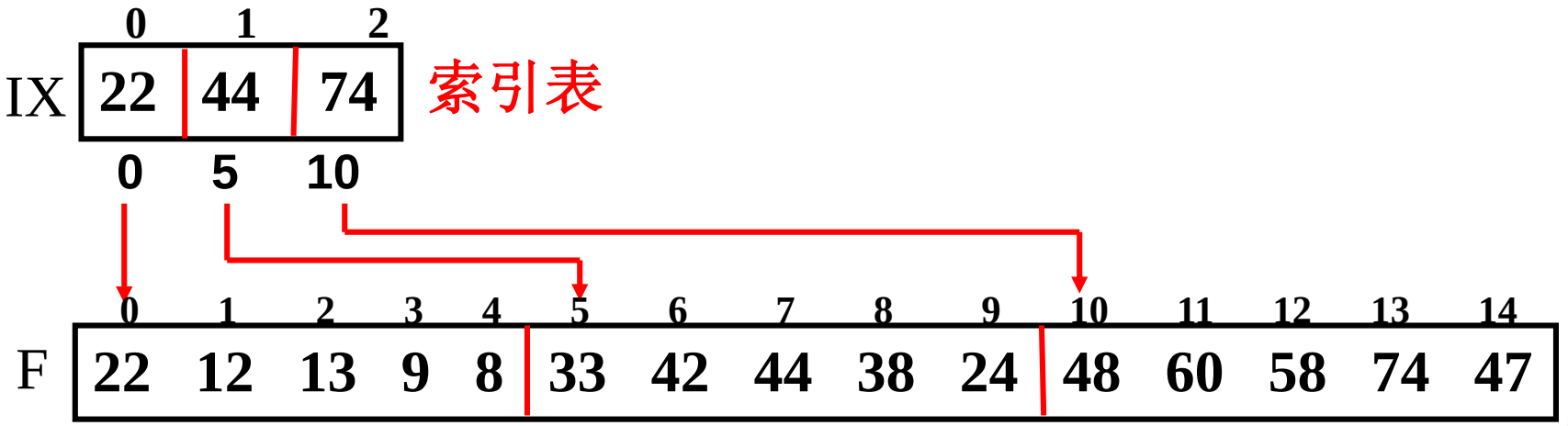
线性查找和折半查找的比较

	线性查找	折半查找
表的特征	表中元素无序	表中元素有序
存储结构	顺序和链表结构	顺序结构
插删操作	链表方便	尽量避免
时间复杂度	O(n)	O（logn）
ASL	大	小
适用范围	表长较短	表长较长 有序

分析：如何结合顺序表和有序表的特点？

8.4 分块查找----线性查找+折半查找

- 分块查找（索引顺序查找）的基本思想
  - 块间有序，块内无序：首先将表中的元素分成若干块，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.



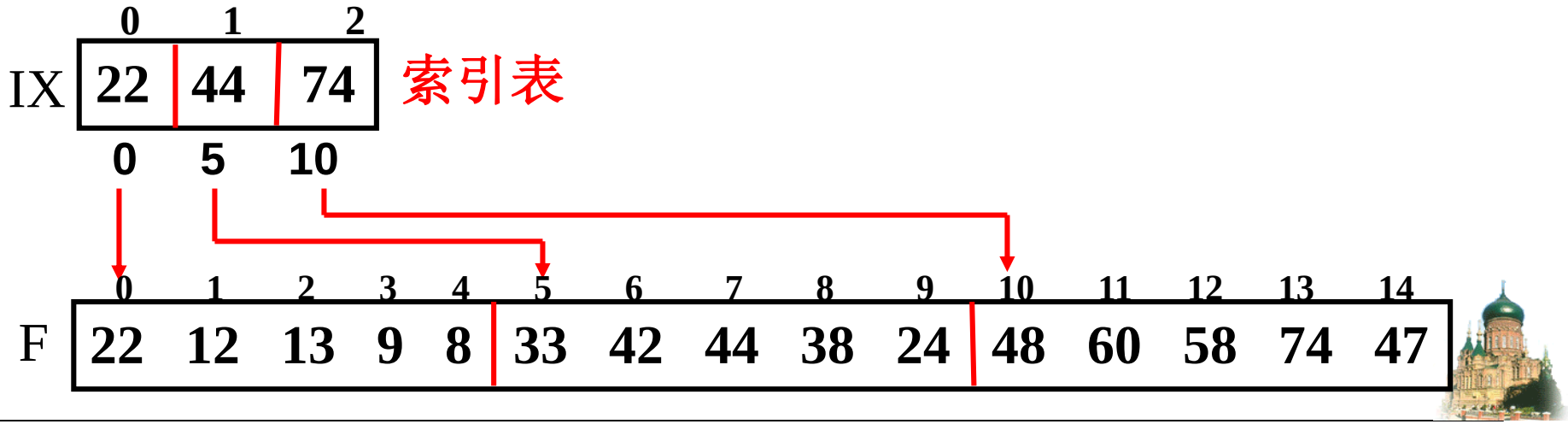
8.4 分块查找

- 算法的实现

```
typedef struct //索引表结点
{   keytype key;
    int addr;
}indextype;
```

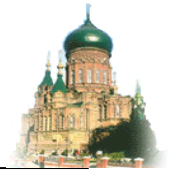
```
typedef struct ;// 索引表
{   indextype index[maxsize];
    int block;
} INtable;
```

- 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；
  - 然后到此块中进行进行顺序查找。



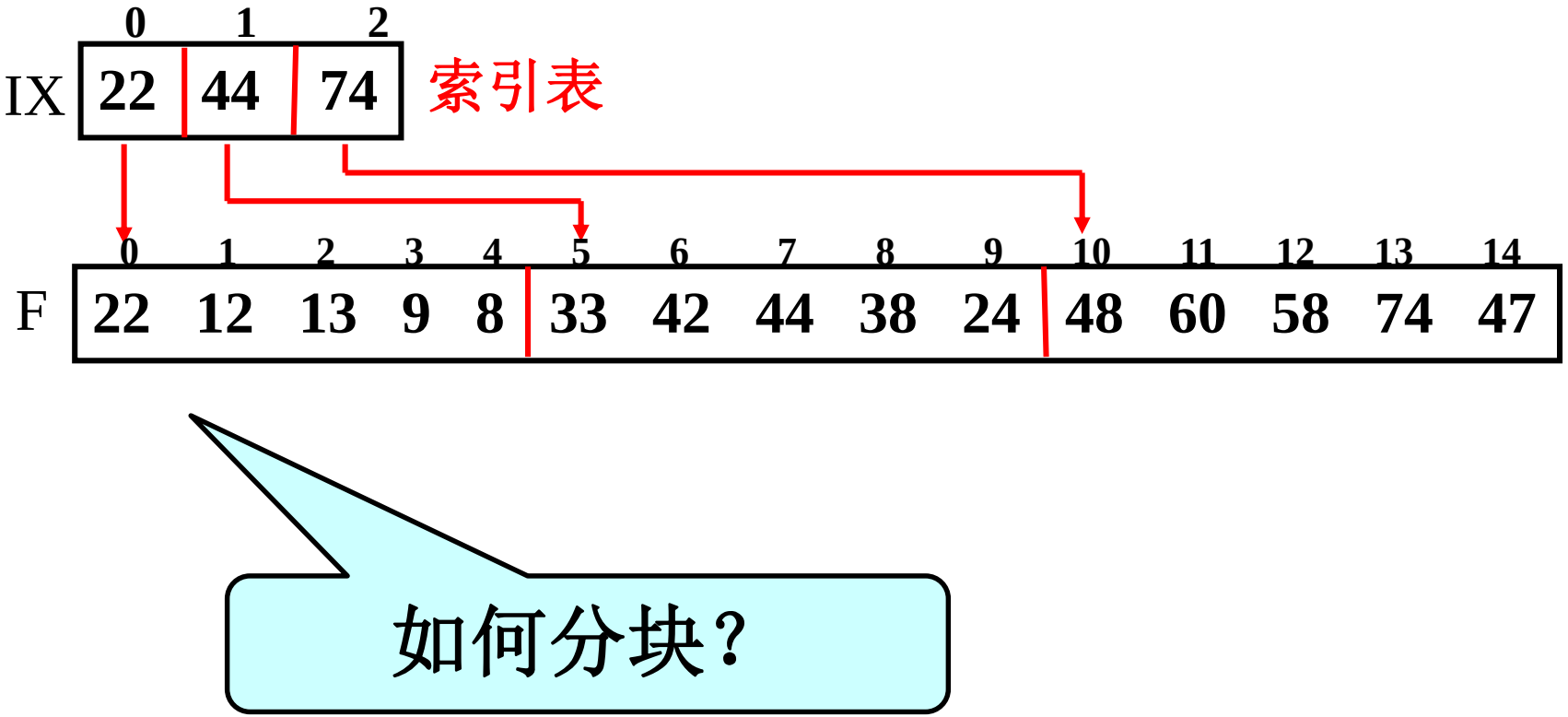
8.4 分块查找

```
typedef   keytype  INDEX[maxblock] ;// 线性表—索引表
int   index_search(keytype k, int blocks, INDEX ix, LIST F, int L )
//在查找表F（均匀成blocks块，每块长度为L）中查找关键字k，ix为索引表
 {   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 = j + 1 ;
        if ( k == F[ j ].key )  return   j ; // 查找成功
     }
     return   –1 ; /* 查找失败 */
}
```



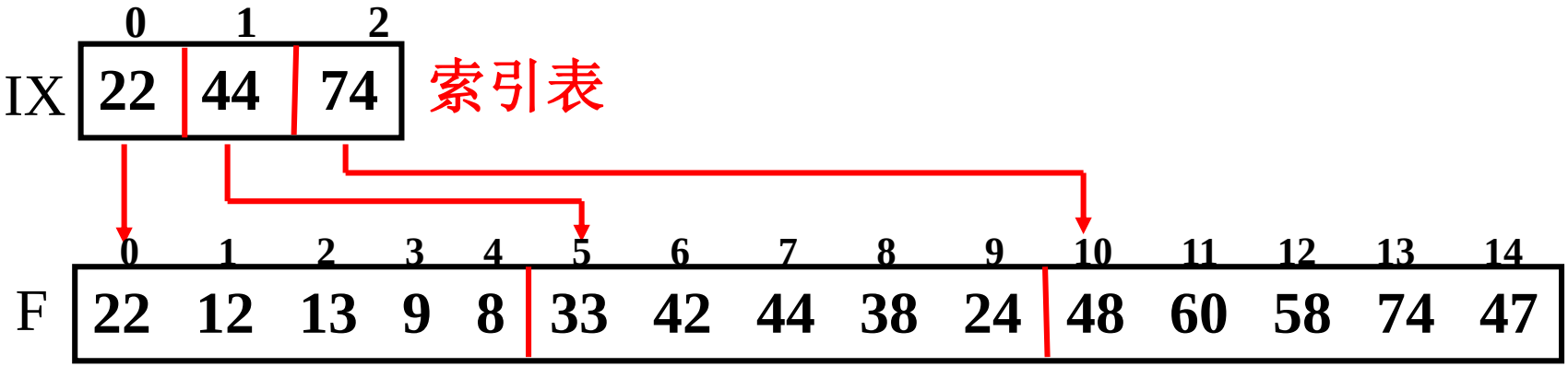
8.4 分块查找

- 分块查找性能分析



8.4 分块查找

- 分块查找性能分析



思考题：如果索引表长度为b，每块平均长度为L平均查找长度是多少？

答案：(b+1)/2+(L+1)/2。

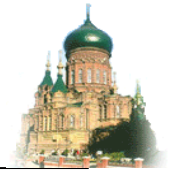
8.4 分块查找

分块查找性能分析

- 设长度为n 的表被均匀分成b 块，每块长度为L，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：

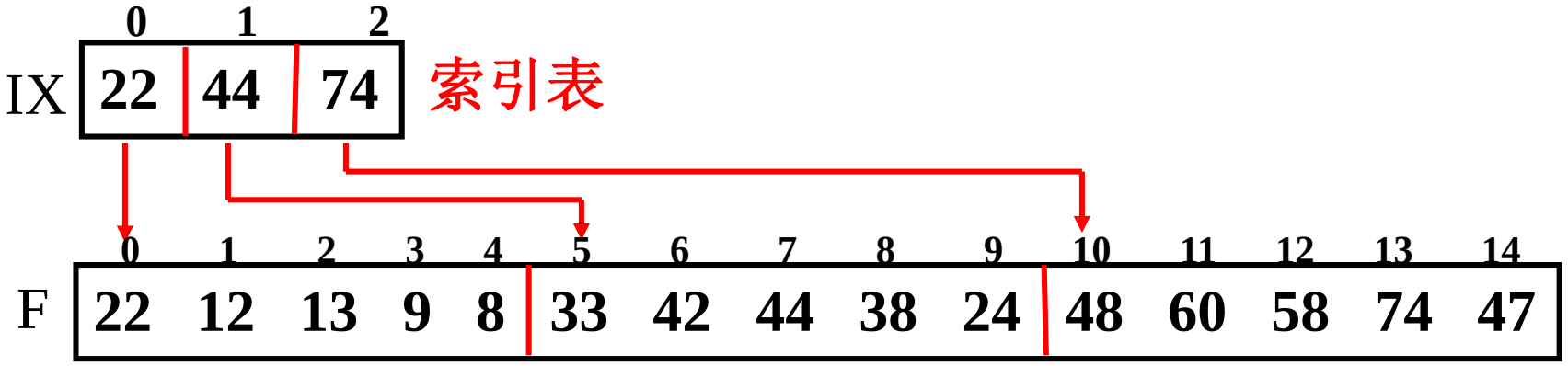
$$ASL(L)=ASL_{ix}+ASL_{blk} = (b+1)/2+(L+1)/2=(n/L+L)/2+1$$

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



8.4 分块查找

分块查找优化

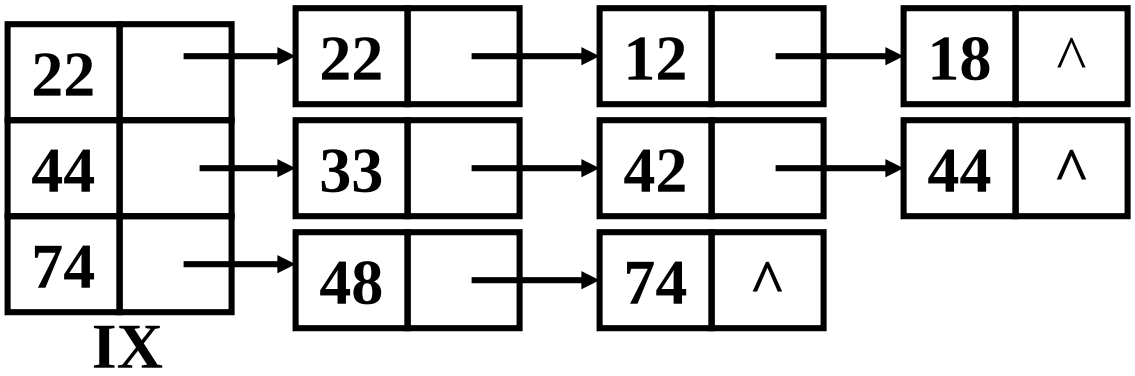


- 索引表非常大的情况
  - 折半查找索引表
  - 若索引表中不包含目标关键字，则折半查找索引表最终停在low>up，要在low所指分块中查找

- 只适合静态查找；
  - 把同一块中的元素组织成一个链表。

8.4 分块查找

- 分块查找局限性和改进
- 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



- 优点

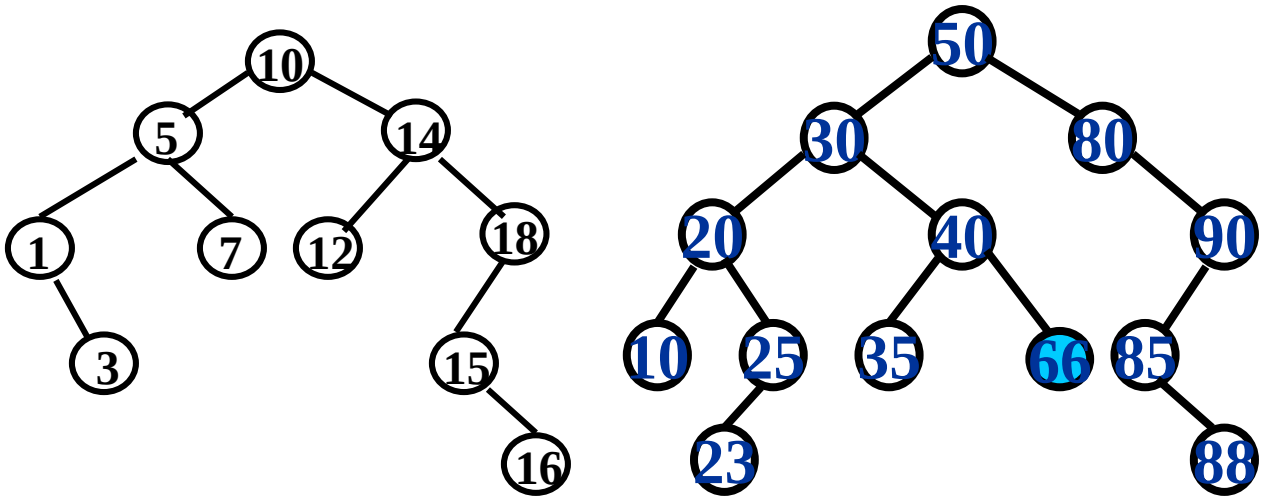
①在表中插入或删除一个记录时，只要找到该记录所属的块，就在该块内进行插入和删除运算。

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。

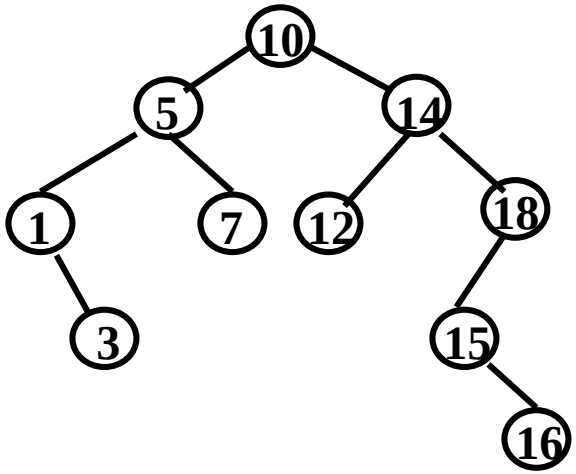
8.5 二叉查找树BST

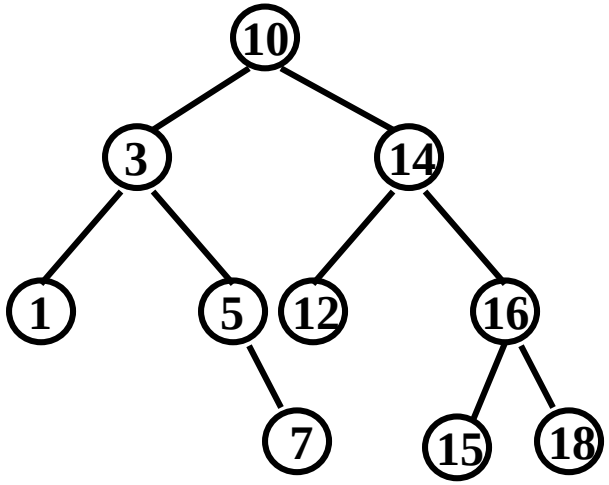
- 二叉查找树——二叉搜索树、二叉分类（排序）树
  - 二叉查找树或者是空树，或者是满足下列性质的二叉树：
    - 若它的左子树不空，则左子树上所有结点的关键字的值都小于根结点关键字的值；
    - 若它的右子树不空，则右子树上所有结点的关键字的值都大于根结点关键字的值；
    - 它的左、右子树本身又是一个二叉查找树。



8.5 二叉查找树BST

- 二叉查找树（二叉排序树）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，建立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。



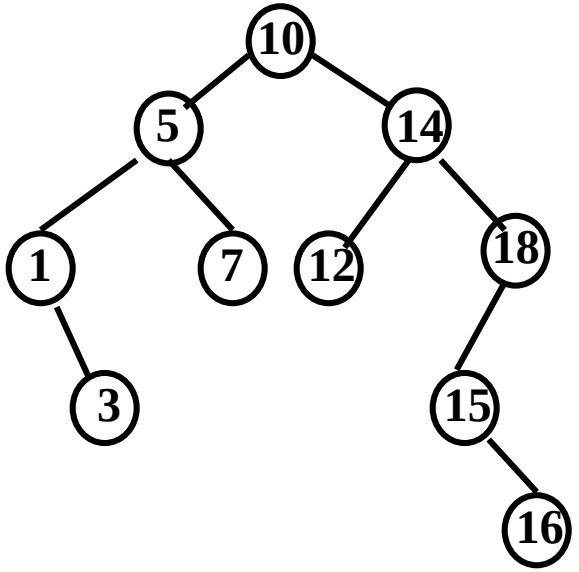


8.5 二叉查找树BST

- 二叉查找树的结构特点：
  - 能像有序表那样进行高效查找；
  - 能像链表那样灵活删除。

- 二叉查找树的存储结构：

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

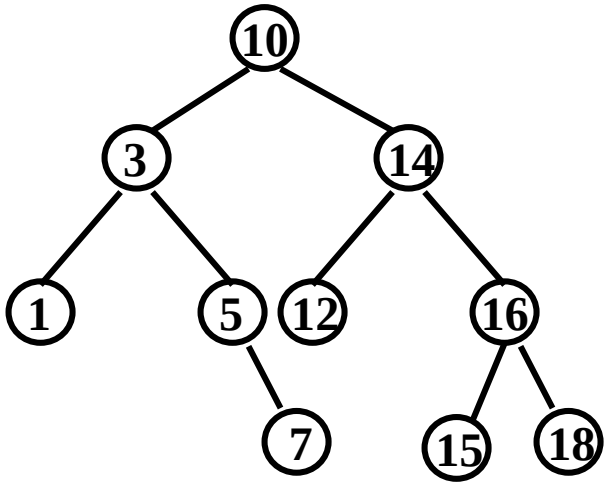


8.5 二叉查找树BST

- 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：

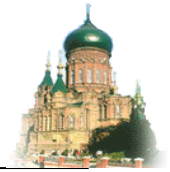
- 若F = Null，则查找失败；否则，
- k ==F->data.key，则查找成功；否则，
- k < F->data.key，则递归地在F 的左子树查找k；否则
- k > F->data.key，则递归地在F 的右子树查找k。



8.5 二叉查找树BST

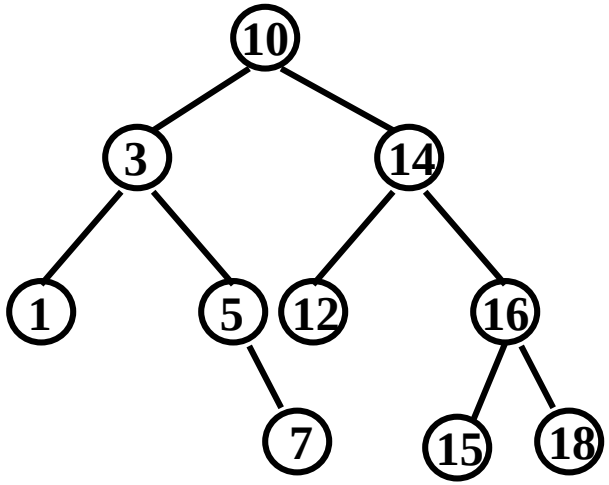
- 二叉查找树的查找操作（递归）：

```
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
}
```



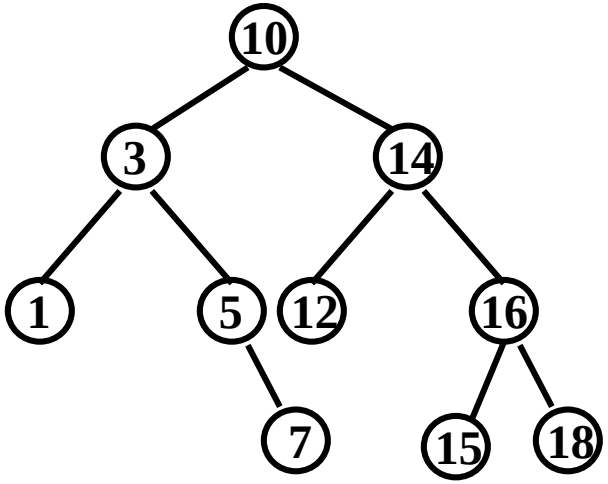
8.5 二叉查找树BST

- 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;
   if(F == NULL)
      F = p;
    while(F!=NULL)
   {     if(R.key < F->data.key)
           if(F->lchild) F =F->lchild;
           else {F->lchild=p; return;}
         else if (R.key > F->data.key)
            if(F->rchild) F =F->rchild;
            else {F->rchild=p; return;}
         else
            return;
      }
    return;
}
```



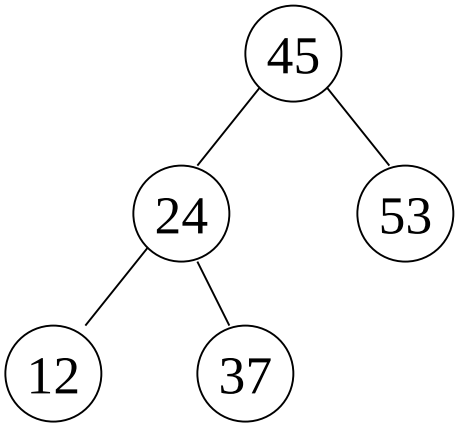
8.5 二叉查找树BST

二叉查找树的建立

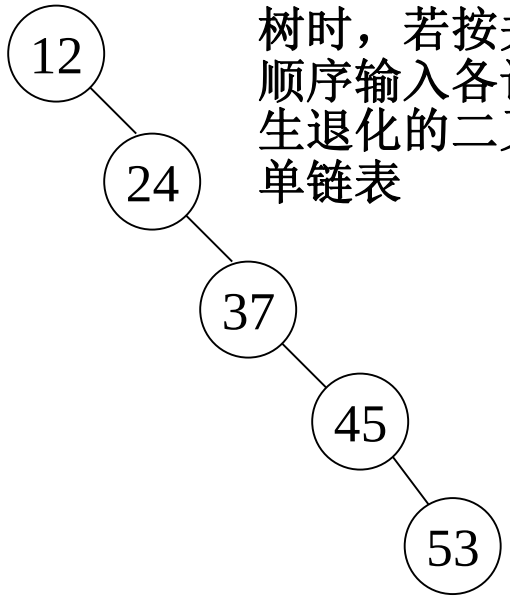
```
BST  CreateBST ( void )
{  BST  F = NULL; //初始时F为空
   keytype  key;
   cin>>key>>其他字段;//读入一个记录
   while( key ){  //假设key=0是输入结束标志
      InsertBST( R , F );  // 插入记录R
      cin>>key>>其他字段 ;//读入下个记录
    }
    return   F;//返回建立的二叉查找树的根
}
```

8.5 二叉查找树BST

- 二叉查找树的建立
- 对序列{45,24,53,12,37}建立二叉树
- 对序列{12,24,37,45,53}建立二叉树



ASL =（1+2+2+3+3）/ 5 = 2.2



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

ASL =（1+2+3+4+5）/ 5 = 3

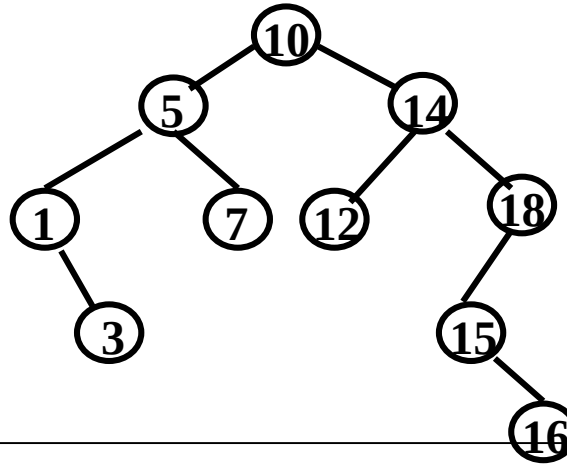
- 如何防止？
- ✓ 随机输入各结点
- ✓ 在建立、插入和删除各结点过程中平衡相关结点的左、右子树。

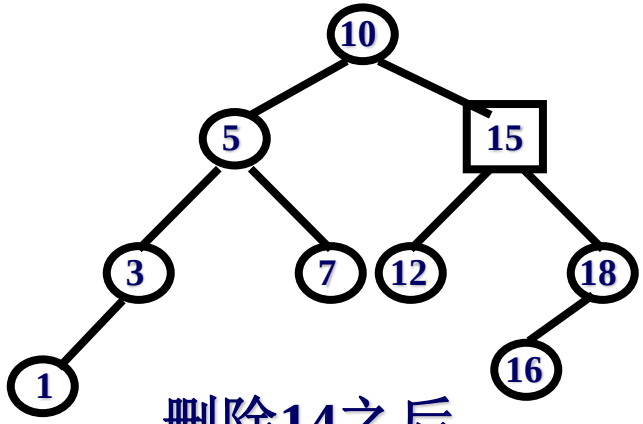
8.5 二叉查找树BST

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：
  - 把左子树作为右子树中最小结点的左子树。或者把右子树作为左子树中最大结点的右子树。
  - 则用继承结点（中序后继结点）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。



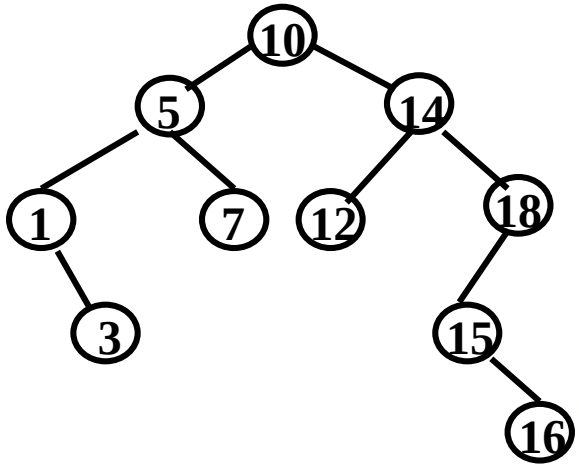


删除14之后

8.5 二叉查找树BST

- 二叉查找树的删除操作的实现步骤

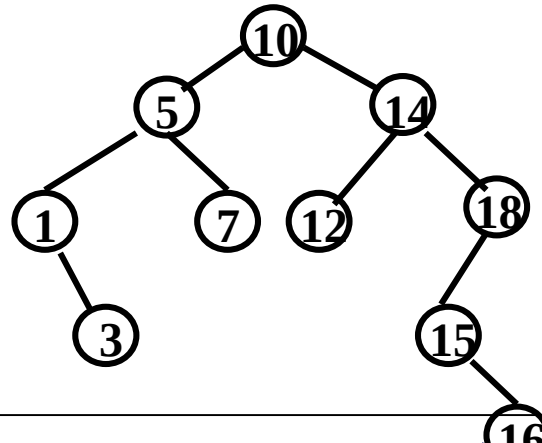
1. 若结点p是叶子，则直接删除结点p；
2. 若结点p只有左子树，则只需重接p的左子树；
   若结点p只有右子树，则只需重接p的右子树；
3. 若结点p的左右子树均不空，则
   3.1 查找结点p的右子树上的最左下结点s；
   3.2 将结点s数据域替换到被删结点p的数据域；
   3.3 若节点s有右子树，则重接s的右子树；
   3.4 删除结点s；



8.5 二叉查找树BST

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) ;
}
```

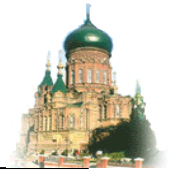



```
void  DeleteB( keytype  k, BST &F )
{ if ( F != Null)
    if ( k < F->data.key )
      DeleteB( k, f->lchild ) ;
    else if  ( k > F->data.key )
      DeleteB( k, f->rchild );
    else
      if ( F->rchild == Null )
        F = F->lchild ;
      else if( F->lchild == Null )
        F = F->rchild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

8.5 二叉查找树BST

二叉查找树的性能

- 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$\log_2 n$。
- 二叉查找树的插入删除时间复杂度同查找一致，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。

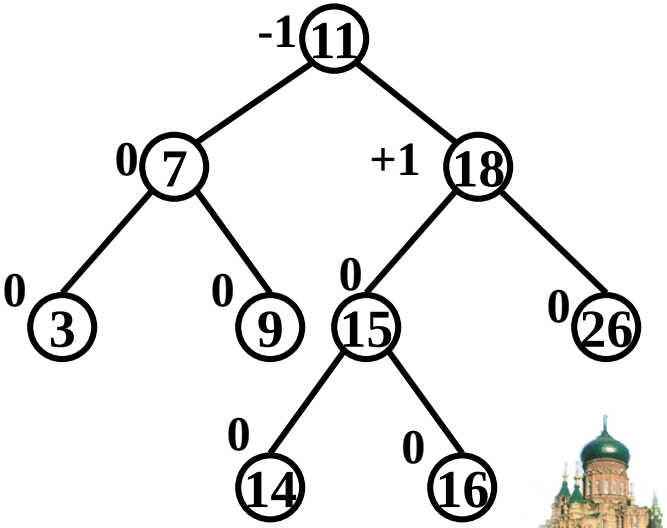


8.6 平衡二叉树（AVL树）

1962年,Adelson-Velskii和Landis提出的

为了保证树的高度为logn，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(logn)，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(logn)，从而确保树上的基本操作在最坏情况下的时间均为O(logn)。

- **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**



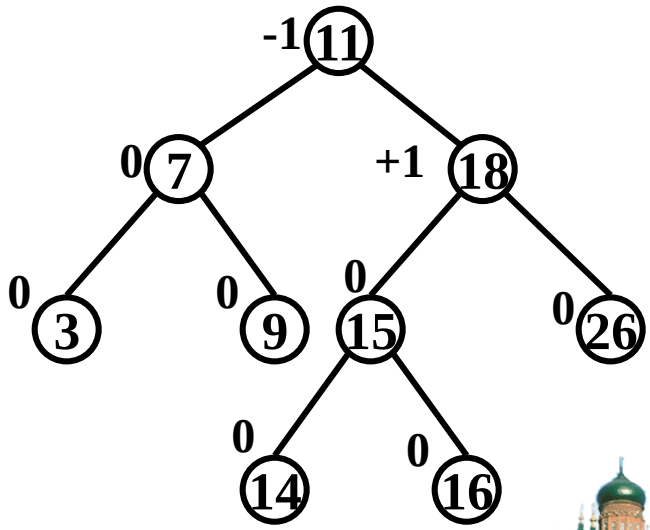
8.6 平衡二叉树（AVL树）

1962年,Adelson-Velskii和Landis提出的

为了保证树的高度为logn，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(logn)，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(logn)，从而确保树上的基本操作在最坏情况下的时间均为O(logn)。

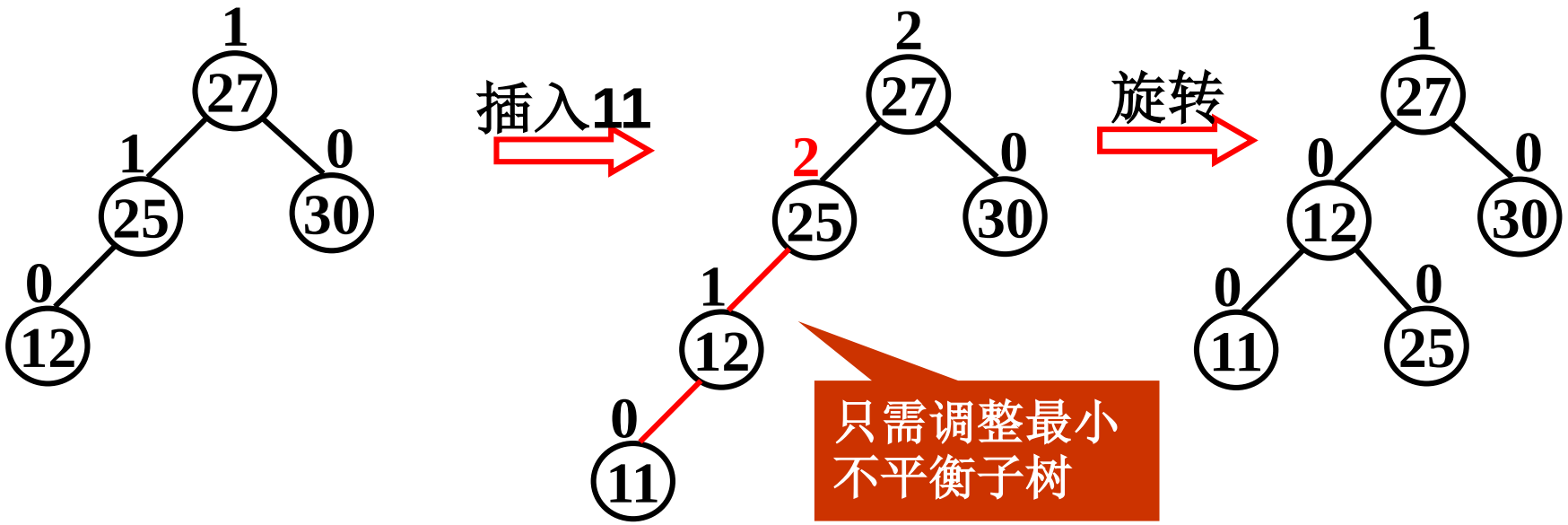
- **AVL树的查找操作**
  - 与BST的相同
  - AVL树的ASL可保持在$O(\log_2 n)$

```
struct Node {
      ElementType  data ;
                   int   bf ;
 struct Node *lchild, *rchild ;
}
Typedef   Node  *AVLT ;
```



8.6 AVL树

- 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



查找路径上的所有结点
都有可能受到影响

8.6 AVL树

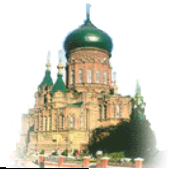
- AVL树的平衡化处理
  - 在一棵AVL树上插入结点可能会破坏树的平衡性，需要平衡化处理恢复平衡，且保持BST的结构性质。
  - 若用Y表示新插入的结点，A表示离新插入结点Y最近的，且平衡因子变为±2的祖先结点。
  - 调整即以A为根的子树。
  - 可以用4种旋转进行平衡化处理：

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）

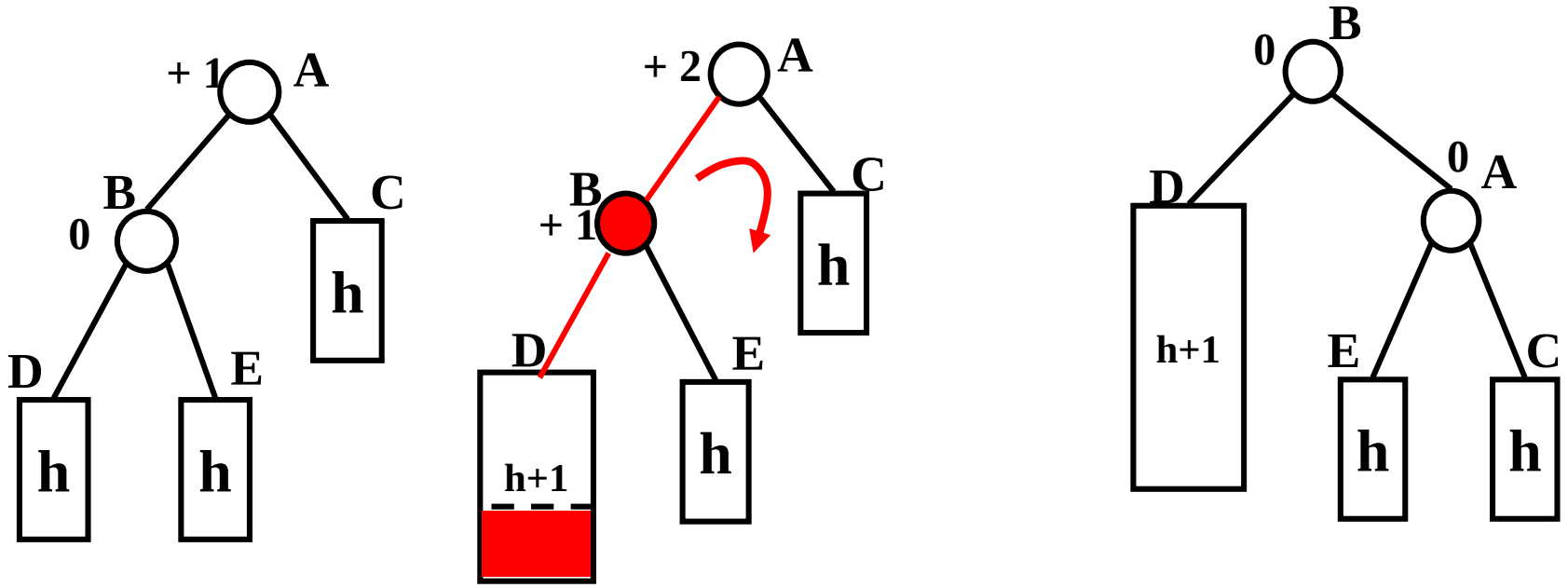
③ LR型：新结点Y 被插入到 A 的左子树的右子树上（逆、顺）

④ RL型：新结点Y 被插入到 A 的右子树的左子树上（顺、逆）



8.6 AVL树

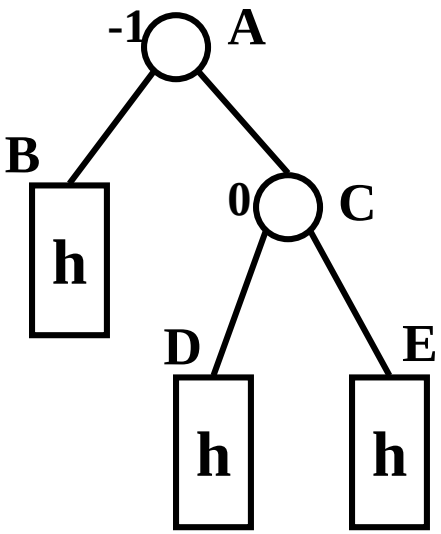
- AVL树的平衡化处理
  - LL型：新结点Y 被插入到 A 的左子树的左子树上（顺）



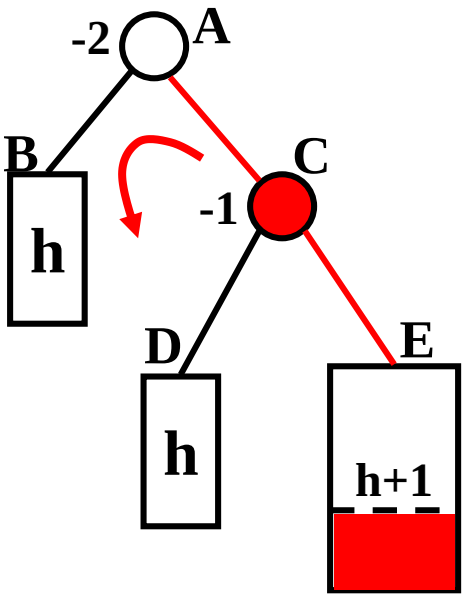
(a)AVL树　(b) D子树中插入结点　(c) 右向旋转后的AVL树

8.6 AVL树

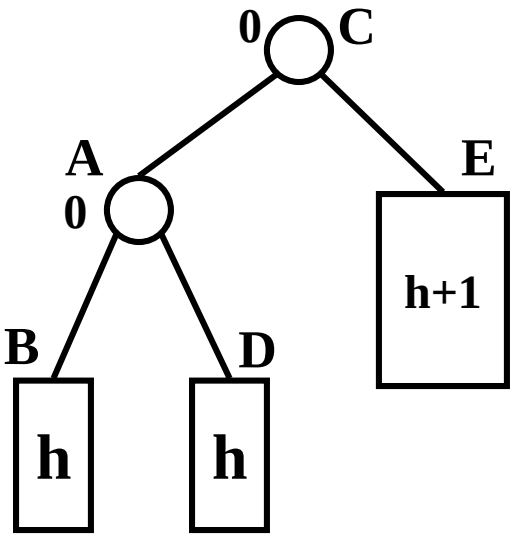
- AVL树的平衡化处理
  - RR型：新结点Y 被插入到 A 的右子树的右子树上（逆）



(a)AVL树



(b)E子树中插入结点

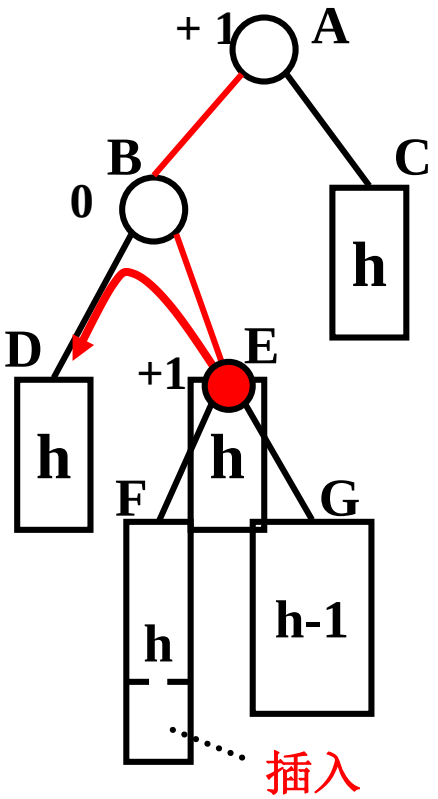


(c)左向旋转后的AVL树

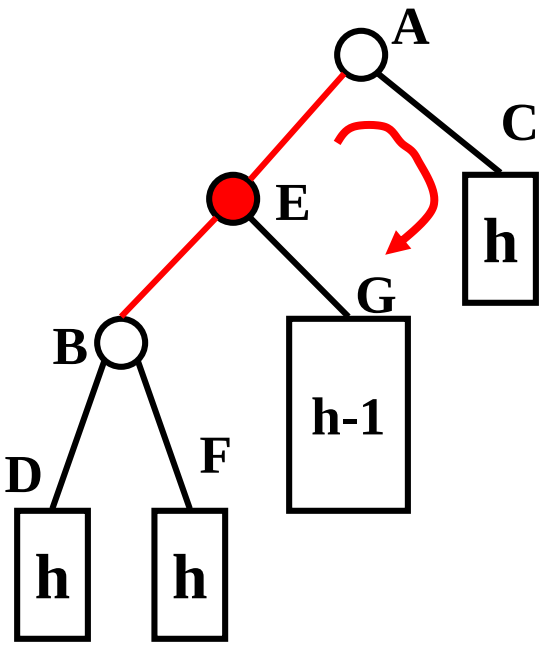
8.6 AVL树

AVL树的平衡化处理

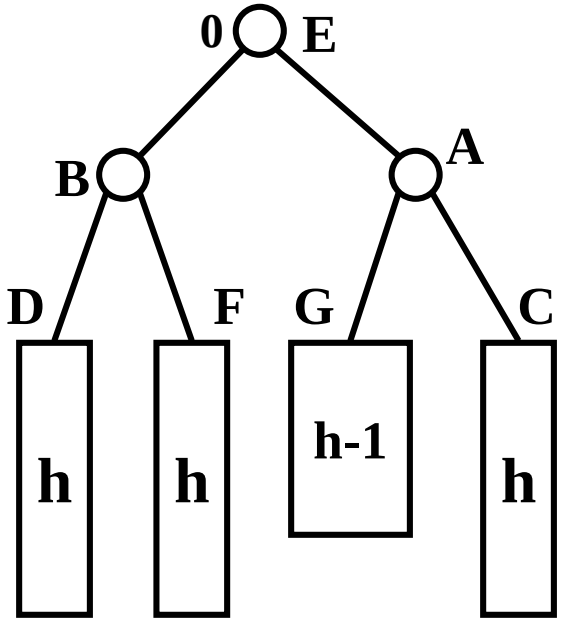
- **LR型**：新结点Y 被插入到 A 的左子树的右子树上(逆,顺)



(a)F子树插入结点
高度变为h



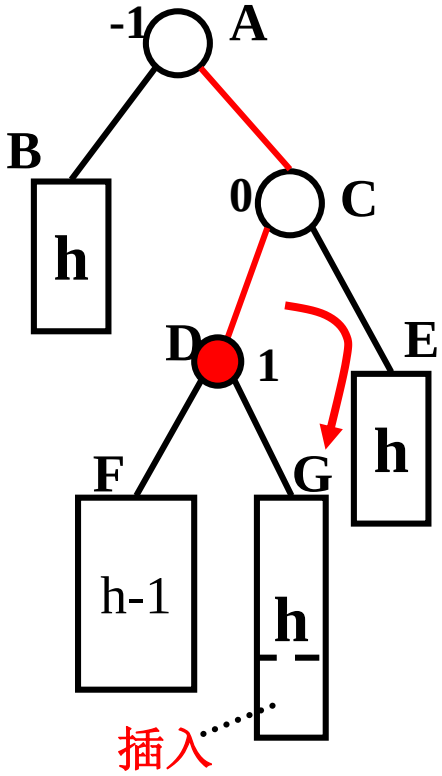
(b)绕E，将B
逆时针转后



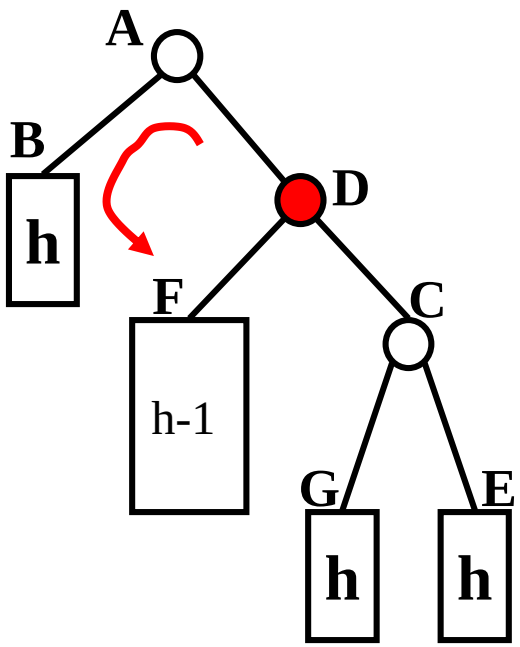
(c)绕E，将A
顺时针转后

8.6 AVL树

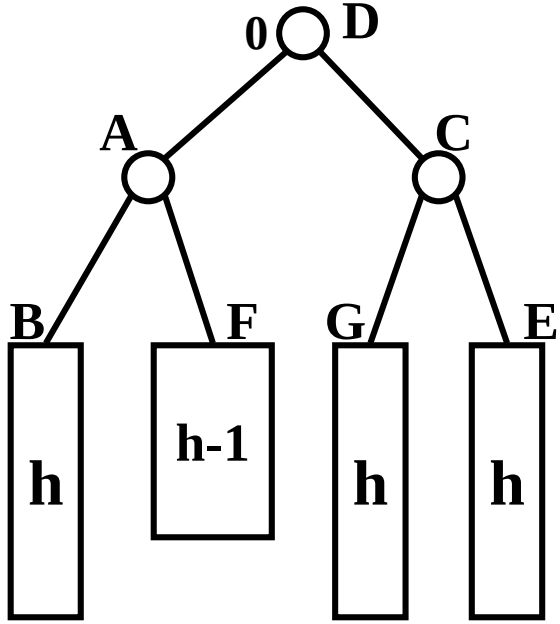
- AVL树的平衡化处理
  - RL型：新结点Y 被插入到 A 的右子树的左子树上(顺, 逆)



(a)G子树插入结点高度变为h



(b)绕D，C顺时针转之后

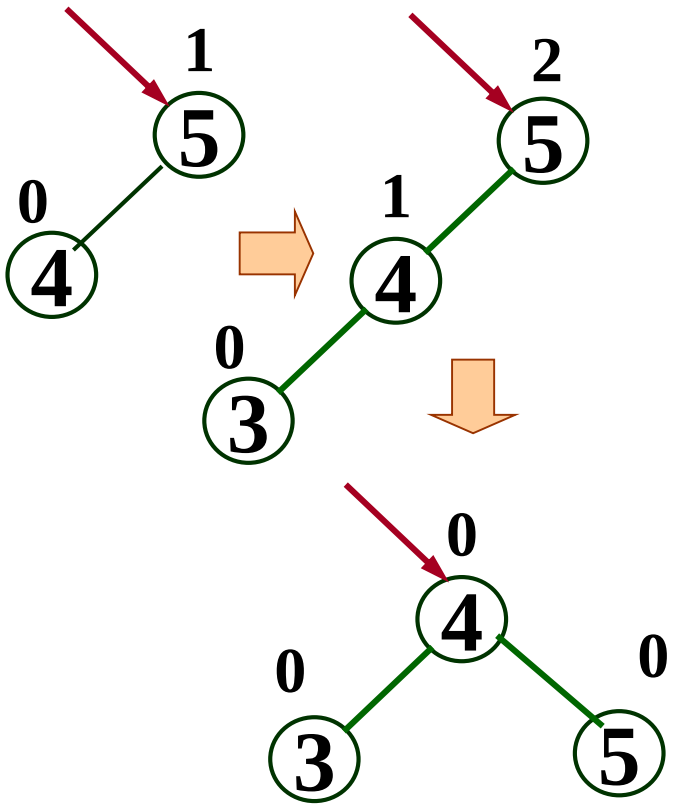


(c)绕D，A逆时针转之后

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5，4，3，8，19,1,2，25，23

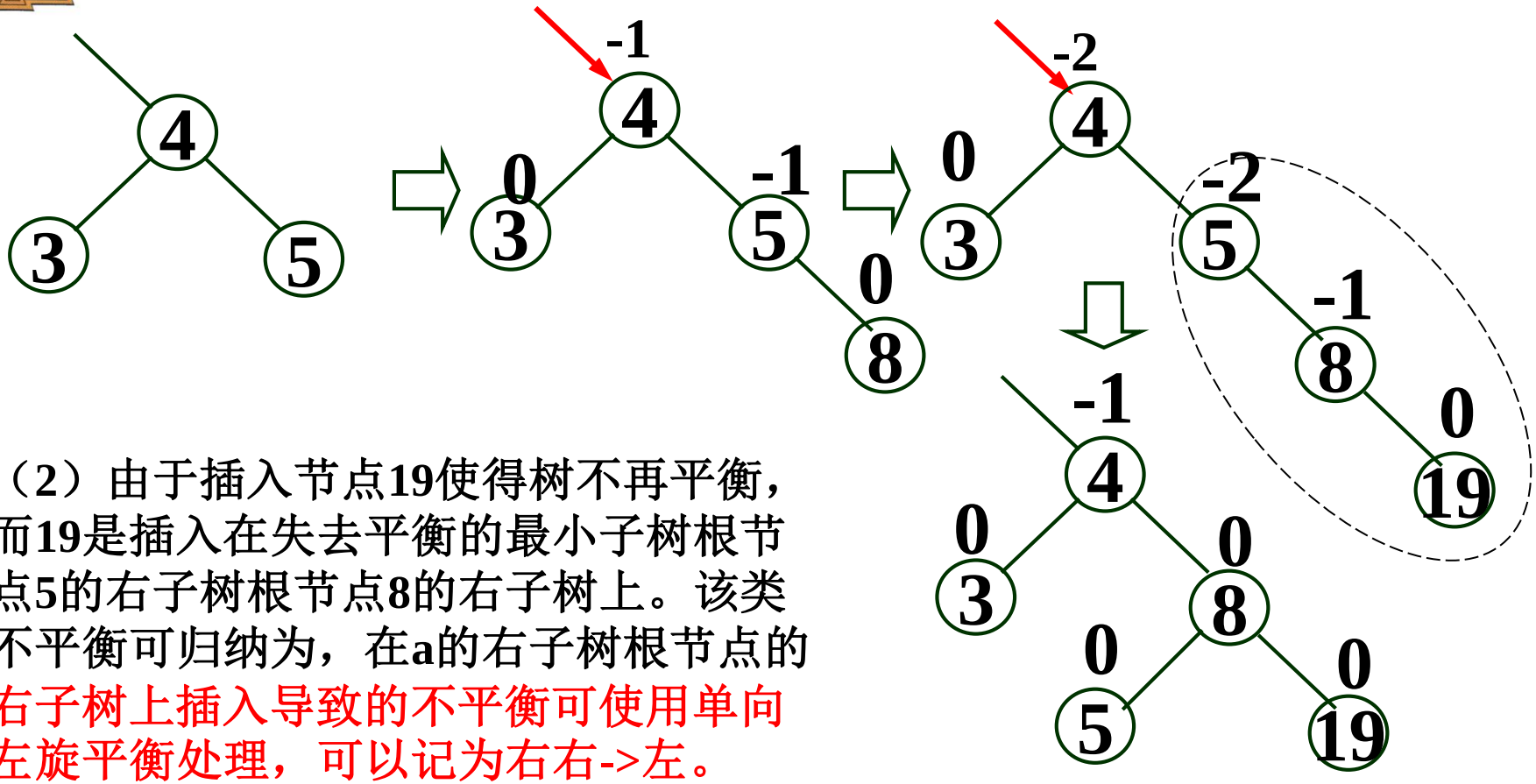


（1）由于插入节点3使得树不再平衡，而3是插入在失去平衡的最小子树根节点5的左子树根节点4的左子树上。

可以定义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）

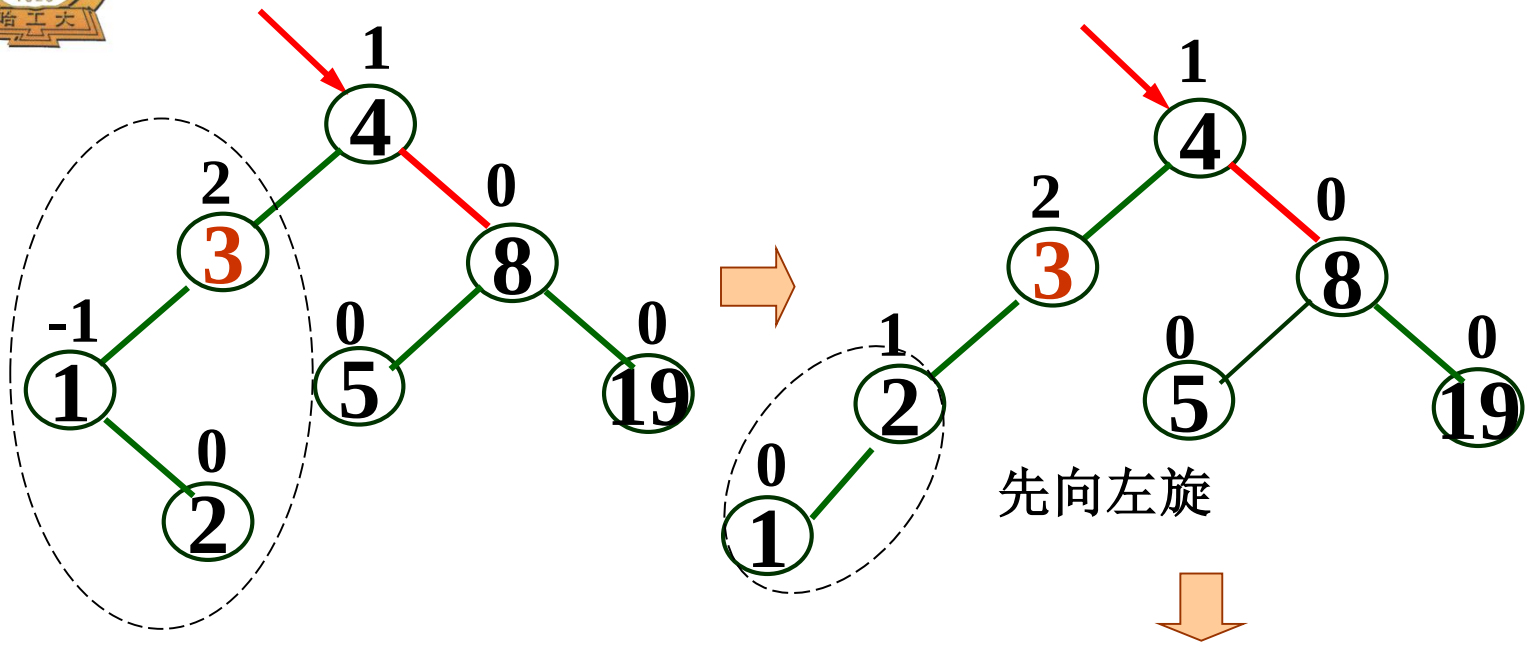
依次插入的关键字为5, 4, 3, 8, 19,1,2，25，23



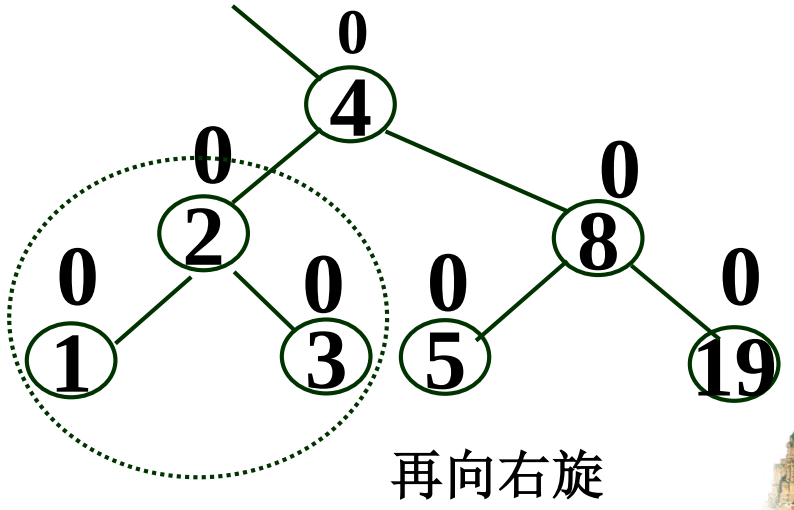
（2）由于插入节点19使得树不再平衡，而19是插入在失去平衡的最小子树根节点5的右子树根节点8的右子树上。该类不平衡可归纳为，在a的右子树根节点的右子树上插入导致的不平衡可使用单向左旋平衡处理，可以记为右右->左。

② RR型：新结点Y 被插入到 A 的右子树的右子树上（逆）

依次插入的关键字为5, 4, 3, 8, 19,1,2，25，23

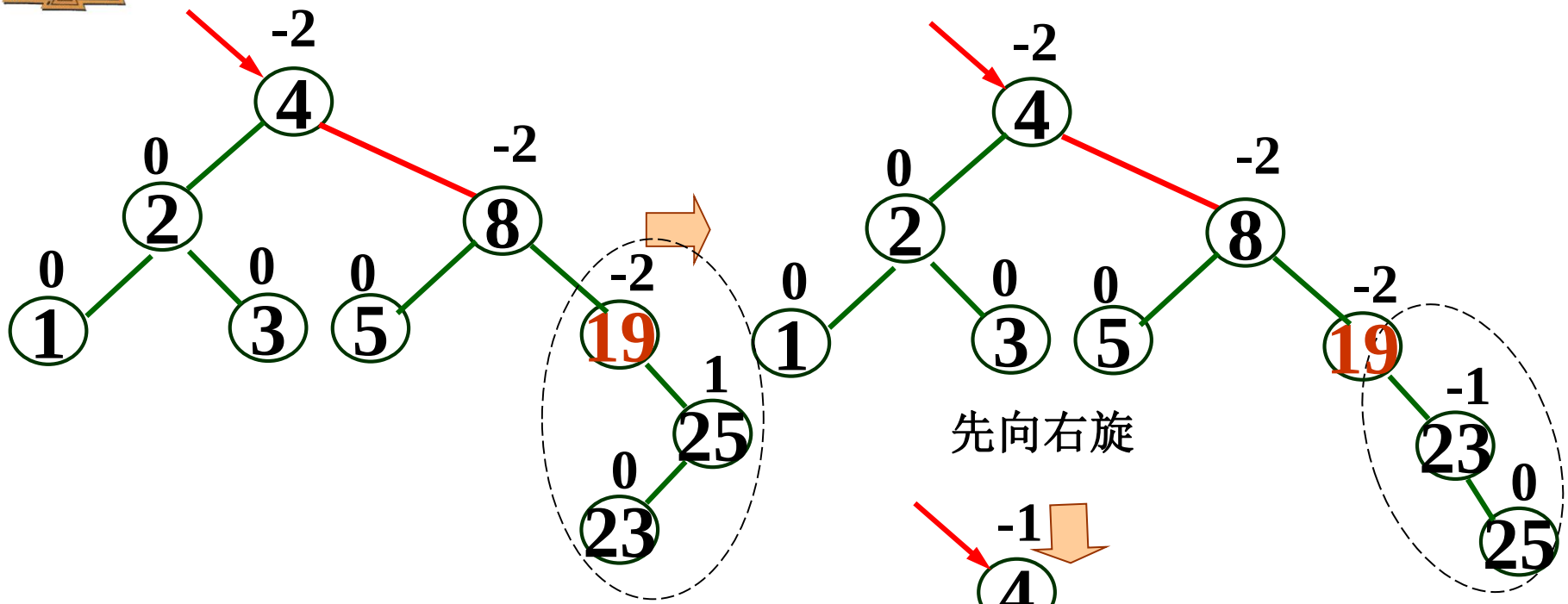


（3）在3的左子树根节点1的右子树上插入节点2导致不平衡，可使用双向旋转：先使其子树左旋再整棵树右旋，可记为左右->左右。

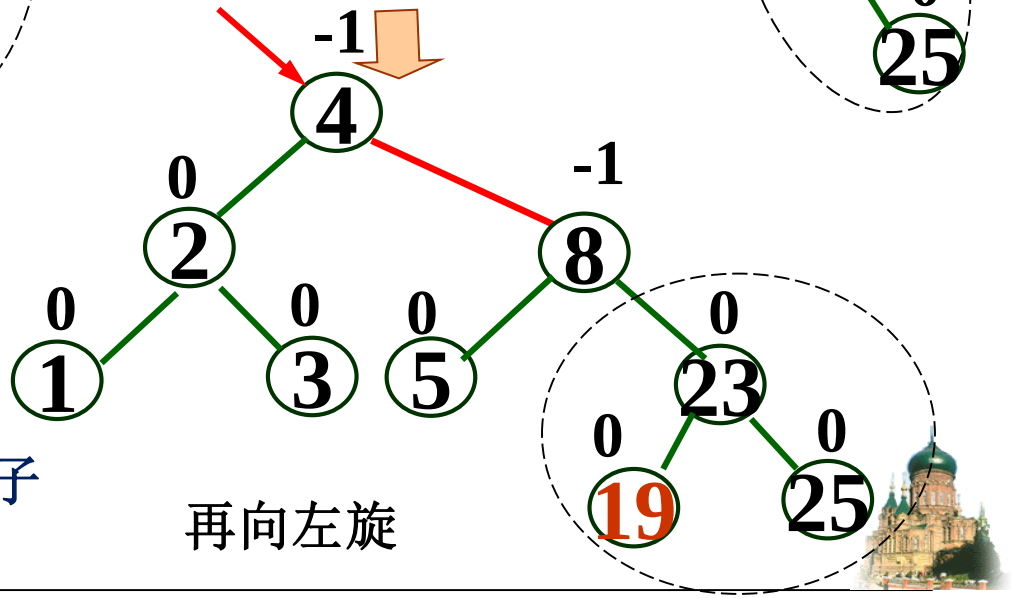


③ LR型：新结点Y 被插入到 A 的左子树的右子树上（逆、顺）

依次插入的关键字为5, 4, 3, 8, 19,1,2，25，23

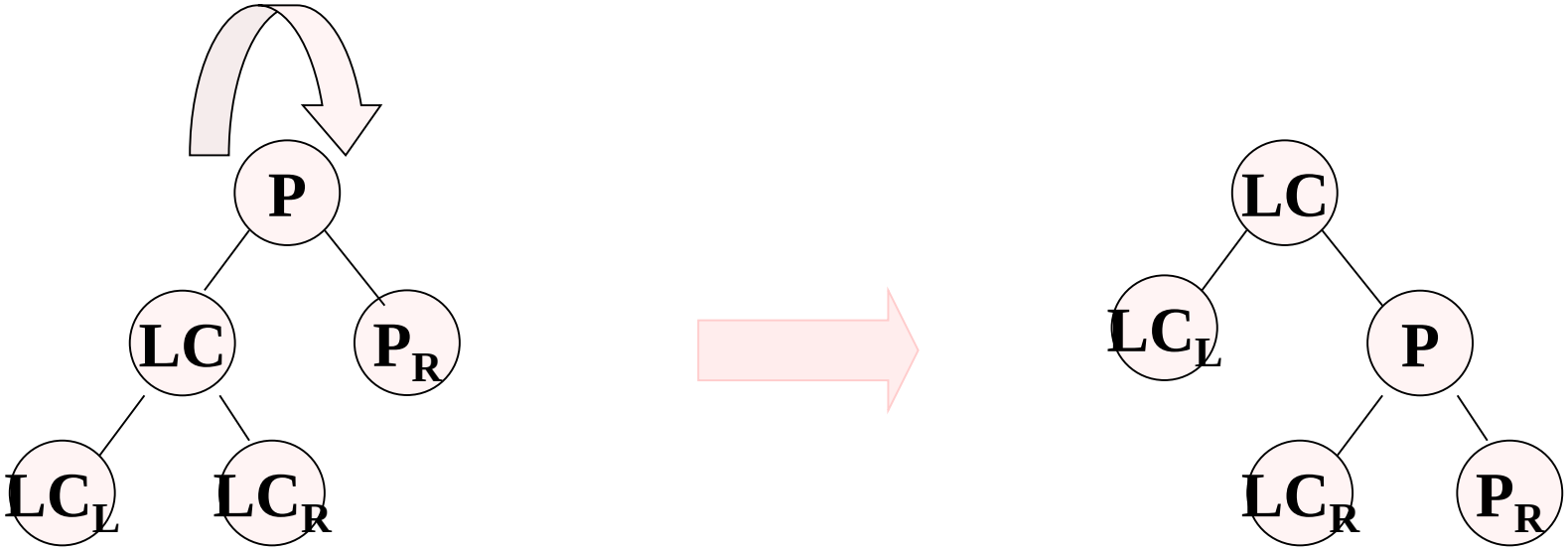


（4）在19的右子树根节点25的左子树上插入节点23导致不平衡，可使用双向旋转：先使其子树右旋再整棵树左旋，可记为右左->右左。

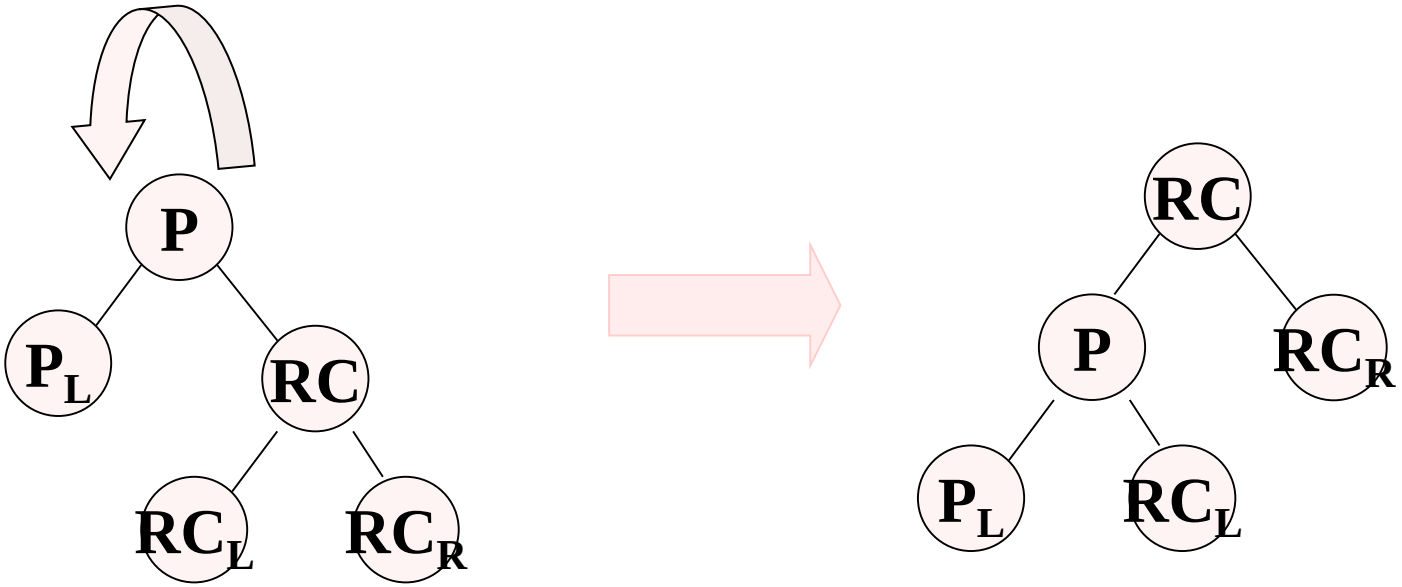


④ RL型：新结点Y 被插入到 A 的右子树的左子树上（顺、逆）

```
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=lc;
}
```



```
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;
}
```



8.6 AVL树

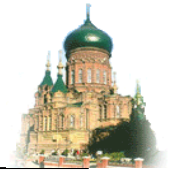
- AVL树的平衡化处理
  - 在一棵AVL树上插入结点可能会破坏树的平衡性，需要平衡化处理恢复平衡，且保持BST的结构性质。
  - 若用Y表示新插入的结点，A表示离新插入结点Y最近的，且平衡因子变为±2的祖先结点。
  - 调整即以A为根的子树。
  - 可以用4种旋转进行平衡化处理：

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）

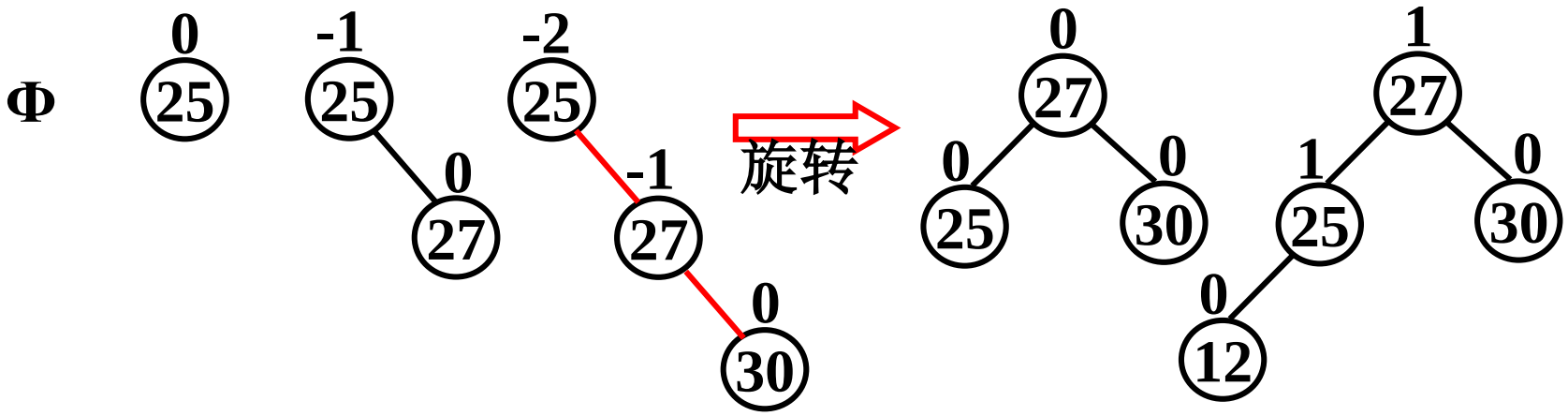
③ LR型：新结点Y 被插入到 A 的左子树的右子树上（逆、顺）

④ RL型：新结点Y 被插入到 A 的右子树的左子树上（顺、逆）



8.6 AVL树

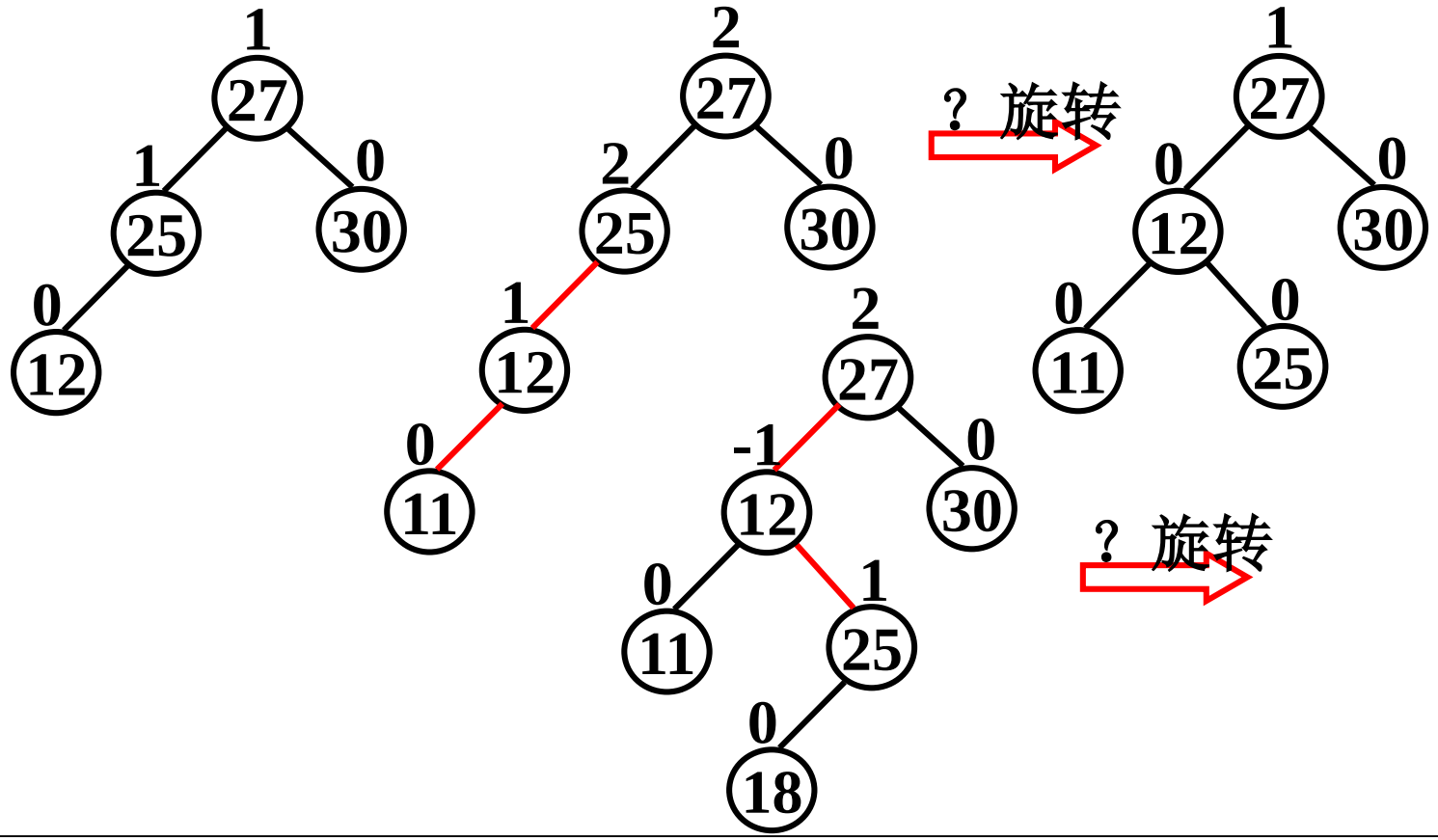
- AVL树的平衡化处理
  - 示例：假设25，27，30，12，11，18，14，20，15，22是一关键字序列，并以上述顺序建立AVL树。



8.6 AVL树

AVL树的平衡化处理

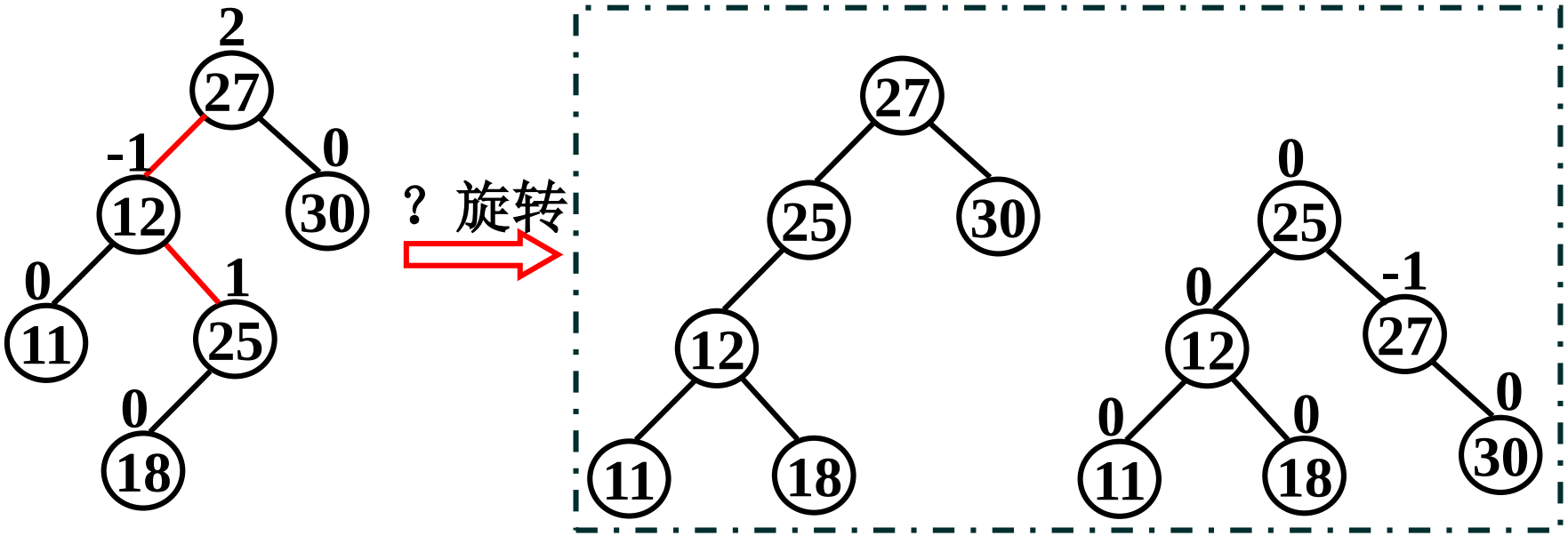
- 示例：假设25，27，30，12，11，18，14，20，15，22是一关键字序列，并以上述顺序建立AVL树。



8.6 AVL树

AVL树的平衡化处理

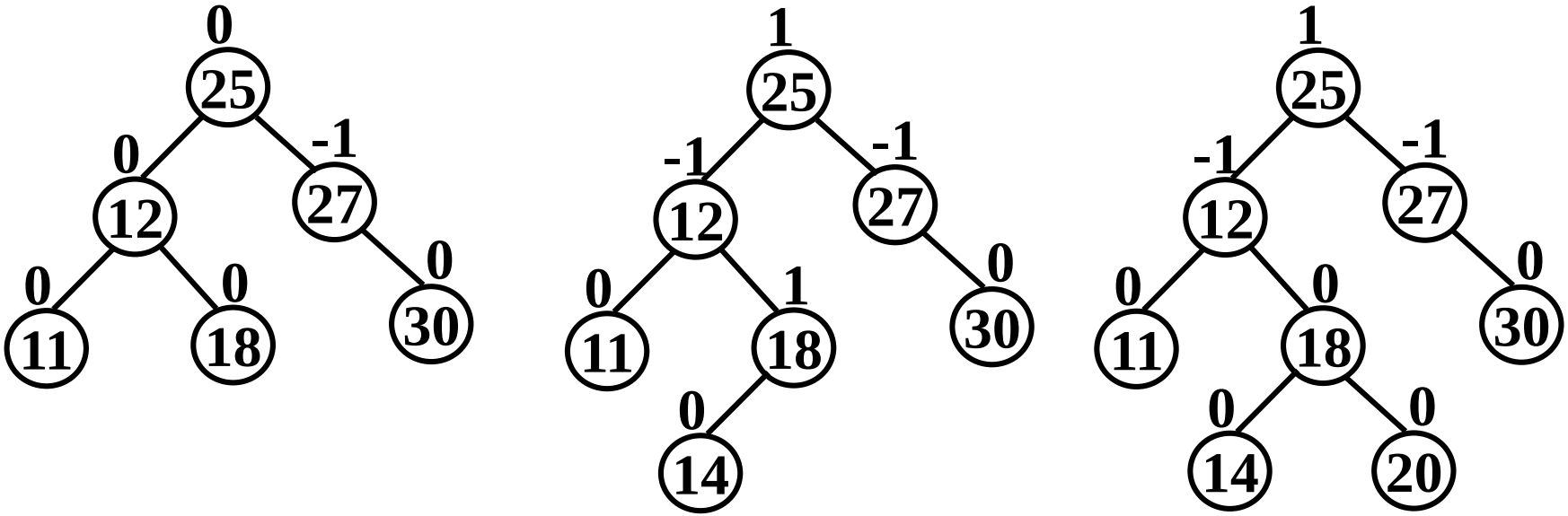
- 示例：假设25，27，30，12，11，18，14，20，15，22是一关键字序列，并以上述顺序建立AVL树。



8.6 AVL树

AVL树的平衡化处理

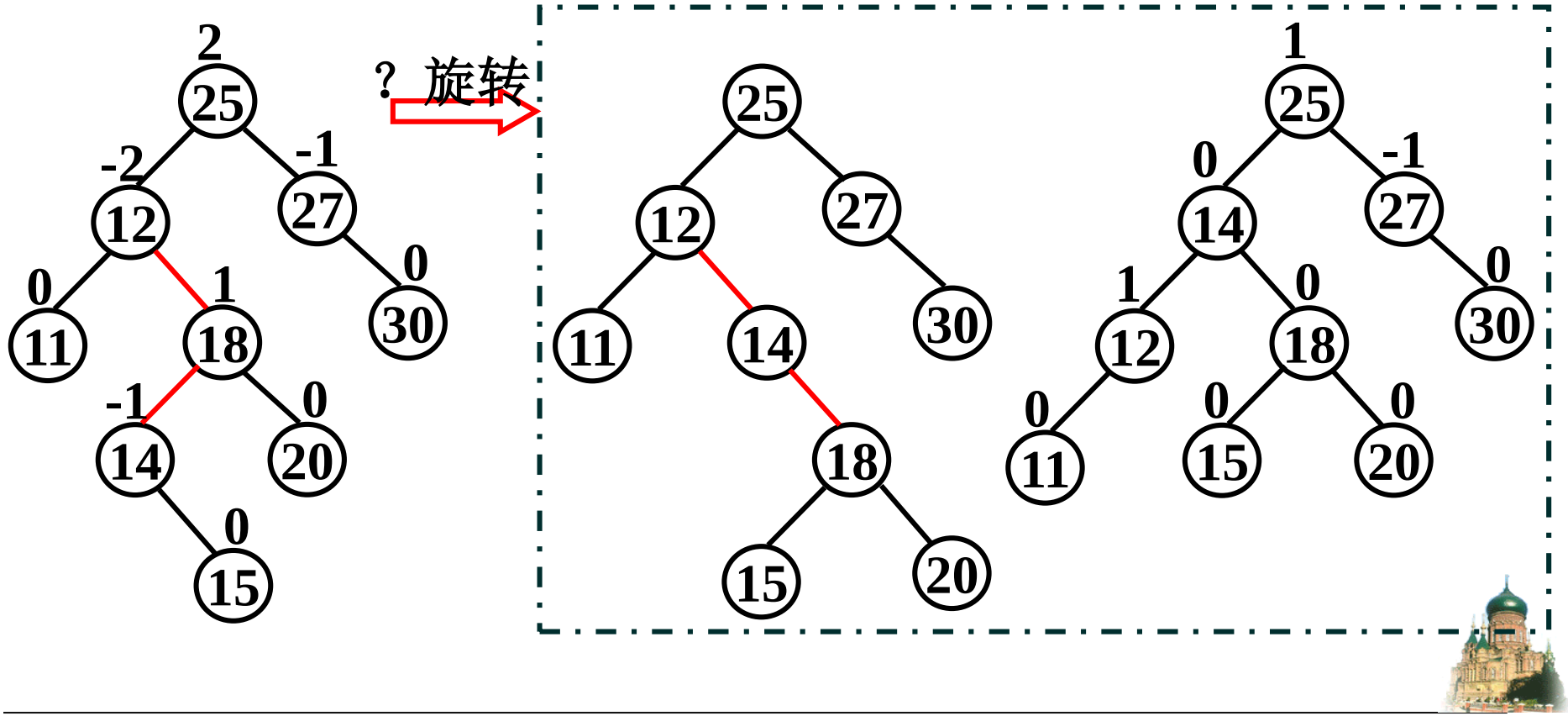
- 示例：假设25，27，30，12，11，18，14，20，15，22是一关键字序列，并以上述顺序建立AVL树。



8.6 AVL树

AVL树的平衡化处理

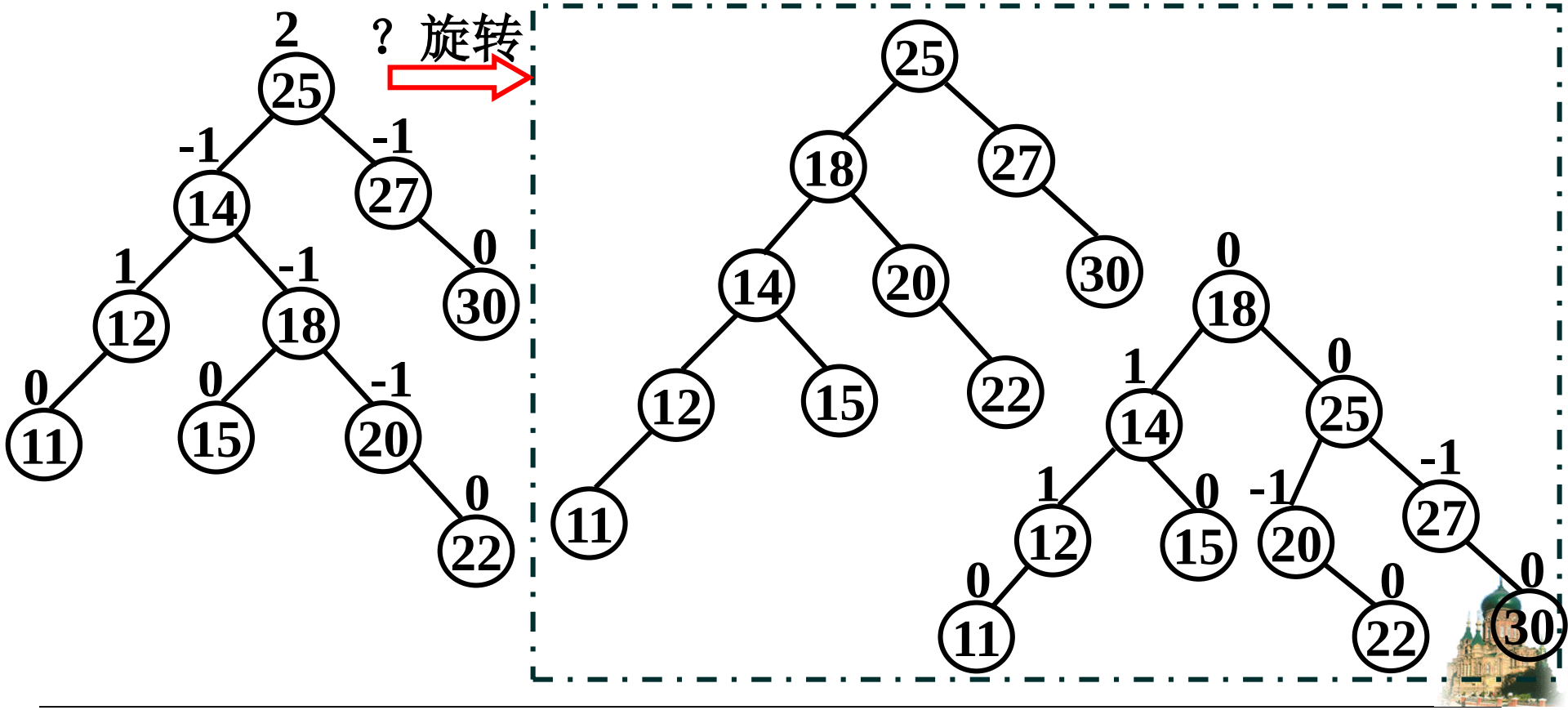
- 示例：假设25，27，30，12，11，18，14，20，15，22是一关键字序列，并以上述顺序建立AVL树。



8.6 AVL树

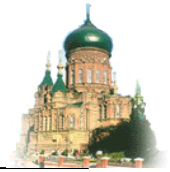
AVL树的平衡化处理

- 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。



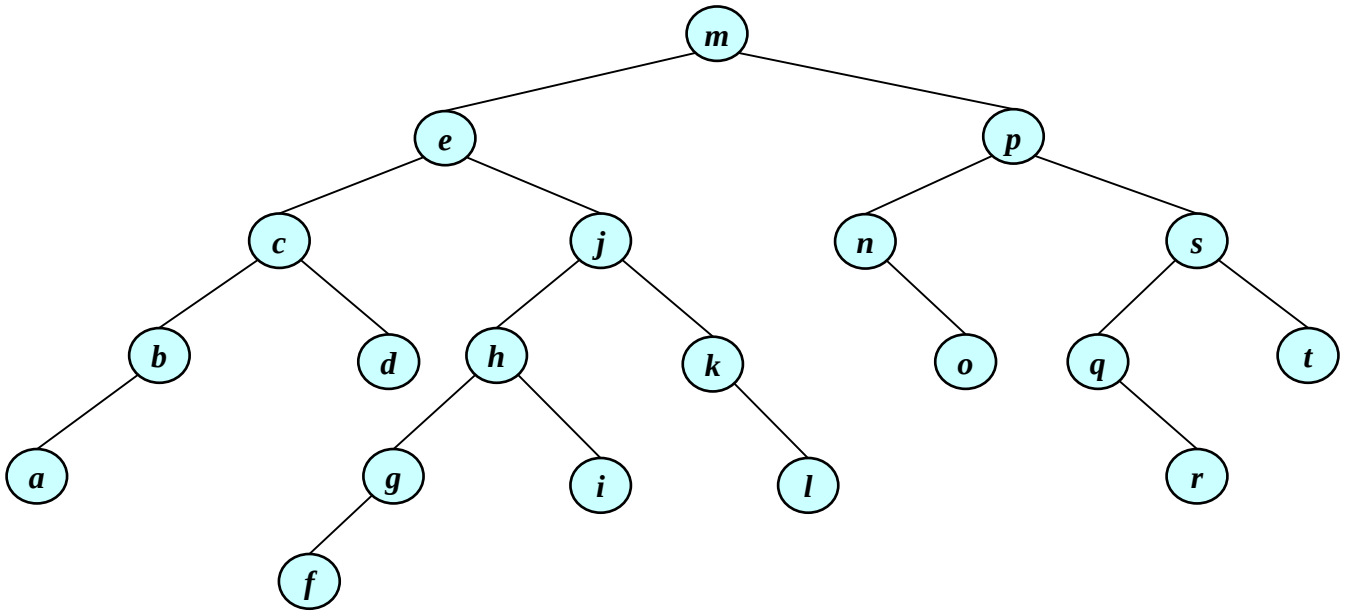
8.6 AVL树

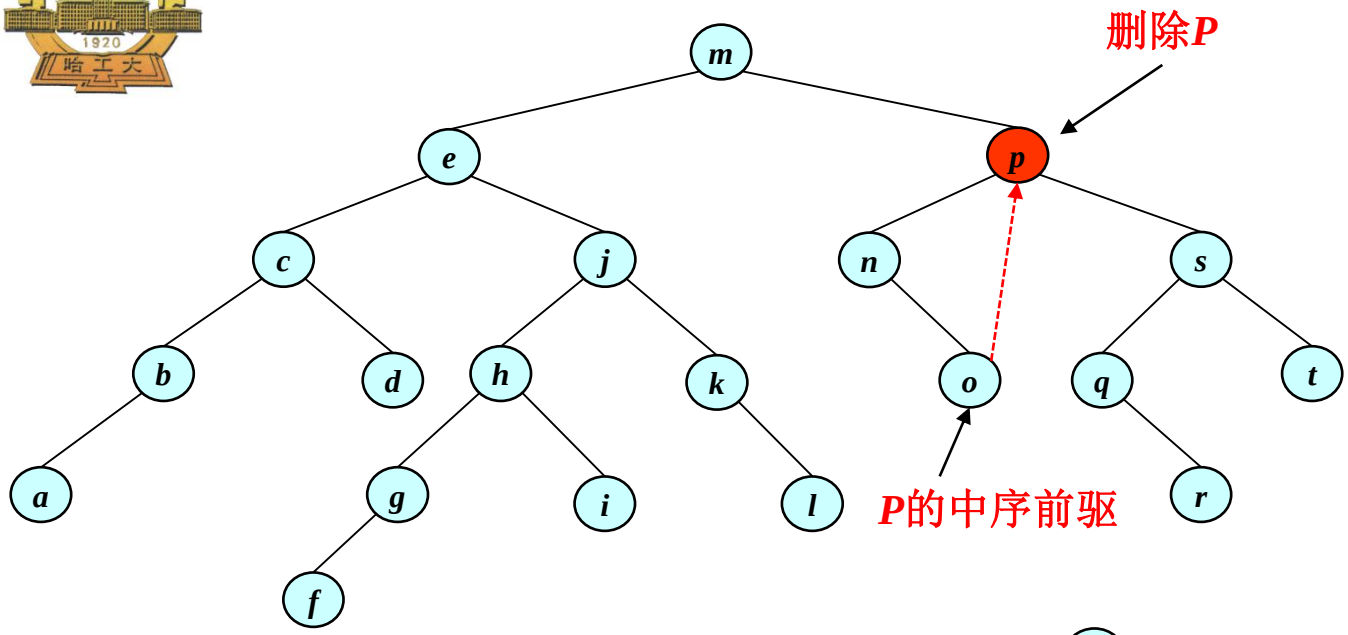
- AVL树的插入操作与建立
  - 对于一组关键字的输入序列，从空开始不断地插入结点，最后构成AVL树
  - 每插入一个结点后就应判断从该结点到根的路径上有无结点发生不平衡
  - 如有不平衡问题，利用旋转方法进行树的调整，使之平衡化
  - 建AVL树过程是不断插入结点和必要时进行平衡化的过程



【例】有一棵平衡二叉树的初始状态如图所示，

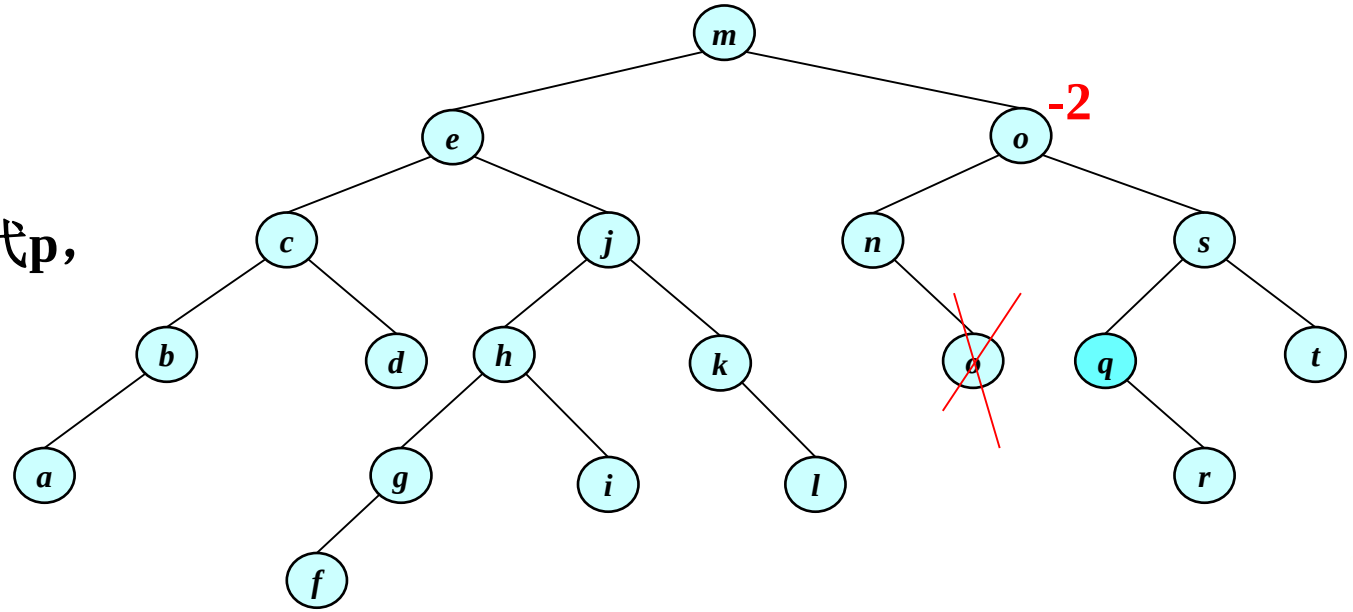
请给出删除图中结点p后经调整得到的新的平衡二叉树。

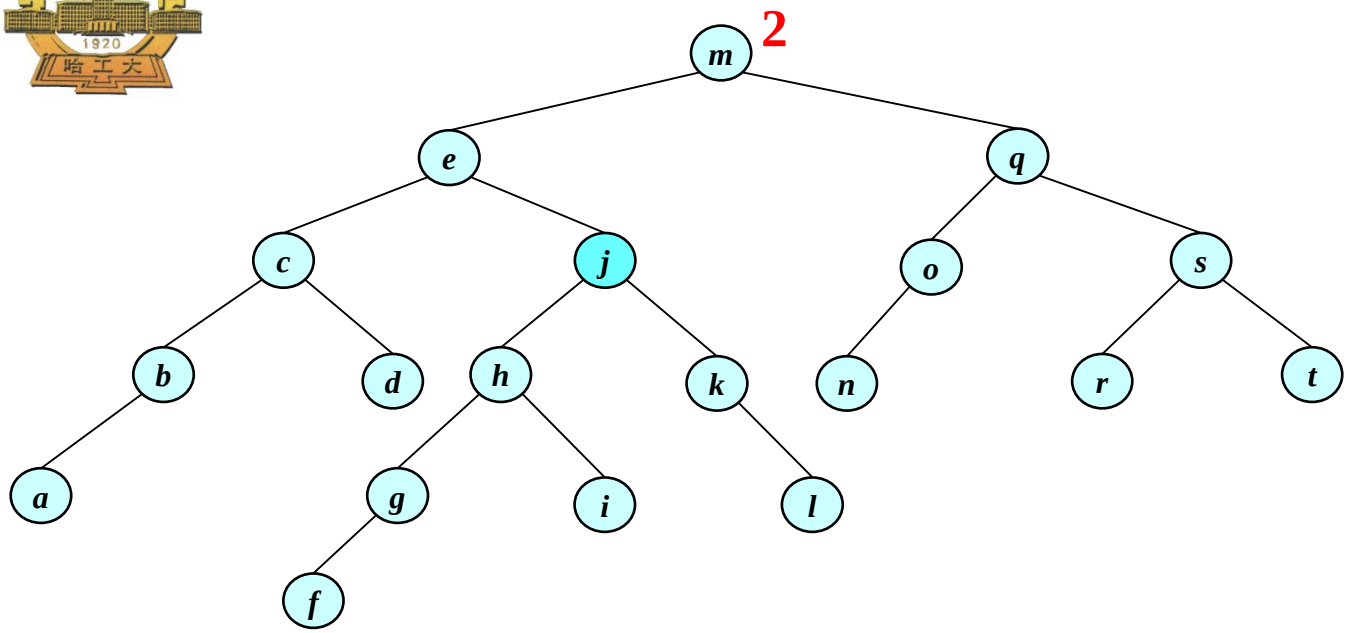




由于结点p既有左孩子，又有右孩子，故在删除结点p的时候应该先找到中序遍历中结点p的直接前驱结点，即结点o。

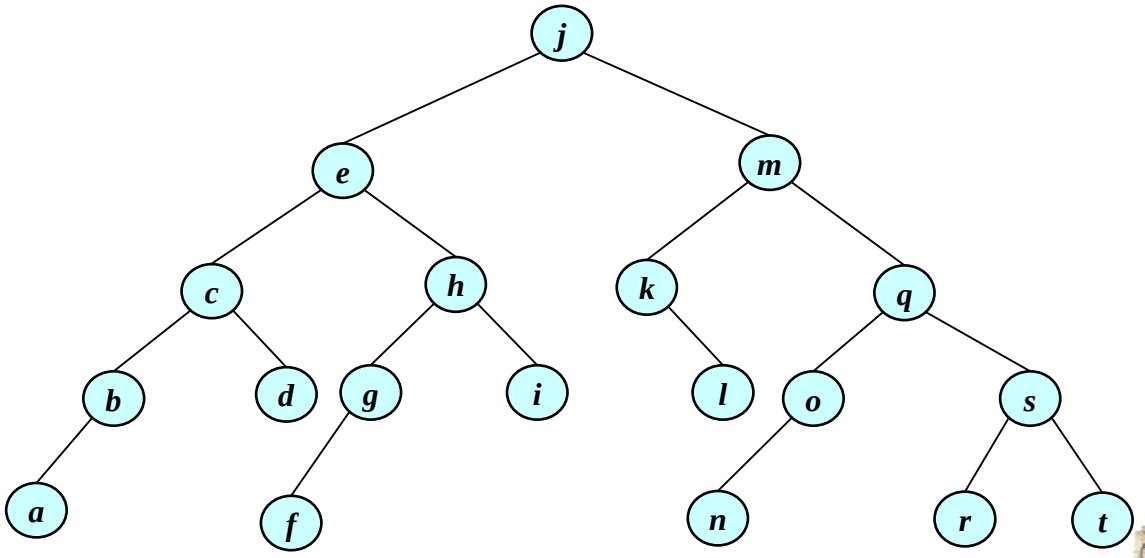
然后用o取代p，删除o，





此时结点o的平衡度变为2，发生不平衡，故应对子树o,r,t进行RL型调整。

结点m的平衡度变为-2，发生不平衡，故应对子树m,e,j进行LR调整。



8.6 AVL树

AVL树的删除操作

- 删除操作与插入操作是对称的（镜像），但可能需要的平衡化次数多。
- 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。