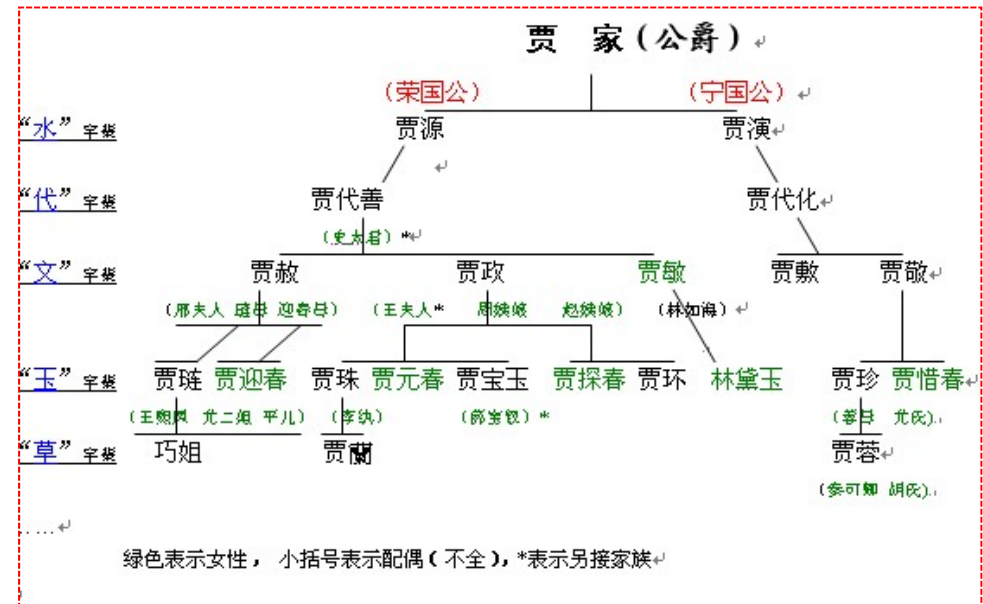


第六章 树和二叉树

线性结构：线性表, 栈, 队列
串, 数组, 广义表

非线性结构：树和二叉树
图, 网



树/层次关系结构的典型实例：

人物家谱结构、社会组织结构、文件目录结构、html文档结构

6.1 树的定义

6.1.1 定义和术语

1. 树 (tree):

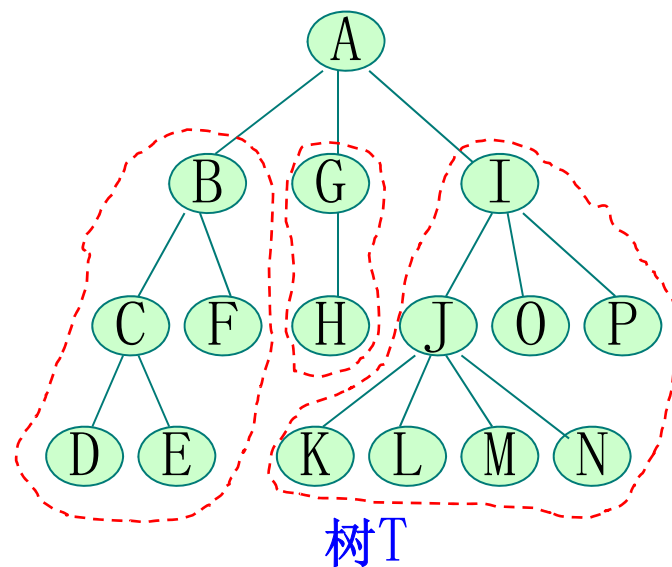
树是 n ($n \geq 0$) 个结点的有限集 T ,

当 $n=0$ 时, T 为空树;

当 $n>0$ 时,

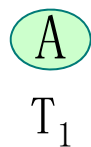
(1) 有且仅有一个称为 T 的**根**的结点,

(2) 当 $n>1$ 时, 余下的结点分为 m ($m>0$) 个互不相交的有限集 T_1, T_2, \dots, T_m , 每个 T_i ($1 \leq i \leq m$) 也是一棵树, 且称为根的**子树**。



例1. 一个结点的树

$$T_1 = \{A\}$$



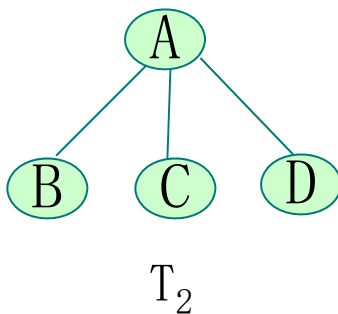
例2. 四个结点的树

$$T_2 = \{A, B, C, D\}$$

$$T_{21} = \{B\}$$

$$T_{22} = \{C\}$$

$$T_{23} = \{D\}$$



例3. 有16个结点的树

$T = \{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P\}$

$T_1 = \{B, C, D, E, F\}$

$T_{11} = \{C, D, E\}$

$T_{111} = \{D\}$

$T_{112} = \{E\}$

$T_{12} = \{F\}$

$T_2 = \{G, H\}$

$T_{21} = \{H\}$

$T_3 = \{I, J, K, L, M, N, O, P\}$

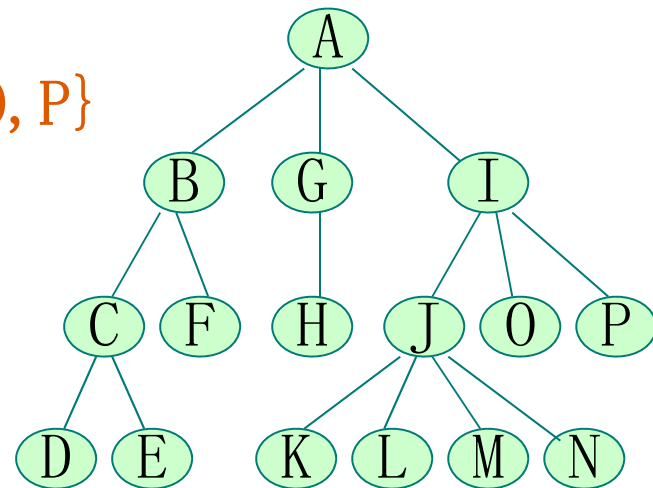
$T_{31} = \{J, K, L, M, N\}$

$T_{311} = \{K\}$

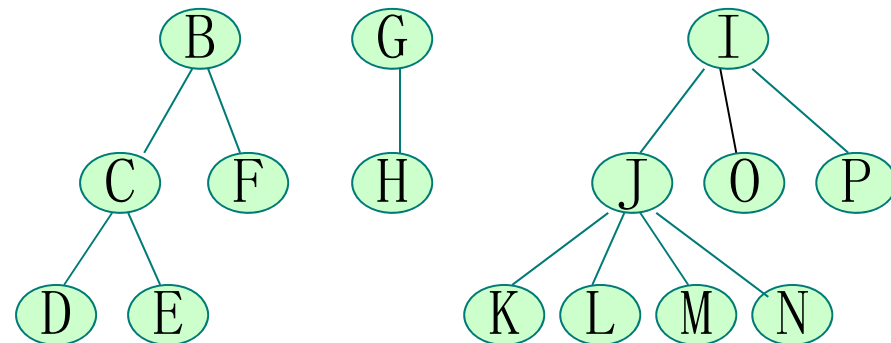
$T_{312} = \{L\} \quad \dots$

$T_{32} = \{O\}$

$T_{33} = \{P\}$



树T



T_1

T_2

T_3

2. 结点的度 (degree): 结点的子树数目

3. 树的度: 树中各结点的度的最大值

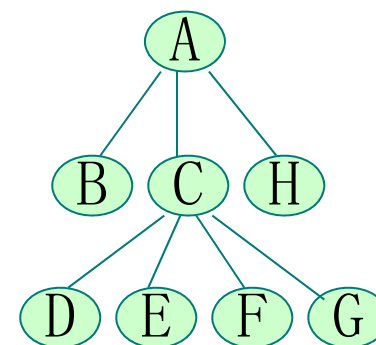
4. n度树: 度为n的树

5. 叶子(终端结点): 度为0的结点

6. 分枝结点(非终端结点, 非叶子): 度不为0的结点

7. 双亲(父母, parent)和孩子(儿子, child):

若结点C是结点P的子树的根, 称P是C的双亲, C是P的孩子。



4度树

8. 结点的层(level):

规定树T的根的层为1，其余任一结点的层等于其双亲的层加1。

9. 树的深度(depth, 高度):

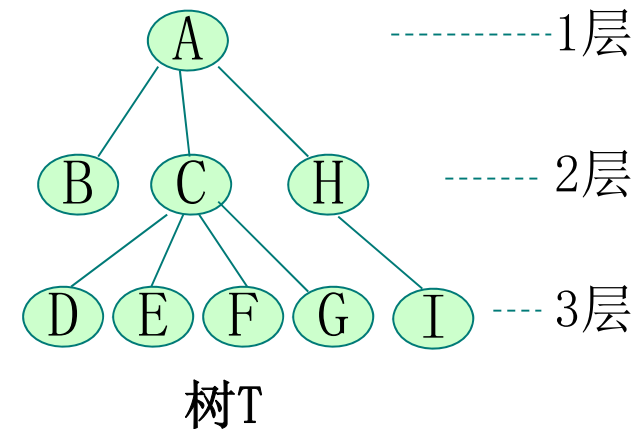
树中各结点的层的最大值。

10. 兄弟(sibling):

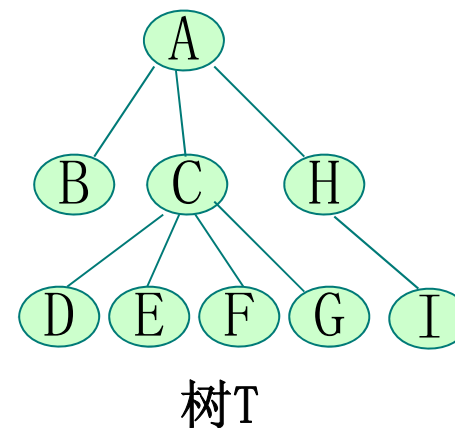
同一双亲的结点之间互为兄弟。

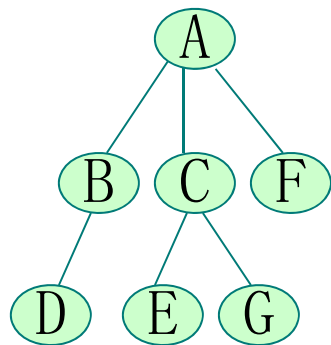
11. 堂兄弟:

同一层号的结点互为堂兄弟。

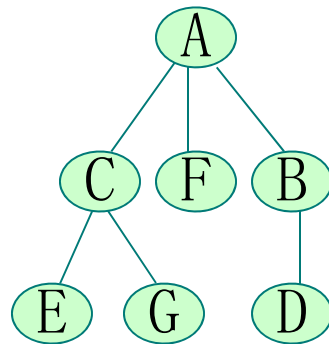


- 12. 祖先: 从树根到某结点所经分枝上的所有结点为该结点的祖先。
- 13. 子孙: 一个结点的所有子树的结点为该结点的子孙。

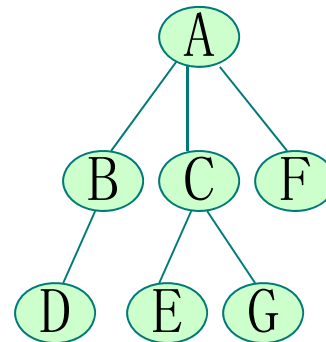




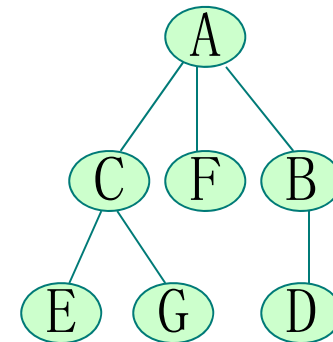
无序树 T_3



无序树 T_4



有序树 T_1

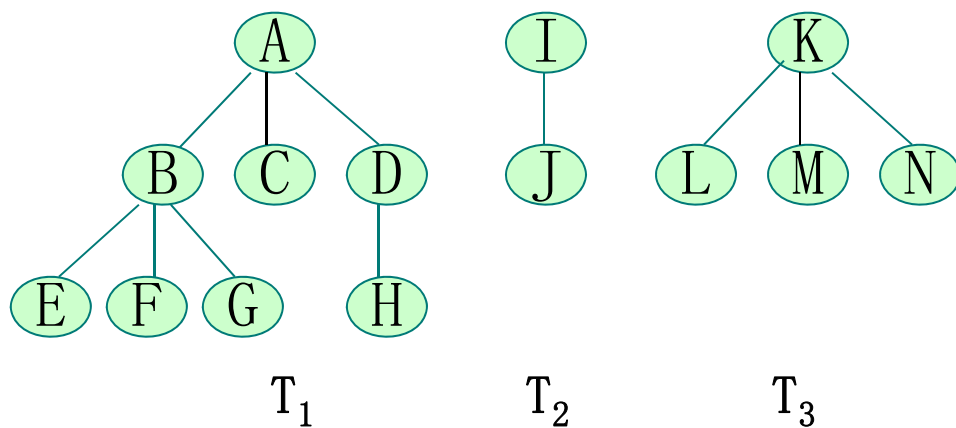


有序树 T_2

14. **有序树**: 若任一结点的各棵子树, 规定从左至右是有次序的, 即不能互换位置, 则称该树为有序树。
15. **无序树**: 若任一结点的各棵子树, 规定从左至右是无次序的, 即能互换位置, 则称该树为无序树。

16. 森林:

$m (m \geq 0)$ 棵互不相交的树的集合。

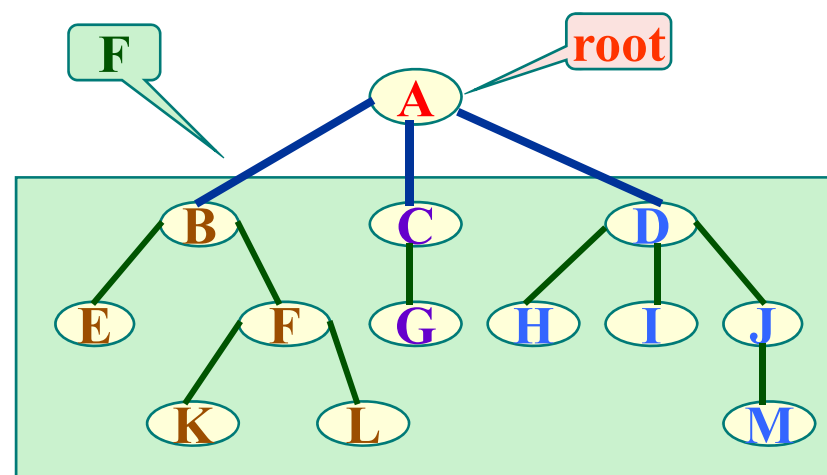


森林 $F = \{T_1, T_2, T_3\}$

任何一棵非空树可表示为一个二元组 **Tree = (root, F)**

其中: **root** 为根结点

F 被称为子树森林



6.1.3 树的基本操作

查找类:

Root(T)	// 求树的根结点
Value(T, cur_e)	// 求当前结点的元素值
Parent(T, cur_e)	// 求当前结点的双亲结点
LeftChild(T, cur_e)	// 求当前结点的最左孩子
RightSibling(T, cur_e)	// 求当前结点的右兄弟
TreeEmpty(T)	// 判定树是否为空树
TreeDepth(T)	// 求树的深度
TraverseTree(T, Visit())	// 遍历

分为三类:
查找类
插入类
删除类

插入类:

InitTree(&T)	// 初始化置空树
CreateTree(&T, definition)	// 按定义构造树
Assign(T, cur_e, value)	// 给当前结点赋值
InsertChild(&T, &p, i, c)	
	// 将以c为根的树插入为结点p的第i棵子树

删除类:

ClearTree(&T)	// 将树清空
DestroyTree(&T)	// 销毁树的结构
DeleteChild(&T, &p, i)	// 删除结点p的第i棵子树

树结构的特点

线性结构	树型结构
第一个数据元素 (无前驱)	根结点 (无前驱)
最后一个数据元素 (无后继)	多个叶子结点 (无“后继”)
其它数据元素 (一个前驱、 一个后继)	其它数据元素 (一个前驱、 多个“后继”)

6.2 二叉树(binary tree)

6.2.1 定义和术语

1. 二叉树的递归定义

❖ **定义：** 二叉树是 $n(n \geq 0)$ 个结点的有限集，

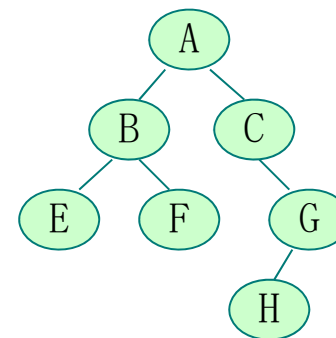
它或为空树($n=0$)；

或由一个**根结点**和两棵分别称为**左子树**和**右子树**的互不相交的二叉树构成。

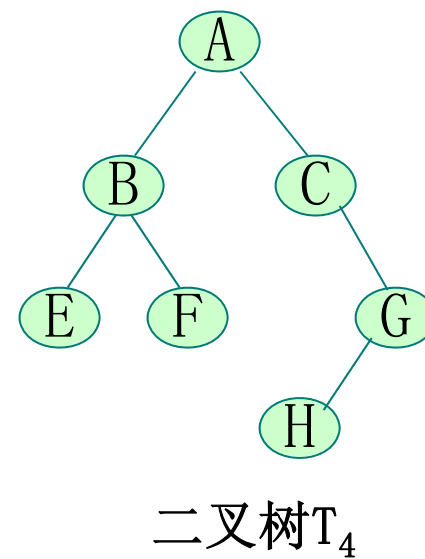
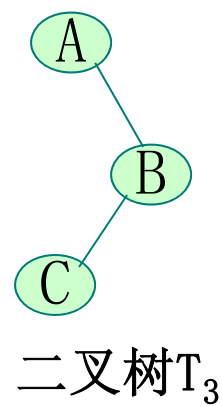
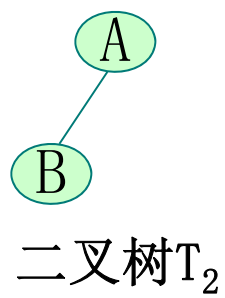
❖ 特点

☞ 每个结点至多有二棵子树(即不存在度大于**2**的结点)；

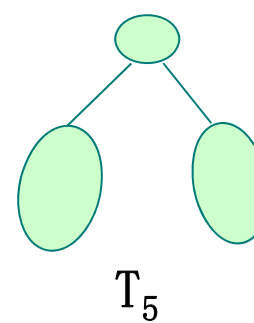
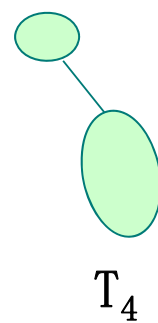
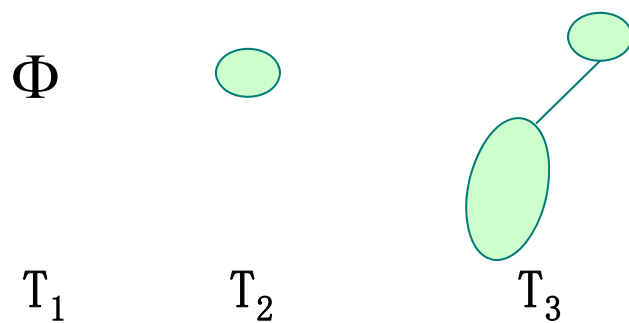
☞ 二叉树的子树有左、右之分，且其次序不能任意颠倒。



例:

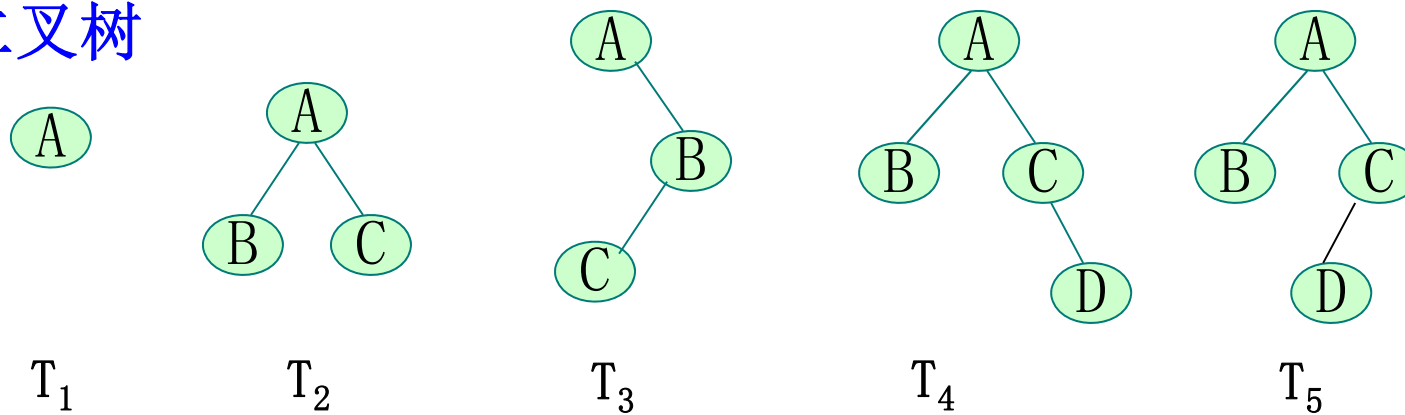


2. 二叉树的5种基本形态:

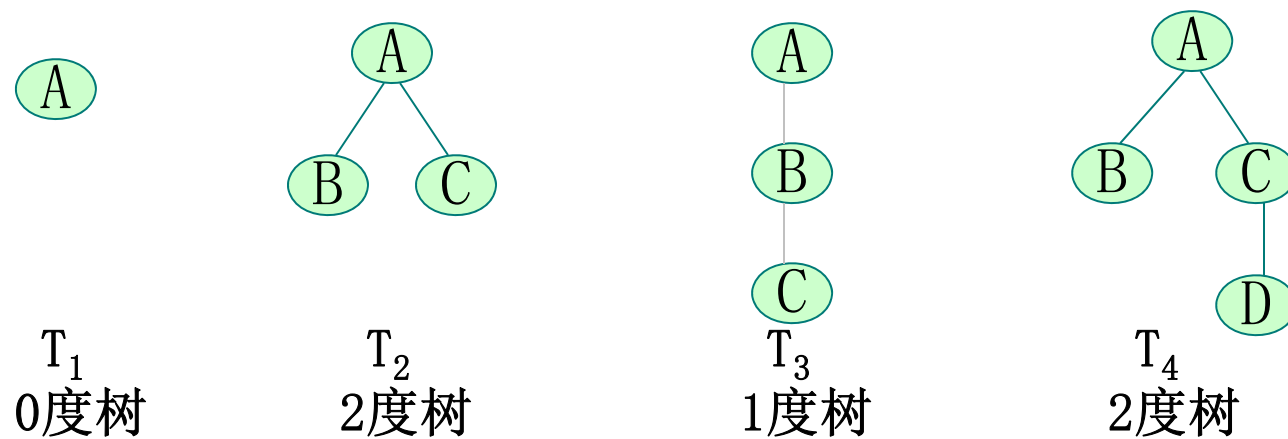


3. 二叉树与2度树的区别

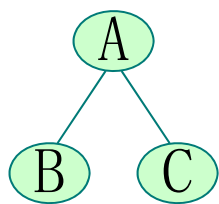
(1) 二叉树



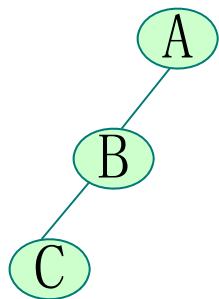
(2) 树



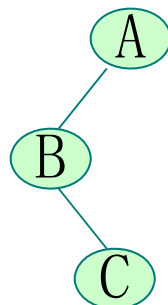
4. 三个结点不同形态的二叉树(5种)



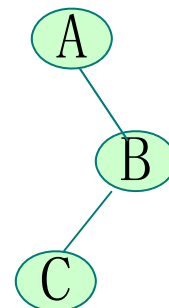
T_1



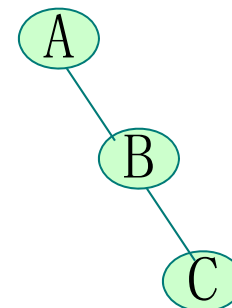
T_2



T_3

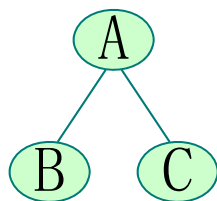


T_4

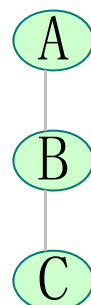


T_5

5. 三个结点不同形态的树(2种)



T_1



T_2

6. 二叉树的基本操作

1. 置T为空二叉树: $T = \{ \}$

2. 销毁二叉树T

3. 生成二叉树T: 生成哈夫曼树、二叉排序树、平衡二叉树、堆

4. 遍历二叉树T:

按某种规则访问T的每一个结点一次且仅一次的过程。

5. 二叉树 \longleftrightarrow 树

6. 二叉树 \rightarrow 平衡二叉树

7. 求结点的层号

8. 求结点的度

9. 求二叉树T的深度

10. 插入一个结点

11. 删除一个结点

12. 求二叉树T的叶子/非叶子

分为三类:

查找类 插入类 删除类

Root(T); **Value(T, e);** **Parent(T, e);**

LeftChild(T, e); **RightChild(T, e);**

LeftSibling(T, e); **RightSibling(T, e);**

BiTreeEmpty(T); **BiTreeDepth(T);**

PreOrderTraverse(T, Visit());

InOrderTraverse(T, Visit());

PostOrderTraverse(T, Visit());

LevelOrderTraverse(T, Visit());



InitBiTree(&T);

Assign(T, &e, value);

CreateBiTree(&T, definition);

InsertChild(T, p, LR, c);

ClearBiTree(&T);

DestroyBiTree(&T);

DeleteChild(T, p, LR);

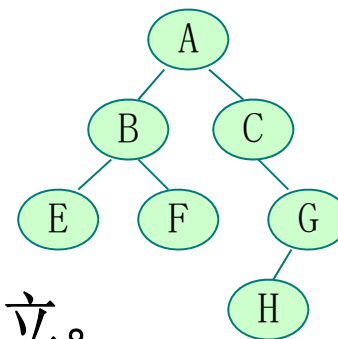
6.2.2 二叉树的性质和特殊二叉树

性质1. 在二叉树的第 i 层上至多有 2^{i-1} 个结点 ($i \geq 1$)。

证明: 用归纳法

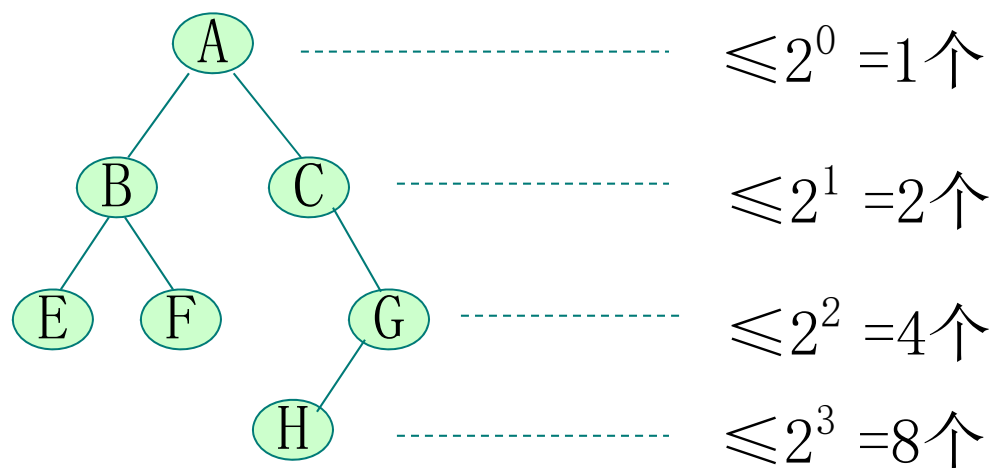
1. 当 $i=1$, 即第一层只有一个根结点, 显然 $2^{i-1} = 2^0 = 1$ 成立。
2. 假设对所有的 j ($1 \leq j < i$) 上述性质成立,
即第 j 层上至多有 2^{j-1} 个结点 ($1 \leq j < i$)。
3. 要证明 $j=i$ 时, 命题也成立。

由归纳假设: 第 $i-1$ 层上至多有 2^{i-2} 个结点, 又由于二叉树每个结点的度最大为2, 所以第 i 层上结点总数最多为第 $i-1$ 层最大结点数的2倍, 即 $2 * 2^{i-2} = 2^{i-1}$ 。



二叉树性质续

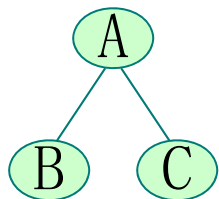
性质1 二叉树的第 i ($i \geq 1$) 层最多有 2^{i-1} 个结点。



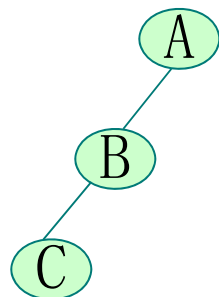
深度为 k 的二叉树
最多共有多少个
结点？

$$2^0 + 2^1 + \dots + 2^{k-1} = \frac{2^0(1 - 2^k)}{1 - 2} = 2^k - 1$$

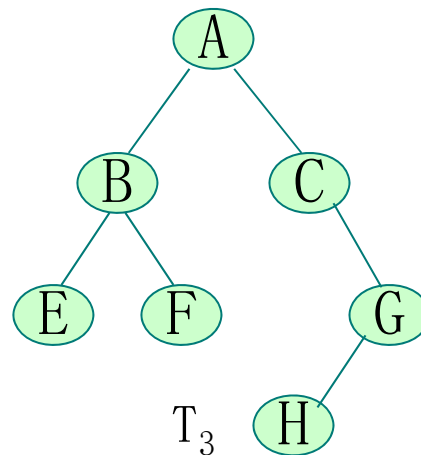
性质2 深度为 k 的二叉树最多有 $2^k - 1$ 个结点。



T_1



T_2



T_3

$$T_1: n_0=2, \quad n_2=1, \quad 2=1+1$$

$$T_2: n_0=1, \quad n_2=0, \quad 1=0+1$$

$$T_3: n_0=3, \quad n_2=2, \quad 3=2+1$$

叶子的数目
= 度为2的结点数目+1

$$n_0 = n_2 + 1$$

性质3. 二叉树中, 终端结点数 n_0 与度为2的结点数 n_2 有如下关系:

$$n_0 = n_2 + 1.$$

证明: 设二叉树中度为 i 的结点数为 n_i ,

$$\text{则结点总数 } n = n_0 + n_1 + n_2 \quad (1)$$

除根结点外, 每个结点都是另一结点的孩子

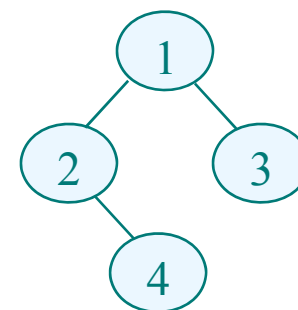
$$\text{孩子数} = n - 1; \quad (2)$$

度为 i ($i=0, 1, 2$) 的结点, 有 i 个孩子。

$$\text{孩子数} = 2n_2 + n_1, \quad (3)$$

$$\text{由 } (3) = (2) \text{ 得 } n - 1 = 2n_2 + n_1, \quad (4)$$

$$\text{由 } (4) - (1) \text{ 得 } -1 = n_2 - n_0, \text{ 故 } n_0 = n_2 + 1.$$



满二叉树 (full binary tree)-----

深度为 k 且有 2^k-1 个结点的二叉树。

特点： (1) 每一层上结点数都达到最大；叶子结点都在第 k 层。

(2) 度为1的结点 $n_1=0$

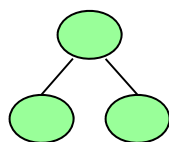
(3) n 个结点的满二叉树的深度 $=\log_2(n+1)$

设深度为 k , $\because 2^k - 1 = n$, 即 $2^k = n + 1$

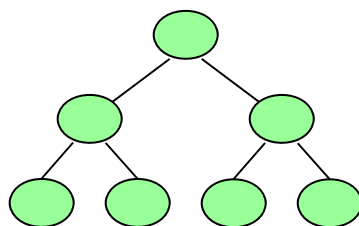
$\therefore k = \log_2(n+1)$



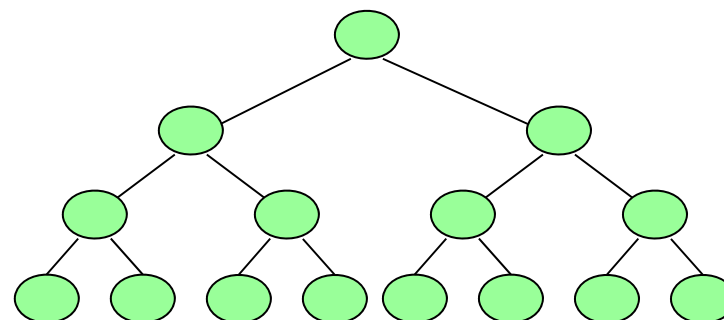
T_1



T_2

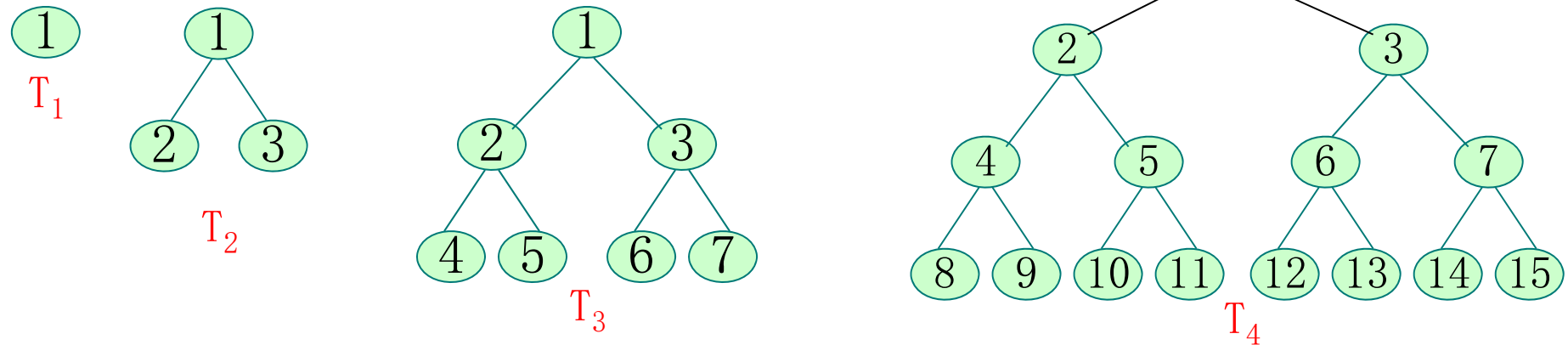


T_3



T_4

顺序编号的满二叉树：从根结点起从上到下逐层（层内从左到右）对二叉树的结点进行连续编号。



设满二叉树有 n 个结点, 编号为 $1, 2, \dots, n$

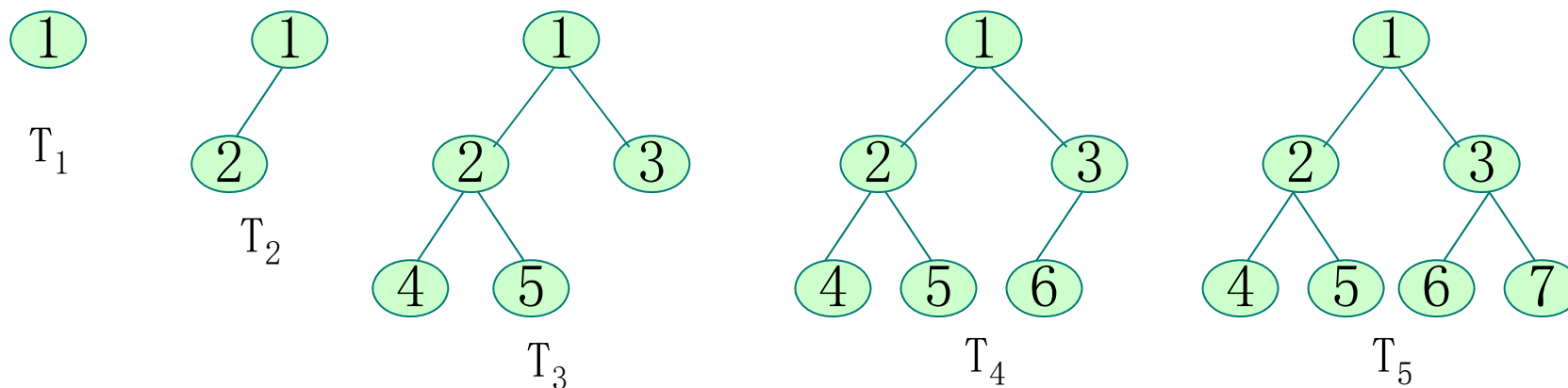
- 左小孩为偶数, 右小孩为奇数;
- 结点 i 的左小孩是 $2i$, $2i \leq n$; 结点 i 的右小孩是 $2i+1$, $2i+1 \leq n$;
结点 i 的双亲是 $\lfloor i/2 \rfloor$, $2 \leq i \leq n$;
- 结点 i 的层号 = $\lfloor \log_2 i \rfloor + 1 = \lceil \log_2(i+1) \rceil \quad 1 \leq i \leq n$

完全二叉树:

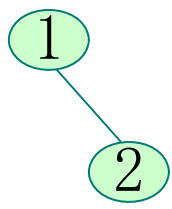
深度为 k 有 n 个结点的二叉树, 当且仅当每一个结点都与同深度的满二叉树中编号从1至 n 的结点一一对应, 则称之为完全二叉树 (或称为“顺序二叉树”)。

例. 完全二叉树:

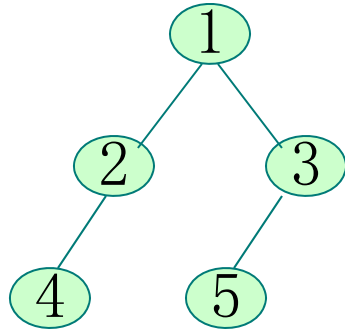
满二叉树一定是完全二叉树,
反之不成立。



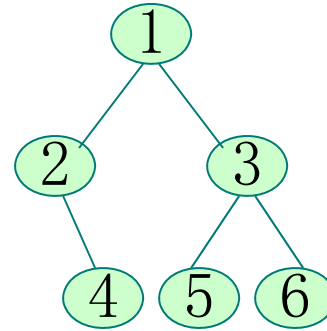
例 非完全二叉树:



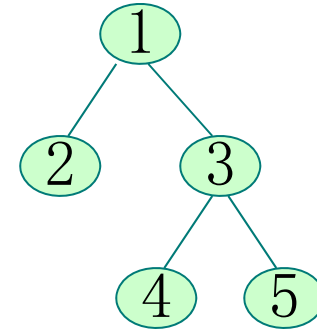
T_1



T_2

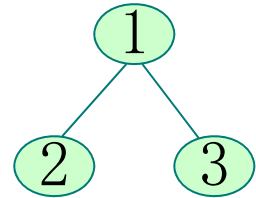


T_3

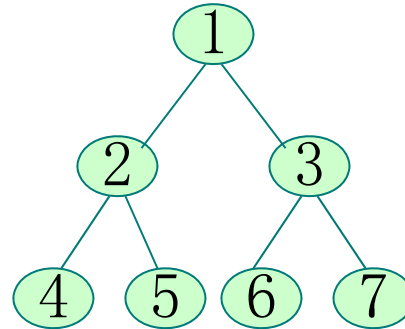


T_4

例 满二叉树:



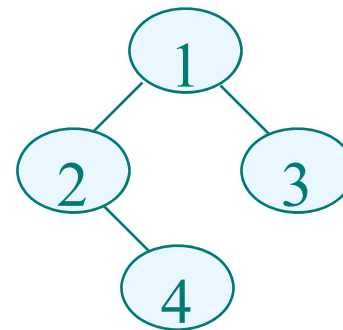
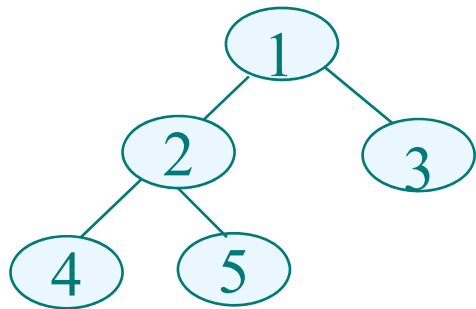
T_{01}



T_{02}

深度为 k 的完全二叉树的特点或性质:

- (1) 任意结点 i , 其左右子树的深度分别表示为 Lh_i 与 Rh_i , 则 $Lh_i - Rh_i$ 等于 0 或 1, 即叶结点只可能出现在层次最大或次最大的两层上。
- (2) 完全二叉树结点数 n 满足 $2^{k-1} - 1 < n \leq 2^k - 1$.



$$LH_2 - RH_2 = 0 - 1 = -1$$

性质4. 结点数为n的完全二叉树，其深度为

$$\lfloor \log_2 n \rfloor + 1 = \lceil \log_2 (n+1) \rceil$$

证明：设深度为k，则由性质2和完全二叉树定义有：

结点数n满足： $2^{k-1} - 1 < n \leq 2^k - 1$

或写为 $2^{k-1} \leq n < 2^k$

于是有： $k-1 \leq \log_2 n < k$

因为 $k-1$ 和 k 均为整数

显然有 $\lfloor \log_2 n \rfloor = k-1$ ，故 $k = \lfloor \log_2 n \rfloor + 1$.

性质 5 :

若对含 n 个结点的完全二叉树从上到下且从左至右进行 1 至 n 的编号, 则对完全二叉树中任意一个编号为 i 的结点:

(1) 若 $i=1$, 则该结点是二叉树的根, 无双亲,

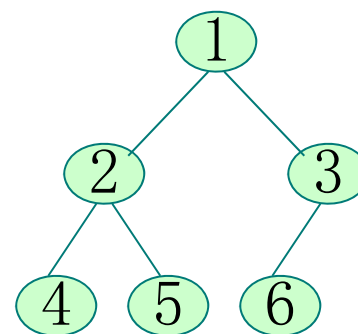
否则, 编号为 $\lfloor i/2 \rfloor$ 的结点为其双亲结点;

(2) 若 $2i > n$, 则该结点无左孩子,

否则, 编号为 $2i$ 的结点为其左孩子结点;

(3) 若 $2i+1 > n$, 则该结点无右孩子结点,

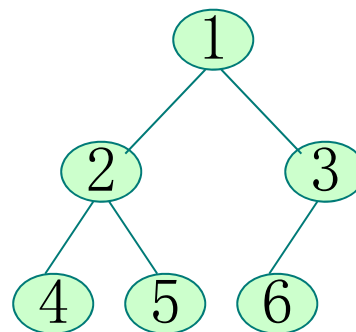
否则, 编号为 $2i+1$ 的结点为其右孩子结点。



基于性质5思考如下问题：

若对含 n 个结点的完全二叉树从上到下且从左至右进行 1 至 n 的编号，则：

- (1) n 号结点的双亲结点编号是多少？
- (2) 最后一个非终端节点编号是多少？
- (3) 度为1的结点具有什么特征？
- (4) 最小的叶子节点编号是多少？



6.2.3 二叉树的存储结构

1. 顺序结构

(1) 使用一维数组存储完全二叉树:

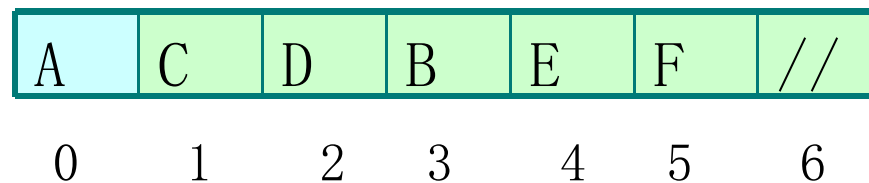
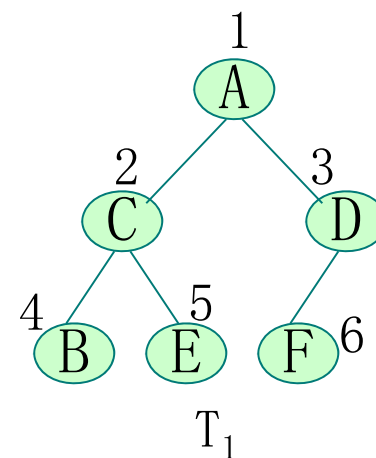
```
#define MAX_TREE_SIZE 100
```

// 二叉树的最大结点数

```
typedef TElemType SqBiTree[MAX_TREE_SIZE];
```

// 0号单元存储根结点

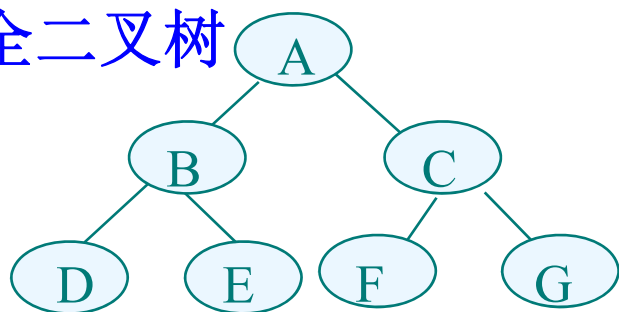
```
SqBiTree bt;
```



T_1 的顺序存储结构

顺序存储特点：用一组地址连续的存储单元，以层序顺序存放二叉树的数据元素，结点的相对位置蕴含着结点之间的关系。

完全二叉树



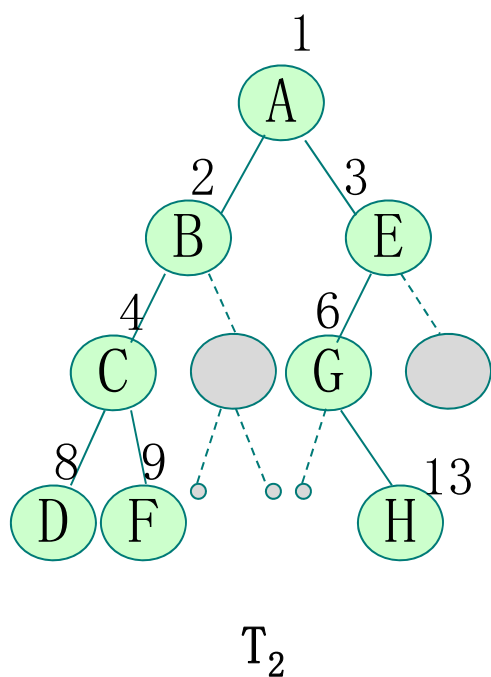
1	2	3	4	5	6	7	8	9	10	11
A	B	C	D	E	F	G	0	0	0	0

若数组从“1”编址, 对于 $i=3$,

- $bt[i]$ 的双亲为 $\lfloor 3/2 \rfloor = 1$, 即在 $bt[1]$ 中;
- 其左孩子在 $bt[2i]=bt[6]$ 中;
- 其右孩子在 $bt[2i+1]=bt[7]$ 中。

(2) 一般二叉树:

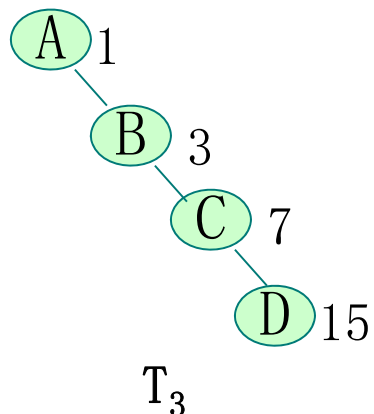
按完全二叉树形式存储,没结点处用0表示,表示“虚结点”。



A	B	E	C	0	G	0	D	F	0	0	0	H
1	2	3	4	5	6	7	8	9	10	11	12	13

T_2 的顺序结构

(3) 右单枝树



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

T_3 的顺序结构

缺点：①浪费空间；②插入、删除不便

深度为k的二叉树，最多需长度为 2^k-1 的一维数组。

若是右单枝树，空间利用率为：

$$\alpha = \frac{k}{2^k - 1}$$

$$k=4, \quad \alpha=4/15 ;$$

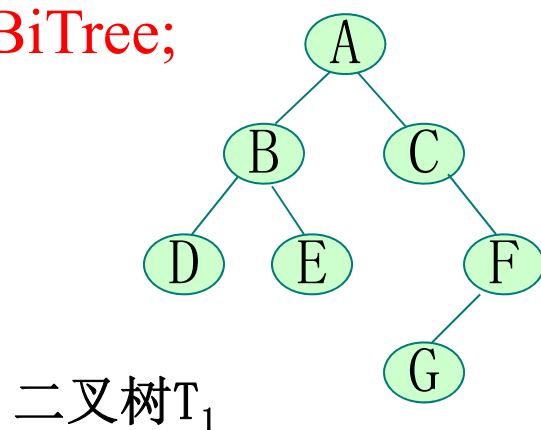
$$k=10, \quad \alpha=10/1023 \approx 0.0098$$

2. 链式存储结构

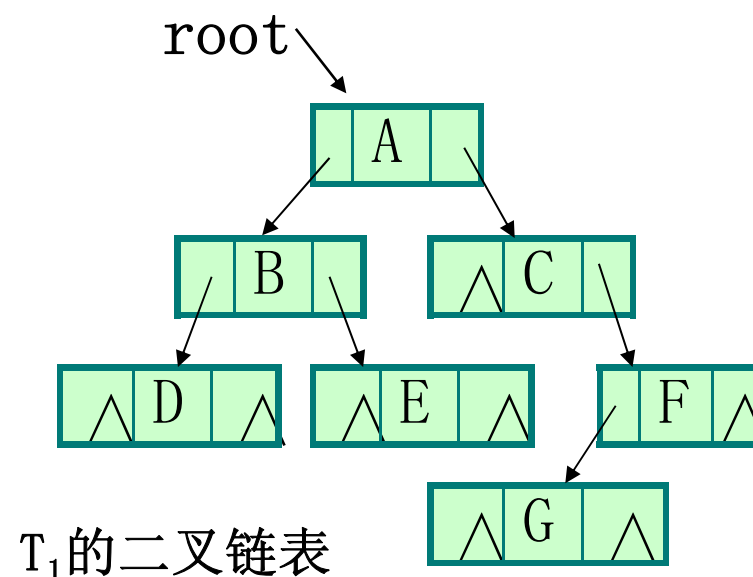
(1) 二叉链表

```
typedef struct BiTNode { // 结点结构
    TElemType    data;
    struct BiTNode *lchild, *rchild;
} BiTNode, *BiTree;
```

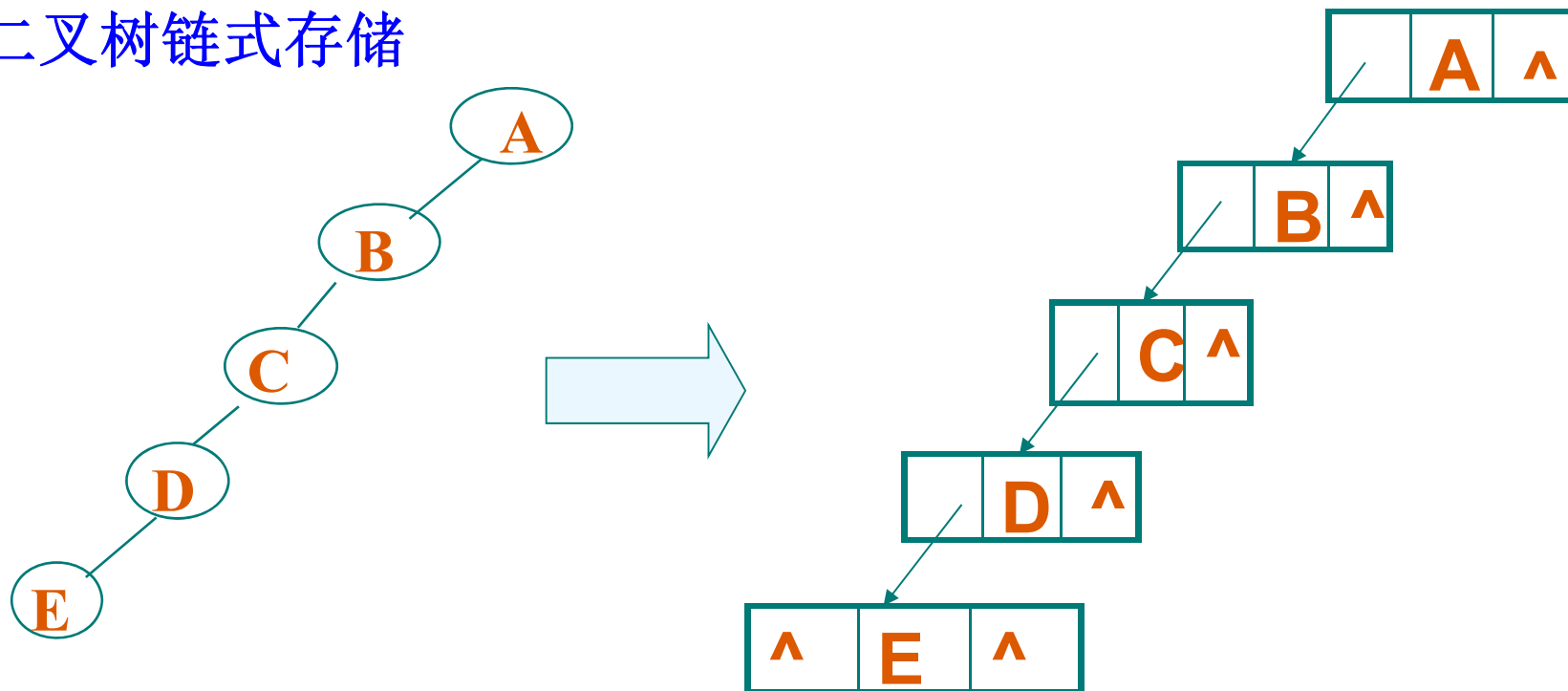
// 左右孩子指针



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



例：二叉树链式存储



优点：①不浪费空间；②插入、删除方便

性质6. 含有 n 个结点的二叉链表中，有 $n+1$ 个空链域。

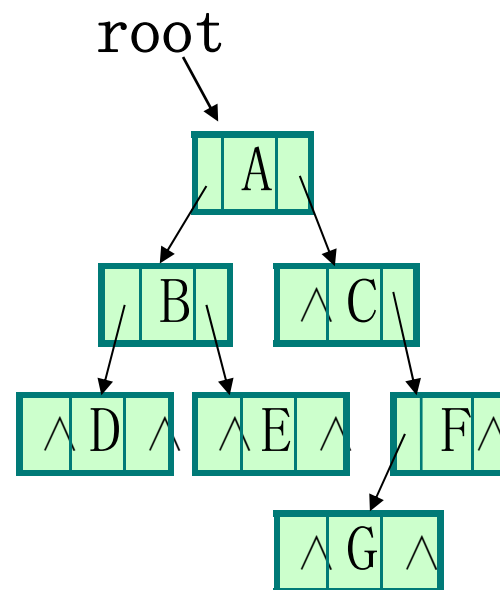
证明一： 空链域数 $=2n_0+n_1 = n_0+n_1+n_0$ (1)

又有 $n_0=n_2+1$

所以 (1) 式空链域数 $=n_0+n_1+n_2+1$

又因为 $n=n_0+n_1+n_2$

故空链域数 $=n+1$.



证明二：

有 $n-1$ 个结点有链引入，所以非空链域数为： $n-1$,

而 n 个结点共有 $2n$ 个链域，故空链域数： $n+1$.

(2) 三叉链表

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

typedef struct TriTNode { // 结点结构

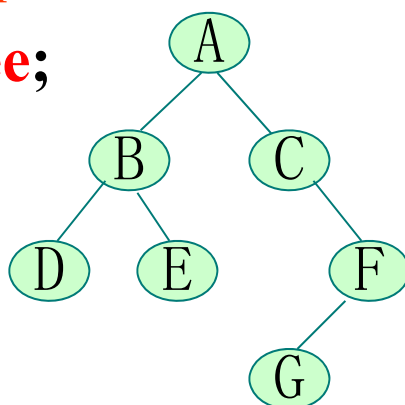
TElemType data;

struct TriTNode *lchild, *rchild;

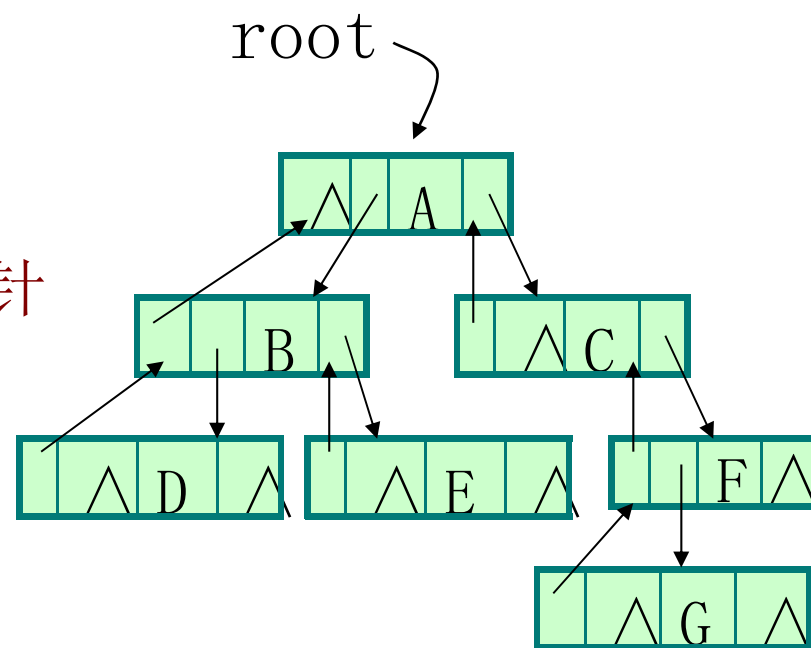
// 左右孩子指针

struct TriTNode *parent; // 双亲指针

} TriTNode, *TriTree;



二叉树 T_2

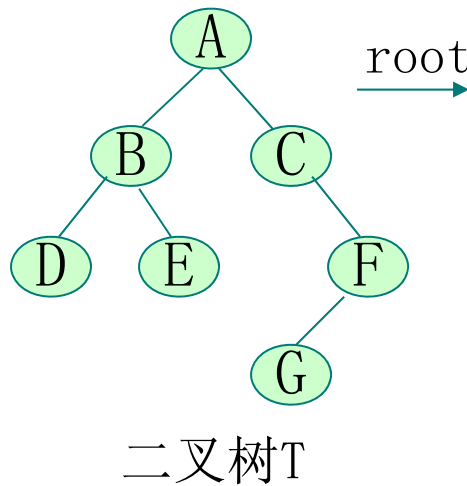


T_2 的三叉链表

- 
- 思考：含 n 个结点的三叉链表有多少个空链域？
 - 线索链表

(3) 静态链表

```
struct SBITNode
{ ElemType data;
  int lchild, rchild;
} SBITree[n+1];
```



	lchild	data	rchild
0	///	///	///
1	2	A	3
2	4	B	5
3	0	C	6
4	0	D	0
5	0	E	0
6	7	F	0
7	0	G	0

一维数组t[0..7]

6.3 遍历二叉树和线索二叉树

6.3.1 遍历二叉树

按某种规则访问二叉树的每一个结点一次且仅一次的过程。

- “遍历”是任何类型均有的操作，对线性结构而言，只有一条搜索路径(因为每个结点均只有一个后继)。
- 二叉树是非线性结构，每个结点可能有两个后继，则存在如何遍历即按什么样的搜索路径遍历的问题。
- 一次遍历后，使二叉树树中结点的非线性排列，按访问的先后顺序变为某种线性排列。
- 遍历是树结构插入、删除、修改、查找等运算的基础。

设：D----访问根结点, 输出根结点;

L----递归遍历左子树; R----递归遍历右子树。

遍历规则(方案):

□ 先左后右

DLR----先序遍历(先根, preorder)

LDR----中序遍历(中根, inorder)

LRD----后序遍历(后根, postorder)

□ 先右后左

DRL----逆先序遍历 RDL----逆中序遍历

RLD----逆后序遍历

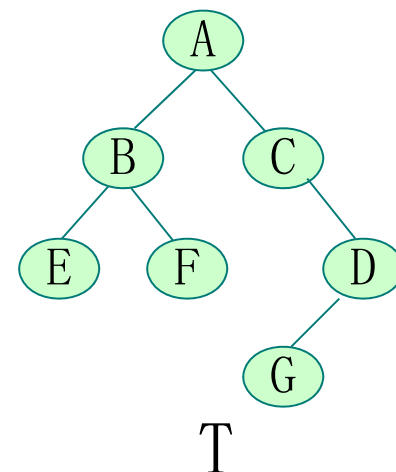
先序遍历

先序遍历二叉树递归定义：

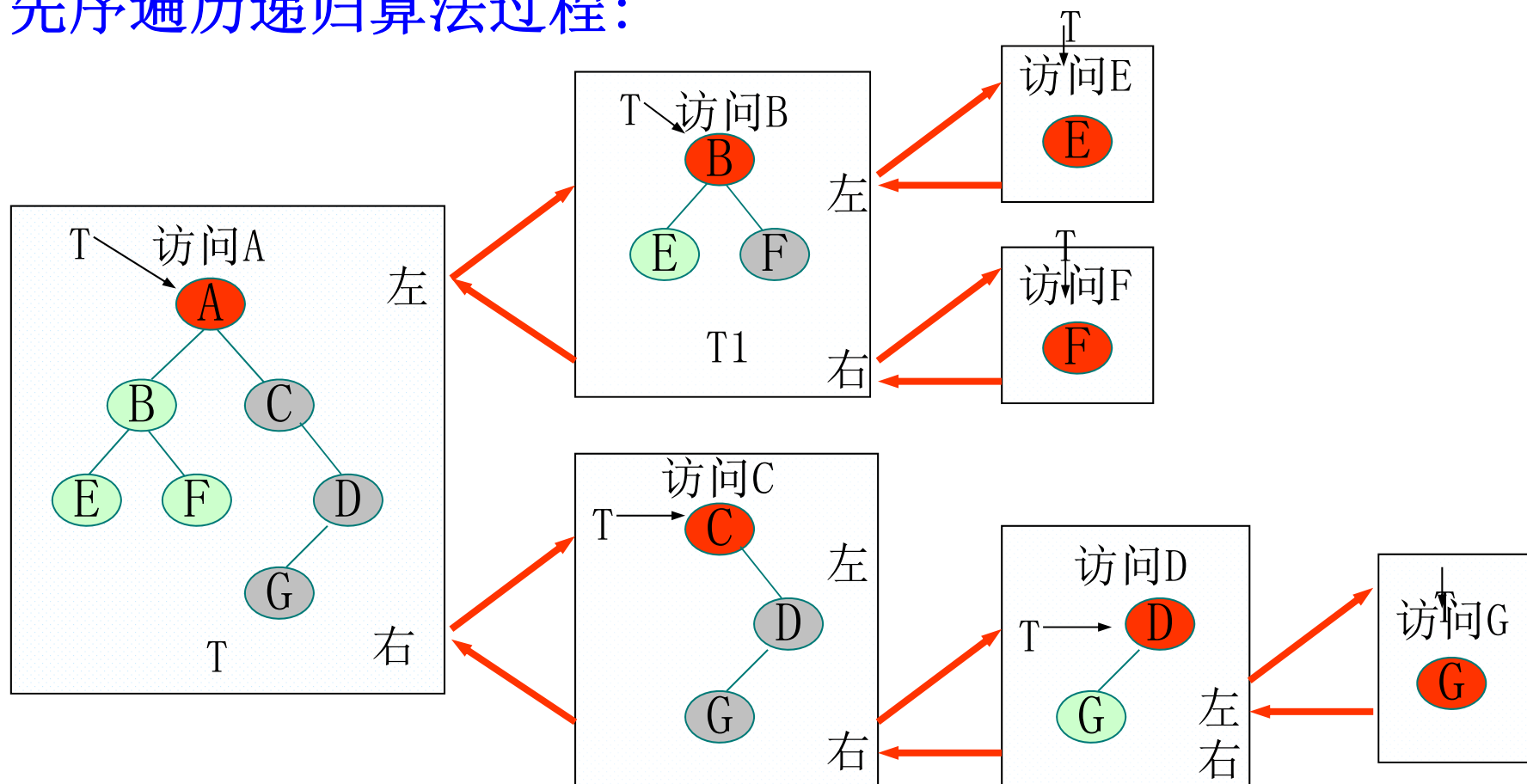
若二叉树为空，则遍历结束；

否则，执行下列步骤：

- (1) 访问根结点；
- (2) 先序遍历根的左子树；
- (3) 先序遍历根的右子树。



先序遍历递归算法过程：



先序遍历递归算法(基于二叉链表)：

```
typedef struct BiTNode *BiTree; //结点指针类型
status PreOrderTraverse(BiTree T, status( *visit)(TElemType &e))
{
    // 先序遍历二叉树
    if (T) {
        visit(T->data); // 访问结点
        PreorderTraverse (T->lchild, visit); // 遍历左子树
        PreorderTraverse (T->rchild, visit); // 遍历右子树
    }
}
```

中序遍历

中序遍历二叉树递归定义：

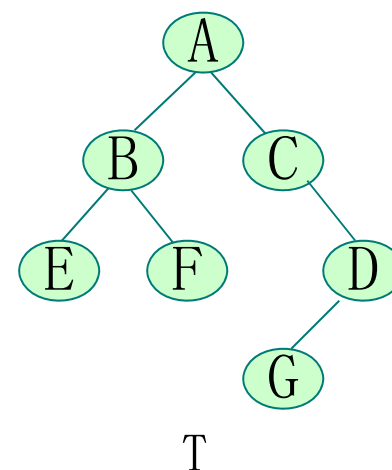
若二叉树为空，则遍历结束；

否则，执行下列步骤：

(1) 中序遍历根的左子树；

(2) 访问根结点；

(3) 中序遍历根的右子树。



中序遍历递归算法

```
typedef struct BiTNode *BiTree;    //结点指针类型
void InOrderTraverse(BiTree T)
    //T是指向二叉链表根结点的指针
{
    if (T)
    {
        InOrderTraverse(T->lchild); //递归访问左子树
        printf( "%c" , T->data);    //访问结点
        InOrderTraverse(T->rchild); //递归访问右子树
    }
    return;
}
```

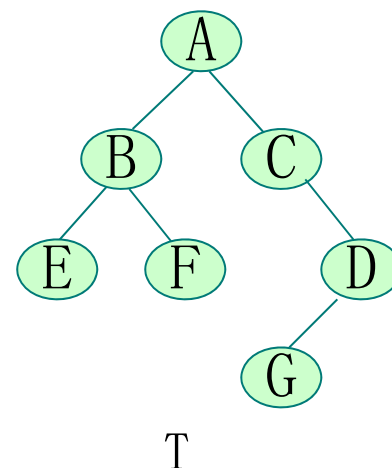

后序遍历

后序遍历二叉树递归定义：

若二叉树为空，则遍历结束；

否则，执行下列步骤：

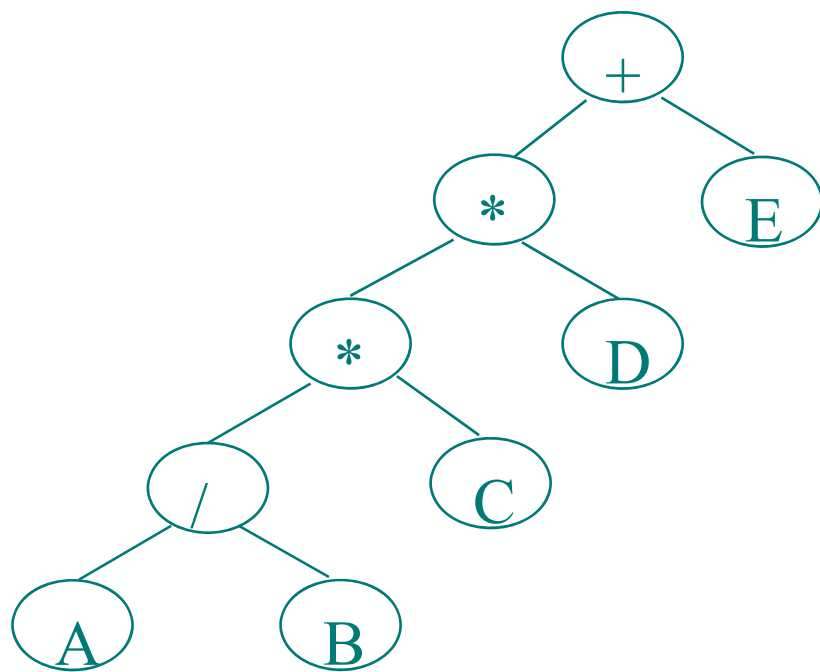
- (1) 后序遍历根的左子树；
- (2) 后序遍历根的右子树；
- (3) 访问根结点。



后序遍历递归算法

```
typedef struct BiTNode *BiTree; //结点指针类型
void PostOrderTraverse(BiTree T)
    //T是指向二叉链表根结点的指针
{
    if (T)
    {
        PostOrderTraverse(T->lchild); //递归访问左子树
        PostOrderTraverse(T->rchild); //递归访问右子树
        printf( "%c" , T->data);      //visit(T->data),访问结点
    }
    return;
}
```

例：用二叉树表示算术表达式 $(A/B)*C*D+E$



先序遍历结果

$+ * * / A B C D E$

—前缀表示法（波兰式）

中序遍历结果

$A / B * C * D + E$

—中缀表示法

后序遍历结果

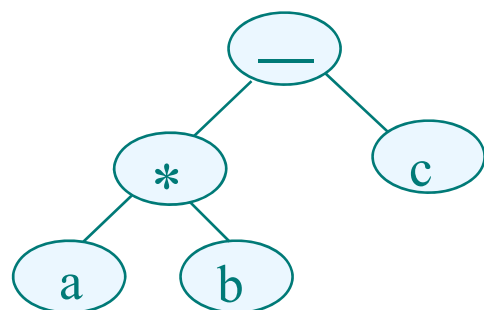
$A B / C * D * E +$

—后缀表示法（逆波兰式）

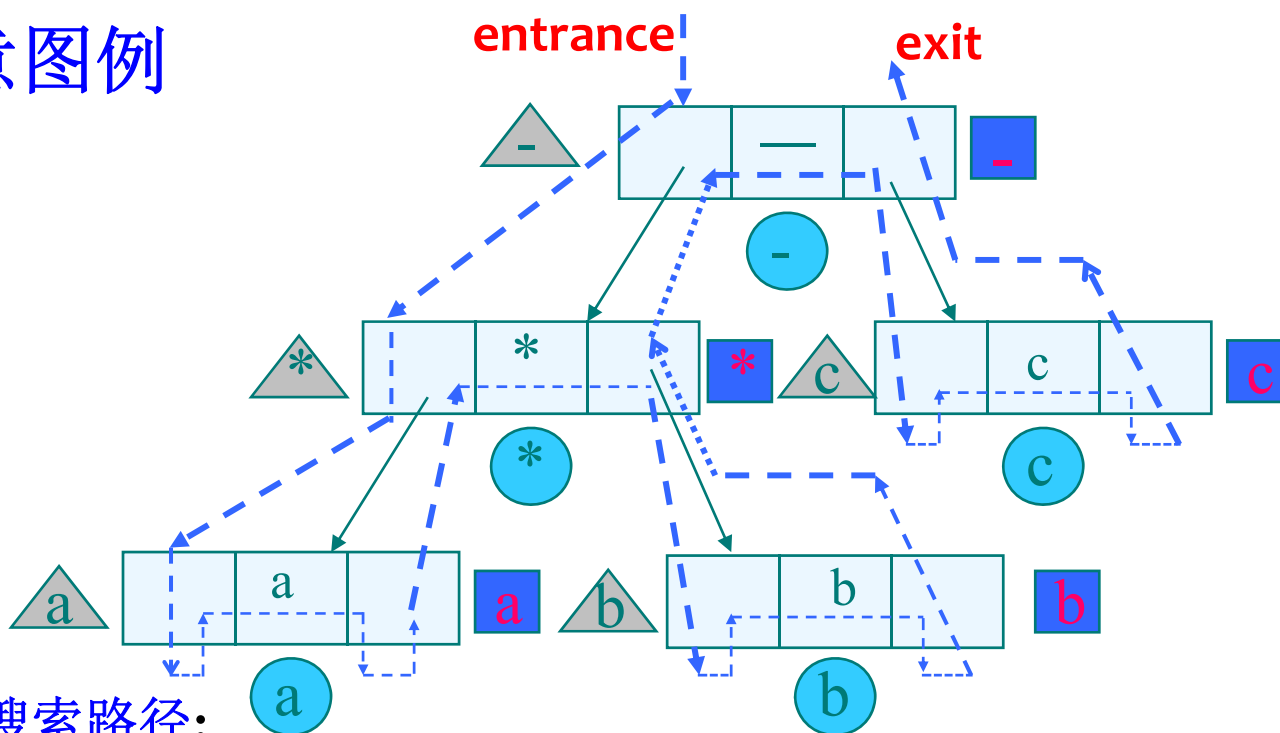
层次遍历结果

$+ * E * D / C A B$

三种遍历过程示意图例



$a*b-c$



□ 虚线表示执行过程/搜索路径:

✓ 向下表示更深层的递归调用;

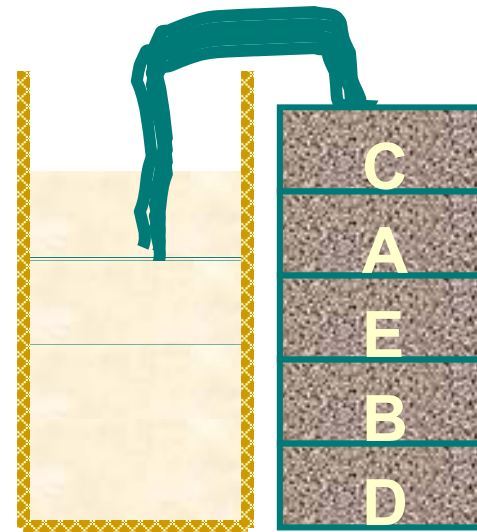
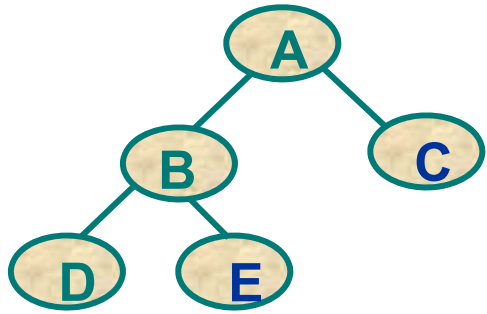
✓ 向上表示递归调用返回;

□ 沿虚线记下各类符号,便得到遍历的结果。

非递归算法(中序遍历)

- ❑ 递归算法简明精炼，但效率较低；
- ❑ 某些高级语言不支持递归；
- ❑ 非递归算法思想：
 1. 设置栈S存放所经过的根结点指针信息；初始化S；
 2. 遇到根结点并不访问，而是入栈；
 3. 中序遍历它的左子树；
左子树遍历结束后，将根结点指针退栈，并访问根结点；
然后中序遍历它的右子树。
 4. 当需要退栈时，如果栈为空则结束。

中序遍历二叉树的非递归算法



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

```
{//中序遍历非递归算法,s为存储二叉树结点的指针栈
```

```
InitStack(S); push(S,T);    //根指针进栈 (空指针也进栈)
```

```
while (!StackEmpty(S)){
```

```
    while (GetTop(S,p)&& p) push(S,p->lchild); //向左走到尽头
```

```
    pop(S,p);    //空指针退栈
```

```
    if (!StackEmpty(S)){ //访问结点与其右子树
```

```
        pop(S,p); visit(p->data); push(S,p->rchild)
```

```
    }//if
```

```
//while
```

```
return OK;
```

```
}//InOrderTraverse
```

- 根先进栈，左孩子紧随其后进栈，右孩子在根出栈后入栈

- 每个结点都进一次和出一次栈，且总访问栈顶元素，故时间复杂度为 $O(n)$

```

void Inorder(struct BiTNode *t)    //t为根指针，用顺序栈
{ struct BiTNode *st[maxleng+1]; //定义指针栈st[1..maxleng]
  int top=0;                      //置空栈
  do
  { while(t)                      //根指针t表示的为非空二叉树
    { if (top==maxleng) exit(OVERFLOW); //栈已满,退出
      st[++top]=t;                  //根指针进栈(非空指针)
      t=t->lchild;                 //t移向左子树
    } //循环结束表示以栈顶元素为根指针的二叉树的左子树遍历结束
    if (top)                       //为非空栈
    { t=st[top--]; printf("%c", t->data); //弹出根指针访问根结点
      t=t->rchild;                  //遍历右子树
    }
  } while(top || t);              //父结点未访问，或右子树未遍历
}

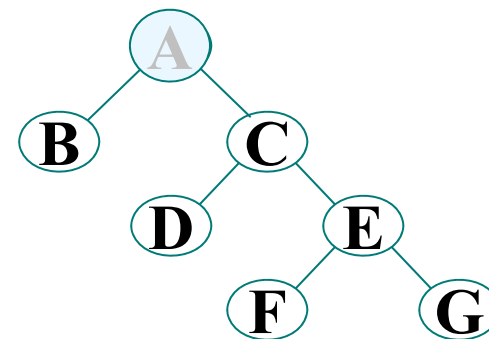
```


讨论：

- 如何实现先序、后序非递归遍历算法？
- （选择题）根据前面描述的搜索路径，在中序遍历的非递归算法中，栈中结点都未被访问；那么如果利用栈设计先序遍历的非递归算法，栈中结点的访问情况是：
(1) 都未被访问 (2) 都已被访问 (3) 不确定

层序遍历算法

- 按从上往下逐层，同层从左至右的次序访问各结点
- 访问根之后，通过根访问其左孩子，然后右孩子
- 如何暂存没访问过的结点？
- 采用队列



层序遍历序列: A

A出队

队头

队列

A的子树

根结点B、C入队

层序遍历算法 (利用队列) 习题集6.47

```
void LayerOrder(Bitree T)
```

```
{ InitQueue(Q);           //初始化队列
  if(T) EnQueue(Q,T);      //T非空则入队
  while( !QueueEmpty(Q) )
  { DeQueue(Q, &p);        //队首结点出队(送入p)
    visit(p);              //出队后即刻访问该结点
    if(p->lchild) EnQueue(Q,p->lchild); //p的左孩子入队
    if(p->rchild) EnQueue(Q,p->rchild); //p的右孩子入队
  }
} //LayerOrder
```

当孩子为空时不将
空指针入队



基于遍历的应用举例

- 1、建立二叉树的存储结构
- 2、基于遍历序列构造二叉树
- 3、求二叉树的深度(后序遍历)

建立(生成)二叉树

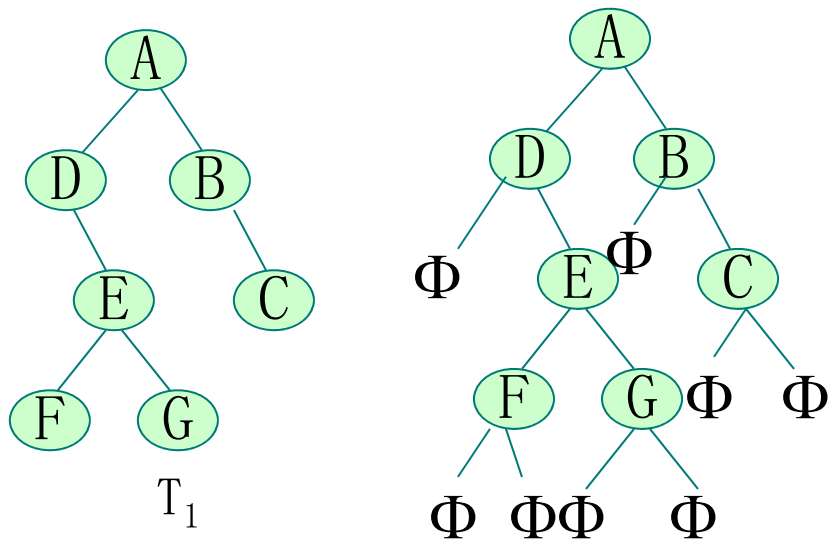
- 二叉树 T_1 的先序序列:

"ADEFGBC"

带空二叉树的先序序列:

"AD Φ EF $\Phi\Phi$ G $\Phi\Phi$ B Φ C $\Phi\Phi$ "

其中: ' Φ ' 表示空二叉树

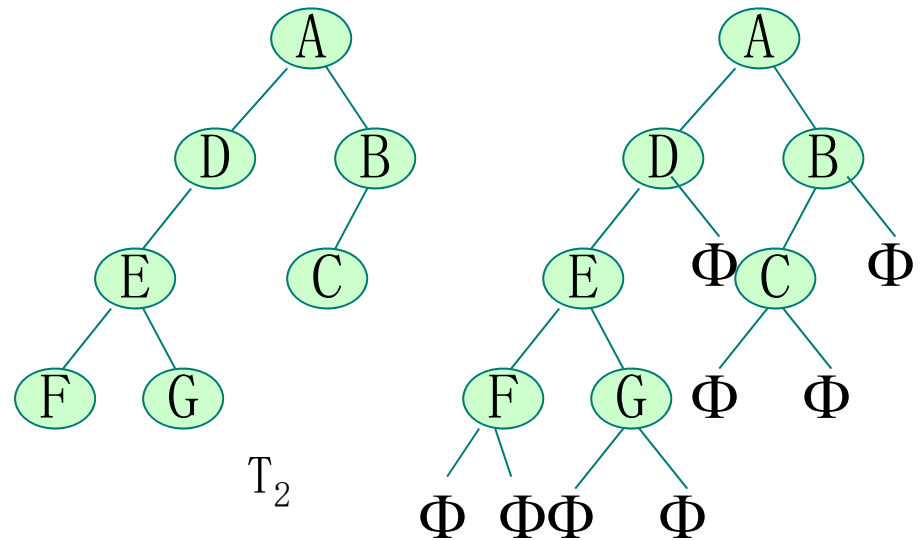


- 二叉树 T_2 的先序序列:

"ADEFGBC"

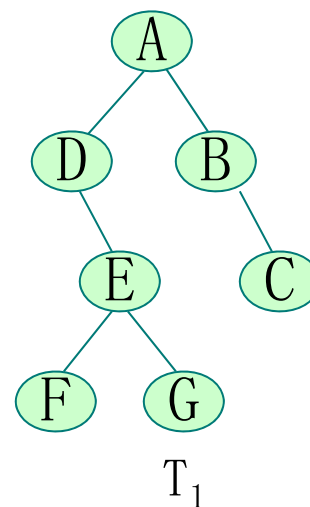
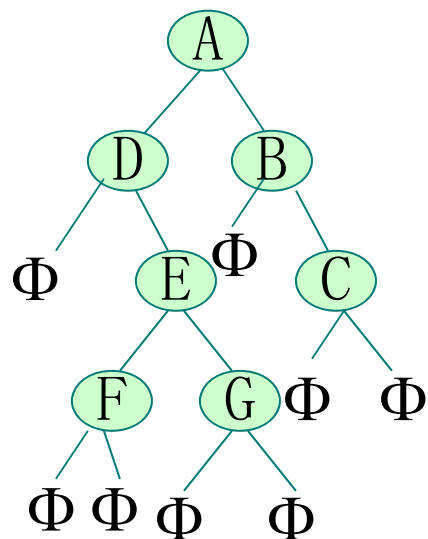
带空二叉树的先序序列:

"ADEF $\Phi\Phi$ G $\Phi\Phi\Phi$ B $\Phi\Phi\Phi$ "



输入二叉树的先序遍历次序序列，建立对应的二叉树：

A D Φ E F Φ Φ G Φ Φ B Φ C Φ Φ



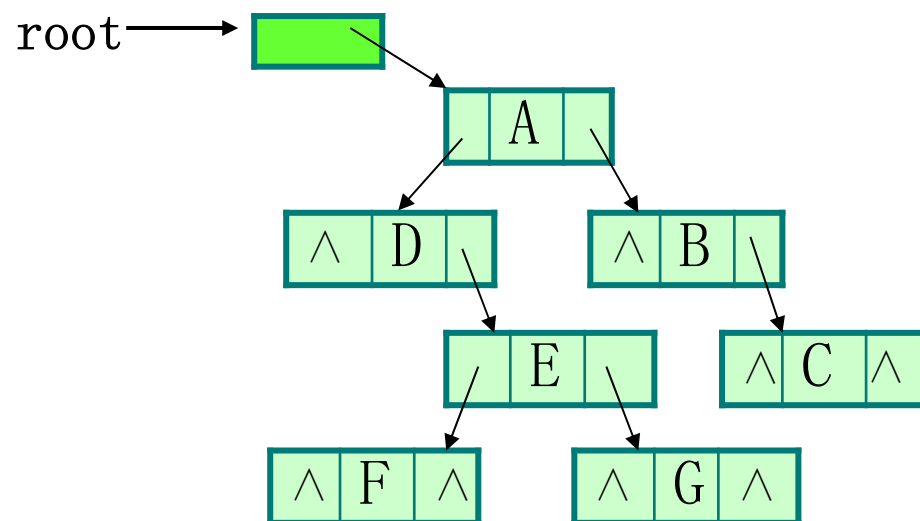
算法：创建二叉树

输入：带空结点的二叉树的先序序列

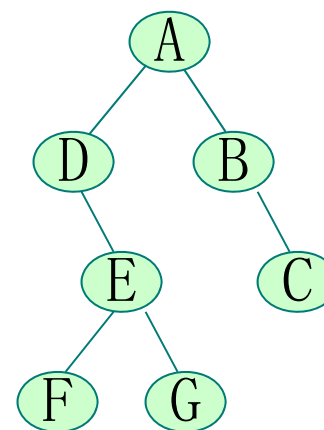
输出：二叉树的根指针

`#define leng sizeof(BiTNode)` //结点所占空间大小

假设输入先序序列：AD Φ EF $\Phi\Phi$ G $\Phi\Phi$ B Φ C $\Phi\Phi$



二叉链表



二叉树

```
Status CreateBiTree(BiTree &T) {
```

```
//按先序次序输入二叉树结点的值(空格符表空树), 构造其二叉链表
```

```
scanf(&ch);
```

```
if (ch=='Φ ') T = NULL;
```

```
else {
```

```
    if (!(T = (BiTNode *) malloc(sizeof(BiTNode)))) exit(OVERFLOW);
```

```
    T->data = ch;                // 生成根结点
```

```
    CreateBiTree(T->lchild);    // 构造左子树
```

```
    CreateBiTree(T->rchild);    // 构造右子树
```

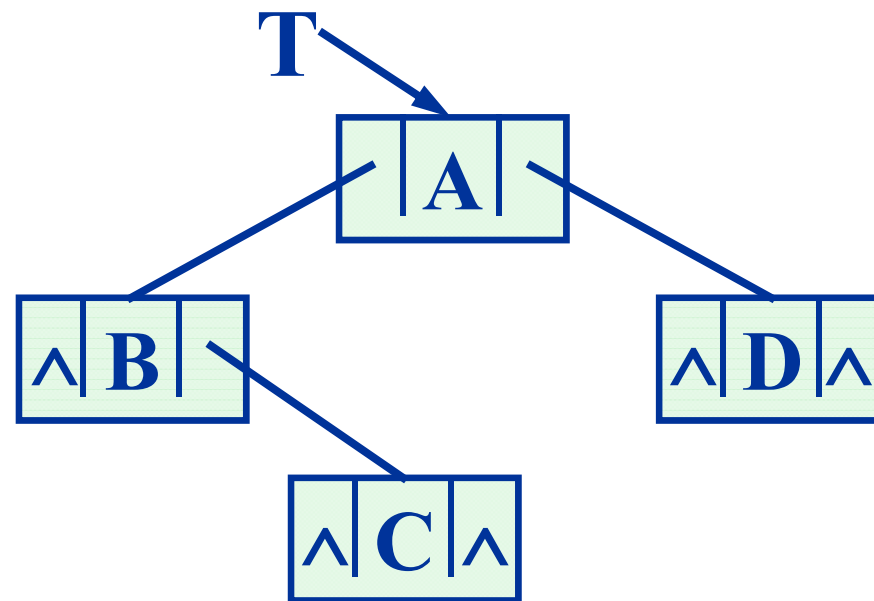
```
}
```

```
return OK;                //C语言程序如何将T返回给主调函数
```

```
} // CreateBiTree
```


上页算法执行过程举例如下：

A B ■ C ■ ■ D ■ ■



用非递归算法创建二叉树

输入： 各结点的值及其在满二叉树中的编号

输出： 二叉树根指针

```
#define MAXSIZE 100
```

```
BiTree CreatTree()
```


```
{ BiTree    s[MAXSIZE+1], root, q;  //指针数组
```

```
    int i, j;
```


```
    ElemType x;
```

```
    printf("i, x=");
```

```
    scanf("%d%d", &i, &x);
```



```
while(i!=0)
{
    q=(BiTNode *) malloc(sizeof(BiTNode));
    q->data=x; q->lchild=q->rchild=NULL;
    s[i]=q;
    if (i==1) root=q;
    else { j=i/2;
          if (i%2) s[j]->rchild=q;
          else    s[j]->lchild=q; }
    printf("i, x="); scanf("%d%d",&i,&x);
}
return root;
}
```



讨论：若知先序（或后序）遍历结果和**中序**遍历结果，
能否“恢复”出二叉树？

证明：由一棵二叉树的先序序列和中序序列可唯一确定这棵二叉树。

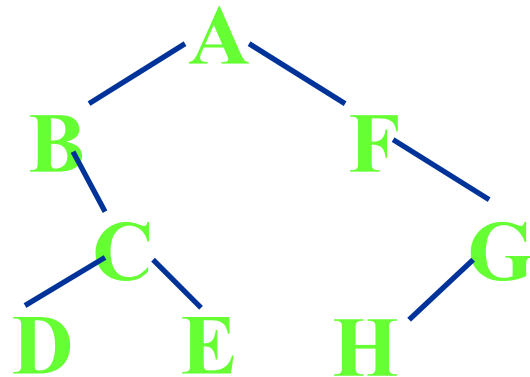
2、已知二叉树的中序序列和后序序列分别是BDCEAFHG 和 DECBHGFA，请画出这棵二叉树。

分析：

- ①由后序遍历特征，根结点必在后序序列尾部（即A）；
- ②由中序遍历特征，根结点必在中间，而且其左部必全部是左子树的子孙（即BDCE），其右部必全部是右子树的子孙（即FHG）；
- ③继而，根据后序中的DECB子树可确定B为A的左孩子，根据HGF子串可确定F为A的右孩子；以此类推。

已知中序遍历: B D C E A F H G

已知后序遍历: D E C B H G F A

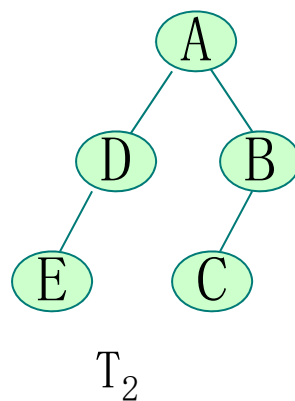
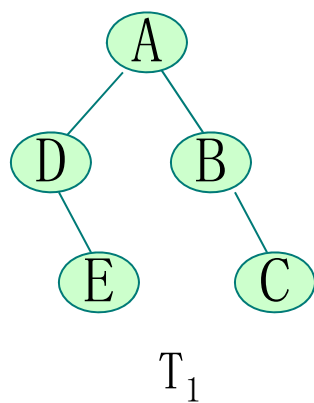


(D C E)

(F H G)

由先序序列: A D E B C
和(或)后序序列: E D C B A
能否确定唯一一棵二叉树?

不能



3、求二叉树的深度(后序遍历)

算法基本思想：

首先分析二叉树的深度和它的左、右子树深度之间的关系。

从二叉树深度的定义可知，二叉树的深度应为其左、右子树深度的最大值加1。

由此，需先分别求得左、右子树的深度。

算法中“访问结点”的操作为：求得左、右子树深度的最大值，然后加 1 。

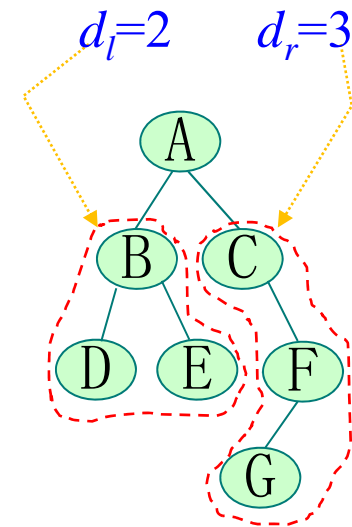
2、求二叉树的深度

$$d = \max(d_l, d_r) + 1$$

算法基本思想: 基于后序遍历

“访问结点”：求左、右子树深度的最大值，再加1.

```
int Depth (BiTree T){  
    if ( !T )    depthval = 0;  
    else {  
        depthLeft = Depth( T->lchild );  
        depthRight = Depth( T->rchild );  
        depthval = 1 +  
            (depthLeft > depthRight ? depthLeft : depthRight);  
    }  
    return depthval;  
}
```



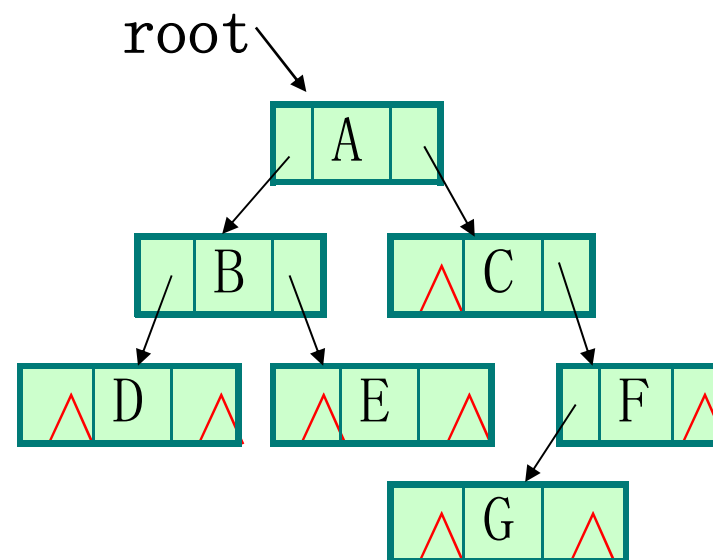
6.3.2 线索二叉树

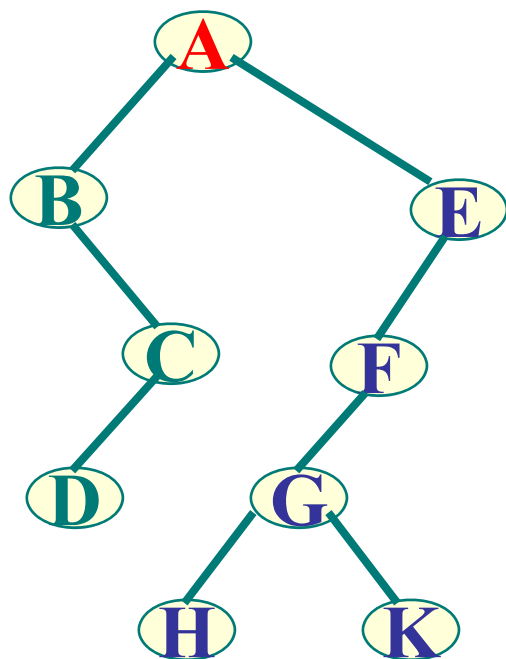
遍历二叉树是按某种规则将非线性结构的二叉树结点线性化。

- ❑ 遍历二叉树可得到结点的一个线性序列，在线性序列中，就存在结点的前驱和后继，但是在二叉链表上只能找到结点的左孩子、右孩子。
- ❑ 二叉树结点中没有相应前驱和后继的信息。
 - ✓ 结点的前驱和后继只有在每次遍历时动态产生。
 - ✓ 能否通过结点的两个链域查找出任一结点的前驱和后继？

6.3.2 线索二叉树

- ❑ **n个节点的二叉链表：**
 - ✓ 有： $n * 2$ 个指针域
 - ✓ 使用： $n - 1$ 个指针，除根以外，
每个结点被一个指针指向
- ❑ 空指针域数： $n * 2 - (n - 1) = n + 1$
- ❑ **线索二叉树：** 利用 $n + 1$ 个空链域
存放结点的前驱和后继信息。

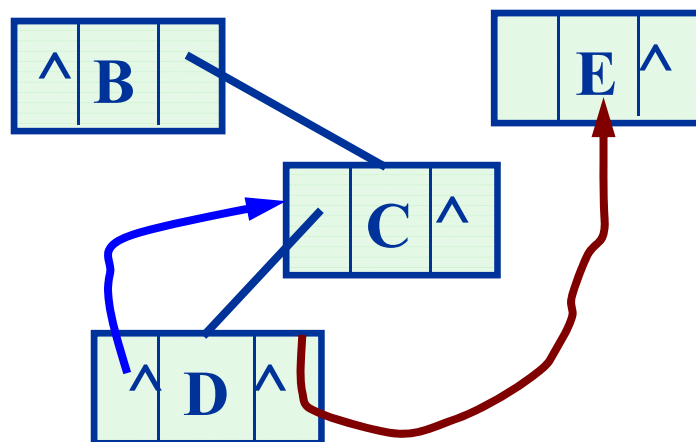




先序序列: **A B C D E F G H K**

指向该序列中的“前驱”和“后继”的指针，称作“**线索**”。

A B C **D** E F G H K



(1) 结点结构

在二叉链表中增加 Ltag 和 Rtag 两个标志域

lchild	Ltag	data	Rtag	rchild
--------	------	------	------	--------

□ 考虑结点的左子树

- ✓若有，则左链域lchild指示其左孩子（Ltag= 0）；
- ✓否则，令左链域指示其前驱（Ltag=1）；

□ 考虑结点的右子树

- ✓若有，则右链域rchild指示其右孩子（Rtag= 0）；
- ✓否则，令右链域指示其后继（Rtag= 1）。

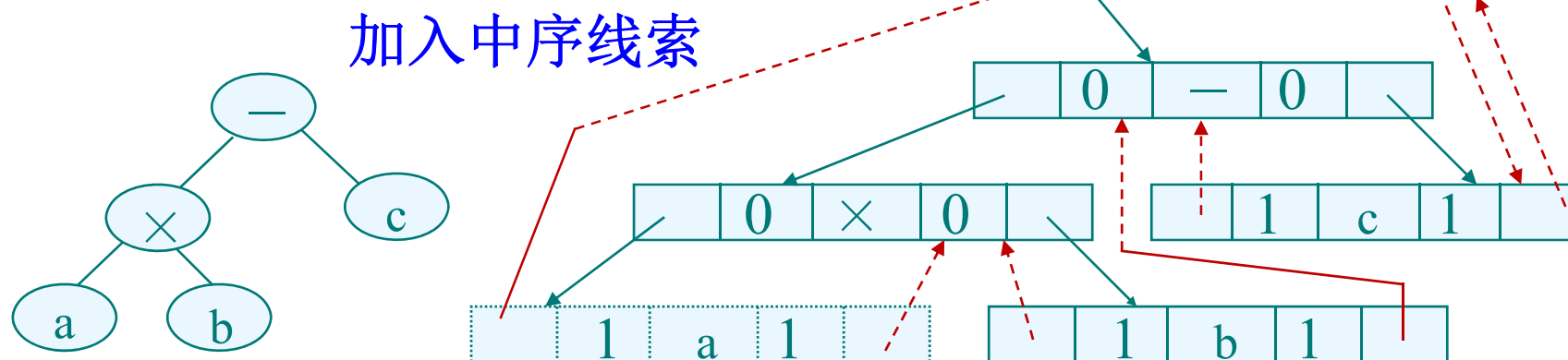
(2) 整体结构

讨论：空二叉树的线索链表形态？

增设一个**头结点**，令其lchild指向二叉树的根结点，
Ltag=0、Rtag=1；

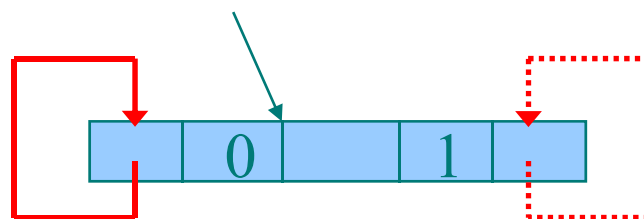
并将该结点作为遍历访问的第一个结点的前驱和最后
一个结点的后继；

最后用头指针指示该头结点。



(3) 空二叉树的线索链表

只有一个**头结点**，其 $Ltag=0$ 、 $Rtag=1$ ； $lchild$ 与 $rchild$ 都指向头结点自身。



线索链表的类型描述:

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

```
// Link==0:指针, Thread==1:线索
```

```
typedef struct BiThrNode {
```

```
    TElemType    data;
```

```
    struct BiThrNode *lchild, *rchild; // 左右指针
```

```
    PointerTag      LTag, RTag;      // 左右标志
```

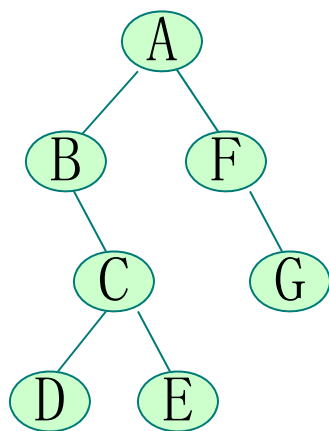
```
} BiThrNode, *BiThrTree;
```


- 称以这种结点结构构成的二叉链表为二叉树的**线索链表**；
- 其中指示前驱和后继的链域称为**线索**；
- 加上线索的二叉树称为**线索二叉树**；
- 对二叉树以**某规则**遍历使其变为线索二叉树的过程称为**线索化**。
 - ✓ 按中序遍历得到的线索二叉树称为**中序线索二叉树**；
 - ✓ 按先序遍历得到的线索二叉树称为**先序线索二叉树**；
 - ✓ 按后序遍历得到的线索二叉树称为**后序线索二叉树**。

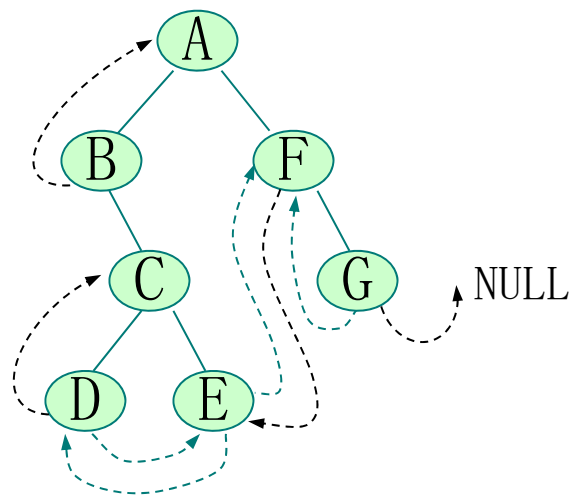
先序线索二叉树:

线索指向先序遍历中前趋、后继的线索二叉树。

例. T的先序序列: A, B, C, D, E, F, G



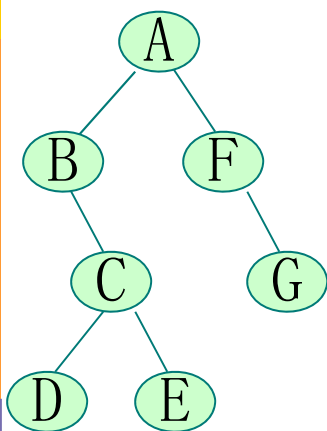
二叉树T



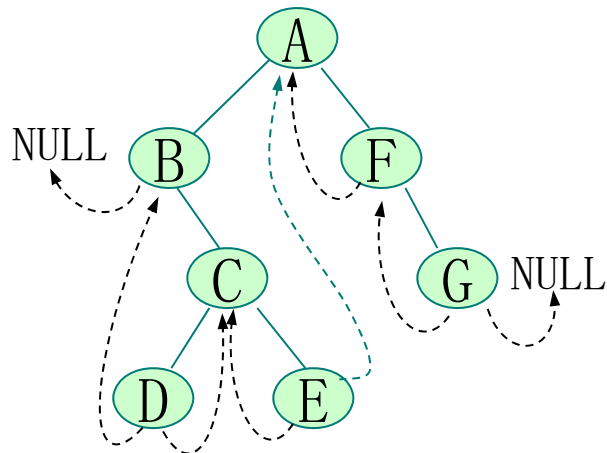
T的先序线索二叉树

中序线索二叉树: 线索指向中序遍历中前趋、后继的线索二叉树。

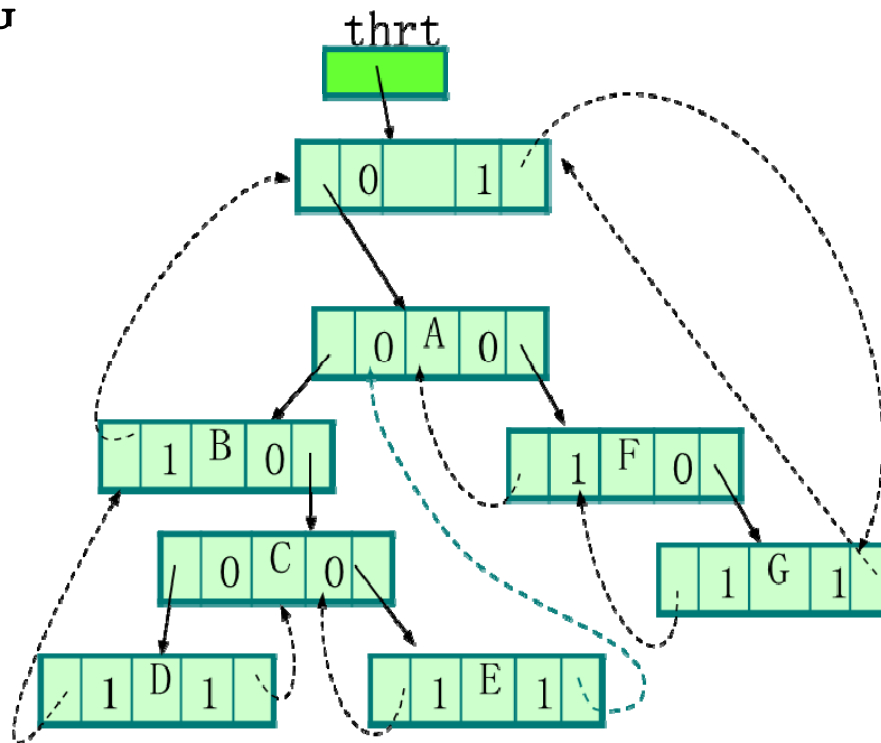
例. T的中序序列: B, D, C, E, A, F, G



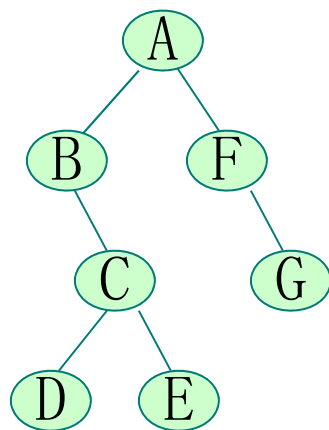
二叉树T



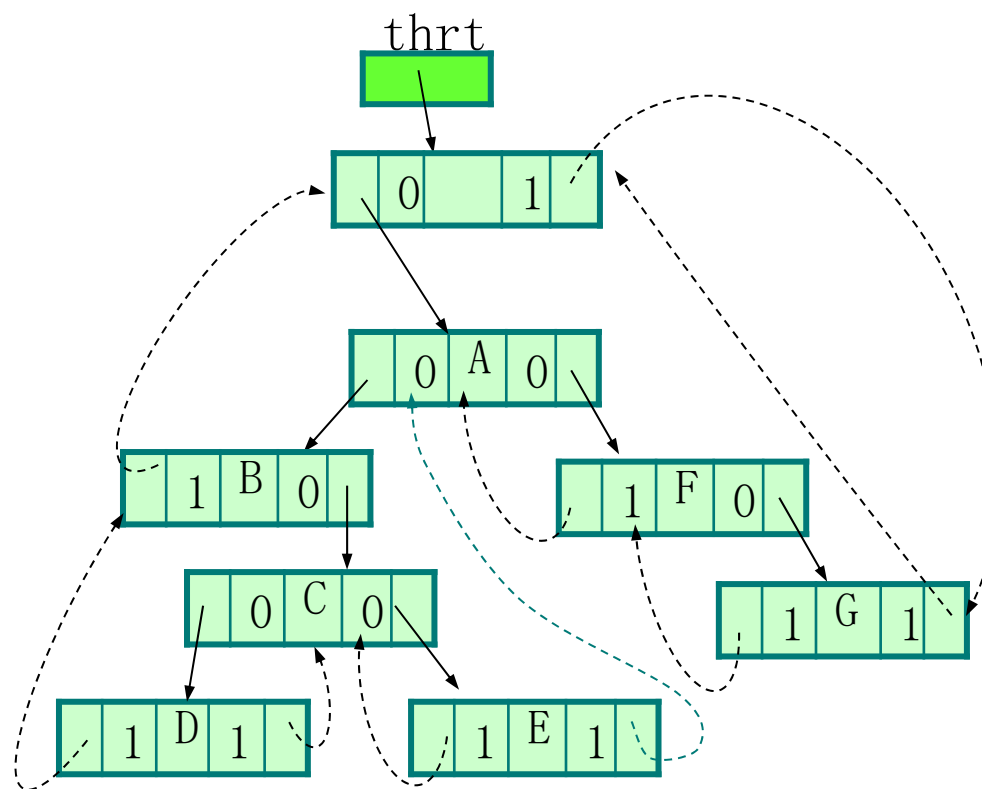
T的中序线索二叉树



T的中序线索二叉树链表



二叉树T

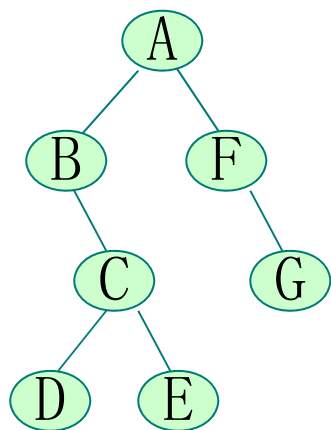


T的中序线索二叉树链表

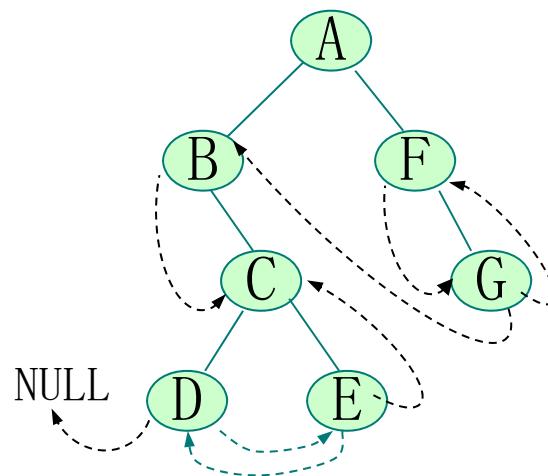
后序线索二叉树:

线索指向后序遍历中前趋、后继的线索二叉树。

例. T的后序序列: **D,E,C,B,G,F,A**



二叉树T

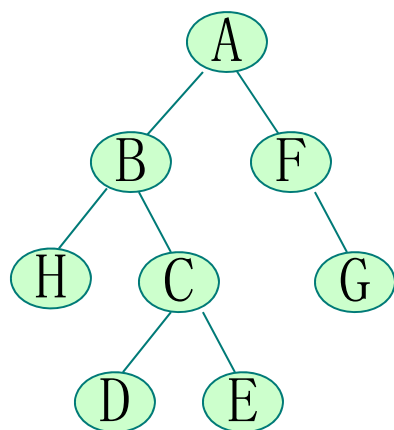


T的后序线索二叉树

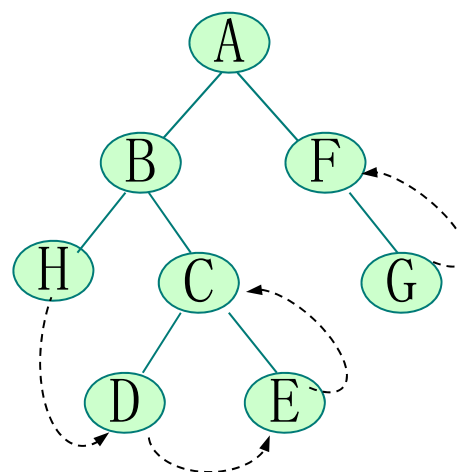
后序后继线索二叉树:

只设指向后序遍历中后继线索的线索二叉树。

例. T的后序序列: **H,D,E,C,B,G,F,A**



二叉树T



T的后序后继线索二叉树

线索链表的遍历算法：（无需堆栈）

由于在线索链表中添加了遍历中得到的“前驱”和“后继”的信息，从而简化了遍历算法。

例：对中序线索链表的遍历算法

中序遍历的第一个结点？

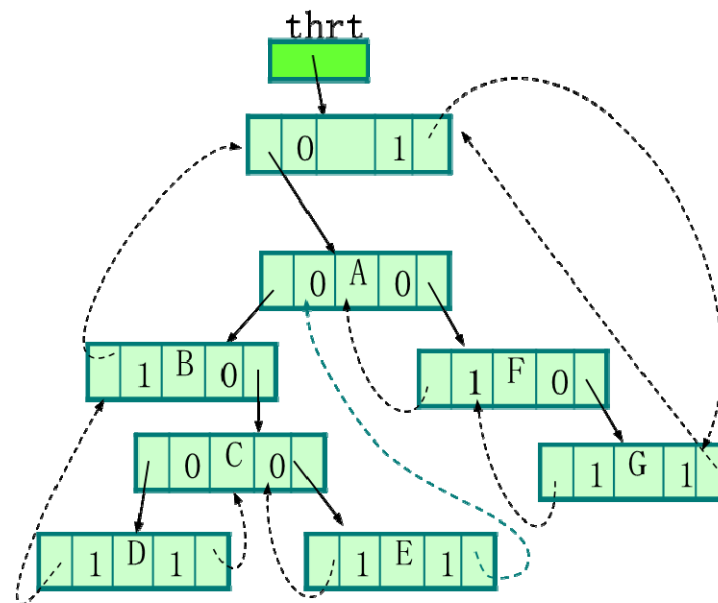
左子树上处于“最左下”

（没有左子树）的结点。

在中序线索链表中结点的后继？

若无右子树，则为后继线索所指结点；

否则为对其右子树进行中序遍历时访问的第一个结点。



T的中序线索二叉树链表

中序线索二叉树遍历步骤（算法6.5）：

有后继找后继，
无后继找右子树的
最左子孙

- 1) 设置一个搜索指针p;
- 2) 先寻找中序遍历之首结点（即最左下角结点）：
当LTag=0时（表示有左孩子）， $p=p->lchild$ ；直到LTag=1
（无左孩子，已到最左下角）；首先访问 $p->data$ ；
- 3) 接着进入该结点的右子树，检查RTag 和 $p->rchild$ ；
- 4) 若该结点的RTag=1（表示有后继线索），则 $p=p->rchild$ ；
访问 $p->data$ ；并重复4），直到后继结点的RTag=0；
- 5) 当RTag=0时（表示有右孩子），则应从该结点的右孩子
开始（ $p=p->rchild$ ）查找左下角的子孙结点；即重复2）。


```
void InOrderTraverse_Thr(BiThrTree T, void (*Visit)(TElemType e)) {
```

```
    p = T->lchild;    // p指向根结点
```

```
    while (p != T) {    // 空树或遍历结束时， p==T
```

```
        while (p->LTag==Link) p = p->lchild; // 第一个结点
```

```
        if(!visit(p->data)) return ERROR; //访问其左子树为空的节点
```

```
        while (p->RTag==Thread && p->rchild!=T) {
```

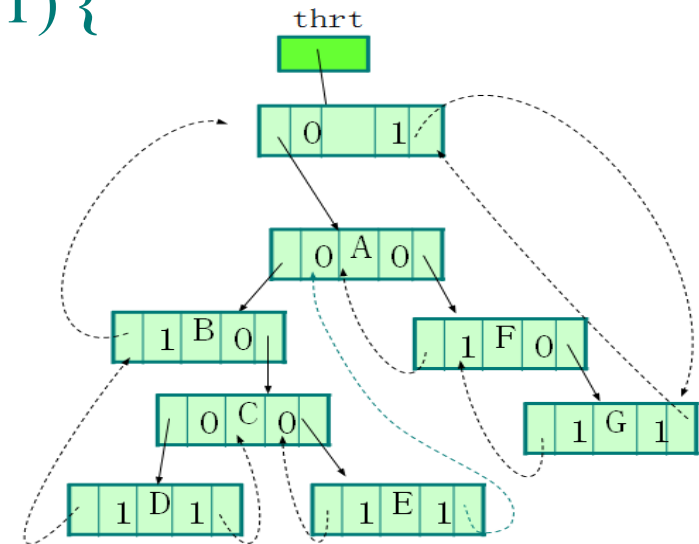
```
            p = p->rchild; Visit(p->data);
```

```
        }                // 访问后继结点
```

```
        p = p->rchild;    // 处理其右子树
```

```
    }
```

```
} // InOrderTraverse_Thr
```



如何建立线索链表？ 算法6.6

在中序遍历过程中修改结点的左、右指针域，以保存当前访问结点的“前驱”和“后继”信息。

遍历过程中，附设指针pre, 并始终保持指针pre指向当前访问的、指针p所指结点的前驱。

每次只修改前驱结点的右指针(后继)和本结点的左指针(前驱)。

若 $p \rightarrow lchild = \text{NULL}$, 则 $\{p \rightarrow Ltag = 1; p \rightarrow lchild = pre;\}$

//p的前驱线索应存p结点的左边

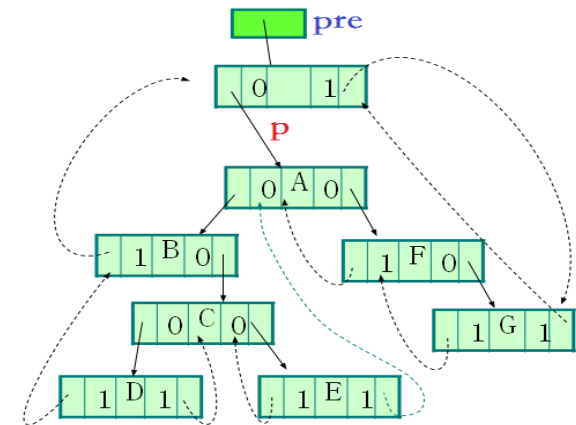
若 $pre \rightarrow rchild = \text{NULL}$, 则 $\{pre \rightarrow Rtag = 1; pre \rightarrow rchild = p;\}$

//pre的后继线索应存pre结点的右边

```

void InThreading(BiThrTree p) {
    //对以p为根的非空二叉树进行线索化
    if (p) {
        InThreading(p->lchild); //左子树线索化
        if (!p->lchild)          //建前驱线索
            { p->LTag = Thread;  p->lchild = pre; }
        if (!pre->rchild)        //建后继线索
            { pre->RTag = Thread; pre->rchild = p; }
        pre = p;                //保持 pre 指向 p 的前驱
        InThreading(p->rchild); //右子树线索化
    }
    //if, 退出时, pre指向中序遍历的最后结点
} // InThreading

```



Status InOrderThreading (BiThrTree &Thrt, BiThrTree T)

```
{ // 构建中序线索链表
    if (!(Thrt = (BiThrTree)malloc(sizeof(BiThrNode))))
        exit (OVERFLOW);
    Thrt->LTag = Link; Thrt->RTag = Thread;
    Thrt->rchild = Thrt;           // 添加头结点
    if (!T) Thrt->lchild = Thrt;  // T为空二叉树
    ...
    return OK;
} // InOrderThreading
```

if (!T) Thrt->lchild = Thrt; // T为空二叉树

else {

Thrt->lchild = T; pre = Thrt;

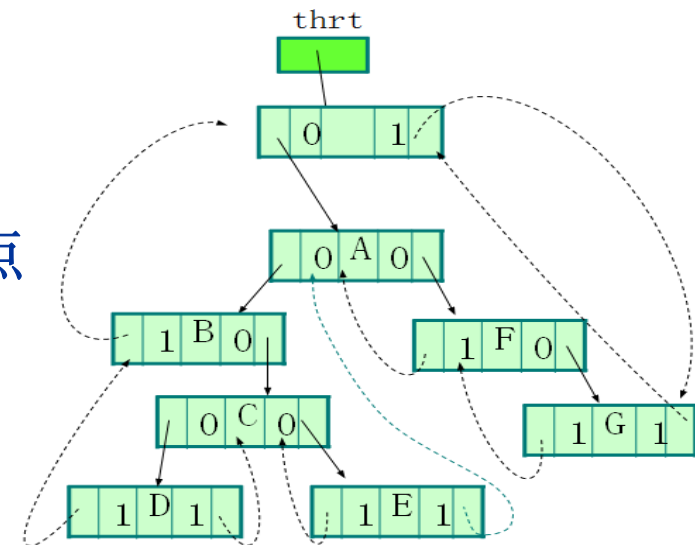
InThreading(T);

pre->rchild = Thrt; // 处理最后一个结点

pre->RTag = Thread;

Thrt->rchild = pre;

}



建立线索链表的参考算法（用顺序栈）

```
void creat_thread(struct BiTNode *t)
{ struct BiTNode *st[maxleng+1]; //指针栈
  int top=0;                      //置空栈
  struct BiTNode *pre=NULL;       //前驱结点指针
  do
  { while(t)                      //根指针t表示的为非空二叉树
    { if (top==maxleng)
      { exit(OVERFLOW);           //栈已满, 退出
        st[++top]=t;              //根指针进栈
        t=t->lchild;              //t移向左子树
      }
    }
  }
```

```

if (top)                                     //为非空栈
{ t=st[top--];                             //弹出根指针
  printf("%c", t->data);                     //访问根结点
  if (t->lchild!=NULL) t->ltag=0;             //左指针为孩子
  else {t->ltag=1; t->lchild=pre; }          //左指针为线索
  if (pre!=NULL)
    if (pre->rchild!=NULL) pre->rtag=0;      //右指针为孩子
    else {pre->rtag=1; pre->rchild=t; }      //右指针为线索
  pre=t;                                   //pre与t保持前后
  t=t->rchild;                             //遍历右子树
}
} while(top||t);
pre->rtag=1;                               //最后一节点右标记线索
}

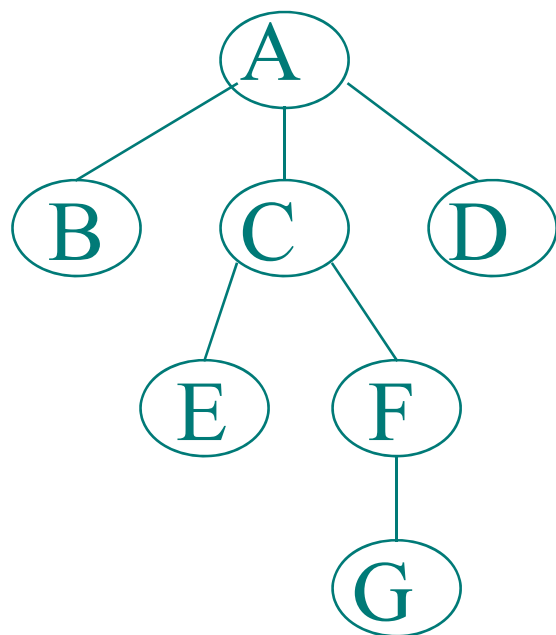
```

6.4 树和森林

6.4.1 树的存储结构

- 一、双亲表示法/顺序表示法
- 二、孩子表示法/链接表表示法
- 三、孩子链表表示法
- 四、带双亲的孩子链表表示法
- 五、树的二叉链表(孩子-兄弟) 存储表示法

1. 双亲表示法/数组表示法/顺序表示法? :



	data	parent
0	A	-1
1	B	0
2	C	0
3	D	0
4	E	2
5	F	2
6	G	5

$r = 0$
 $n = 7$

C语言的类型描述:

```
#define MAX_TREE_SIZE 100
```

结点结构:

```
typedef struct PTNode {  
    TElemType data;  
    int parent; // 双亲位置域
```

```
} PTNode;
```

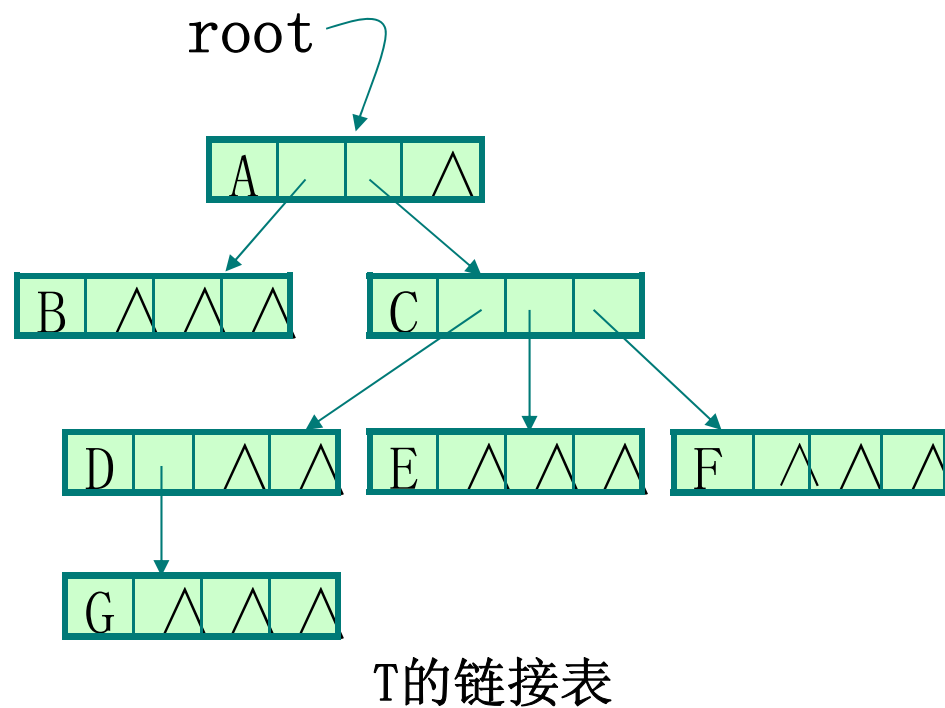
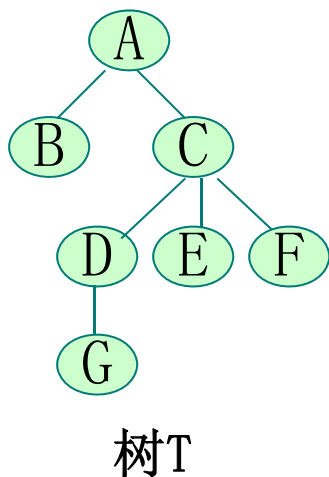
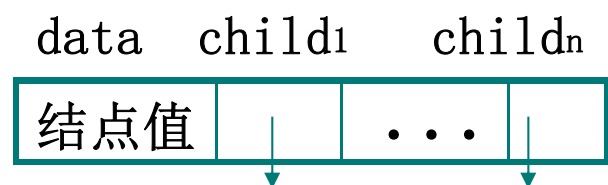
树结构:

```
typedef struct {  
    PTNode nodes[MAX_TREE_SIZE];  
    int r, n; // 根结点的位置和结点个数  
} PTree;
```

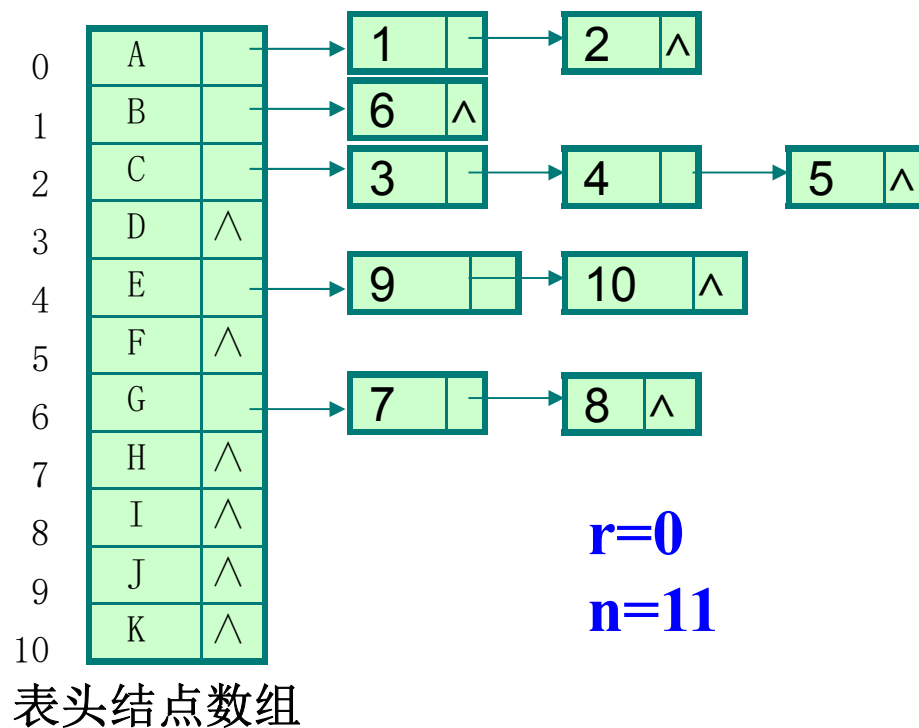
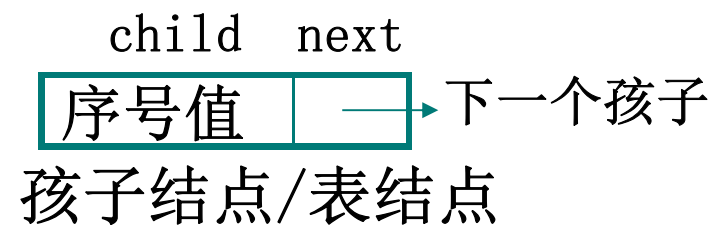
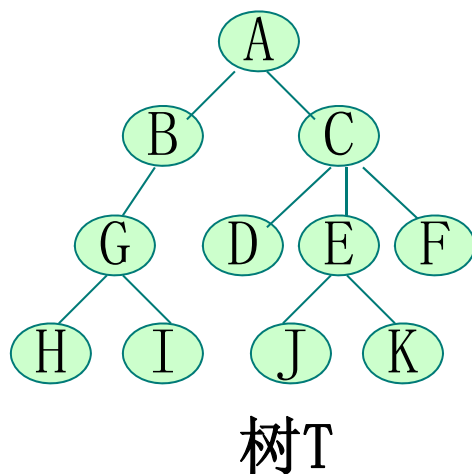
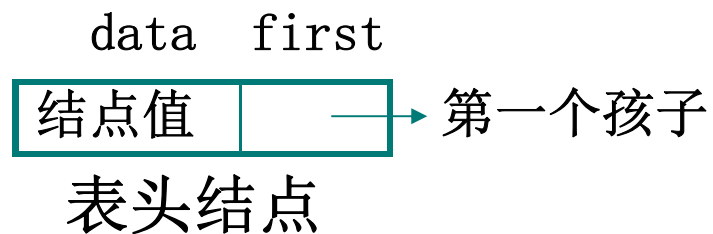
data	parent
------	--------

2. 孩子表示法/链接表表示法

(1) 固定大小的结点格式，设树T的度为n



3. 孩子链表表示法/单链表表示法



C语言的类型描述:

```
typedef struct CTNode {  
    int      child;  
    struct CTNode *next;  
} *ChildPtr;  
typedef struct {  
    TElemType  data;  
    ChildPtr  firstchild; // 孩子链的头指针  
} CTBox;  
typedef struct {  
    CTBox nodes[MAX_TREE_SIZE];  
    int  n, r;    // 结点数和根结点的位置  
} CTree;
```

孩子结点结构



双亲结点结构



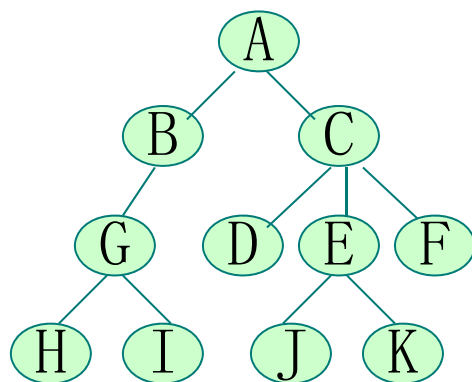
树结构

4.带双亲的孩子链表表示法

data parent first

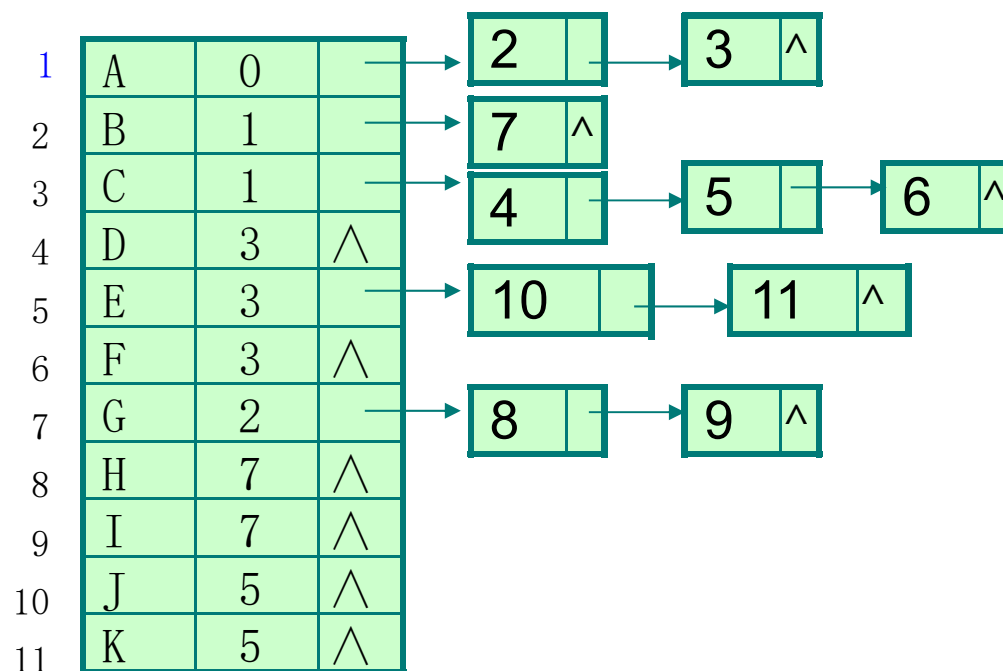


表头结点



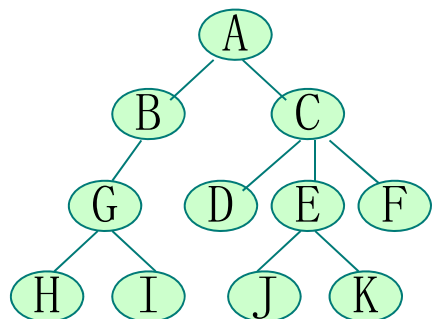
树T

child next



表头结点数组

5. 孩子兄弟表示法/二叉链表



firstchild data nextsibling



左孩子

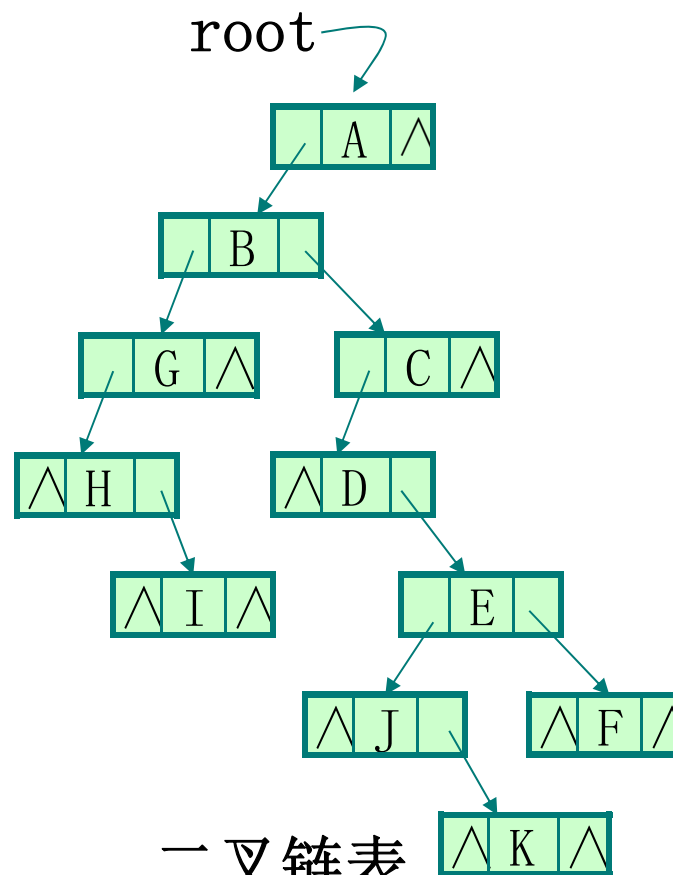
右兄弟

```
typedef struct CSNode{
```

```
    ElemType    data;
```

```
    struct CSNode *firstchild, *nextsibling;
```

```
} CSNode, *CSTree;
```

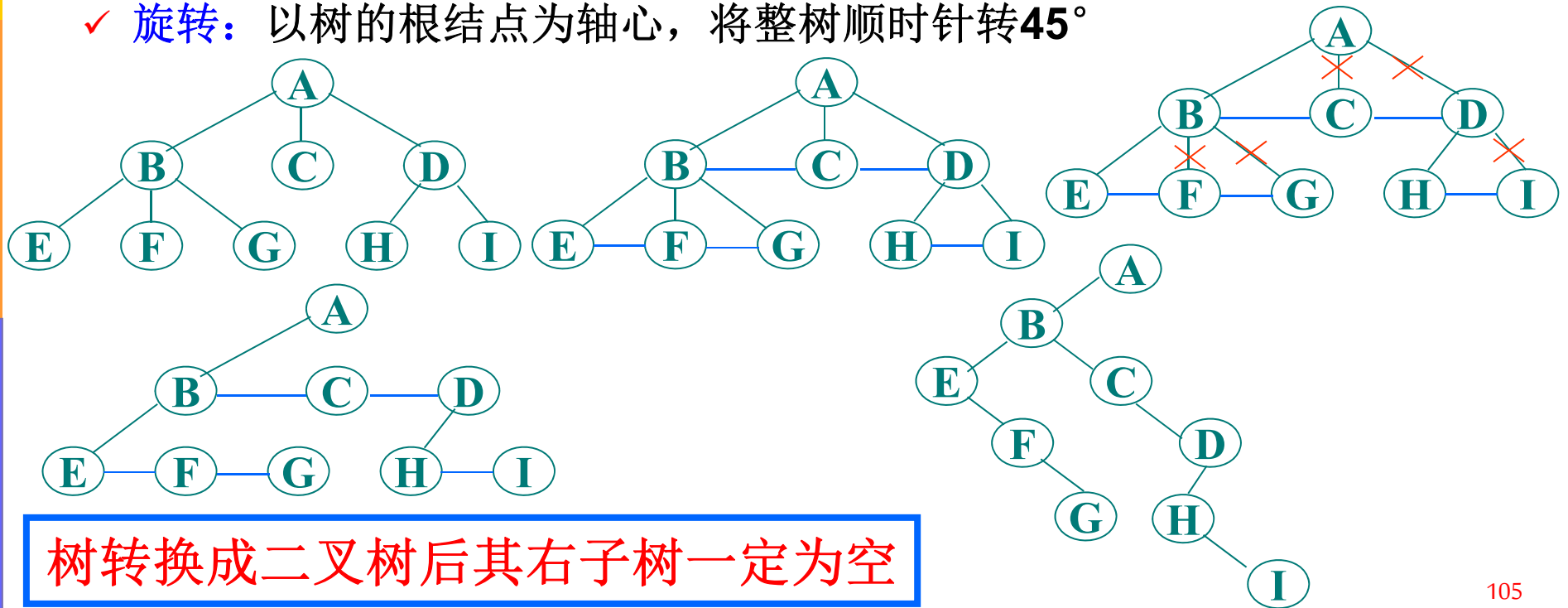


二叉链表

6.4.2 树与二叉树的转换

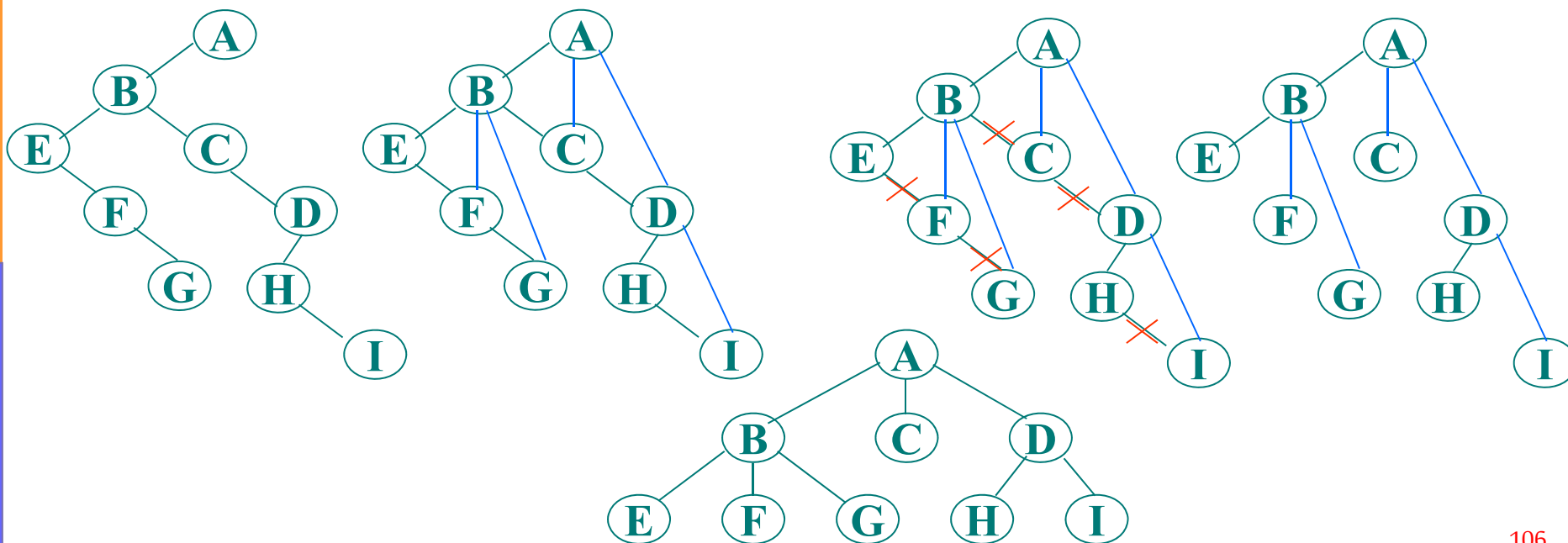
□ 将树转换成二叉树（基于树的二叉链表表示）

- ✓ 加线：在兄弟之间加一连线
- ✓ 抹线：对每个结点，除了其左孩子外，去除其与其余孩子之间的关系
- ✓ 旋转：以树的根结点为轴心，将整树顺时针转 45°



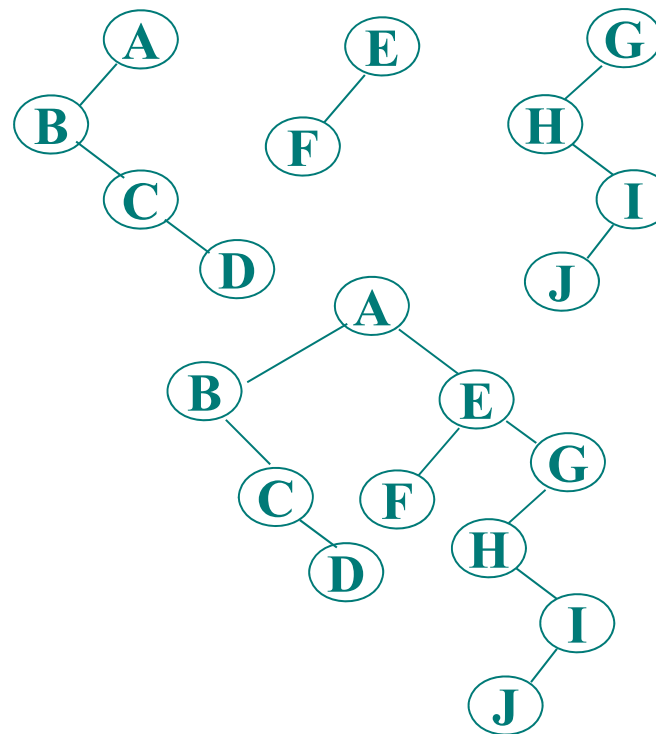
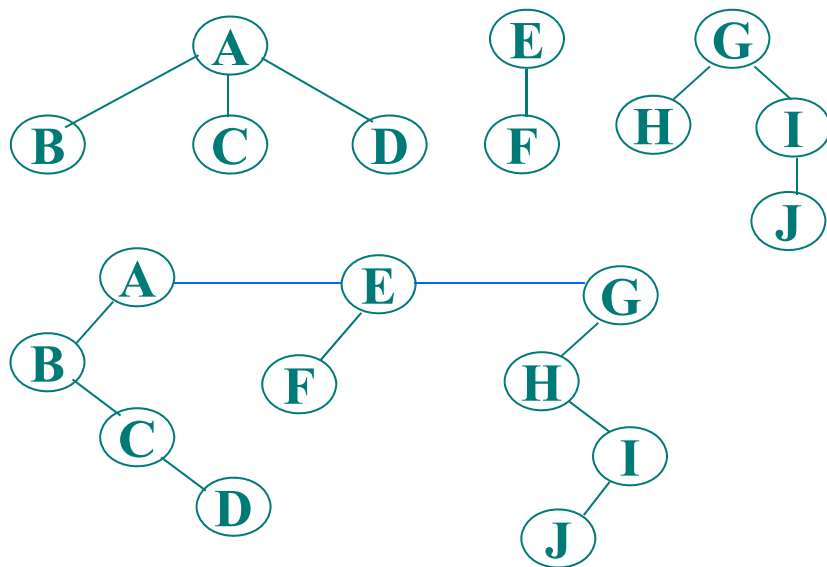
□ 将二叉树转换成树

- ✓ **加线：**若p结点是双亲结点的左孩子，则将p的右孩子，右孩子的右孩子，……沿分支找到的所有右孩子，都与p的双亲用线连起来
- ✓ **抹线：**抹掉原二叉树中双亲与右孩子之间的连线
- ✓ **调整：**将结点按层次排列，形成树结构



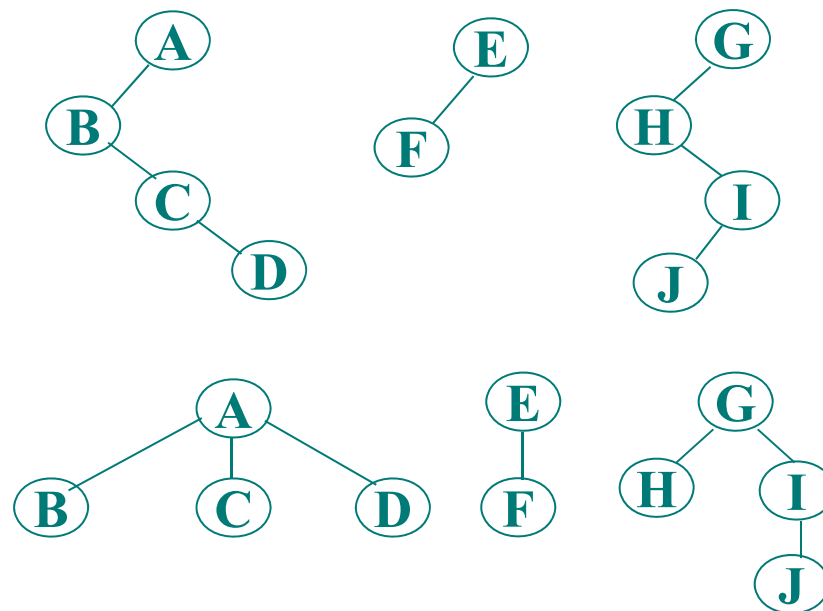
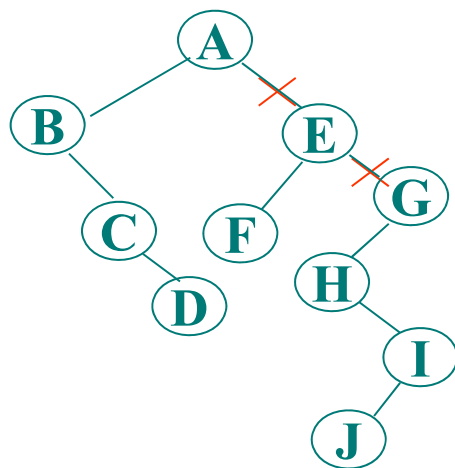
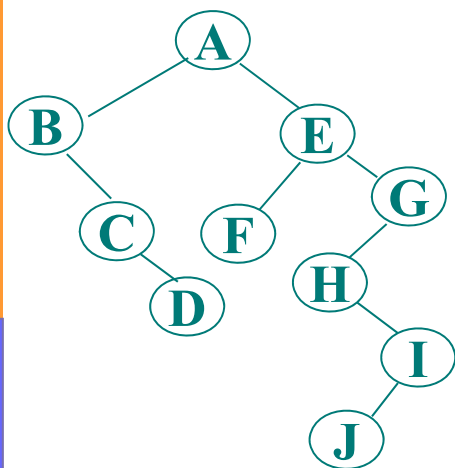
□ 森林转换成二叉树


- ✓ 将各棵树分别转换成二叉树
- ✓ 将每棵**二叉树**的根结点用线相连
- ✓ 以第一棵树根结点为二叉树的根，再以根结点为轴心，顺时针旋转，构成二叉树型结构



❑ 二叉树转换成森林

- ✓ **抹线：** 将二叉树中根结点与其右孩子连线，及沿右分支搜索到的所有右孩子间连线全部抹掉，使之变成孤立的二叉树
- ✓ **还原：** 将孤立的二叉树还原成树





- 树的各种操作均可对应二叉树的操作来完成。

- 注意：

- ✓和树对应的二叉树，其左、右子树的概念已改变为：

- 左是孩子，右是兄弟

森林和二叉树的对应关系

森林

$$F = (T_1, T_2, \dots, T_n);$$

$$T_1 = (\text{root}, T_{11}, T_{12}, \dots, T_{1m});$$

$$F_1 = (T_{11}, T_{12}, \dots, T_{1m});$$

二叉树

$$B = (\text{root}, \text{LBT}, \text{RBT});$$

由森林转换成二叉树的转换规则为：

若 $F = \Phi$ ，则 $B = \Phi$ ；

否则，

由 $\text{ROOT}(T_1)$ 对应得到 **root**；

由 $F_1 = (T_{11}, T_{12}, \dots, T_{1m})$ 对应得到 **LBT**；

由 (T_2, T_3, \dots, T_n) 对应得到 **RBT**。

由二叉树转换为森林的转换规则为:

若 $B = \Phi$, 则 $F = \Phi$;

否则,

由 **root** 对应得到 **ROOT(T_1)**;

由**LBT** 对应得到 **$F_1 = (T_{11}, T_{12}, \dots, T_{1m})$** ;

$T_1 = (\text{root}, T_{11}, T_{12}, \dots, T_{1m})$;

由**RBT** 对应得到 **(T_2, T_3, \dots, T_n)** 。

树和森林的遍历

一、树的遍历

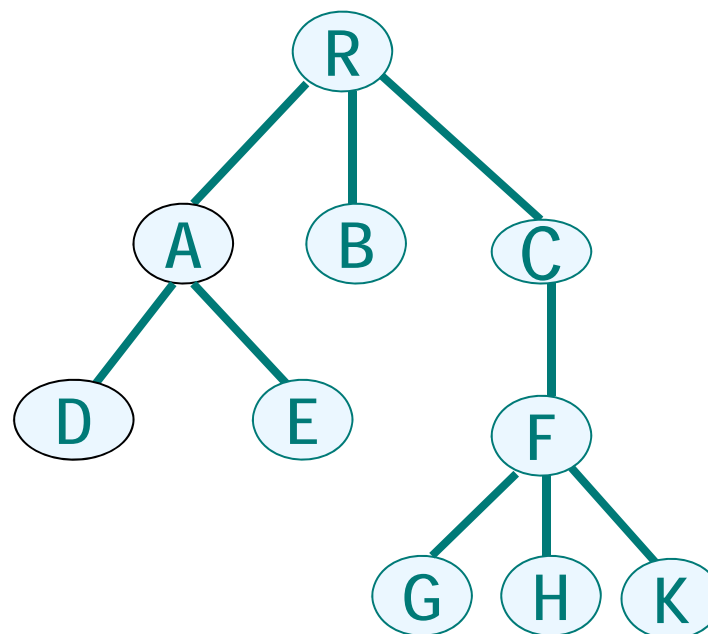
□ 先根遍历

若树为空，则空操作

否则

(1) 访问树的根结点

(2) 依次先根遍历每棵子树



例：右图所示树的先根遍历序列：RADEBCFGHK

□后根遍历

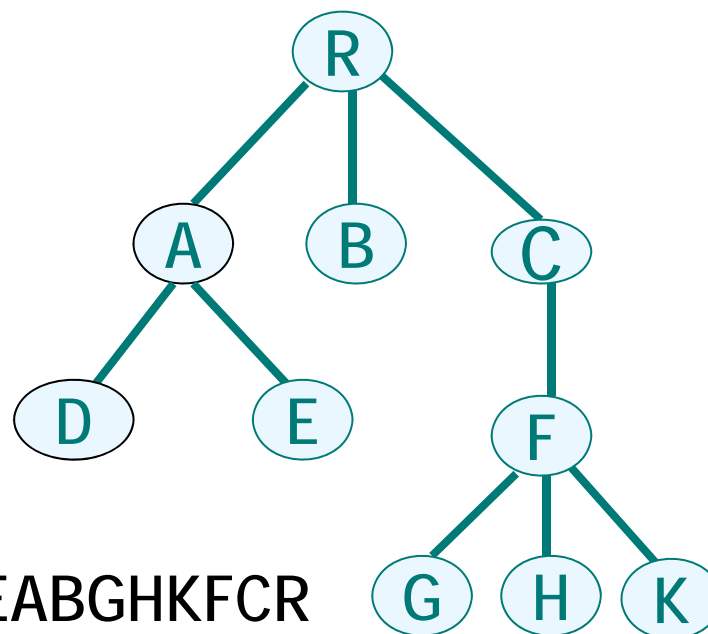
若树为空，则空操作

否则

(1) 依次后根遍历每棵子树

(2) 访问树的根结点

例：右图所示树的后根遍历序列：DEABGHKFCR



与对应二叉树的中序遍历序列一致

二、森林的遍历

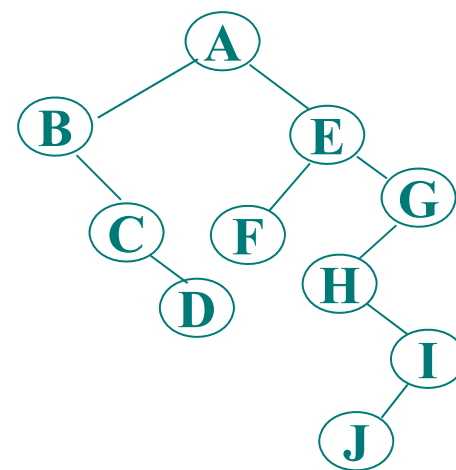
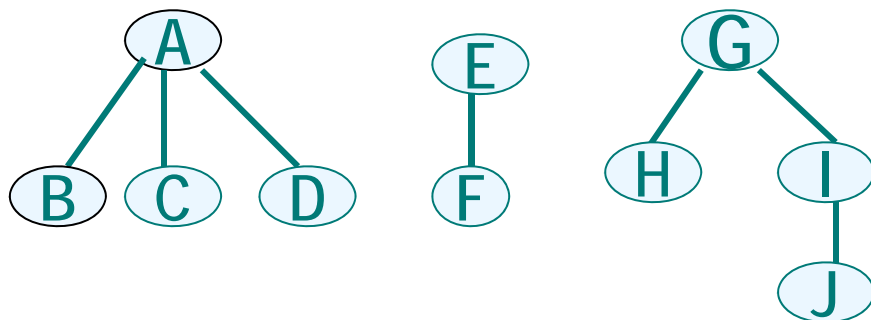
□ 先序遍历森林（相当于对对应的二叉树进行先序遍历）

若森林为空，则空操作，否则

- (1) 访问第一棵树的根结点
- (2) 先序遍历第一棵树中根结点的子树森林
- (3) 先序遍历除去第一棵树后余下的树构成的森林

例：下图所示森林的先序遍历序列为：

ABCDEFGHIJ



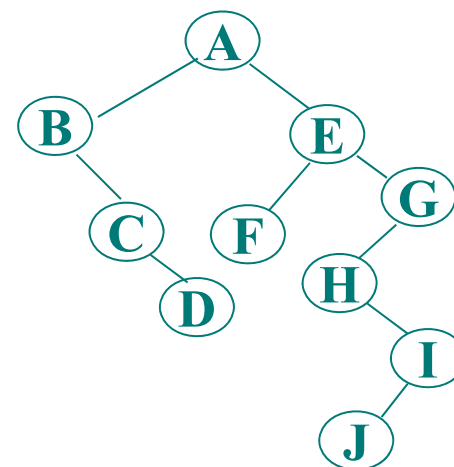
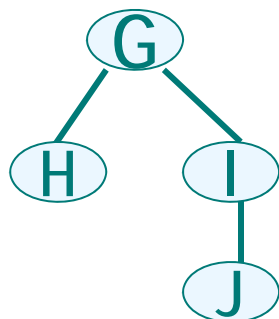
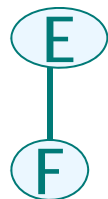
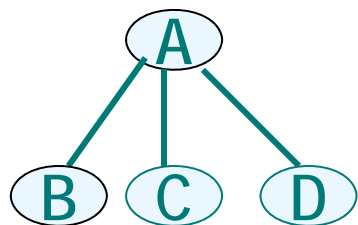
□ 中序遍历森林（相当于对对应的二叉树进行中序遍历）

若森林为空，则空操作，否则

- （1）中序遍历第一棵树根结点的子树森林
- （2）访问第一棵树的根结点
- （3）中序遍历除第一棵树后余下的树构成的森林

例：下图所示森林的中序遍历序列为：

BCDAFEHJIG



树的遍历和二叉树遍历的对应关系

树	森林	二叉树
先根遍历	先序遍历	先序遍历
后根遍历	中序遍历	中序遍历

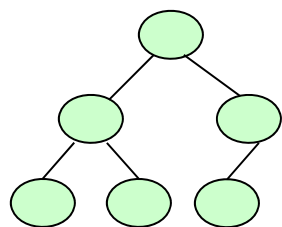
6.6 哈夫曼 (Huffman) 树及其应用

6.6.1. 最优二叉树 (Huffman 树)

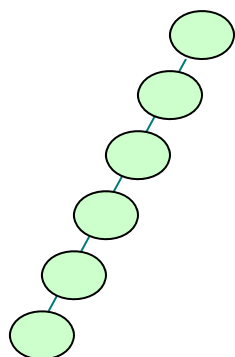
□ 路径长度

从树中一个结点到另一个结点之间的分支构成这两个结点之间的**路径**，路径上的分支数目称做**路径长度**。

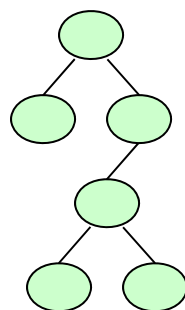
□ **树的路径长度**：从树根到每一结点的路径长度之和。



T₁



T₂



T₃

$$PL(T_1) = 1 + 1 + 2 + 2 + 2 = 8$$

$$PL(T_2) = 1 + 2 + 3 + 4 + 5 = 15$$

$$PL(T_3) = 1 + 1 + 2 + 3 + 3 = 10$$

➤ 当n个结点的二叉树为完全二叉树时, $PL(T)$ 具有最小值

$$\because \text{结点} i \text{的层} = \lfloor \log_2 i \rfloor + 1$$

$$\begin{aligned} \text{树} T \text{的根到结点} i \text{的路径长度} &= \text{结点} i \text{的层} - 1 \\ &= \lfloor \log_2 i \rfloor \end{aligned}$$

$$\therefore PL(T) = \lfloor \log_2 1 \rfloor + \lfloor \log_2 2 \rfloor + \dots + \lfloor \log_2 n \rfloor$$

$$= \sum_{i=1}^n \lfloor \log_2 i \rfloor$$

➤ 当n个结点的二叉树为单枝树时, $PL(T)$ 具有最大值:

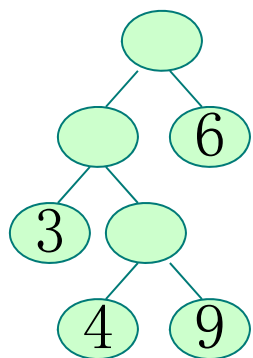
$$PL(T) = 0 + 1 + 2 + \dots + (n-1) = n(n-1)/2$$

□ 树或二叉树T的带权路径长度——每个叶子的权与根到该叶子的路径长度的乘积之和，记作WPL(T)。

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

其中：n --- 树T的叶子数目 w_k --- 叶子k的权

l_k --- 树T的根到叶子k的路径长度

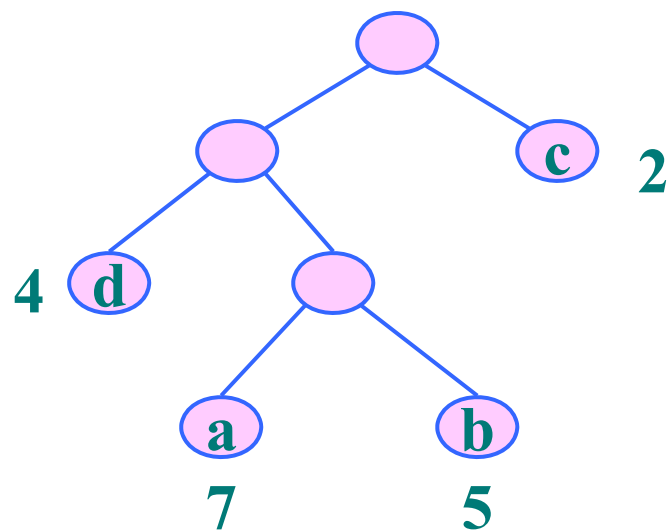


T₁

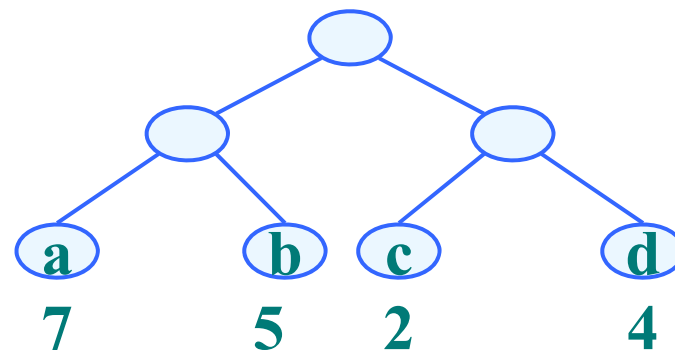
$$WPL(T_1) = 3*2 + 4*3 + 9*3 + 6*1 = 51$$

例 以权值分别为7, 5, 2, 4 的4个结点为叶子结点构造二叉树

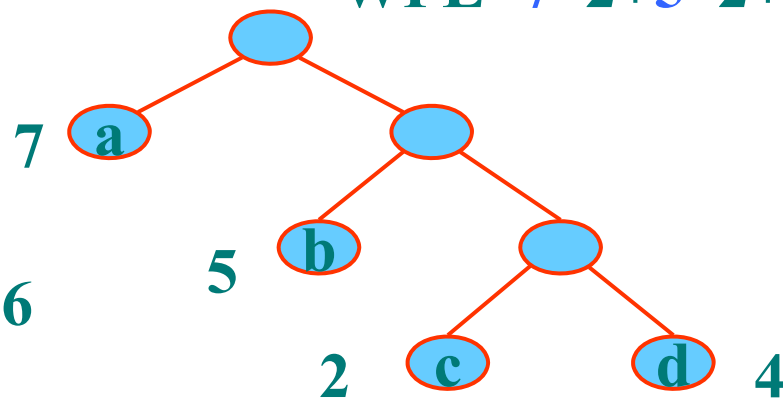
$$WPL = \sum_{k=1}^n W_K L_K$$



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



$$WPL = 7*2 + 5*2 + 2*2 + 4*2 = 36$$



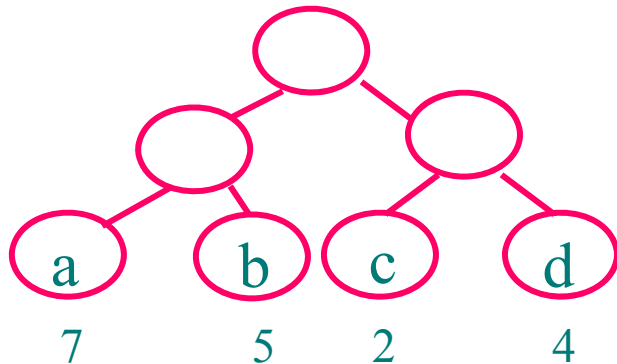
$$WPL = 7*1 + 5*2 + 2*3 + 4*3 = 35$$

带权路径长度达到最小

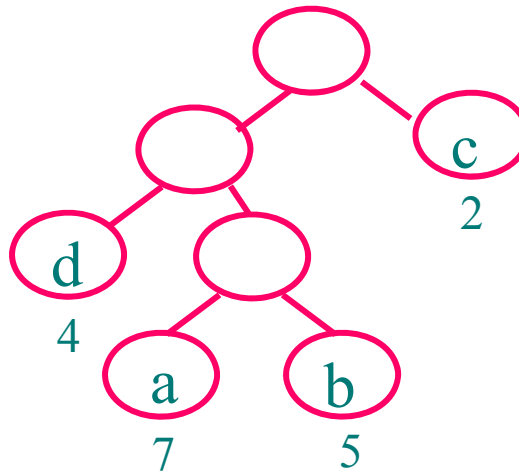
□ 哈夫曼树/最佳树/最优树

在具有n个相同叶子的各二叉树中，WPL最小的二叉树。

完全二叉树

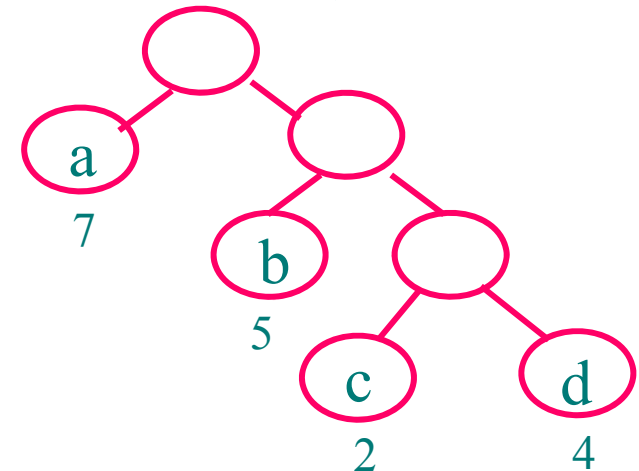


$$WPL_a = 36$$



$$WPL_b = 46$$

Huffman 树



$$WPL_c = 35$$

$$= 7 * 1 + 5 * 2 + 2 * 3 + 4 * 3$$

- (1) 完全二叉树并不一定是Huffman树;
- (2) 在赫夫曼树中, 权值大的结点离根近;
- (3) Huffman树不唯一, 但WPL一定相等。

Huffman算法

- (1) 以权值分别为 W_1, W_2, \dots, W_n 的 n 个结点，构成 n 棵二叉树 T_1, T_2, \dots, T_n ，并组成森林 $F = \{T_1, T_2, \dots, T_n\}$ ，
 - ✓ 每棵二叉树 T_i 仅有一个权值为 W_i 的根结点；
- (2) 在 F 中选取两棵根结点权值最小的树作为左右子树构造一棵新二叉树，并且置新二叉树根结点权值为左右子树上根结点的权值之和
 - ✓ 根结点的权值=左右孩子权值之和，
 - ✓ 叶结点的权值= W_i
- (3) 从 F 中删除这两棵二叉树，同时将新二叉树加入到 F 中；
- (4) 重复(2)，(3)直到 F 中只含一棵二叉树为止(这棵就是Huffman树)。

下面实例中在哈夫曼算法基础上引入了排序

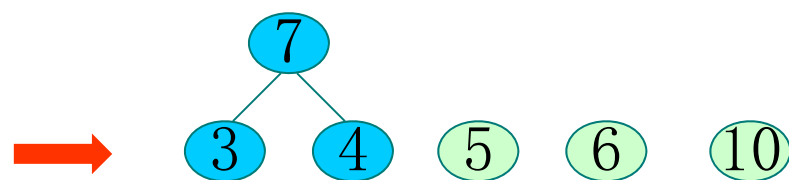
例 给定权集合 {4, 5, 3, 6, 10}，构造哈夫曼树

1. 按权值大小排序： 3, 4, 5, 6, 10

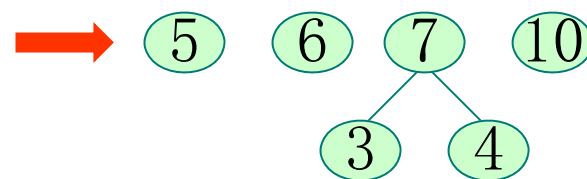
2. 生成森林：



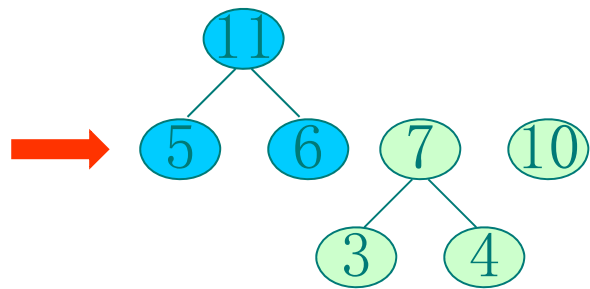
3. 合并两棵权最小的二叉树, 并排序, 直到为一棵二叉树:



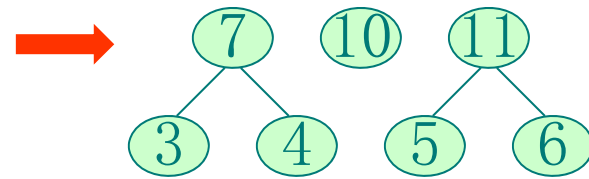
选择合并



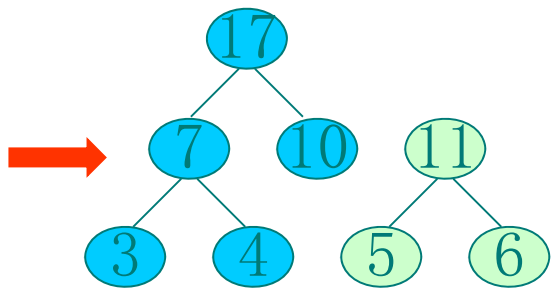
排序



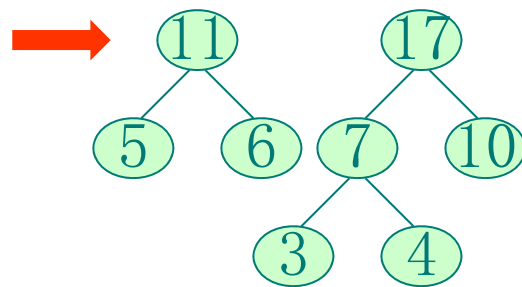
选择合并



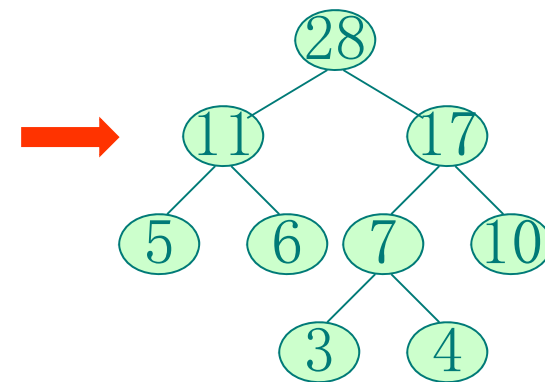
排序



选择合并



排序



哈夫曼树

6.6.2 哈夫曼编码/最小冗余码

➤ ASCII码/定长码

ab12: 01100001 01100010 00110001 00110010

97

98

49

50

➤ 哈夫曼码/不定长码

能按字符的使用频度, 使文本代码的总长度具有最小值。

例. 给定有18个字符组成的文本:

A A D A T A R A E F R T A A F T E R

求各字符的哈夫曼码。

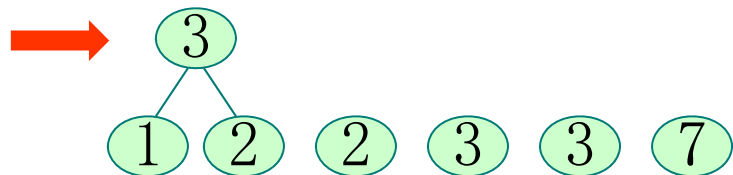
(1) 统计:

字符	A	D	E	F	T	R
频度	7	1	2	2	3	3

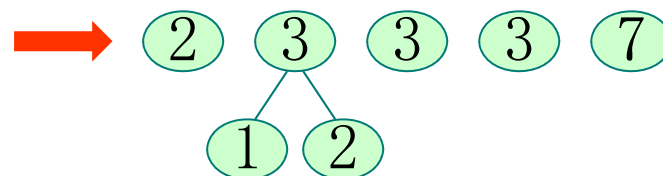
(2) 构造Huffman树:

① ② ② ③ ③ ⑦

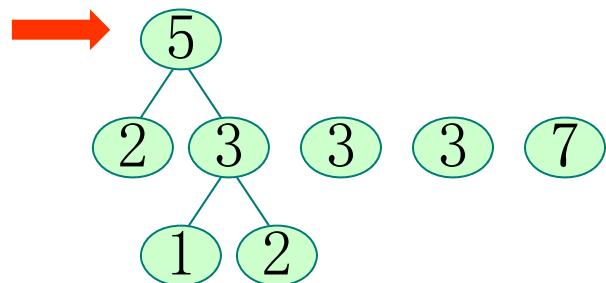
1 2 2 3 3 7



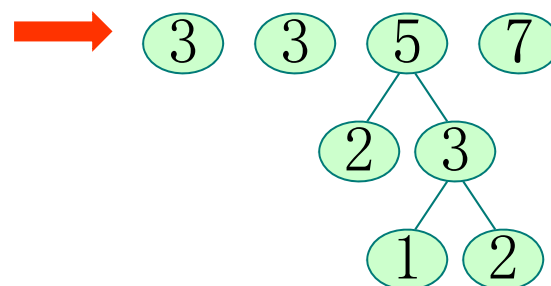
合并1和2



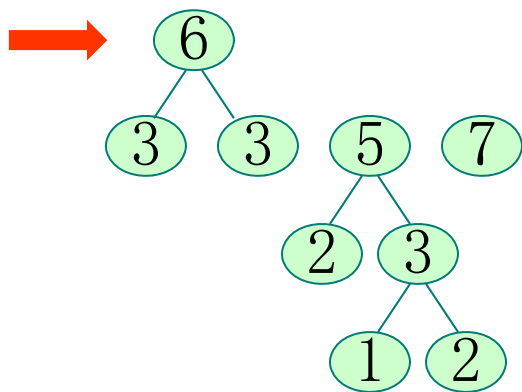
排序



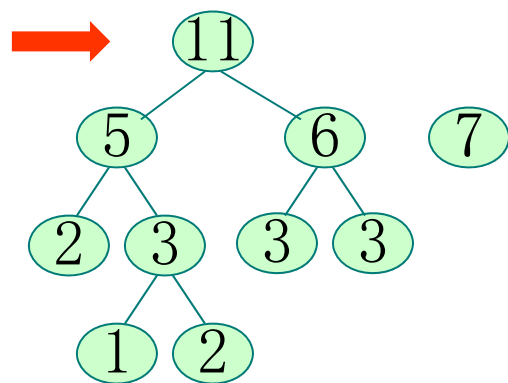
合并2和3



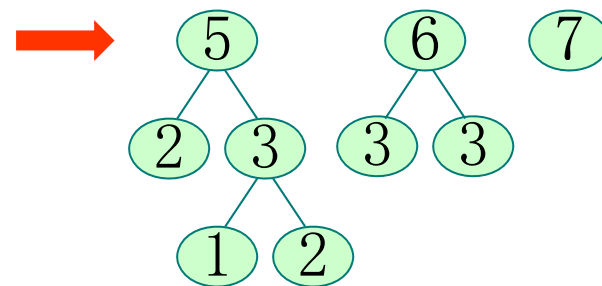
排序



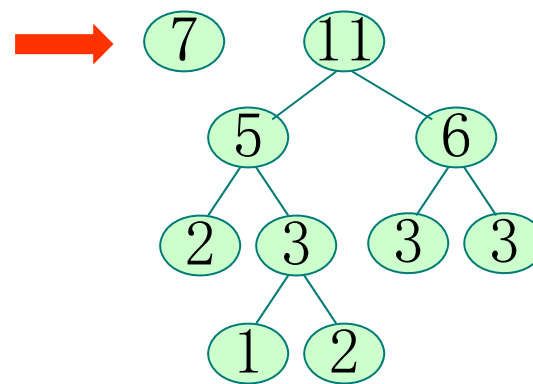
合并3和3



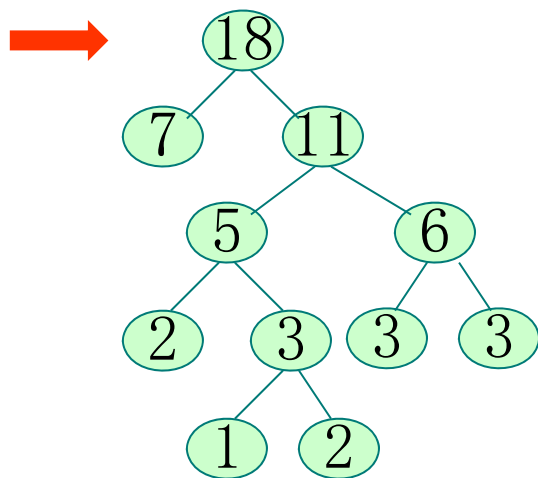
合并5和6



排序

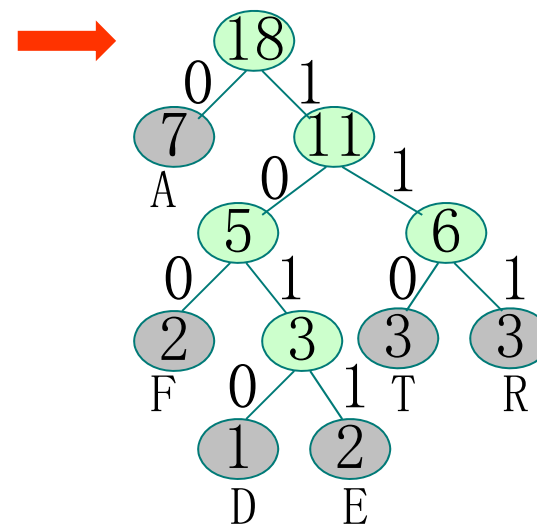


排序



合并7和11得Huffman树

(4) 确定Huffman编码:

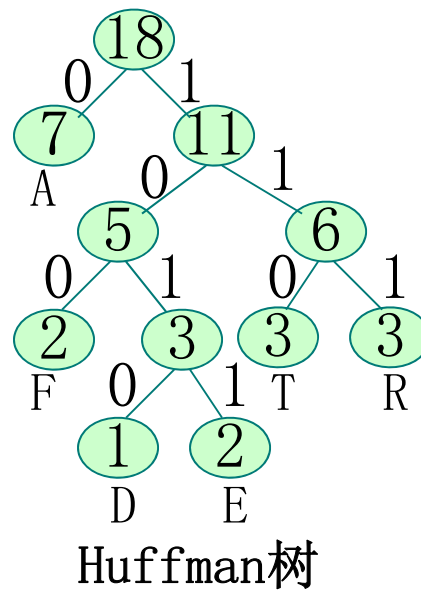


叶结点根据权值附上字符，
分支标数的Huffman树

字 符	A	D	E	F	T	R
频 度	7	1	2	2	3	3
编 码	0	1010	1011	100	110	111

特点：任一编码不是其它编码的前缀

如何译码？



例. 给定代码序列:

0 0 1 0 0 0 1 1 1 0 1 0 1 0 1 0 1 1 1 10

文本为: A A F A R A D E T

哈夫曼编码算法的实现

Huffman树和编码的特点:

- 哈夫曼树中没有度为1的结点。
- 若有 n 个叶子结点，则其共有 $2n-1$ 个结点。
- Huffman编码时是从叶子走到根；而译码时又要从根走到叶子，因此每个结点需要增开双亲指针分量。
- 实现时，要用到顺序和链式两种存储结构。

```
typedef struct{
    unsigned int weight; //权值分量（可放大取整）
    unsigned int parent, lchild, rchild; //双亲和孩子分量
}HTNode, *HuffmanTree; //用动态数组存储Huffman树
typedef char** HuffmanCode; //动态数组存储Huffman编码表
```

先构造Huffman树HT，再求出n个字符的Huffman编码HC。

Void HuffmanCoding(HuffmanTree &HT, HuffmanCode &HC,

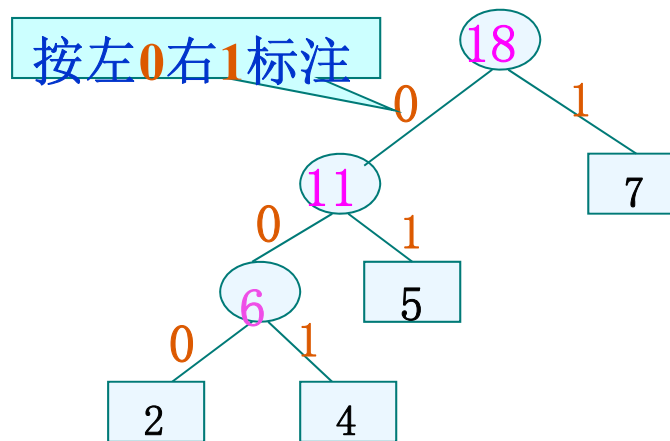
int *w, int n) //算法6.12

*w存放n个字符的权值

按左0右1标注

1
2
3
4
5
6
7

w	P	L	R
7	7	0	0
5	6	0	0
2	5	0	0
4	5	0	0
6	6	3	4
11	7	5	2
18	0	6	1



w	7	5	2	4
Code	1	01	000	001

□ 树的定义和术语

- ✓ 掌握树的相关概念, 包括树、结点的度、树的度、分支结点、叶子结点、孩子结点、双亲结点、树的深度、森林等定义。

□ 二叉树的定义及性质

- ✓ 二叉树、满二叉树和完全二叉树的定义与性质

□ 重点掌握二叉树、树的存储结构

- ✓ 二叉树顺序存储结构和链式存储结构（二叉链表、**线索链表**）
- ✓ 树的双亲表示法、孩子表示法、二叉链表表示法

□ 重点掌握二叉树的基本运算和各种遍历算法的实现

□ 掌握线索二叉树的概念和相关算法的实现

□ 树、森林与二叉树之间的相互转换

□ 最优二叉树(哈夫曼树)的定义、构造及应用

□ 灵活运用二叉树解决一些综合应用问题