

---

# 第5章 树和二叉树

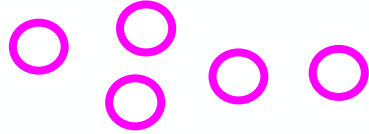
王迪

wangd@sdas.org

---

# 逻辑结构

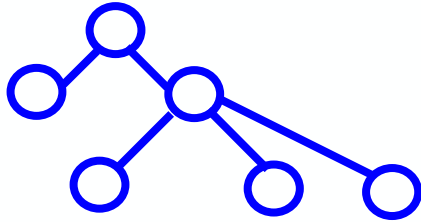
**集合**——数据元素间除“同属于一个集合”外，无其它关系



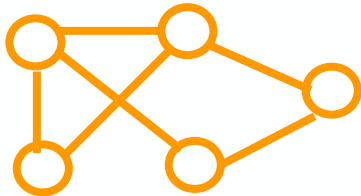
**线性结构**——一个对一个，如线性表、栈、队列



**树形结构**——一个对多个，如树



**图形结构**——多个对多个，如图



# 第5章 树和二叉树



5.1 树和二叉树的定义

5.2 案例引入

5.3 树和二叉树的抽象数据类型定义

5.4 二叉树的性质和存储结构

5.5 遍历二叉树和线索二叉树

5.6 树和森林

5.7 哈夫曼树及其应用

5.8 案例分析与实现



# 教学目标

1. 掌握二叉树的基本概念、性质和存储结构
2. 熟练掌握二叉树的前、中、后序遍历方法
3. 了解线索化二叉树的思想
4. 熟练掌握：哈夫曼树的实现方法、构造哈夫曼编码的方法
5. 了解：森林与二叉树的转换，树的遍历方法

# 5.1 树和二叉树的定义

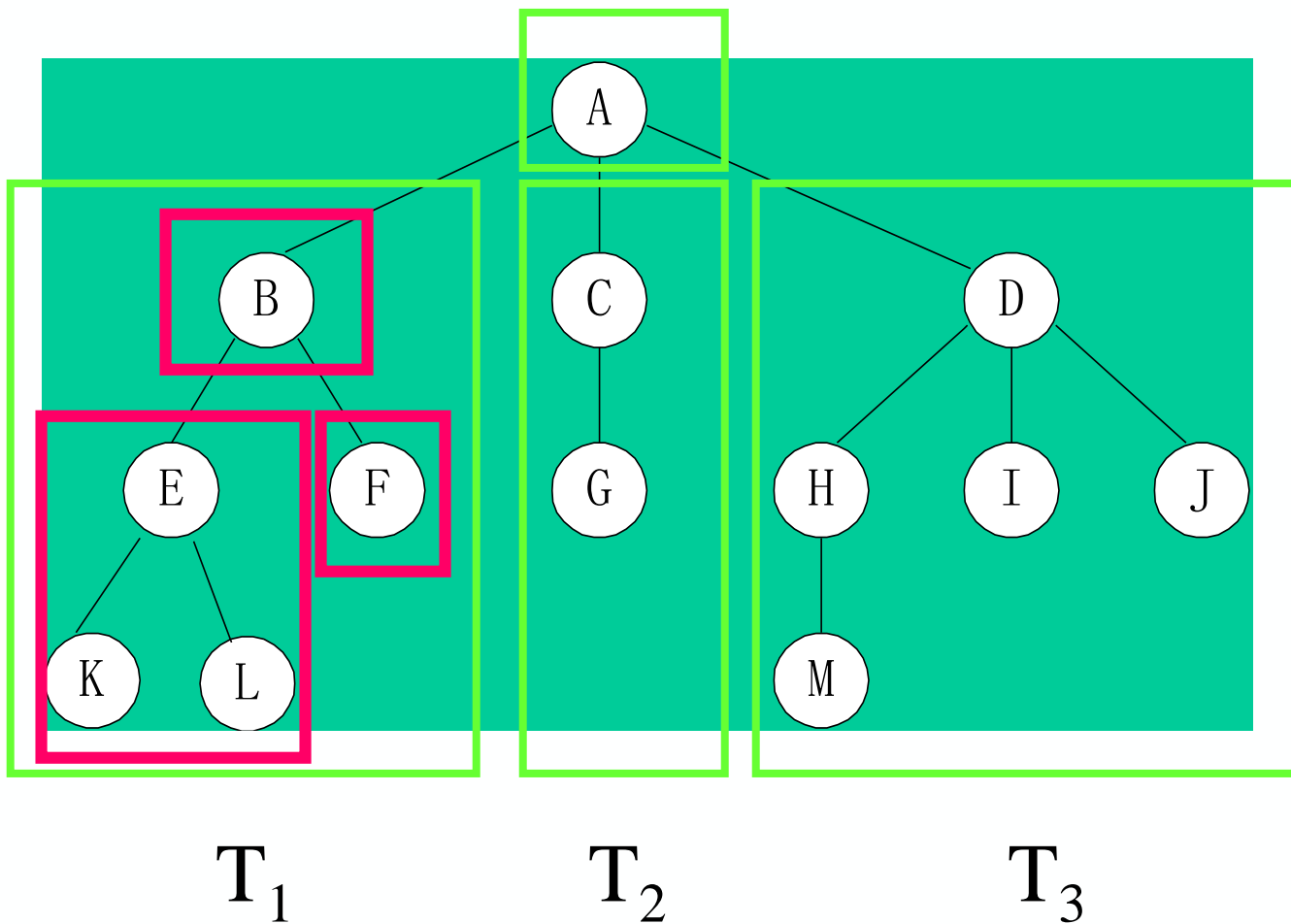
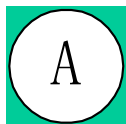


## 树的定义

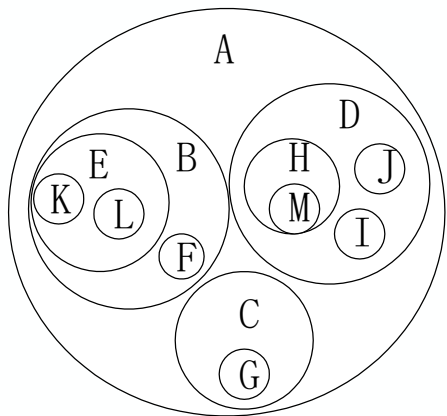
树 (Tree) 是  $n$  ( $n \geq 0$ ) 个结点的有限集, 它或为空树 ( $n = 0$ ) ; 或为非空树, 对于非空树  $T$ :

- (1) 有且仅有一个称之为根的结点;
- (2) 除根结点以外的其余结点可分为  $m$  ( $m > 0$ ) 个互不相交的有限集  $T_1, T_2, \dots, T_m$ , 其中每一个集合本身又是一棵树, 并且称为根的子树 (SubTree) 。

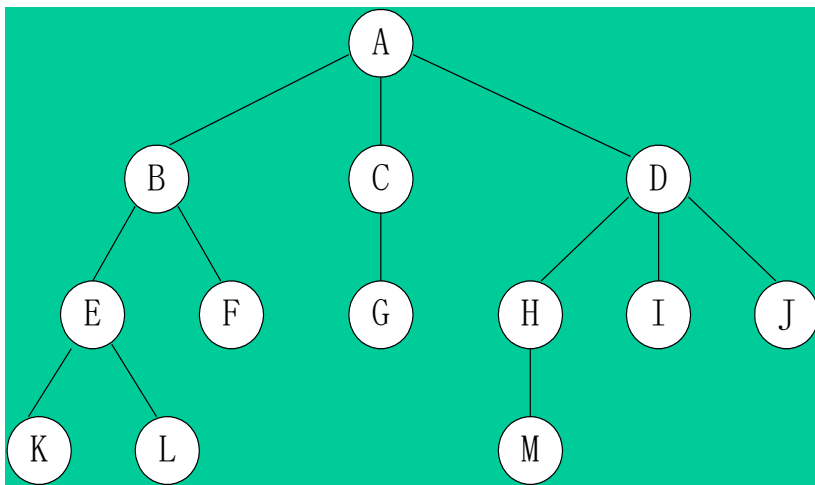
树是n个结点的有限集



# 树的其它表示方式

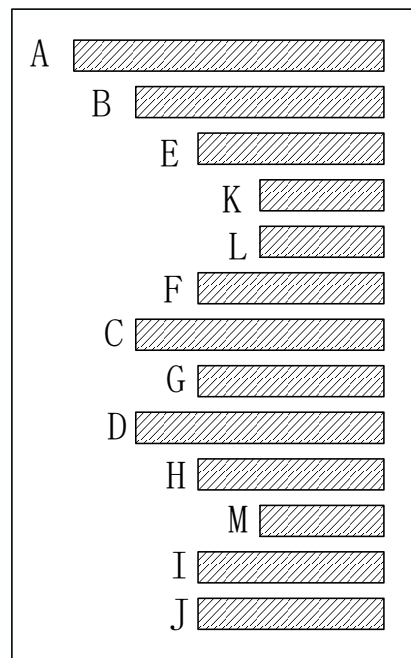


嵌套集合



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

广义表



凹入表示

# 基本术语

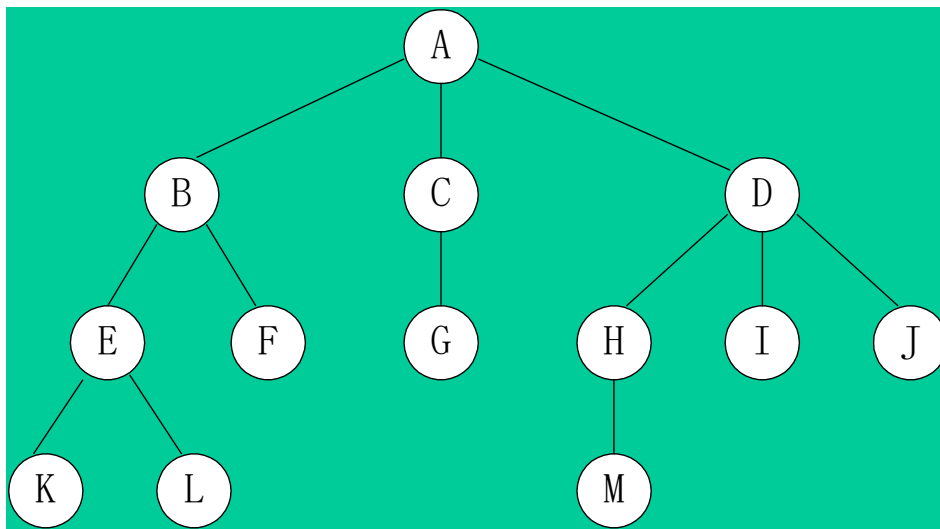
根 —— 即根结点(没有前驱)

叶子 —— 即终端结点(没有后继)

森林 —— 指 $m$ 棵不相交的树的集合(例如删除A后的子树个数)

有序树 —— 结点各子树从左至右有序, 不能互换 (左为第一)

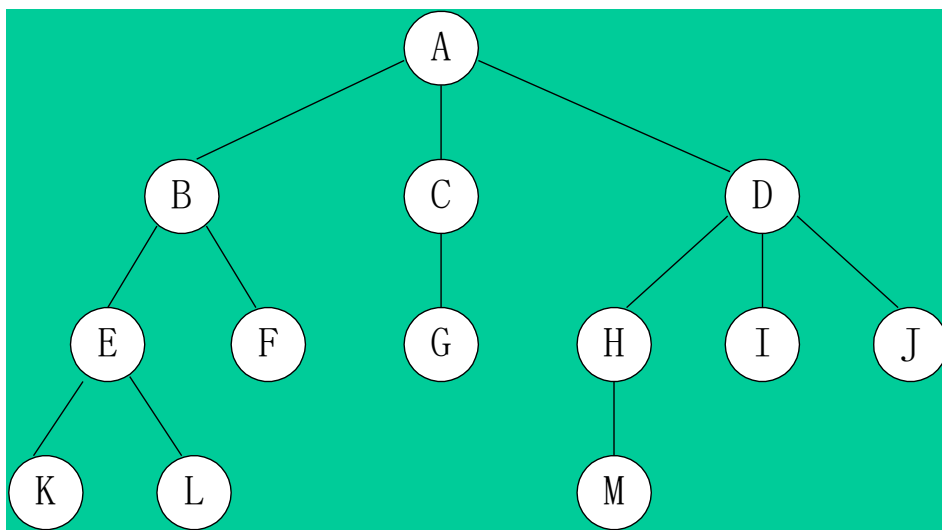
无序树 —— 结点各子树可互换位置。





# 基本术语

- 双亲 —— 即上层的那个结点(直接前驱)
- 孩子 —— 即下层结点的子树的根(直接后继)
- 兄弟 —— 同一双亲下的同层结点 (孩子之间互称兄弟)
- 堂兄弟 —— 即双亲位于同一层的结点 (但并非同一双亲)
- 祖先 —— 即从根到该结点所经分支的所有结点
- 子孙 —— 即该结点下层子树中的任一结点



# 基本术语

结点 —— 即树的数据元素

结点的度 —— 结点挂接的子树数

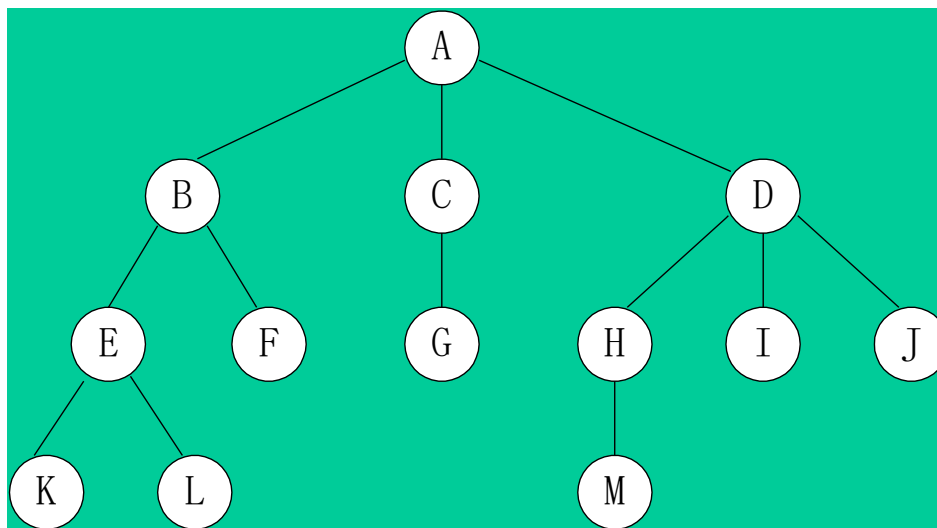
结点的层次 —— 从根到该结点的层数（根结点算第一层）

终端结点 —— 即度为0的结点，即叶子

分支结点 —— 即度不为0的结点（也称为内部结点）

树的度 —— 所有结点度中的最大值

树的深度  
(或高度) —— 指所有结点中最大的层数



层次

1

2

3

4

# 树结构和线性结构的比较

## 线性结构

第一个数据元素 无前驱

最后一个数据元素 无后继

其它数据元素

一个前驱，一个后继

一对一

## 树结构

根节点（只有一个）无双亲

叶子节点（可以有多个）无孩子

其它节点—中间节点

一个双亲，多个孩子

一对多

# 二叉树的定义

二叉树 (Binary Tree) 是  $n$  ( $n \geq 0$ ) 个结点所构成的集合, 它或为空树 ( $n = 0$ ) ; 或为非空树, 对于非空树  $T$ :

- (1) 有且仅有一个称之为根的结点;
- (2) 除根结点以外的其余结点分为两个互不相交的子集  $T_1$  和  $T_2$ , 分别称为  $T$  的左子树和右子树, 且  $T_1$  和  $T_2$  本身又都是二叉树。

---

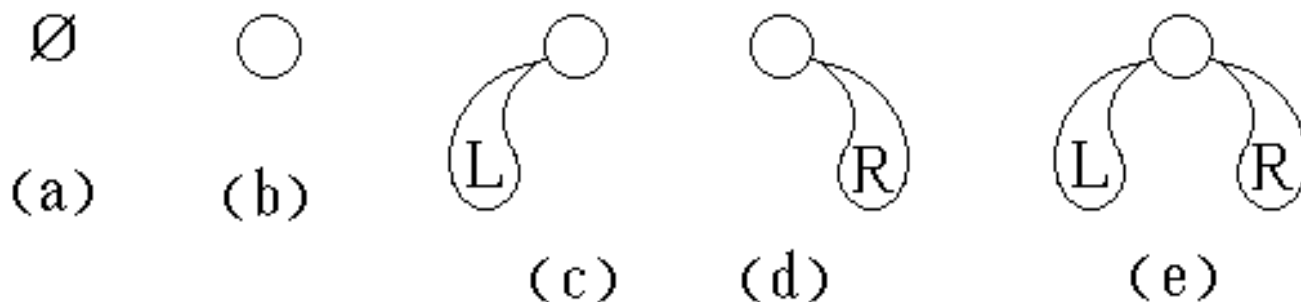
普通树（多叉树）若不转化为二叉树，则运算很难实现

为何要重点研究每结点最多只有两个“叉”的树？

- ✓ 二叉树的结构最简单，规律性最强；
- ✓ 可以证明，所有树都能转为唯一对应的二叉树，不失一般性。

## 二叉树基本特点:

- 结点的度小于等于2
- 有序树（子树有序，不能颠倒）

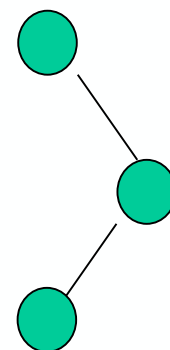
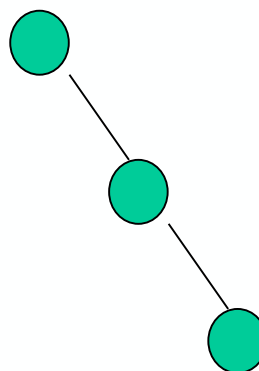
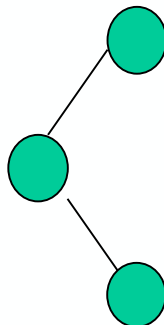
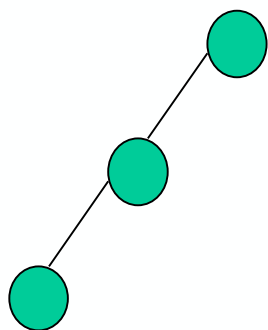
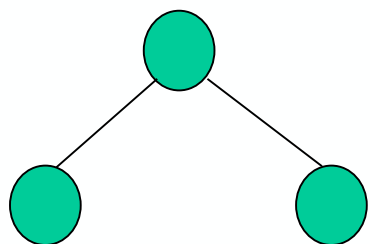


**二叉树的五种不同形态**

# 练习

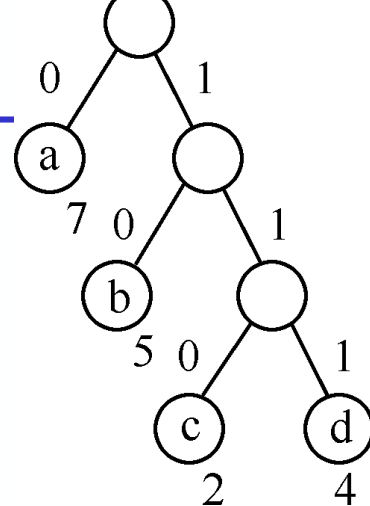
具有3个结点的二叉树可能有几种不同形态？普通树呢？

5种/2种



# 5.2 案例引入

## 案例5.1：数据压缩问题



将数据文件转换成由0、1组成的二进制串，称之为编码。

(a) 等长编码方案

字符	编码
a	00
b	01
c	10
d	11

(b) 不等长编码方案1

字符	编码
a	0
b	10
c	110
d	111

(c) 不等长编码方案2

字符	编码
a	0
b	01
c	010
d	111

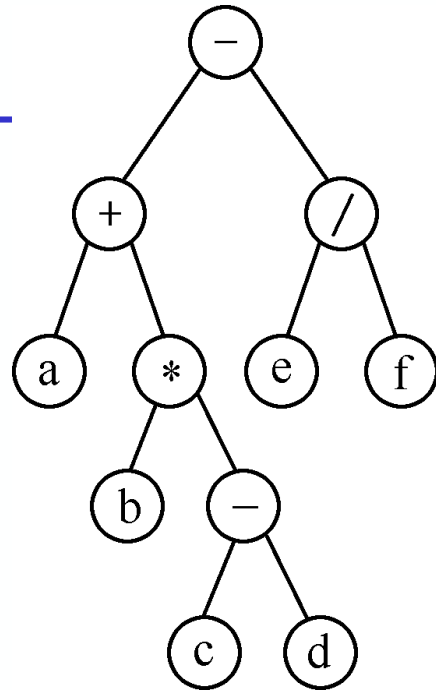


## 案例5.2：利用二叉树求解表达式的值

以二叉树表示表达式的递归定义如下：

(1) 若表达式为数或简单变量，则相应二叉树中仅有一个根结点，其数据域存放该表达式信息；

(2) 若表达式为“第一操作数 运算符 第二操作数”的形式，则相应的二叉树中以左子树表示第一操作数，右子树表示第二操作数，根结点的数据域存放运算符（若为一元运算符，则左子树为空），其中，操作数本身又为表达式。



$(a + b * (c - d) - e / f)$ 的二叉树

## 5.3 树和二叉树的抽象数据类型定义



### 二叉树的抽象数据类型定义

ADT BinaryTree{

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 $D=\Phi$ , 则 $R=\Phi$ ;

若 $D\neq\Phi$ , 则 $R=\{H\}$ ; 存在二元关系:

- ① root 唯一           //关于根的说明
- ②  $D_j \cap D_k = \Phi$    //关于子树不相交的说明
- ③ .....           //关于数据元素的说明
- ④ .....           //关于左子树和右子树的说明

基本操作 P:       //至少有20个

}ADT BinaryTree

CreateBiTree(&T,definition)

初始条件：definition给出二叉树T的定义。

操作结果：按definition构造二叉树T。

PreOrderTraverse(T)

初始条件：二叉树T存在。

操作结果：先序遍历T，对每个结点访问一次。

InOrderTraverse(T)

初始条件：二叉树T存在。

操作结果：中序遍历T，对每个结点访问一次。

PostOrderTraverse(T)

初始条件：二叉树T存在。

操作结果：后序遍历T，对每个结点访问一次。

## 5.4 二叉树的性质和存储结构



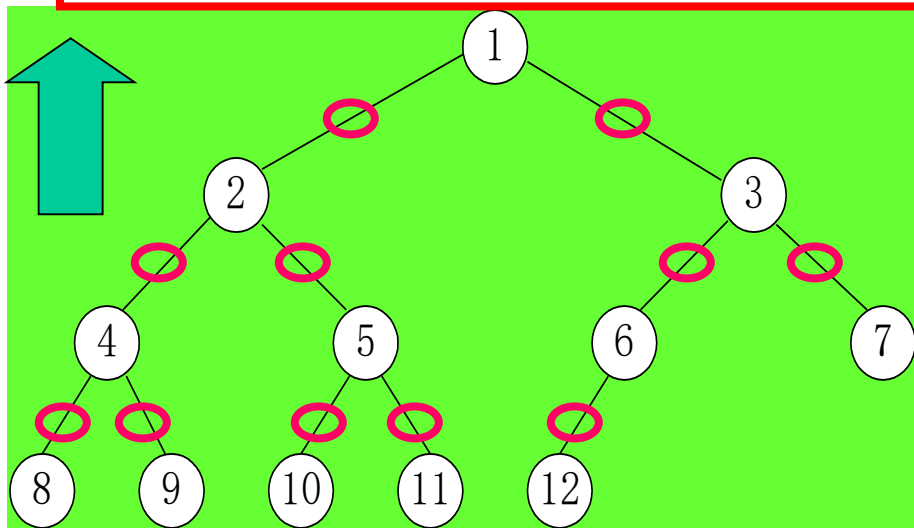
性质1：在二叉树的第 $i$ 层上至多有 $2^{i-1}$ 个结点

提问：第 $i$ 层上至少有\_\_1\_\_个结点？

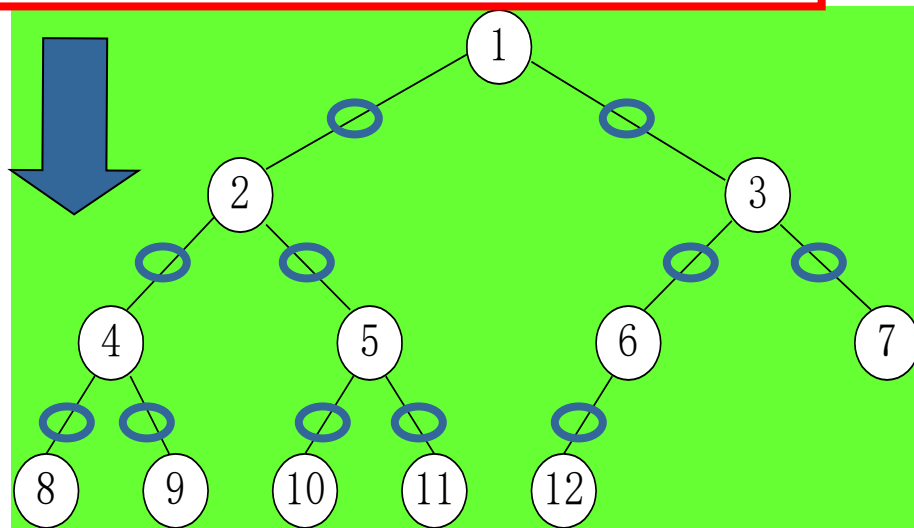
性质2：深度为 $k$ 的二叉树至多有 $2^k-1$ 个结点

提问：深度为 $k$ 时至少有\_\_k\_\_个结点？

**性质3:** 对于任何一棵二叉树，若2度的结点数有 $n_2$ 个，则叶子数 $n_0$ 必定为 $n_2+1$ （即 $n_0=n_2+1$ ）



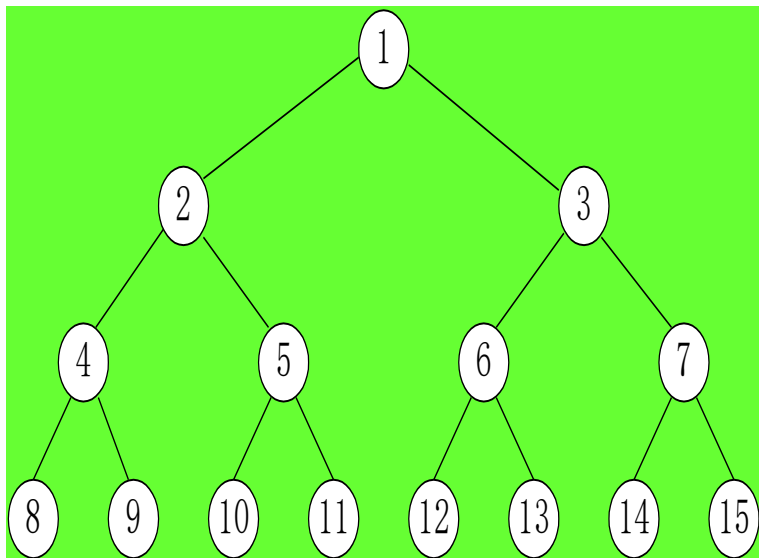
$$B = n - 1$$



$$B = n_2 \times 2 + n_1 \times 1$$

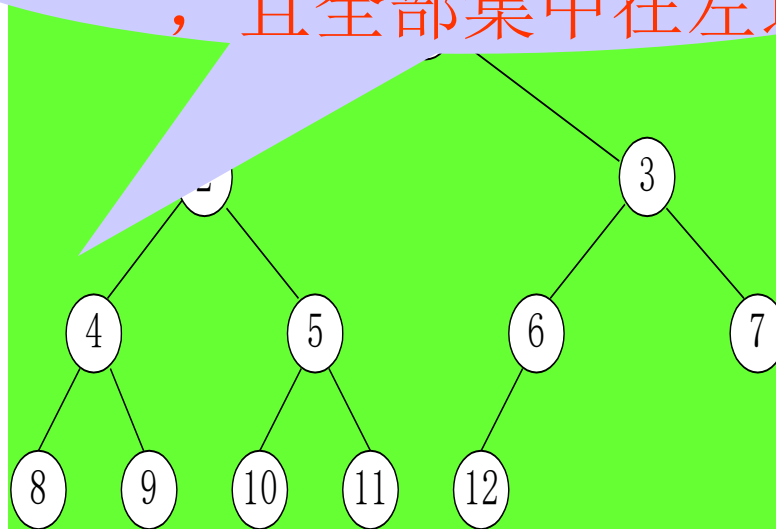
$$n = n_2 \times 2 + n_1 \times 1 + 1 = n_2 + n_1 + n_0$$

# 特殊形态的二叉树



**满二叉树：**一棵深度为 $k$  且有 $2^k - 1$ 个结点的二叉树。（特点：每层都“充满”了结点）

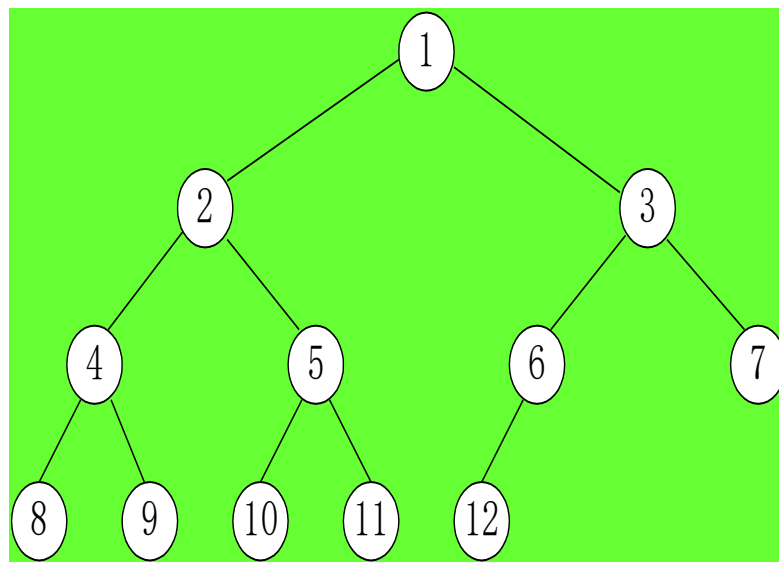
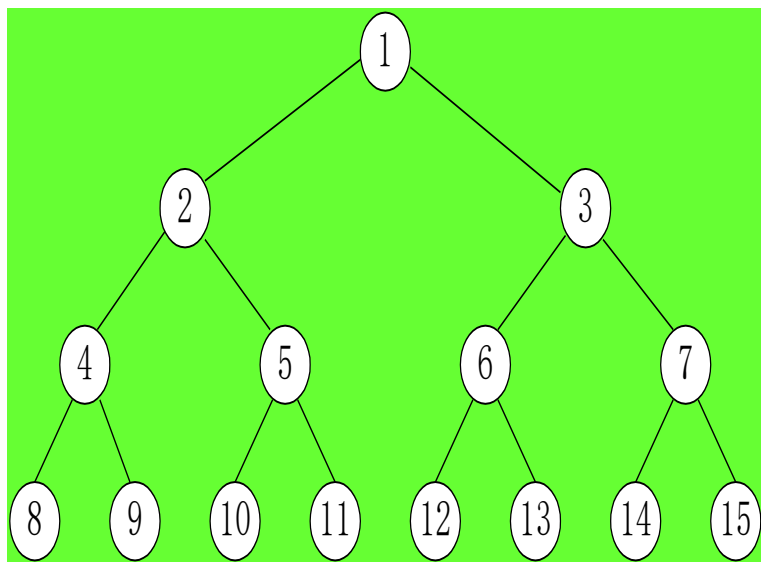
只有最后一层叶子不满，且全部集中在左边



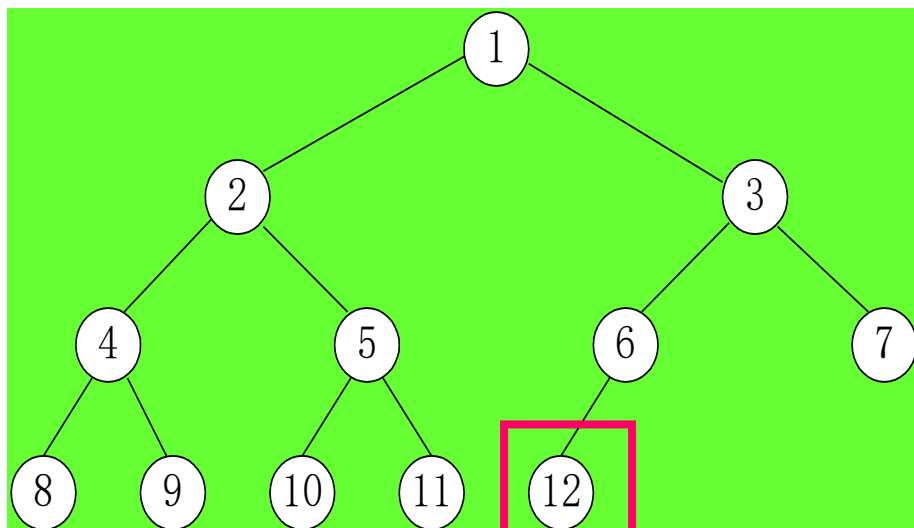
**完全二叉树：**深度为 $k$  的，有 $n$ 个结点的二叉树，当且仅当其每一个结点都与深度为 $k$  的满二叉树中编号从1至 $n$ 的结点一一对应

# 满二叉树和完全二叉树的区别

满二叉树是叶子一个也不少的树，而完全二叉树虽然前 $n-1$ 层是满的，但最底层却允许在右边缺少连续若干个结点。**满二叉树是完全二叉树的一个特例。**



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



$k-1$ 层

$$2^{k-1} - 1$$

$k$ 层

$$2^k - 1$$

$n$

$2^{k-1} - 1 < n \leq 2^k - 1$  或  $2^{k-1} \leq n < 2^k$   
 $k-1 \leq \log_2 n < k$ , 因为 $k$ 是整数  
所以 $k = \lfloor \log_2 n \rfloor + 1$



**性质5:** 对完全二叉树，若从上至下、从左至右编号，则编号为 $i$  的结点，其左孩子编号必为 $2i$ ，其右孩子编号必为 $2i+1$ ；其双亲的编号必为 $i/2$ 。

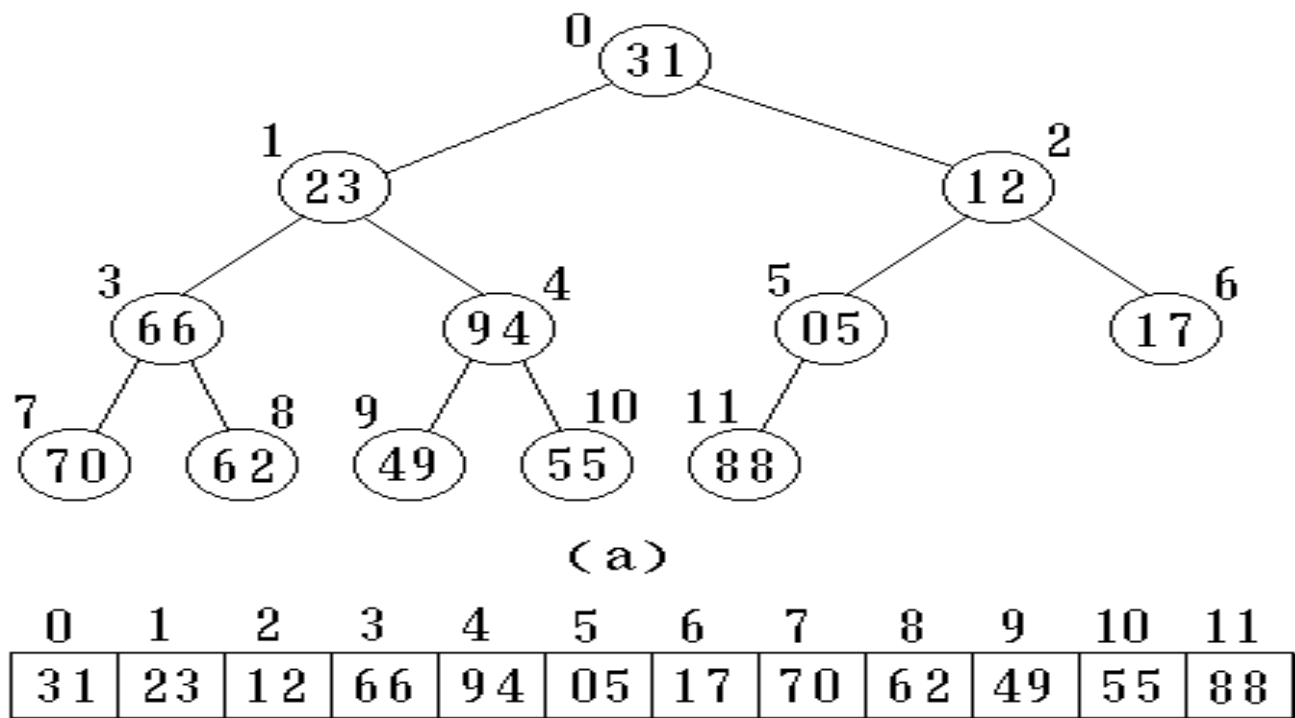
# 练习

---

一棵完全二叉树有5000个结点，可以计算出其叶结点的个数是（ 2500 ）。

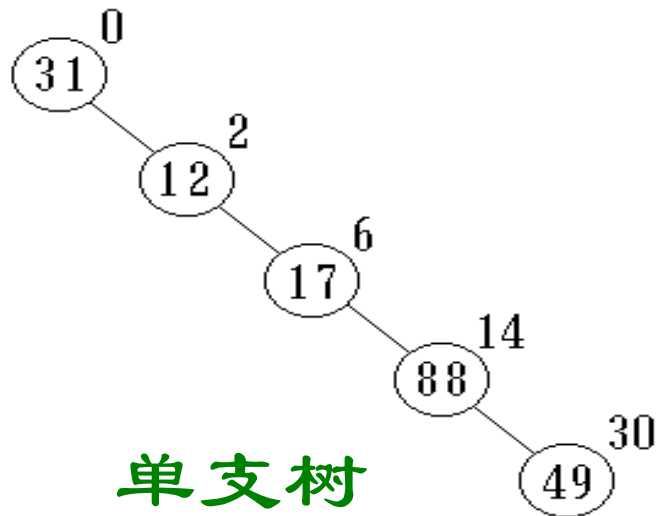
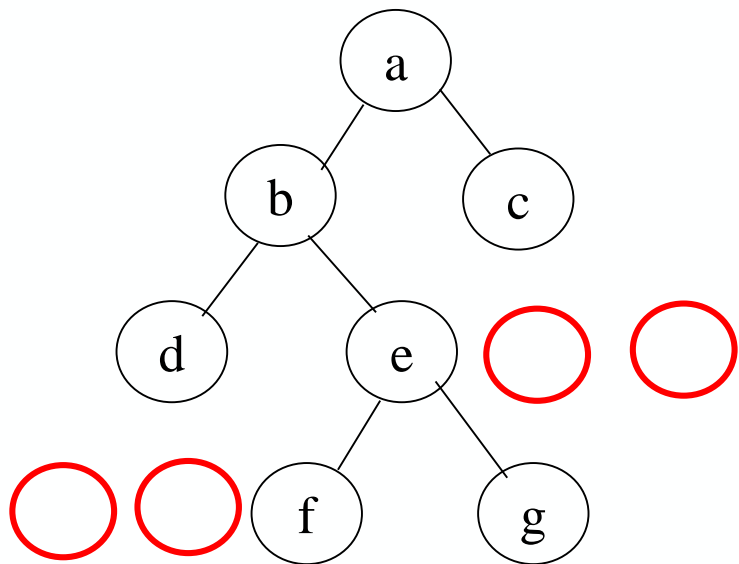
# 二叉树的顺序存储

实现：按**满二叉树**的结点层次编号，依次存放二叉树中的数据元素。



# 二叉树的顺序存储

0	1	2	3	4	5	6	7	8	9	10
a	b	c	d	e	0	0	0	0	f	g

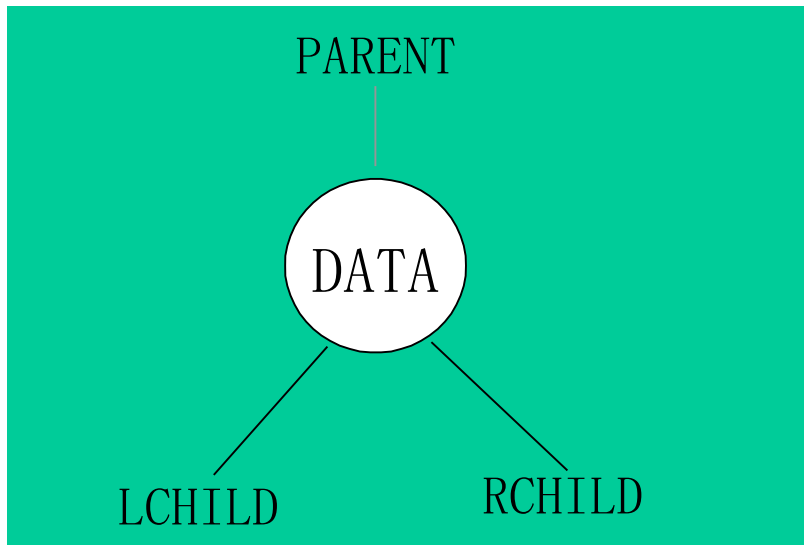


特点:

结点间关系蕴含在其存储位置中

浪费空间, 适于存满二叉树和完全二叉树

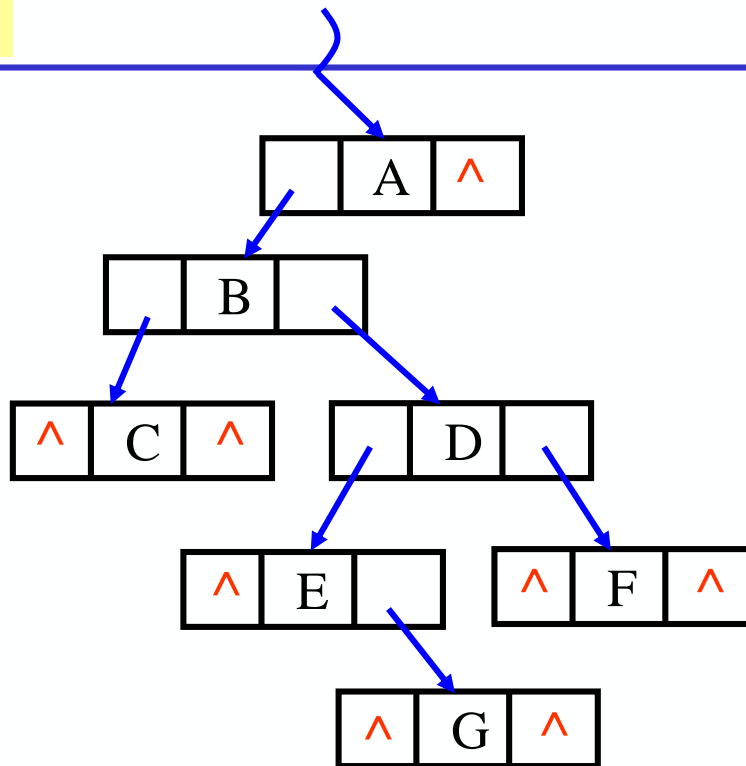
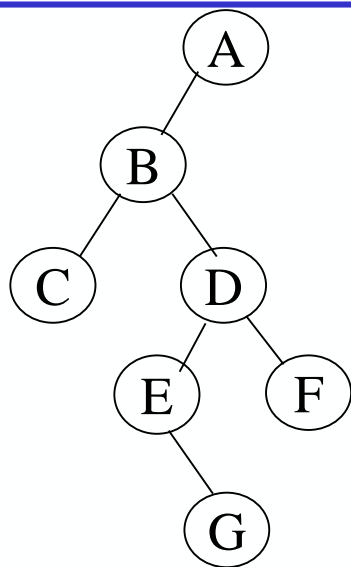
# 二叉树的链式存储



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

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

# 二叉链表



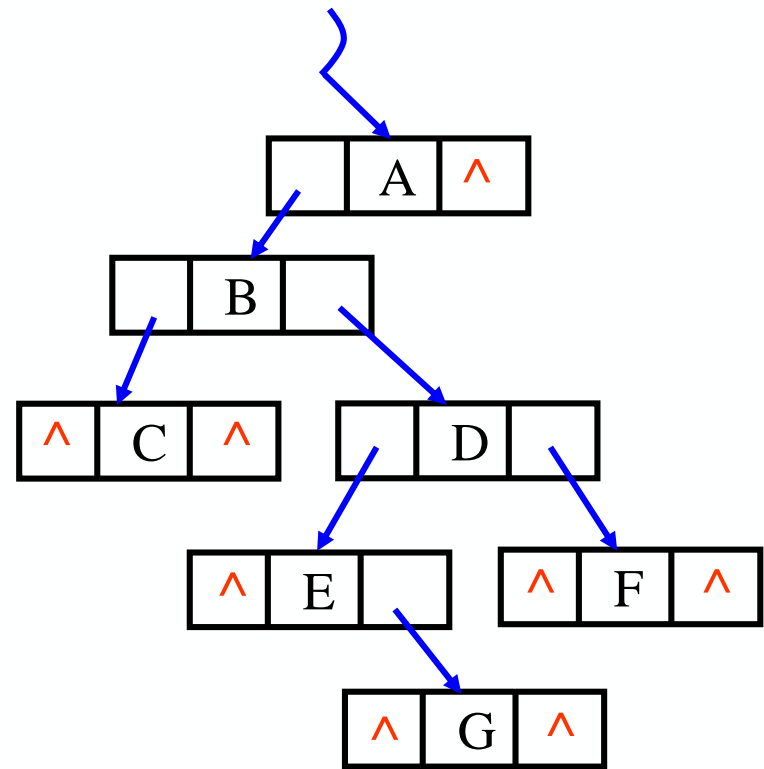
```
typedef struct BiNode{  
    TElemType  data;  
    struct BiNode  *lchild,*rchild; //左右孩子指针  
}BiNode,*BiTree;
```

# 练习

在 $n$ 个结点的二叉链表中，有  $n+1$  个空指针域

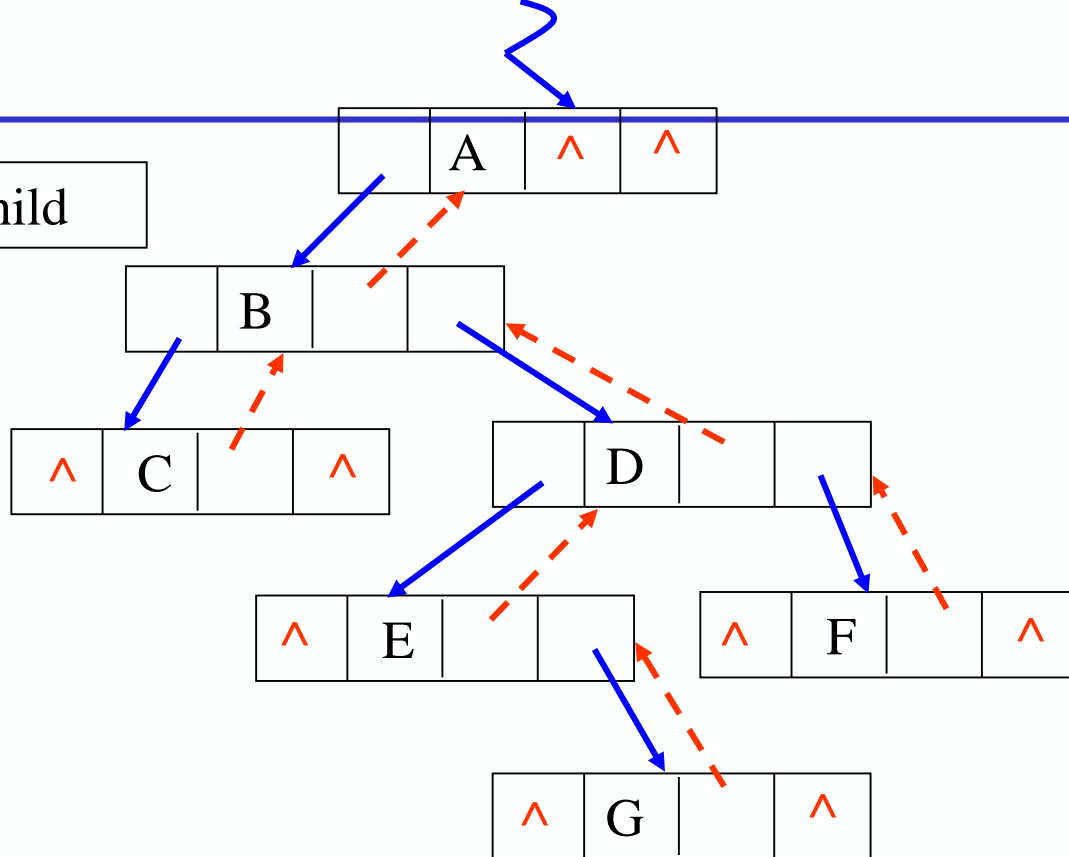
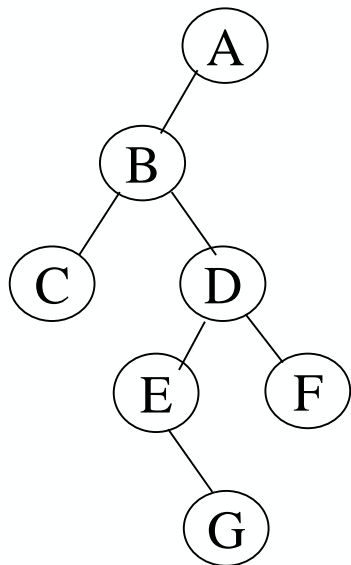
分析：必有  $2n$  个链域。除根结点外，每个结点有且仅有一个双亲，所以只会有  $n-1$  个结点的链域存放指针，指向非空子女结点。

$$\text{空指针数目} = 2n - (n-1) = n+1$$



# 三叉链表

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



```
typedef struct TriTNode
{
    TelemType data;
    struct TriTNode *lchild,*parent,*rchild;
}TriTNode,*TriTree;
```



## 5.5 遍历二叉树和线索二叉树

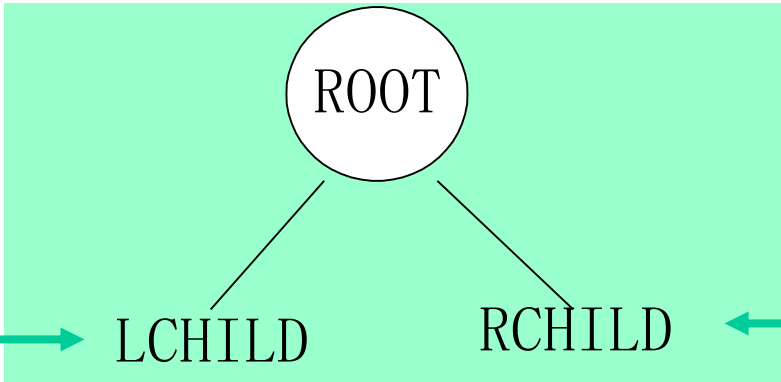


**遍历定义**——指按某条搜索路线遍访每个结点且不重复（又称周游）。实质是得到树中所有节点的一个线性排列。

**遍历用途**——它是树结构插入、删除、修改、查找和排序运算的前提，是二叉树一切运算的基础和核心。

# 遍历规则

D



先左后右

L



LCHILD

RCHILD

R

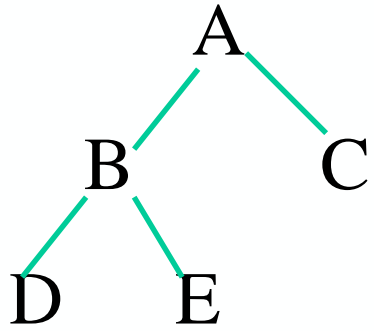


DLR LDR LRD

DRL

RDL

RLD



先序遍历: **A B D E C**

中序遍历: **D B E A C**

后序遍历: **D E B C A**

口诀:

DLR—先序遍历, 即先根再左再右

LDR—中序遍历, 即先左再根再右

LRD—后序遍历, 即先左再右再根

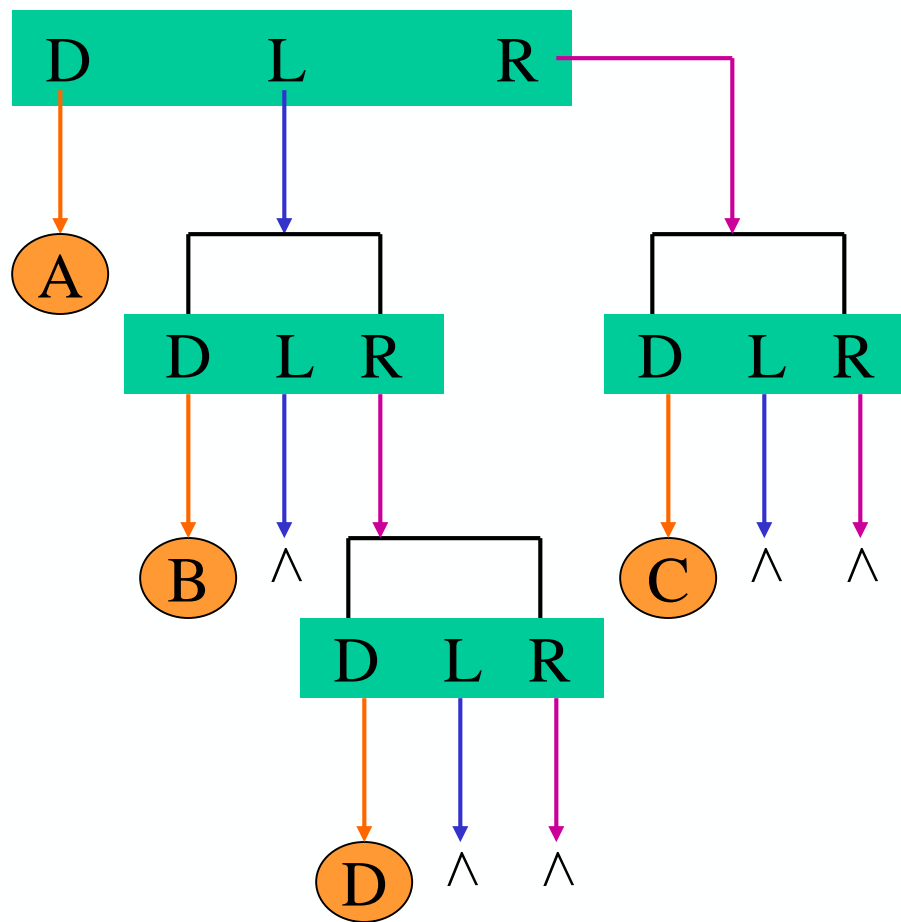
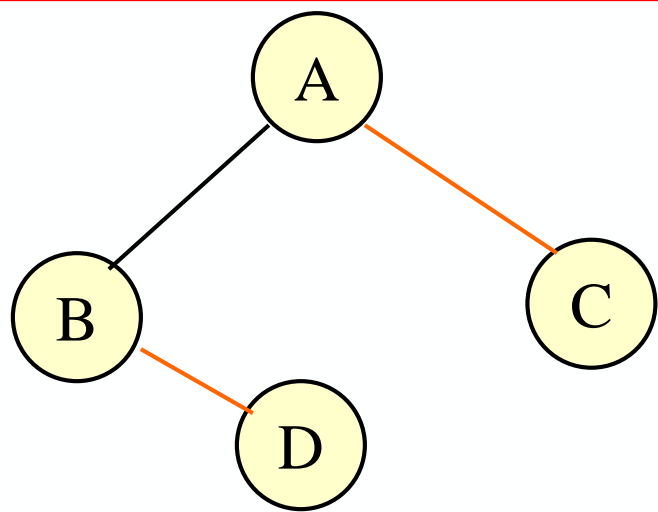
# 遍历的算法实现—先序遍历

若二叉树为空，则空操作  
否则

访问根结点 (D)

前序遍历左子树 (L)

前序遍历右子树 (R)



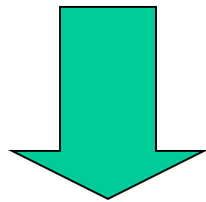
先序遍历序列: A B D C

# 遍历的算法实现——用递归形式格外简单！

回忆：

```
long Factorial ( long n ) {  
    if ( n == 0 ) return 1; //基本项  
    else return n * Factorial (n-1); //归纳项}
```

则三种遍历算法可写出：



# 先序遍历算法

```
Status PreOrderTraverse(BiTree T){  
    if(T==NULL) return OK; //空二叉树  
    else{  
        cout<<T->data; //访问根结点  
        PreOrderTraverse(T->lchild); //递归遍历左子树  
        PreOrderTraverse(T->rchild); //递归遍历右子树  
    }  
}
```

```

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

```

先序序列: **ABDC**

左是空返回

左是空返回  
右是空返回

左是空返回  
右是空返回

主程序

Pre( T )

T → A  
printf(A);  
pre(T → L);  
pre(T → R);

T → B  
printf(B);  
pre(T → L);  
pre(T → R);

T → C  
printf(C);  
pre(T → L);  
pre(T → R);

T → ^

返回

T → D

printf(D);  
pre(T → L);  
pre(T → R);

T → ^

返回

T → ^

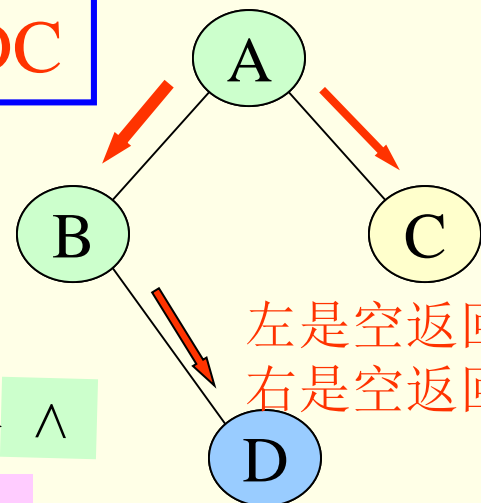
返回

T → ^

返回

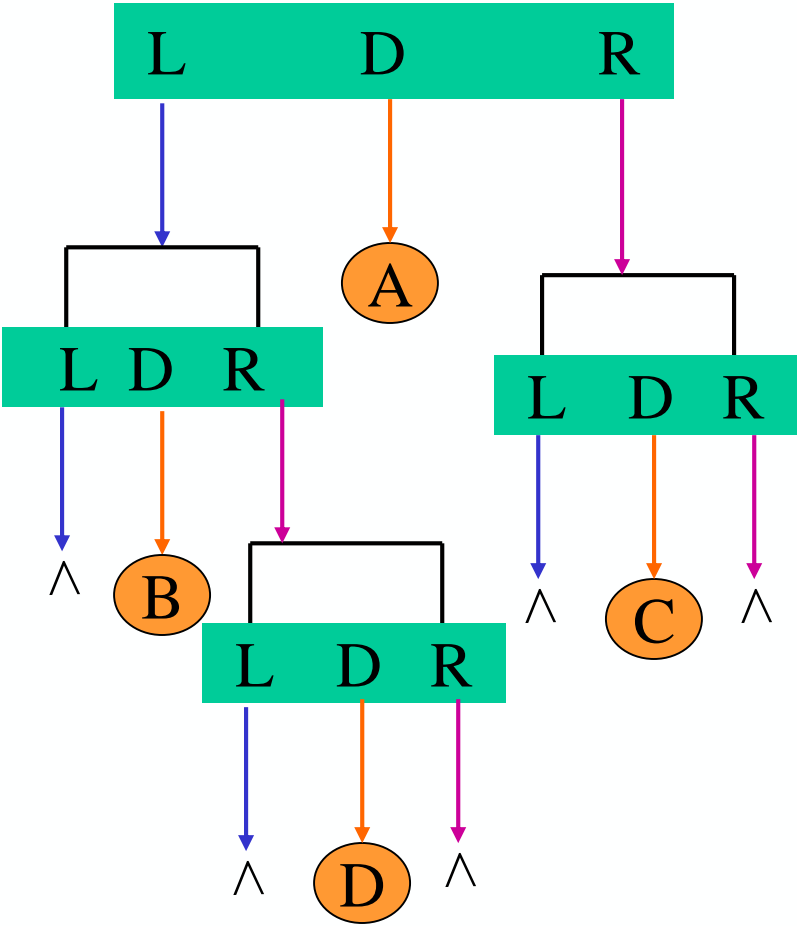
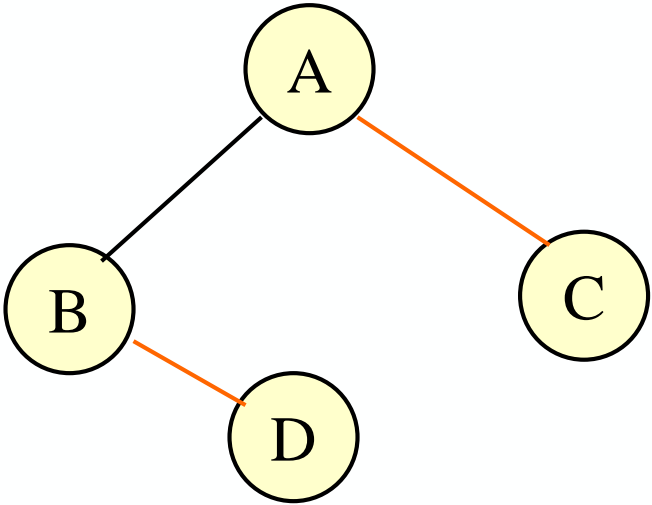
T → ^

返回



# 遍历的算法实现—中序遍历

若二叉树为空，则空操作  
否则：  
    中序遍历左子树 (L)  
    访问根结点 (D)  
    中序遍历右子树 (R)



中序遍历序列：B D A C



# 中序遍历算法

```
Status InOrderTraverse(BiTree T){  
    if(T==NULL) return OK; //空二叉树  
    else{  
        InOrderTraverse(T->lchild); //递归遍历左子树  
        cout<<T->data; //访问根结点  
        InOrderTraverse(T->rchild); //递归遍历右子树  
    }  
}
```

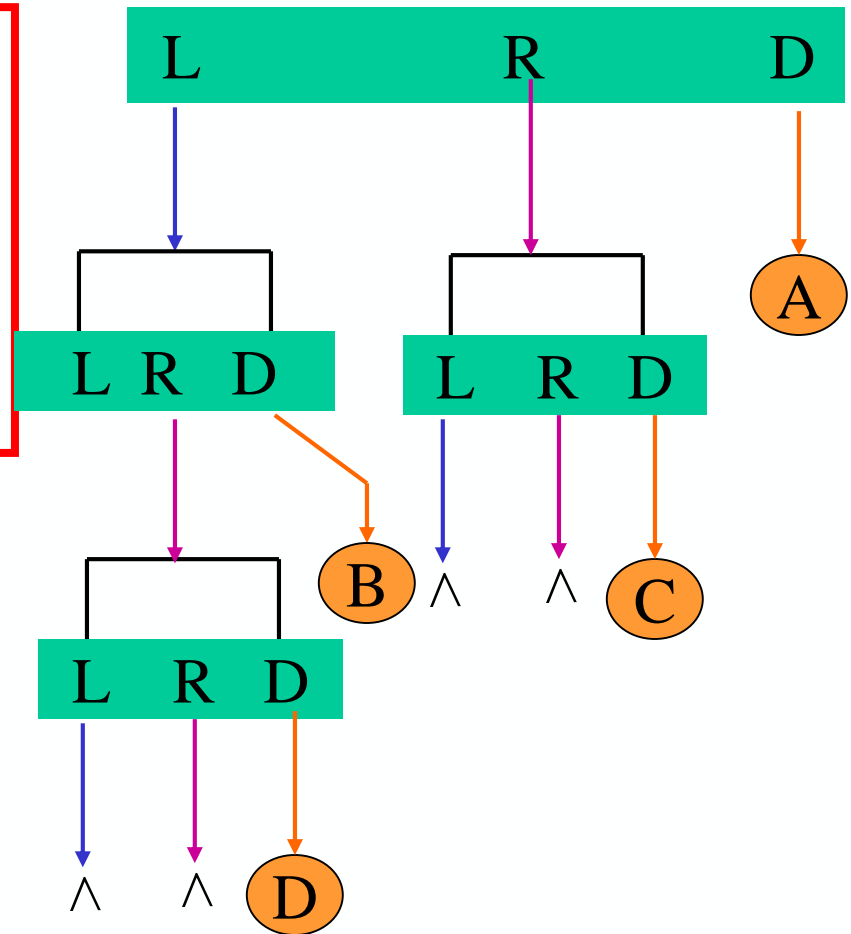
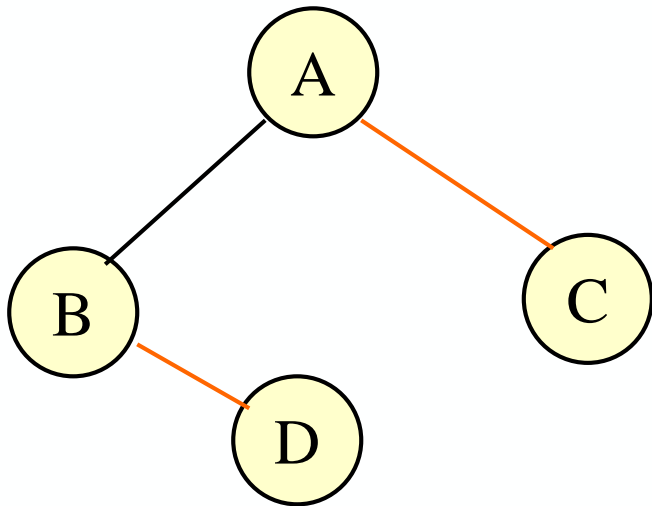
## 遍历的算法实现一后序遍历

若二叉树为空，则空操作  
否则

## 后序遍历左子树 (L)

## 后序遍历右子树 (R)

## 访问根结点 (D)

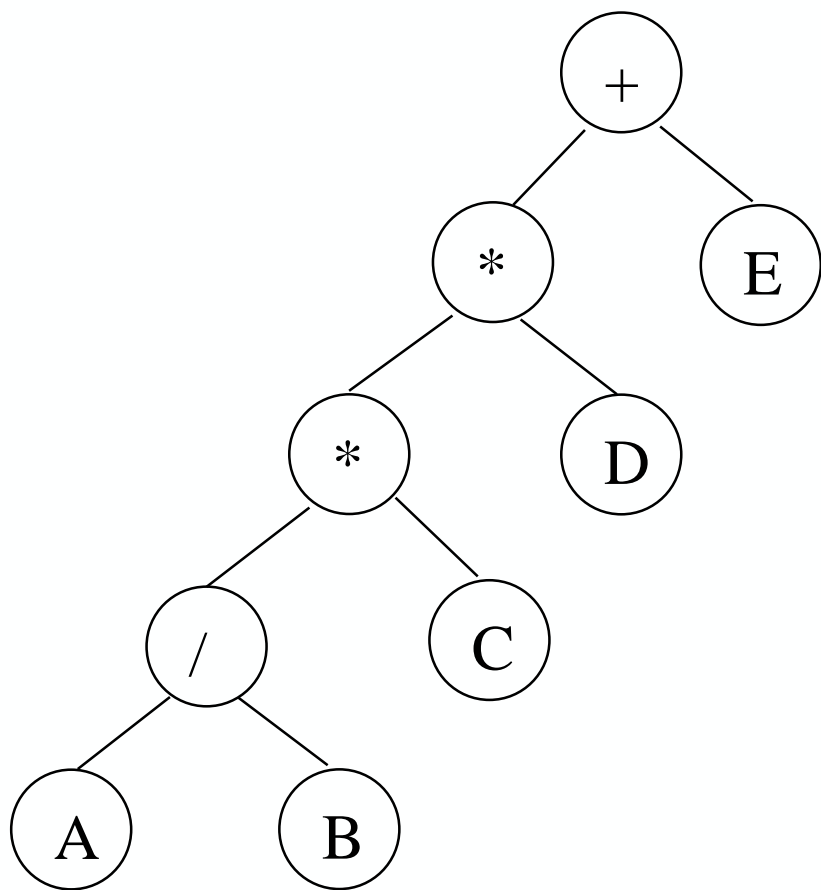


后序遍历序列: D B C A

# 后序遍历算法

```
Status PostOrderTraverse(BiTree T){  
    if(T==NULL) return OK; //空二叉树  
    else{  
        PostOrderTraverse(T->lchild); //递归遍历左子树  
        PostOrderTraverse(T->rchild); //递归遍历右子树  
        cout<<T->data; //访问根结点  
    }  
}
```

# 用二叉树表示算术表达式



先序遍历

$+ * * / A B C D E$

前缀表示

中序遍历

$A / B * C * D + E$

中缀表示

后序遍历

$A B / C * D * E +$

后缀表示

层序遍历

$+ * E * D / C A B$

# 遍历算法的分析

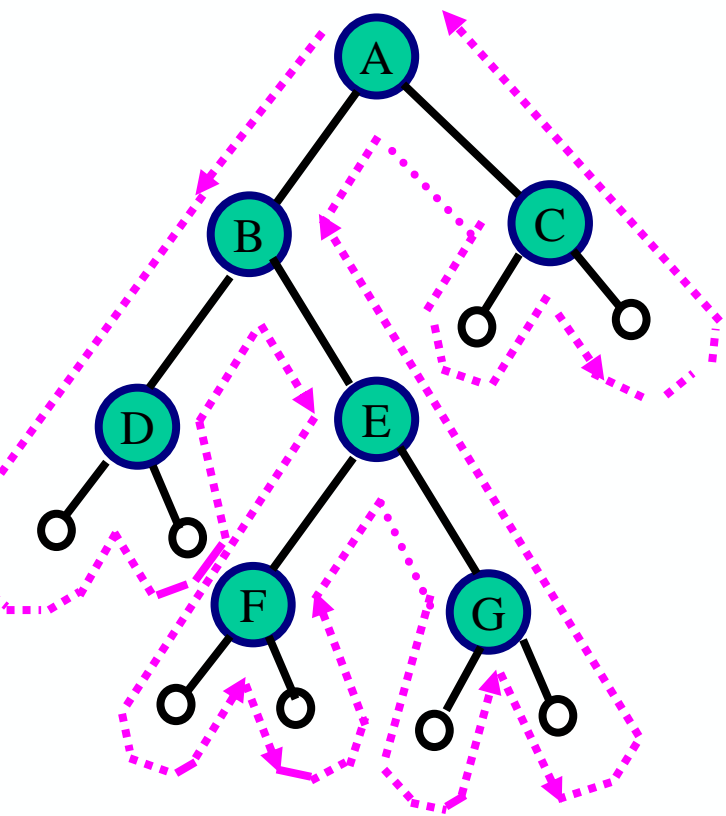
```
Status PreOrderTraverse(BiTree T){  
    if(T==NULL) return OK;  
    else{  
        cout<<T->data;  
        PreOrderTraverse(T->lchild);  
        PreOrderTraverse(T->rchild);  
    }  
}
```

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

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

# 遍历算法的分析

如果去掉输出语句，从递归的角度看，三种算法是完全相同的，或说这三种算法的访问路径是相同的，只是访问结点的时机不同。



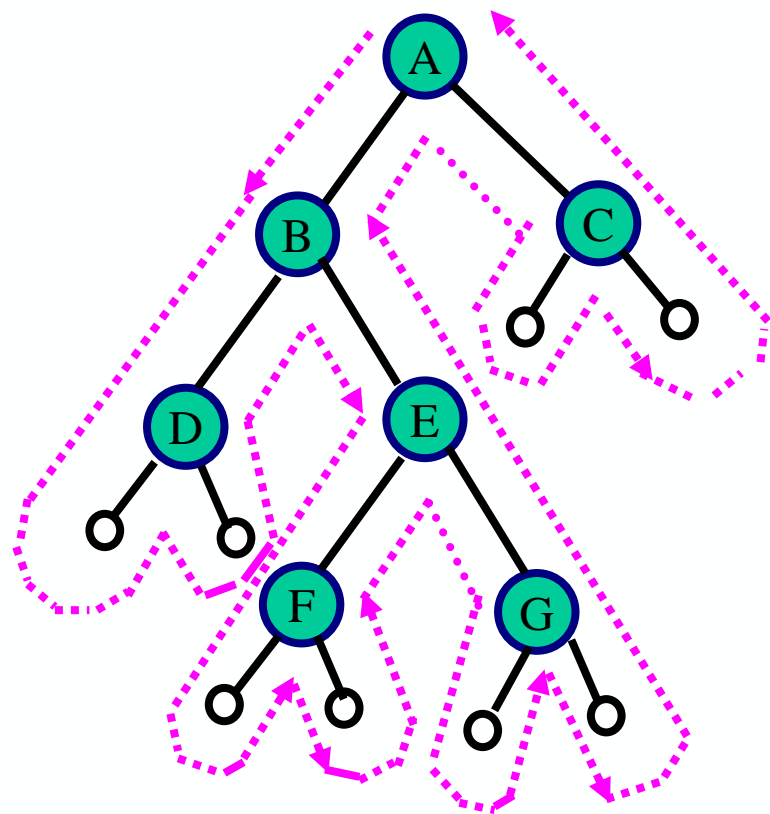
从虚线的出发点到终点的路径上，每个结点经过3次。

第1次经过时访问 = 先序遍历  
第2次经过时访问 = 中序遍历  
第3次经过时访问 = 后序遍历

# 遍历算法的分析

时间效率:  $O(n)$  // 每个结点只访问一次

空间效率:  $O(n)$  // 栈占用的最大辅助空间



# 重要结论

若二叉树中各结点的值均不相同，则：  
由二叉树的前序序列和中序序列，或由其后序序列和中序序列均**能唯一**地确定一棵二叉树，  
但由前序序列和后序序列却**不一定能唯一**地确定一棵二叉树。



# 练习

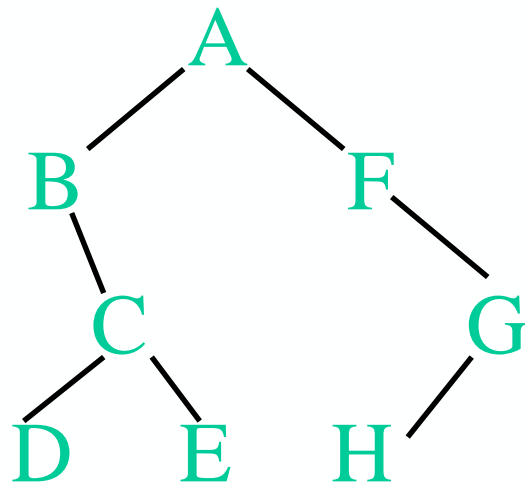
已知一棵二叉树的中序序列和后序序列分别是  
BDCEAFHG 和 DECBHGFA，请画出这棵二叉树。

- ①由后序遍历特征，根结点必在后序序列尾部（A）；
- ②由中序遍历特征，根结点必在其中间，而且其左部必全部是左子树子孙（BDCE），其右部必全部是右子树子孙（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。

---

中序遍历: B D C E A F H G

后序遍历: D E C B H G F A



( D C E )

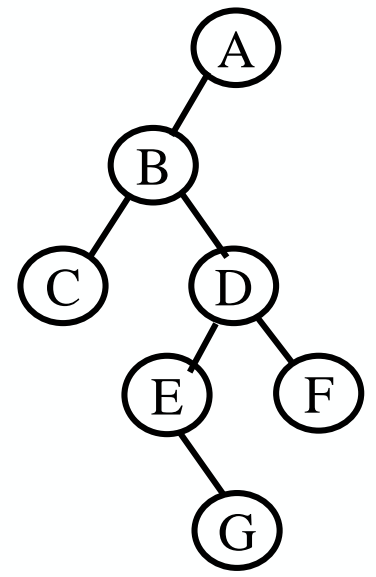
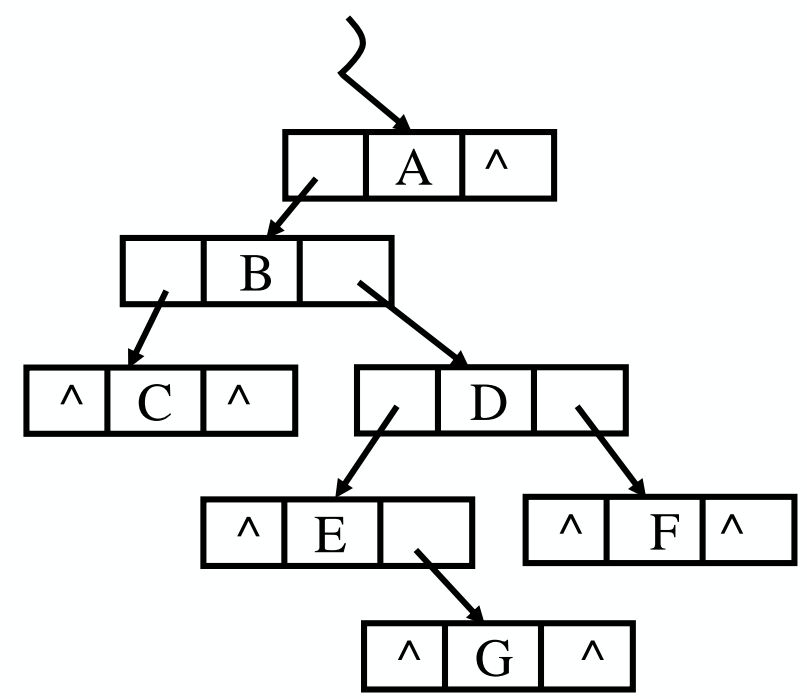
( H G )

# 二叉树的建立（算法5.3）

按先序遍历序列建立二叉树的二叉链表

例：已知先序序列为：

A B C  $\Phi$   $\Phi$  D E  $\Phi$  G  $\Phi$   $\Phi$  F  $\Phi$   $\Phi$   $\Phi$  （动态演示）



## 二叉树的建立（算法5.3）

```
void CreateBiTree(BiTree &T) {  
    cin>>ch;  
    if (ch=='#')    T=NULL;    //递归结束，建空树  
    else {  
        T=new BiTNode;    T->data=ch;  
        //生成根结点  
        CreateBiTree(T->lchild); //递归创建左子树  
        CreateBiTree(T->rchild); //递归创建右子树  
    }  
}
```

# 二叉树遍历算法的应用

## ✓ 计算二叉树深度

- 如果是空树，则深度为0；
- 否则，递归计算左子树的深度记为 $m$ ，递归计算右子树的深度记为 $n$ ，二叉树的深度则为 $m$ 与 $n$ 的较大者加1。

# 二叉树遍历算法的应用

## ✓ 计算二叉树结点总数

- 如果是空树，则结点个数为0；
- 否则，结点个数为左子树的结点个数+右子树的结点个数再+1。

### 算法5.6

```
int NodeCount(BiTree T) {  
    if(T == NULL ) return 0;  
    else return NodeCount(T-  
        >lchild)+NodeCount(T->rchild)+1;  
}
```

# 二叉树遍历算法的应用

## ✓ 计算二叉树叶子结点总数

- 如果是空树，则叶子结点个数为0；
- 否则，为左子树的叶子结点个数+右子树的叶子结点个数。
- 什么是叶子节点？（判断左孩子和右孩子是否为空）

```
int LeafCount(BiTree T) {  
    if (T==NULL)           //如果是空树返回0  
        return 0;  
    if (T->lchild == NULL && T->rchild == NULL)  
        return 1; //如果是叶子结点返回1  
    else return LeafCount(T->lchild) + LeafCount(T->rchild);  
}
```

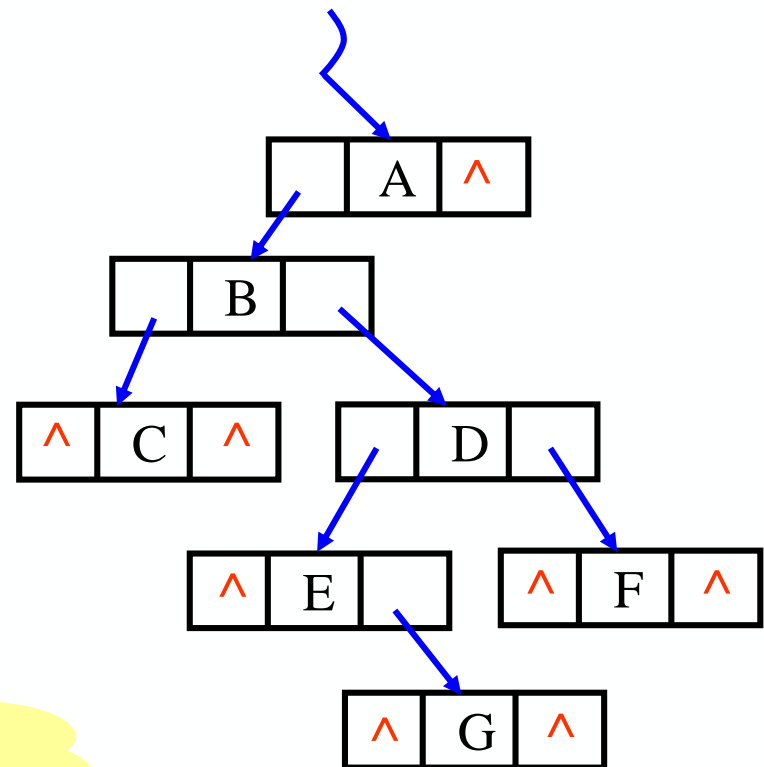
# 思考

在 $n$ 个结点的二叉链表中，有 $n+1$ 个空指针域

二叉链表空间效率这么低，能否利用这些空闲区存放有用的信息或线索？

——可以用它来存放当前结点的直接前驱和后继等线索，以加快查找速度。

线索化二叉树





# 线索化二叉树

普通二叉树只能找到结点的左右孩子信息，而该结点的直接前驱和直接后继只能在遍历过程中获得

若将遍历后对应的有关前驱和后继预存起来，则从第一个结点开始就能很快“顺藤摸瓜”而遍历整个树

可能是根、或最左（右）叶子

例如中序遍历结果：B D C E A F H G，实际上已将二叉树转为线性排列，显然具有唯一前驱和唯一后继！

# 线索化二叉树

如何保存这类信息？

两种解决方法 { 增加两个域：fwd和bwd;  
利用空链域（n+1个空链域）

# 线索化二叉树

- 1) 若结点有左子树，则lchild指向其左孩子；  
否则， lchild指向其直接前驱(即线索)；
- 2) 若结点有右子树，则rchild指向其右孩子；  
否则， rchild指向其直接后继(即线索)。

为了避免混淆，增加两个标志域

lchild	LTag	data	RTag	rchild
--------	------	------	------	--------

# 线索化二叉树

lchild	LTag	data	RTag	rchild
--------	------	------	------	--------

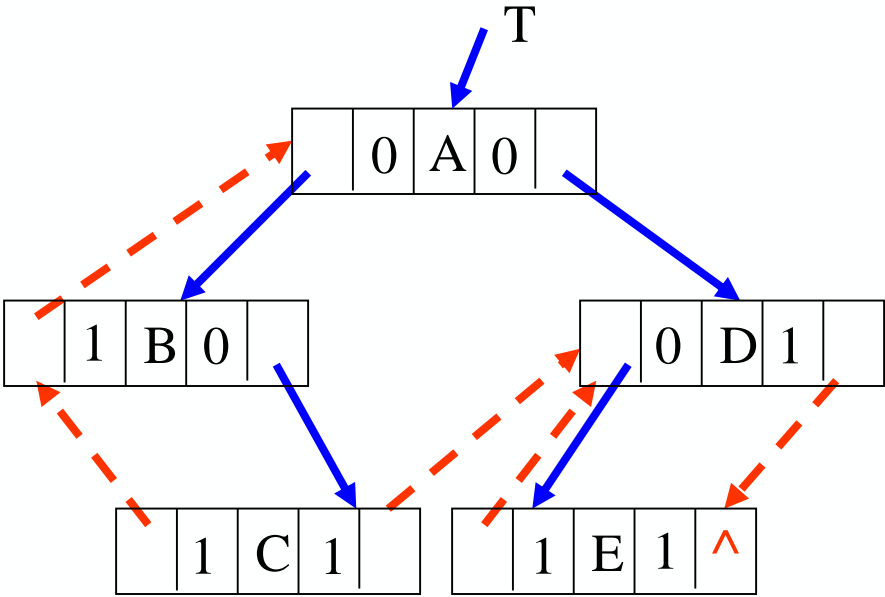
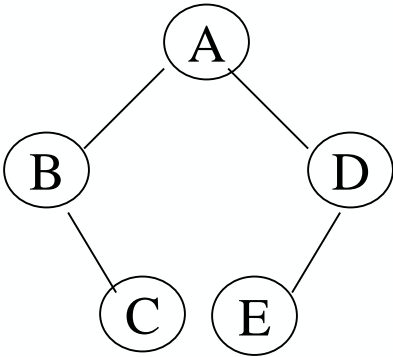
LTag : 若 LTag=0, lchild域指向左孩子;  
若 LTag=1, lchild域指向其前驱。

RTag : 若 RTag=0, rchild域指向右孩子;  
若 RTag=1, rchild域指向其后继。

# 先序线索二叉树

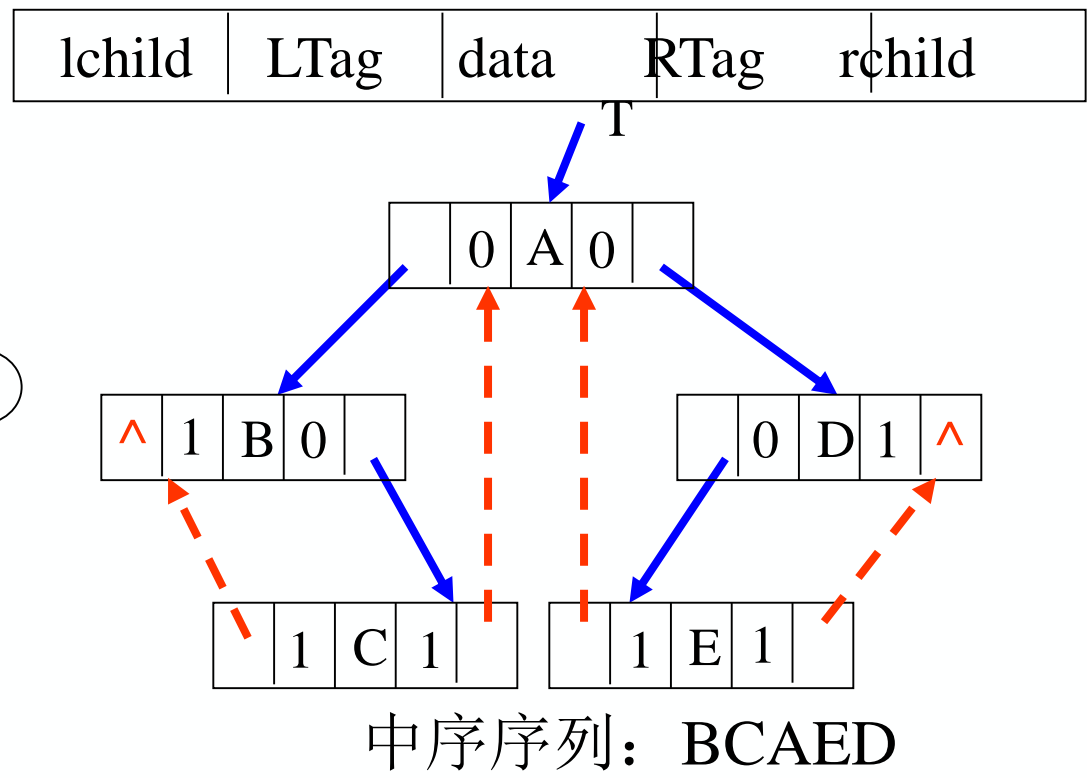
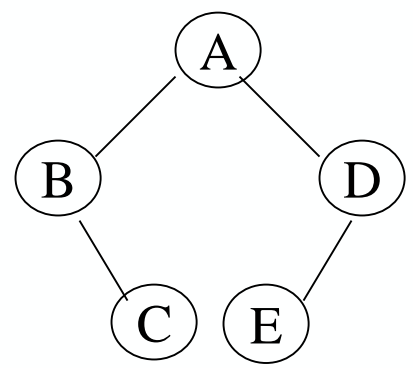
LTag=0, lchild域指向左孩子  
LTag=1, lchild域指向其前驱  
RTag=0, rchild域指向右孩子  
RTag=1, rchild域指向其后继

lchild	LTag	data	RTag	rchild
--------	------	------	------	--------



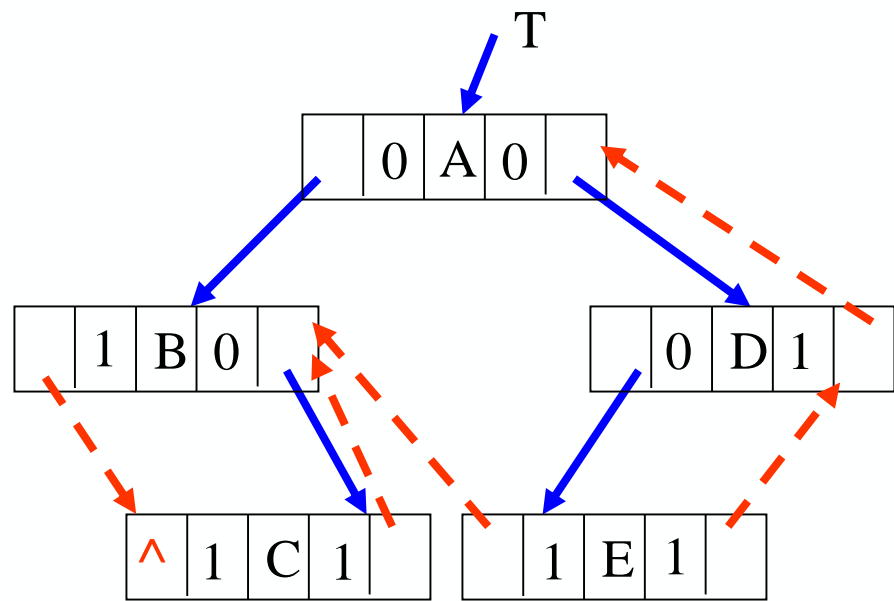
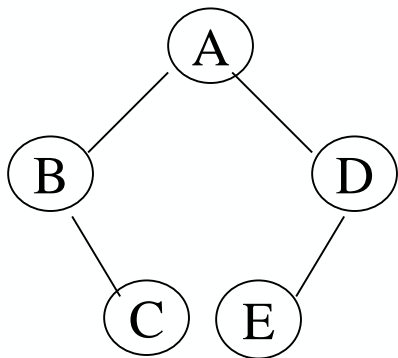
先序序列：ABCDE

# 中序线索二叉树



# 后序线索二叉树

lchild	LLag	data	RTag	rchild
--------	------	------	------	--------



后序序列：CBEDA

# 线索化二叉树的几个术语

**线索：**指向结点前驱和后继的指针

**线索链表：**加上线索二叉链表

**线索二叉树：**加上线索的二叉树（图形式样）

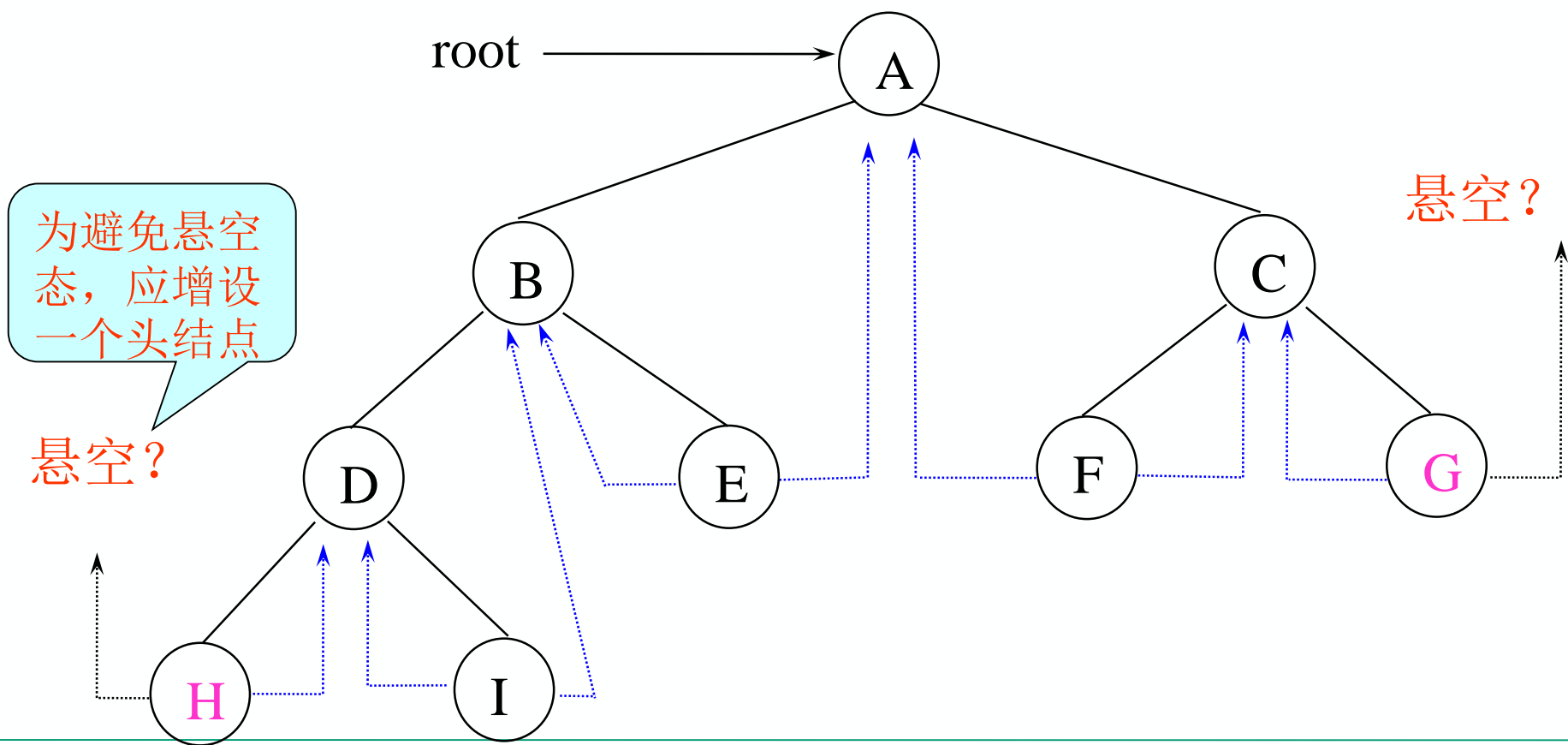
**线索化：**对二叉树以某种次序遍历使其变为线索二叉树的过程



# 练习

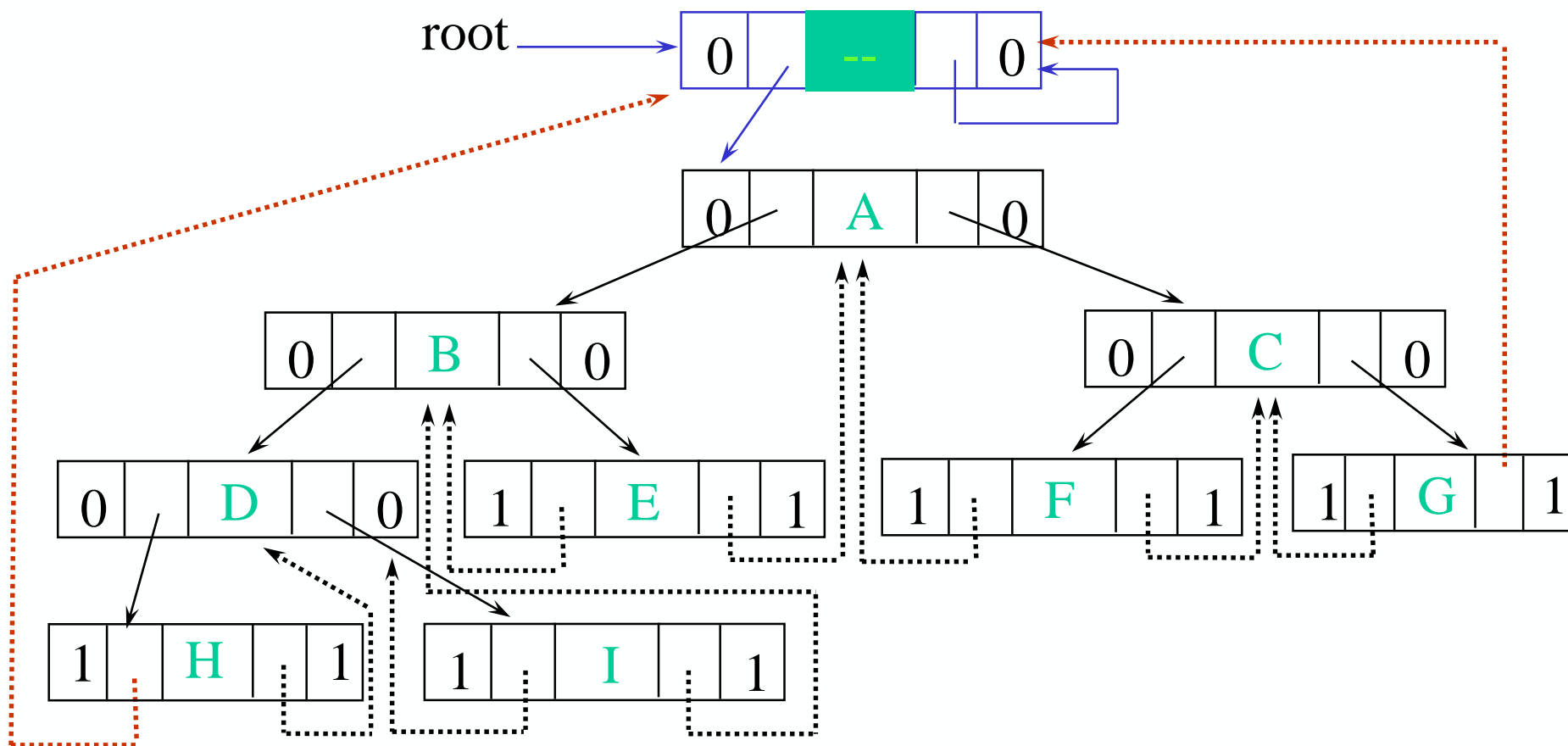
画出以下二叉树对应的中序线索二叉树。

该二叉树中序遍历结果为： H, D, I, B, E, A, F, C, G



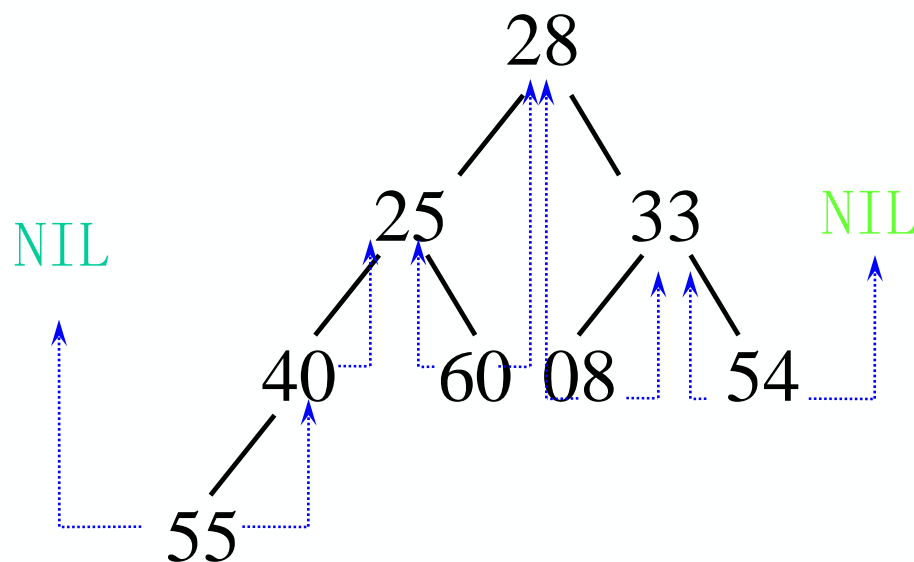
对应的中序线索二叉树存储结构如图所示：

注：此图中序遍历结果为： **H**, D, I, B, E, A, F, C, **G**



# 练习

画出与二叉树对应的中序线索二叉树



因为中序遍历序列是：55 40 25 60 28 08 33 54  
对应线索树应当按此规律连线，即在原二叉树中添加虚线。