

# 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；
- 掌握堆、选择树的定义，森林(树)与二叉树的对应关系和相互转换方法；
- 了解树型结构的应用，重点掌握并查集的理论、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**



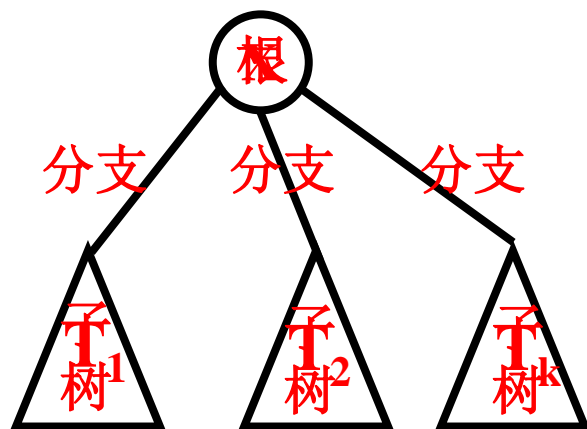


## 3.1 树与二叉树的基本术语

### 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， $T_1, T_2, \dots, T_k$  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  $T_1, T_2, \dots, T_k$ 。这些边也叫做分支， $T_1, T_2, \dots, T_k$  称作根为  $X$  的树之子树（SubTree）。

### 说明：

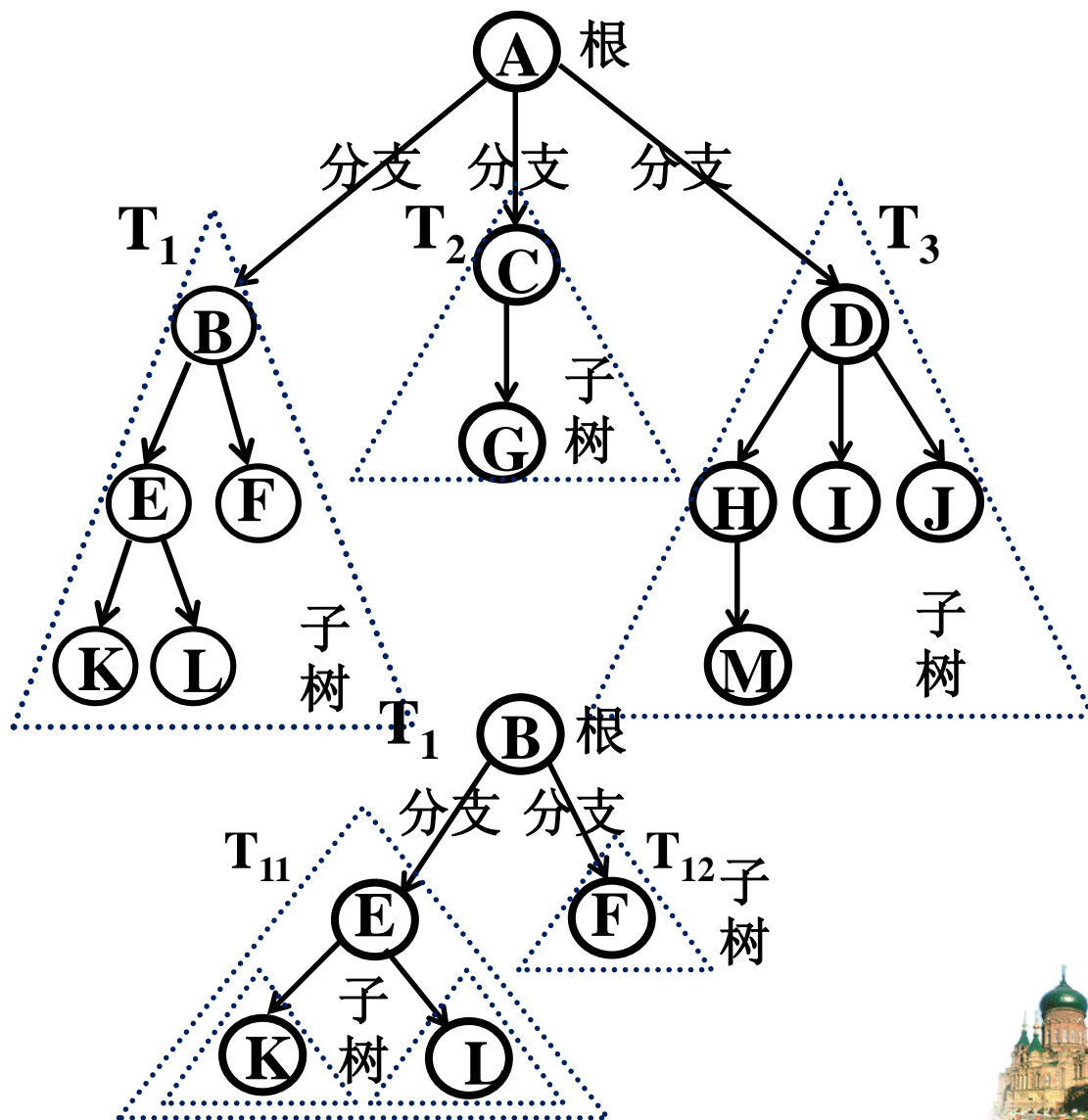
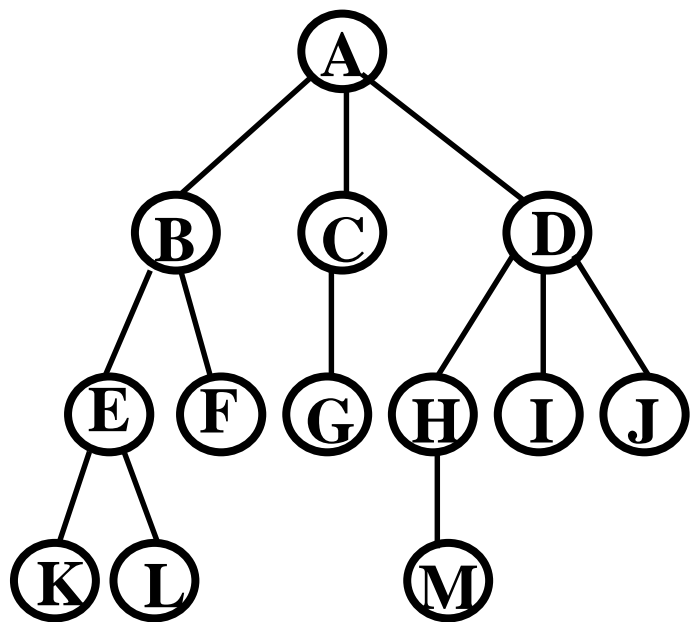


- 递归定义，但不会产生循环定义；
- 构造性定义便于树型结构的建立；
- 一株树的每个结点都是这株树的某株子树的根；



## 3.1 树与二叉树的基本术语 (Cont.)

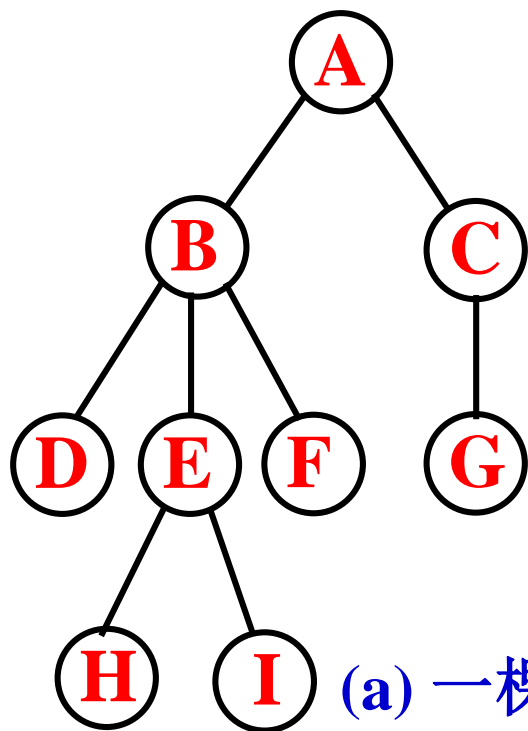
➡ 树的构造性递归定义:



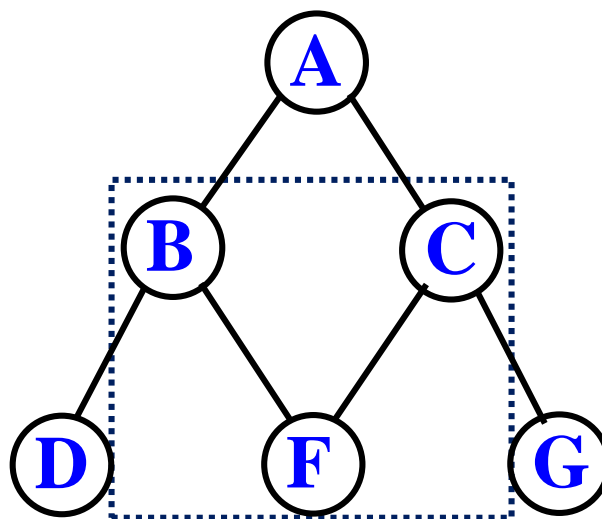
## 3.1 树与二叉树的基本术语 (Cont.)

➡ 树的逻辑结构特点:

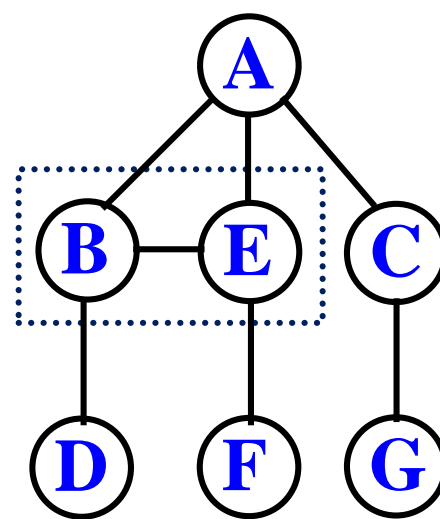
- 除根结点之外，每棵子树的根结点有且仅有一个直接前驱，但可以有**0**个或多个直接后继。
- 即**一对多**的关系，反映了结点之间的**层次关系**。



(a) 一棵树结构



(b) 一个非树结构



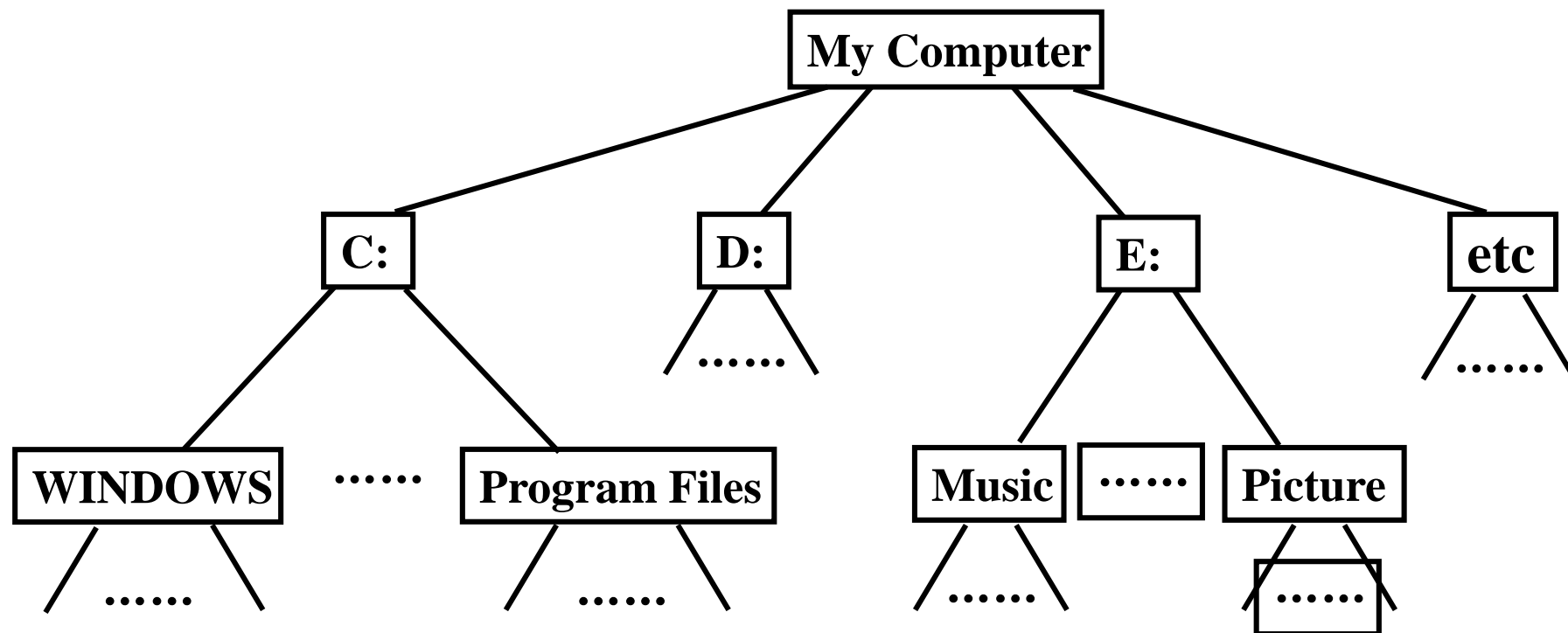
(c) 一个非树结构





## 3.1 树与二叉树的基本术语 (Cont.)

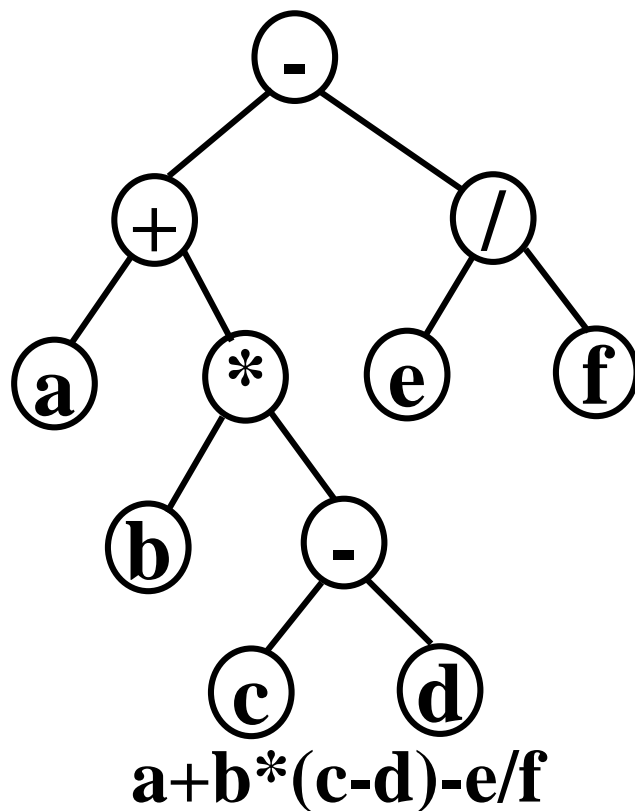
➡ 树型结构的应用示例----文件（目录）结构：





## 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:



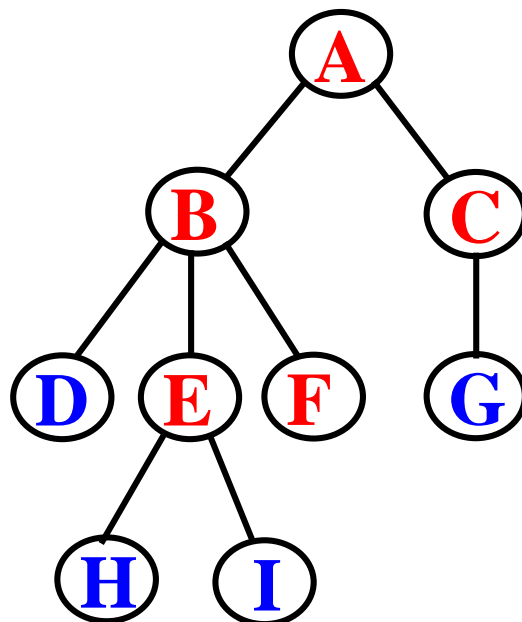




## 3.1 树与二叉树的基本术语 (Cont.)

### 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。

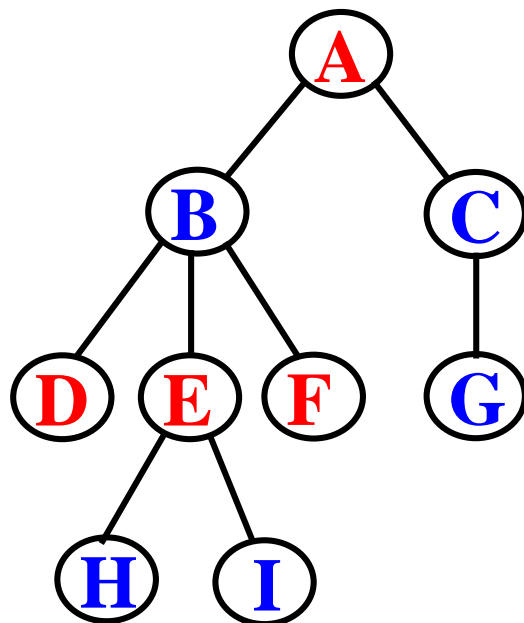




## 3.1 树与二叉树的基本术语 (Cont.)

### 基本术语:

- **结点孩子、双亲**: 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。



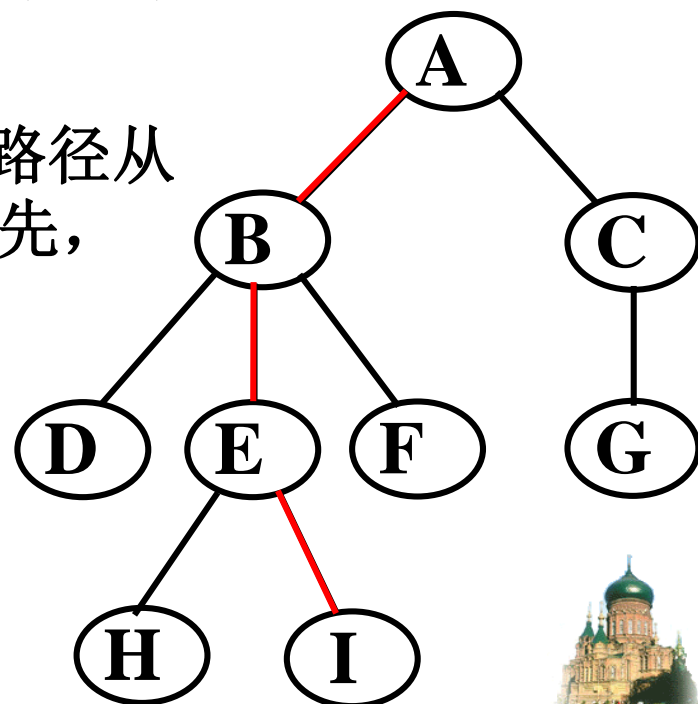


## 3.1 树与二叉树的基本术语 (Cont.)

### 基本术语:

■ **路(径)和路(径) 长度**: 如果树的结点序列  $n_1, n_2, \dots, n_k$  有如下关系: 结点  $n_i$  是  $n_{i+1}$  的双亲 ( $1 \leq i < k$ ), 则把  $n_1, n_2, \dots, n_k$  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。

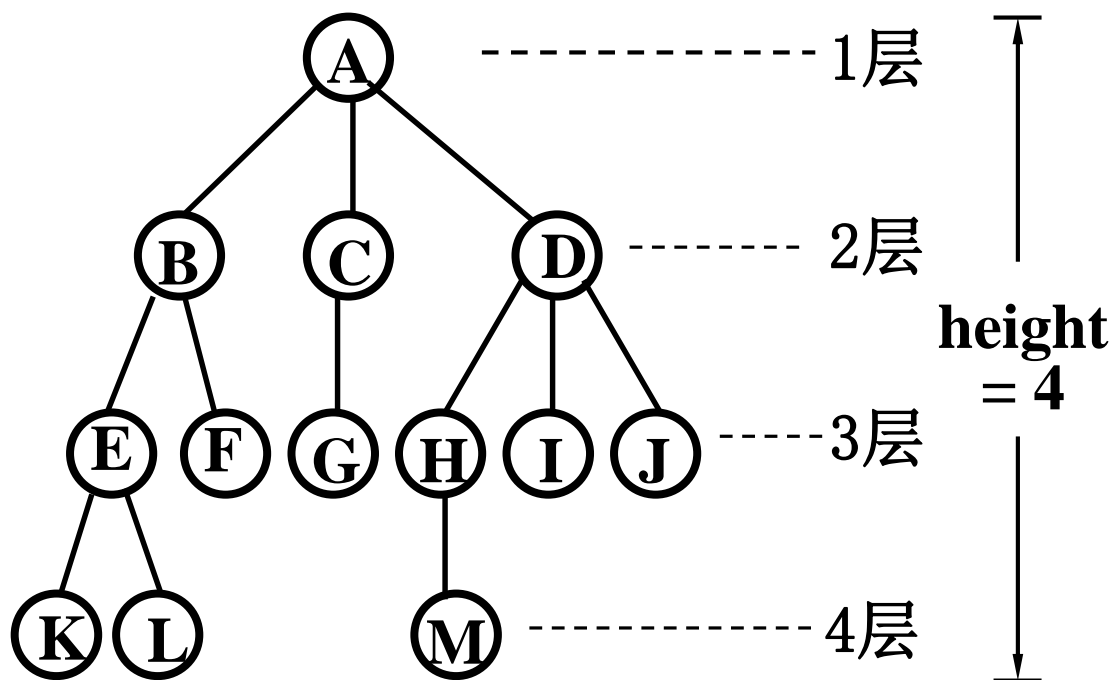




## 3.1 树与二叉树的基本术语 (Cont.)

### 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。
- **树的深度**: 树中所有结点的最大层数, 也称**高度**。

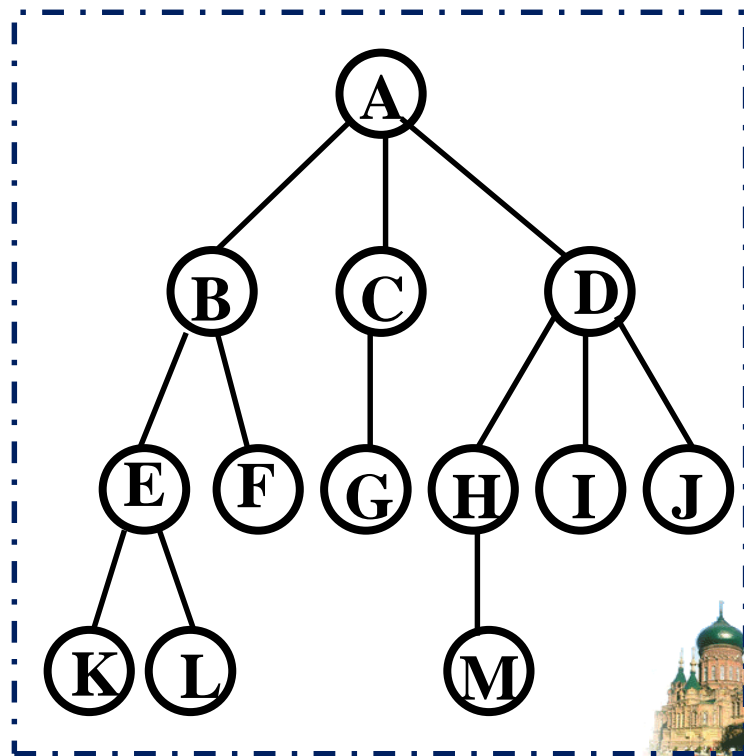
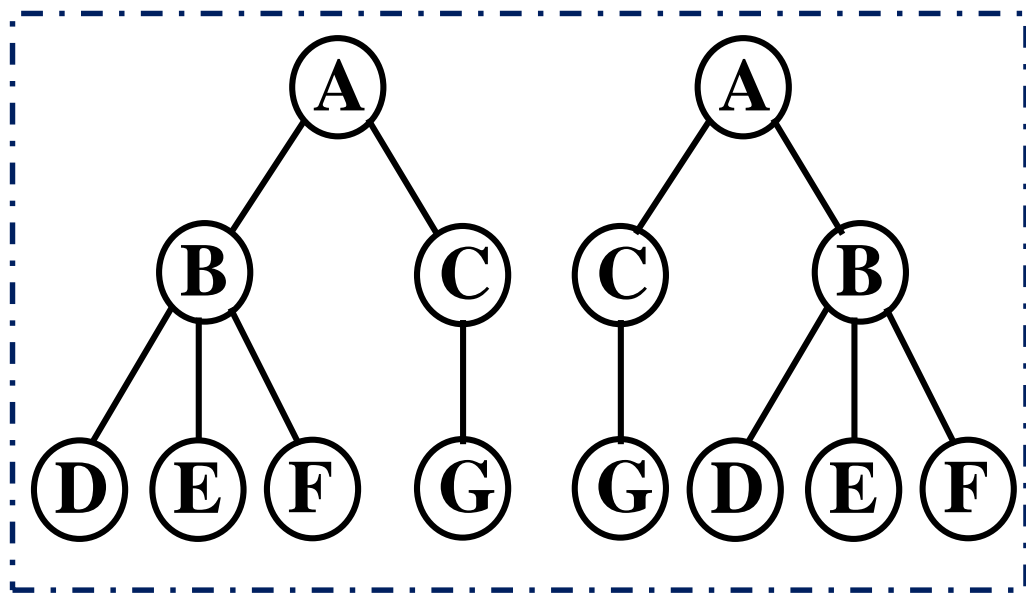




## 3.1 树与二叉树的基本术语 (Cont.)

### 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

第一个数据元素

无前驱

最后一个数据元素

无后继

其它数据元素

一个前驱,一个后继

一对一

#### 树型结构

根结点 (只有一个)

无双亲

叶子结点(可以有多个)

无孩子

其它结点

一个双亲,多个孩子

一对多





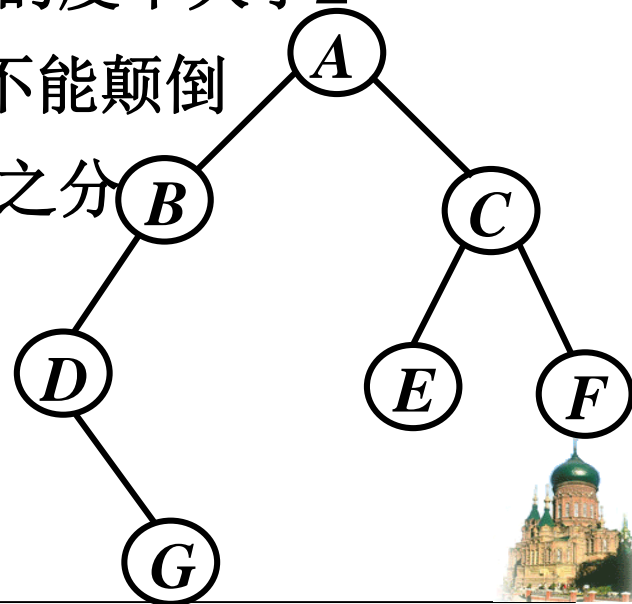
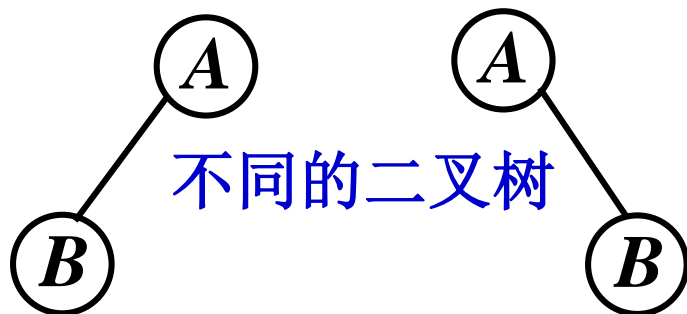
## 3.2 二叉树

### ➡ 二叉树(Binary Tree)的定义:

- 二叉树一个是由  $n$  ( $n \geq 0$ ) 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
- 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.)

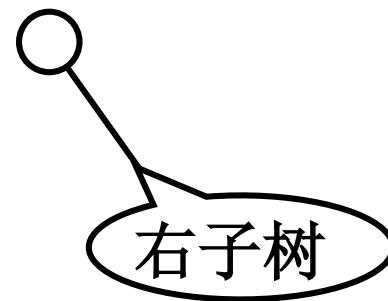
➡ 二叉树的基本形态:

$\Phi$

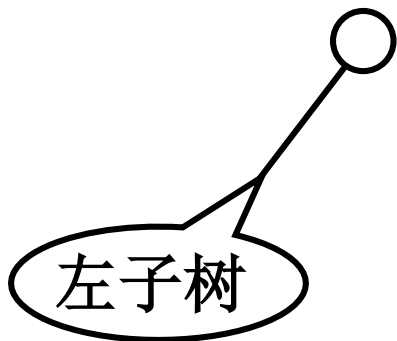
空二叉树



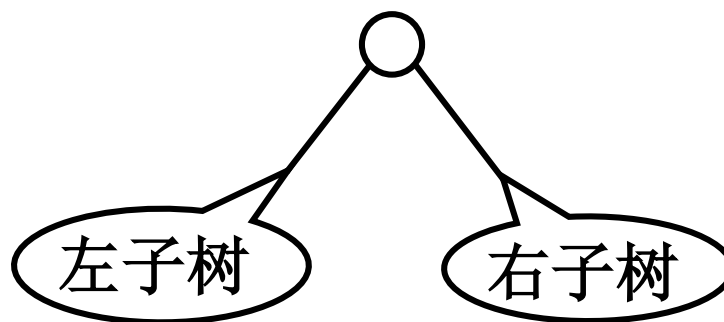
只有一个根结点



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

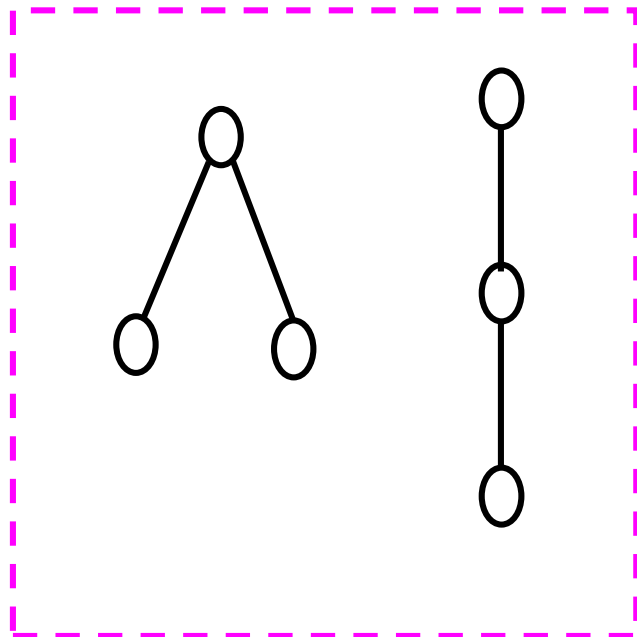




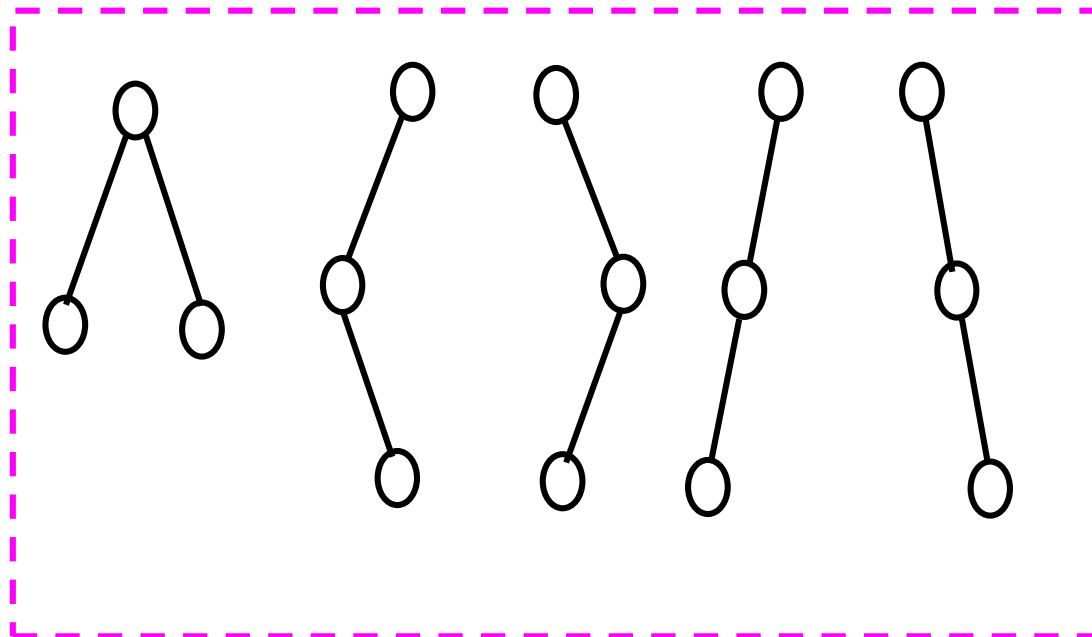


## 3.2 二叉树 (Cont.)

➡ 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

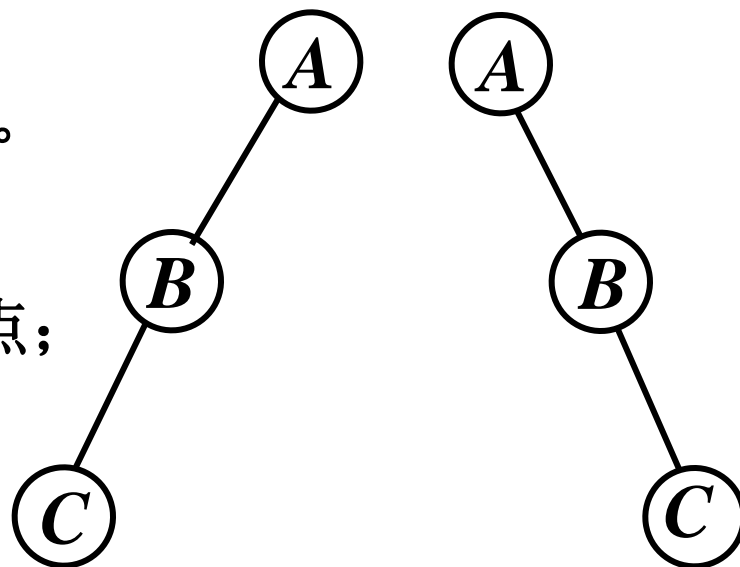
- 所有结点都只有右子树的二叉树称为右斜树；

#### 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

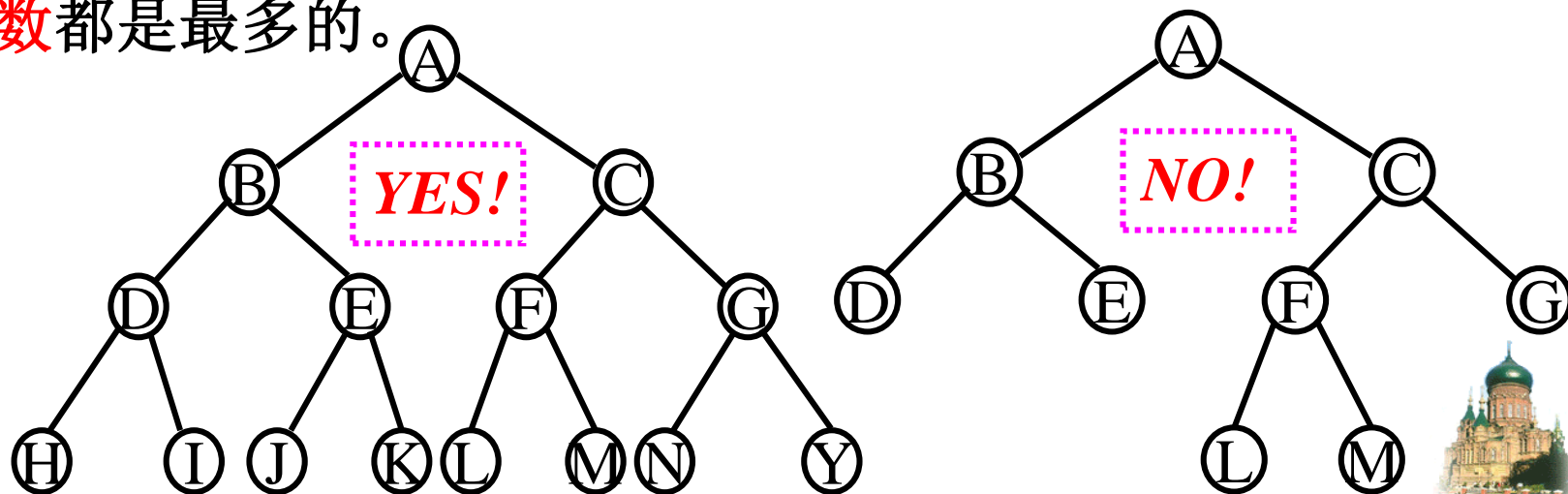
➤ 定义：高度为 $K$ 且有 $2^K-1$ 个结点的二叉树称为**满二叉树**。

➤ 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。



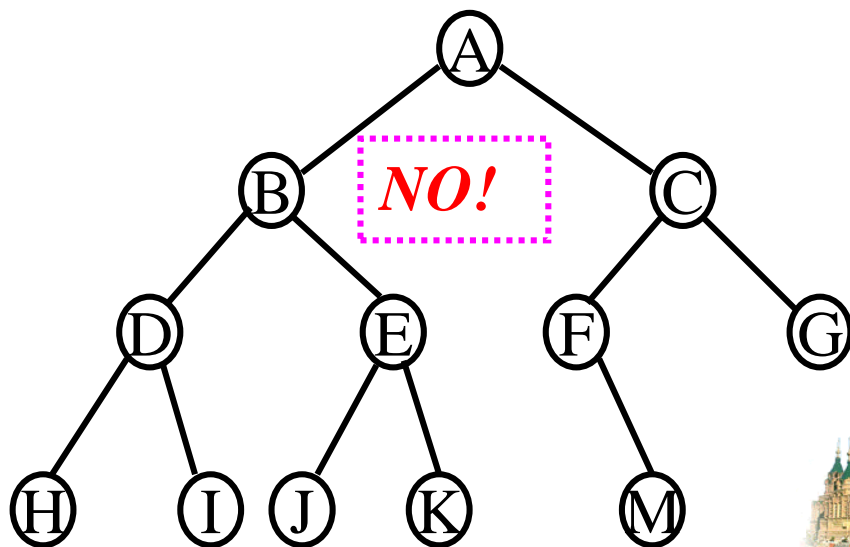
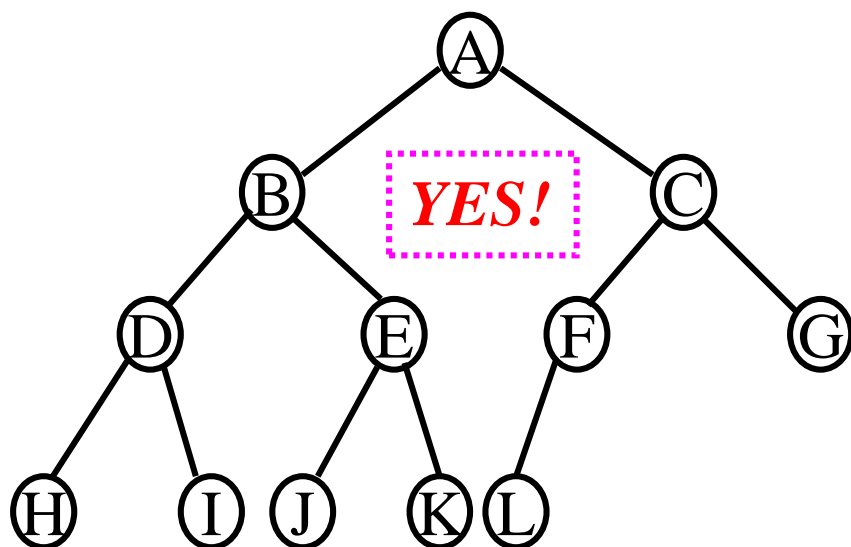


## 3.2 二叉树 (Cont.)

### 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 $k$ )为**完全二叉树**：

- 1. 所有的叶都出现在 $k$ 或 $k-1$ 层；
- 2.  $k-1$ 层的所有叶都在非终结结点的右边；
- 3. 除了 $k-1$ 层的最右非终结结点可能有一个（只能是左分支）或两个分支之外，其余非终结结点都有两个分支。



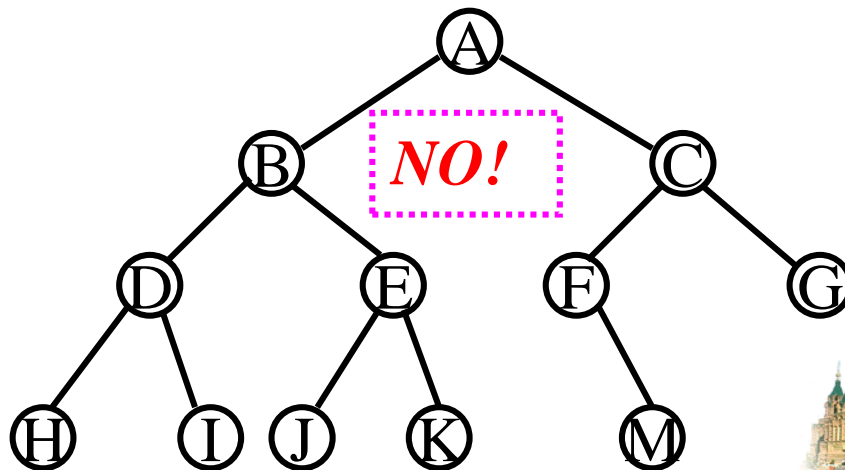
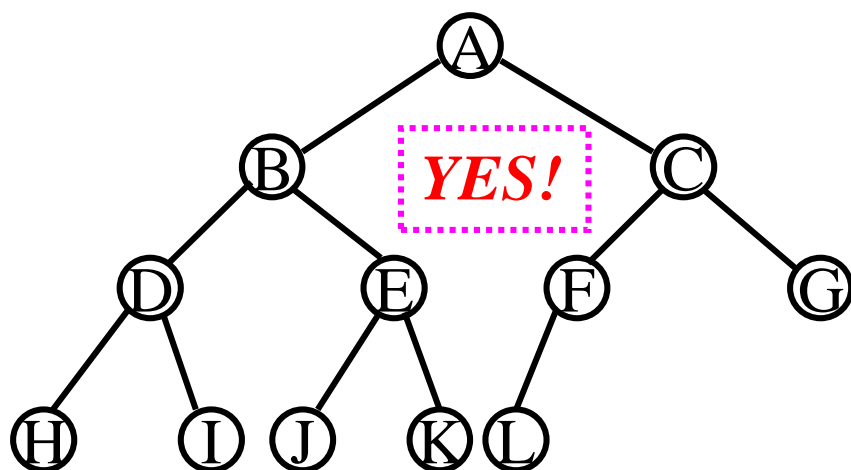


## 3.2 二叉树 (Cont.)

### 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 $k$ 的完全二叉树的前 $k-1$ 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二倍, 即  $2 \times 2^{k-1} = 2^k$ 。结论成立。  $\square$





## 3.2 二叉树 (Cont.)

### 二叉树的性质

#### 性质2:

■ 高度为  $k$  ( $k \geq 1$ ) 的二叉树最多有  $2^k - 1$  个结点，最少有  $k$  个结点。

■ 证明：由性质1可知，高度为  $k$  的二叉树中结点个数最多  
$$= \sum_{i=1}^k (\text{第 } i \text{ 层上结点的最大个数}) = 2^0 + 2^1 + 2^2 + \dots + 2^{k-1} = 2^k - 1;$$

另外，每一层至少要有一个结点，因此，高度为  $k$  的二叉树，至少有  $k$  个结点。□

➡ 高度为  $k$  且具有  $2^k - 1$  个结点的二叉树一定是满二叉树，

➡ 高度为  $k$  且具有  $k$  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  $\lceil \log_2(n+1) \rceil$

**证明:** 设完全二叉树的高度为  $k$ , 则根据**性质2**有:

$$2^{k-1} - 1 < n \leq 2^k - 1$$

前  $k-1$  层最多结点数

前  $k$  层最多结点数

变形  $2^{k-1} < n+1 \leq 2^k$

取对数  $k-1 < \log_2(n+1) \leq k$

因此有  $\lceil \log_2(n+1) \rceil = k \quad \square$



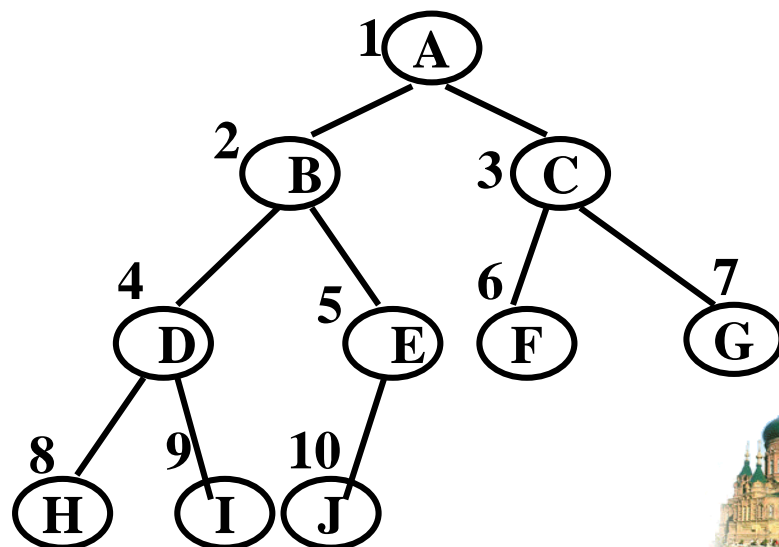
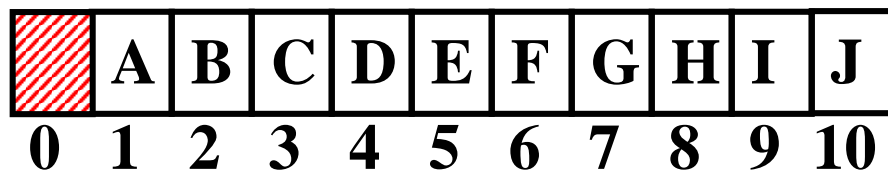


## 3.2 二叉树 (Cont.)

### 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:  $1, 2, \dots, n$ ，且使该编号对应于数组的下标，即编号为  $i$  ( $1 \leq i \leq n$ ) 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。

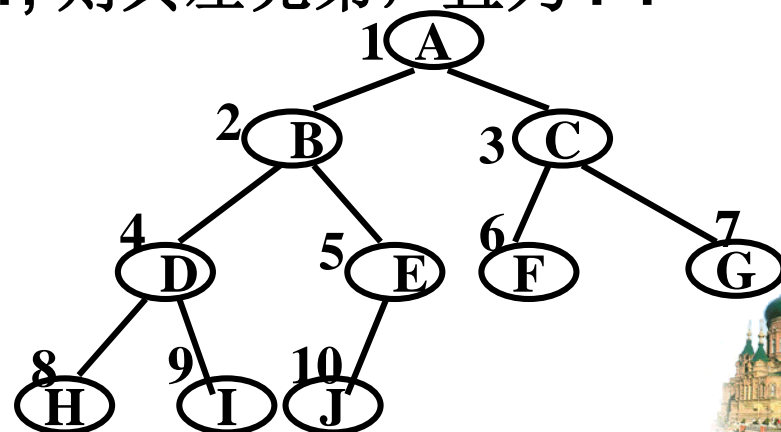
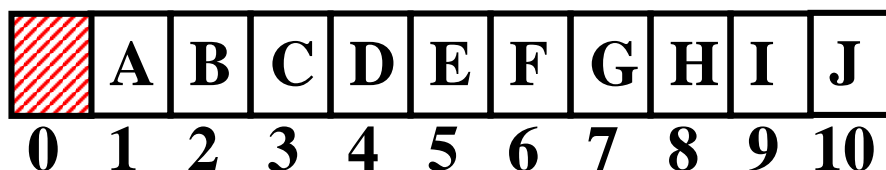




## 3.2 二叉树 (Cont.)

### 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  $\lfloor i/2 \rfloor$
- 若  $2*i \leq n$ , 则  $i$  有左儿子且为  $2*i$ ; 否则,  $i$  无左儿子。
- 若  $2*i+1 \leq n$ , 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  $i \leq n$  &&  $i \neq 1$ , 则其左兄弟, 且为  $i-1$





## 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
  - **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，

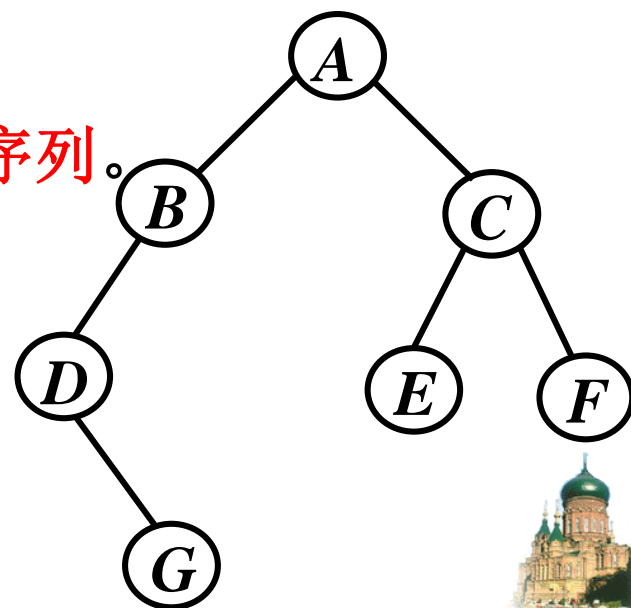
● ①访问根结点；

● 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，

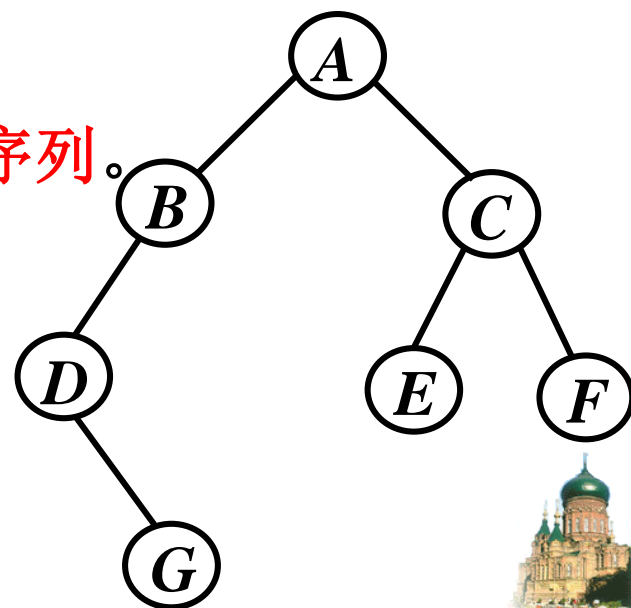
● ①中序遍历根结点的左子树；

● 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，

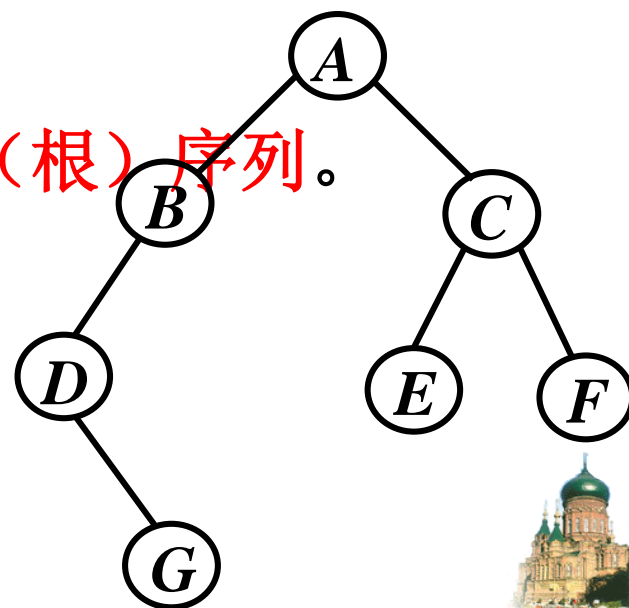
● ①后序遍历根结点的左子树；

● 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***





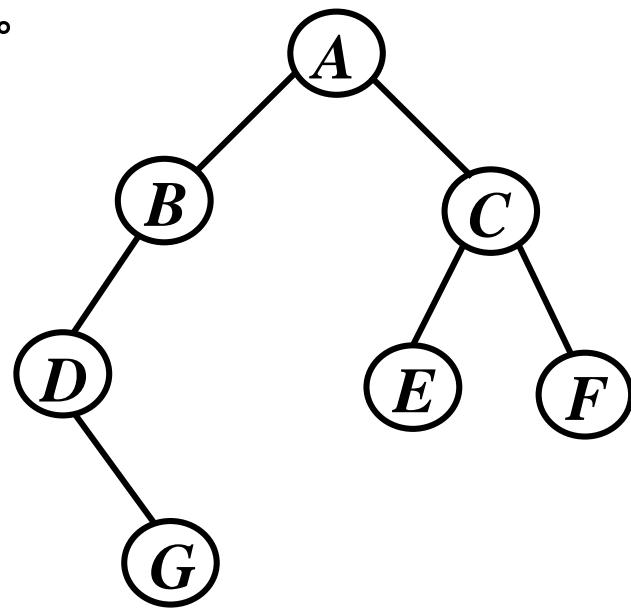
## 3.2 二叉树 (Cont.)

### 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***







## 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。



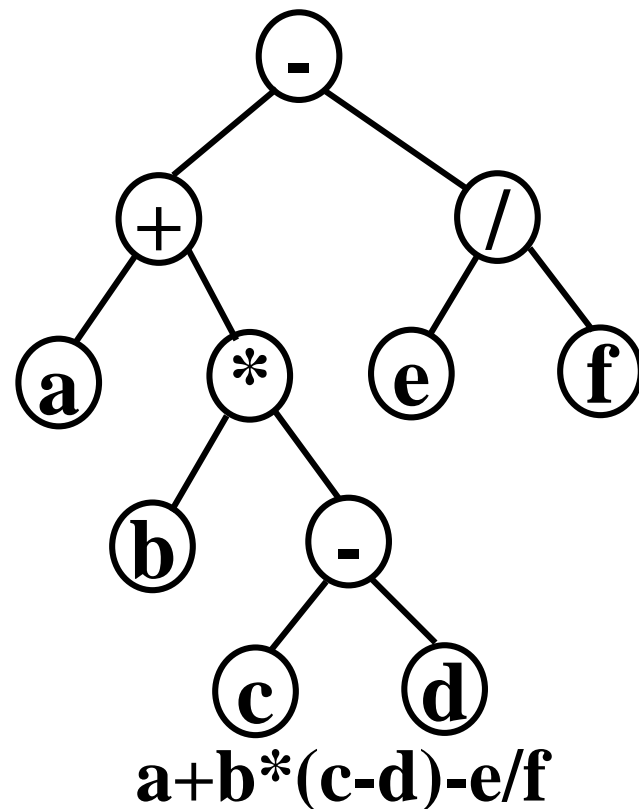


## 3.2 二叉树 (Cont.)

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 ) );
    }
}
```



先序序列: - + a \* b - c d / e f



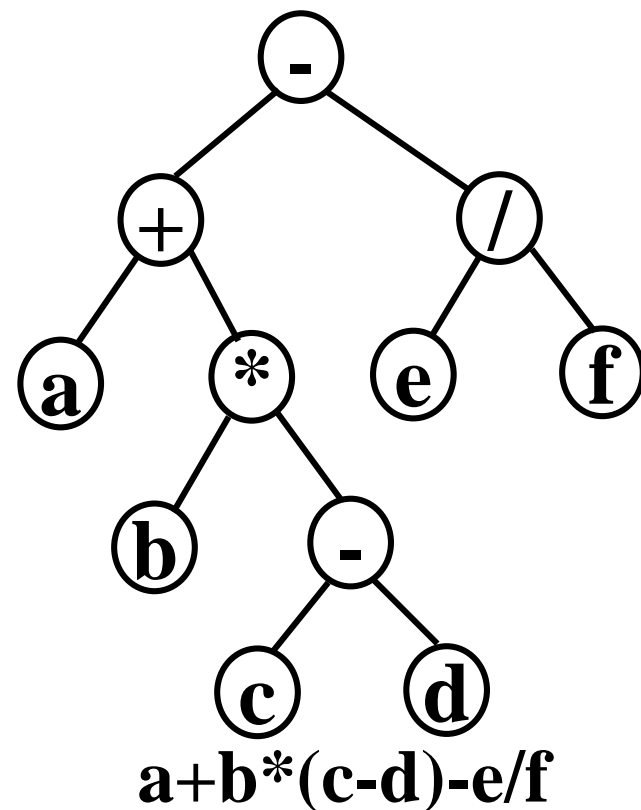


## 3.2 二叉树 (Cont.)

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 ) );
    }
}
```



中序序列:  $a + b * c - d - e / f$



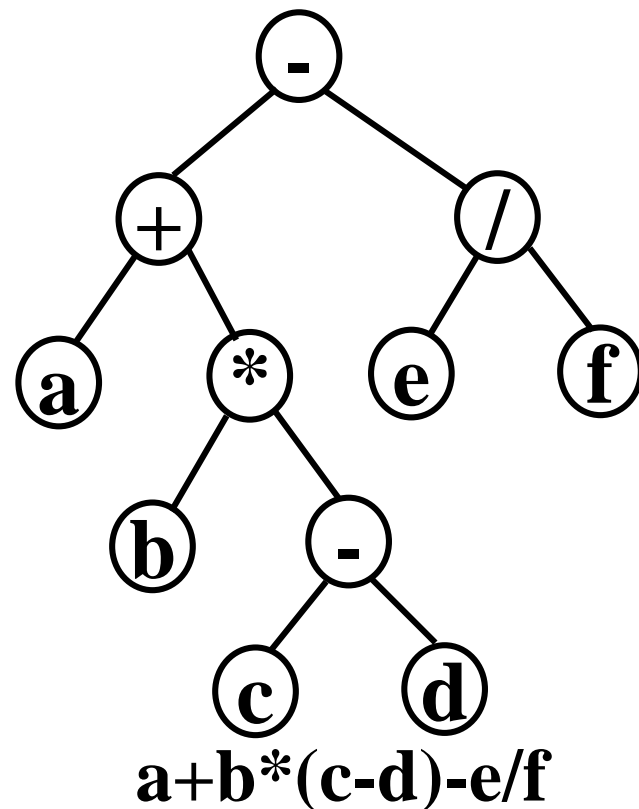


## 3.2 二叉树 (Cont.)

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 ) )
    {
        PostOrder ( Lchild ( BT ) );
        PostOrder ( Rchild ( BT ) );
        visit ( Data ( BT ) );
    }
}
```



后序序列:  $a\ b\ c\ d\ -\ *\ +\ e\ f\ /\ -$





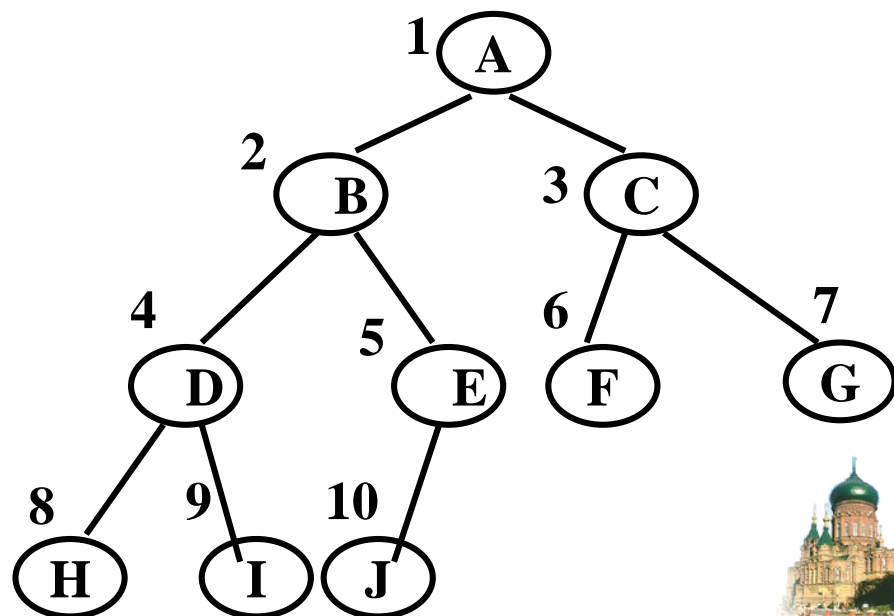
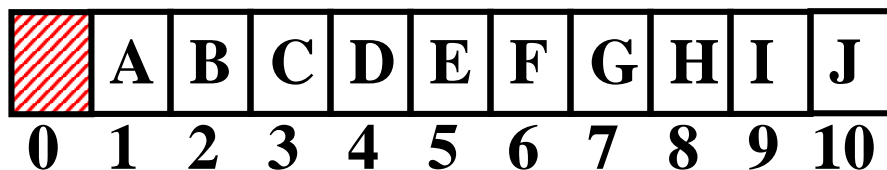
## 3.2 二叉树 (Cont.)

### 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：

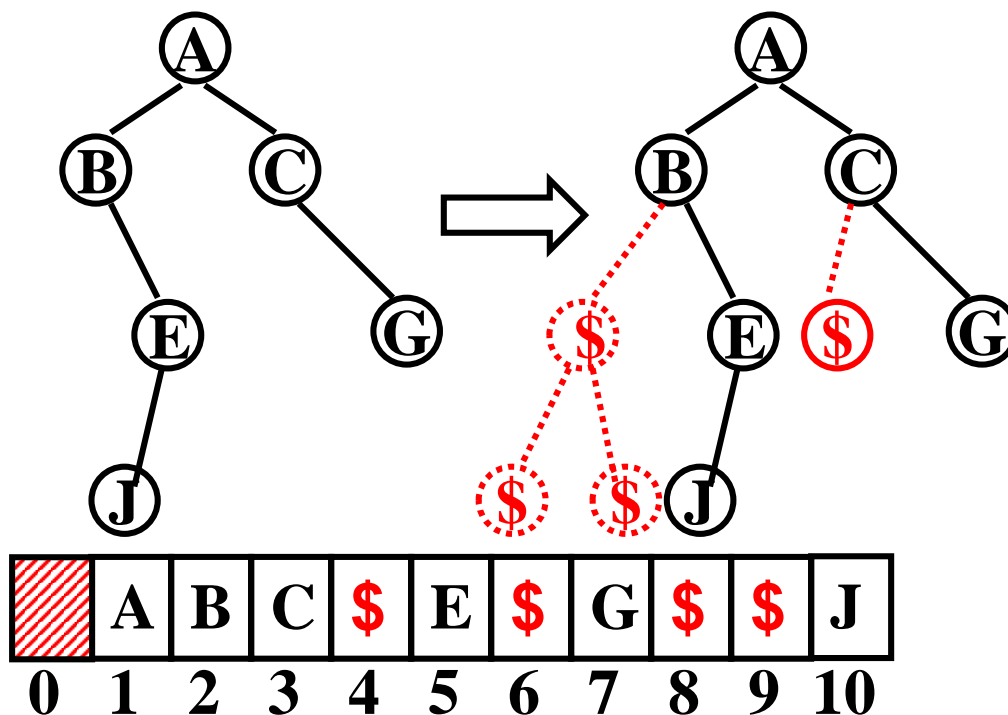




## 3.2 二叉树 (Cont.)

### 一般二叉树的顺序存储结构

- 实现：按完全二叉树的结点层次编号，依次存放二叉树中的数据元素
- 特点：结点间关系蕴含在其存储位置中



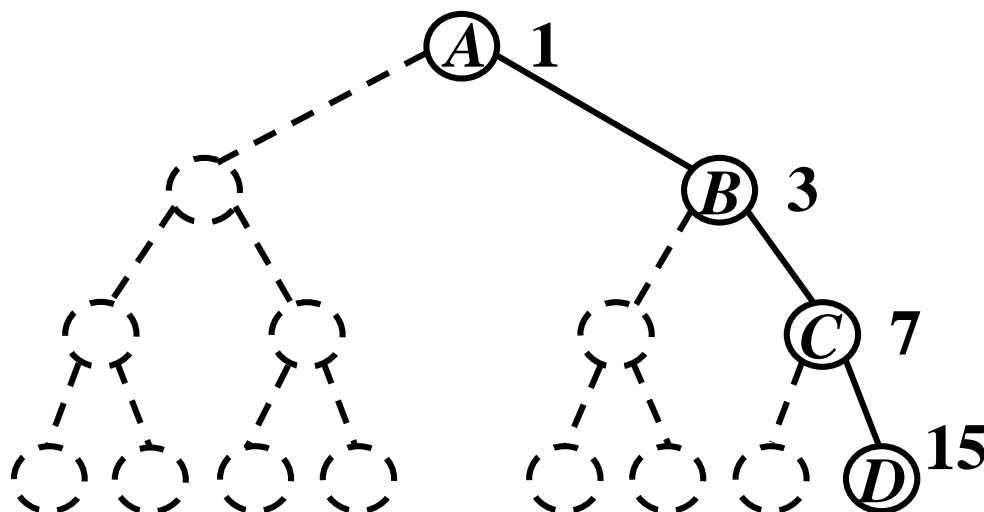


## 3.2 二叉树 (Cont.)

### ■ 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 $k$ 的右斜树， $k$ 个结点需分配 $2^k - 1$ 个存储单元。
- ◆ 需增加很多空结点，造成存储空间的浪费。





## 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。



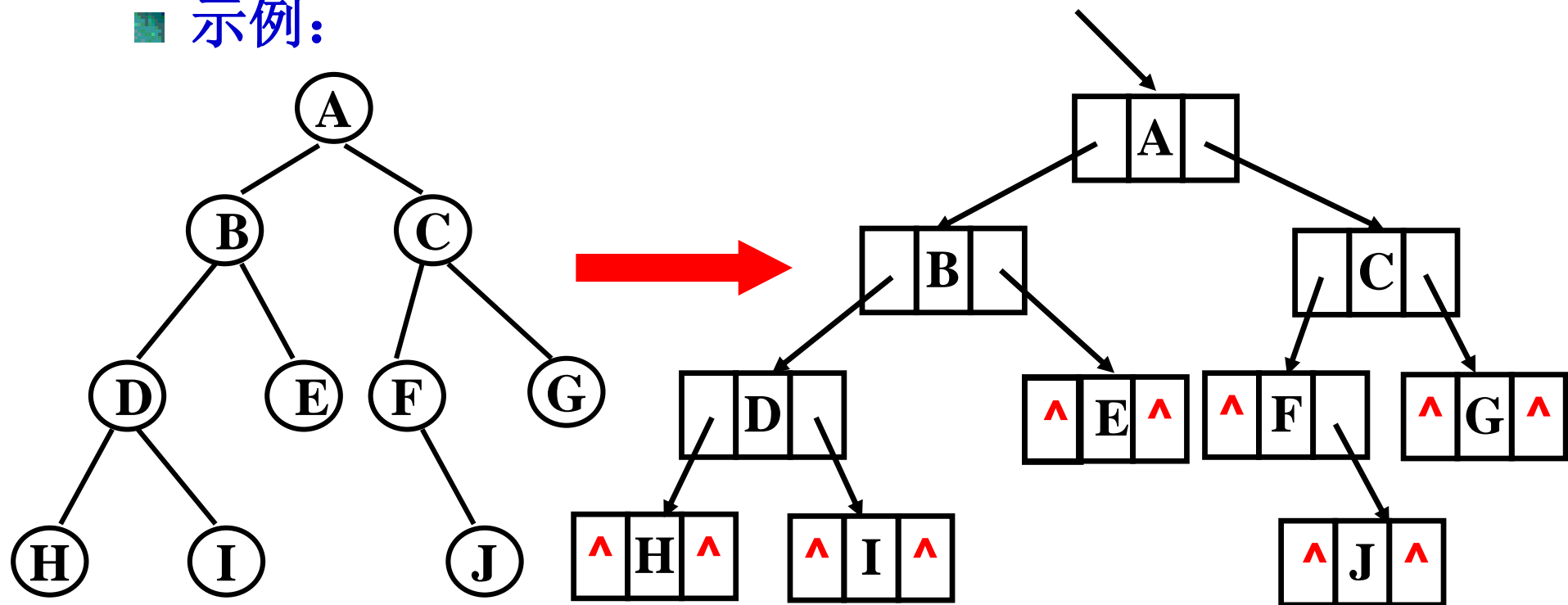




## 3.2 二叉树 (Cont.)

➡ 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?



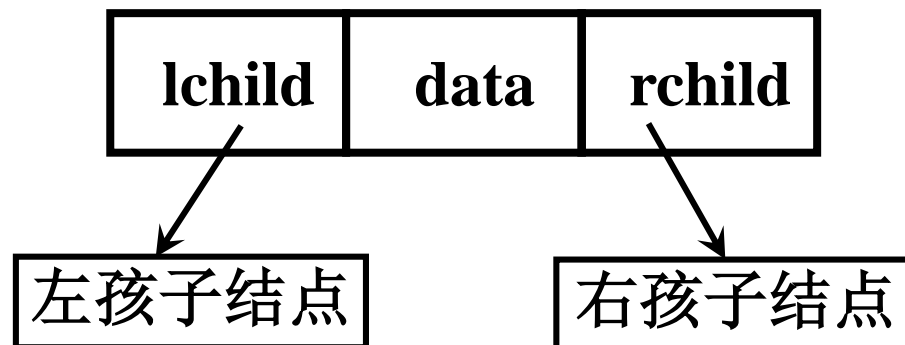


## 3.2 二叉树 (Cont.)

### ➡ 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 ;**

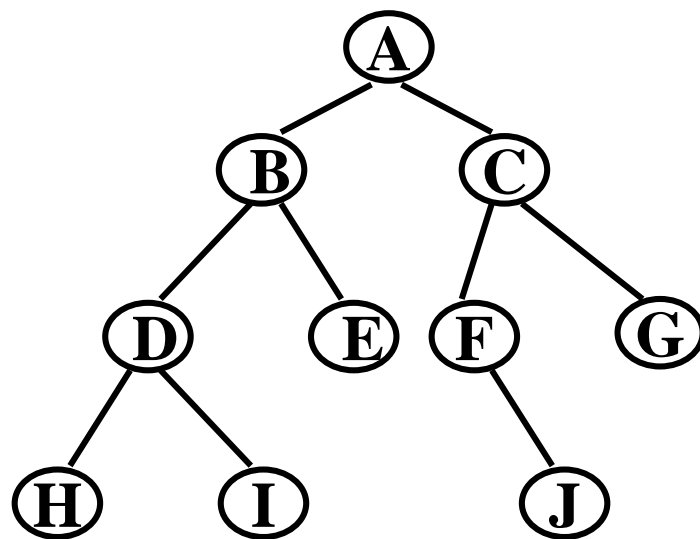
**root →data = v ;**

**root →lchild = ltree ;**

**root →rchild = rtree ;**

**return root ;**

}



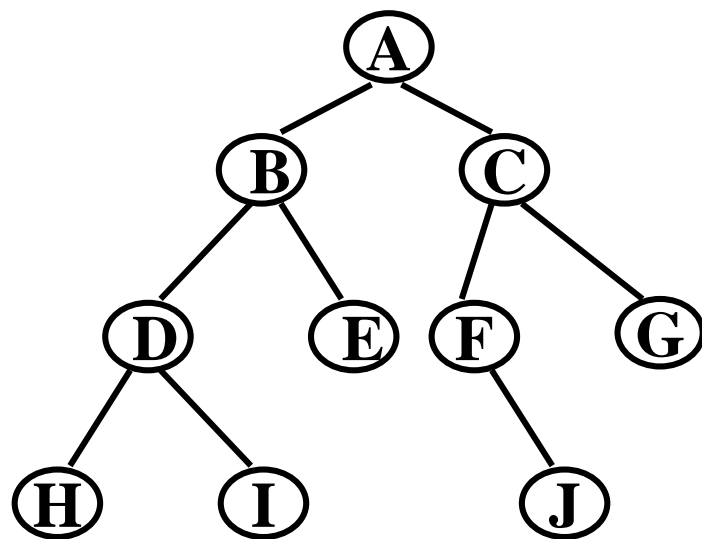


## 3.2 二叉树 (Cont.)

■ **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

```
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 → rchild=NULL;`

`s[ i ]=p;`

`if ( i == 1 ) bt = p ;`

`else {   j=i /2; // 父结点的编号`

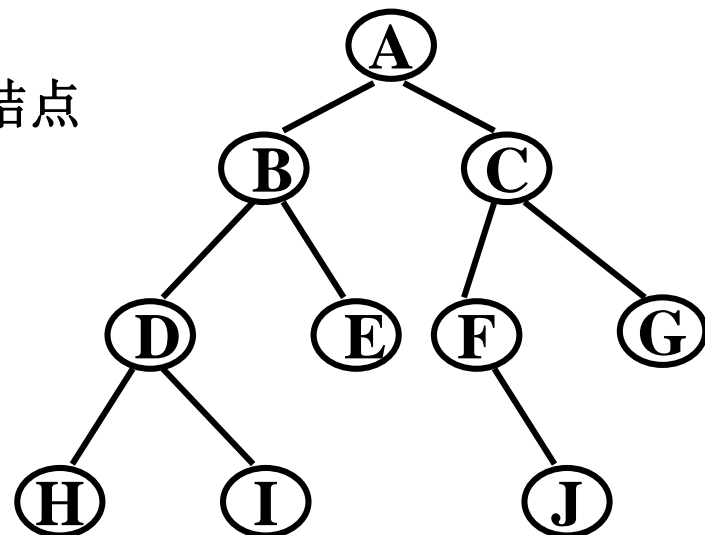
`if ( i %2==0 ) s[ j ]→lchild=p; // i 是 j 的左儿子`

`else         s[ j ]→rchild=p; // i 是 j 的右儿子`

`}cin >> i>>ch ;}`

`return bt;`

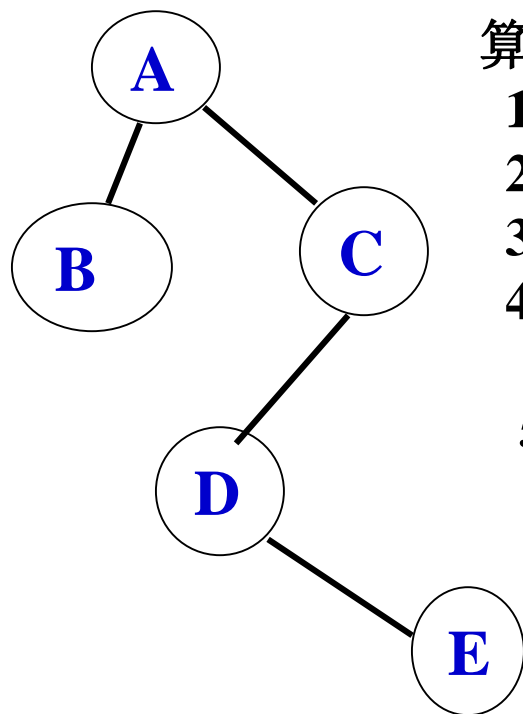
`}`





例1：已知二叉树的逻辑结构如下，试写出建立二叉树的算法。

二叉树用三元组表示 (data, parent, tag)，左图表示如下：  
(A,#,#),(B,A,L),(C,A,R),(D,C,L)  
(E,D,R),(#,#,#)结束标志



算法思想：

- 1、建立一个空树；
- 2、用一个队列存放输入的点；
- 3、每输入一个结点，建立并入队；
- 4、若其parent为‘#’，则为根，否则到队列中查找父结点，没找到出队；
- 5、根据读入的tag与双亲建立关系  
重复3—5。





```
typedef struct
    char data, parent, tag ;
    } Bnode;
typedef struct
    Btree ele[max];
    int front, rear;
    }Squeue;
```

```
Btree creat( )
{
    Bnode p;
    Squeue q;
    Btree root, s;
```

```
typedef struct node{
    char data;
    node *lc,*rc}*Btree;
```





```
root=new node;  
root=NULL;  
MakeNull(q);  
cin>>p.data>>p.parent>>p.tag;  
while(p.data!='#')  
{  
    s=new node;  
    s->data=p.data;  
    s->lc=s->rc=NULL;  
    q.rear=(q.rear+1)%max;  
    q.ele[q.rear]=s; //入队  
    if (p.parent=='#')  
        root=s;  
}
```







**else//查找父结点**

**{**

**while(!empty(q) && q.ele[q.front]->data!=p.parent)**

**q.front=(q.front+1)%max;//出队**

**if (q.ele [q.front]->data==p.parent)**

**if(p.tag=='L')**

**q.ele[q.front]->lc=s;**

**else**

**q.ele[q.front]->rc=s;**

**//else**

**cin>>p.data>>p.parent>>p.tag;**

**}//while**

**return root;**

**}**



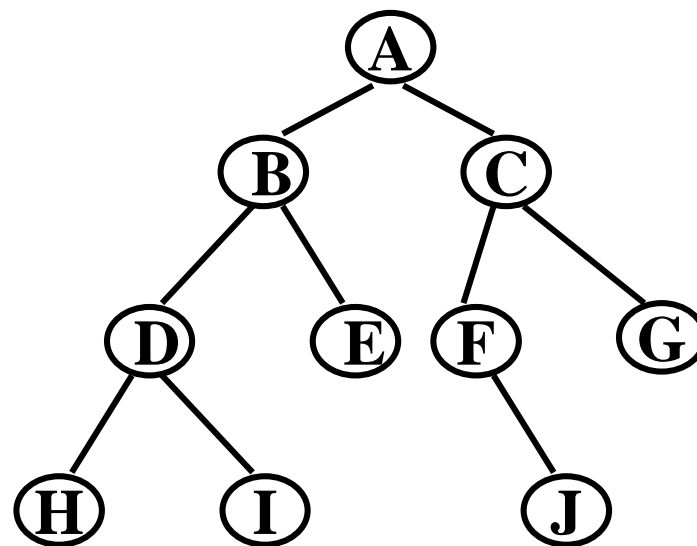


## 3.2 二叉树 (Cont.)

➡ 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 ) ;  
    }  
}
```



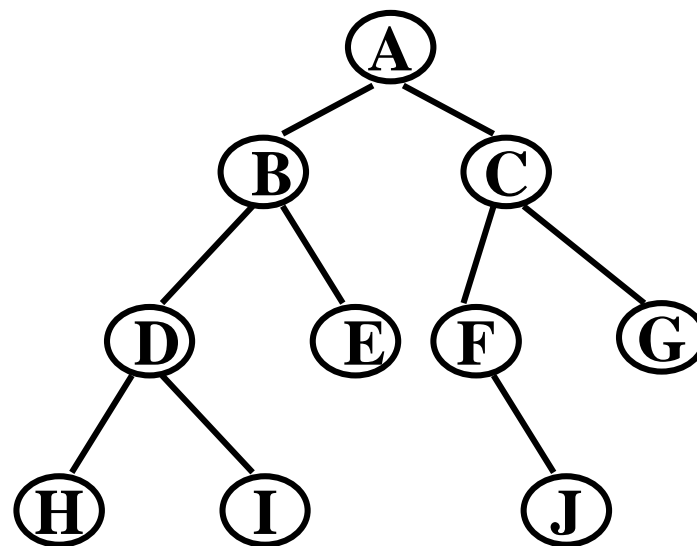


## 3.2 二叉树 (Cont.)

➡ 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 );  
    }  
}
```



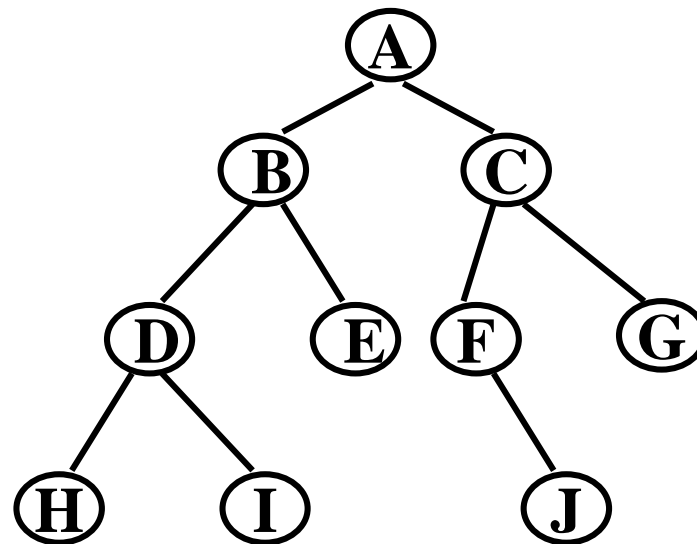


## 3.2 二叉树 (Cont.)

➡ 二叉树左右链存储结构下的递归遍历算法.

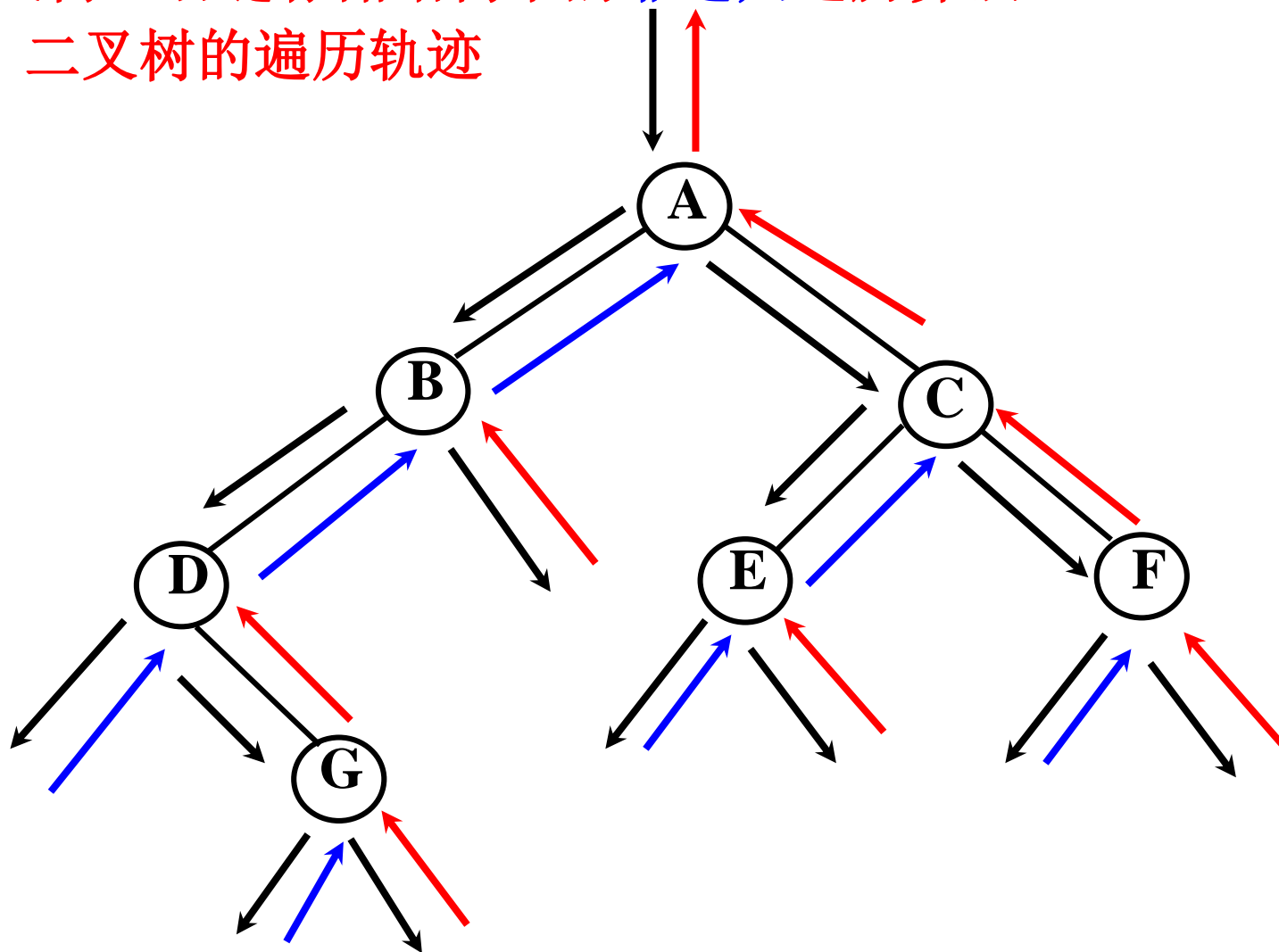
### ■ 后序遍历

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



## ➡ 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;

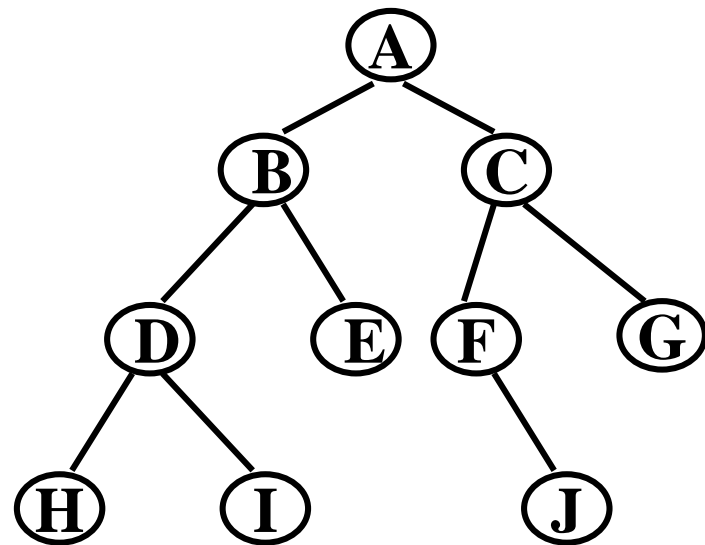
2.1.2 将指针root的值保存到栈中;

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 准备遍历root的右子树;





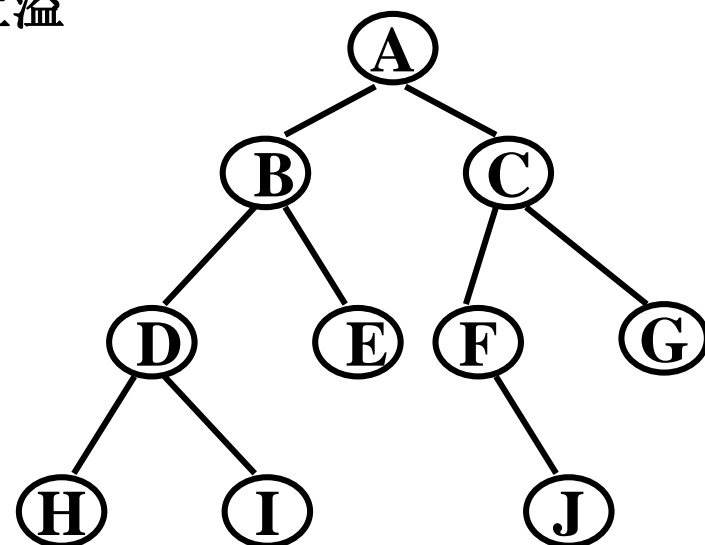
## 3.2 二叉树 (Cont.)

### ■ 先序遍历非递归算法---栈顶保存当前结点左子树

```

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;
        }
    }
}

```



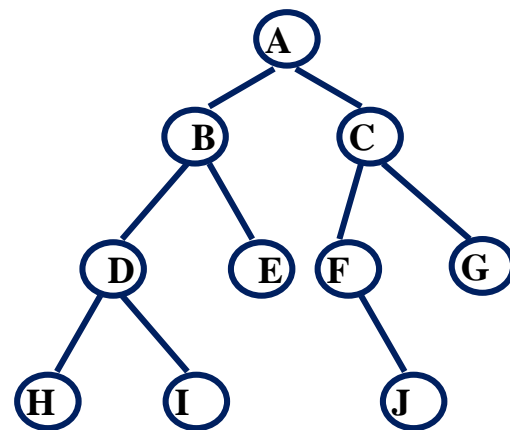


## 先序遍历的非递归算法——栈顶保存当前结点的右子树

```

void PreOrder( Btree T )
{ Stack S; MakeNull(S); //递归工作栈
  struct node* p = T;
  while ( p != Null ) {
    cout << p->data << endl;
    if ( p->rchild != Null )
      Push (S, p->rchild );
    if ( p->lchild != Null )
      p = p->lchild;    //进左子树
    else {p=Top(S);Pop(S);} //左子树空, 访问右子树 }
  }

```

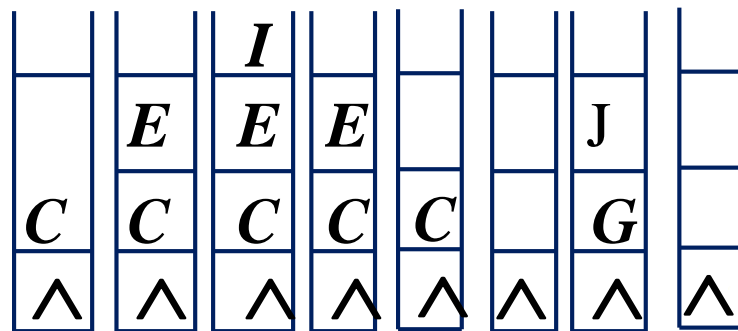


访问    **A   B   D   H   I   E   C   F   J   G**

进栈    **C   E   I   \*   \*   \*   G   J**

P向左前进   **B   D   H   -   -   -   F**

退栈    **+   +   +   I   E   C   J   G**



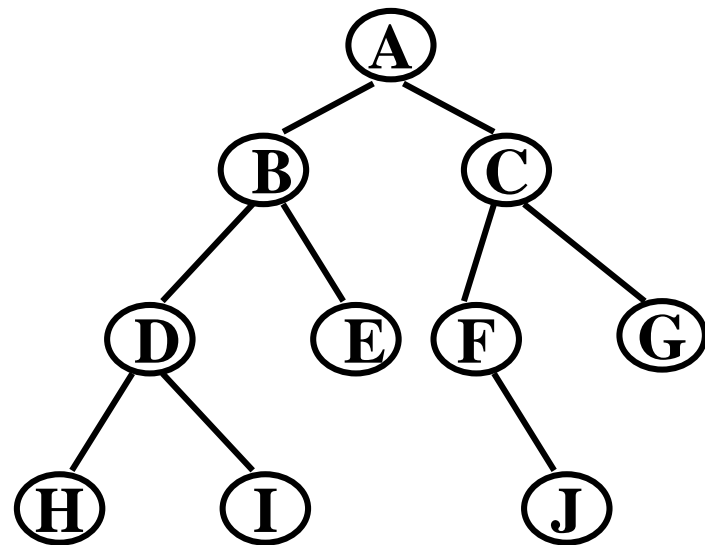




## 3.2 二叉树 (Cont.)

### ■ 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 准备遍历root的右子树;

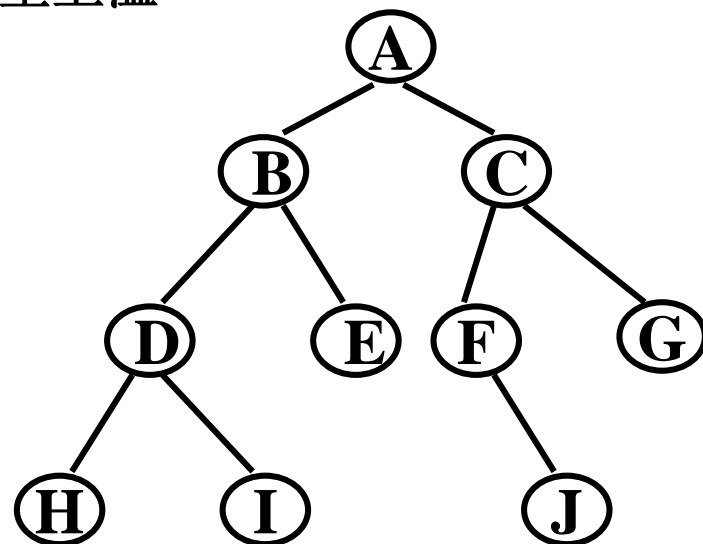




## 3.2 二叉树 (Cont.)

### ■ 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;
    }
  }
}
```

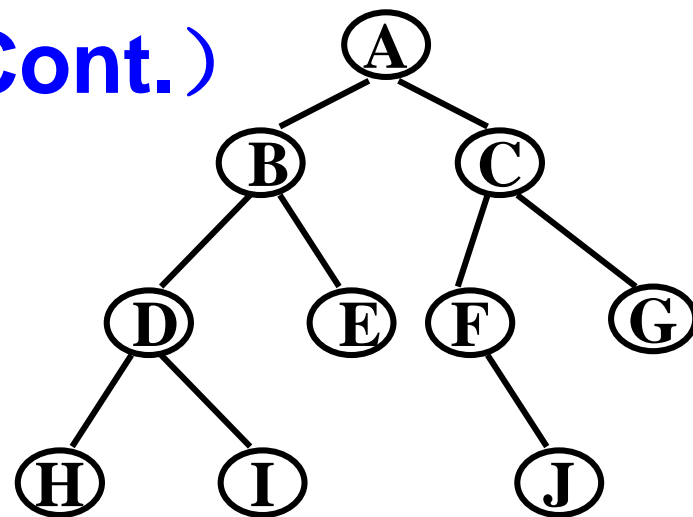




## 3.2 二叉树 (Cont.)

### ■ 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 非空且栈顶元素的标志为2 时, 出栈并输出栈顶结点;
  - 2.3 若栈非空, 将栈顶元素的标志改为2, 准备遍历栈顶结点的右子树;





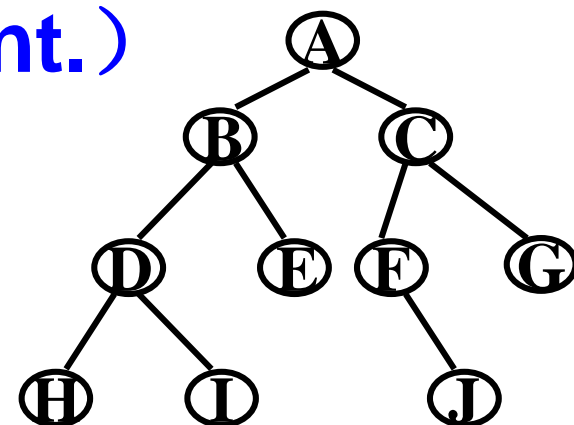
## 3.2 二叉树 (Cont.)

### 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) {
            top++;
            s[top].ptr=root;
            s[top].flag=1;
            root=root->lchild; }
        while (top!= -1 && s[top].flag==2)
            cout<<s[top--].ptr->data;
        if (top!= -1) {
            s[top].flag=2;
            root=s[top].ptr->rchild; }
    }
}

```





**void PostOrder (Btree t) (不设标志, 设一变量)**

```
{  
    Btree p, pr; Stack s;  
    MakeNull(s);  
    p=t;  
    while(p!=Null||!Empty(s))  
    {  
        while (p!=Null)  
        { Push(p,s);  
            pr=p->rc;p=p->lc;  
            if(p==Null) p=pr;  
        }  
        p=Pop(s);visit(p->data);  
        if(!Empty(s)&&Top(s)->lc==p)  
            p=Top(s)->rc;  
        else p=Null;  
    }  
}
```





思考题：在二叉树中增加两个域parent父结点, flag标志, 写出不用栈进行后序遍历的非递归算法

**flag**用于区分在遍历过程中达到该点时的走向（初始时每个结点的flag均为0）。

```
struct node {  
    char data;  
    node *lc,*rc,*parent;  
    int flag;  
};
```





```
struct node {      char data;      node *lc,*rc,*parent;
    int flag;
};

void PostOrder ( node *t) //后序遍历二叉树/
{
    node *p;
    p=t; //t指向二叉树的根, 建立时, 所有结点标志为0/
    while( p!=Null)
    switch (p->flag)
    {
        case 0:  p->flag=1;
                  if (p->lc!=Null) p=p->lc;
                  break;
        case 1:  p->flag=2;
                  if (p->rc!=Null) p=p->rc;
                  break;
        case 2:  p->flag=0;
                  cout<<p->data;
                  p=p->parent;
                  break;
    }
}
```





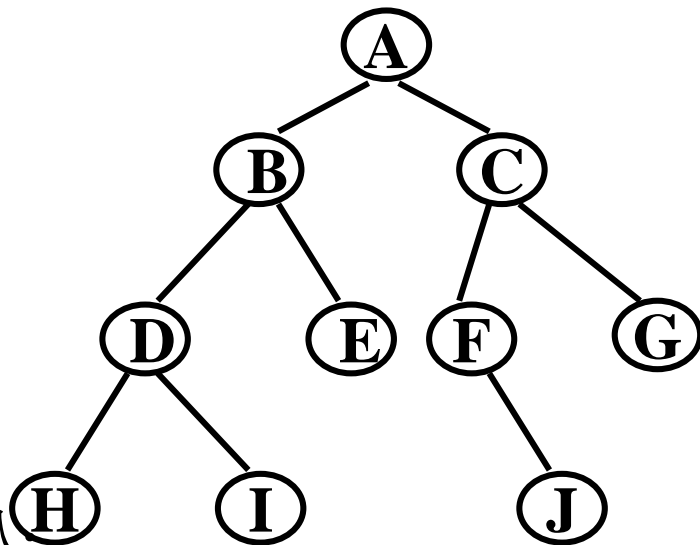
## 3.2 二叉树 (Cont.)

### ■ 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];
```

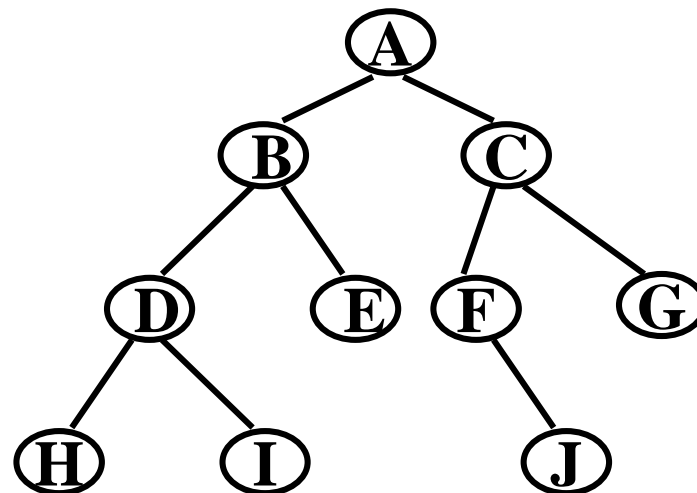
```
    cout<<q->data;
```

```
    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

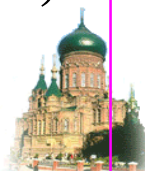
- 二叉树的遍历是二叉树各种操作和算法的基础，遍历算法中对每个结点的“访问”操作可以是对结点进行的各种处理
- 根据遍历算法的框架，适当修改访问操作的内容，可以派生出很多关于二叉树的应用算法。
- 因此，二叉树遍历算法是有关二叉树算法中最核心的算法。

### ➡ 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 ( Btree T )**

```
{  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 PreOrder(Btree T )
```

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

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





## 3.2 二叉树 (Cont.)

➡ 求二叉树宽度算法(宽度: 二叉树各层节点个数的最大值)

```
void Btrwidth ( Btree T)
{ int front=-1,rear=-1,count=0,max=0,right;
  if (T!=Null)
    q[++rear]=T; max=1; right=rear;
  while(front!=rear)
    { T=q[++front];
      if(T->lc!=Null)q[++rear]=T->lc; count++;
      if(T->rc!=Null) q[++rear]=T->rc; count++;
      if(front==right)
        { if(max<count)max=count; count=0;right=rear;}
    }
}
```

```
struct node {
    struct node *lc ;
    struct node *rc ;
    datatype data ;
} ;
typedef node * Btree ;
```

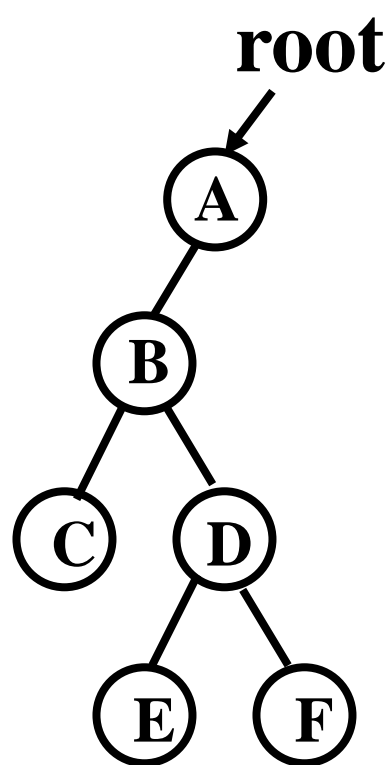




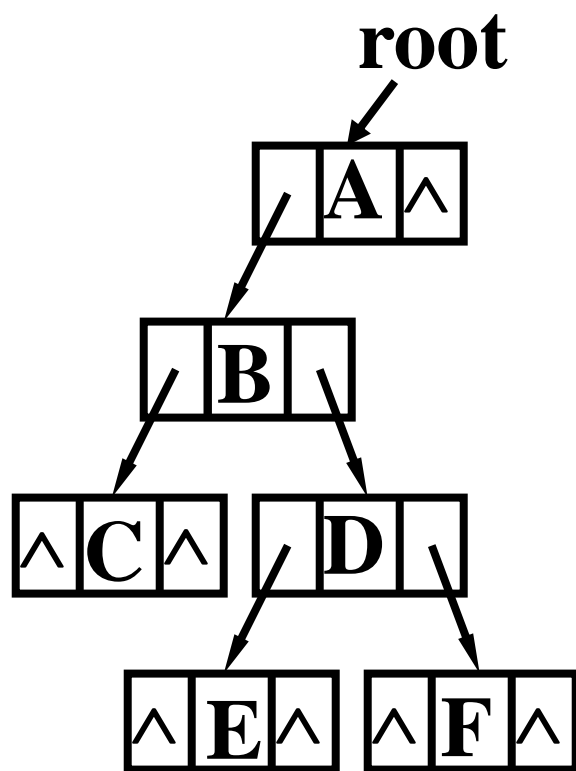
## 3.2 二叉树 (Cont.)

### ➡ 二叉树的其他链式存储结构----动态三叉链表

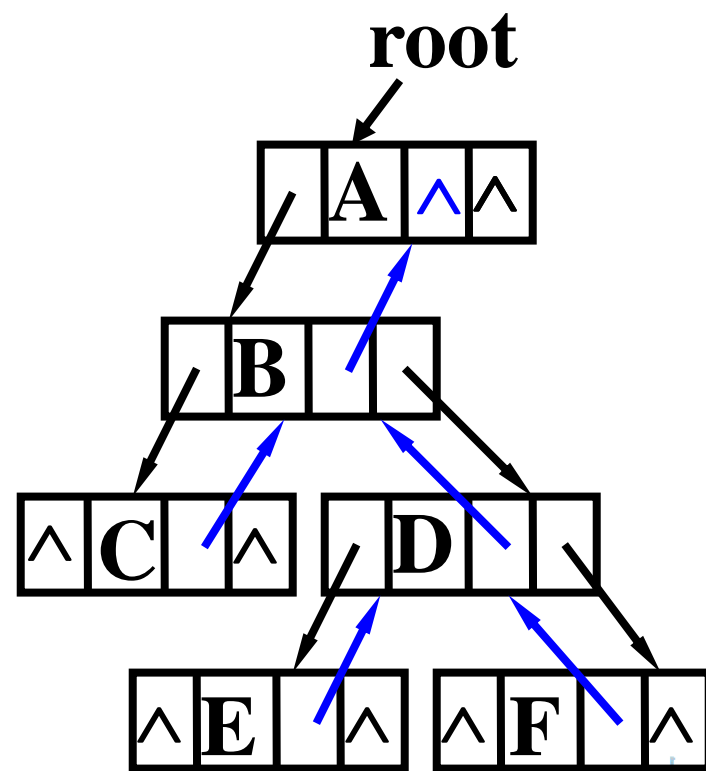
■ 在二叉链表的基础上增加了一个指向双亲的指针域。



二叉树



动态二叉链表



动态三叉链表

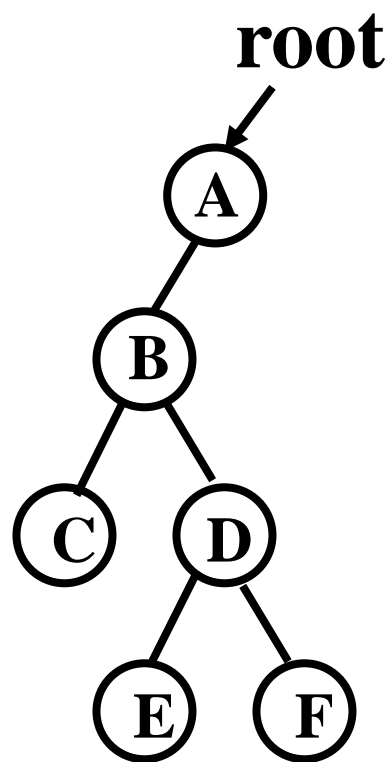






## 3.2 二叉树 (Cont.)

➡ 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 $n$  ( $n \geq 1$ ) 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。

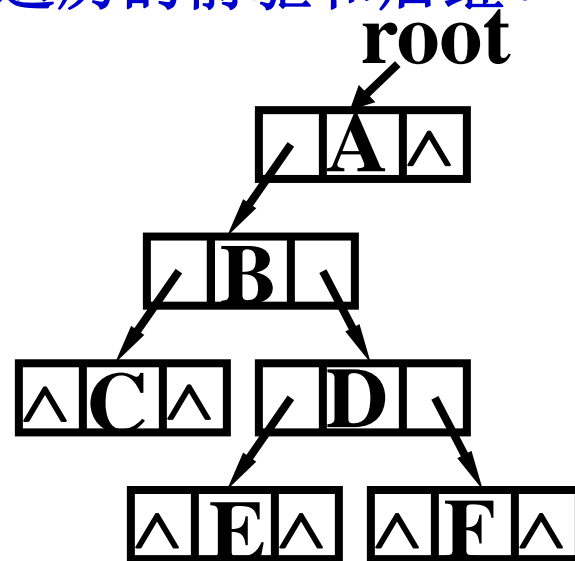
➤ 在二叉链表中如何找某个结点的某种遍历的前驱和后继？

■ 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

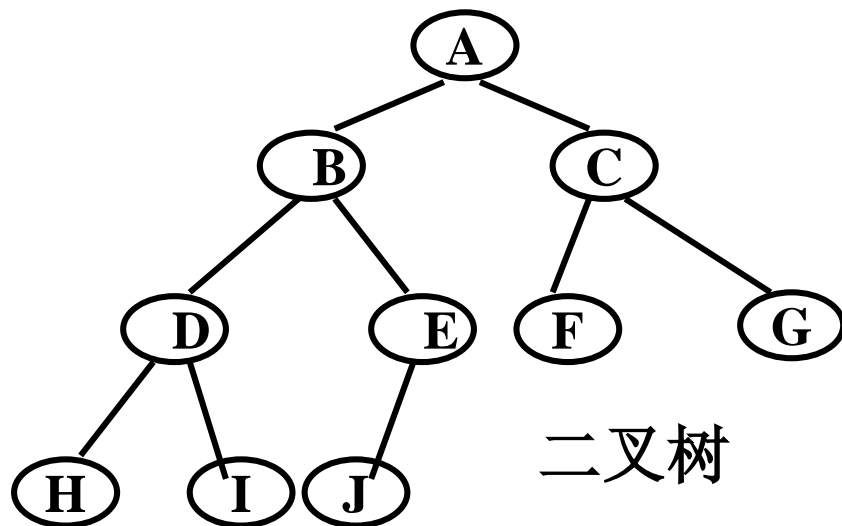
- 若结点 $p$ 有左孩子，则 $p \rightarrow lchild$ 指向其左孩子结点，否则令其指向其（先序、中序、后序、层序）前驱；
- 若结点 $p$ 有右孩子，则 $p \rightarrow rchild$ 指向其右孩子结点，否则令其指向其（先序、中序、后序、层序）后继；
- ➡ 如何区分指针是指向其左/右孩子的指针还是指向某种遍历的前驱/后继？
  - 在每个结点中增加两个标志位，以区分该结点的两个链域是指向其左/右孩子还是指向某种遍历的前驱/后继。





## 3.2 二叉树 (Cont.)

### 二叉树的线索链表存储结构----线索二叉树



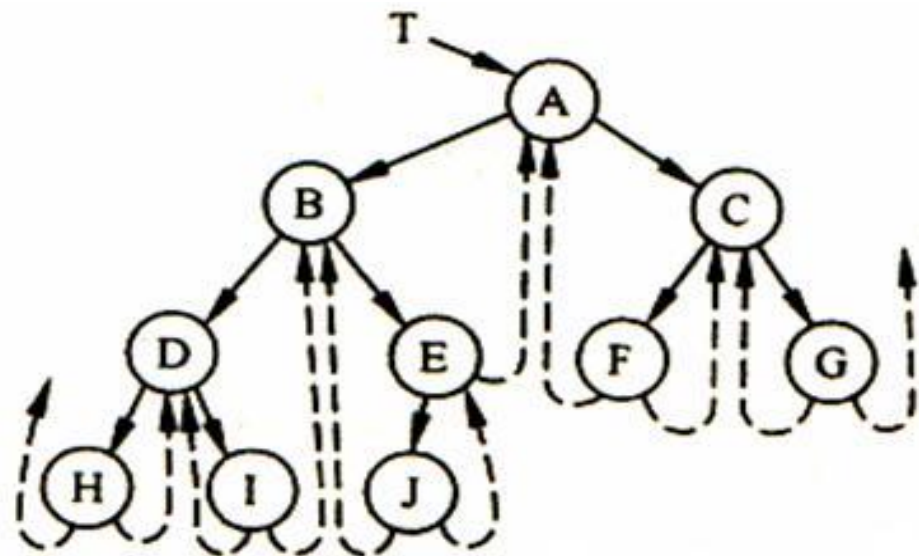
二叉树

#### 结点结构

lchild	ltag	data	rchild	rtag
--------	------	------	--------	------

$p \rightarrow ltag = \begin{cases} \text{True} & p \rightarrow lchild \text{ 指向左孩子} \\ \text{False} & p \rightarrow lchild \text{ 指向 (中序) 前驱} \end{cases}$

$p \rightarrow rtag = \begin{cases} \text{True} & p \rightarrow rchild \text{ 指向右孩子} \\ \text{False} & p \rightarrow rchild \text{ 指向 (中序) 后继} \end{cases}$



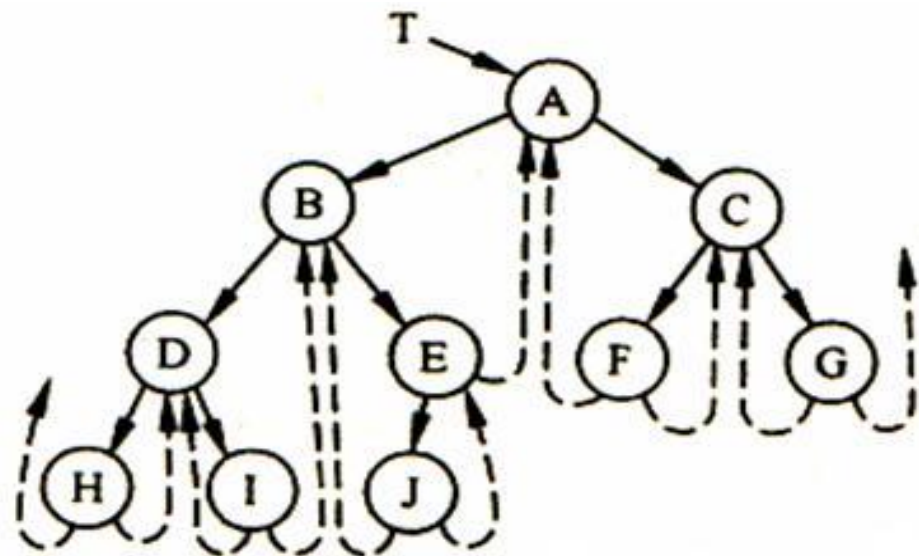
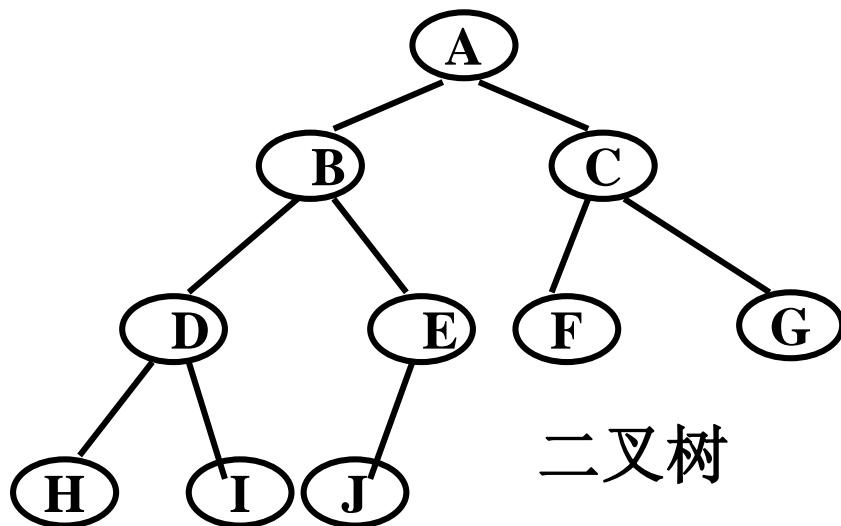
中序线索二叉树





## 3.2 二叉树 (Cont.)

### 二叉树的线索链表存储结构----线索二叉树



中序线索二叉树

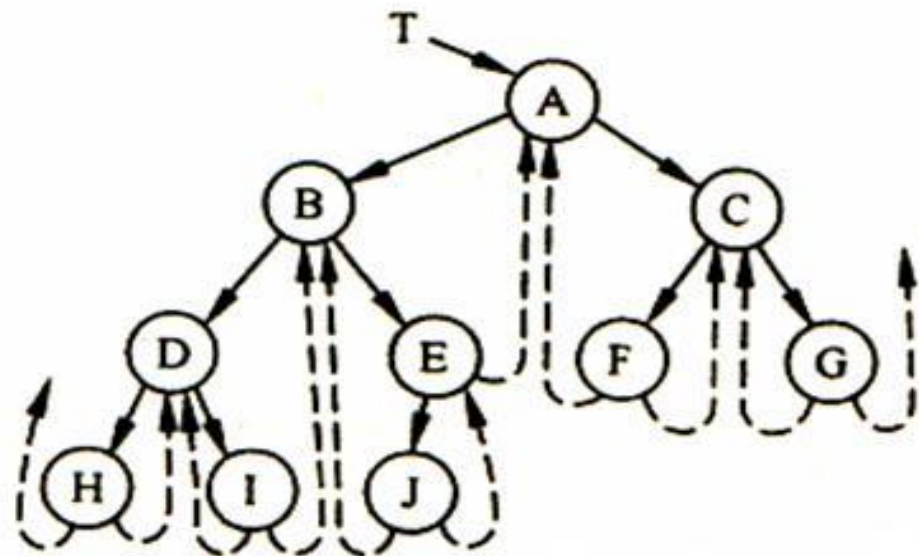
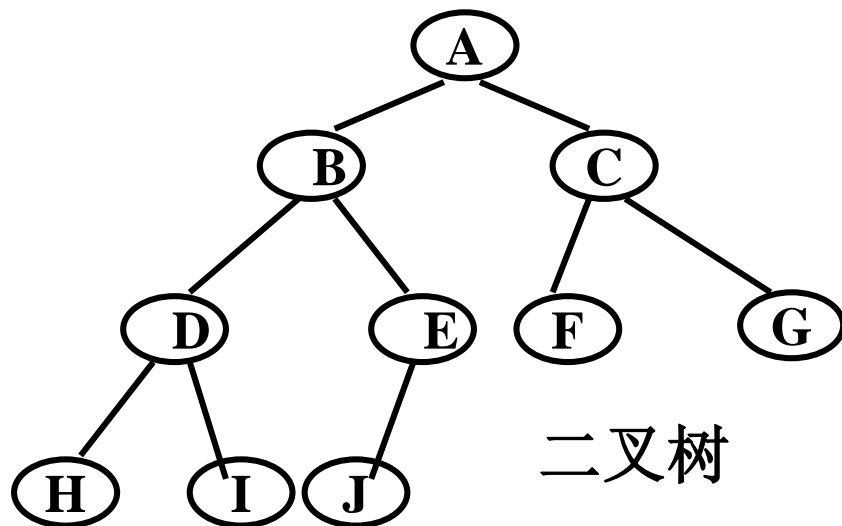
- **线索**: 将结点的空指针域指向其前驱/后继的指针被称为**线索**;
- **线索化**: 结点的空链域存放其前驱/后继的过程称为**线索化**;
- **线索二叉树**: 线索化的二叉树称为**线索二叉树**。





## 3.2 二叉树 (Cont.)

### 二叉树的线索链表存储结构----线索二叉树



### 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

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

结点结构

lchild	ltag	data	rchild	rtag
--------	------	------	--------	------

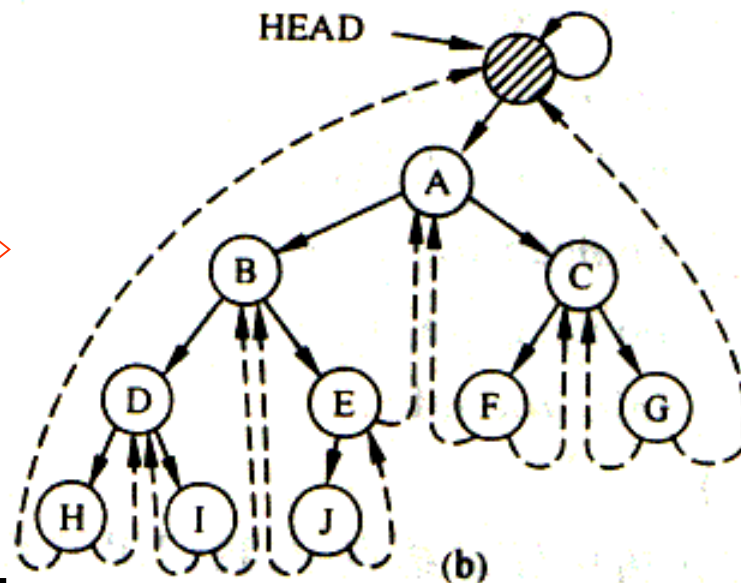
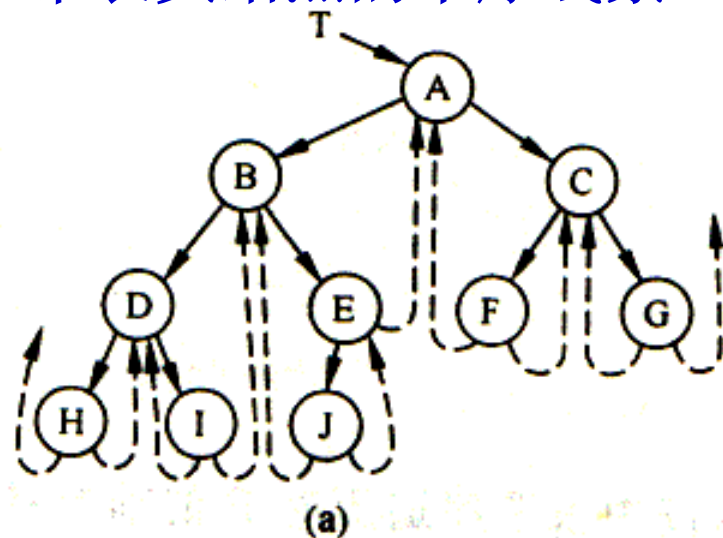






## 3.2 二叉树 (Cont.)

### 带表头结点的中序线索二叉树



lchild	ltag	data	rchild	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

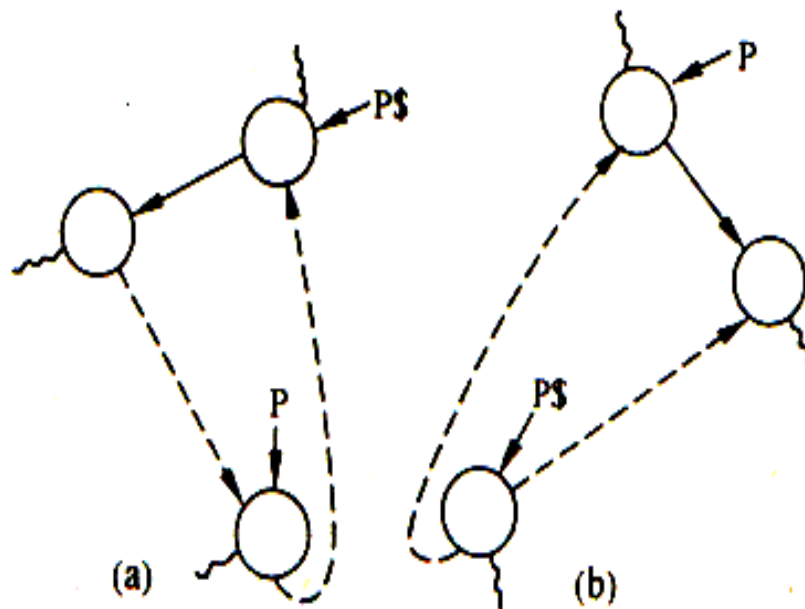
➤ **算法1:** 在中序线索二叉树中求一个结点 $p$ 的中序后继 $p\$$

➤ **分析:**

- (1) 当 $p \rightarrow rtag == \text{False}$ 时,  $p \rightarrow rchild$  即为所求(线索)。
- (2) 当 $p \rightarrow rtag == \text{True}$ 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 ThInOrder(ThTree HEAD)
```

```
{ ThTree tmp ;
```

```
  tmp = HEAD ;
```

```
  do {
```

```
    tmp = InNext ( tmp ) ;
```

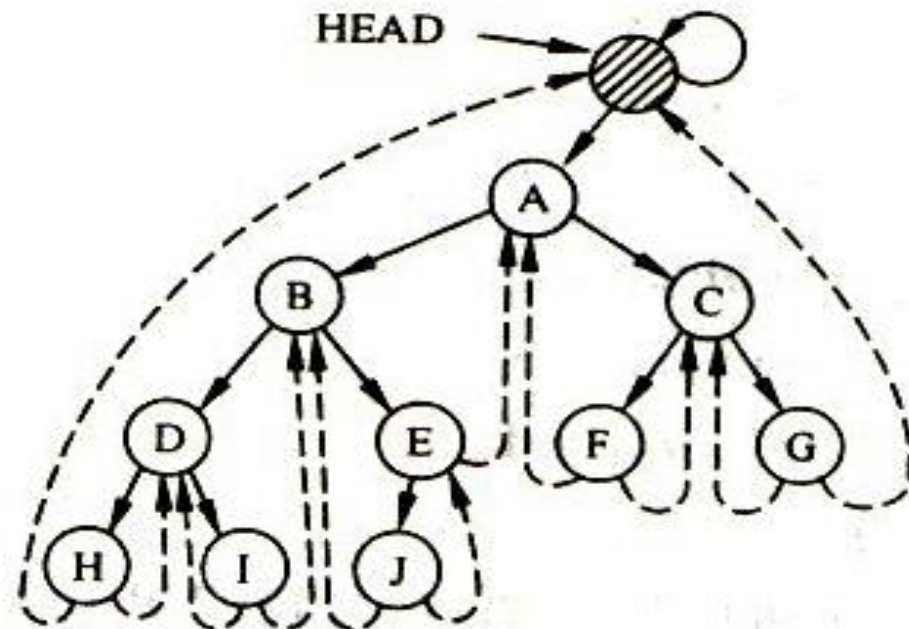
```
    if ( tmp != HEAD )
```

```
        visit ( tmp -> data ) ;
```

```
  } while ( tmp != HEAD ) ;
```

```
}
```

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



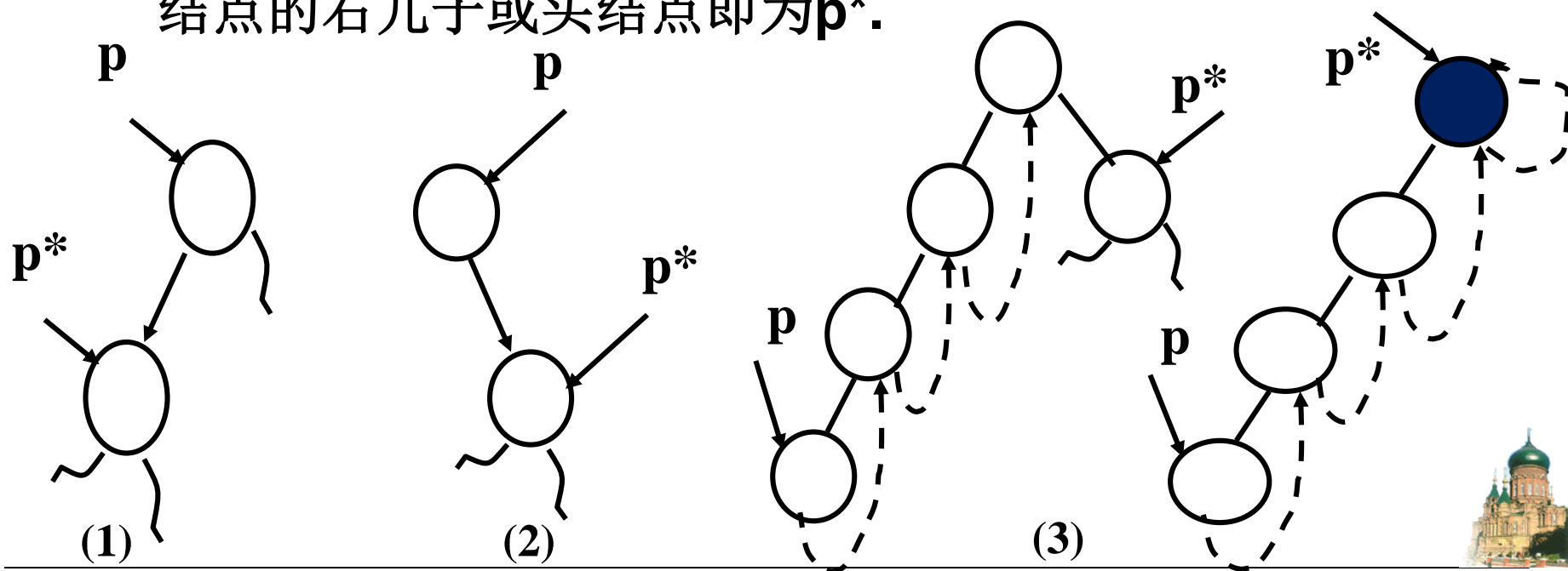


## 3.2 二叉树 (Cont.)

➡ **算法3:** 求中序线索二叉树中结点 $p$ 的先序顺序后继结点 $p^*$

➡ **分析:**

- (1)  $p$  的左子树不空时,  $p$  的左儿子 $p \rightarrow lchild$  即为  $p^*$ ;
- (2)  $p$  的左子树空但右子树不空时,  $p$  的 $p \rightarrow rchild$  为 $p^*$ ;
- (3)  $p$  的左右子树均空时, 右线索序列中第一个有右孩子结点的右儿子或头结点即为 $p^*$ .





## 3.2 二叉树 (Cont.)

➡ **算法3:** 求中序线索二叉树中结点 $p$ 的先序顺序后继结点 $p^*$

➡ **算法实现:**

**THTREE PreNext( ThTree p)**

```
{ ThTree Q ;  
    if (p->ltag == True )  
        Q=p->lchild ;  
    else{ Q = p;  
        while(Q->rtag == False)  
            Q = Q->rchild ;  
        Q = Q->rchild ;  
    } return ( Q ) ;  
}
```





## 3.2 二叉树 (Cont.)

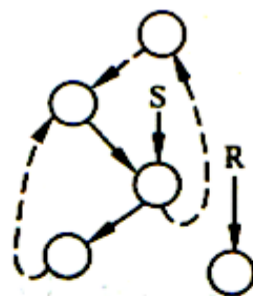
### 线索二叉树的若干算法

➡ **算法4:** 中序线索二叉树的插入算法

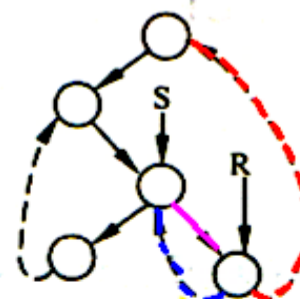
➡ **分析:** 如将结点 **R** 插入作为结点 **S** 的右孩子结点。

(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

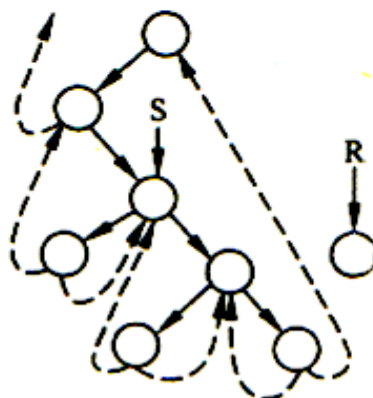


插入前

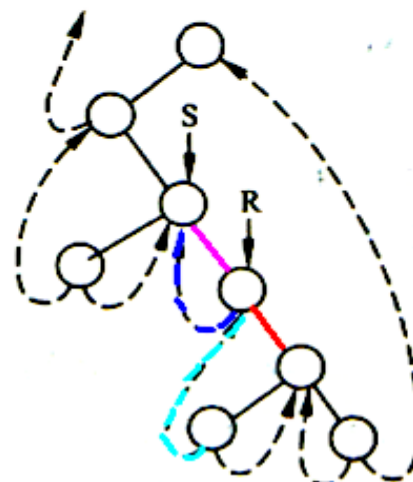


插入后

(a)



插入前



插入后

(b)





## 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 ;
```

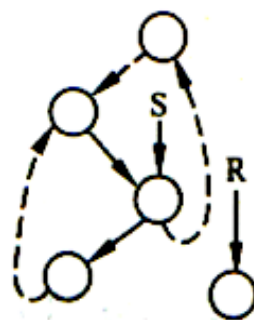
```
  S->rtag = True ;
```

```
  if (R->rtag==True) {
```

```
    w = InNext( R ) ;
```

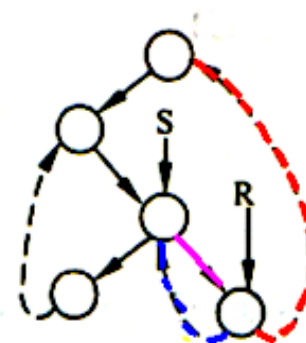
```
    w->lchild = R ; }
```

```
}
```

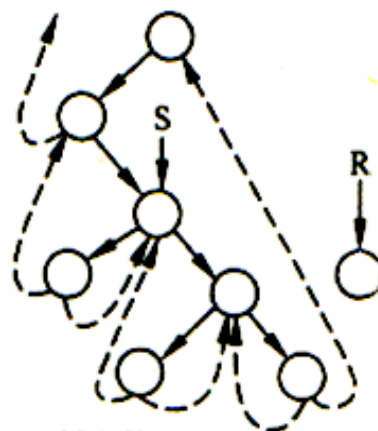


插入前

(a)

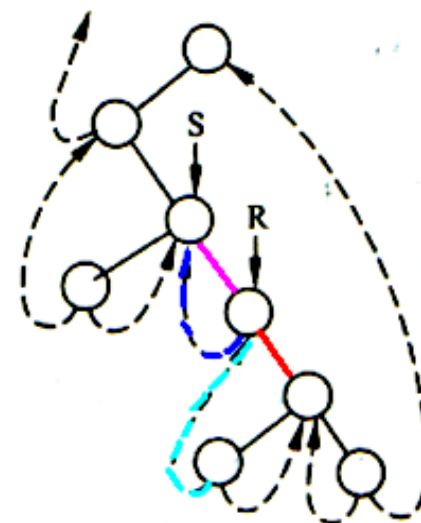


插入后



插入前

(b)



插入后



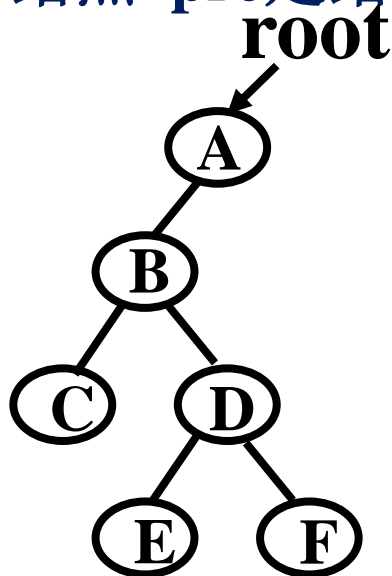


## 3.2 二叉树 (Cont.)

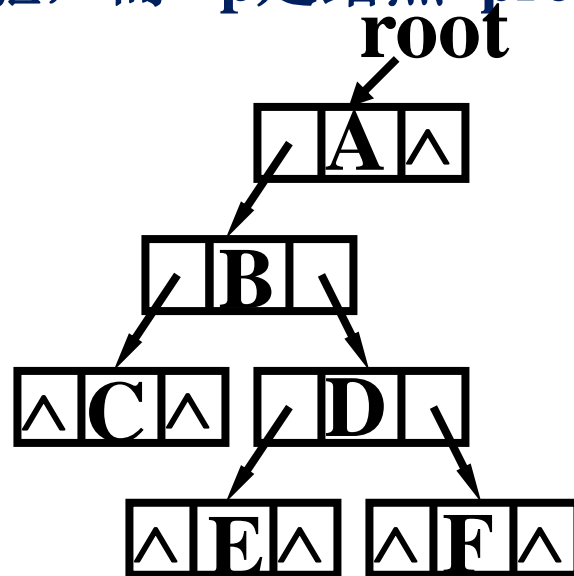
➤ **算法5:** 二叉树的 (中序) 线索化算法-----递归算法

➤ **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；

3.2 若**root**没有右孩子，则将**root**右标志置为**False**；

3.3 若结点**pre**右标志为**False**，则为**pre**加上后继线索；

3.4 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(Btree *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 ); //左子树线索化
    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 ==False) // *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 ==False) // *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 ); //右子树线索化
}
}

```





## 3.2 二叉树 (Cont.)

### ➡ 二叉树的复制

- 两株二叉树具有**相同结构**指：**“形状”相同**
  - (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**。
- **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 \*/**





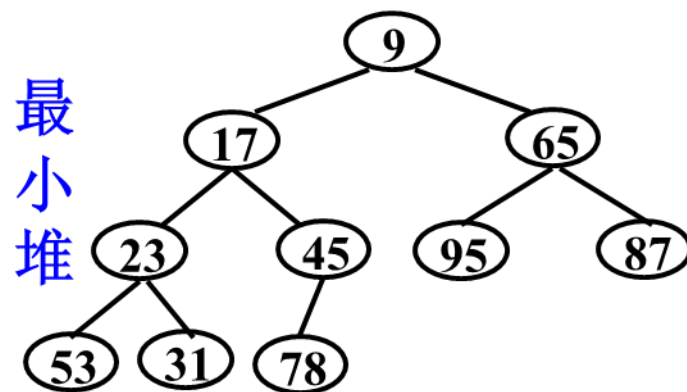
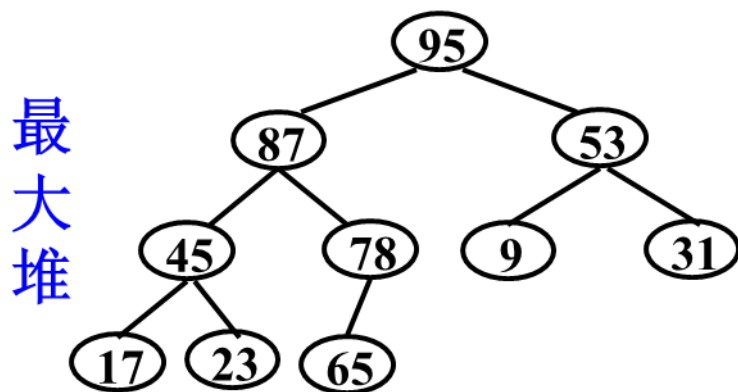
## 3.3 堆

一种特殊形式的完全二叉树——堆（heap）。它有两种基本形式：**最大堆**和**最小堆**。

如果一棵完全二叉树的任意一个非终结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**。

类似地，如果一棵完全二叉树的任意一个非终结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**。

**特点：**根结点的元素是最大（小）的。





### 基本操作:

- (1) `MaxHeap( heap )`: 创建一个空堆, 最多可容纳`MaxSize`个元素。
- (2) `HeapFull( heap )`: 判断堆是否为满。若堆中元素个数达到最大容量`MaxSize`, 则返回`TRUE`, 否则返回`FALSE`。
- (3) `Insert( heap, item )`: 插入一个元素。若堆不满, 则将`item`插入`heap`, 否则不能插入。
- (4) `HeapEmpty( heap)`: 判断堆是否为空。若堆中元素个数小于0, 则返回`TRUE`, 否则返回`FALSE`。
- (5) `DeleteMax( heap)`: 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。





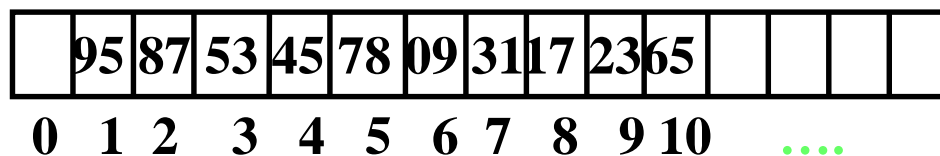
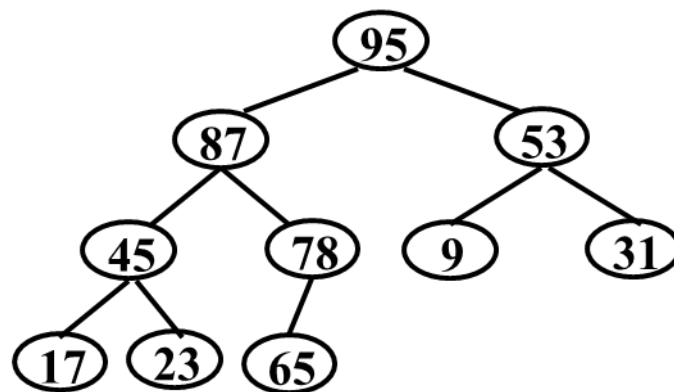
## 最大堆的实现

### 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;
```





```
void MaxHeap ( Heap &heap)//创建一个空堆  
{  
    heap.n=0;  
}
```

```
bool HeapEmpty (Heap heap)//判断堆是否为空  
{  
    return (!heap.n);  
}
```

```
bool HeapFull ( Heap heap)//判断堆是否为满  
{  
    return ( heap.n == MaxSize-1);  
}
```





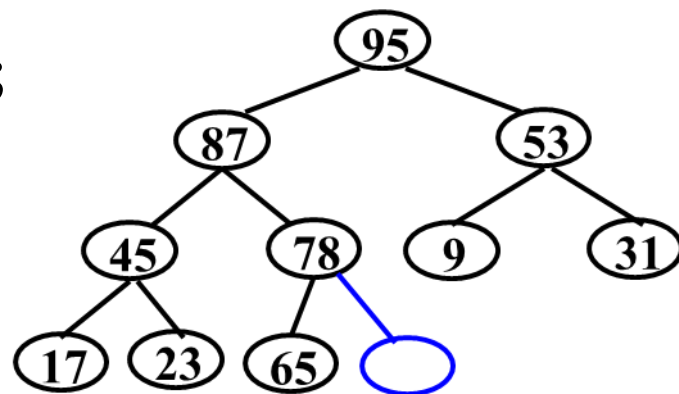
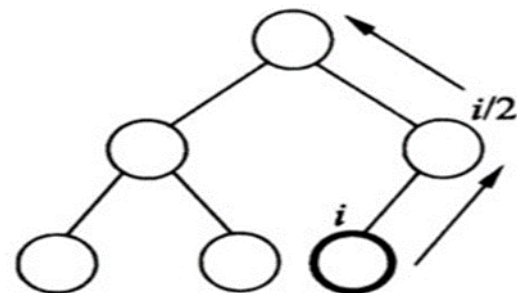


**void Insert ( Heap &heap, Elementtype x) //大根堆插入一个元素**

```

{
    int i;
    if ( ! HeapFull ( heap ) )
    {
        i=heap.n+1;
        while ( (i!=1)&&(x >heap.ele [i/2] ) )
        {
            heap.ele [i]=heap.ele [i/2];
            i=i/2;
        }
        heap.ele [i]=x;
        heap.n++;
    }
} //时间复杂度O (logn)

```



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





```

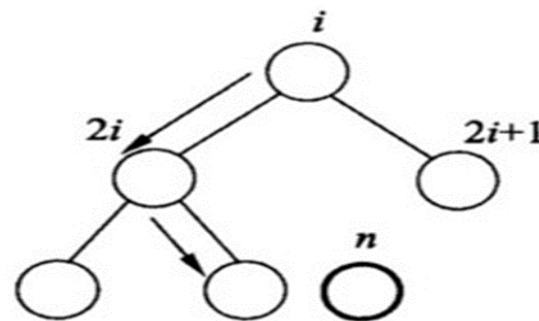
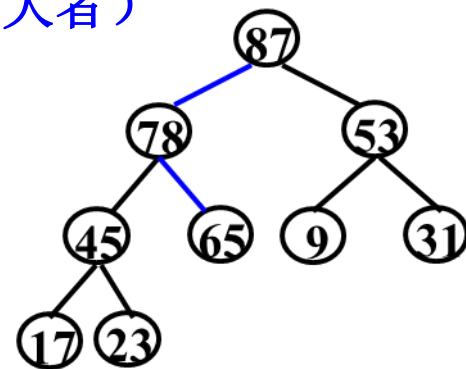
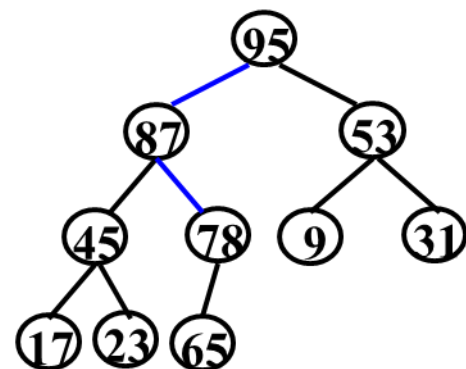
void DeleteMax (Heap & heap )//大根堆删除
{   int parent=1, child=2; Elementtype ele, tmp;
    if (! HeapEmpty(heap))
    {   ele=heap.ele [1];
        tmp=heap.ele [heap.n--];
        while (child<= heap.n){
            if(child<heap.n)&&(heap.ele [child]<heap.ele [child+1]))
                child++; //找最大子结点（左右儿子的大者）
            if (tmp>=heap.ele [child]) break;
            heap.ele [parent]=heap.ele [child];
            parent=child;
            child*=2;
        }//while
        heap.ele [parent]=tmp;
        return ele;
    }//if
}

```

**//时间复杂度  $O(\log n)$**

	95	87	53	45	78	09	31	17	23	65				
--	----	----	----	----	----	----	----	----	----	----	--	--	--	--

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





经常使用堆来实现优先级队列（priority queue）。与第二章所讨论的队列不同的是，优先级队列只对最高（或最低）优先级的元素进行删除。但是在任何时候，都可以把任意优先级的元素插入到优先级队列。

操作系统中的进程管理是优先级队列的一个应用实例，系统中使用一个优先队列来管理进程。

每个进程有进程任务号和优先级两部分组成。当有多个进程排队时，优先级高的先操作。





**作业：**设计一个程序模仿操作系统的进程管理问题，进程服务按优先级高的先服务，同优先级的先到先服务的管理原则。设文件task.dat中存放了仿真进程服务请求，其中第一列是进程任务号，第二列是进程的优先级。

1 30

2 20

3 40

4 20

5 0

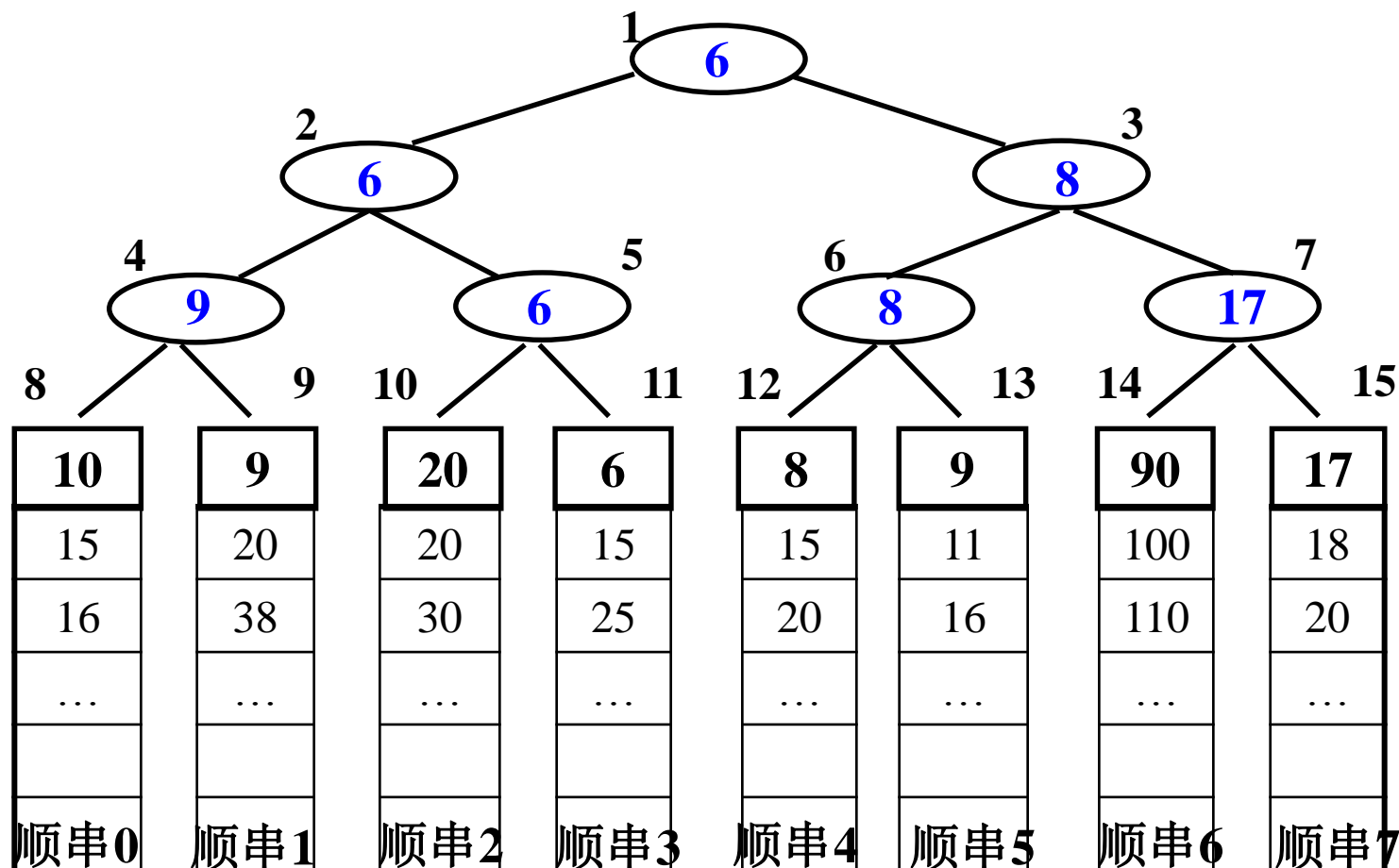
算法：1) 建堆  
2) 循环输出堆顶。





## 3.4 选择树 (Selection Tree)

- 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？



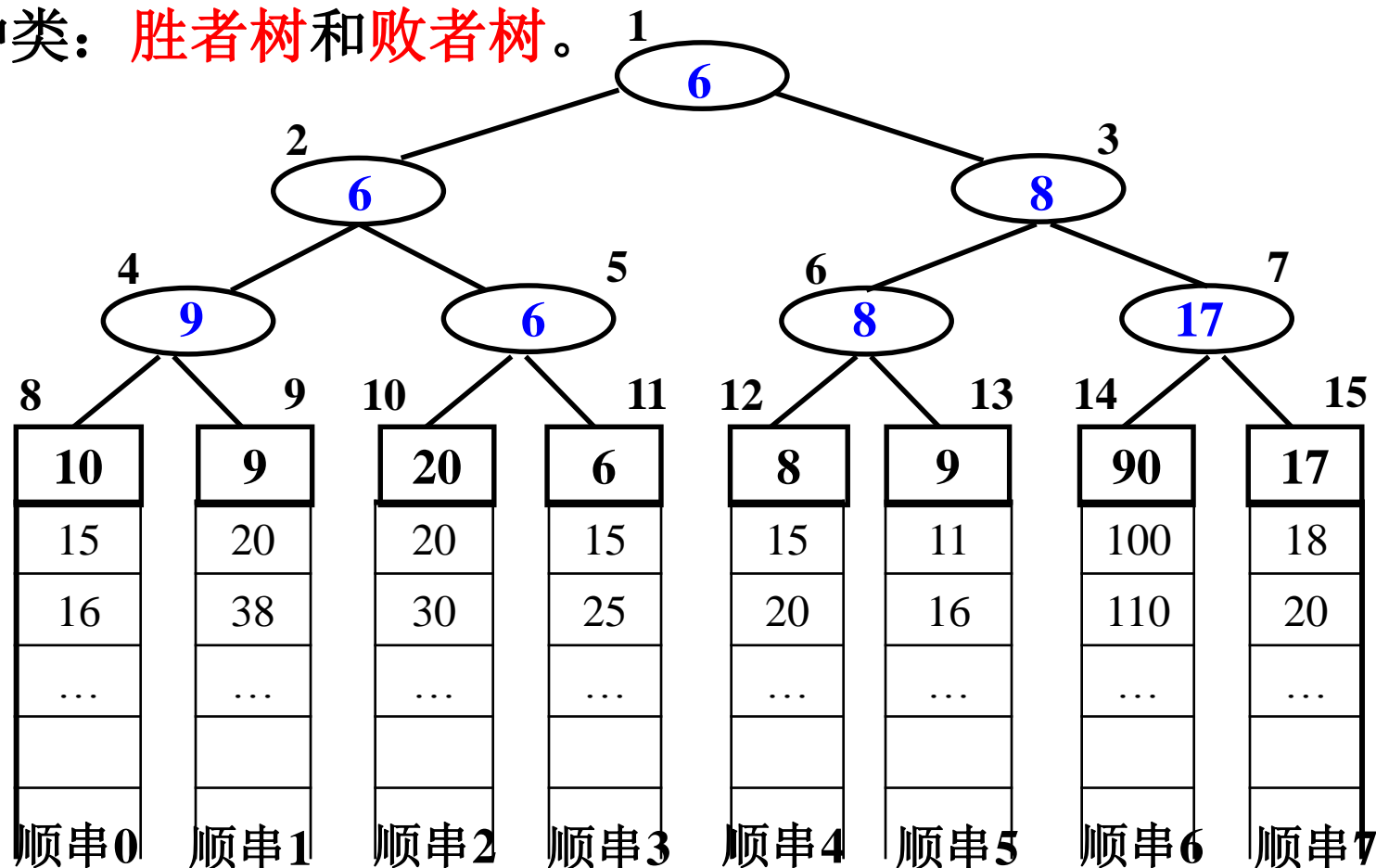


## 3.4 选择树 (Selection Tree)

**选择树** (也称**Tournament Tree**)

➤ **选择树**就是能够记载上一次比较获得的结果的完全二叉树

➤ 种类: **胜者树**和**败者树**。

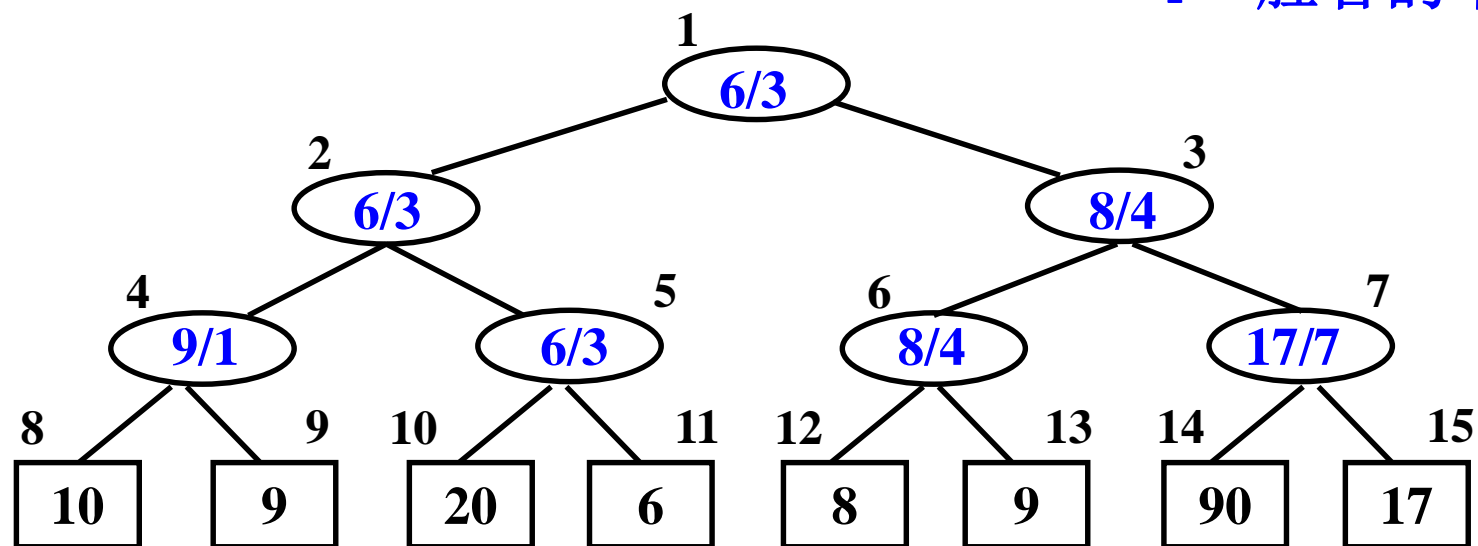




## 3.4 选择树 (Selection Tree)

### 胜者树 (Winner Tree)

$X/Y$  X—胜者关键字  
Y—胜者的下标



顺串0 顺串1 顺串2 顺串3 顺串4 顺串5 顺串6 顺串7

- 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

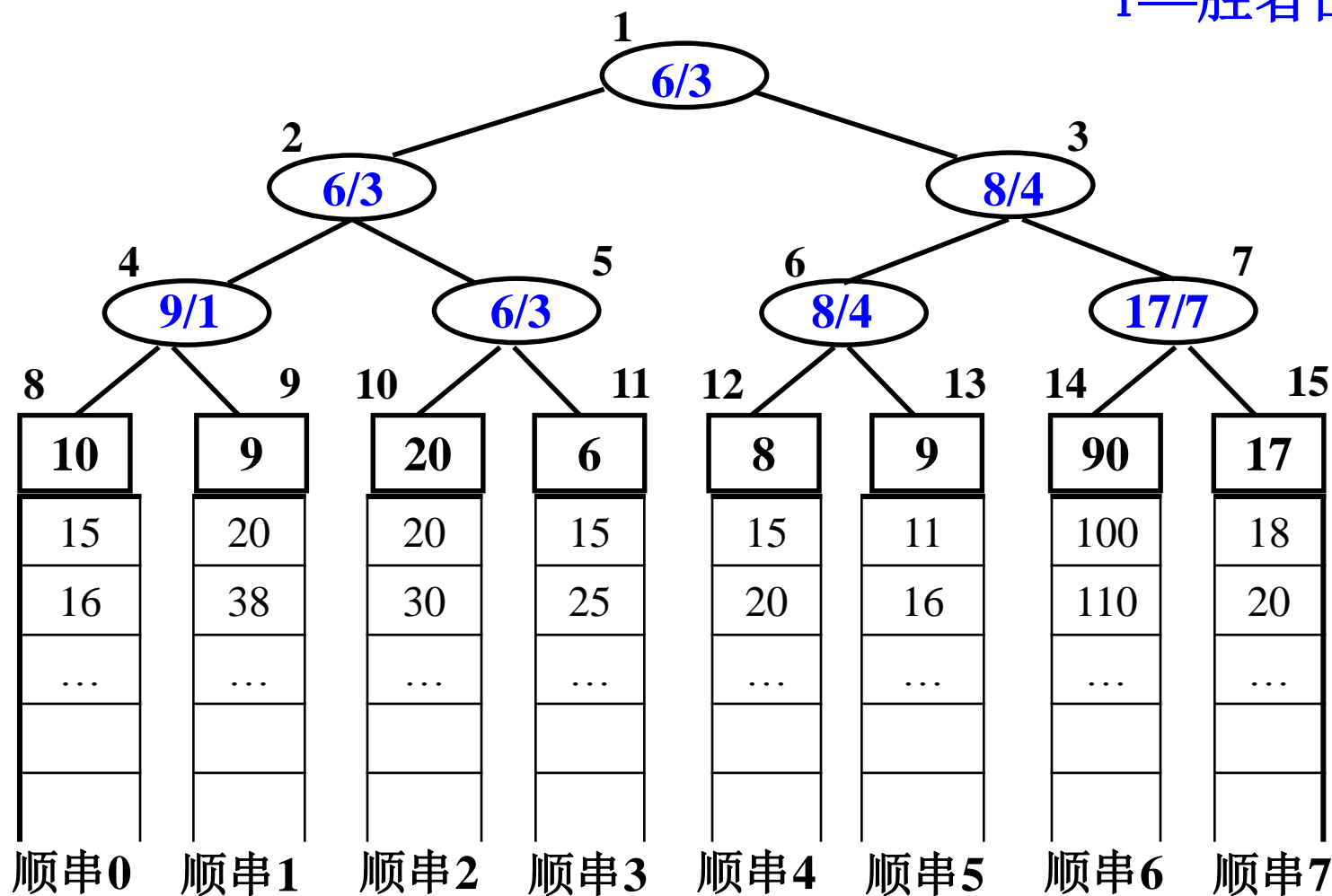




## 3.4 选择树 (Selection Tree)

➡ 胜者树的构建

$\text{X/Y}$  X—胜者关键字  
Y—胜者的下标

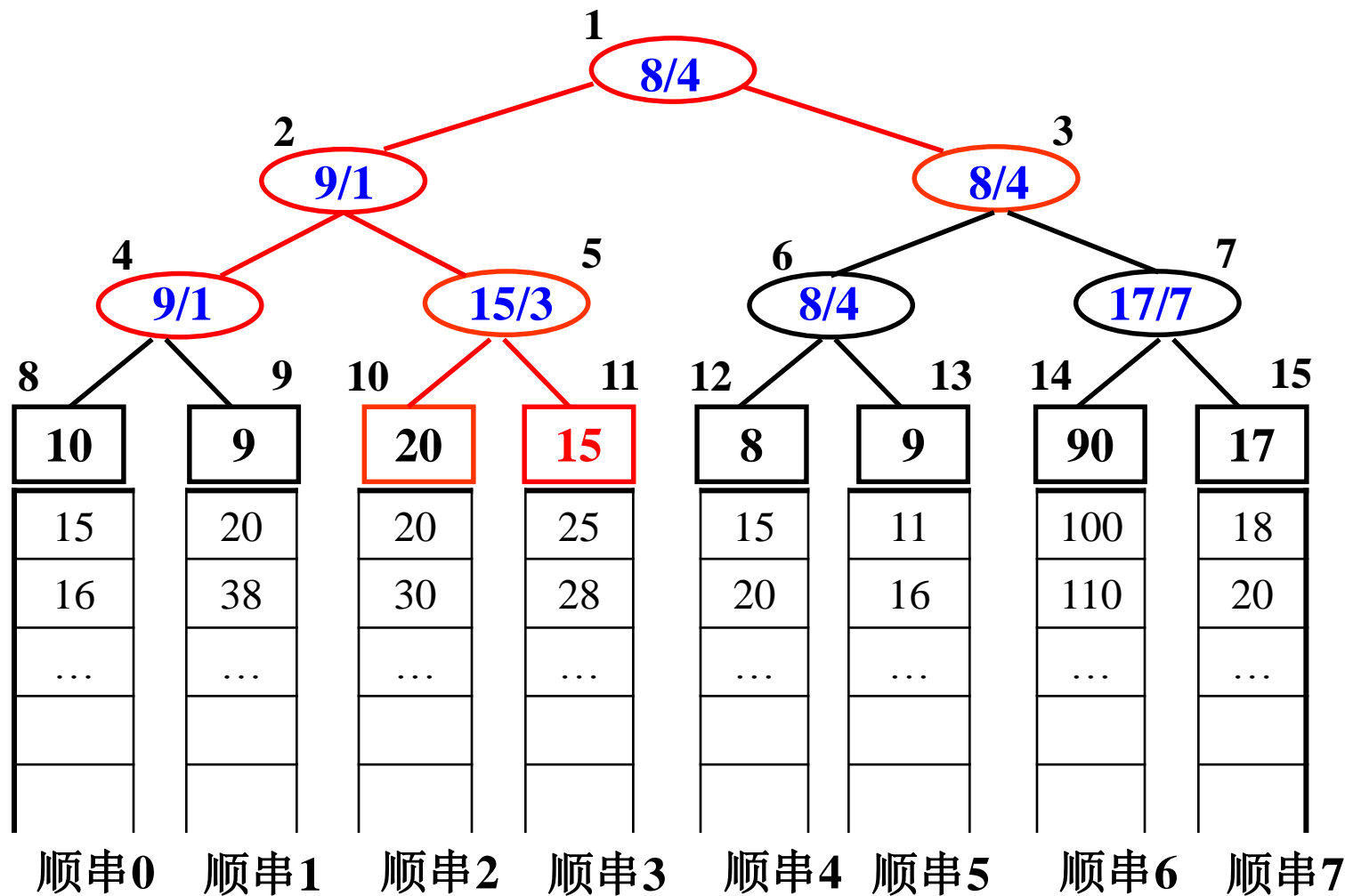






## 3.4 选择树 (Selection Tree)

➡ 胜者树的重构：新进入的结点与兄弟结点比较，直到树根

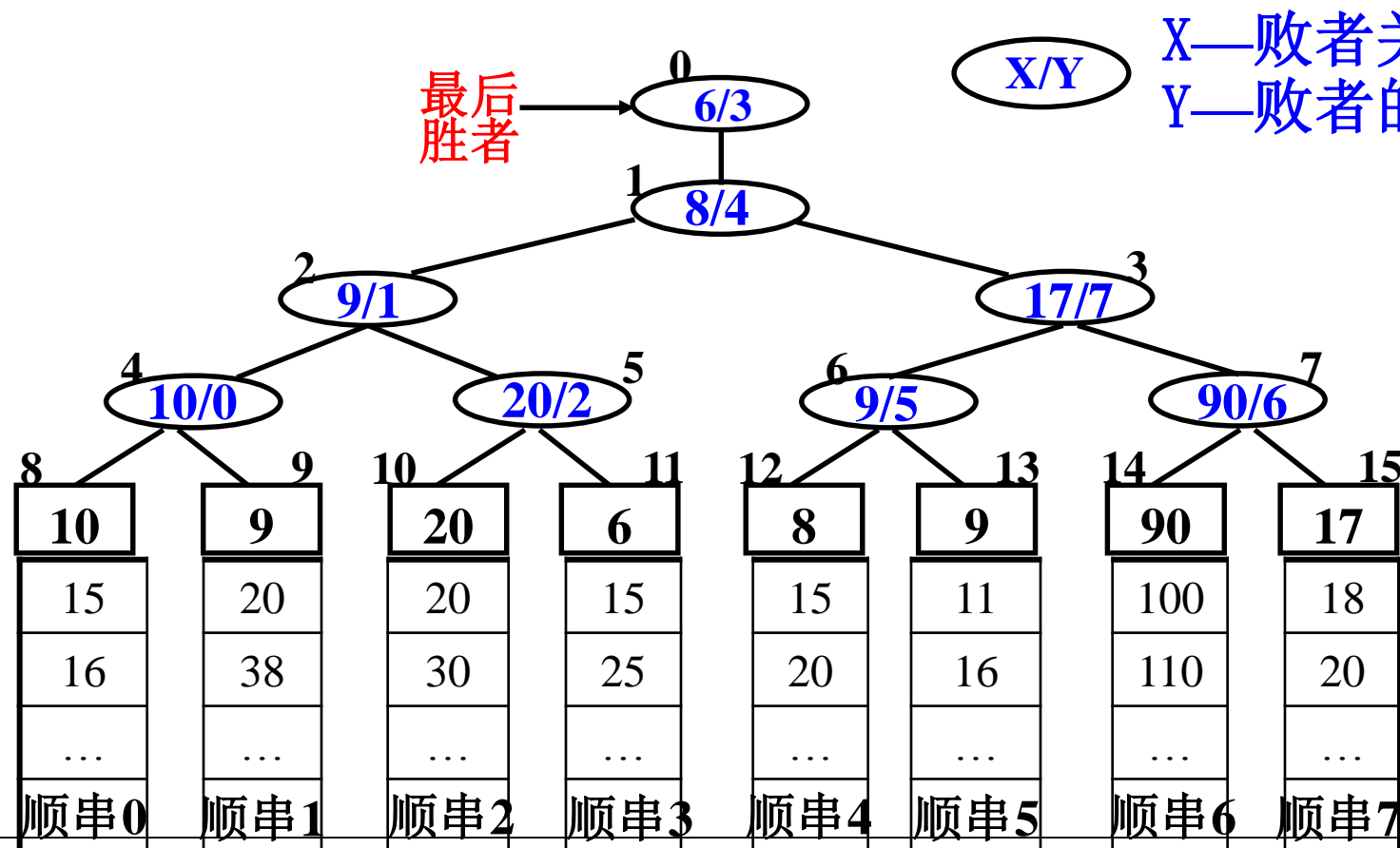




## 3.4 选择树 (Selection Tree)

### 败者树的构建(Loser Tree)

- 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

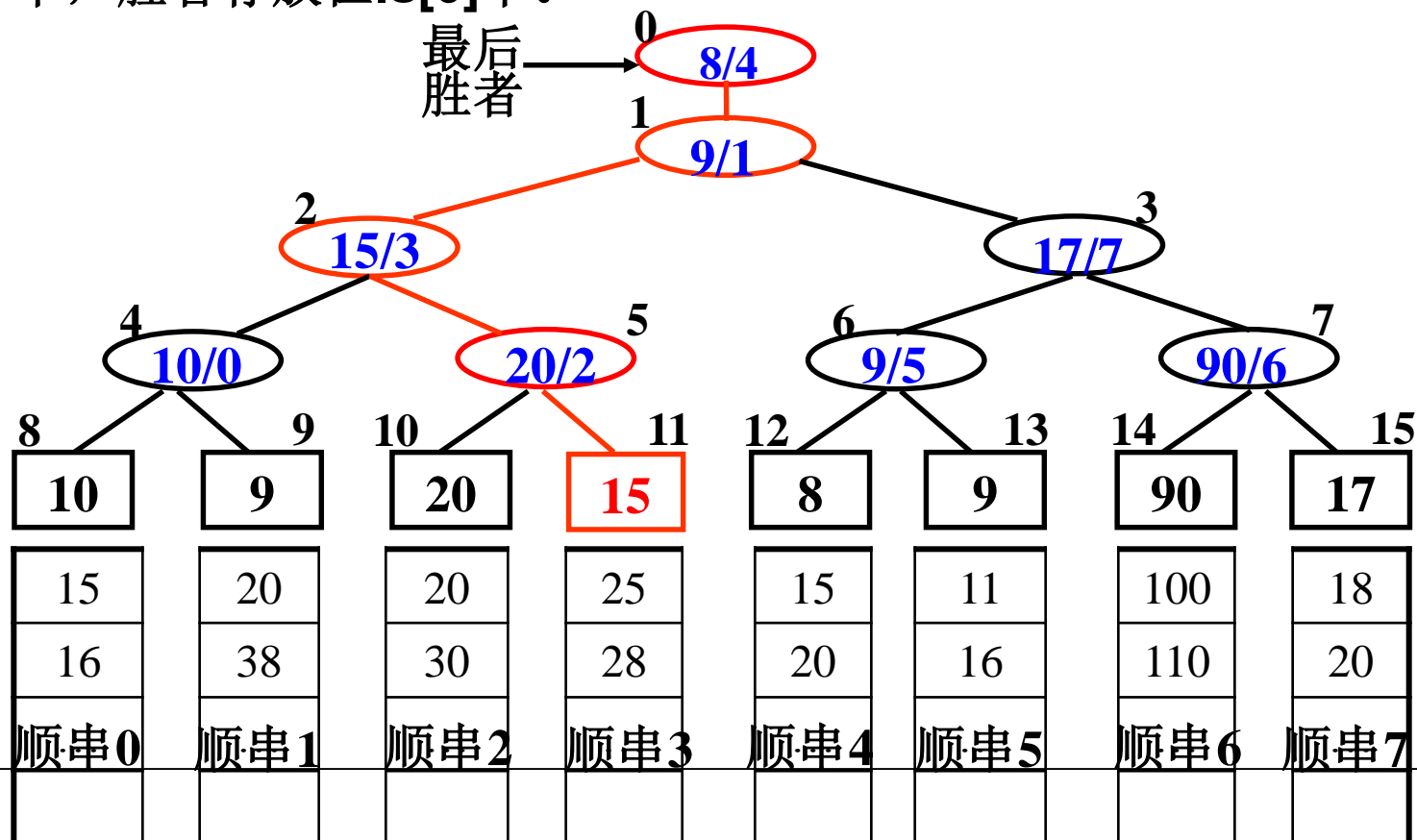




## 3.4 选择树 (Selection Tree)

### 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。

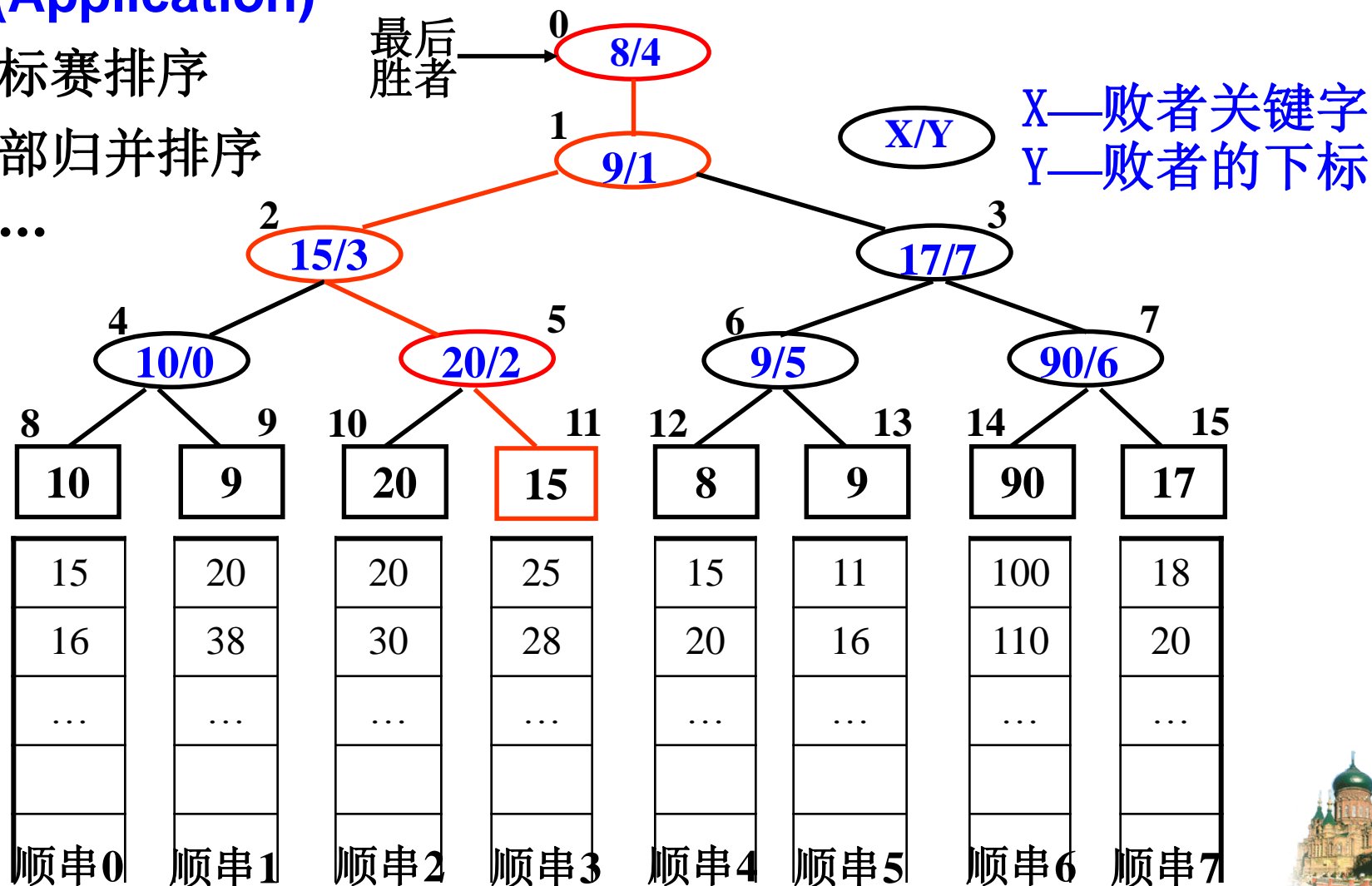




## 3.4 选择树 (Selection Tree)

### 应用(Application)

- 锦标赛排序
- 外部归并排序
- .....





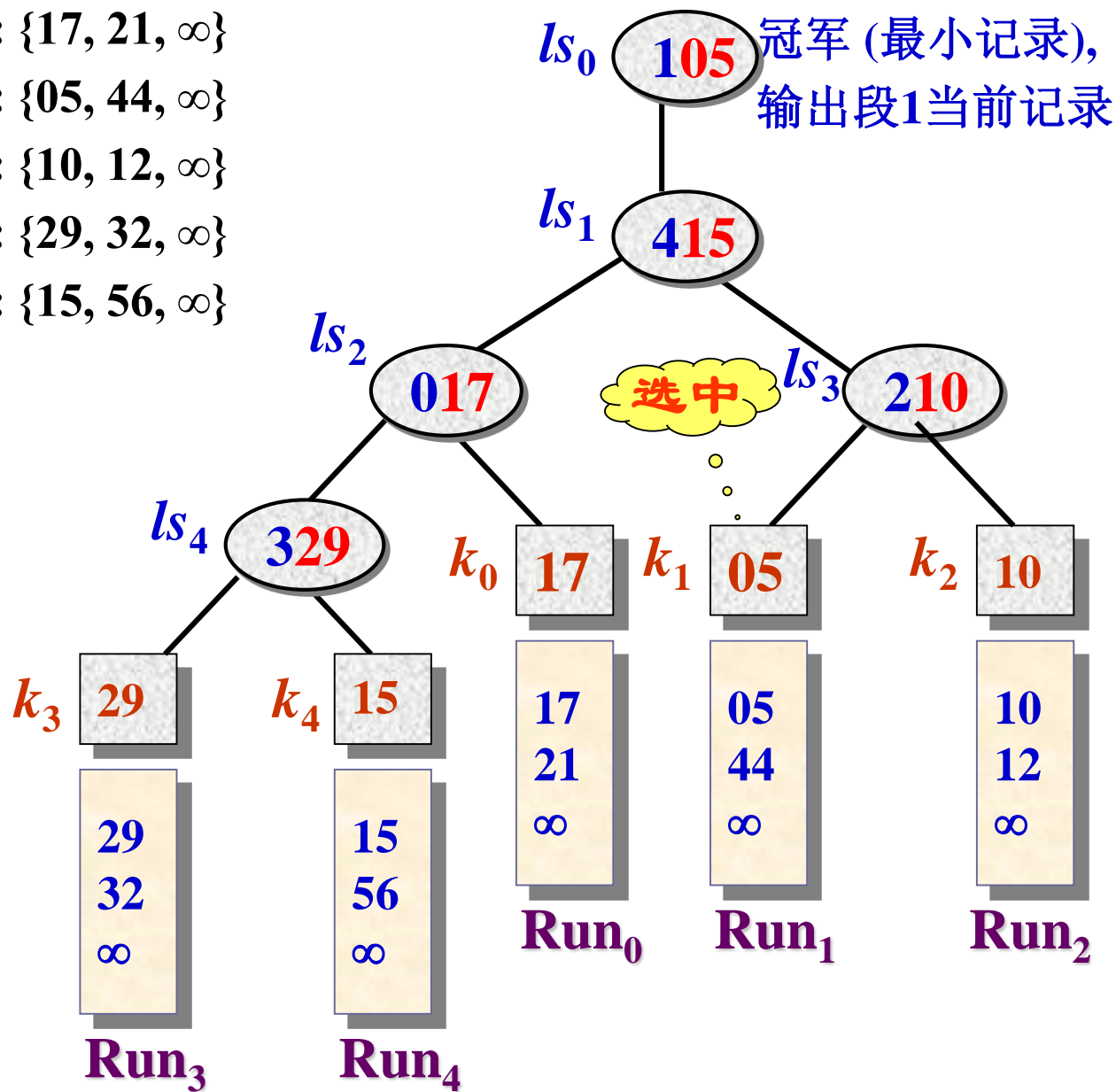
Run0: {17, 21,  $\infty$ }

Run1: {05, 44,  $\infty$ }

Run2: {10, 12,  $\infty$ }

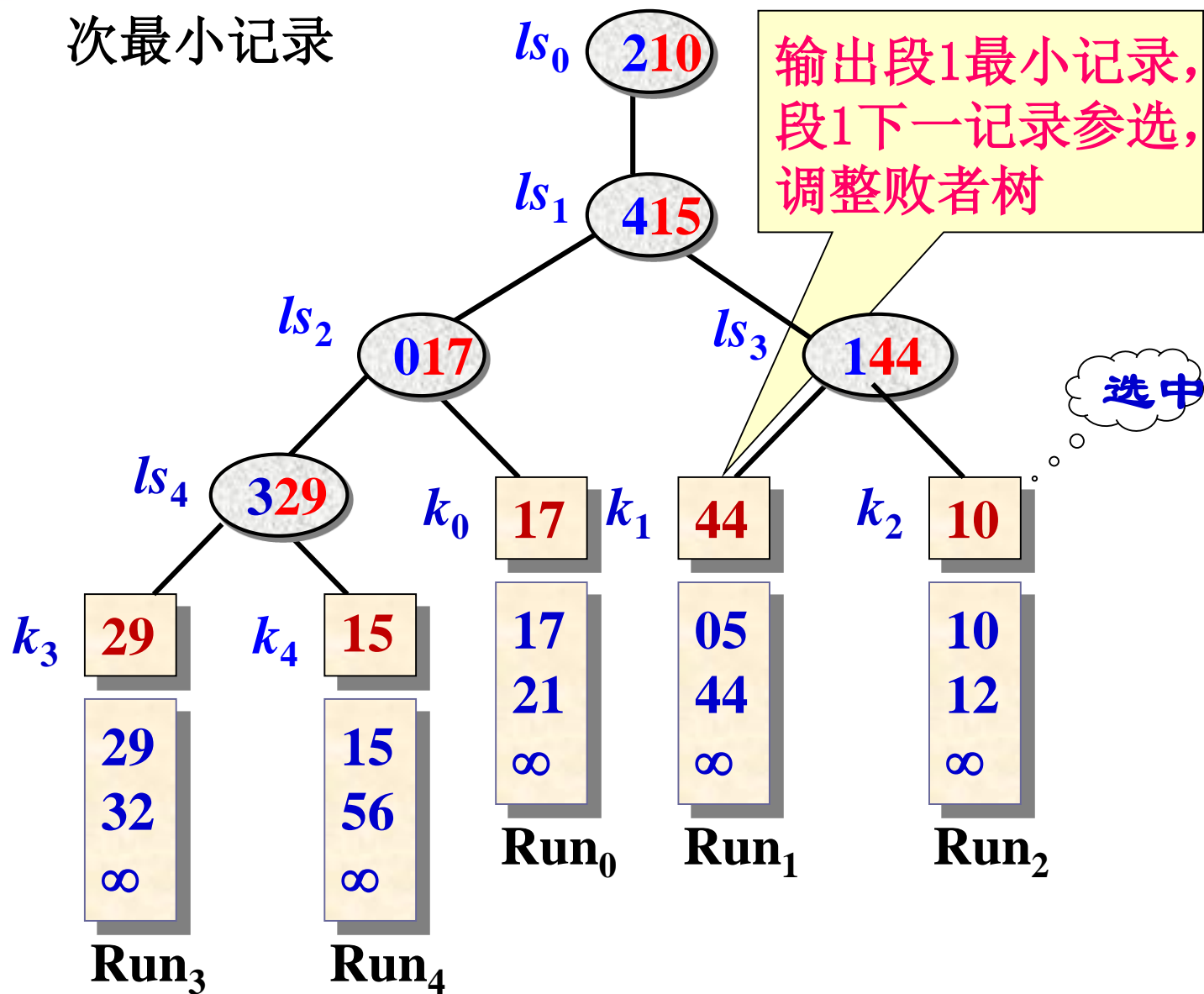
Run3: {29, 32,  $\infty$ }

Run4: {15, 56,  $\infty$ }





次最小记录





■败者树的高度为  $\lceil \log_2(k+1) \rceil$ ，在每次调整，找一个具有最小关键字记录时，最多做  $\lceil \log_2 k \rceil$  次关键字比较。

**作业：**已知顺串

$R1[10, 15, 16], R2[9, 20, 38], R3[20, 20, 30],$   
 $R4[6, 15, 25], R5[8, 15, 20], R6[9, 11, 16],$   
 $R7[90, 100, 110], R8[17, 18, 20]$  建立败者树。





## 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

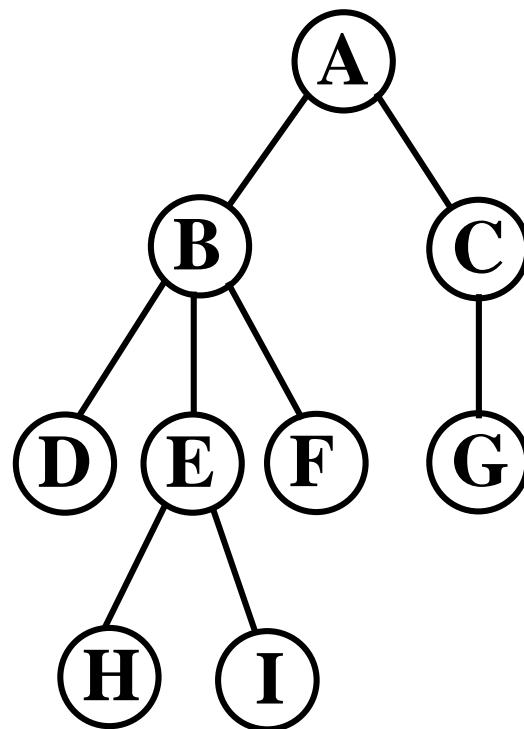
- 访问根结点；
- 按照从左到右的顺序先序遍历根结点的每一棵子树

先序遍历序列：A B D E H I F C G

### 后序遍历

- 按照从左到右的顺序后序遍历根结点的每一棵子树；
- 访问根结点。

后序遍历序列：D H I E F B G C A





## 3.5 树 (Cont.)

### 层序遍历

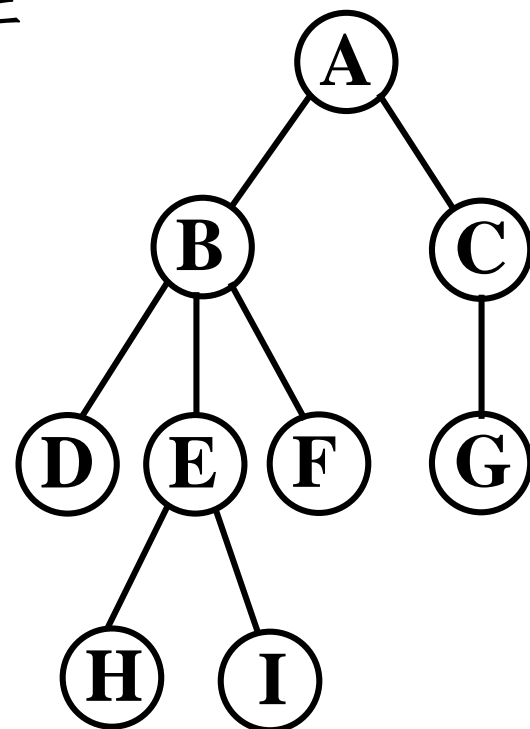
从树的第一层（即根结点）开始，自上而下逐层遍历，在同一层中，按从左到右的顺序对结点逐个访问。

层序遍历序列：A B C D E F G H I

### 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

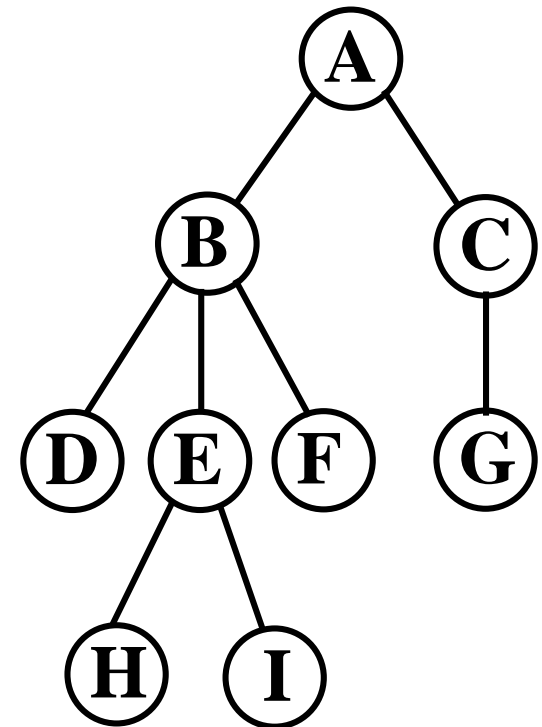




## 3.5 树 (Cont.)

➡ 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( T ) ) ;  
  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

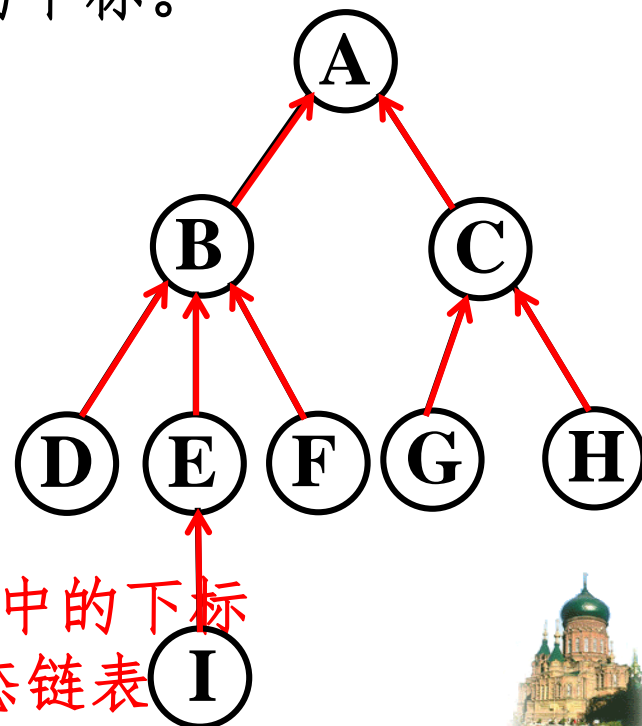
#### 双亲表示法 (单链表示、父链表示)

- 每个结点 (根结点除外) 都只有**唯一**的**双亲**结点
- 因此, 可以把各个结点 (一般按**层序**) 存储一维数组中, 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:

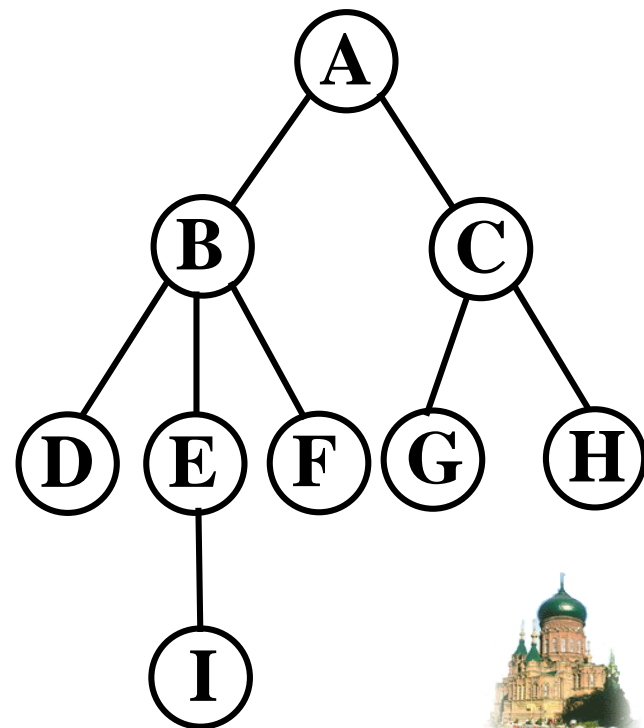
- 每个结点均保存父结点所在的数组单元下标
- 兄弟结点的编号连续。

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)

- 把每个结点的孩子看成是一个线性表，且以单链表存储，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，组织成一个线性表，为了便于进行查找采用顺序存储。
- 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。

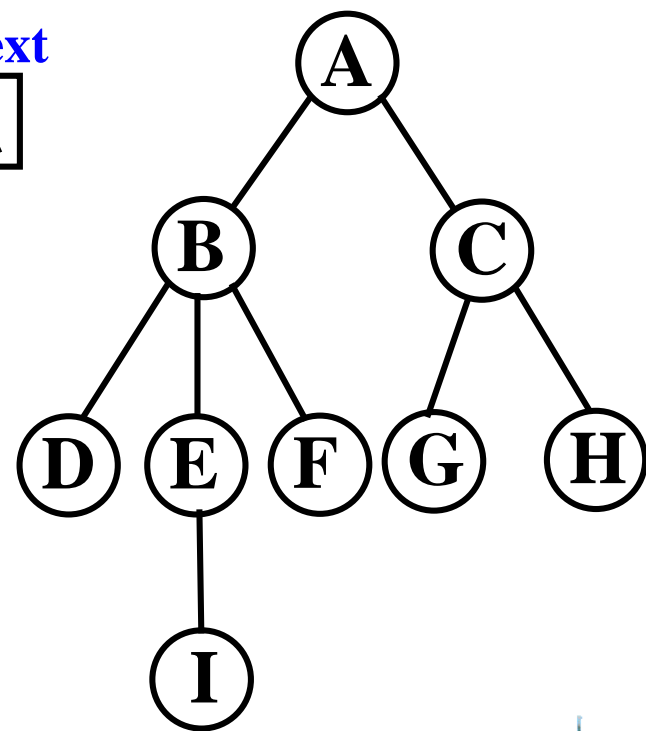
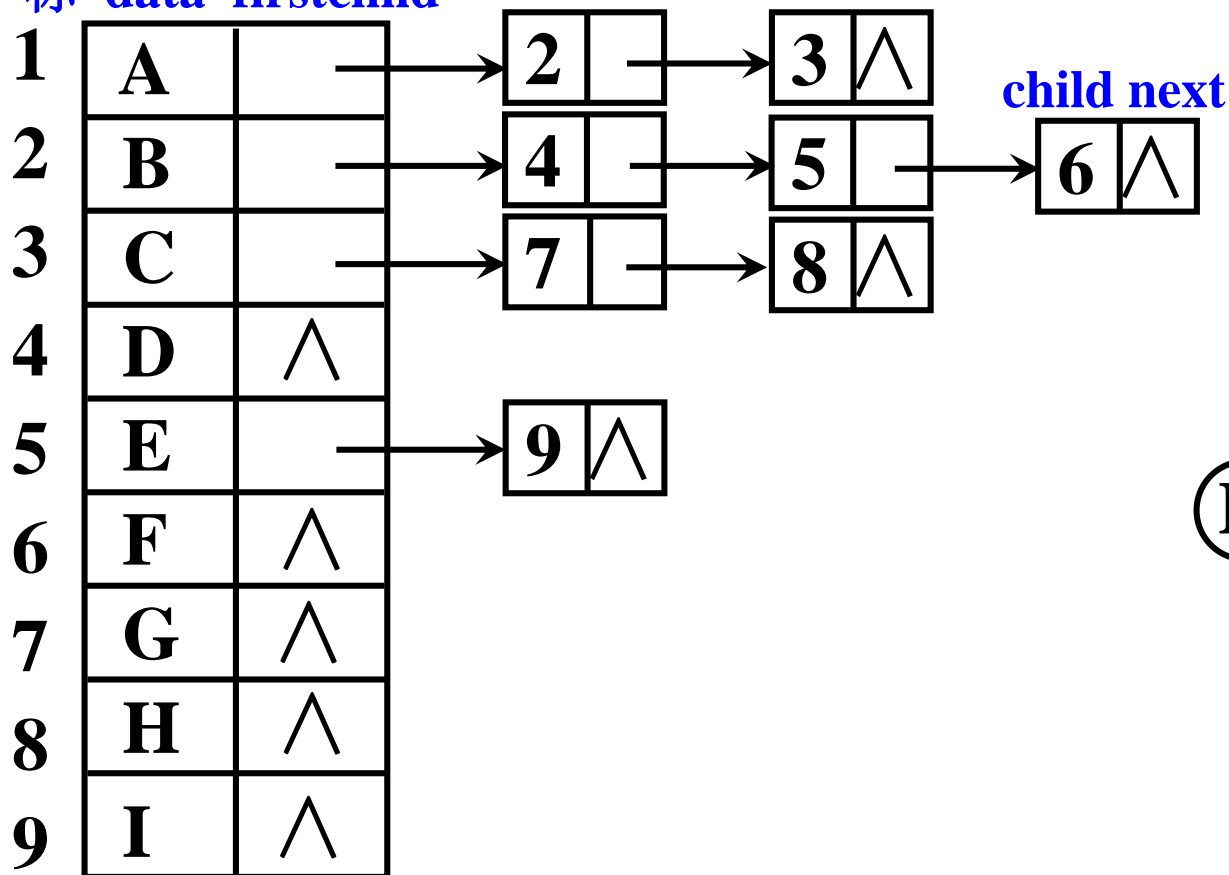




## 3.5 树 (Cont.)

### 孩子链表表示法 (邻接表表示)

下标 data firstchild





## 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



表头结点





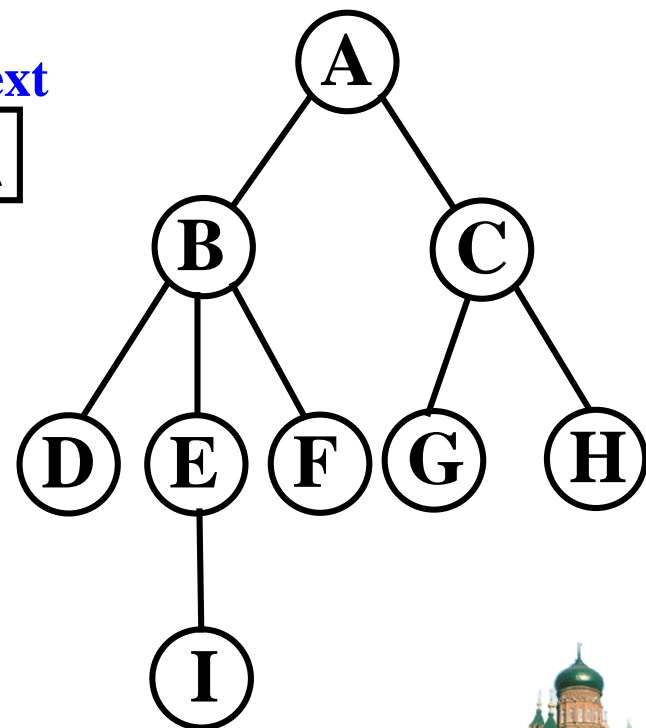
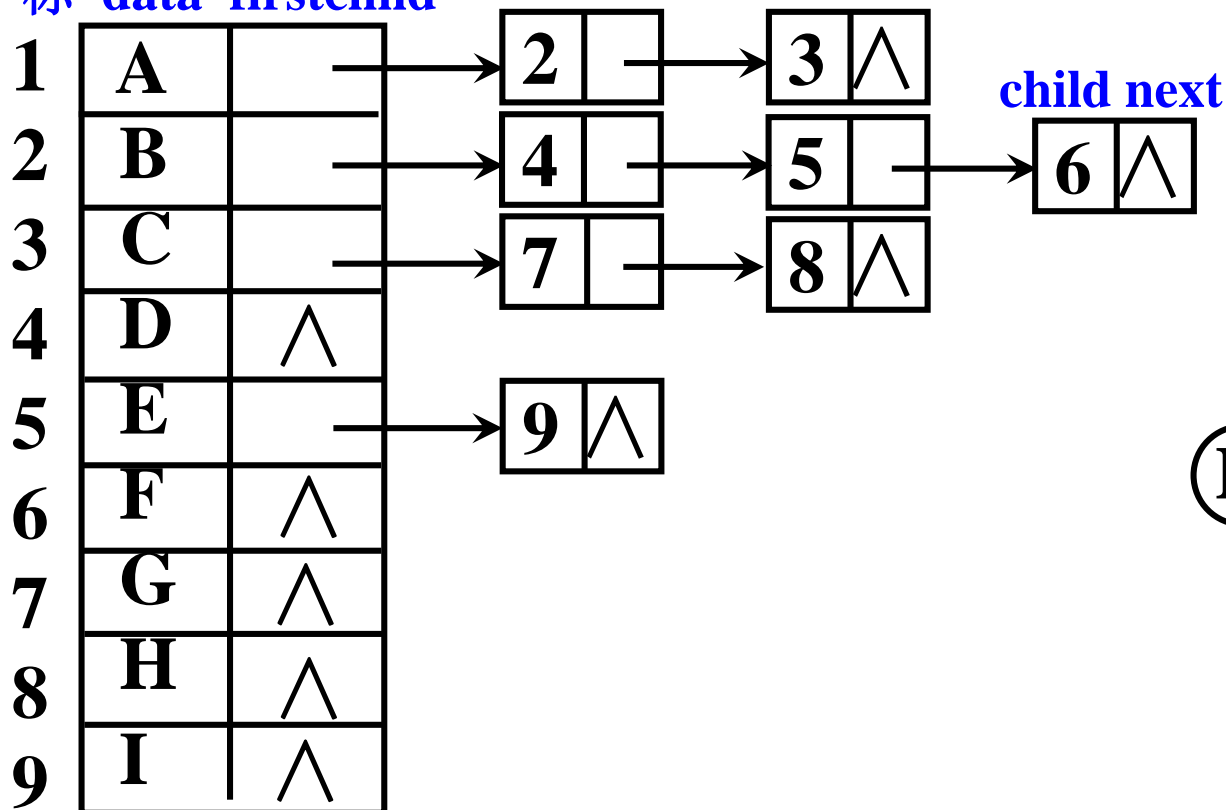


## 3.5 树 (Cont.)

### 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?

下标 data firstchild



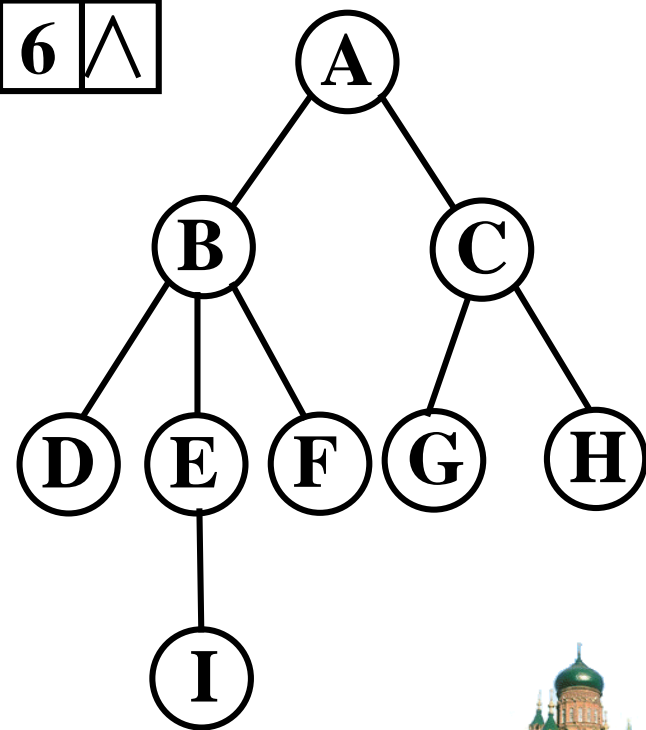


## 3.5 树 (Cont.)

### 双亲孩子表示法

data parent firstchild

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

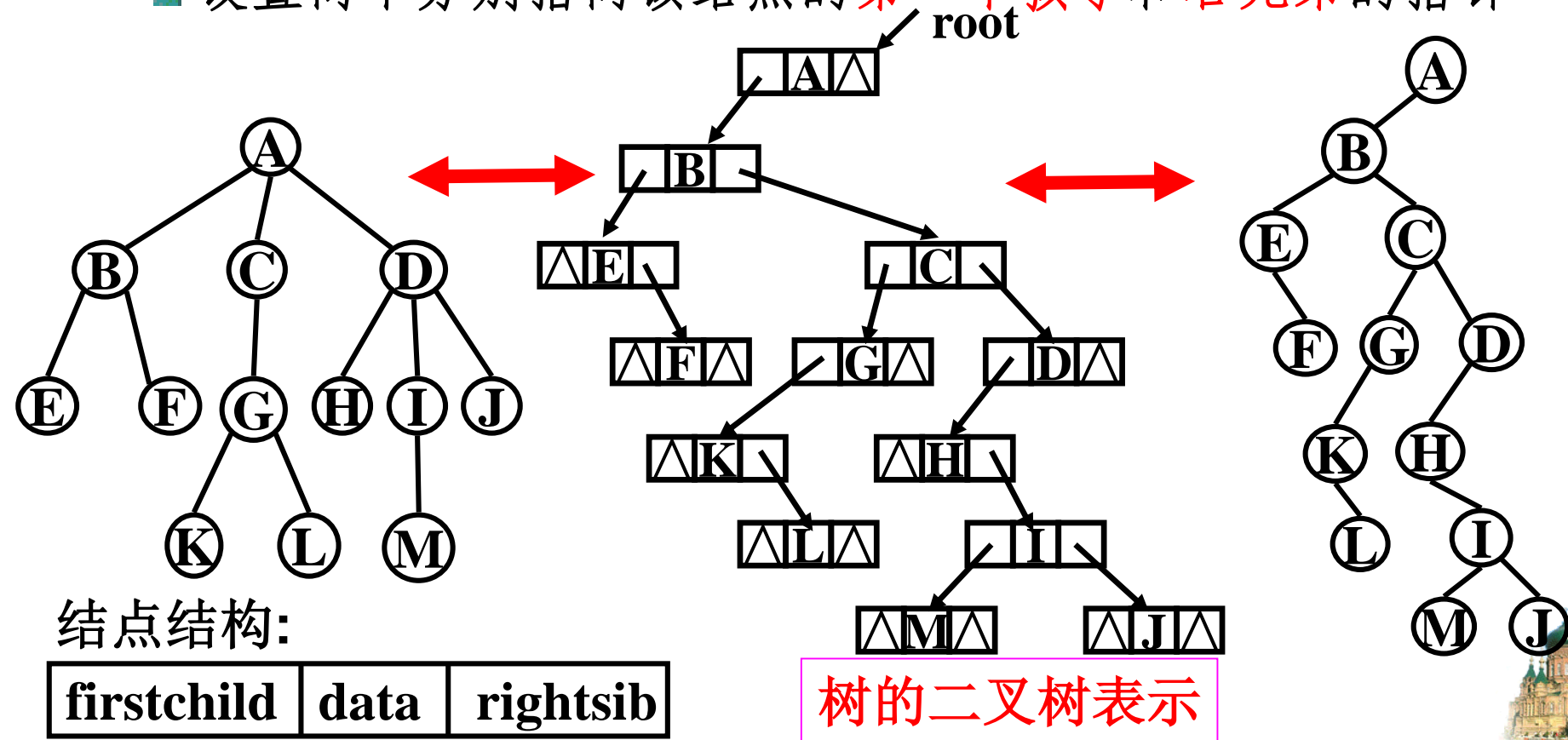




## 3.5 树 (Cont.)

### ➤ 二叉链表表示法 ((左)孩子—(右)兄弟链表表示)

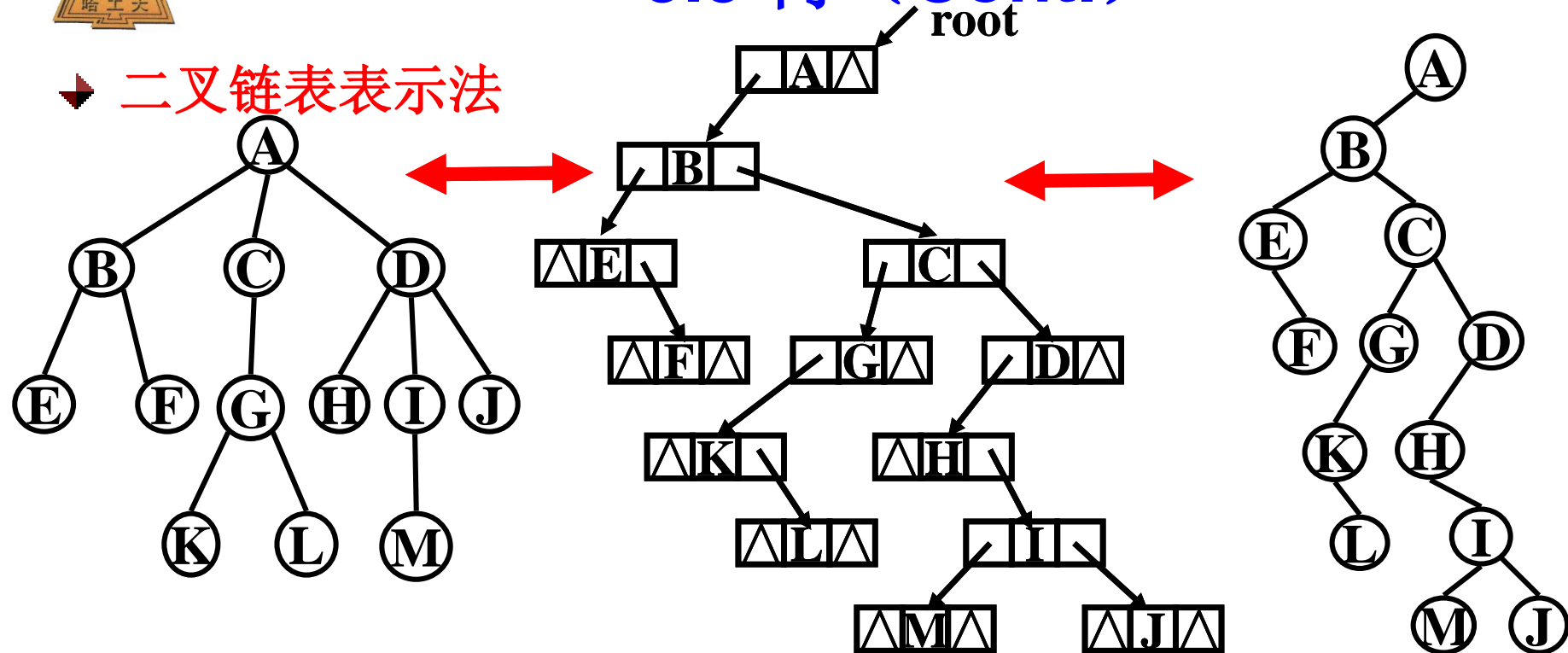
- 某结点的**右兄弟**是唯一的
- 设置两个分别指向该结点的**第一个孩子**和**右兄弟**的指针





## 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.）

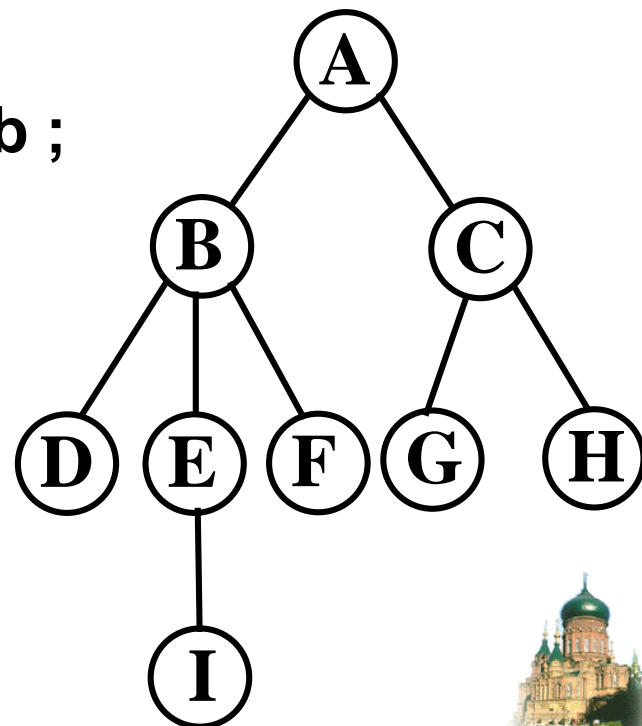
### 二叉链表表示法（（左）孩子—（右）兄弟链表表示）

■ 结点结构: 



firstchild	data	rightsib
------------	------	----------

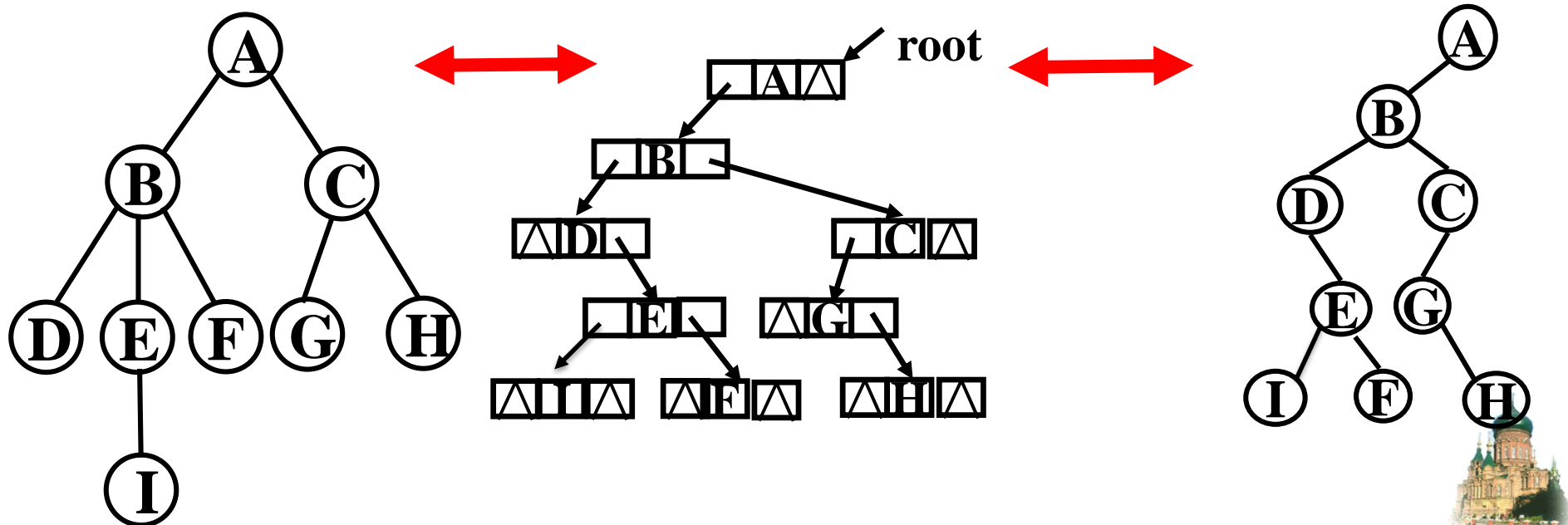
■ 类型定义:

```
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	二叉树
结点关系	兄弟关系 	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度角 (左链竖画, 右链横画)

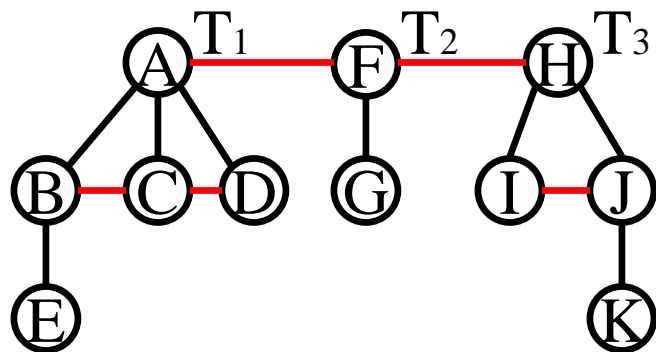




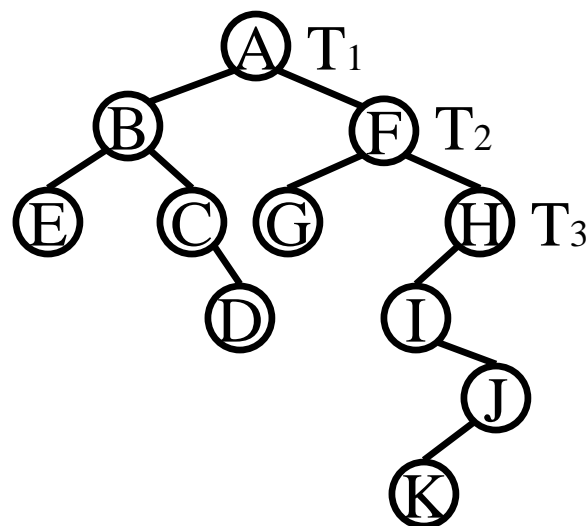
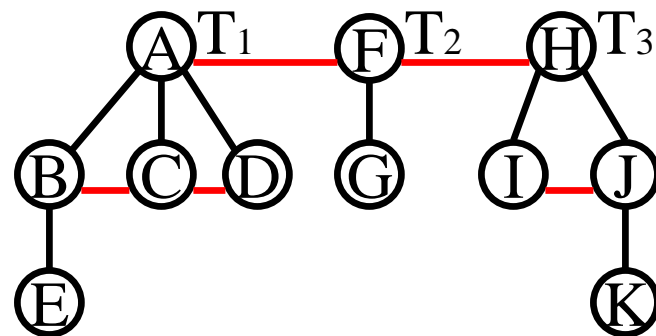
## 3.6 森林( 树)与二叉树间的转换 (Cont.)

### 森林( 树)转换成二叉树

连线:



抹线:



旋转:







## 3.6 森林( 树)与二叉树间的转换 (Cont.)

### ➡ 二叉树转换成森林( 树)

#### ■ 连线:

- 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;

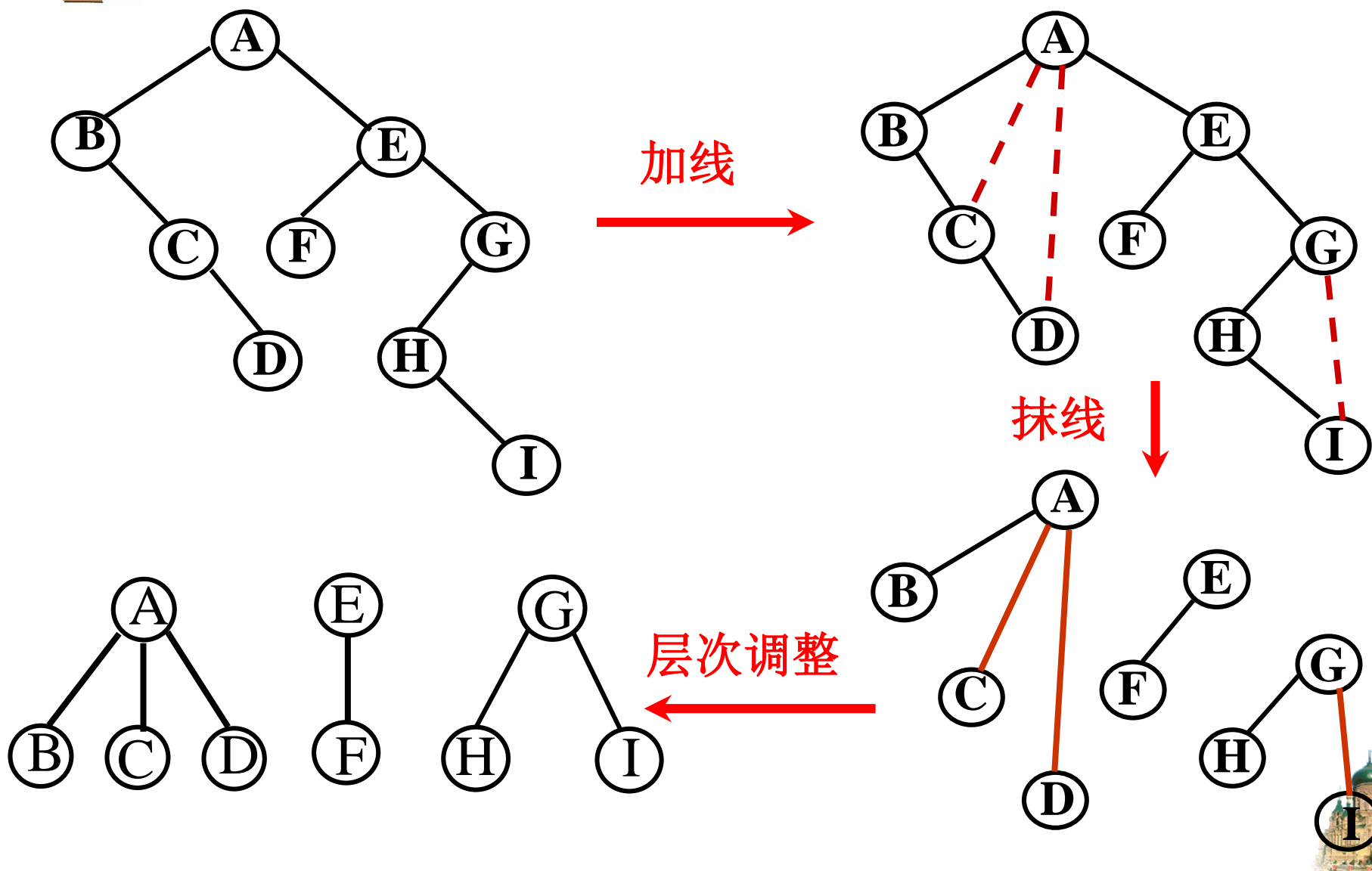
#### ■ 旋转:

- 按逆时针旋转45度角 (即把结点按层次排列)





## 3.6 森林( 树)与二叉树间的转换 (Cont.)





## 3.6 森林( 树)与二叉树间的转换 (Cont.)

### 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，该没有右子树；
- 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；





## 3.6 森林( 树)与二叉树间的转换 (Cont.)

### 森林(树)转换成二叉树的递归算法

■  $F = \{T_1, T_2, \dots, T_n\}$       二叉树  $B(F)$

■ 若  $n=0$ , 则  $B(F)$  为空; 否则,  $n > 0$ , 则

●  $B(F)$  的根就是  $\text{root}(T_1)$  ;

●  $B(F)$  的左子树是  $F$  的第一棵树  $T_1$  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  $T_1$  的根对应二叉树  $B$  的根;

●  $T_1$  中根结点的子树森林  $F_1$  是由  $B$  的左子树转换来的;

●  $F$  中除  $T_1$  之外其余子树组成的森林  $F' = \{T_2, \dots, T_n\}$  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 (Cont.)

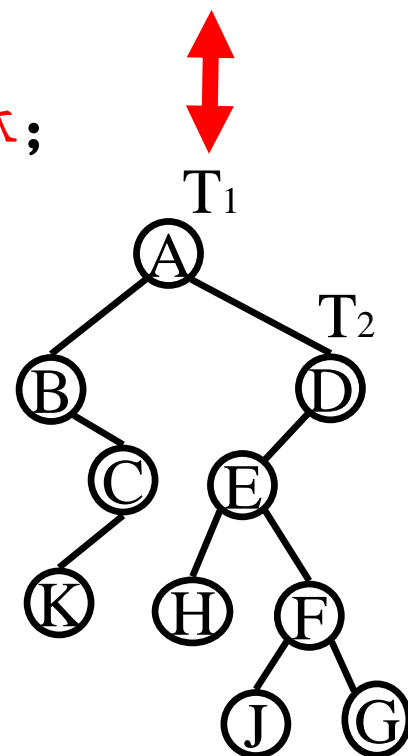
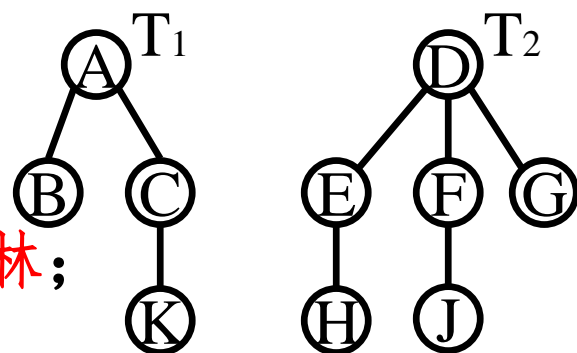
### 非空森林的基本遍历

#### 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	√	√	√

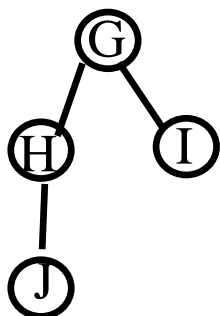
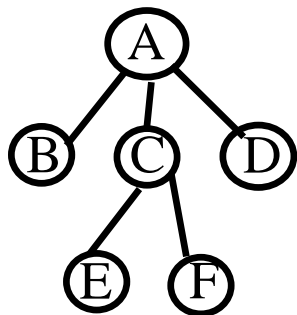




**思考题：**森林采用带度数的后序遍历序列存储，设计算法将其转换成二叉树形式，二叉树的存储方案为 [child][data][sibling]。

森林的存储形式如下：

data	B	E	F	C	D	A	J	H	I	G
degree	0	0	0	2	0	3	0	1	0	2

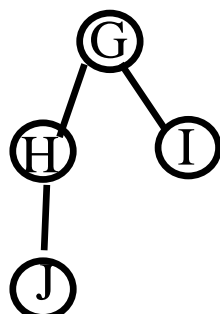
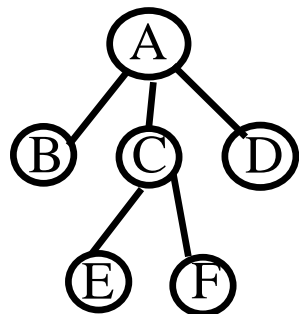




**思考题：**森林采用带度数的后序遍历序列存储，设计算法将其转换成二叉树形式，二叉树的存储方案为 [child][data][sibling]。

森林的存储形式如下：

data	B	E	F	C	D	A	J	H	I	G
degree	0	0	0	2	0	3	0	1	0	2



算法思想：

1. 如果结点度数 $m$ 为0，入栈；
2. 度数 $m$ 不为0，则弹出 $m$ 个节点作为该节点的子树，新生成的树入栈；
3. 重复执行直到数组中所有节点处理完毕，栈中的节点即为每颗树的根。
4. 依次弹出栈中节点，连接为右子树。





## 3.7 树型结构的应用

### 用树结构表示集合

#### ADT 集合 MFSET

##### 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(  $S_i, S_j, S$  ) : If  $S_i \cap S_j = \Phi$ ,  $S = S_i \cup S_j$ ;

● Find(  $i, S$  ) : 求包含  $i$  的集合;

● Initial(  $A, x$  ) : 建立集合  $A$ , 使之只包含  $x$ 。

■ 例如,  $S_1 = \{1, 7, 8, 9\}$ ,  $S_2 = \{2, 5, 10\}$ ,  $S_3 = \{3, 4, 6\}$ , 则  $S_1 \cup S_2 = \{1, 2, 5, 7, 8, 9, 10\}$







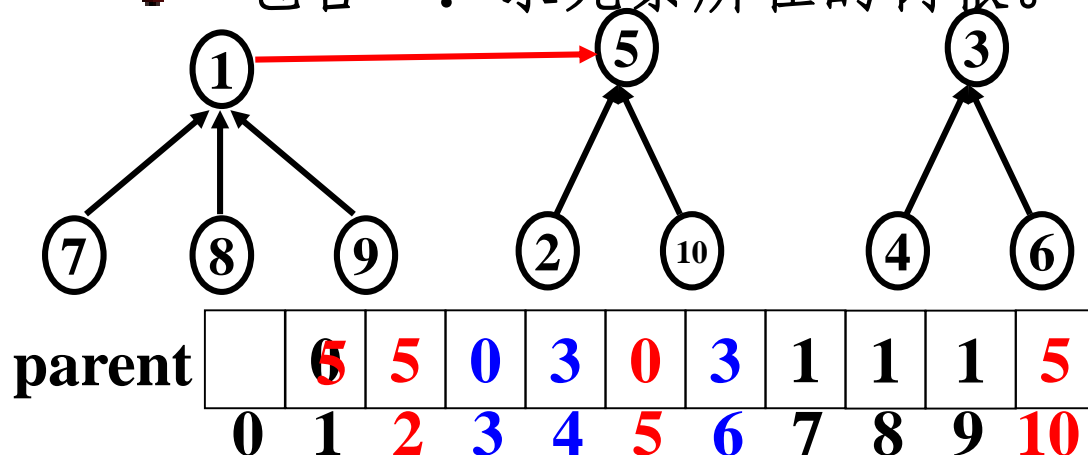
## 3.7 树型结构的应用 (Cont.)

### 用树结构表示集合

#### 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。



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.)

用树结构表示集合

➡ 集合的存储结构

#define n 元素的个数

typedef int MFSET[n+1];

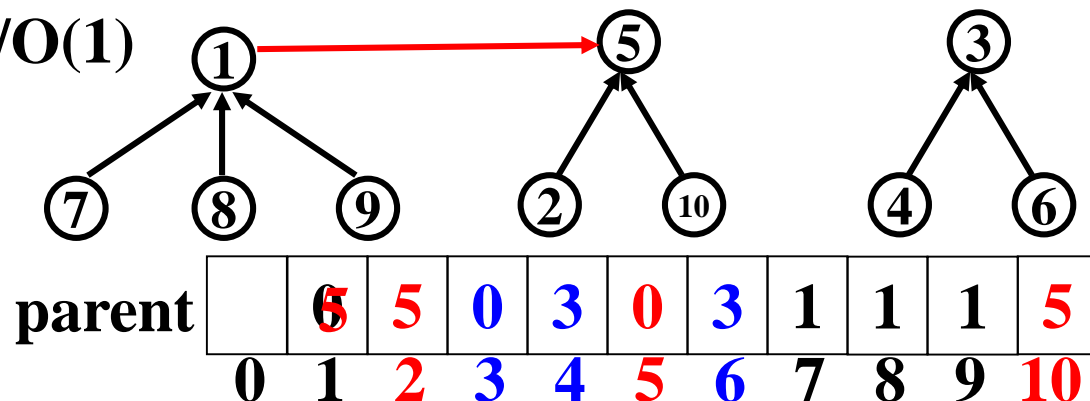
/\* 集合的“型”为MFSET,元素的“型”为int \*/

➡ 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)





## 3.7 树型结构的应用 (Cont.)

用树结构表示集合

➡ 基本操作的实现

int Find(int i, MFSET parent)

{ int tmp=i;

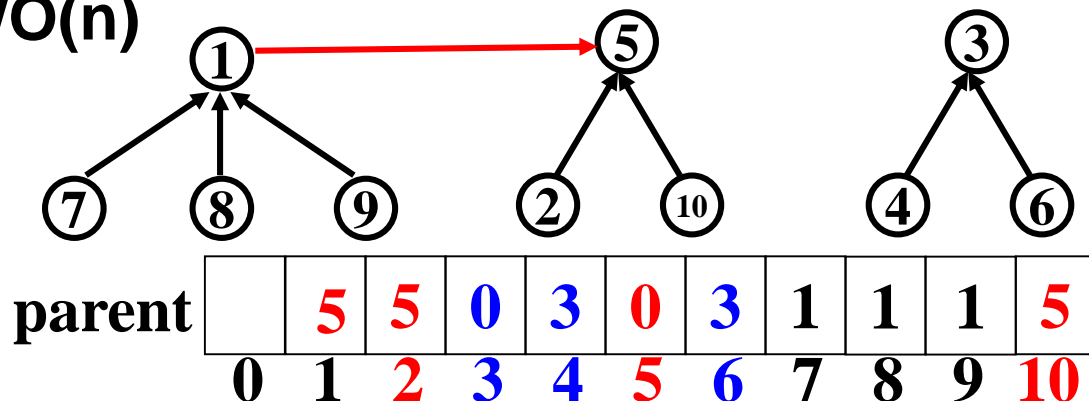
while(parent[tmp]!=0)/\* >0,未到根 \*/

tmp=parent[tmp]; /\* 上溯 \*/

return tmp;

}//O(n)

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





## 3.7 树型结构的应用 (Cont.)

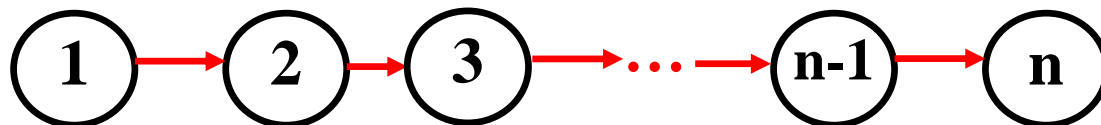
### 用树结构表示集合

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.)

用树结构表示集合

➡ 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**；  
另外，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.)

用树结构表示集合

➡ 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;
```

```
}
```

```
}
```





## 3.7 树型结构的应用 (Cont.)

### 用树结构表示集合

#### ➡ 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;
}

void Initial(int A ,MFSET C)
{
    C[x].father=0;
    C[x].count=1;
}
```





## 3.7 树型结构的应用 (Cont.)

### 用树结构表示集合

#### 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**:  $x \in S, y \in S, x \equiv y \Leftrightarrow (x, y) \in R$  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:  $1 \equiv 2, 5 \equiv 6, 3 \equiv 4, 1 \equiv 4$







## 3.7 树型结构的应用 (Cont.)

### 用树结构表示集合

#### ➡ 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,  $S_1, S_2, \dots, S_7$
- 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 $S_k$  和 $S_m$  (不失一般性)
  - ◆ 2.2 若 $S_k \neq S_m$ , 则将 $S_k$  并入 $S_m$ , 并将 $S_k$  置空。

- 当所有的等价对处理过后,  $S_1, S_2, \dots, S_7$  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
        cin<<i<<j;
    }
}
```





## 3.7 树型结构的应用 (Cont.)

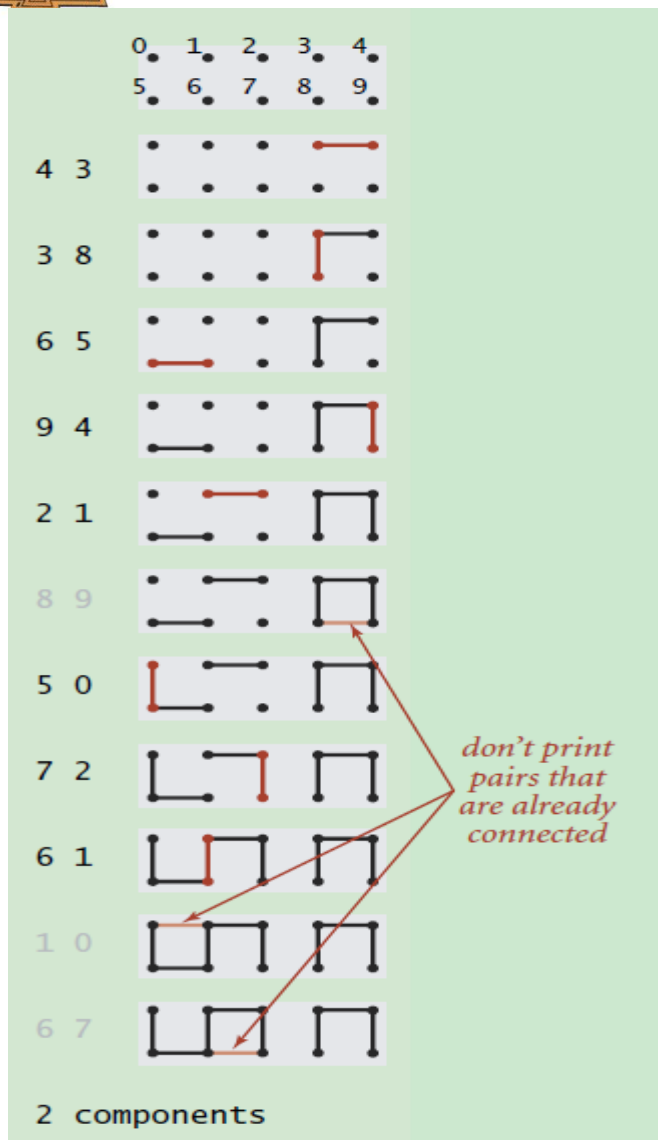
### 问题1：动态连通性

输入了一组整数对，即上图中的 (4, 3) (3, 8) 等等，每对整数代表这两个 points/sites 是连通的。那么随着数据的不断输入，整个图的连通性也会发生变化，从上图中可以很清晰的发现这一点。同时，对于已经处于连通状态的 points/sites，直接忽略，比如上图中的 (8, 9)

动态连通性的应用场景：

**网络连接判断：**如果每个 pair 中的两个整数分别代表一个网络节点，那么该 pair 就是用来表示这两个节点是需要连通的。那么为所有的 pairs 建立了动态连通图后，就能够尽可能少的减少布线的需要。

**变量名等同性 (类似于指针的概念)：**在程序中，可以声明多个引用来指向同一对象，就可以通过为程序中声明的引用和实际对象建立动态连通图来判断哪些引用实际上是指向同一对象。





## 3.7 树型结构的应用 (Cont.)

### 对问题建模:

解决的问题是什么，模型中选择的数据结构和算法显然会根据问题的不同而不同，就动态连通性这个场景，需要解决的问题可能是：

- 给出两个节点，判断它们是否连通，如果连通，不给出具体的路径
- 给出两个节点，判断它们是否连通，如果连通，给出具体的路径

就以上问题而言，区别是能够给出具体路径，致了选择算法的不同，数据结构也不同。不需要具体路径的Union-Find算法，而第二种情况可以使用基于DFS的算法。





## 3.7 树型结构的应用 (Cont.)

### 问题2：查找朋友圈

班上有  $N$  名学生。其中有些人是朋友，有些则不是。他们的友谊具有传递性。如果已知  $A$  是  $B$  的朋友， $B$  是  $C$  的朋友，那么可以认为  $A$  也是  $C$  的朋友。（所谓的朋友圈，是指所有朋友的集合）





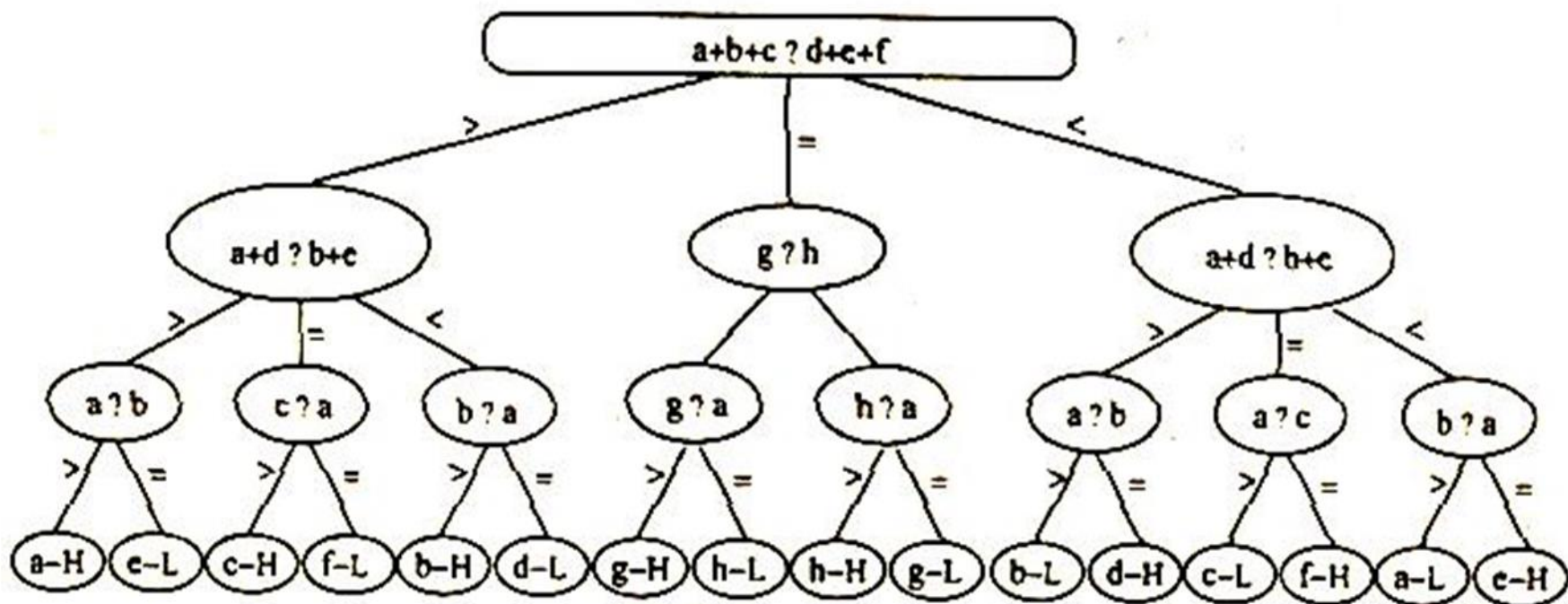
## 3.7 树型结构的应用 (Cont.)

### 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。

#### 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.)

### 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**;
- 一个解决过程的执行对应于通过**根到叶的一条路**
- 一个判定树是**所有可能的解的集合**



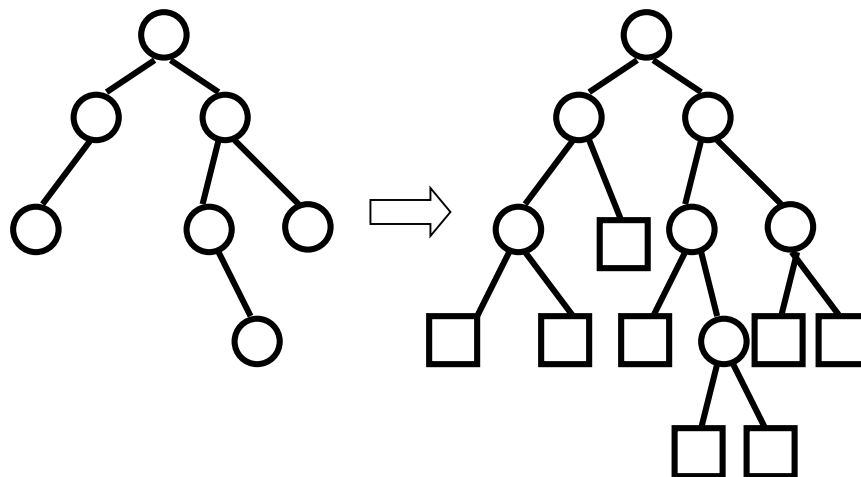


## 3.7 树型结构的应用 (Cont.)

### 哈夫曼 (Huffman) 树

#### 一、哈夫曼树的由来及其构造

1. 扩充二叉树 (增长树) { 内结点 ○  
外结点 □



在二叉树中，对于每个结点，若出现空（左/右）子树，则为其加上一个特殊的结点（称为外结点），由此得到的新二叉树称为原二叉树的扩充二叉树，而原二叉树的结点称为内结点。

- 没有度数为1的结点
- 外结点数 = 内结点数 + 1 (为什么?)
- 有  $n$  个外结点的扩充二叉树共有  $2n-1$  个结点。



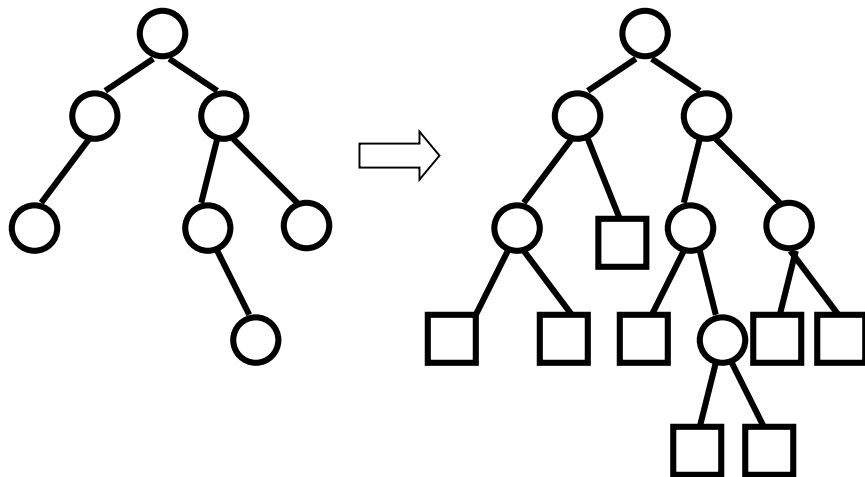




## 3.7 树型结构的应用 (Cont.)

### 2. 扩充二叉树的外路径长、内路径长及相互关系

- 外路径长 $E$ : 从根结点到每个外结点的路径长之和,  $E = \sum l_j$
- 内路径长  $I$ : 从根结点到每个内结点的路径长之和
- 关系:  $E = I + 2 \times n$  ( $n$ 为内结点的个数)



内路径长  $I = 2 \times 1 + 3 \times 2 + 1 \times 3 = 11$

外路径长  $E = 1 \times 2 + 5 \times 3 + 2 \times 4 = 25$





## 3.7 树型结构的应用 (Cont.)

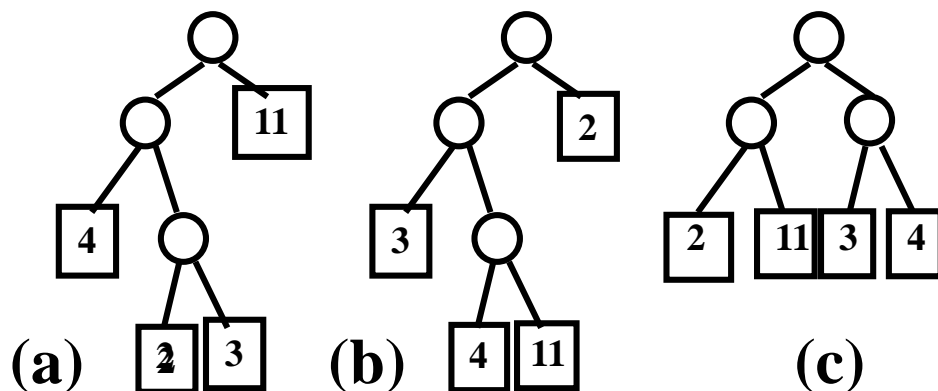
### 3. 扩充二叉树的加权路径长

#### ■ 外结点的加权路径长:

设每个外结点  $j$  对应一个实数  $w_j$  (称为外结点的权值),  $l_j$  是从根到外结点  $j$  的路长, 则  $w_j \cdot l_j$  称为外结点  $j$  的加权路径长。

#### ■ 扩充二叉树的加权路径长:

$WPL = \sum w_j \cdot l_j$  称为扩充二叉树的加权路径长。



设:  $w_i = \{2, 3, 4, 11\}$

求:  $\sum w_j \cdot l_j$  (加权路径长)

(a)  $11 \times 1 + 4 \times 2 + 2 \times 3 + 3 \times 3 = 34$

(b)  $2 \times 1 + 3 \times 2 + 4 \times 3 + 11 \times 3 = 53$

(c)  $2 \times 2 + 11 \times 2 + 3 \times 2 + 4 \times 2 = 40$





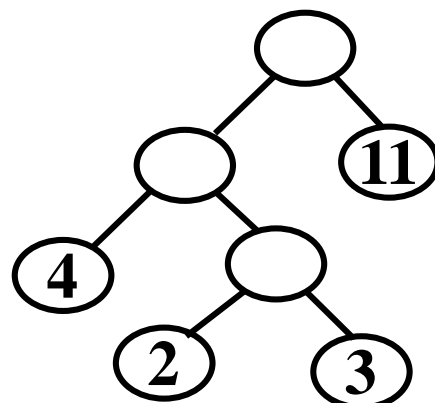
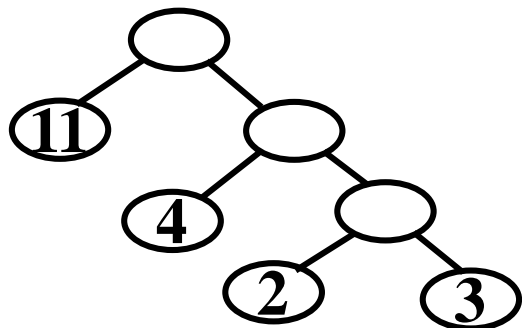
## 3.7 树型结构的应用 (Cont.)

### 4. 哈夫曼树 (最优二叉树)

在给定权值为 $w_1, w_2, \dots, w_n$ 的 $n$ 个叶结点所构成的所有扩充二叉树中,  $WPL = \sum w_j \cdot l_j$ 最小的称为huffman树。

#### ➡ 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。





## 3.7 树型结构的应用 (Cont.)

### ➤ 哈夫曼树的构造方法:

- (1) **初始化**: 由给定的 $n$ 个权值 $\{w_1, w_2, \dots, w_n\}$ 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 $F=\{T_1, T_2, \dots, T_n\}$ ;
- (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**。



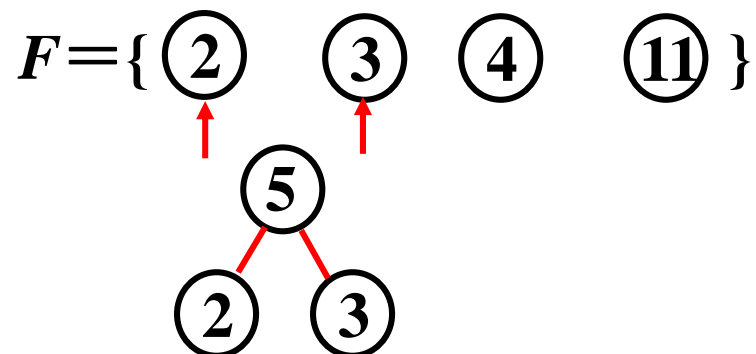


## 3.7 树型结构的应用 (Cont.)

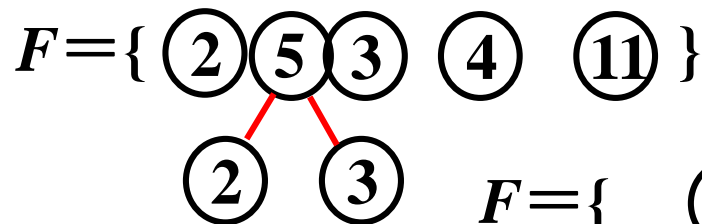
➤ 哈夫曼树的构造示例:  $W=\{2, 3, 4, 11\}$

■ 初始化:

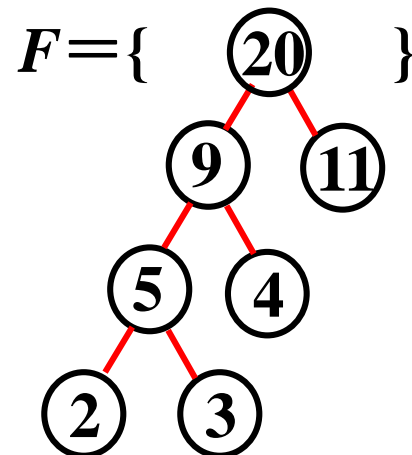
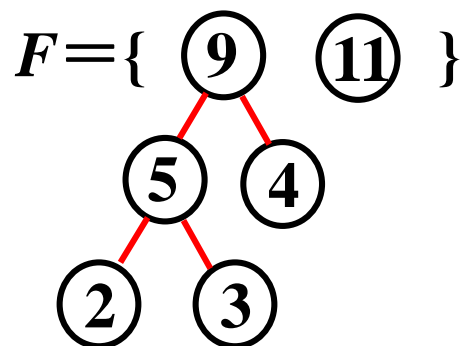
■ 选取与合并:



■ 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 $T[0], \dots, T[2n-2]$ 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 $T[0], \dots, T[n]$ , 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 $T[0], \dots, T[i-1]$ 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 $0 \leq p_1, p_2 \leq i-1$ ;
  - **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:

	weight	parent	lchild	rchild
⑦	7	-1	-1	-1
⑤	5	-1	-1	-1
②	2	-1	-1	-1
④	4	-1	-1	-1
		-1	-1	-1
		-1	-1	-1
		-1	-1	-1

初始化



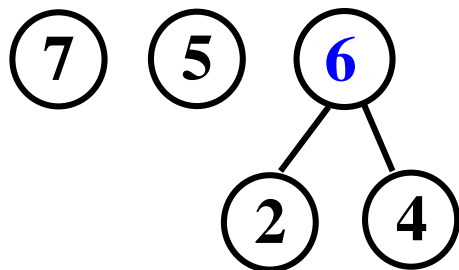




## 3.7 树型结构的应用 (Cont.)

➡ 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
$p1 \rightarrow$ 2	2	<del>4</del> -1	-1	-1
$p2 \rightarrow$ 3	4	<del>4</del> -1	-1	-1
$i \rightarrow$ 4	6	-1	<del>2</del> -1	<del>3</del> -1
5		-1	-1	-1
6		-1	-1	-1



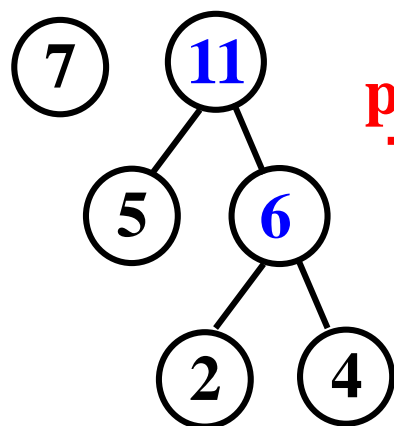
过程





## 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
<b>p1</b> → 1	5	<del>5</del> -1	-1	-1
2	2	4	-1	-1
3	4	4	-1	-1
<b>p2</b> → 4	6	<del>5</del> -1	2	3
<b>i</b> → 5	11	-1	<del>1</del> -1	<del>4</del> -1
6		-1	-1	-1

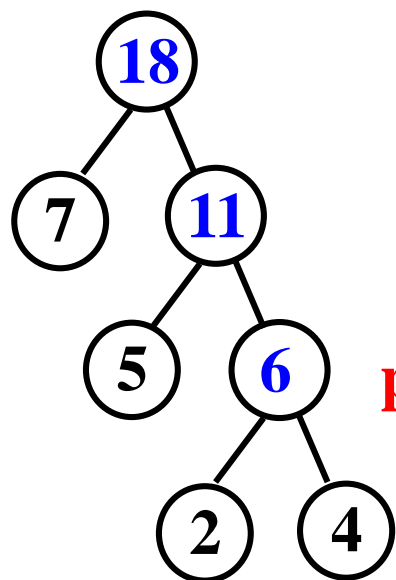
过程





## 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
<b>p1</b> → 0	7	<del>6 -1</del>	-1	-1
1	5	5	-1	-1
2	2	4	-1	-1
3	4	4	-1	-1
4	6	5	2	3
<b>p2</b> → 5	11	<del>6 -1</del>	1	4
<b>i</b> → 6	18	-1	<del>0 -1</del>	<del>5 -1</del>

过程





## 3.7 树型结构的应用 (Cont.)

### 哈夫曼树构造算法的实现

**void CreatHT(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：是指将一组对象(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





## 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 $c_j$ 的出现概率为 $p_j$ ，其二进制位串长度（码长）为 $l_j$ ，则 $\sum p_j \cdot l_j$ 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** $\sum p_j \cdot l_j$ 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），则从根到叶的路上的0、1序列就是相应字符的编码----哈夫曼编码



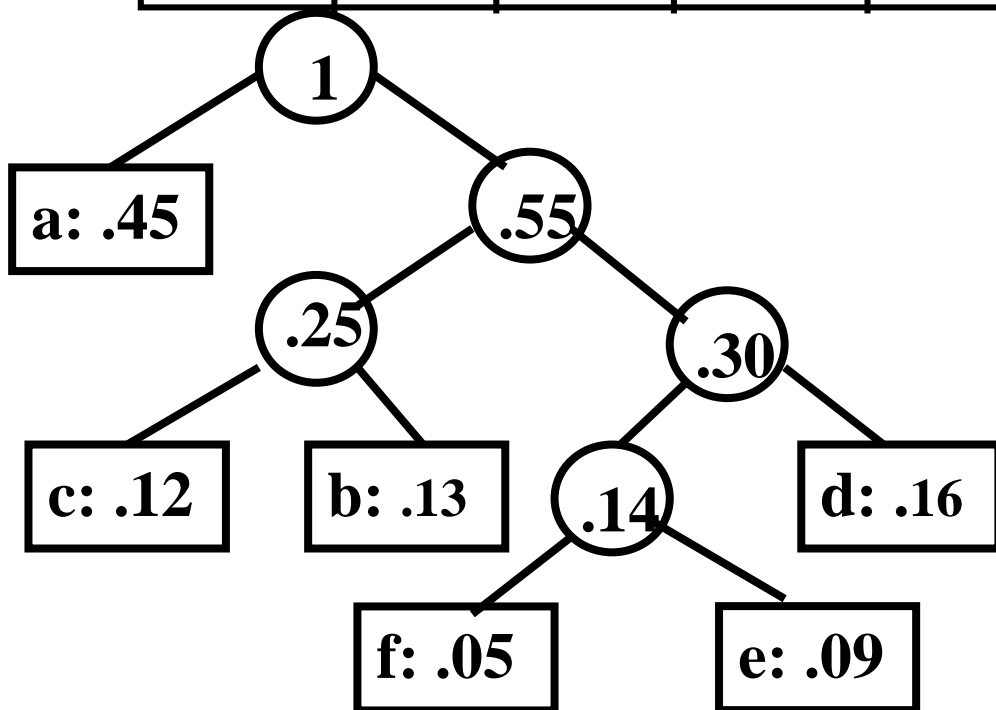




## 3.7 树型结构的应用 (Cont.)

### 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 = $\lceil \log_2  C  \rceil$
变长	0	101	100	111	1101	1100	2.24 = $\sum p_j \cdot l_j$



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

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	1 0 1 \0
2	c	1 0 0 \0
3	d	1 1 1 \0
4	e	1 1 0 1 \0
5	f	1 1 0 0 \0
6		

编码表 H





## 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	1 0 1 \0
2	c	1 0 0 \0
3	d	1 1 1 \0
4	e	1 1 0 1 \0
5	f	1 1 0 0 \0
6		

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.)

- 贪心算法是指对问题求解时，总是做出在当前看来，是最好的选择。也就是说，不从整体最优上加以考虑，某种意义上的局部最优解。
- 贪心算法注重贪心策略的选择。贪心算法不是对所有问题，都能得到整体最优解，选择的贪心策略必须具备无后效性。即某个状态以后的过程不会影响以前的状态，只与当前状态有关。所以，对所采用的贪心策略一定要仔细分析其是否满足无后效性。

问题：

某人准备修复农场围栏，需要 $n$ 段木板，这 $n$ 段木板的长短不一。他需要借锯子锯木板，每次锯都需要付费，付费标准是：当前要锯成的的木板长度。现在他拿着一个长度刚好为 $n$ 段子木板的长度总和的大木板，要把它锯成各个子木板，问：最小花费是多少？





## 3.7 树型结构的应用 (Cont.)

➡ 贪心法存在的问题:

- 1) 不能保证求得最后解是最佳的;
- 2) 不能用来求最大或最小解问题;
- 3) 只能求满足某些约束条件的可行解的范围

上述的这些问题可以在动态规划，回溯算法，分支限界算法里面得到相应的解决



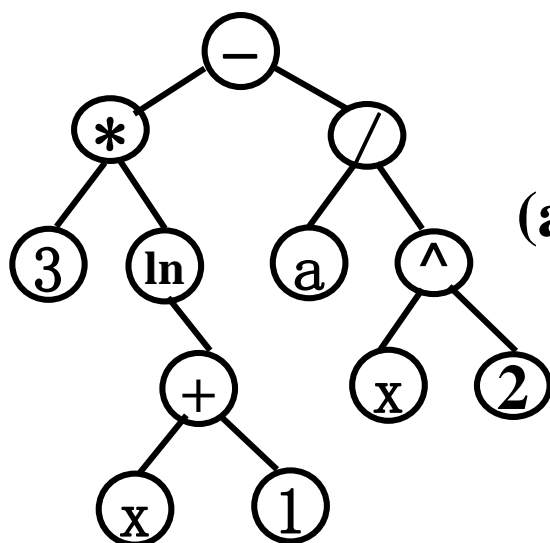


## 3.7 树型结构的应用 (Cont.)

### 树的应用—表达式求值

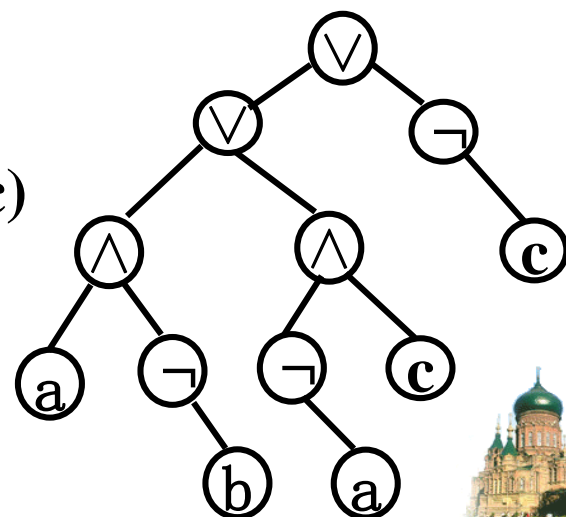
#### 用树结构表示表达式---表达式树

- 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。



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

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



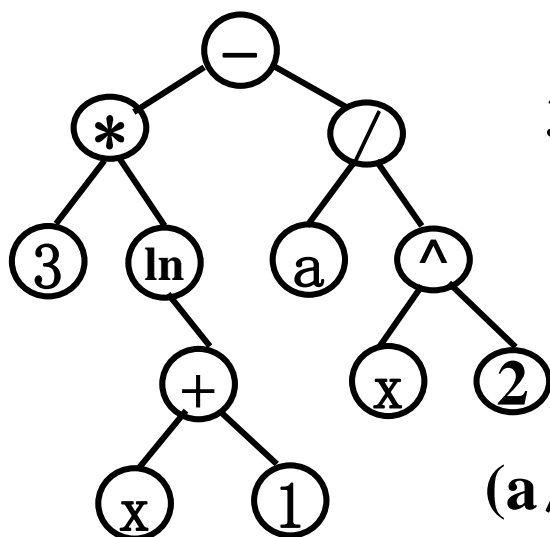


## 3.7 树型结构的应用 (Cont.)

### 树的应用—表达式求值

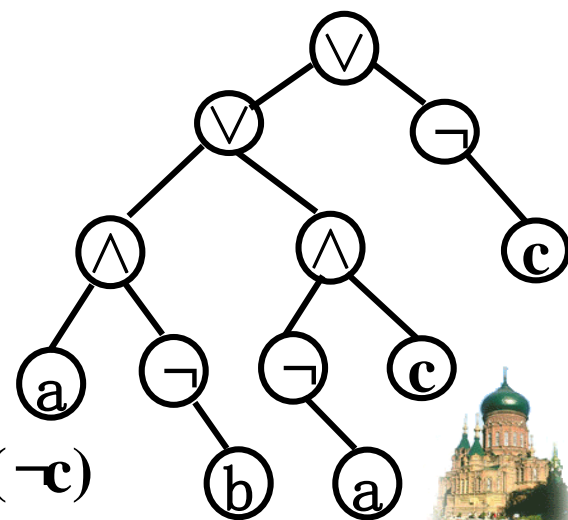
#### 表达式求值方法

- 把中缀表达式转换为后缀表达式（栈结构、树结构）；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以求出。



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

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





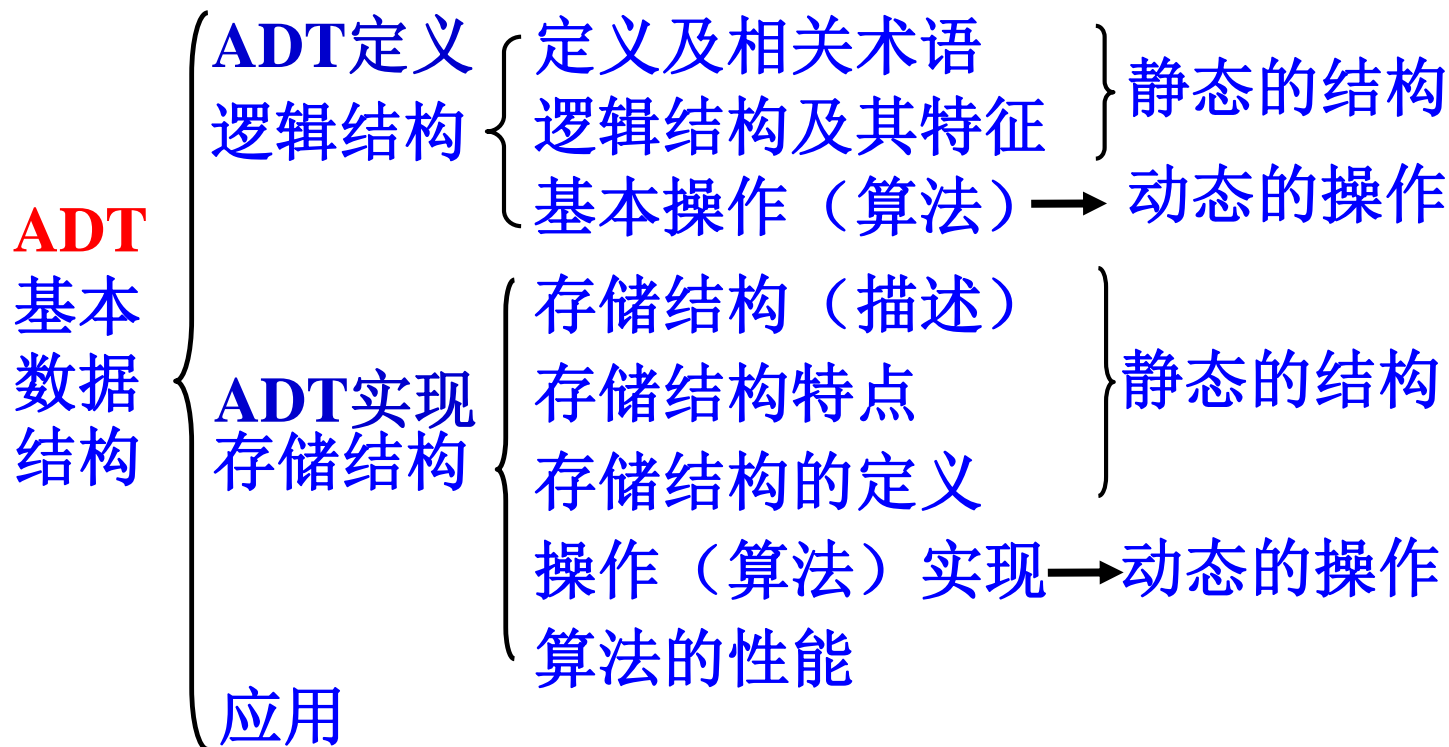


# 本章的知识点结构

## ➤ 基本的数据结构（ADT）

### ■ 二叉树、树（森林）

## ➤ 知识点结构



## ➤ 遍历算法是最核心的算法





# 本章小结

## 知识点总结

