



数据结构与算法 树

臧天仪 教授

tianyi.zang@hit.edu.cn

哈尔滨工业大学计算学学部



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，应用广泛。
- **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

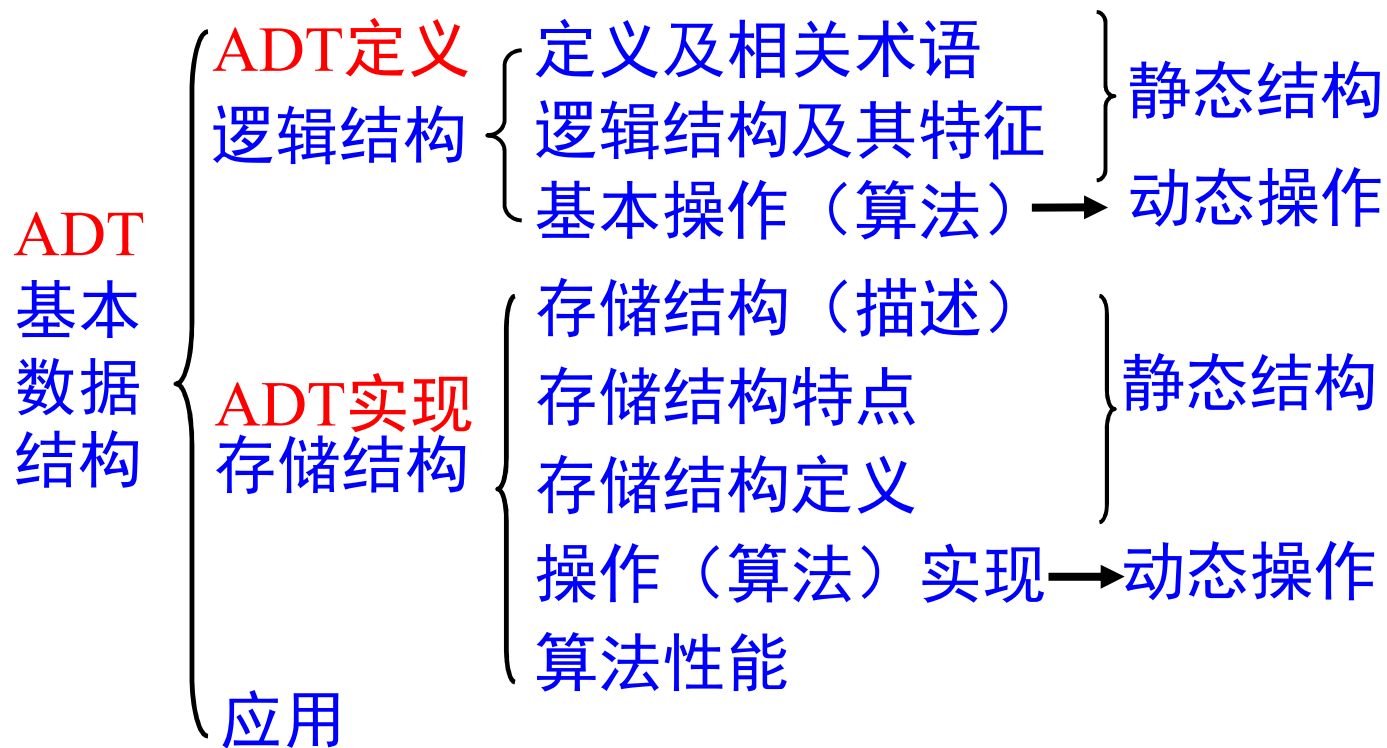


本章的知识点结构

- 基本数据结构 (ADT)

- 二叉树、树 (森林)

- 知识点结构



- 遍历算法是最核心算法



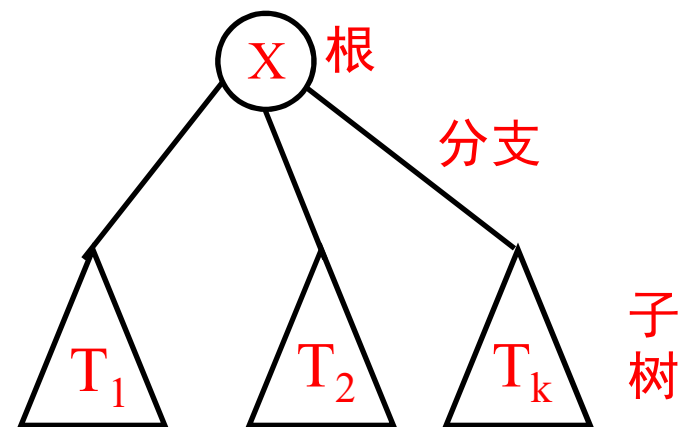
3.1 树与二叉树的基本术语

- 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, \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；

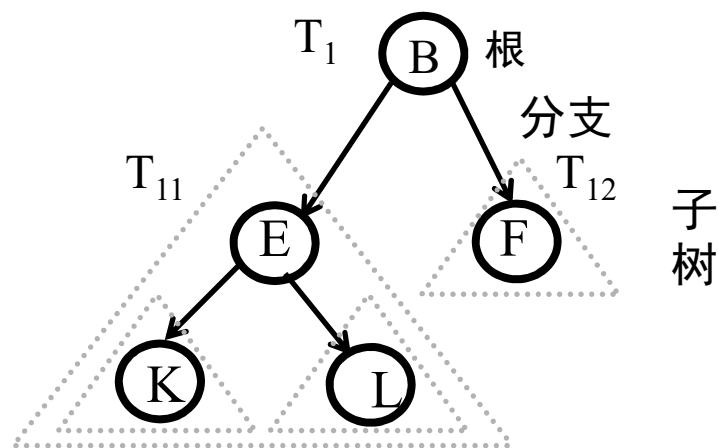
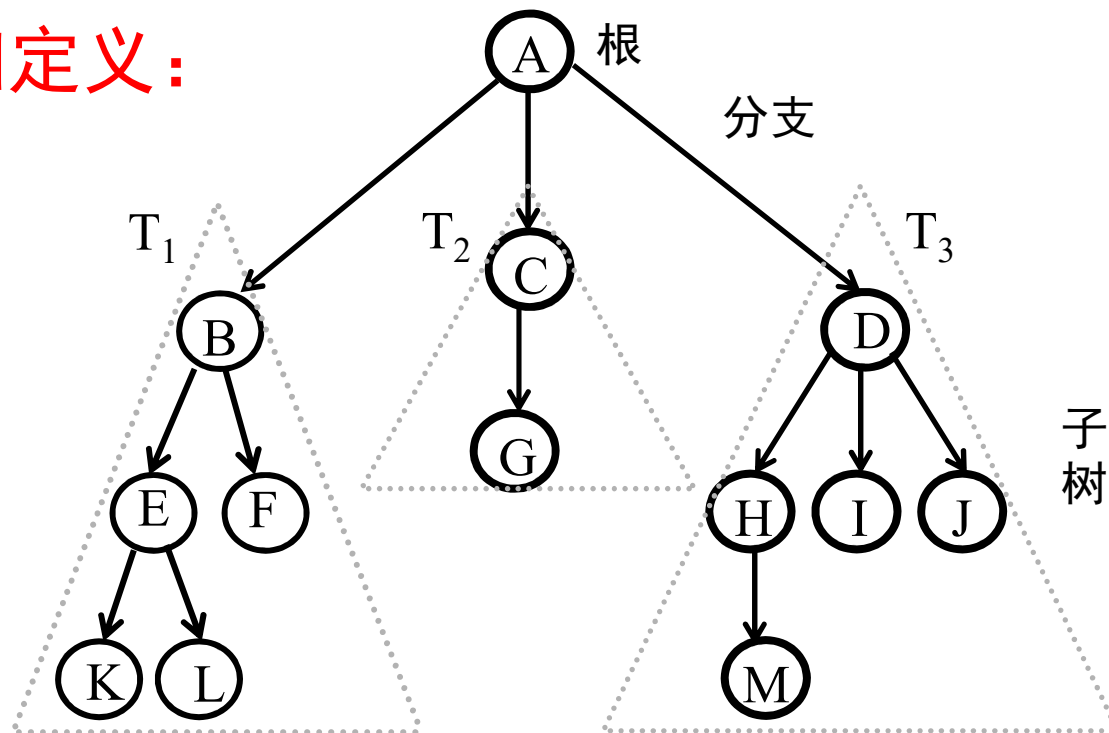
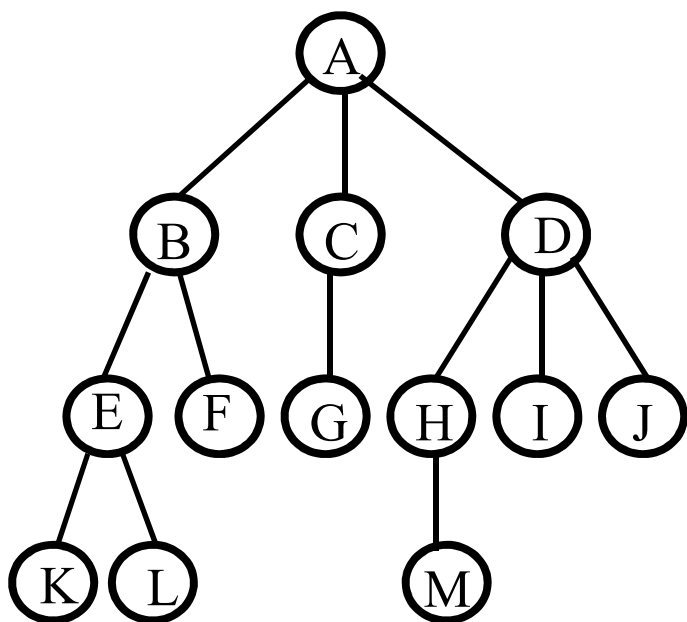




3.1 树与二叉树的基本术语(Cont.)



- 树的构造性递归定义:



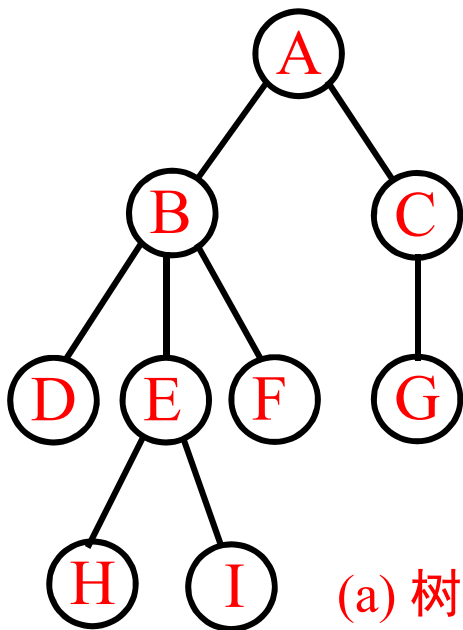


3.1 树与二叉树的基本术语(Cont.)

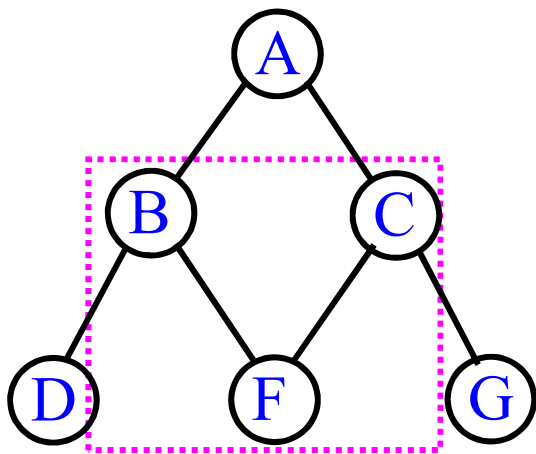


- 树的逻辑结构特点:

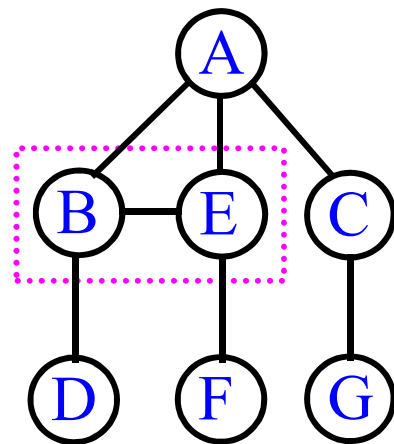
- 除根结点之外，每棵子树的根结点有且仅有一个直接前驱，但可以有0个或多个直接后继。
- 即一对多的关系，反映了结点之间的层次关系。



(a) 树结构



(b) 非树结构

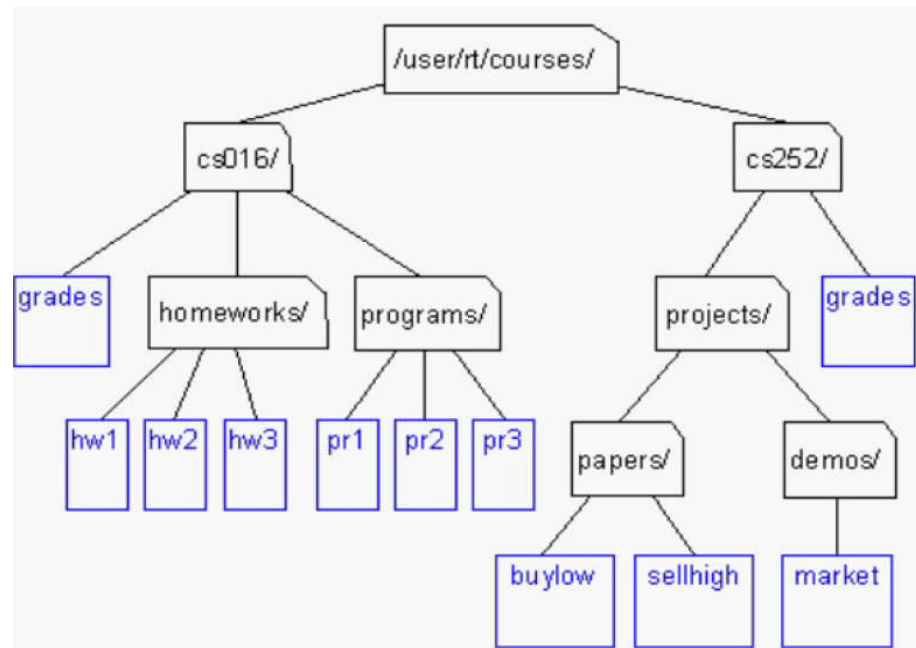


(c) 非树结构



3.1 树与二叉树的基本术语(Cont.)

- 树型结构应用：
 - 操作系统的文件管理
 - 层级架构
 - 家谱树
 - 企事业组织架构图
 - 书目录
 - 程序多层结构

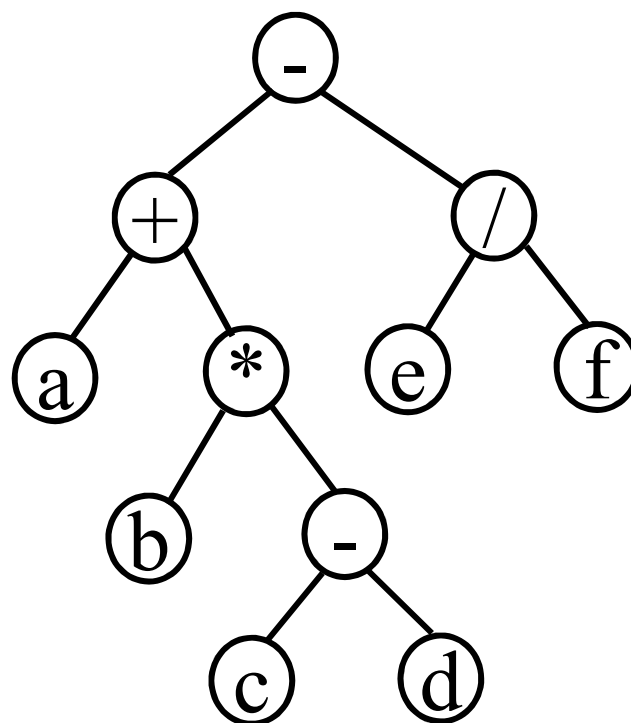




3.1 树与二叉树的基本术语(Cont.)



- 树型结构应用示例:
 - (无公共子式的)表达式的表示:



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

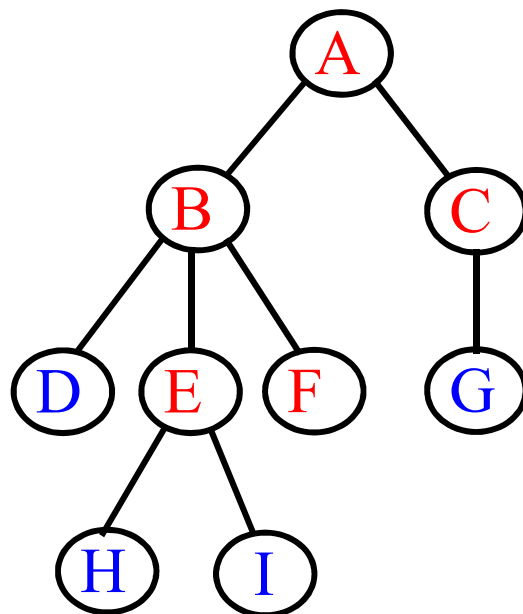


3.1 树与二叉树的基本术语(Cont.)



- **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。



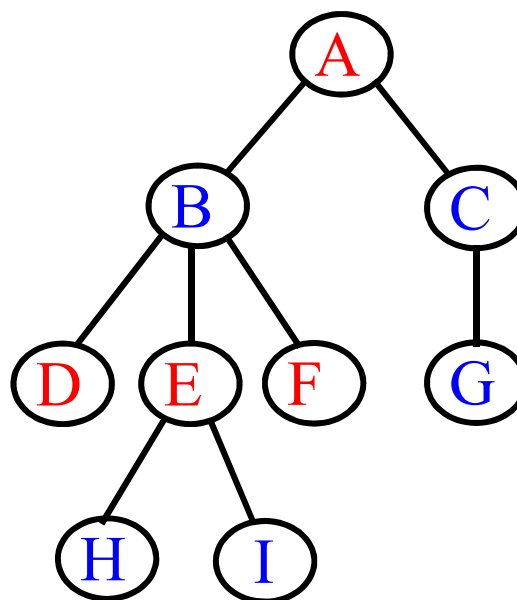


3.1 树与二叉树的基本术语(Cont.)



- 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。

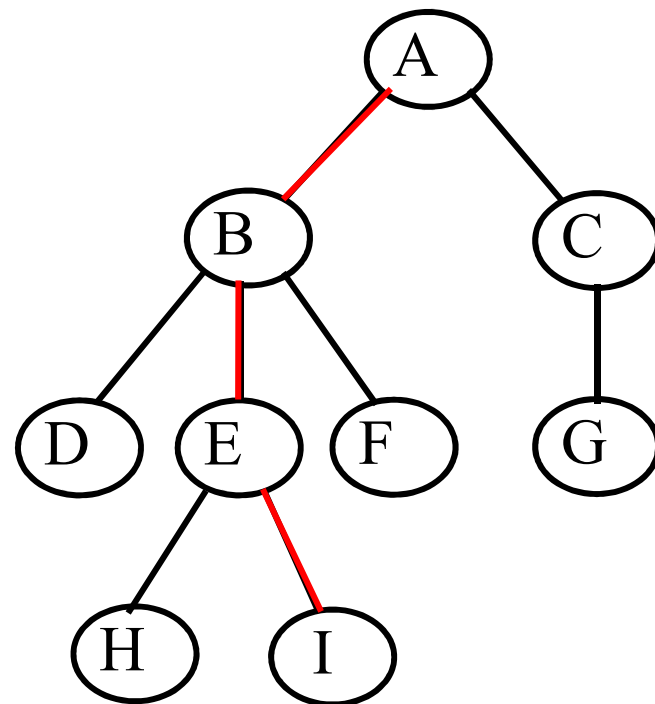




3.1 树与二叉树的基本术语(Cont.)

- 基本术语:

- **路（径）**：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）长度**：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。



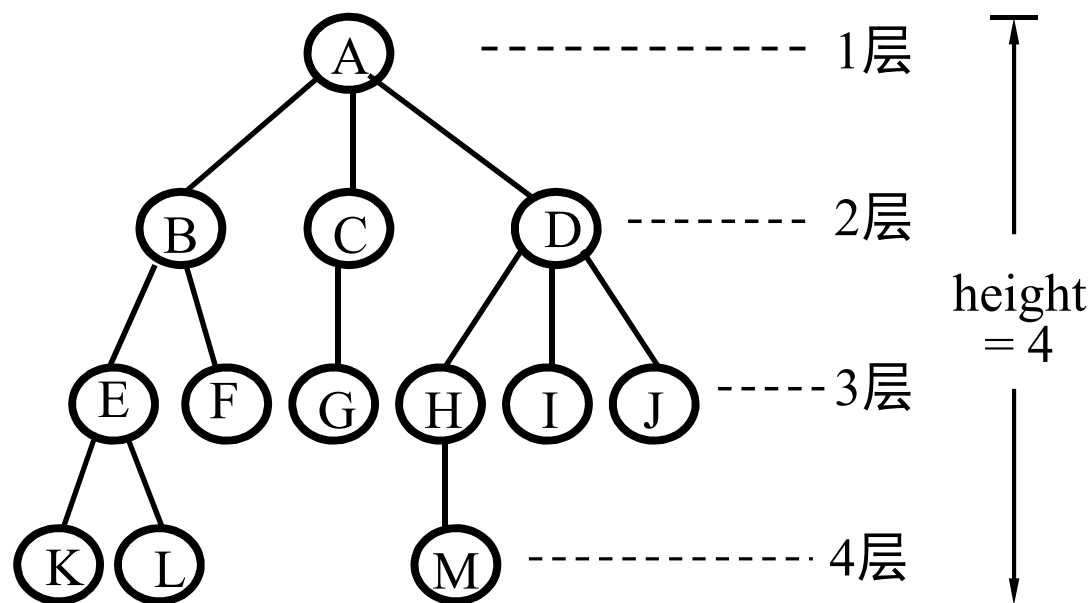


3.1 树与二叉树的基本术语(Cont.)



- 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。
 - 最长的路径长度+1



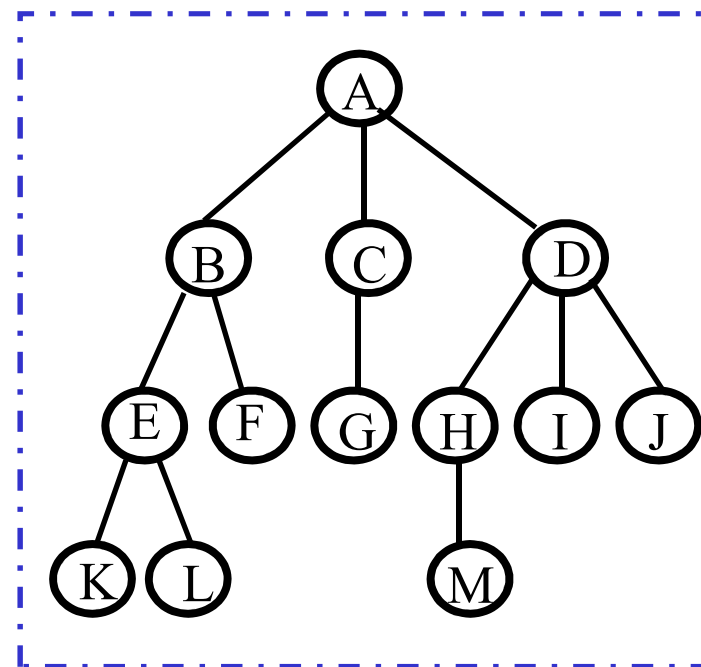
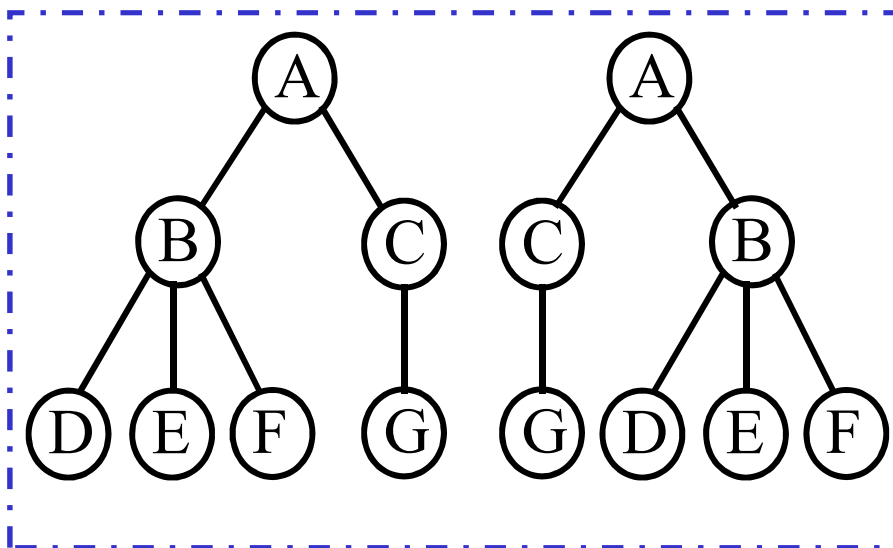


3.1 树与二叉树的基本术语(Cont.)



- 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



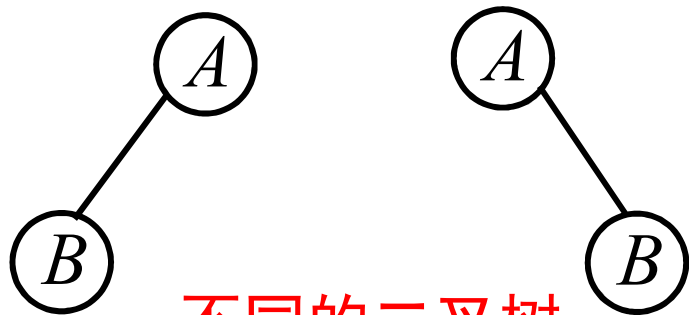
3.2 二叉树

- 二叉树(Binary Tree)的定义:

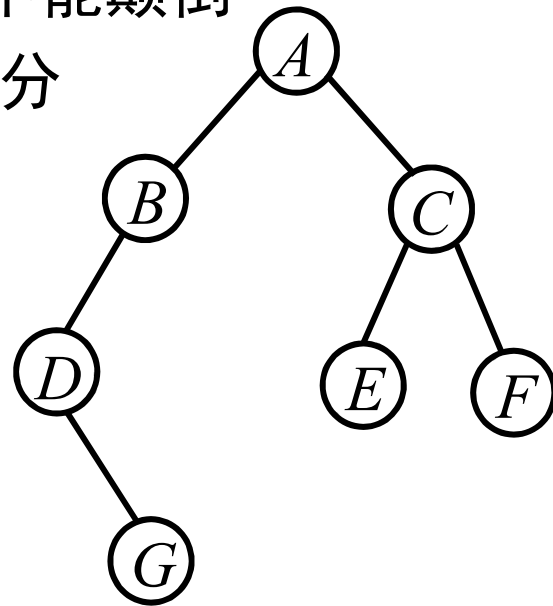
- 二叉树一个是 n ($n \geq 0$) 个结点的有限集合, 该集合或者为空 (称为空二叉树); 或者是由一个根结点和两棵互不相交的、分别称为左子树和右子树的二叉树组成。

- 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
- 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



不同的二叉树





3.2 二叉树(Cont.)

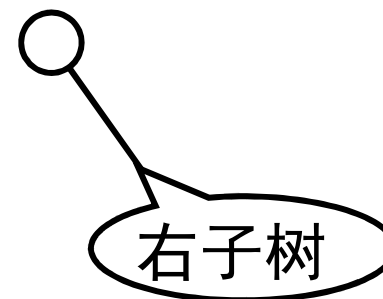
- 二叉树的基本形态:

Φ

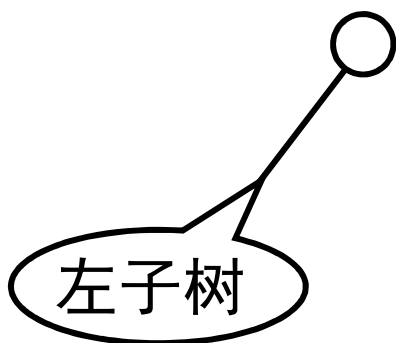
空二叉树



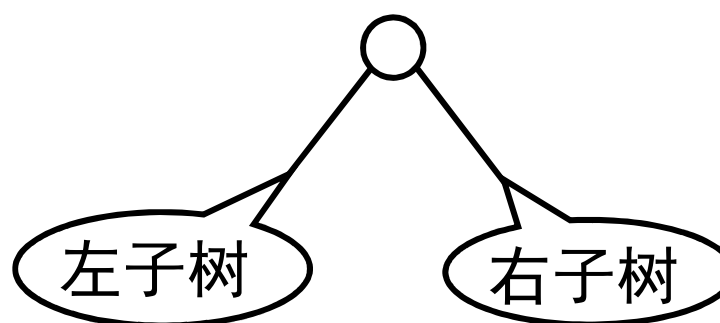
只有一个根结点



根结点只有右子树



根结点只有左子树

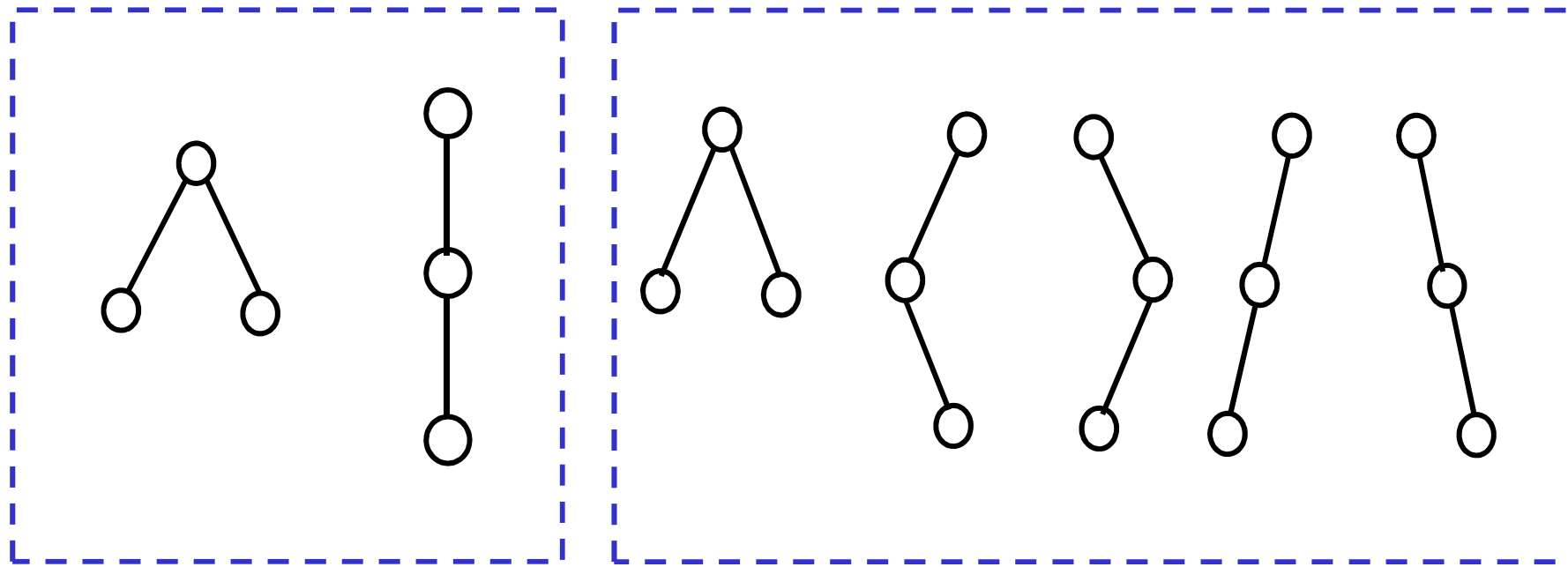


根结点同时有左右子树



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

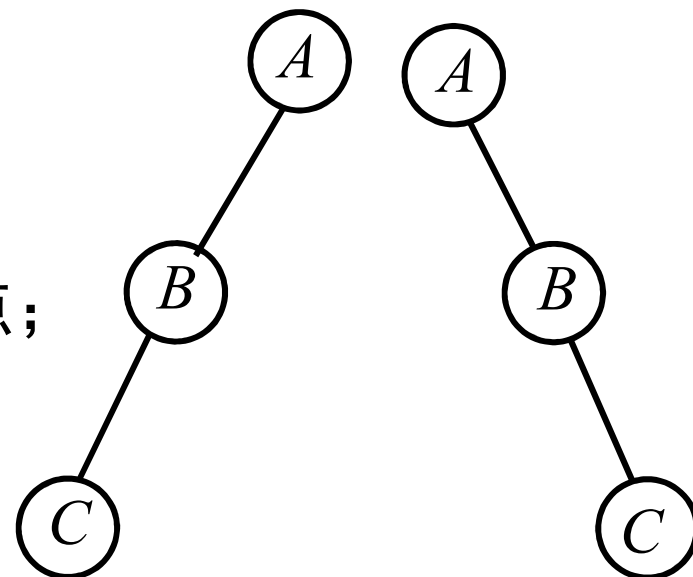
- 所有结点都只有右子树的二叉树称为右斜树

- 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

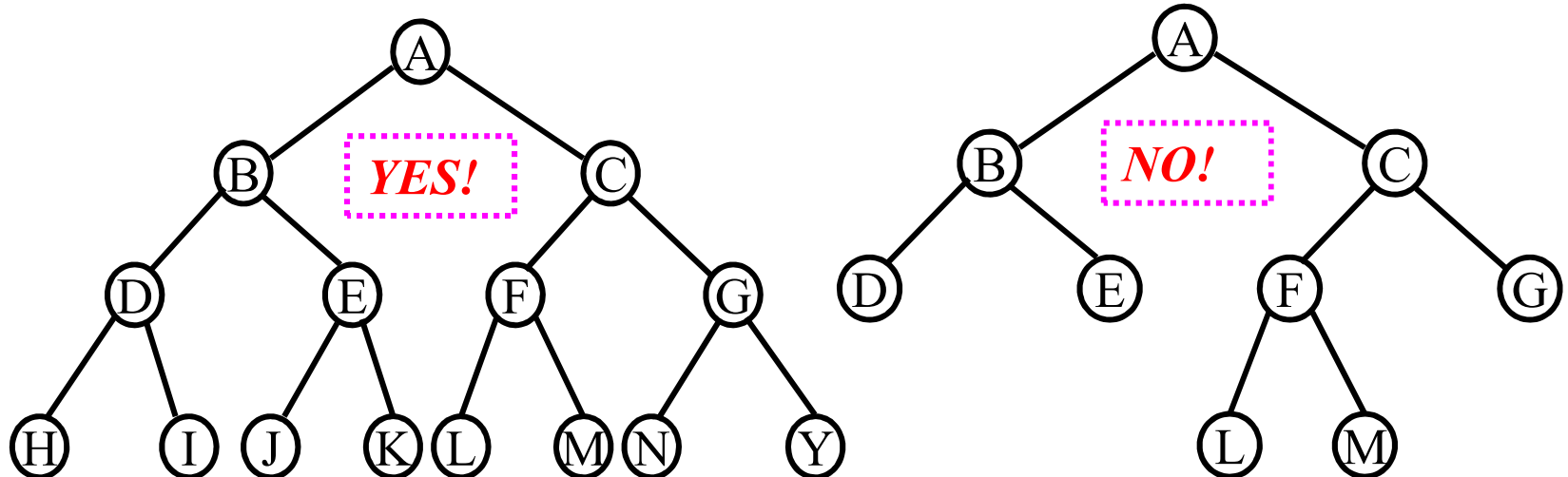




3.2 二叉树(Cont.)

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



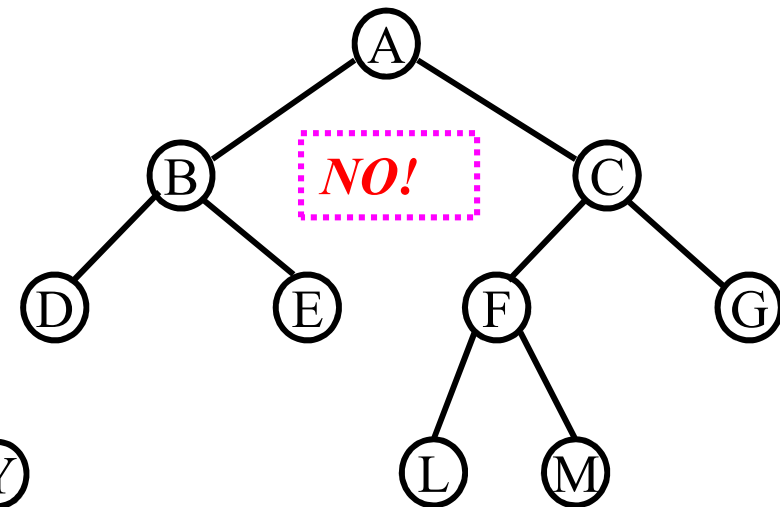
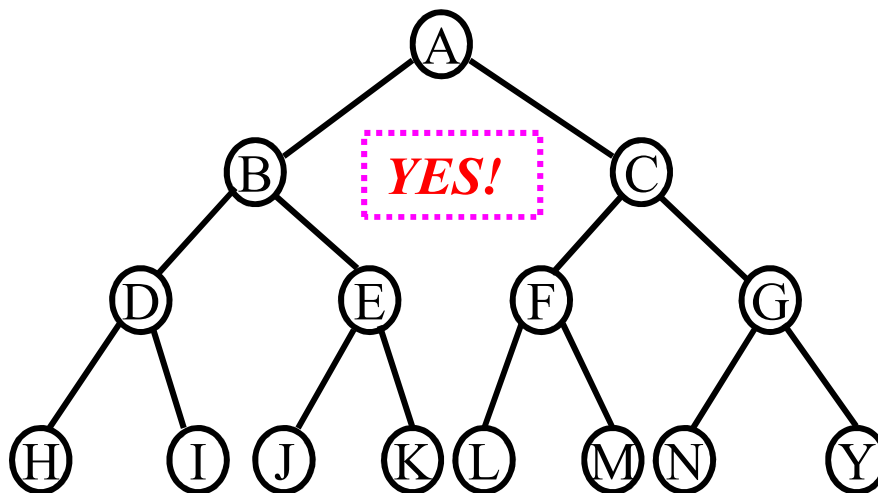


3.2 二叉树(Cont.)

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. $h-1$ 层所有叶都在非终端结点的右边；
3. 除了 $h-1$ 层**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。



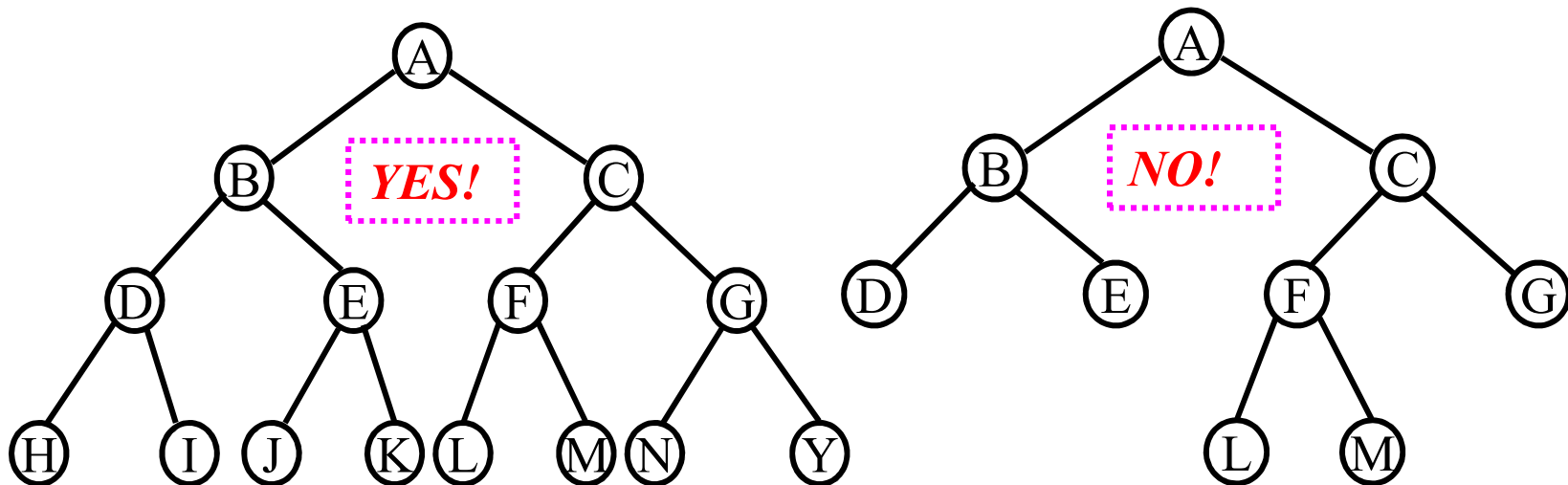


3.2 二叉树(Cont.)

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。 \square



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，最多结点数

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

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。 \square

- 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。□

- 思考:** 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 $\lceil \log_2(n+1) \rceil$ 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:

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

前 $h-1$ 层最多结点数

前 h 层最多结点数

转换 $2^{h-1} < n + 1 \leq 2^h$

取对数 $h-1 < \log_2(n+1) \leq h$

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

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

最少结点数

最多结点数

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

$$h = \lfloor \log_2 n \rfloor + 1 \square$$

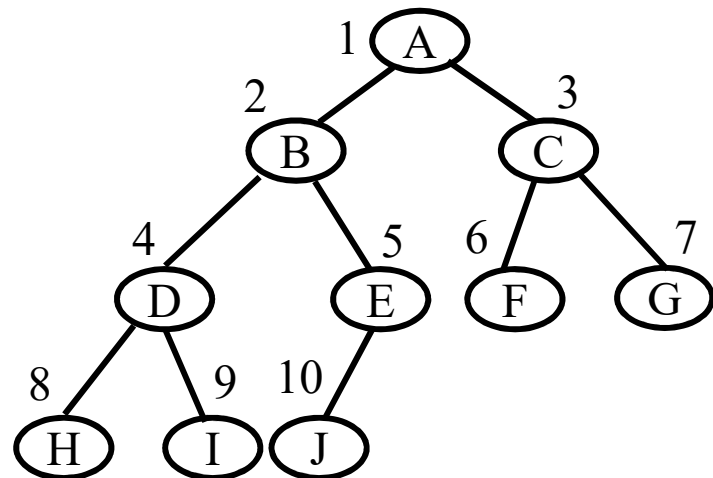
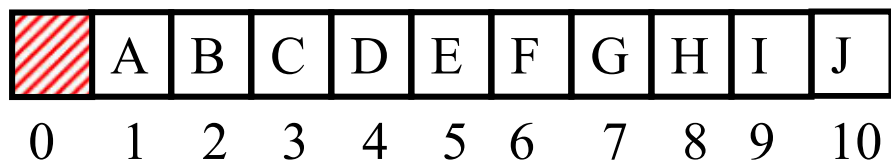


3.2 二叉树(Cont.)

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, ..., n ，且使该编号对应于数组下标，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。



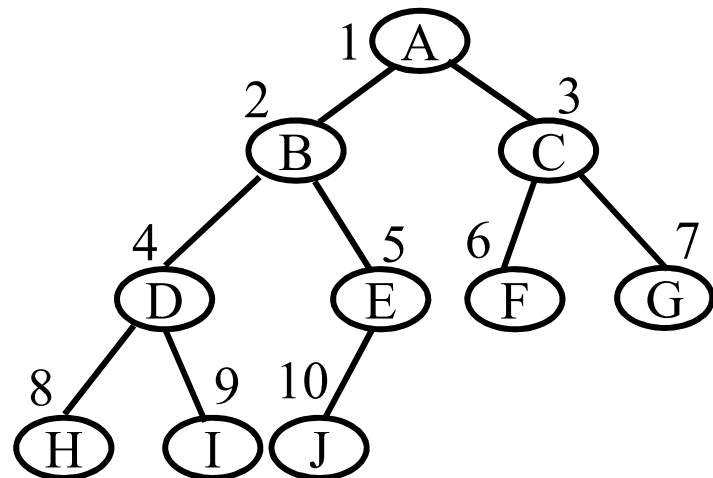
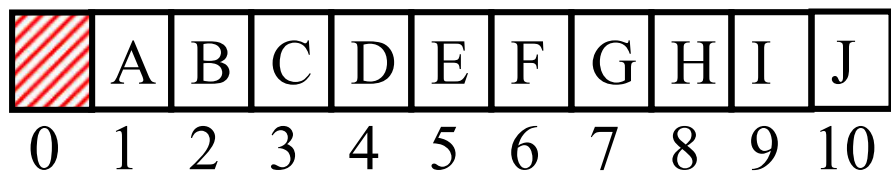


3.2 二叉树(Cont.)

性质5：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 ($1 \leq i \leq n$) 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。

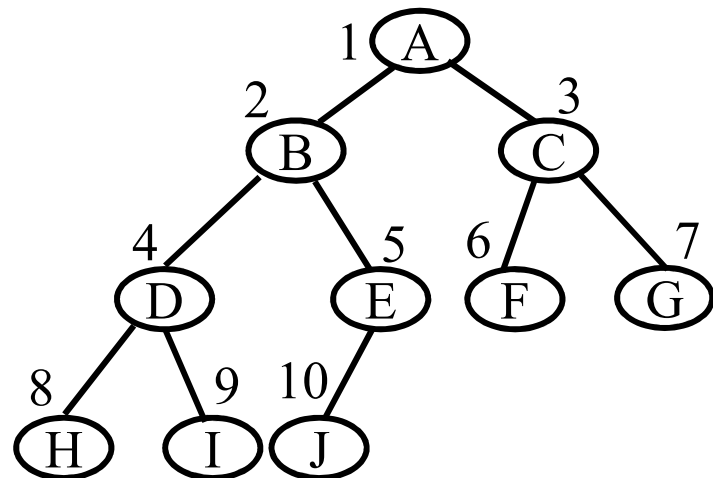
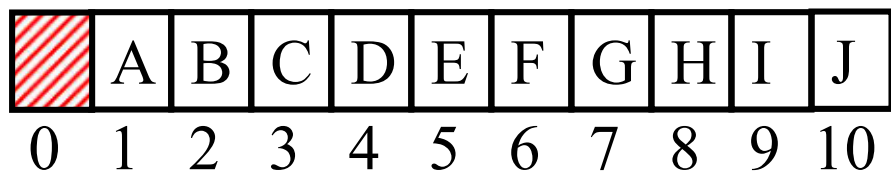




3.2 二叉树(Cont.)

- 性质5：完全二叉树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 $i \leq n \ \&\& \ i \neq 1$, 则有左兄弟, 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，

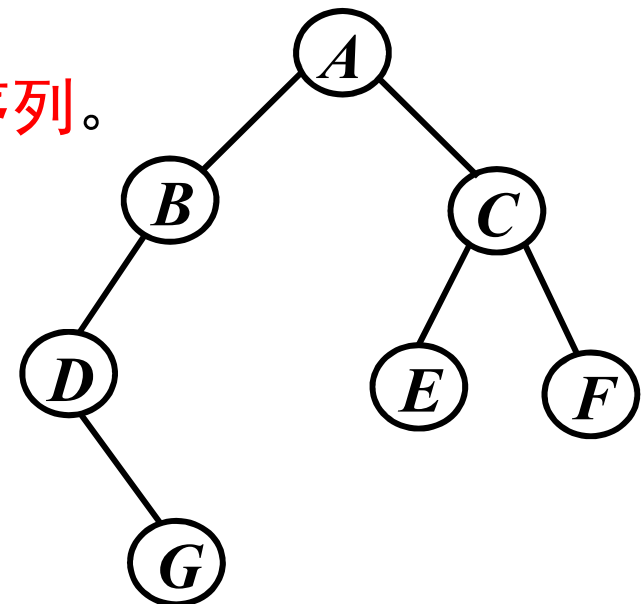
- ①访问根结点

- 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，

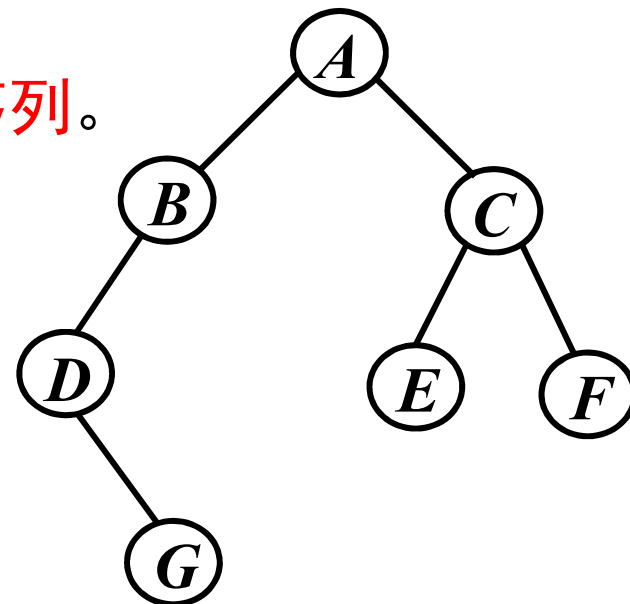
- ① 中序遍历根结点的左子树

- 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.)



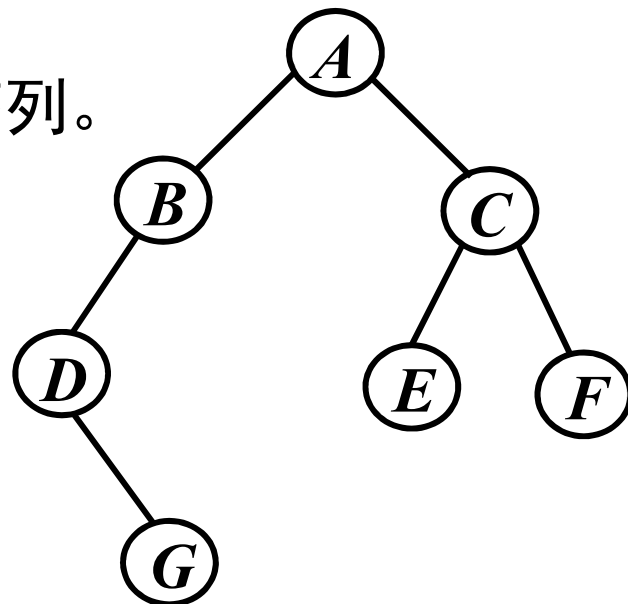
二叉树遍历的定义

- 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.)

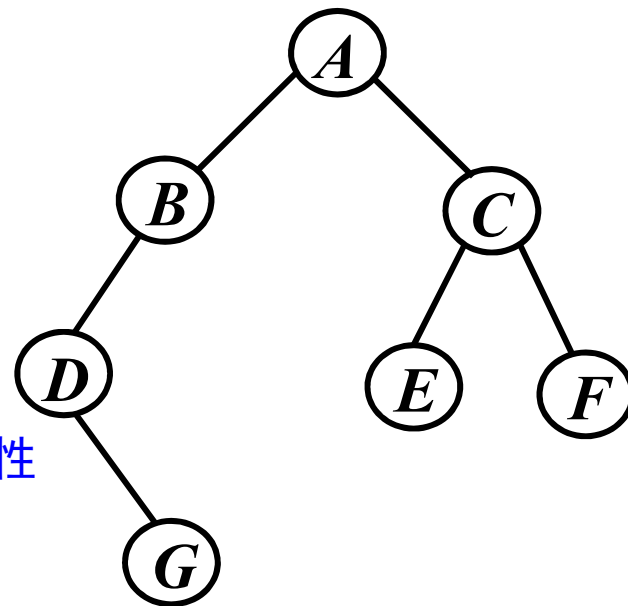
二叉树遍历的定义

- 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： $A B C D E F G$

- 三种遍历之间的关系

- 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 的根结点数据域的值。



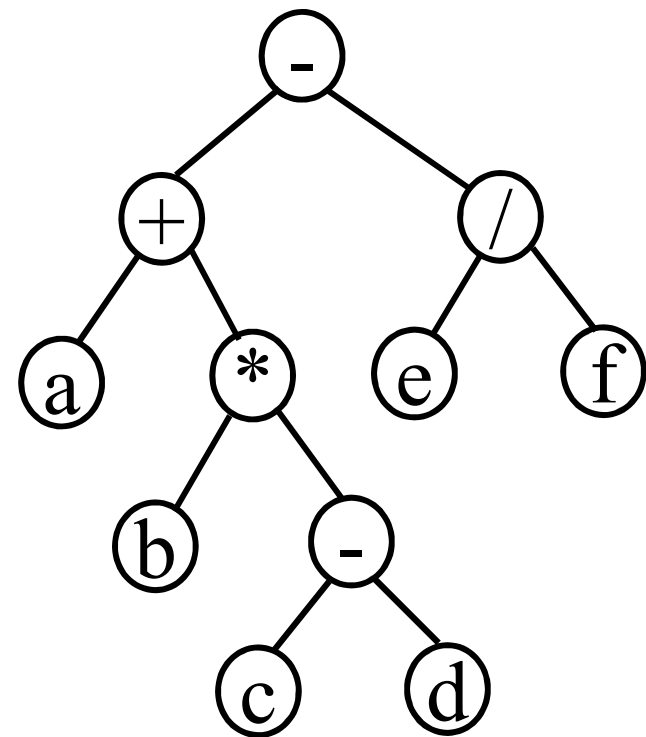
3.2 二叉树(Cont.)

利用二叉树的基本操作，编写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： $- + a * b - c d / e f$



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

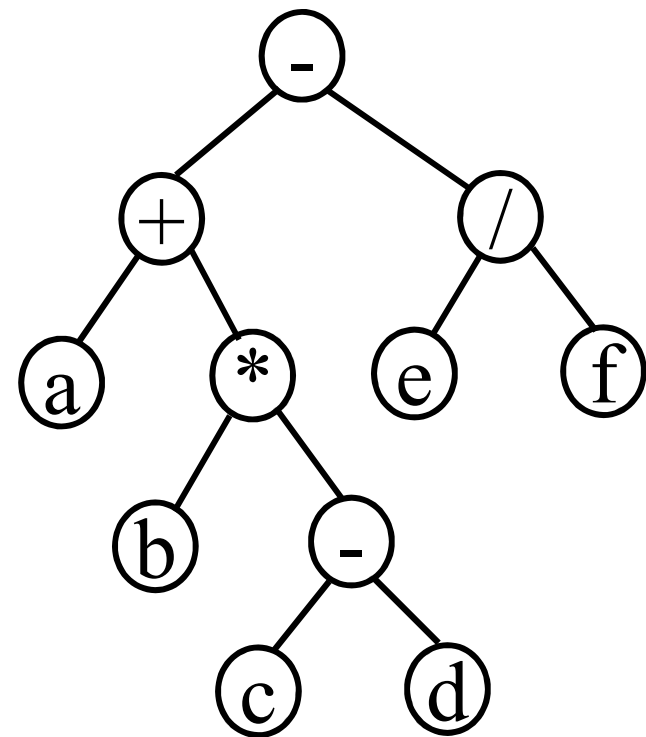


3.2 二叉树(Cont.)

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: $a + b * c - d - e / f$

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

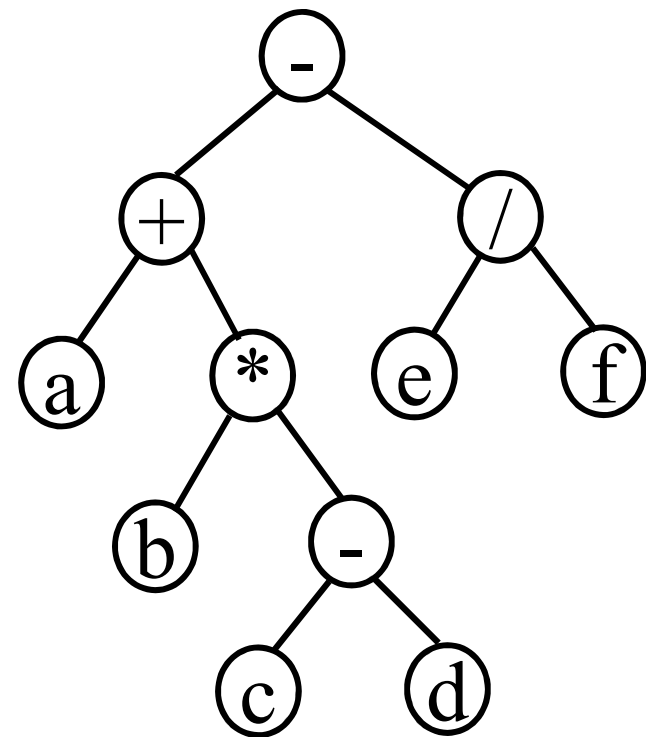


3.2 二叉树(Cont.)

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 / -

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

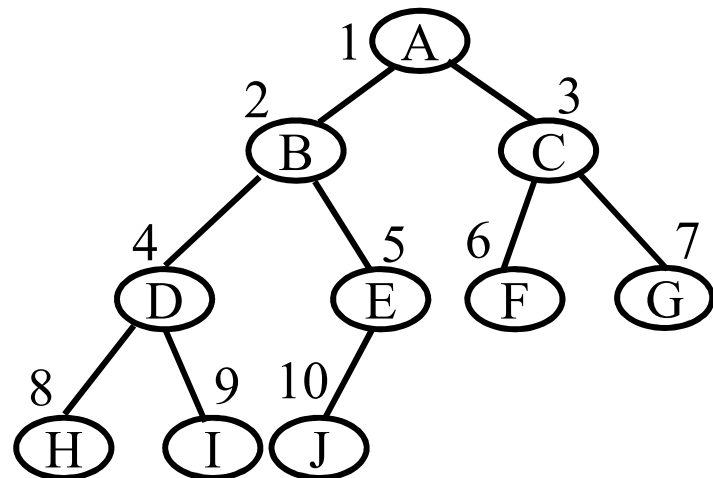
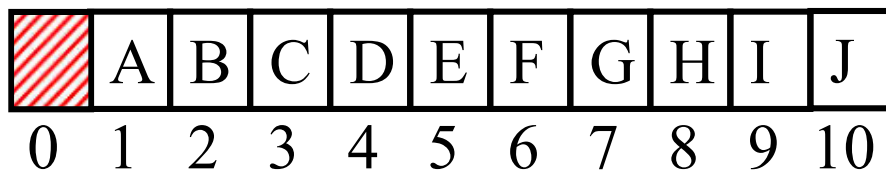


3.2 二叉树(Cont.)

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：

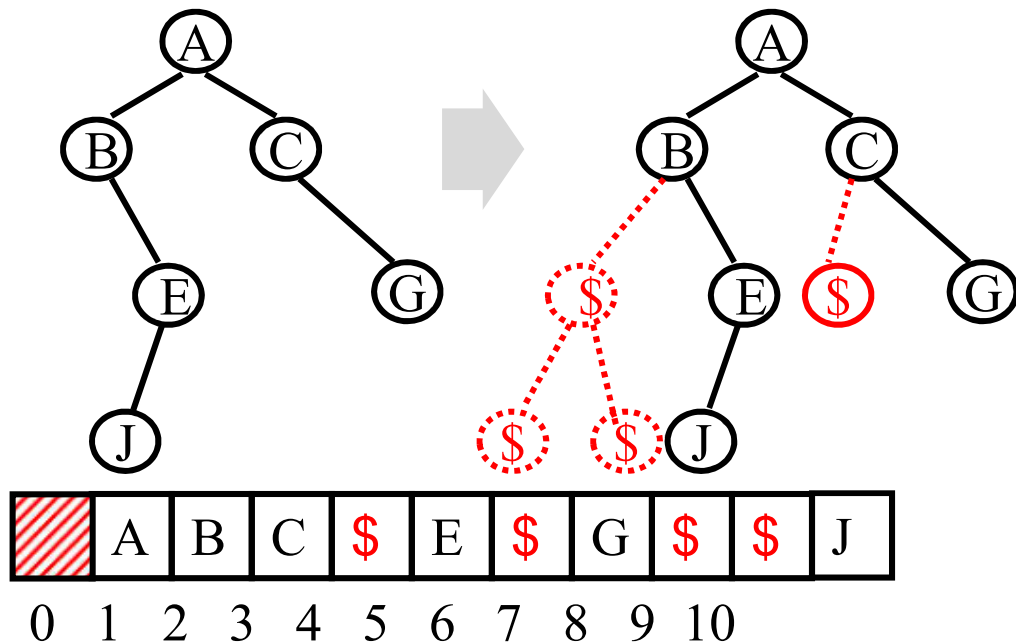




3.2 二叉树(Cont.)

一般二叉树的顺序存储结构

- 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\$。

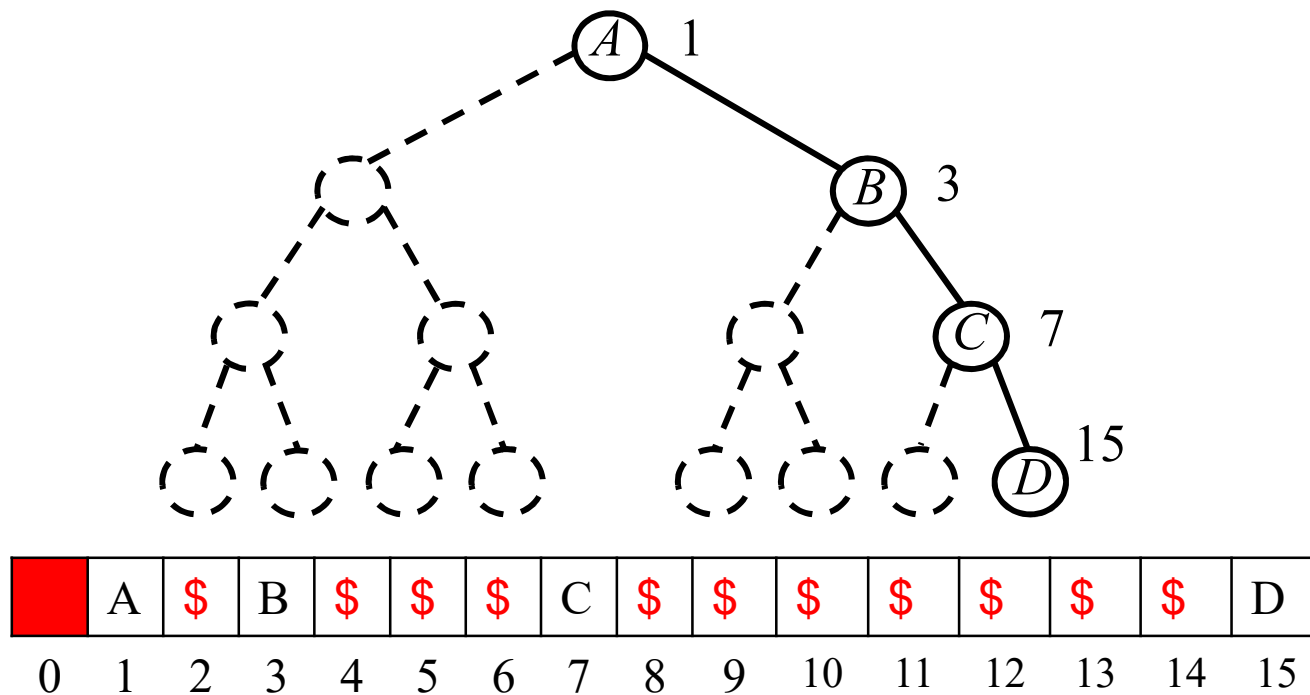




3.2 二叉树(Cont.)

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。





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。
- 用一个指向根结点的指针标识这个二叉树。

- 结点结构：

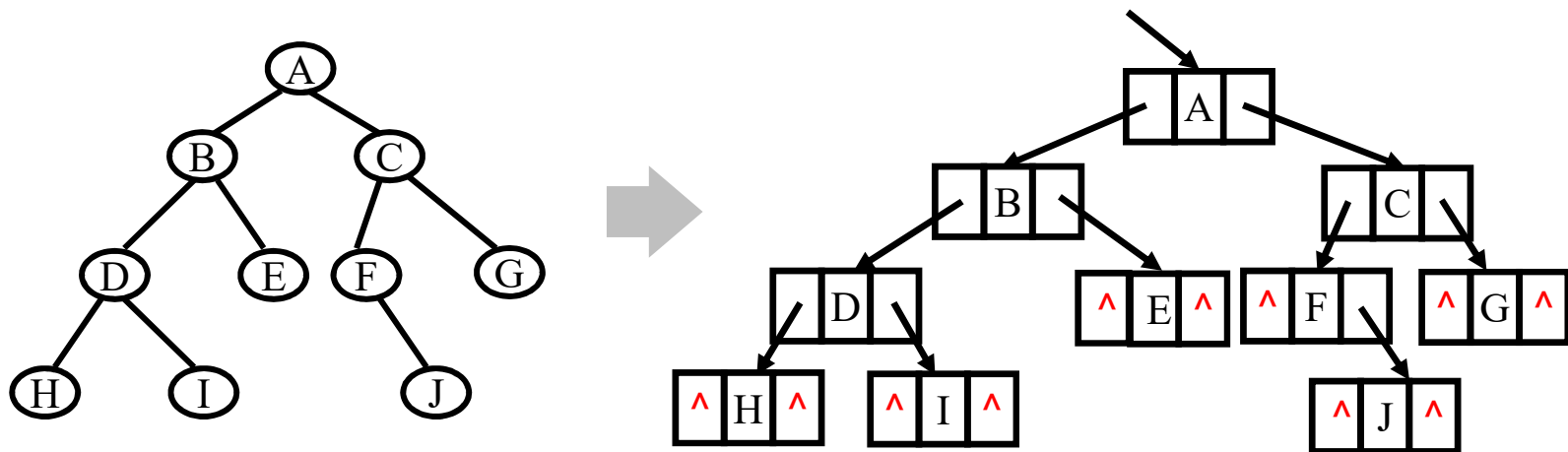
lchild	data	rchild
--------	------	--------

- 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：



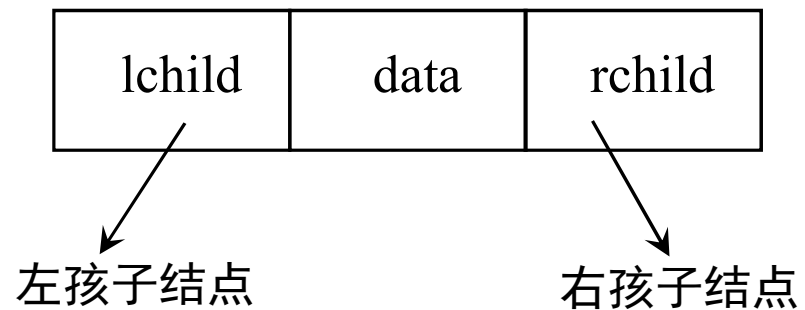
- 具有 n 个结点的二叉链表中，有多少个空指针？有多少指向孩子结点的指针？



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;
```





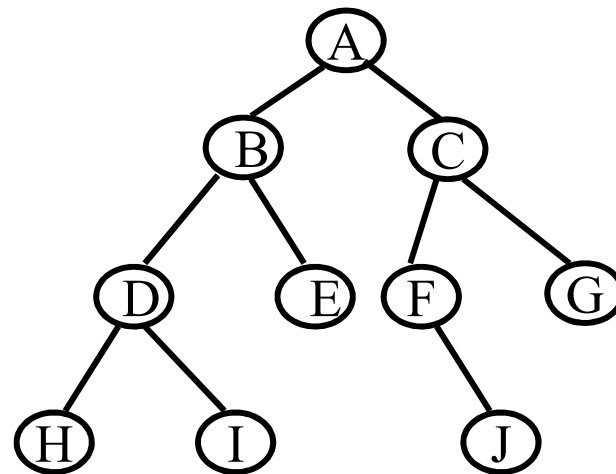
3.2 二叉树(Cont.)

二叉树左右链存储结构(二叉链表)的构建

- 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 ;  
}
```

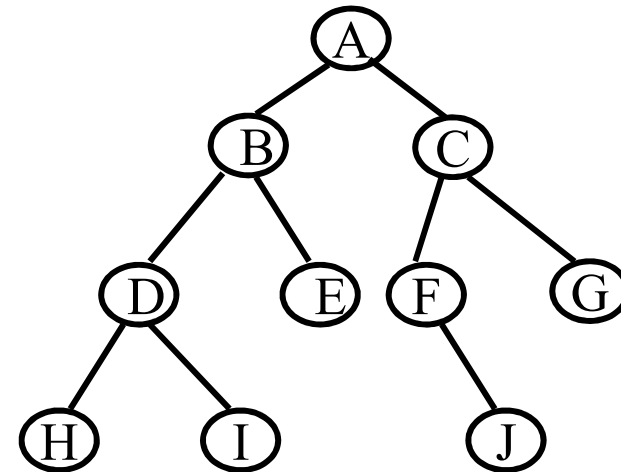




3.2 二叉树(Cont.)

- **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 );
        }
}
```

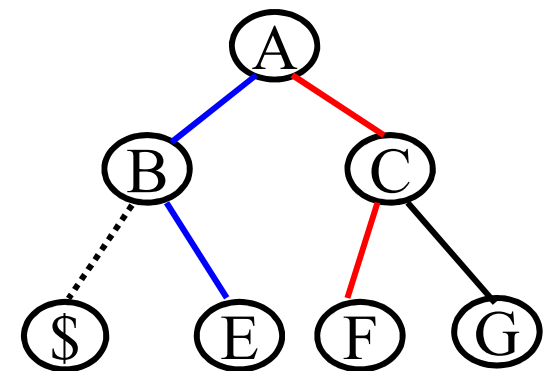




3.2 二叉树(Cont.)

- **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 → 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 是父结点j 的左儿子
           else s[ j ] → rchild = p; // i 是父结点j 的右儿子
         } cin >> i >> ch ;
  }
}
```

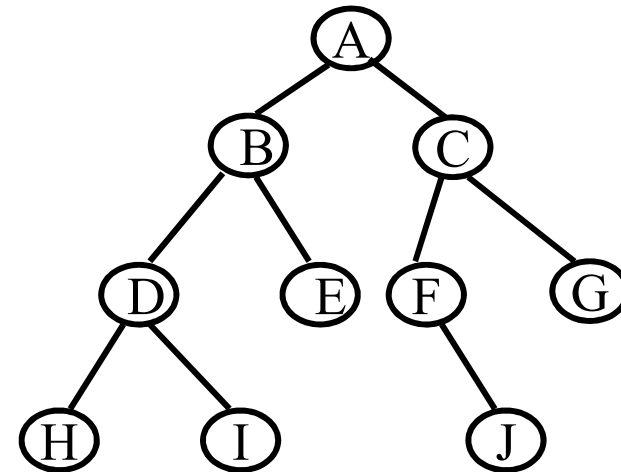




3.2 二叉树(Cont.)

- 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 ) ;
    }
}
```



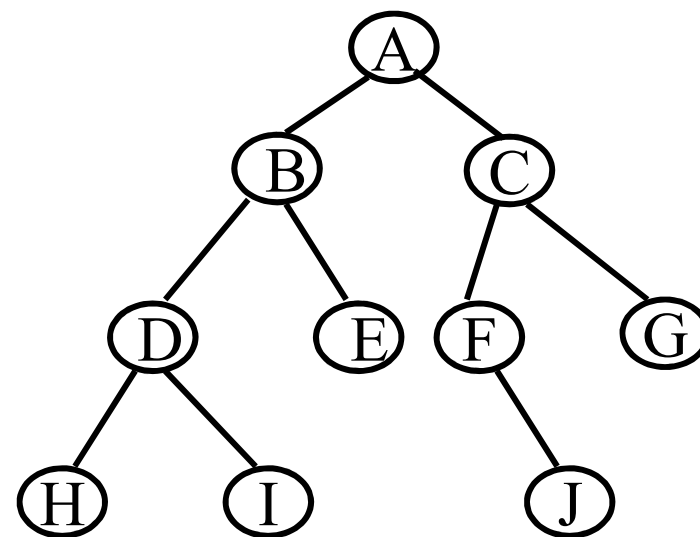


3.2 二叉树(Cont.)

- 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 );
    }
}
```



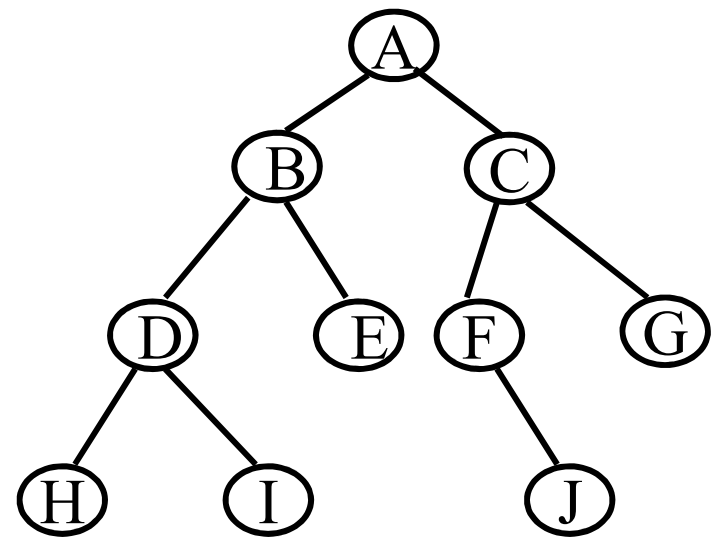
[[[[H] D [I]] B [E]] A [[F [J]] C [G]]]



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

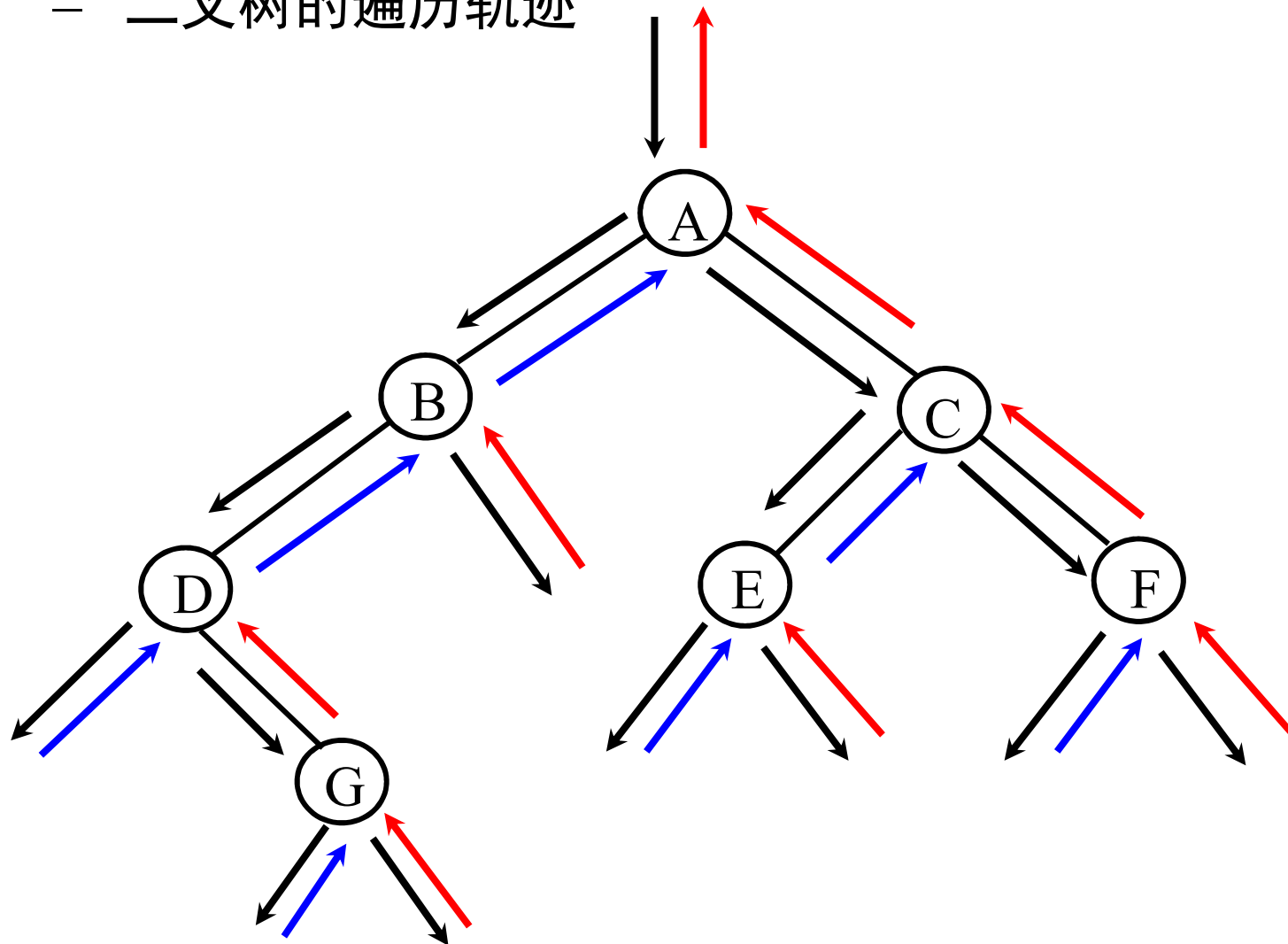
```
void PostOrder (BTREE BT)
{
    if ( BT != NULL)
    {
        PostOrder ( BT->lchild );
        PostOrder ( BT->rchild );
        cout<< BT->data ;
    }
}
```





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;

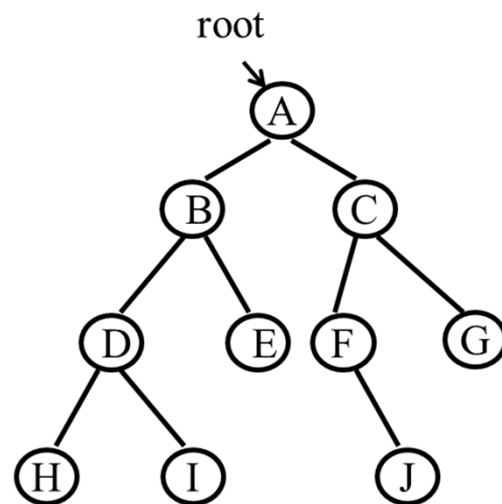
2.1.2 将指针root的值保存到栈中;

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;



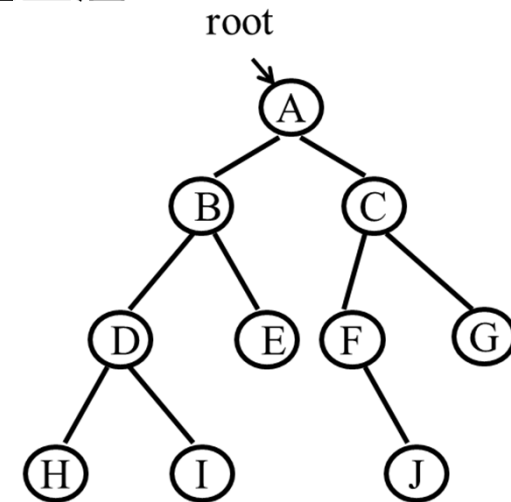


3.2 二叉树(Cont.)



- 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;

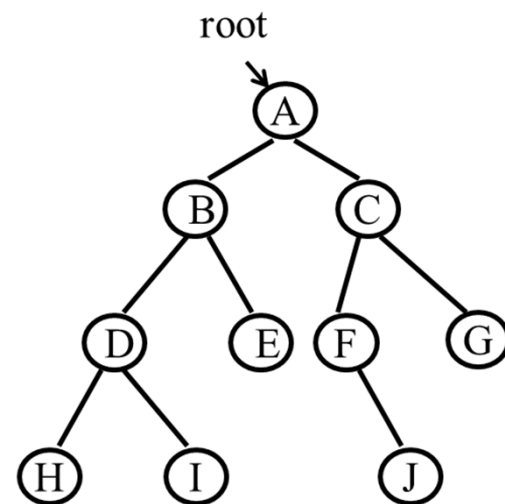
2.1.2 继续遍历root的左子树;

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;



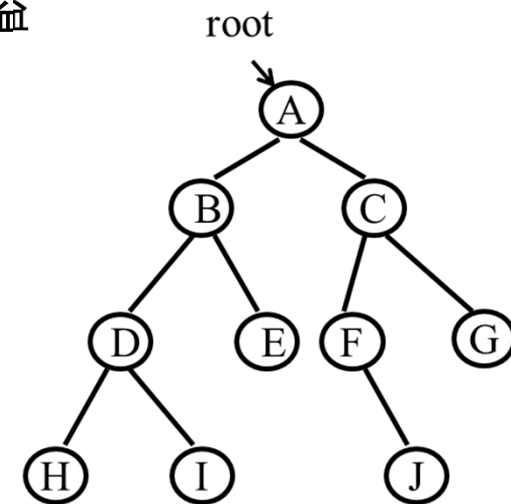


3.2 二叉树(Cont.)



- 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

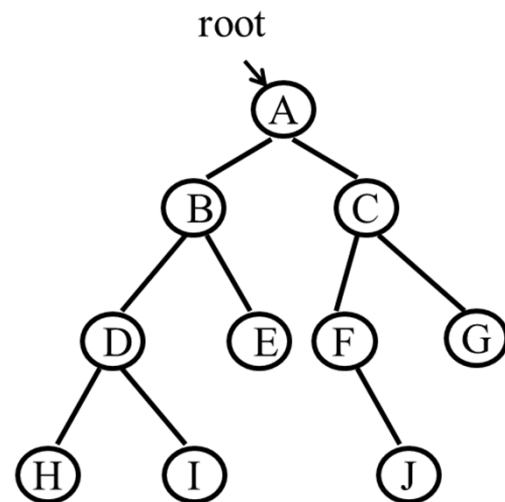
2.1 当root非空时循环

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.)

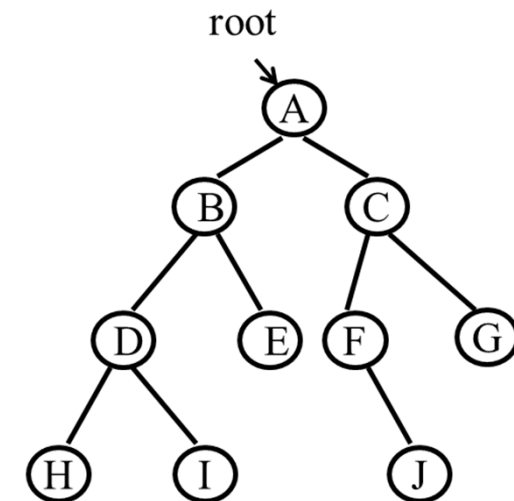


- 后序遍历非递归算法

```
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

ptr	flag
node*	bool





3.2 二叉树(Cont.)

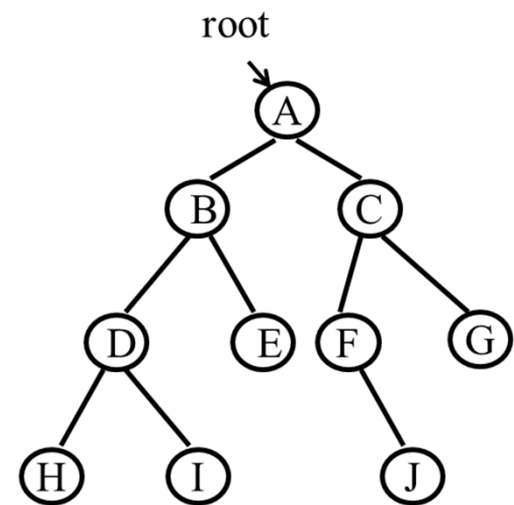
层序遍历算法

- 基本思想:

按层次顺序遍历二叉树，原则是先被访问结点的左、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**数据结构。

- 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];
```

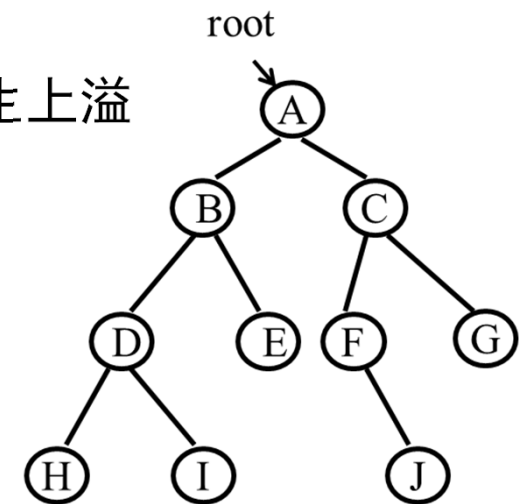
```
    cout<<q->data;
```

```
    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.)

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

```
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 struct 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);  
    }  
}
```

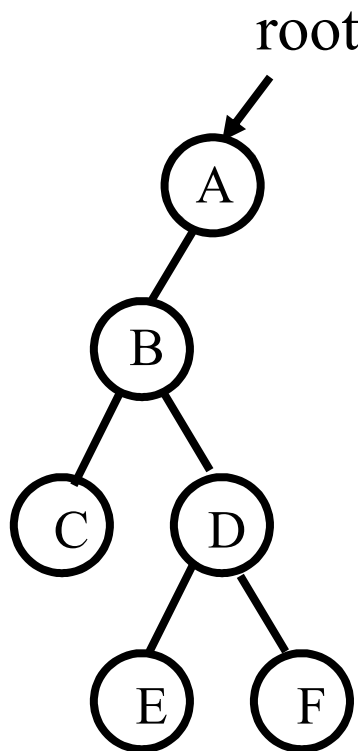
```
struct node {  
    struct node *lchild;  
    struct node *rchild;  
    datatype data;  
};  
typedef node * BTREE;
```



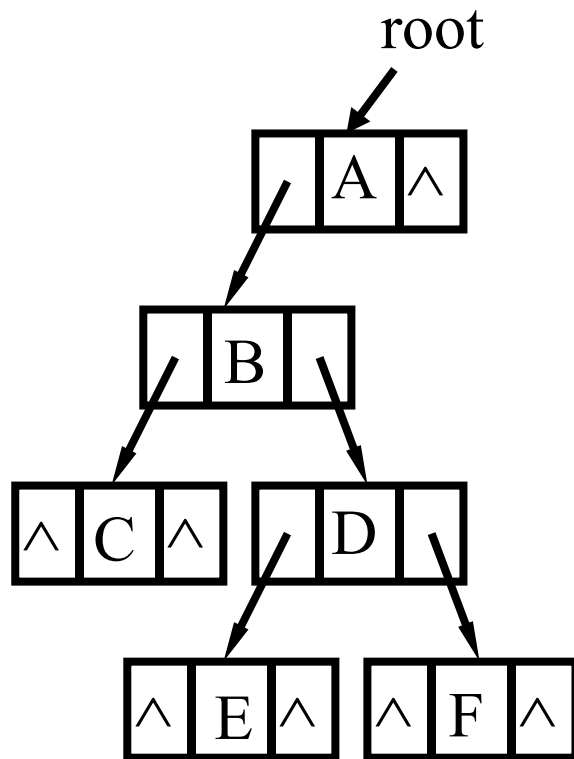

3.2 二叉树(Cont.)

- 二叉树的其他链式存储结构：动态三叉链表

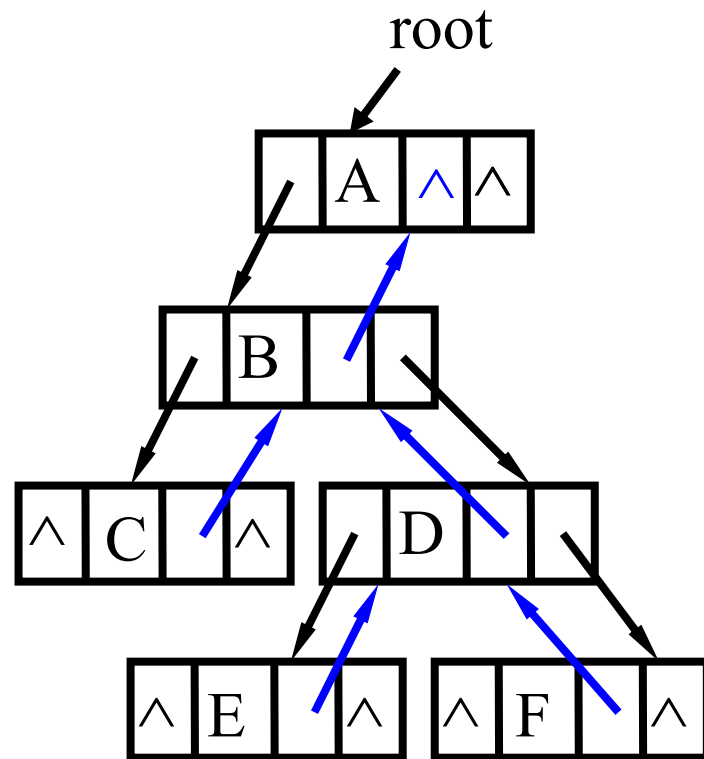
- 在二叉链表的基础上增加了一个指向双亲的指针域。



二叉树



动态二叉链表

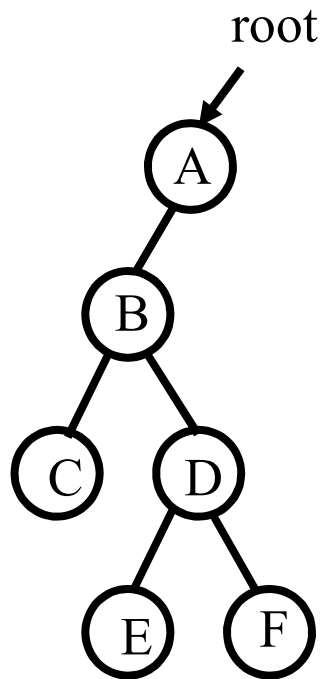


动态三叉链表



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

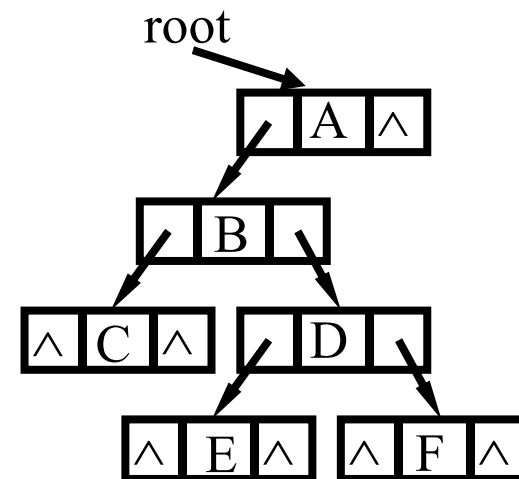
静态二叉链表和三叉链表



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

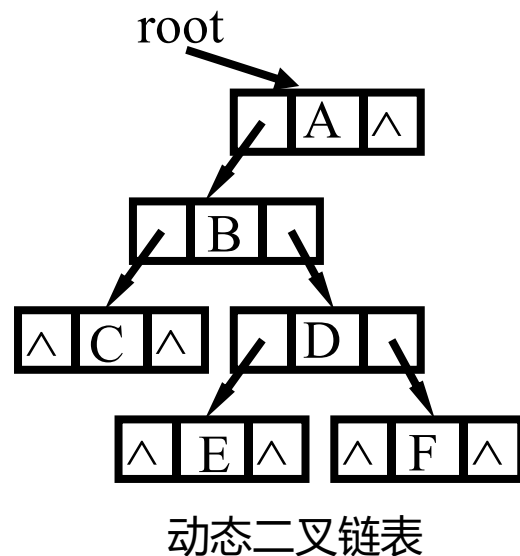


3.2 二叉树(Cont.)



二叉树的线索链表存储结构：线索二叉树

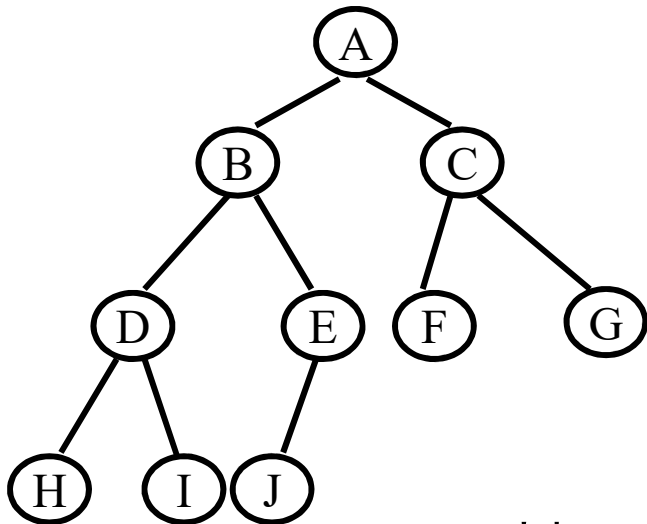
- 如何利用空指针域解决上述问题？
 - 若结点p有左孩子，则p->lchild指向其左孩子结点，否则令其指向其（先序、中序、后序、层序）前驱；
 - 若结点p有右孩子，则p->rchild指向其右孩子结点，否则令其指向其（先序、中序、后序、层序）后继；
- 如何区分指针是指向其左/右孩子的指针还是指向某种遍历的前驱/后继？
 - 在每个结点中增加两个标志位，以区分该结点的两个链域是指向其左/右孩子，还是指向某种遍历的前驱/后继。



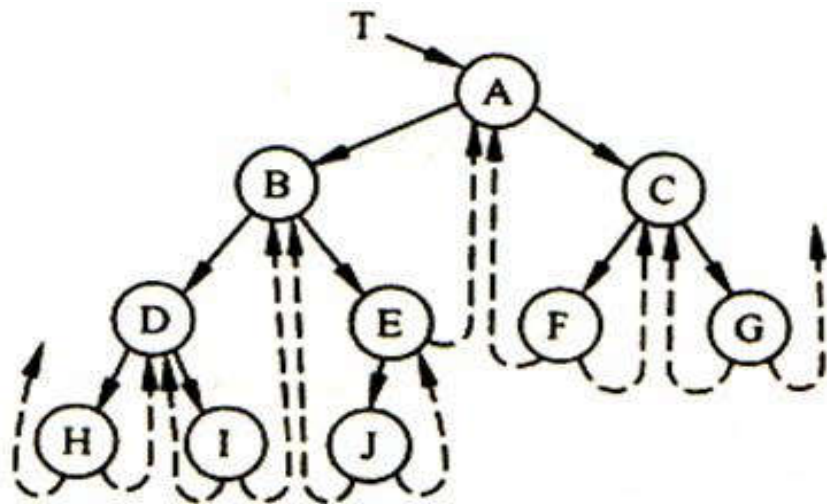


3.2 二叉树(Cont.)

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	rchild	rtag
--------	------	------	--------	------

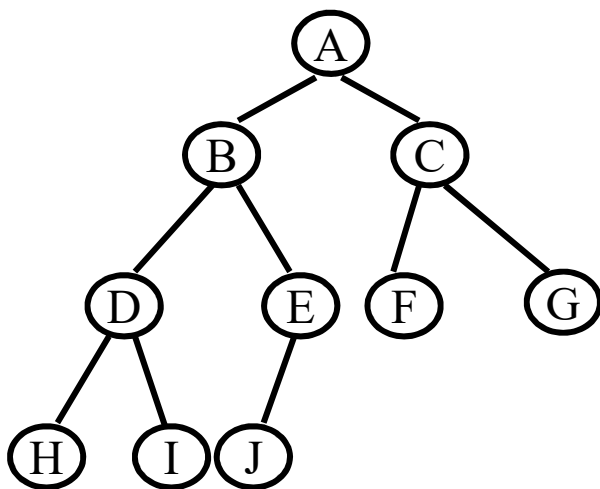
$p \rightarrow ltag = \begin{cases} \text{TRUE} & p \rightarrow lchild \text{ 指向左孩子} \\ \text{FALSE} & p \rightarrow lchild \text{ 指向 (中序) 前驱} \end{cases}$

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

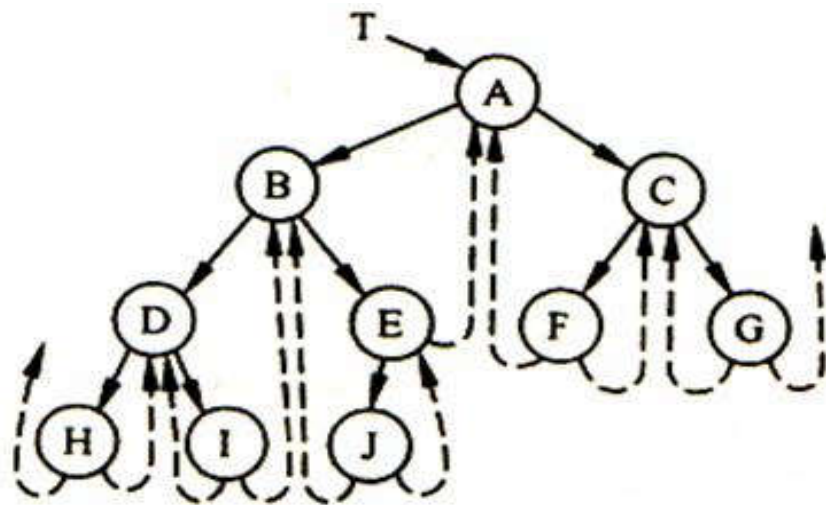


3.2 二叉树(Cont.)

二叉树的线索链表存储结构：线索二叉树



二叉树



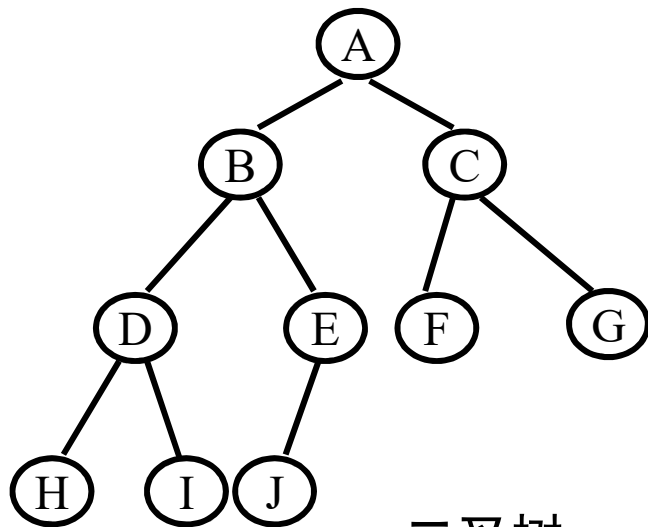
中序线索二叉树

- **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**。

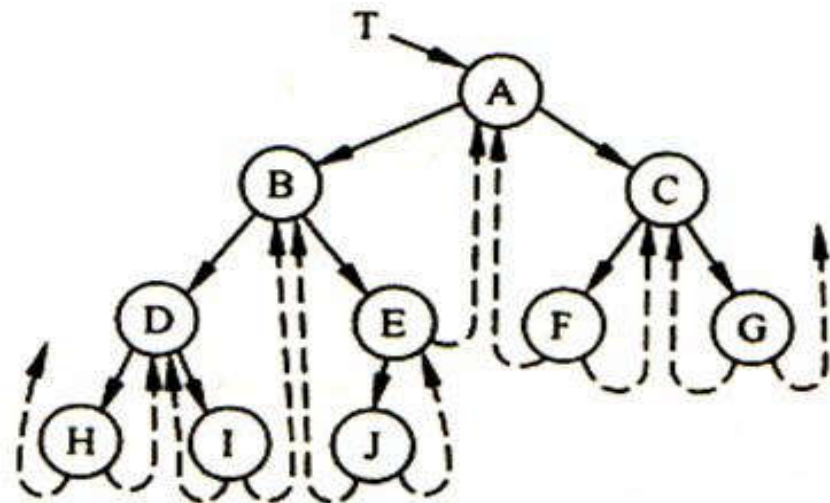


3.2 二叉树(Cont.)

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.)

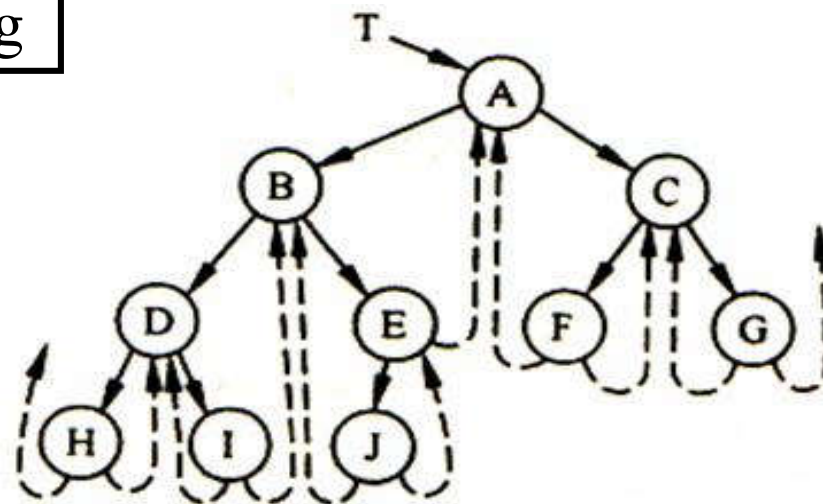


• 线索链表的存储结构定义

结点结构

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

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

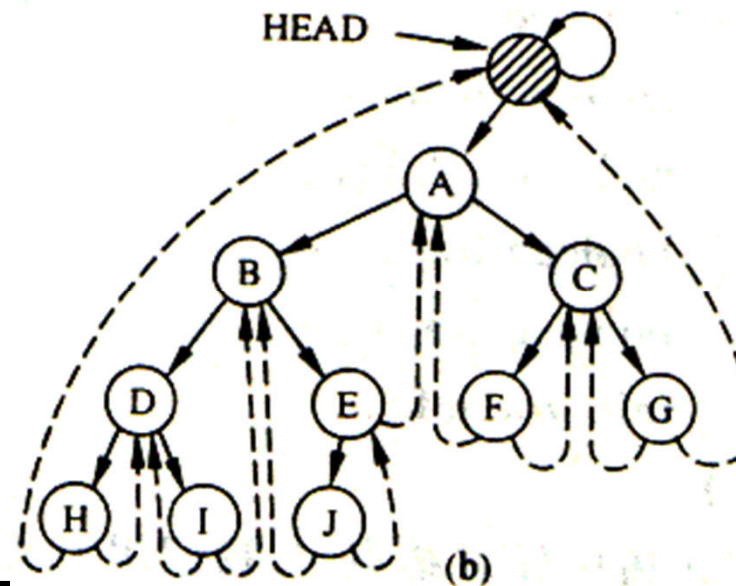
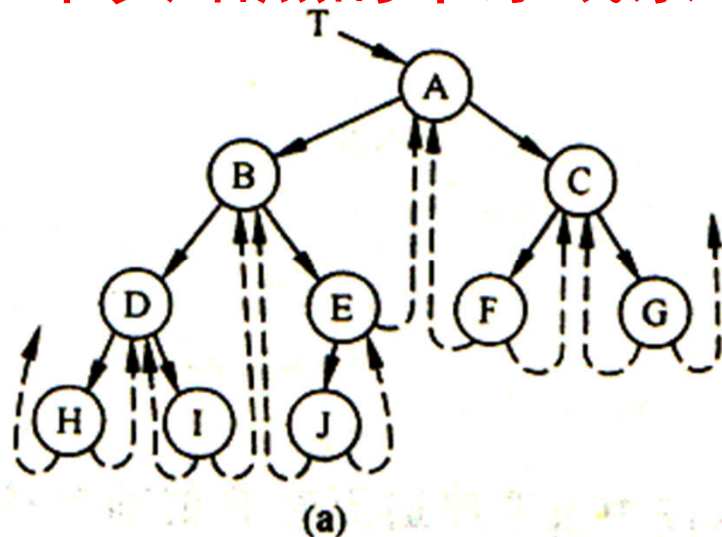




3.2 二叉树(Cont.)



• 带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;



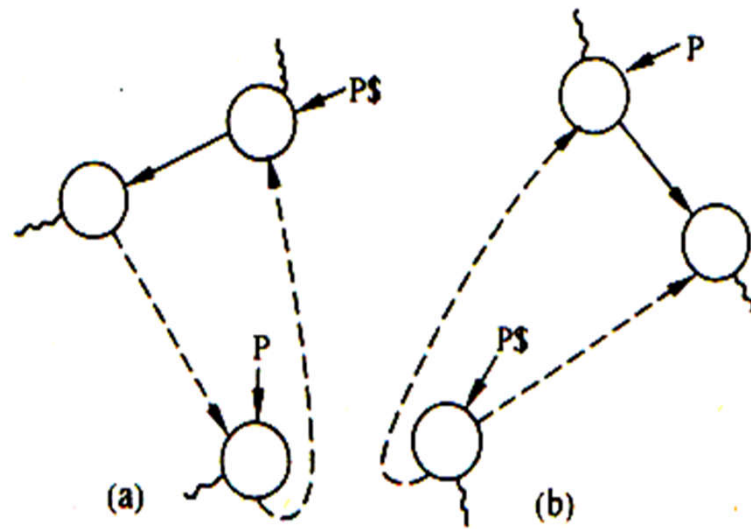
3.2 二叉树(Cont.)



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->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:

// 利用InNext()

```
void InOrderTh (THTREE head)
```

```
{ THTREE tmp ;
```

```
tmp = head ;
```

```
do {
```

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

```
if ( tmp != head )
```

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

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

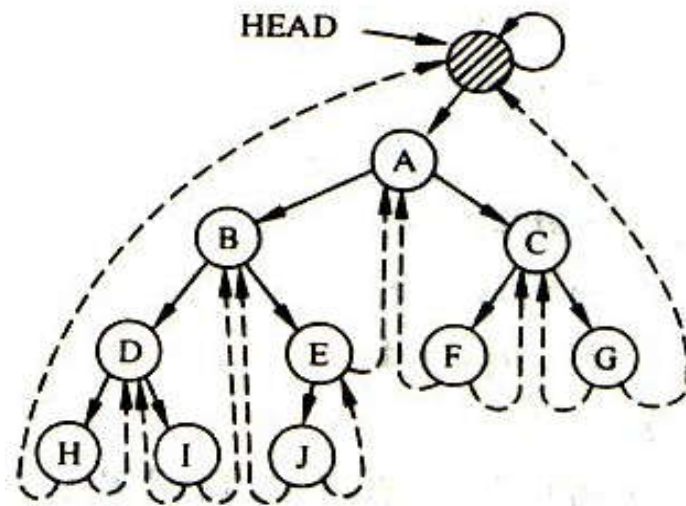
```
}
```

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, 可以更改为其他结点操作
```

```
       while(p->rtag == FALSE && p->rchild != head) { //结点H
```

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

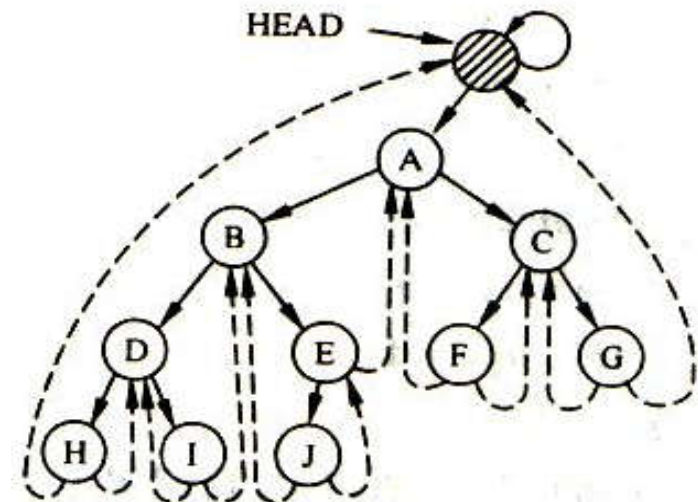
```
           printf("%c ", p->data);
```

```
       }
```

```
       p = p->rchild; //p进入其右子树
```

```
   }
```

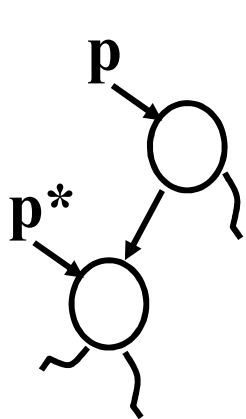
```
}
```



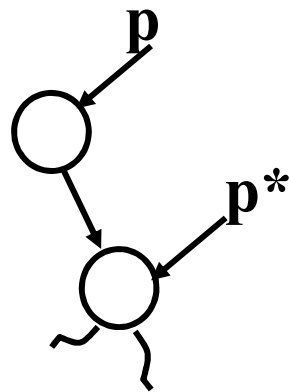


3.2 二叉树(Cont.)

- **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**: LDR, DLR
 - (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**第一个有右儿子**结点的**右儿子或表头结点**即为 p^* .

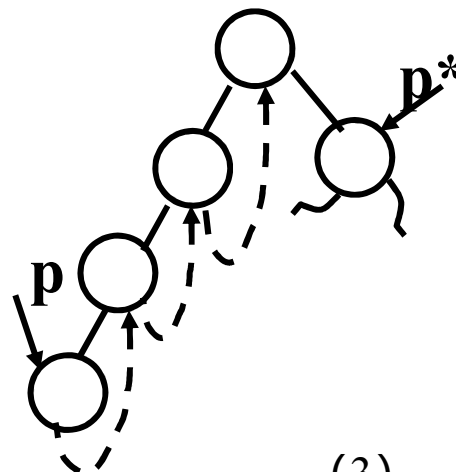


(1)

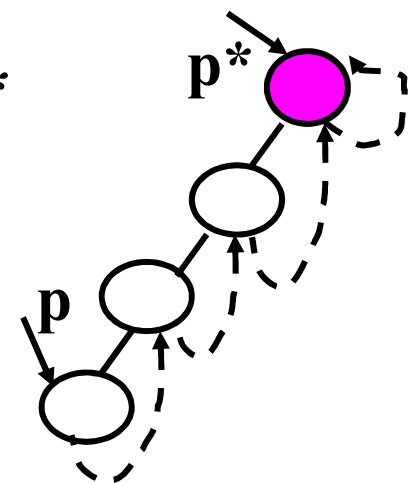


(2)

DLR



(3)

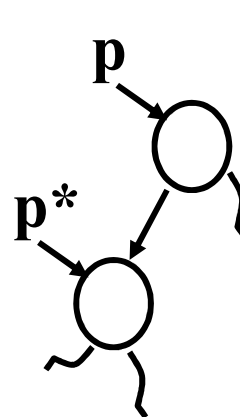




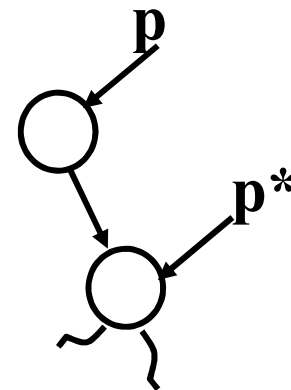
3.2 二叉树(Cont.)

- **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：** LDR, DLR

```
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;
}
```

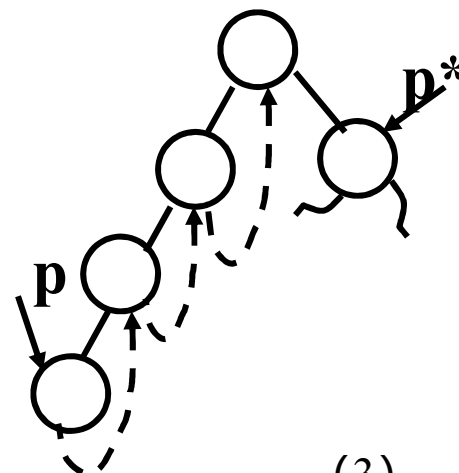


(1)



(2)

DLR



(3)



3.2 二叉树(Cont.)



线索二叉树的若干算法

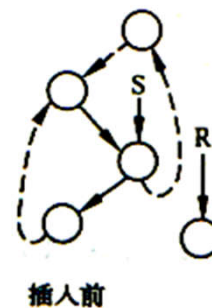
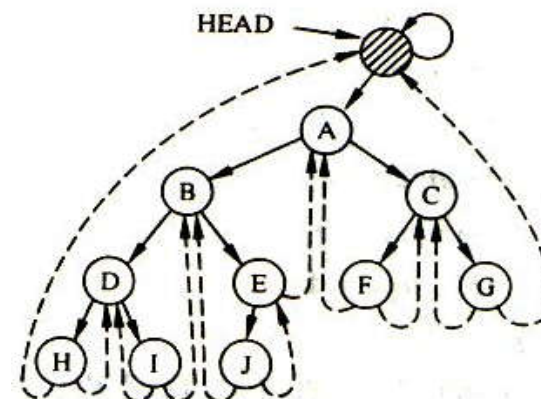
算法4：中序线索二叉树的插入

- 分析：如将结点 R 插入作为结点 S 的右孩子结点。

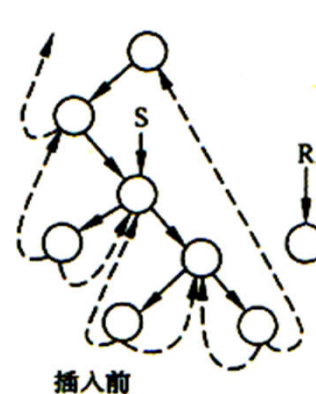
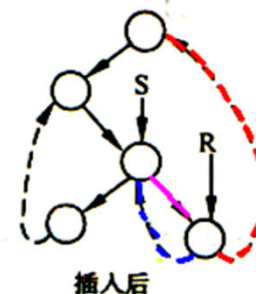
(1)若S的右子树为空，直接插入；

(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

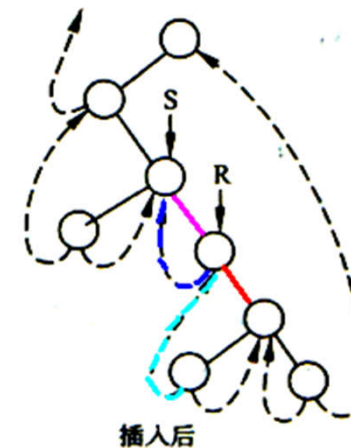
R完全继承原来S的右子树



(a)



(b)





3.2 二叉树(Cont.)



```
void RInsert (THTREE S ,THTREE R) //LDR
```

```
{ THTREE W ;
```

```
  R->rchild = S->rchild; //a&b(1)
```

```
  R->rtag = S->rtag ;
```

```
  R->lchild = S ;           // b(2)
```

```
  R->ltag = FALSE ;        // thread
```

```
  S->rchild = R ;           //b(3)
```

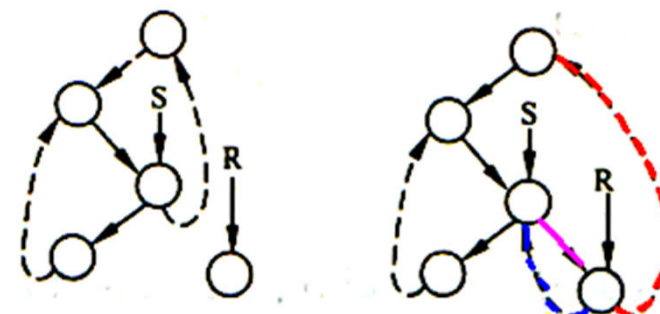
```
  S->rtag = TRUE ;
```

```
  if (R->rtag==TRUE) { // b(4)
```

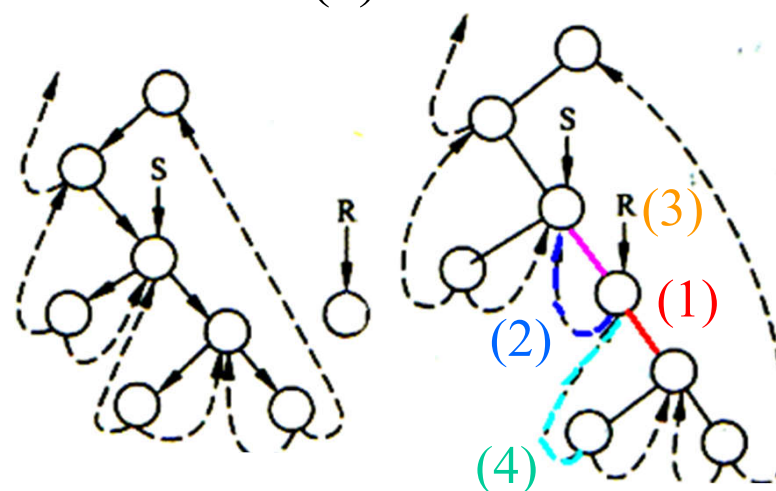
```
    w = InNext( R ) ;
```

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

```
}
```



(a)

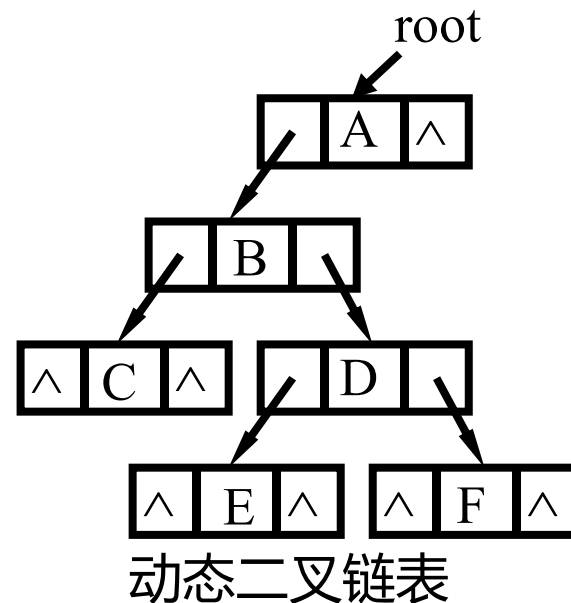
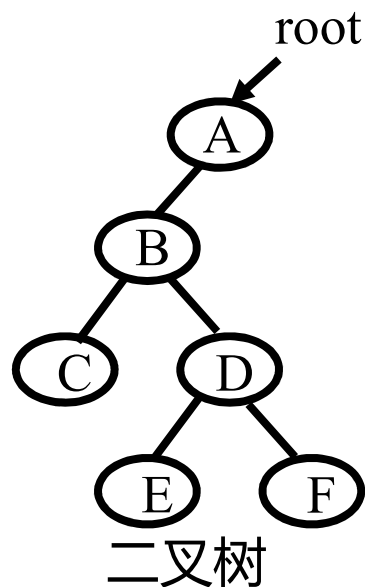


(b)



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。**





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.)

算法5：递归中序线索化二叉树

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; //当前根节点：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

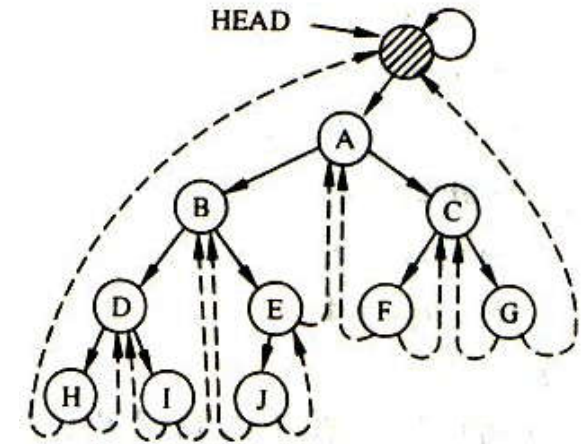
 p->lchild=pre; //令 *p的左线索指向中序前驱

 }

 pre = p; // 令pre 是下一个访问的中序前驱

 InOrderTh(p->rchild); //递归地线索化右子树

}}





3.2 二叉树(Cont.)

二叉树的复制

- 两株二叉树具有**相同结构**满足：
 - (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**。“**形状**”相同。
- 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算法：

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



3.2 二叉树(Cont.)

判断两株二叉树是否等价的算法

```
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.)

二叉树的复制

```
BTREE Copy( BTREE oldtree )
{
    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 */
```

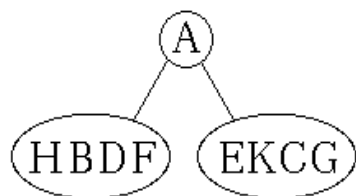


3.2 二叉树(Cont.)

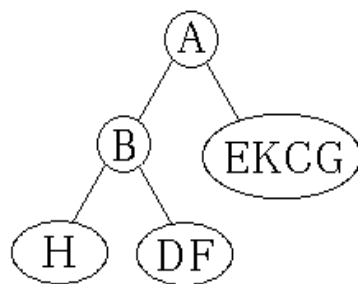


由序列确定二叉树

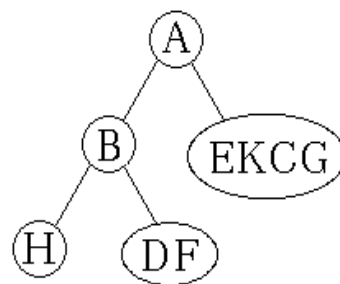
- 由二叉树的**先序序列**和**中序序列**可唯一地确定一棵二叉树
- 例：先序序列 {ABHFDECKG} 和中序序列 {HBDFAEKCG}, **构造二叉树**过程如下：



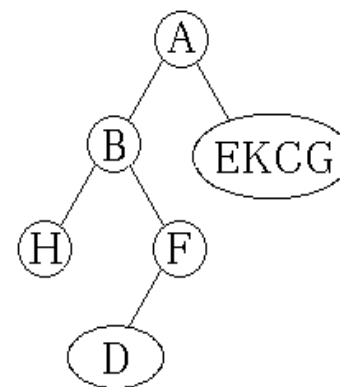
(a) 取A



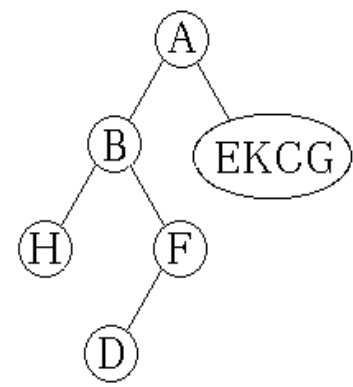
(b) 取B



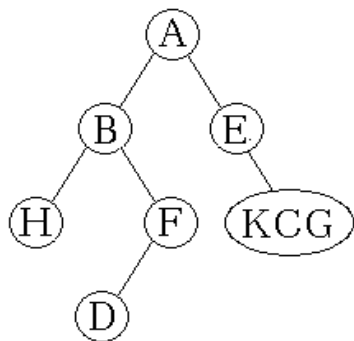
(c) 取H



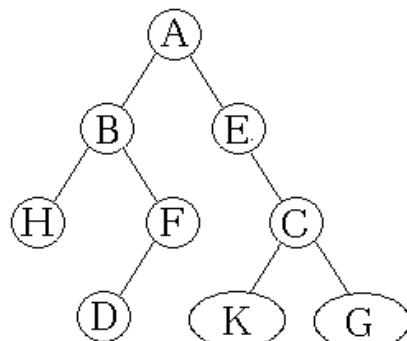
(d) 取F



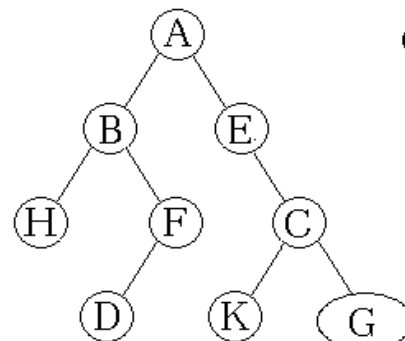
(e) 取D



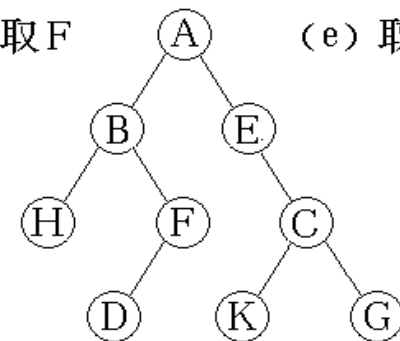
(f) 取E



(g) 取C



(h) 取K



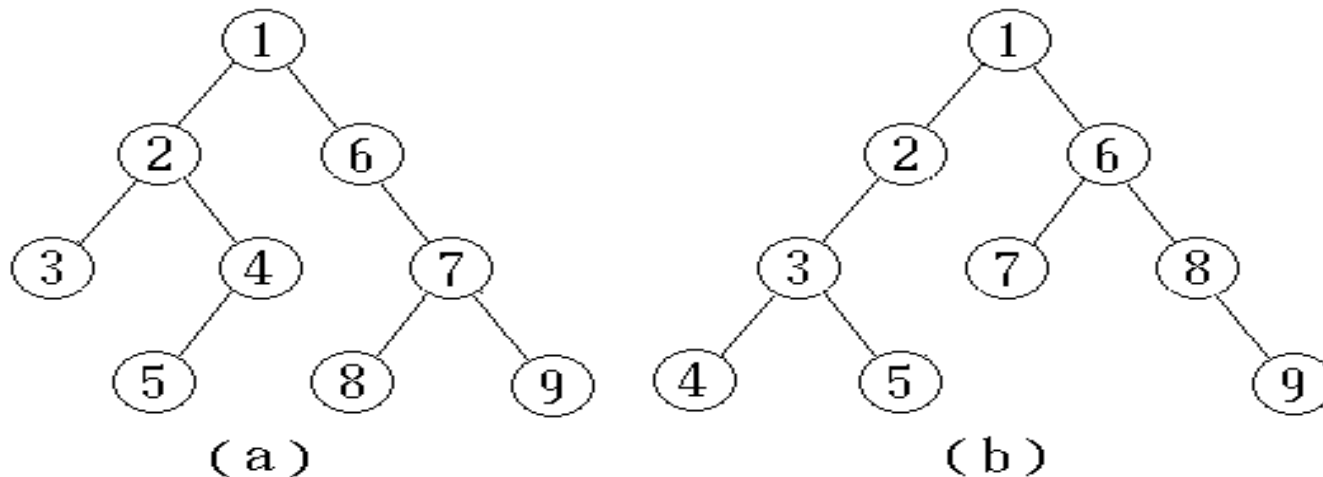
(i) 取G



3.2 二叉树(Cont.)

由序列确定二叉树

- 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。



- 思考**：设有 n 个结点（数据值），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

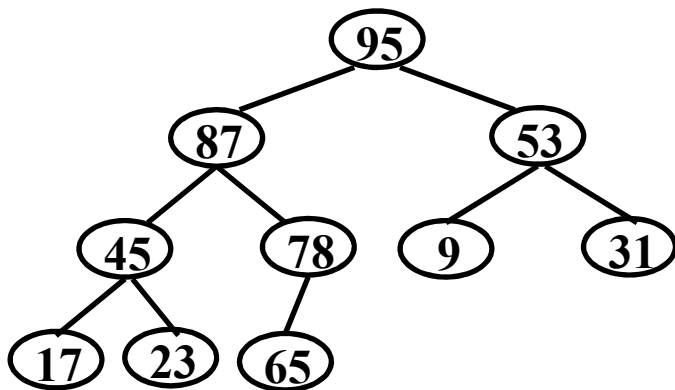


3.3 堆 (Heap)

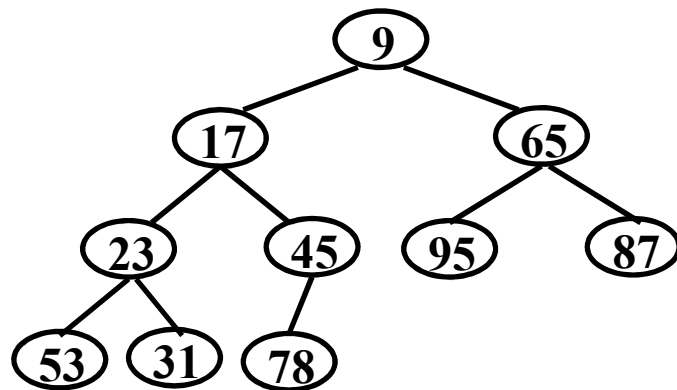
一、ADT 堆

• 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。
- HeapEmpty(heap, n): 判断堆是否为空。若堆中元素数量n大于0，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。
- 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)

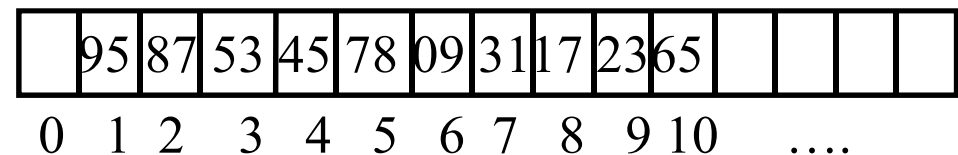
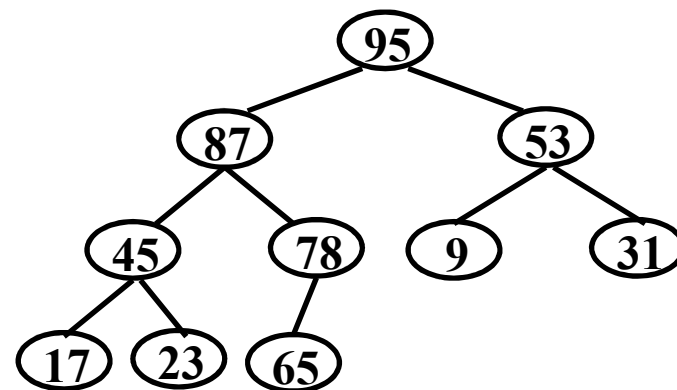
二、ADT堆的实现：最大堆实现

- 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;
```





3.3 堆 (Heap)

二、ADT堆的实现—最大堆的实现

• 堆基本操作的实现

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：最大堆实现

• 堆基本操作的实现

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]; //键值下推
```

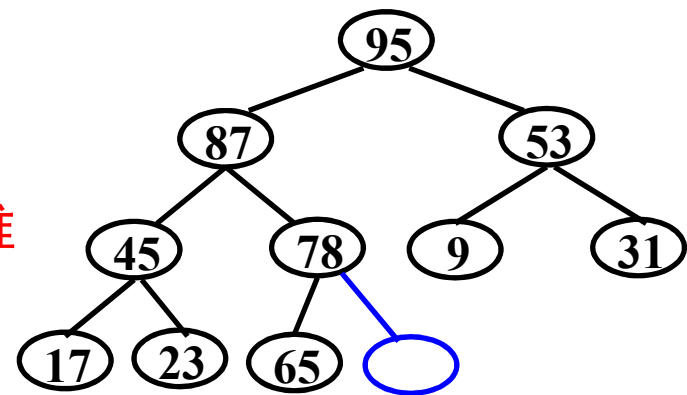
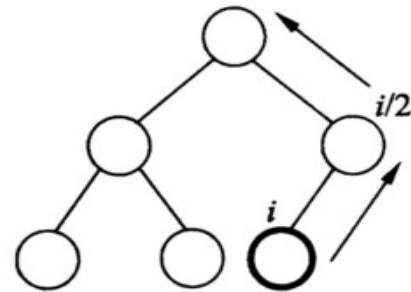
```
            i/=2;
```

```
        }
```

```
    }
```

```
    heap.data[i]= elem;
```

```
    }//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				

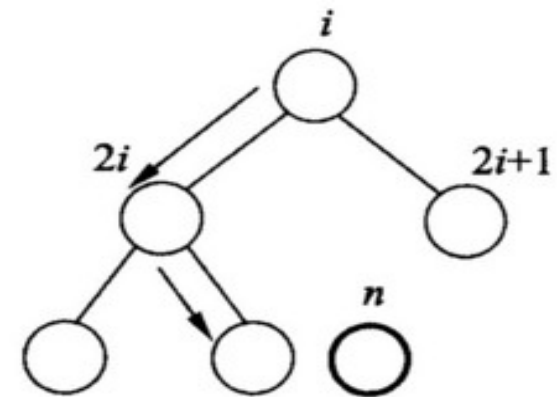
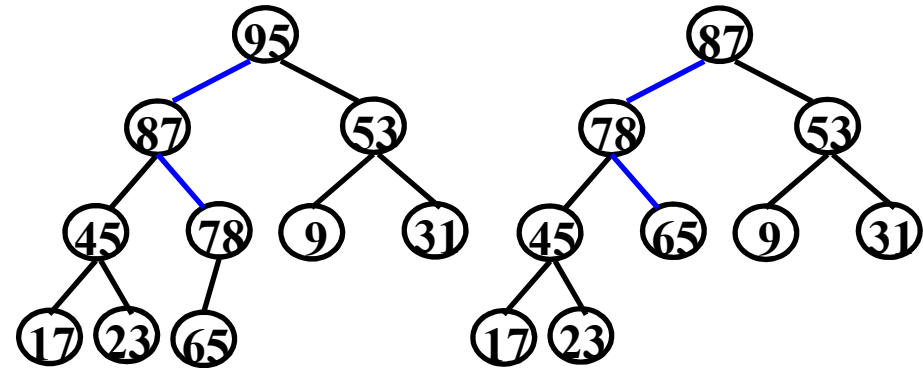


Todo 3.3 堆 (Heap)

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){ //下推
            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;
    }
} //时间复杂度O (logn)
```





3.3 堆 (Heap)

三、堆与优先级队列 (Priority Queue)

- 优先级队列应用广泛
 - 计算机操作系统任务调度、日常工作安排等领域。
- 在优先级队列中，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级队列。
- 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级队列。

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

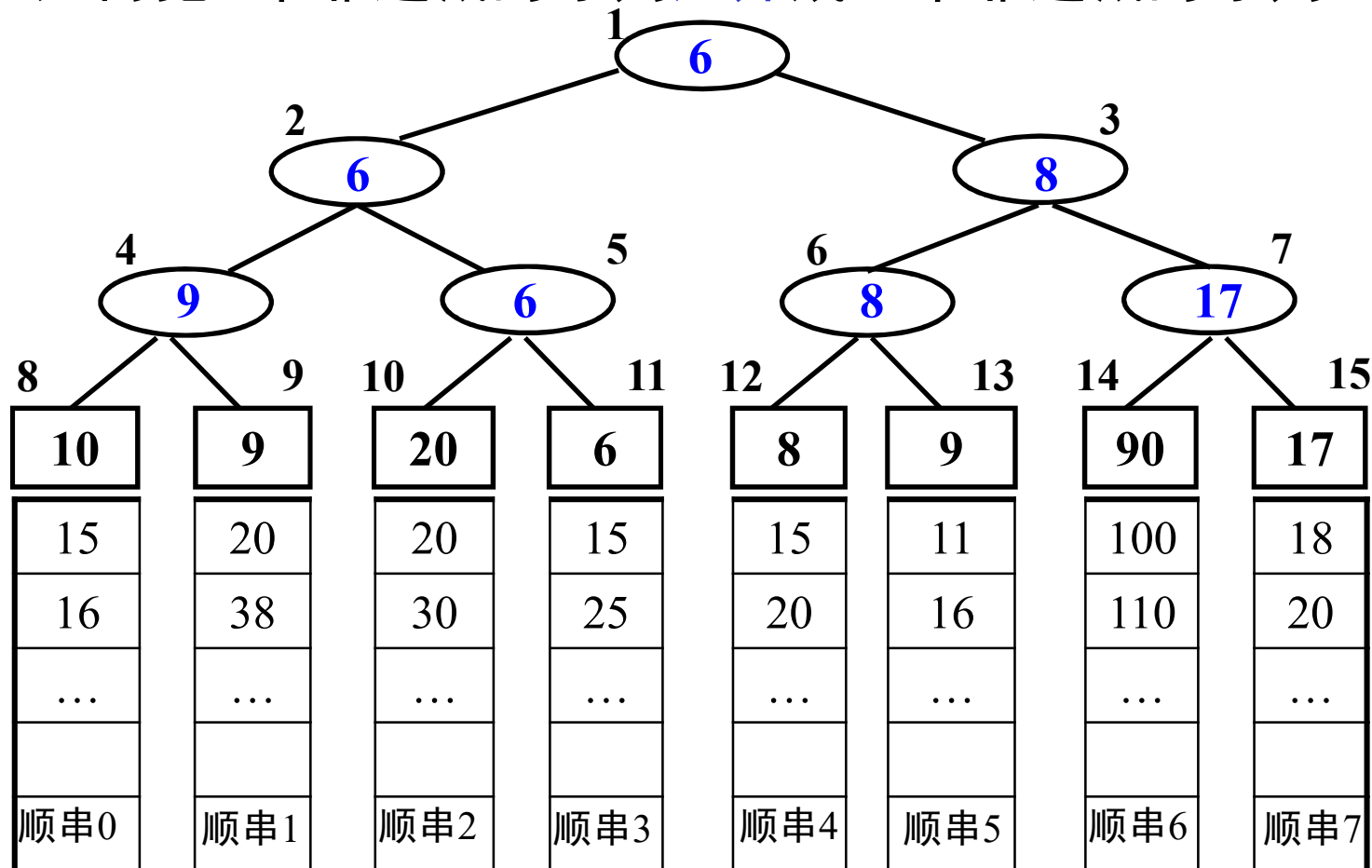
优先级队列的各种存储表示与操作性能比较



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？

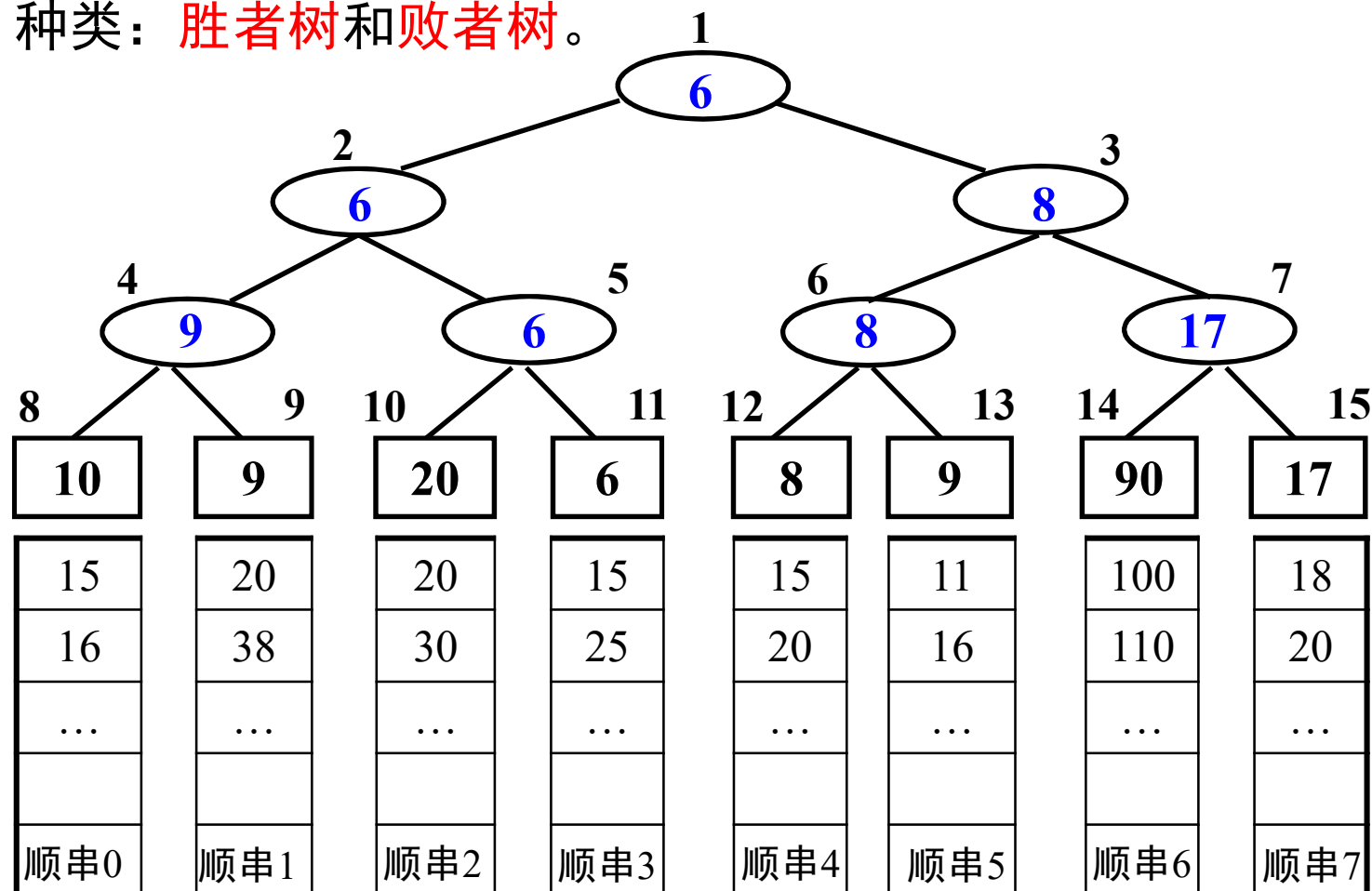




3.4 选择树 (Selection Tree)

二、选择树 (也称Tournament Tree)

- 选择树就是能够记载上一次比较**所得知识**的完全二叉树
- 种类：**胜者树**和**败者树**。

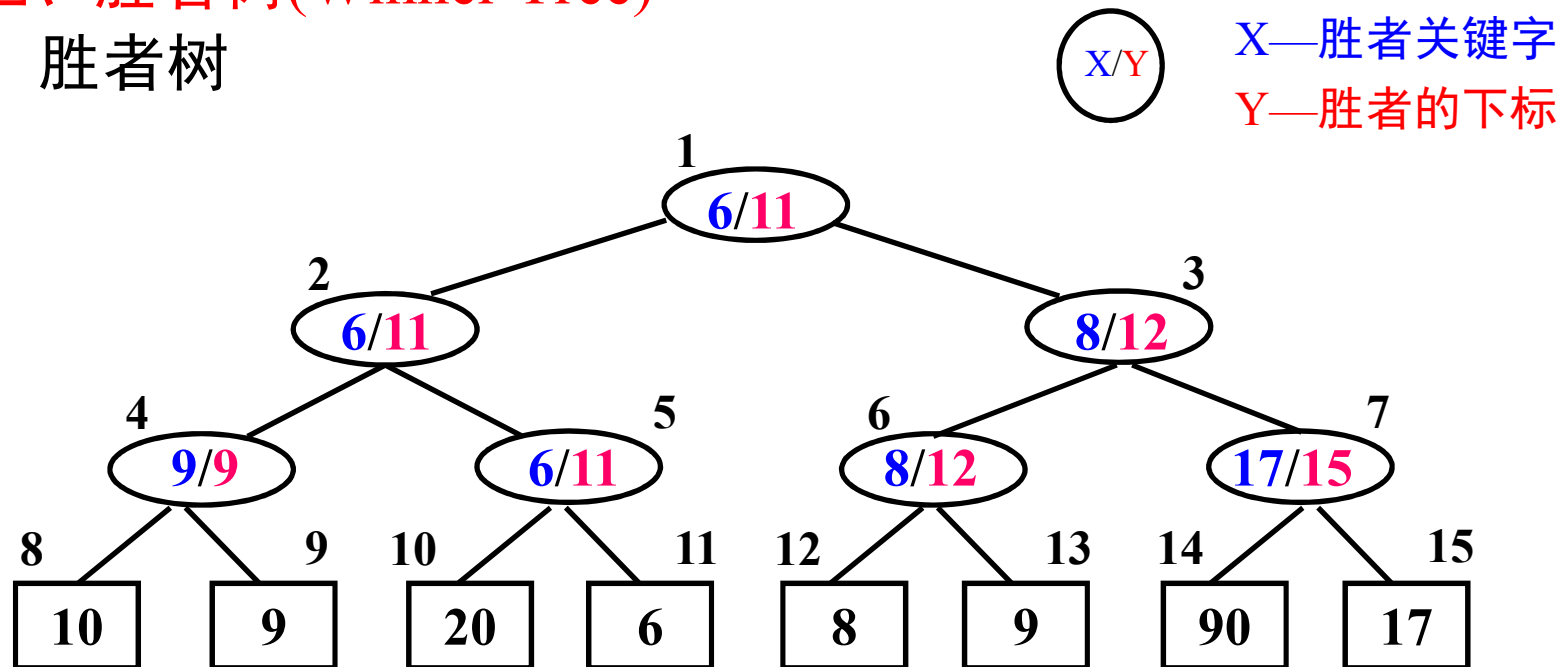




3.4 选择树 (Selection Tree)

三、胜者树(Winner Tree)

- 胜者树



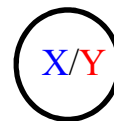
- 具有 n 个外结点和 $n-1$ 个内结点
- 外结点为比赛选手，内结点为一次比赛，每一层为一轮比赛
- 比赛在兄弟结点间进行，胜者保存到父结点中
- 根结点保存最终的胜者



3.4 选择树 (Selection Tree)

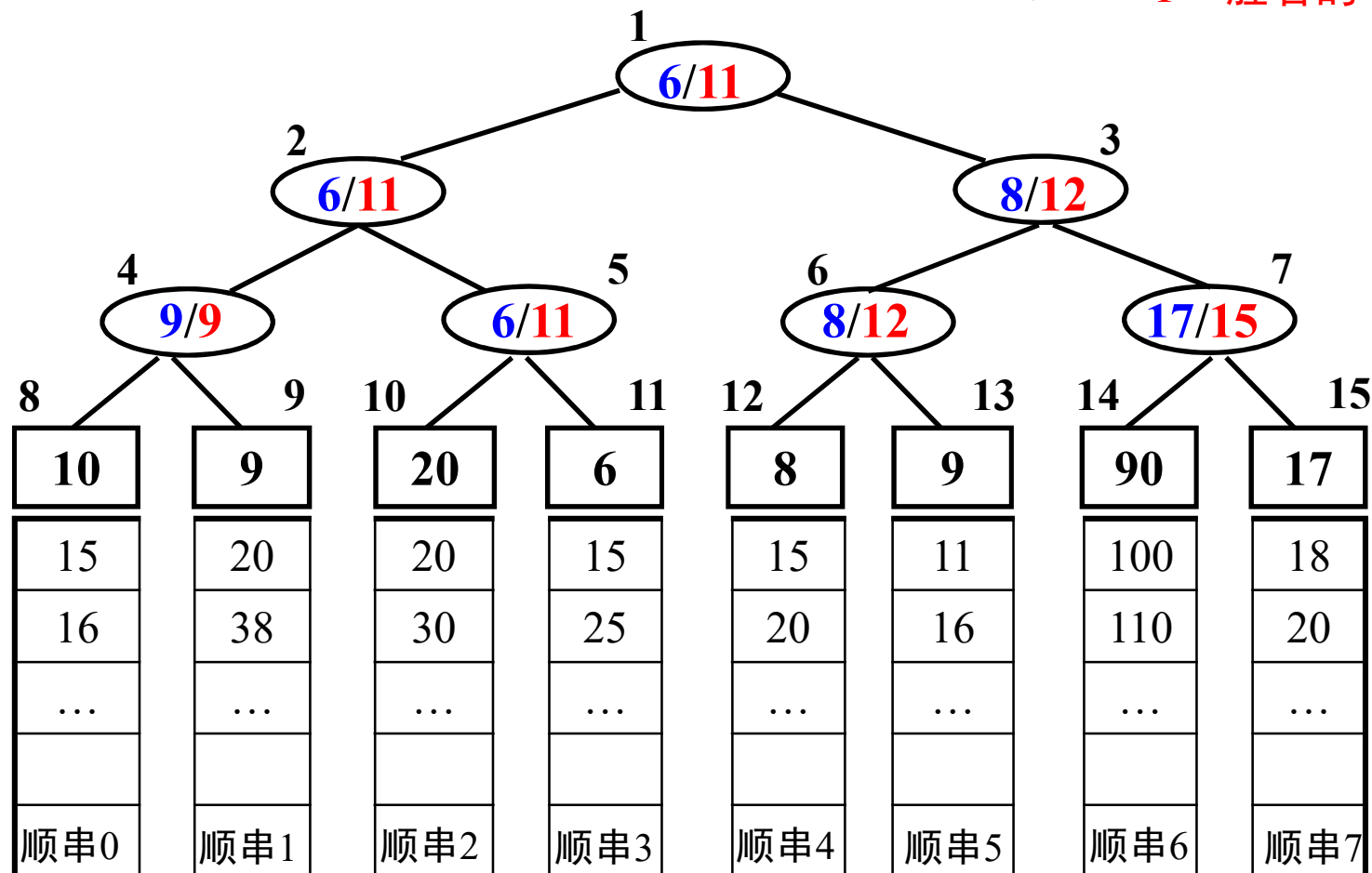
三、胜者树(Winner Tree)

- 胜者树的构建



X—胜者关键字

Y—胜者的下标

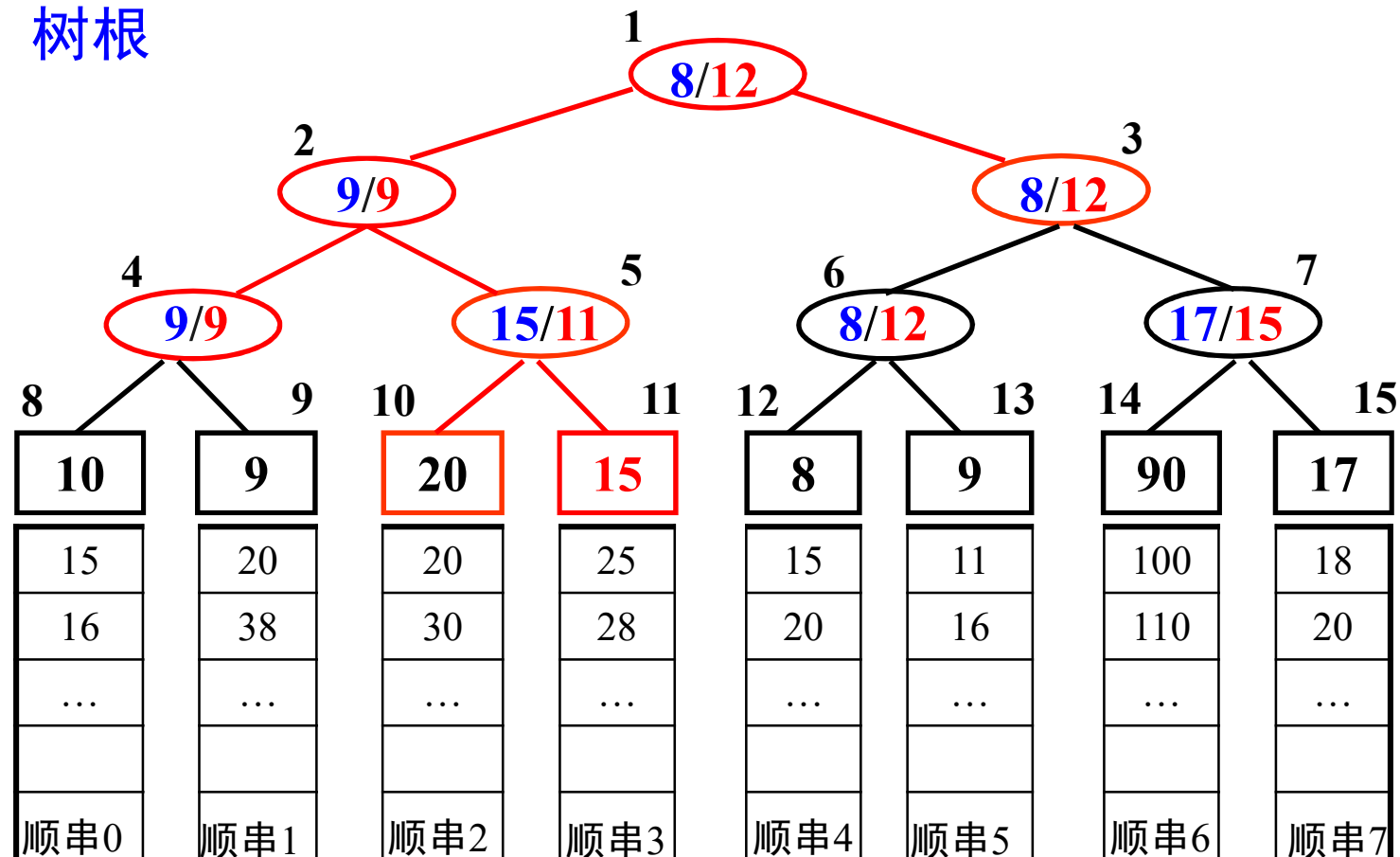




3.4 选择树 (Selection Tree)

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



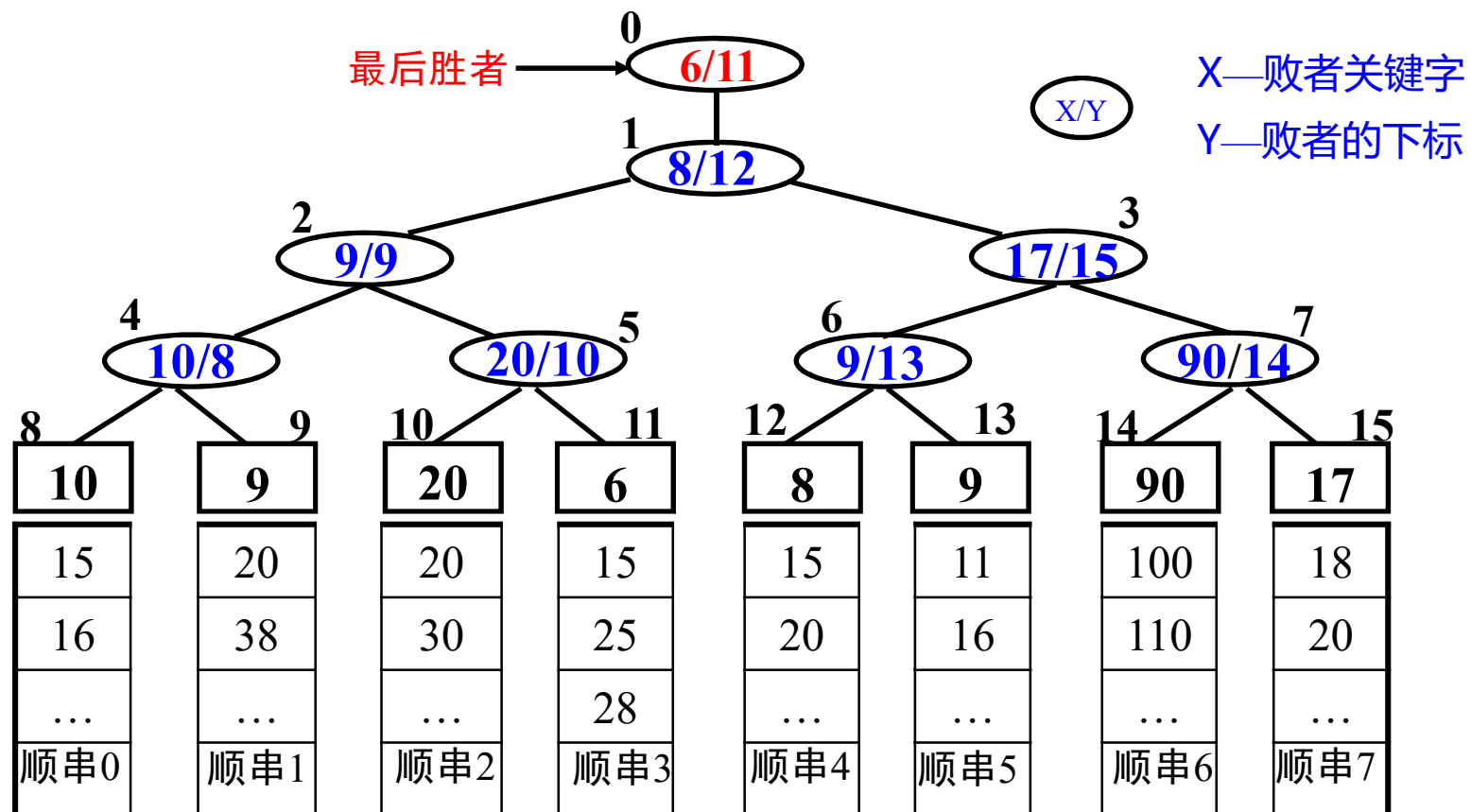


3.4 选择树 (Selection Tree)

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

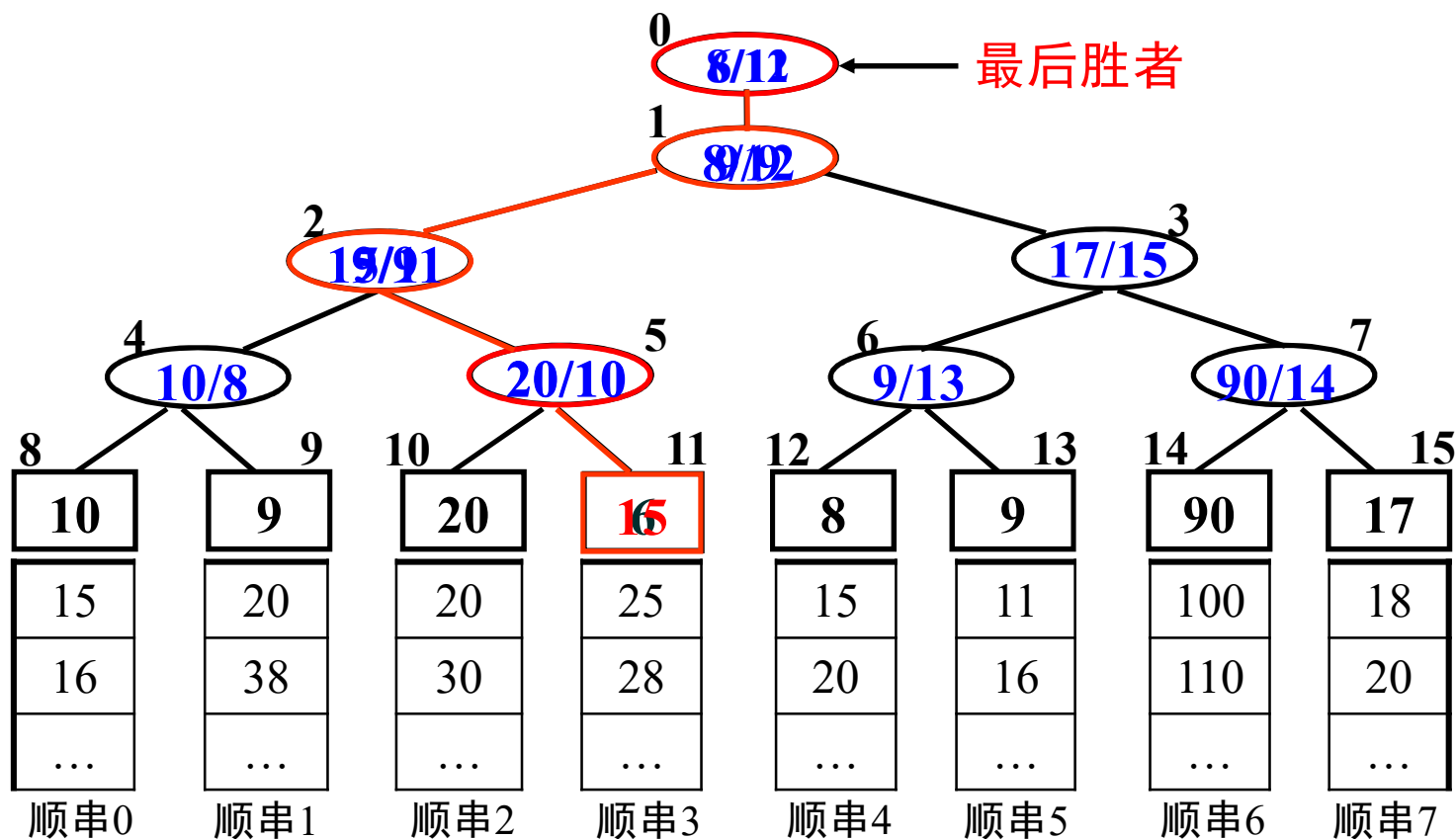




3.4 选择树 (Selection Tree)

- 败者树的重构

- 新进入的结点与其父结点进行比赛：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。

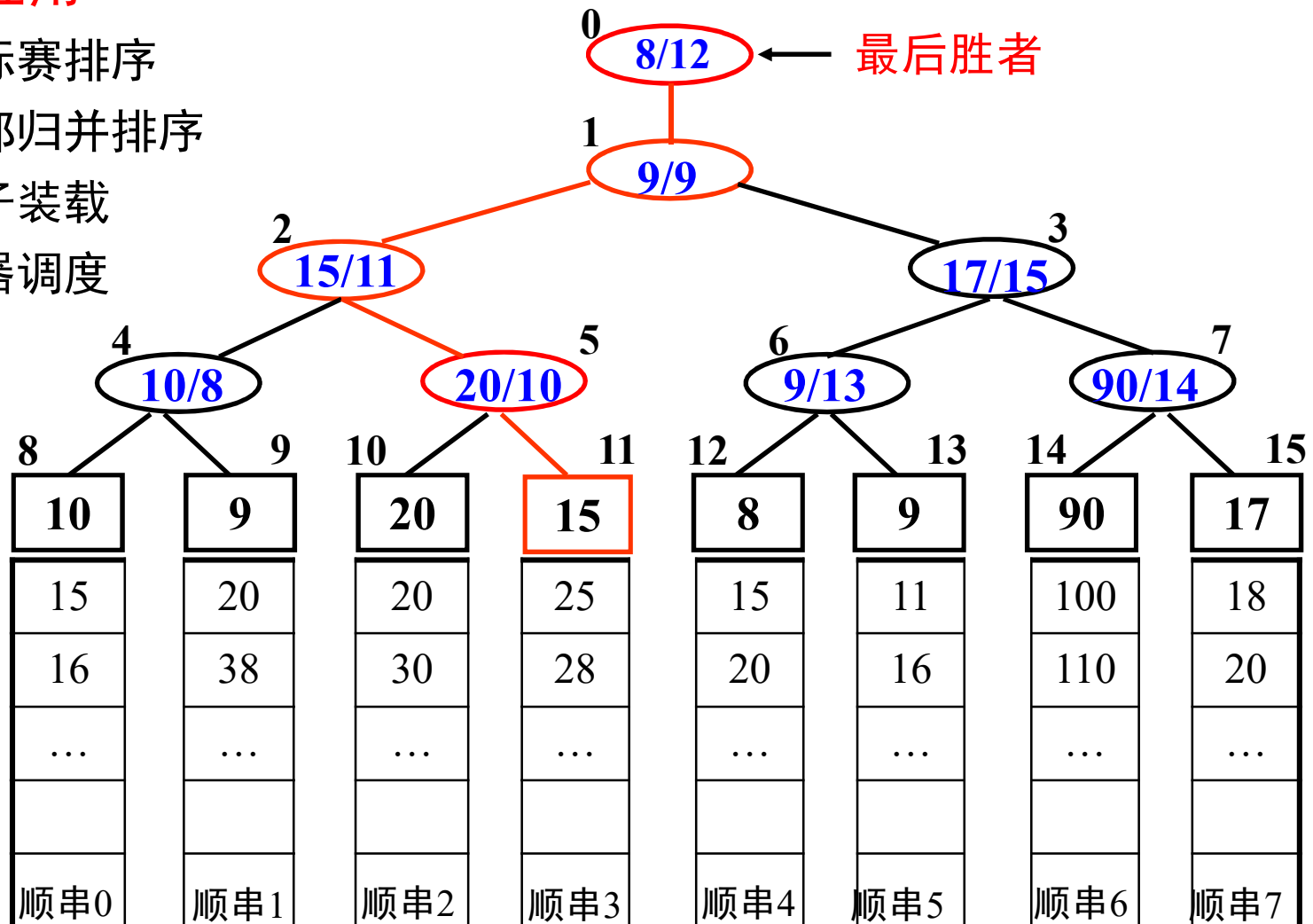




3.4 选择树 (Selection Tree)

五、应用

- 锦标赛排序
- 外部归并排序
- 箱子装载
- 机器调度





3.5 树

树基本操作

- $\text{Parent}(n, T)$ 求结点 n 的双亲
- $\text{LeftMostChild}(n, T)$ 返回结点 n 的最左儿子
- $\text{RightSibling}(n, T)$ 返回结点 n 的右兄弟
- $\text{Data}(n, T)$ 返回结点 n 的信息
- $\text{CreateK}(v, T_1, T_2, \dots, T_k), k = 1, 2, \dots$
 - 建立data域值为 v 的根结点 r , 有 k 株子树 T_1, T_2, \dots, T_k , 且自左至右排列;
 - 返回根结点 r 。
- $\text{Root}(T)$ 返回树 T 的根结点
- 树遍历操作
 - 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.)



- 树遍历操作

- 树三种遍历方式：先序（根）遍历、后序（根）遍历和层序（次）遍历。

- 先序遍历

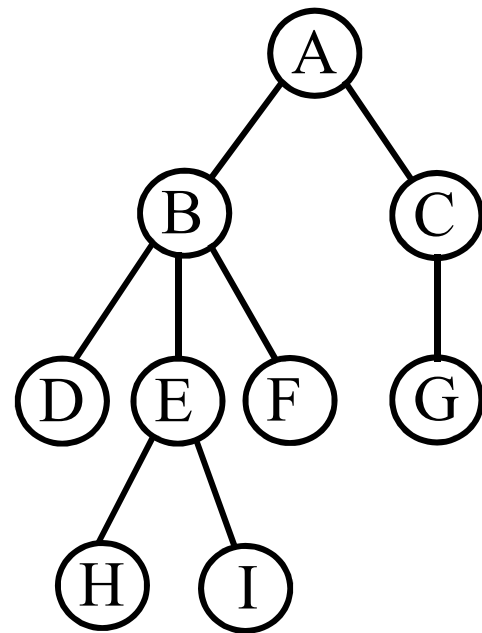
- (1) 访问根结点；
- (2) 按照从左到右的顺序先序遍历根结点的每一棵子树

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

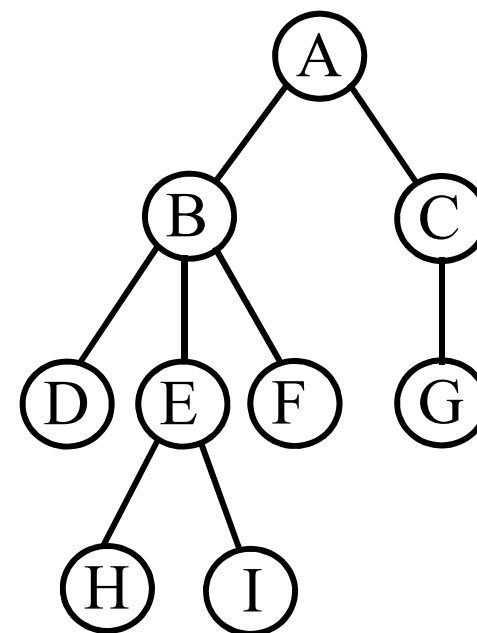
- (1)中序遍历第一棵子树;
- (2)访问根结点;
- (3)按照从左到右的顺序中序遍历根结点的其他子树

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



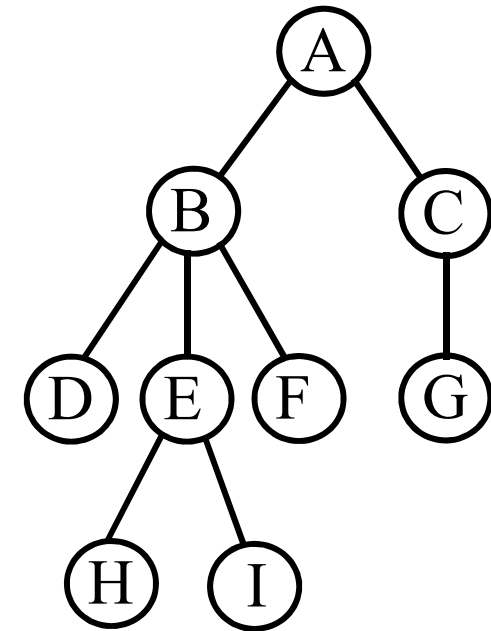


3.5 树(Cont.)

- 使用树的基本操作完成先序遍历递归算法

```
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.)

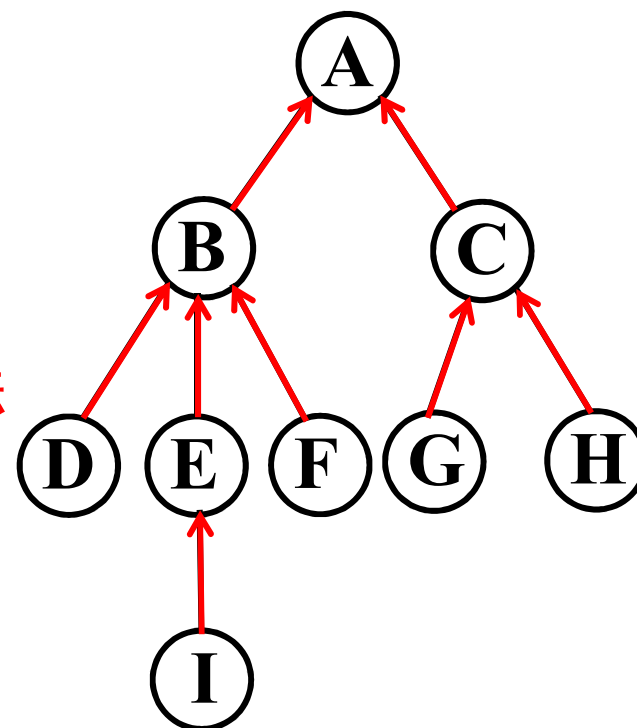


树的存储结构

- **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**唯一双亲**结点
 - 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。

- **结点结构定义**

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





3.5 树(Cont.)

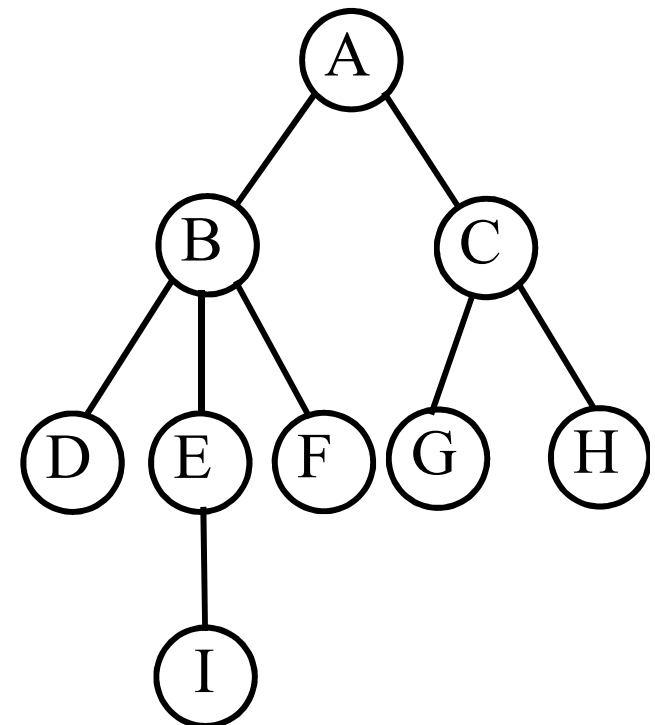
- 双亲表示法（单链表示、父链表示）

- 存储特点：

- 每个结点均保存父结点的数组下标
 - 兄弟结点编号连续。

- 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



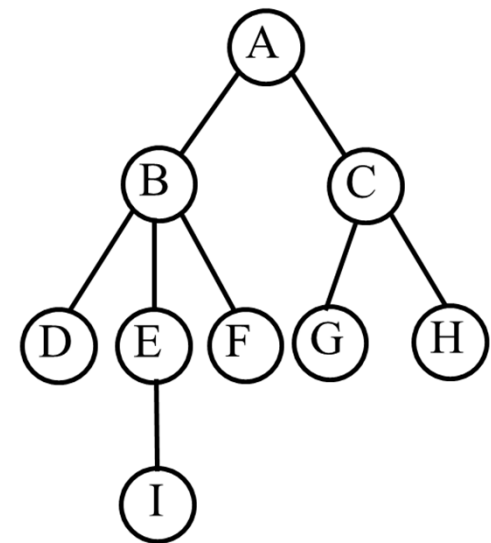
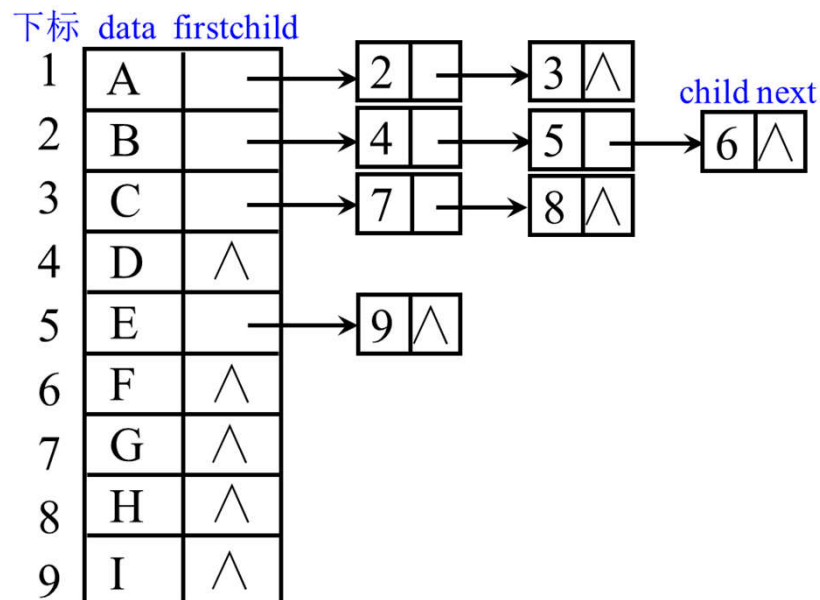


3.5 树(Cont.)

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。





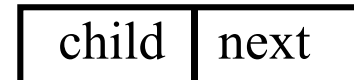
3.5 树(Cont.)

- 孩子链表表示法（邻接表表示）

- 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



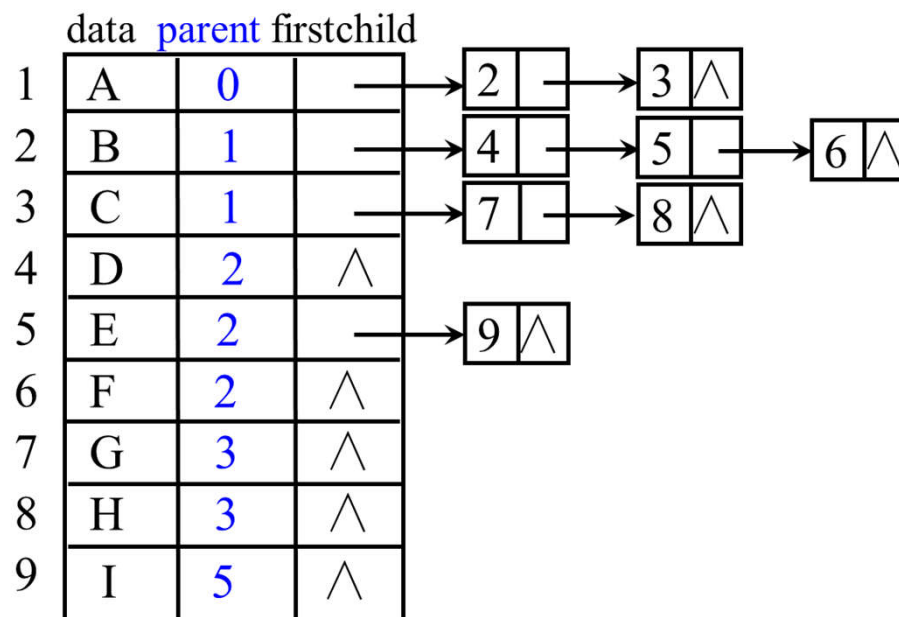
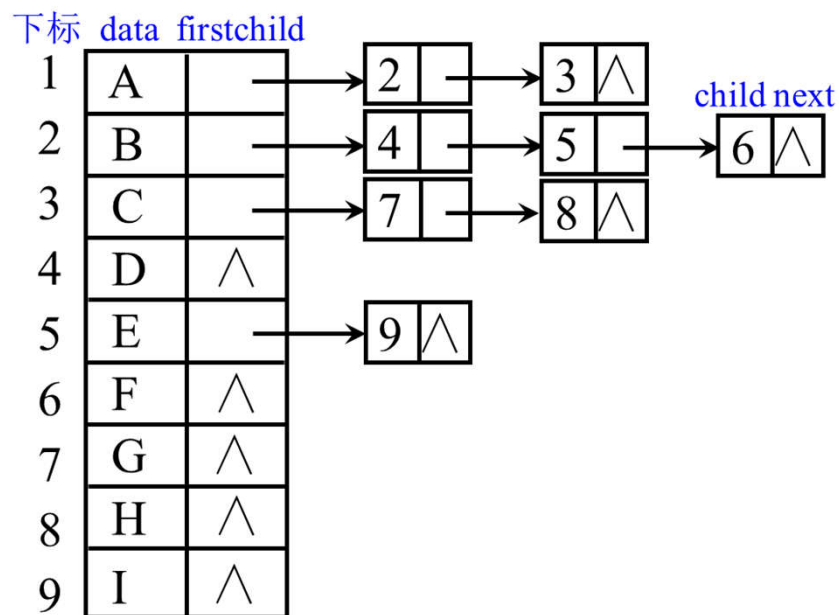
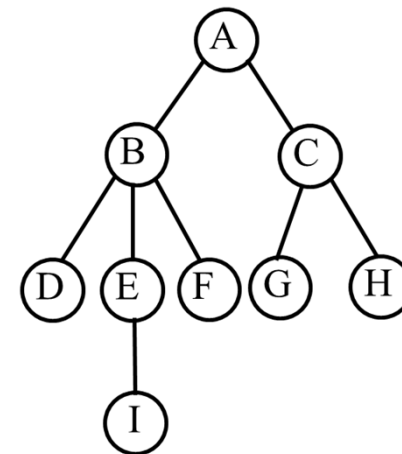
表头结点





3.5 树(Cont.)

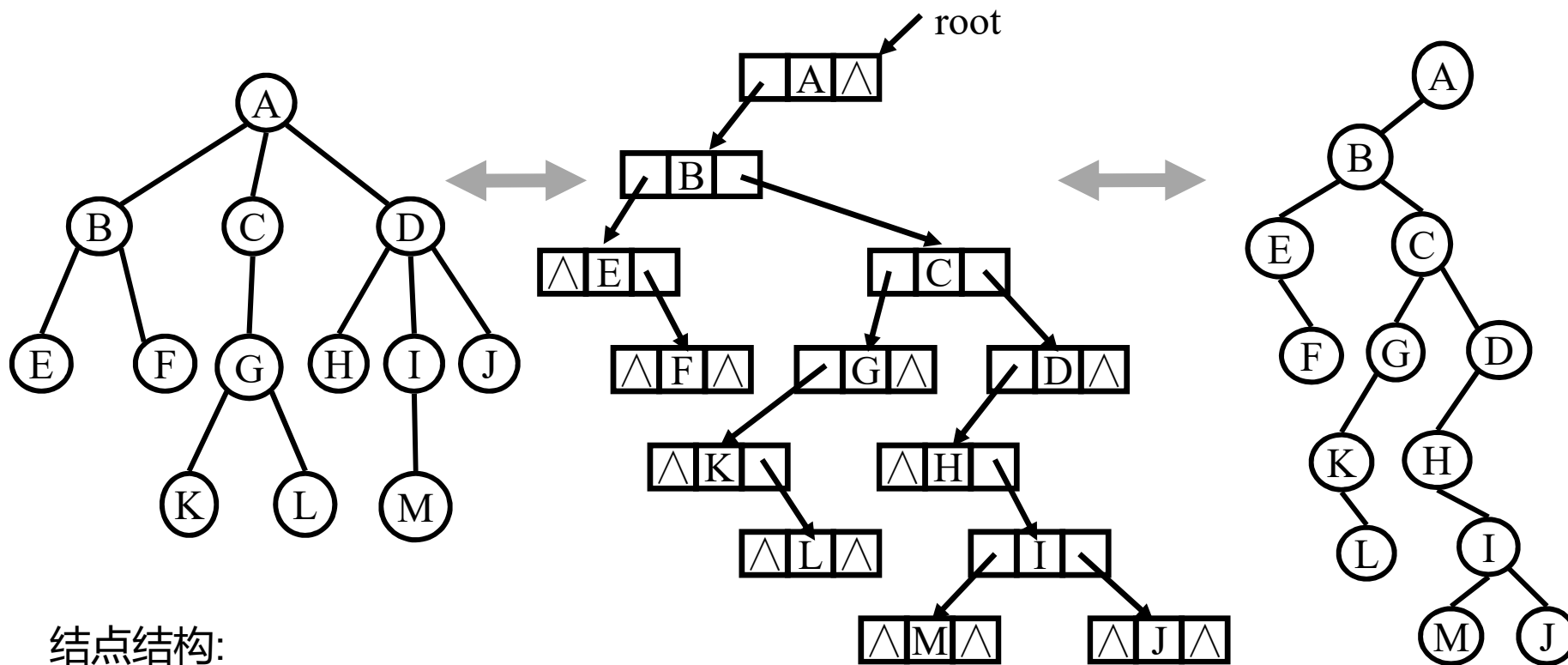
- 孩子链表表示法（邻接表表示）
 - 如何查找孩子结点？时间性能？
 - 如何查找双亲结点？时间性能？
- 双亲孩子表示法





3.5 树(Cont.)

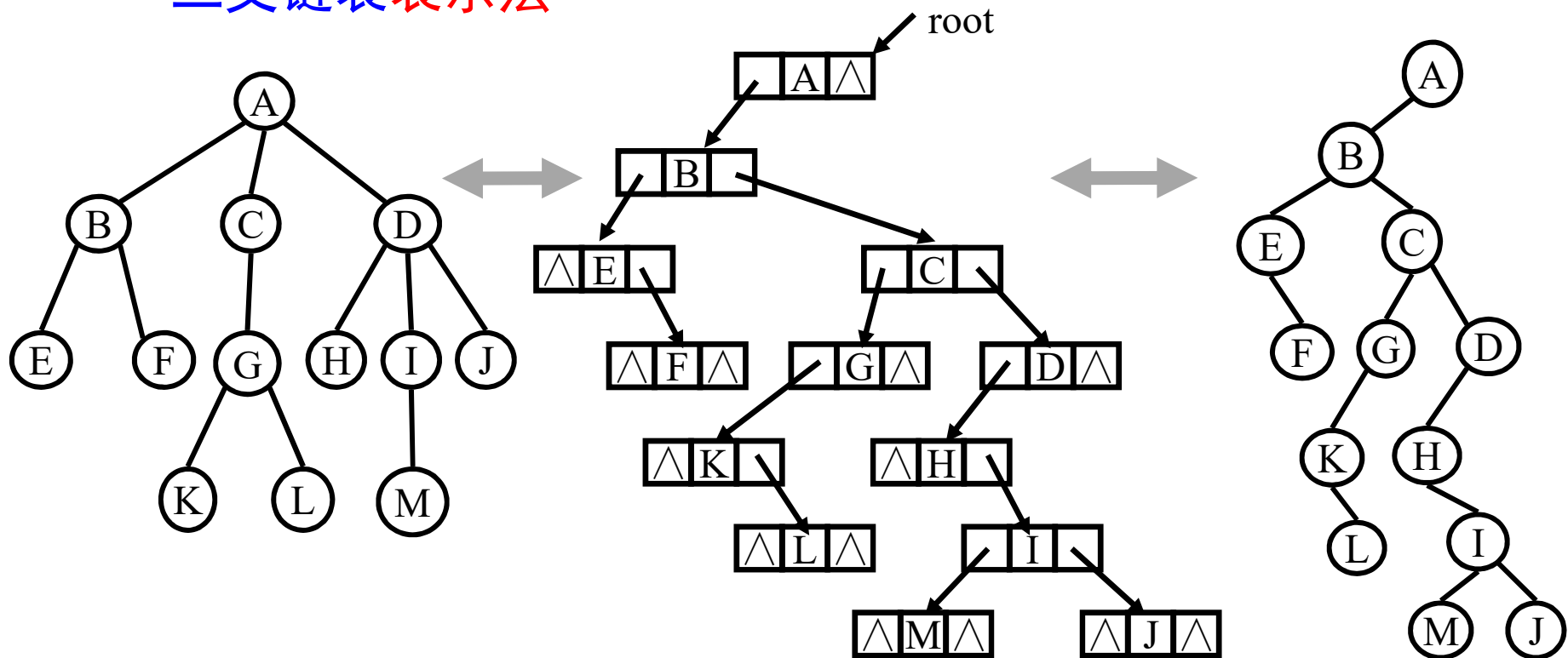
- 二叉链表表示法：(左)孩子—(右)兄弟链表表示
 - 某结点的右兄弟是唯一的
 - 设置两个分别指向该结点的第一个孩子及其右兄弟的指针



树的二叉树表示



- 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.)

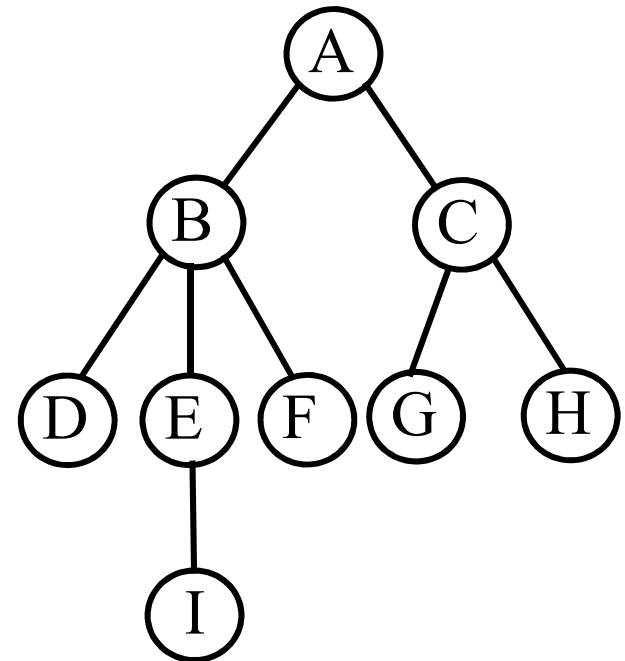
- 二叉链表表示法 ((左)孩子-(右)兄弟链表表示)

- 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 ;
```



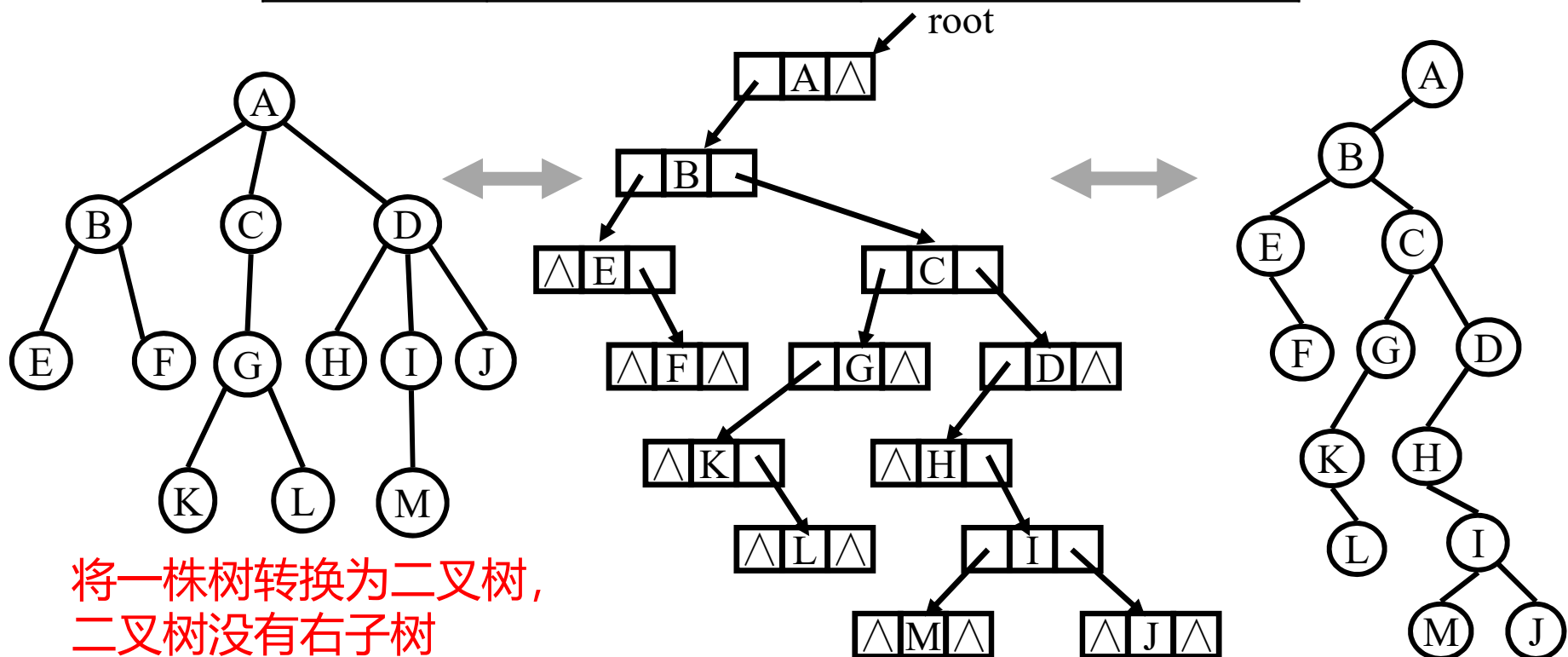


3.6 森林(树)与二叉树间的转换

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 \longleftrightarrow	双亲和右孩子
	双亲和长子 \longleftrightarrow	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:

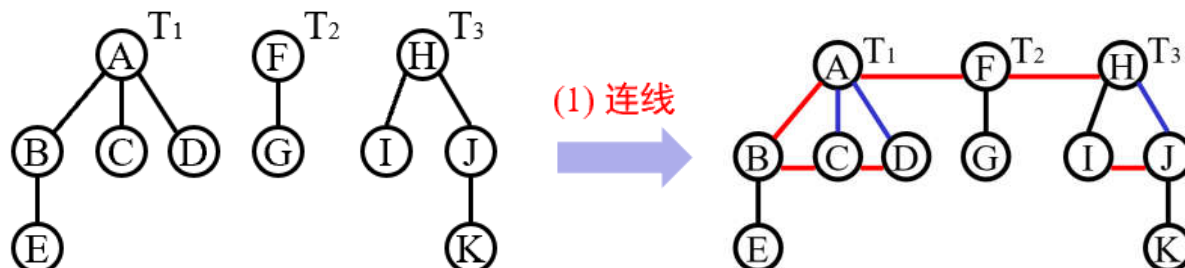
- 把每株树的各兄弟结点连起来;
- 把各株树的根结点连起来 (视为兄弟)

- 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

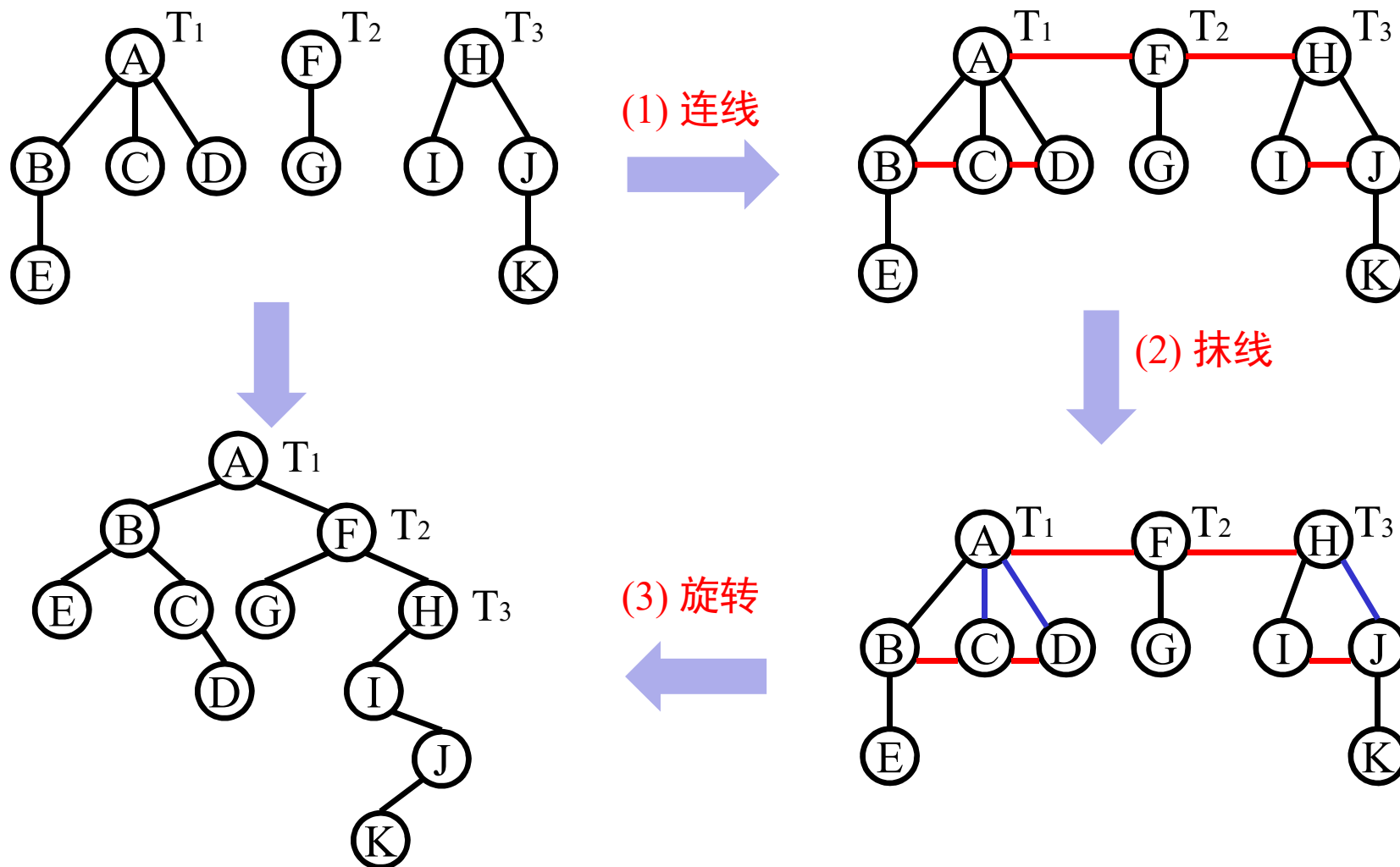




3.6 森林(树)与二叉树间的转换



• 森林(树)转换成二叉树



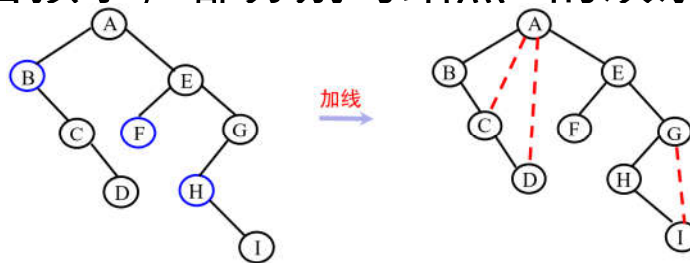


3.6 森林(树)与二叉树间的转换

- 二叉树转换成森林(树)

- 连线:

- 若某个结点 k 是其双亲结点的左孩子, 则将结点 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;

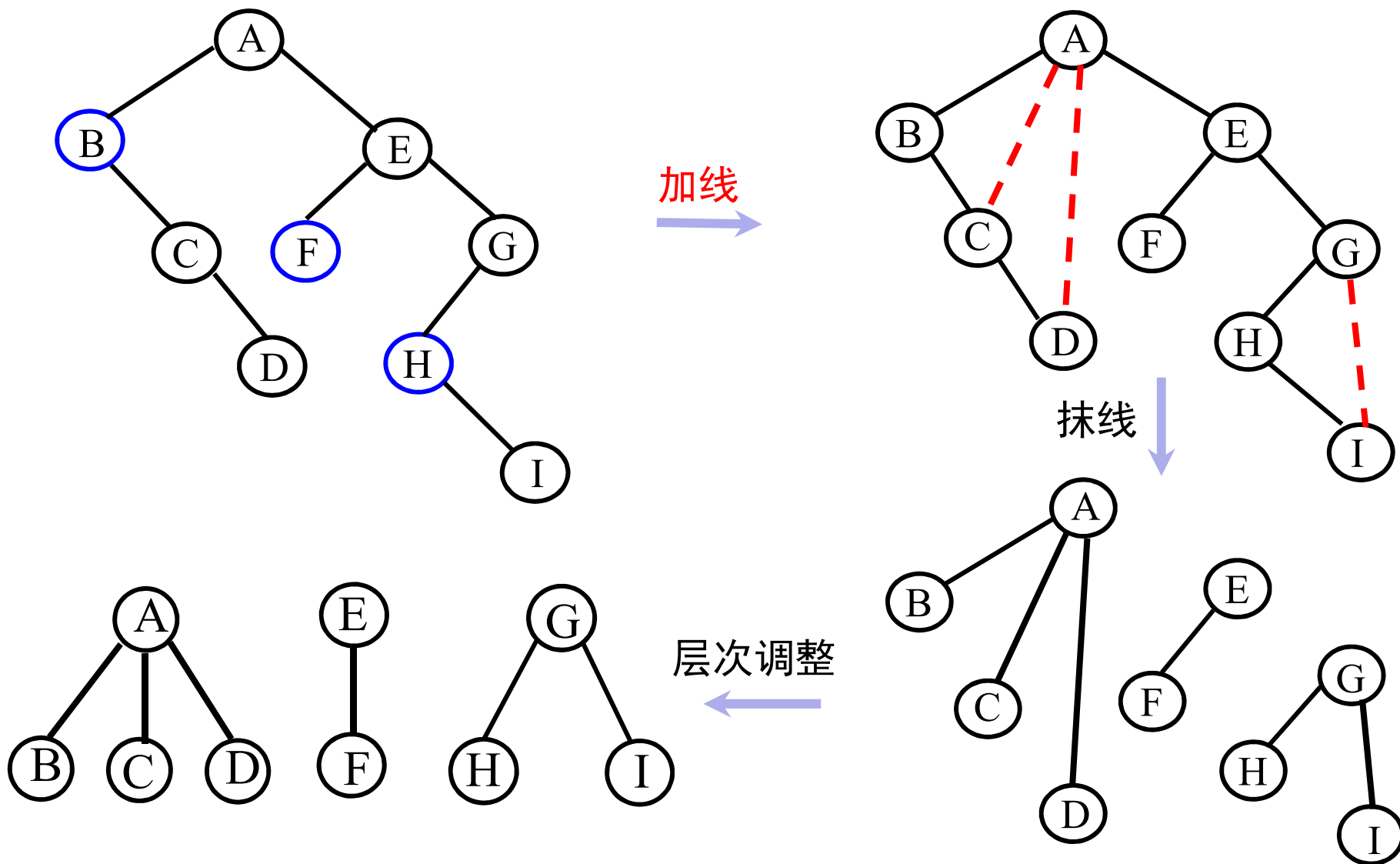
- 旋转: (不是必要的)

- 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

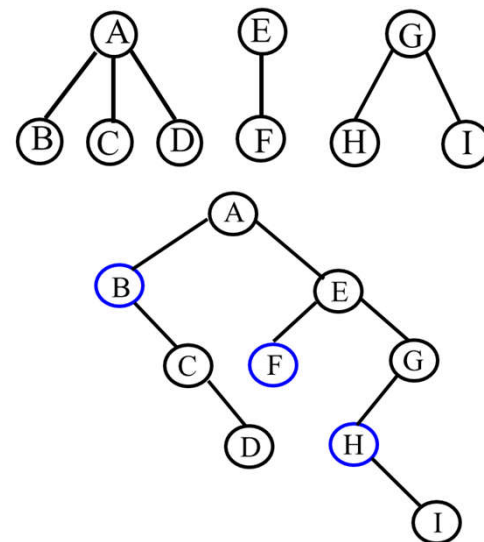




3.6 森林(树)与二叉树间的转换

- 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
- 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**；
 - 其余树对应二叉树的右子树。
- 思考：
 - 森林中的**非终端结点数**
 $F = \text{对应二叉树中右子树为空的结点数} B - 1$

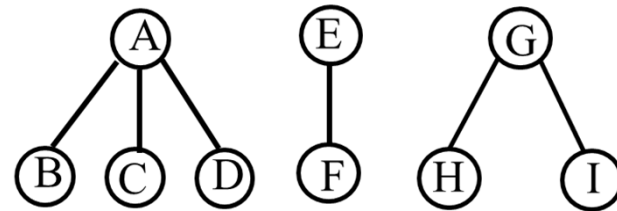




3.6 森林(树)与二叉树间的转换

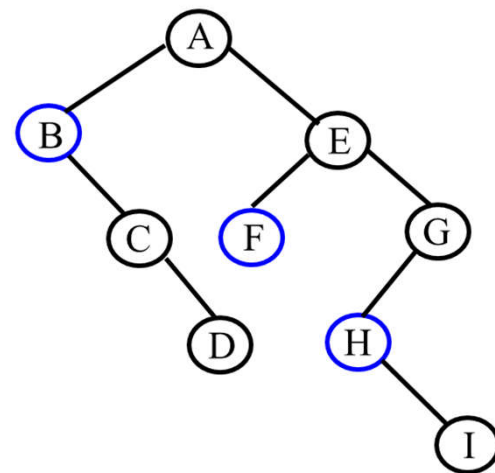
- 森林(树)转换成二叉树的递归算法

- $F = \{T_1, T_2, \dots, T_n\}$ 二叉树 $B(F)$
- 若 $n=0$, 则 $B(F)$ 为空; 否则, $n>0$, 则
 - $B(F)$ 的根就是 $\text{root}(T_1)$;
 - $B(F)$ 的左子树对应 F 第一棵树 T_1 的子树;
 - $B(F)$ 的右子树对应 F 其余树。



- 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 F_1 是由 B 的左子树转换来的;
 - F 中除 T_1 外其余子树组成的森林 $F' = \{T_2, \dots, T_n\}$ 是由 B 右子树转换而来的。





3.6 森林(树)与二叉树间的转换

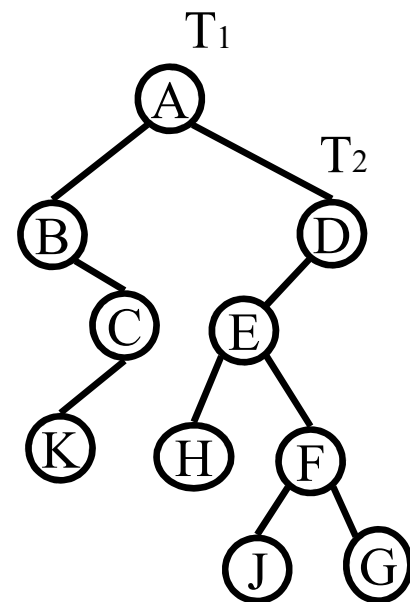
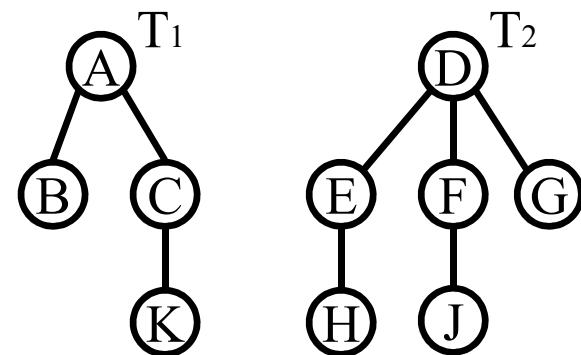
• 非空森林的基本遍历

– 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**全部子树**；
- 按先根顺序遍历其余**树**。

– 后根遍历

- 按后根顺序遍历第一株树的**子树**；
- 访问第一株树的根结点；
- 按后根顺序遍历其余**树**。



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, Union-Find Disjoint Sets)

- 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(x, S): 建立集合 S , 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.)

一、非相交集合的树结构表示

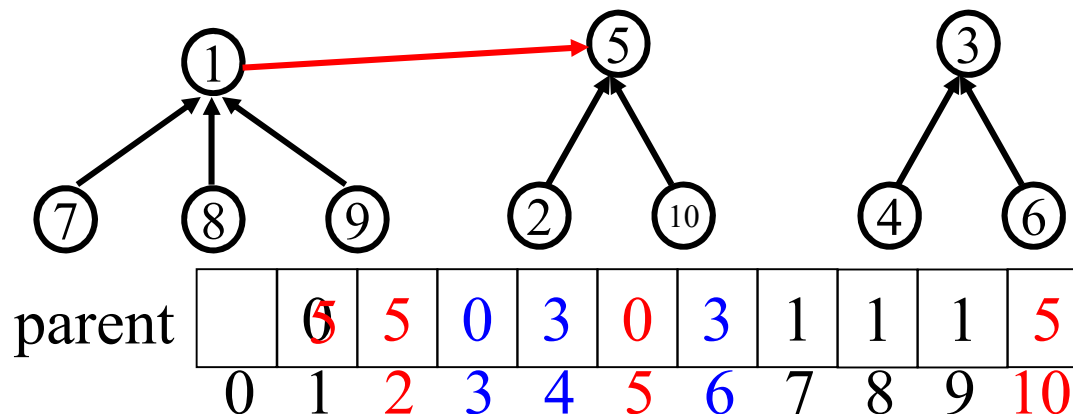
• ADT集合MFSET的实现

– 集合树结构表示（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。

– 集合操作实现

- “**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.)

一、非相交集合的树结构表示

- 集合的存储结构

#define n 元素的数量

typedef int MFSET[n+1];

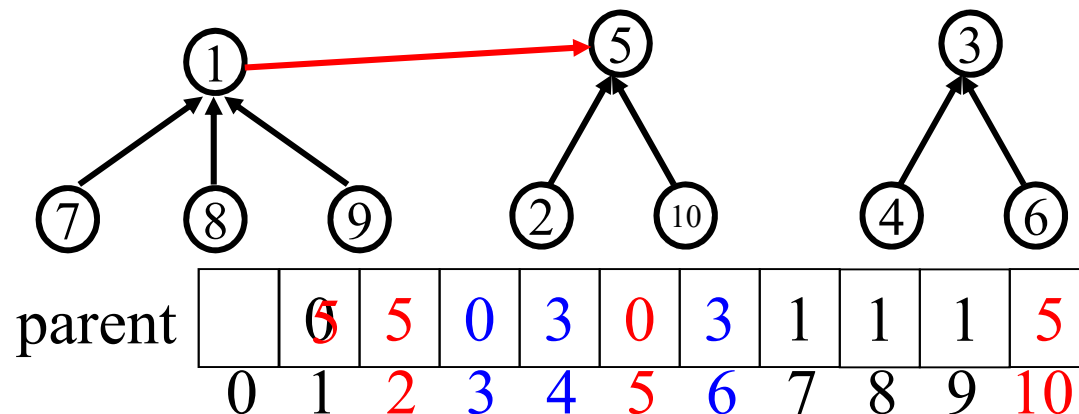
/* 集合的“型”为MFSET, 元素的“型”为int */

- 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)





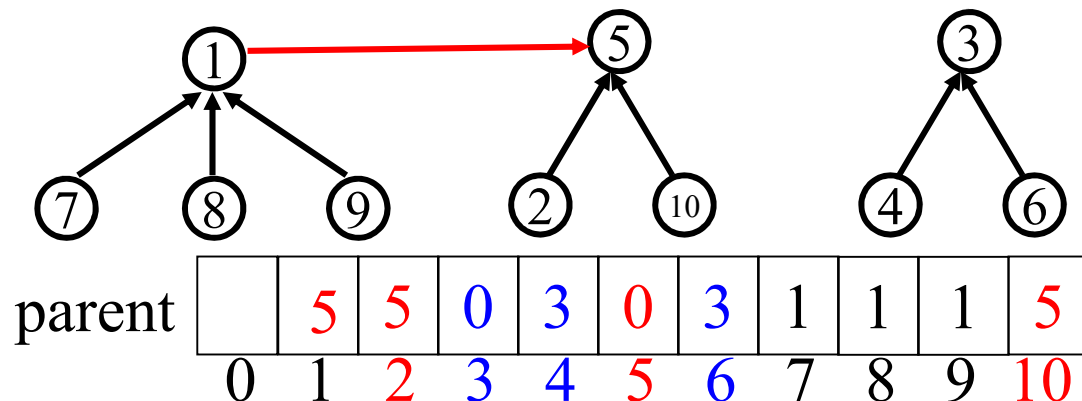
3.7 树型结构的应用(Cont.)

一、非相交集合的树结构表示

• 基本操作实现

```
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)
```





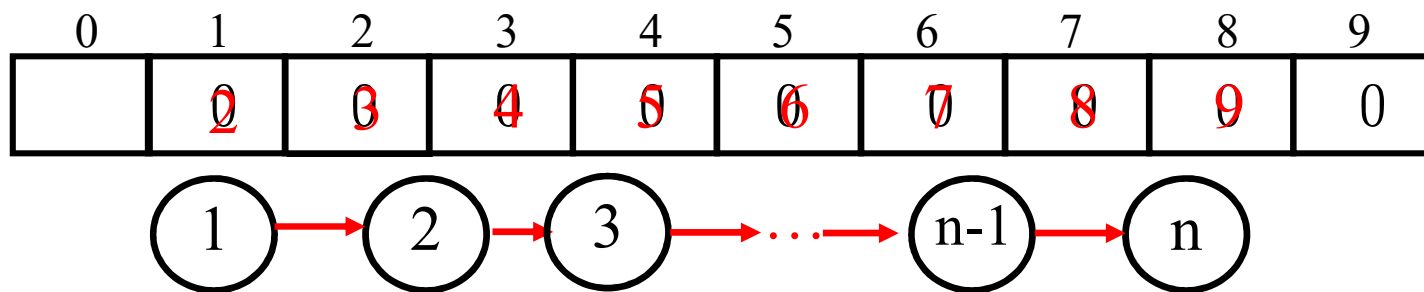
3.7 树型结构的应用(Cont.)

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。



- 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:

```
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:  $h \leq \lfloor \log_2 n \rfloor + 1$ 
```

```
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： $x \in S, y \in S, x \equiv y, (x, y) \in R$ 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：
 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_n
 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
 - 2.2 若 $S_k \neq S_m$, 则将 S_k 并入 S_m , 并将 S_k 置空。
- 当所有等价对处理后, S_1, S_2, \dots, S_n 中的非空集合即为S的R等价类。
- 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.)



等价分类算法实现:

```
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;
    }
}
```

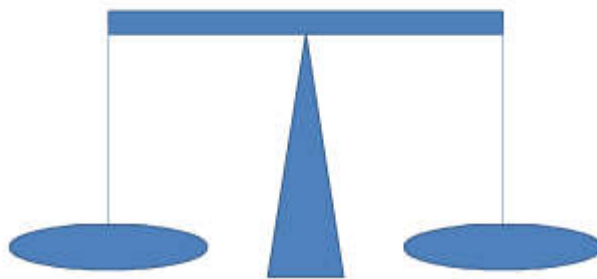


3.7 树型结构的应用(Cont.)

二、判定树

- 八枚硬币问题：

- 假设有八枚硬币a、b、c、d、e、f、g、h，已知其中1枚是假币，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。

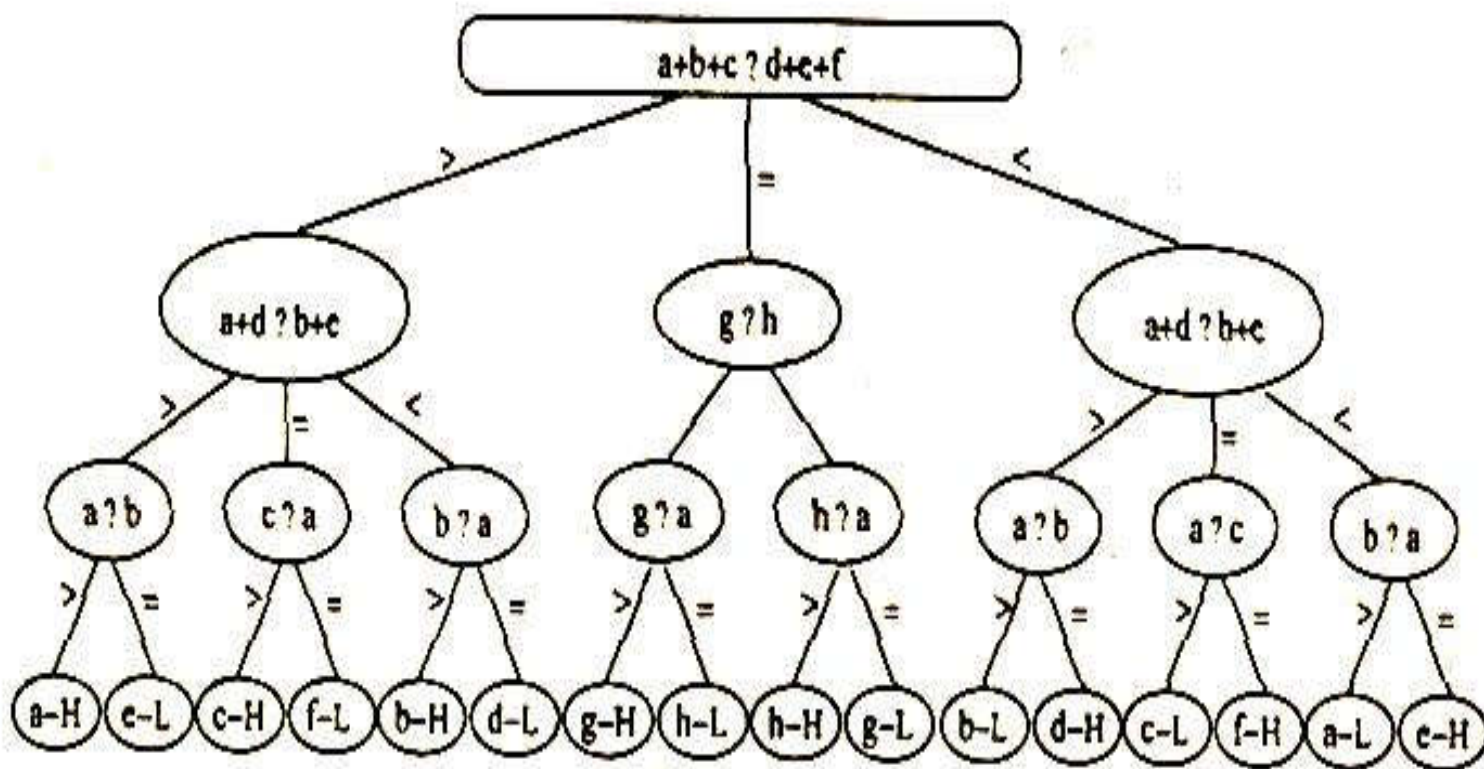




3.7 树型结构的应用(Cont.)

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

所有可能答案集合

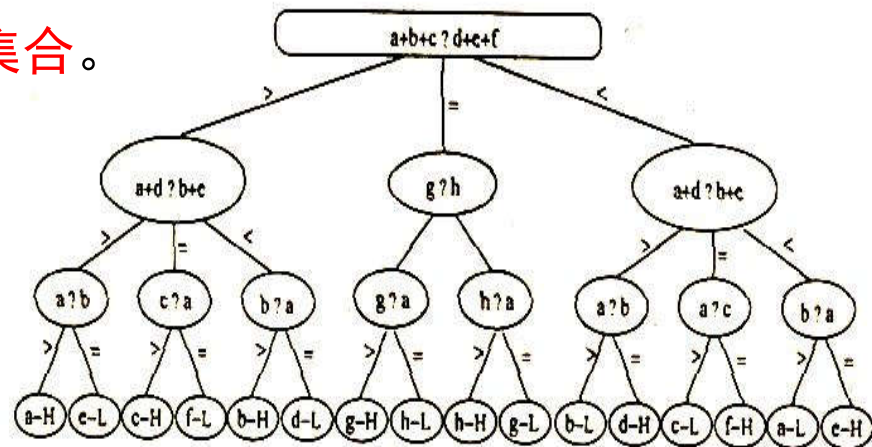


3.7 树型结构的应用(Cont.)

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；
- 一个求解过程对应于从根到叶的一条路；
- 一个判定树是所有可能解的集合。





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**权值**：

- 对叶子结点赋予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：

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

w_k 第 k 个叶子的权值

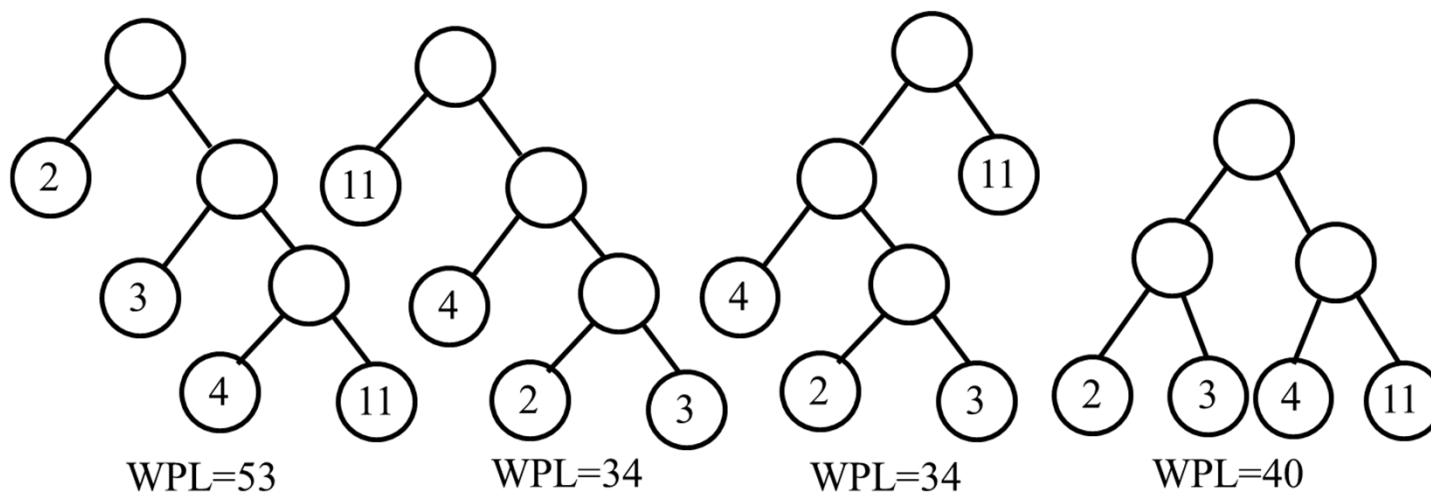
l_k 从根结点到第 k 个叶子的路径长度



3.7 树型结构的应用(Cont.)

- 相关术语:

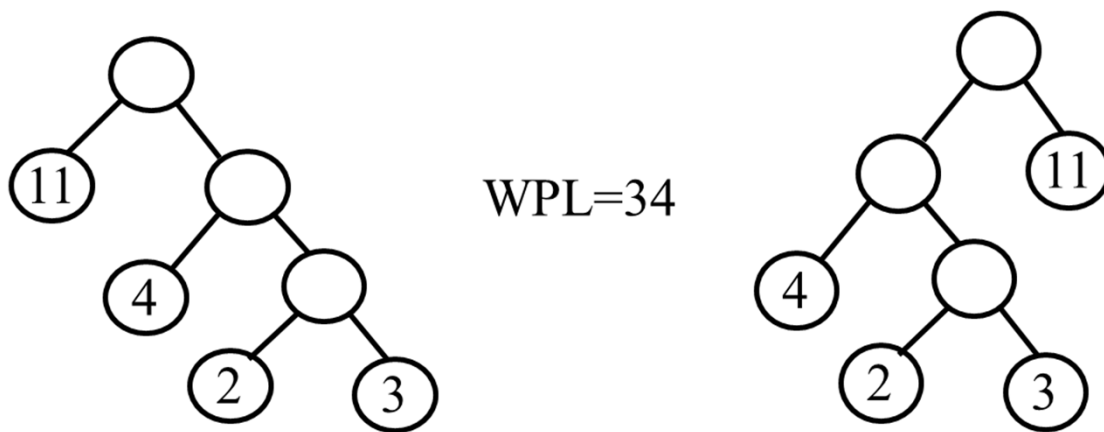
- 举例: 给定4个叶子结点, 其权值分别为 $\{2, 3, 4, 7\}$, 可以构造出形状不同的二叉树
- 哈夫曼树: 给定一组具有确定权值的叶子结点, 带权路径长度最小的二叉树, 称为哈夫曼树, 亦称最优二叉树。





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。（构造哈夫曼树的指导思想）
 - 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.)

- 哈夫曼树的构造算法:

- 输入: n 个权值 $\{w_1, w_2, \dots, w_n\}$
- 输出: 哈夫曼树

- 步骤:

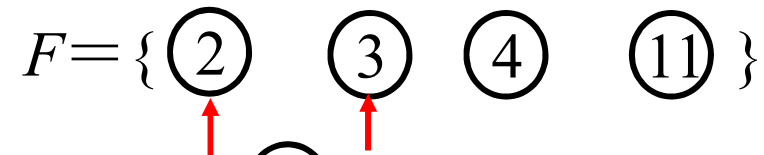
- (1) 初始化: 由给定的 n 个权值 $\{w_1, w_2, \dots, w_n\}$ 构造 n 棵只有一个根结点、左右子树均空的二叉树, 从而得到一个二叉树集合 $F = \{T_1, T_2, \dots, T_n\}$;
- (2) 选取与合并: 在 F 中选取根结点权值最小的两棵二叉树, 分别作为左、右子树构造一棵新的二叉树, 这棵新二叉树的根结点的权值为其左、右子树根结点的权值之和;
- (3) 删除与加入: 在 F 中删除作为左、右子树的两棵二叉树, 并将新建立的二叉树加入到 F 中;
- (4) 重复(2)、(3)两步, 当集合 F 中只剩下一棵二叉树时, 这棵二叉树便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



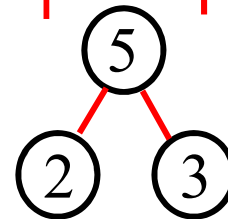
3.7 树型结构的应用(Cont.)

- 哈夫曼树的构造示例: $W = \{2, 3, 4, 11\}$

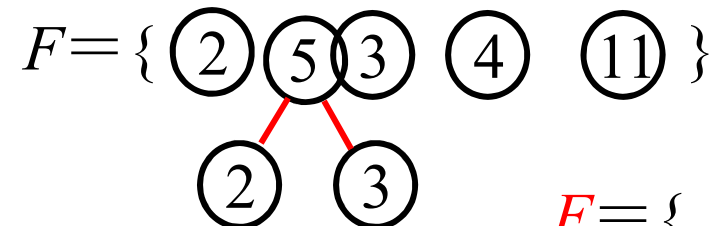
– 初始化:



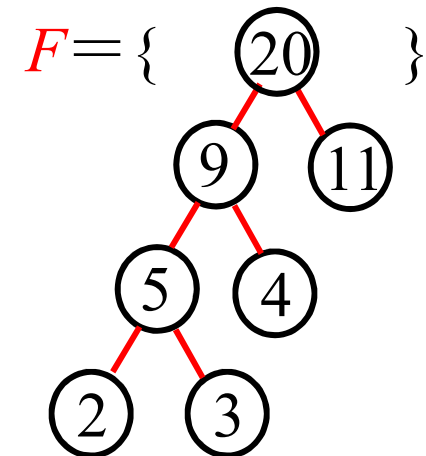
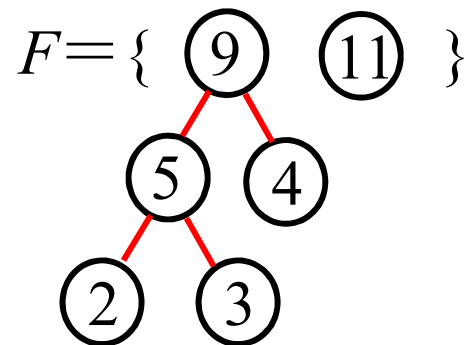
– 选取与合并:



– 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 个独立根结点的权值;

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, 其中,

$$0 \leq p_1, p_2 \leq i-1;$$

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	0	7	-1	-1	-1
⑤	1	5	-1	-1	-1
②	2	2	-1	-1	-1
④	3	4	-1	-1	-1
	4		-1	-1	-1
	5		-1	-1	-1
	6		-1	-1	-1

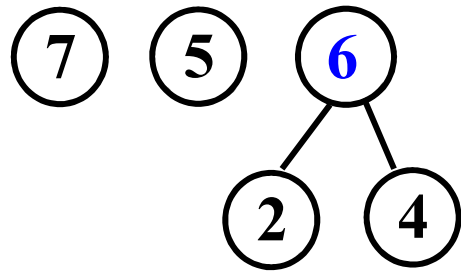
初始化



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

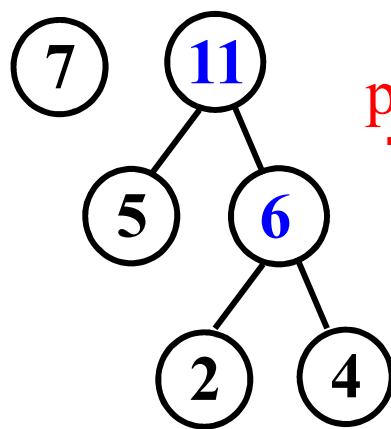


构造过程



3.7 树型结构的应用(Cont.)

- 哈夫曼树构造算法的实现示例:



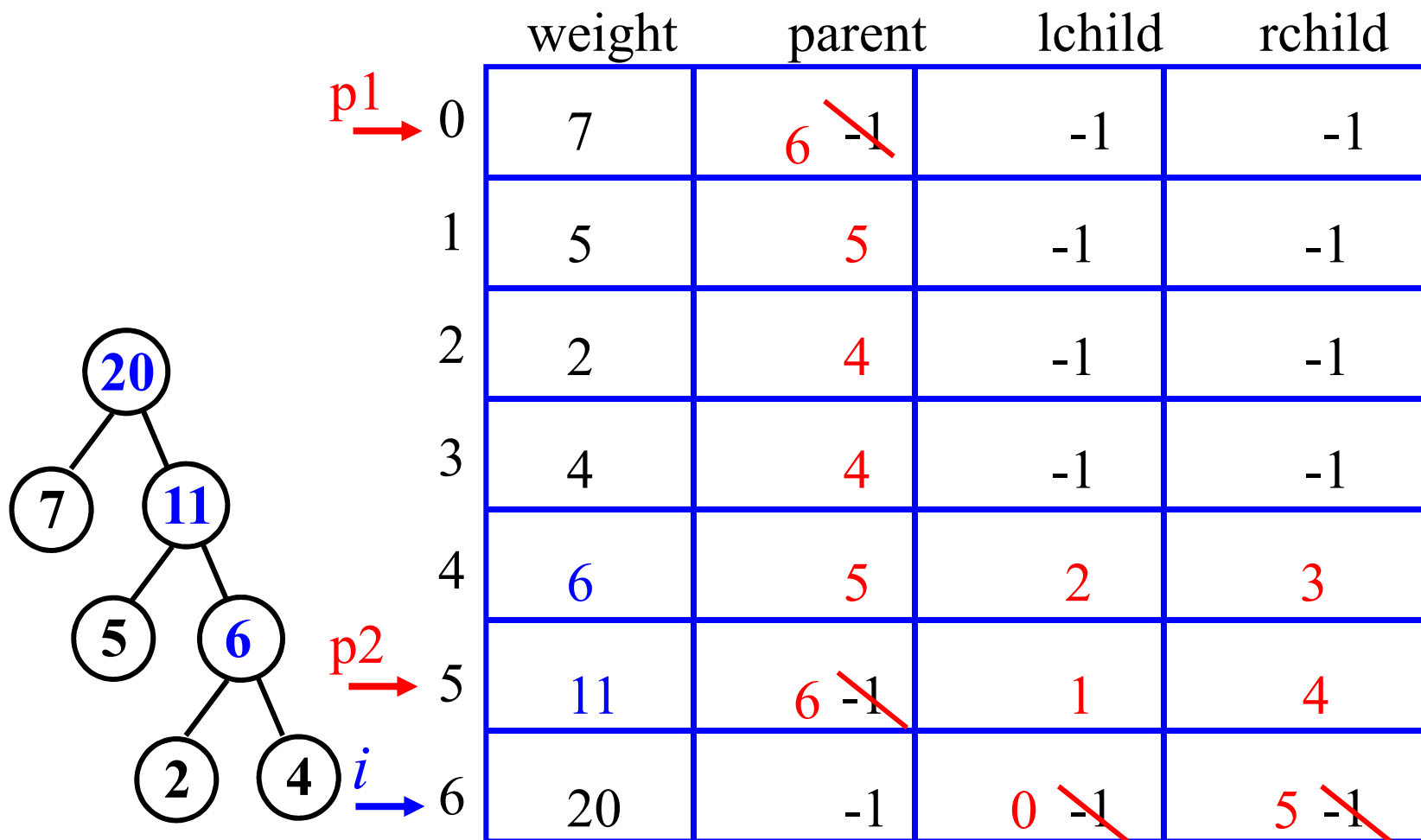
	weight	parent	lchild	rchild
0	7	-1	-1	-1
$p1 \rightarrow$ 1	5	5 -1	-1	-1
2	2	4	-1	-1
3	4	4	-1	-1
$p2 \rightarrow$ 4	6	5 -1	2	3
$i \rightarrow$ 5	11	-1	1 -1	4 -1
6		-1	-1	-1

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示例:



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 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 选取最小两个根结点
    T[p1].parent = T[p2].parent = i; //3.2 合并为一棵新二叉树
    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，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

- 哈夫曼算法：求字符集最优前缀编码

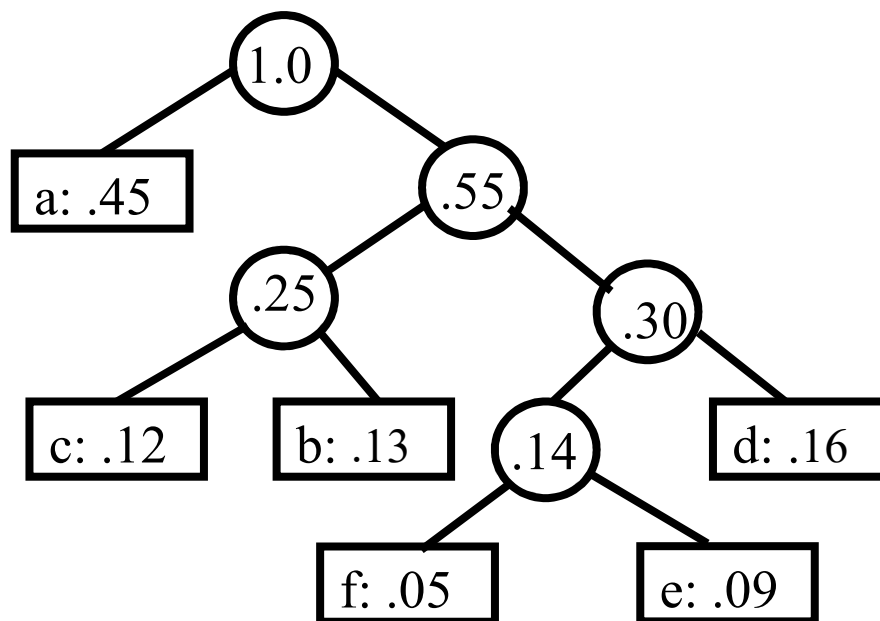
- **初始化**：使字符集中的每个字符对应一棵只有叶结点的二叉树，**叶的权值为对应字符的使用频率**。
- **构造算法**：利用huffman算法来**构造**一棵huffman树。
- **哈夫曼编码**：对huffman树上的每个结点，其左支附以0，右支附以1(**或者相反**)，则**从根到叶**的路径上的0、1序列就是相应字符的编码，**且为最优前缀码**



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 = $\lceil \log_2 C \rceil$
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

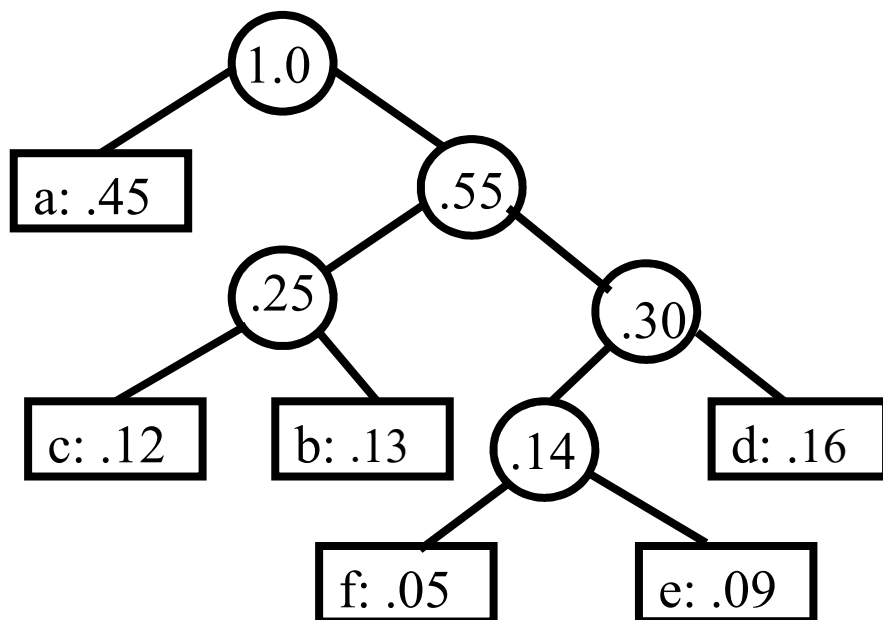


3.7 树型结构的应用(Cont.)

- 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。

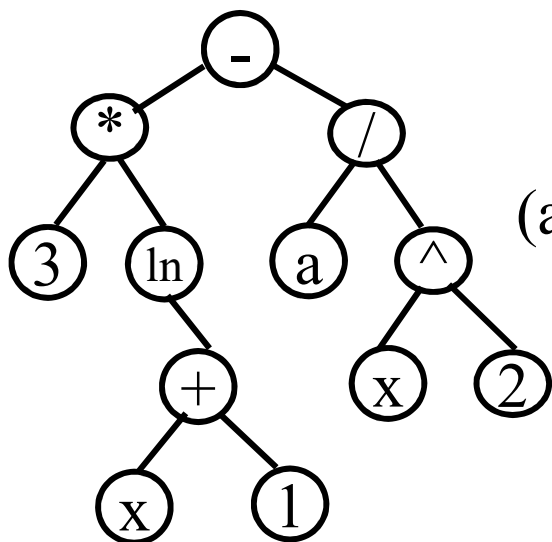


3.7 树型结构的应用(Cont.)

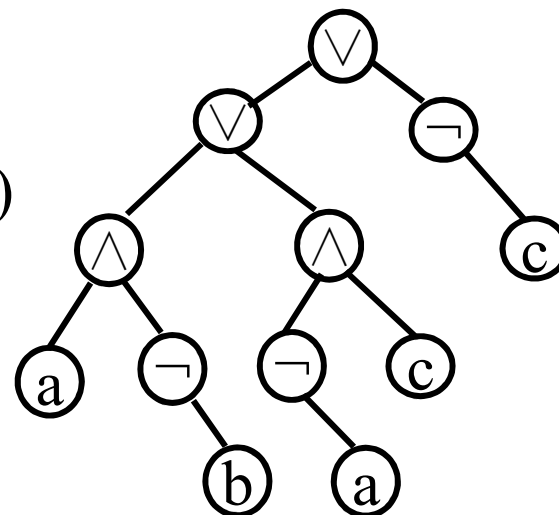
四、树的应用：表达式求值

- 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。



$$3 * \ln(x + 1) - a / x^2$$
$$(a \wedge \neg b) \vee (\neg a \wedge c) \vee (\neg c)$$



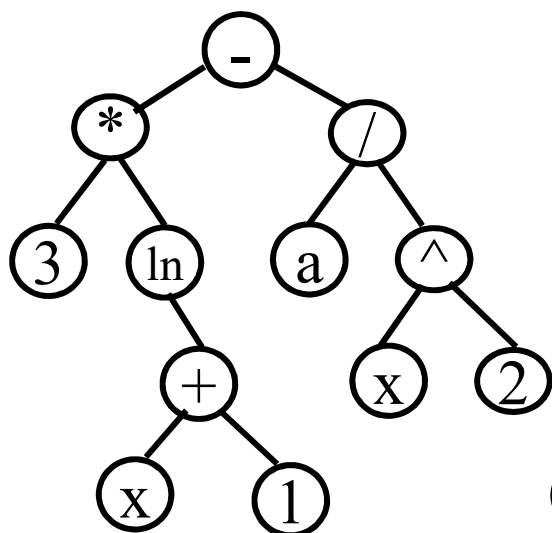


3.7 树型结构的应用(Cont.)

四、树的应用：表达式求值

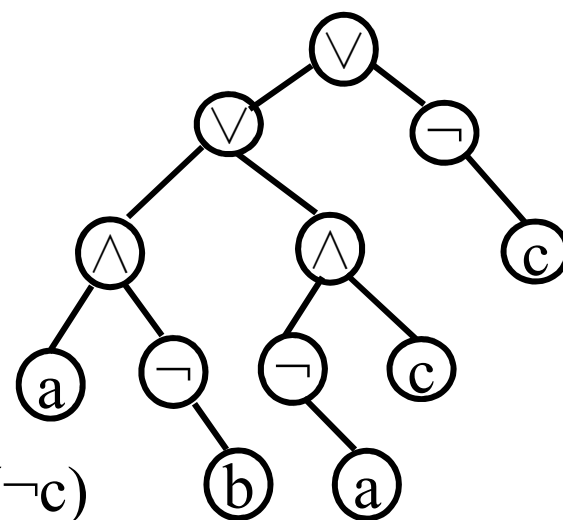
• 表达式求值方法

- 把中缀表达式转换为**后缀表达式**（**栈结构、树结构**），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。



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

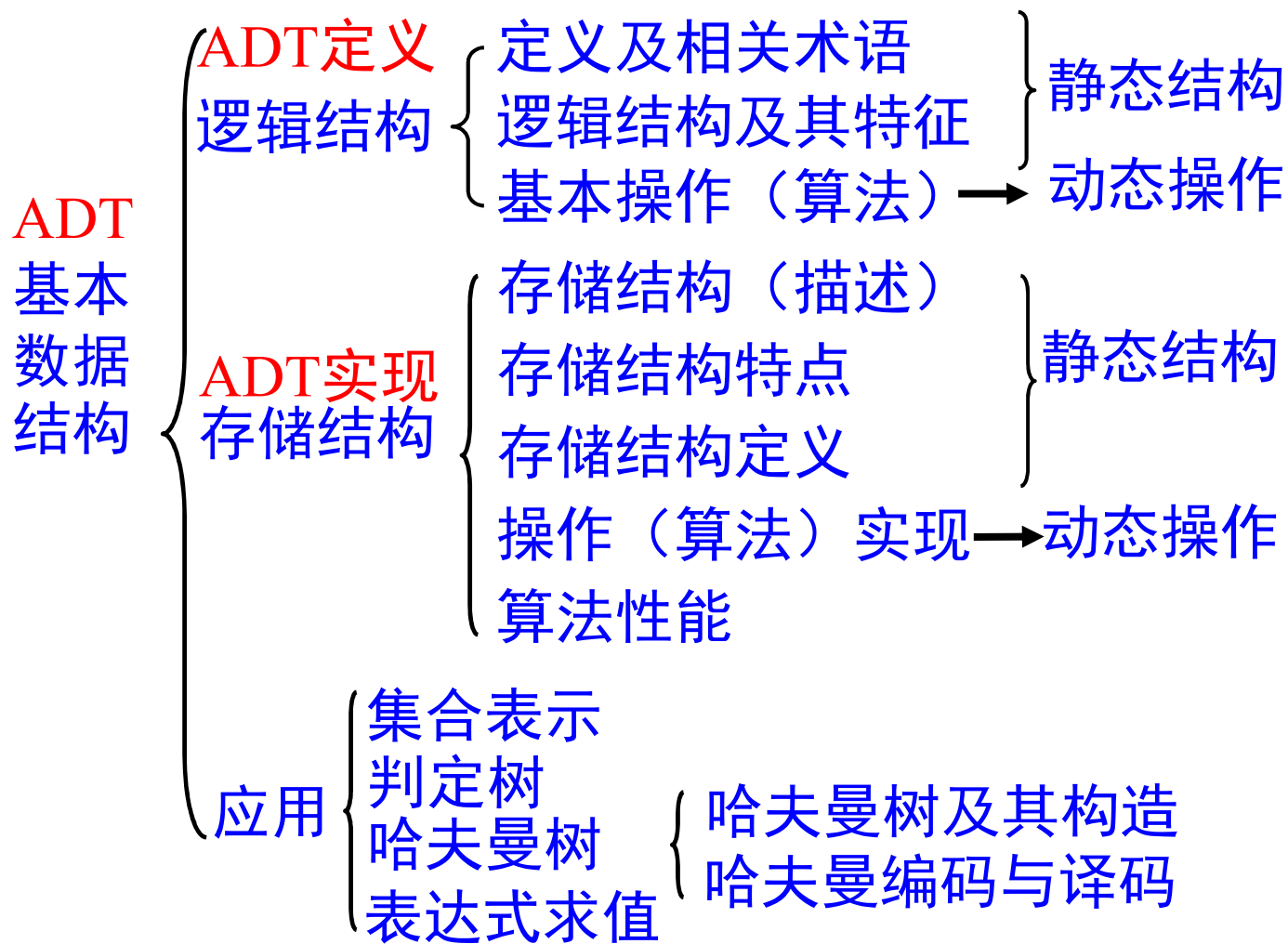
$$(a \wedge \neg b) \vee (\neg a \wedge c) \vee (\neg c)$$





本章小结

- 知识点：二叉树、森林（树）
- 知识点体系结构





本章小结

• 知识点总结

