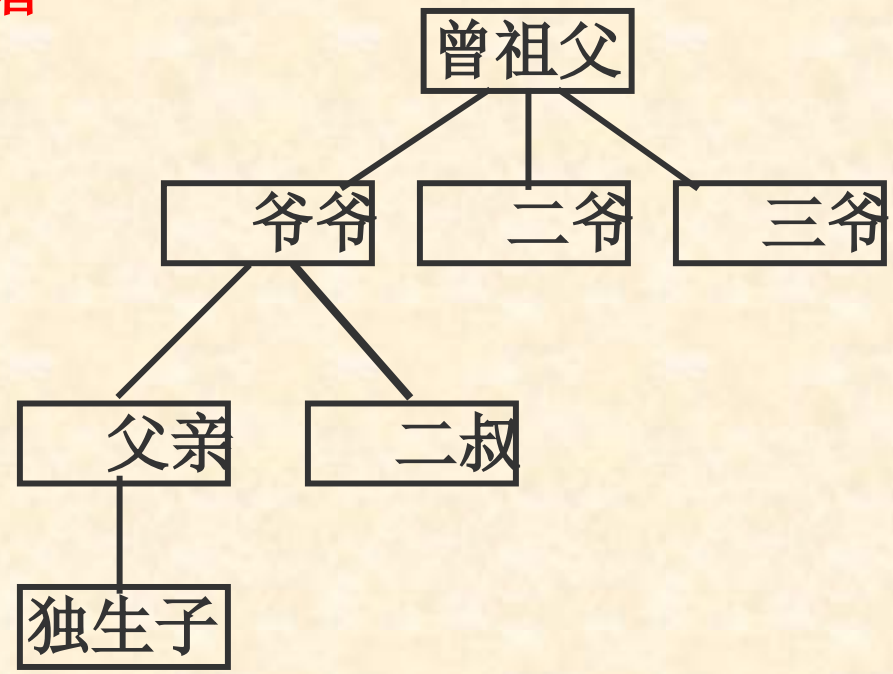
第六章 树和二叉树

族谱



树是以分支关系定义的层次结构。

树型结构是一类重要的非线性结构。

学习重点：

- 树的基本概念
- 二叉树的基本概念、存储结构、相关操作
- 树和森林与二叉树之间的相互转换
- 二叉树的应用

数据对象 D：

D 是具有相同特性的数据元素的集合。

数据关系 R：

若 D 为空集，则称为空树；

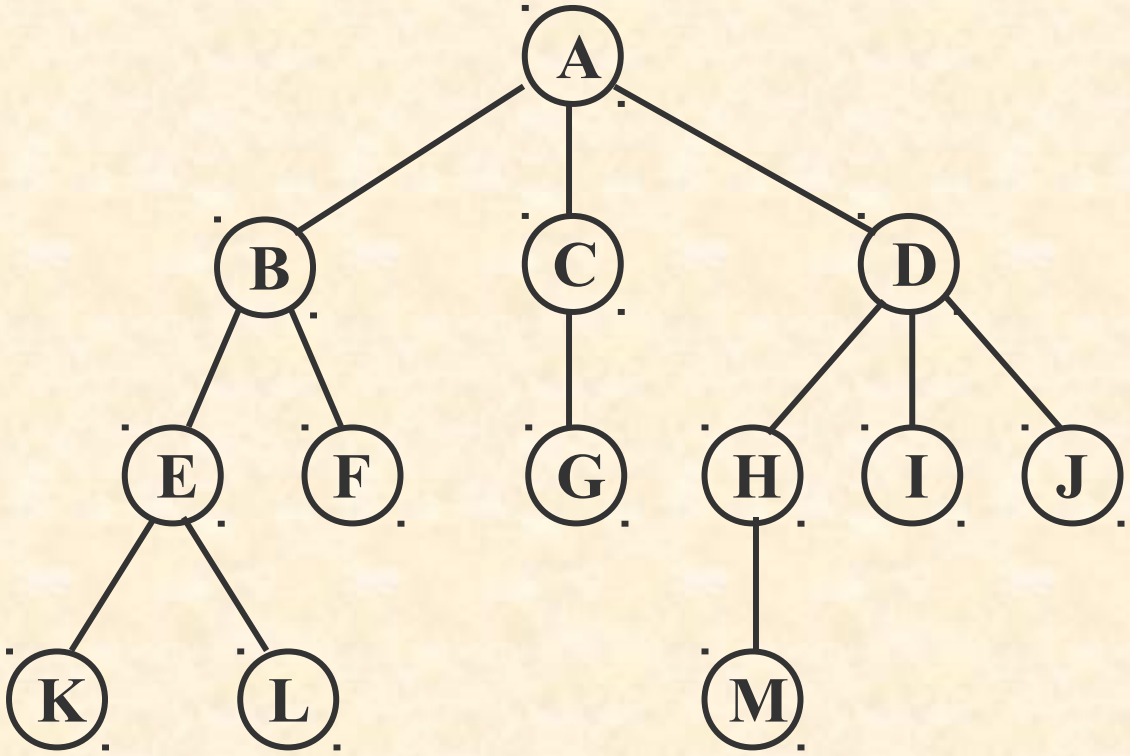
否则：

(1) 在 D 中存在唯一的称为根的数据元素 root，

(2) 当 n>1 时，其余结点可分为 m (m>0) 个互不相交的有限集 $T_1, T_2, \cdots, T_m$，其中每一棵子集本身又是一棵符合本定义的树，称为根 root 的子树。

6.1　树的定义和基本术语

树 (Tree)： 是具有层次结构的　**n(n≥0)** 个结点的有限集。



一般树

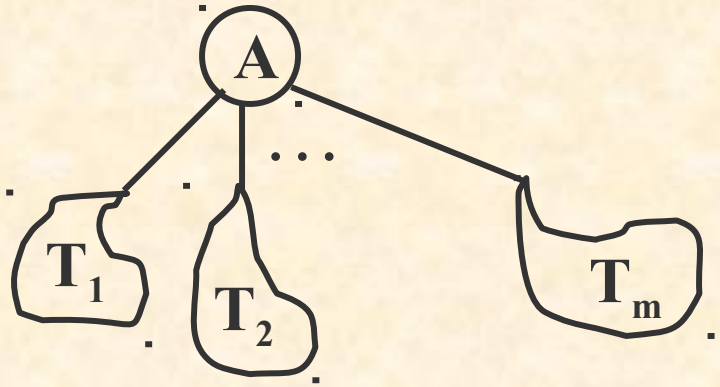
树 (Tree)： 是 $n(n\geq 0)$ 个结点的有限集。

$n = 0$ ， 空树 。

$n=1$ ，有且仅有一个称为根的结点。

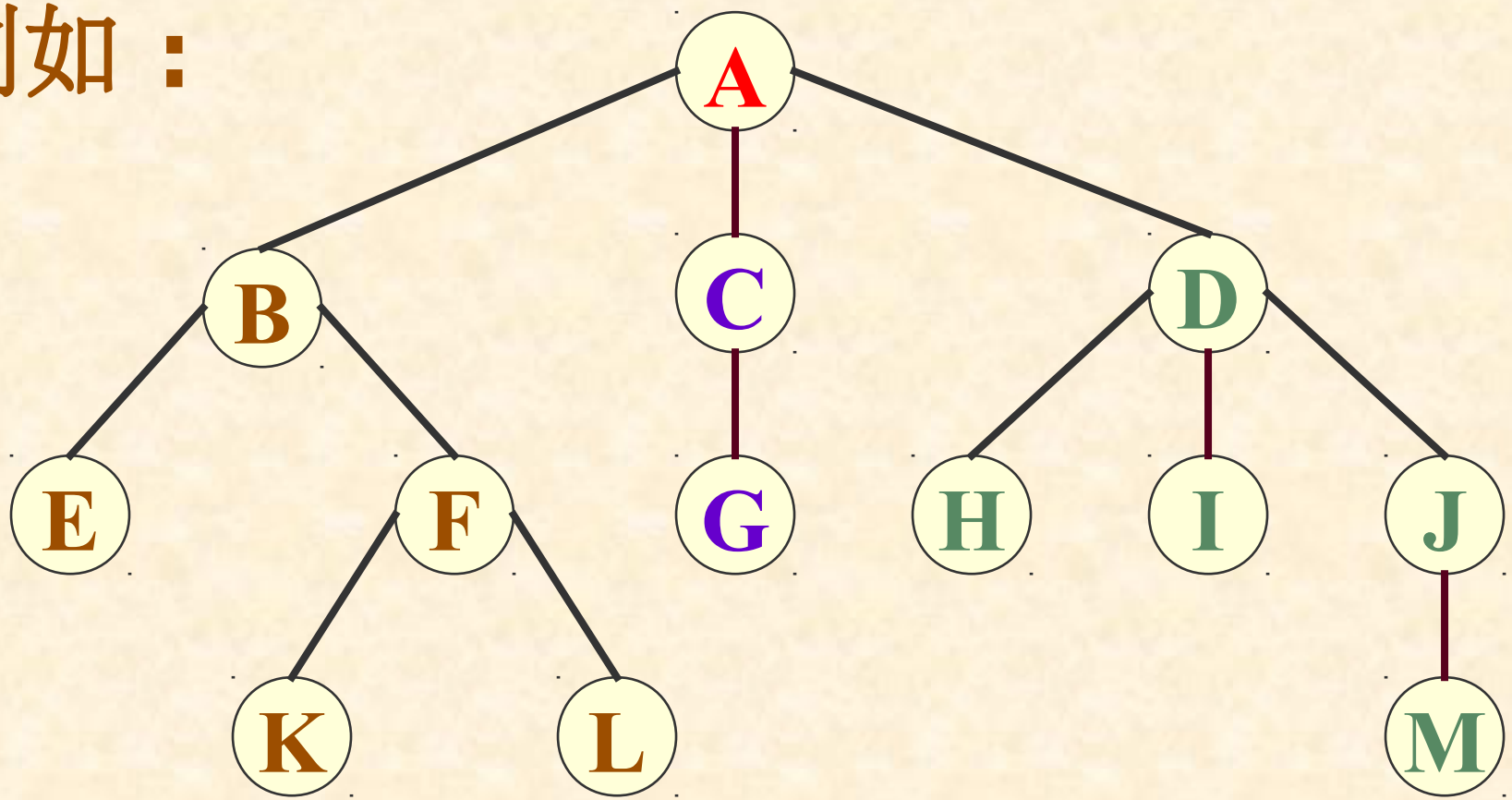
$n > 1$ ，除根结点外，其余结点可分为 $m(m > 0)$ 个互不相交的有限子集 $T_1, T_2, \ldots T_m$ ，其中每个子集都称为根结点的子树 。

只有根结点的树



$n > 1$

例如：



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

树根

T_1 T_2 T_3

基本操作:

- 查　找
- 插　入
- 删　除

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

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

基　本　术　语

结点：数据元素 + 若干指向子树的分支

结点的度：分支的个数（**拥有的子树数**）

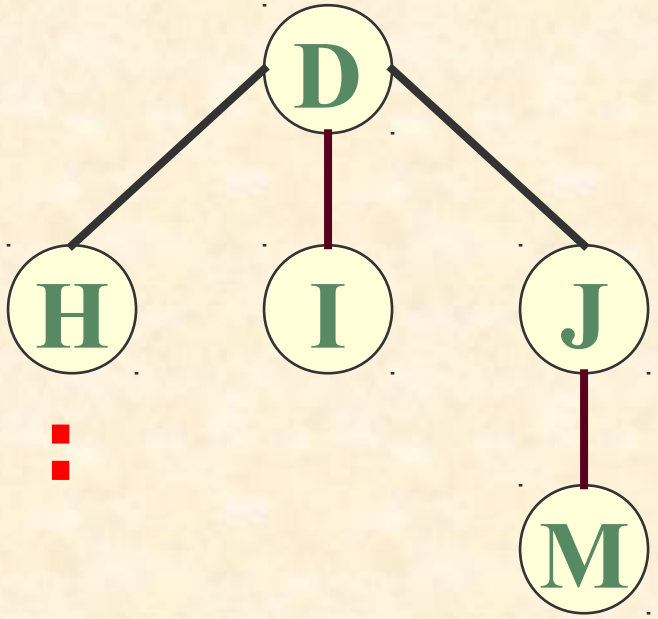
树的度：树中所有结点的度的最大值

叶子结点（终端节点）：

度为零的结点

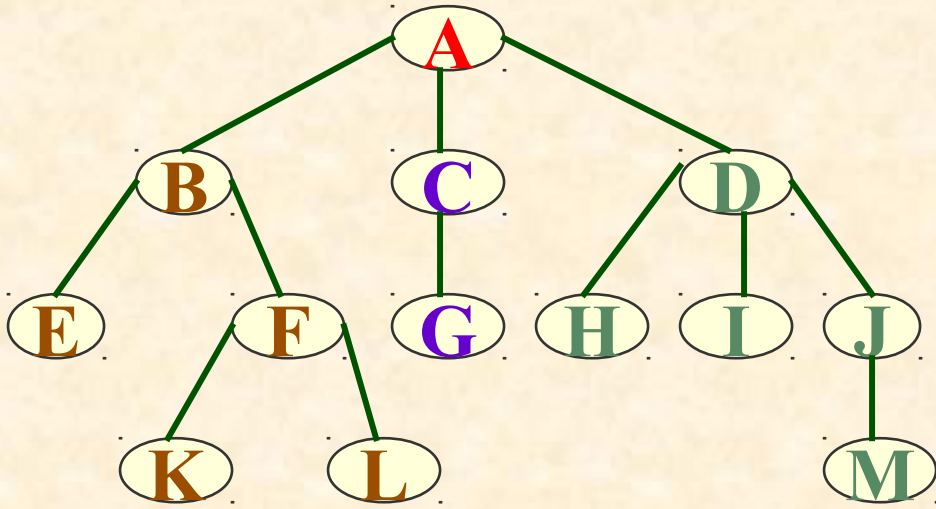
分支结点（非终端节点）：

度大于零的结点



孩子结点（节点的子树的根）、双亲结点、兄弟结点、堂兄弟

祖先结点、子孙结点



结点的层次： 结点的层次从根开始定义起，根为第一层，根的孩子为第二层。

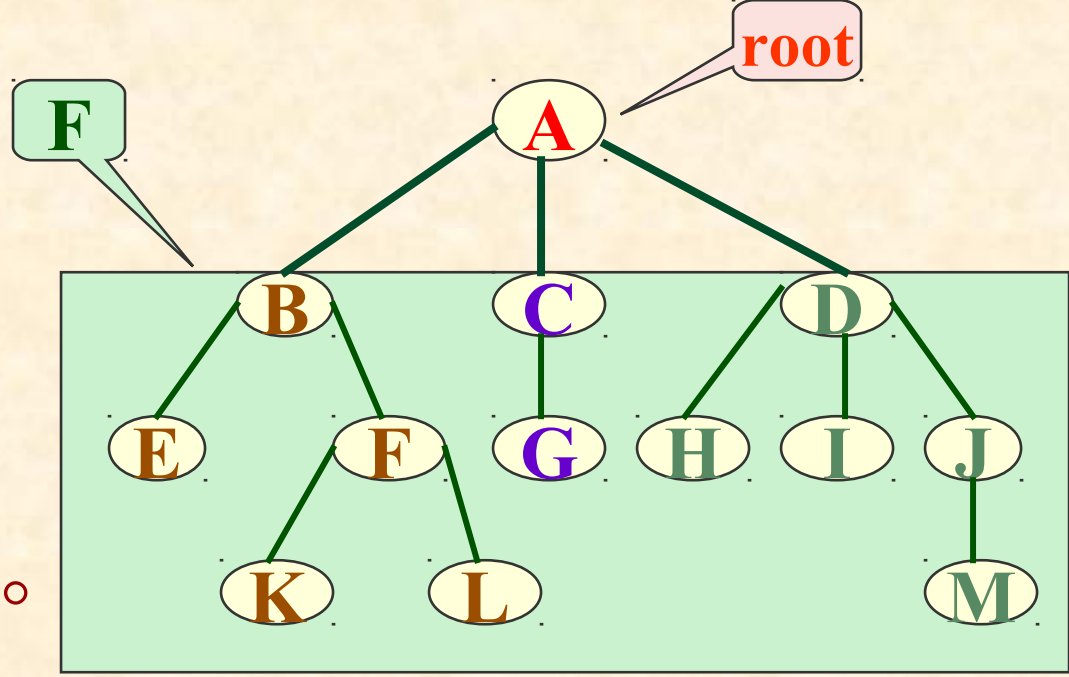
树的深度： 树中叶子结点所在的最大层次

如果将树中结点的各子树看成从左到右是有序的 (即不能互换) ，则称该树为有序树，否则称为无序树。

例， B,C,D 分别称为 A 的第 1 ， 2 ， 3 个儿子。

森林：

是 **m** （m≥0）棵互
不相交的树的集合。



任何一棵非空树是一个二元组

Tree = （root， F）

其中： root 被称为根结点，

F 被称为子树森林

对比树型结构和线性结构的结构特点

线性结构

第一个数据元素
（无前驱）

最后一个数据元素
（无后继）

其它数据元素
（一个前驱、
一个后继）

树型结构

根结点
（无前驱）

多个叶子结点
（无后继）

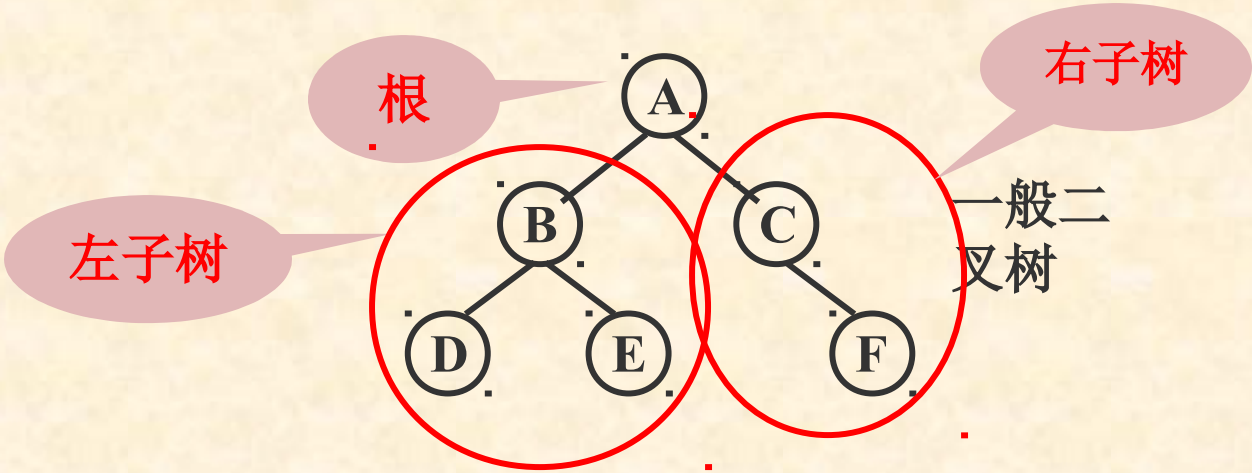
其它数据元素
（一个前驱、
多个后继）

6.2

二叉树的类型定义

二叉树或为**空树**；或是由一个**根结点**加上**两棵**分别称为**左子树**和右子树的、**互不交的二叉树**组成。

二叉树是一个**递归定义**。



树的子树次序不作规定，二叉树的两个子树有**左、右之分**。

树中结点的度没有限制，二叉树中结点的度只能取
0 、 1 、 2 。

6.1 节关于树的基本术语也都适用于二叉树。

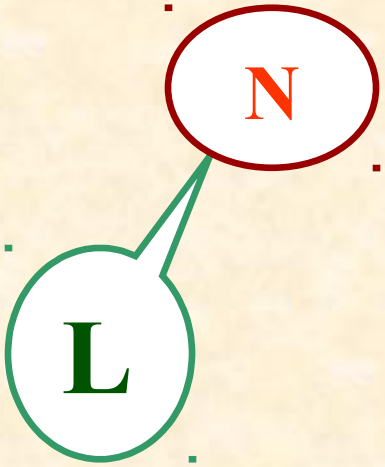
根据定义，二叉树通常具有 5 种基本形态：

空树

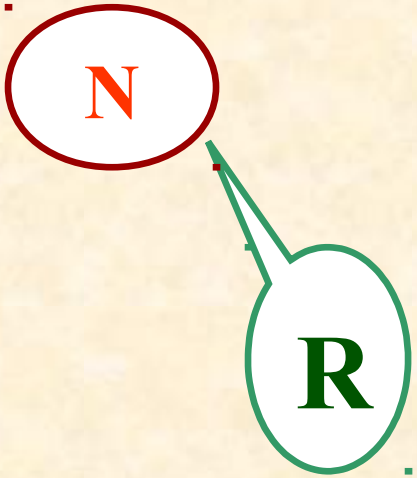
只含根结点

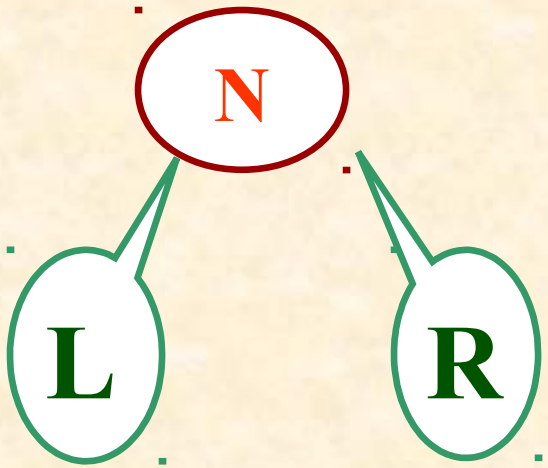
左右子树均不为空树

右子树为空树



左子树为空树





二叉树的主要基本操作：

- 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);

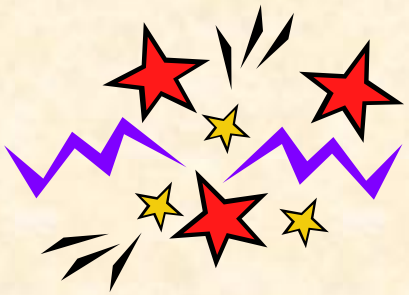
InsertChild(T, p, LR, c);

ClearBiTree(&T);

DestroyBiTree(&T);

DeleteChild(T, p, LR);

二叉树
的重要特性



性质 1：

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

用归纳法证明：

(1) $i = 1$，只有一个根结点，$2^{i-1} = 2^0 = 1$，正确；

(2) 假设 $i-1$ 成立，即第 $i-1$ 层上至多有 2^{i-2} 个结点；

(3) 由于二叉树的结点的度至多为 2，故在第 i 层上的最大结点数为第 $i-1$ 层上的最大结点数的 2 倍，即 $2 \times 2^{i-2} = 2^{i-1}$。

性质 2：

深度为 k 的二叉树上至多含 2^k-1 个结点（k≥1）

证明：

基于上一条性质，深度为 k 的二叉树上的结点数至多为

$$2^0+2^1+\cdots\cdots+2^{k-1}=2^k-1$$

引论： 一棵树有 n 个结点，则必有 n – 1 条分支。

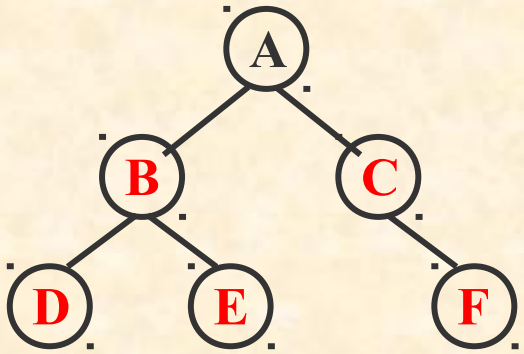
证明：

除根结点外，其它结点都有一个分支进入，

设 B 为分支总数，故 $n = B + 1$ ，

故 $B = n - 1$

得证。



性质 3： 对任何一棵二叉树 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) 已知，终端结点数为 n_0 ，度为 2 的结点数为 n_2

设度为 1 的结点数为 n_1 ，

由于二叉树中的所有结点的度只能为 0 、 1 、 2 ，

故二叉树的结点总数为 $n = n_0 + n_1 + n_2$ ；

(2) 除根结点外，其它结点都有一个分支进入，

设 B 为分支总数，故 $n = B + 1$ ，

由于这些分支均是由度为 1 或 2 的结点引出的，

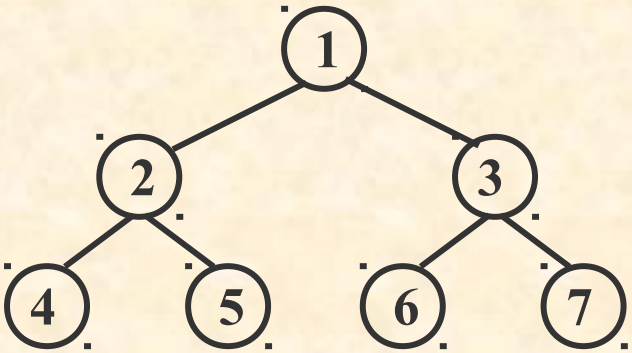
所以有 $B = n_1 + 2n_2$ ，故 $n = n_1 + 2n_2 + 1$ ，

由 (1) 和 (2) ，可得 $n_0 + n_1 + n_2 = n_1 + 2n_2 + 1$ ，

故有 $n_0 = n_2 + 1$ 。

两种特殊形态二叉树： 满二叉树、完全二叉树。

一棵深度为 k 且有 2^k-1 个结点的二叉树称为满二叉树。



满二叉树

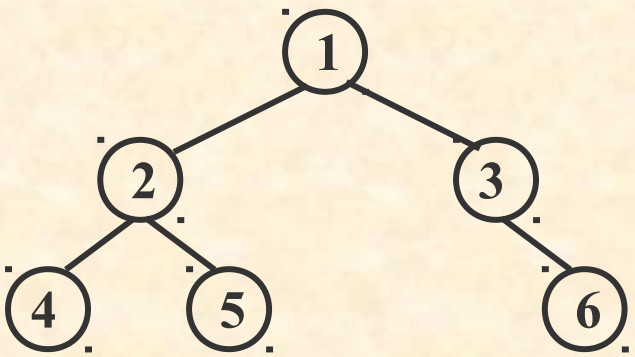
特点：(1) 每一层的结点数都达到最大结点数。

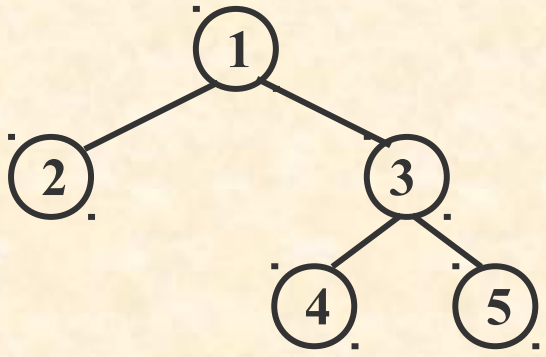
(2) 叶子结点在最大层。

(3) 任一结点，其左、右分支下的子孙的最大层次相等。

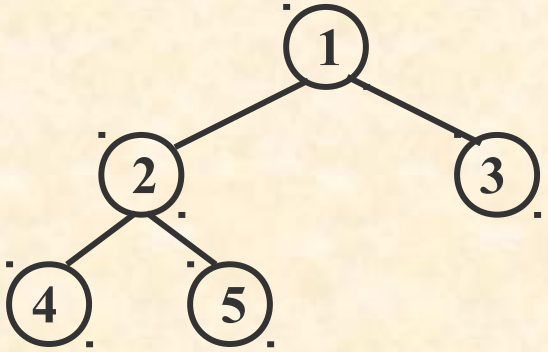
对满二叉树的结点进行连续编号，从根结点起，自上而下，自左至右，1、2、3、……、2^k-1。

深度为 k 的，有 n 个结点的二叉树，当且仅当其每一个结点都与深度为 k 的满二叉树中编号从 1 至 n 的结点一一对应时，称为完全二叉树。

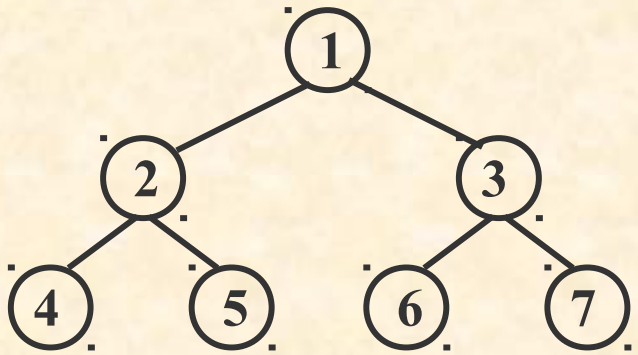




非完全二叉树



完全二叉树 (1)



完全二叉树 (2)

特点：

(1) 叶子结点只可能在层次最大的两层上出现。

(2) 对任一结点，若其右分支的子孙的最大层次为 l ，则其左分支下的子孙的最大层次必为 l 或 $l+1$ 。

若设二叉树的高度为 h ，则共有 h 层。除第 h 层外，其它各层 (0 ~ h-1) 的结点数都达到最大个数，第 h 层从右向左连续缺若干结点，这就是完全二叉树。

性质 4：具有 n 个结点的完全二叉树的深度为 $\lfloor \log_2 n \rfloor + 1$。

证明：设 n 结点完全二叉树的深度为 k ，

因为，

k−1 层满二叉树结点数	<	k 层完全二叉树结点数	≤	k 层满二叉树结点数

故 $2^{k-1}-1 < n \le 2^k-1$

有 $2^{k-1} \le n < 2^k$

又 $k-1 \le \log_2 n < k$

因为 k 是整数，所以 $k = \lfloor \log_2 n \rfloor + 1$ 。

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。

6.3

二叉树的存储结构

一、 二叉树的顺序存储表示

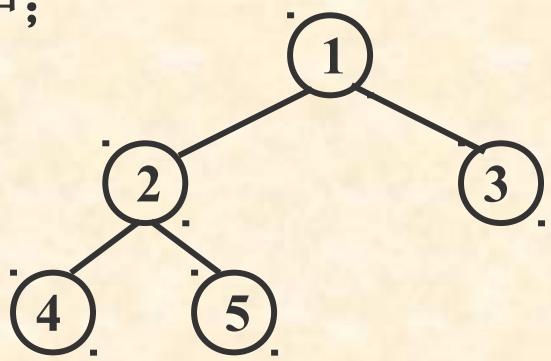
二、二叉树的链式存储表示

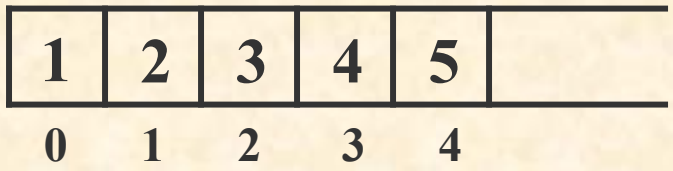
1. 顺序存储结构

用一组**地址连续**的存储单元依次*自上而下、自左至右*存储二叉树上的结点元素。

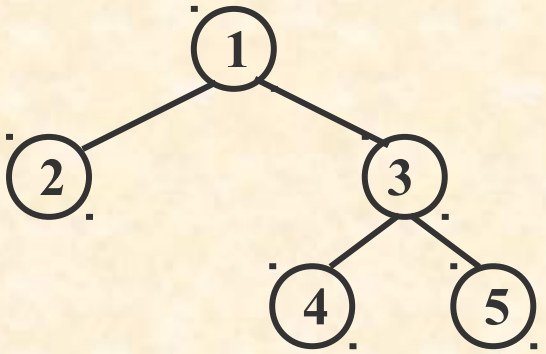
```
# define   MAX_TREE_SIZE    100
typedef   TElemType   SqBiTree[MAX_TREE_SIZE]
```

完全二叉树： 编号为 **i** 的元素存储在数组下标为 **i−1** 的分量中；





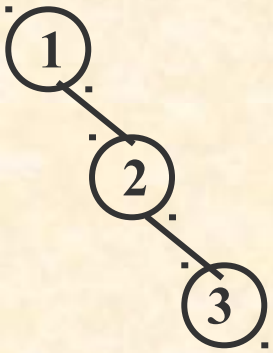
一般二叉树： 对照完全二叉树，存储在数组的相应分量中；



1	2	3	0	0	4	5	
0	1	2	3	4	5	6	

0 表示不存在此结点

在最坏情况下，深度为 k 的右单支二叉树需要 2^k-1 个存储空间。



1	0	2	0	0	0	3	
0	1	2	3	4	5	6	

空间浪费

结论： 顺序存储结构适用于完全二叉树。

二、二叉树的链式存储表示

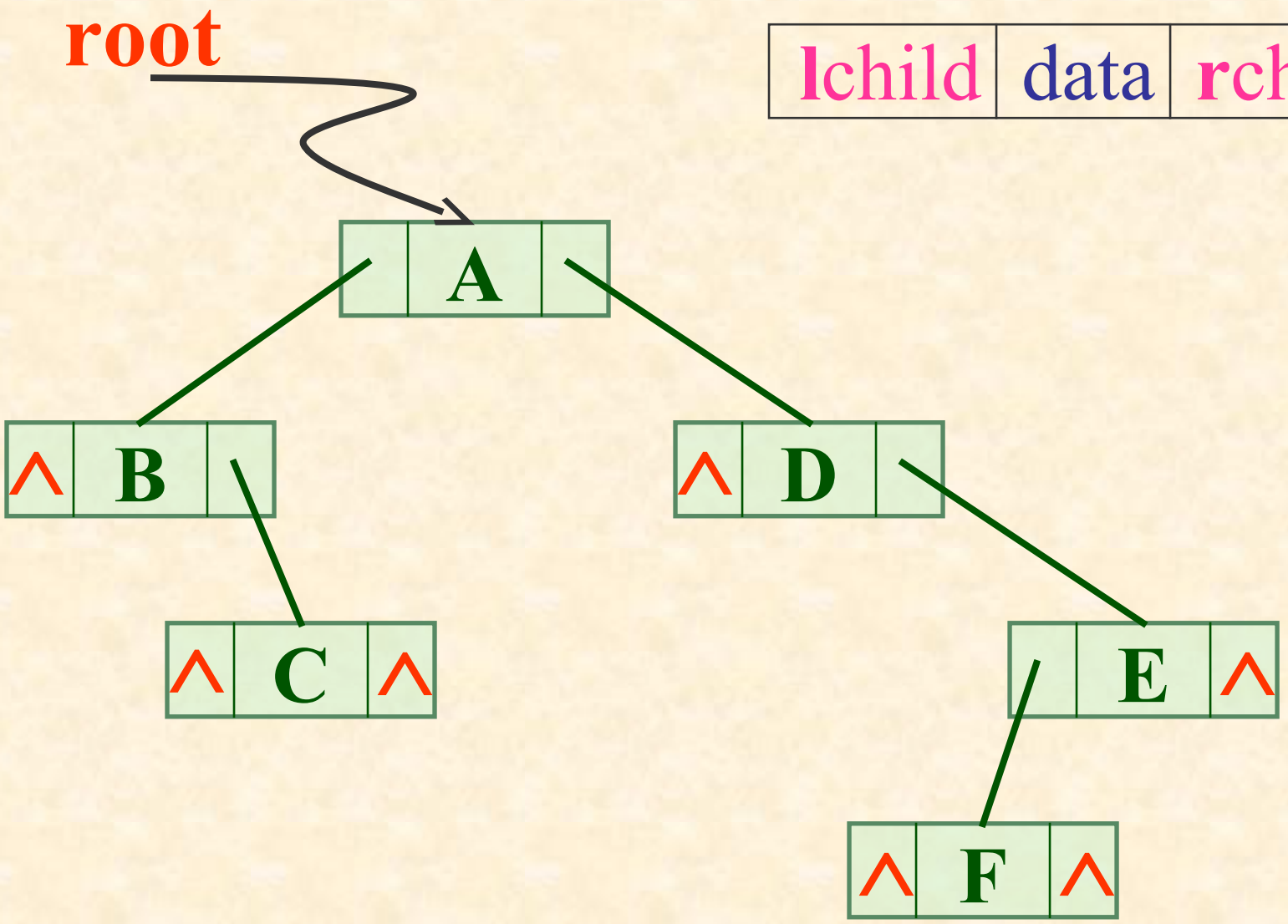
1. 二叉链表

2. 三叉链表

1. 二叉链表

结点结构：

lchild	data	rchild



C 语言的类型描述如下：

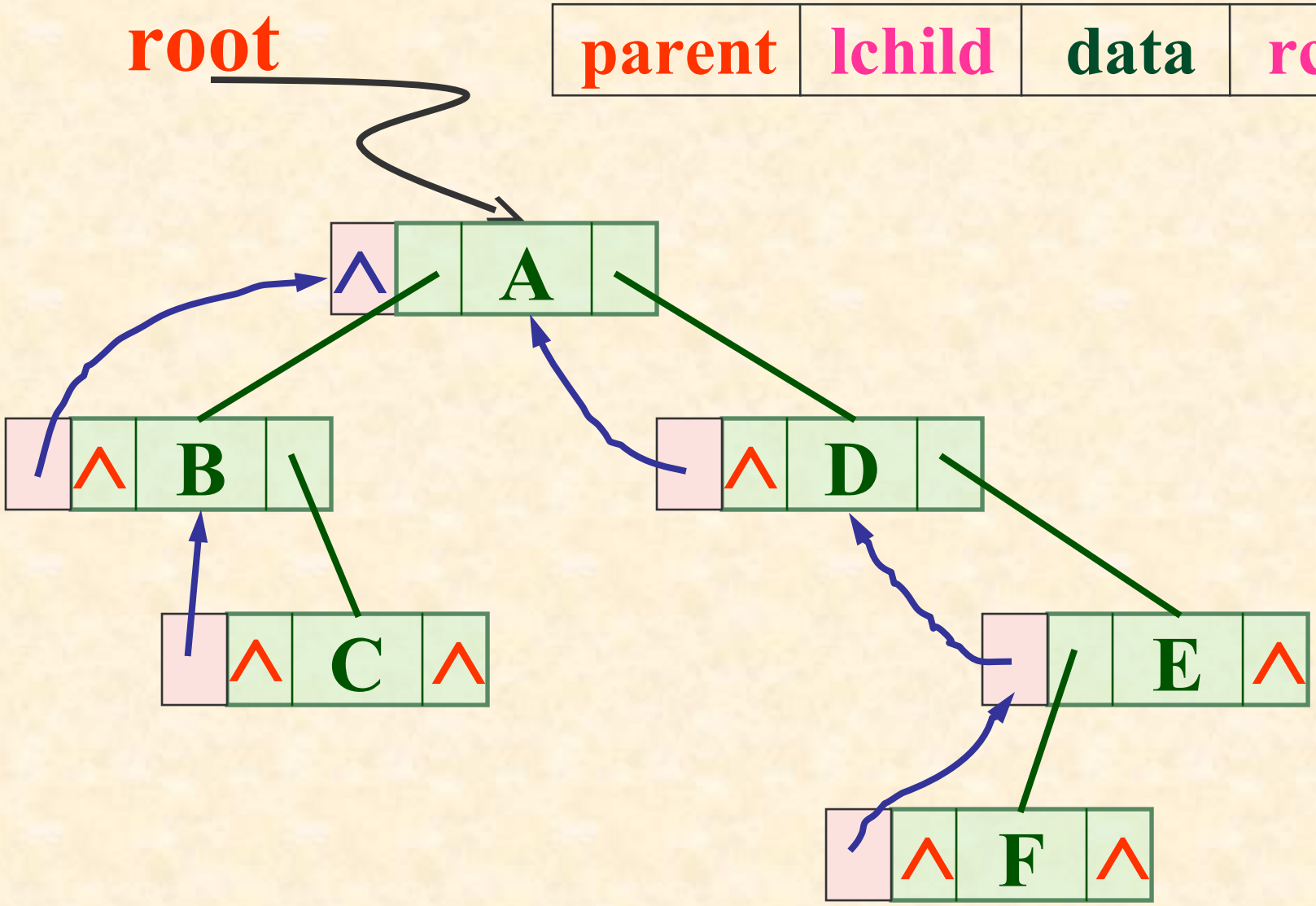
```
typedef struct BiTNode { //  结点结构
    TElemType          data;
    struct BiTNode  *lchild, *rchild;
                                  //  左右孩子指针
} BiTNode, *BiTree;
```

结点结构：

2．三叉链表

结点结构：

parent	lchild	data	rchild



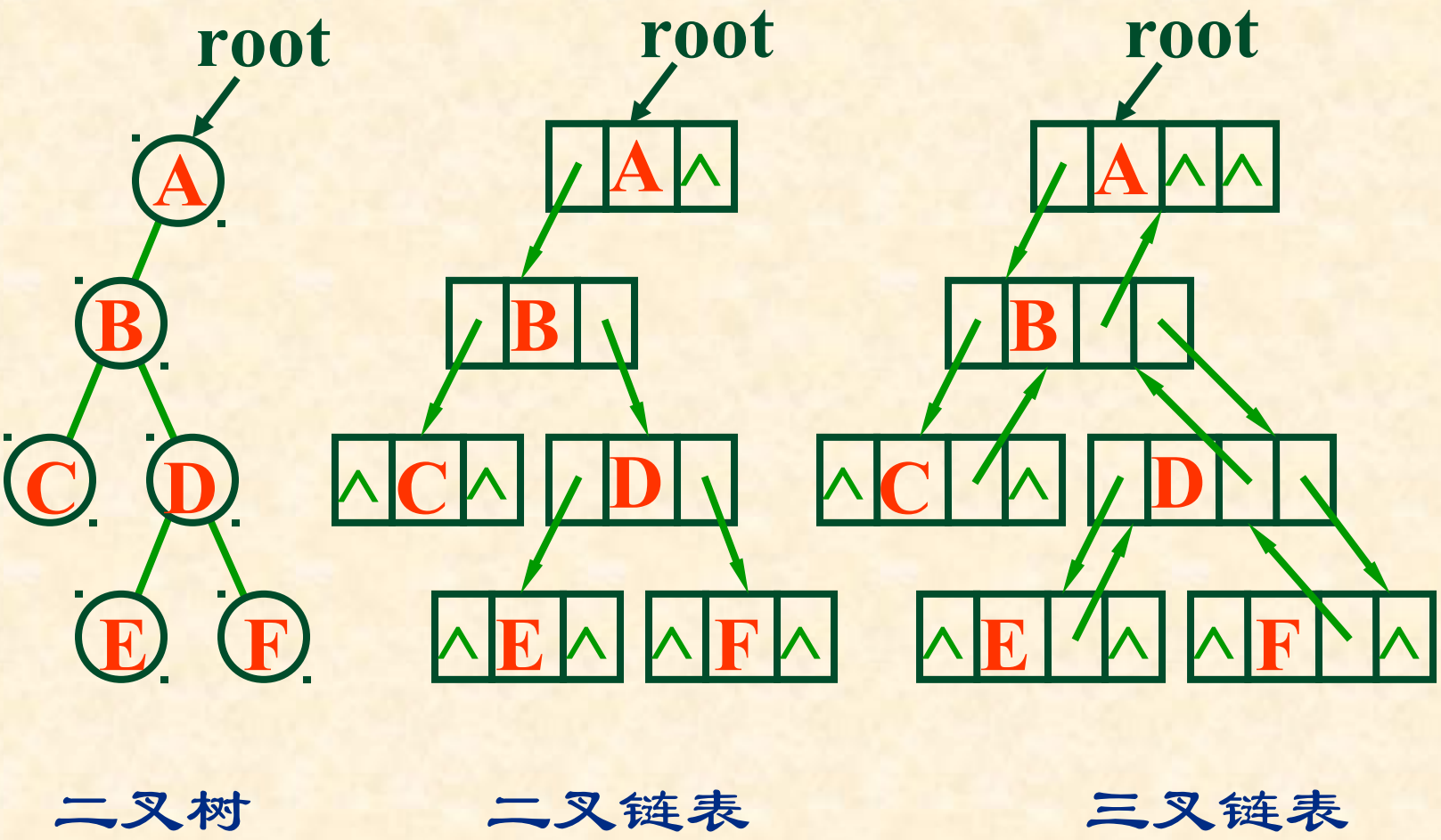
C 语言的类型描述如下：

```
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;
```

结点结构：

parent	lchild	data	rchild

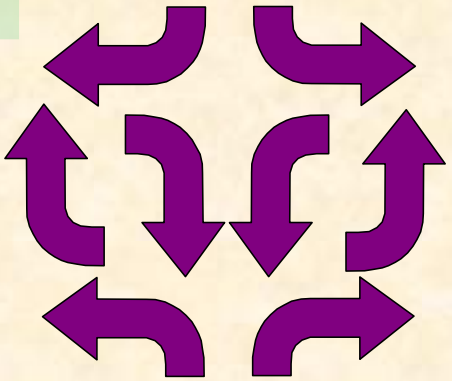
二叉树链表表示的示例



二叉树　　二叉链表　　三叉链表

6.4

二叉树的遍历



一、问题的提出

顺着某一条搜索路径**巡访**二叉树中的结点，使得每个结点**均被访问一次**，而且**仅被访问一次**。

“**访问**”的含义可以很广，如：输出结点的信息等。

“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），

故不需要另加讨论。而二叉树是非

线性结构，每个结点有两个后

在如何遍历即按什么样的搜索

进行遍历的问题。

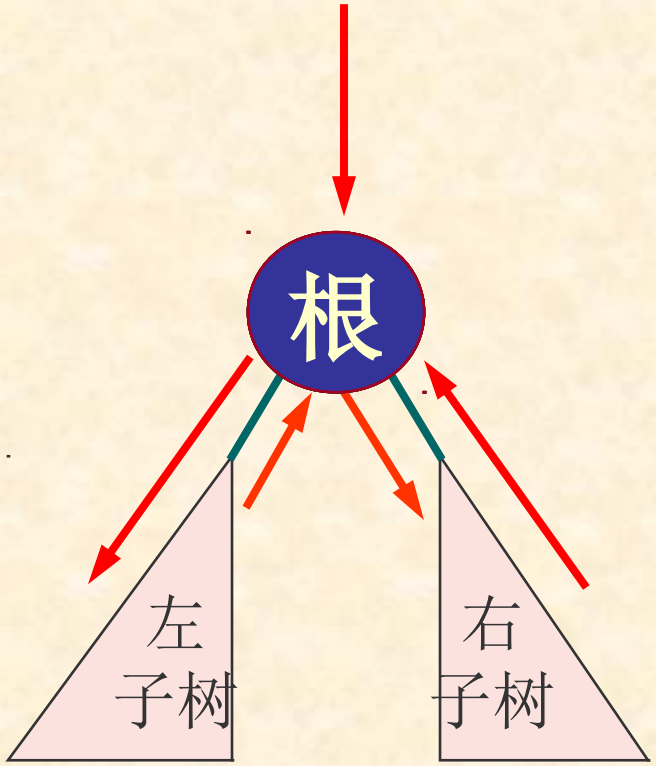
$D = (root, D_L, D_R)$。

如果能依次遍历这三部分，就可以遍历整个二叉树；

设以 D、L、R 分别表示访问根结点、遍历左子树、遍历右子树，则可以存在 6 种遍历方案：DLR、DRL、LDR、LRD、RDL、RLD；

若限定先左后右的原则，则只有 3 种情况：先(根)序遍历、中(根)序遍历、后(根)序遍历。

二、先左后右的遍历算法



先（根）序的遍历算法

中（根）序的遍历算法

后（根）序的遍历算法

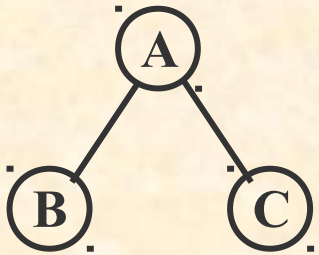
先（根）序的遍历算法：

若二叉树为空树，则空操作；否则，

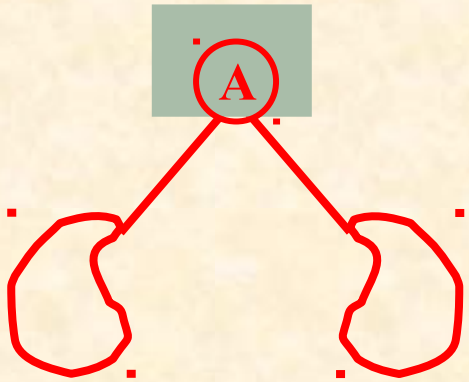
（1）访问根结点；

（2）先序遍历左子树；

（3）先序遍历右子树。



A B C



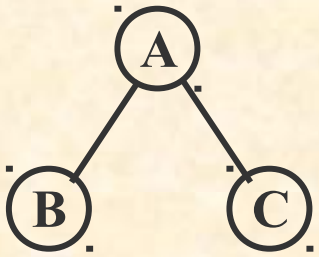
中（根）序的遍历算法：

若二叉树为空树，则空操作；否则，

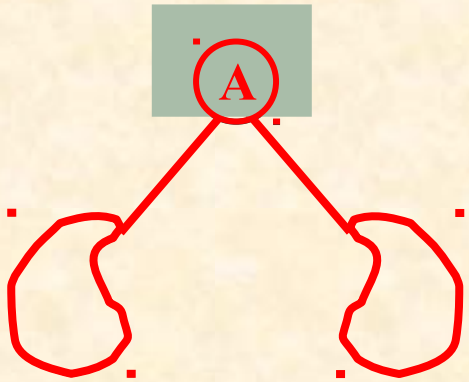
（1）中序遍历左子树；

（2）访问根结点；

（3）中序遍历右子树。



B A C



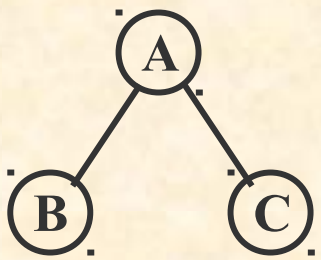
后（根）序的遍历算法：

若二叉树为空树，则空操作；否则，

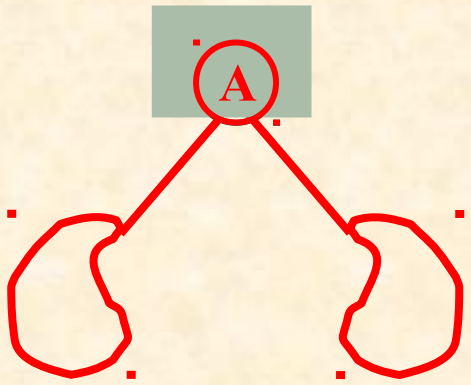
（1）后序遍历左子树；

（2）后序遍历右子树；

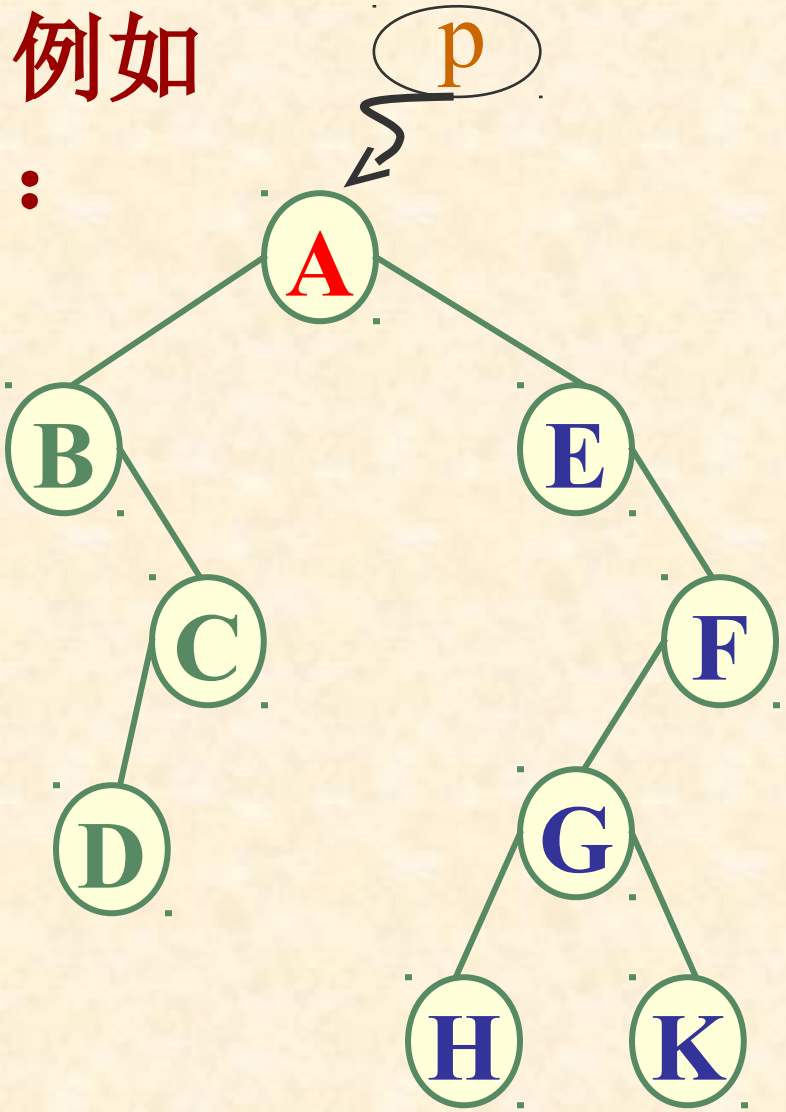
（3）访问根结点。



B C A



例如：



先序序列：

A B C D E F G H K

中序序列：

B D C A E H G K F

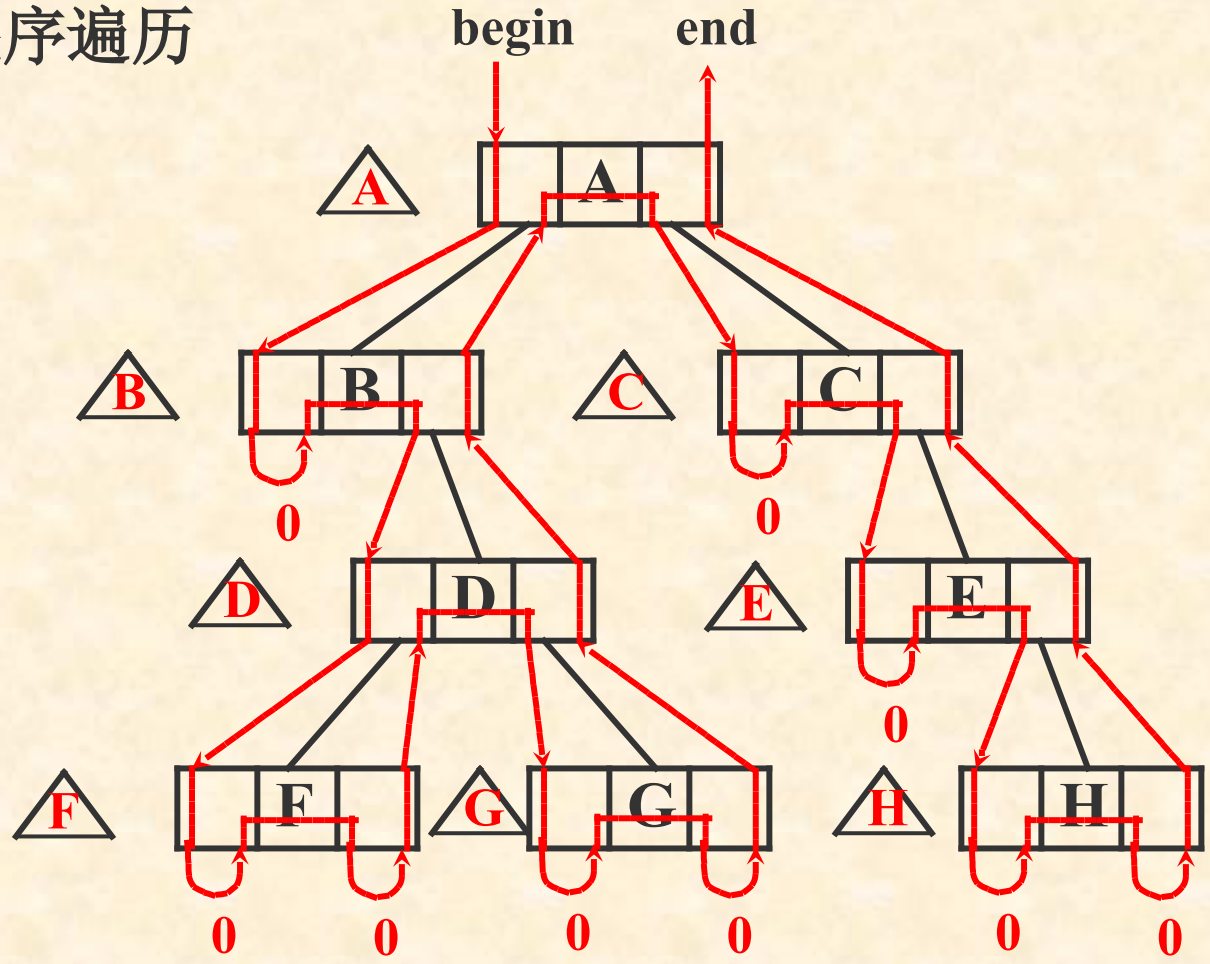
后序序列：

D C B H K G F E A

算法 6.1 先序遍历递归算法

```
Status PreOrderTraverse ( BiTree T , printf() )
 {
                                  // visit(e)  函数可以看作   printf (e)
        if  (T)
        {
            printf(T->data) ;
            PreOrderTraverse ( T->lchild, printf ) ;
            PreOrderTraverse ( T->rchild, printf ) ;
            return OK;
        }
       else   return OK ;
}
```

先序遍历

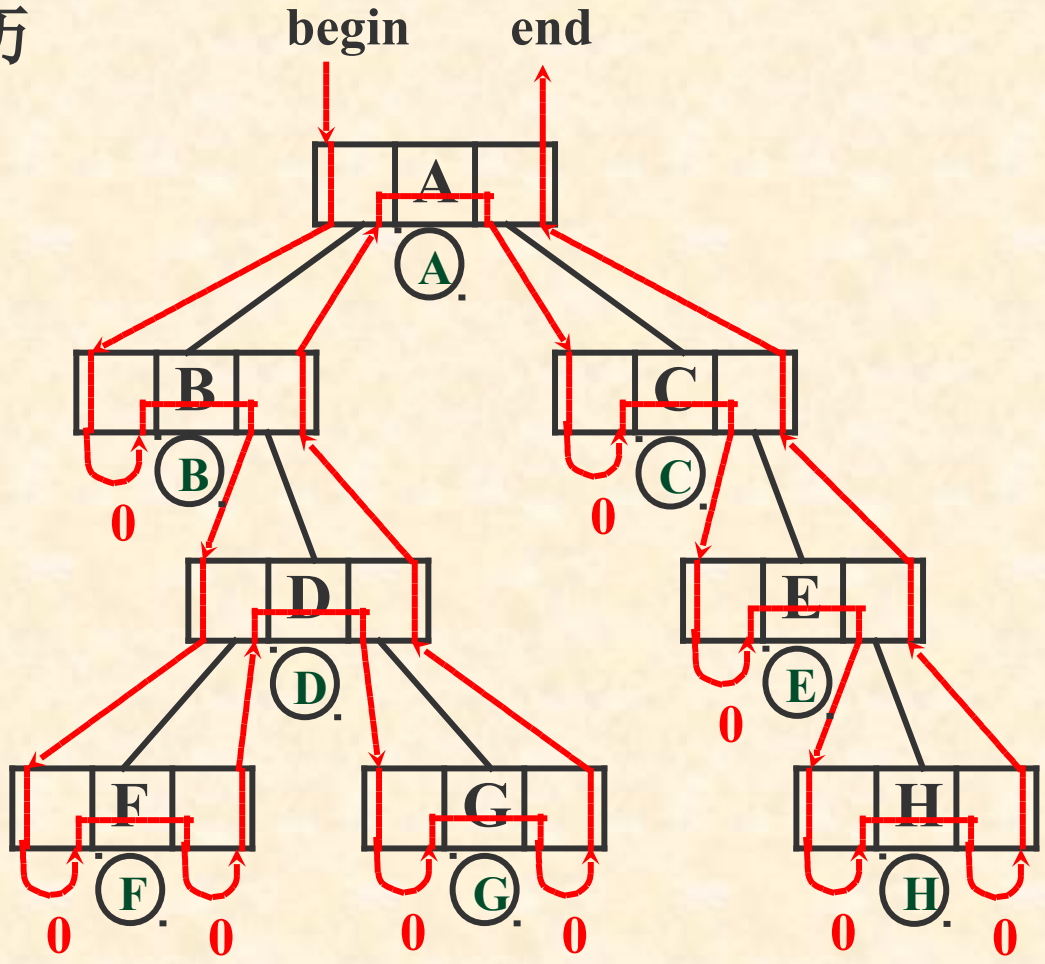


先序遍历顺序： A B D F G C E H

算法 6.2 中序遍历递归算法

```
Status InOrderTraverse ( BiTree T , printf() ) {
                                    // visit(e) 函数可以看作 printf (e)
    if (T) {
        InOrderTraverse ( T->lchild, printf ) ;
        printf(T->data) ;
        InOrderTraverse ( T->rchild, printf ) ;
    }
    else return OK ;
}
```

中序遍历

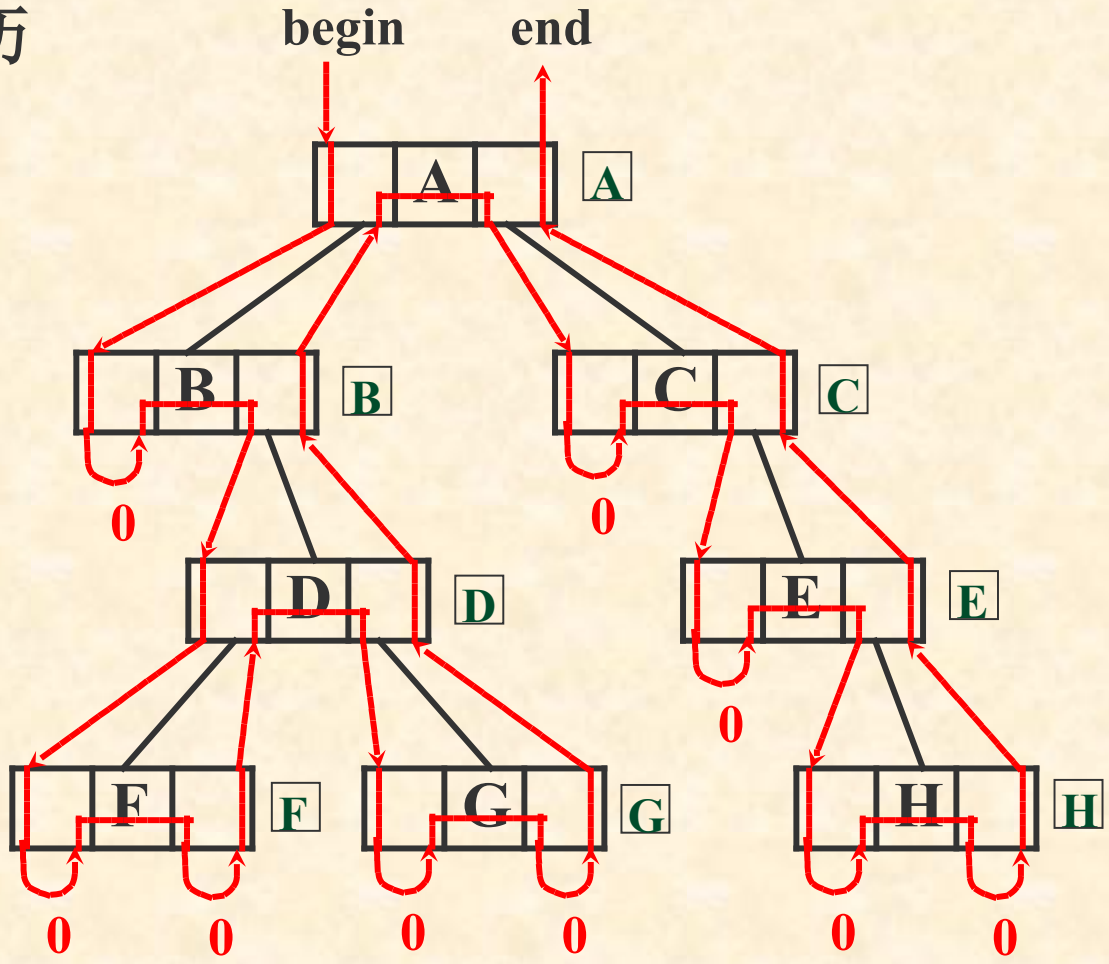


中序遍历顺序： B F D G A C E H

算法 6.3 后序遍历递归算法

```
Status PostOrderTraverse ( BiTree T , printf() ) {
                                    // visit(e) 函数可以看作 printf (e)
    if (T) {
        PostOrderTraverse ( T->lchild, printf ) ;
        PostOrderTraverse ( T->rchild, printf ) ;
        printf(T->data) ;
    }
    else return OK ;
}
```

后序遍历



后序遍历顺序：F G D B H E C A

遍历二叉树的非递归算法

- **先序遍历**：算法 1，将右子树根结点入栈，算法 2 将根结点入栈
- **中序遍历**：在遍历左子树之前，先把根结点入栈，当左子树遍历结束后，从栈中弹出，访问，再遍历右子树
- **后序遍历**：

  1）设定一个指针，指向最近访问过的结点。在退栈取出根结点时，需判断：若根结点的右子树为空，或它的右子树非空，但已遍历完毕，即它的右子树根结点恰好是最近一次访问过的结点时，应该遍历该根结点。反之，该根结点应重新入栈，先遍历它的右子树。

  2）还可同时设定一个标记，指示该根结点是第一次还是第二次入栈

先序遍历的非递归算法

```
void preorder(BiTree T)
  {     SqStack S;          BiTree P=T;
        InitStack(S);       Push(S,NULL);
        while (P)
          {     printf("%c",P->data);
                if (P->rchild)
                        Push(S,P->rchild);
                if (P->lchild)
                        P=P->lchild;
                else Pop(S,P);
        }
 }
```

先序遍历的非递归算法 2

```
void preorder(BiTree T){
        int top=0;
        BiTree stack[20], P=T;
        do {  while (P){
                        printf("%c",P->data);
                        stack[top]=P; top++;
                        P=P->lchild;     }
                if (top){
                        top--;        P=stack[top];
                        P=P->rchild;}
        }while (top || P);
}
```

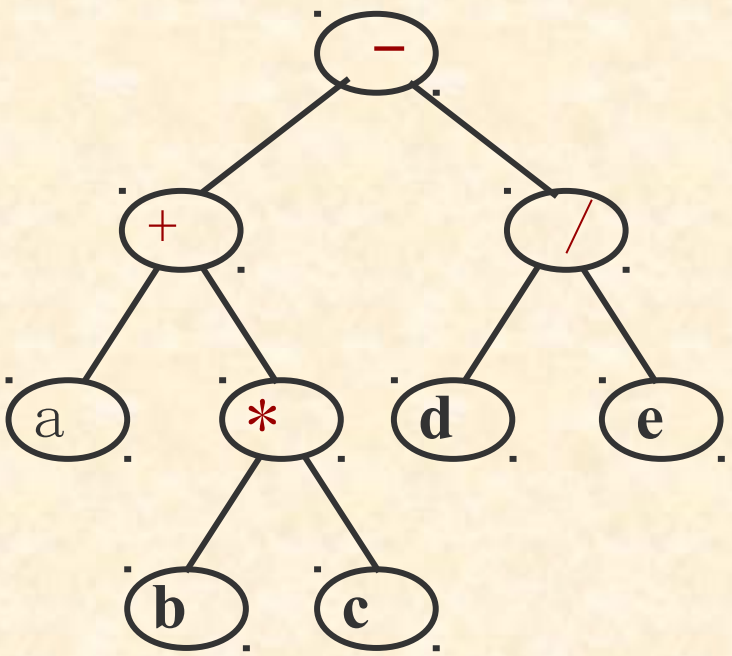
中序遍历的非递归算法 1 （ p131 算法 6.3)

```
void inorder(BiTree T)
 {
      SqStack S;        BiTree P=T;
      InitStack(S);
      while( P || ! StackEmpty(S)   )
       { if (P) {  Push(S,P);
                   P=P->lchild; }
         else    {  Pop(S,P);
                    printf("%c",P->data);
                    P=P->rchild;}
       }
 }
```

中序遍历的非递归算法 2

```
void inorder(BiTree T){
        SqStack S;          BiTree P=T;
        InitStack(S);
        do{   while(P){
                        Push( S, P);
                        P=P->lchild;     }
              if (!StackEmpty(S)){
                        Pop(S,P);
                        printf("%c",P->data);
                        P=P->rchild;}
        }while((S.top!=S.base) || P);
}
```

例，表达式 a + b * c – d / e



先序遍历：

– + a * b c / d e

前缀式，波兰式

中序遍历：

a + b * c – d / e

中缀式，算术表达式

后序遍历：

a b c * + d e / –

后缀式，逆波兰式

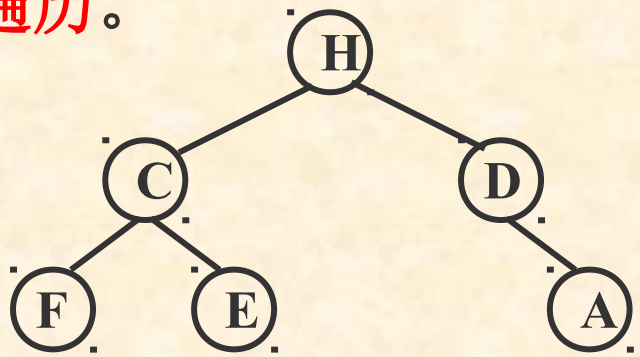
中缀表示： 适于人的思维

后缀表示： 适于计算机的思维

a + b * c – d / e

a b c * + d e / −

对二叉树除可以进行先序、中序、后序的遍历外，还可以进行层次遍历。



层次遍历：H，C，D，F，E，A

过程：打印 H；

栈实现？

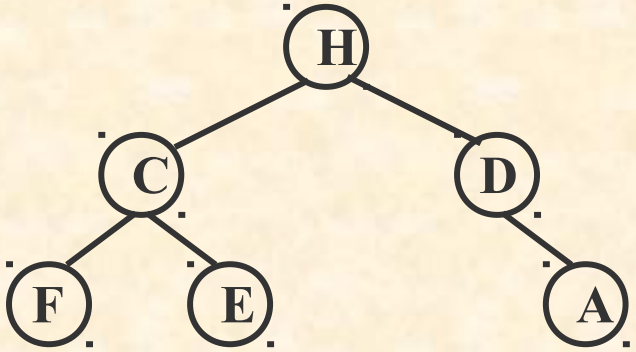
打印 H 的左儿子 C；

打印 H 的右儿子 D；

打印 C 的左、右儿子 F、E；

打印 D 的左、右儿子 A；

队列实现层次遍历



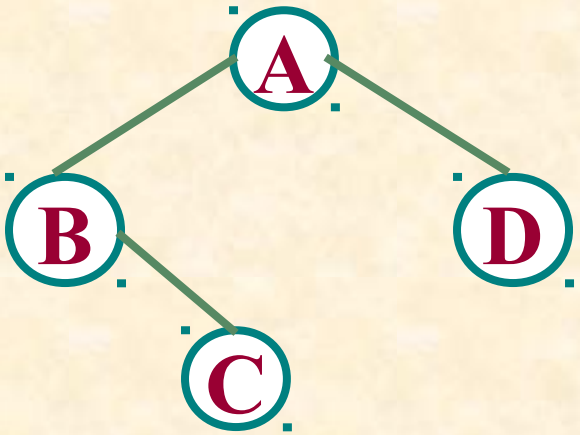
H C D F E A

- 以字符串的形式 “根 左子树 右子树”定义一棵二叉树

例如：

空树　　以空白字符“■”表示

只含一个根结点的二叉树　以字符串“A■■”表示



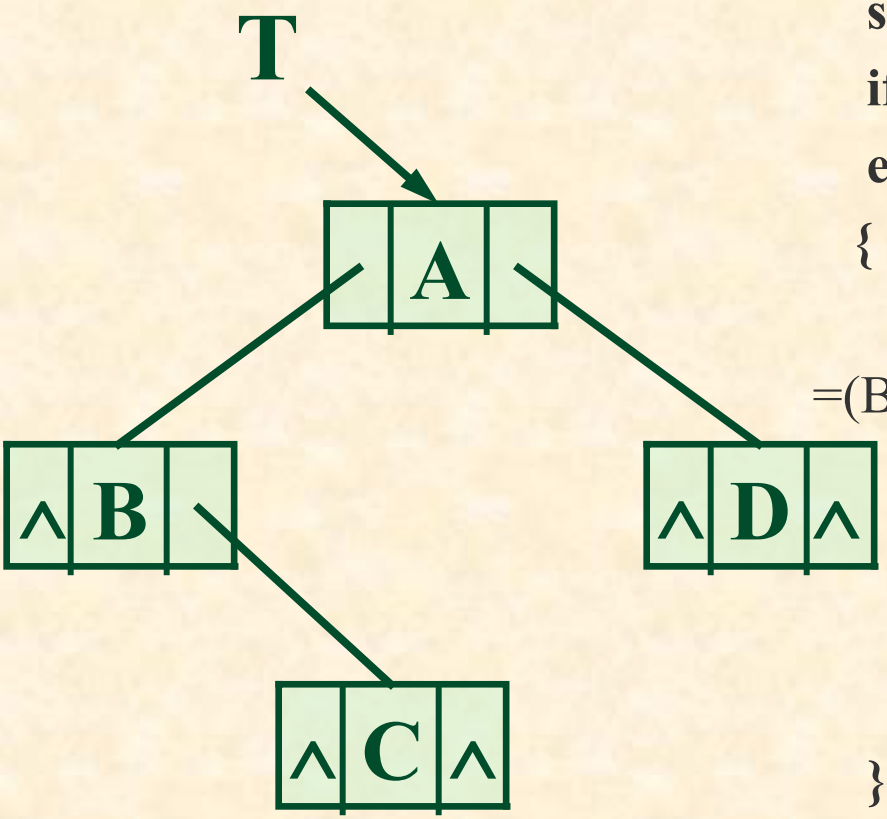
以下列字符串表示

A(B(■,C(■,■)),D(■,■))

```
Status CreateBiTree(BiTree &T) {
   scanf(&ch);
   if (ch==' ') T = NULL;
   else
  {
    if (!(T =(BiTNode*)malloc(sizeof(BiTNde))))
       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;  }                           // CreateBiTree
```

上页算法执行过程举例如下：

A B ■ C ■ ■ D ■ ■




```
Status CreateBiTree(BiTree &T) {
    scanf(&ch);
    if (ch==' ') 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;  } // CreateBiTree
```

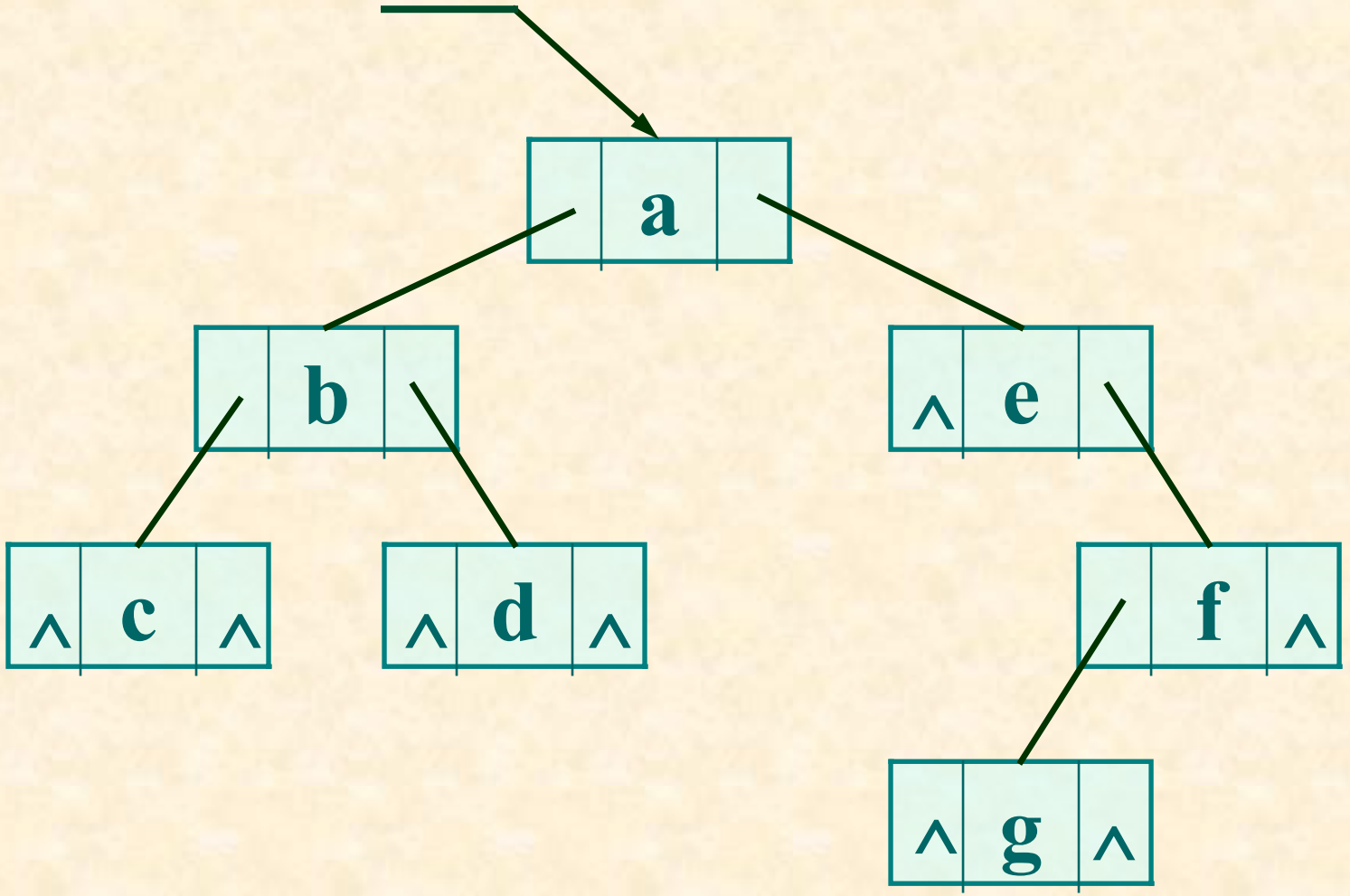
由二叉树的先序和中序序列建树

仅知二叉树的先序序列“**abcdefg**”不能唯一确定一棵二叉树，如果同时已知二叉树的中序序列“**cbdaegf**”，则会如何？

例如：

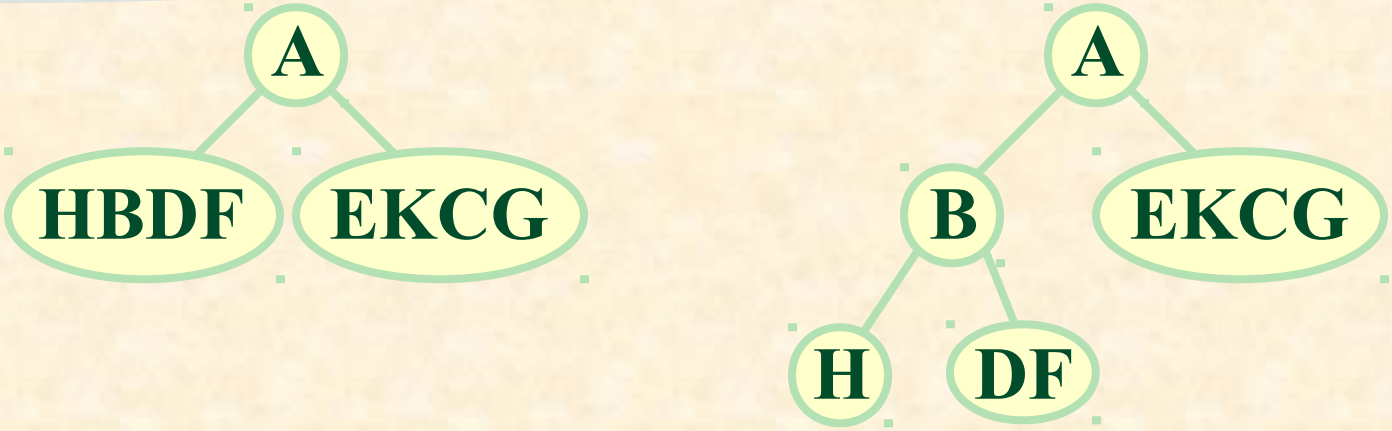
a b c d e f g　先序序列

c b d a e g f　中序序列



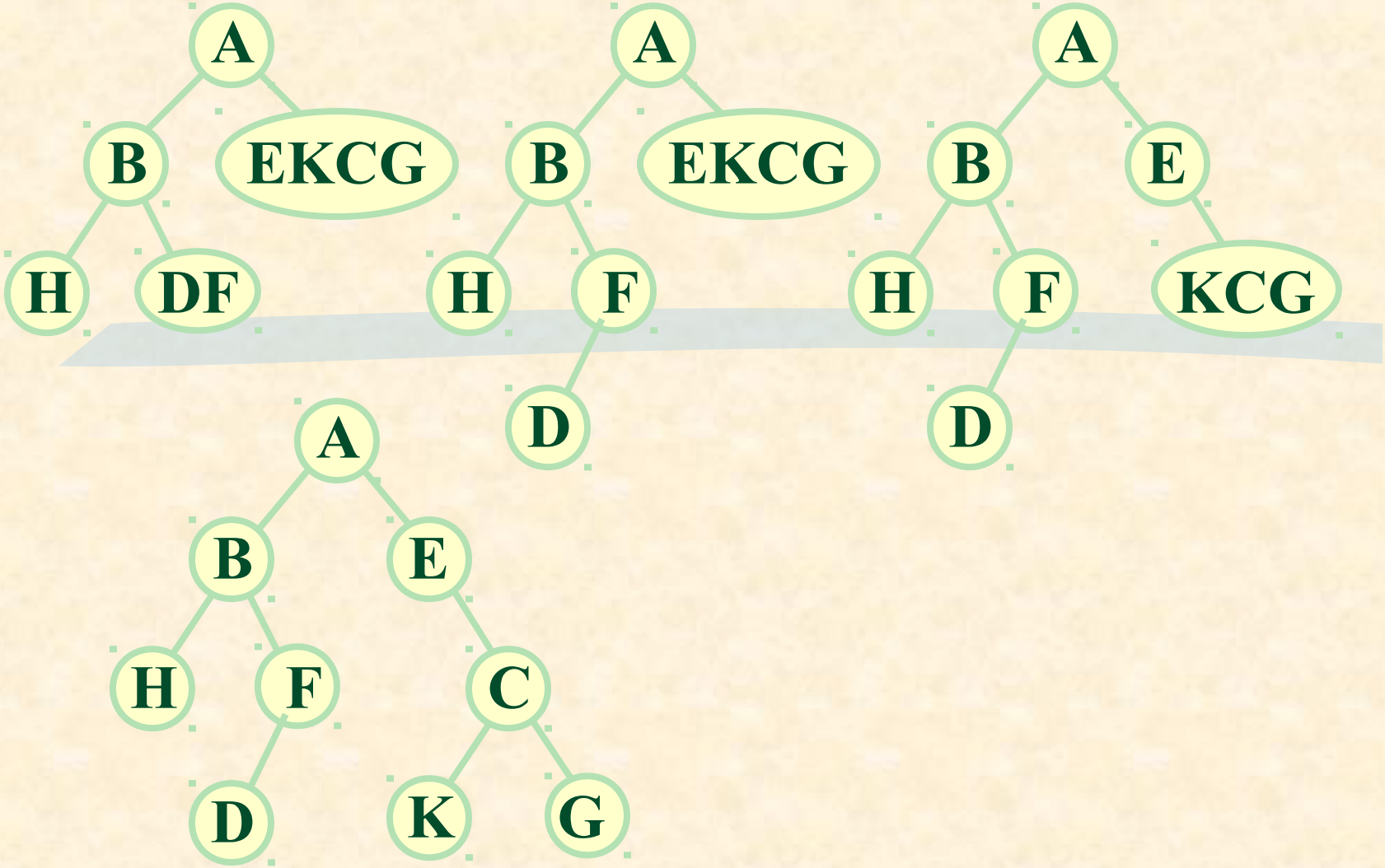
二叉树的建树

前序序列 { ABHFDECKG } 和中序序列 { HBDFAEKCG }, 构造二叉树过程如下：



前序序列 { ABHFDECKG }

中序序列 { HBDFAEKCG }



二叉树遍历应用

1. 计算二叉树结点个数 (递归算法)

```
int Count ( BinTreeNode *T )
{
    if ( T == NULL )
        return 0;
    else
        return ( 1 + Count ( T->leftChild )
                   + Count ( T->rightChild ) );
}
```

2. 求二叉树中叶子结点的个数

```
int Leaf_Count(Bitree T)  // 求二叉树中叶子结点的数目
{
  if(!T)    return 0;                    // 空树没有叶子
  else if(!T->lchild&&!T->rchild)   return 1;
  else return
       Leaf_Count(T-> lchild) + Leaf_Count( T->
rchild );                 // 左子树的叶子数加上右子树的叶子数
}                          //Leaf_Count
```

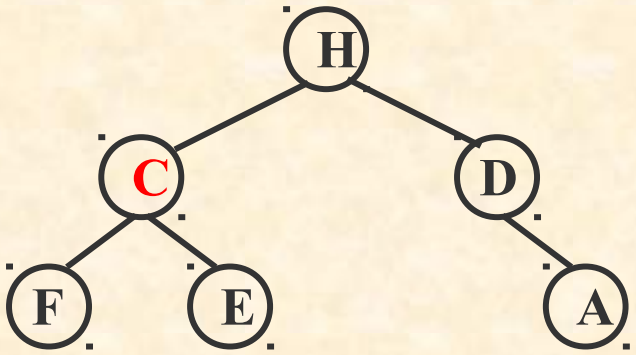
3. 求二叉树高度 (递归算法)

```
int Height ( BinTreeNode * T ) {
    if ( T == NULL ) return 0;
    else {
        int m = Height ( T->leftChild );
        int n = Height ( T->rightChild ) );
        return (m > n) ? m+1 : n+1;
        }
}
```

6.3.2　线索二叉树

二叉树的遍历实现了对一个非线性结构进行线性化的操作，从而使每个结点在线性序列中有且仅有一个直接前驱和直接后继。

但以二叉链表作为存储结构时，



如何直接找到任意结点的前驱和后继结点？

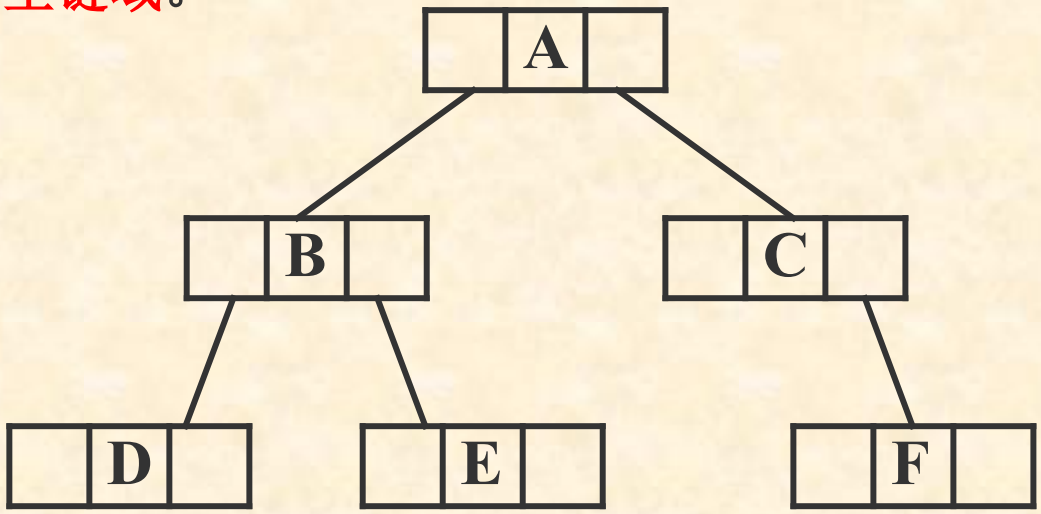
方法 1： 增加两个指针域 **fwd** 和 **bkwd**，分别指示其前驱和后继。

优点： 实现方便、简单。

缺点： 需要大量额外空间。

fwd	lchild	data	rchild	bkwd
前驱	左指针域	数据域	右指针域	后继

方法 2： 利用闲余的**空链域**。



性质： 含有 n 个结点的二叉链表中有 $n+1$ 个空链域。

证明： (1) 设，终端结点数为 n_0 ，

度为 1 的结点数为 n_1 ，

度为 2 的结点数为 n_2 ，

故二叉树的结点总数为 $n = n_0 + n_1 + n_2$ ；

(2) 空链域个数为 $2n_0 + n_1$ ，

已知， $n_0 = n_2 + 1$ ，

故， $2n_0 + n_1$

$= n_0 + n_1 + n_2 + 1$

$= n + 1$

如何利用空链域描述前驱和后继信息？

lchild	LTag	data	RTag	rchild

其中：

$$LTag = \begin{cases} 0 & \text{lchild 域指示结点的左儿子} \\ 1 & \text{lchild 域指示结点的前驱结点} \end{cases}$$

$$RTag = \begin{cases} 0 & \text{rchild 域指示结点的右儿子} \\ 1 & \text{rchild 域指示结点的后继结点} \end{cases}$$

增加线索的二叉树称之为线索二叉树。

对二叉树按照某种次序使其成为线索二叉树的过程叫做线索化

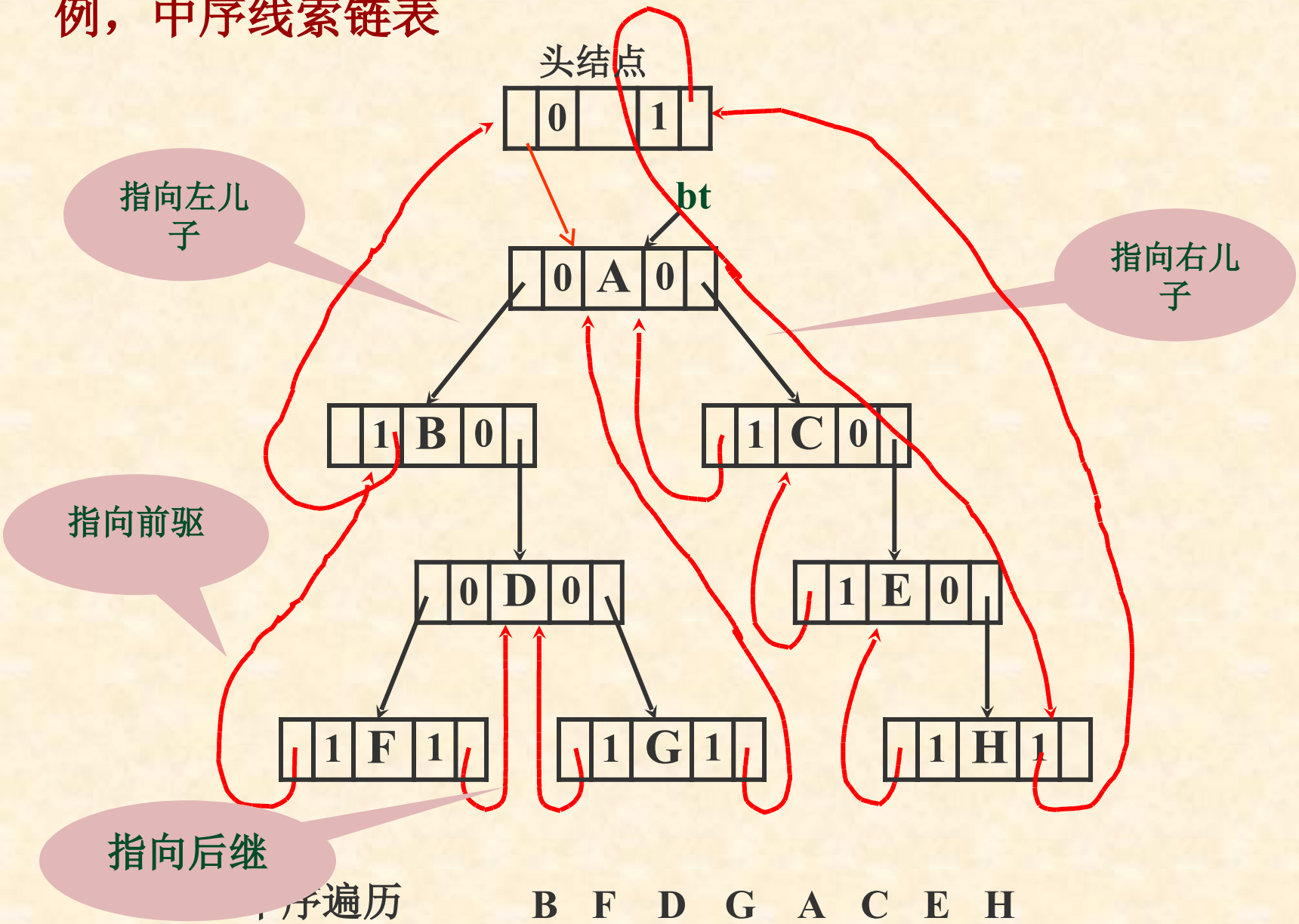
二叉线索树的存储表示

```
typedef   enum  PionterTag  ( Link ,  Thread ) ;      // 0 ,  1

typedef   struct   BiThrNode  {

    TElemType              data  ;

    struct   BiThrNode     * lchild  ,  * rchild  ;

    PointerTag             LTag  ,  RTag  ;

}
```

线索二叉链表的建立是依据二叉树的遍历顺序的。

说明：在线索树中的前驱和后继是指按某种次序遍历所得到的序列中的前驱和后继。

例，中序线索链表



序遍历　　B F D G A C E H

```
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== 0)  p = p->lchild;  // 第一个结点
      if( !visit(p->data) ) return ERROR
       while (p->RTag== 1 && p->rchild!=T)
       {
          p = p->rchild;  Visit(p->data);  // 访问后继结点
       }
       p = p->rchild;                       // p 进至其右子树根
    }
}                                           // InOrderTraverse_Thr
```

三、如何建立线索链表?

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

```
void InThreading(BiThrTree p) {
  if (p) {    // 对以 p 为根的非空二叉树进行线索化
    InThreading(p->lchild);      // 左子树线索化
    if (!p->lchild)                    // 建前驱线索
      { p->LTag = 1;    p->lchild = pre; }
    if (!pre->rchild)                  // 建后继线索
      { pre->RTag = 1;   pre->rchild = p; }
    pre = p;                  // 保持 pre 指向 p 的前驱
    InThreading(p->rchild);      // 右子树线索化
```

```
Status InOrderThreading(BiThrTree &Thrt,
                        BiThrTree T) { //  构建中序线索链表
if (!(Thrt =(BiThrNode*)malloc(sizeof(BiThrNode))))
    exit (OVERFLOW);
   Thrt->LTag = Link;  Thrt->RTag =Thread;
   Thrt->rchild = Thrt;      //  添加头结点
         …   …
   return OK;
}                                    // InOrderThreading
```

```
if (!T)  Thrt->lchild = Thrt;
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;
}
```

6.4 树和森林的表示方法

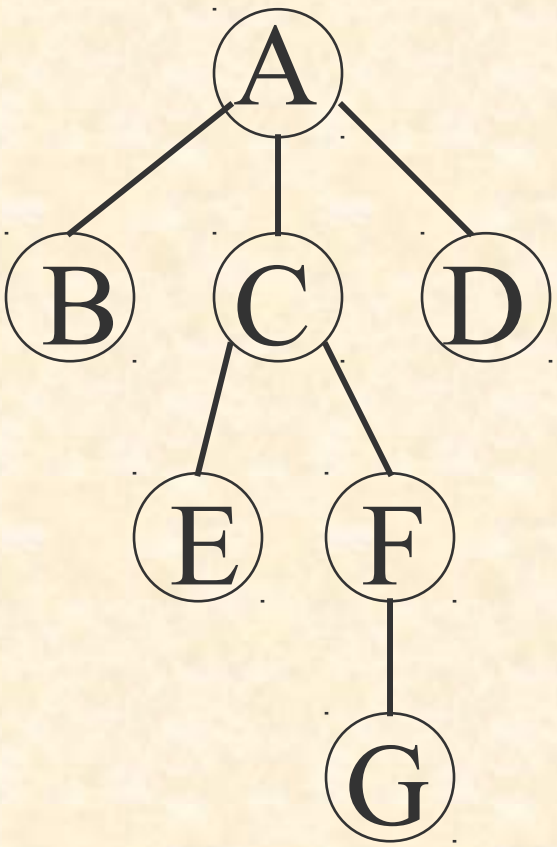
树的三种存储结构

一、双亲表示法

二、孩子链表表示法

三、树的二叉链表（孩子－兄弟）存储表示法

一、双亲表示法：



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

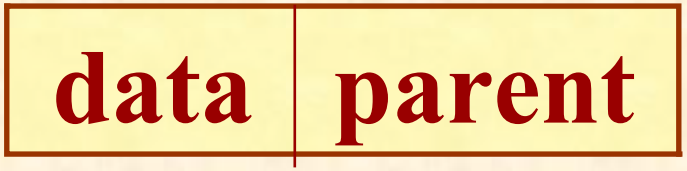
r=0

n=7

C 语言的类型描述：

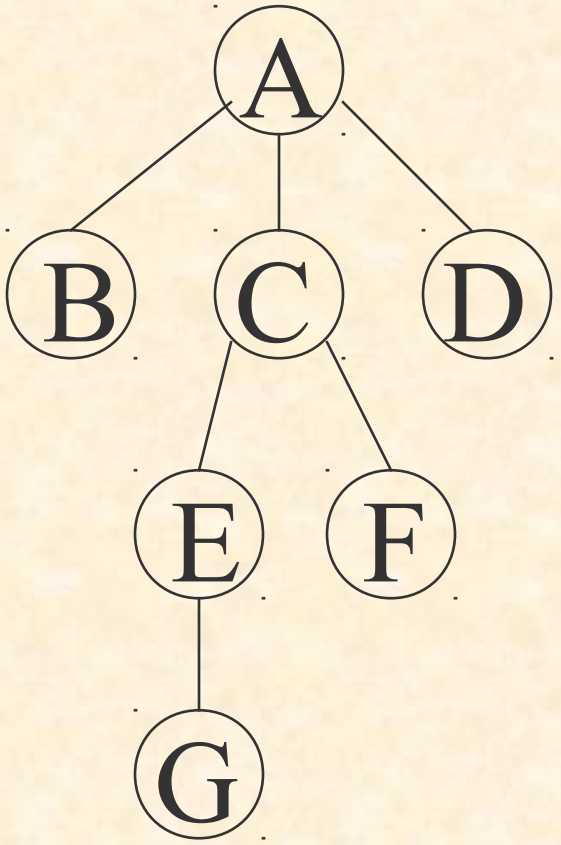
```
#define MAX_TREE_SIZE  100
```

结点结构：




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

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

二、孩子链表表示法：



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

r=0

n=7

C 语言的类型描述：

孩子结点结构：

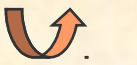

```
typedef struct CTNode {
    int         child;
    struct CTNode *nextchild;
} *ChildPtr;
```

双亲结点结构

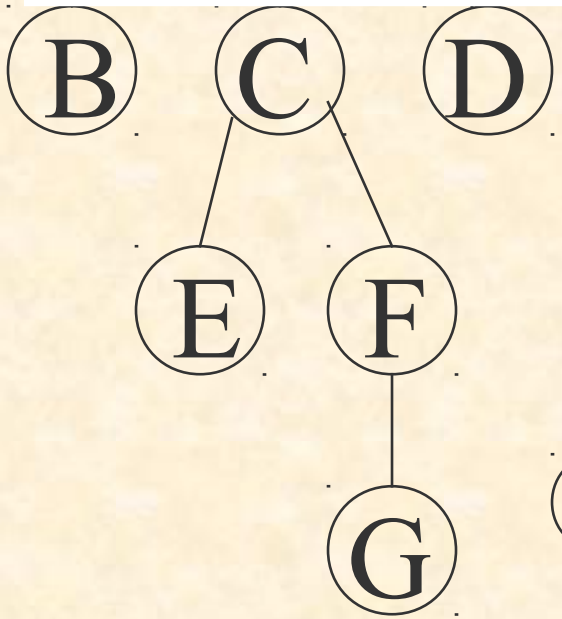

```
typedef struct {
  Elem    data;
  ChildPtr  firstchild;
                // 孩子链的头指针
} CTBox;
```

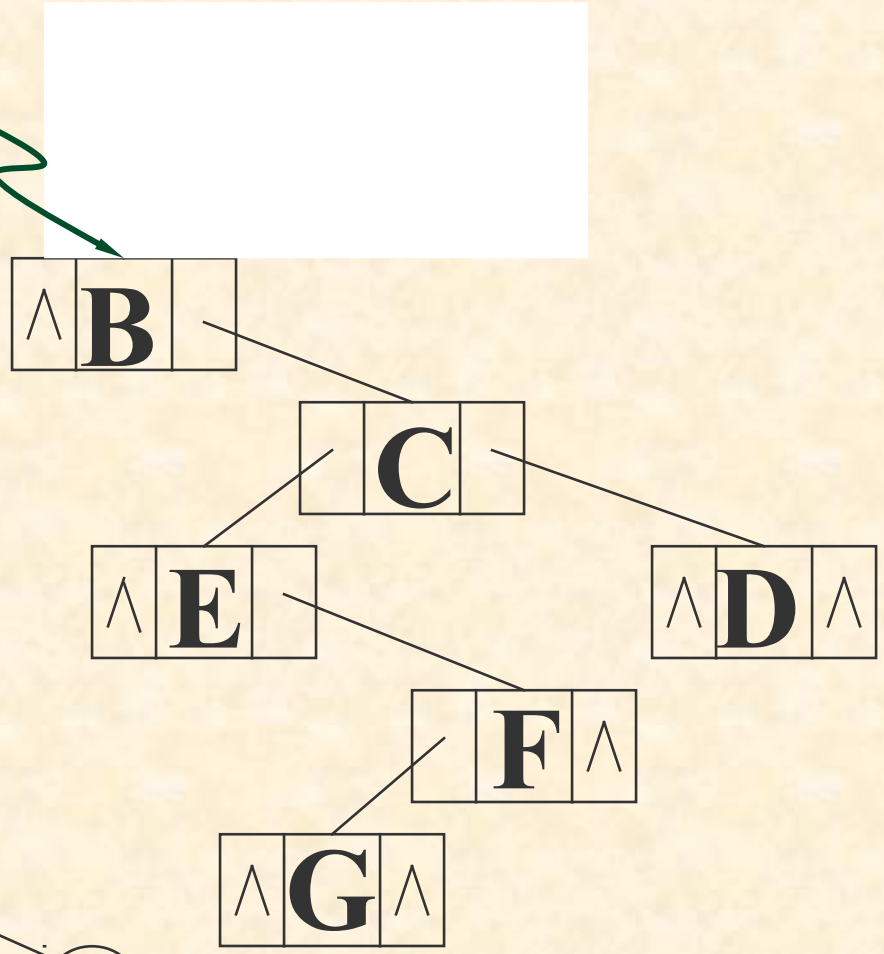
树结构：

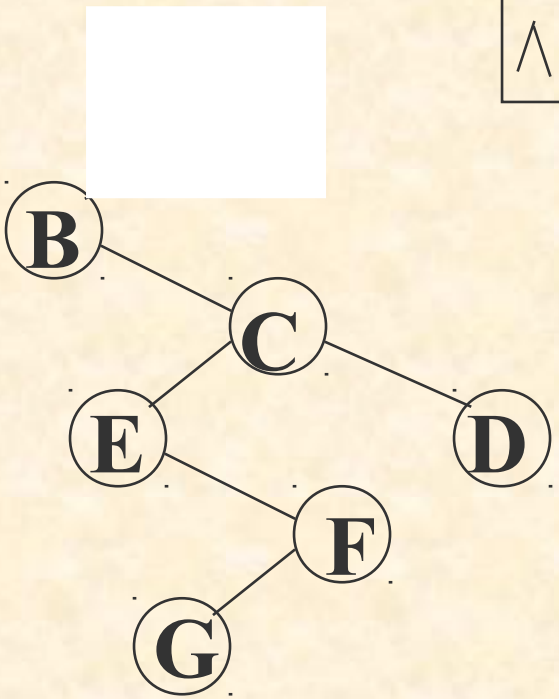
```
typedef struct {
    CTBox  nodes[MAX_TREE_SIZE];
    int    n, r;
           // 结点数和根结点的位置
} CTree;
```



三、树的二叉链表（孩子-兄弟）存储表示



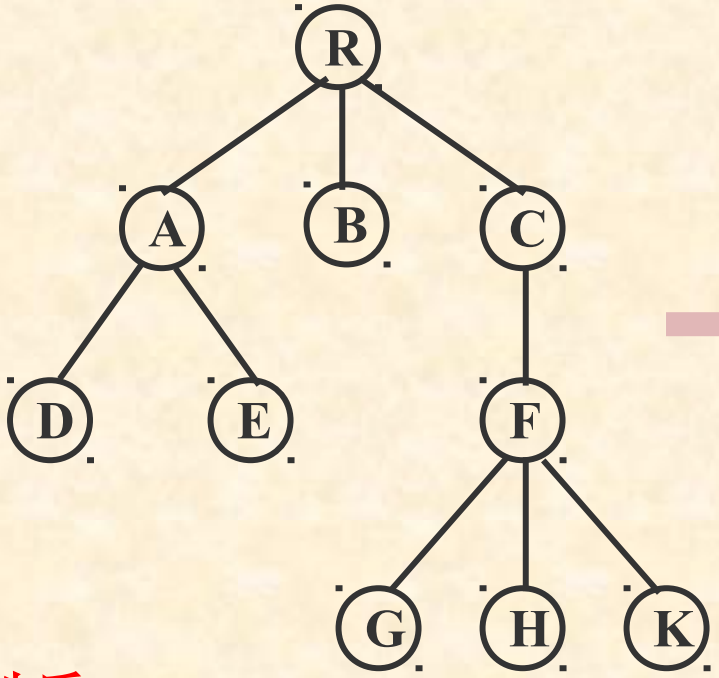


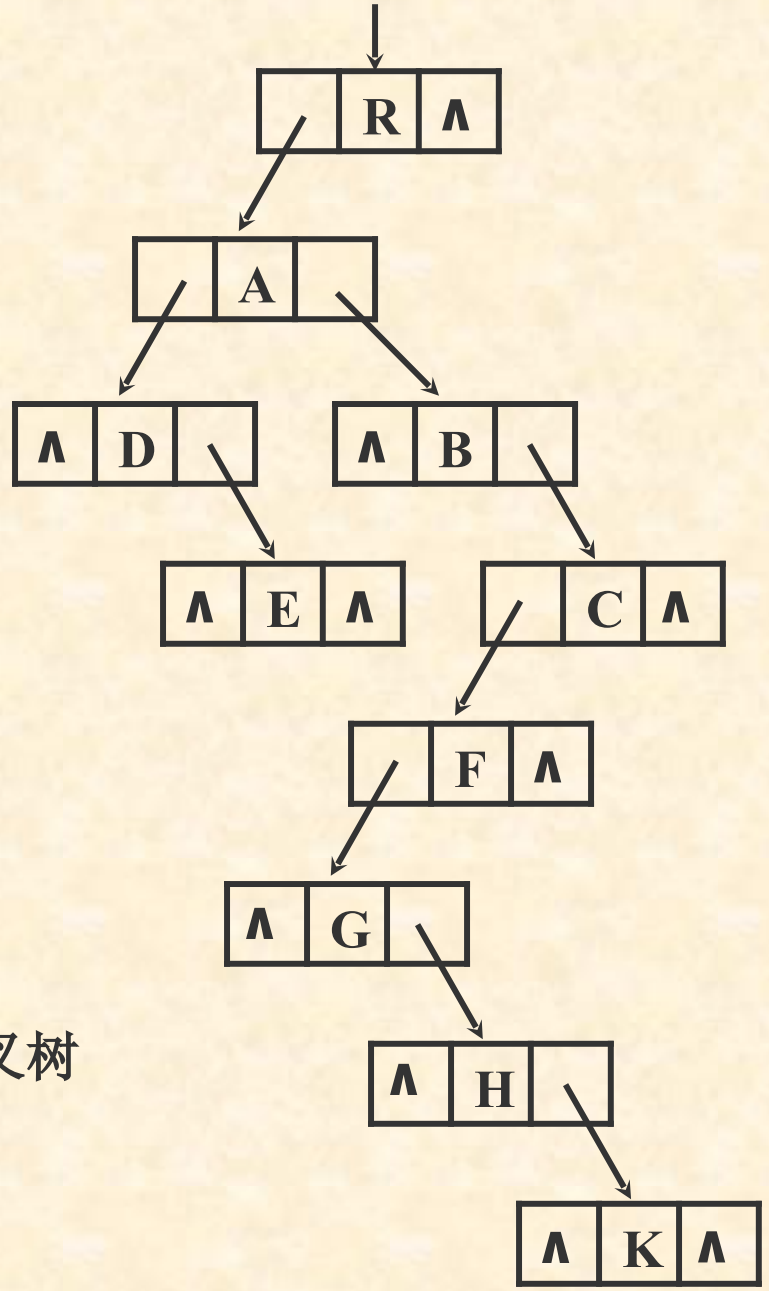


C 语言的类型描述：

结点结构：


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





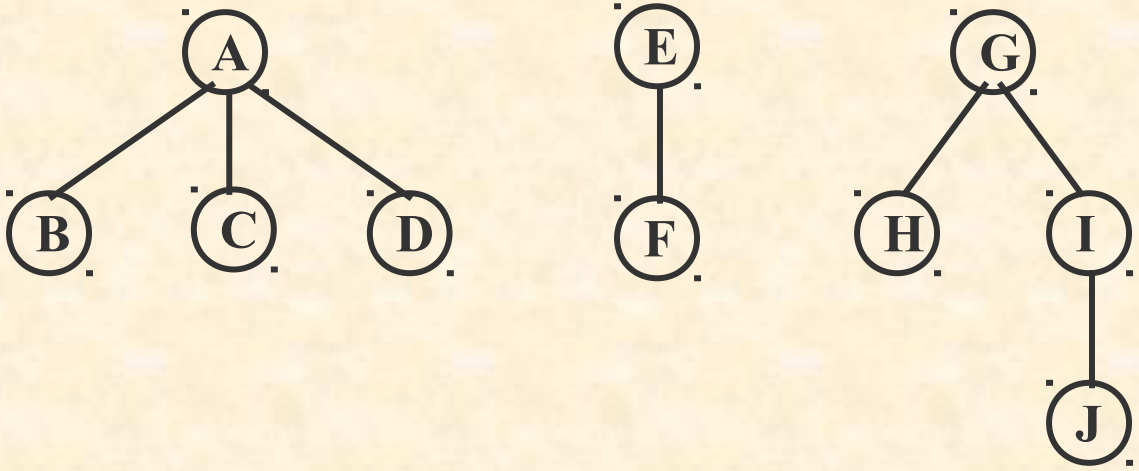
性质：

1. 树可以表示成二叉树的形式

   启示： 树与二叉树的转换

2. 树转换成一棵只有左子树的二叉树

6.4.2　森林与二叉树的转换

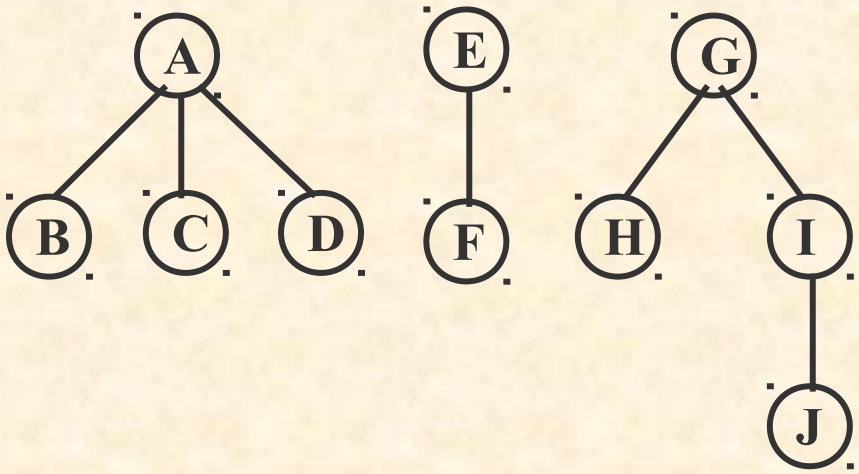


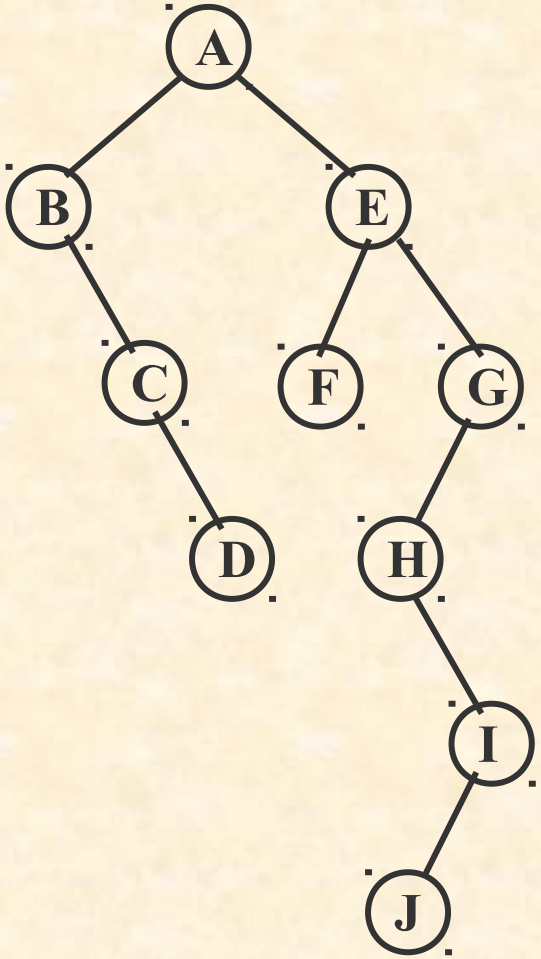
(1). 任何一棵树都可以转换为一棵只有左子树的二叉树。

(2). 森林是由若干棵树构成的集合，若把森林中前一棵树的根结点看成是后一棵树的根结点兄弟，就可以导出森林与二叉树的转换。

1. 森林转换成二叉树

把森林中第二棵树的根节点看成是第一棵树的根节点的兄弟，则可得出森林和二叉树的对应关系。





由此，树和森林的各种操作均可与二叉树的各种操作相对应。

应当注意的是，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**

树的遍历可有 3 条搜索路径：

先根 (次序) 遍历：

若树不空，则先访问根结点，然后依次先根遍历各棵子树。

后根 (次序) 遍历：

若树不空，则先依次后根遍历各棵子树，然后访问根结点。

按层次遍历：

若树不空，则自上而下自左至右访问树中每个结点。

先根遍历时顶点的访问次序

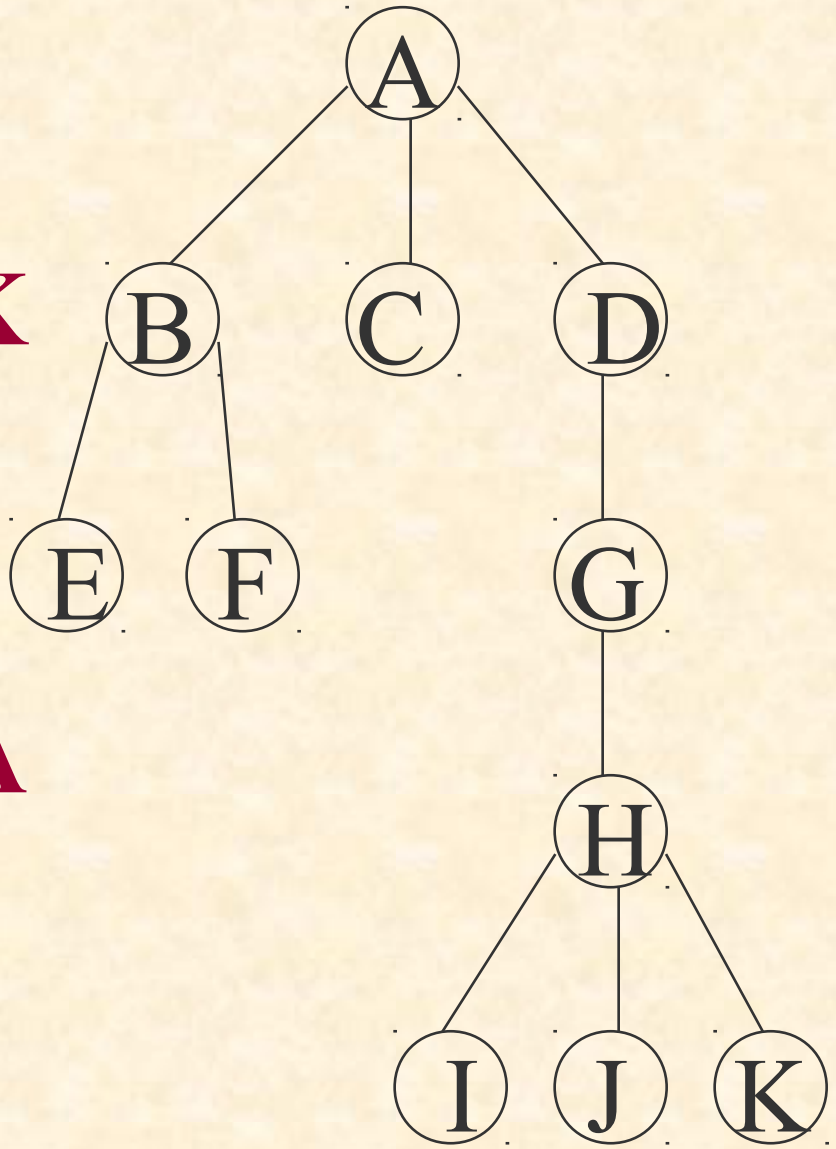
A B E F C D G H I J K

后根遍历时顶点的访问次序

E F B C I J K H G D A

层次遍历时顶点的访问次序:

A B C D E F G H I J K



森林的遍历

先序遍历

若森林不空，则

访问森林中第一棵树的根结点；

先序遍历森林中第一棵树的子树森林；

先序遍历森林中（除第一棵树之外）剩余树构成的森林。

即：依次从左至右对森林中的每一棵树进行**先根遍历**。

中序遍历

若森林不空，则

中序遍历森林中第一棵树的子树森林；

访问森林中第一棵树的根结点；

中序遍历森林中（除第一棵树之外）其余树构成的森林。

即：**依次从左至右**对森林中的每一棵**树**进行**后根遍历**。

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 哈夫曼树与哈夫曼编码

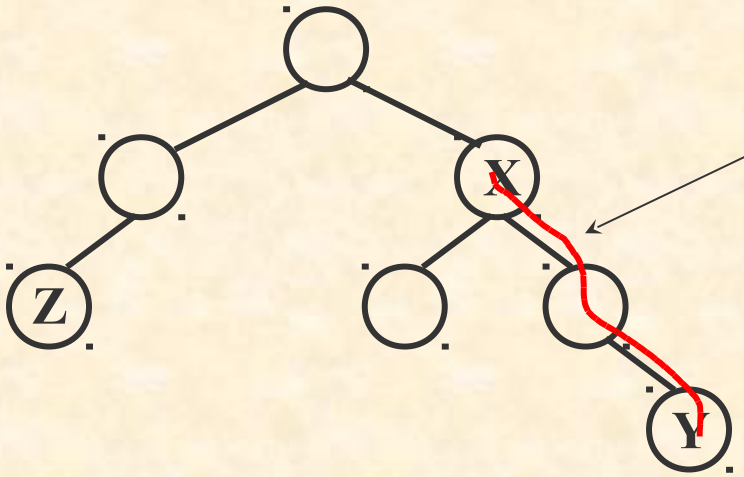
- 最优树的定义
- 如何构造最优树
- 前缀编码

6. 赫夫曼树 Huffman (最优二叉树)

6.1 基本概念：

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。



X 到 Y 的路径

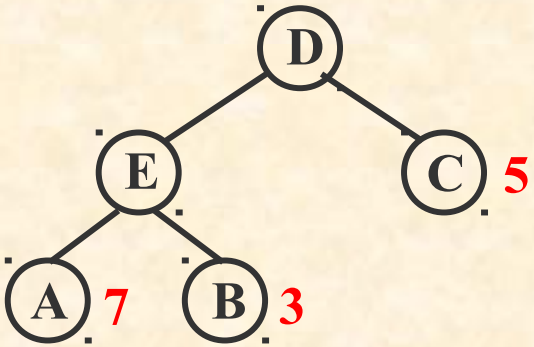
路径长度为 2

X 到 Z 的路径

树的路径长度是从树根到每一个结点的路径长度之和。

在具有相同结点数的所有二叉树中，完全二叉树 的路径长度是最短的。

推广，为结点加权 w 。



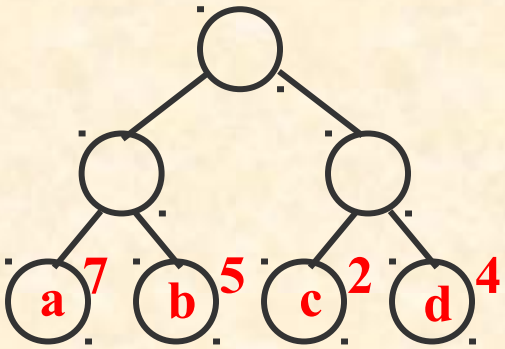
结点的带权路径长度为从根结点到该结点之间的路径长度与结点上权值的乘积。

树的带权路径长度为树中所有叶子结点的带权路径长度之和，通常记做 $WPL=\sum_{k=1}^{n} w_k L(v_k)$

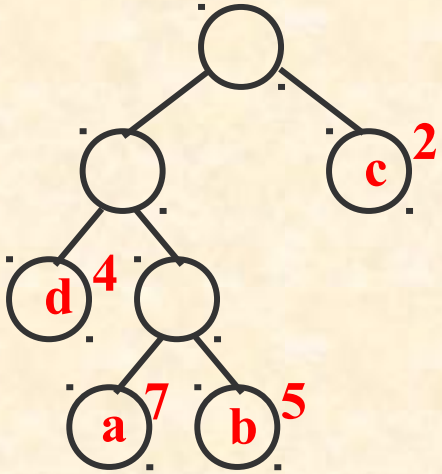
w_k 为叶子结点 v_k 的权值

$L(v_k)$ 为叶子结点 v_k 的路径长度

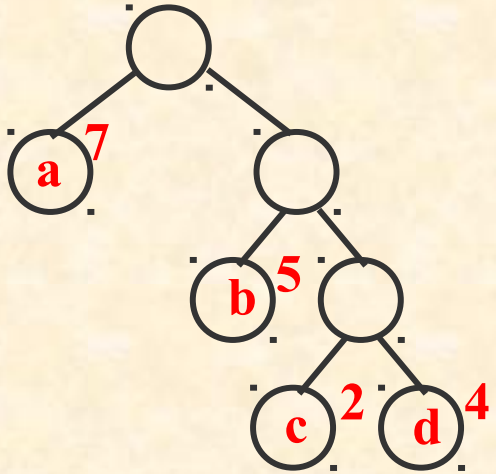
例，3 棵二叉树，都有 4 个叶子结点 a、b、c、d，分别带权 7、5、2、4，求它们各自的带权路径长度。



(1)



(2)



(3)

(1) WPL = 7×2 + 5×2 + 2×2 + 4×2 = 36

(2) WPL = 7×3 + 5×3 + 2×1 + 4×2 = 46

(3) WPL = 7×1 + 5×2 + 2×3 + 4×3 = 35

假设有 n 个权值 $\{w_1, w_2, \ldots w_n\}$，试构造一棵有 n 个叶子结点的二叉树，每个叶子结点带权为 w_i，则其中带权路径长度 WPL 最小的二叉树称做最优二叉树或赫夫曼树。在霍夫曼树中，权值大的结点离根最近。

如何构造赫夫曼树？

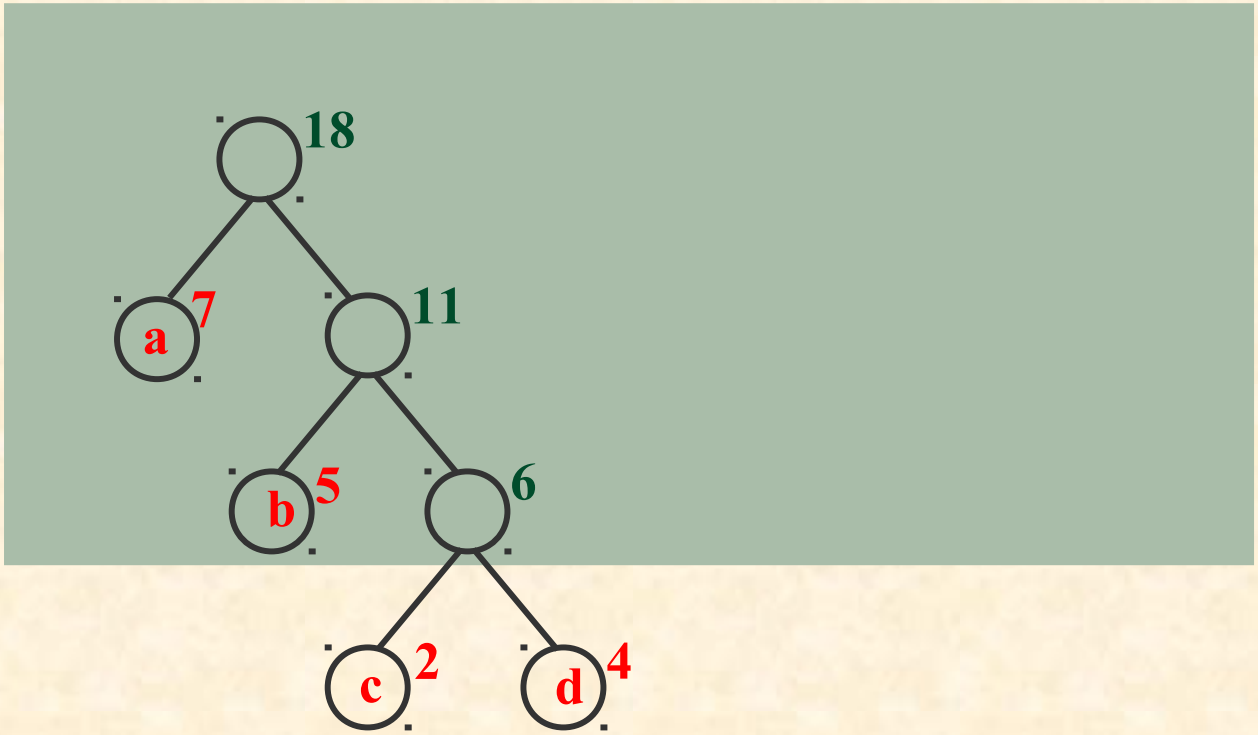
(1) 根据给定的 n 个权值 $\{w_1, w_2, \ldots w_n\}$ 构成 n 棵二叉树的集合 $F = \{T_1, T_2, \ldots T_n\}$，其中每棵二叉树 T_i 中只有一个权值为 w_i 的根结点。

(2) 在 F 中选取两棵根结点权值最小的树作为左、右子树构造一棵新的二叉树，且置新二叉树的根结点的权值为其左、右子树根结点的权值之和。

(3) 在 F 中删除这两棵树，同时将新得到的二叉树加入集合 F 中。

(4) 重复 (2) 和 (3)，直到 F 中只含一棵树为止。

例， 4 个叶子结点 a、b、c、d，分别带权 7、5、2、4。



3.2 赫夫曼编码

1. 编码

例，传送 ABACCD，四种字符，可以分别编码为 00,01,10,11。

则原电文转换为 00 01 00 10 10 11 。

对方接收后，采用二位一分进行译码。

当然，为电文编码时，总是希望总长越短越好，

如果对每个字符设计长度不等的编码，且让电文中出现次数较多的字符采用较短的编码，则可以减短电文的总长。

例，对 ABACCD 重新编码，分别编码为 0 , 00 , 1 , 01 。

A B C D

则原电文转换为 0 00 0 1 1 01 。 减短了。

00 01 00 10 10 11

问题： 如何译码？

前四个二进制字符就可以多种译法。

AAAA BB

2. 前缀编码

若设计的长短不等的编码，满足任一个编码都不是另一个编码的前缀，则这样的编码称为**前缀编码**。

例， A , B , C , D 前缀编码可以为 0 , 110 , 10 , 111

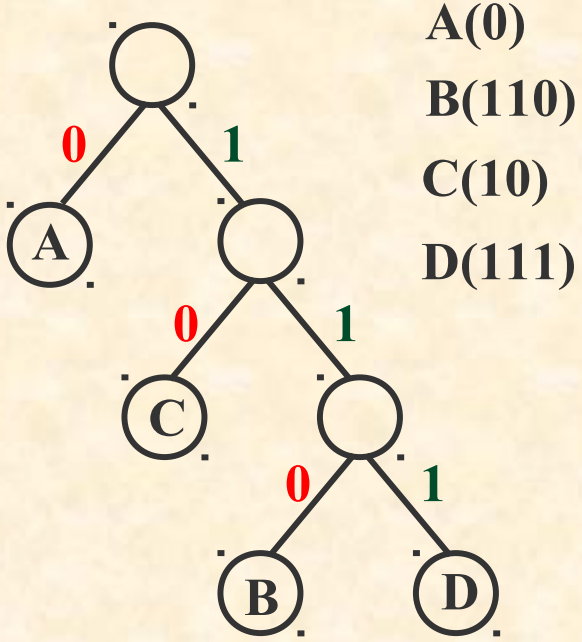
利用**二叉树**设计二进制前缀编码。

叶子结点表示 A , B , C , D 这 4 个字符

左分支表示 ‘ 0’，右分支表示 ‘ 1’

从根结点到叶子结点的路径上经过的二进制符号串作为该叶子结点字符的编码

路径长度为编码长度



A(0)

B(110)

C(10)

D(111)

如何得到最短的二进制前缀编码？

3. 赫夫曼编码

设每种字符在电文中出现的概率 w_i 为，则依此 n 个字符出现的概率做权，可以设计一棵赫夫曼树，使

$$WPL = \sum_{i=1}^{n} w_i \, l_i \quad \text{最小}$$

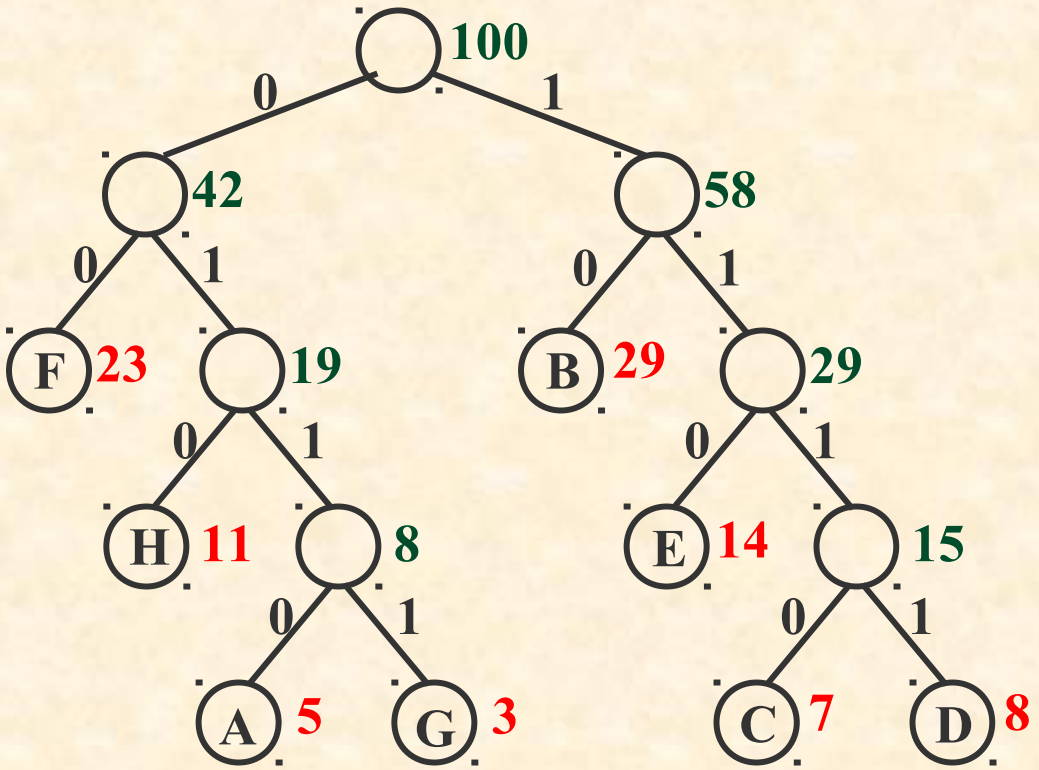
w_i 为叶子结点的出现概率 （ 权 ）

l_i 为根结点到叶子结点的路径长度

例，某通信可能出现 A B C D E F G H 8 个字符，其概率分别为 0.05 , 0.29 , 0.07 , 0.08 , 0.14 , 0.23 , 0.03 , 0.11 ， 试设计赫夫曼编码

不妨设 w = { 5 , 29 , 7 , 8 , 14 , 23 , 3 , 11 }

排序后 w { 100 }



A (0110)

B (10)

C (1110)

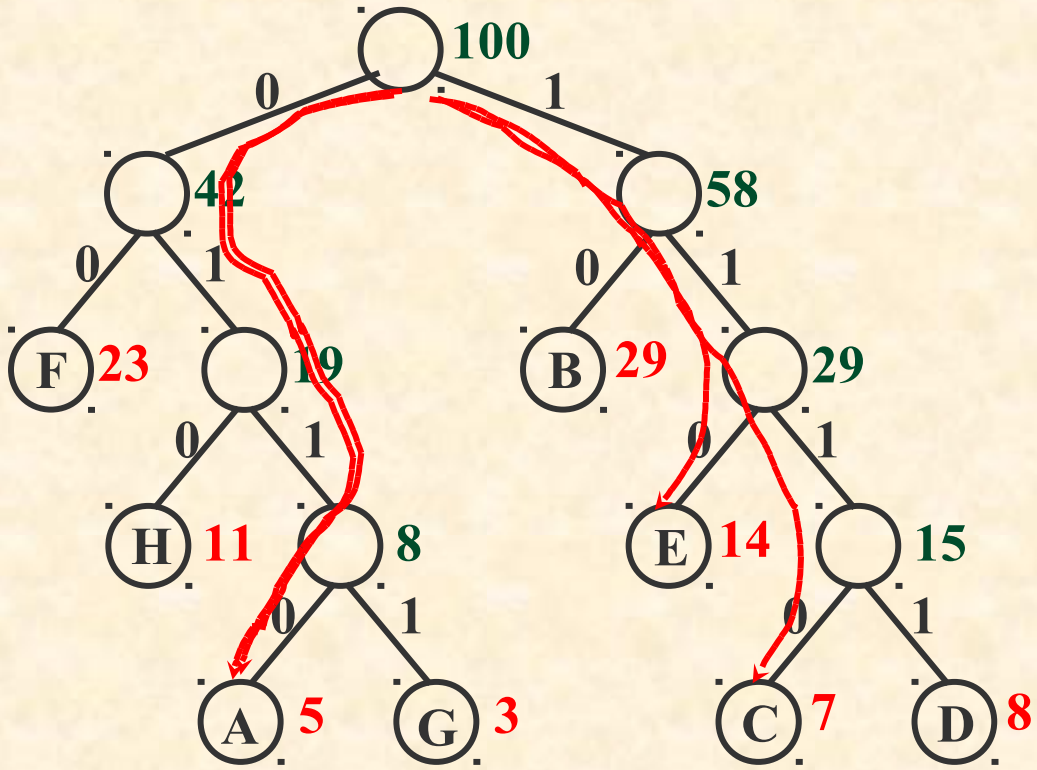
D (1111)

E (110)

F (00)

G (0111)

H (010)



A (0110)
B (10)
C (1110)
D (1111)
E (110)
F (00)
G (0111)
H (010)

ACEA 编码为 0110 1110 110 0110

如何译码？ A C E A

1. 从根结点出发，从左至右扫描编码，

2. 若为 ‘0’则走左分支，若为‘1’则走右分支，直至叶结点为止，

3. 取叶结点字符为译码结果，返回重复执行 1,2,3 直至全部译完为止

建立赫夫曼树及求赫夫曼编码的算法 (p147 算法 6.12)

```
typedef struct {
        unsigned int weight;
        unsigned int parent,lchild,rchild;
         }  HTNode, *HuffmanTree;
typedef char  **HuffmanCode;
void HuffmanCoding(   HuffmanTree &HT,
HuffmanCode &HC, int *w, int n  )
  {  HuffmanTree p;          char *cd;      int i,s1,s2,start;
     unsigned int c,f;
     if (n<=1) return;     // n 为字符数目，  m 为结点数目
     int m=2*n-1;
     HT = (HuffmanTree)malloc((m+1)*sizeof(HTNode));
                              //  0 号单元未用
```

```
 for (p=HT, i=1; i<=n; ++i,++p,++w)
     { p->weight = *w; p->parent=0; p->lchild=0;p->rchild=0; }
                 // *p = { *w,0,0,0 };
 for (; i<=m;++i,++p)
     {  p->weight = 0; p->parent=0;  p->lchild=0; p->rchild=0; }
                //*p={ 0,0,0,0 };
 for (i=n+1; i<=m;++i)    //   建赫夫曼树
     {
          Select(HT,i-1,s1,s2);
         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;
     }
```

```
// 从叶子到根逆向求赫夫曼编码
  HC=  (HuffmanCode) malloc ((n+1)*sizeof(char *));
  cd = (char*)malloc(n*sizeof(char));
  cd[n-1]='\0';
  for (i=1;i<=n;++i)
      {
        start = n-1;
        for (c=i,f=HT[c].parent; f!=0; c=f,f=HT[f].parent)
        if (HT[f].lchild ==c) cd[--start]='0';
        else cd[--start]='1';
        HC[i]=(char *)malloc((n-start)*sizeof(char));
        strcpy(HC[i],&cd[start]);
        printf("%s\n",HC[i]);
      }
  free(cd);
}
```

- 深度为 5 的二叉树至多有 (　　) 个结点。

  A．16　　B．32

  C．31　　D．10

- 具有 10 个叶子结点的二叉树中有_____个度为 2 的结点。

  A．8　　B．9　　C．10　　D．11

- 将一棵有多个结点的完全二叉树从根这一层开始，每一层上从左到右依次对结点进行编号，根结点的编号为 1，则编号为 49 的结点的左孩子编号为 (　　)。

  A．98　　B．99　　C．50

  D．48

- **按照二叉树的定义，具有 3 个结点的二叉树有（　　）种形态。**

A．3　　B．4　　C．5　　D．6

- **某二叉树的先序序列和后序序列正好相反，则该二叉树一定是（　　）的二叉树。**

A．空或只有一个结点

B．高度等于其结点数

C．任一结点无左孩子

D．任一结点无右孩子

- 假定一棵二叉树的结点个数为 50，则它的最小深度为 _____，最大深度为 _____。
- 一棵树的后根序列与其转换的二叉树的_____ 序列相同，先根序列与其转换的二叉树的_____ 序列相同。
- 具有 400 个结点的完全二叉树的深度为 _____。
- 假定一棵二叉树的结点数为 18，则它的最小深度为 _____，最大深度为 _____。

- 已知二叉树的后序和中序序列如下，画出该二叉树。

  后序序列： DEABFCR

  中序序列： DAERBCF

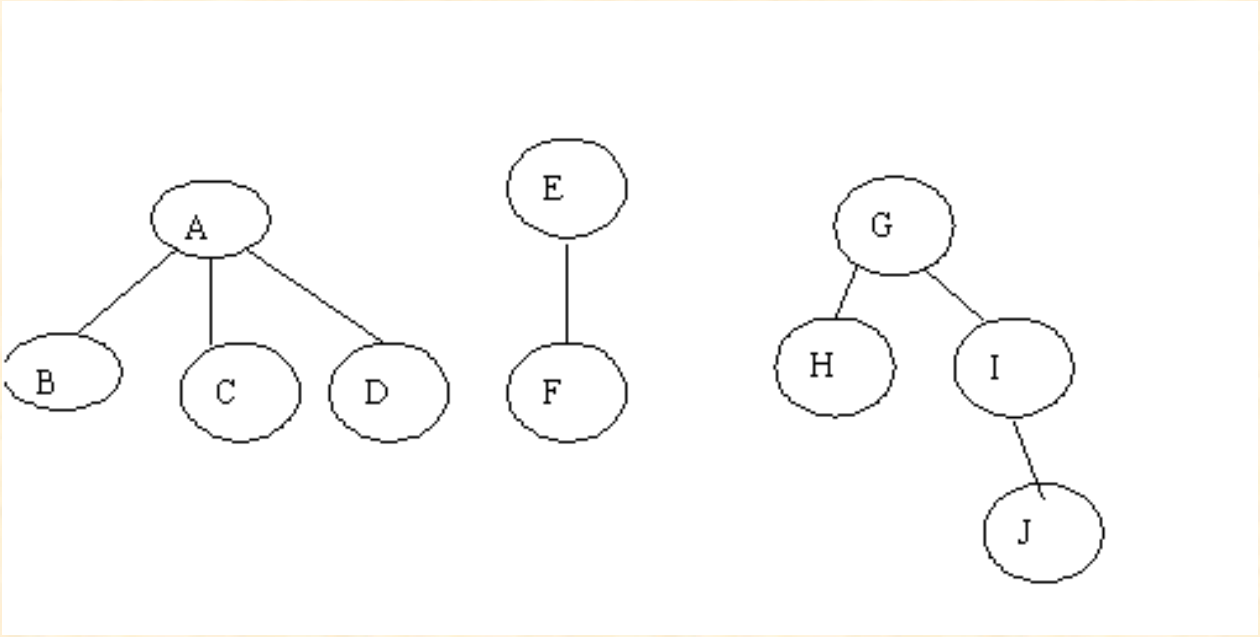
- 已知二叉树的后序和中序序列如下，画出该二叉树。

  后序序列： ABCDEFG

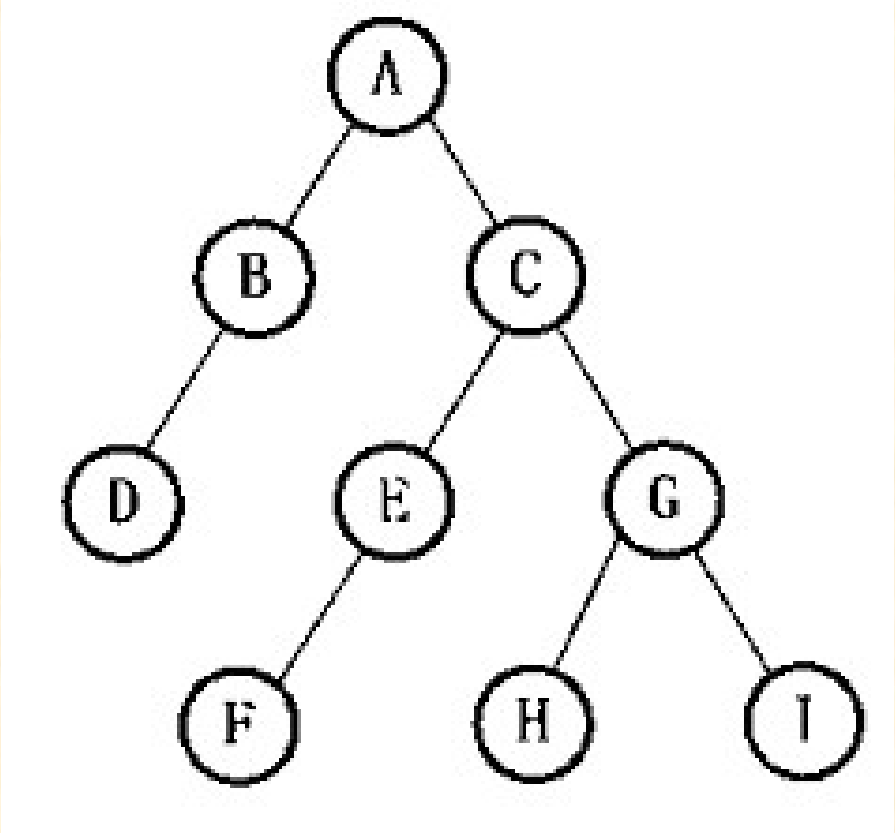
  中序序列： ACBGEDF

- 有 7 个带权结点，其权值分别为 3，7，8，2，6，10，14，试以它们为叶子结点生成一棵哈夫曼树，画出相应的哈夫曼树 (**左子树根结点的权小于等于右子树根结点的权**) 。

- 已知如下树林，画出对应的二叉树。



- 已知二叉树，画出中序的线索。



- 有一份电文中共使用五个字符： a 、 b 、 c 、 d 、 e, 它们的出现频率依次为 8 、 14 、 10 、 4 、 18 ，请构造相应的哈夫曼树 (左子树根结点的权小于等于右子树根结点的权) ，求出每个字符的哈夫曼编码。

已知二叉树以二叉链表作为存储结构，阅读下列算法，指出其功能。

```
void unknown(BinTree p,int &n)
{//*n 的初值为 0
 if (p)
 {
 if ((!p->lchild)&&(!p->rchild))
     n++;
 unknown(p->lchild,n);
 unknown(p->rchild,n);
 }
}
```

- 已知二叉树以二叉链表作为存储结构，阅读下列算法，说出它的功能， k 为全局变量，初值为 0 ，首次调用时 i 值为 0 。

```
void unknown(struct node *t,  int i)
    {   if (t!=NULL)
                {   cout<< t->data;                    /*  访问根结点  */
        i++;
        if(k<i)  k=i;
        unknown(t->lch, i);              /*  先根遍历左子树  */
        unknown(t->rch, i);              /*  先根遍历右子树  */
                }
    }                                                  /* unknown  */
```

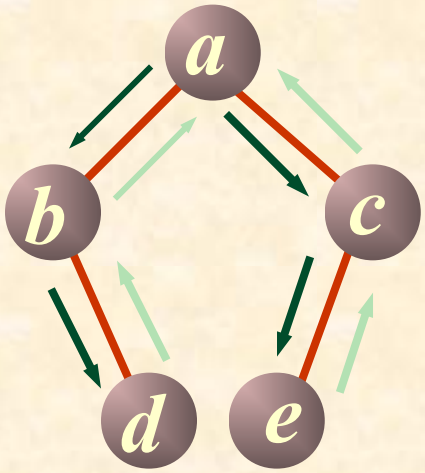
- 按照二叉树的定义，具有 3 个结点的二叉树有 (　　) 种形态。　　A．3　　B．4　　C．5　　D．6

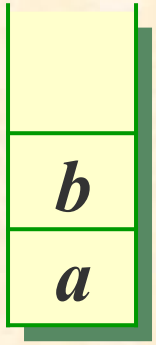
- 已知二叉树的中序和后序序列如下，画出该二叉树。

中序序列： daerbcf

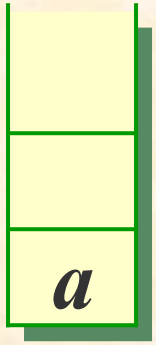
后序序列： deabfcr

中序遍历二叉树 (非递归算法) 用栈实现

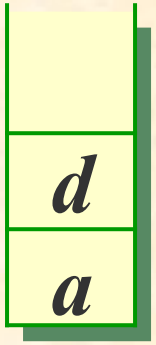




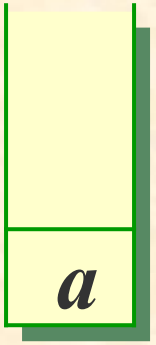
a b 入栈



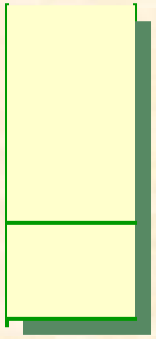
b 退栈访问



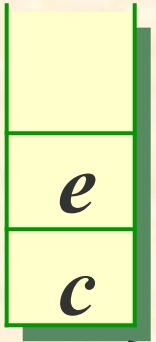
d 入栈



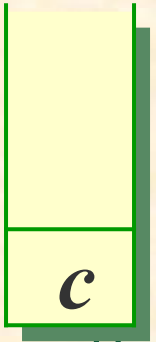
d 退栈访问



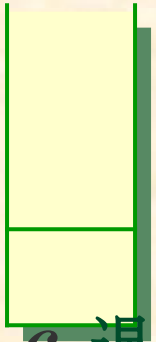
a 退栈访问



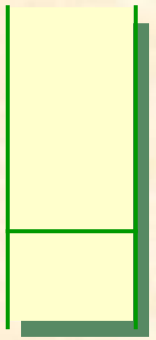
c e 入栈



e 退栈访问

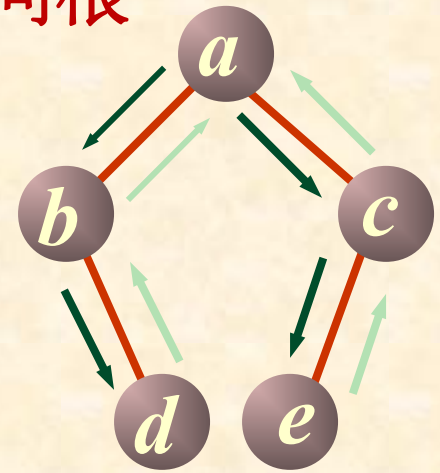


c 退栈访问

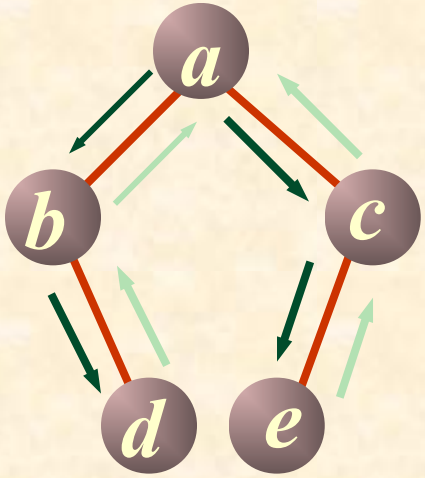


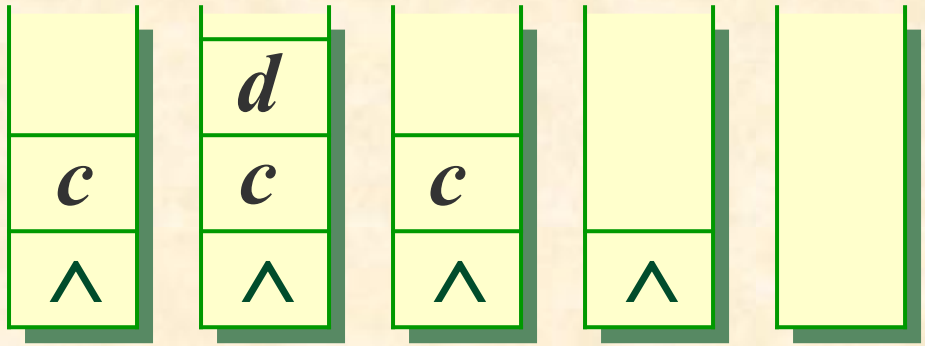
栈空

```
void InOrder ( BinTree T ) {
   stack S;   InitStack( &S );    // 递归工作栈
   BinTreeNode *p = T;            // 初始化
 while ( p != NULL || !StackEmpty(&S) ) {
        if( p != NULL )
         { Push(&S, p);   p = p->leftChild; }
      else if ( !StackEmpty(&S) ) {        // 栈非空
         Pop(&S, p);                       // 退栈
         printf("%c",p->data);             // 访问根
         p = p->rightChild;
       }//if
   }//while
 return ok;
}
```



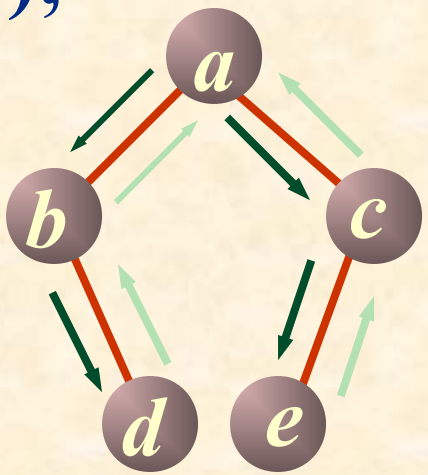
前序遍历二叉树 (非递归算法) 用栈实现





访问	访问	退栈	退栈	访问	
a	*b*	*d*	*c*	*e*	
进栈	进栈	访问	访问	左进	
c	*d*	*d*	*c*	空	
左进	左进	左进	左进	退栈	结束
b	空	空	*e*	^	

```
void PreOrder( BinTree T ) {
    stack S;   InitStack(&S); // 递归工作栈
    BinTreeNode * p = T;  Push (&S, NULL);
    while ( p != NULL ) {
        printf("%c",p->data);
        if ( p->rightChild != NULL )
            Push ( &S, p->rightChild );
        if  ( p->leftChild != NULL )
            p = p->leftChild;// 进左子树
        else Pop( &S, p );
    }
}
```



❖后序遍历二叉树 (非递归算法) 用栈实现

后序遍历时使用的栈的结点定义

```
typedef  struct {
    BinTreeNode *ptr;      // 结点指针
    enum tag{ L, R };      // 该结点退栈标记
} StackNode;
```

根结点的

tag = L ， 表示从左子树退出，访问右子树。

tag = R， 表示从右子树退出，访问根。

```
void PostOrder ( BinTree T ) {
    stack S;   InitStack(&S);  StackNode w;
    BinTreeNode * p = T;
    do {
            while ( p != NULL )
            {   // 向左子树走
                w.ptr = p;   w.tag = L;   Push (&S,
    w);
                p = p->leftChild;
            }
            int continue = 1;              // 继续循环标记
```

```
    while ( continue && !StackEmpty(&S) ) {
        Pop (&S, w);  p = w.ptr;
        switch ( w.tag ) {          // 判断栈顶 tag 标记
          case L :  w.tag = R;   Push (&S, w);
                    continue = 0;
                    p = p->rightChild;   break;
          case R :  printf("%c",p->data);
        }
      }
   } while ( p != NULL ||  !StackEmpty(&S) );
}
```

交换二叉树各结点的左、右子树 (递归算法)

```
void  unknown ( BinTreeNode * T ) {
   BinTreeNode *p = T, *temp;
   if ( p != NULL ) {
      temp = p->leftChild;
      p->leftChild = p->rightChild;
      p->rightChild = temp;
      unknown ( p->leftChild );
      unknown ( p->rightChild );
   }
}
```

使用栈消去递归算法中的两个递归语句

```
void unknown ( BinTreeNode * T ) {
    BinTreeNode *p, *temp;
    stack S;   InitEmpty (&S);
    if ( T != NULL ) {
        push(&S, T);
        while ( ! StackEmpty(&S) ) {
            Pop(&S, p);              // 栈中退出一个结
    点
            temp = p->leftChild;     // 交换子女
            p->leftChild = p->rightChild;
            p->rightChild = temp;
```

```
            if ( p->rightChild != NULL )
                push (&S, p->rightChild );
            if ( p->leftChild != NULL )
                push (&S, p->leftChild );
        }
    }
}
```