

§ 6. 树和二叉树

6.1. 树的定义和基本术语

6.1.1. 树的基本特征及描述

基本特性：树由 n 个结点组成, 元素间存在一对多的关系

$n=0$ 空树

$n>0$ 非空树

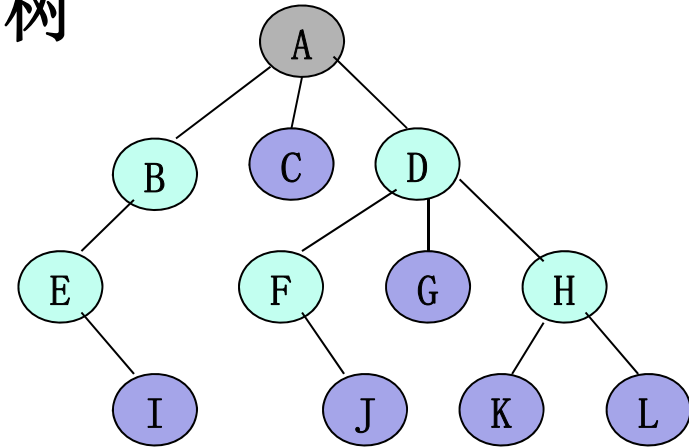
- ★ 有唯一一个点没有前驱, 有若干后继, 称“树根”
- ★ 有若干点仅有一个前驱, 没有后继, 称“树叶”
- ★ 有若干点仅有一个前驱, 有若干后继, 称“树枝”

树的递归描述：

- ① 树是有 n ($n \geq 0$) 个结点组成的有限集合, 当 $n=0$ 时称为空树, 否则称为非空树
 - ② 在任一非空树中, 有且仅有一个称为根(root)的结点, 其余结点分属 m 颗 ($m \geq 0$) 互不相交的子树 (称为根的子树)
 - ③ 每颗子树同样又是一棵树
- ★ 一棵树中, 每个结点被定义为它的子树的根结点的前驱, 而它的每颗子树的根结点就成为它的后继
 - ★ 每个结点都可以理解为子树的根结点

树的非递归描述：

- ① 树是有 n ($n \geq 0$) 个结点组成的有限集合, 当 $n=0$ 时称为空树, 否则称为非空树
- ② 对任一非空树, 有且仅有一个结点没有前驱, 有 m ($m \geq 0$) 个后继, 称为树根
- ③ 其它结点仅有一个前驱, 有 m ($m \geq 0$) 个后继, 称为树枝 ($m > 0$) 和树叶 ($m = 0$)



§ 6. 树和二叉树

6.1. 树的定义和基本术语

6.1.2. 树的形式定义 (P. 118-119)

ADT Tree {

数据对象D: $D = \{d_i \mid 1 \leq i \leq n, n \geq 0, d_i \in D_0\}$

数据关系R: $R = \{H\}$

$H = \{\langle d_i, d_j \rangle \mid 1 \leq i, j \leq n, n \geq 0, i \neq j, d_i, d_j \in D\}$

① 若 $D = \emptyset$, 则为空树

② 若D中只有一个元素, 则该元素为root, 且 $R = \emptyset$

③ 若D中元素为一个以上, 即 $D - \{\text{root}\} \neq \emptyset$, 则R理解为
存在 $D - \{\text{root}\}$ 的一个划分 D_1, D_2, \dots, D_m ($m > 0$)

★ 对任 j, k ($1 \leq j, k \leq m$ 且 $j \neq k$), 有 $D_j \cap D_k = \emptyset$ (互不相交)

★ 对任 i ($1 \leq i \leq m$), 唯一有元素 $x_i \in D_i$, 有 $\langle \text{root}, x_i \rangle \in H$

④ 对应 $D - \{\text{root}\} \neq \emptyset$ 的划分, $H - \{\langle \text{root}, x_1 \rangle, \langle \text{root}, x_2 \rangle, \dots, \langle \text{root}, x_n \rangle\}$
有唯一的一个划分 H_1, H_2, \dots, H_m ($m > 0$)

★ 对任 j, k (若 $1 \leq j, k \leq m$ 且 $j \neq k$), 有 $H_j \cap H_k = \emptyset$ (互不相交)

★ 对任 i ($1 \leq i \leq m$), H_i 为 D_i 上的二元关系,

即 $(D_i, \{H_i\})$ 是一颗符合本定义棵树, 称为根root的子树

基本操作P:

共15种

} ADT Tree

例：如图所示的树，请填写相关信息

$D = \{ \underline{\hspace{2cm}} \}$

$H = \{ \underline{\hspace{4cm}} \}$

root=a

$D - \{\text{root}\} = \{ \underline{\hspace{2cm}} \}$

存在划分 $D_1 = \{ \underline{\hspace{2cm}} \}$ 唯一元素 $\underline{\hspace{2cm}} \in D_1$ $\langle a, \underline{\hspace{2cm}} \rangle \in H$

$D_2 = \{ \underline{\hspace{2cm}} \}$ 唯一元素 $\underline{\hspace{2cm}} \in D_2$ $\langle a, \underline{\hspace{2cm}} \rangle \in H$

$D_3 = \{ \underline{\hspace{2cm}} \}$ 唯一元素 $\underline{\hspace{2cm}} \in D_3$ $\langle a, \underline{\hspace{2cm}} \rangle \in H$

且 $D_1 \cap D_2 = \emptyset$ $D_2 \cap D_3 = \emptyset$ $D_1 \cap D_3 = \emptyset$

对应 $D - \{\text{root}\}$, $H - \{ \underline{\hspace{2cm}} \}$

$= \{ \underline{\hspace{4cm}} \}$

存在划分 $H_1 = \{ \underline{\hspace{2cm}} \}$

H_1 是 $D_1 = \{ \underline{\hspace{2cm}} \}$ 上的二元关系 $\{D_1, H_1\}$ 是树

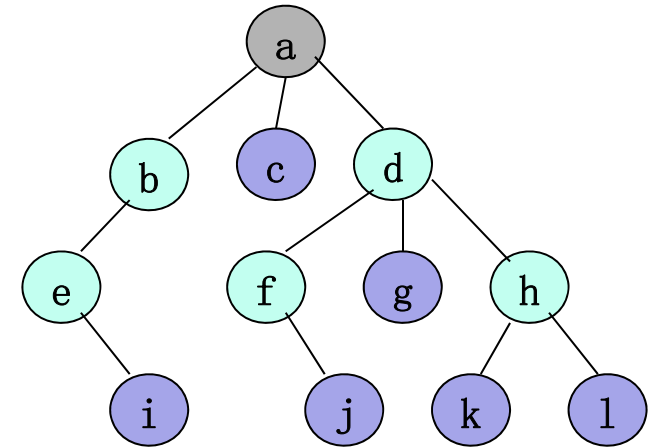
$H_2 = \{ \emptyset \}$

H_2 是 $D_2 = \{ \underline{\hspace{2cm}} \}$ 上的二元关系 $\{D_2, H_2\}$ 是树

$H_3 = \{ \underline{\hspace{2cm}} \}$

H_3 是 $D_3 = \{ \underline{\hspace{2cm}} \}$ 上的二元关系 $\{D_3, H_3\}$ 是树

且 $H_1 \cap H_2 = \emptyset$ $H_2 \cap H_3 = \emptyset$ $H_1 \cap H_3 = \emptyset$



§ 6. 树和二叉树

6.1. 树的定义和基本术语

6.1.1. 树的基本特征及描述

6.1.2. 树的形式定义

6.1.3. 树的表示方法

★ 图示法结点表示元素，连线表示关系 (无箭头，隐含从上至下)

★ 二元组表示法

TREE = (D, R)

D = {数据元素的集合}

R = {序偶集合}

$T = (D, R)$

$D = \{A, B, C, D, E, F, G, H, I, J, K, L\}$

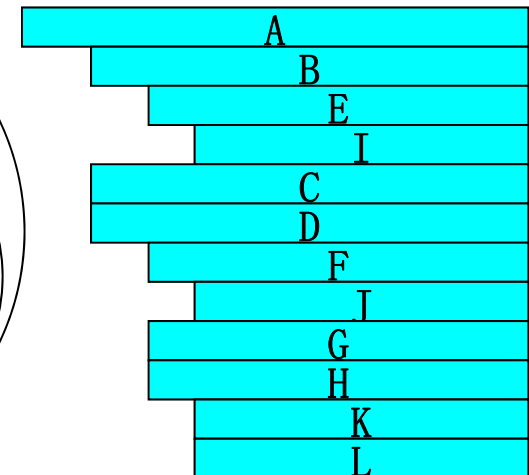
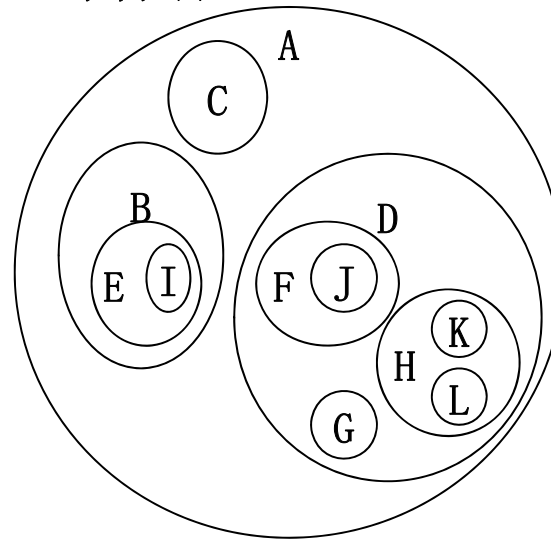
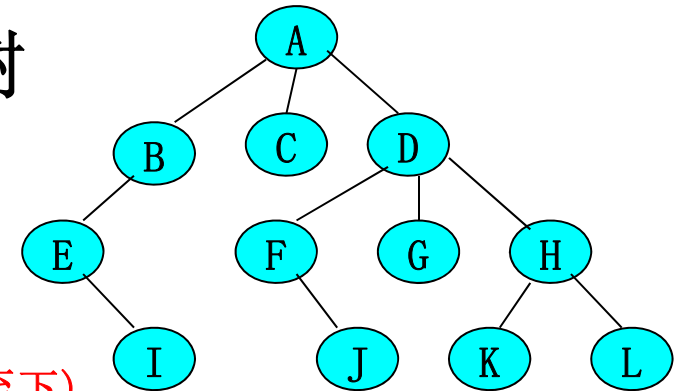
$R = \{\langle A, B \rangle, \langle A, C \rangle, \langle A, D \rangle, \langle B, E \rangle, \langle E, I \rangle, \langle D, F \rangle, \langle F, J \rangle, \langle D, G \rangle, \langle D, H \rangle, \langle H, K \rangle, \langle H, L \rangle\}$

★ 嵌套集合表示法：每个圆表示一棵树，包含一个元素(根)及若干内嵌圆

★ 凹入表示法：每个根结点对应一个矩形，树根在上(长)，子树根在下(短)

★ 广义表表示法：(根节点(子树1, ..., 子树m))

(A(B(E(I)), C, D(F(J), G, H(K, L))))



§ 6. 树和二叉树

6.1. 树的定义和基本术语

6.1.4. 树的基本术语

- ★ 结点的度 : 每个结点拥有的子树数目
- ★ 树的度 : 树中所有结点的度的最大值
- ★ 叶子(终端结点) : 度为0的结点
- ★ 分支(非终端结点) : 度不为0的结点
- ★ 孩子结点 : 某结点的子树的根 (某结点的直接后继)
- ★ 双亲结点 : 某结点的直接前驱
- ★ 兄弟结点 : 同一双亲结点的孩子之间互称兄弟结点
- ★ 子孙 : 某结点所有子树中的结点
- ★ 祖先 : 从根结点到该结点的路径上经过的所有结点
- ★ 堂兄弟 : 双亲为兄弟结点的结点
- ★ 结点的层数 : 从树根开始的层次数
- ★ 树的深度(高度) : 结点层次的最大值
- ★ 有序树 : 结点的子树是左右有序的, 不能互换
(左: 第一个孩子, 右: 最后一个孩子)
- ★ 无序树 : 结点的子树可以互换
- ★ 树的遍历 : 每个结点均访问到且每个结点只能访问一次
- ★ 森林 : m 棵互不相交的树的集合

树的另一种定义:

$Tree = (root, F)$

root为根结点

$F = (T_1, T_2, \dots, T_m)$ 为 m 颗子树组成的森林

其中 $T_i = (r_i, F_i)$ 称为根root的第 i 颗子树

依照下图给出答案:

树的度为_____

树的深度为_____

终端结点的个数_____

单分支结点的个数_____

双分支结点的个数_____

三分支结点的个数_____

H的双亲结点_____

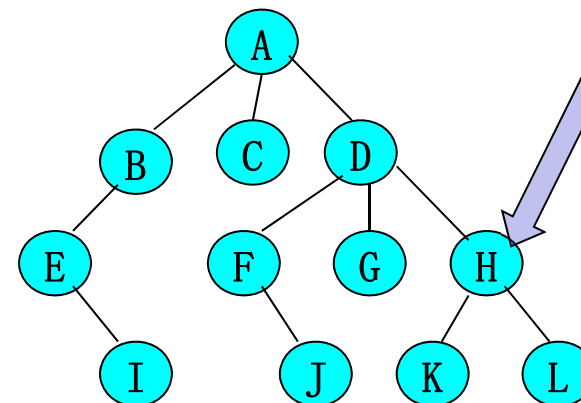
H的孩子结点____和____

B和D互为_____

E和F互为_____

F是A的_____

D是L的_____



§ 6. 树和二叉树

6. 1. 树的定义和基本术语

6. 1. 5. 树的性质 (补充)

性质1: 树中的结点数 = 所有结点的度数+1

证明: 除根外, 每个结点仅有一个前驱, 即每个结点与指向的分支一一对应

=> 除树根外的结点数 = 所有结点的分支数(度数)

=> 成立

性质2: 度为 k ($k \geq 1$) 的树中第 i 层 ($i \geq 1$) 上最多有 k^{i-1} 个结点

证明: $i=1$ 时, $k^{i-1}=k^0=1$ (根) => 成立

设 $i-1$ 时成立 ($i > 1$), 即第 $i-1$ 层最多有 k^{i-2} 个结点

当 i 时, 第 $i-1$ 层上每个结点最多有 k 个后继

=> $k^{i-2} * k = k^{i-1}$

=> 成立

性质3: 深度为 h ($h \geq 1$) 的 k 叉树 ($k > 1$), 至多有 $\frac{k^h-1}{k-1}$ 个结点

证明: 若 k 叉树, 每层都达到最多结点, 则最多为

$$\sum_{i=1}^h k^{i-1} = k^0 + k^1 + \dots + k^{h-1} = \frac{k^h-1}{k-1}$$

§ 6. 树和二叉树

6.1. 树的定义和基本术语

6.1.5. 树的性质 (补充)

性质4: 具有 $n(n>0)$ 个结点的 k 叉树($k>1$)的最小深度为 $\lceil \log_k^{n(k-1)+1} \rceil$ 或 $\lfloor \log_k^{n(k-1)} \rfloor + 1$

证明: 设深度为 h , n 个结点要具有最小深度, 则前 $h-1$ 层必须满, 第 h 层为 $1 - k^{h-1}$ 个

$$\Rightarrow \frac{k^{h-1}-1}{k-1} < n \leq \frac{k^h-1}{k-1}$$

$$\Rightarrow k^{h-1}-1 < n(k-1) \leq k^h-1$$

$$\Rightarrow k^{h-1} \leq n(k-1) < k^h \quad (\text{都是整数})$$

$$\Rightarrow h-1 \leq \log_k^{n(k-1)} < h$$

$$\Rightarrow \log_k^{n(k-1)} < h \leq \log_k^{n(k-1)+1}$$

$$\Rightarrow \log_k^{n(k-1)+1} \leq h \leq \log_k^{n(k-1)+1}$$

$$\Rightarrow h \text{只能是整数, } h = \lceil \log_k^{n(k-1)+1} \rceil \text{ 或 } h = \lfloor \log_k^{n(k-1)} \rfloor + 1$$

§ 6. 树和二叉树

6.2. 二叉树

6.2.1. 二叉树的定义

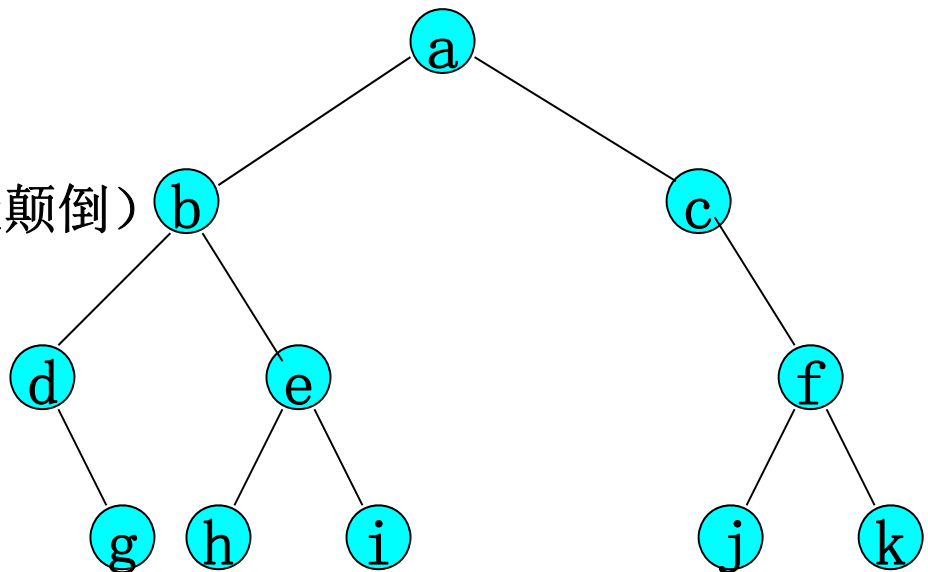
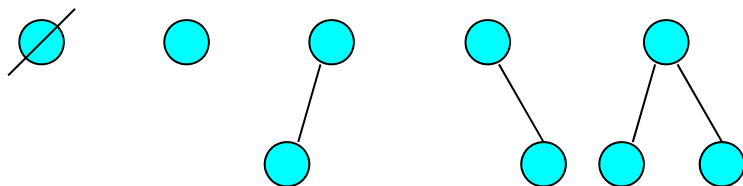
★ 二叉树的递归定义

- ① 二叉树是有 n ($n \geq 0$) 个结点组成的有限集合，当 $n=0$ 时称为空树，否则称为非空树
- ② 在任一非空二叉树中，有且仅有一个称为根(root)的结点
- ③ 二叉树的其余结点分属2颗互不相交的子树 (从左到右分别称为左子树和右子树)，每颗子树同样又是一棵二叉树

★ 二叉树的基本特征

- 树的度为2
- 有序树（子树有左右之分，不能颠倒）

★ 二叉树的五种基本形态



§ 6. 树和二叉树

6.2. 二叉树

6.2.1. 二叉树的定义

★ 二叉树的形式定义 (P. 121-123)

ADT BinaryTree {

数据对象D: $D = \{d_i \mid 1 \leq i \leq n, n \geq 0, d_i \in D_0\}$

数据关系R: $R = \{H\}$

$H = \{\langle d_i, d_j \rangle \mid 1 \leq i, j \leq n, n \geq 0, i \neq j, d_i, d_j \in D\}$

★ 若 $D = \emptyset$, 则 $R = \emptyset$, BinaryTree为空树

★ 若 $D \neq \emptyset$, 则 $R = \{H\}$, H是如下的二元关系:

① 在D中存在唯一的称为根的数据元素root, 它在关系H下无前驱

② 若 $D - \{\text{root}\} \neq \emptyset$, 则存在 $D - \{\text{root}\} = \{D_1, D_r\}$, 且 $D_1 \cap D_r = \emptyset$

③ 若 $D_1 \neq \emptyset$, 则 D_1 中存在唯一的元素 x_1 , $\langle \text{root}, x_1 \rangle \in H$, 且存在 D_1 上的关系 $H_1 \subset H$;
若 $D_r \neq \emptyset$, 则 D_r 中存在唯一的元素 x_r , $\langle \text{root}, x_r \rangle \in H$, 且存在 D_r 上的关系 $H_r \subset H$;

$H = \{\langle \text{root}, x_1 \rangle, \langle \text{root}, x_r \rangle, H_1, H_r\}$

④ $(D_1, \{H_1\})$ 是一颗符合本定义的二叉树, 称为根的左子树,

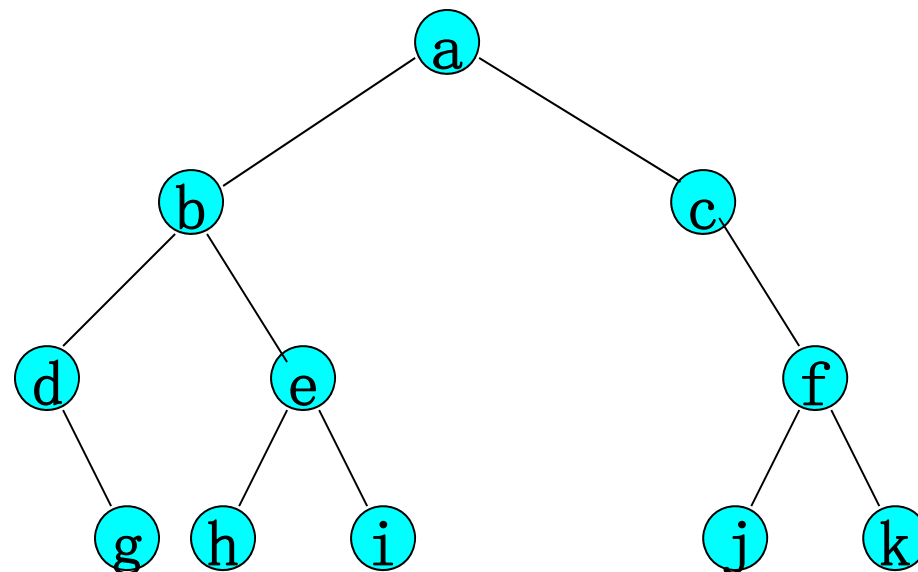
$(D_r, \{H_r\})$ 是一颗符合本定义的二叉树, 称为根的右子树,

基本操作P:

共20种

} ADT binaryTree

填空完成:



$D = \{ \underline{\hspace{2cm}} \}$

$H = \{ \underline{\hspace{4cm}} \}$

root=a

$D - \{\text{root}\} = \{ \underline{\hspace{2cm}} \}$

存在划分 $D_1 = \{ \underline{\hspace{2cm}} \}$ 唯一元素 $\underline{\hspace{1cm}} \in D_1$ $\langle a, \underline{\hspace{1cm}} \rangle \in H$

存在 D_1 上的关系 $H_1 = \{ \underline{\hspace{2cm}} \} \subset H$

$D_r = \{ \underline{\hspace{2cm}} \}$ 唯一元素 $\underline{\hspace{1cm}} \in D_r$ $\langle a, \underline{\hspace{1cm}} \rangle \in H$

存在 D_r 上的关系 $H_r = \{ \underline{\hspace{2cm}} \} \subset H$

且 $D_1 \cap D_r = \emptyset$, $H = \{ \langle a, \underline{\hspace{1cm}} \rangle, \langle a, \underline{\hspace{1cm}} \rangle, \{H_1\}, \{H_r\} \}$

$(D_1, \{H_1\})$ / $(D_r, \{H_r\})$ 都是一颗二叉树

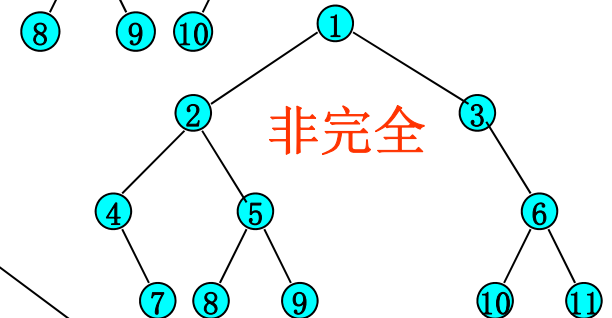
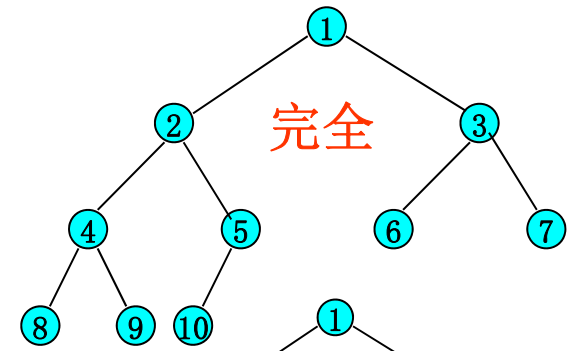
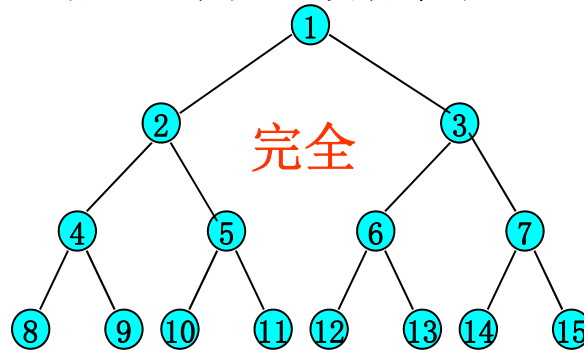
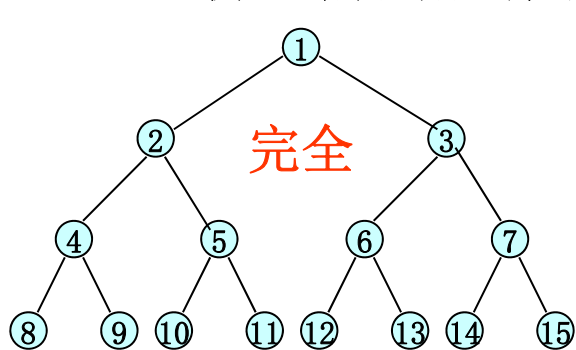
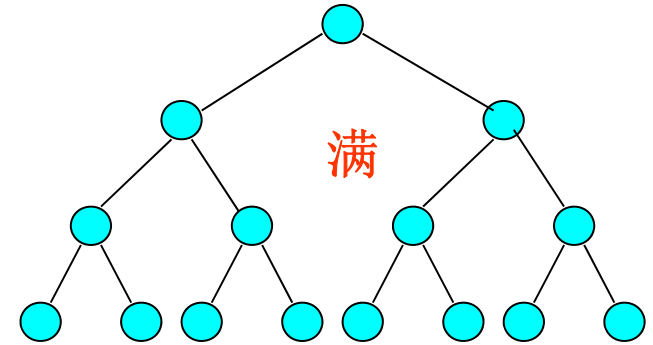
§ 6. 树和二叉树

6.2. 二叉树

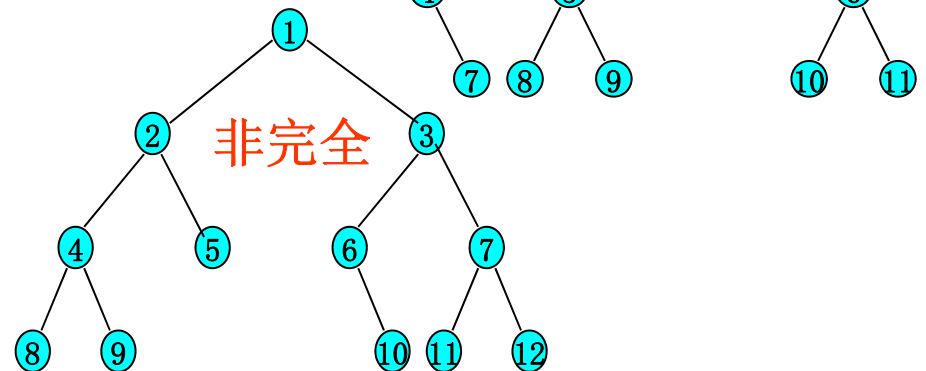
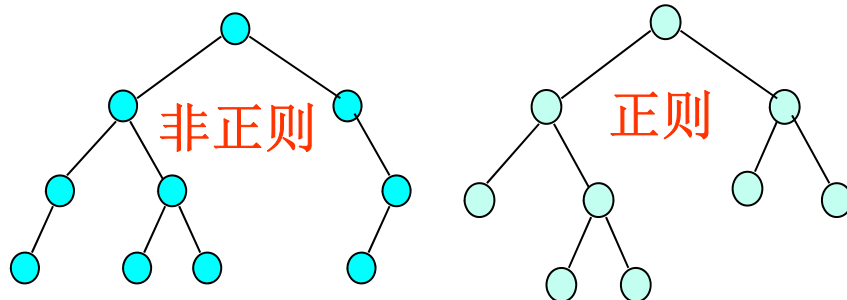
6.2.1. 二叉树的定义

★ 几种特殊形态的二叉树

- 满二叉树：二叉树的每一层都是满的 (具有最大结点数)
- 完全二叉树：对满二叉树进行编号，规定顺序从上到下，从左到右，当一个具有 n 个结点的二叉树从1- n 进行编号并与满二叉树1- n 的编号一一对应时，称为完全二叉树
 - ◆ 除最后一层外，其余层均满
 - ◆ 最后一层或者是满的，或者右边缺少连续若干结点



- 正则二叉树：没有度为1的结点的二叉树



§ 6. 树和二叉树

6.2. 二叉树

6.2.2. 二叉树的性质

性质1: 在二叉树的第 i 层至多有 2^{i-1} 个结点 (参考树的性质2)

证明: $i=1$ 时, $2^{i-1}=2^0=1$ (根) \Rightarrow 成立

设 $i-1$ 时成立 ($i>1$), 即第 $i-1$ 层最多有 2^{i-2} 个结点

当 i 时, 第 $i-1$ 层上每个结点最多有2个后继

$$\Rightarrow 2^{i-2} \times 2 = 2^{i-1}$$

\Rightarrow 成立

性质2: 深度为 k 的二叉树至多有 2^k-1 个结点 (参考树的性质3)

证明: $\sum_{i=1}^k (\text{第 } i \text{ 层最大结点数}) = \sum_{i=1}^k 2^{i-1} = 2^k - 1$

性质3: 对任何一颗非空二叉树 T , 如果其终端结点数为 n_0 , 度为2的结点数为 n_2 , 则 $n_0 = n_2 + 1$

证明: ① 设度为1的结点为 n_1 , 则总结点数 $n = n_0 + n_1 + n_2$

② 除根外, 每个结点一个前驱, 设 B 为分支, 则 $B = n - 1$ (仅根没有前驱)

③ 度1的结点1个后继, 度2的结点2个后继, 则 $n_1 + 2n_2 = B = n - 1$

④ $n_1 + 2n_2 = n_0 + n_1 + n_2 - 1$, 得 $n_0 = n_2 + 1$

性质4: n 个结点的完全二叉树的深度为 $\lfloor \log_2 n \rfloor + 1$ (参考树的性质4)

证明: 假设 n 个结点的完全二叉树深度为 k , 则前 $k-1$ 层必是满的, 第 k 层从左开始 $1 - 2^{k-1}$ 个结点

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

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

$$\Rightarrow k-1 \leq \log_2 n < k$$

$$\Rightarrow k \text{ 为整数} \Rightarrow k = \lfloor \log_2 n \rfloor + 1$$

性质5: n 个结点的完全二叉树, 按从上到下, 从左到右的顺序进行编号, 对任一 i ($1 \leq i \leq n$), 则:

- ① 如果 $i=1$, 则 i 为二叉树的根, 无双亲; 如果 $i>1$, 则双亲为 $\lfloor i/2 \rfloor$
- ② 如果 $2i>n$, 则结点 i 无左孩子(i 为叶子结点), 否则($i \leq \frac{n}{2}$ 或 $2i \leq n$) i 有左孩子, 编号为 $2i$
- ③ 如果 $2i+1>n$, 则结点 i 无右孩子(不一定叶子), 否则($i \leq \frac{n-1}{2}$ 或 $2i+1 \leq n$) i 有右孩子, 编号为 $2i+1$

补充结论:

- ④ 若 i 为奇数且 $i>1$, 则结点 i 的左兄弟为 $i-1$
- ⑤ 若 i 为偶数且 $i<n$, 则结点 i 的右兄弟为 $i+1$

证明: $i=1$ 时, i 为根 若有左孩子, 为 2 ($n \geq 2$)

若有右孩子, 为 3 ($n \geq 3$)

\Rightarrow ②③成立

$i>1$ 时, 设 i 为第 j 层 ($1 \leq j \leq \lfloor \log_2 n \rfloor$) (最后 $\lfloor \log_2 n \rfloor + 1$ 层全部叶子, 不讨论) 上的

第 k 个 ($1 \leq k \leq 2^{j-1}$) 结点, 前 $j-1$ 层共有 $2^0 + 2^1 + \dots + 2^{j-2} = 2^{j-1} - 1$ 个结点, 编号 $1 \sim 2^{j-1} - 1$

\Rightarrow 第 j 层的第1个结点编号为 2^{j-1} , 第 k 个编号为 $2^{j-1} + (k-1)$, 即 $i = 2^{j-1} + (k-1)$

若 i 有左、右孩子, 则必在 $j+1$ 层上, 第 j 层的前 $k-1$ 个结点在第 $j+1$ 层共有 $2(k-1)$ 个孩子

\Rightarrow 第 k 个结点的左孩子为第 $j+1$ 层的第 $2*(k-1)+1$ 个结点

第 k 个结点的右孩子为第 $j+1$ 层的第 $2*(k-1)+2$ 个结点

第 $j+1$ 层上第1个结点的编号为 2^j

\Rightarrow 左孩子为 $2^j + (2*(k-1)+1) - 1 = 2*(2^{j-1} + (k-1)) = 2i$

右孩子为 $2^j + (2*(k-1)+2) - 1 = 2*(2^{j-1} + (k-1)) + 1 = 2i+1$

\Rightarrow ②③成立 \Rightarrow ①成立

§ 6. 树和二叉树

6.2. 二叉树

6.2.3. 二叉树的存储结构

6.2.3.1. 顺序存储结构

/ P.126 定义 */*

```
#define MAX_TREE_SIZE 100
#define UNDEFINE_VALUE 0 //此句增加
typedef ElemType SqBiTree[MAX_TREE_SIZE];
SqBiTree bt;
```

- ★ 将二叉树按完全二叉树形式进行编号，依次存放在一维数组的相应下标中
- ★ 用数组元素中不会出现的值表示没有对应结点
- ★ 为了对应完全二叉树，下标从[1]开始，[0]不用

换算关系：

某结点 ($2 \leq i \leq n$)，父结点： $i/2$

左孩子： $2i$ (若有)

右孩子： $2i+1$ (若有) *不再进行详细的讨论*

- ★ 下标直接为 $[i]$ 、 $[i/2]$ 、 $[2*i]$ 、 $[2*i+1]$

适应性：

完全二叉树：适合，换算容易， n 个结点，数组 $n+1$

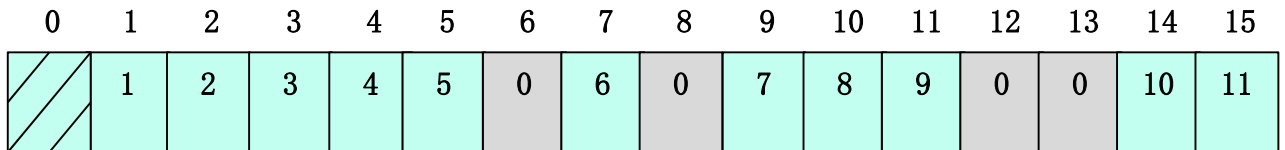
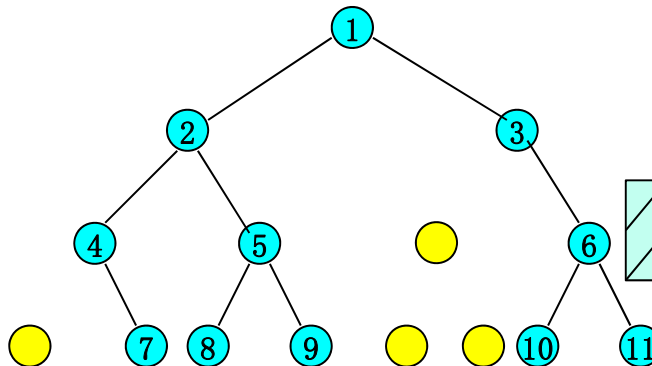
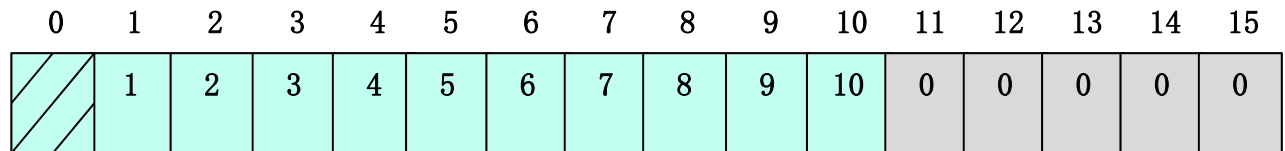
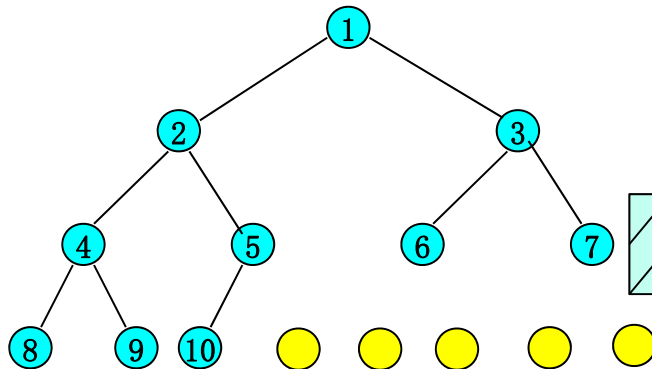
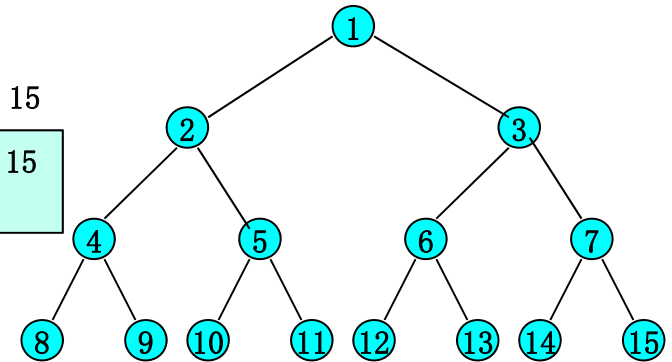
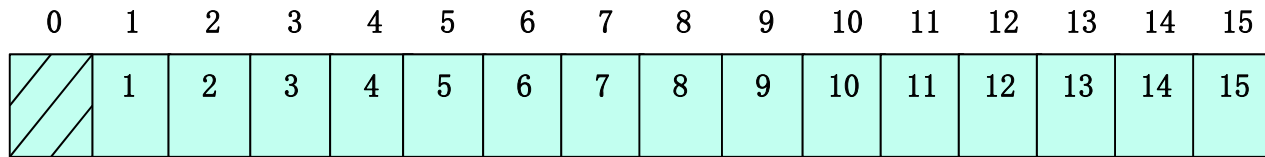
一般二叉树：空间浪费， n 个结点，数组最大 2^n

§ 6. 树和二叉树

6.2. 二叉树

6.2.3. 二叉树的存储结构

6.2.3.1. 顺序存储结构

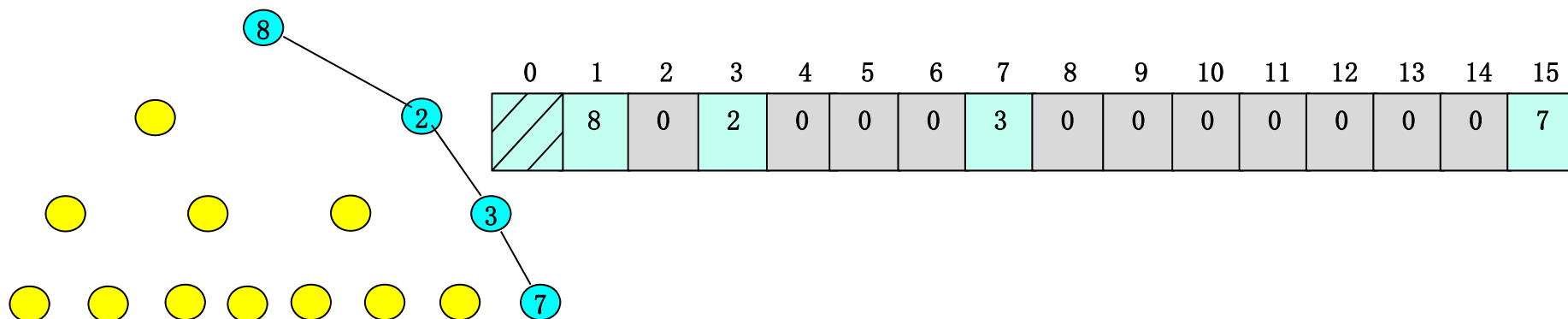
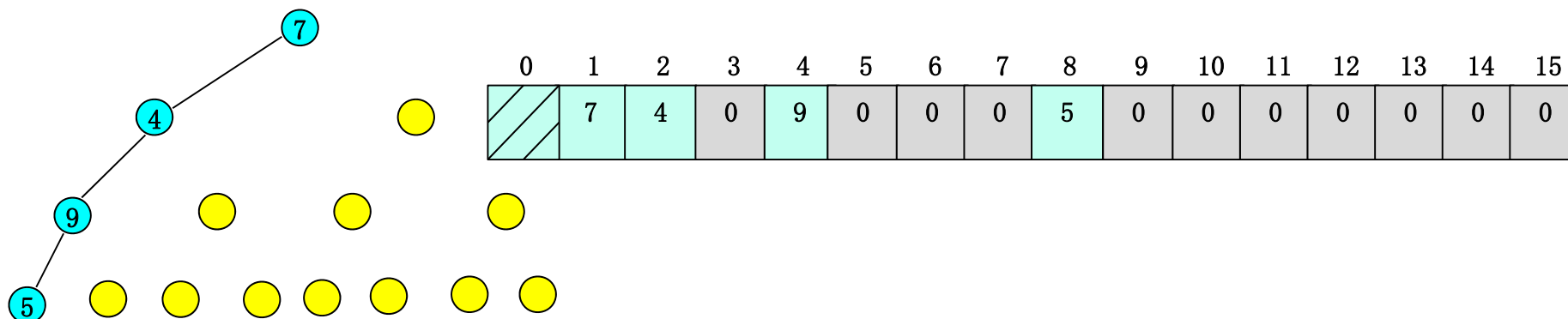


§ 6. 树和二叉树

6.2. 二叉树

6.2.3. 二叉树的存储结构

6.2.3.1. 顺序存储结构



§ 6. 树和二叉树

6.2. 二叉树

6.2.3. 二叉树的存储结构

6.2.3.2. 链式存储结构

★ 二叉链表

★ 三叉链表

/* P.127 定义 */

```
typedef struct BiTNode {  
    TElemType data;  
    struct BiTNode *lchild, *rchild;  
    struct BiTNode *parent; //二叉链表无  
} BiTNode, *BiTree
```

★ n 个结点的二叉链表有 $n+1$ 个空指针域

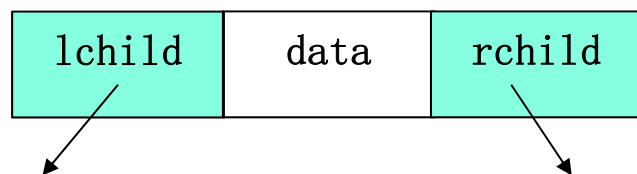
n 个结点， $2n$ 个指针域， $n-1$ 个：被 $n-1$ 个分支占用

$n+1$ 个：空闲

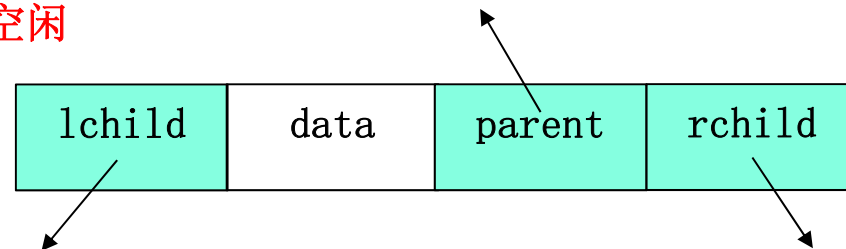
★ n 个结点的三叉链表有 $n+2$ 个空指针域

n 个结点， $3n$ 个指针域，孩子指针： $n+1$ 个空闲

双亲指针：仅1个空闲



二叉链表



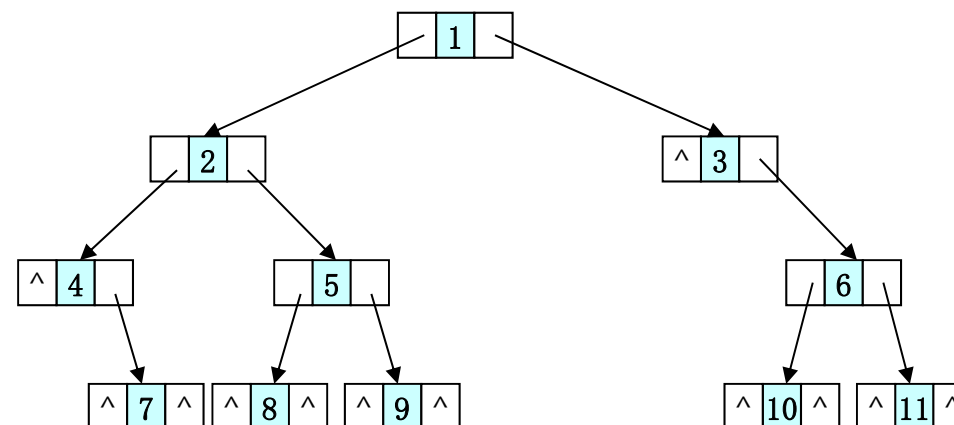
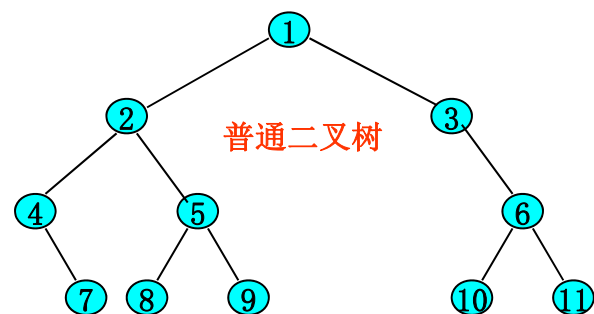
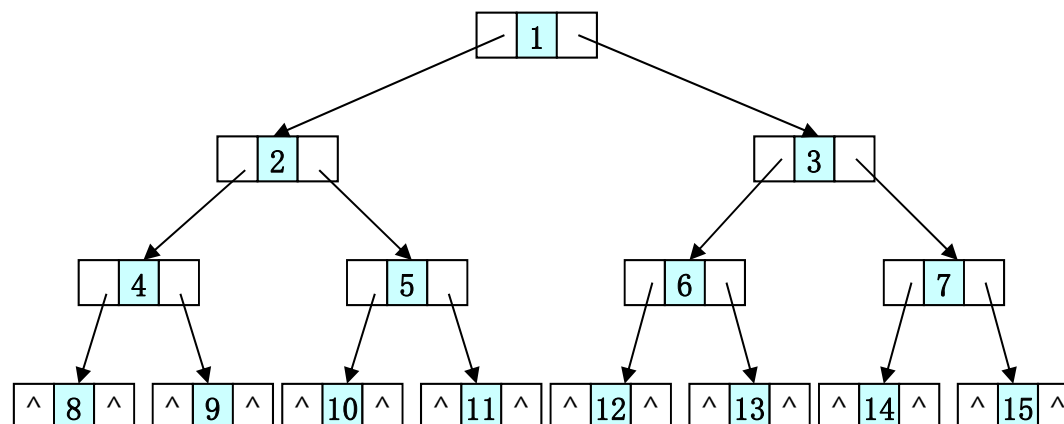
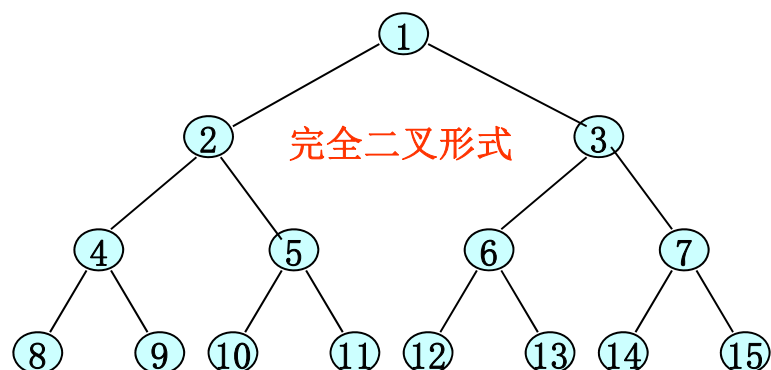
三叉链表

§ 6. 树和二叉树

6.2. 二叉树

6.2.3. 二叉树的存储结构

6.2.3.2. 链式存储结构

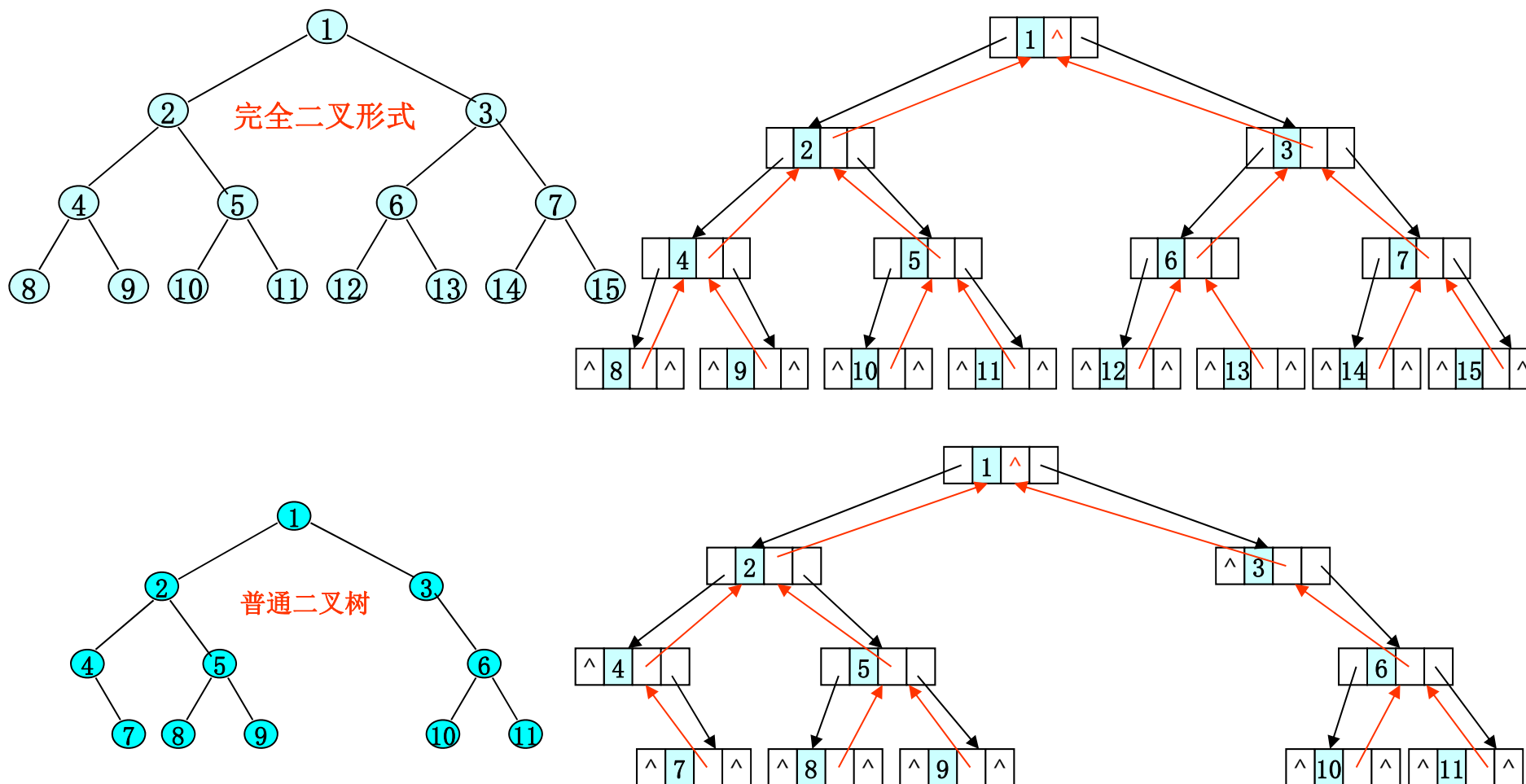


§ 6. 树和二叉树

6.2. 二叉树

6.2.3. 二叉树的存储结构

6.2.3.2. 链式存储结构



§ 6. 树和二叉树

6.2. 二叉树

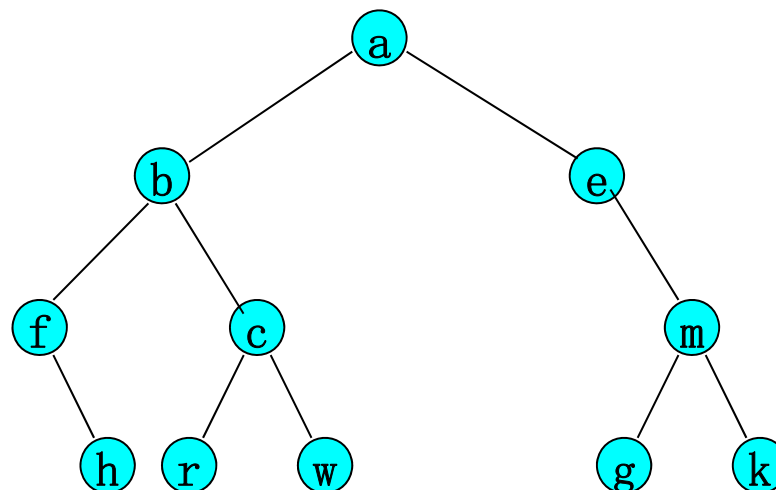
6.2.3. 二叉树的存储结构

6.2.3.2. 链式存储结构

★ 静态二叉链表和三叉链表(补充)

和第二章静态链表相似，用数组模拟链表

	data	lchild	rchild	parent
0	a	1	2	-1
1	b	3	4	0
2	e	-1	5	0
3	f	-1	6	1
4	c	7	8	1
5	m	9	10	2
6	h	-1	-1	3
7	r	-1	-1	4
8	w	-1	-1	4
9	g	-1	-1	5
10	k	-1	-1	5



- lchild/rchild/parent均为整数，data按需定义
- 二叉链表无parent域
- 下标[0]开始，用-1表示无，也可从[1]开始，用0表示无
- 数据的排放可任意(本例按层次)

§ 6. 树和二叉树

6.3. 遍历二叉树和线索二叉树

6.3.1. 遍历二叉树

★ 遍历的含义

① 每个结点均被访问到

② 每个结点只能被访问一次

● 对于线性结构比较简单，对于非线性结构，则需要寻找一种规律

★ 二叉树的遍历方案(递归)

二叉树 { 根 D - 表示访问根的操作
 左子树 L - 表示遍历左子树的操作
 右子树 R - 表示遍历右子树的操作

● 排列组合可得六种方案

DLR DRL : D最先被访问

LDR RDL : D在中间被访问

LRD RLD : D在最后被访问

● 若规定必须先左后右，则得到三种方案

DLR : 先序遍历(先根遍历)

LDR : 中序遍历(中根遍历)

LRD : 后序遍历(后根遍历)

● 还有按层次遍历的方法

§ 6. 树和二叉树

6.3. 遍历二叉树和线索二叉树

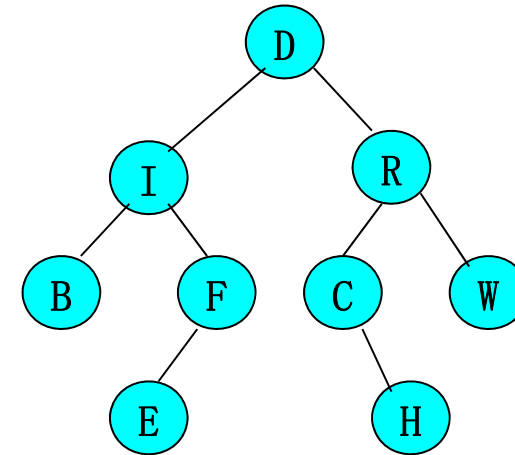
6.3.1. 遍历二叉树

例：二叉树如图，按层次遍历得到的结点序列为_____

按先序遍历得到的结点序列为_____

按中序遍历得到的结点序列为_____

按后序遍历得到的结点序列为_____



§ 6. 树和二叉树

6.3. 遍历二叉树和线索二叉树

6.3.1. 遍历二叉树

★ 表达式的二叉树表示

以二叉树表示表达式的递归定义：

- ① 若表达式为数或简单变量，则相应二叉树中仅有一个根结点，其数据域存放表达式信息；
- ② 若表达式=(第一操作数)(运算符)(第二操作数)，则相应的二叉树中以左子树表示第一操作数；右子树表示第二操作数；根结点的数据域存放运算符
(若为单目运算符，则左子树为空)
- ③ 操作数本身又是表达式

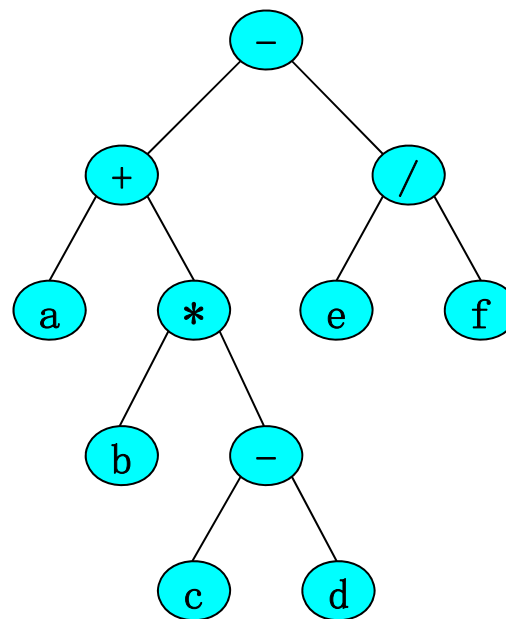
例： $a+b*(c-d)-e/f$

(操作数为树叶, 运算符为树枝、树根)

先序遍历：前缀表达式（波兰式） $-+a*b-cd/ef$

中序遍历：中缀表达式 $a+b*c-d-e/f$

后序遍历：后缀表达式（逆波兰式） $abcd-*+ef/-$

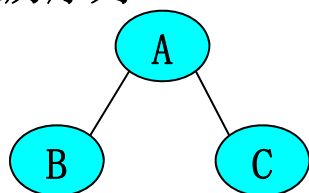


§ 6. 树和二叉树

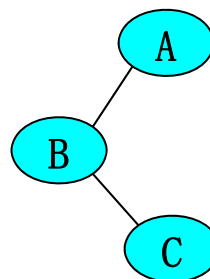
6.3. 遍历二叉树和线索二叉树

6.3.1. 遍历二叉树

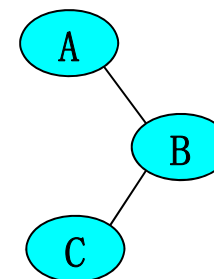
- ★ 遍历的含义
- ★ 二叉树的遍历方案(递归)
- ★ 表达式的二叉树表示
- ★ 通过遍历序列复原二叉树
 - 单个遍历序列



先: ABC
中: BAC
后: BCA



先: ABC
中: BCA
后: CBA



先: ABC
中: ACB
后: CBA

=> 结论: 不可行

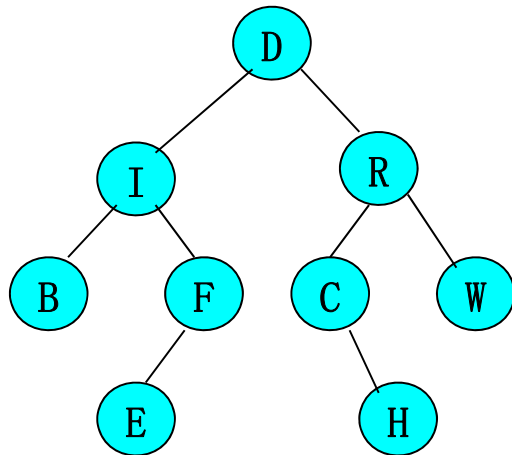
§ 6. 树和二叉树

6.3.1. 遍历二叉树

★ 通过遍历序列复原二叉树

● 多个遍历序列

中序和其它序列
的组合可行



先序+后序：不行
层次+先序：不行
层次+后序：不行

先序+中序：先序第一个结点是根，对应中序位置，
左边是左子树，右边是右子树(递归)

~~层: D I R B F C W E H~~
先: D I B F E R C H W
中: B I E F D C H R W
~~后: B E F I H C W R D~~

中序+后序：后序最后一个结点是根，对应中序位置，
左边是左子树，右边是右子树(递归)

~~层: D I R B F C W E H~~
~~先: D I B F E R C H W~~
中: B I E F D C H R W
后: B E F I H C W R D

层次+中序：层次第一个结点是根，对应中序，分开左右子树，某子树的
中序序列，对应层次顺序的第一个结点为子树的根，依次类推

层: D I R B F C W E H
~~先: D I B F E R C H W~~
中: B I E F D C H R W
~~后: B E F I H C W R D~~

§ 6. 树和二叉树

6.3. 遍历二叉树和线索二叉树

6.3.1. 遍历二叉树

- ★ 遍历的含义
- ★ 二叉树的遍历方案(递归)
- ★ 表达式的二叉树表示
- ★ 通过遍历序列复原二叉树
- ★ 二叉树遍历的递归实现

● 二叉树先序遍历的递归实现 (P. 129 算法 6.1)

```
Status PreOrderTraverse(BiTree T, Status (*visit)(TElemType e))
{
    if (T) {
        if ((*visit)(T->data)==OK)
            if (PreOrderTraverse(T->lchild, visit)==OK)
                if (PreOrderTraverse(T->rchild, visit)==OK)
                    return OK;
        return ERROR;
    }
    else //空树返回OK
        return OK;
}
```

考虑到Visit可能返回ERROR, 所以要判断

```
Status PreOrderTraverse(BiTree T, Status (*visit)(TElemType e))
{
    if (T) {
        (*visit)(T->data);
        PreOrderTraverse(T->lchild, visit);
        PreOrderTraverse(T->rchild, visit);
    }
    return OK;
}
```

假设visit就是简单的打印, 则可以简化

```
Status PreOrderTraverse(BiTree T)
{
    if (T) {
        cout << T->data;
        PreOrderTraverse(T->lchild);
        PreOrderTraverse(T->rchild);
    }
    return OK;
}
```

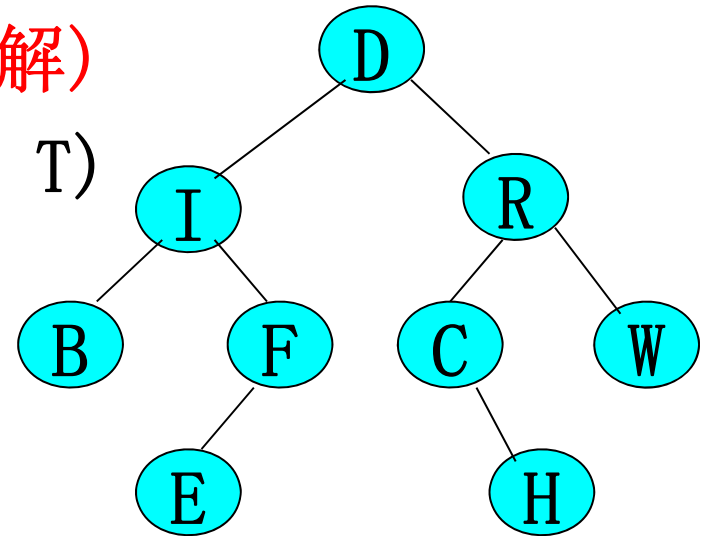
直接打印

```
Status MyVisit(TElemType e)
{
    printf("%d", e);
    return OK;
}
```

调用时:
PreOrderTraverse(T, MyVisit);

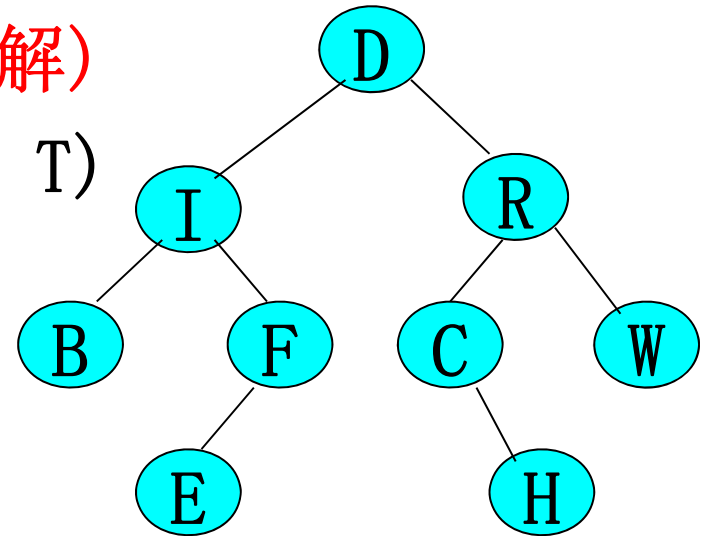
● 二叉树先序遍历的递归实现 (理解)

```
Status PreOrderTraverse(BiTree T)
{
    if (T) {
        cout << T->data;
        PreOrderTraverse(T->lchild);
        PreOrderTraverse(T->rchild);
    }
    return OK;
}
```



● 二叉树先序遍历的递归实现 (理解)

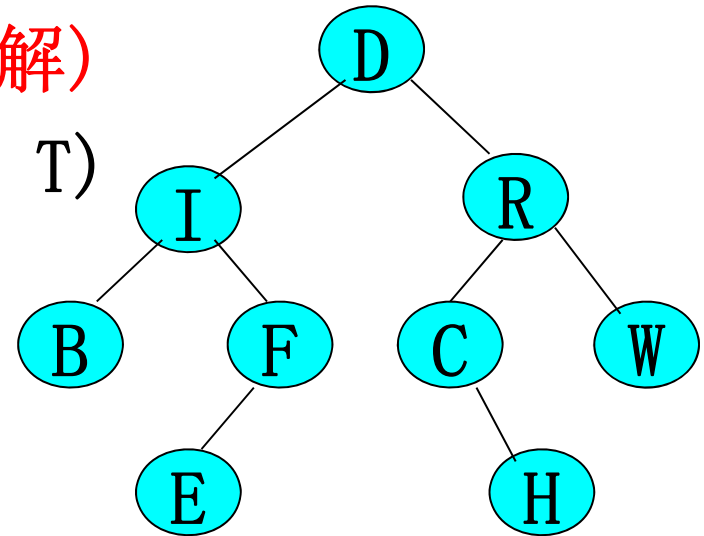
```
Status PreOrderTraverse(BiTree T)
{
    if (T) {
        cout << T->data;
        PreOrderTraverse(T->lchild);
        PreOrderTraverse(T->rchild);
    }
    return OK;
}
```



D

● 二叉树先序遍历的递归实现 (理解)

```
Status PreOrderTraverse(BiTree T)
{
    if (T) {
        cout << T->data;
        PreOrderTraverse(T->lchild);
        PreOrderTraverse(T->rchild);
    }
    return OK;
}
```



D

D	D
---	---

● 二叉树先序遍历的递归实现 (理解)

```
Status PreOrderTraverse(BiTree T)
```

```
{
```

```
    if (T) {
```

```
        cout << T->data;
```

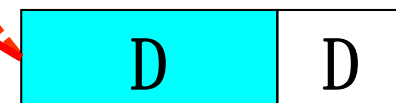
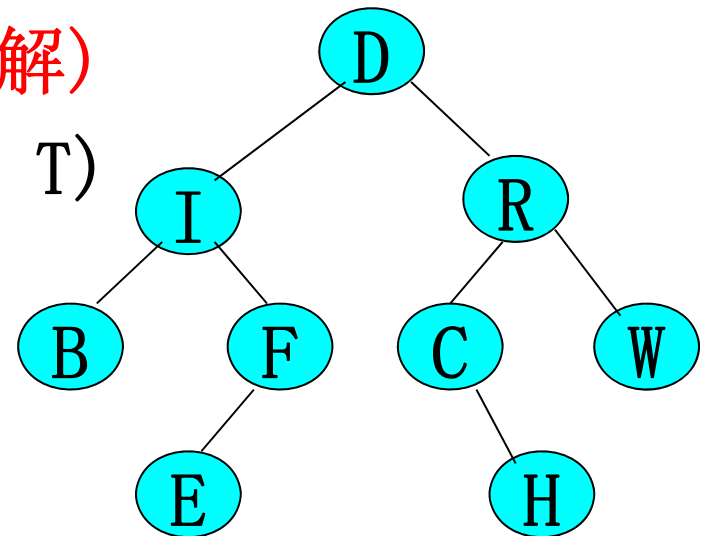
```
        PreOrderTraverse(T->lchild);
```

```
        PreOrderTraverse(T->rchild);
```

```
    }
```

```
    return OK;
```

```
}
```



● 二叉树先序遍历的递归实现 (理解)

```
Status PreOrderTraverse(BiTree T)
```

```
{
```

```
    if (T) {
```

```
        cout << T->data;
```

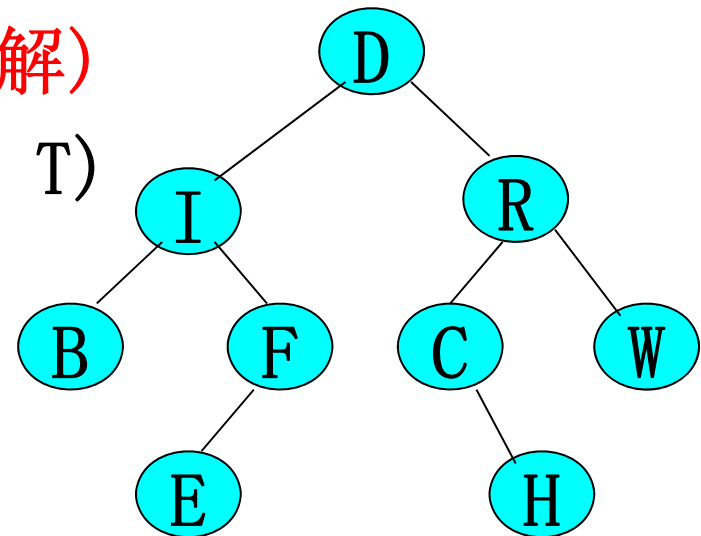
```
        PreOrderTraverse(T->lchild);
```

```
        PreOrderTraverse(T->rchild);
```

```
    }
```

```
    return OK;
```

```
}
```



D I

I	I
D	D

● 二叉树先序遍历的递归实现 (理解)

```
Status PreOrderTraverse(BiTree T)
```

```
{
```

```
    if (T) {
```

```
        cout << T->data;
```

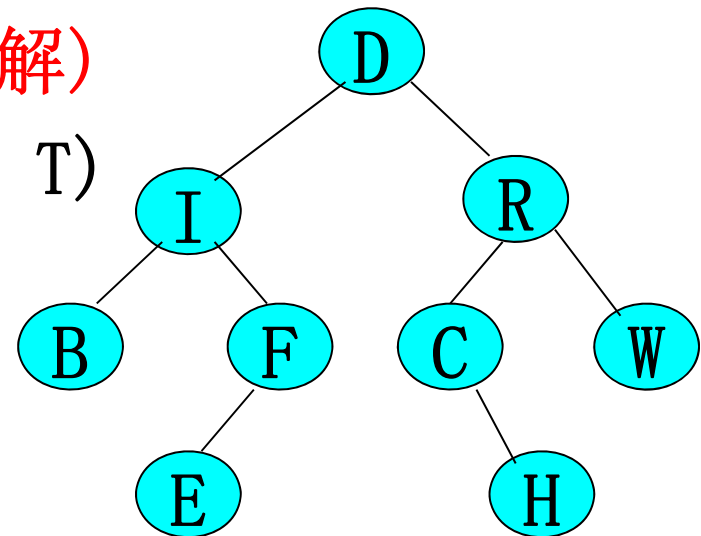
```
        PreOrderTraverse(T->lchild);
```

```
        PreOrderTraverse(T->rchild);
```

```
    }
```

```
    return OK;
```

```
}
```



D I

I	I
D	D

● 二叉树先序遍历的递归实现 (理解)

```
Status PreOrderTraverse(BiTree T)
```

```
{
```

```
    if (T) {
```

```
        cout << T->data;
```

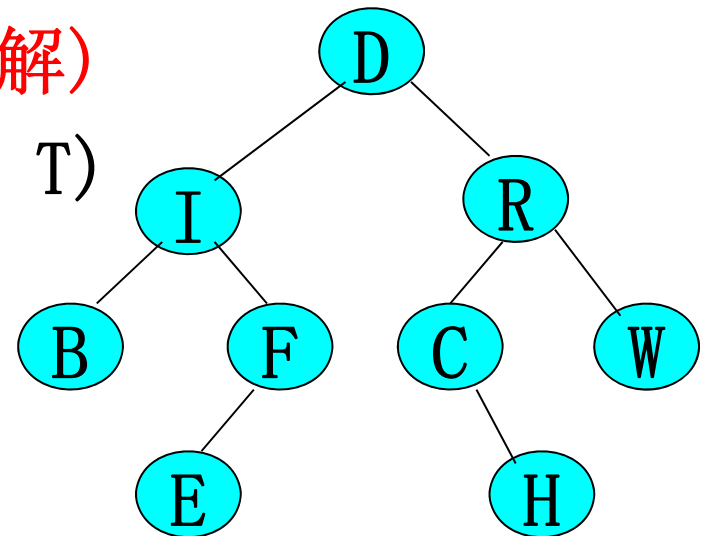
```
        PreOrderTraverse(T->lchild);
```

```
        PreOrderTraverse(T->rchild);
```

```
    }
```

```
    return OK;
```

```
}
```



B	B
I	I
D	D

D I B

● 二叉树先序遍历的递归实现 (理解)

```
Status PreOrderTraverse(BiTree T)
```

```
{
```

```
    if (T) {
```

```
        cout << T->data;
```

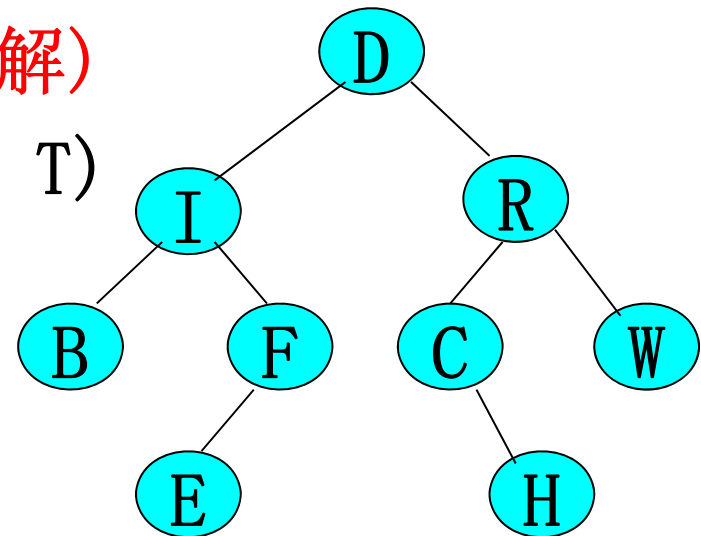
```
        PreOrderTraverse(T->lchild);
```

```
        PreOrderTraverse(T->rchild);
```

```
    }
```

```
    return OK;
```

```
}
```



D I B

B	B
I	I
D	D

● 二叉树先序遍历的递归实现 (理解)

```
Status PreOrderTraverse(BiTree T)
```

```
{
```

```
    if (T) {
```

```
        cout << T->data;
```

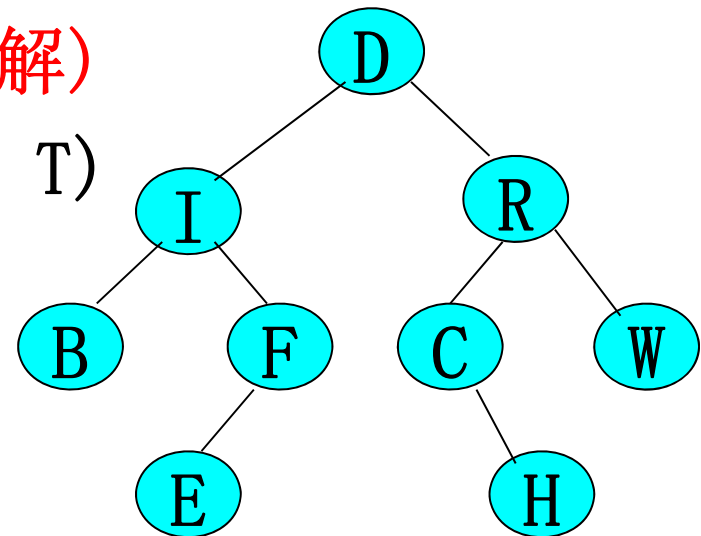
```
        PreOrderTraverse(T->lchild);
```

```
        PreOrderTraverse(T->rchild);
```

```
    }
```

```
    return OK;
```

```
}
```



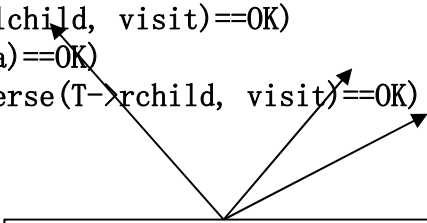
未完，后续省略，自行理解

D I B

NULL	/
B	B
I	I
D	D

● 二叉树中序遍历的递归实现 (P. 129 算法 6.1)

```
Status InOrderTraverse(BiTree T, Status (*visit)(TElemType e))
{
    if (T) {
        if (InOrderTraverse(T->lchild, visit)==OK)
            if ((*visit)(T->data)==OK)
                if (InOrderTraverse(T->rchild, visit)==OK)
                    return OK;
        return ERROR;
    }
    else //空树返回OK
        return OK;
}
```



考虑到Visit可能返回ERROR, 所以要判断

```
Status InOrderTraverse(BiTree T, Status (*visit)(TElemType e))
{
    if (T) {
        InOrderTraverse(T->lchild, visit);
        (*visit)(T->data);
        InOrderTraverse(T->rchild, visit);
    }
    return OK;
}
```

假设visit就是简单的打印, 则可以简化

```
Status InOrderTraverse(BiTree T)
{
    if (T) {
        InOrderTraverse(T->lchild);
        cout << T->data;
        InOrderTraverse(T->rchild);
    }
    return OK;
}
```

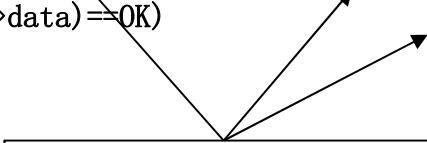
直接打印

```
Status MyVisit(TElemType e)
{
    printf("%d", e);
    return OK;
}
```

调用时:
InOrderTraverse(T, MyVisit);

● 二叉树后序遍历的递归实现 (P. 129 算法 6.1)

```
Status PostOrderTraverse(BiTree T, Status (*visit)(TElemType e))
{
    if (T) {
        if (PostOrderTraverse(T->lchild, visit)==OK)
            if (PostOrderTraverse(T->rchild, visit)==OK)
                if ((*visit)(T->data)==OK)
                    return OK;
        return ERROR;
    }
    else //空树返回OK
        return OK;
}
```



考虑到Visit可能返回ERROR, 所以要判断

```
Status PostOrderTraverse(BiTree T, Status (*visit)(TElemType e))
{
    if (T) {
        PostOrderTraverse(T->lchild, visit);
        PostOrderTraverse(T->rchild, visit);
        (*visit)(T->data);
    }
    return OK;
}
```

假设visit就是简单的打印, 则可以简化

```
Status PostOrderTraverse(BiTree T)
{
    if (T) {
        PostOrderTraverse(T->lchild);
        PostOrderTraverse(T->rchild);
        cout << T->data;
    }
    return OK;
}
```

直接打印

```
Status MyVisit(TElemType e)
{
    printf("%d", e);
    return OK;
}
```

调用时:
PostOrderTraverse(T, MyVisit);

§ 6. 树和二叉树

6.3. 遍历二叉树和线索二叉树

6.3.1. 遍历二叉树

★ 二叉树遍历的非递归实现

● 中序遍历的非递归实现

基本思路：

- ① 从根开始，左子树进栈，直到左子树为空，此时栈顶元素为中序遍历的第一个结点
- ② 元素出栈，其右子树当做根，重复①
- ③ 重复① ②直到栈为空

共3种实现方法，具体参考附件的源代码
(后面给出了其中两种方法的图示理解方式)

● 二叉树中序遍历的非递归实现 (P. 130-131 算法 6.2)

```
Status InOrderTraverse(BiTree T, Status (*visit)(TElemType e))
```

```
{  
    InitStack(S); //栈的元素类型为BiTree, 顺序栈还是链栈不限  
    Push(S, T);  
    while(!StackEmpty(S)) {  
        while(GetTop(S, p) && p) //左子树到尽头  
            Push(S, p->lchild); //注意, 最后进栈的是NULL指针  
        Pop(S, p); //NULL指针出栈  
        if (!StackEmpty(S)) {  
            Pop(S, p);  
            if (!(*visit)(p->data)) {  
                DestroyStack(S); //书上漏掉, 丢内存  
                return ERROR;  
            }  
            Push(S, p->rchild); //访问右子树  
        }  
    }  
    DestroyStack(S); //书上漏掉, 丢内存  
    return OK;  
}
```


● 二叉树中序遍历的非递归实现 (P. 130-131 算法 6.2)

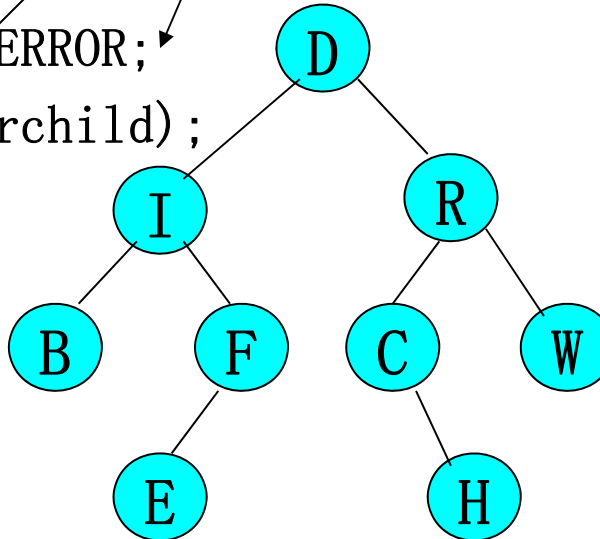
```
Status InOrderTraverse(BiTree T, Status (*visit)(TElemType e))
```

```
{  InitStack(S);  
   Push(S, T);  
   while(!StackEmpty(S)) {  
       while(GetTop(S, p) && p)  
           Push(S, p->lchild);  
       Pop(S, p);  
       if (!StackEmpty(S)) {  
           Pop(S, p);  
           if (!(*visit)(p->data))  
               return ERROR;  
           Push(S, p->rchild);  
       }  
   }  
   return OK;  
}
```

栈(num) p 输
 (0)

少两句

DestroyStack(S), 下同

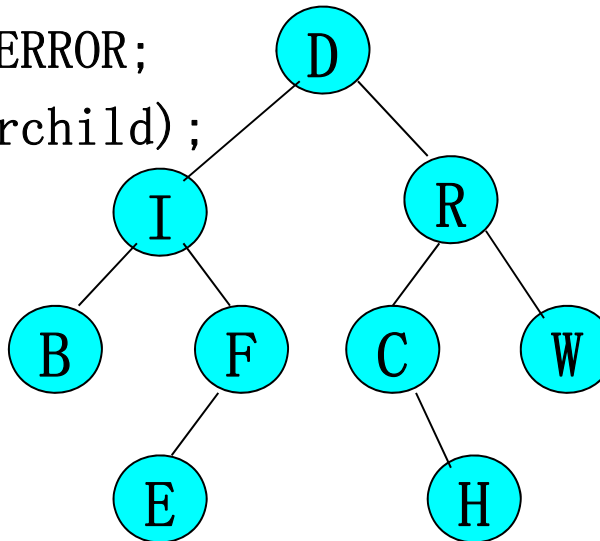


● 二叉树中序遍历的非递归实现 (P. 130-131 算法 6.2)

```
Status InOrderTraverse(BiTree T, Status (*visit)(TElemType e))
```

```
{  InitStack(S);
   Push(S, T);
   while(!StackEmpty(S)) {
       while(GetTop(S, p) && p)
           Push(S, p->lchild);
       Pop(S, p);
       if (!StackEmpty(S)) {
           Pop(S, p);
           if (!(*visit)(p->data))
               return ERROR;
           Push(S, p->rchild);
       }
   }
   return OK;
}
```

栈(num)	p	输
(0)		
D (1)		

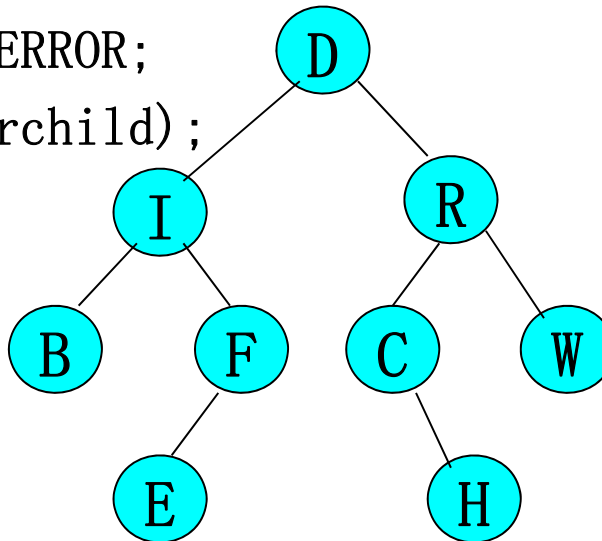


● 二叉树中序遍历的非递归实现 (P. 130-131 算法 6.2)

```
Status InOrderTraverse(BiTree T, Status (*visit)(TElemType e))
```

```
{  InitStack(S);
   Push(S, T);
   while(!StackEmpty(S)) {
       while(GetTop(S, p) && p)
           Push(S, p->lchild);
       Pop(S, p);
       if (!StackEmpty(S)) {
           Pop(S, p);
           if (!(*visit)(p->data))
               return ERROR;
           Push(S, p->rchild);
       }
   }
   return OK;
}
```

栈(num)	p	输
(0)		
D (1)		
D (1)	D	

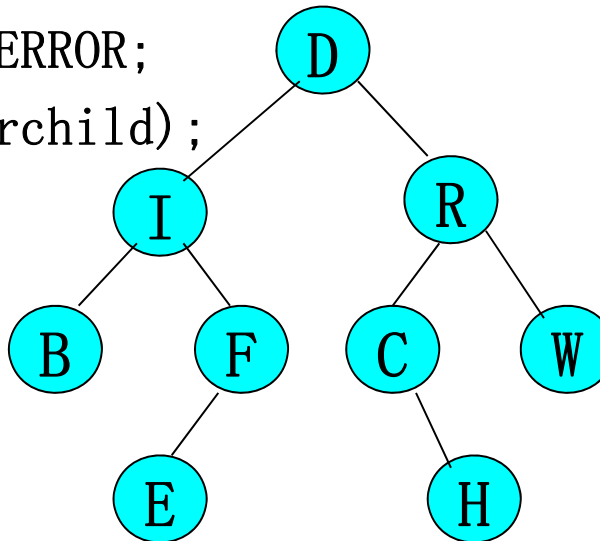


● 二叉树中序遍历的非递归实现 (P. 130-131 算法 6.2)

```
Status InOrderTraverse(BiTree T, Status (*visit)(TElemType e))
```

```
{  InitStack(S);
   Push(S, T);
   while(!StackEmpty(S)) {
       while(GetTop(S, p) && p)
           Push(S, p->lchild);
       Pop(S, p);
       if (!StackEmpty(S)) {
           Pop(S, p);
           if (!(*visit)(p->data))
               return ERROR;
           Push(S, p->rchild);
       }
   }
   return OK;
}
```

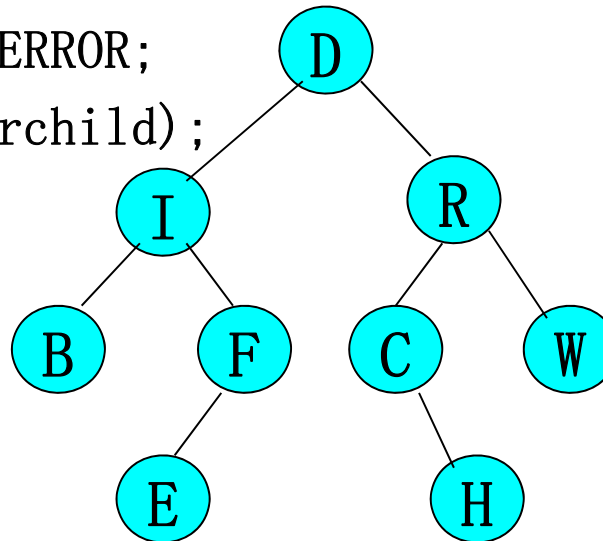
栈(num)	p	输
(0)		
D (1)		
D (1)	D	
DI (2)	D	



● 二叉树中序遍历的非递归实现 (P. 130-131 算法 6.2)

```
Status InOrderTraverse(BiTree T, Status (*visit)(TElemType e))
```

```
{  InitStack(S);
   Push(S, T);
   while(!StackEmpty(S)) {
       while(GetTop(S, p) && p)
           Push(S, p->lchild);
       Pop(S, p);
       if (!StackEmpty(S)) {
           Pop(S, p);
           if (!(*visit)(p->data))
               return ERROR;
           Push(S, p->rchild);
       }
   }
   return OK;
}
```

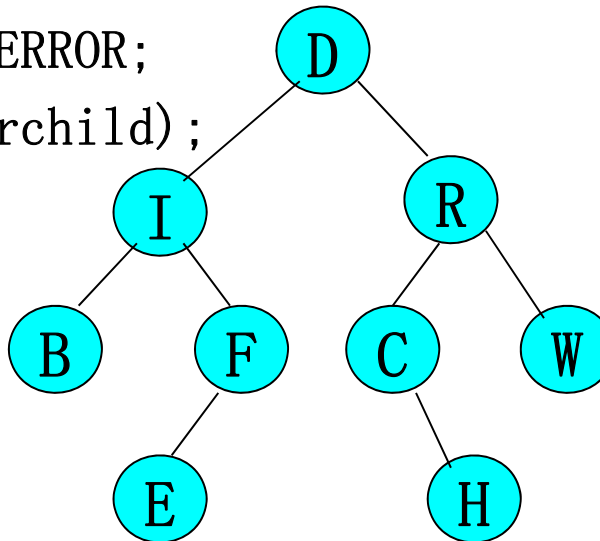


栈 (num)	p	输
(0)		
D (1)		
D (1)	D	
DI (2)	D	
DI (2)	I	

● 二叉树中序遍历的非递归实现 (P. 130-131 算法 6.2)

```
Status InOrderTraverse(BiTree T, Status (*visit)(TElemType e))
```

```
{  InitStack(S);
   Push(S, T);
   while(!StackEmpty(S)) {
       while(GetTop(S, p) && p)
           Push(S, p->lchild);
       Pop(S, p);
       if (!StackEmpty(S)) {
           Pop(S, p);
           if (!(*visit)(p->data))
               return ERROR;
           Push(S, p->rchild);
       }
   }
   return OK;
}
```

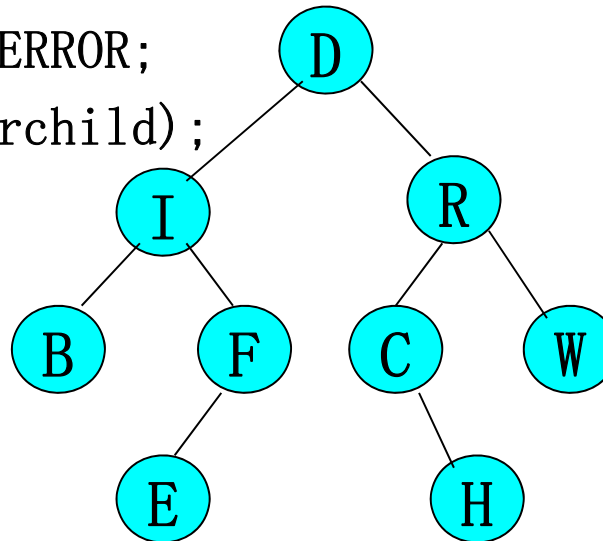


栈 (num)	p	输
(0)		
D (1)		
D (1)	D	
DI (2)	D	
DI (2)	I	
DIB (3)	I	

● 二叉树中序遍历的非递归实现 (P. 130-131 算法 6.2)

```
Status InOrderTraverse(BiTree T, Status (*visit)(TElemType e))
```

```
{  InitStack(S);
   Push(S, T);
   while(!StackEmpty(S)) {
       while(GetTop(S, p) && p)
           Push(S, p->lchild);
       Pop(S, p);
       if (!StackEmpty(S)) {
           Pop(S, p);
           if (!(*visit)(p->data))
               return ERROR;
           Push(S, p->rchild);
       }
   }
   return OK;
}
```

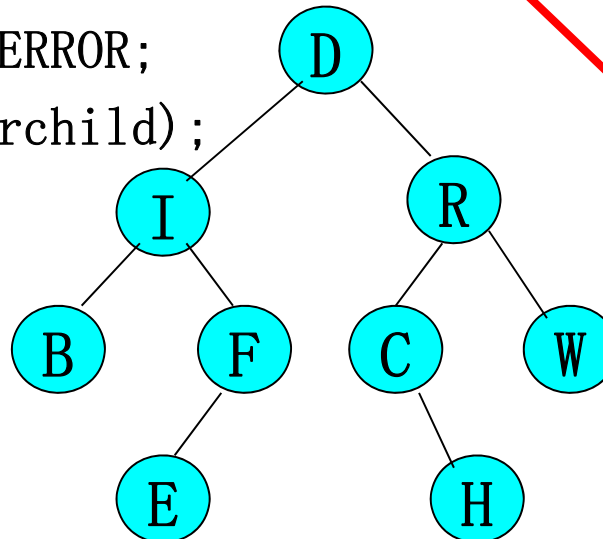


栈 (num)	p	输
(0)		
D (1)		
D (1)	D	
DI (2)	D	
DI (2)	I	
DIB (3)	I	
DIB (3)	B	

● 二叉树中序遍历的非递归实现 (P. 130-131 算法 6.2)

```
Status InOrderTraverse(BiTree T, Status (*visit)(TElemType e))
```

```
{  InitStack(S);
   Push(S, T);
   while(!StackEmpty(S)) {
       while(GetTop(S, p) && p)
           Push(S, p->lchild);
       Pop(S, p);
       if (!StackEmpty(S)) {
           Pop(S, p);
           if (!(*visit)(p->data))
               return ERROR;
           Push(S, p->rchild);
       }
   }
   return OK;
}
```

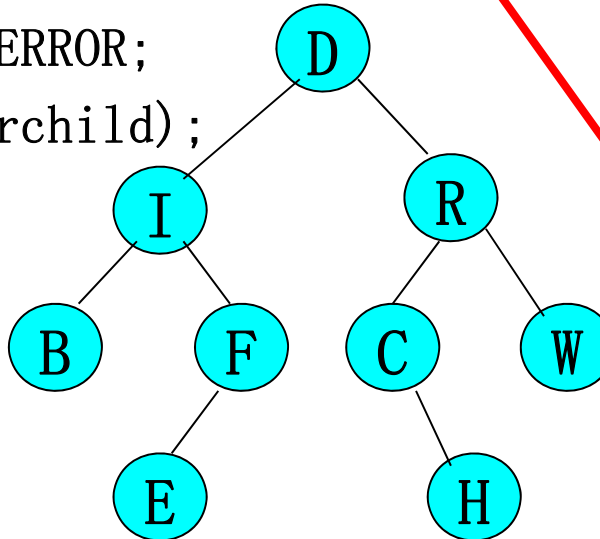


栈(num)	p	输
(0)		
D (1)		
D (1)	D	
DI (2)	D	
DI (2)	I	
DIB (3)	I	
DIB (3)	B	
DIB/ (4)	B	

● 二叉树中序遍历的非递归实现 (P. 130-131 算法 6.2)

```
Status InOrderTraverse(BiTree T, Status (*visit)(TElemType e))
```

```
{  InitStack(S);
   Push(S, T);
   while(!StackEmpty(S)) {
       while(GetTop(S, p) && p)
           Push(S, p->lchild);
       Pop(S, p);
       if (!StackEmpty(S)) {
           Pop(S, p);
           if (!(*visit)(p->data))
               return ERROR;
           Push(S, p->rchild);
       }
   }
   return OK;
}
```

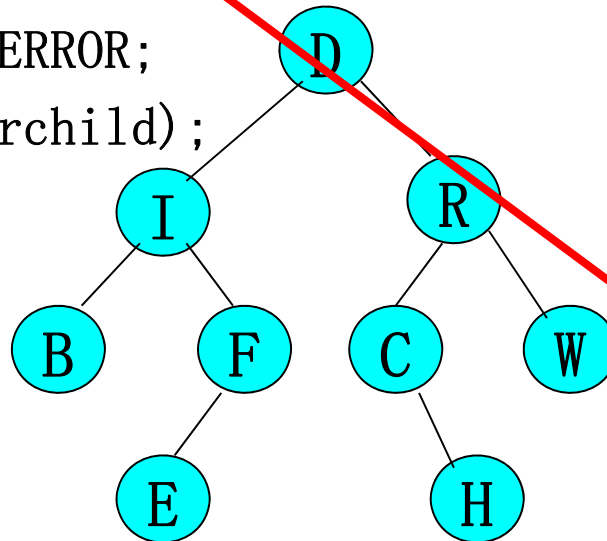


栈 (num)	p	输
(0)		
D (1)		
D (1)	D	
DI (2)	D	
DI (2)	I	
DIB (3)	I	
DIB (3)	B	
DIB/ (4)	B	
DIB/ (4)	/	

● 二叉树中序遍历的非递归实现 (P. 130-131 算法 6.2)

```
Status InOrderTraverse(BiTree T, Status (*visit)(TElemType e))
```

```
{  InitStack(S);
   Push(S, T);
   while(!StackEmpty(S)) {
       while(GetTop(S, p) && p)
           Push(S, p->lchild);
       Pop(S, p);
       if (!StackEmpty(S)) {
           Pop(S, p);
           if (!(*visit)(p->data))
               return ERROR;
           Push(S, p->rchild);
       }
   }
   return OK;
}
```

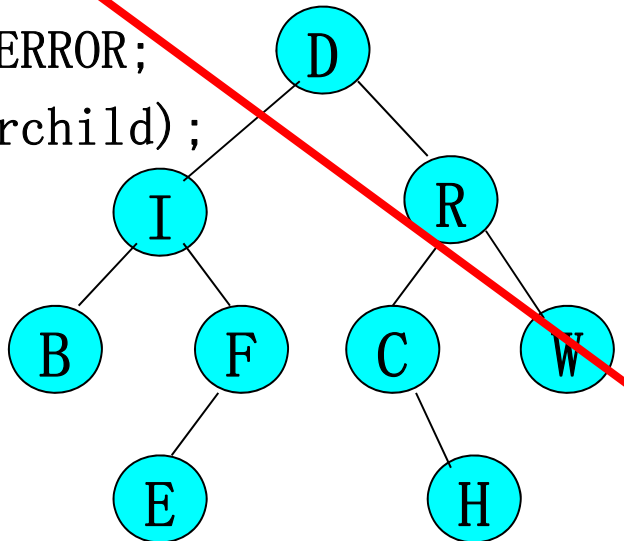


栈 (num)	p	输
(0)		
D (1)		
D (1)	D	
DI (2)	D	
DI (2)	I	
DIB (3)	I	
DIB (3)	B	
DIB/ (4)	B	
DIB/ (4)	/	
DIB (3)	/	

● 二叉树中序遍历的非递归实现 (P. 130-131 算法 6.2)

```
Status InOrderTraverse(BiTree T, Status (*visit)(TElemType e))
```

```
{  InitStack(S);
   Push(S, T);
   while(!StackEmpty(S)) {
       while(GetTop(S, p) && p)
           Push(S, p->lchild);
       Pop(S, p);
       if (!StackEmpty(S)) {
           Pop(S, p);
           if (!(*visit)(p->data))
               return ERROR;
           Push(S, p->rchild);
       }
   }
   return OK;
}
```



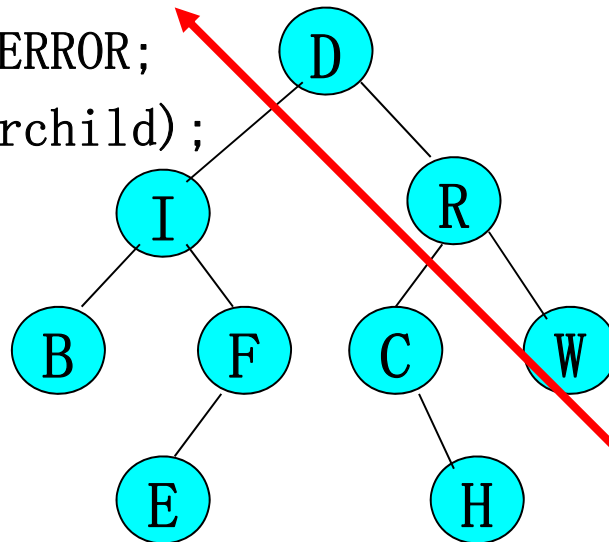
栈 (num)	p	输
(0)		
D (1)		
D (1)	D	
DI (2)	D	
DI (2)	I	
DIB (3)	I	
DIB (3)	B	
DIB/ (4)	B	
DIB/ (4)	/	
DIB (3)	/	
DI (2)	B	

● 二叉树中序遍历的非递归实现 (P. 130-131 算法 6.2)

Status InOrderTraverse(BiTree T, Status (*visit)(TElemType e))

```
{
    InitStack(S);
    Push(S, T);
    while(!StackEmpty(S)) {
        while(GetTop(S, p) && p)
            Push(S, p->lchild);
        Pop(S, p);
        if (!StackEmpty(S)) {
            Pop(S, p);
            if (!(*visit)(p->data))
                return ERROR;
            Push(S, p->rchild);
        }
    }
    return OK;
}
```

未完，后续省略，自行理解



栈 (num)	p	输
(0)		
D (1)		
D (1)	D	
DI (2)	D	
DI (2)	I	
DIB (3)	I	
DIB (3)	B	
DIB/ (4)	B	
DIB/ (4)	/	
DIB (3)	/	
DI (2)	B	
DI (2)	B	B

● 二叉树中序遍历的非递归实现 (P. 131 算法 6.3)

```
Status InOrderTraverse(BiTree T, Status (*visit)(TElemType e))
```

```
{
```

```
    InitStack(S);    //栈的元素类型为BiTree, 顺序栈还是链栈不限
```

```
    p = T;
```

```
    while(p || !StackEmpty(S)) {    //p不为空 或 栈不为空
```

```
        if (p) {
```

```
            Push(S, p);    //根进栈
```

```
            p = p->lchild; //访问左子树
```

```
        }
```

```
        else {
```

```
            Pop(S, p);    //根退栈
```

```
            if (!(*visit)(p->data)) { //访问根
```

```
                DestroyStack(S);    书上漏掉, 丢内存
```

```
                return ERROR;
```

```
            }
```

```
            p = p->rchild;    //访问右子树
```

```
        }
```

```
    } // end of while
```

```
    DestroyStack(S);    书上漏掉, 丢内存
```

```
    return OK;
```

```
}
```

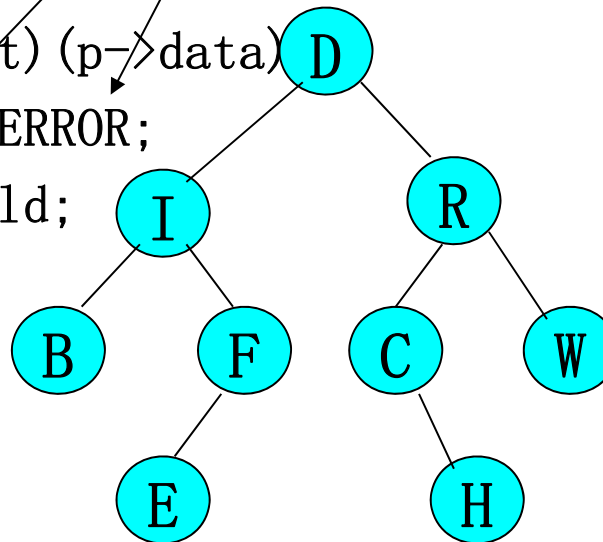
● 二叉树中序遍历的非递归实现 (P. 131 算法 6.3)

```
Status InOrderTraverse(BiTree T, Status (*visit)(TElemType e))
```

```
{  InitStack(S);  
  p = T;  
  while(p || !StackEmpty(S)) {  
    if (p) {  
      Push(S, p);  
      p = p->lchild;  
    }  
    else {  
      Pop(S, p);  
      if (!(*visit)(p->data))  
        return ERROR;  
      p = p->rchild;  
    }  
  }  
  return OK;  
}
```

栈(num)	p	输
(0)		

少两句
DestroyStack(S), 下同

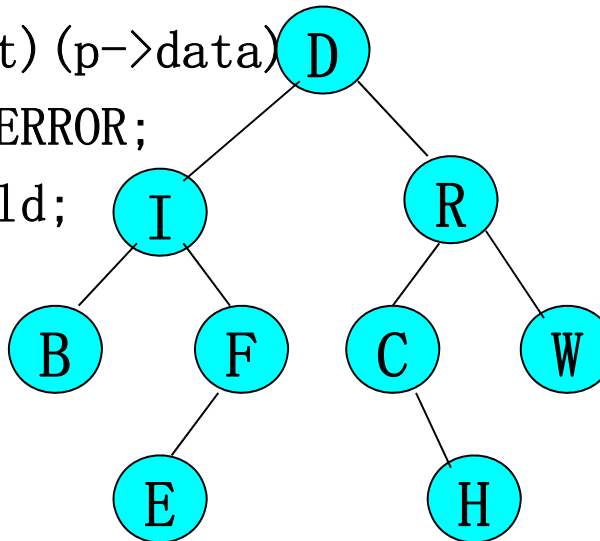


● 二叉树中序遍历的非递归实现 (P. 131 算法 6.3)

```
Status InOrderTraverse(BiTree T, Status (*visit)(TElemType e))
```

```
{  InitStack(S);
   p = T;
   while(p || !StackEmpty(S)) {
       if (p) {
           Push(S, p);
           p = p->lchild;
       }
       else {
           Pop(S, p);
           if (!(*visit)(p->data))
               return ERROR;
           p = p->rchild;
       }
   }
   return OK;
}
```

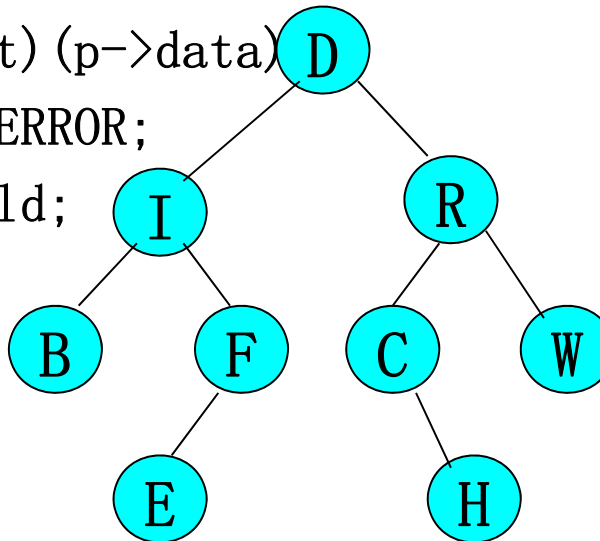
栈(num)	p	输
(0)		
(0)	D	



● 二叉树中序遍历的非递归实现 (P. 131 算法 6.3)

```
Status InOrderTraverse(BiTree T, Status (*visit)(TElemType e))
```

```
{  InitStack(S);
   p = T;
   while(p || !StackEmpty(S)) {
       if (p) {
           Push(S, p);
           p = p->lchild;
       }
       else {
           Pop(S, p);
           if (!(*visit)(p->data))
               return ERROR;
           p = p->rchild;
       }
   }
   return OK;
}
```

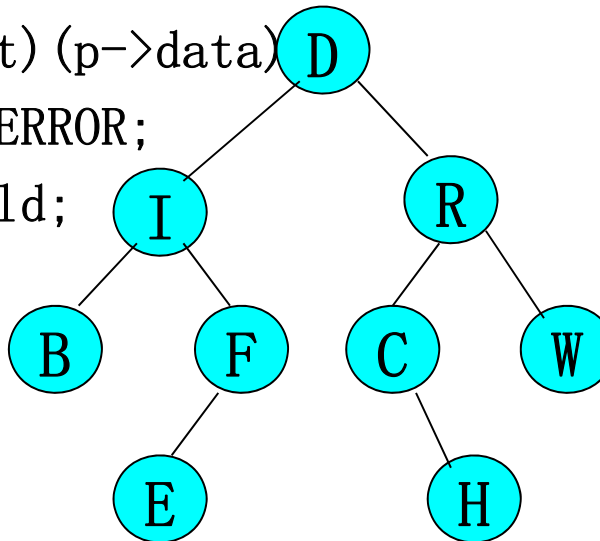


栈(num)	p	输
(0)		
(0)	D	
D (1)	D	

● 二叉树中序遍历的非递归实现 (P. 131 算法 6.3)

```
Status InOrderTraverse(BiTree T, Status (*visit)(TElemType e))
```

```
{  InitStack(S);
   p = T;
   while(p || !StackEmpty(S)) {
       if (p) {
           Push(S, p);
           p = p->lchild;
       }
       else {
           Pop(S, p);
           if (!(*visit)(p->data))
               return ERROR;
           p = p->rchild;
       }
   }
   return OK;
}
```



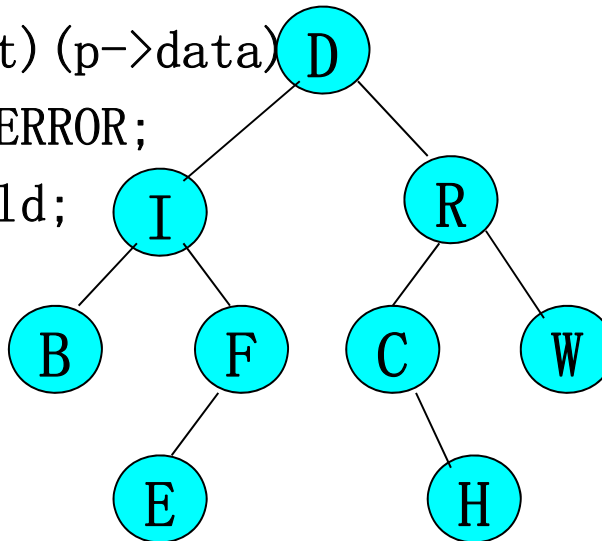
栈(num)	p	输
(0)		
(0)	D	
D (1)	D	
D (1)	I	

● 二叉树中序遍历的非递归实现 (P. 131 算法 6.3)

```
Status InOrderTraverse(BiTree T, Status (*visit)(TElemType e))
```

```
{  InitStack(S);
   p = T;
   while(p || !StackEmpty(S)) {
       if (p) {
           Push(S, p);
           p = p->lchild;
       }
       else {
           Pop(S, p);
           if (!(*visit)(p->data))
               return ERROR;
           p = p->rchild;
       }
   }
   return OK;
}
```

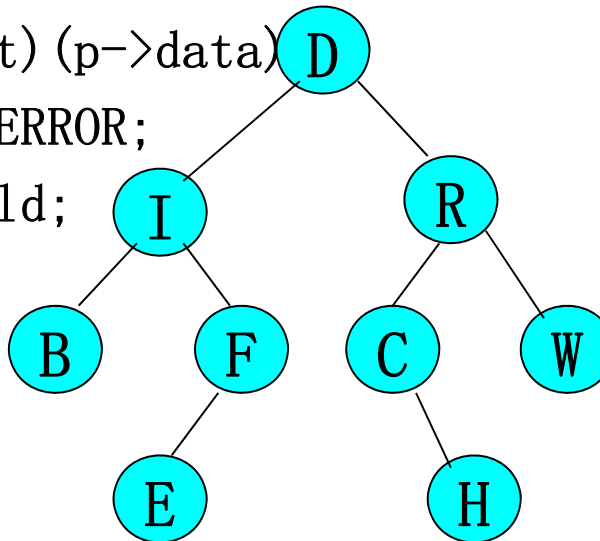
栈 (num)	p	输
(0)		
(0)	D	
D (1)	D	
D (1)	I	
DI (2)	I	



● 二叉树中序遍历的非递归实现 (P. 131 算法 6.3)

```
Status InOrderTraverse(BiTree T, Status (*visit)(TElemType e))
```

```
{  InitStack(S);
   p = T;
   while(p || !StackEmpty(S)) {
       if (p) {
           Push(S, p);
           p = p->lchild;
       }
       else {
           Pop(S, p);
           if (!(*visit)(p->data))
               return ERROR;
           p = p->rchild;
       }
   }
   return OK;
}
```

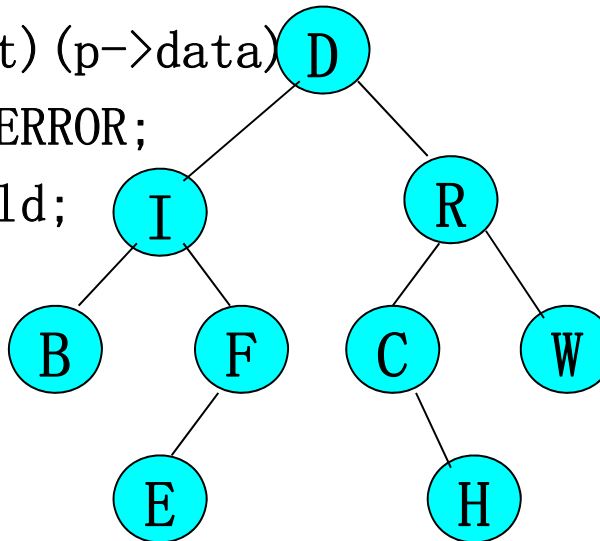


栈 (num)	p	输
(0)		
(0)	D	
D (1)	D	
D (1)	I	
DI (2)	I	
DI (2)	B	

● 二叉树中序遍历的非递归实现 (P. 131 算法 6.3)

```
Status InOrderTraverse(BiTree T, Status (*visit)(TElemType e))
```

```
{  InitStack(S);
   p = T;
   while(p || !StackEmpty(S)) {
       if (p) {
           Push(S, p);
           p = p->lchild;
       }
       else {
           Pop(S, p);
           if (!(*visit)(p->data))
               return ERROR;
           p = p->rchild;
       }
   }
   return OK;
}
```

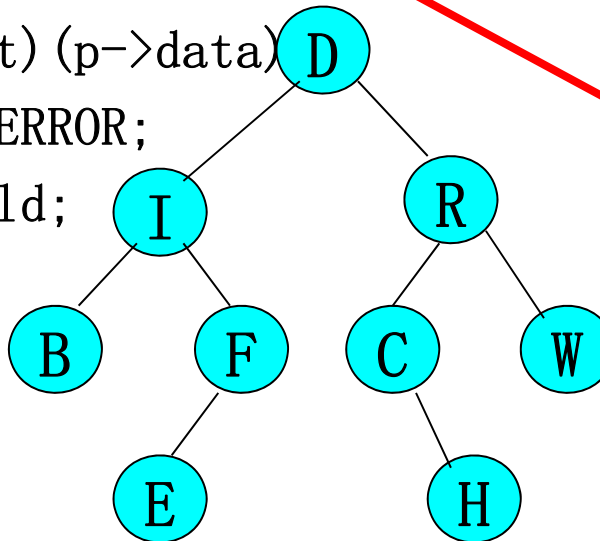


栈 (num)	p	输
(0)		
(0)	D	
D (1)	D	
D (1)	I	
DI (2)	I	
DI (2)	B	
DIB (3)	B	

● 二叉树中序遍历的非递归实现 (P. 131 算法 6.3)

```
Status InOrderTraverse(BiTree T, Status (*visit)(TElemType e))
```

```
{  InitStack(S);
   p = T;
   while(p || !StackEmpty(S)) {
       if (p) {
           Push(S, p);
           p = p->lchild;
       }
       else {
           Pop(S, p);
           if (!(*visit)(p->data))
               return ERROR;
           p = p->rchild;
       }
   }
   return OK;
}
```

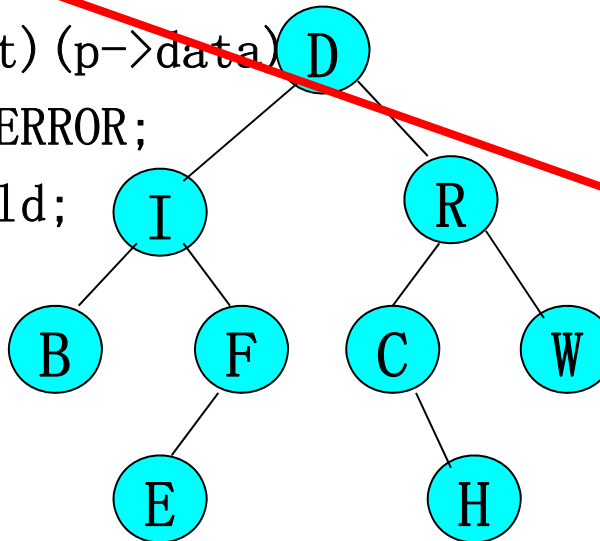


栈 (num)	p	输
(0)		
(0)	D	
D (1)	D	
D (1)	I	
DI (2)	I	
DI (2)	B	
DIB (3)	B	
DIB (3)	/	

● 二叉树中序遍历的非递归实现 (P. 131 算法 6.3)

```
Status InOrderTraverse(BiTree T, Status (*visit)(TElemType e))
```

```
{  InitStack(S);
   p = T;
   while(p || !StackEmpty(S)) {
       if (p) {
           Push(S, p);
           p = p->lchild;
       }
       else {
           Pop(S, p);
           if (!(*visit)(p->data))
               return ERROR;
           p = p->rchild;
       }
   }
   return OK;
}
```



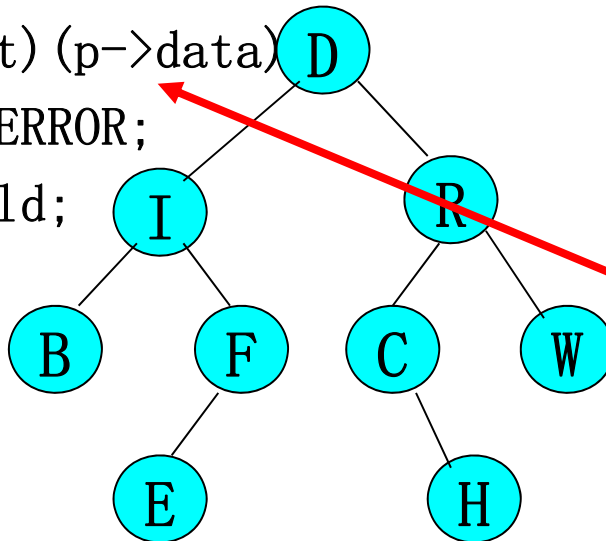
栈 (num)	p	输
(0)		
(0)	D	
D (1)	D	
D (1)	I	
DI (2)	I	
DI (2)	B	
DIB (3)	B	
DIB (3)	/	
DI (2)	B	

● 二叉树中序遍历的非递归实现 (P. 131 算法 6.3)

```
Status InOrderTraverse(BiTree T, Status (*visit)(TElemType e))
```

```
{  InitStack(S);
   p = T;
   while(p || !StackEmpty(S)) {
       if (p) {
           Push(S, p);
           p = p->lchild;
       }
       else {
           Pop(S, p);
           if (!(*visit)(p->data))
               return ERROR;
           p = p->rchild;
       }
   }
   return OK;
}
```

未完，后续省略，自行理解



栈 (num)	p	输
(0)		
(0)	D	
D (1)	D	
D (1)	I	
DI (2)	I	
DI (2)	B	
DIB (3)	B	
DIB (3)	/	
DI (2)	B	
DI (2)	B	B

§ 6. 树和二叉树

6.3. 遍历二叉树和线索二叉树

6.3.1. 遍历二叉树

★ 二叉树遍历的非递归实现

● 先序遍历的非递归实现

基本思路：

- ① 首先访问根结点，然后其右子树、左子树依次进栈
- ② 元素出栈，当做子树的根，重复①
- ③ 重复① ②直到栈为空

共3种实现方法，具体参考附件的源代码

§ 6. 树和二叉树

6.3. 遍历二叉树和线索二叉树

6.3.1. 遍历二叉树

★ 二叉树遍历的非递归实现

● 后序遍历的非递归实现

和前/中序的区别:

- ① 遇见根结点，不访问，先进栈，访问左子树
- ② 根结点退栈，仍不能访问
- ③ 根结点再次进栈，先访问右子树
- ④ 根结点退栈，访问

后序中某子树的根结点被访问的时机？

- 1、右子树为空
- 2、右子树已被访问过
=> 右子树的根一定是整个右子树中最后被访问的
=> 右子树的根是后序遍历序列中该结点的直接前驱

问：根结点能否不进栈？ 答：不行，不进栈就再也找不到了

基本思路一：

- ① 从根结点开始，沿左子树依次进栈，直到左子树为空
- ② 元素出栈，若已置访问标记，则访问，否则取得其右子树后，元素再次进栈并置访问标记，随后其右子树重复①（需进出栈两次）
- ③ 重复① ②直到栈为空

基本思路二：

- ① 从根结点开始，沿左子树依次进栈，直到左子树为空
- ② 取栈顶(不出栈)，若有右子树，则置标记，随后则其右子树当做根，重复①，若无右子树，则出栈访问
- ③ 重复① ②直到栈为空

共4种实现方法，具体参考附件的源代码

§ 6. 树和二叉树

6.3. 遍历二叉树和线索二叉树

6.3.1. 遍历二叉树

★ 二叉树遍历的非递归实现

● 层次遍历的非递归实现

基本思路：

- ① 从根开始，根进队列
- ② 元素出队列，其左/右子树依次进队列
- ③ 重复②直到队列为空

共2种实现方法，具体参考附件的源代码

§ 6. 树和二叉树

6.3. 遍历二叉树和线索二叉树

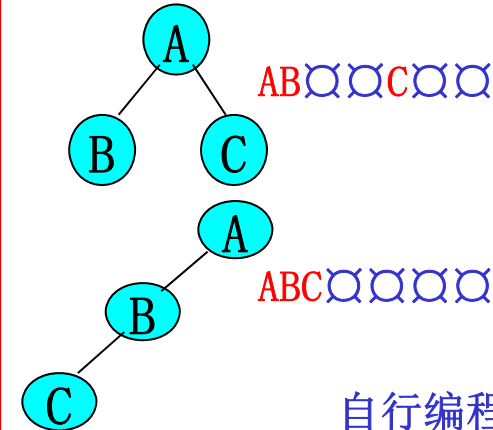
6.3.1. 遍历二叉树

★ 二叉树建立的递归实现

- 二叉树按**先序**建立的递归实现 (P. 131 算法 6.4)

```
Status CreateBiTree(BiTree &T)
{
    scanf("%c", &ch);
    //也可以 cin >> ch / ch = getchar()
    if (ch == ' ')
        T = NULL;
    else {
        if (!(T=new BiTNode))
            exit(LOVERFLOW);
        T->data = ch;
        CreateBiTree(T->lchild);
        CreateBiTree(T->rchild);
    }
    return OK;
}
```

注:和先序遍历序列不同,输入时要
包含子树为空的情况,即输入空的位置
比有值的位置多一个



自行编程并思考:
建立能否中后序?

中序:

第1个 □B□A□C□

第2个 □C□B□A□

后序:

第1个 □□B□□CA

第2个 □□C□□BA

§ 6. 树和二叉树

6.3. 遍历二叉树和线索二叉树

6.3.1. 遍历二叉树

★ 二叉树遍历的第一个结点及最后一个结点

● 第一个结点

先序：根结点

中序：从根出发，沿着**左子树链**向下，第一个**没有左孩子**的结点；若无左子树，则为根

后序：从根出发，若仅有左或右子树，则选择该子树，若同时有左右子树，则**左子树优先**，以此规则找到的第一个叶子结点；若左右子树都不存在，则为根

● 最后一个结点

先序：从根出发，若仅有左或右子树，则选择该子树，若同时有左右子树，则**右子树优先**，以此规则找到的第一个叶子结点；若左右子树都不存在，则为根

中序：从根出发，沿着**右子树链**向下，第一个**没有右孩子**的结点；若无右子树，则为根

后序：根结点

求中序的第一个结点的算法：

```
p = T;  
while (p->lchild)  
    p = p->lchild;  
return p;
```

求中序的最后一个结点的算法：

```
p = T;  
while (p->rchild)  
    p = p->rchild;  
return p;
```

自行思考：

如何找先序/后序遍历序列中的第一个/最后一个结点？

§ 6. 树和二叉树

6.3. 遍历二叉树和线索二叉树

6.3.1. 遍历二叉树

★ 通过表达式的二叉树表示求解表达式

先序遍历：前缀表达式（波兰式） $-+a*b-cd/ef$

中序遍历：中缀表达式 $a+b*c-d-e/f$

后序遍历：后缀表达式（逆波兰式） $abcd-*+ef/-$

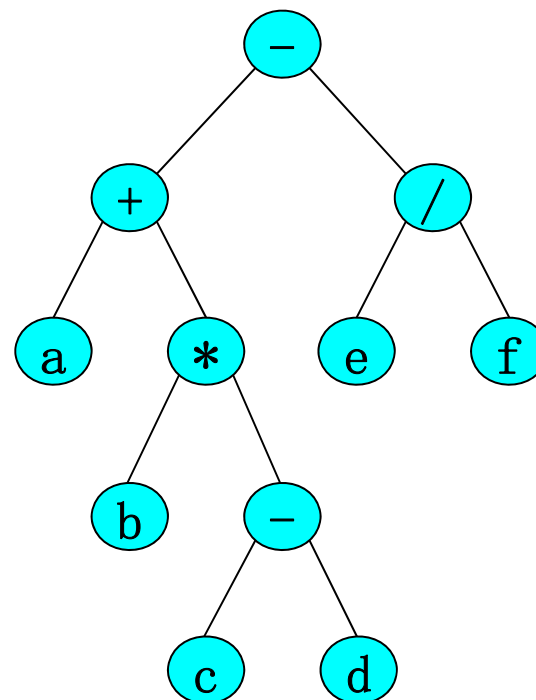
波兰式，自左到右依次进同一个栈：

- ① 当前**栈顶元素**和**次栈顶元素**均为操作数时，依次出栈，对这两个操作数进行**栈顶运算符**的运算（**次栈顶为左值**）
- ② 运算结果（**当做操作数**）进栈
- ③ 重复①②至表达式结束，栈中仅有一个操作数，即为最终结果

逆波兰式，自左到右依次扫描：

- ① 是**操作数**则依次进栈，是**运算符**则退出**栈顶及次栈顶**的操作数，进行该运算符的运算（**次栈顶为左值**）
- ② 运算结果（**当做操作数**）进栈
- ③ 重复①②至表达式结束，栈中仅有一个操作数，即为最终结果

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



§ 6. 树和二叉树

6.3. 遍历二叉树和线索二叉树

6.3.1. 遍历二叉树

★ 二叉树的常见应用

- 二叉树的输出(按层次缩进, 且区分左右子树)
- 求二叉树的高度(深度)
- 求二叉树中结点的数量
- 求二叉树中叶子结点的数量
- 二叉树左右子树互换(所有左右子树全部要互换)
- 二叉树的复制
- 在二叉树中查找值为e的结点(返回该结点的指针)
(若有多个, 按某序列返回第1个即可)

具体方法课上已讲, 再通过作业方式完成, 不再给出

§ 6. 树和二叉树

6.3. 遍历二叉树和线索二叉树

6.3.2. 线索二叉树

★ 遍历二叉树的作用

遍历二叉树可以理解为以一定的规则将二叉树中的结点排列为一个线性序列，即对非线性结构进行线性化操作，变为一个线性表

★ 线性信息的获取

二叉树的静态存储信息只包含左右子树，只有在遍历的动态过程中才能得到结点的前驱、后继信息

★ 静态获取线性信息的方法

- 每个结点加两个指针域，初始为NULL，遍历过程中分别将前驱、后继的信息赋值

(空间浪费，存储密度降低)

- n 个结点的二叉链表必然有 $n+1$ 个空指针域，能否有效利用这些空间？

★ 二叉链表的改进

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

ltag { 0 lchild域指向结点的左孩子
1 lchild域指向结点的某序列遍历的前驱

rtag { 0 rchild域指向结点的右孩子
1 rchild域指向结点的某序列遍历的后继

§ 6. 树和二叉树

6.3. 遍历二叉树和线索二叉树

6.3.2. 线索二叉树

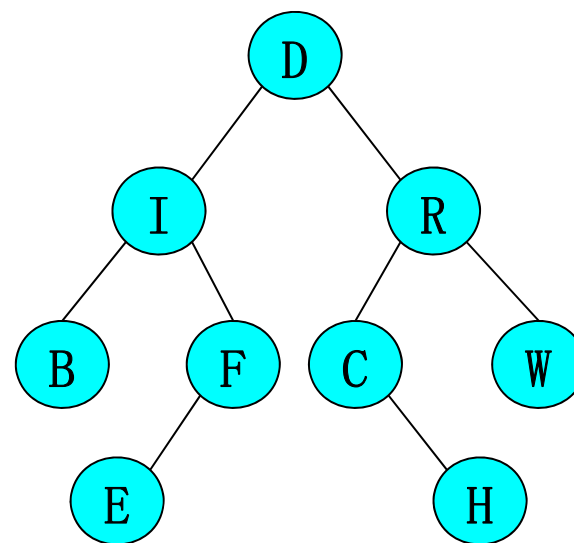
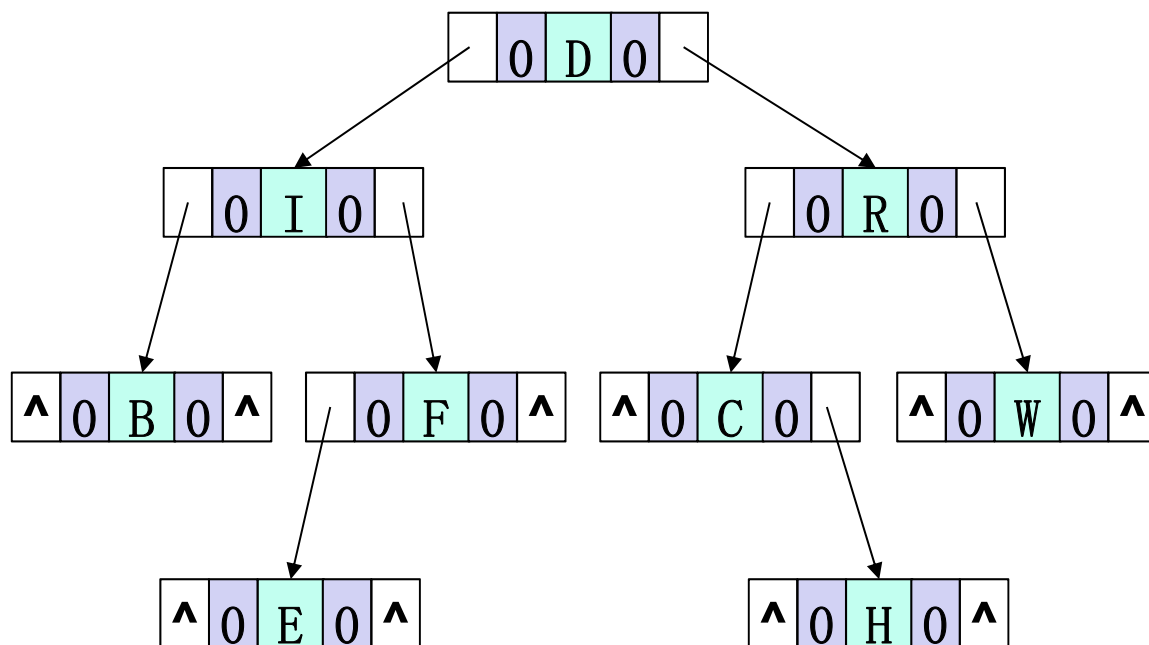
★ 线索链表与线索二叉树

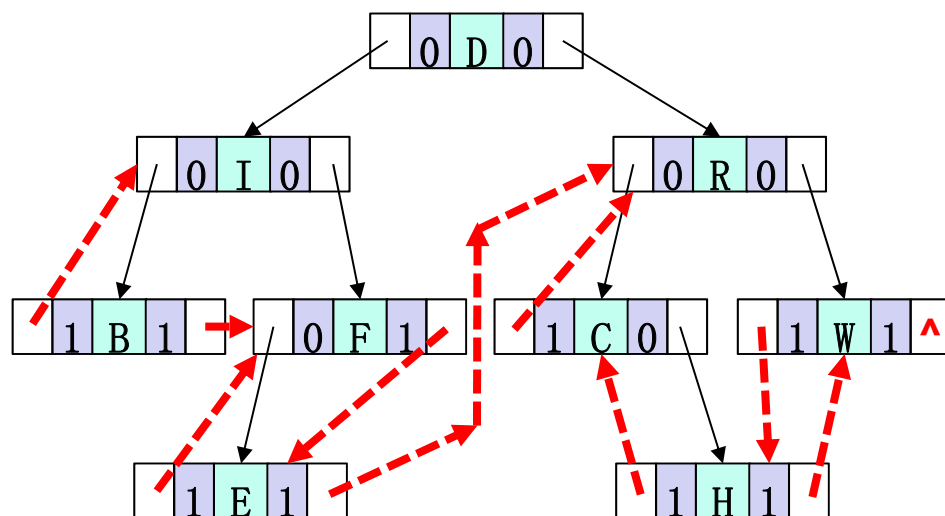
线索：指向前驱/后继的指针称为线索

线索链表：已扩充标志位结点构成的二叉链表称为线索链表

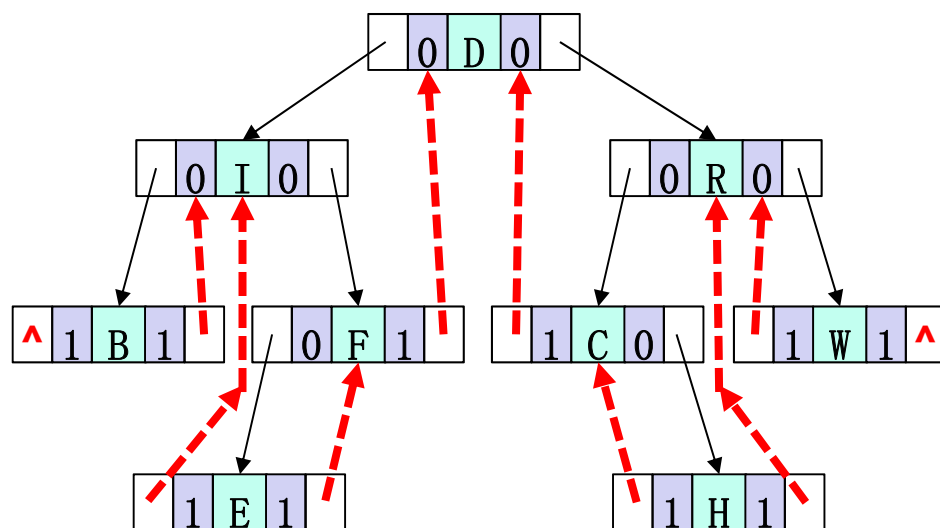
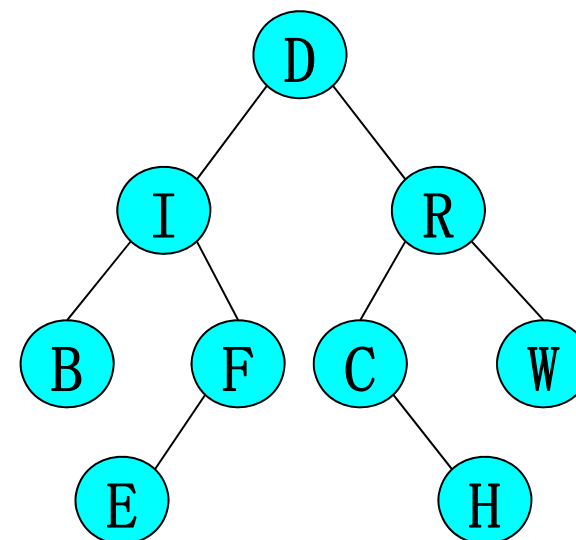
线索二叉树：加上线索的二叉树称为线索二叉树

线索化：对二叉树以某种次序遍历使其成为线索二叉树的过程称为线索化

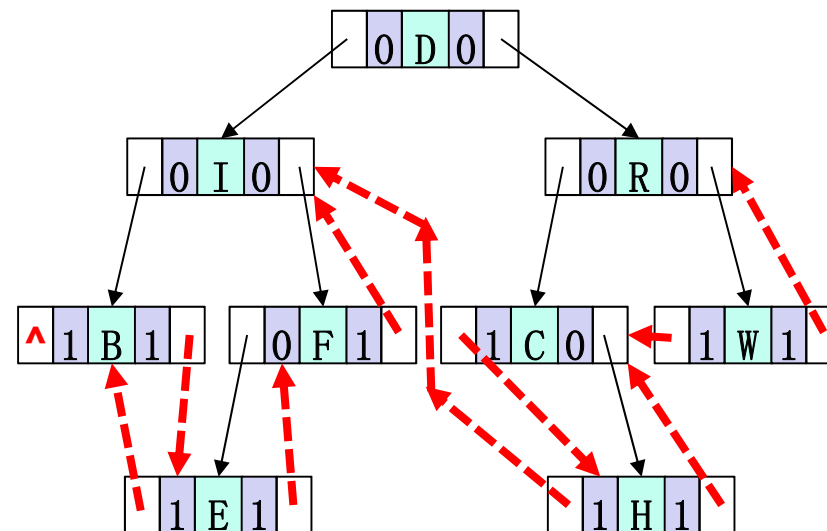




先序: DIBFERCHW



中序: BIEFDCHRW



后序: BEFIHCWRD

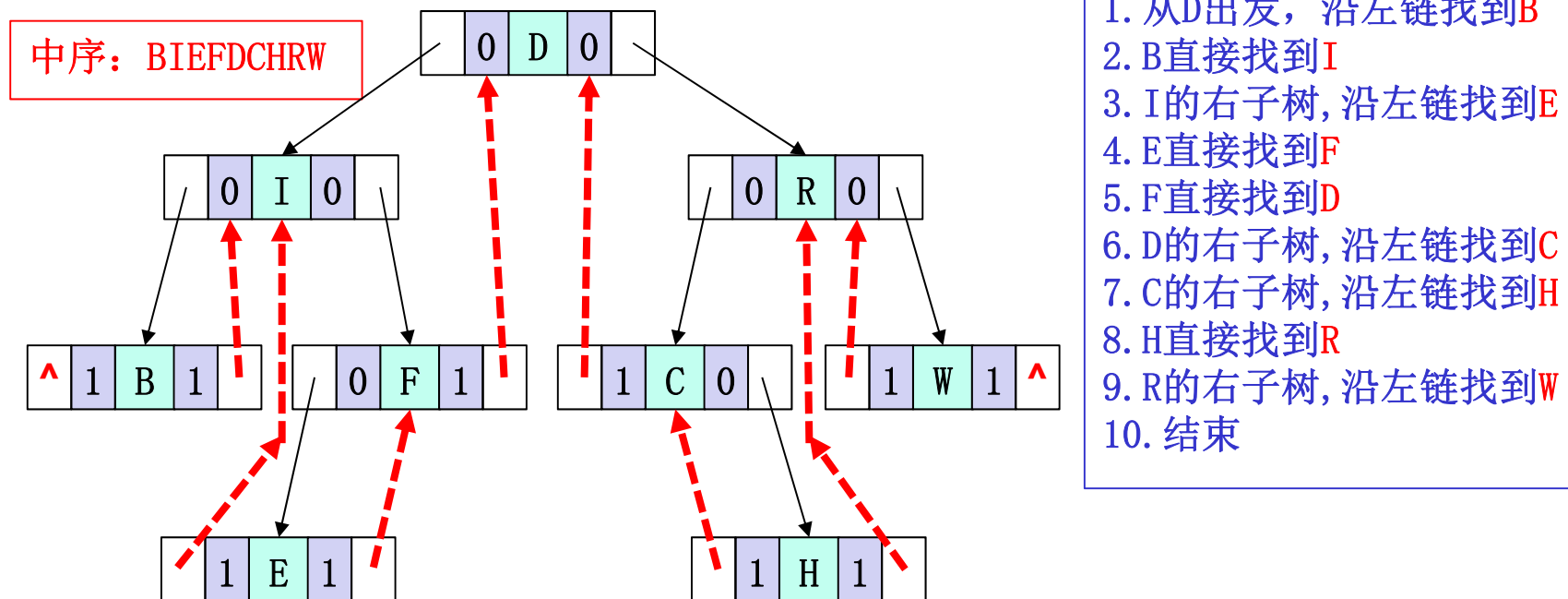
§ 6. 树和二叉树

6.3. 遍历二叉树和线索二叉树

6.3.2. 线索二叉树

★ 线索二叉树的遍历(以中序为例)

- ① 从根结点出发, 若ltag=0则沿左链一直向左, 到ltag=1为止该结点为中序遍历的首结点, 访问该结点
- ② 从该结点出发, 若rtag=1则rchild为后继结点, 访问该结点并沿右链一直向右, 若rtag=0则以其右子树的根结点重复①
- ③ 重复①②至遍历完成



§ 6. 树和二叉树

6.3. 遍历二叉树和线索二叉树

6.3.2. 线索二叉树

★ 线索链表与线索二叉树

★ 线索二叉树的遍历

● 如果逆中序遍历 (BIEFDCHRW => WRHCDFEIB)，则刚才的算法中左右互换即可

- ① 从根结点出发，若rtag=0则沿右链一直向右，到rtag=1为止
该结点为逆中序遍历的首结点，访问该结点
- ② 从该结点出发，若ltag=1则lchild为后继结点，访问该结点
并沿左链一直向左，若ltag=0则以其左子树的根结点重复①
- ③ 重复①②至遍历完成

● 前序、后序不再深入讨论，其中后序要用带双亲结点的三叉链表才能遍历

§ 6. 树和二叉树

6.3. 遍历二叉树和线索二叉树

6.3.2. 线索二叉树

★ 线索链表与线索二叉树

★ 线索二叉树的遍历

★ 线索二叉树的存储表示 (P. 133-134)

```
typedef enum { Link, Thread } PointerTag;
```

//Link和Thread都是int型，具体参考C++第7章

//Link ==0: 指针，指向孩子结点

//Thread==1: 线索，指向前驱或后继结点

```
typedef struct BiThrNode{  
    TElemType          data;  
    struct BiThrNode *lchild, *rchild;  
    PointerTag          LTag,    RTag;  
} BiThrNode, *BiThrTree;
```

§ 6. 树和二叉树

6.3. 遍历二叉树和线索二叉树

6.3.2. 线索二叉树

- ★ 线索链表与线索二叉树
- ★ 线索二叉树的遍历
- ★ 线索二叉树的存储表示
- ★ 中序遍历线索二叉树的算法实现

```
Status InOrderTraverse_Thr(BiThrTree T, Status (*Visit)(TElemType e))
```

```
{ BiThrTree p=T; //指向根结点
```

```
while (p) { //非空则循环
```

```
    while(p->LTag==Link)                沿左链一直向左  
        p=p->lchild;                    直到LTag==1为止
```

```
    if (!(*Visit)(p->data))  
        return ERROR;
```

```
    while(p->RTag==Thread && p->rchild) {  
        p=p->rchild;  
        if (!(*Visit)(p->data))  
            return ERROR;  
    }  
    若ltag==1且未结束  
    则沿右链一直向右访问结点
```

```
    p=p->rchild; //指向右子树的根结点
```

```
    }  
    return OK;
```

```
}
```

- ★ 本程序不带头结点, P. 134算法6.5带头结点
- ★ 与二叉树遍历的非递归相比, 虽然都是 $O(n)$ 但效率高(常数小), 且不需辅助栈

带头结点方式:

ltag=0

lchild=根结点

rtag=1

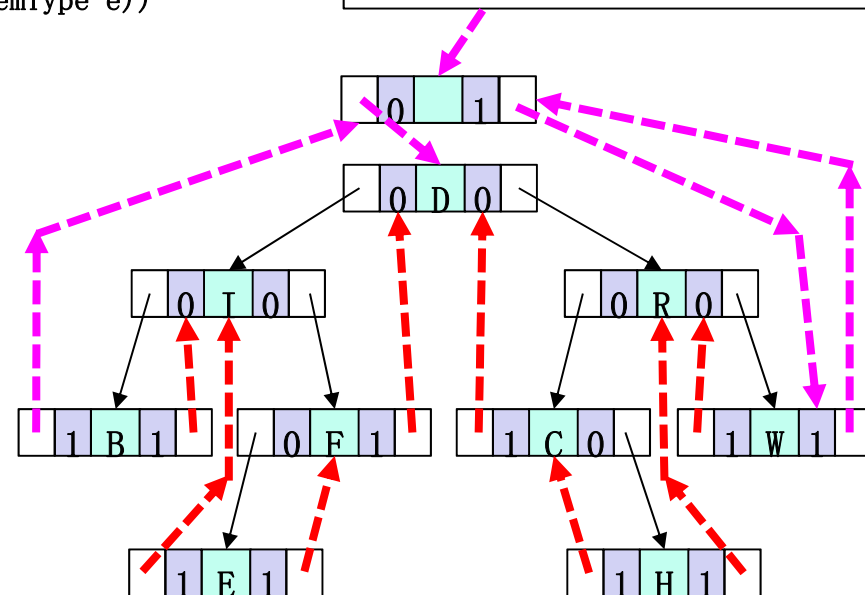
rchild=尾结点

另:

首结点lchild = 头

尾结点rchild = 头

中序: BIEFDCHRW
带头结点的中序线索二叉



§ 6. 树和二叉树

6.3. 遍历二叉树和线索二叉树

6.3.2. 线索二叉树

★ 线索链表与线索二叉树

★ 线索二叉树的遍历

★ 线索二叉树的存储表示

★ 中序遍历线索二叉树的算法实现

★ 建立中序遍历线索二叉树

① 找到首结点，若首结点的lchild为空则置ltag=1，否则不做任何操作

② 记住刚才遍历的结点指针pre，继续向后遍历，

若当前结点lchild为空，则置ltag=1/lchild=pre

若pre结点rchild为空，则rtag=1/rchild=当前，否则不做任何操作

③ 重复②至遍历完成

★ 建立中序遍历线索二叉树

```
void InThreading(BiThrTree P, BiThrTree &pre)
```

pre指向当前访问结点的前驱

```
{  if (P) {
```

```
    InThreading(P->lchild, pre);  //中序线索化左子树
```

```
    if (!P->lchild) {
```

```
        P->LTag = Thread;
```

当前结点的左子树为空

```
        P->lchild = pre;
```

则线索化，置LTag和lchild

```
    }
```

```
    if (!pre->rchild) //pre必有值，不需要pre&&!pre->rchild
```

```
        pre->RTag = Thread;
```

```
        pre->rchild = P;
```

前驱结点的右子树为空

```
    }
```

则线索化，置RTag和rchild

```
    pre = P;  //当前结点成为前驱
```

```
    InThreading(P->rchild, pre);  //中序线索化右子树
```

```
}
```

```
}
```

本算法中pre通过形参传入，书P.135算法6.7中pre要理解为一个全局指针

★ 建立中序遍历线索二叉树 (和P. 134算法6. 6有差别)

Status InOrderThreading(BiThrTree &Thrt, BiThrTree T) //T是未线索化的二叉树

{

BiThrTree pre; //定义前驱指针(书上的算法, 则pre须是全局指针变量)

if (!(Thrt = (BiThrTree)malloc(sizeof(BiThrNode))))

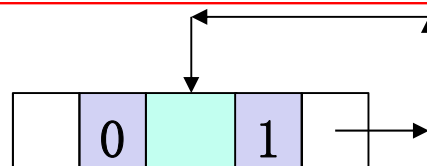
exit(LOVERFLOW);

申请线索二叉树的头结点

Thrt->LTag = Link;

Thrt->RTag = Thread;

Thrt->rchild = Thrt; //先指向自己



头结点初始化

if (!T)

Thrt->lchild = Thrt; //空树则头结点的左指针指向自己

else {

Thrt->lchild = T; //非空树则指向根结点

pre = Thrt; //前驱指针赋值

InThreading(T, pre); //对非空二叉树进行中序线索化(和书略有不同)

pre->rchild = Thrt;

InThreading执行完, pre指向最后一个结点

pre->RTag = Thread;

将RTag和rchild赋值, 指向头结点即可

Thrt->rchild = pre; //头结点的rchild赋值

}

return OK;

}

§ 6. 树和二叉树

6. 4. 树和森林

6. 4. 1. 树的存储结构

- ★ 双亲表示法：每个结点一个指针，指向其双亲结点
- ★ 孩子表示法：每个结点的孩子形成一个单链表
- ★ 孩子兄弟表示法：二叉链表，左指针指向第一个孩子，右指针指向下一个兄弟

§ 6. 树和二叉树

6. 4. 树和森林

6. 4. 1. 树的存储结构

★ 双亲表示法：每个结点一个指针，指向其双亲结点

```
#define MAX_TREE_SIZE 100
```

```
typedef struct PTNode {
```

```
    TElemType data;    //结点的值
```

```
    int        parent; //双亲的下标
```

```
} PTNode;
```

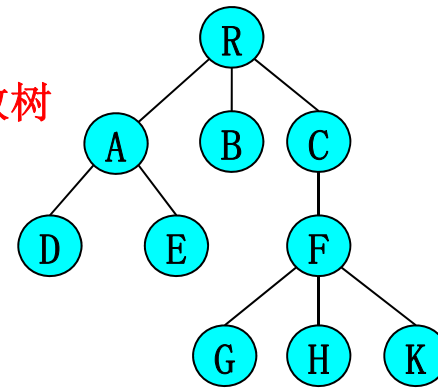
```
typedef struct {
```

```
    PTNode nodes[MAX_TREE_SIZE]; //数组存放树
```

```
    int r, n; //r:根的位置 n:结点数量
```

```
} PTree;
```

P. 135 定义



类似于静态链表

0	R	-1
1	A	0
2	B	0
3	C	0
4	D	1
5	E	1
6	F	3
7	G	6
8	H	6
9	K	6

已知F，求双亲和孩子

双亲：[6].parent

孩子：扫描整个数组，

找所有parent域为6的

§ 6. 树和二叉树

6. 4. 树和森林

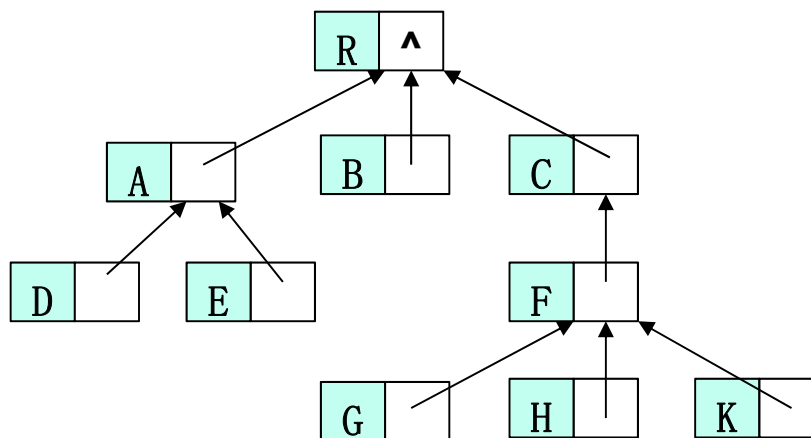
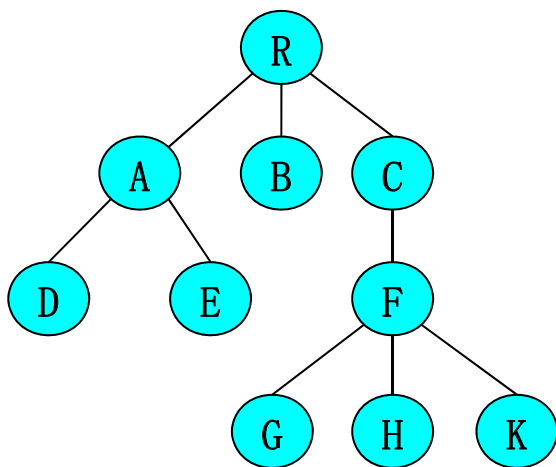
6. 4. 1. 树的存储结构

★ 双亲表示法：每个结点一个指针，指向其双亲结点

```
typedef struct ptnode {  
    TElemType    data;  
    struct ptnode *parent;  
}ptnode, *ptree;
```

链式表示

双亲表示法
无继续讨论价值



如果仅有一个指向根结点的指针，则无法找到其它结点，必须保留所有叶子结点的指针

§ 6. 树和二叉树

6. 4. 树和森林

6. 4. 1. 树的存储结构

★ 双亲表示法：每个结点一个指针，指向其双亲结点

★ 孩子表示法：每个结点的孩子形成一个单链表

方法①：同构结点方式 - 数组

```
#define MAX_TREE_SIZE    100
```

```
#define MAX_TREE_DEGREE  3 //树的度, 按需调整
```

```
typedef struct CTNode {
```

```
    TElemType data;
```

```
    int      child[MAX_TREE_DEGREE]; //子树下标
```

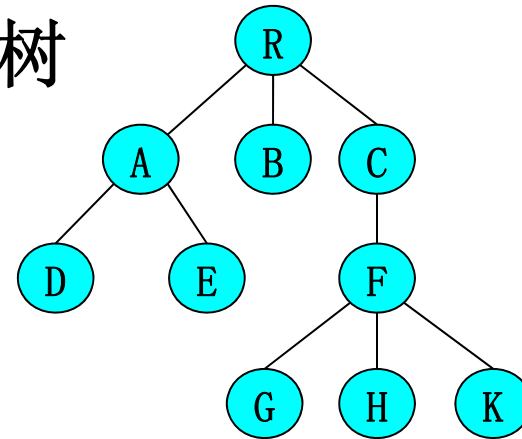
```
} CTNode;
```

```
typedef struct {
```

```
    CTNode nodes[MAX_TREE_SIZE];
```

```
    int      r, n; //根的位置和结点数
```

```
} CTree;
```



0	R	1	2	3
1	A	4	5	-1
2	B	-1	-1	-1
3	C	6	-1	-1
4	D	-1	-1	-1
5	E	-1	-1	-1
6	F	7	8	9
7	G	-1	-1	-1
8	H	-1	-1	-1
9	K	-1	-1	-1

类似于静态链表

n个结点，度为k的树

必有 $n(k-1)+1$ 个空链域

§ 6. 树和二叉树

6. 4. 树和森林

6. 4. 1. 树的存储结构

★ 双亲表示法：每个结点一个指针，指向其双亲结点

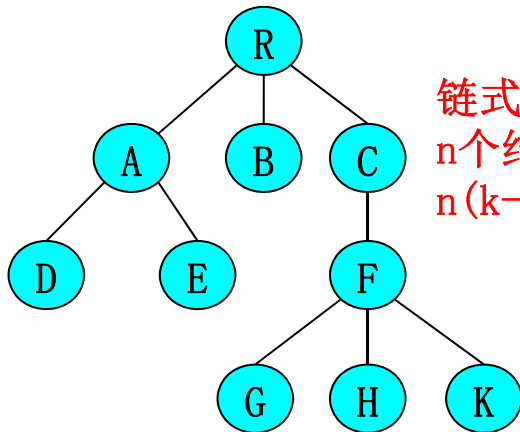
★ 孩子表示法：每个结点的孩子形成一个单链表

方法①：同构结点方式 - 链式

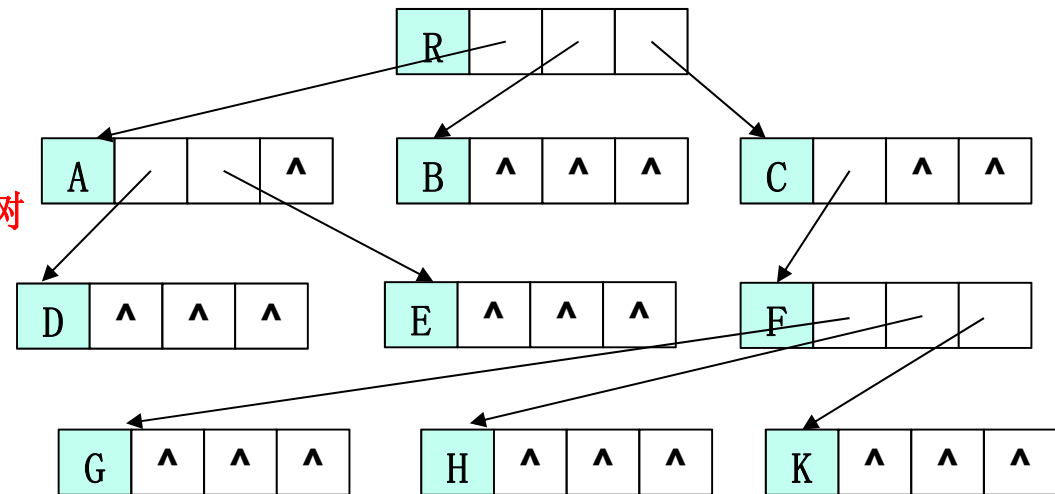
```
#define MAX_TREE_DEGREE 3 //树的度, 按需调整
```

```
typedef struct ctnode {  
    TElemType      data;  
    struct ctnode *child[MAX_TREE_DEGREE];  
} ctnode, *ctree;
```

孩子表示法的方式①
不再讨论



链式表示
n个结点，度为k的树
 $n(k-1)+1$ 个空链域



§ 6. 树和二叉树

6. 4. 树和森林

6. 4. 1. 树的存储结构

★ 双亲表示法：每个结点一个指针，指向其双亲结点

★ 孩子表示法：每个结点的孩子形成一个单链表

方法②：异构结点方式 - 数组

```
#define MAX_TREE_SIZE    100
```

```
#define MAX_TREE_DEGREE  3 //树的度, 按需调整
```

```
typedef struct CTreeNode {
```

```
    TElemType data;
```

```
    int      ndegree; //结点的度
```

```
    int      *child;  //孩子指针, 按需申请
```

```
    } CTreeNode;
```

```
typedef struct {
```

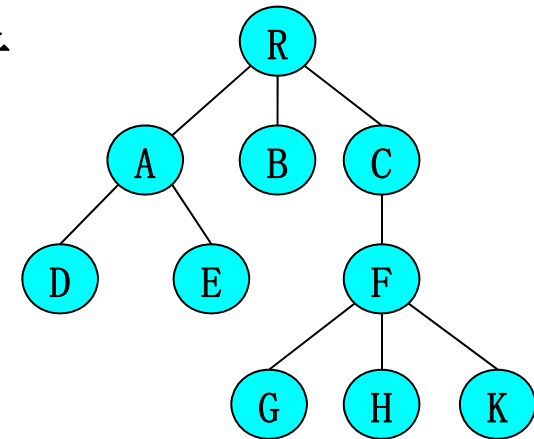
```
    CTreeNode nodes[MAX_TREE_SIZE];
```

```
    int      r, n; //根的位置和结点数
```

```
    } CTree;
```

使用时，每个结点

```
child = new int[ndegree];
```



类似于静态链表
空指针域比方式①少
但操作更为复杂

0	R	3		→	1	2	3
1	A	2		→	4	5	
2	B	0	^				
3	C	1		→	6		
4	D	0	^				
5	E	0	^				
6	F	3		→	7	8	9
7	G	0	^				
8	H	0	^				
9	K	0	^				

§ 6. 树和二叉树

6. 4. 树和森林

6. 4. 1. 树的存储结构

★ 双亲表示法：每个结点一个指针，指向其双亲结点

★ 孩子表示法：每个结点的孩子形成一个单链表

方法②：异构结点方式 - 链式

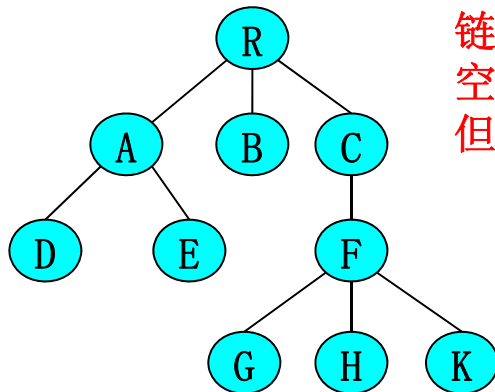
#define MAX_TREE_DEGREE 3 //树的度, 按需调整

```
typedef struct ctnode {  
    TElemType    data;  
    int          ndegree;  
    struct ctnode **child;  
} ctnode, *ctree;
```

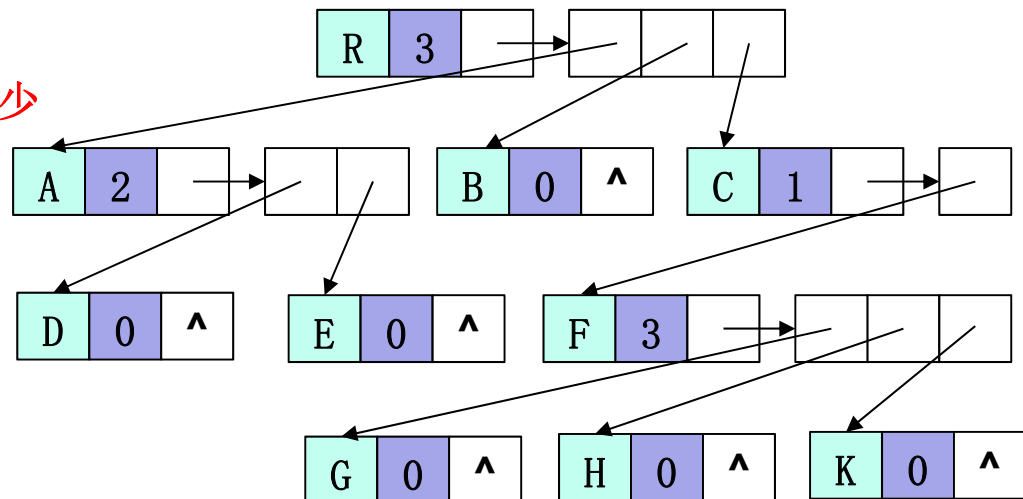
孩子表示法的方式②
不再讨论

使用时，每个结点

```
child = (struct ctnode **)malloc(ndegree *  
    sizeof(struct ctnode *));
```



链式结构
空指针域比方式①少
但操作更为复杂



§ 6. 树和二叉树

6. 4. 树和森林

6. 4. 1. 树的存储结构

★ 双亲表示法：每个结点一个指针，指向其双亲结点

★ 孩子表示法：每个结点的孩子形成一个单链表

方法③：孩子链表方式 - 结点的所有孩子为单链表

```
#define MAX_TREE_SIZE    100
```

P. 136 定义

```
typedef struct CTNode {
```

```
    int          child;    //该孩子的下标位置
```

```
    struct CTNode *next;   //下一个孩子
```

```
} *Childptr;
```

```
typedef struct {
```

```
    TElemType data;
```

```
    Childptr  firstchild;  //孩子链表头指针
```

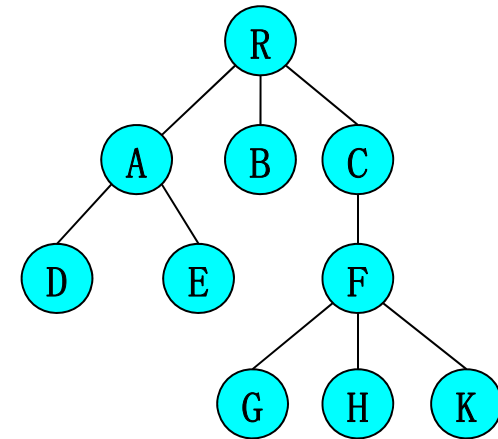
```
} CBox;
```

```
typedef struct {
```

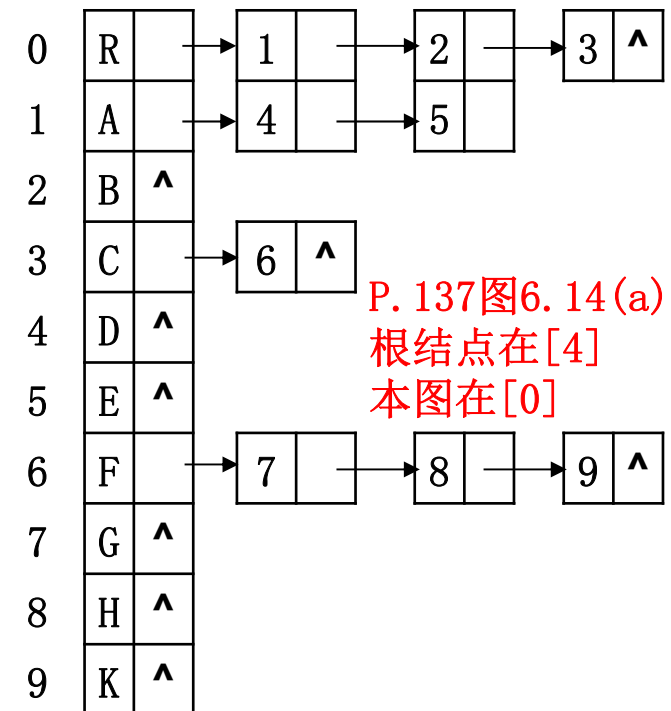
```
    CBox  nodes[MAX_TREE_SIZE];
```

```
    int   r, n;  //根的位置和结点数
```

```
} CTree;
```



数组方式存放结点
结点间关系是指针形式



P. 137图6. 14(a)
根结点在[4]
本图在[0]

§ 6. 树和二叉树

6. 4. 树和森林

6. 4. 1. 树的存储结构

★ 双亲表示法：每个结点一个指针，指向其双亲结点

★ 孩子表示法：每个结点的孩子形成一个单链表

方法③：孩子链表方式 - 结点的所有孩子为单链表

```
#define MAX_TREE_SIZE 100
```

变化：P. 136 定义+双亲

```
typedef struct CTNode {
```

```
    int          child;    //该孩子的下标位置
```

```
    struct CTNode *next;   //下一个孩子
```

```
} *Childptr;
```

```
typedef struct {
```

```
    TElemType data;
```

```
    int          parent;    //双亲的下标位置
```

```
    Childptr     firstchild; //孩子链表头指针
```

```
} CTBox;
```

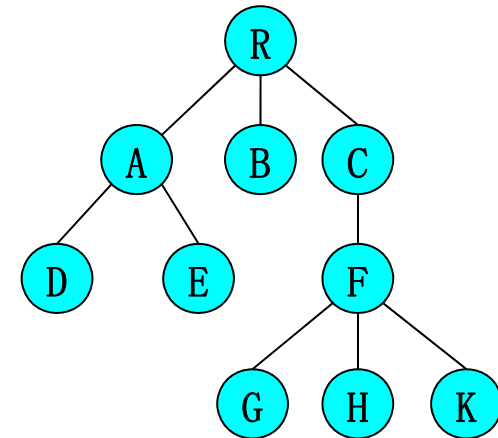
```
typedef struct {
```

```
    CTBox     nodes[MAX_TREE_SIZE];
```

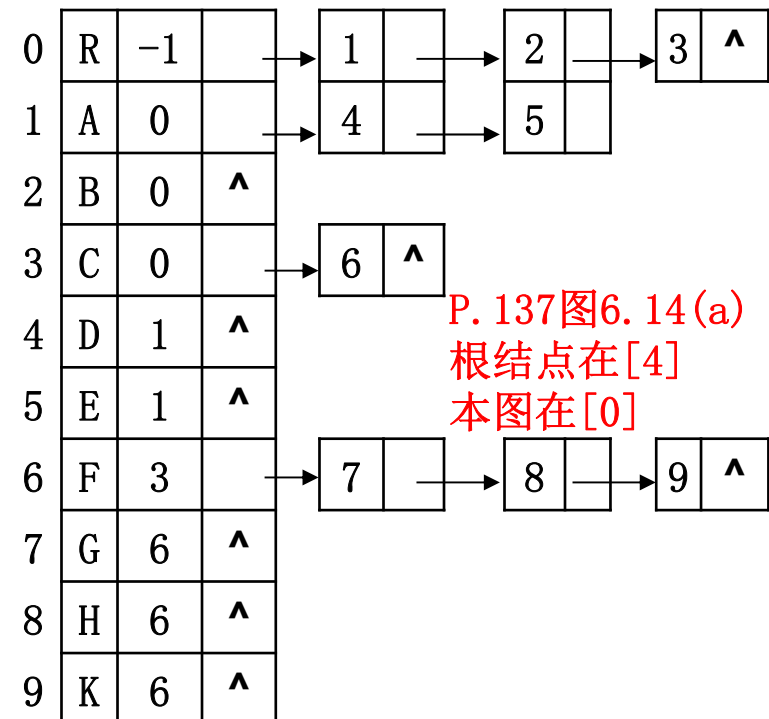
```
    int      r, n;    //根的位置和结点数
```

```
} CTree;
```

孩子表示法的方式③不再讨论



数组方式存放结点
结点间关系是指针形式



§ 6. 树和二叉树

6. 4. 树和森林

6. 4. 1. 树的存储结构

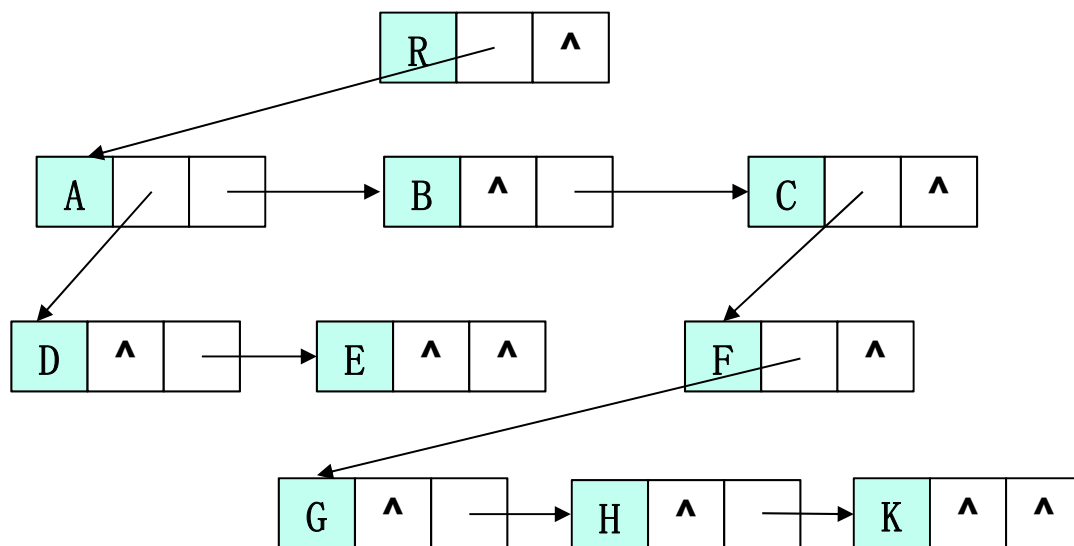
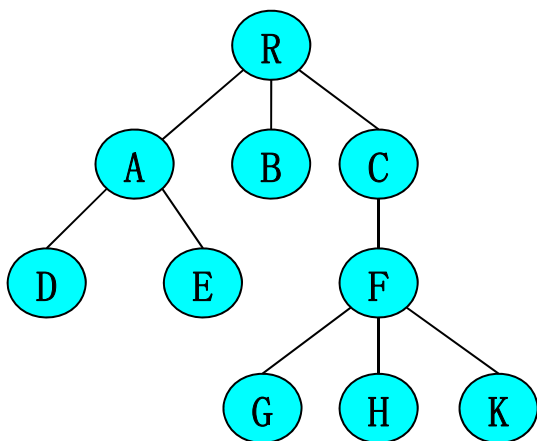
★ 孩子兄弟表示法：二叉链表，左指针指向第一个孩子，右指针指向下一个兄弟

/* P. 136 定义 */

```
typedef struct CSNode {  
    TElemType data;  
    struct CSNode *firstchild, *nextsibling;  
} CSNode, *CSTree;
```

/* P. 127 二叉链表的定义 */

```
typedef struct BiTNode {  
    TElemType data;  
    struct BiTNode *lchild, *rchild;  
} BiTNode, *BiTree
```



§ 6. 树和二叉树

6. 4. 树和森林

6. 4. 1. 树的存储结构

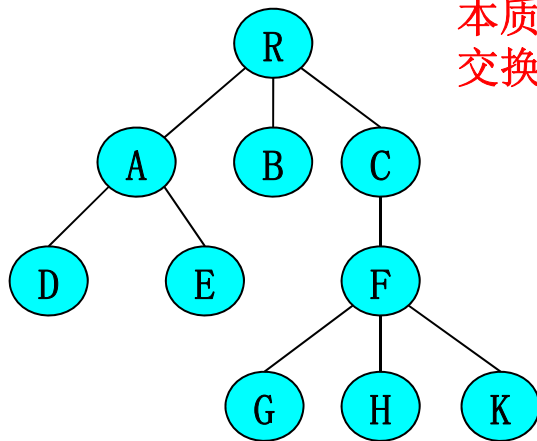
★ 孩子兄弟表示法：二叉链表，左指针指向第一个孩子，右指针指向下一个兄弟

/* P. 136 定义 */

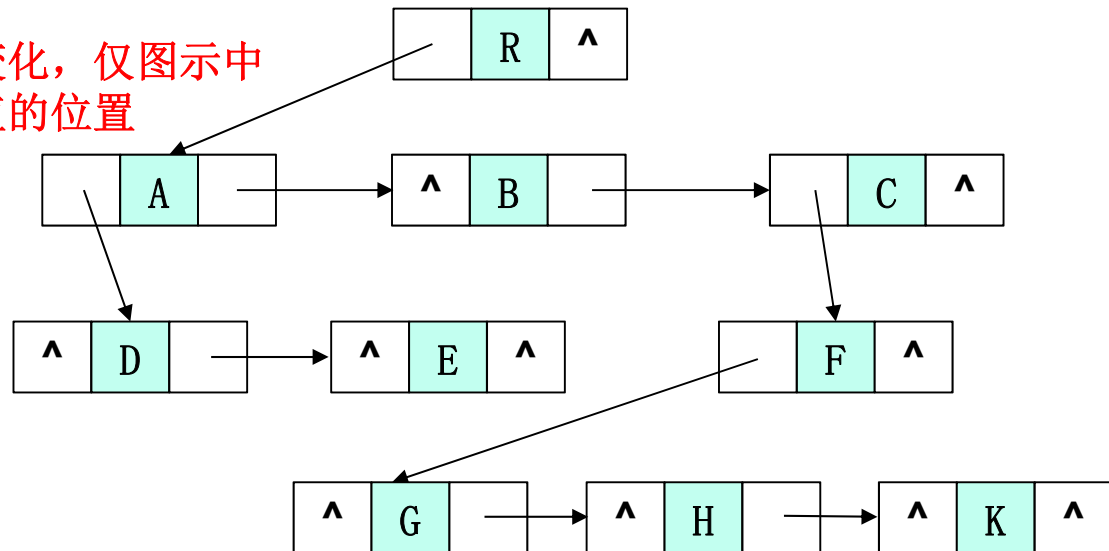
```
typedef struct CSNode {  
    TElemType data;  
    struct CSNode *firstchild, *nextsibling;  
} CSNode, *CSTree;
```

/* P. 127 二叉链表的定义 */

```
typedef struct BiTNode {  
    TElemType data;  
    struct BiTNode *lchind, *rchild;  
} BiTNode, *BiTree
```



本质无任何变化，仅图示中
交换指针和值的位置



§ 6. 树和二叉树

6. 4. 树和森林

6. 4. 1. 树的存储结构

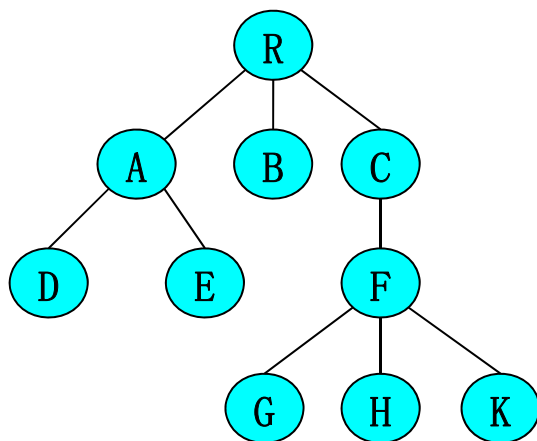
★ 孩子兄弟表示法：二叉链表，左指针指向第一个孩子，右指针指向下一个兄弟

/* P. 136 定义 */

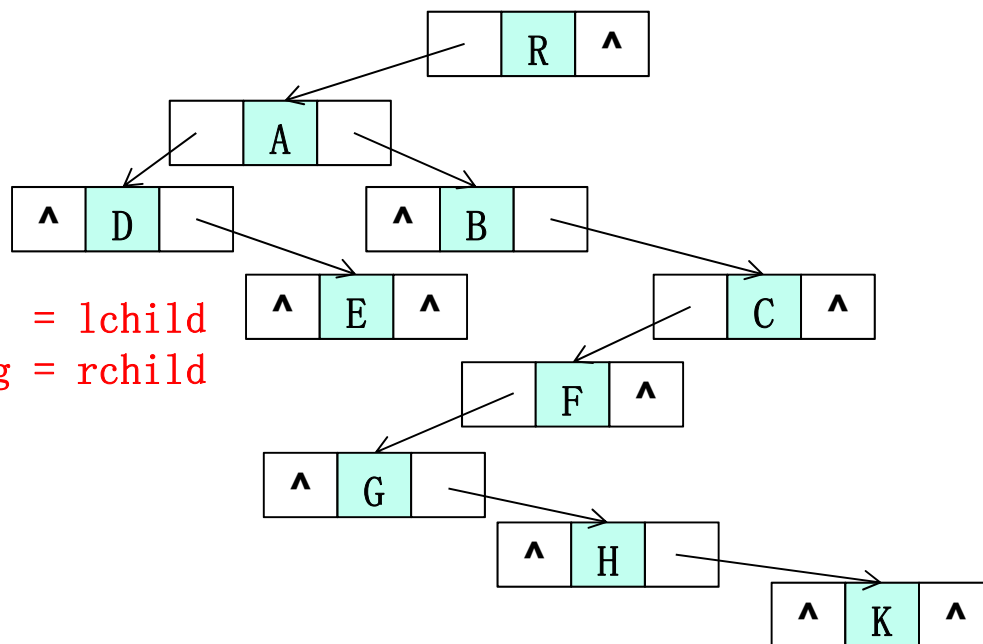
```
typedef struct CSNode {  
    TElemType data;  
    struct CSNode *firstchild, *nextsibling;  
} CSNode, *CSTree;
```

/* P. 127 二叉链表的定义 */

```
typedef struct BiTNode {  
    TElemType data;  
    struct BiTNode *lchild, *rchild;  
} BiTNode, *BiTree
```



firstchild = lchild
nextsibling = rchild
就是二叉树



§ 6. 树和二叉树

6. 4. 树和森林

6. 4. 1. 树的存储结构

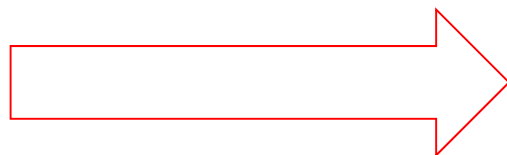
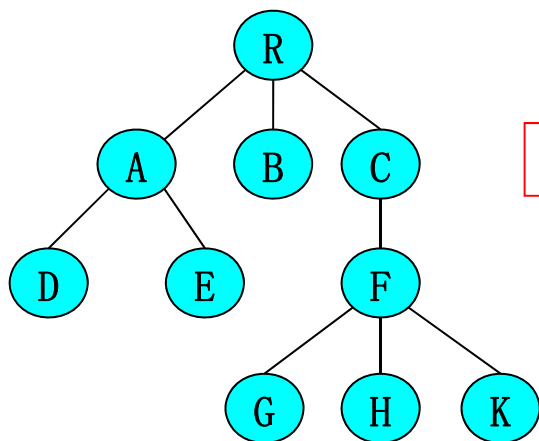
★ 孩子兄弟表示法：二叉链表，左指针指向第一个孩子，右指针指向下一个兄弟

/* P. 136 定义 */

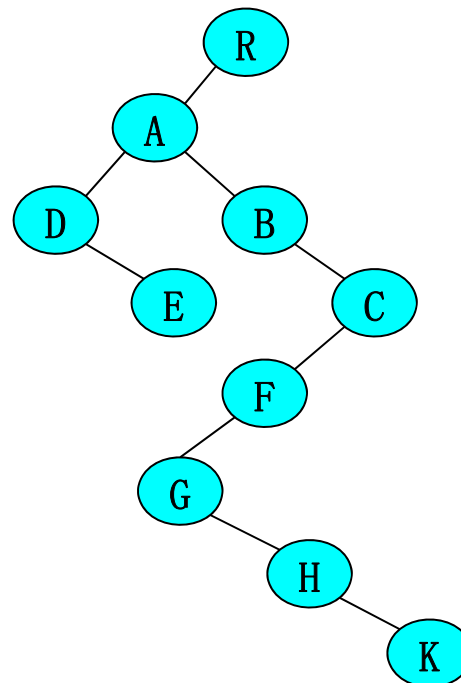
```
typedef struct CSNode {  
    TElemType data;  
    struct CSNode *firstchild, *nextsibling;  
} CSNode, *CSTree;
```

/* P. 127 二叉链表的定义 */

```
typedef struct BiTNode {  
    TElemType data;  
    struct BiTNode *lchild, *rchild;  
} BiTNode, *BiTree
```



firstchild = lchild
nextsibling = rchild
就是二叉树



§ 6. 树和二叉树

6. 4. 树和森林

6. 4. 1. 树的存储结构

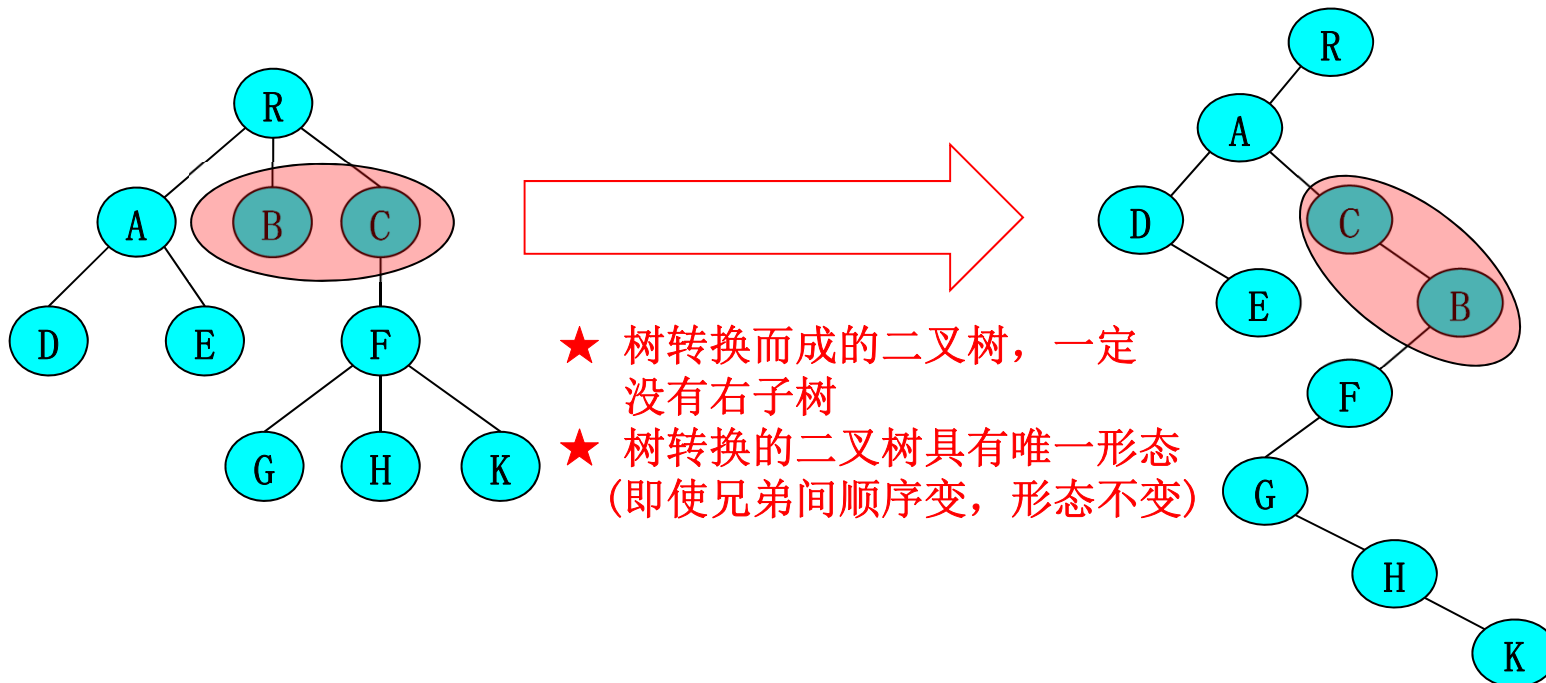
★ 孩子兄弟表示法：二叉链表，左指针指向第一个孩子，右指针指向下一个兄弟

/* P. 136 定义 */

```
typedef struct CSNode {  
    TElemType data;  
    struct CSNode *firstchild, *nextsibling;  
} CSNode, *CSTree;
```

/* P. 127 二叉链表的定义 */

```
typedef struct BiTNode {  
    TElemType data;  
    struct BiTNode *lchild, *rchild;  
} BiTNode, *BiTree
```



§ 6. 树和二叉树

6. 4. 树和森林

6. 4. 1. 树的存储结构

- ★ 双亲表示法：每个结点一个指针，指向其双亲结点(不再继续讨论)
- ★ 孩子表示法：每个结点的孩子形成一个单链表(不再继续讨论)
- ★ 孩子兄弟表示法：二叉链表，左指针指向第一个孩子，右指针指向下一个兄弟(最常用)

§ 6. 树和二叉树

6. 4. 树和森林

6. 4. 2. 森林和二叉树的转换

★ 树的孩子兄弟表示法与二叉树的对应关系：将下一个兄弟看作右孩子，即为对应的二叉树

● 树转换而成的二叉树，一定没有右子树

★ 森林和二叉树的对应关系：二叉树根结点的右结点对应下一棵树，右结点的右结点对应再下一棵树

● 二叉树右链结点(全部右结点)的数量，就是森林中树的数量

★ 森林转二叉树的递归定义

设 $F = \{T_1, T_2, \dots, T_m\}$ 为森林， $B = \{\text{root}, \text{LB}, \text{RB}\}$ 为二叉树

① 若F为空，即 $m=0$ ，则B为空

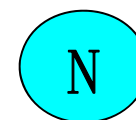
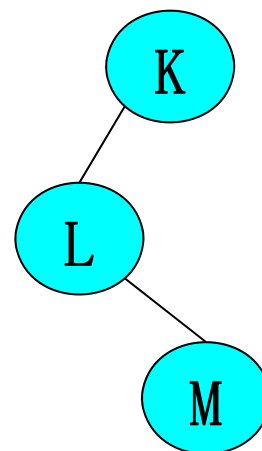
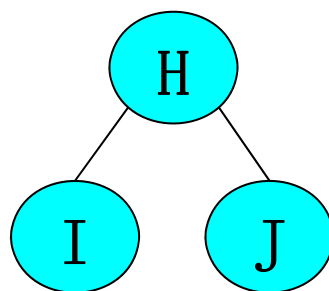
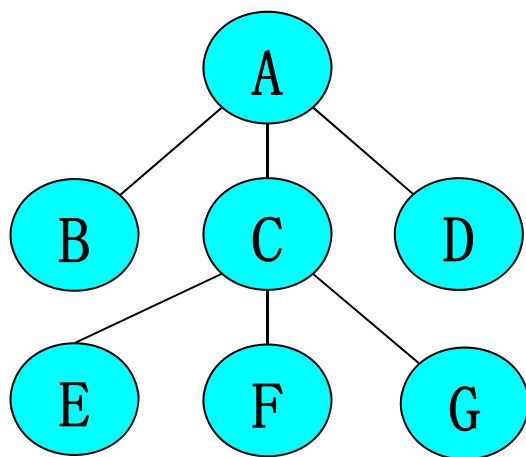
② 若F非空，即 $m \neq 0$ ，则B的root为F的第一棵树的根，即 $\text{ROOT}(T_1)$ ；B的左子树LB是第一棵树 T_1 去根后的子树森林 $F_1 = \{T_{11}, T_{12}, \dots, T_{1m_1}\}$ 转换而成的二叉树，B的右子树RB是F中其余树 $F' = \{T_2, T_3, \dots, T_m\}$ 转换而成的二叉树

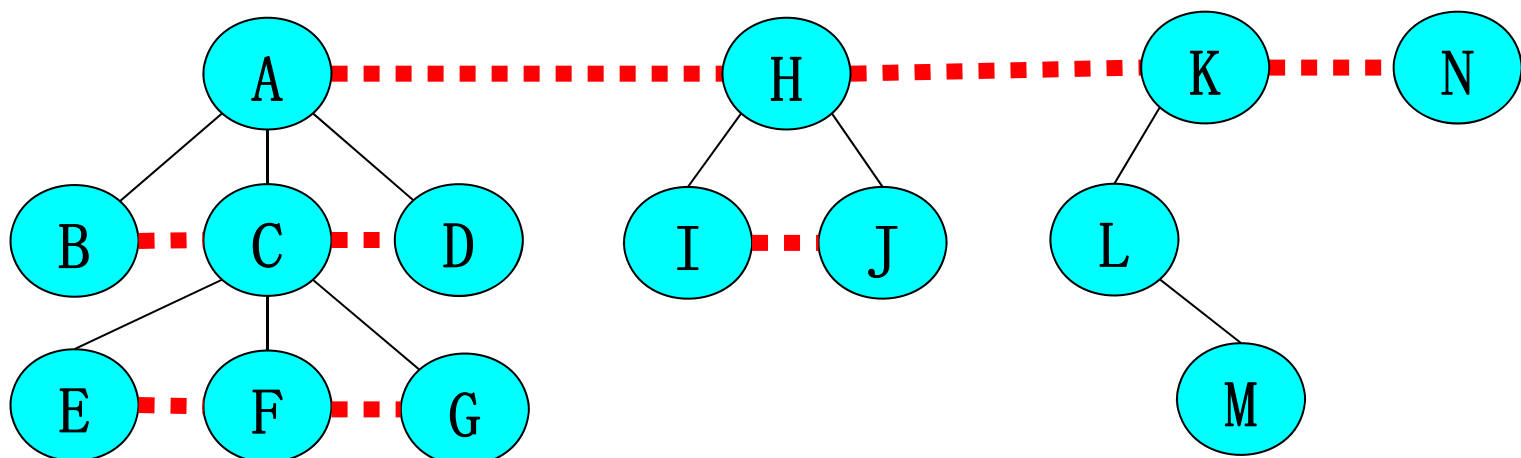
★ 森林转二叉树的转换方法

① 连兄弟：所有兄弟间横线相连(所有树的根为兄弟)

② 断孩子：每个结点只保留与第一个孩子的连线，断开与其它孩子的连线

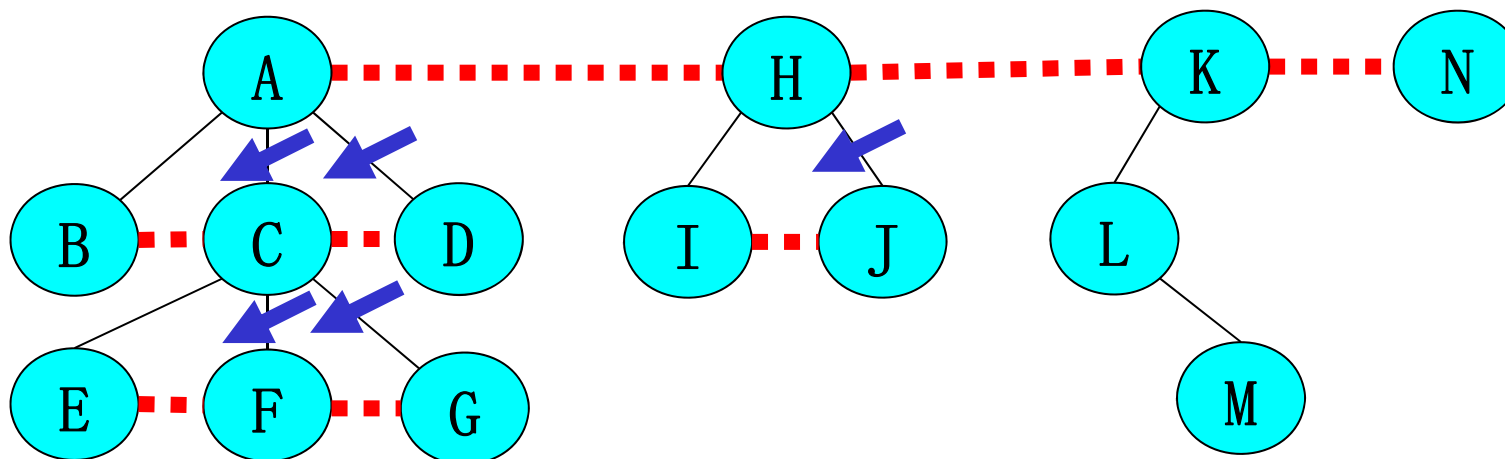
③ 顺时针旋转45度：使兄弟关系变为右孩子关系





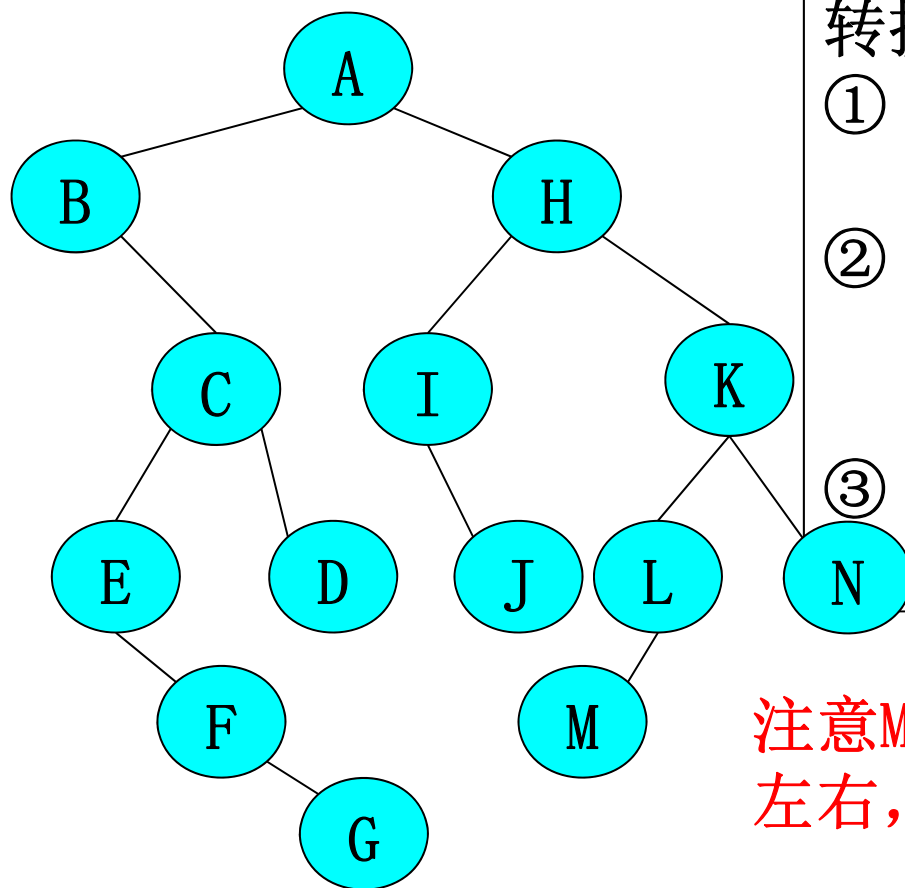
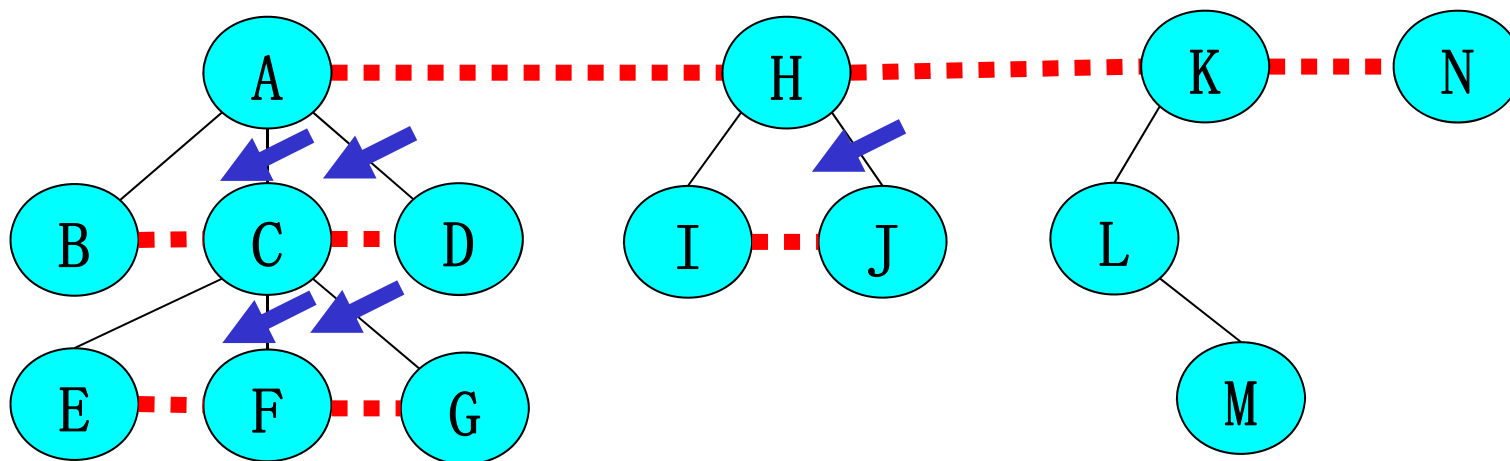
转换方法:

- ① 连兄弟: 所有兄弟间横线相连,
----- (所有树的根为兄弟)



转换方法:

- ① 连兄弟: 所有兄弟间横线相连,
----- (所有树的根为兄弟)
- ② 断孩子: 每个结点只保留与第一个孩子的连线, 断开与
----- 其它孩子的连线



转换方法:

- ① 连兄弟: 所有兄弟间横线相连,
----- (所有树的根为兄弟)
- ② 断孩子: 每个结点只保留与第一个孩子的连线, 断开与
↙ 其它孩子的连线
- ③ 顺时针旋转45度: 使兄弟关系
 变为右孩子关系

注意M: 树的孩子不分
左右, 图示位置表示左右

§ 6. 树和二叉树

6. 4. 树和森林

6. 4. 2. 森林和二叉树的转换

★ 二叉树转森林的递归定义

设 $B = \{\text{root}, \text{LB}, \text{RB}\}$ 为二叉树, $F = \{T_1, T_2, \dots, T_m\}$ 为森林

① 若B为空, 则F为空

② 若B非空, 则F中第一棵树 T_1 的根 $\text{ROOT}(T_1)$ 即为二叉树的根root; T_1 中根结点的子树森林

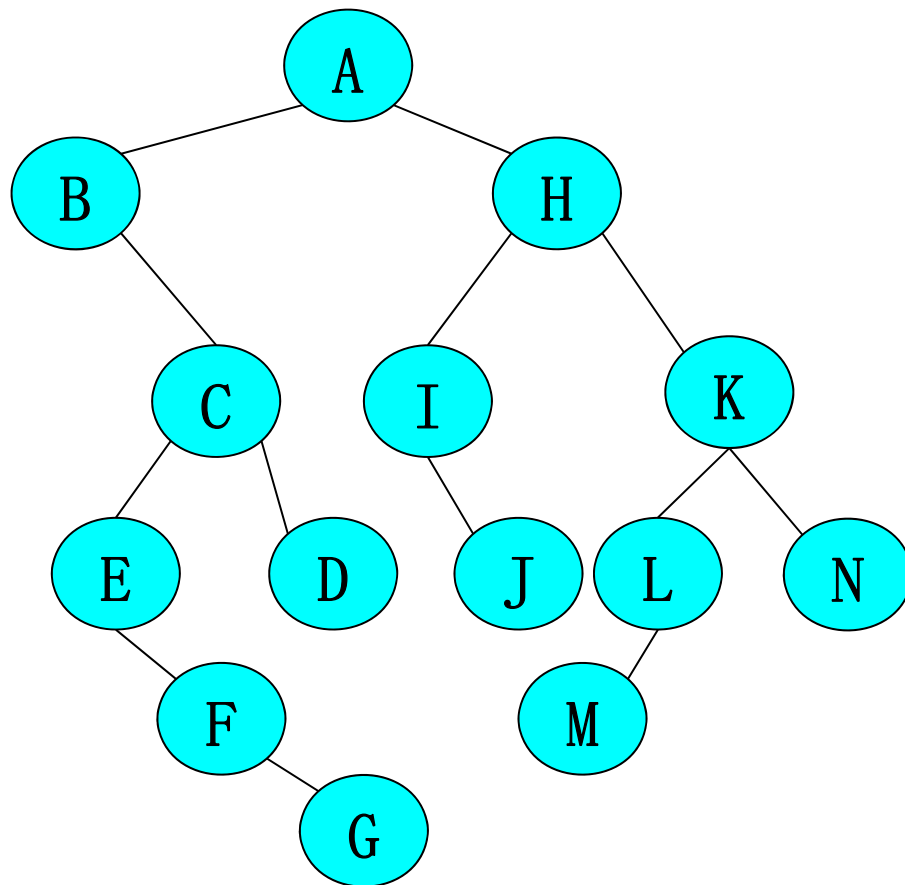
F_1 是由B的左子树LB转换而成的森林; F中除 T_1 之外其余树组成的森林 $F' = \{T_2, T_3, \dots, T_m\}$ 是由B的右子树RB转换而成的森林

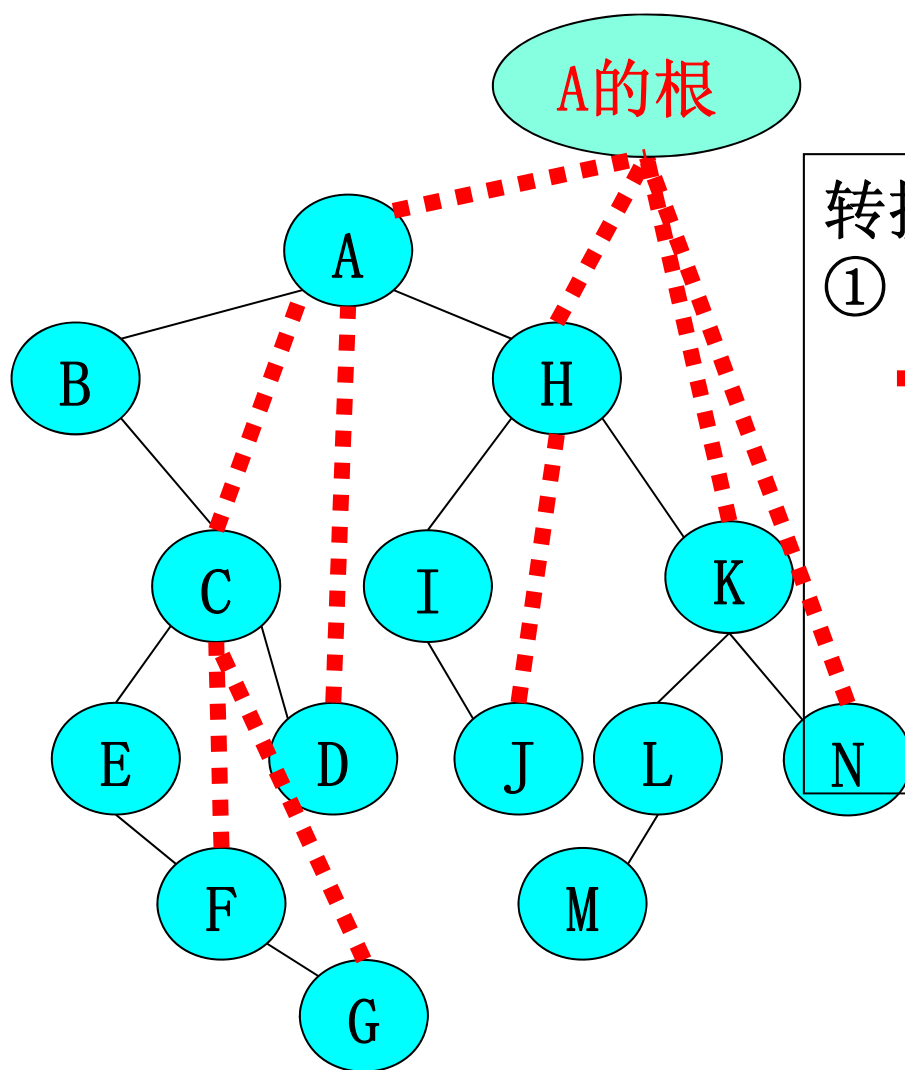
★ 二叉树转森林的转换方法

① 连双亲: 所有结点右链(全部右孩子)与其双亲结点相连(根的所有右孩子指向空)

② 断右链: 断开所有右链(全部右孩子)的连接

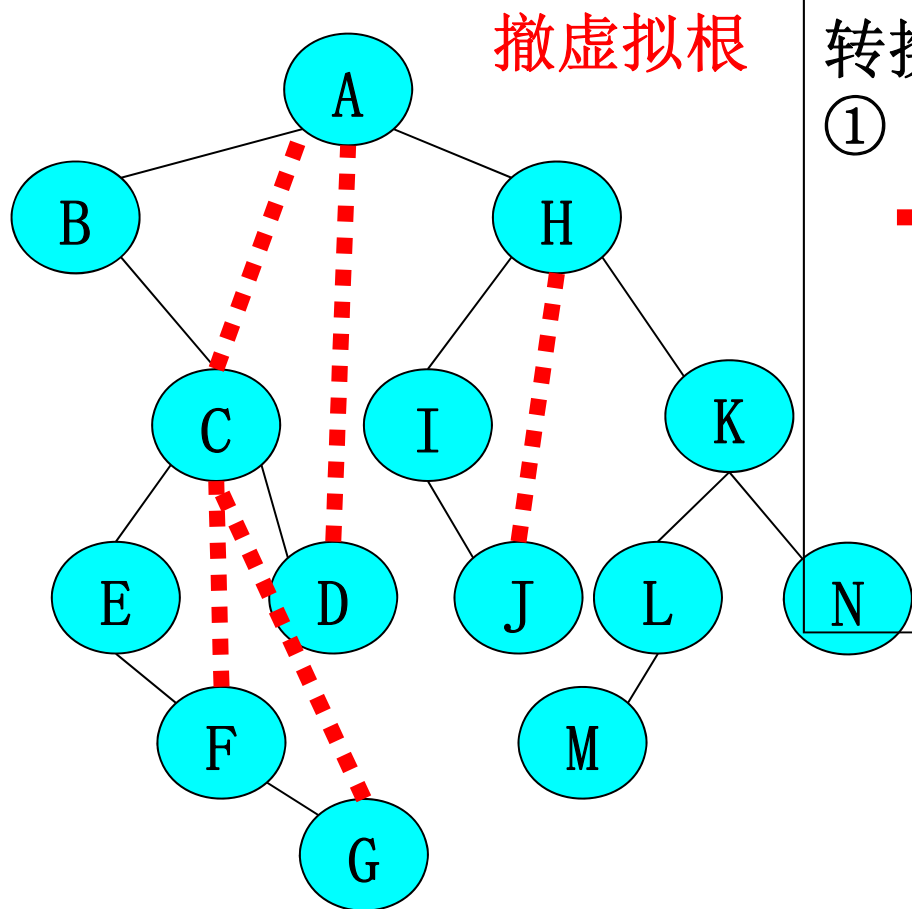
③ 逆时针旋转45度: 使孩子关系变为兄弟关系





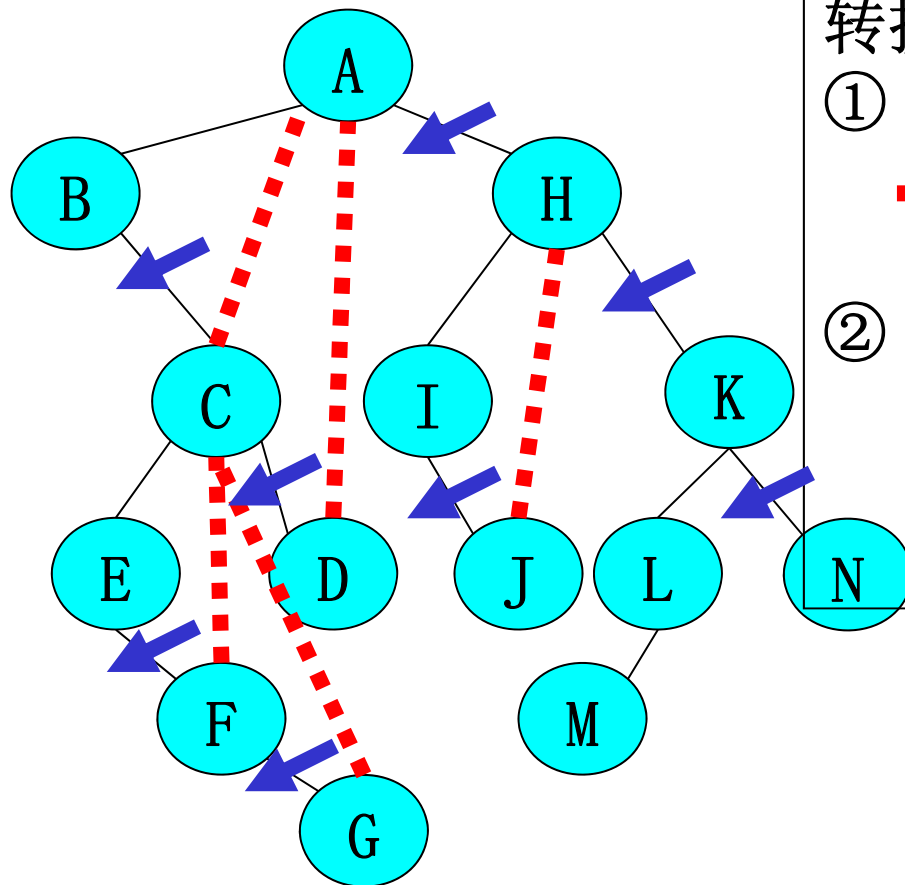
转换方法:

- ① 连双亲: 所有结点右链(全部右孩子)与其双亲结点相连
(根的所有右孩子指向空)



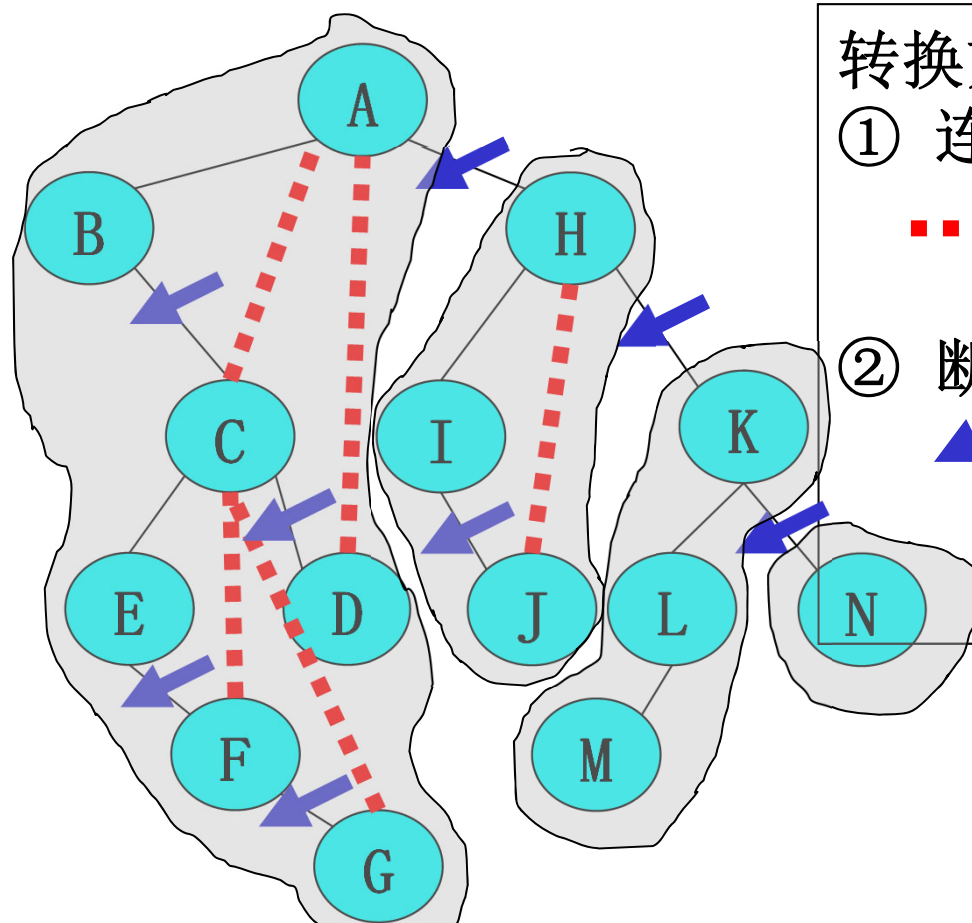
转换方法:

- ① 连双亲: 所有结点右链(全部右孩子)与其双亲结点相连
(根的所有右孩子指向空)



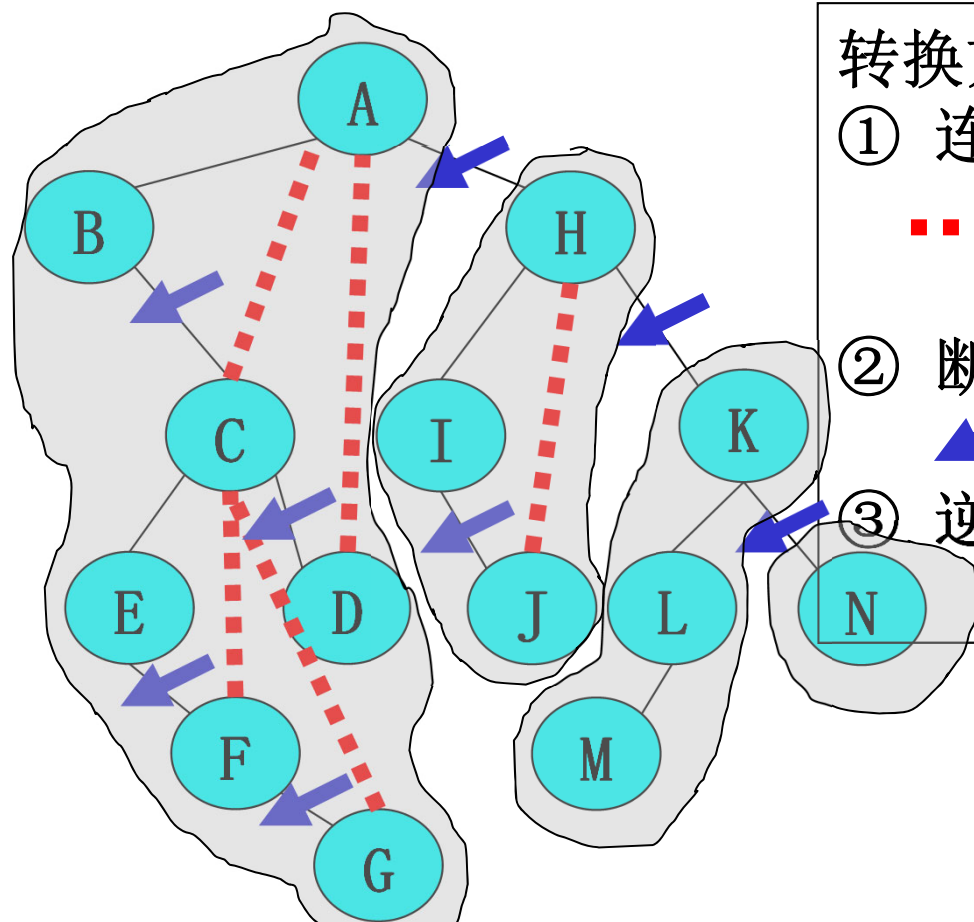
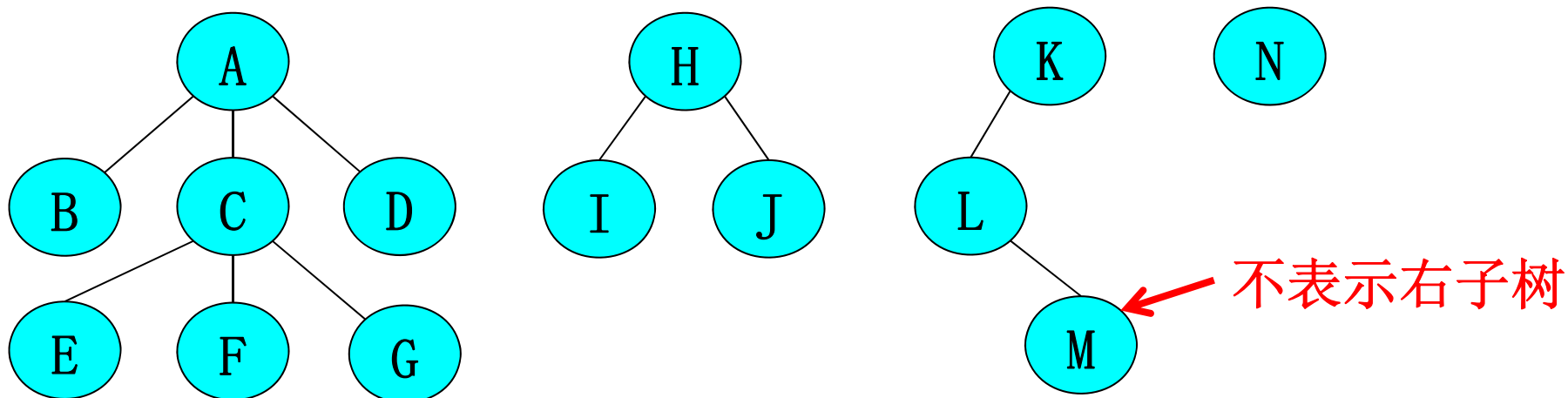
转换方法:

- ① 连双亲: 所有结点右链(全部右孩子)与其双亲结点相连
.....
- ② 断右链: 断开所有右链(全部右孩子)的连接
➡



转换方法:

- ① 连双亲: 所有结点右链(全部右孩子)与其双亲结点相连
..... (根的所有右孩子指向空)
- ② 断右链: 断开所有右链(全部右孩子)的连接
.....



转换方法:

- ① 连双亲: 所有结点右链(全部右孩子)与其双亲结点相连
(根的所有右孩子指向空)
- ② 断右链: 断开所有右链(全部右孩子)的连接
- ③ 逆时针旋转45度: 使孩子关系变为兄弟关系

§ 6. 树和二叉树

6. 4. 树和森林

6. 4. 3. 树和森林的遍历

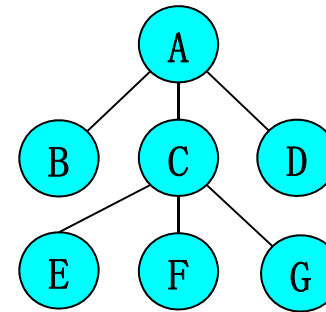
★ 树的遍历:

先序: 根、第一棵子树、第二棵子树、...、第k棵子树

(对应二叉树的先序遍历)

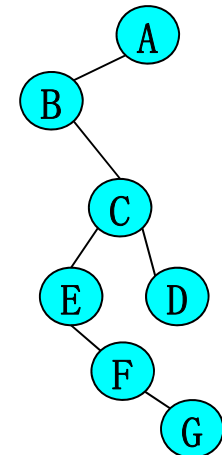
后序: 第一棵子树、第二棵子树、...、第k棵子树根

(对应二叉树的中序遍历)



树的先序遍历: _____
二叉树的先序遍历: _____

树的后序遍历: _____
二叉树的中序遍历: _____



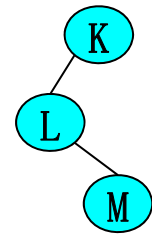
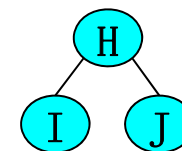
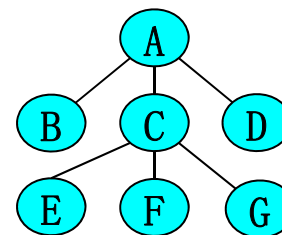
★ 森林的遍历:

先序: 每棵树依次用先序遍历 (对应二叉树的先序)

- ① 访问森林中第一颗树的根结点
- ② 先序遍历第一棵树中根结点的子树森林
- ③ 先序遍历除第一棵树外剩余的子树森林

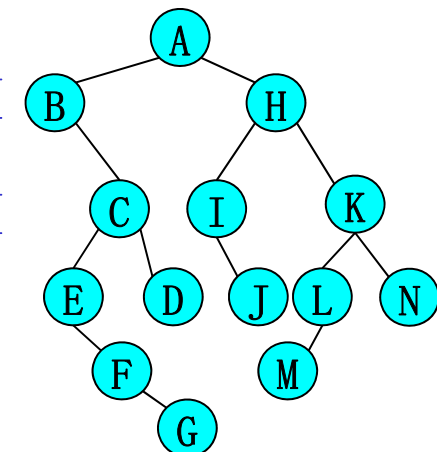
中序: 每棵树依次用后序遍历 (对应二叉树的中序)

- ① 中序遍历第一棵树中根结点的子树森林
- ② 访问森林中第一颗树的根结点
- ③ 中序遍历除第一棵树外剩余的子树森林



森林的先序遍历: _____
二叉树的先序遍历: _____

森林的中序遍历: _____
二叉树的中序遍历: _____



§ 6. 树和二叉树

6.5. 树与等价问题(略)

§ 6. 树和二叉树

6.6. 赫夫曼树及其应用

6.6.1. 最优二叉树(赫夫曼树)

6.6.1.1. 基本概念

★ 路径：从一个结点到另一个结点的连线(唯一)

红：A-H 蓝：F-K

★ 路径长度：路径上分支的数目 (经过的结点数+1)

例：A-H 的长度为3

F-K 的长度为3

★ 树的路径长度：从根到每一个结点的路径长度之和

例：树的路径长度为22 ($2*1+4*2+4*3$)

A : 0

B, C : 1

D, E, F, G : 2

H, I, J, K : 3

★ 结点的权值：根据某种分布给结点所赋的值

某种分布：与具体的应用相关

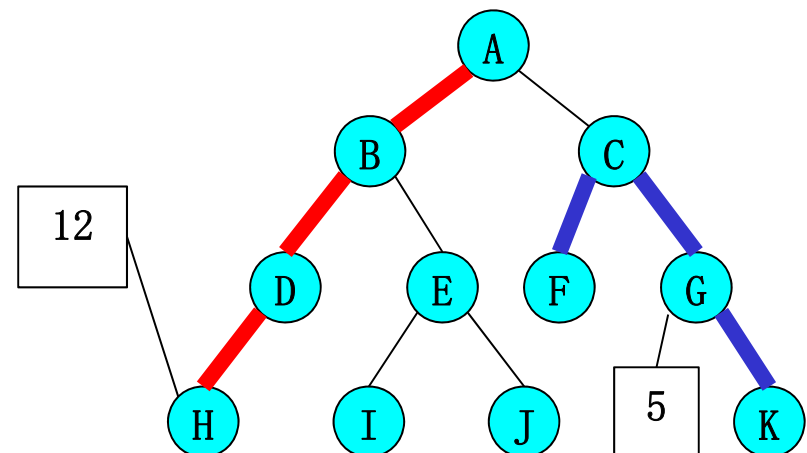
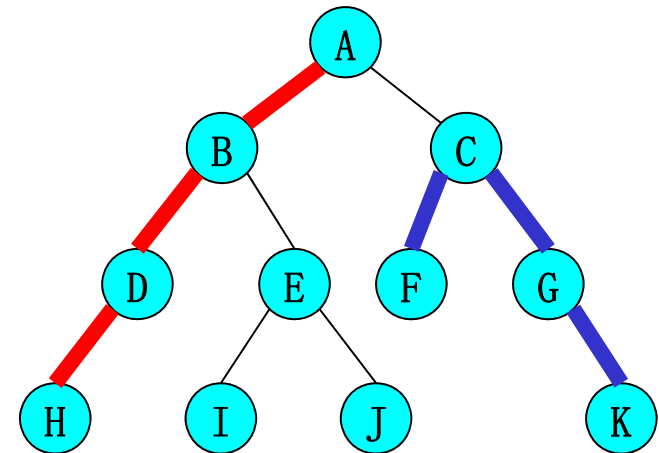
H房子中有12个人

G零件的重量是5吨

★ 结点的带权路径长度：从该结点到根之间路径长度与权值的乘积

例：H 结点的带权路径长度为 $12*3=36$

G 结点的带权路径长度为 $5*2=10$



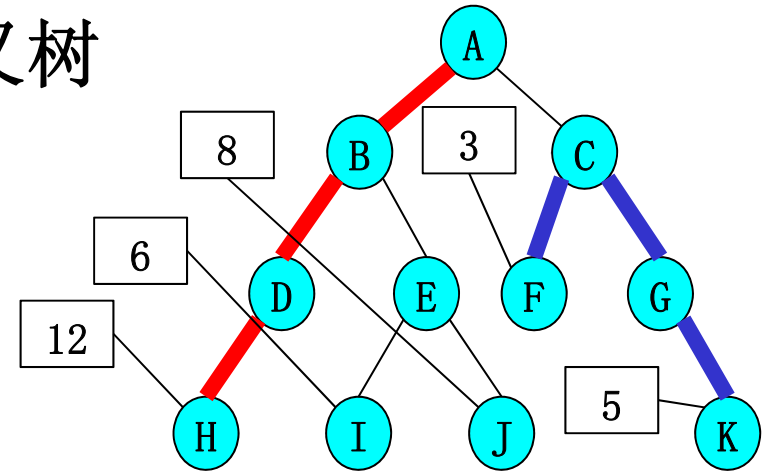
§ 6. 树和二叉树

6.6. 赫夫曼树及其应用

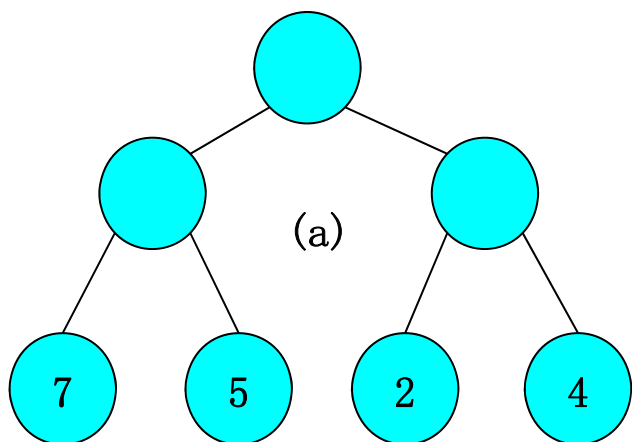
6.6.1. 最优二叉树(赫夫曼树)

6.6.1.1. 基本概念

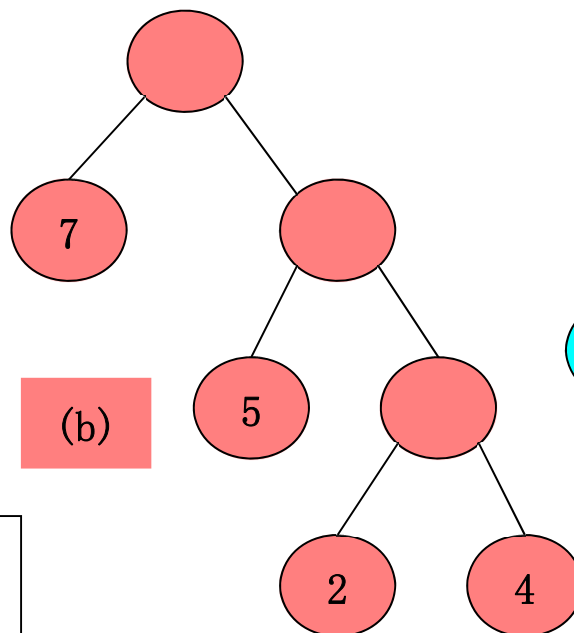
- ★ 路径：从一个结点到另一个结点的连线(唯一)
- ★ 路径长度：路径上分支的数目 (经过的结点数+1)
- ★ 树的路径长度：从根到每一个结点的路径长度之和
- ★ 结点的权值：根据某种分布给结点所赋的值
- ★ 结点的带权路径长度：从该结点到根之间路径长度与权值的乘积
- ★ 树的带权路径长度(WPL)：树中所有叶子结点的带权路径长度之和
例： $WPL = 12 \times 3 + 6 \times 3 + 8 \times 3 + 3 \times 2 + 5 \times 3 = 99$
- ★ 赫夫曼树(最优二叉树)：n个带有权值的结点作为叶子结点构成的不同形式的二叉树中WPL最小的树



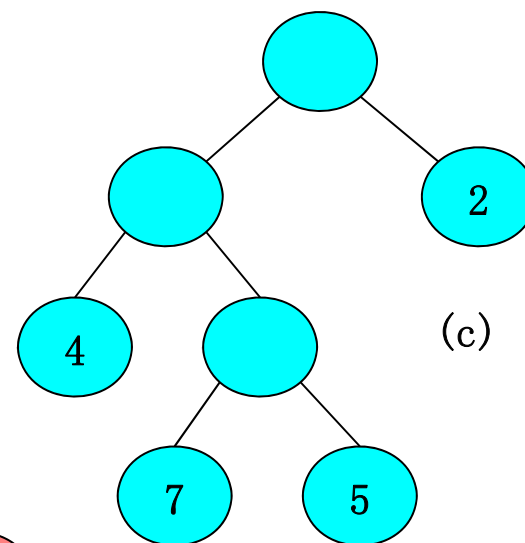
例：4个结点, 权值分别为7, 5, 2, 4 (构造的部分树如下)



(a)



(b)



(c)

$$(a) WPL = 7 \times 2 + 5 \times 2 + 2 \times 2 + 4 \times 2 = 36$$

$$(b) WPL = 7 \times 1 + 5 \times 2 + 2 \times 3 + 4 \times 3 = 35$$

$$(c) WPL = 4 \times 2 + 7 \times 3 + 5 \times 3 + 2 \times 1 = 46$$

注意：4个叶子结点的树，
形态远不止这三种

问题1：还有WPL没有更小的？
问题2：如何构造WPL最小的树？

§ 6. 树和二叉树

6.6. 赫夫曼树及其应用

6.6.1. 最优二叉树(赫夫曼树)

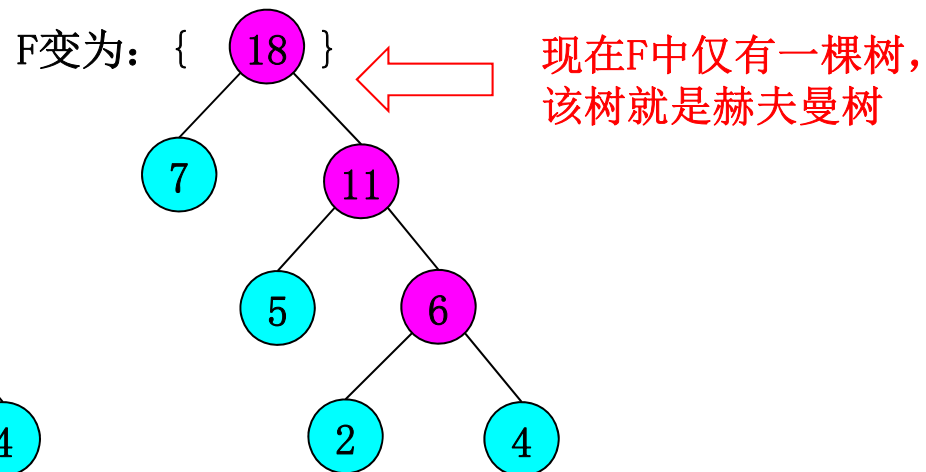
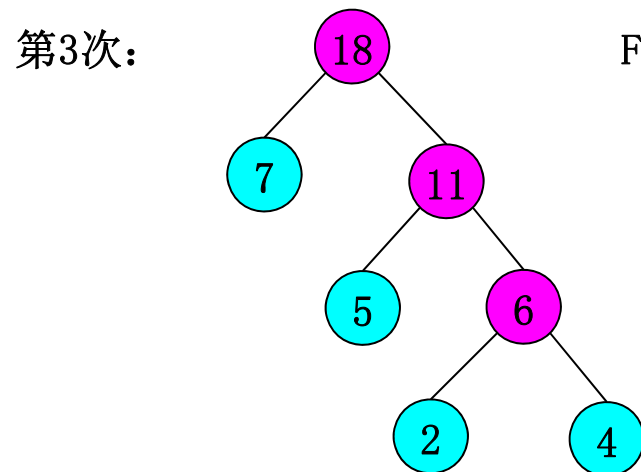
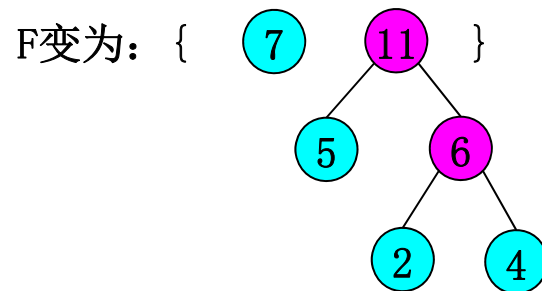
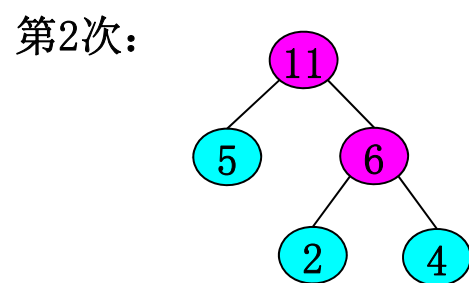
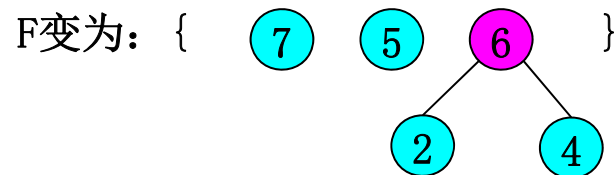
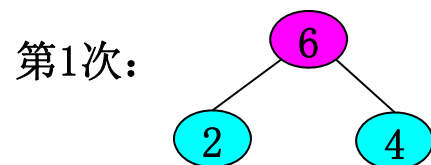
6.6.1.1. 基本概念

6.6.1.2. 赫夫曼树的构造 (赫夫曼算法的描述)

- ① n 个带权结点, 构成 n 棵只有根结点的二叉树的森林集合, 根结点的权值即为带权结点的权值 $F = \{ T_1, T_2, T_3, \dots, T_n \}$
- ② 在 F 中选取根结点权值最小的两棵子树 $T_i, T_j (i \neq j)$ 作为左、右子树构造新的二叉树 T' ; 其根结点的权值为左右子树之和
- ③ 在 F 中删除 T_i, T_j , 将 T' 加入
- ④ 重复步骤②③至 F 中仅有一棵树, 该树即为赫夫曼树

例：将7，5，4，2分别作为树的叶子结点的权值，
则构造过程为：

初始：F={ 7 5 2 4 }



§ 6. 树和二叉树

6.6. 赫夫曼树及其应用

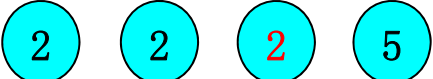
6.6.1. 最优二叉树(赫夫曼树)

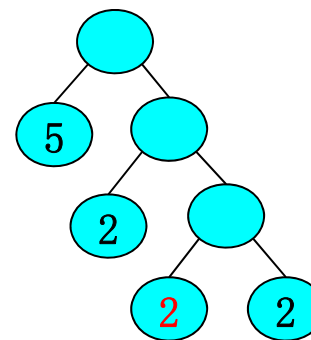
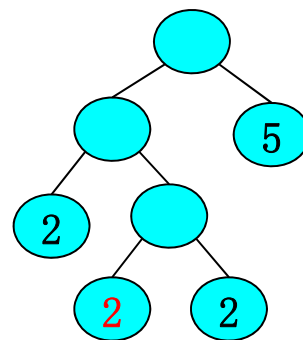
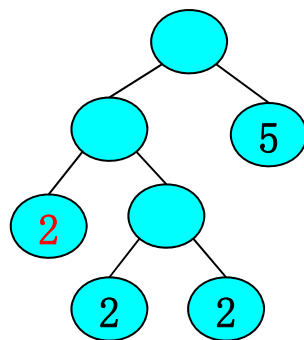
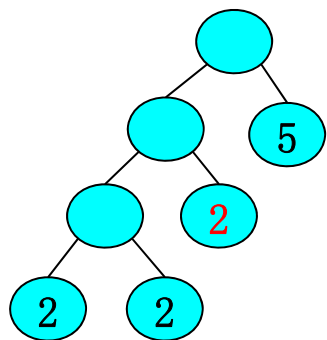
6.6.1.1. 基本概念

6.6.1.2. 赫夫曼树的构造 (赫夫曼算法的描述)

★ 由赫夫曼树的构造过程可知，赫夫曼树中不存在度为1的结点，即赫夫曼树是**正则二叉树**

★ 赫夫曼树的形态可能有多种，但WPL都相同

例：  下面4种WPL=21, 都是赫夫曼树



如果是考试，看清楚是否有左值比右值小之类的限定要求!!!

§ 6. 树和二叉树

6.6. 赫夫曼树及其应用

6.6.1. 最优二叉树(赫夫曼树)

6.6.1.1. 基本概念

6.6.1.2. 赫夫曼树的构造 (赫夫曼算法的描述)

6.6.1.3. 赫夫曼树的应用 - 判定问题求最佳算法

例：百分制的等级转换规则如下：

90~100 : 优

80~89 : 良

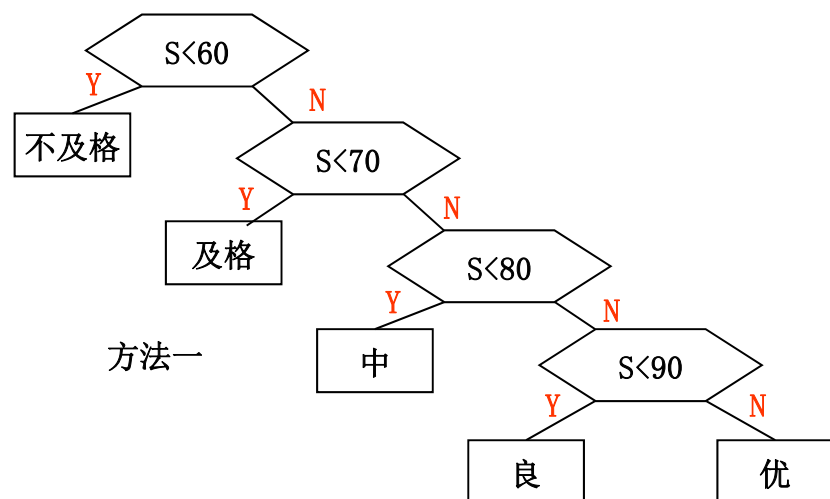
70~79 : 中

60~69 : 及格

0~59 : 不及格

要求根据输入的百分制分数，输出转换的结果

解：判定算法可以有若干种，以下是其中的三种
(假设用S表示成绩，且保证值合法(0~100之间))

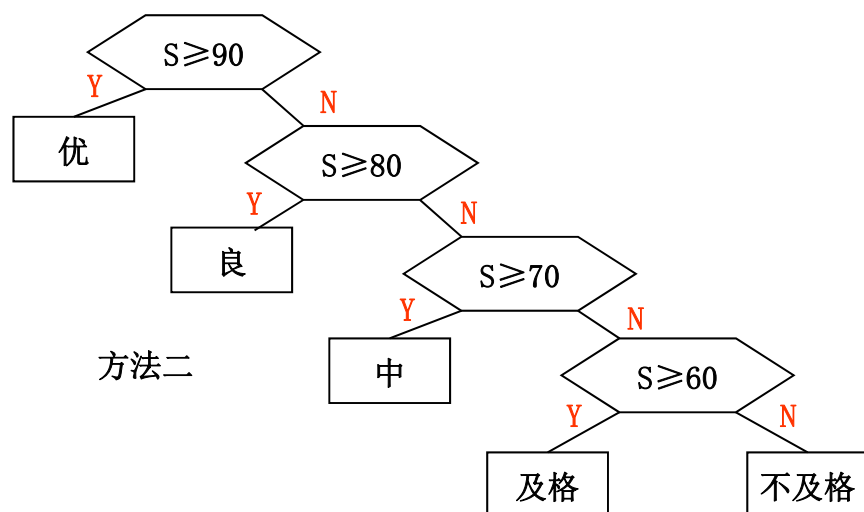


方法一对应的程序段：

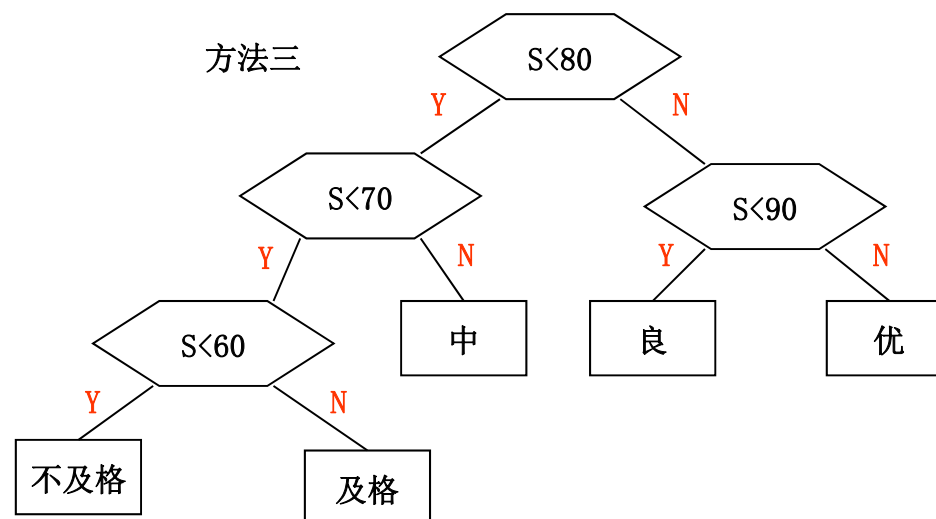
```
if (s<60)
    cout << "不及格";
else if (s<70)
    cout << "及格";
else if (s<80)
    cout << "中";
else if (s<90)
    cout << "良";
else
    cout << "优";
```

问题：1、哪个方法是正确的？
2、哪个方法效率最高？

结论：1、三种方法都是正确的
2、对于大批量的数据，
因为分布规律不均匀，
三种方法效率不同的
(算法效率依赖于数据)

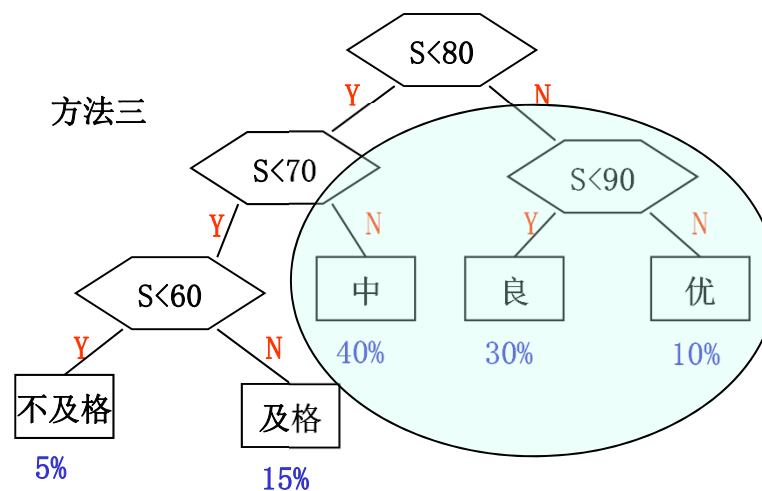
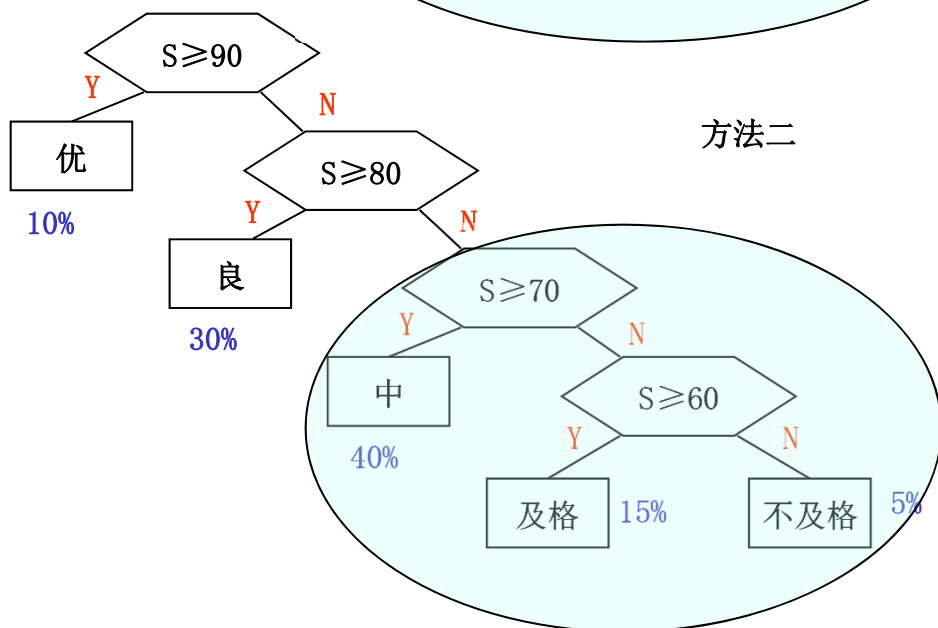
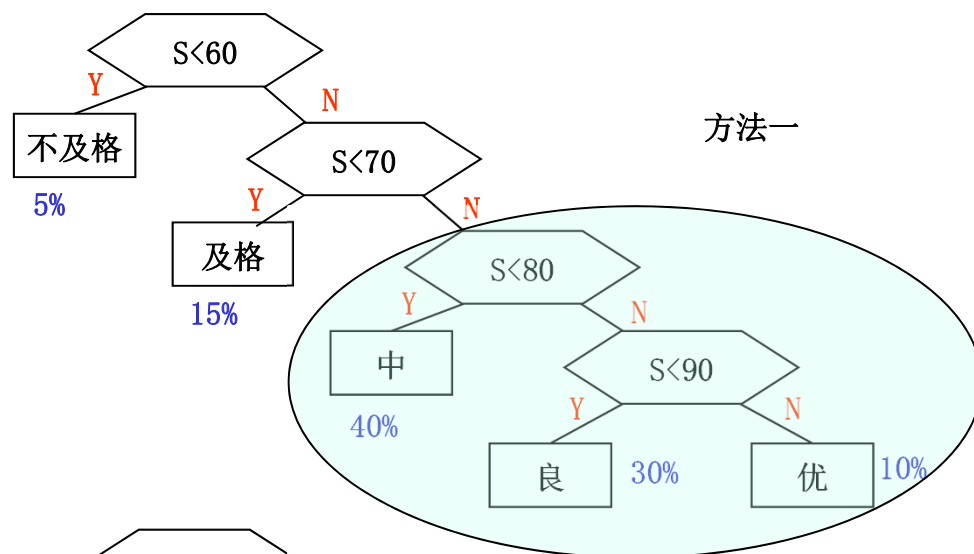


方法三



假设共10000个输入数据，按以下比例分布：

分数	0~59	60~69	70~79	80~89	90~100
比例	5%	15%	40%	30%	10%



方法一：80%的数据需三次以上的比较才能得到
共需 31500次比较

$(500 \times 1 + 1500 \times 2 + 4000 \times 3 + 3000 \times 4 + 1000 \times 4)$
用分布规律的百分比做判定树叶子结点的权值，则
 $WPL = 5 \times 1 + 15 \times 2 + 40 \times 3 + 30 \times 4 + 10 \times 4 = 315$

方法二：60%的数据需三次以上的比较才能得到
共需27000次比较

$(1000 \times 1 + 3000 \times 2 + 4000 \times 3 + 1500 \times 4 + 500 \times 4)$
 $WPL = 10 \times 1 + 30 \times 2 + 40 \times 3 + 15 \times 4 + 5 \times 4 = 270$

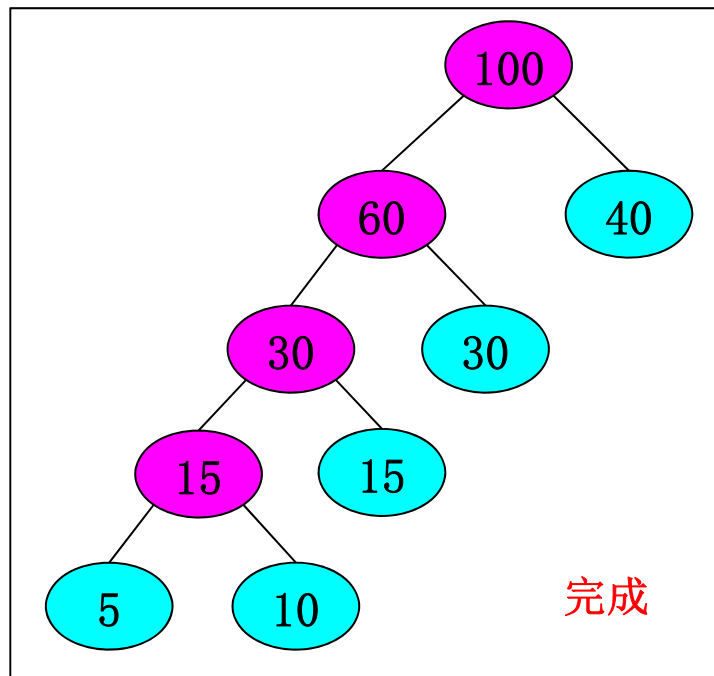
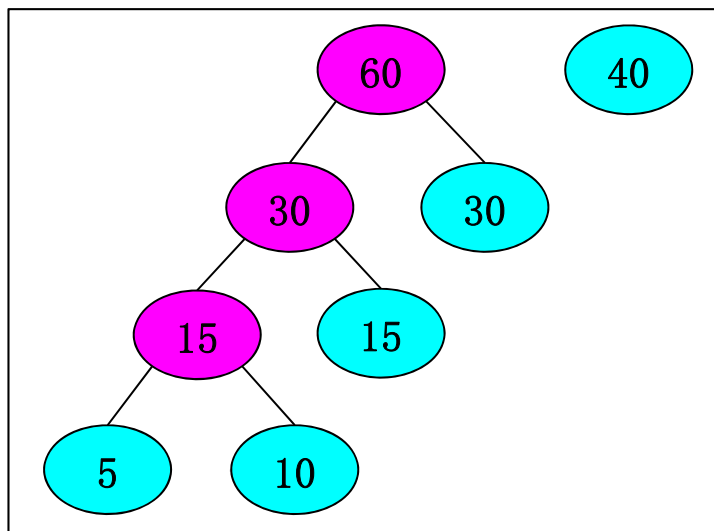
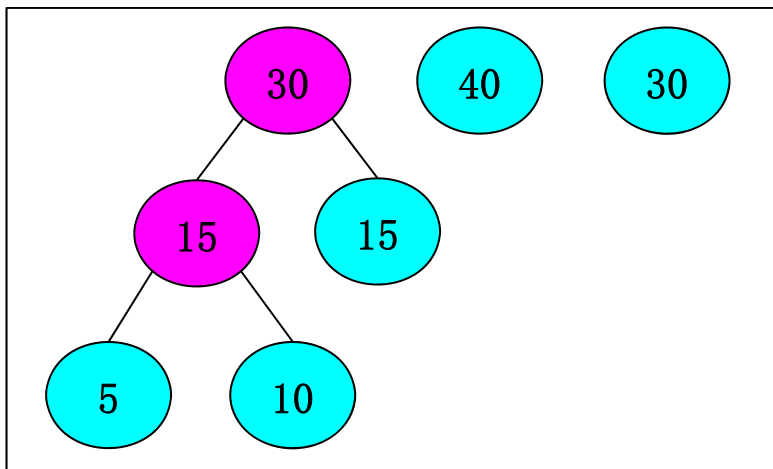
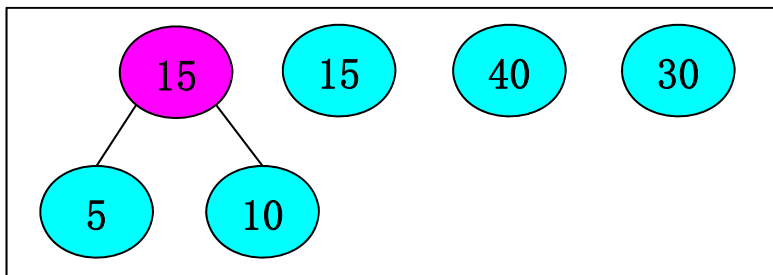
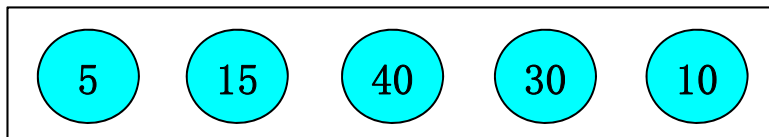
方法三：80%的数据仅需两次比较就可得到
共需22000次比较

$(500 \times 3 + 1500 \times 3 + 4000 \times 2 + 3000 \times 2 + 1000 \times 2)$
 $WPL = 5 \times 3 + 15 \times 3 + 40 \times 2 + 30 \times 2 + 10 \times 2 = 220$

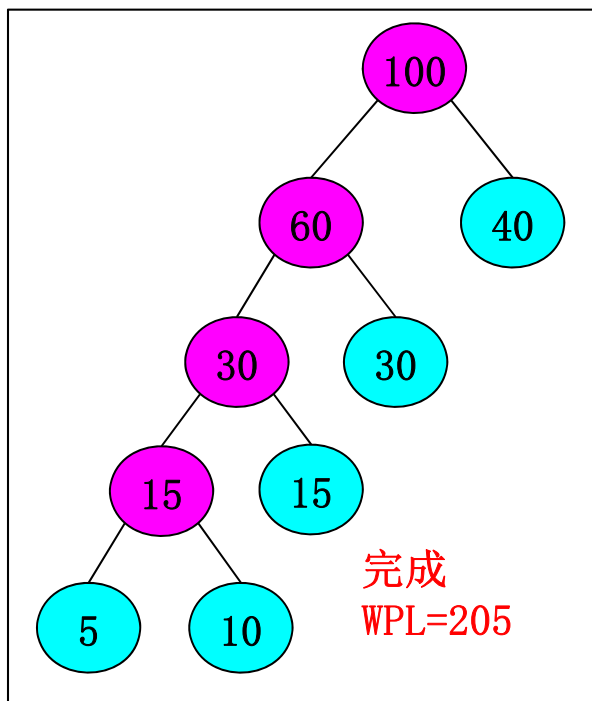
结论：WPL的值越小，算法的效率越高

推论：赫夫曼树的效率最高

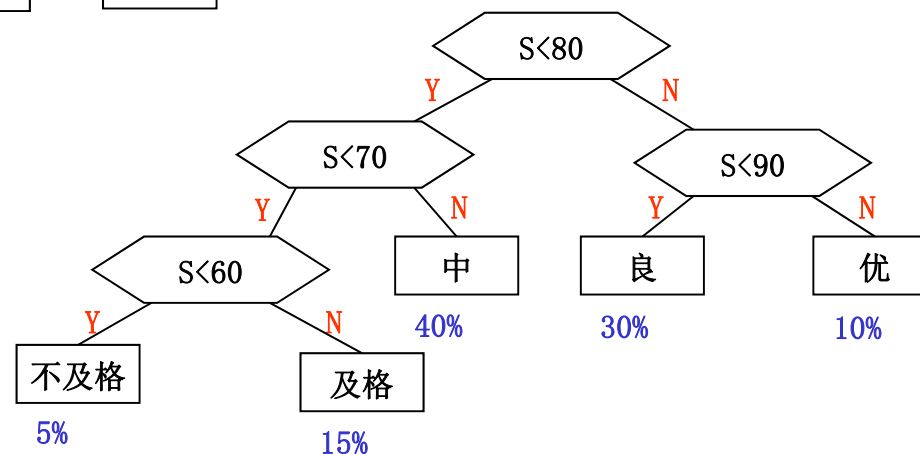
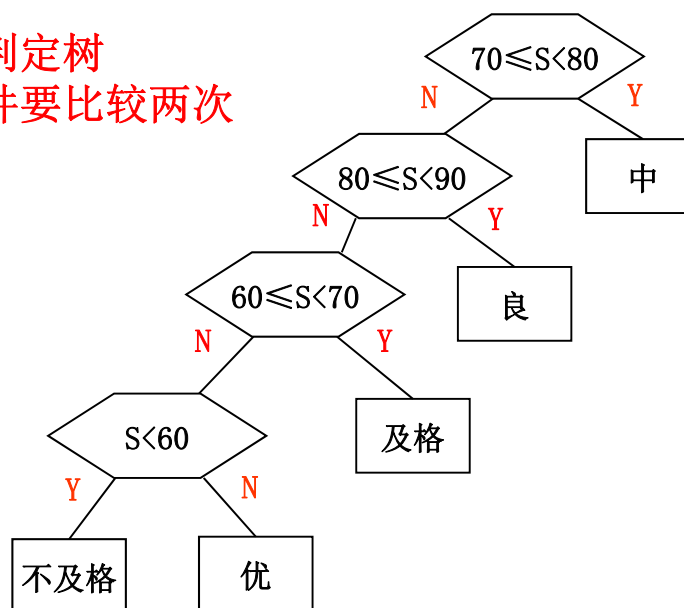
将5, 15, 40, 30, 10分别作为判定树的叶子结点的权值, 构造赫夫曼树



将5, 15, 40, 30, 10分别作为判定树的叶子结点的权值, 构造赫夫曼树



对应的判定树
三个条件要比较两次



方法三: 在赫夫曼树基础上改为单条件, WPL=220
最优解 => 次优解

§ 6. 树和二叉树

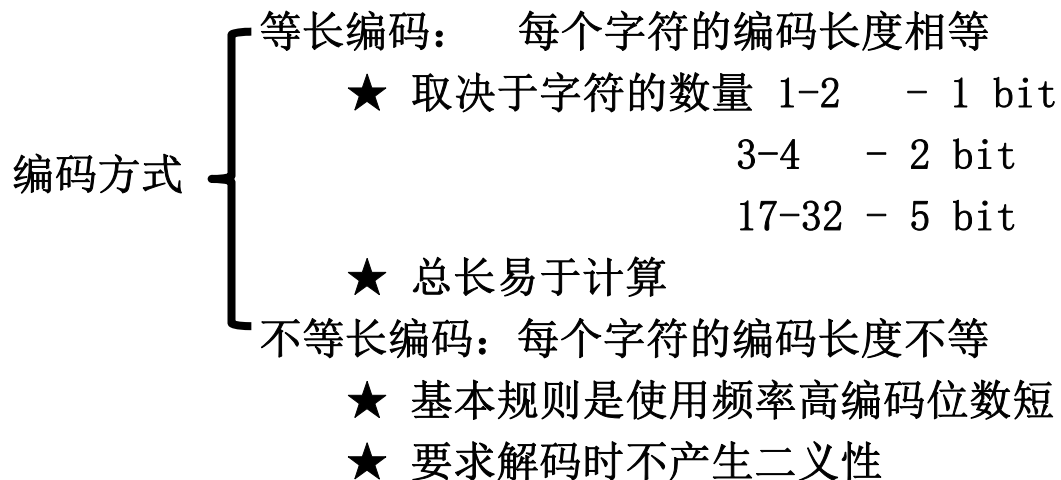
6.6. 赫夫曼树及其应用

6.6.1. 最优二叉树(赫夫曼树)

6.6.2. 赫夫曼编码

6.6.2.1. 引入

字符通过编码方式传送, 在其它条件都相同的情况下, 编码总长越短则传输时间越短



00如何译码?
方法1: aa或b
方法2: aa或c

例: 序列aadbcbadcbababab, 采用编码方式传送
(a-7次 b-5次 c-2次 d-4次)

- ★ 等长编码 4个不同字符, 2bit编码即可

假设编码为: a-00 b-01 c-10 d-11
总编码: 000011011001000011100100010011010011
总编码长度: $18 \times 2 = 36$
译码方法: 每2位对应一个字符

- 换为 a-01 b-10 c-11 d-00, 编码变总长不变

例: 序列aadbcbadcbababab, 采用编码方式传送
(a-7次 b-5次 c-2次 d-4次)

- ★ 不等长编码

方法一: 假设编码为: a-0 b-00 c-1 d-01
总编码: 000100100000110000000100001
总编码长度: $7 \times 1 + 5 \times 2 + 2 \times 1 + 4 \times 2 = 27$

方法二: 假设编码为: a-0 b-1 c-00 d-01
总编码: 000110010001001010011001
总编码长度: $7 \times 1 + 5 \times 1 + 2 \times 2 + 4 \times 2 = 24$

§ 6. 树和二叉树

6.6. 赫夫曼树及其应用

6.6.1. 最优二叉树(赫夫曼树)

6.6.2. 赫夫曼编码

6.6.2.1. 引入

6.6.2.2. 前缀编码

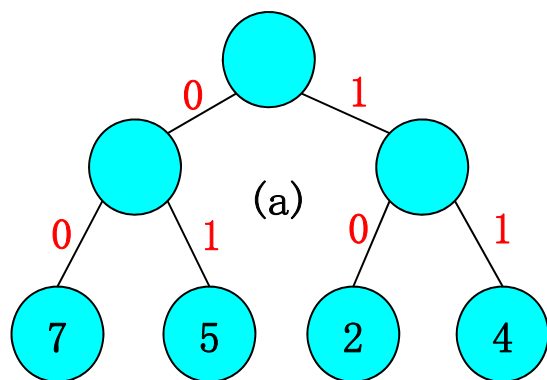
任一编码不是另一个编码的前缀

★ a-0 b-00: 不是前缀编码, 所以会产生二义性

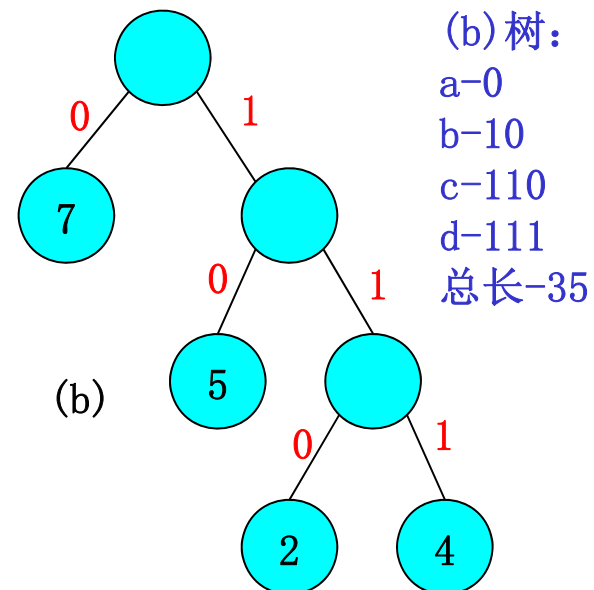
★ 前缀编码的设计方法:

所有字符做为二叉树的**叶子结点**, 二叉树的左分支编码为0, 右分支编码为1, 则从**根到叶子**的路径上的编码组合在一起即为该字符的编码

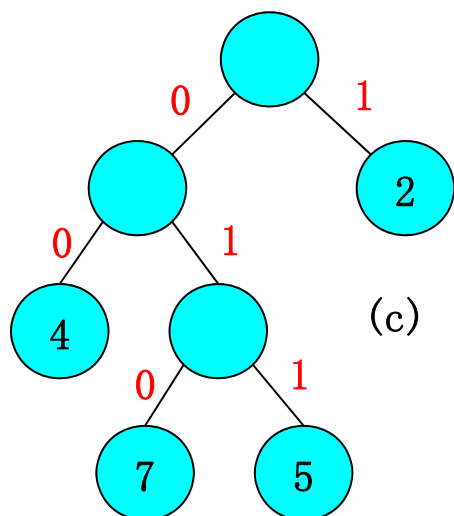
例：aadbcbadcbababdbad(a-7次 b-5次 c-2次 d-4次)



(a) 树:
a-00
b-01
c-10
d-11
总长-36



(b) 树:
a-0
b-10
c-110
d-111
总长-35



(c) 树:
a-010
b-011
c-1
d-00
总长-46

三种情况都是前缀编码

可举例观察译码时的二义性

(a) $WPL=7*2+5*2+2*2+4*2=36$

(b) $WPL=7*1+5*2+2*3+4*3=35$

(c) $WPL=4*2+7*3+5*3+2*1=46$

注意：4个叶子结点的树，
形态远不止这三种

§ 6. 树和二叉树

6.6. 赫夫曼树及其应用

6.6.1. 最优二叉树(赫夫曼树)

6.6.2. 赫夫曼编码

6.6.2.1. 引入

6.6.2.2. 前缀编码

6.6.2.3. 赫夫曼编码

假设编码中有 n 种字符，每个字符出现的频率为 w_i ，该字符的编码长度为 l_i ，则电文总长为 $\sum_{i=1}^n w_i l_i$ ，对应到二叉树，设 n 个结点为叶子结点， w_i 是叶子节点的权值， l_i 为从根到叶子的路径长度，则 $\sum_{i=1}^n w_i l_i$ 就是树的带权路径长度（WPL）

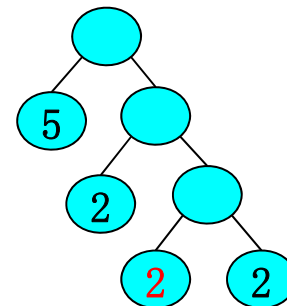
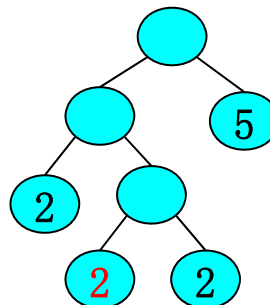
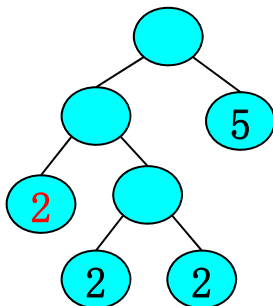
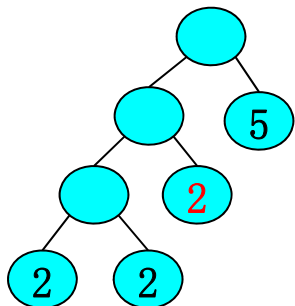
赫夫曼树是WPL最小的树，即编码总长度最短

以赫夫曼树形式构造的前缀编码就是赫夫曼编码

★ 不定长编码中总码长最短

★ 赫夫曼编码可能有多种（总码长，即WPL都相同）

例： 2 2 2 5 下面4种WPL=21，编码不同



§ 6. 树和二叉树

6.6. 赫夫曼树及其应用

6.6.1. 最优二叉树(赫夫曼树)

6.6.2. 赫夫曼编码

6.6.2.1. 引入

6.6.2.2. 前缀编码

6.6.2.3. 赫夫曼编码

★ 赫夫曼树赫夫曼编码的存储表示

★ 赫夫曼编码的编码实现

★ 赫夫曼编码的译码实现

★ 赫夫曼树赫夫曼编码的存储表示(P. 147)

/* 定义赫夫曼树的结点 */

```
typedef struct {  
    unsigned int weight; //权值  
    unsigned int parent, lchild, rchild; //下标  
} HTNode, *HuffmanTree;
```

/* 存储赫夫曼编码的编码表 */

```
typedef char ** HuffmanCode; //二级指针
```

★ 赫夫曼编码的编码实现 (P. 147 算法 6.12)

```
void HuffmanCoding(HuffmanTree &HT, HuffmanCode &HC, int *w, int n)
{ //HT : 赫夫曼树
  //HC : 赫夫曼编码
  //w : int型数组, 存放n的字符的出现频率表, { 7, 5, 2, 3, ... }形式
  //n : 字符的数量
```

```
HuffmanTree p;           局部变量定义, 书上略
int      i, m, start, sl, s2, c, f;
char     *cd;
```

```
if (n<=1) //仅一个结点无法构造赫夫曼树
    return;
```

```
m=2*n-1; //n是字符数, 即叶子结点数n0, n0=n2+1, 无n1, 总数=2n-1
HT = (HuffmanTree)malloc((m+1)*sizeof(HTNode)); // [0]不用
```

```
for (p = HT+1, i=1; i<=n; i++, p++, w++) {
    p->weight = *w;
    p->parent = 0;
    p->lchild = 0;
    p->rchild = 0;
}
```

HT的前n个结点赋值 (当做叶子用)
书上直接 *p = { *w, 0, 0, 0 } 代替
p=HT 不行, 因为HT[0]不用

```
for (; i<=m; i++, p++) {
    p->weight = 0;
    p->parent = 0;
    p->lchild = 0;
    p->rchild = 0;
}
```

HT的后n-1个结点赋值 (当非叶子用)
书上直接 *p = { 0, 0, 0, 0 } 代替

执行完这两个循环后, 结果可参考P. 149 图6.27(a)

已知8种字符的出现概率为0.05, 0.29, 0.07, 0.08, 0.14, 0.23, 0.03, 0.11, 请设计赫夫曼编码

0不用	weight	parent	lchild	rchild
1	5	0	0	0
2	29	0	0	0
3	7	0	0	0
4	8	0	0	0
5	14	0	0	0
6	23	0	0	0
7	3	0	0	0
8	11	0	0	0
9	0	0	0	0
10	0	0	0	0
11	0	0	0	0
12	0	0	0	0
13	0	0	0	0
14	0	0	0	0
15	0	0	0	0

★ 赫夫曼编码的编码实现 (P. 147 算法 6.12)

```
void HuffmanCoding(HuffmanTree &HT, HuffmanCode &HC, int *w, int n)
{
    ...

```

/* n+1到m都是用做非叶结点的，依次使用即可 */

```
for (i=n+1; i<=m; i++) {
```

/* Select 是在HT[1..i-1]中选择parent为0条件下的权值最小结点
返回的下标在s1, s2中 (需要单独给出其实现) */

```
Select(HT, i-1, s1, s2);
```

/* 第i个结点称为s1, s2的父结点，权值为两者之和
s1/s2的parent被置为i，下次不会被Select选中 */

```
HT[s1].parent = i;
```

```
HT[s2].parent = i;
```

```
HT[i].lchild = s1;
```

```
HT[i].rchild = s2;
```

```
HT[i].weight = HT[s1].weight + HT[s2].weight;
```

```
}
```

```
HC = (HuffmanCode)malloc((n+1)*sizeof(char *)); //n+1, [0]不用
```

```
cd = (char *)malloc(n * sizeof(char)); //长度为n，当临时字符串
```

```
cd[n-1] = '\0'; //注意，书上是双引号，算法可以，程序不行
```

```
for (i=1; i<=n; i++) { //对前n个结点（叶子结点）循环
```

```
start = n-1; //从叶子结点开始逆序，所以start指向 '\0'
```

```
for (c=i, f=HT[i].parent; f!=0; c=f, f=HT[f].parent)
```

```
if (HT[f].lchild==c) //判断左右子树分取01
```

```
cd[--start] = '0'; //在临时空间的尾部填充
```

```
else
```

```
cd[--start] = '1'; //尾部填充
```

循环结束条件f!=0

表示没到根结点

```
HC[i]=(char *)malloc((n-start)*sizeof(char)); //按实际度申请
```

```
strcpy(HC[i], &cd[start]); //从临时空间复制到 HC[i] 中
```

```
}
```

```
free(cd);
```

```
}
```

执行完这个循环后，结果
可参考P. 149 图6.27(b)

执行完这部分后，结果可参考P. 149 图6.27(c)

已知8种字符的出现概率为0.05, 0.29, 0.07, 0.08,
0.14, 0.23, 0.03, 0.11，请设计赫夫曼编码

0不用	weight	parent	lchild	rchild
1	5	9	0	0
2	29	14	0	0
3	7	10	0	0
4	8	10	0	0
5	14	12	0	0
6	23	13	0	0
7	3	9	0	0
8	11	11	0	0
9	8	11	1	7
10	15	12	3	4
11	19	13	8	9
12	29	14	5	10
13	42	15	6	11
14	58	15	2	12
15	100	0	13	14

1	—→	0	1	1	0
2	—→	1	0		
3	—→	1	1	1	0
4	—→	1	1	1	1
5	—→	1	1	0	
6	—→	0	0		
7	—→	0	1	1	1
8	—→	0	1	0	

★ 赫夫曼编码的编码实现(P. 147 算法 6. 12)

```
void HuffmanCoding(HuffmanTree &HT, HuffmanCode &HC, int *w, int n)
```

```
{
```

```
    //刚才就是从叶子到根逆向求每个字符的赫夫曼编码，这部分的实现还可以通过正向遍历赫夫曼树来实现，即 P. 148 算法 6. 13
```

```
    HC = (HuffmanCode)malloc((n+1)*sizeof(char *)); //n+1, [0]不用
```

```
    cd = (char *)malloc(n * sizeof(char)); //长度为n，当临时字符串
```

```
    p = m; //p初始指向第m个结点，即根结点
```

```
    cdlen = 0;
```

```
    for(i=1; i<=m; i++)
```

```
        HT[i].weight = 0; //赫夫曼树构造完成后权值已无用，借来做标记
```

```
    /* 从根结点出发，循环求各结点的赫夫曼编码 */
```

```
    while(p) {
```

```
        if (HT[p].weight == 0) { //==0表示没被遍历过
```

```
            HT[p].weight = 1; //置1表示已访问过左分支
```

```
            if (HT[p].lchild != 0) { //lchild!=0表示还没到叶子
                p = HT[p].lchild; //此时需要继续向左
                cd[cdlen++] = '0'; //同时编码值0
            }
```

```
            else if (HT[p].rchild == 0) { //左右孩子都是0，p是叶子
                HC[p] = (char *)malloc((cdlen+1)*sizeof(char));
                cd[cdlen] = '\0';
                strcpy(HC[p], cd); //复制完后，cd值仍继续要用
            }
```

```
        }
```

```
        else if (HT[p].weight==1) { //==1表示左分支已被访问过
```

```
            HT[p].weight = 2; //置2表示已访问过右分支
```

```
            if (HT[p].rchild != 0) { //rchild!=0表示还没到叶子
                p = HT[p].rchild; //此时需要继续向右
                cd[cdlen++] = '1'; //同时编码值1
            }
```

```
        }
```

```
        else { //已经置2的，则回退一层，以备下次访问
```

```
            HT[p].weight = 0; //置0，下次又可以先左后右
```

```
            p = HT[p].parent; //回退一层
```

```
            cdlen --; //编码-1
```

```
        }
```

```
    } // end of while
```

```
    free(cd);
```

```
}
```

★ 赫夫曼编码的译码实现(通过作业完成)

已知: $a=0110$ $b=***$

收到的编码为: $0110*****$

求译码结果 $a****$

§ 6. 树和二叉树

6.7. 回溯法与树的遍历 (略)

6.8. 树的计数 (略)