数据结构与算法

Data Structures and Algorithms

张岩

海量数据计算研究中心

哈工大计算机科学与技术学院

第3章 树与二叉树

学习目标

- 树型结构是一种非线性结构，反映了结点之间的层次关系，在计算机科学与软件工程中有着广泛的应用。
- 掌握树(森林)和二叉树的定义及其相关的术语；
- 重点掌握二叉树的结构、性质，存储表示和四种遍历算法；二叉树线索化的实质及线索化的过程；
- 了解树的结构性质、存储表示方法和遍历算法；
- 掌握森林(树)与二叉树的对应关系和相互转换方法；
- 掌握判定树、并查集、表达式求值等树型结构应用，重点掌握哈夫曼树的概念和构造方法，哈夫曼编码和译码的原理及实现方法。

本章主要内容

- 3.1 树与二叉树的基本术语
- 3.2 二叉树
- 3.3 堆
- 3.4 选择树
- 3.5 树
- 3.6 森林（树）与二叉树的相互转换
- 3.7 树的应用
- 本章小结

本章的知识点结构

- 基本的数据结构（ADT）
  - 二叉树、树（森林）
- 知识点结构

ADT 基本数据结构

- **ADT定义**
- 逻辑结构
  - 定义及相关术语
  - 逻辑结构及其特征
  - 基本操作（算法）→ 动态的操作

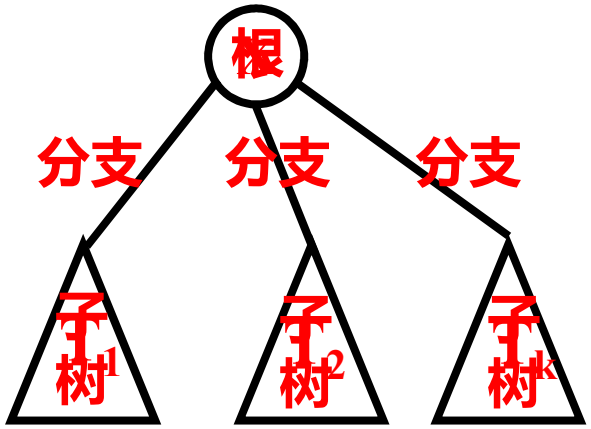
  （定义及相关术语、逻辑结构及其特征：静态的结构）
- **ADT实现** 存储结构
  - 存储结构（描述）
  - 存储结构特点
  - 存储结构的定义
  - 操作（算法）实现→动态的操作
  - 算法的性能

  （存储结构（描述）、存储结构特点、存储结构的定义：静态的结构）
- 应用

- 遍历算法是最核心的算法

3.1 树与二叉树的基本术语

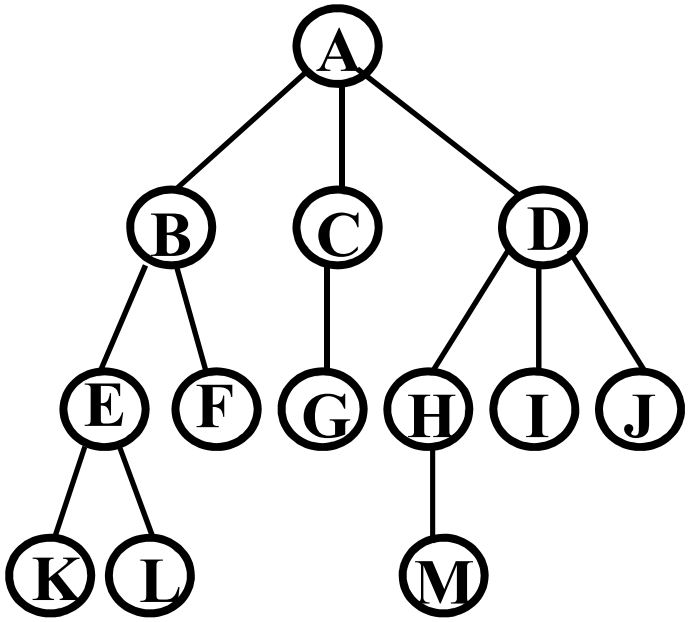
- 树的构造性递归定义：
  - 一个结点 X 组成的集合{ X }是一棵树，这个结点 X 称为这棵树的根（root）。
  - 假设 X 是一个结点，T_1，T_2，…，T_k是 k棵互不相交的树，可以构造一棵新树：令 X 为根，并有k 条边由 X 指向树T_1，T_2，…，T_k。这些边也叫做分支，T_1，T_2，…，T_k称作根为X的树之子树（SubTree）。

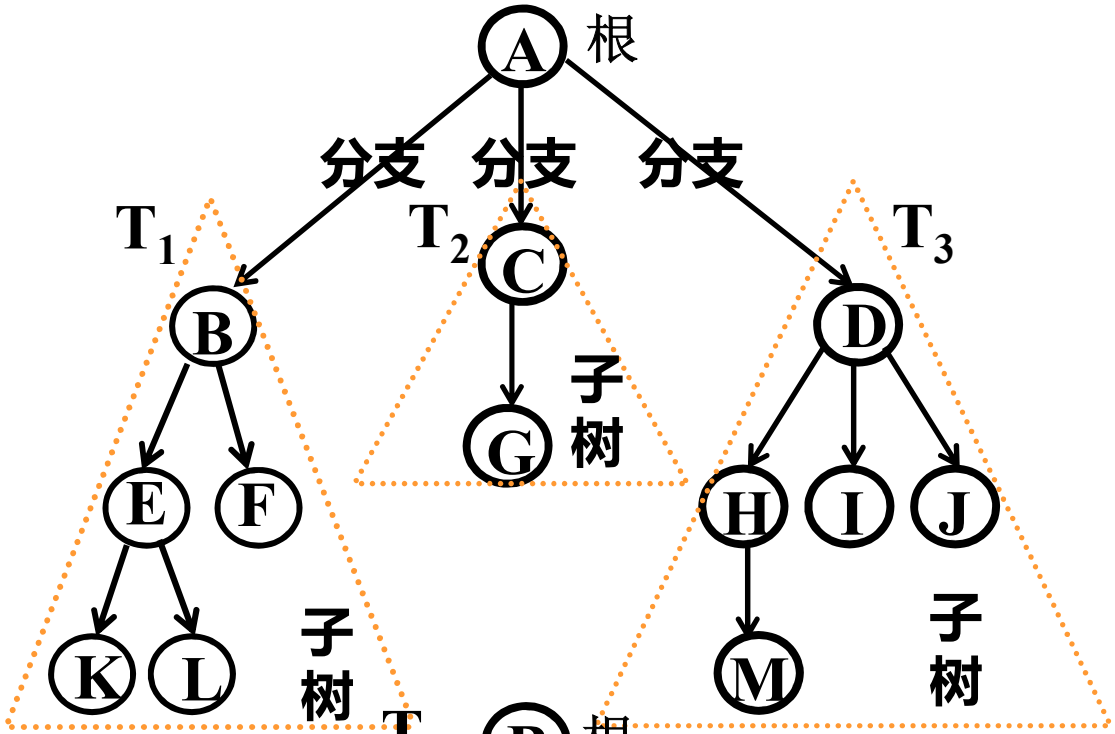


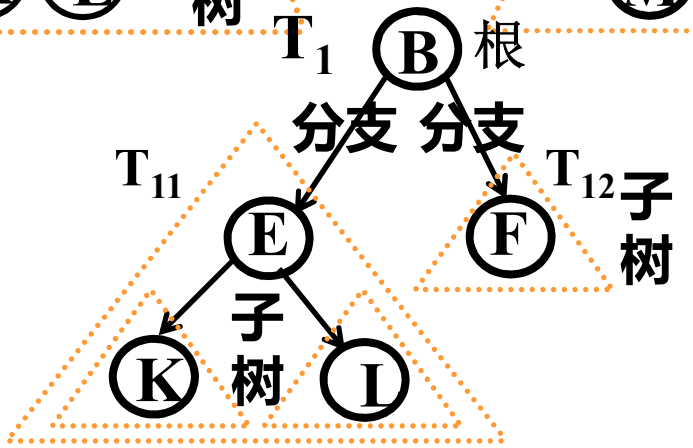
- **说明：**
  - **递归定义，但不会产生循环定义；**
  - **构造性定义便于树型结构的建立；**
  - **一株树的每个结点都是这株树的某株子树的根；**

3.1 树与二叉树的基本术语(Cont.)

- 树的构造性递归定义：

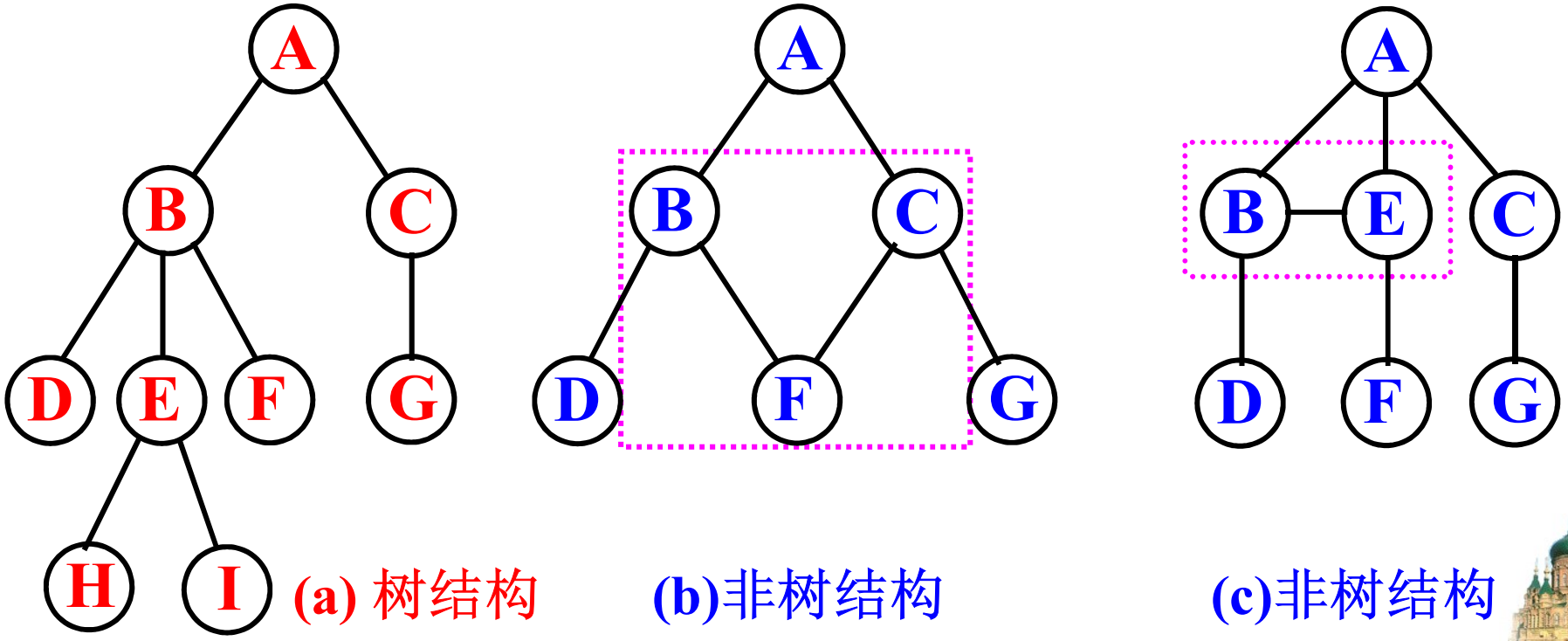






3.1 树与二叉树的基本术语(Cont.)

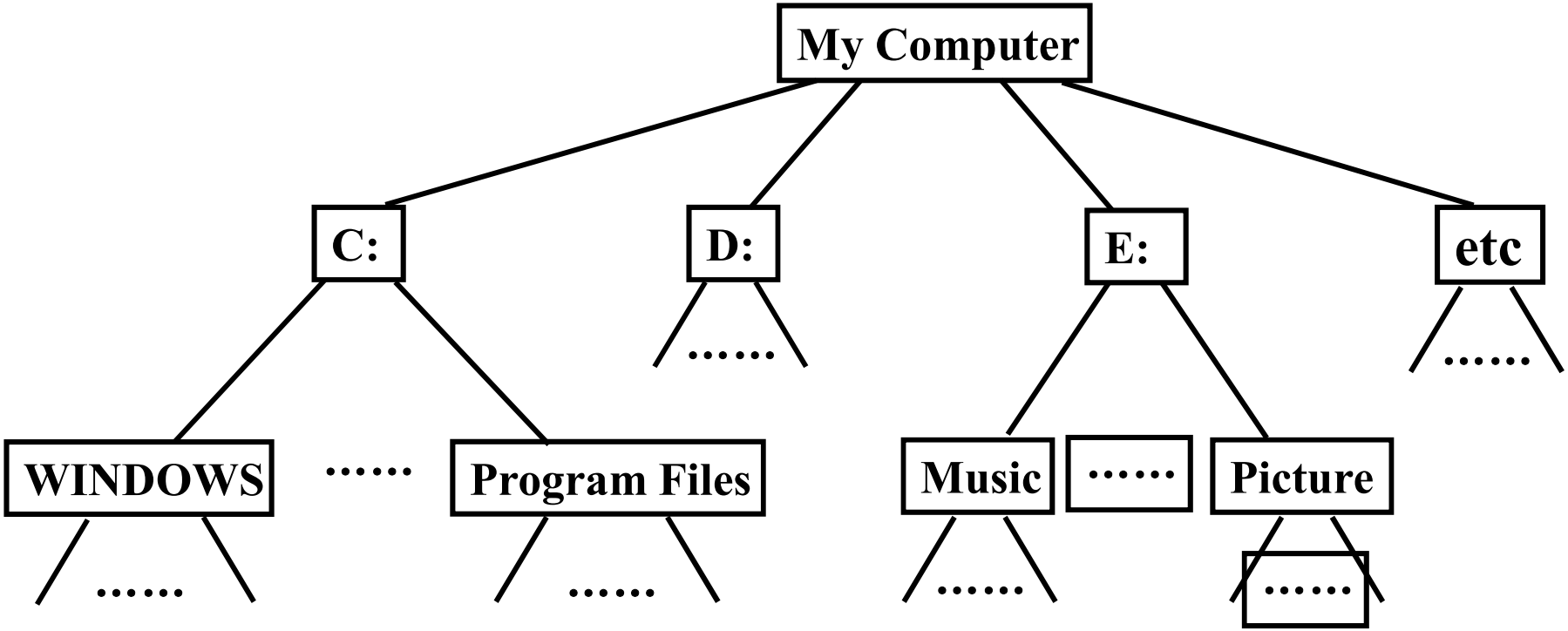
- 树的逻辑结构特点：
  - 除根结点之外，每棵子树的根结点有且仅有一个直接前驱，但可以有0个或多个直接后继。
  - 即一对多的关系，反映了结点之间的层次关系。



(a) 树结构　　(b)非树结构　　(c)非树结构

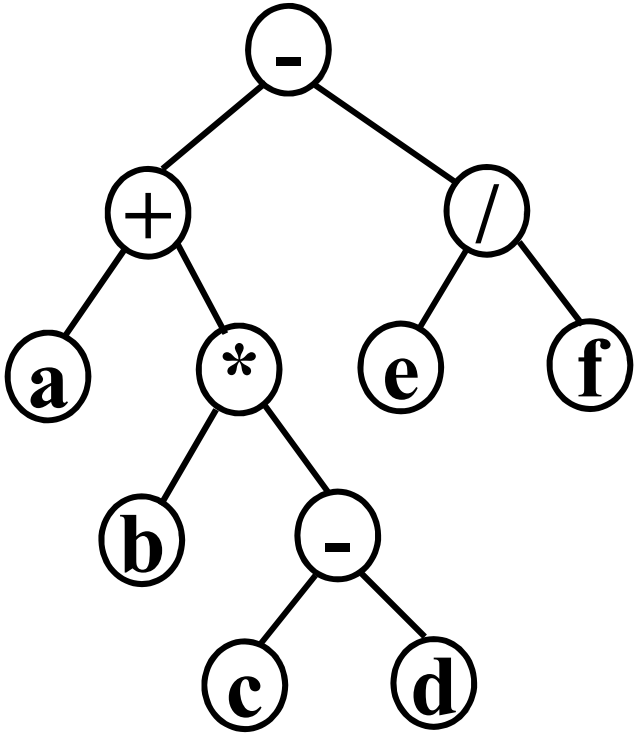
3.1 树与二叉树的基本术语(Cont.)

- 树型结构的应用示例----文件（目录）结构：



3.1 树与二叉树的基本术语(Cont.)

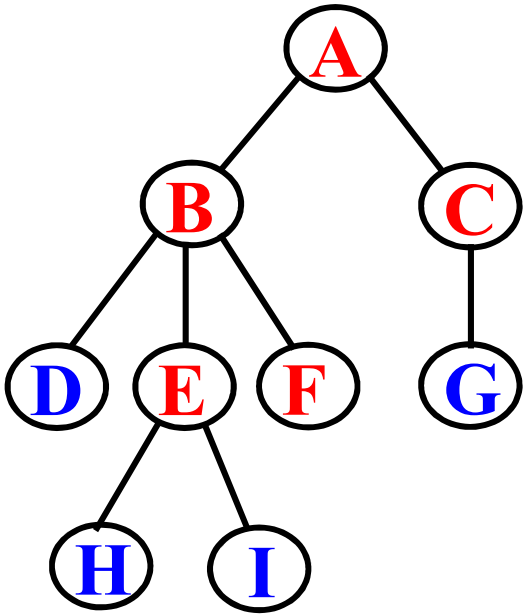
- 树型结构的应用示例----(无公共子式的)表达式的表示：



a+b*(c-d)-e/f

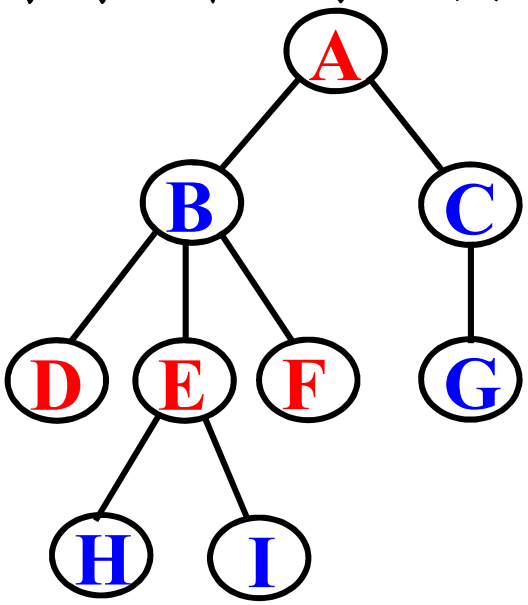
3.1 树与二叉树的基本术语(Cont.)

- 基本术语：
  - 结点的度：结点所具有的子树的个数。
  - 树的度：树中各结点度的最大值。
  - 叶子结点：度为0的结点，也称为终端结点。
  - 分支结点：度不为0的结点，也称为非终端结点。



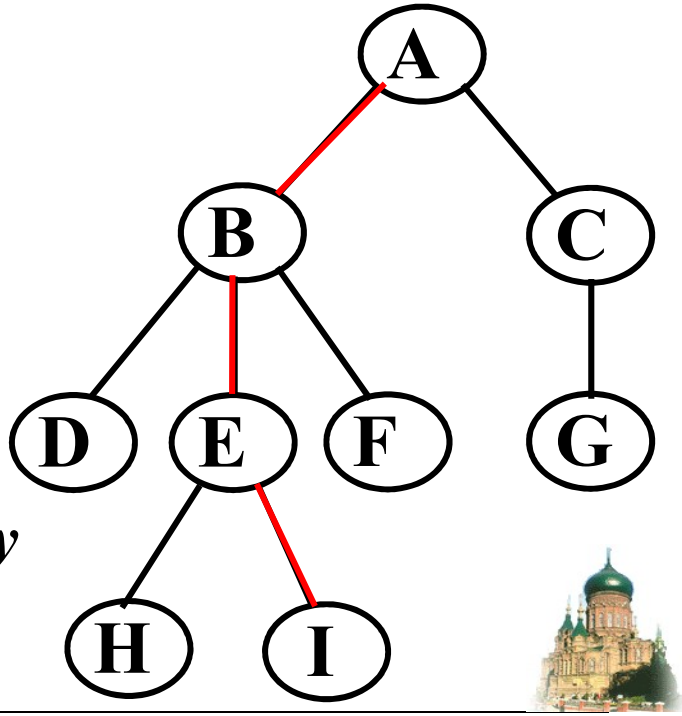
3.1 树与二叉树的基本术语(Cont.)

- 基本术语：
  - 孩子结点、双亲结点：树中某结点子树的根结点称为这个结点的孩子结点（子结点、儿子），这个结点称为它孩子结点的双亲结点（父结点）；
  - 兄弟：具有同一个双亲的孩子结点互称为兄弟。



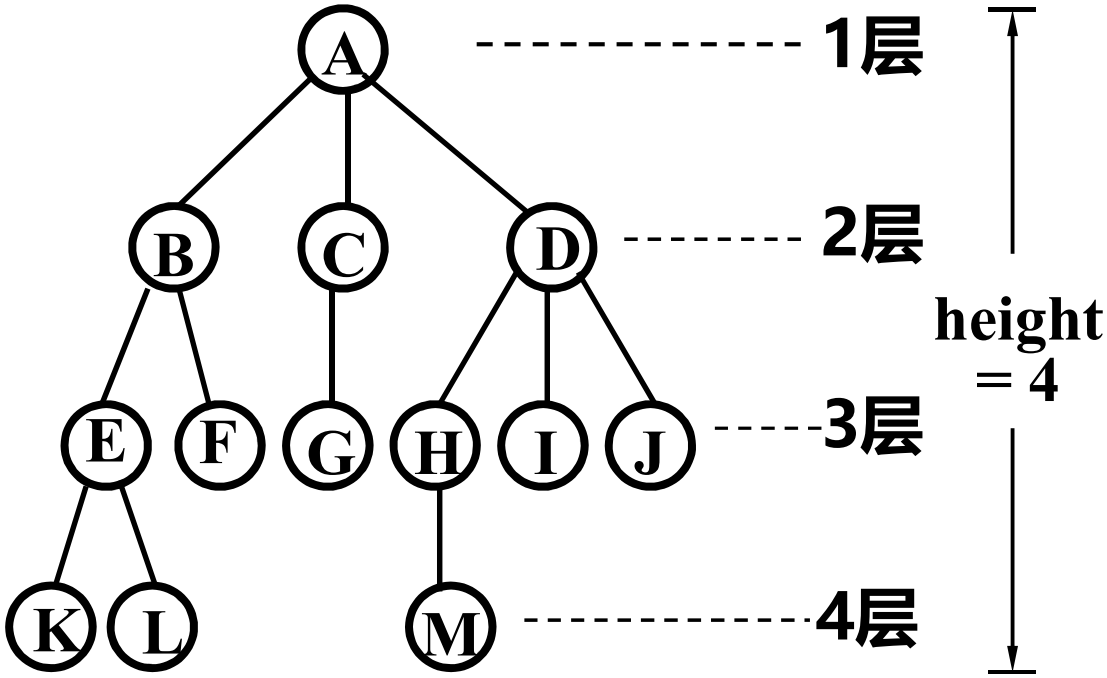
3.1 树与二叉树的基本术语(Cont.)

- 基本术语：
  - 路（径）和路（径）长度：如果树的结点序列$n_1, n_2, \ldots, n_k$有如下关系：结点n_i是n_{i+1}的双亲（$1<=i<k$），则把$n_1, n_2, \ldots, n_k$称为一条由n_1至n_k的路径；路径上经过的边的个数称为路径长度。
  - 祖先、子孙：在树中，如果有一条路径从结点x到结点y，那么x就称为y的祖先，而y称为x的子孙。



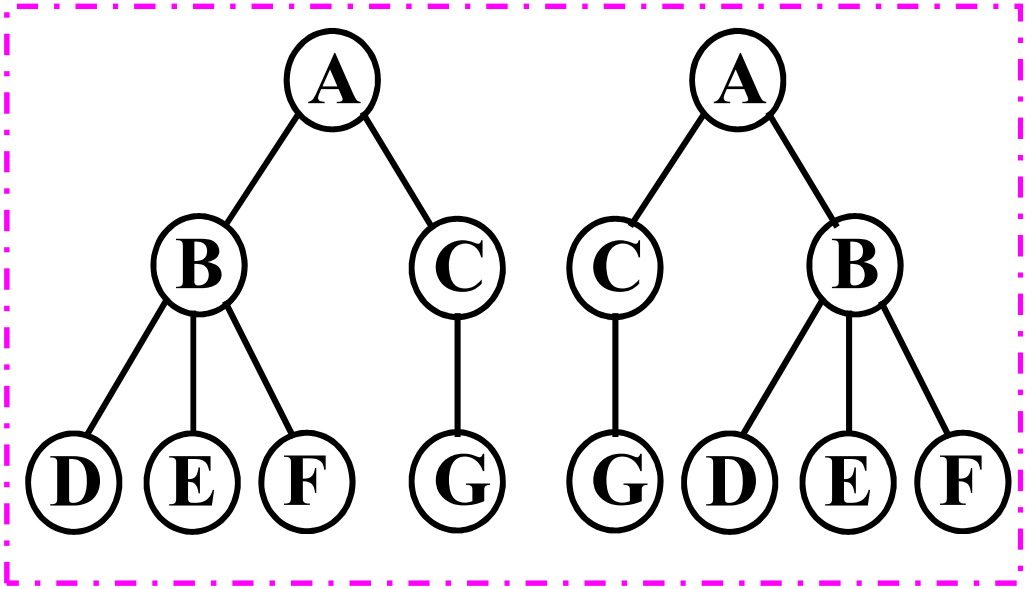
3.1 树与二叉树的基本术语(Cont.)

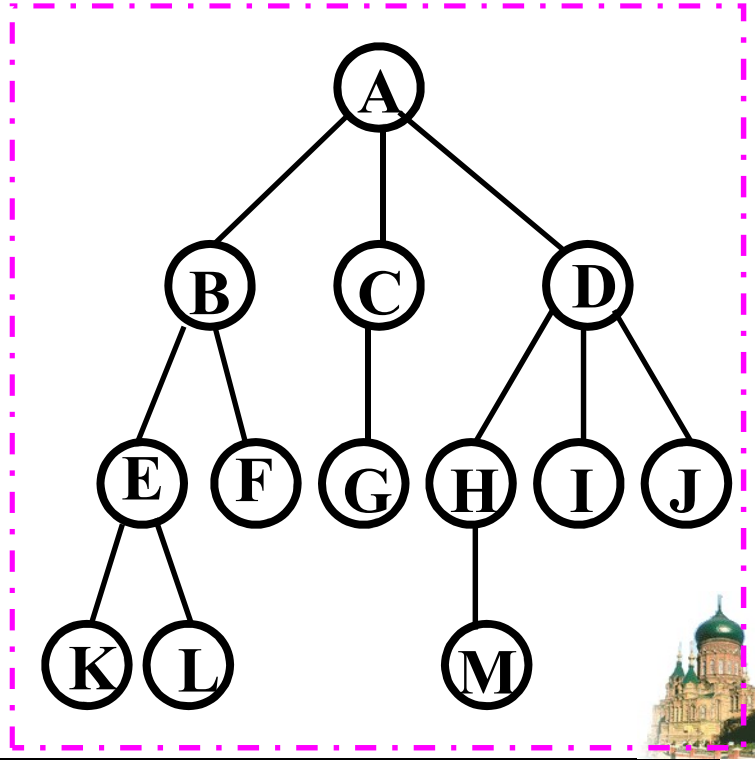
- 基本术语：
  - 结点的层数：根结点的层数为1；对其余任何结点，若某结点在第k层，则其孩子结点在第k+1层。
  - 树的高度：树中所有结点的最大层数，也称深度。



3.1 树与二叉树的基本术语(Cont.)

- 基本术语：
  - 有序树、无序树：如果一棵树中结点的各子树从左到右是有次序的，称这棵树为有序树；反之，称为无序树。
  - 森林：m ($m \geq 0$)棵互不相交的树的集合。





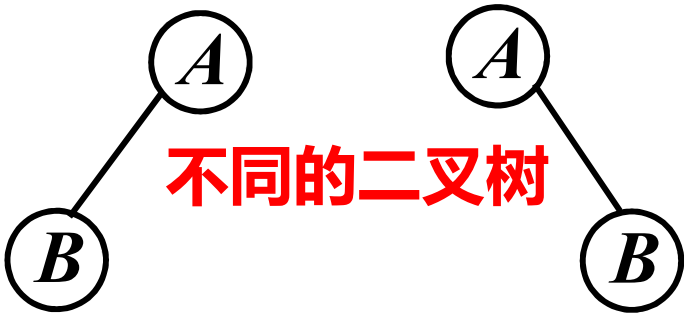
3.1 树与二叉树的基本术语(Cont.)

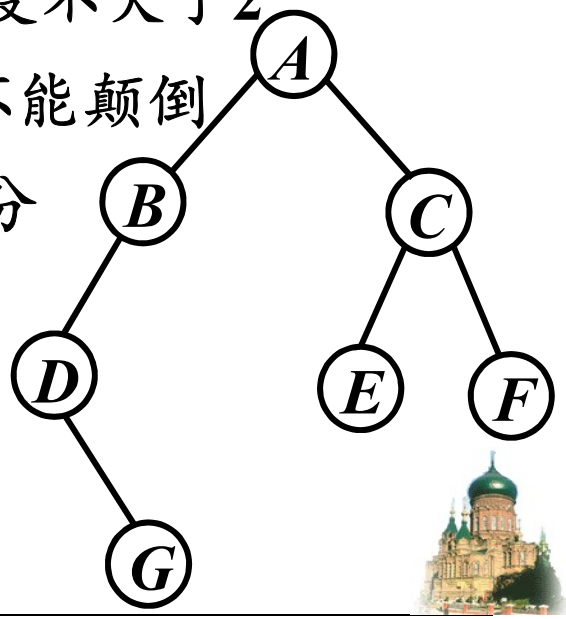
- 树型结构和线性结构的比较

线性结构	树型结构
第一个数据元素	根结点（只有一个）
无前驱	无双亲
最后一个数据元素	叶子结点(可以有多个)
无后继	无孩子
其它数据元素	其它结点
一个前驱,一个后继	一个双亲,多个孩子
一对一	一对多

3.2 二叉树

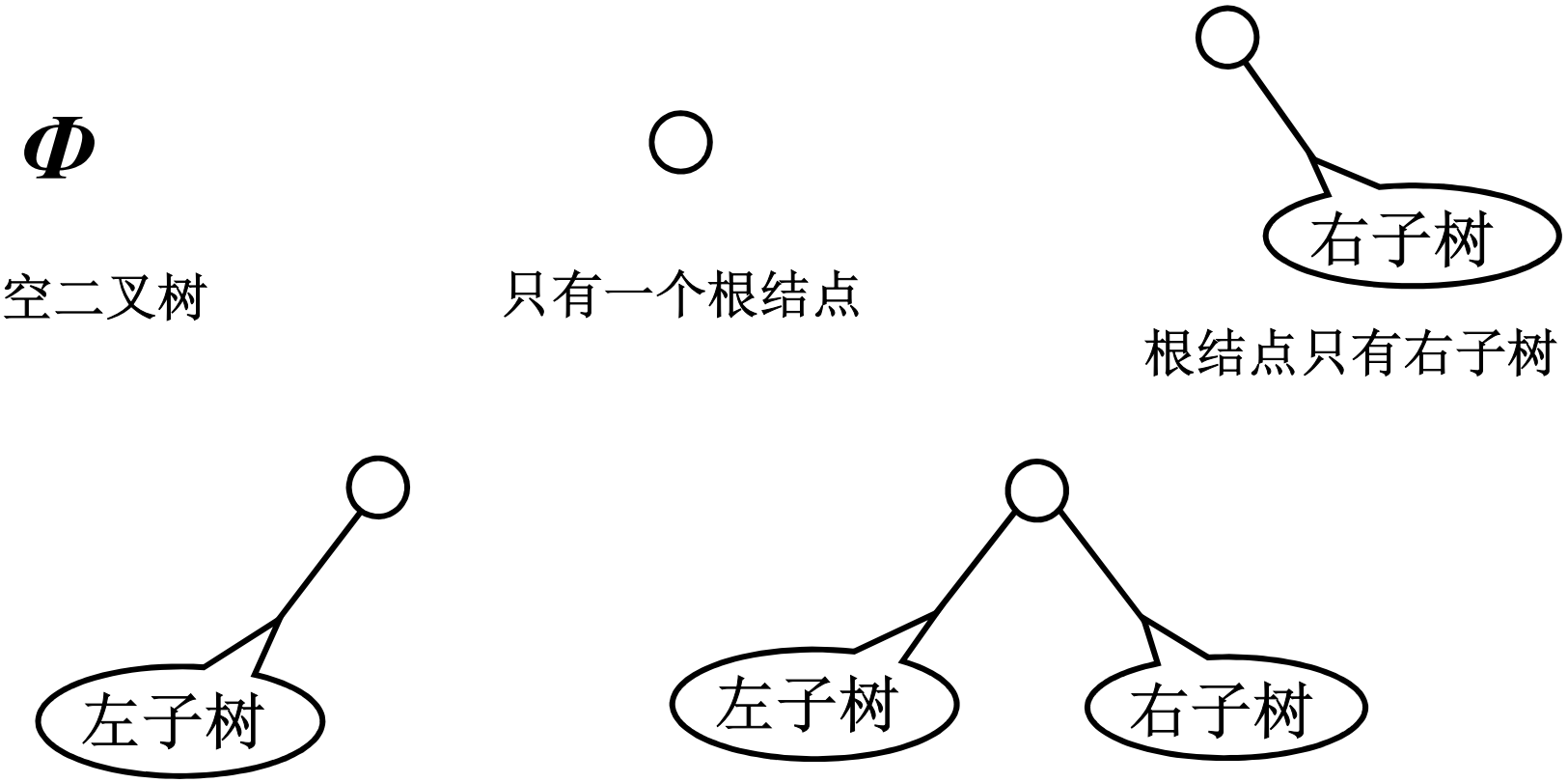
- 二叉树(Binary Tree)的定义：
  - 二叉树一个是n（n≥0）个结点的有限集合，该集合或者为空（称为空二叉树）；或者是由一个根结点和两棵互不相交的、分别称为左子树和右子树的二叉树组成。
- 结构特点：
  - 每个结点最多只有两棵子树，即结点的度不大于2
  - 且子树有左右之别，子树的次序(位置)不能颠倒
  - 即使某结点只有一棵子树，也有左右之分





3.2 二叉树(Cont.)

- 二叉树的基本形态：

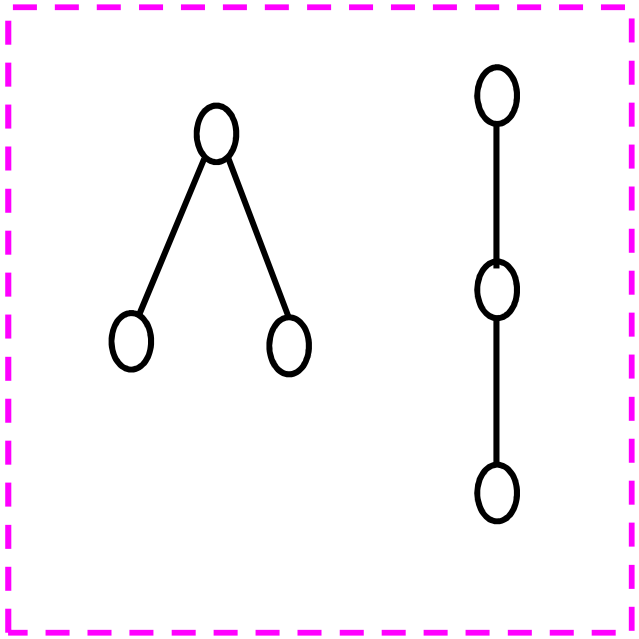


空二叉树　　只有一个根结点　　根结点只有右子树

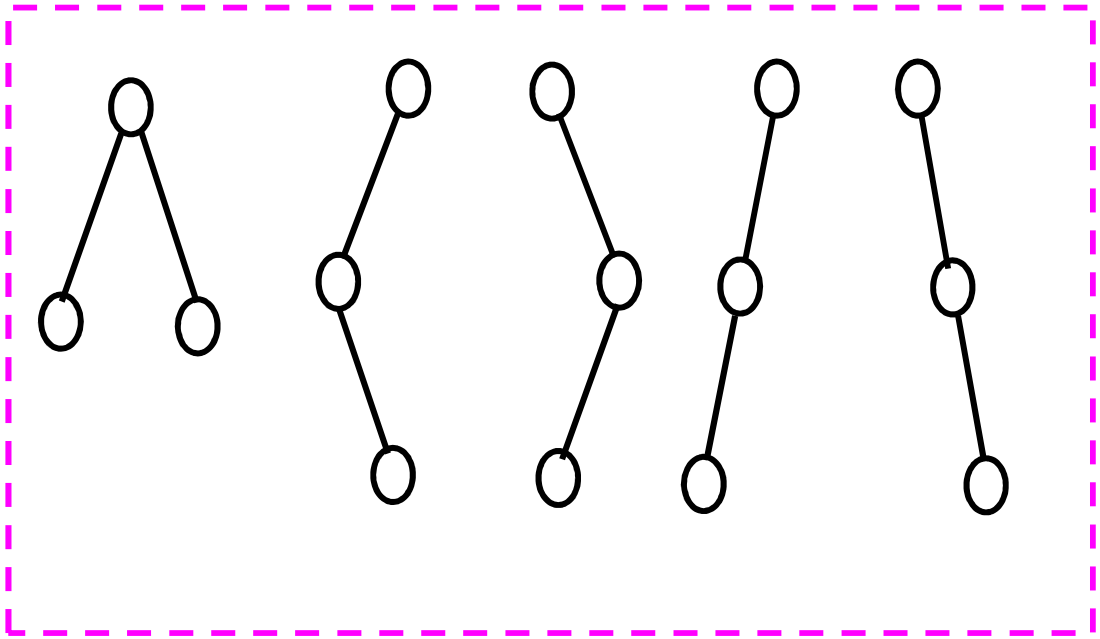
根结点只有左子树　　根结点同时有左右子树

3.2 二叉树(Cont.)

- 具有3个结点的树和二叉树的不同构的形态：



树的不同构形态

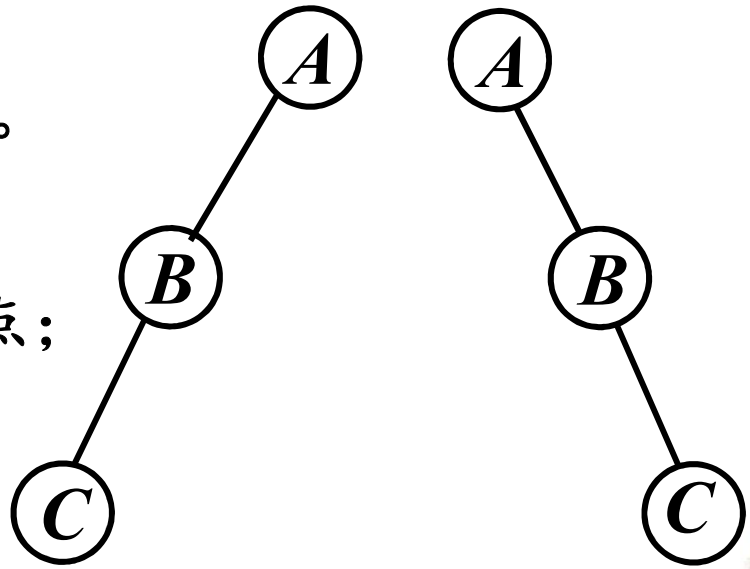


二叉树的不同构形态

3.2 二叉树(Cont.)

特殊的二叉树----斜树

- 左斜树
  - 所有结点都只有左子树的二叉树称为左斜树；
- 右斜树
  - 所有结点都只有右子树的二叉树称为右斜树；
- 斜树：
  - 左斜树和右斜树统称为斜树。



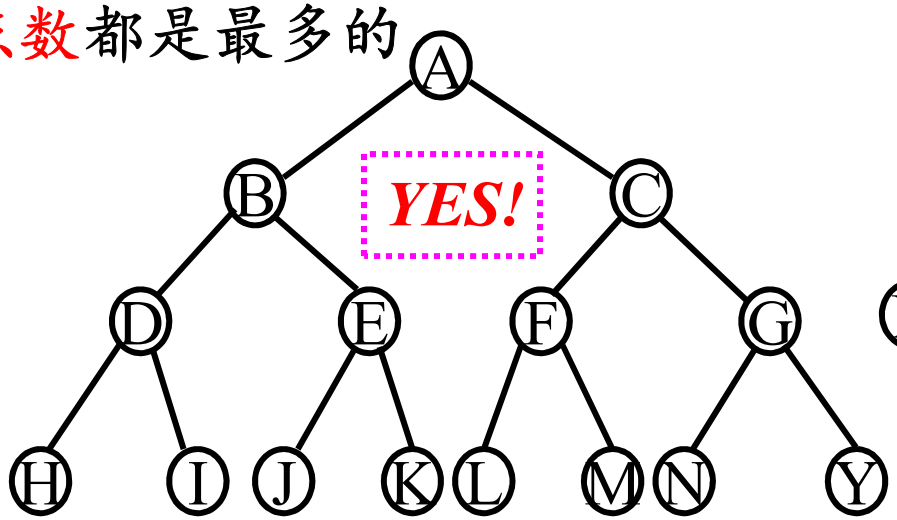
斜树的结构特点：

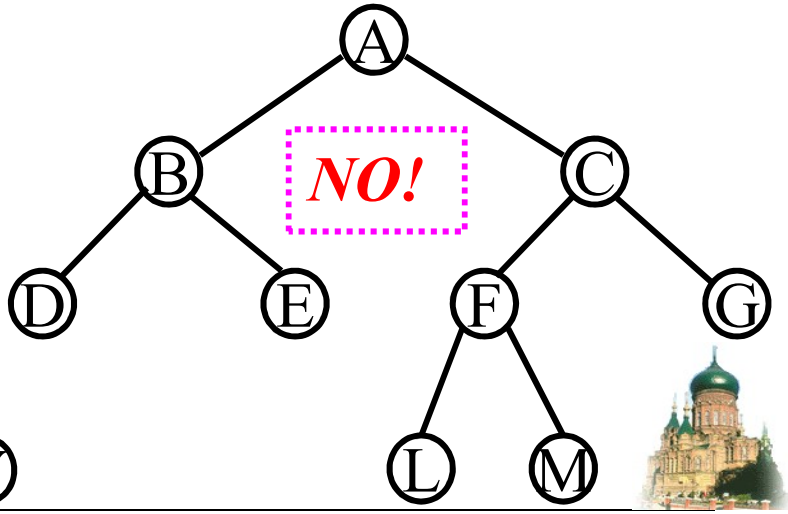
- 在斜树中，每一层只有一个结点；
- 斜树的结点个数与其高度相同。

3.2 二叉树(Cont.)

特殊的二叉树----满二叉树

- 定义：高度为K且有2^K-1个结点的二叉树称为满二叉树。
- 结构特点：
  - 分支结点都有两棵子树
  - 叶子结点都在最后一层
- 满二叉树在相同高度的二叉树中，结点数、分支结点数和叶结点数都是最多的

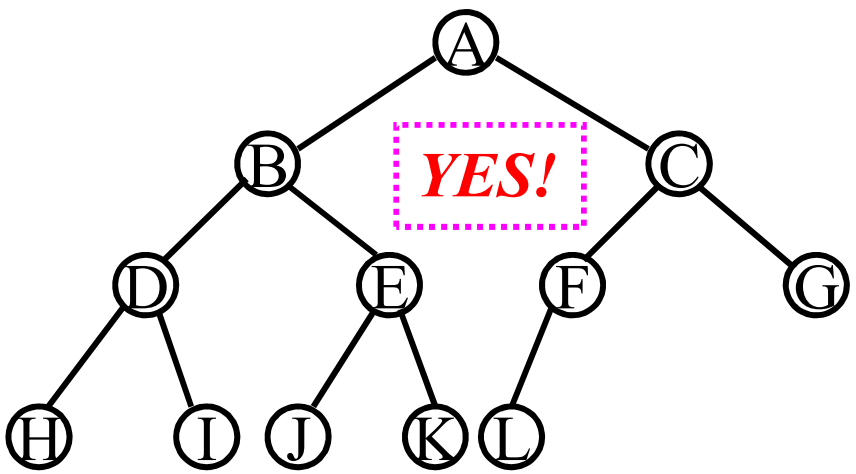


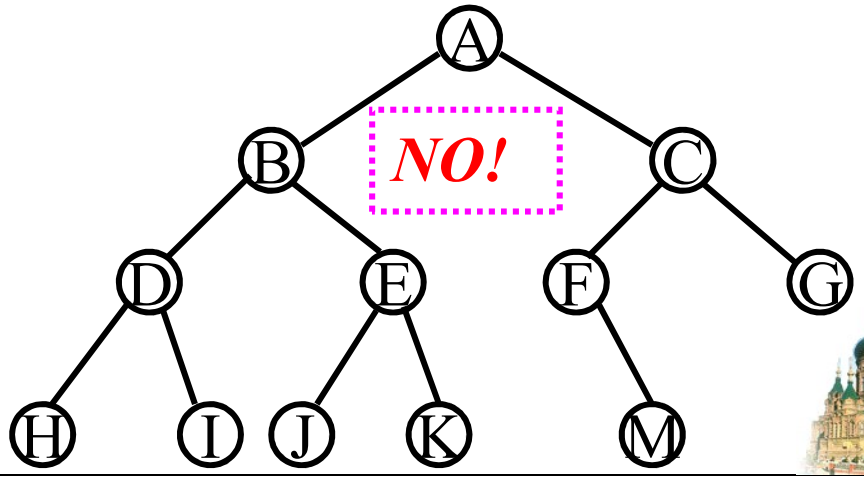


3.2 二叉树(Cont.)

特殊的二叉树----完全二叉树

- 定义：称满足下列性质的二叉树(假设高度为h)为完全二叉树:
  - 1.所有的叶都出现在h或h-1层；
  - 2.h-1层的所有叶都在非终端结点的右边；
  - 3. 除了h-1层的最右非终端结点可能有一个（只能是左分支）或两个分支之外，其余非终端结点都有两个分支。

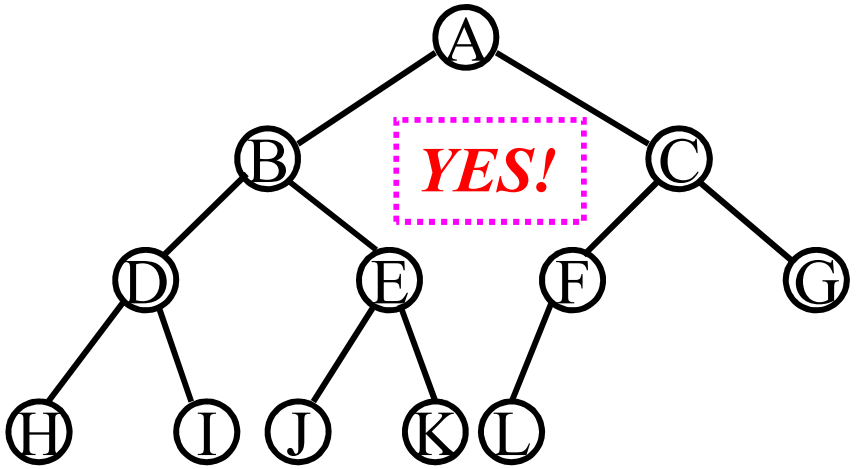


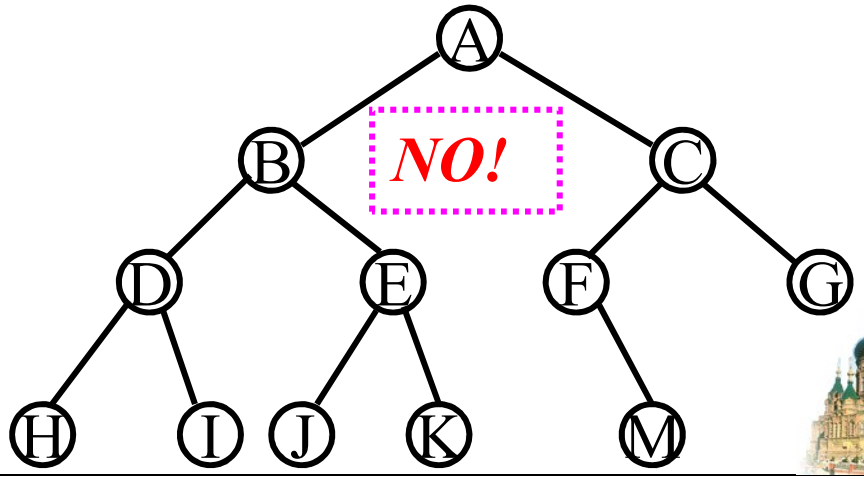


3.2 二叉树(Cont.)

特殊的二叉树----完全二叉树

- 结构特点：
  - 1. 叶子结点只能出现在最下两层，且最下层的叶子结点都集中在二叉树的左部；
  - 2. 完全二叉树中如果有度为1的结点，只可能有一个，且该结点只有左孩子。
  - 3. 深度为h的完全二叉树的前h-1层一定是满二叉树。





3.2 二叉树(Cont.)

二叉树的性质

- 性质1:
  - 二叉树的第 i 层最多有 2^{i-1} 个结点。($i \geq 1$)
  - 证明(数学归纳法):

当i=1时，第1层只有一个根结点，而

$$2^{i-1}=2^0=1,$$

结论成立。

假定i=k（$1 \leq k < i$）时结论成立，即第k层上至多有2^{k-1}个结点，则 $i=k+1$时，因为第k+1层上的结点是第k层上结点的孩子，而二叉树中每个结点最多有2个孩子，故在第k+1层上最大结点个数为第k层上的最大结点个数的2倍，即$2 \times 2^{k-1}=2^k$。结论成立。□

3.2 二叉树(Cont.)

二叉树的性质

- 性质2:
  - 高度为 $h(h \geq 1)$的二叉树最多有 2^h-1个结点，最少有h个结点。
  - 证明：由性质1可知，高度为h的二叉树中结点个数最多

$$=\sum_{i=1}^{h}(\text{第}i\text{层上最多结点数}) = 2^0+2^1+2^2+\ldots+2^{h-1}=2^h-1;$$

另外，每一层至少要有一个结点，因此，高度为h的二叉树，至少有h个结点。□

- 高度为h且具有2^h-1个结点的二叉树一定是满二叉树，
- 高度为h且具有h个结点的二叉树不一定是斜树。

3.2 二叉树(Cont.)

二叉树的性质

- 性质3:
  - 在非空二叉树中，如果叶子结点数为n_0，度为2的结点数为n_2，则有：$n_0=n_2+1$，而与度数为1的结点数n_1无关。
  - 证明：设n为二叉树的结点总数，则有：

$$n=n_0+n_1+n_2$$

在n个结点的二叉树中，共有$n-1$条分支(边)；在这些分支中，度为1和度为2的结点分别提供1条和2条分支。所以有：

$$n-1=n_1+2n_2$$

因此可以得到：$n_0=n_2+1$。与度数1的结点数n_1无关。□

- 在有n个结点的满二叉树中，有多少个叶子结点？

3.2 二叉树(Cont.)

完全（满）二叉树的性质

- **性质4:** 具有 n $(n \geq 0)$ 个结点的完全二叉树的高度为

$$\lceil \log_2(n+1) \rceil \text{ 或 } \lfloor \log_2 n \rfloor + 1$$

证明：设完全二叉树的高度为 h，则根据性质2有：

$$\underbrace{2^{h-1}-1}_{\text{前h-1层最多结点数}} < n \leq \underbrace{2^h-1}_{\text{前h层最多结点数}}$$

变形 $2^{h-1} < n+1 \leq 2^h$

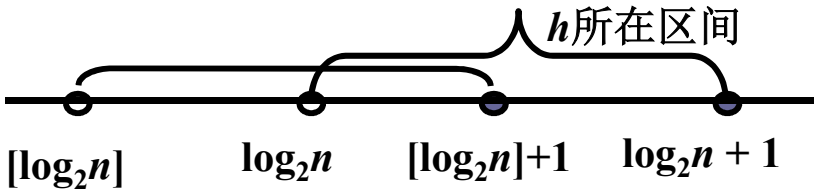
取对数 $h-1 < \log_2(n+1) \leq h$

因此有 $\lceil \log_2(n+1) \rceil = h$ □

$$\underbrace{2^{h-1}}_{\text{最少结点数}} \leq n < \underbrace{2^h}_{\text{最多结点数}}$$

$h-1 \leq \log_2 n < h$

$h = \lfloor \log_2 n \rfloor + 1$ □

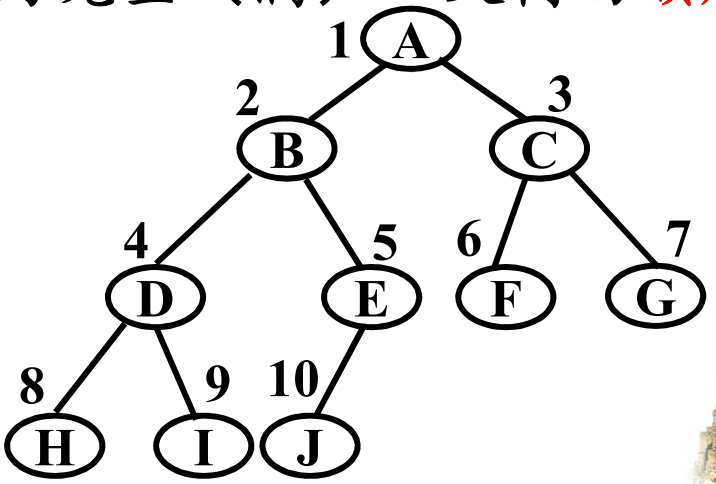


3.2 二叉树(Cont.)

完全（满）二叉树的性质

- 完全二叉树的顺序存储结构：
  - 如果把一棵完全二叉树的具有n个结点自顶向下，同一层自左向右连续编号:1, 2,…,n，且使该编号对应于数组的下标，即编号为i（$1 \le i \le n$）的结点存储在下标为i的数组单元中，则这种存储表示方法称为完全（满）二叉树的顺序存储结构。

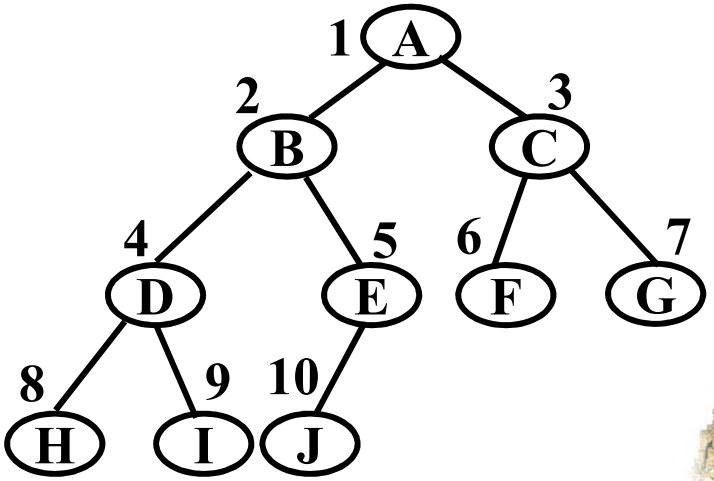
	A	B	C	D	E	F	G	H	I	J
0	1	2	3	4	5	6	7	8	9	10



3.2 二叉树(Cont.)

- 完全二叉树的顺序存储结构的性质：
  - 若i = 1, 则 i 是根结点，无父结点；
    若i > 1, 则 i 的父结点为⌊i/2⌋
  - 若 2*i ≤ n, 则 i 有左儿子且为 2*i；否则，i 无左儿子。
  - 若2*i+1 ≤ n, 则 i 有右儿子且为2*i+1；否则，i 无右儿子
  - 若 i 为偶数, 且 i < n , 则有右兄弟，且为 i + 1。
  - 若 i 为奇数, 且 i ≤ n && i != 1, 则其左兄弟，且为 i-1

	A	B	C	D	E	F	G	H	I	J
0	1	2	3	4	5	6	7	8	9	10



3.2 二叉树(Cont.)

二叉树的遍历操作

- 遍历的定义：根据某种策略，按照一定的次序访问二叉树中的每一个结点，使每个结点被访问一次且只被访问一次。这个过程称为二叉树的遍历。
- 遍历的结果是二叉树结点的线性序列。非线性结构线性化。
  - 策略：左孩子结点一定要在右孩子结点之前访问
  - 次序：先序（根）遍历、中序（根）遍历、后序（根）遍历和层序（次）遍历

二叉树：根结点D、左儿子L、右儿子R → **三个不同“部件”的可能顺序：**

DLR　LDR　LRD

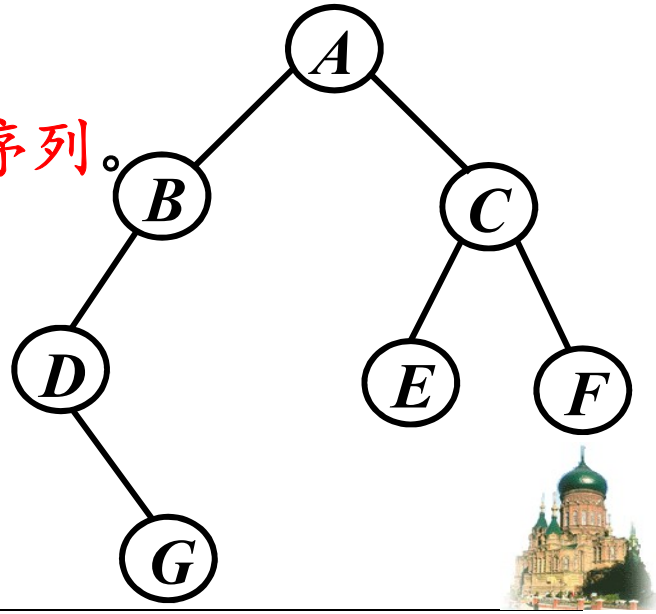
~~**DRL**~~　~~**RDL**~~　~~**RLD**~~

  - 访问：抽象操作，可以是对结点进行的各种处理，这里简化为输出结点的数据。

3.2 二叉树(Cont.)

二叉树遍历的定义

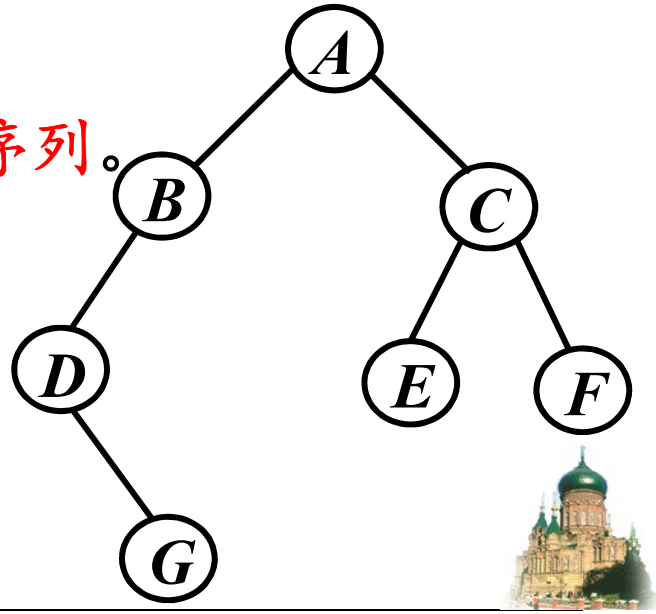
- 先序（根）遍历二叉树
  - 若二叉树为空，则返回；否则，
    - ①访问根结点；
    - ②先序遍历根结点的左子树;
    - ③先序遍历根结点的右子树;
- 所得到的线性序列分别称为先序（根）序列。
- 先序遍历序列为：*A B D G C E F*



3.2 二叉树(Cont.)

二叉树遍历的定义

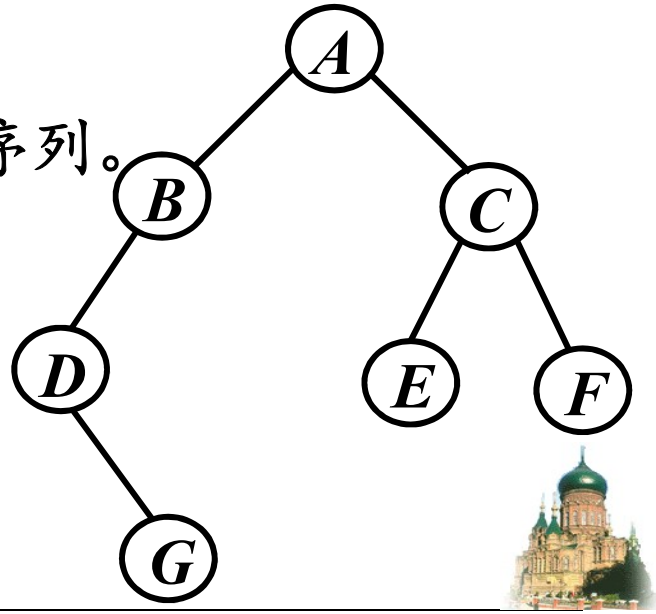
- 中序（根）遍历二叉树
  - 若二叉树为空，则返回；否则，
    - ①中序遍历根结点的左子树;
    - ②访问根结点；
    - ③中序遍历根结点的右子树;
- 所得到的线性序列分别称为中序（根）序列。
- 中序遍历序列为：*D G B A E C F*



3.2 二叉树(Cont.)

二叉树遍历的定义

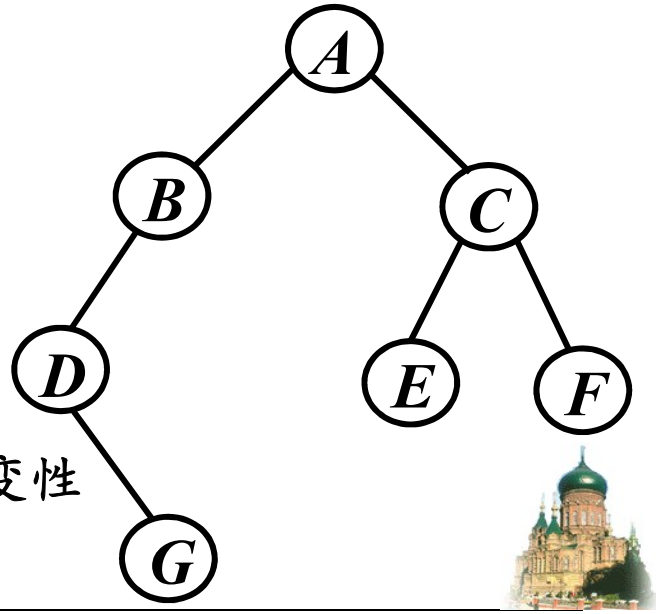
- 后序（根）遍历二叉树
  - 若二叉树为空，则返回；否则，
    - ①后序遍历根结点的左子树;
    - ②后序遍历根结点的右子树;
    - ③访问根结点;
- 所得到的线性序列分别称为后序（根）序列。
- 后序遍历序列为： *G D B E F C A*



3.2 二叉树(Cont.)

二叉树遍历的定义

- 层序（次）遍历二叉树
  - 从二叉树的第一层（即根结点）开始，从上至下逐层遍历，在同一层中，则按从左到右的顺序对结点进行访问。
  - 所得到的线性序列分别称为层序序列。
  - 层序遍历序列为：*A B C D E F G*



- 先序遍历序列为：*A B D G C E F*
- 中序遍历序列为：*D G B A E C F*
- 后序遍历序列为：*G D B E F C A*
  - 三种遍历顺序的叶子结点的相对顺序具有不变性
  - 二叉树的恢复算法

3.2 二叉树(Cont.)

二叉树的基本操作

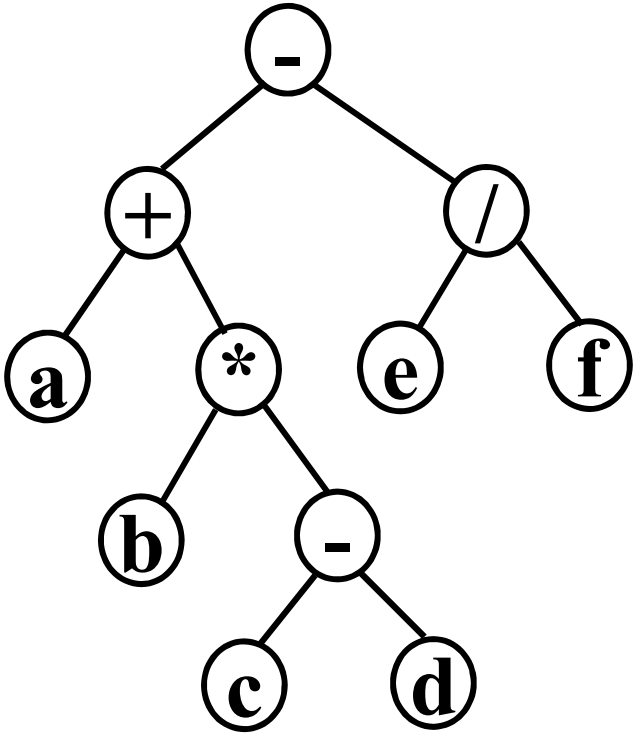
- ① Empty (BT) ： 建立一株空的二叉树。
- ② IsEmpty (BT) ： 判断二叉树是否为空,若是空则返回TRUE;否则返回FALSE。
- ③ CreateBT (V, LT , RT) ： 建立一株新的二叉树。这棵新二叉树根结点的数据域为V,其作右子树分别为LT，RT。
- ④ Lchild (BT): 返回二叉树BT的左儿子。若无左儿子，则返回空。
- ⑤ Rchild (BT) ： 返回二叉树BT的右儿子。若无右儿子，则返回空。
- ⑥ Data (BT) ： 返回二叉树BT的根结点的数据域的值。

3.2 二叉树(Cont.)

利用二叉树的基本操作，写出前三种遍历算法的递归形式

- 先序遍历算法

```
void  PreOrder (BTREE BT )
{
    if ( ! IsEmpty ( BT ) )
    {   visit ( Data ( BT ) ) ;
        PreOrder ( Lchild ( BT ) ) ;
        PreOrder ( Rchild ( BT ) ) ;
    }
}
```



a+b*(c-d)-e/f

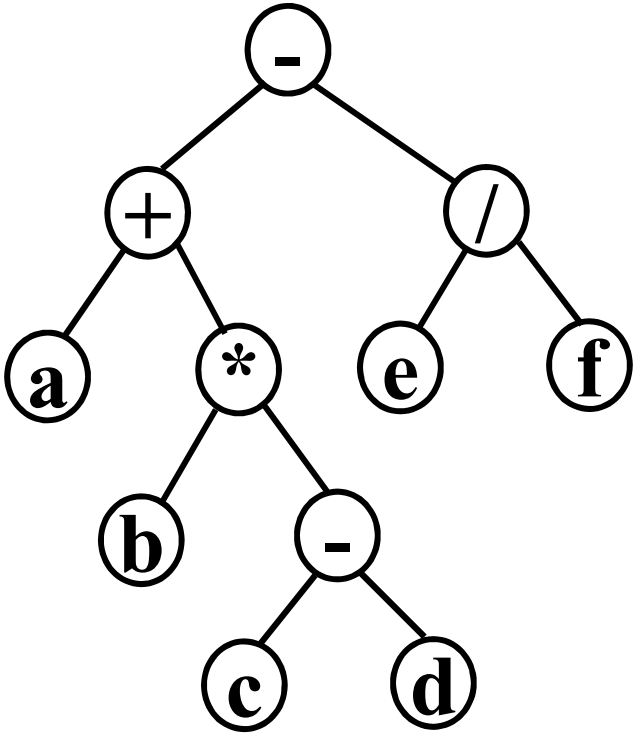
- 先序序列：- + a * b – c d / e f

3.2 二叉树(Cont.)

利用二叉树的基本操作，写出前三种遍历算法的递归形式

- 中序遍历算法

```
void  InOrder (BTREE BT )
{
    if ( ! IsEmpty ( BT ) )
    {   InOrder ( Lchild ( BT ) ) ;
        visit ( Data ( BT ) ) ;
        InOrder ( Rchild ( BT ) ) ;
    }
}
```



a+b*(c-d)-e/f

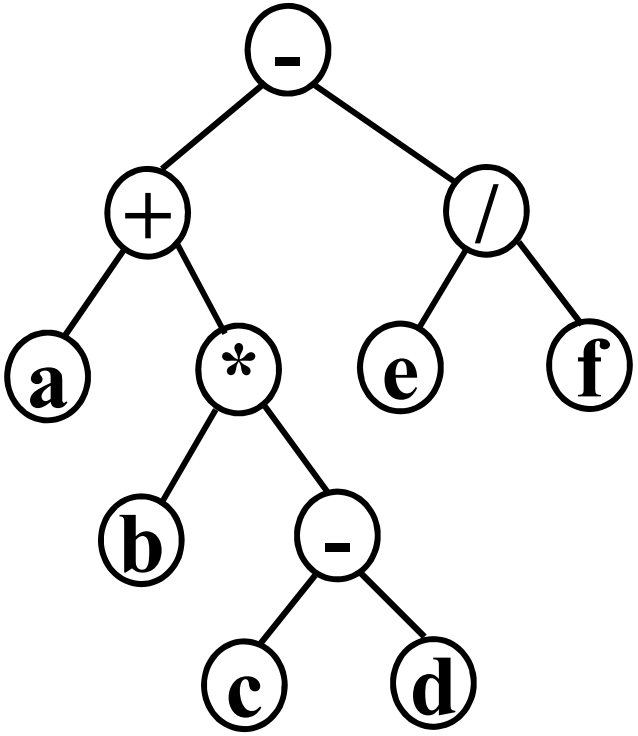
- 中序序列： a + b * c – d – e / f

3.2 二叉树(Cont.)

利用二叉树的基本操作，写出前三种遍历算法的递归形式

- 后序遍历算法

```
void  PostOrder (BTREE BT )
{
    if ( ! IsEmpty ( BT ) )
   {     PostreOrder ( Lchild ( BT ) ) ;
         PostOrder ( Rchild ( BT ) ) ;
         visit ( Data ( BT ) ) ;
    }
}
```



a+b*(c-d)-e/f

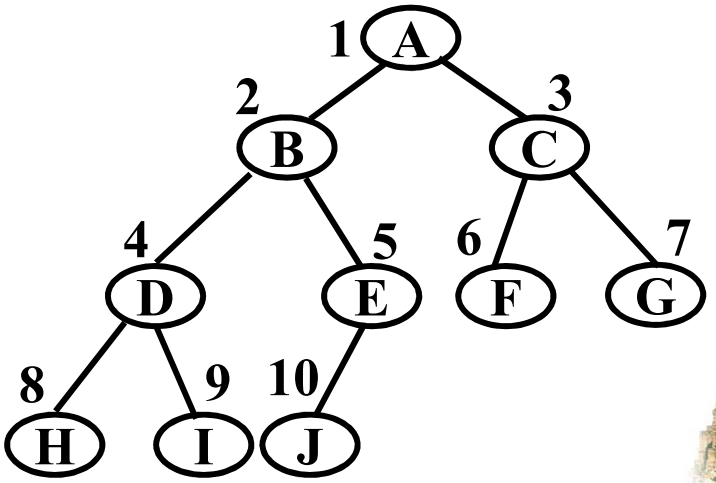
- 后序序列：a b c d - * + e f / -

3.2 二叉树(Cont.)

二叉树的存储结构

- 二叉树的顺序存储结构
  - 完全（满）二叉树---参见“完全二叉树的顺序存储结构”
    - 采用一维数组，按层序顺序依次存储二叉树的每一个结点。如下图所示：

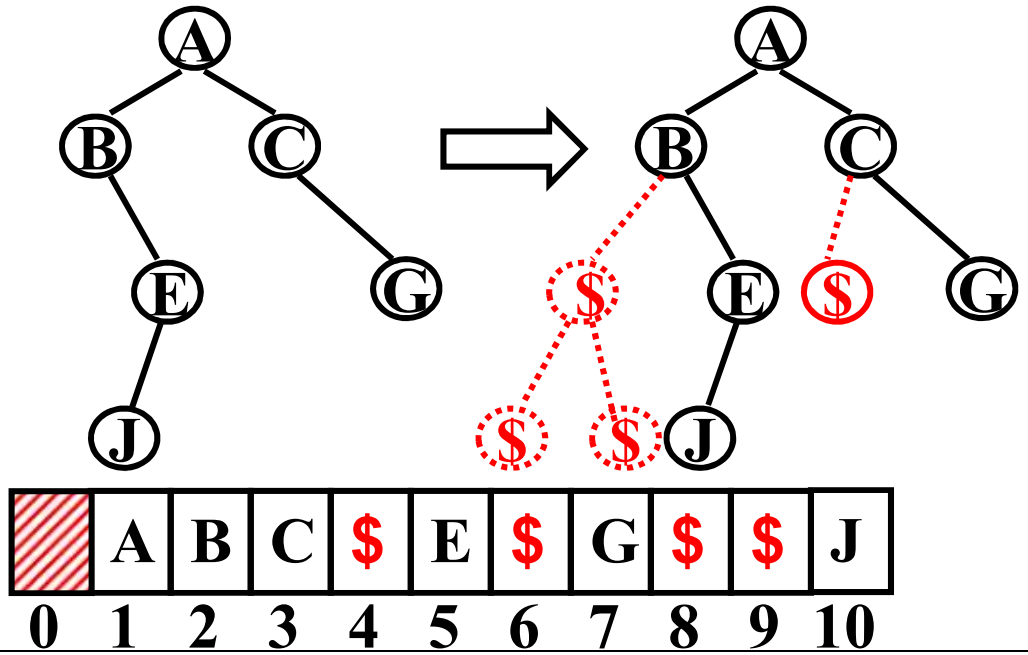
	A	B	C	D	E	F	G	H	I	J
0	1	2	3	4	5	6	7	8	9	10



3.2 二叉树(Cont.)

一般二叉树的顺序存储结构

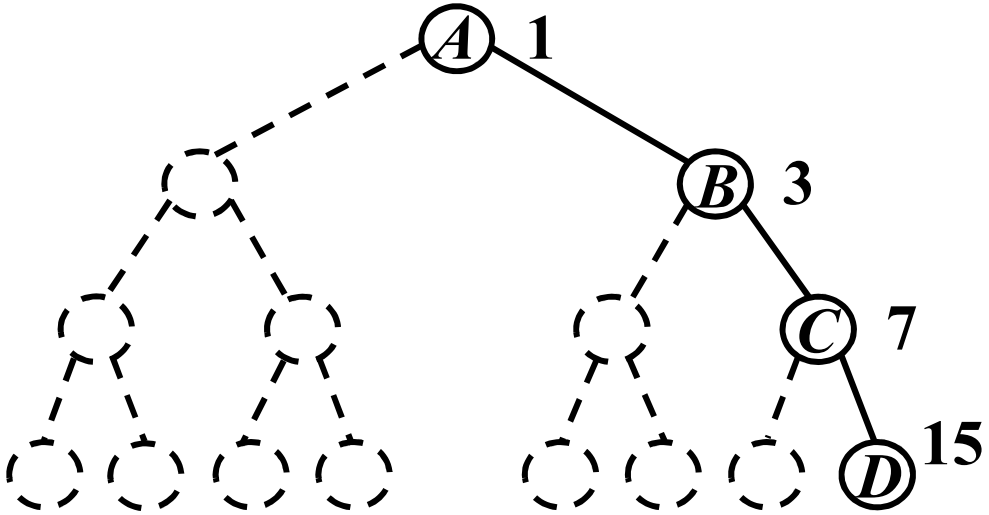
- 通过虚设部分结点，使其变成相应的完全二叉树。
- 根据性质5，如已知某结点的层序编号*i*,则可求得该结点的双亲结点、左孩子结点和右孩子结点，然后检测其值是否为虚设的特殊结点$。



3.2 二叉树(Cont.)

一般二叉树的顺序存储结构

- 一棵斜树的顺序存储会怎样呢？
  - 高度为h的右斜树，h个结点需分配2^h-1个存储单元.
  - 需增加很多空结点，造成存储空间的浪费.



	A	$	B	$	$	$	C	$	$	$	$	$	$	$	D
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

3.2 二叉树(Cont.)

二叉树的左右链存储结构----动态二叉链表

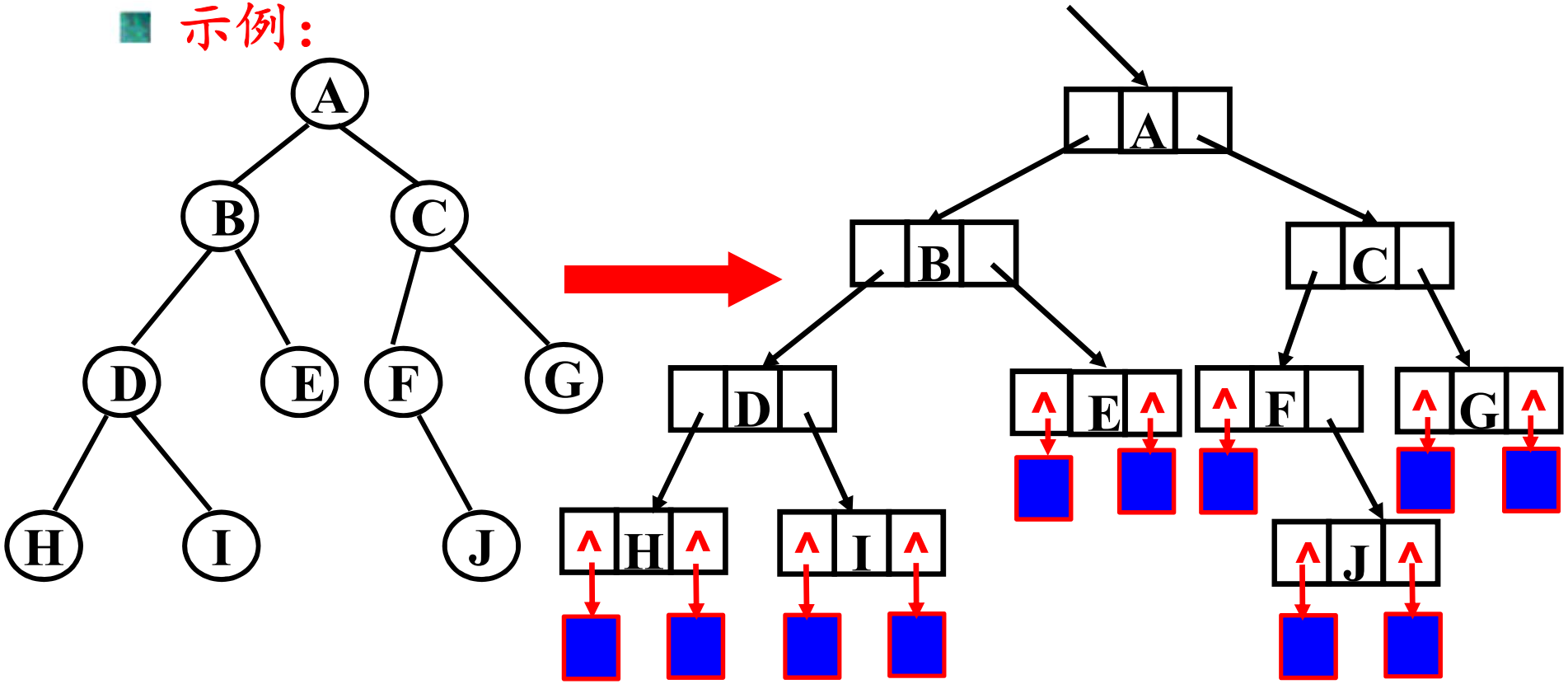
- 二叉树左右链表示：每个结点除了存放结点数据信息外，还设置两个指示左、右孩子的指针，如果该结点没有左或右孩子，则相应的指针为空。并用一个指向根结点的指针标识这个二叉树 。
- 结点结构：

lchild	data	rchild

- **data**：数据域，存放该结点的数据信息；
- **lchild**：左指针域，存放指向左孩子的指针；
- **rchild**：右指针域，存放指向右孩子的指针。

3.2 二叉树(Cont.)

- 二叉树的左右链存储结构----动态二叉链表
  - 示例：

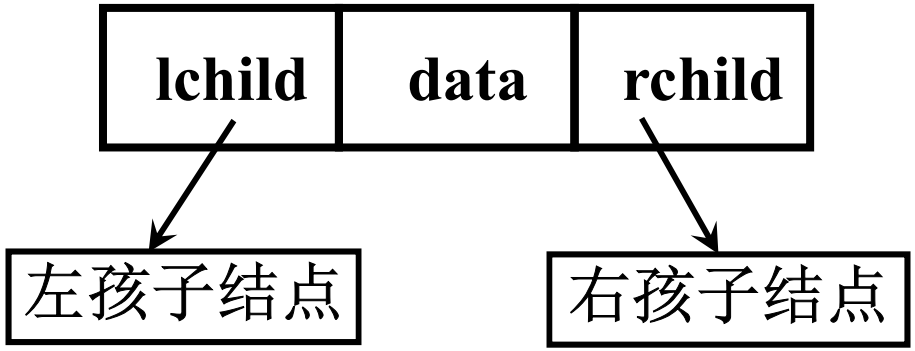


- 具有*n*个结点的二叉链表中，有多少个空指针？有多少指向孩子结点的指针？

3.2 二叉树(Cont.)

- 二叉树的左右链存储结构----动态二叉链表
  - 存储结构定义：

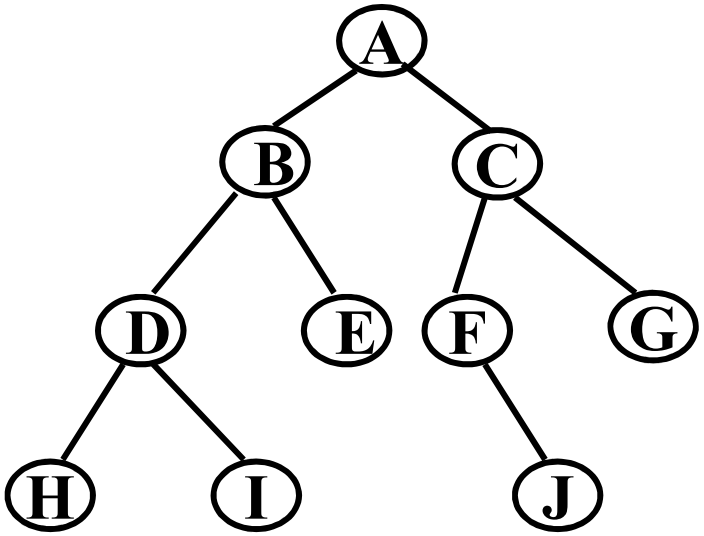
```
struct node {
    struct node *lchild ;
    struct node *rchild ;
        datatype data ;
} ;
typedef struct node * BTREE ;
```



3.2 二叉树(Cont.)

- 二叉树的左右链存储结构(二叉链表)的建立
  - 方法1:

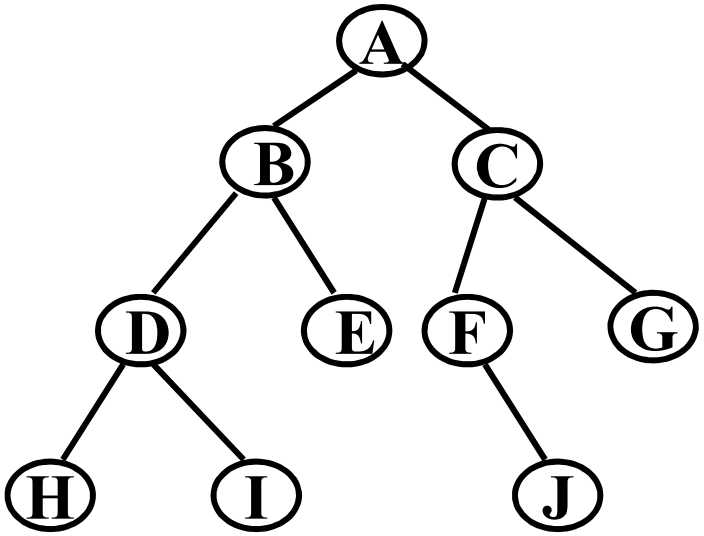
```
BTREE CreateBT(datatype v, BTREE ltree , BTREE rtree )
{
    BTREE  root ;
    root = new node ;
    root →data = v ;
    root →lchild = ltree ;
    root →rchild = rtree ;
    return  root ;
}
```



3.2 二叉树(Cont.)

- 方法2：按先序序列建立二叉树的左右链存储结构.
  - 如由图所示二叉树,输入:ABDH##I##E##CF#J##G##其中：#表示空子树

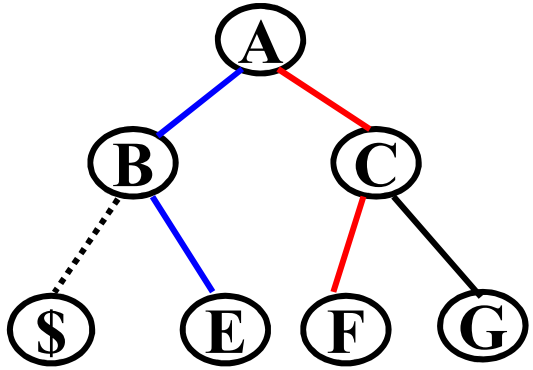
```
void CreateBT(BTREE & T)
{  cin >> ch ;
   if ( ch == '#' ) T = NULL ;
   else{   T=new node;
           T →data = ch ;
           CreateBT ( T →lchild ) ;
           CreateBT ( T →rchild ) ;
   }
}
```



3.2 二叉树(Cont.)

- 方法3：建立二叉树的左右链存储结构的非递归算法.

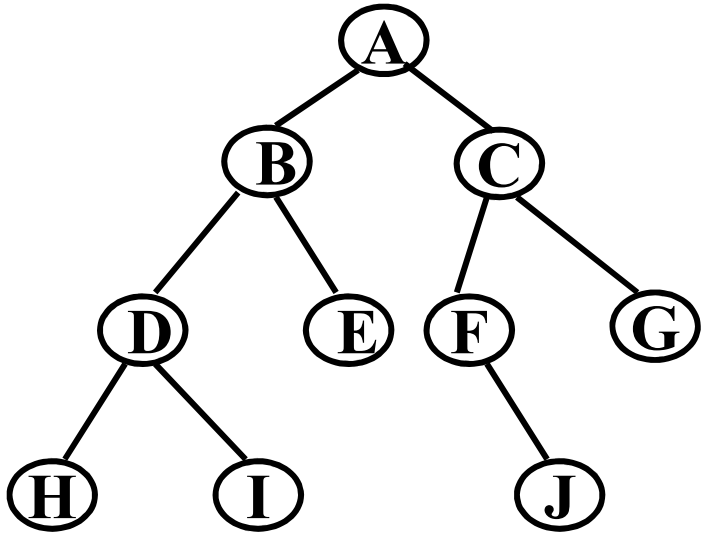
```
struct node *s[max]; /* 辅助指针数组，存放二叉树结点指针 */
BTREE CreateBT ( )
{ int i , j; datatype ch;
   struct node *bt, *p; /* bt为根，p 用于建立结点 */
   cin >> i>>ch ;
   while ( i != 0&&ch != 0) {
        p =new node;        p → data=ch;
        p → lchild=NULL;  p → lchild=NULL;
        s[ i ]=p;
        if  ( i == 1 ) bt = p ;
        else {   j =i /2;  /* 父结点的编号 */
                 if ( i %2==0 )  s[ j ]→lchild=p;  /* i 是 j 的左儿子 */
                 else            s[ j ]→rchild=p; /* i 是 j 的右儿子 */
       }cin >> i>>ch ;}
}
```



3.2 二叉树(Cont.)

- 二叉树左右链存储结构下的递归遍历算法.
  - 先序遍历

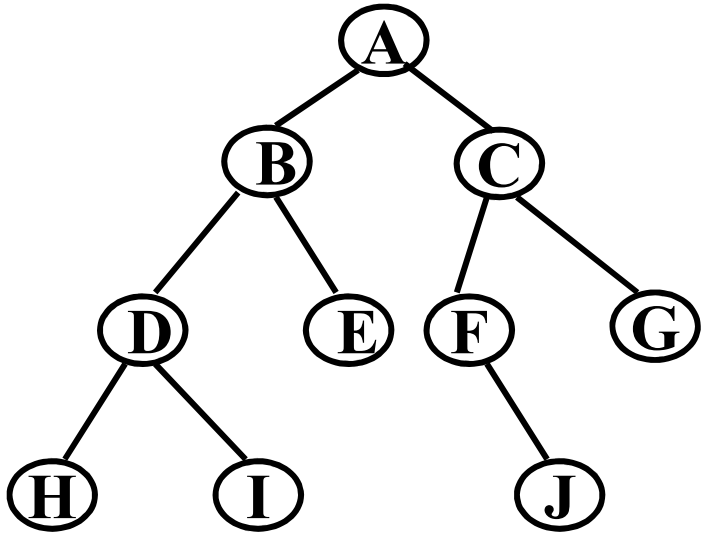
```
void  PreOrder (BTREE  BT)
{
    if ( BT != NULL)
    {
        cout<< BT->data  ;
        PreOrder ( BT->lchild ) ;
        PreOrder ( BT->rchild ) ;
    }
}
```



3.2 二叉树(Cont.)

- 二叉树左右链存储结构下的递归遍历算法.
  - 中序遍历

```
void  InOrder (BTREE  BT)
{
    if ( BT != NULL)
    {
        InOrder ( BT->lchild ) ;
        cout<< BT->data  ;
        InOrder ( BT->rchild ) ;
    }
}
```

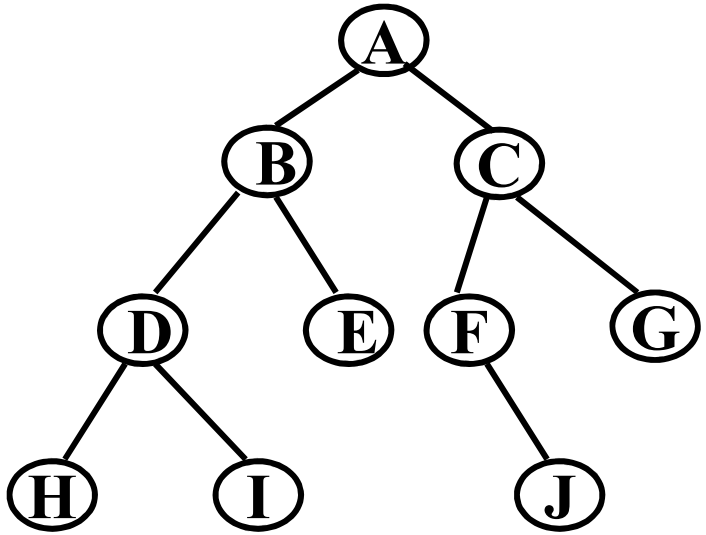


[[[[H] D [I]] B [E]] A [[F [J]] C [G]]]

3.2 二叉树(Cont.)

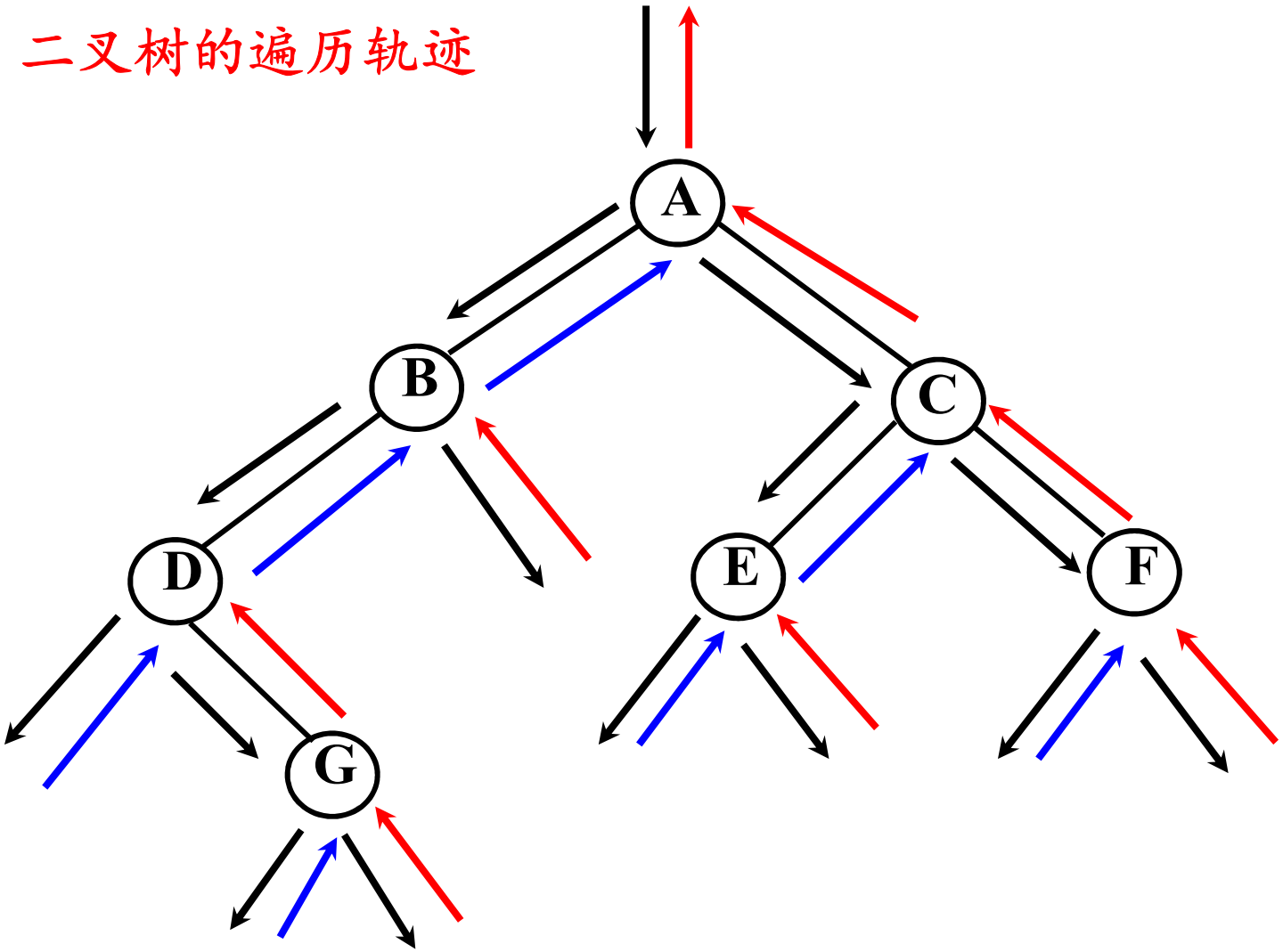
- 二叉树左右链存储结构下的递归遍历算法.
  - 后序遍历

```
void  PostOrder (BTREE  BT)
{
    if ( BT != NULL)
    {
        PostOrder ( BT->lchild ) ;
        PostOrder ( BT->rchild ) ;
        cout<< BT->data  ;
    }
}
```



3.2 二叉树(Cont.)

- 二叉树左右链存储结构下的非递归遍历算法
  - 二叉树的遍历轨迹



3.2 二叉树(Cont.)

- 先序遍历非递归算法

1.栈s初始化；

2.循环直到root为空且栈s为空

2.1 当root不空时循环

2.1.1 输出root->data;

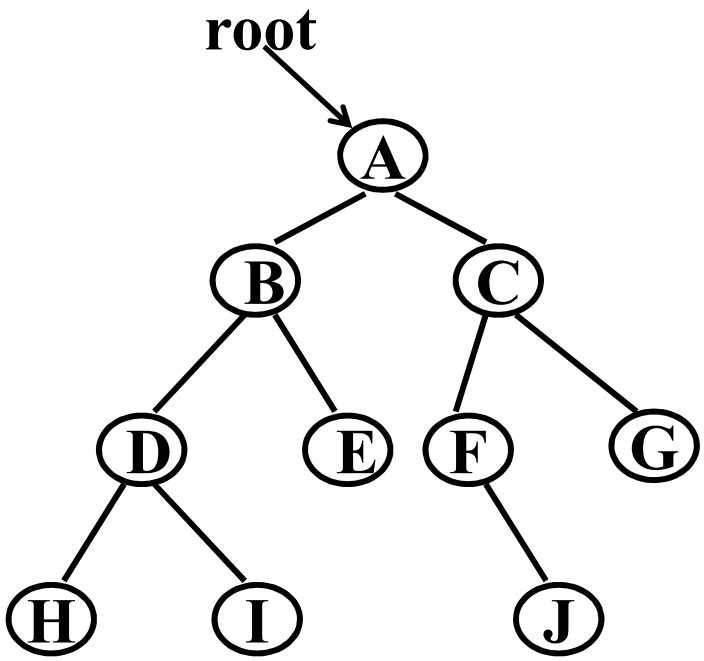
2.1.2 将指针root的值保存到栈中；

2.1.3 继续遍历root的左子树；

2.2 如果栈s不空，则

2.2.1 将栈顶元素弹出至root;

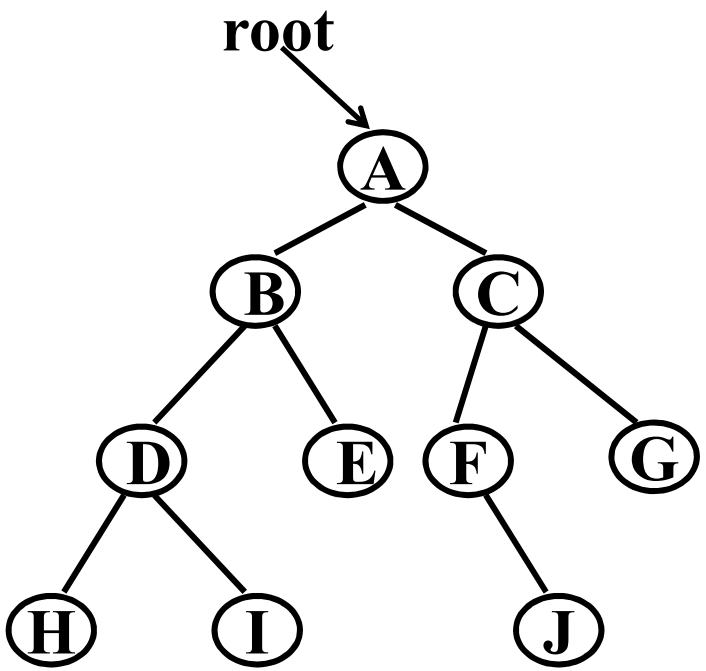
2.2.2 遍历root的右子树；



3.2 二叉树(Cont.)

- 先序遍历非递归算法

```
void PreOrder(BTREE root)
{   top= -1;     //采用顺序栈，并假定不会发生上溢
    while (root!=NULL || top!= -1) {
        while (root!= NULL) {
            cout<<root->data;
            s[++top]=root;
            root=root->lchild;
        }
        if (top!= -1) {
            root=s[top--];
            root=root->rchild;
        }
    }
}
```



3.2 二叉树(Cont.)

- 中序遍历非递归算法

1.栈s初始化；

2.循环直到root为空且栈s为空

2.1 当root不空时循环

2.1.1 将指针root的值保存到栈中；

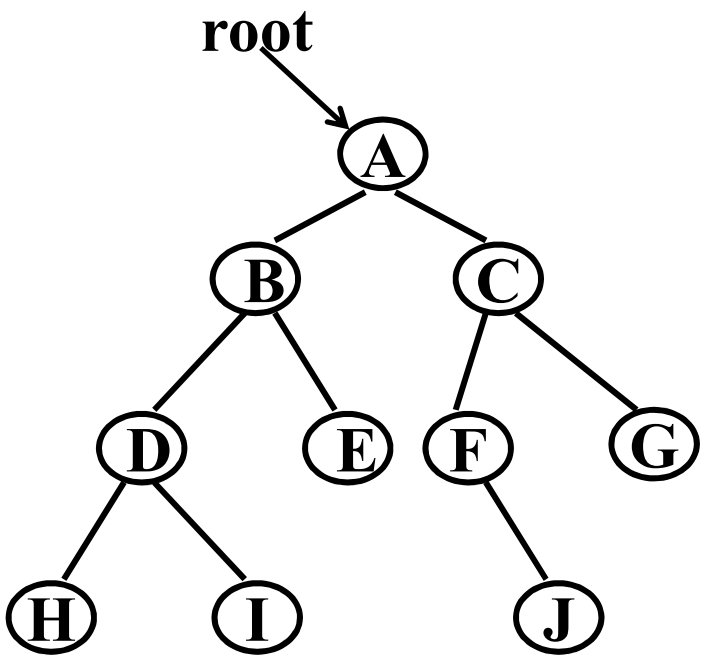
2.1.2 继续遍历root的左子树；

2.2 如果栈s不空，则

2.2.1 将栈顶元素弹出至root；

2.2.2 输出root->data;

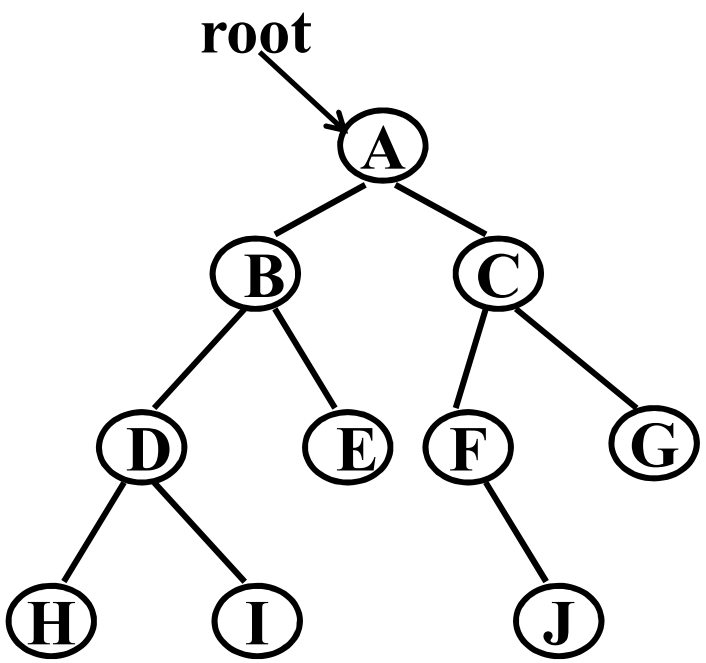
2.2.3 遍历root的右子树；



3.2 二叉树(Cont.)

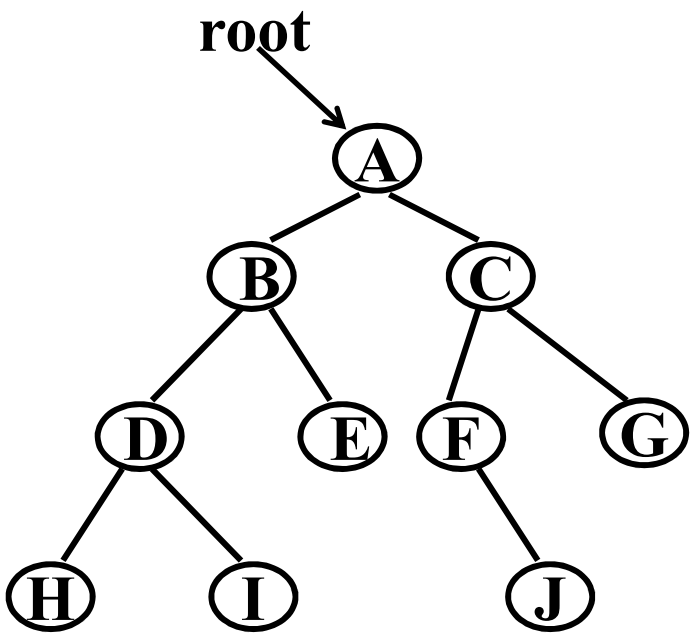
- 中序遍历非递归算法

```
void InOrder(BTREE  root)
{   top= -1;     //采用顺序栈，并假定不会发生上溢
    while (root!=NULL || top!= -1) {
        while (root!= NULL) {
            s[++top]=root;
            root=root->lchild;
        }
        if (top!= -1) {
            root=s[top--];
            cout<<root->data;
            root=root->rchild;
        }
    }
}
```



3.2 二叉树(Cont.)

- 后序遍历非递归算法



1. 栈s初始化；

2. 循环直到root为空且栈s为空

   2.1 当root非空时循环

      2.1.1 将root连同标志flag=1 入栈；

      2.1.2 继续遍历root的左子树；

   2.2 当栈s 非空且栈顶元素标志为2 时，出栈并输出栈顶结点；

   2.3 若栈非空，将栈顶元素标志改为2，遍历栈顶结点的右子树。

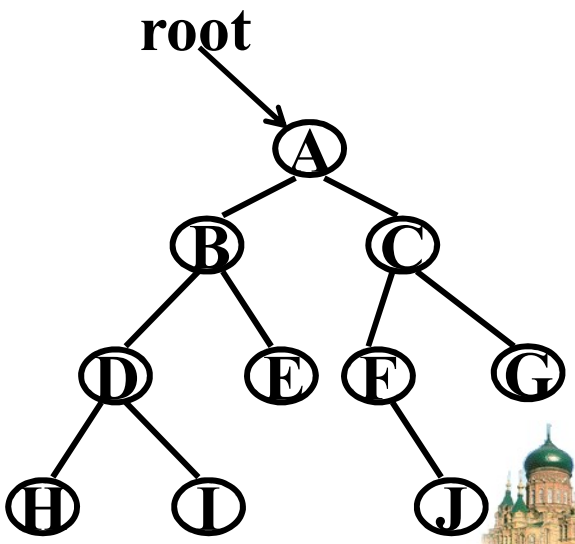
3.2 二叉树(Cont.)

- 后序遍历非递归算法

```
void PostOrder(BTREE root)
{  top= -1; //采用顺序栈，并假定栈不会发生上溢
   while (root!=NULL || top!= -1)  {
      while (root!=NULL)  {
        s[top++].ptr=root; s[top].flag=1; root=root->lchild;
      }
      while (top!= -1 && s[top].flag==2) {
         root=s[top--].ptr; cout<<root->data;
      }
      if (top!= -1) {
         s[top].flag=2; root=s[top].ptr->rchild;
      }
   }
}
```

栈单元结构

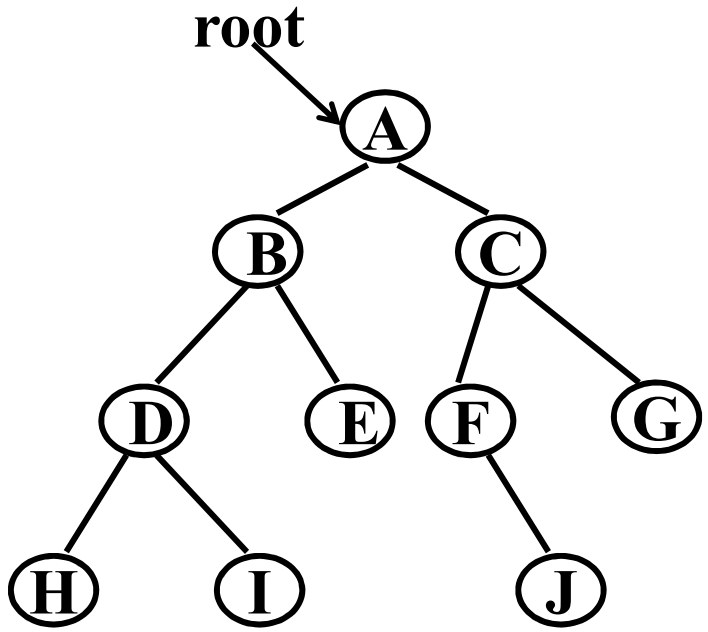
node *	flag



3.2 二叉树(Cont.)

层序遍历算法

- 基本思想：按层次顺序遍历二叉树的原则是先被访问的结点的左、右儿子结点先被访问，因此，在遍历过程中需利用具有先进先出特性的队列结构。



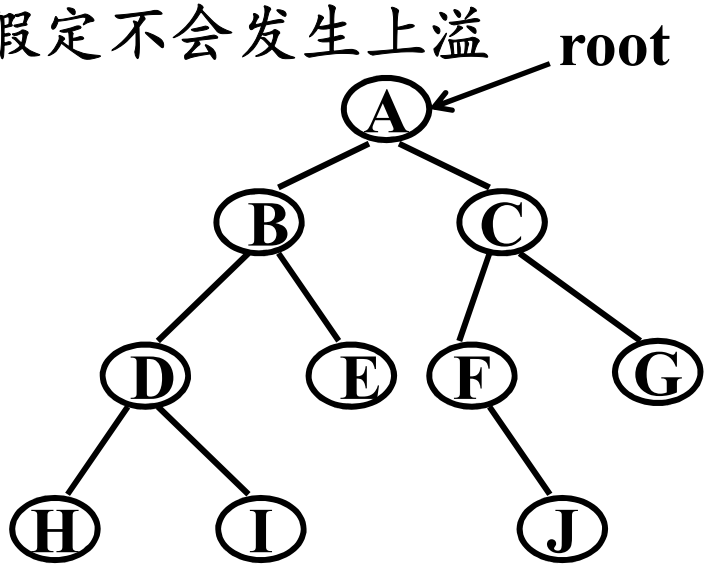
- 实现步骤:

1. 队列Q初始化；
2. 如果二叉树非空，将根指针入队；
3. 循环直到队列Q为空
   - 3.1 q=队列Q的队头元素出队；
   - 3.2 访问结点q的数据域；
   - 3.3 若结点q存在左孩子，则将左孩子指针入队；
   - 3.4 若结点q存在右孩子，则将右孩子指针入队；

3.2 二叉树(Cont.)

- 层序遍历算法

```
void LeverOrder (BTREE root)
{   front=rear=0; //采用顺序队列，并假定不会发生上溢
    if (root==NULL) return;
        Q[++rear]=root;
    while (front!=rear) {
        q=Q[++front];
        cout<<q->data;
        if (q->lchild!=NULL)  Q[++rear]=q->lchild;
        if (q->rchild!=NULL)  Q[++rear]=q->rchild;
    }
}
```



3.2 二叉树(Cont.)

- 遍历算法应用举例
  - 二叉树的遍历是二叉树各种操作和算法的基础，遍历算法中对每个结点的“访问”操作可以是对结点进行的各种处理
  - 根据遍历算法的框架，适当修改访问操作的内容，可以派生出很多关于二叉树的应用算法。
  - 因此，二叉树遍历算法是有关二叉树算法中最核心的算法.
- 计算二叉树结点个数的递归算法。

```
int Count ( BTREE T )
{    if ( T == NULL )  return 0;
     else  return 1 + Count ( T->lchild )
                    + Count ( T->rchild );
}
```

```
struct  node  {
    struct  node  *lchild ;
    struct  node  *rchild ;
            datatype  data ;
} ;
typedef node  * BTREE ;
```

3.2 二叉树(Cont.)

```
void Count(BiNode *root)
{//n为全局量并已初始化为0
    if (root) {
        Count(root->lchild);
        n+ +;
        Count(root->rchild);
    }
}
```

- 求二叉树高度的递归算法

```
int Height (BTREE T )
{  if ( T == NULL ) return 0;
   else {int m = Height ( T->lchild );
         int n = Height ( T->rchild ) );
         return (m > n) ? (m+1) : (n+1);
         }
}
```

```
void Destroy (BTREE  T)
{
   if ( T != NULL ) {
      Destroy ( T->lchild );
      Destroy ( T->rchild );
      delete T;
   }
}
```

删除二叉树的递归算法

```
struct  node  {
    struct  node  *lchild ;
    struct  node  *rchild ;
          datatype  data ;
} ;
typedef node  * BTREE ;
```

3.2 二叉树(Cont.)

- 交换二叉树所有结点子树的算法.

```
void  Exchange ( BTREE T )
{   Node *p = T, *tmp;
    if ( p != NULL ) {
        temp = p->lchild;
        p->lchild = p->rchild;
        p->rchild = tmp;
        Exchange ( p->lchild );
        Exchange ( p->rchild );
    }
}
```

```
struct  node  {
    struct  node  *lchild;
    struct  node  *rchild;
        datatype   data;
};
typedef node * BTREE;
```

3.2 二叉树(Cont.)

```
void Exchange ( BREE T )  //非递归算法
{    struct node *p, *tmp;
     top = -1;    //采用顺序栈，并假定不会发生上溢
     if ( T != NULL ) {
        s[++top] = T;
        while ( top !  =-1 ) {
            p = s[top--]; //栈中退出一个结点
            tmp = p->lchild//交换子女
            p->lchild = p->rchild;
            p->rchild = tmp;
            if ( p->lchild != NULL )
                s[++top] = p->lchild;
            if ( p->rchild != NULL )
                s[++top] = p->rchild;
        }使用栈消去递归算法中的两个递归语句
     }
}
```

```
struct  node  {
    struct  node  *lchild;
    struct  node  *rchild;
        datatype   data;
};
typedef node * BTREE;
```

3.2 二叉树(Cont.)

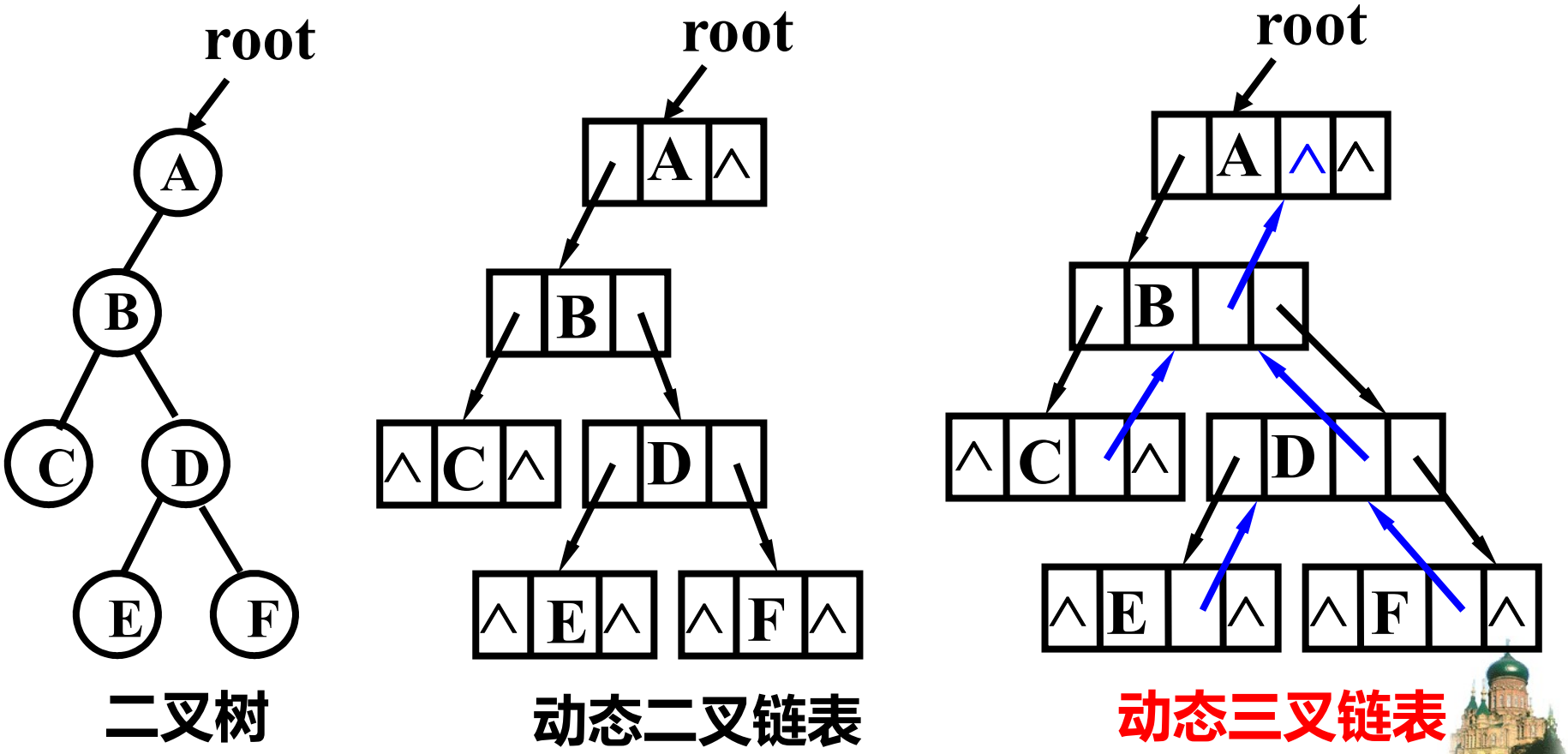
- 按先序次序打印二叉树中的叶子结点的算法.

```
void PreOrderPrnLeaf(BTREE T )
{
   if (T) {
      if (!T->lchild && !T->rchild)
            cout<<T->data;
      PreOrderPrnLeaf (T->lchild);
      PreOrderPrnLeaf (T->rchild);
   }
}
```

```
struct  node  {
    struct  node  *lchild;
    struct  node  *rchild;
            datatype   data;
};
typedef node * BTREE;
```

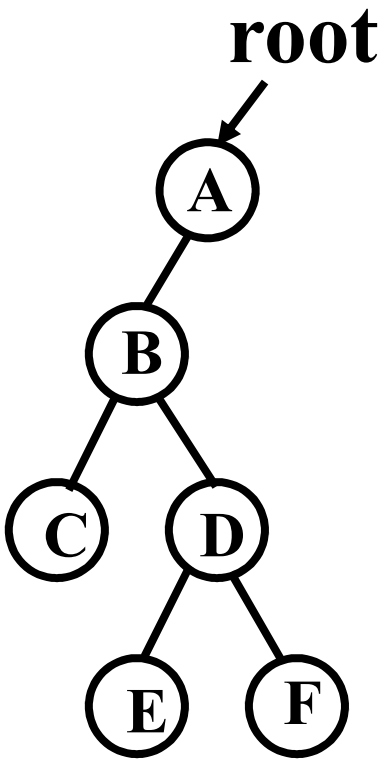
3.2 二叉树(Cont.)

- 二叉树的其他链式存储结构----动态三叉链表
  - 在二叉链表的基础上增加了一个指向双亲的指针域。



3.2 二叉树(Cont.)

- 二叉树的其他链式存储结构----静态二叉链表和三叉链表



二叉树

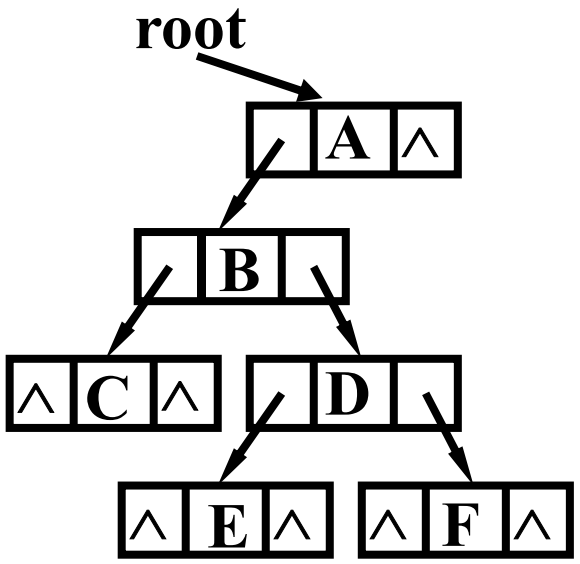
	data	parent	lchild	rchild
0	A	-1	1	-1
1	B	0	2	3
2	C	1	-1	-1
3	D	1	4	5
4	E	3	-1	-1
5	F	3	1	1

静态二叉链表和三叉链表

3.2 二叉树(Cont.)

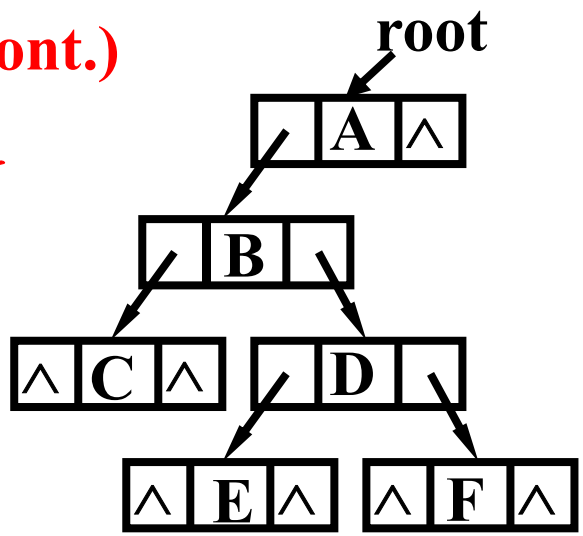
二叉树的线索链表存储结构----线索二叉树

- 二叉链表的空间利用情况如何？
  - 在n（n≥1）个结点的二叉树左右链表示中，只有n-1个指向子树的指针，却有n+1个空指针域。
- 在二叉链表中如何找某个结点的某种遍历的前驱和后继？
  - 每次都要从根结点进行遍历？！
- 如何遍历二叉链表表示的二叉树？
  - 利用栈或队列，能否不用？
- 如何表示二叉链表的遍历序列？
- 如何利用空指针域解决上述问题？



动态二叉链表

3.2 二叉树(Cont.)

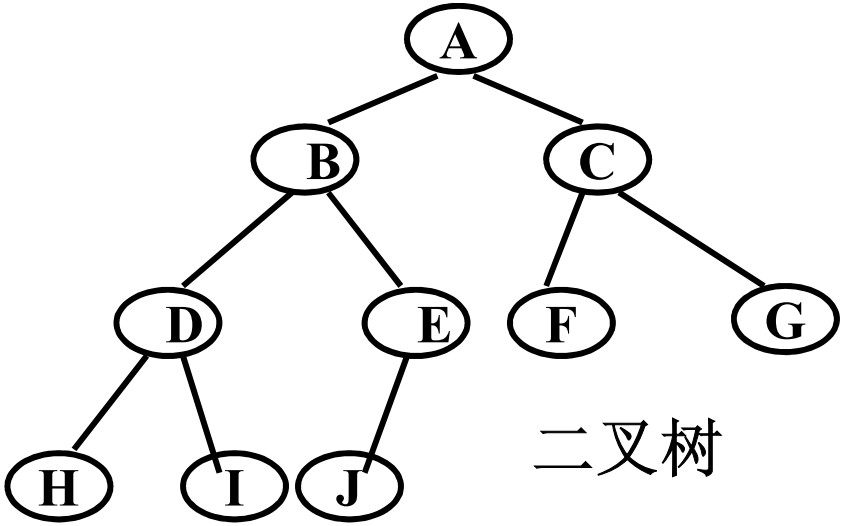


二叉树的线索链表存储结构----线索二叉树

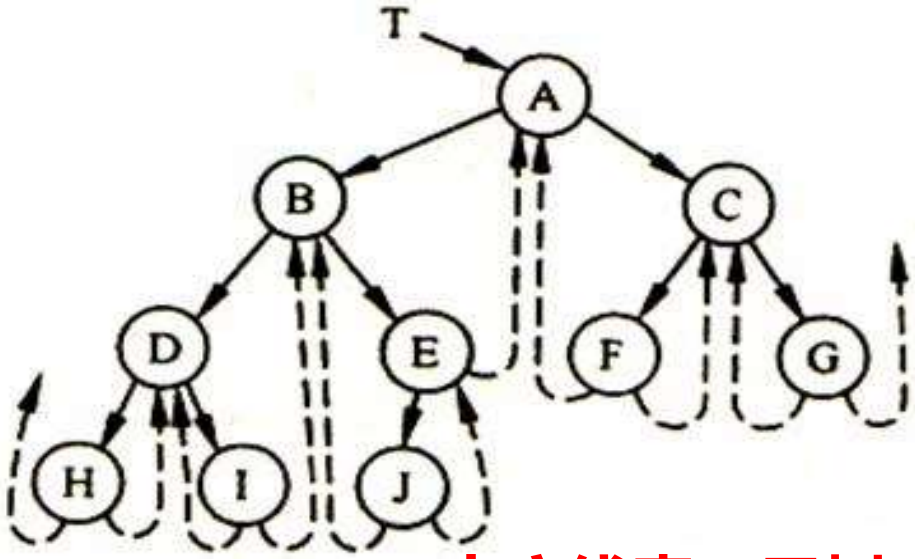
- 如何利用空指针域解决上述问题？
  - 若结点p有左孩子，则p->lchild指向其左孩子结点，否则令其指向其（先序、中序、后序、层序）前驱；
  - 若结点p有右孩子，则p->rchild指向其右孩子结点，否则令其指向其（先序、中序、后序、层序）后继；
- 如何区分指针是指向其左/右孩子的指针还是指向某种遍历的前驱/后继？
  - 在每个结点中增加两个标志位，以区分该结点的的两个链域是指向其左/右孩子还是指向某种遍历的前驱/后继。

3.2 二叉树(Cont.)

二叉树的线索链表存储结构----线索二叉树



二叉树



中序线索二叉树

结点结构

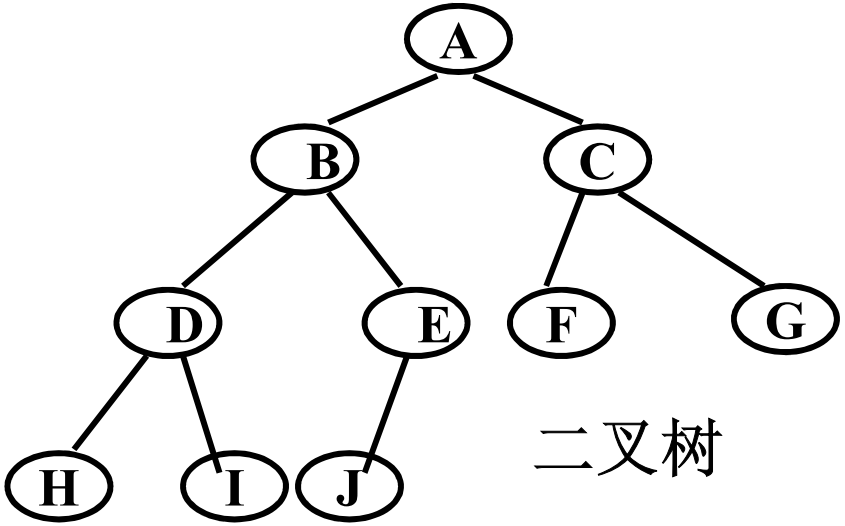
lchild	ltag	data	rchlid	rtag

$$p\text{->ltag} = \begin{cases} \text{TRUE} & \text{p->lchild 指向左孩子} \\ \text{FALSE} & \text{p->lchild 指向（中序）前驱} \end{cases}$$

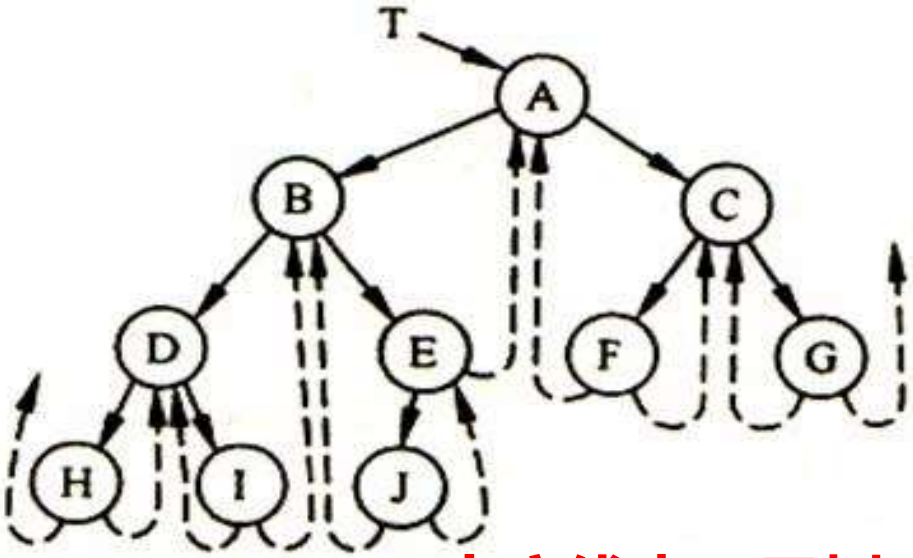
$$p\text{->rtag} = \begin{cases} \text{TRUE} & \text{p->rchild 指向右孩子} \\ \text{FALSE} & \text{p->lchild 指向（中序）后继} \end{cases}$$

3.2 二叉树(Cont.)

二叉树的线索链表存储结构----线索二叉树



二叉树

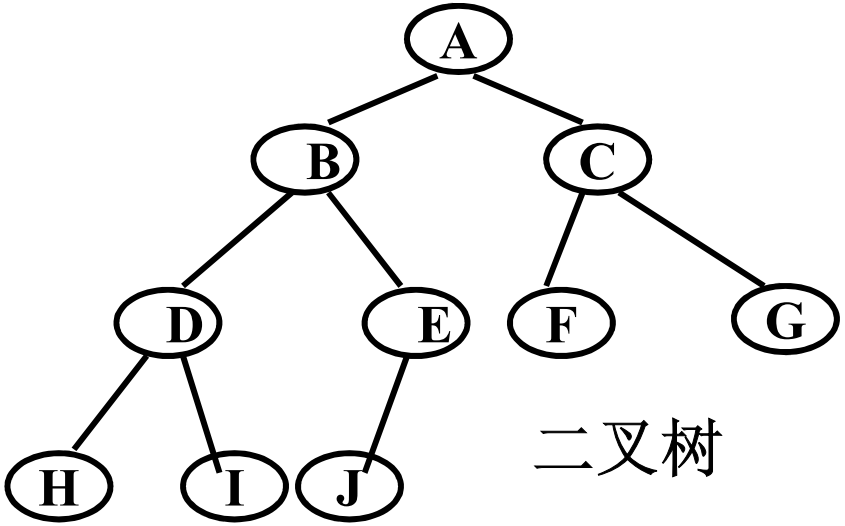


中序线索二叉树

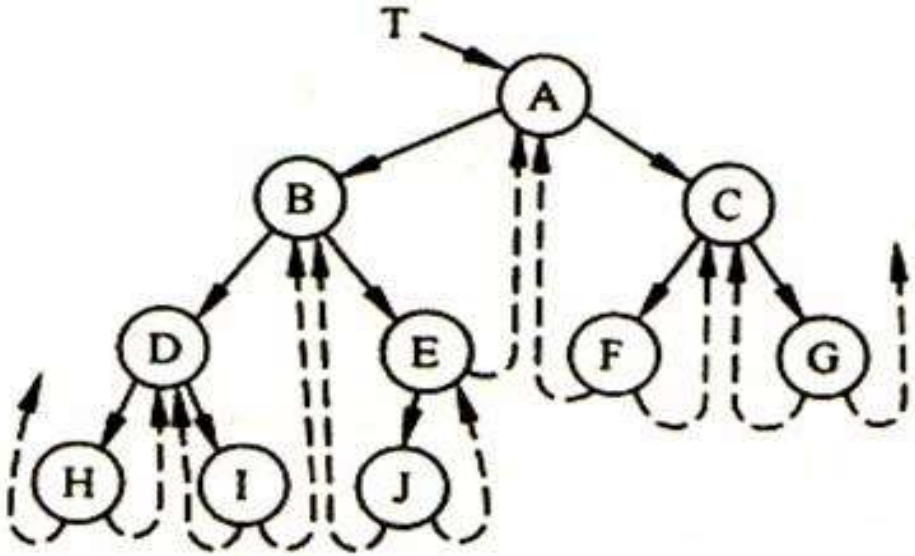
- 线索：将结点的空指针域指向其前驱/后继的指针被称为线索；
- 线索化：结点的空链域存放其前驱/后继的过程称为线索化；
- 线索二叉树：线索化的二叉树称为线索二叉树。

3.2 二叉树(Cont.)

二叉树的线索链表存储结构----线索二叉树



二叉树



中序线索二叉树

- 二叉树的遍历方式有4种，故有4种意义下的前驱和后继，相应的有4种线索二叉树：

  (1) 先序线索二叉树；(2) 中序线索二叉树；

  (3) 后序线索二叉树；(4) 层序线索二叉树。

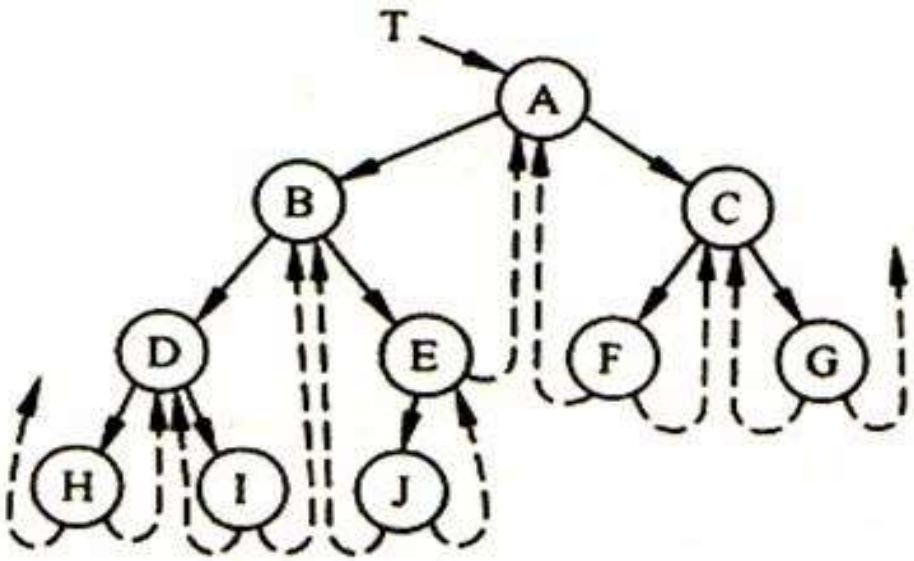
3.2 二叉树(Cont.)

- 线索链表的存储结构定义

结点结构

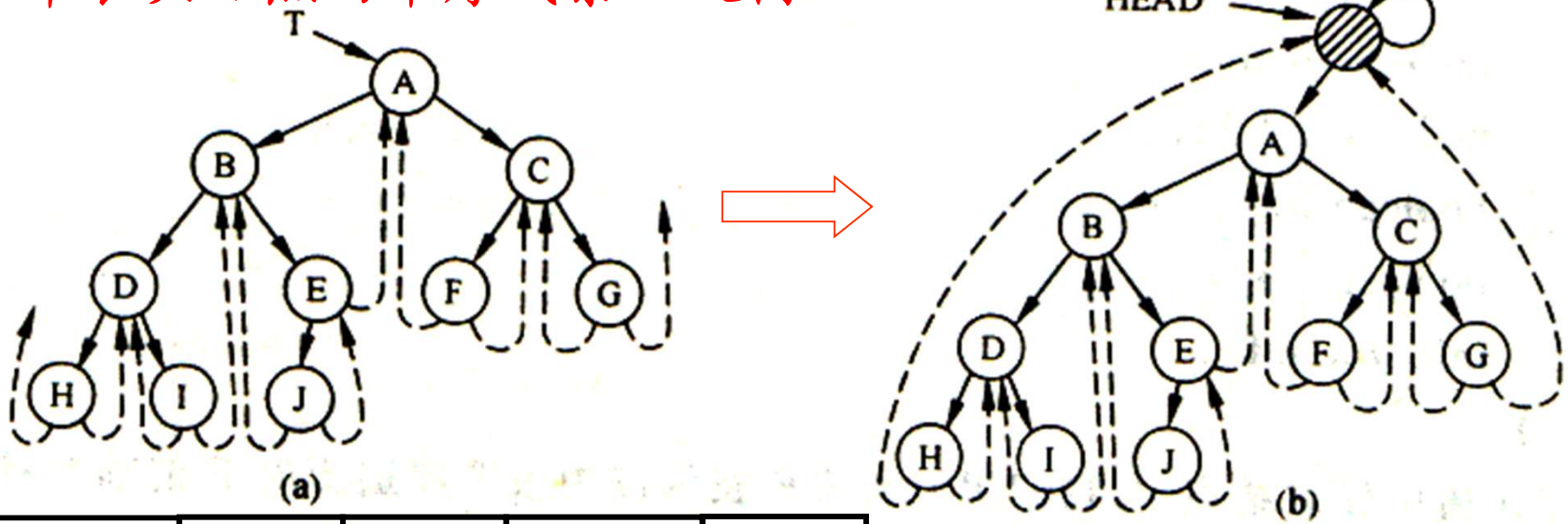
lchild	ltag	data	rchlid	rtag

```
struct  node {
      datatype  data ;
      struct  node *lchild, *rchild;
                bool   ltag, rtag;
};
typedef  struct node  *  THTREE;
```



3.2 二叉树(Cont.)

- 带表头结点的中序线索二叉树



lchild	ltag	data	rchlid	rtag

非空二叉树：

head->lchild = T；　（根）

head->ltag = TRUE ;

head->rchild = head ;

head->rtag = TRUE ;

空二叉树：

head->lchild = head ;

head->ltag = FALSE ;

head->rchild = head ;

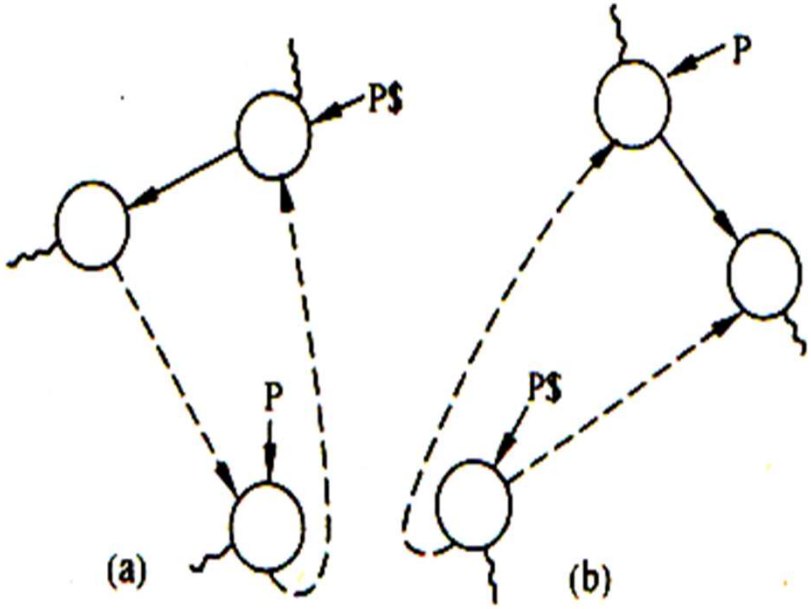
head->rtag = TRUE ;

3.2 二叉树(Cont.)

线索二叉树的若干算法

- 算法1：在中序线索二叉树中求一个结点p的中序后继p$
- 分析:
  - (1)当p->rtag==FALSE时，p->rchild 即为所求(线索)。
  - (2)当p->rtag==TRUE时，p$为p 的右子树的最左结点。
- 算法实现:

```
THTREE  InNext( THTREE  p)
{    THTREE  q ;
     q =p->rchild ;
     if (p->rtag = = TRUE )
        while( q->ltag = = TRUE )
        q = q->lchild ;
    return  q;
}
```

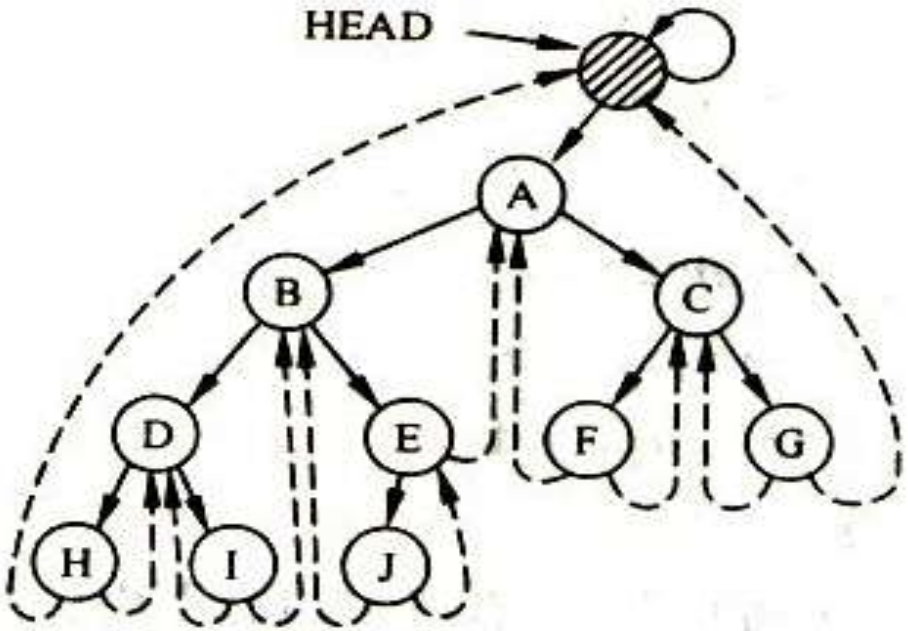


3.2 二叉树(Cont.)

- 算法2：利用InNext算法，中序遍历线索二叉树
- 算法实现：

```
void InOrderTh (THTREE head)
{   THTREE   tmp ;
    tmp = head ;
    do {
        tmp = InNext ( tmp ) ;
        if ( tmp !=  head )
            visit ( tmp -> data ) ;
    } while ( tmp !=  head ) ;
}
```

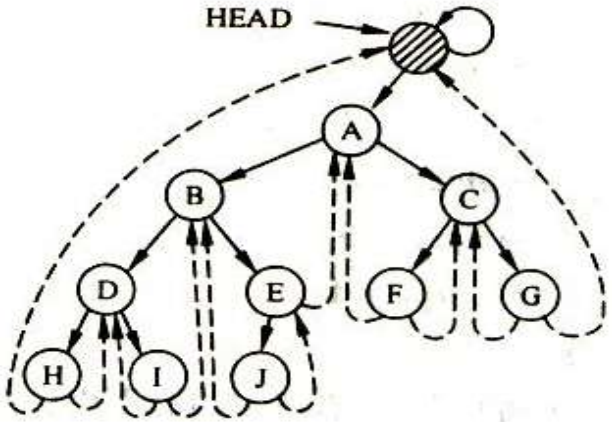
```
head->lchild = T
head->rchild = head ;
head->ltag = TRUE ;
head->rtag = TRUE ;
```



3.2 二叉树(Cont.)

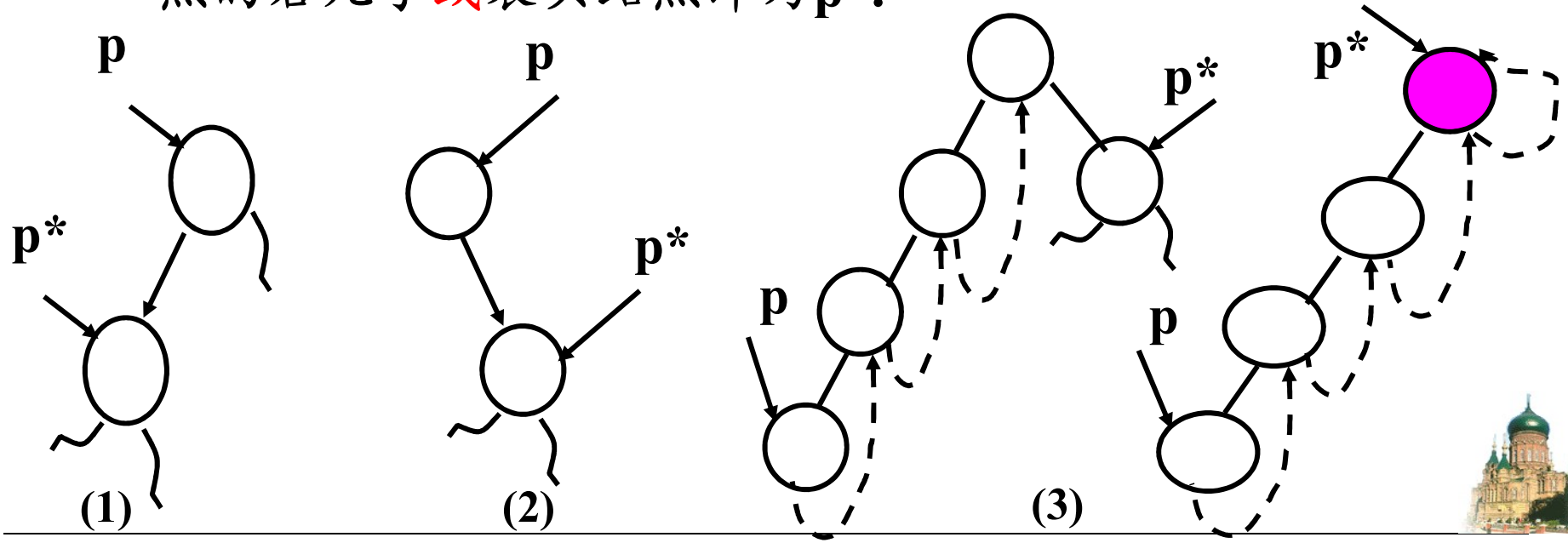
- 算法2'：中序遍历线索二叉树

```
void InOrderTh2 (THTREE head)
{    THTREE p = head->lchild;  //p指向根结点
     while(p != head) {                //空树或遍历结束时p == head
        while(p->ltag == TRUE ) //当ltag = 0时循环到中序序列的第1个结点
            p = p->lchild;
        printf("%c ", p->data); //显示结点数据，可以更改为其他对结点的操作
        while(p->rtag == FALSE && p->rchild != head) {
            p = p->rchild;
            printf("%c ", p->data);
        }
        p = p->rchild;  //p进入其右子树
     }
}
```



3.2 二叉树(Cont.)

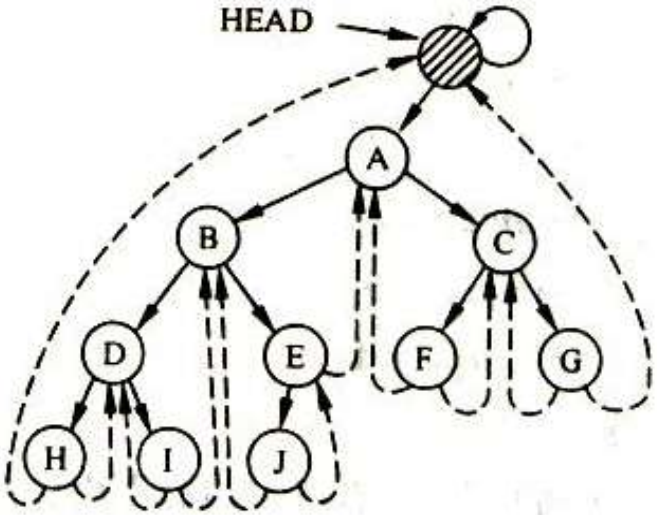
- 算法3：求中序线索二叉树中结点p 的先序序列的后继结点p*
- 分析：
  - (1) p 的左子树不空时，p 的左儿子p->lchild 即为 p*；
  - (2) p 的左子树空但右子树不空时，p 的p->rchild 为p*；
  - (3) p 的左右子树均空时，右线索序列中第一个有右儿子结点的右儿子或表头结点即为p*.



3.2 二叉树(Cont.)

- 算法3：求中序线索二叉树中结点p 的先序序列的后继结点p*
- 算法实现：

```
THTREE   PreNext( THTREE  p)
{
    THTREE   q ;
    if (p->ltag = = TRUE )
            q = p->lchild ;   //(1)
     else{  q = p ;            //(3)
            while(q->rtag = = FALSE)
                q = q->rchild ;
            q = q->rchild ;  //(2)
      }
      return  q ;
}
```



3.2 二叉树(Cont.)

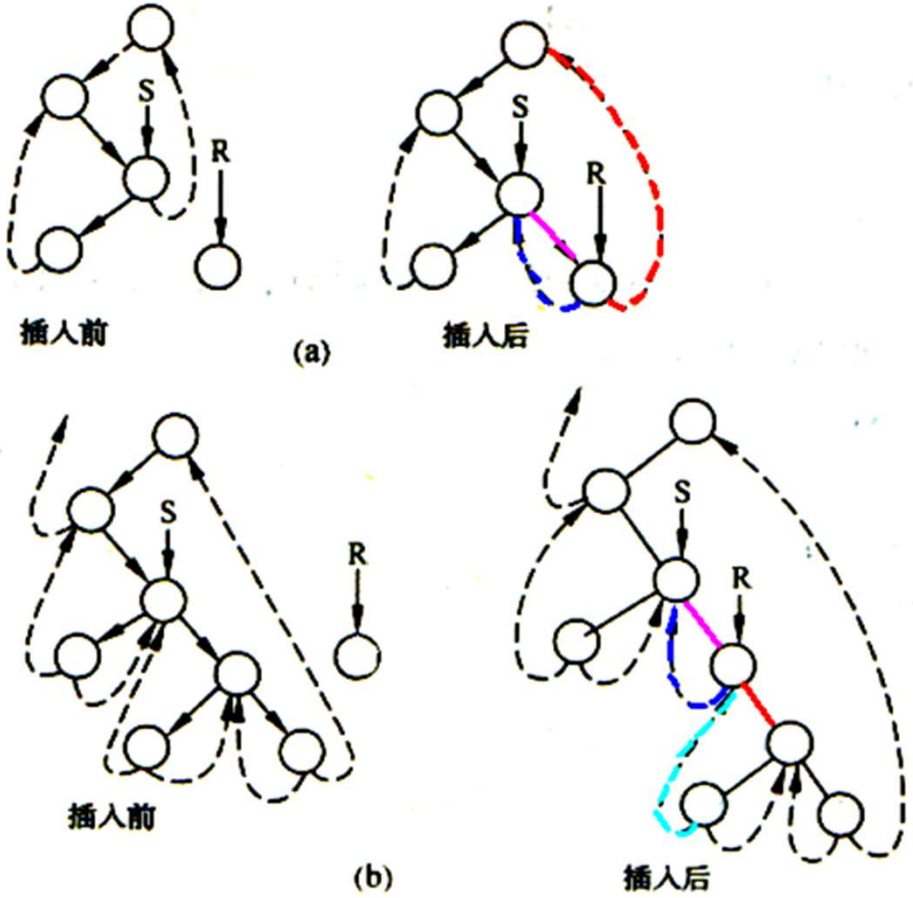
线索二叉树的若干算法

- 算法4：中序线索二叉树的插入算法
- 分析：如将结点 R 插入作为结点 S 的右孩子结点。

(1)若S的右子树为空，直接插入;

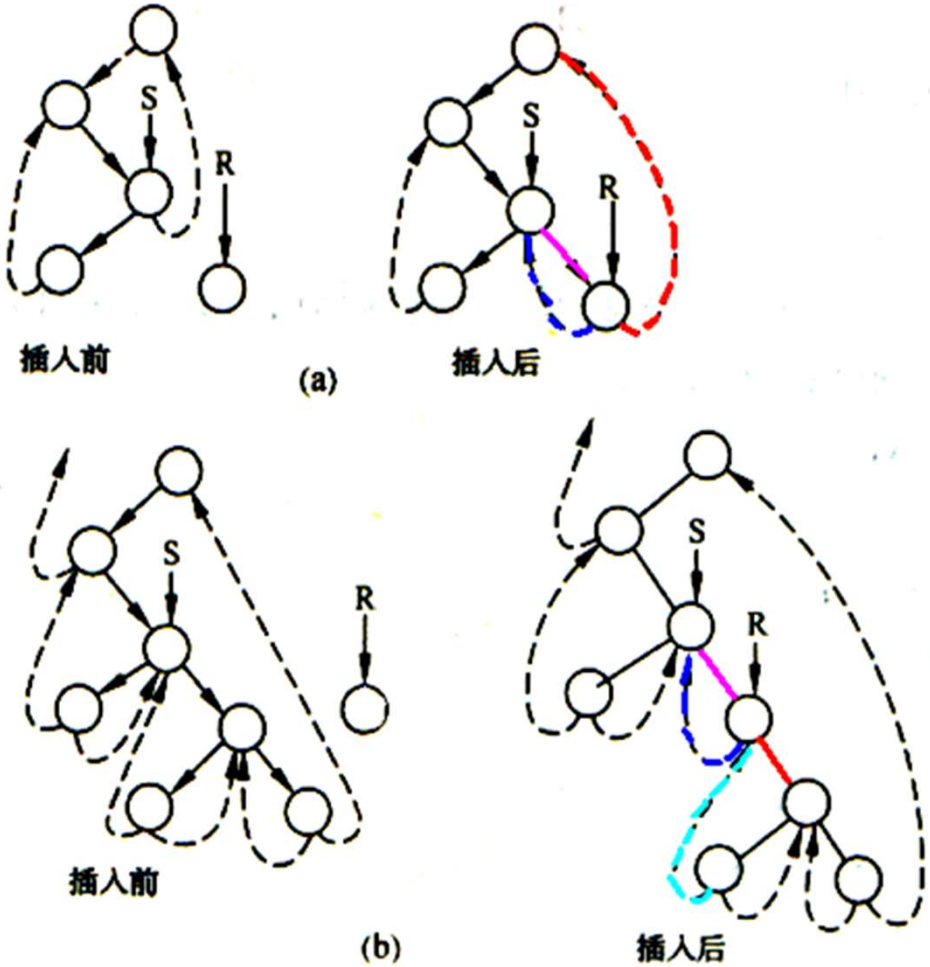
(2)若S的右子树非空，则 R 插入后，原来 S 的右子树作为 R 的右子树

R完全继承原来S的右子树



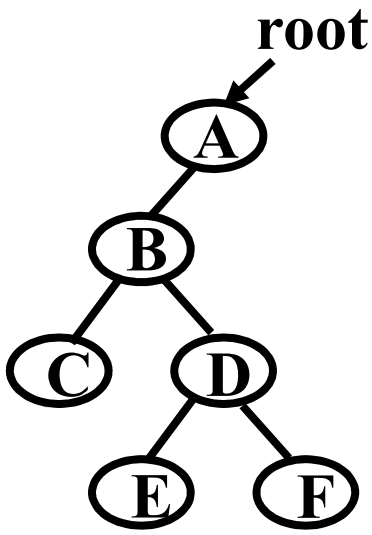
3.2 二叉树(Cont.)

```
void RInsert (THTREE S ,THTREE R)
{  THTREE   w ;
   R->rchild = S->rchild;
   R->rtag = S->rtag ;
   R->lchild = S ;//--
   R->ltag = FALSE ;//--
   S->rchild = R ;
   S->rtag = TRUE ;
   if (R->rtag==TRUE) {
        w = InNext( R ) ;
        w->lchild = R ;  }
}
```

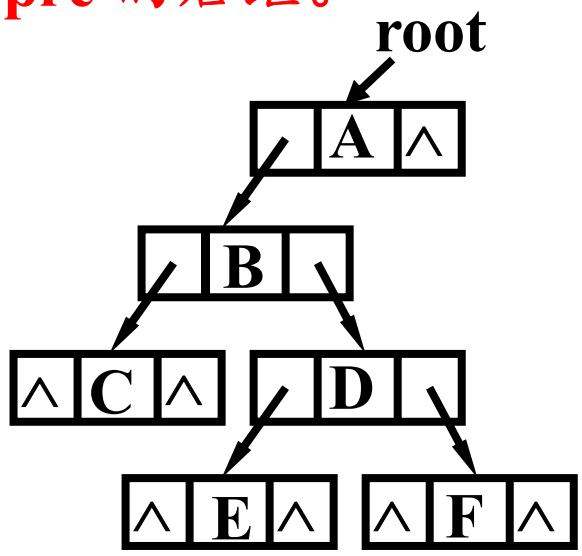


3.2 二叉树(Cont.)

- 算法5：二叉树的（中序）线索化算法---递归算法
- 基本思想：
  - 二叉树线索化，只要按某种次序遍历二叉树，在遍历过程中用线索取代空指针即可。
  - 为此，附设一个指针pre---始终指向刚刚访问过的结点，而指针p 指示当前正在访问的结点。显然，结点*pre是结点*p 的前驱，而 *p是结点*pre 的后继。



二叉树



动态二叉链表

3.2 二叉树(Cont.)

- 算法5：二叉树的（中序）线索化算法--递归算法
- 实现步骤：

1 如果二叉链表root为空，则返回；否则，

2 对root的左子树建立线索；

3 对根结点root建立线索；

3.1 若root没有左孩子，则为root加上前驱线索pre;

3.2 若结点pre右标志为FALSE，则为pre加上后继线索root;

3.3 令pre指向刚刚访问的结点root;

4 对root的右子树建立线索。

3.2 二叉树(Cont.)

```
BTREE *pre=NULL;  //全局量
void  InOrderTh(THTREE p) //将二叉树 p中序线索化
{   if( p ){ //p 非空时，当前访问的结点是 p
        InOrderTh( p->lchild ); //递归地线索化左子树
        p->ltag=( p->lchild ) ? TRUE : FALSE; //左(右)孩子非空
        p->rtag=( p->rchild )? TRUE : FALSE; //时,标志1,否: 0
        if ( pre ) { //若*p 的前驱*pre 存在
            if ( pre->rtag ==FASLE) //  *p的前驱右标志为线索
                pre->rchild=p; // 令 *pre 的右线索指向中序后继
            if ( p->ltag ==FASLE) //   *p的左标志为线索
                p->lchild=pre;        //令 *p的左线索指向中序前驱
        }
        pre = p;   // 令pre 是下一个访问的中序前驱
        InOrderTh( p->rchild ); //递归地线索化右子树
    }
}
```

3.2 二叉树(Cont.)

线索二叉树前驱和后继

结点p	先序的后继	中序的后继	后序的后继	先序的前驱	中序的前驱	后序的前驱
先序线索树	若p->ltag==TRUE，则p->lchild为后继；若p->ltag=FALSE，则p->rchild为后继	√	√	当p->ltag==FALSE时，则p->lchild为前驱；当p->ltag==TRUE时，需遍历才能求得前驱	√	√
中序线索树	p*	p$	√	√	$p	√
后序线索树	√	√	若p->rtag==FALSE，则p->rchild为后继；若p->rtag==TRUE，需遍历才能求得后继	√	√	若p->rtag==TRUE，则p->rchild为前驱；若p->rtag==FALSE，p->lchild为前驱

3.2 二叉树(Cont.)

二叉树的复制

- 两株二叉树具有相同结构指：
  - （1）它们都是空的；
  - （2）它们都是非空的，且左右子树分别具有相同结构.

“形状”相同

- 相似二叉树:具有相同结构的二叉树为相似二叉树。
- 相似且对应结点包含相同信息的二叉树称为等价二叉树。
- 判断两株二叉树是否等价的算法：

```
struct node {
    struct node *lchild ;
    struct node *rchild ;
    datatype data ;
};
typedef struct node * BTREE;
```

3.2 二叉树(Cont.)

```
int  Equal( BTREE firstbt, BTREE secondbt )
{    int  x ;
     x = 0 ;
     if ( IsEmpty(firstbt) && IsEmpty(secondbt) )
         x = 1 ;
     else if ( !IsEmpty( firstbt ) && ! IsEmpty( secondbt ) )
         if (  Data( firstbt ) == Data( secondbt )  )
            if ( Equal( Lchild( firstbt ) , Lchild( secondbt ) ) )
              x= Equal( Rchild( firstbt ) , Rchild( secondbt) )
     return( x ) ;
} /* Equal */
```

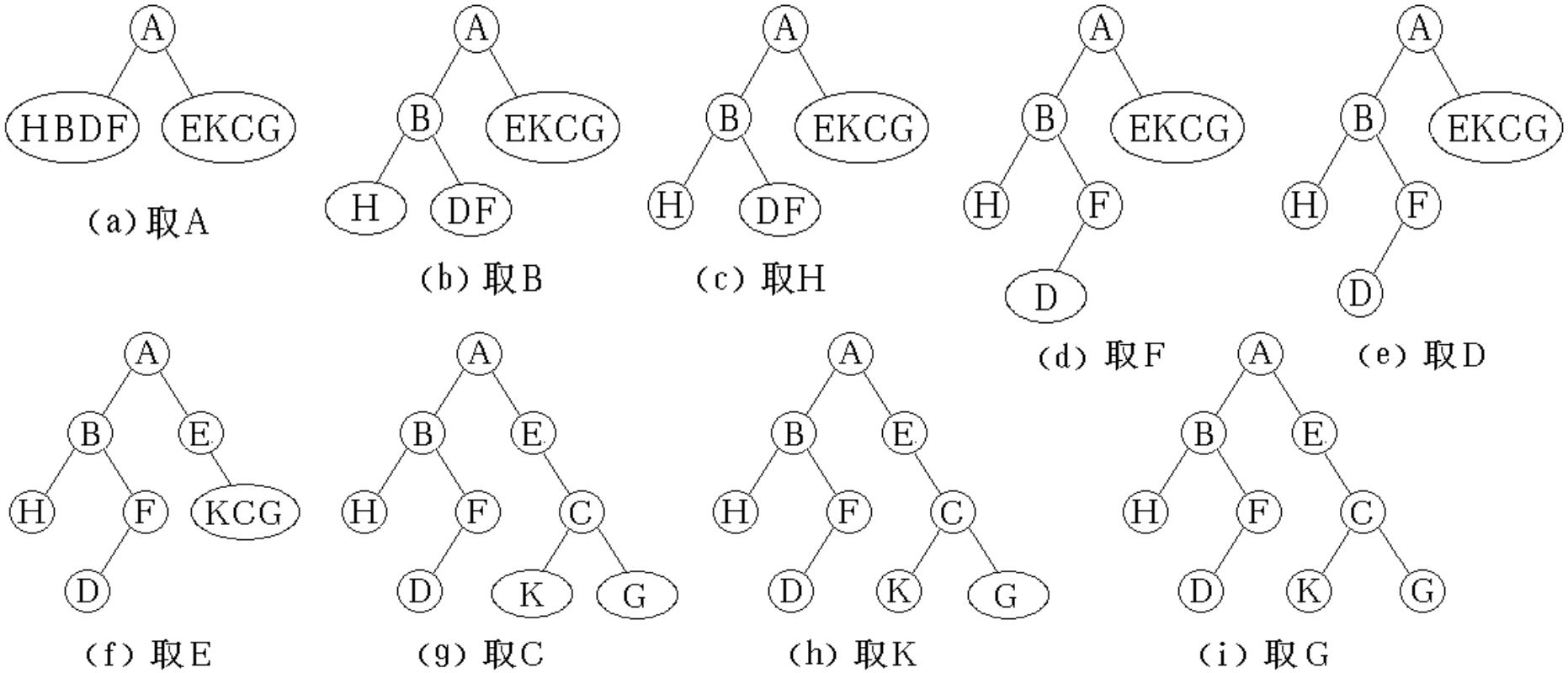
3.2 二叉树(Cont.)

```
BTREE  Copy( BTREE  oldtree )
{  //二叉树的复制
   BTREE   temp ;
   if ( oldtree != NULL ) {
        temp = new  Node ;
        temp -> data = oldtree->data ;
        temp -> lchild = Copy( oldtree->lchild ) ;
        temp -> rchild = Copy( oldtree->rchild ) ;
        return ( temp );
   }
   return ( NULL ) ;
}  /* Copy */
```

3.2 二叉树(Cont.)

二叉树的计数

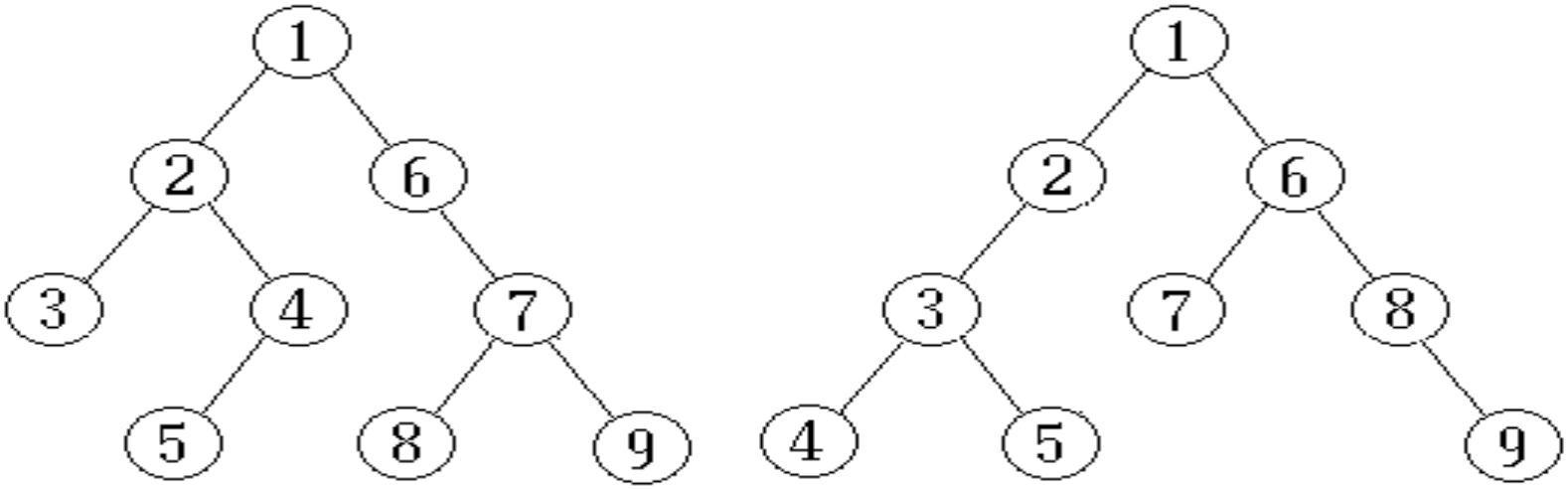
- 由二叉树的先序序列和中序序列可唯一地确定一棵二叉树。例, 先序序列 {ABHFDECKG} 和中序序列 {HBDFAEKCG}, 构造二叉树过程如下：



3.2 二叉树(Cont.)

二叉树的计数

- 如果先序序列固定不变，给出不同的中序序列，可得到不同的二叉树。



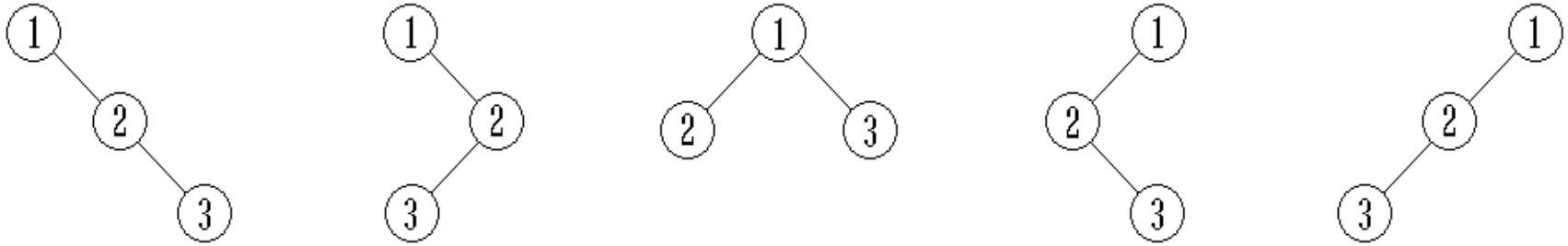
（a）　　（b）

- 问题：设有 n 个数据值（结点），可能构造多少种不同构的二叉树？
- 思路：固定先序排列，找出所有可能的中序排列

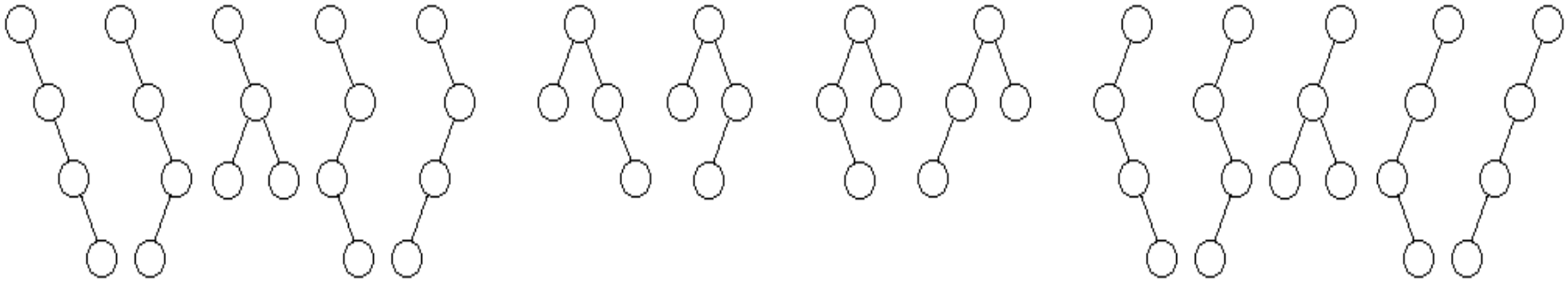
3.2 二叉树(Cont.)

二叉树的计数

- 例如，有 3 个数据 { 1, 2, 3 }，可得 **5 种**不同构的二叉树。它们的**先序排列均为 123**，所有可能中序序列是 123，132，213，231，321。



- 具有4个结点的不同构的二叉树，**14**种，对应14个中序排列

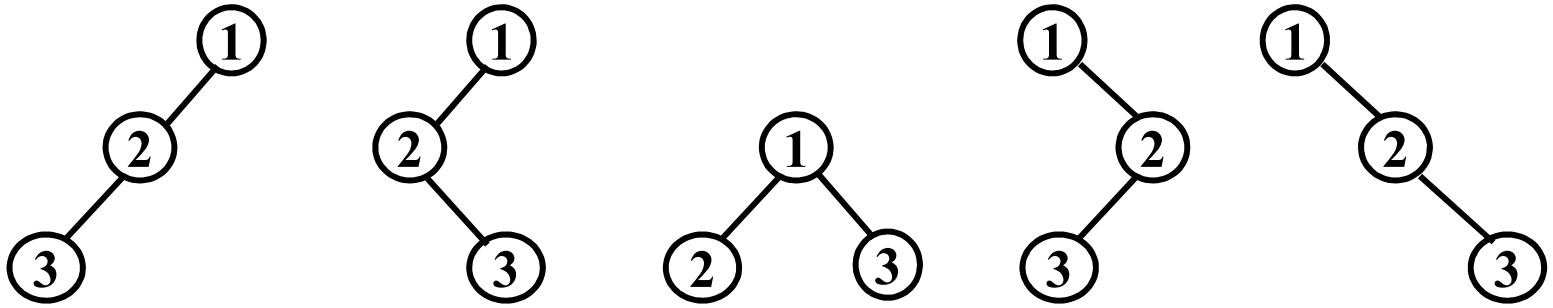


(a) $i=0$　(b) $i=1$　(c) $i=2$　(d) $i=3$

3.2 二叉树(Cont.)

二叉树的计数

- 3个结点二叉树有5种不同形态(同构)



- 中序遍历序列（5个）
  - 3,2,1　　2,3,1　　2,1,3　　1,3,2　　1,2,3
  - 1,2,3,3,2,1　　1,2,2,3,3,1　　1,2,2,1,3,3　　1,1,2,3,3,2　　1,1,2,2,3,3
- 3个元素按1,2,3的顺序进栈，所有可能出栈顺序也是5种
  - 3,2,1　　2,3,1　　2,1,3　　1,3,2　　1,2,3　　3,1,2
  - 1,2,3,3,2,1　　1,2,2,3,3,1　　1,2,2,1,3,3　　1,1,2,3,3,2　　1,1,2,2,3,3
  - 1,1,1,0,0,0　　1,1,0,1,0,0　　1,1,0,0,1,0　　1,0,1,1,0,0　　1,0,1,0,1,0

3.2 二叉树(Cont.)

二叉树的计数

- n个结点不同构的二叉树与不同的出栈序列数量是否一样多？
  - 递推关系

$$\begin{cases} b_0 = 1 \\ b_n = \sum_{i=0}^{n-1} b_i b_{n-i-1}, n \geq 1 \end{cases}$$

  - Catalan数

$$b_n = \frac{1}{n+1} C_{2n}^{n} = \frac{1}{n+1} \frac{(2n)!}{n! \cdot n!}$$

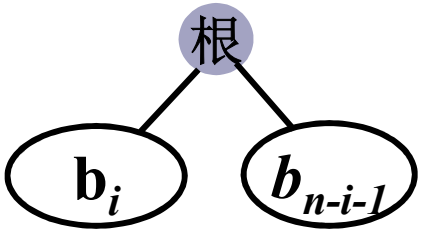
$$= C_{2n}^{n} - C_{2n}^{n+1} \quad for\ n \geq 1$$

  - 前几个卡特兰数：规定$C_0 = 1$，而
    - $C_1=1$，$C_2=2$，$C_3=5$，$C_4=14$，$C_5=42$，
    - $C_6=132$，$C_7=429$，$C_8=1430$，$C_9=4862$，$C_{10}=16796$

3.2 二叉树(Cont.)

二叉树的计数

- 给定n个结点，能构成多少种形状不同的二叉树？
- 给定n个结点二叉树的先序序列,有多少种不同的中序序列？
- 一个进栈序列为1,2,3,…,n，问有多少个不同的出栈序列？
- 分析：设n个结点的二叉树不同构的有b_n个，则



$$\begin{cases} b_0 = 1 \\ b_n = \sum_{i=0}^{n-1} b_i b_{n-i-1}, n \geq 1 \end{cases}$$

- $b_0 = 1$
- $b_n = b_0 * b_{n-1} + b_1 * b_{n-2} + \cdots + b_{n-1} * b_0$

$$= \sum_{i=0}^{n-1} b_i b_{n-i-1}, n \geq 1$$

3.2 二叉树(Cont.)

二叉树的计数

- 一个进栈序列为1,2,3,…,n，问有多少个不同的出栈序列？
- 分析：进栈序列为1,2,3,…,n，设出栈序列总数为b_n。假设最后一个出栈的数为i，则1,2,...,i-1必定在i入栈之前出栈了，否则i就不是最后一个出栈的数。
- 对给定的i，由乘法原理，合法出栈总数为1,2,...,i-1的出栈序列总数b_{i-1}乘以i+1,i+2,...,n的出栈序列总数b_{n-i}，即$b_{i-1}\ b_{n-i}$。
- 由于i的取值可以从1到n，所以再根据加法原理，将i取不同值的序列总数相加，得到的总序列总数为：
- $b_n = b_0 * b_{n-1} + b_1 * b_{n-2} + \cdots + b_{n-1} * b_0$
  $= \sum_{i=0}^{n-1} b_i b_{n-i-1}, n \geq 1$
- 令$b_0 = 1$

$$\begin{cases} b_0 = 1 \\ b_n = \sum_{i=0}^{n-1} b_i b_{n-i-1}, n \geq 1 \end{cases}$$

3.2 二叉树(Cont.)

二叉树的计数问题的生成函数法

$$\begin{cases} b_0 = 1 \\ b_n = \sum_{i=0}^{n-1} b_i b_{n-i-1}, n \geq 1 \end{cases}$$

$$B(x) = b_0 + b_1 x + b_2 x^2 + \ldots + b_n x^n + \ldots = \sum_{i \geq 0} b_i x^i,$$

$$B^2(x) = b_0 b_0 + (b_0 b_1 + b_1 b_0)x + (b_0 b_2 + b_1 b_1 + b_2 b_0)x^2 + \ldots = \sum_{p \geq 0} (\sum_{i=0}^{p} b_i b_{p-i}) x^p = \sum_{p \geq 0} b_{p+1} x^p$$

于是, $xB^2(x) = B(x) - 1$,注意: $B(0) = b_0 = 1$, 解得

$$B(x) = \frac{1 - \sqrt{1 - 4x}}{2x},$$ 利用扩展二项式定理将 $(1-4x)^{1/2}$ 展开, 则

$$B(x) = \frac{1}{2x}\left[1 - \sum_{n \geq 0} \binom{1/2}{n} (-4x)^n\right] = \sum_{m \geq 0} \binom{1/2}{m+1} (-1)^m 2^{2m+1} x^m$$

$$= 1 + x + 2x^2 + 5x^3 + 14x^4 + 42x^5 + \cdots$$

在 $B(x)$ 中，x^n 的系数就是 b_n，即

$$b_n = \binom{1/2}{n+1} (-1)^n 2^{2n+1} = \frac{\frac{1}{2}(\frac{1}{2} - 1)(\frac{1}{2} - 2)\ldots(\frac{1}{2} - n)}{(n+1)!} (-1)^n 2^{2n+1}$$

$$b_n = \frac{1}{n+1}\binom{2n}{n} = \binom{2n}{n} - \binom{2n}{n+1} \qquad b_n = \frac{1}{n+1} C_{2n}^n = C_{2n}^n - C_{2n}^{n+1}$$

3.2 二叉树(Cont.)

二叉树的计数

- 一个进栈序列为1,2,3,…,n，问有多少个不同的出栈序列？

- 组合分析法

- 对于每一个数来说，必须进栈一次、出栈一次。将入栈用1表示，出栈用0表示，则n个数的出入栈为n个1和n个0组成的2n位二进制数。而且从左往右1个数一定不能比0的个数少(否则某个时刻出栈次数多于入栈次数)。
- 因此，出栈序列的总数目=由左而右扫描由n个1和n个0组成的2n位二进制数，1的个数不小于0的个数的方案数。
- 在2n位二进制数中填入n个1的方案数为C(2n,n),不填1的其余n位自动填0。
- 但要从中减去不符合要求（由左而右扫描，0的个数大于1的个数）的方案数，即为所求。

3.2 二叉树(Cont.)

二叉树的计数

- 一个进栈序列为1,2,3,…,n，问有多少个不同的出栈序列？
- 组合分析法
- 不符合要求的2n位二进制数（非法的进出栈序列）与一个由n+1个0和n-1个1组成的2n位二进制数一一对应。
- 首先，不符合要求的特征是由左而右扫描时，必然在某一奇数位2m+1位上，首先出现m+1个0和m个1的，此后的2(n-m)-1位上有n-m个1和n-m-1个0。若把后面这2(n-m)-1位上的0和1互换，使之成为n-m个0和n-m-1个1，结果得一个由n+1个0和n-1个1组成的2n位数。
- 即一个不合要求的2n位数对应于一个由n+1个0和n-1个1组成的排列。

3.2 二叉树(Cont.)

二叉树的计数

- 一个进栈序列为1,2,3,…,n，问有多少个不同的出栈序列？
- 组合分析法
- 反过来，任何一个由n+1个0和n-1个1组成的2n位二进制数，由于0的个数多2个，2n为偶数，故必在某一个奇数位上出现0的个数超过1的个数。同样在后面部分0和1互换，使之成为由n个0和n个1组成的2n位数，即
- n+1个0和n-1个1组成的2n位数必对应一个不符合要求的数
- 因而不合要求的2n位数与n+1个0，n-1个1组成的排列一一对应。
- 显然，不符合要求的方案数为C(2n,n+1)。由此得出合法的出栈序列的总数=C(2n,n)-C(2n,n+1)=C(2n,n)/(n+1)。

3.2 二叉树(Cont.)

二叉树的计数

- 应用
  - 一个进栈序列为1,2,3,…,n，问有多少个不同的出栈序列？
  - 甲乙两人比赛乒乓球，最后结果为11：11，问比赛过程中甲始终领先乙的计分情形的种数？
  - 饭后，姐姐洗碗，妹妹把姐姐洗过的碗一个一个地放进碗橱摞成一摞。共有n个不同的碗，洗前也是摞成一摞的，也许因为小妹贪玩而使碗拿进碗橱不及时，姐姐则把洗过的碗摞在旁边，问：小妹摞起的碗有多少种可能的方式？
  - 一个有n个1和n个-1组成的字串，且前k个数的和均不小于0，那这种字串的总数为多少？
  - 在圆上选择2n个点,将这些点成对连接起来，使得所得到的n条线段不相交的方法数？

3.2 二叉树(Cont.)

二叉树的计数

- 应用
  - 给定n个结点，能构成多少种形状不同的二叉树？
  - 凸多边形的三角划问题:将一个凸多边形区域分成三角形区域的方法数？
  - 一个人在住所以北n个街区和以东n个街区处工作。每天走2n个街区去上班。如果从不穿越（但可以碰到）从家到办公室的对角线，那么有多少条可能的道路？
  - 有2n个人排成一行进入剧场。入场费5元。其中只有n个人有一张5元钞票，另外n人只有10元钞票，剧院无其它钞票，问有多少中方法使得只要有10元的人买票，售票处就有5元的钞票找零？
  - ……

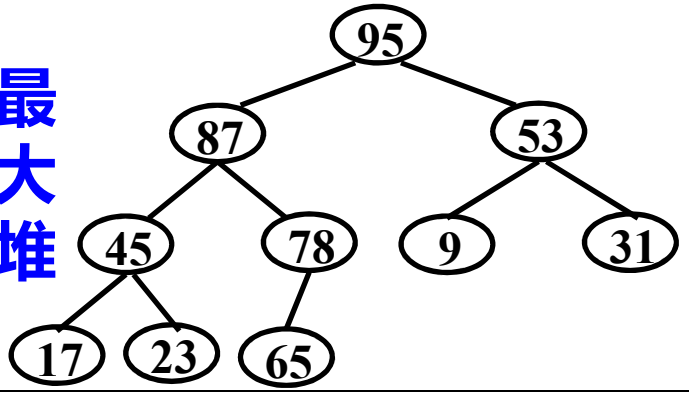
3.3 堆（Heap）

一、ADT堆

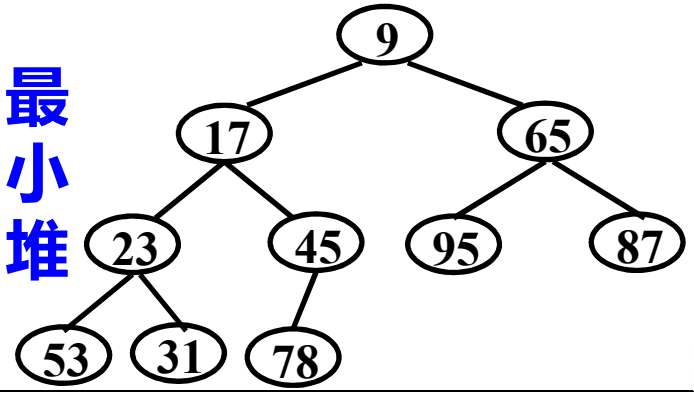
堆的定义

- 如果一棵完全二叉树的任意一个非终端结点的元素都不小于其左儿子结点和右儿子结点（如果有的话）的元素，则称此完全二叉树为最大堆（大顶堆、大根堆）。
- 如果一棵完全二叉树的任意一个非终端结点的元素都不大于其左儿子结点和右儿子结点（如果有的话）的元素，则称此完全二叉树为最小堆（小顶堆、小根堆）。
- 特点：根结点的元素是最大（小）的。。。。。。

最大堆



最小堆



3.3 堆（Heap）

一、ADT堆

ADT堆的基本操作

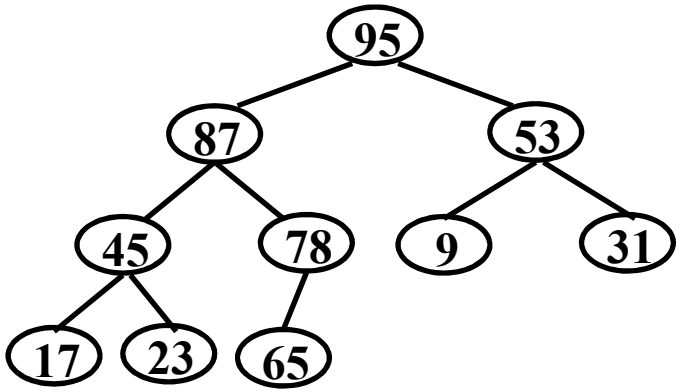
- MaxHeap(maxsize)：创建一个空堆，最多可容纳maxsize个元素
- HeapFull(heap，n)：判断堆是否为满。若堆中元素个数n达到最大容量maxsize，则返回TRUE，否则返回FALSE；
- Insert(heap，item，n)：插入一个元素。若堆不满，则将item 插入heap，否则不能插入；
- HeapEmpty(heap，n)：判断堆是否为空。若堆中元素个数n大于0，则返回TRUE，否则返回FALSE；
- DeleteMax(heap，n)：删除最大元素。若堆为不空，则返回堆中最大元素，并将其删除，否则，返回一个特定值，表明不能进行删除。

3.3 堆（Heap）

二、ADT堆的实现—最大堆的实现

- ADT堆的存储结构
  - 由于堆是一个完全二叉树，所以可以采用完全二叉树的数组表示。
- 堆的存储结构定义

```
#define Maxsize  200
typedef struct {
    int key;
    /* other fields*/
} ElemType;
typedef struct {
    ElemType data[Maxsize];
    int n;
} HEAP;
```



	95	87	53	45	78	09	31	17	23	65		
0	1	2	3	4	5	6	7	8	9	10		

3.3 堆（Heap）

二、ADT堆的实现—最大堆的实现

- 堆的基本操作的实现
  - ①创建空堆

```
void MaxHeap（HEAP heap）
{
        heap.n=0;
}
```

  - ②判空

```
bool HeapEmpty（HEAP heap）
{
        return (!heap.n);
}
```

  - ③判满

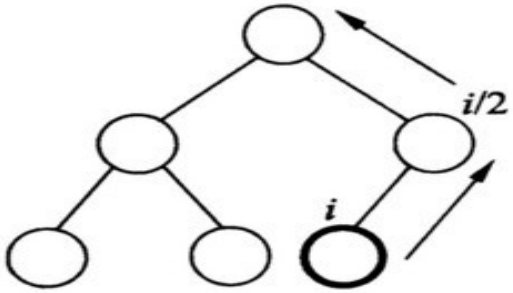
```
bool HeapFull（HEAP heap）
{
        return (heap.n==MaxSize);
}
```

```
#define Maxsize  200
typedef struct {
    int key;
    /* other fields*/
} ElemType;
typedef struct {
    ElemType data[Maxsize];
    int n;
} HEAP;
```

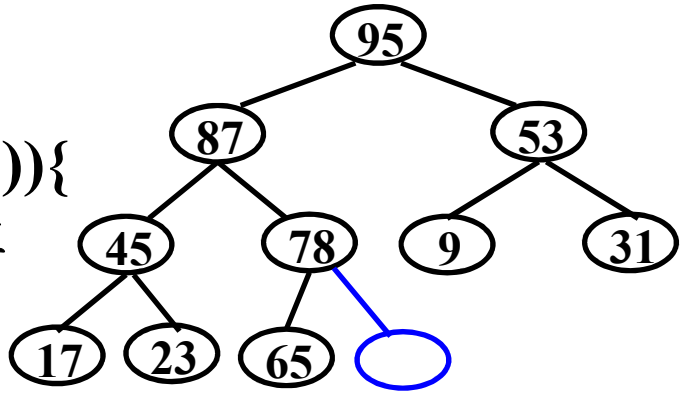
3.3 堆（Heap）

二、ADT堆的实现—最大堆的实现

- 堆的基本操作的实现
  - ④插入




```
void Insert(HEAP& heap，ElemType elem)
{
    int i;
    if (!HeapFull(heap)){
        i=heap.n+1;
        while((i!=1)&&(elem >heap.data[i/2])){
            heap.data[i]=heap.data[i/2];//下推
            i/=2;
        }
    }
    heap.data[i]= elem;
}//时间复杂性O（logn）
```

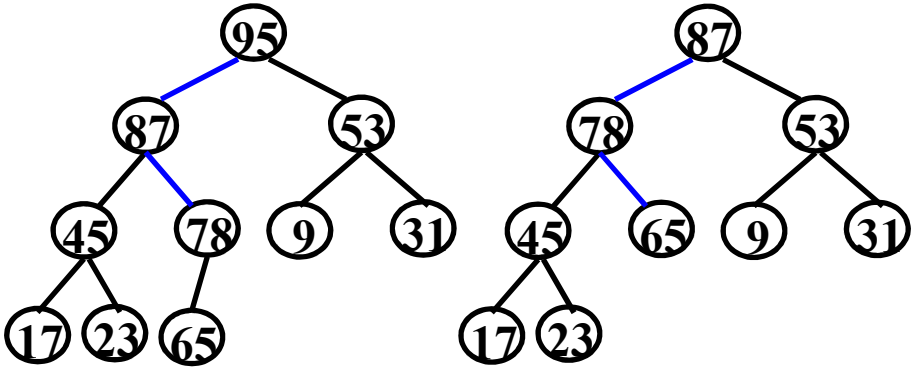


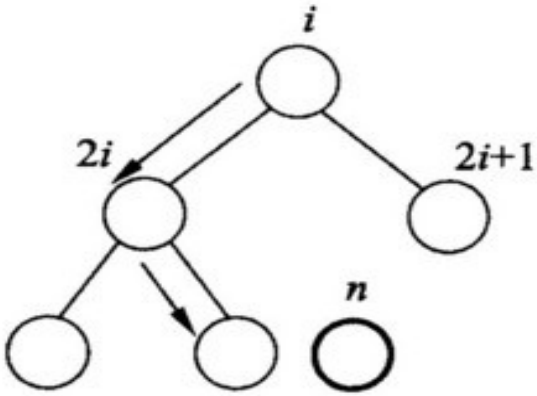
	95	87	53	45	78	09	31	17	23	65				
0	1	2	3	4	5	6	7	8	9	10				

3.3 堆（Heap）

⑤删除最大元素

```
ElemType DeleteMax( HEAP &heap )
{   int parent=1, child=2;
    ElemType elem, tmp;
    if (!HeapEmpty(heap)){
        elem=heap.data[1];
        tmp=heap.data[heap.n--];
        while (child<=heap.n){
            if ((child< heap.n)&&
                (heap.data [child]<heap.data [child+1]) )
                child++; //找最大子结点（左右儿子的大者）
            if (tmp>= heap.data[child])  break;
            heap.data[parent]= heap.data[child];
            parent=child;
            child*=2;
        }
        heap[parent]=tmp;
        return elem;
    }
} //时间复杂性O（logn）
```





3.3 堆（Heap）

三、堆与优先级队列（Priority Queue）

- 优先队列在计算机操作系统任务调度以及日常工作安排等领域具有十分广泛的应用。
- 在优先队列中，被删除的是优先级最高的元素，而在任何时刻可插入任意优先级的元素。支持这两种操作的数据结构称为最大优先队列。
- 如果被删除的是优先级最低的元素，则相应的数据结构称为最小优先队列。

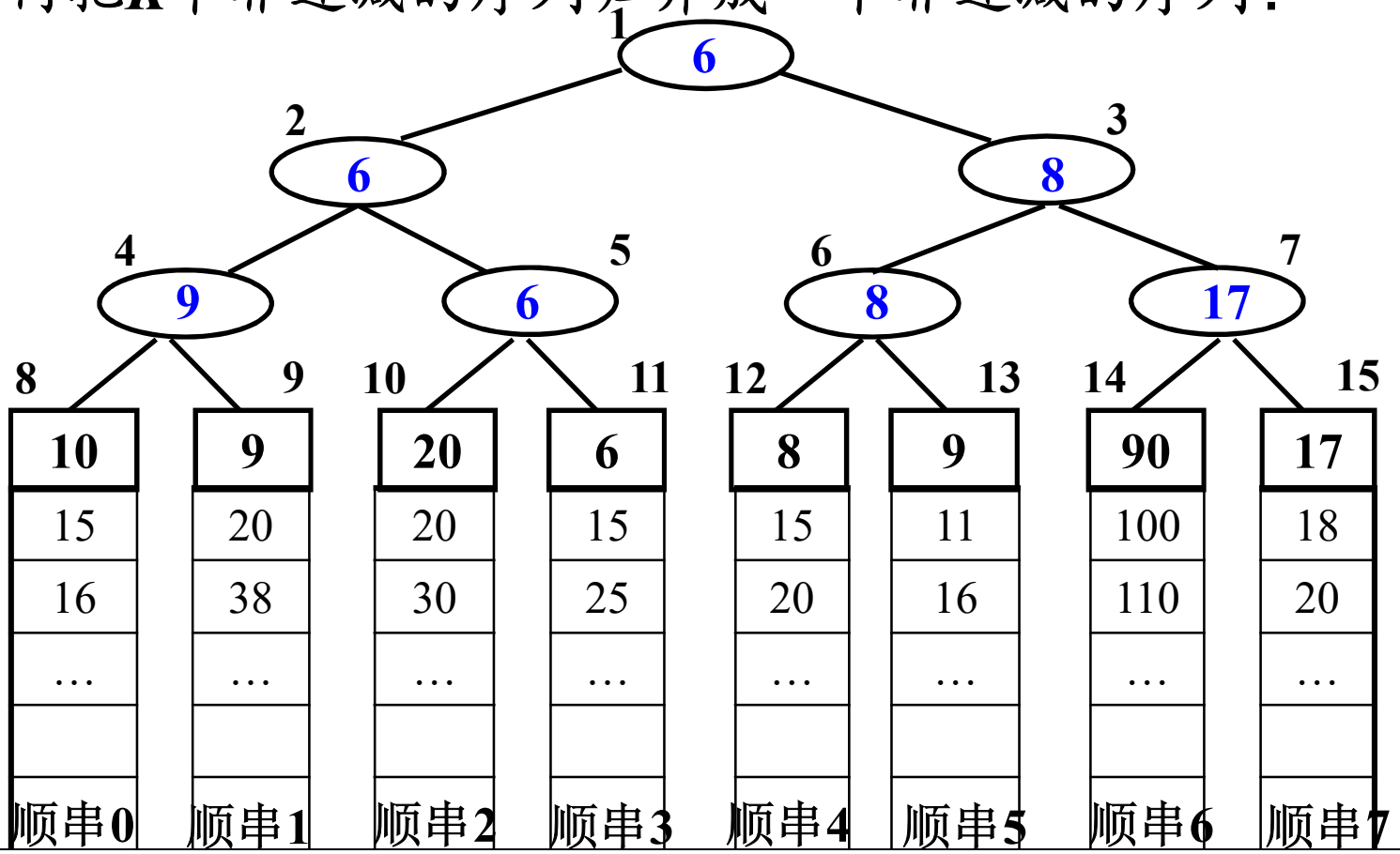
- **优先级队列的各种存储表示和代价分析**

存储表示	插入操作	删除操作
无序数组	Θ（1）	Θ（n）
无序单向链表	Θ（1）	Θ（n）
有序数组	O（n）	Θ（1）
有序单向链表	O（n）	Θ（1）
最大堆/最小堆	O（$\log_2 n$）	O（$\log_2 n$）

3.4 选择树（Selection Tree）

一、背景

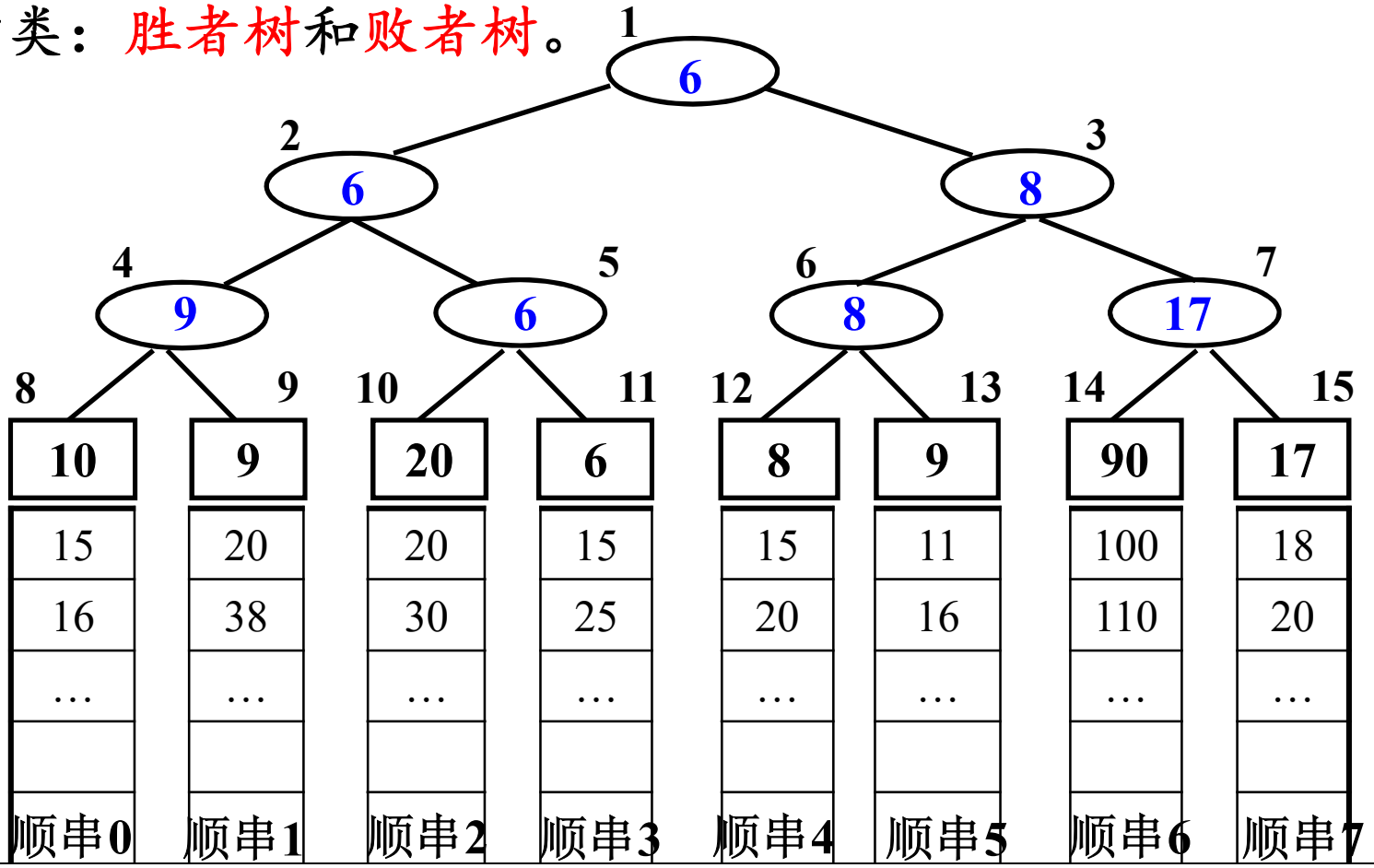
- 如何从*n*个元素中选择最小的，进而对*n*个元素排序？
- 如何把*K*个非递减的序列归并成一个非递减的序列？



3.4 选择树（Selection Tree）

二、选择树（也称Tournament Tree）

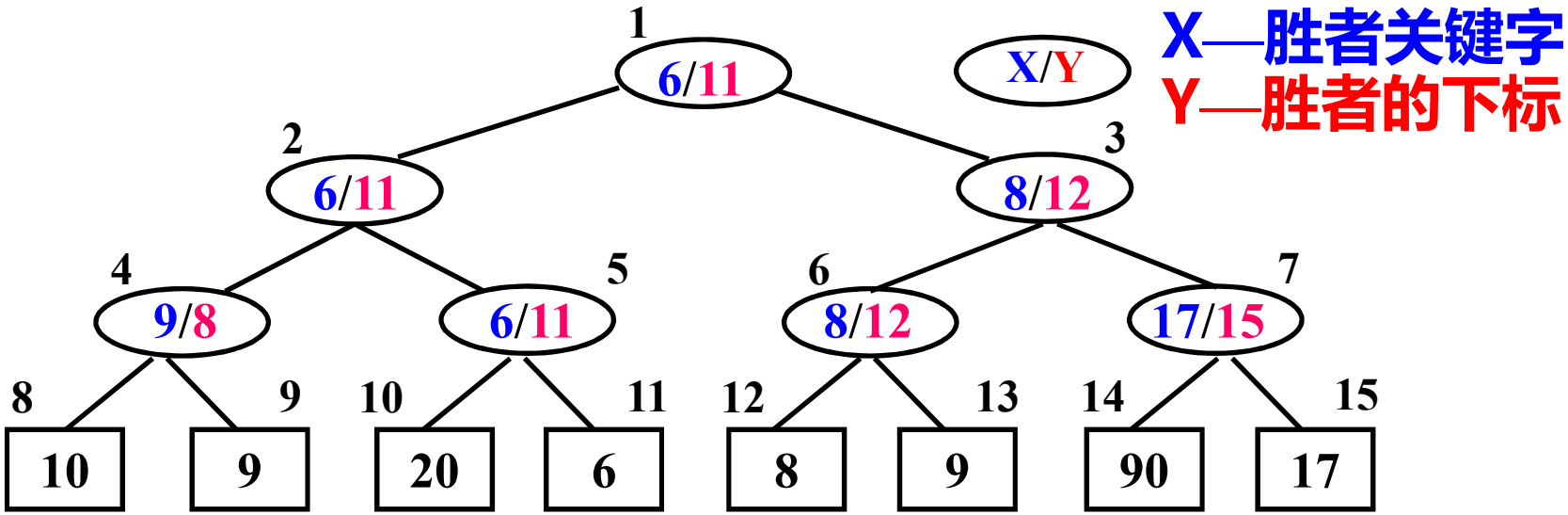
- 选择树就是能够记载上一次比较获得的知识的完全二叉树
- 种类：胜者树和败者树。



3.4 选择树（Selection Tree）

三、胜者树(Winner Tree)

- 胜者树

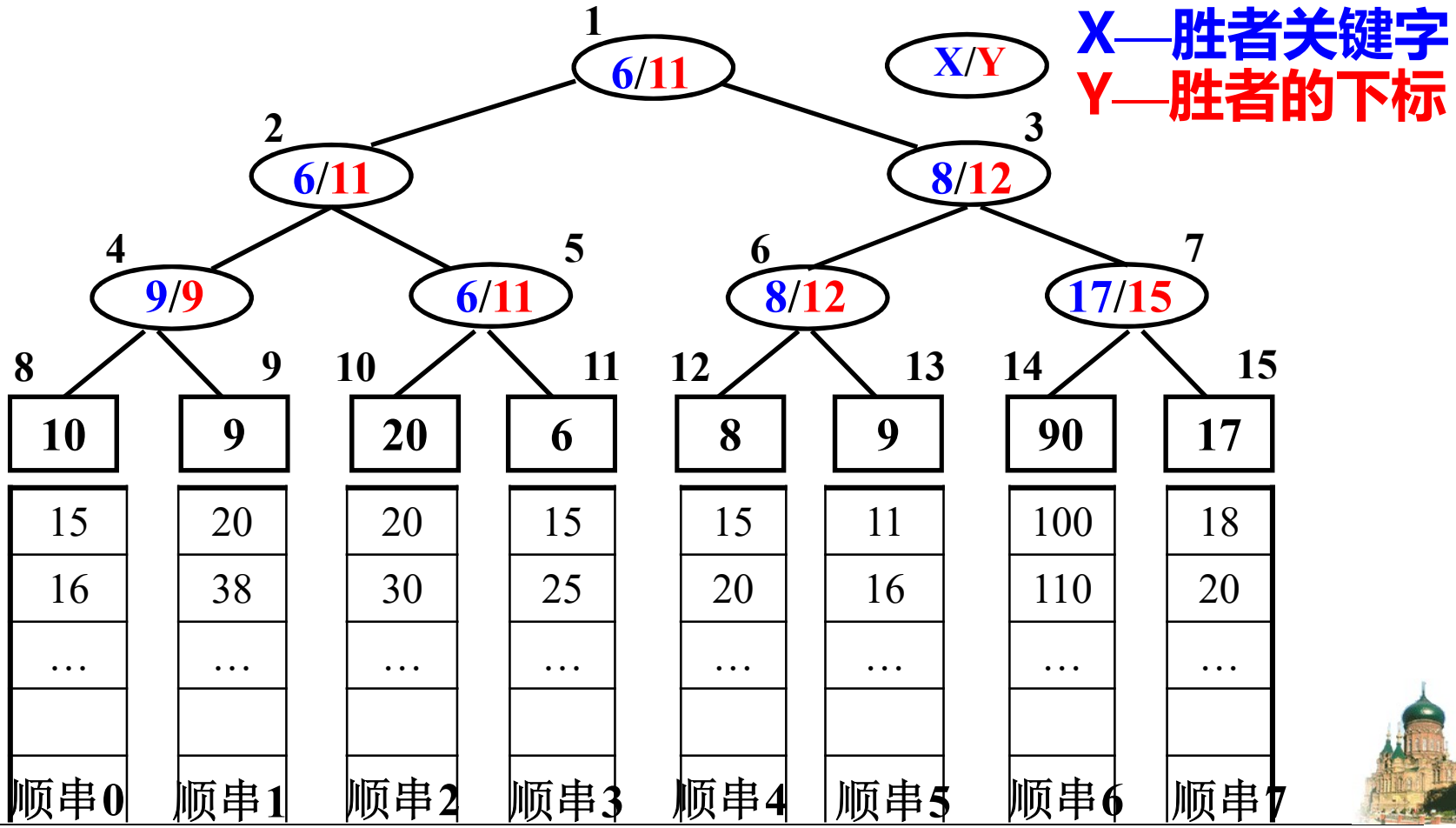


- 具有*n*个外结点和*n*-1个内结点
- 外结点为比赛选手,内结点为一次比赛,每一层为一轮比赛
- 比赛在兄弟结点间进行，胜者保存到父结点中
- 根结点保存最终的胜者

3.4 选择树（Selection Tree）

三、胜者树(Winner Tree)

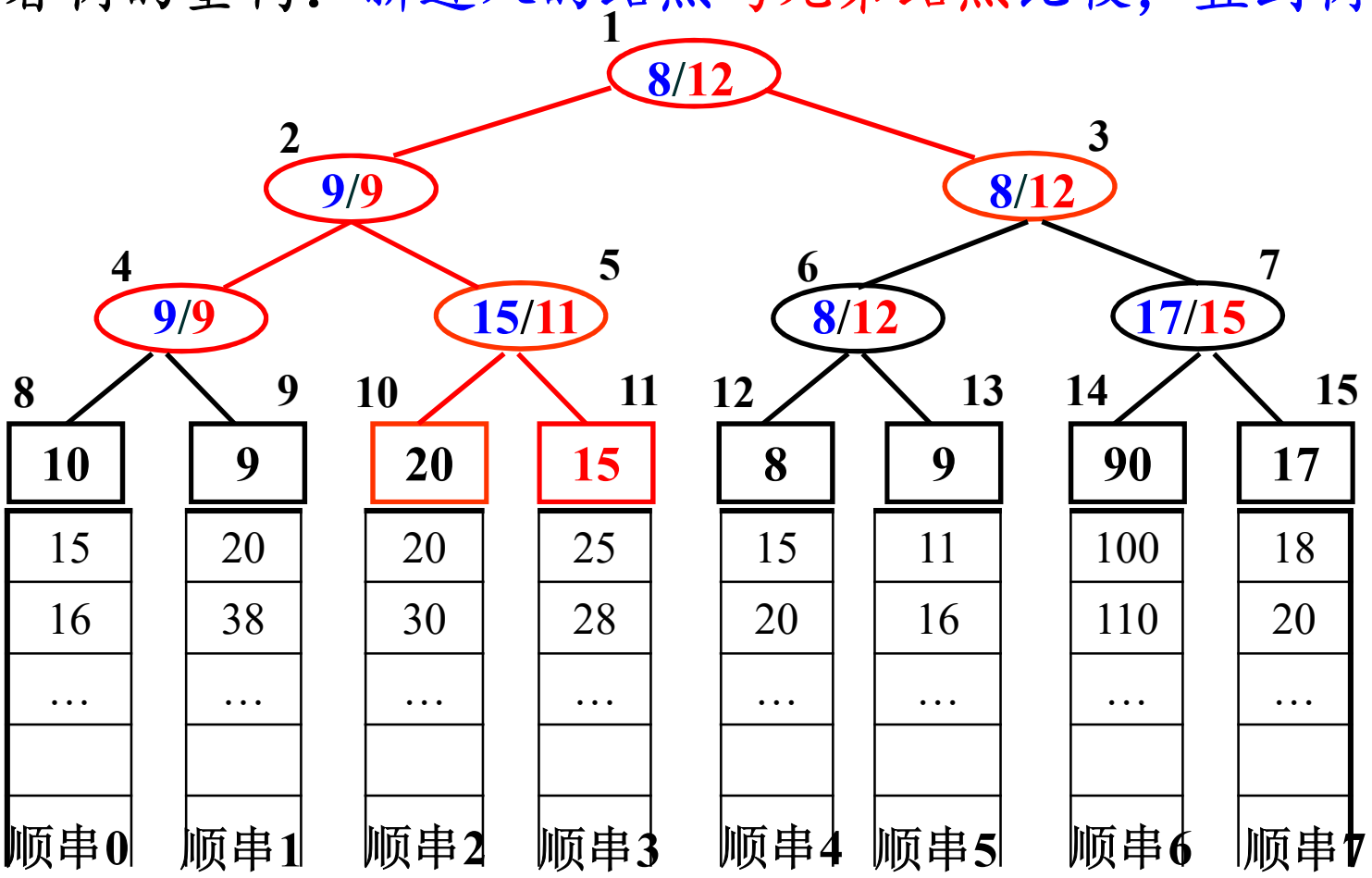
- 胜者树的构建



3.4 选择树（Selection Tree）

三、胜者树(Winner Tree)

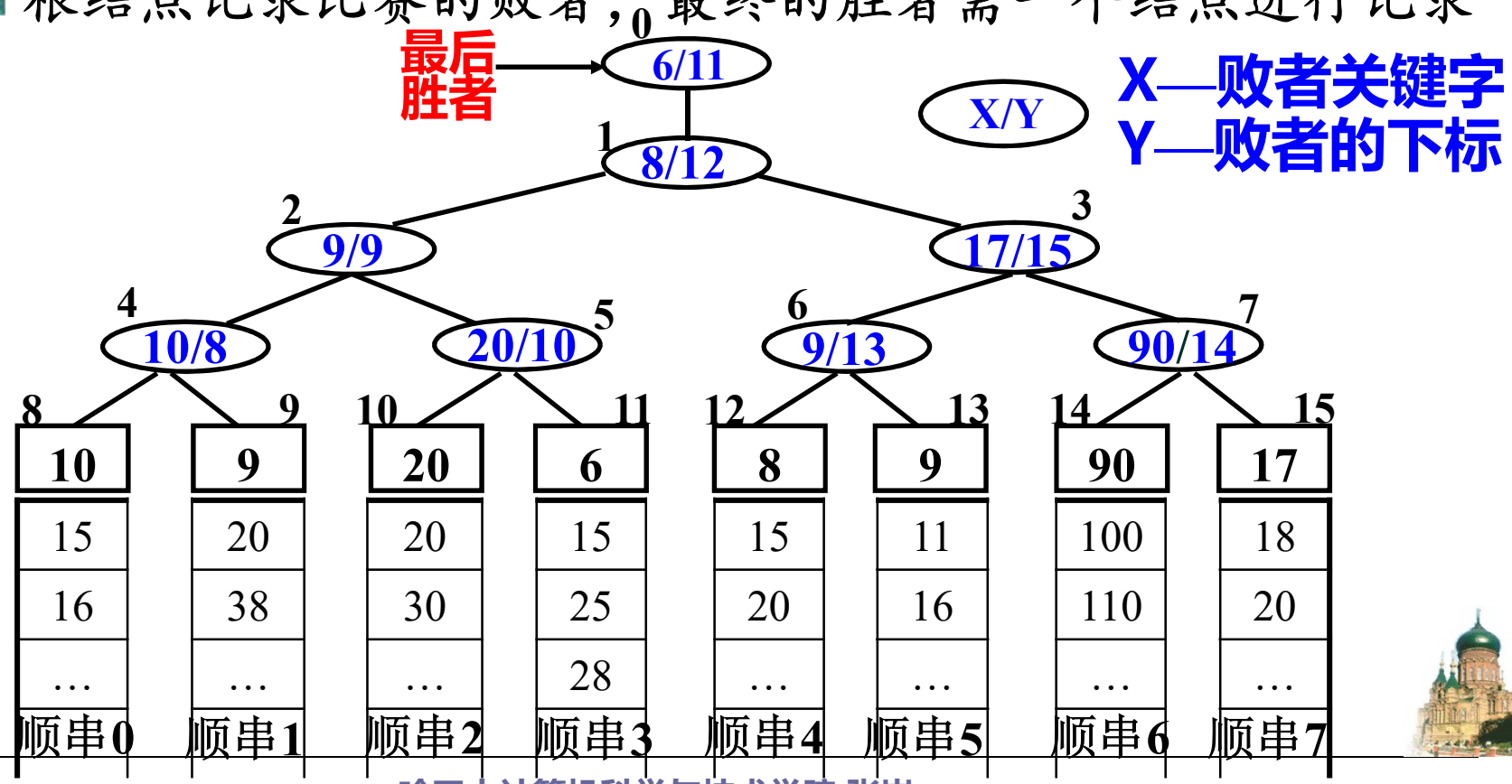
- 胜者树的重构：新进入的结点与兄弟结点比较，直到树根



3.4 选择树（Selection Tree）

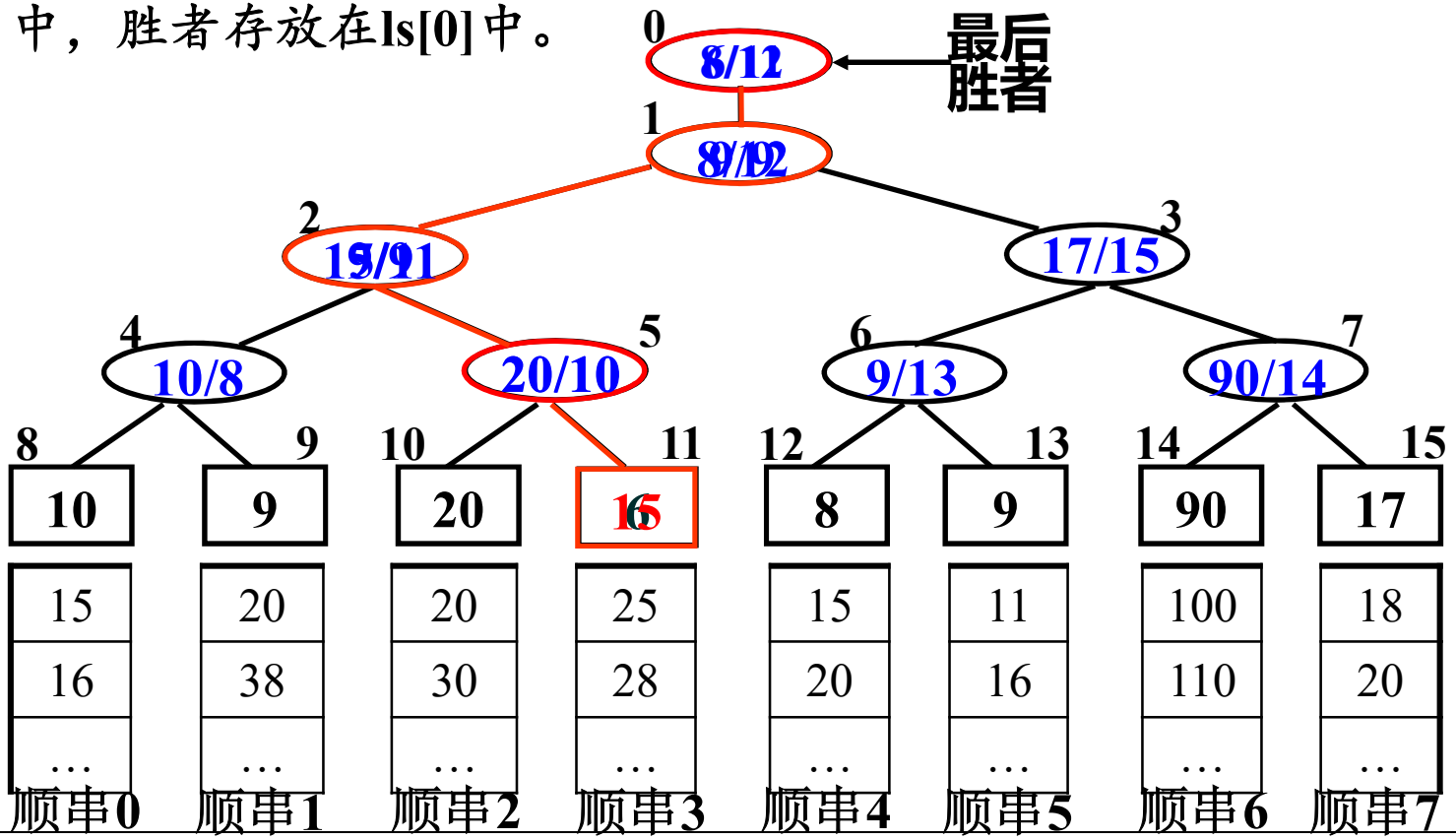
四、败者树(Loser Tree)

- 败者树的构建
  - 内部结点保存败者，胜者参加下一轮比赛
  - 根结点记录比赛的败者，最终的胜者需一个结点进行记录



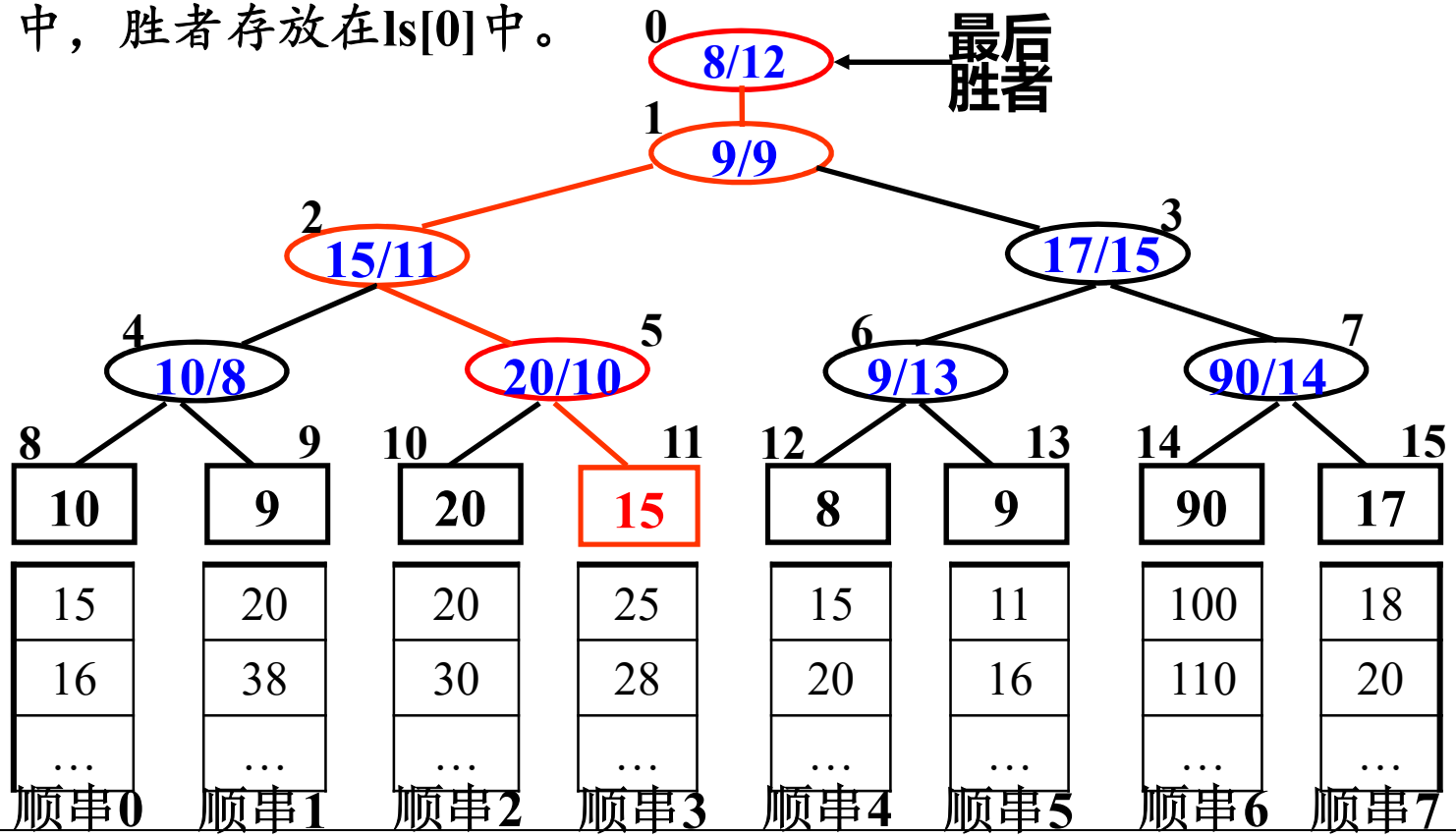
3.4 选择树（Selection Tree）

- 败者树的重构
  - 将新进入的结点与其父结点进行比赛：将败者存入父结点中，胜者再与上一级的父结点比较。
  - 比赛沿着到根的路径不断进行，直到ls[1]处。把败者存放在结点ls[1]中，胜者存放在ls[0]中。



3.4 选择树（Selection Tree）

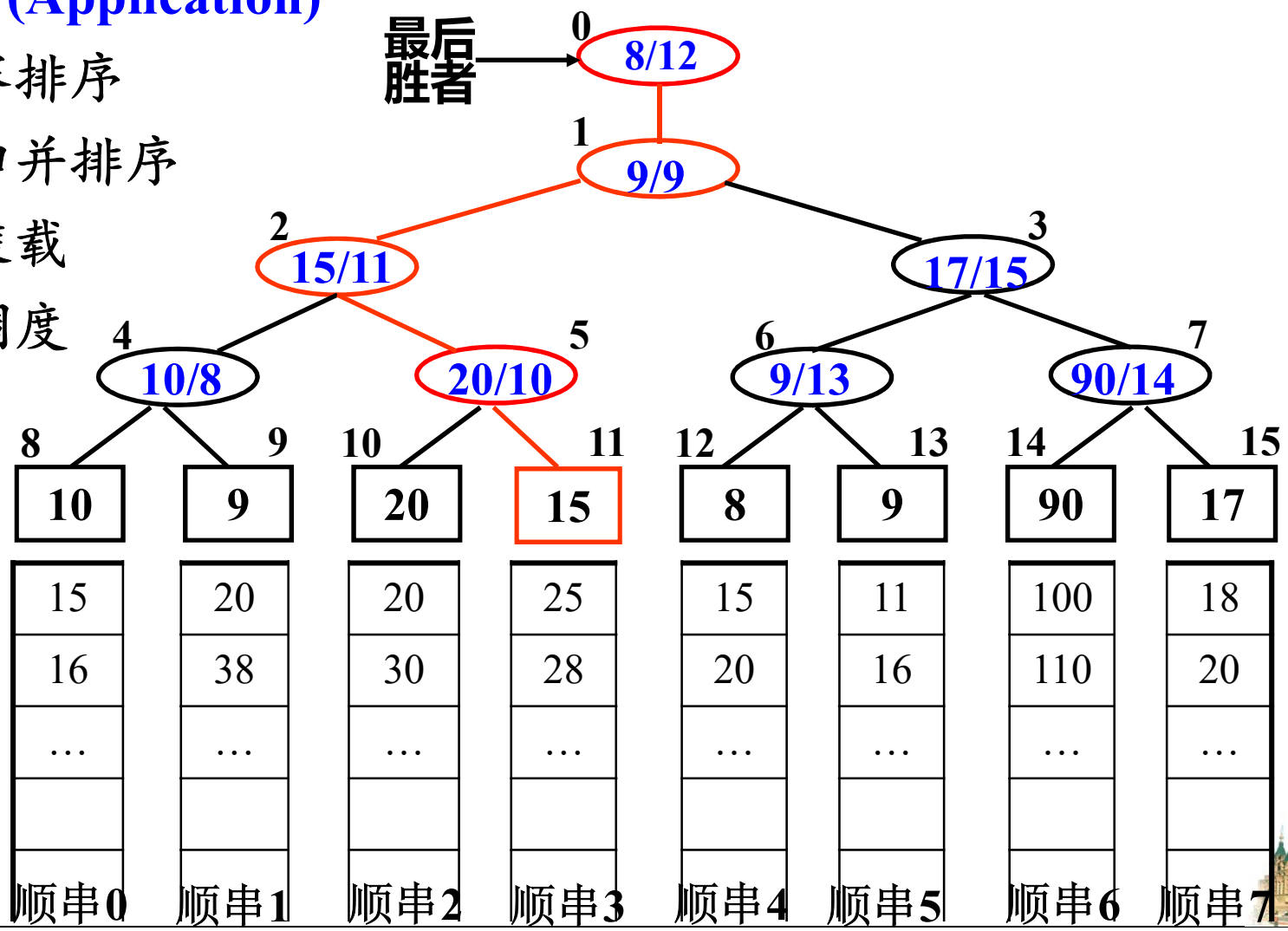
- 败者树的重构
  - 将新进入的结点与其父结点进行比赛：将败者存入父结点中，胜者再与上一级的父结点比较。
  - 比赛沿着到根的路径不断进行，直到ls[1]处。把败者存放在结点ls[1]中，胜者存放在ls[0]中。



3.4 选择树（Selection Tree）

五、应用(Application)

- 锦标赛排序
- 外部归并排序
- 箱子装载
- 机器调度



3.5 树

树的基本操作

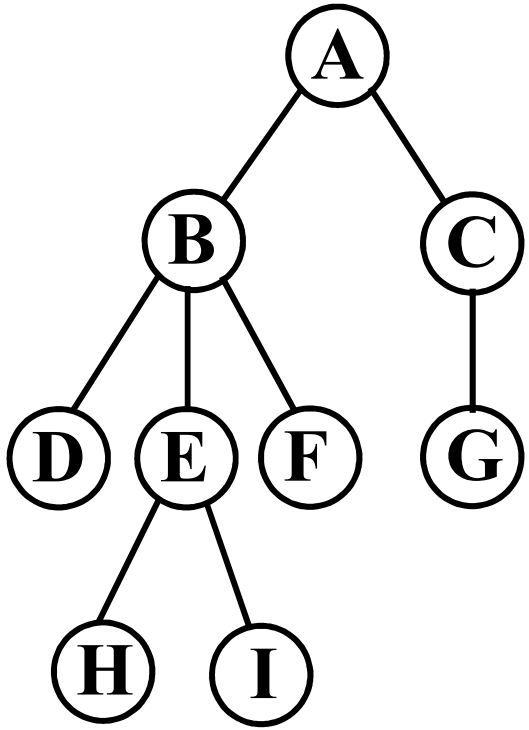
- Parent(n , T) 求结点 n 的双亲
- LeftMostChild(n , T) 返回结点 n 的最左儿子
- RightSibling(n , T) 返回结点 n 的右兄弟
- Data(n , T) 返回结点 n 的信息
- CreateK k (v , T1 , T2 , …… , Tk) , k = 1 , 2 , ……
  - 建立data域值为v的根结点r , 有k株子树T1 , T2 , …… , Tk , 且自左至右排列；返回r。
- Root(T) 返回树T的根结点
- 树的遍历操作
  - 从根结点出发，按照某种次序访问树中所有结点，使得每个结点被访问一次且仅被访问一次。

3.5 树(Cont.)

- 树的遍历操作
  - 树通常有先序（根）遍历、后序（根）遍历和层序（次）遍历三种方式。
- 先序遍历

(1) 访问根结点；
(2) 按照从左到右的顺序先序遍历根结点的每一棵子树

先序遍历序列：*A B D E H I F C G*



- 后序遍历

(1) 按照从左到右的顺序后序遍历根结点的每一棵子树

(2) 访问根结点。

后序遍历序列：*D H I E F B G C A*

3.5 树(Cont.)

- 中序遍历

(1)中序遍历第一棵子树；

(2)访问根结点；

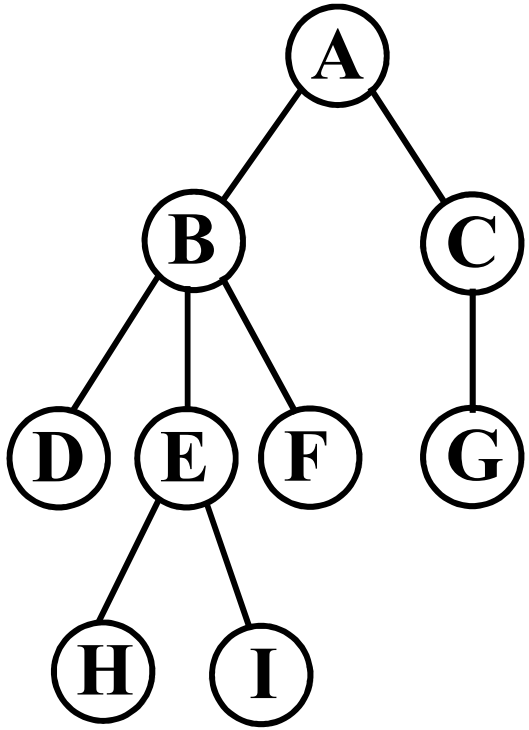
(3)按照从左到右的顺序中序遍历根结点的其他子树

中序遍历序列：*D B H E I F A G C*

- 层序遍历

从树的第一层（即根结点）开始，自上而下逐层遍历，在同一层中，按从左到右的顺序对结点逐个访问。

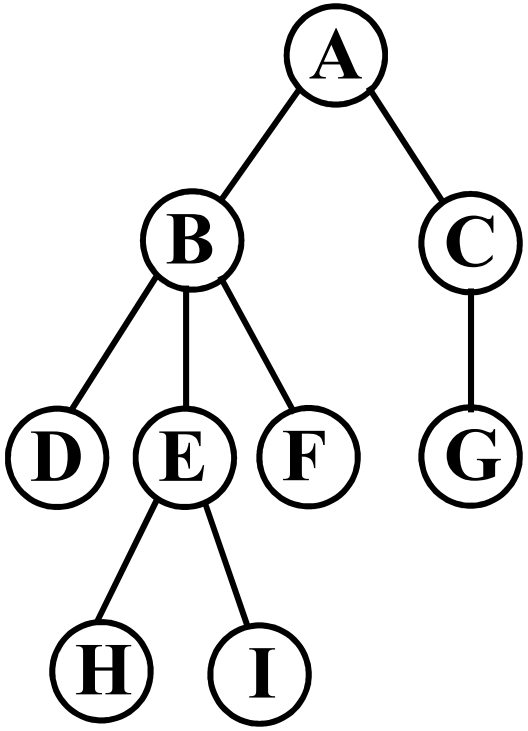
层序遍历序列：*A B C D E F G H I*



3.5 树(Cont.)

- 用树的基本操作写先序遍历的递归算法

```
void  PreOrder(node n, TREE  T )
{   node  c ;
    visit( Data( n ) ) ;
    c = LeftMostChild( n , T ) ;
    while ( c != NULL ) {
        PreOrder( c , T ) ;
        c = RightSibling( c , T ) ;
    }
}
```



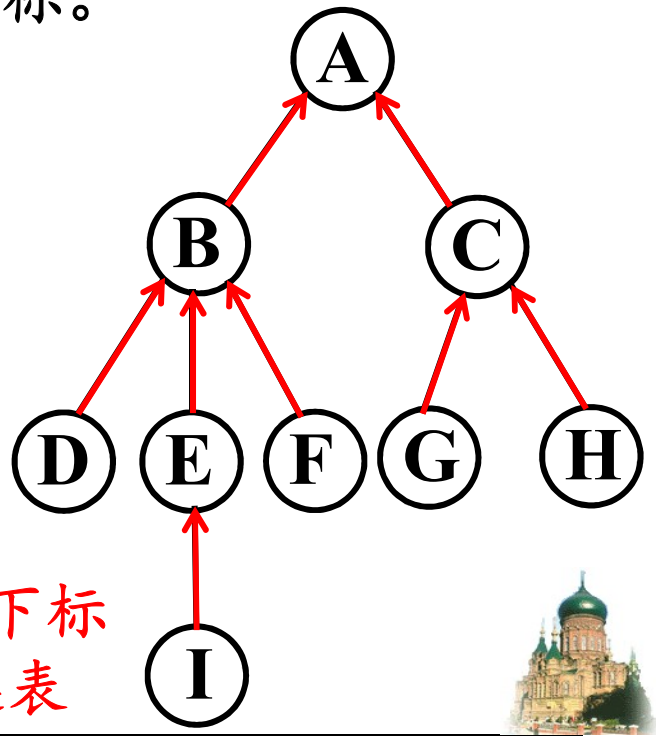
先序遍历树T： PreOrder（Root(T) ， T）

3.5 树(Cont.)

树的存储结构

- 双亲表示法（单链表示、父链表示）
  - 每个结点（根结点除外）都只有唯一的双亲结点
  - 因此，把各个结点（一般按层序）存储在一维数组中，同时记录其唯一双亲结点在数组中的下标。

	1	2	3	4	5	6	7	8	9
data	A	B	C	D	E	F	G	H	I
parent	0	1	1	2	2	2	3	3	5



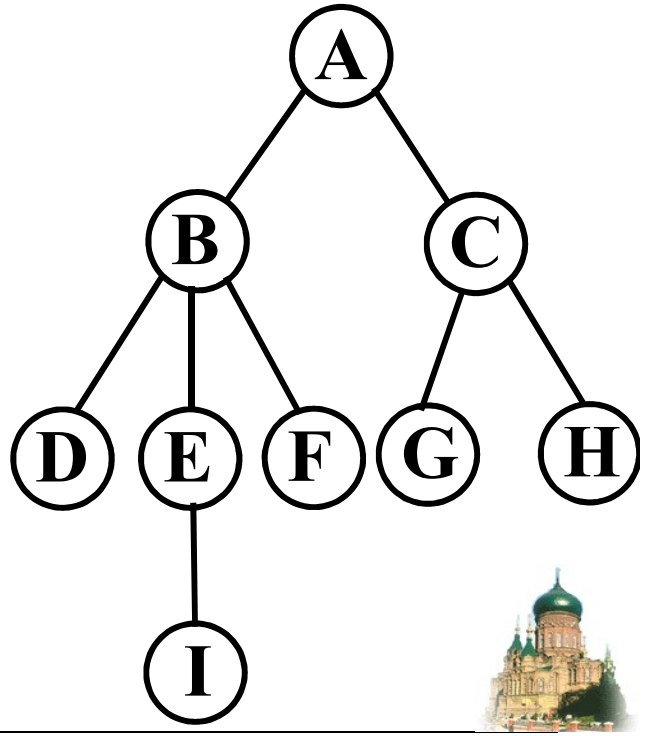
  - 结点结构定义

```
struct node {
    T data;    //数据域
    int parent; //指针域，双亲在数组中的下标
};  //树的双亲表示法实质上是一个静态链表
```

3.5 树(Cont.)

- 双亲表示法（单链表示、父链表示）
  - 存储特点：
    - 每个结点均保存父结点所在的数组单元下标
    - 兄弟结点的编号连续。
  - 如何查找双亲结点和祖先？时间性能？
  - 如何查找孩子结点？时间性能？
  - 如何查找兄弟结点？时间性能？

	1	2	3	4	5	6	7	8	9
data	A	B	C	D	E	F	G	H	I
parent	0	1	1	2	2	2	3	3	5
firstchild	2	4	7	0	9	0	0	0	0
rightsib	0	3	0	5	6	0	8	0	0



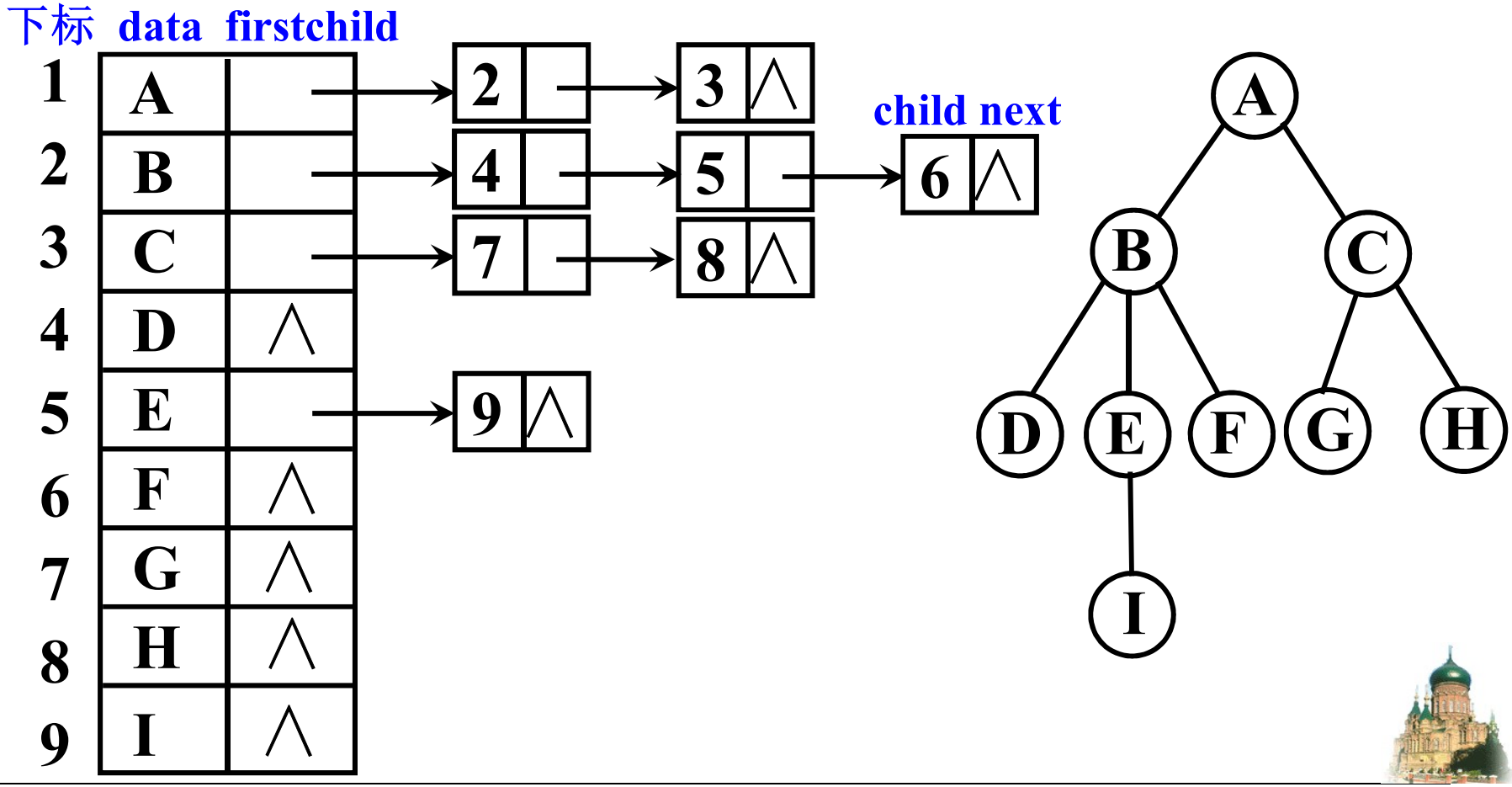
3.5 树(Cont.)

树的存储结构

- 孩子链表表示法（邻接表表示）
  - 把每个结点的孩子结点组成是一个单链表，则*n*个结点共有 *n* 个孩子链表。
  - 再把 每个单链表的表首结点指针，组织成一个顺序表，便于进行查找。
  - 最后，将存放 *n* 个表首结点指针的数组和存放*n*个结点的数组结合起来，构成孩子链表的表头数组。

3.5 树(Cont.)

- 孩子链表表示法（邻接表表示）
  - 示例



3.5 树(Cont.)

- 孩子链表表示法（邻接表表示）
  - 存储结构定义

```
struct  CTNode  {
    int    child ;
    CTNode  *next ;
 };
struct CTBox{
     DataType    data ;
     CTNode  * firstchild ;
} ;
struct  {
     CTBox  nodes[MaxSize] ;
     int   n , r ;
} CTree ;
```

孩子结点

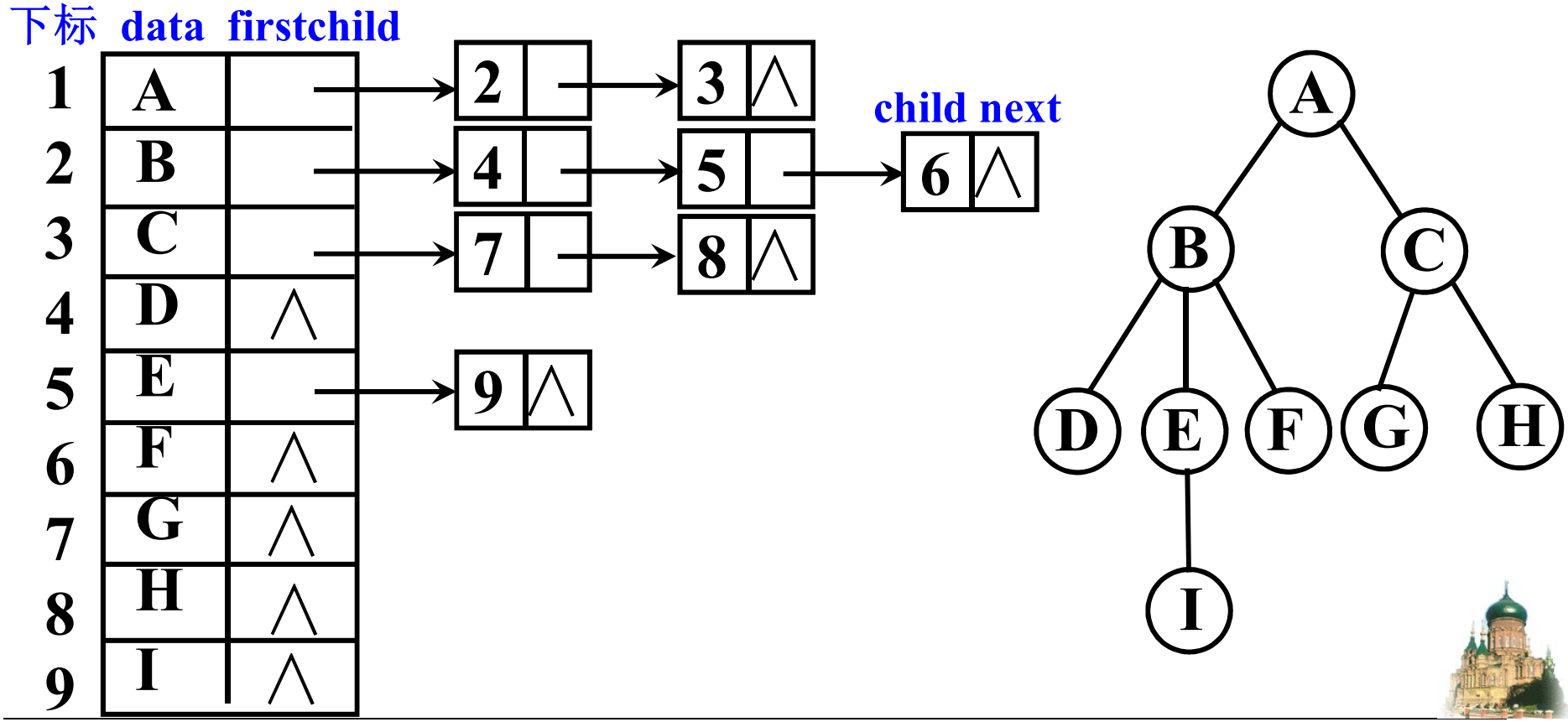
child	next

表头结点

data	firstchild

3.5 树(Cont.)

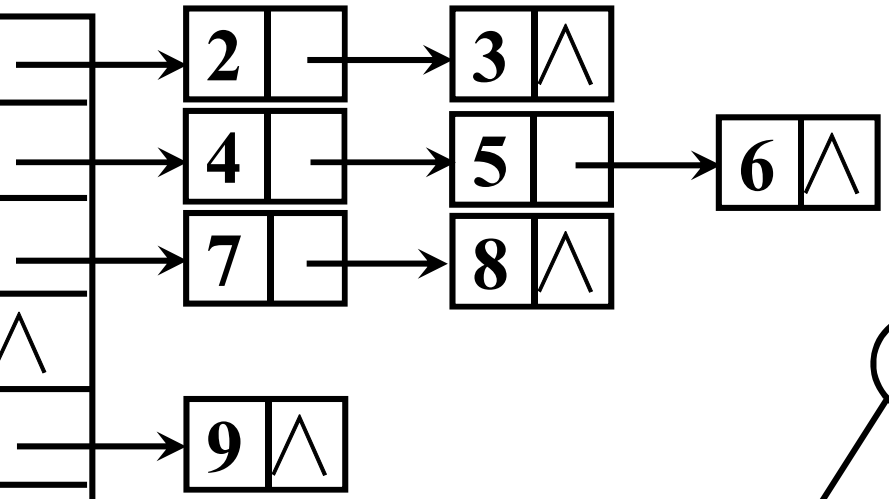
- 孩子链表表示法（邻接表表示）
  - 如何查找孩子结点？时间性能？
  - 如何查找双亲结点？时间性能？

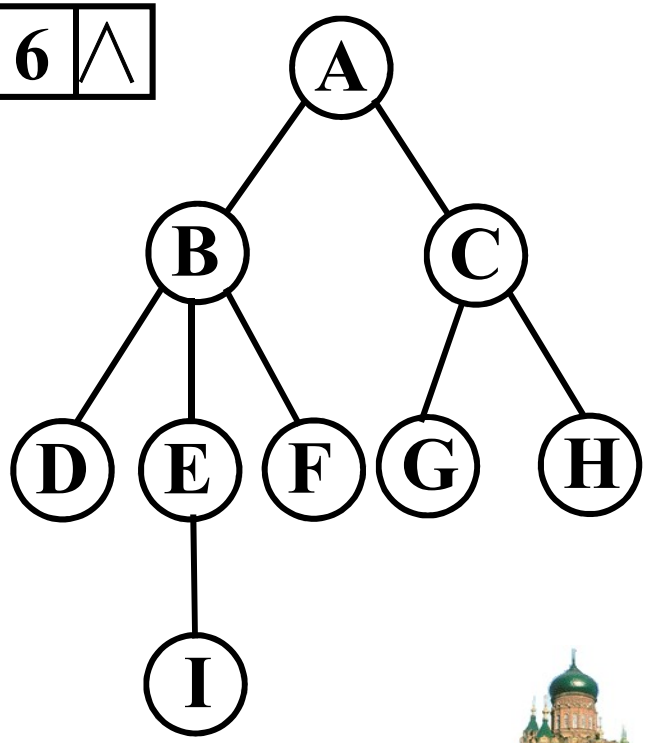


3.5 树(Cont.)

- 双亲孩子表示法

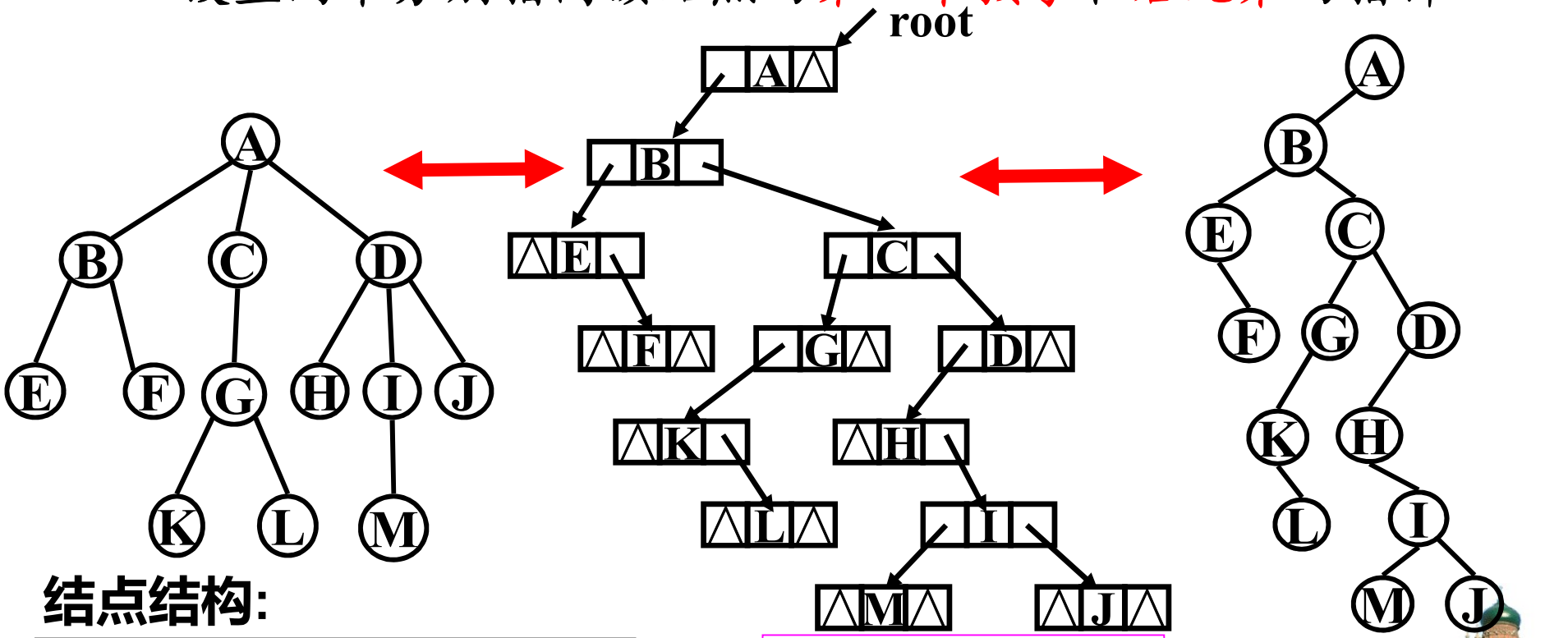
	data	parent	firstchild
1	A	0	
2	B	1	
3	C	1	
4	D	2	∧
5	E	2	
6	F	2	∧
7	G	3	∧
8	H	3	∧
9	I	5	∧





3.5 树(Cont.)

- 二叉链表表示法（(左)孩子—(右)兄弟链表表示）
  - 某结点的右兄弟是唯一的
  - 设置两个分别指向该结点的第一个孩子和右兄弟的指针



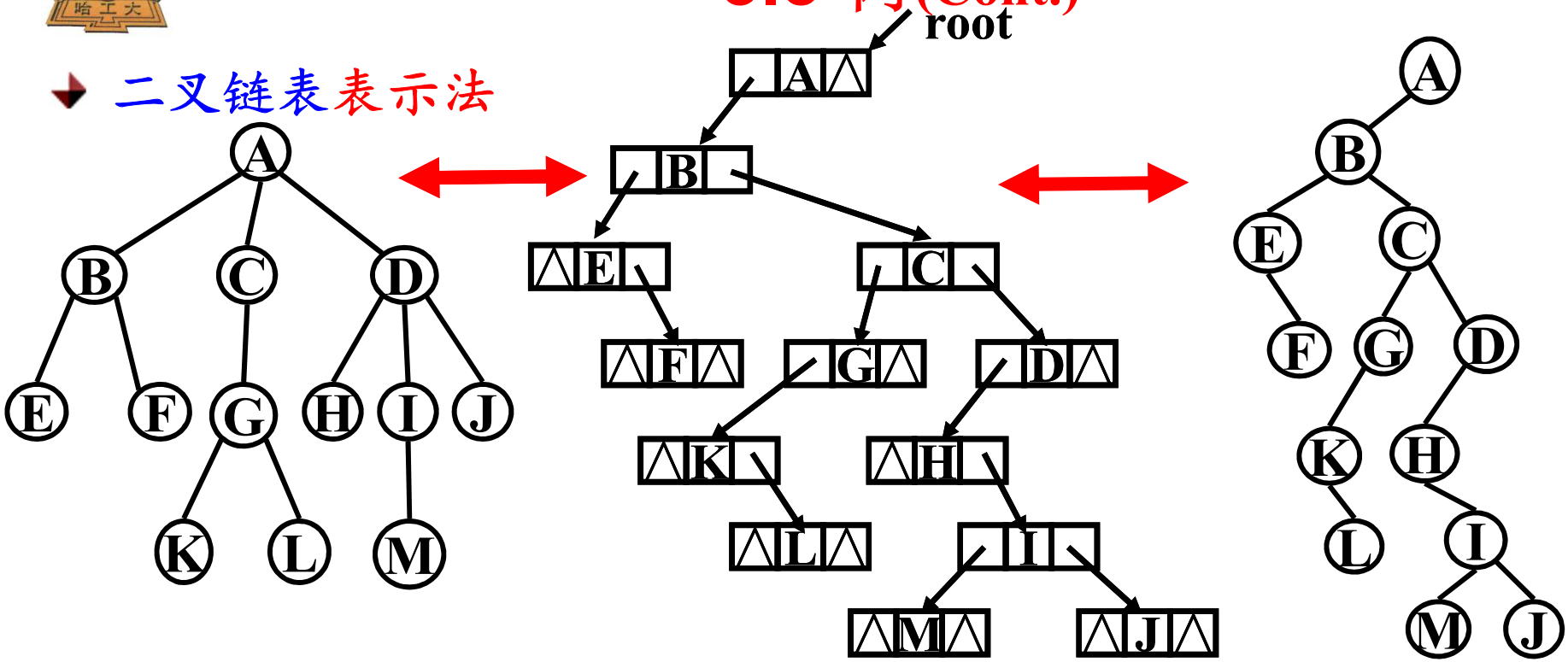
结点结构:

firstchild	data	rightsib

树的二叉树表示

3.5 树(Cont.)

- 二叉链表表示法

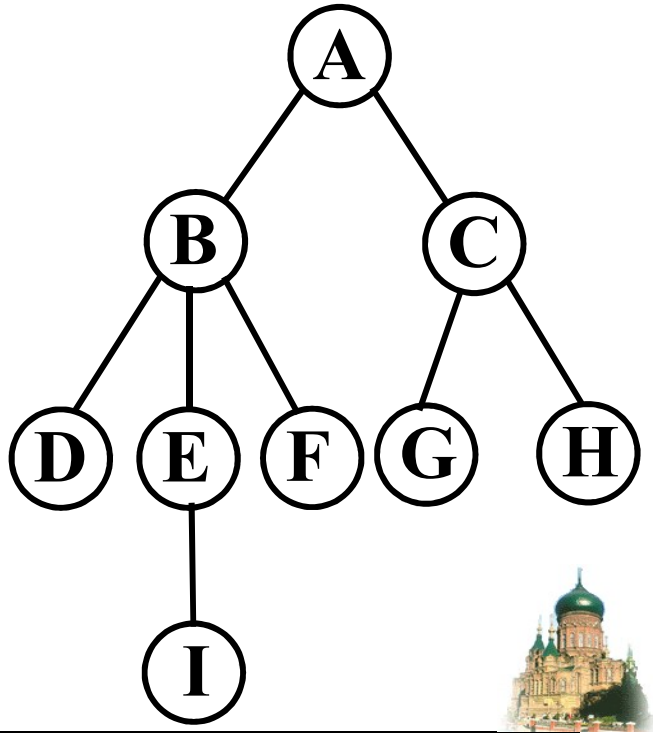


遍历	树	二叉树
先序	ABEFCGKLDHIMJ	ABEFCGKLDHIMJ
中序	EBFAKGLCHDMIJ	EFBKLGCHMIJDA
后序	EFBKLGCHMIJDA	FELKGMJIHDCBA

3.5 树(Cont.)

- 二叉链表表示法（(左)孩子—(右)兄弟链表表示）
  - **结点结构:** | firstchild | data | rightsib |
  - 类型定义:

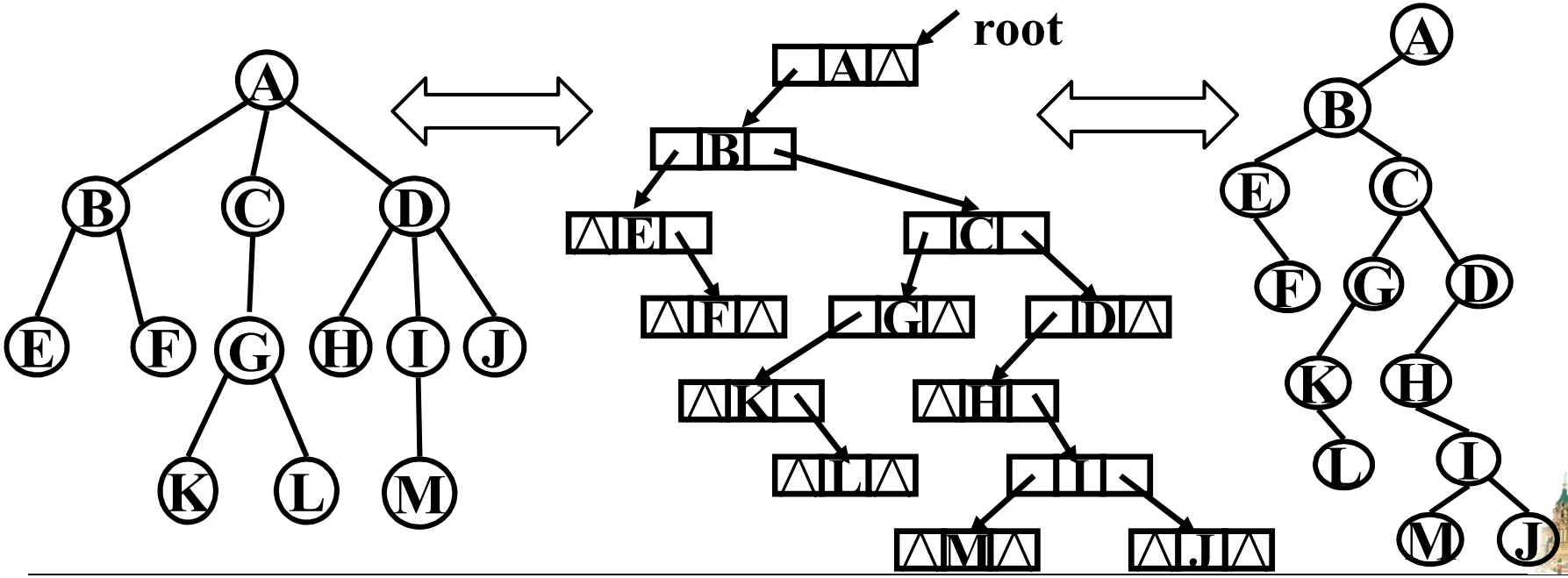
```
struct  CSNode { //动态存储结构
        DataType    data;
        CSNode   *firstchild , *rightsib ;
} ;
typedef struct  CSNode   *CSTREE ;
```



3.6 森林(树)与二叉树间的转换

- 树与二叉树自然对应关系
  - 从树的二叉链表可以看出：

	树		二叉树
结点关系	兄弟	⟷	双亲和右孩子
	双亲和长子	⟷	双亲和左孩子

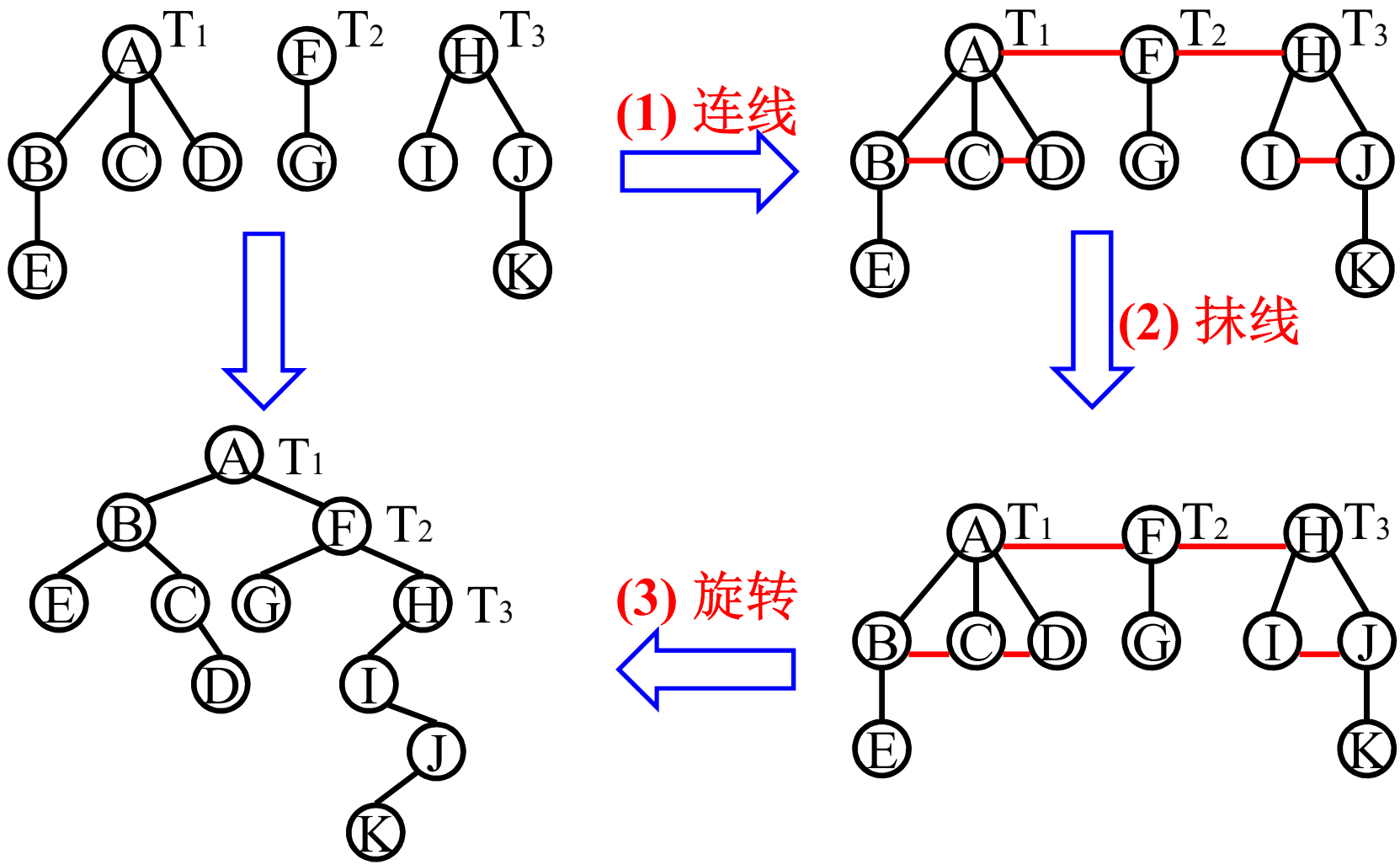


3.6 森林(树)与二叉树间的转换

- 森林(树)转换成二叉树
  - 连线：
    - 把每株树的各兄弟结点连起来；
    - 把各株树的根结点连起来（视为兄弟）
  - 抹线：
    - 对于每个结点，只保留与其最左儿子的连线，抹去该结点与其它结点之间的连线
  - 旋转：
    - 按顺时针旋转45度（左链竖画，右链横画）
    - 不是必要的

3.6 森林(树)与二叉树间的转换

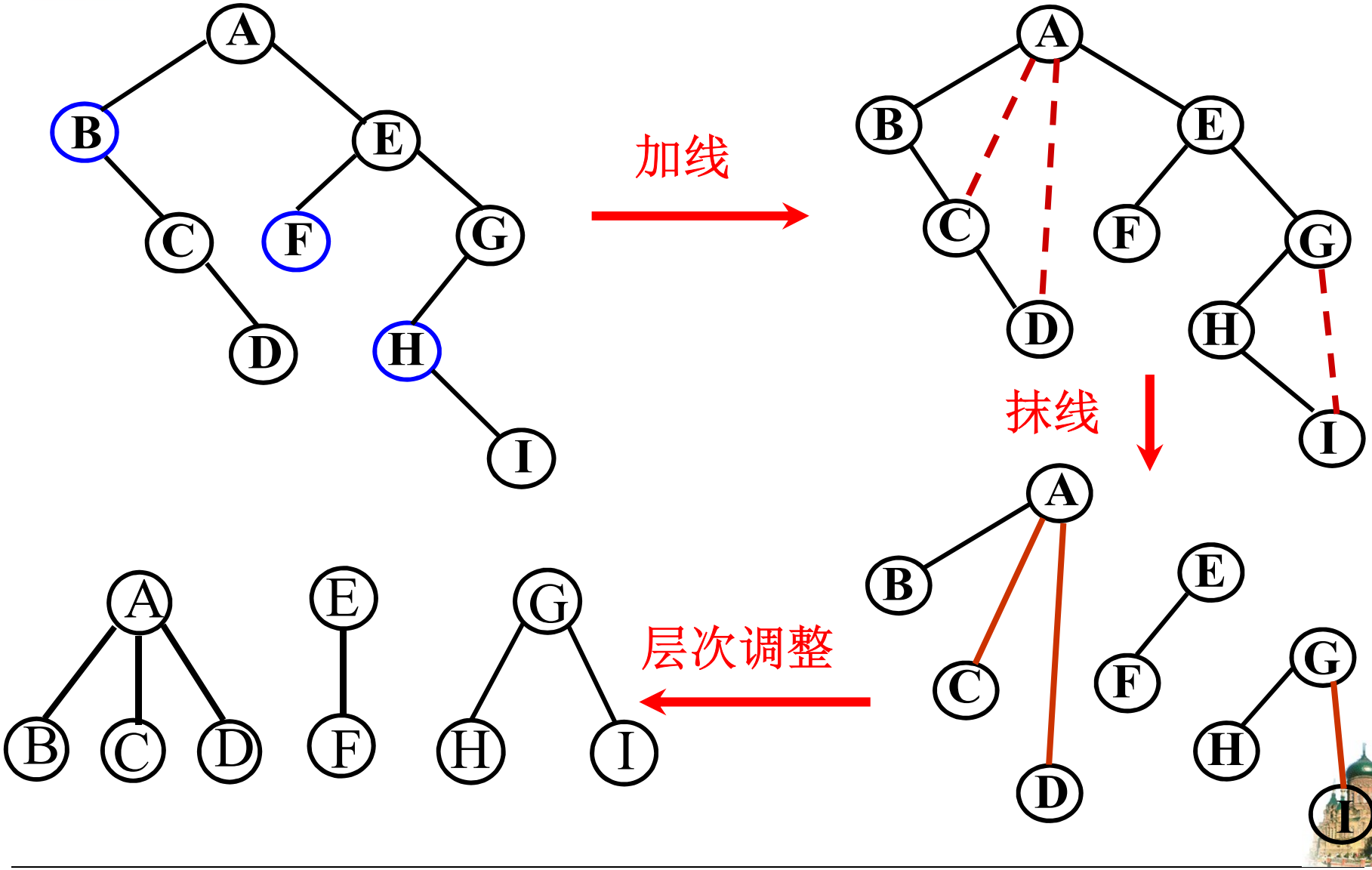
- 森林(树)转换成二叉树



3.6 森林(树)与二叉树间的转换

- 二叉树转换成森林(树)
  - 连线：　　对于有父结点且为父结点的左孩子
    - 若某个结点 k 是其双亲结点的左孩子，则将该结点 k 的右孩子以及（当且仅当）连续地沿着右孩子的右链不断搜索到的所有右孩子，都分别与结点k 的双亲结点相连；
  - 抹线：
    - 把二叉树中的所有结点与其右孩子的连线以及（当且仅当）连续地沿着右孩子的右链不断搜索到的所有右孩子的连线全部抹去；
  - 旋转：（不是必要的）
    - 按逆时针旋转45度角（即把结点按层次排列）
- 只考虑有双亲结点且为左孩子结点

3.6 森林(树)与二叉树间的转换



3.6 森林(树)与二叉树间的转换

森林(树)与二叉树之间的对应关系

- 将一株树转换为二叉树，二叉树一定没有右子树(原因？)
- 一般结论：森林中的任何没有右兄弟的结点在对应的二叉树中，没有右子树；
- 任何一个森林（树）对应唯一的一株二叉树，反之亦然。
  - 且第一株树的根对应二叉树的根；
  - 第一株树的所有子树森林对应二叉树的左子树；
  - 其余子树森林对应二叉树的右子树。

- 思考题：
  - 森林中的非终端结点数F

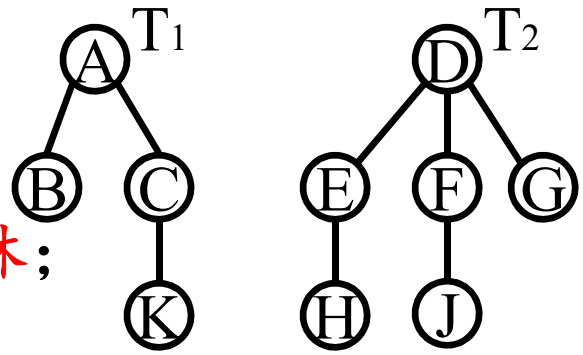
    =对应二叉树中右子树为空的结点数B-1

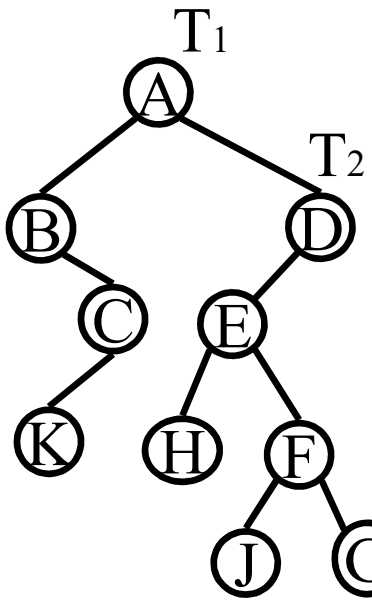
3.6 森林(树)与二叉树间的转换

- 森林(树)转换成二叉树的递归算法
  - F ={T1,T2, …,Tn}　二叉树B(F)
  - 若n=0，则B(F)为空；否则，n>0，则
    - B(F)的根就是root（T1）；
    - B(F)的左子树是F的第一棵树T1的子树森林；
    - B(F)的右子树F的其余子树森林。
- 二叉树转换成森林(树) 的递归算法
  - 若B 为空，则F 为空；若B 不空，则
    - F中的第一株树T1 的根对应二叉树B 的根；
    - T1中根结点的子树森林F1是由B的左子树转换来的；
    - F中除T1之外其余子树组成的森林F’ ={T2,…Tn}是由B的右子树转换而来的。

3.6 森林(树)与二叉树间的转换

- 非空森林的基本遍历
  - 先根遍历
    - 访问第一株树的根结点；
    - 按先根顺序遍历第一棵树的子树森林；
    - 按先根顺序遍历其余子树森林。
  - 后根遍历
    - 按后根顺序遍历第一株树的子树森林；
    - 访问第一株树的根结点；
    - 按后根顺序遍历其余子树森林。





遍历	森林	树	二叉树
先序	√	√	√
中序	×	× √	√
后序	√	√	√

3.7 树型结构的应用

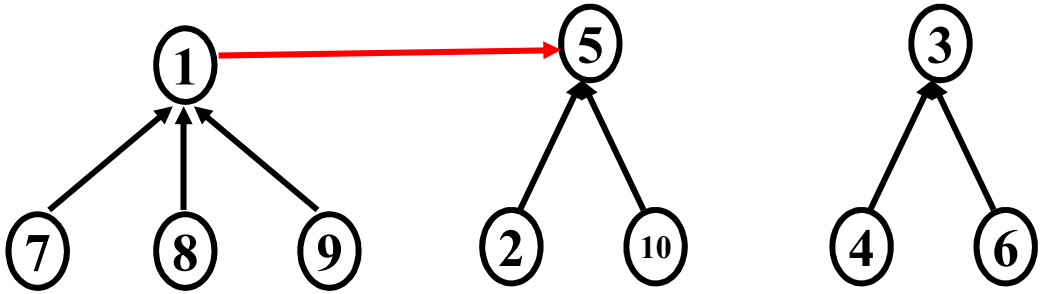
一、非相交集合的树结构表示

- ADT集合MFSET（并查集）
  - 集合：
    - 性质相同的元素所组成的整体（有限且互不相交）
  - 集合上的基本操作
    - Union(Si, Sj, S): if Si∩Sj==Φ， S = Si∪Sj;
    - Find(i, S): 求包含i 的集合;
    - Initial(x, S): 建立集合S，使之只包含x。
  - 例如，S1={1, 7, 8, 9}, S2={2, 5, 10 }, S3={3, 4, 6}，则 S1∪S2={1，2，5，7，8，9，10}

3.7 树型结构的应用(Cont.)

一、非相交集合的树结构表示

- **ADT集合MFSET的实现**
  - 集合的树结构表示（父链表示）
    - 令集合的元素（正整数）对应于数组的下标，
    - 而相应的元素值表示其父结点所对应的数组单元下标。
    - “并”：把其中之一当成另一棵树的子树即可。
    - “包含”：求元素所在的树根。



parent		6 5	5	0	3	0	3	1	1	1	5
	0	1	2	3	4	5	6	7	8	9	10

数组下标：代表元素名

根结点的下标：集合名

3.7 树型结构的应用(Cont.)

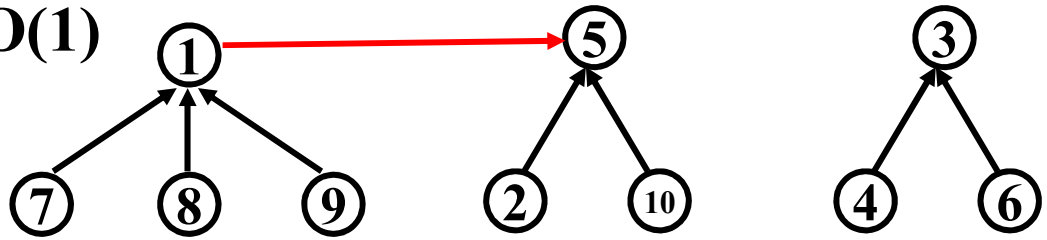
一、非相交集合的树结构表示

- 集合的存储结构

```
#define n 元素的个数
typedef int MFSET[n+1];
/* 集合的“型”为MFSET,元素的“型”为int */
```

- 基本操作的实现

```
void Union(int i, int j ,MFSET parent)
{   parent[i]=j;/* 归并，结果树之根为j */
}//O(1)
```



parent		6 5	5	0	3	0	3	1	1	1	5
	0	1	2	3	4	5	6	7	8	9	10

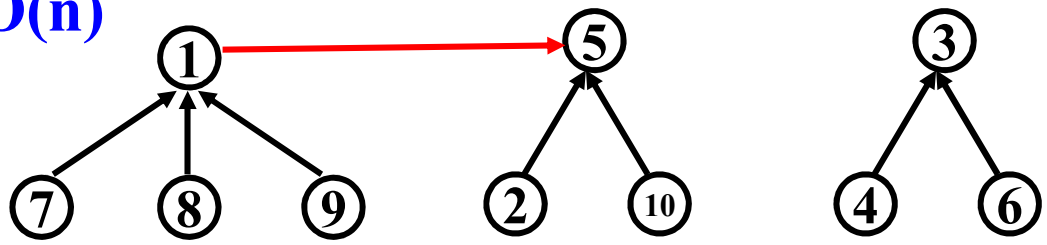
3.7 树型结构的应用(Cont.)

一、非相交集合的树结构表示

- 基本操作的实现

```
void Initial(int x ,MFSET parent)
{    parent[x]=0;
} //O(1)
```

```
int Find(int i, MFSET parent)
{   int tmp=i;
    while(parent[tmp]!=0)/* >0,未到根 */
      tmp=parent[tmp]; /* 上溯 */
    return tmp;
}//O(n)
```



parent		5	5	0	3	0	3	1	1	1	5
	0	1	2	3	4	5	6	7	8	9	10

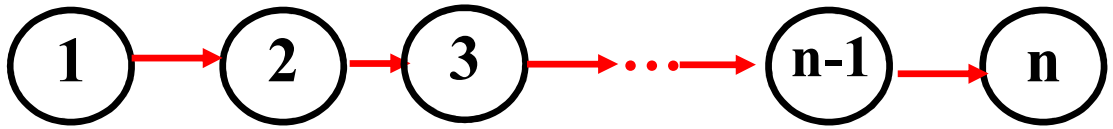
3.7 树型结构的应用(Cont.)

一、非相交集合的树结构表示

- 性能分析：看下列操作序列
  - Union(1，2，parent)， Find(1，parent)
  - Union(2，3，parent)， Find(1，parent)
  - Union(3，4，parent)， Find(1，parent)
  - ……
  - Union(n-1,n，parent)，Find(1，parent)

原因：在“并”操作时，将结点多的并入结点少的，从而形成单链树。

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



每次执行Union的时间都是O(1)，共n-1次，所需时间O(n)；而每个Find(1, parent)，需要从1开始找到根，当1位于第i层时，Find(1, parent) 所需时间为O(i)，共n-2次，所需时间为$O(\sum i) = O(n^2)$。

3.7 树型结构的应用(Cont.)

一、非相交集合的树结构表示

- 改进的ADT MFSET的实现
  - 基本想法：
    - 改进“并”操作的原则，即将结点少的并入结点多的；相应的存储结构也要提供支持—以加权规则压缩高度。
  - 存储结构：

```
typedef struct{
    int father;
    int count; /* 加权 */
} MFSET[ n+1 ];
```

  - 基本操作的实现：

3.7 树型结构的应用(Cont.)

一、非相交集合的树结构表示

- 改进的ADT MFSET的实现

```
void Union(int A,int B,MFSET C)
{   if(C[A].count > C[B].count) { /* |B|<|A| */
        C[B].father = A; /* 并入A */
        C[A].count += C[B].count;
    }
    else {   /*|A|<|B|*/
        C[A].father = B;  /* 并入B */
        C[B].count += C[A].count;
    }
}//O(1)
```

3.7 树型结构的应用(Cont.)

一、非相交集合的树结构表示

- 改进的ADT MFSET的实现

```
int Find(int x, MFSET C)
{     int tmp=x;
      while(C[tmp].father!=0)/*>0,未到根 */
          tmp=C[tmp].father; /* 上溯 */
      return tmp;
} //O(h),h为树的高度,可用数学归纳法证明:h≤ ⌊log2 n ⌋+ 1
void Initial(int A ,MFSET C)
{     C[A].father=0;
      C[A].count=1;
} //O(1)
```

3.7 树型结构的应用(Cont.)

一、非相交集合的树结构表示

- 集合的等价分类
  - 等价关系：集合S上具有自反性、对称性和传递性的二元关系R.
  - 等价类：x∈S，y∈S，x≡y ⇔ (x，y)∈R或 xRy。
  - 集合S上的一个等价关系唯一确定一个等价类的集合S/R(商集)。
  - 等价分类：把一个集合分成若干个等价类的过程(分清、分净)
  - 等价分类算法：
    - 例如集合S = {1,2,3,4,5,6,7}的等价对分别是：1≡2，5≡6，3≡4，1≡4

3.7 树型结构的应用(Cont.)

一、非相交集合的树结构表示

- 集合的等价分类
  - 等价分类算法：
    - 1. 令S中的每一个元素自身构成一个等价类，S1,S2,…S7
    - 2.重复读入等价对（i, j）
      - 2.1对每个读入的等价对(i, j),求出i 和j 所在的集合Sk和Sm（不失一般性）
      - 2.2若Sk≠Sm,则将Sk并入Sm,并将Sk置空。
  - 当所有的等价对处理过后，S1,S2,…S7中的非空集合即为S的R等价类

3.7 树型结构的应用(Cont.)

```
void Equivalence (MFSET S)  /*等价分类算法*/
{    int i ,j , k ,m;
     for(i=1; i<=n+1;i++)
         Initial(i,S);          /*使集合S只包含元素i */
     cin>>i>>j;                 /* 读入等价对*/
     while(!(i==0&&j==0){ /* 等价对未读完*/
          k=Find(i,S);          /*求i的根*/
          m=Find(j,S);          /* 求j的根*/
          if(k!=m)          /*if k==m,i,j已在一个树中，不需合并*/
              Union(i,j,S);     /*合并*/
         cout<<i<<j;
     }
}
```

3.7 树型结构的应用(Cont.)

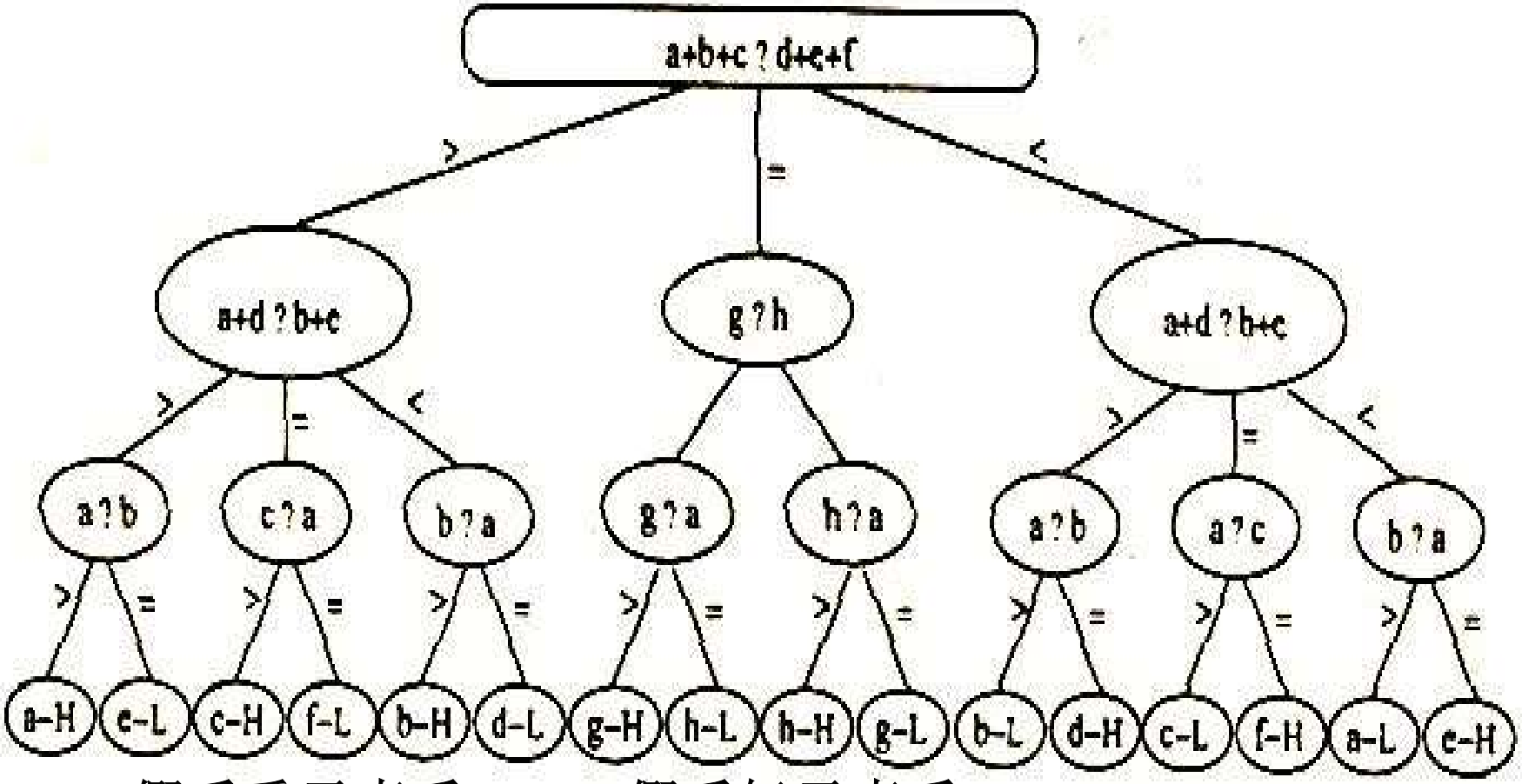
二、判定树

- 八枚硬币问题：
  - 假定有八枚硬币a、b、c、d、e、f、g、h，已知其中1枚是伪造的假币，假币的重量与真币不同，或重或轻。要求以天平为工具，用最少的比较次数挑出假币。

3.7 树型结构的应用(Cont.)

二、判定树

- 八枚硬币问题的判定树



H—假币重于真币；L—假币轻于真币

3.7 树型结构的应用(Cont.)

二、判定树

- 判定树的特点：
  - 一个判定树是一个算法的描述；
  - 每个内部结点对应一个部分解；
  - 每个叶子（外部结点）对应一个解；
  - 每个内部结点连接与一个获得新信息的测试；
  - 从每个结点出发的分支标记着不同的测试结果；
  - 一个求过程对应于从根到叶的一条路；
  - 一个判定树是所有可能的解的集合。

3.7 树型结构的应用(Cont.)

三、哈夫曼（Huffman）树及其应用

- 相关术语：
  - 叶子结点的权值：
    - 对叶子结点赋予的一个有意义的数值量（实数）。
  - 二叉树的带权路径长度：
    - 设二叉树具有n个带权值的叶子结点，从根结点到各个叶子结点的路径长度与相应叶子结点权值的乘积之和。记为：

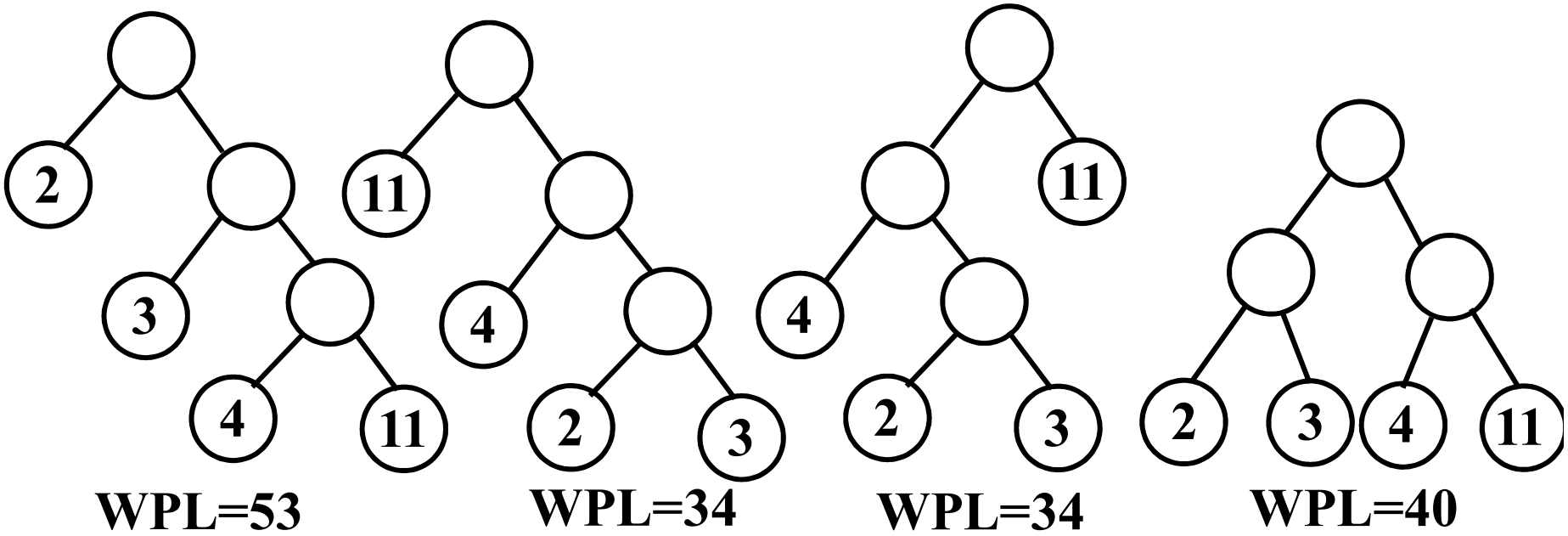
$$WPL=\sum_{k=1}^{n} w_k l_k$$

l_k：从根结点到第k个叶子的路径长度

w_k：第k个叶子的权值；

3.7 树型结构的应用(Cont.)

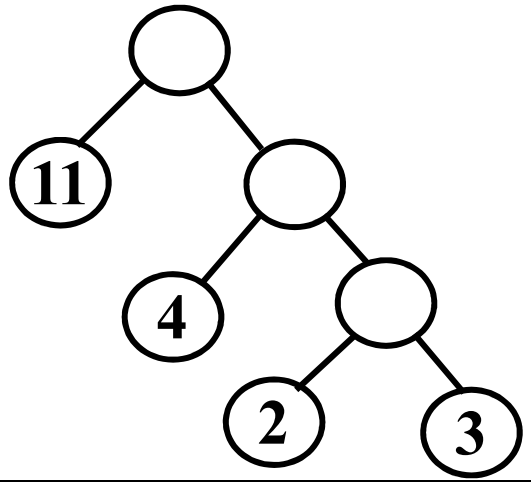
- 相关术语：
  - 举例：给定4个叶子结点，其权值分别为{2，3，4，7}，可以构造出形状不同的二叉树



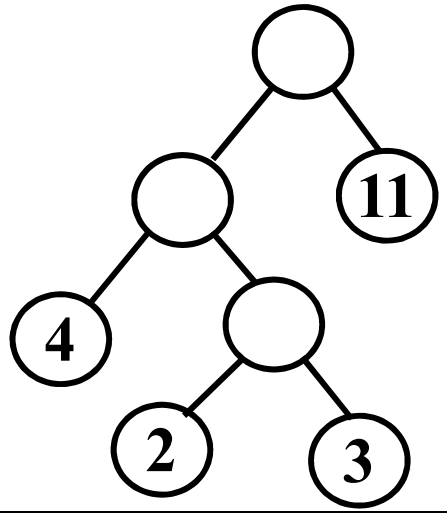
  - 哈夫曼树：给定一组具有确定权值的叶子结点，带权路径长度最小的二叉树，称为哈夫曼树，亦称最优二叉树。

3.7 树型结构的应用(Cont.)

- 哈夫曼树的特点：
  - 权值越大的叶子结点越靠近根结点，而权值越小的叶子结点越远离根结点。（构造哈夫曼树的核心思想）
  - 只有度为0（叶子结点）和度为2（分支结点）的结点，不存在度为1的结点。
  - n个叶结点的哈夫曼树的结点总数为2n-1个。
  - 哈夫曼树不唯一，但WPL唯一。



WPL=34

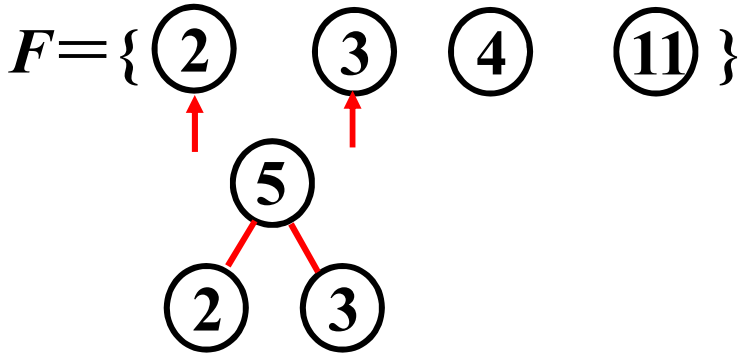


3.7 树型结构的应用(Cont.)

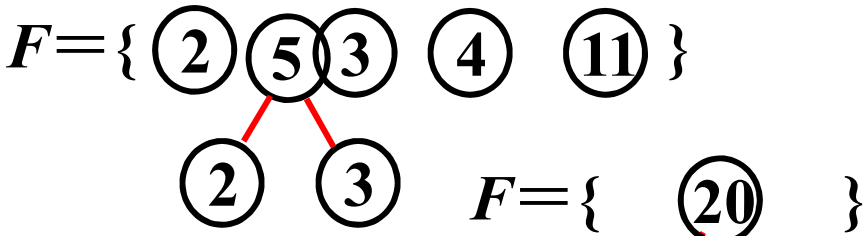
- 哈夫曼树的构造算法：
  - 输入： n个权值$\{w_1, w_2, \ldots, w_n\}$
  - 输出：哈夫曼树
- 步骤：
  - (1) 初始化：由给定的n个权值$\{w_1, w_2, \ldots, w_n\}$构造$n$棵只有一个根结点、左右子树均空的二叉树，从而得到一个二叉树集合$F=\{T_1, T_2, \ldots, T_n\}$；
  - (2) 选取与合并：在F中选取根结点的权值最小的两棵二叉树分别作为左、右子树构造一棵新的二叉树，这棵新二叉树的根结点的权值为其左、右子树根结点的权值之和；
  - (3) 删除与加入：在F中删除作为左、右子树的两棵二叉树，并将新建立的二叉树加入到F中；
  - (4) 重复(2)、(3)两步，当集合F中只剩下一棵二叉树时，这棵二叉树便是哈夫曼树。

3.7 树型结构的应用(Cont.)

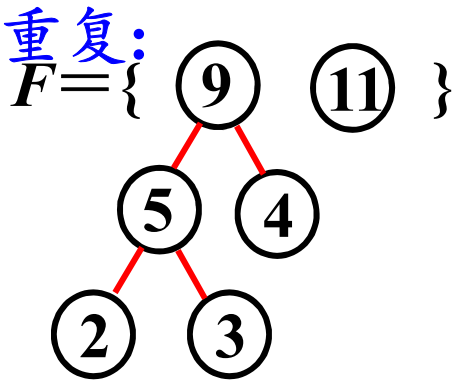
- 哈夫曼树的构造示例：$W=\{2, 3, 4, 11\}$
  - 初始化：
  - 选取与合并：

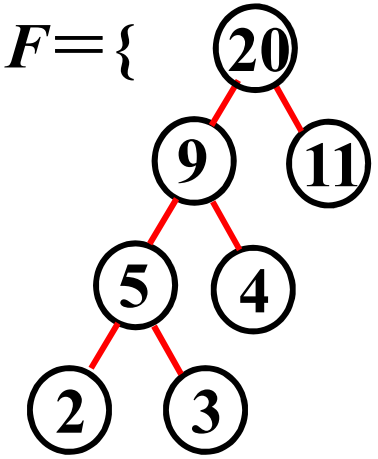


  - 删除与加入：



  - 重复：





3.7 树型结构的应用(Cont.)

- 哈夫曼树的存储结构----静态三叉链表

```
typedef struct { /* 结点型 */
    double weight ; /* 权值 */
    int   lchild ; /* 左孩子链 */
    int   rchild ; /* 右孩子链 */
    int   parent; /* 双亲链 */
}HTNODE ;
typedef HTNODE HuffmanT[ 2n-1 ] ;
```

	weight	parent	lchild	rchild
0				
1				
2				
(2n-1)-1				

HuffmanT T;

3.7 树型结构的应用(Cont.)

- 哈夫曼树构造算法的实现步骤：
  - 1 初始化：将T[0]，…T[2n-2]共 2n-1个结点的三个链域均置空(-1)，权值为 0 ；
  - 2 输入权值：读入 n 个叶子的权值存于T的前 n 个单元T[0]，…T[n], 它们是 n 个独立的根结点上的权值；
  - 3 合并：对二叉树集合进行 n-1次合并：
    - 3.1 在二叉树集合T [0]，…T[i-1]中选取权值最小和次最小的两个根结点$T[p_1]$和$T[p_2]$作为合并对象，这里 $0<= p_1$，$p_2<= i-1$；
    - 3.2将根为$T[p_1]$和$T[p_2]$的两棵二叉树作为左、右子树合并为一棵新二叉树，新二叉树的根结点为T[i]。

3.7 树型结构的应用(Cont.)

- 哈夫曼树构造算法的实现示例：

7 5 2 4

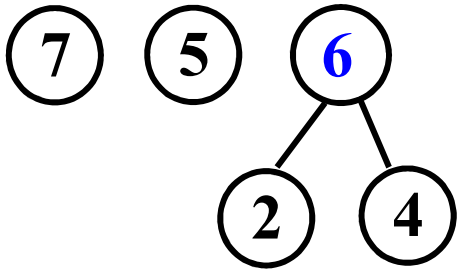
	weight	parent	lchild	rchild
0	7	-1	-1	-1
1	5	-1	-1	-1
2	2	-1	-1	-1
3	4	-1	-1	-1
4		-1	-1	-1
5		-1	-1	-1
6		-1	-1	-1

初始化

3.7 树型结构的应用(Cont.)

- 哈夫曼树构造算法的实现示例：

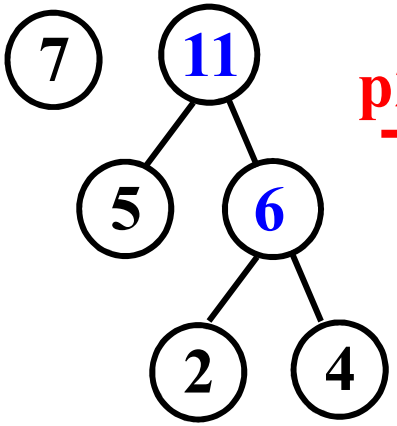
	weight	parent	lchild	rchild
0	7	-1	-1	-1
1	5	-1	-1	-1
p1 → 2	2	4 ~~-1~~	-1	-1
p2 → 3	4	4 ~~-1~~	-1	-1
i → 4	6	-1	2 ~~-1~~	3 ~~-1~~
5		-1	-1	-1
6		-1	-1	-1



过程

3.7 树型结构的应用(Cont.)

- 哈夫曼树构造算法的实现示例：

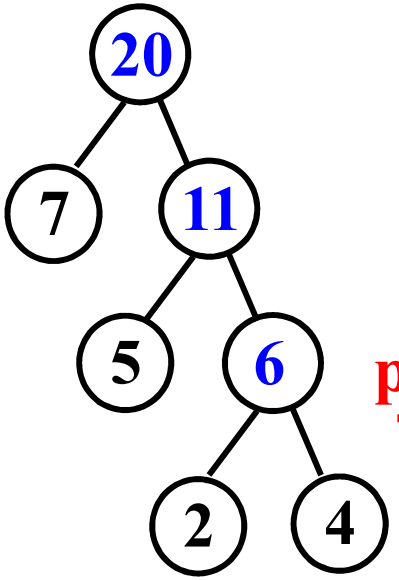


	weight	parent	lchild	rchild
0	7	-1	-1	-1
p1 → 1	5	5 ~~-1~~	-1	-1
2	2	4	-1	-1
3	4	4	-1	-1
p2 → 4	6	5 ~~-1~~	2	3
i → 5	11	-1	1 ~~-1~~	4 ~~-1~~
6		-1	-1	-1

过程

3.7 树型结构的应用(Cont.)

- 哈夫曼树构造算法的实现示例：



	weight	parent	lchild	rchild
p1 → 0	7	6 ~~-1~~	-1	-1
1	5	5	-1	-1
2	2	4	-1	-1
3	4	4	-1	-1
4	6	5	2	3
p2 → 5	11	6 ~~-1~~	1	4
i → 6	20	-1	0 ~~-1~~	5 ~~-1~~

构造过程

3.7 树型结构的应用(Cont.)

- 哈夫曼树构造算法的实现

```
void  CreartHT(HuffmanT T)//构造huffam树,T[2n-2]为其根
{  int i ,p1 ,p2;
   InitHT(T);                                  //1.初始化
   InputW(T);                                  //2.输入权值
   for（i = n; i < 2n-1; i++）{                //3. n-1次合并*/
      SelectMin(T, i-1, &p1, &p2);             //3.1
      T[p1].parent =T[p2].parent = i; //3.2
      T[i].lchild= p1;
      T[i].rchild= p2;
      T[i].weight =T[p1].weight + T[p2].weight;
   }
}
```

3.7 树型结构的应用(Cont.)

哈夫曼树的应用----哈夫曼编码

- 相关术语
  - (二进制)编码：是指把一组对象(如字符集)中的每个对象用唯一的一个二进制位串表示。如ASCII，指令系统
  - 解码(译码)：将二进制串转换为对应对象(字符)的过程。
  - 等长编码：表示一组对象的二进制位串的长度相等。
  - 不等长编码：表示一组对象的二进制位串的长度不相等
    - 等长编码什么情况下空间效率高？
    - 不等长编码什么情况下空间效率高？

3.7 树型结构的应用(Cont.)

哈夫曼树的应用----哈夫曼编码

- 相关术语
  - 编码的前缀性：如果一组编码中，任意一个编码都不是其它任何一个编码的前缀，则称这种编码具有前缀性，简称前缀码。
    - 前缀编码保证了在解码(译码)时的唯一性。
    - 等长编码具有前缀性；
    - 变长编码可能使译码产生二义性，即不具有前缀性。
      - 如, E(00), T(01), W(0001), 则译码时无法确定二进制串0001是ET还是W。

3.7 树型结构的应用(Cont.)

哈夫曼树的应用----哈夫曼编码

- 相关术语
  - 平均编码长度：
    - 对于给定的字符集（一组对象），可能存在多种编码方案，但应选择最优的。
    - 平均编码长度：设每个（对象）字符c_j的出现的概率为p_j，其二进制位串长度（码长）为l_j，则$\sum l_j \cdot p_j$表示该组对象（字符）的平均编码长度。
    - 最优前缀码：使得平均编码长度$\sum l_j \cdot p_j$最小的前缀编码称为最优的前缀码。

3.7 树型结构的应用(Cont.)

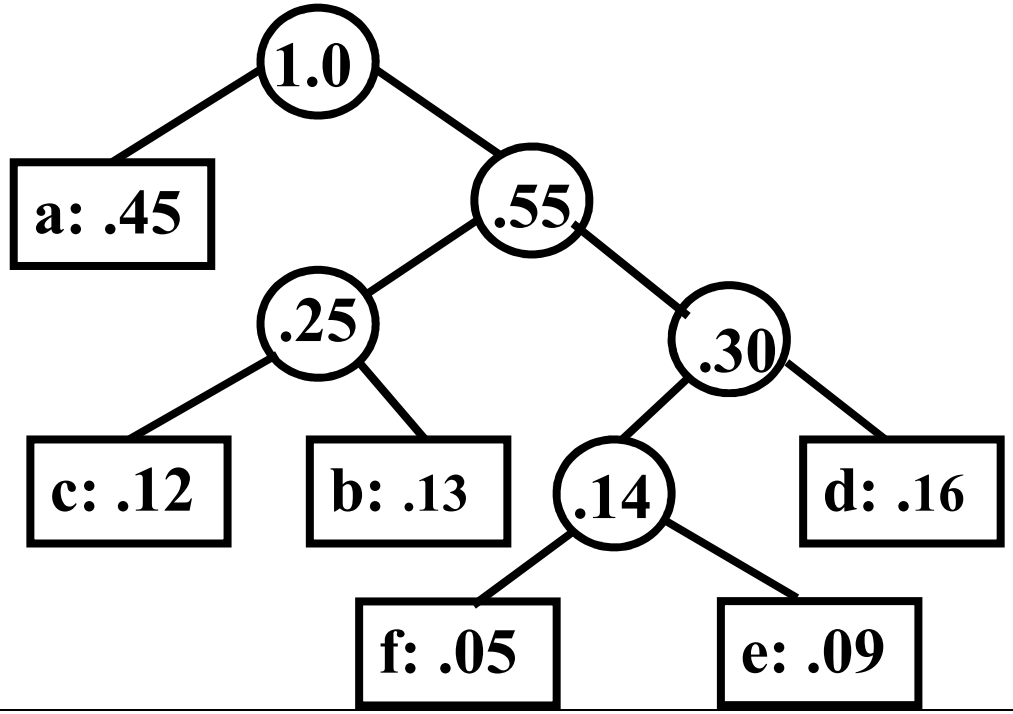
哈夫曼树的应用----哈夫曼编码

- 哈夫曼编码问题
  - 对于给定的字符集及其每个字符出现的概率(使用频度)，求该字符集的最优的前缀性编码—哈夫曼编码问题
- 用哈夫曼算法求字符集最优前缀编码的算法：
  - 使字符集中的每个字符对应一棵只有叶结点的二叉树，叶的权值为对应字符的使用频率----初始化
  - 利用huffman算法来构造一棵huffman树----构造算法
  - 对huffman树上的每个结点，左支附以0，右支附以1(或者相反)，则从根到叶的路径上的0、1序列就是相应字符的编码----哈夫曼编码
  - 且为最优前缀码

3.7 树型结构的应用(Cont.)

- 哈夫曼编码示例

字符	a	b	c	d	e	f	平均码长
概率	0.45	0.13	0.12	0.16	0.09	0.05	
等长	000	001	010	011	100	101	3 $=\lceil \log_2\lvert C\rvert \rceil$
变长	0	101	100	111	1101	1100	2.24 $=\sum l_j \cdot p_j$



	ch	bits
0	a	0\0
1	b	101\0
2	c	100\0
3	d	111\0
4	e	1101\0
5	f	1100\0

编码表H

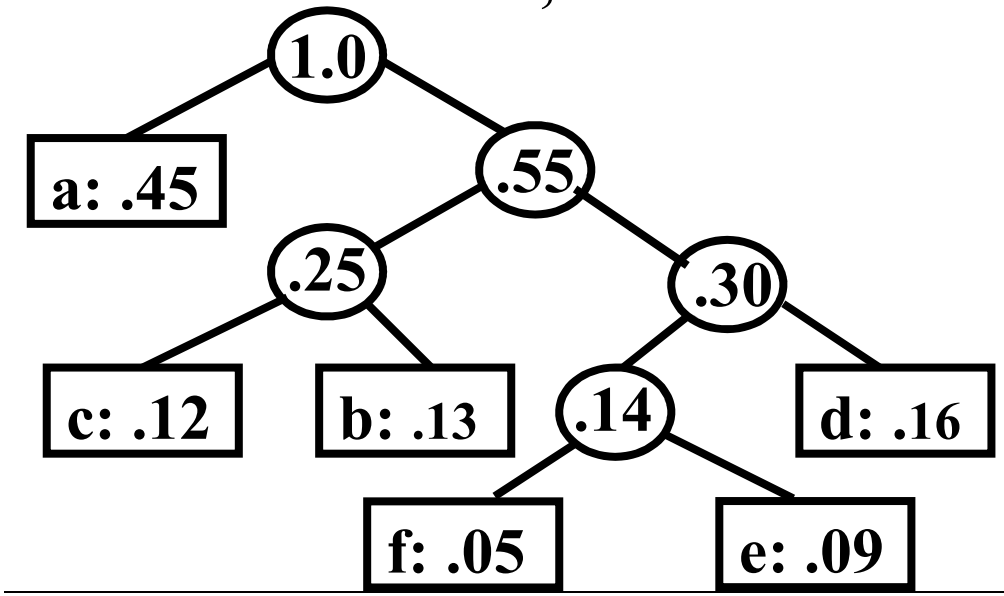
3.7 树型结构的应用(Cont.)

- 哈夫曼编码表的存储结构

```
typedef struct{
  char  ch;  //存储被编码的字符
  char  bits[n+1]; //字符编码位串
}CodeNode;
typedef CodeNode HuffmanCode[n];
HuffmanCode    H;
```

	ch	bits
0	a	0\0
1	b	101\0
2	c	100\0
3	d	111\0
4	e	1101\0
5	f	1100\0

编码表H



	ch	weight	parent	lchild	rchild
0	a	0.45	10	-1	-1
1	b	0.13	7	-1	-1
2	c	0.12	7	-1	-1
3	d	0.16	8	-1	-1
4	e	0.09	6	-1	-1
5	f	0.05	6	-1	-1
6		0.14	8	5	4
7		0.25	9	2	1
8		0.30	9	6	3
9		0.55	10	7	8
10		1.00	-1	0	9

哈夫曼树T

3.7 树型结构的应用(Cont.)

哈夫曼编码算法的实现

```
void CharSetHuffmanEncoding( HuffmanT T, HuffmanCode H)
{ /*根据Huffman树T 求Huffman编码表 H*/
    int c, p, i;          /* c 和p 分别指示T 中孩子和双亲的位置 */
    char cd[n+1];         /* 临时存放编码 */
    int start;            /* 指示编码在cd 中的位置 */
    cd[n]= '\0' ;         /* 编码结束符 */
    for( i =0; i <n;  i++){  /* 依次求叶子T[i]的编码 */
       H[i].ch=getchar();  /* 读入叶子T[i]对应的字符*/
       start=n;            /* 编码起始位置的初值 */
       c =i;               /* 从叶子T[i]开始上溯 */
       while( (p=T[c].parent)>=0){ /* 直到上溯到T[c]是树根位置 */
          cd[--start]=(T[p].lchild==c) ? '0' : '1' ;
          /* 若T[c]是T[p]的左孩子，则生成代码0，否则生成代码1*/
          c=p;                      }  /* 继续上溯*/
      strcpy(H[i].bits, &cd[start]); /*复制编码为串于编码表H*/
    }
}
```

	ch	bits
0	a	0\0
1	b	101\0
2	c	100\0
3	d	111\0
4	e	1101\0
5	f	1100\0

编码表H

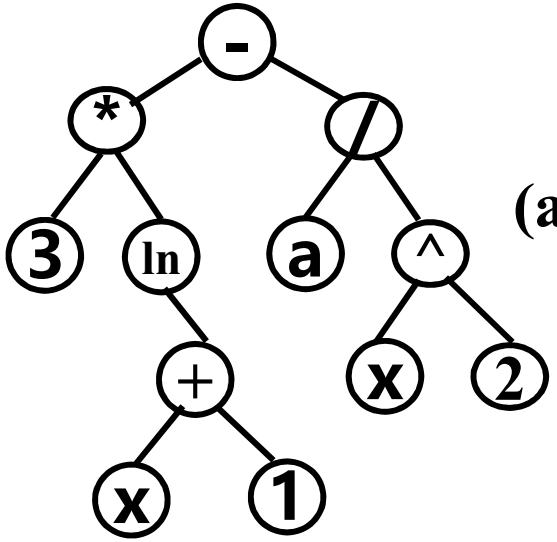
3.7 树型结构的应用(Cont.)

- 利用Huffman编码对数据文件编码和译码
  - 编码：依次读入文件的字符c，在huffman编码表H中找到此字符，若H[i].ch==c，则将c 转换为H[i].bits中的编码串
  - 译码：依次读入文件的二进制码，在huffman树中从根结点T[m-1]出发，若读入0，则走左支，否则，走右支，一旦到达某叶结点T[i]时便译出相应的字符H[i].ch。然后重新从根出发继续译码，直到文件结束。
- 哈夫曼编码一定具有前缀性；
- 哈夫曼编码是最小冗余码；
- 哈夫曼编码方法，使出现概率大的字符对应的码长较短；
- 哈夫曼编码不唯一，可以用于加密；
- 哈夫曼编码译码简单唯一，没有二义性。

3.7 树型结构的应用(Cont.)

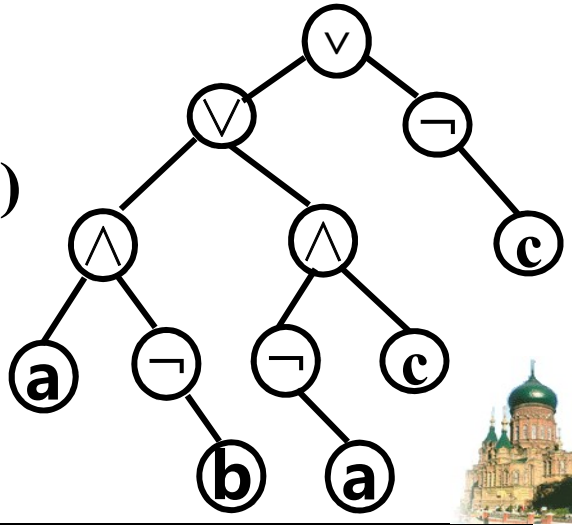
四、树的应用—表达式求值

- 用树结构表示表达式----表达式树
  - 叶结点表示操作数；
  - 非叶结点表示运算符：
    - 二元运算符有两棵子树对应于它的操作数；
    - 一元运算符有一棵子树对应于它的操作数。



$$3 * \ln(x + 1) - a / x^2$$

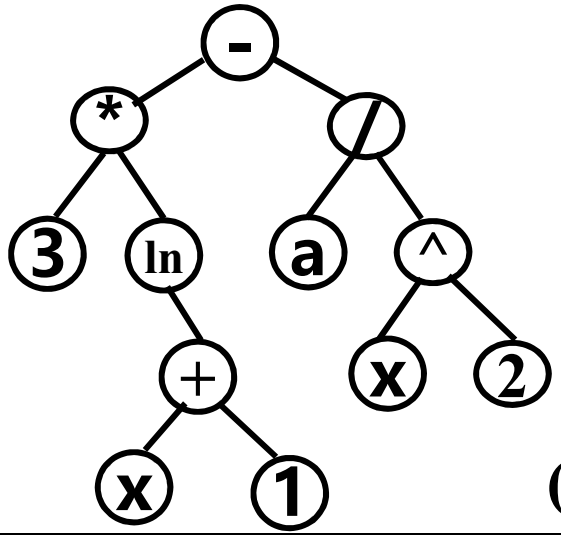
$$(a \wedge \neg b) \vee (\neg a \wedge c) \vee (\neg c)$$



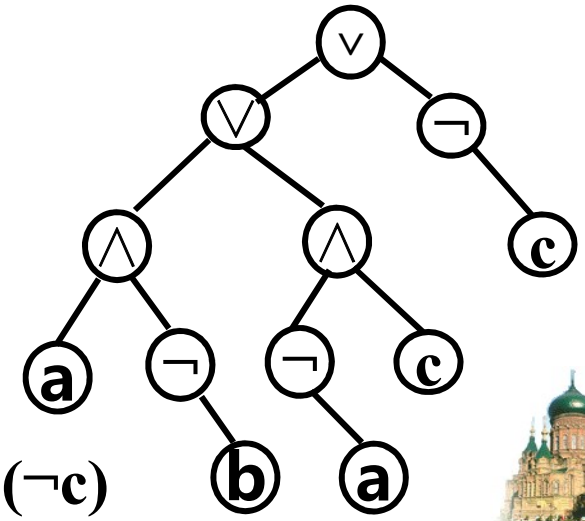
3.7 树型结构的应用(Cont.)

四、树的应用—表达式求值

- 表达式求值方法
  - 把中缀表达式转换为后缀表达式（栈结构、树结构，还有么？）；根据后缀表达式计算表达式的值
  - 利用后序遍历算法，先计算左子树的值，然后再计算右子树的值。当到达某结点时，该结点的左右操作数都已求出



$3 * \ln(x + 1) - a / x^2$



$(a \wedge \neg b) \vee (\neg a \wedge c) \vee (\neg c)$

本章小结

- 知识点：二叉树、森林（树）
- 知识点体系结构

ADT基本数据结构

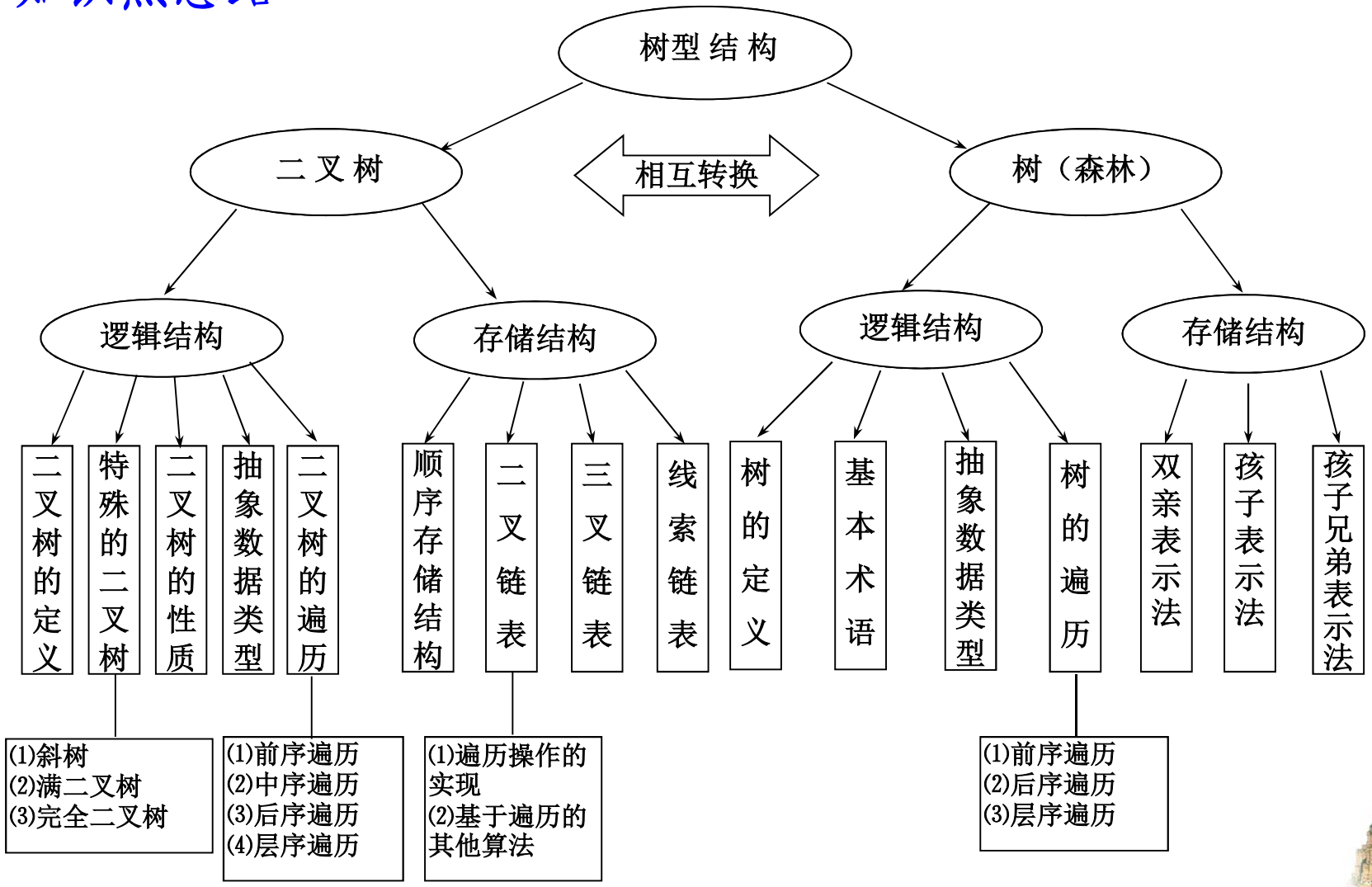
- **ADT定义**逻辑结构
  - 定义及相关术语
  - 逻辑结构及其特征
  - 基本操作（算法）→ 动态的操作

  （定义及相关术语、逻辑结构及其特征：静态的结构）
- **ADT实现**存储结构
  - 存储结构（描述）
  - 存储结构特点
  - 存储结构的定义
  - 操作（算法）实现 → 动态的操作
  - 算法的性能

  （存储结构（描述）、存储结构特点、存储结构的定义：静态的结构）
- 应用
  - 集合表示
  - 判定树
  - 哈夫曼树
    - 哈夫曼树及其构造
    - 哈夫曼编码与译码
  - 表达式求值

本章小结

知识点总结


树型结构
二叉树
相互转换
树（森林）
逻辑结构
存储结构
逻辑结构
存储结构
二叉树的定义
特殊的二叉树
二叉树的性质
抽象数据类型
二叉树的遍历
顺序存储结构
二叉链表
三叉链表
线索链表
树的定义
基本术语
抽象数据类型
树的遍历
双亲表示法
孩子表示法
孩子兄弟表示法
(1)斜树
(2)满二叉树
(3)完全二叉树
(1)前序遍历
(2)中序遍历
(3)后序遍历
(4)层序遍历
(1)遍历操作的实现
(2)基于遍历的其他算法
(1)前序遍历
(2)后序遍历
(3)层序遍历

该奋斗的年龄，不要选择了安逸

- 该奋斗的年龄，不要选择了安逸
- 很多东西
- 就掌握在自己手中。
- 比如快乐，你不快乐，
- 谁会同情你的悲伤；
- 比如坚强，你不坚强，
- 谁会怜悯你的懦弱；
- 比如努力，你不努力，
- 谁会陪你原地停留。
- 只有把命运掌握在自己手中。

- 该奋斗的年龄，不要选择了安逸
- 从今天起要努力，
- 即使看不到希望，
- 也要相信自己。
- 压力不是有人比你努力，
- 而是比你牛几倍的人
- 依然在努力。

该奋斗的年龄，不要选择了安逸

- 该奋斗的年龄，不要选择了安逸
- 学习遇到挫折，
- 你退缩，说难；
- 生活遇到困难，
- 你抱怨，说苦；
- 总怨天尤人，唉声叹气，
- 不过是成全别人的成就，
- 悲观了自己的路。
- 即使今天不如意，
- 但你年轻，
- 努力便有未来。
- 该奋斗的年龄，不要选择了安逸
- 最终
- 你相信什么，
- 就能成为什么。
- 因为世界上最可怕的两个词，
- 一个叫执着，
- 一个叫认真，
- 认真的人改变自己，
- 执着的人改变命运。
- 只要在路上，
- 就没有到不了的地方。