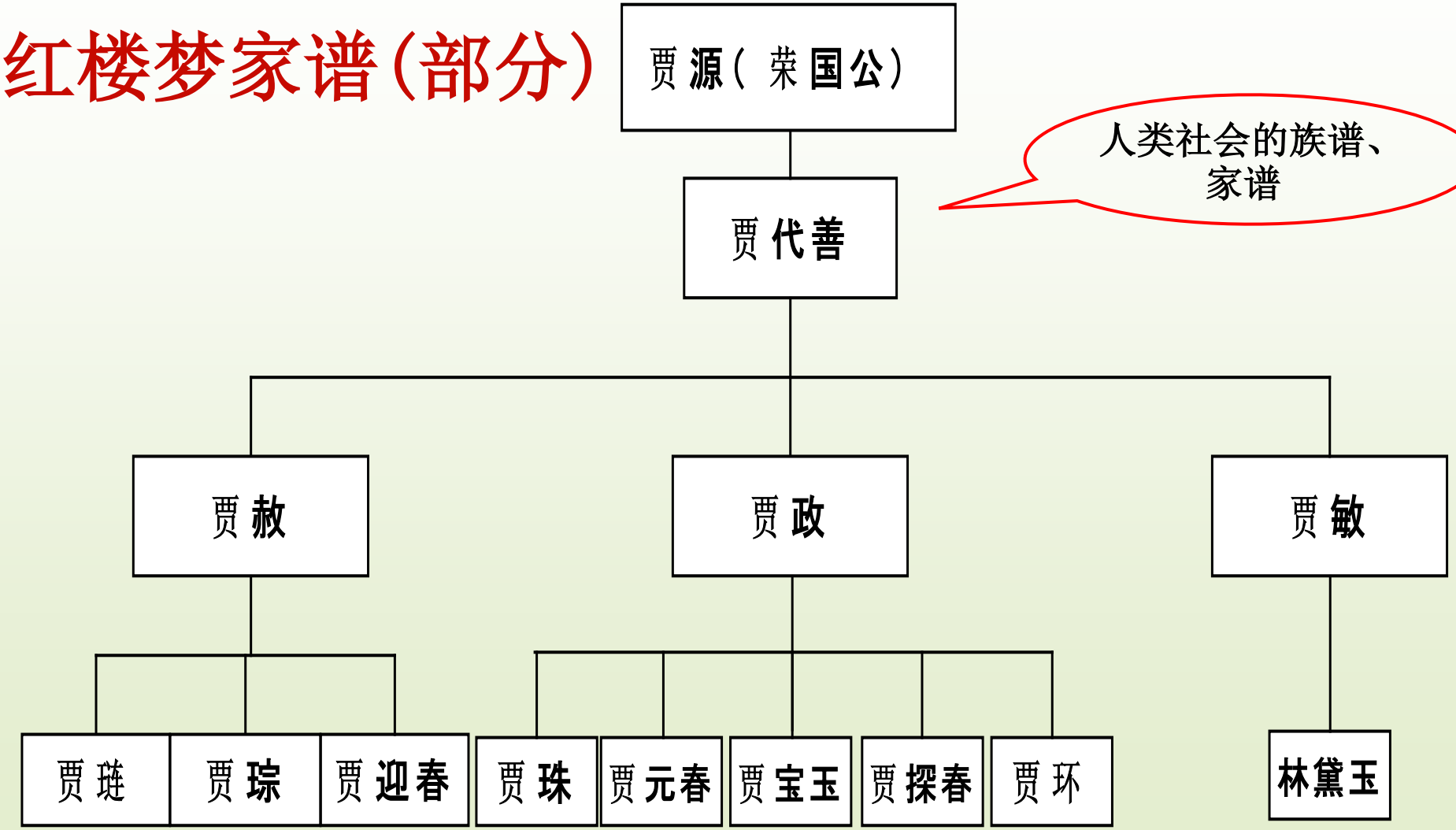
# 第 5 章　树和二叉树

树是一类重要的非线性数据结构，树形结构是以分支关系来定义的层次结构。在客观世界中树形结构广泛存在，并应用于:

- 人类社会的族谱、家谱、行政区域划分管理;
- 各种社会组织机构;
- 在计算机领域中，用树表示源程序的语法结构;
- 在操作系统（OS）中，文件系统、目录等组织结构也是用树来表示的。

# 第 5 章　树和二叉树

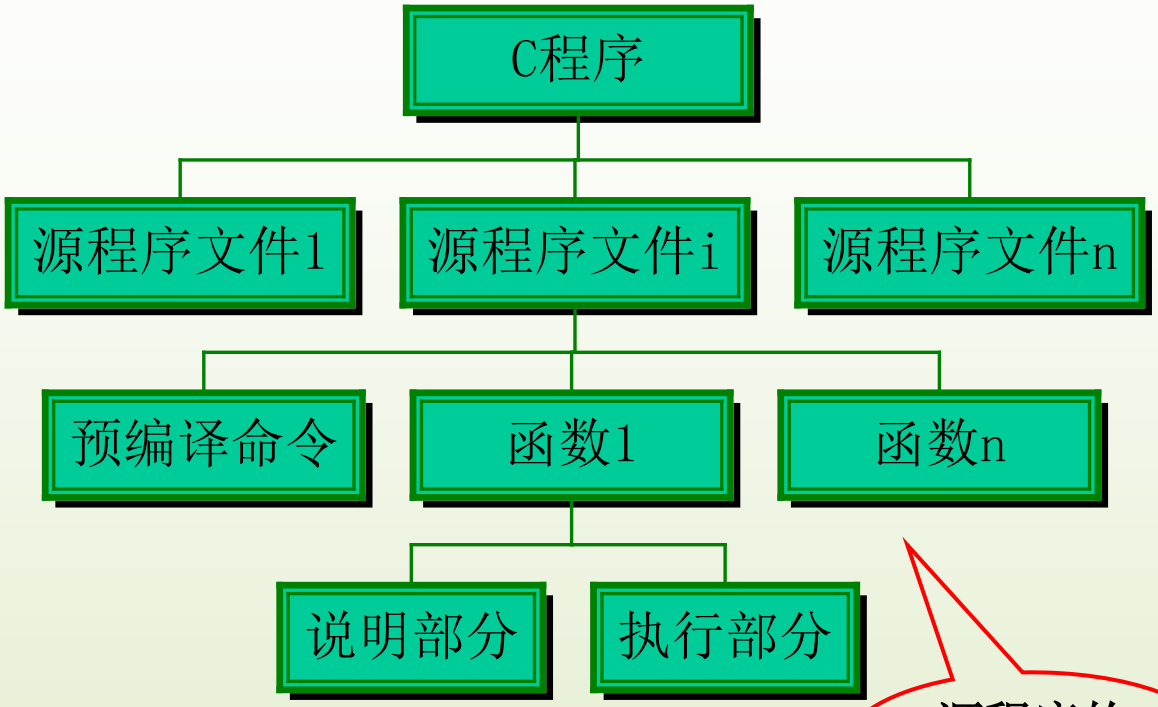
## 红楼梦家谱(部分)




人类社会的族谱、家谱

# 第 5 章　树和二叉树


行政组织机构图



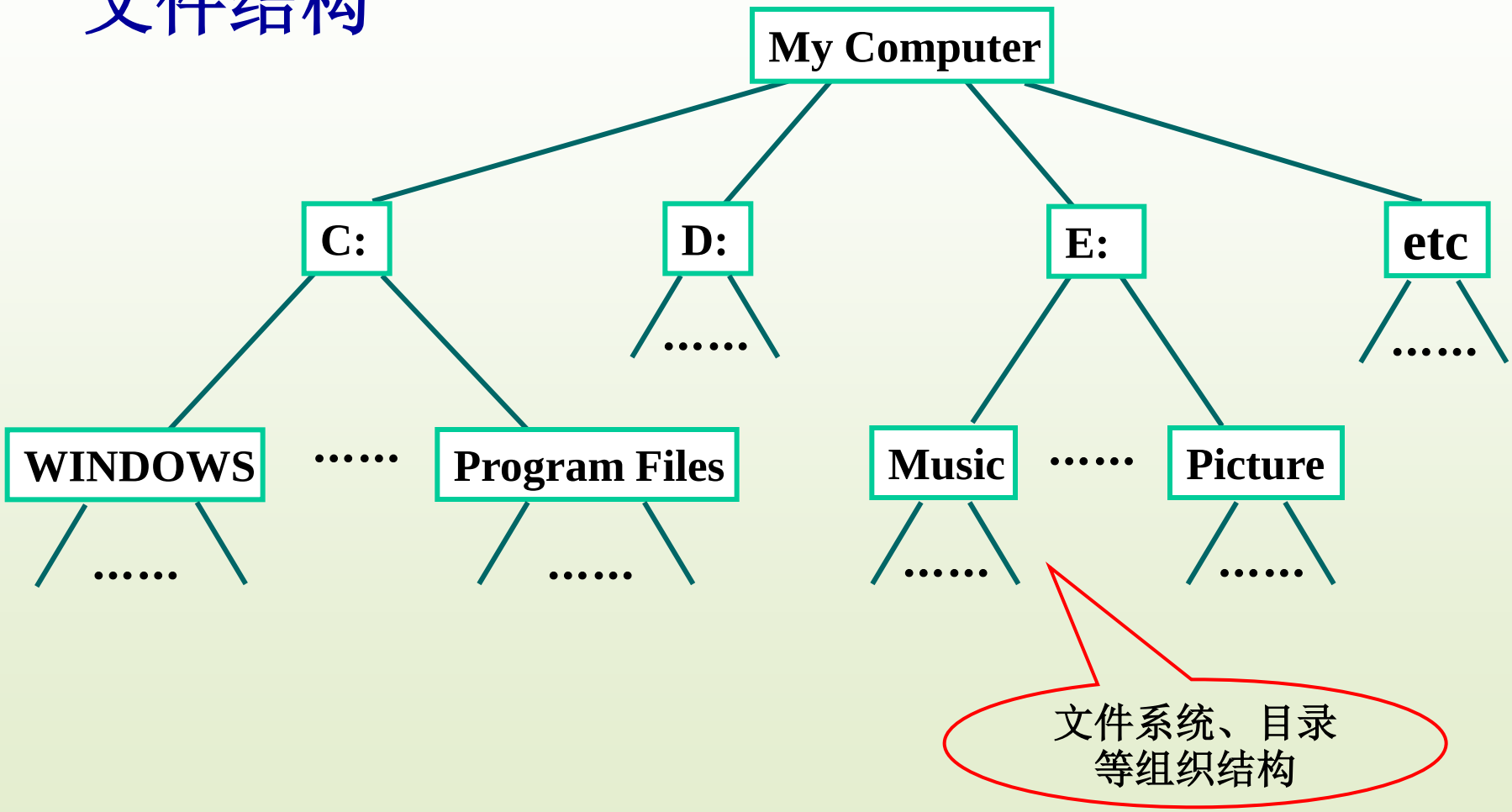

C程序结构

源程序的语法结构

社会组织机构

# 第 5 章　树和二叉树

## 文件结构



# 第 5 章　树和二叉树

## 本章的主要内容是

- 树的逻辑结构
- 树的存储结构
- 二叉树的逻辑结构
- 二叉树的存储结构及实现
- 树、森林与二叉树的转换
- 哈夫曼树

# 5.1 树的逻辑结构

## 树的定义

树：$n$（$n \geq 0$）个结点的有限集合。当$n=0$时，称为空树；任意一棵非空树满足以下条件：

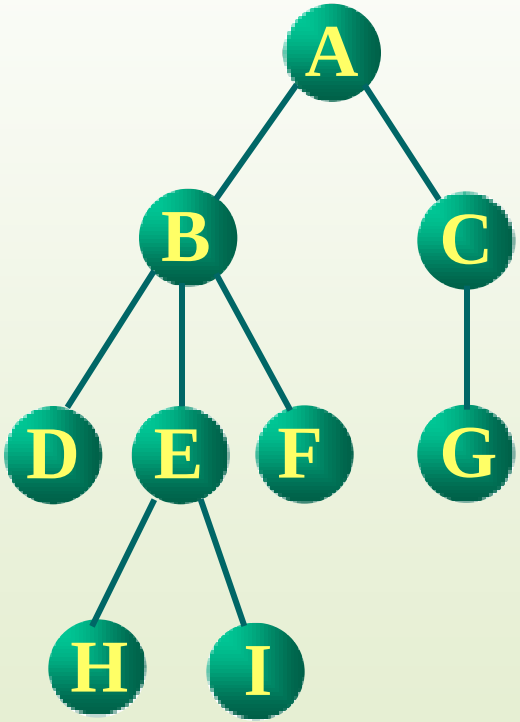
⑴ 有且仅有一个特定的称为根的结点；

⑵ 当$n>1$时，除根结点之外的其余结点被分成$m$（$m>0$）个互不相交的有限集合$T_1, T_2, \cdots, T_m$，其中每个集合又是一棵树，并称为这个根结点的子树。

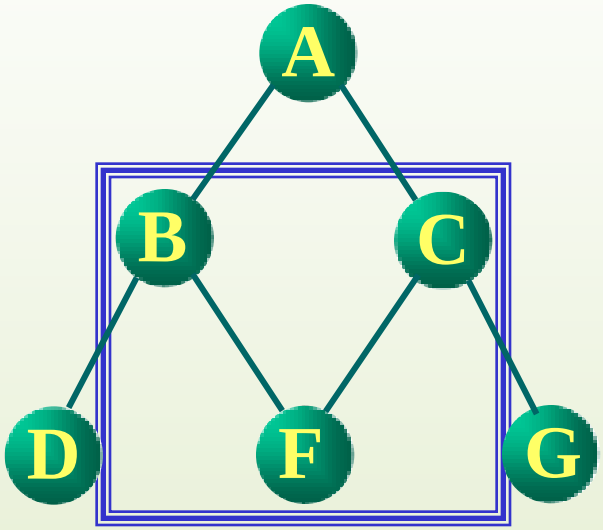
树的定义是采用递归方法

# 5.1 树的逻辑结构

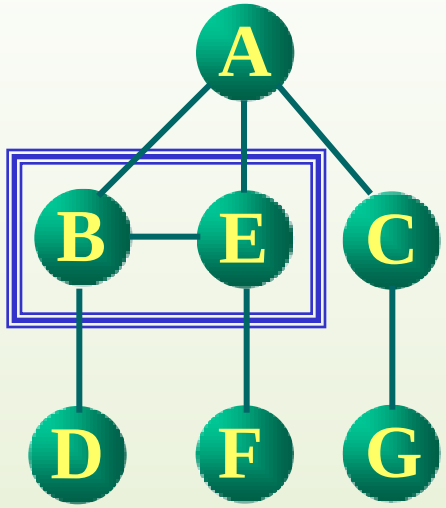
## 树的定义




(a) 一棵树结构




(b)一个非树结构



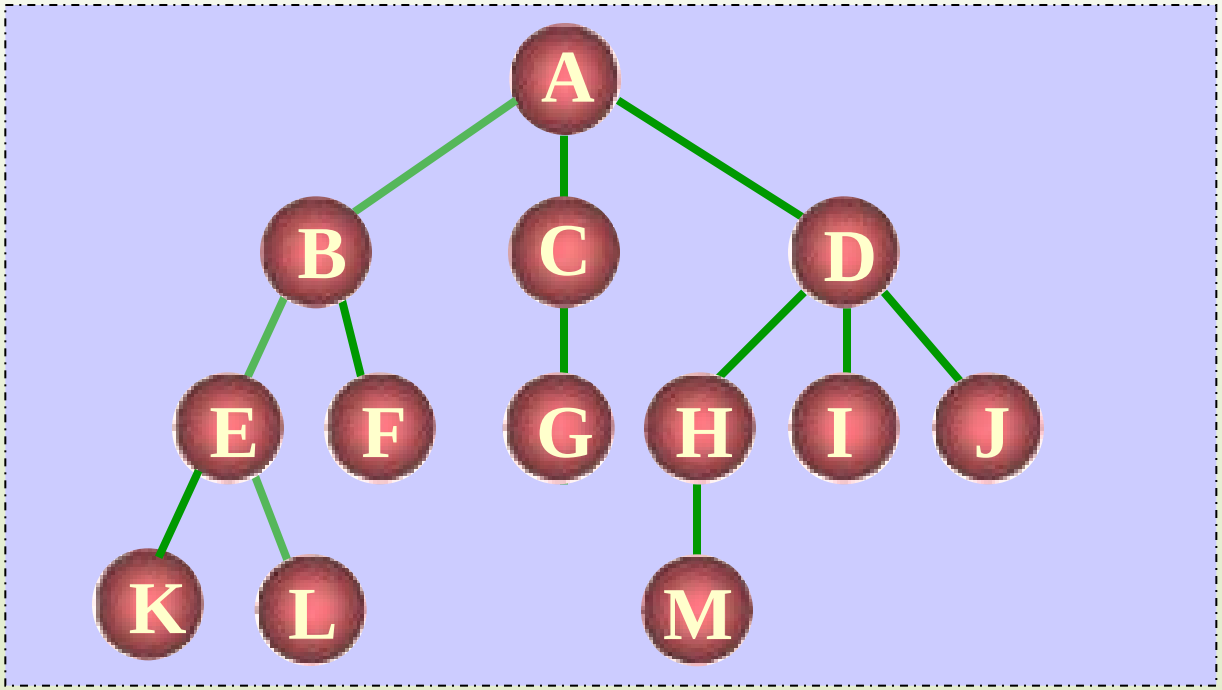

(c)一个非树结构

# 5.1 树的逻辑结构

## 树的基本术语

结点的度：结点所拥有的子树的个数。

树的度：树中各结点度的最大值。

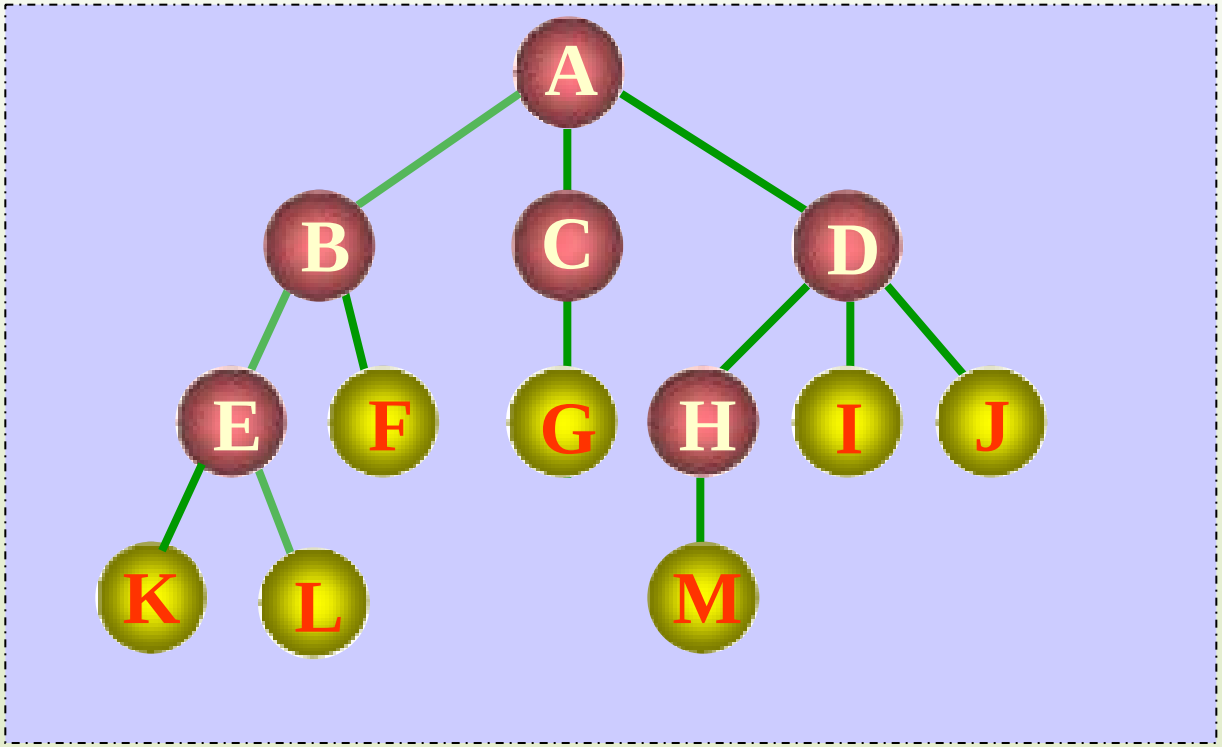


# 5.1 树的逻辑结构

## 树的基本术语

叶子结点：度为0的结点，也称为终端结点。

分支结点：度不为0的结点，也称为非终端结点。

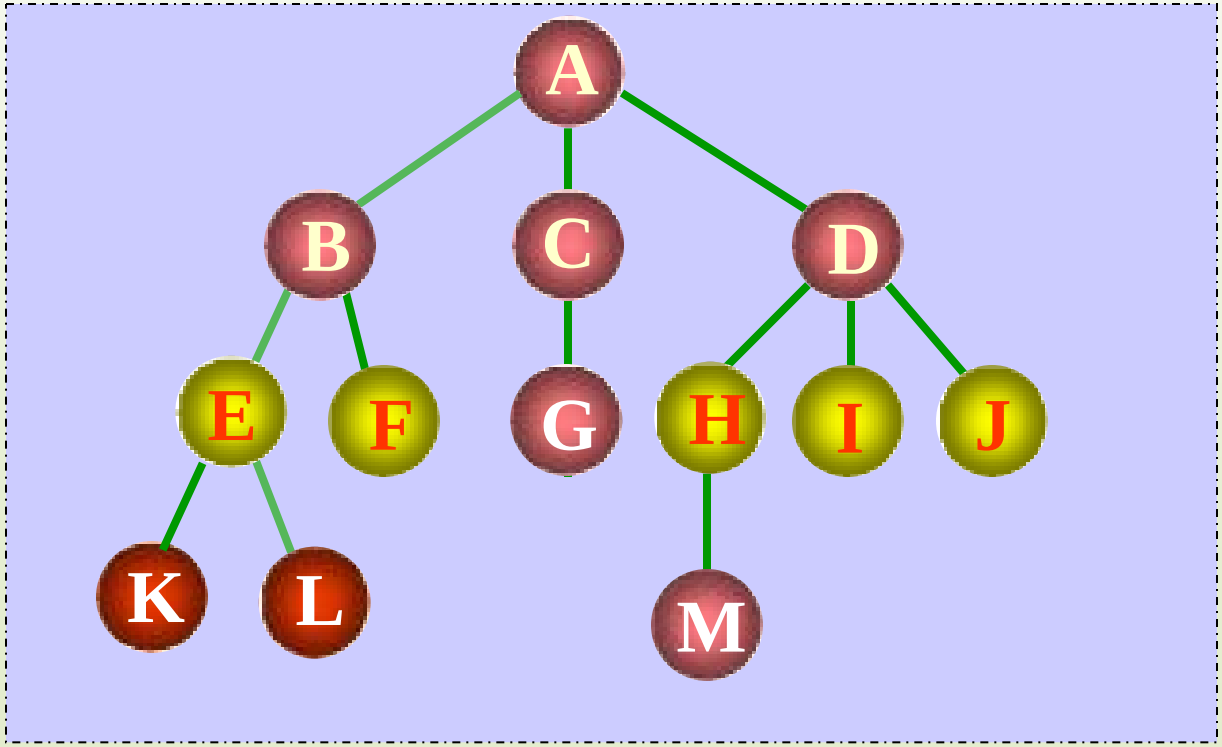


# 5.1 树的逻辑结构

## 树的基本术语

孩子、双亲：树中某结点子树的根结点称为这个结点的孩子结点，这个结点称为它孩子结点的双亲结点；

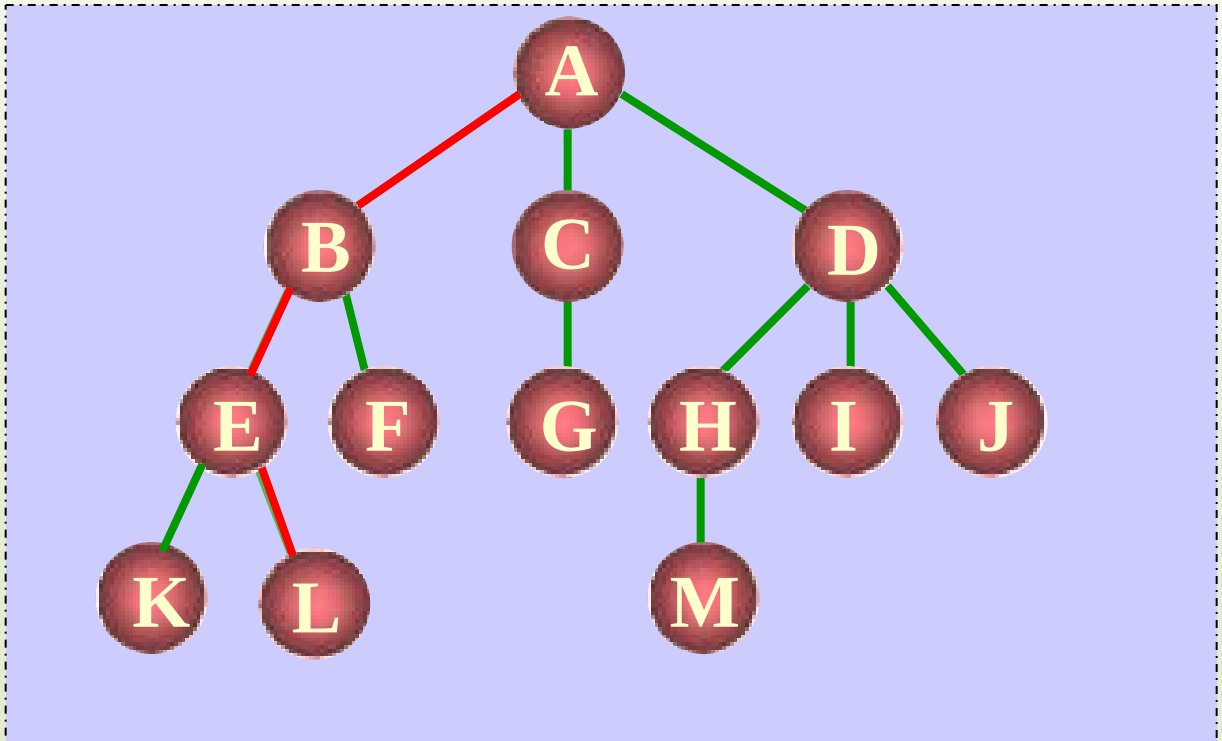
兄弟：具有同一个双亲的孩子结点互称为兄弟。



# 5.1 树的逻辑结构

## 树的基本术语

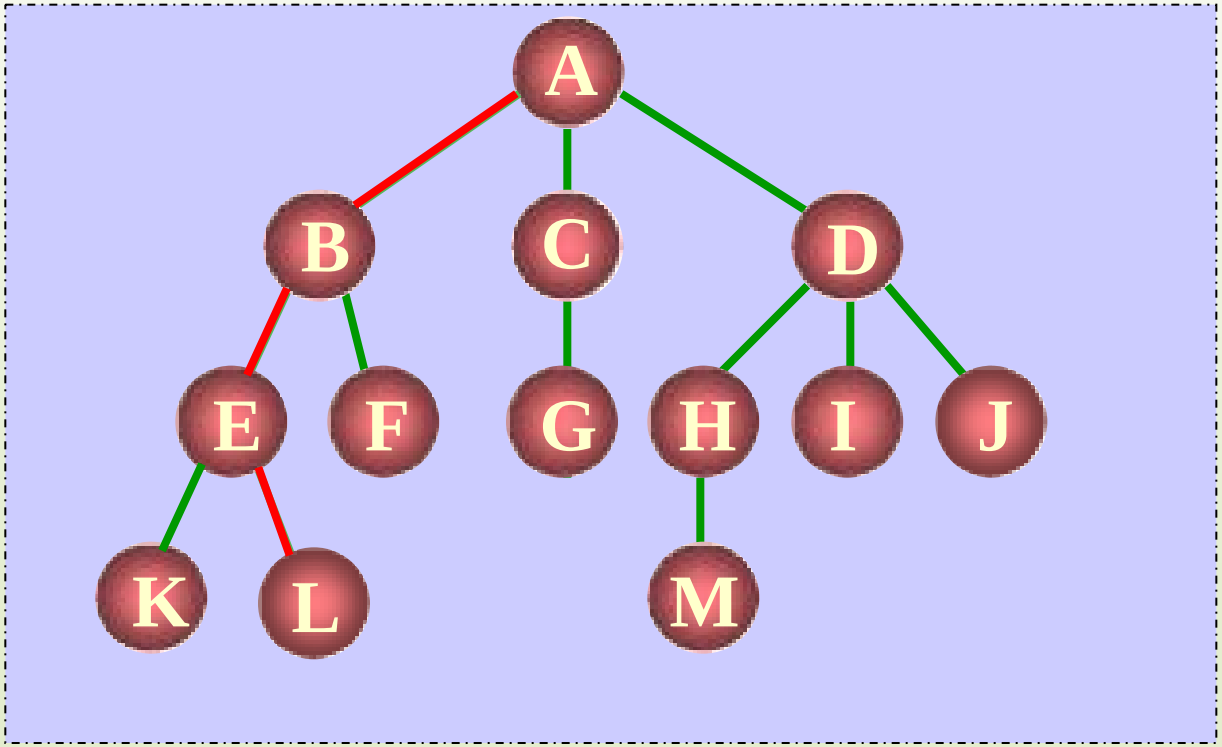
路径：如果树的结点序列$n_1, n_2, \ldots, n_k$有如下关系：结点$n_i$是$n_{i+1}$的双亲（$1<=i<k$），则把$n_1, n_2, \ldots, n_k$称为一条由$n_1$至$n_k$的路径；路径上经过的边的个数称为路径长度。



# 5.1 树的逻辑结构

## 树的基本术语

祖先、子孙：在树中，如果有一条路径从结点*x*到结点*y*，那么*x*就称为*y*的祖先，而*y*称为*x*的子孙。

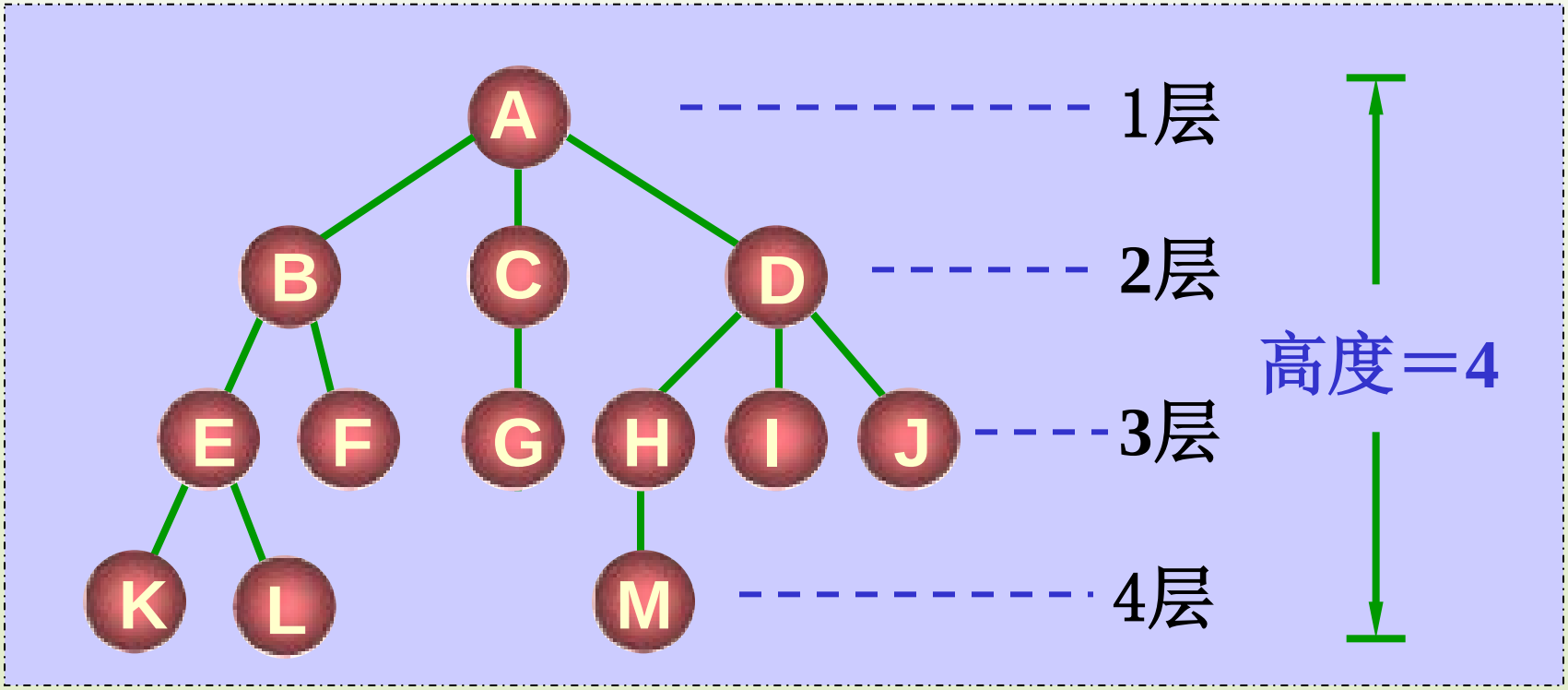


# 5.1 树的逻辑结构

## 树的基本术语

结点所在层数：根结点的层数为1；对其余任何结点，若某结点在第k层，则其孩子结点在第k+1层。

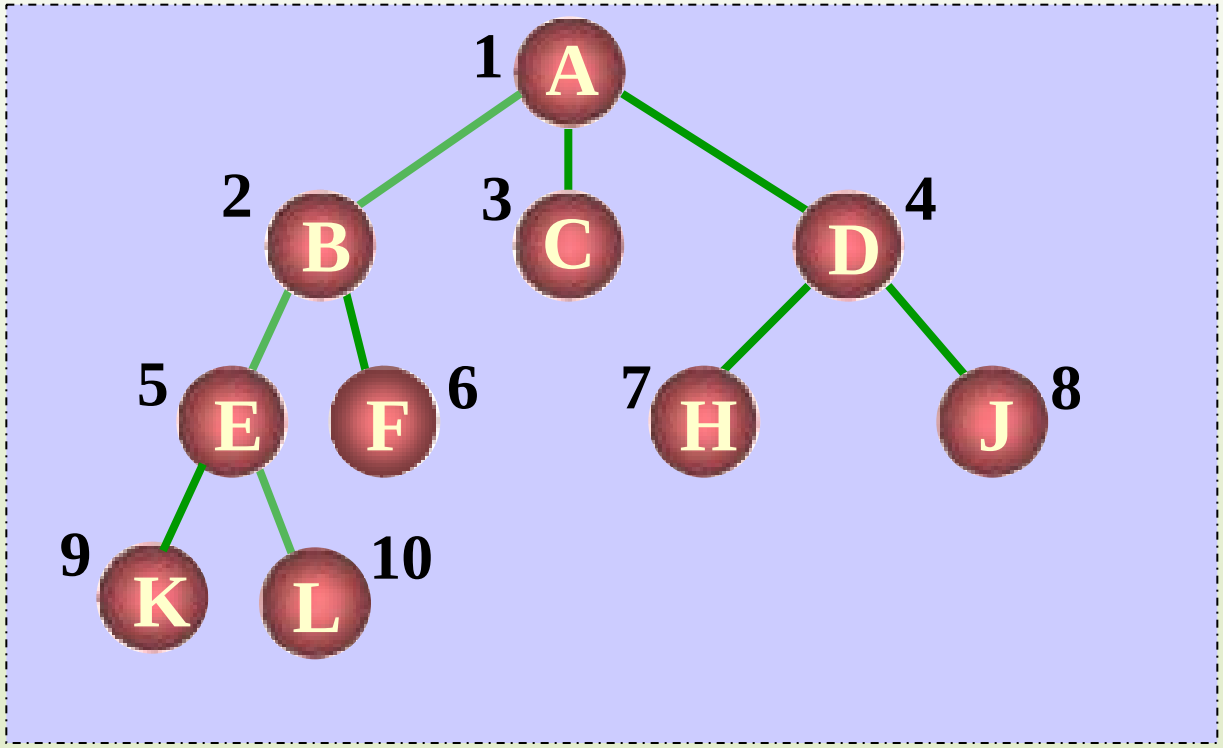
树的深度：树中所有结点的最大层数，也称高度。



# 5.1 树的逻辑结构

## 树的基本术语

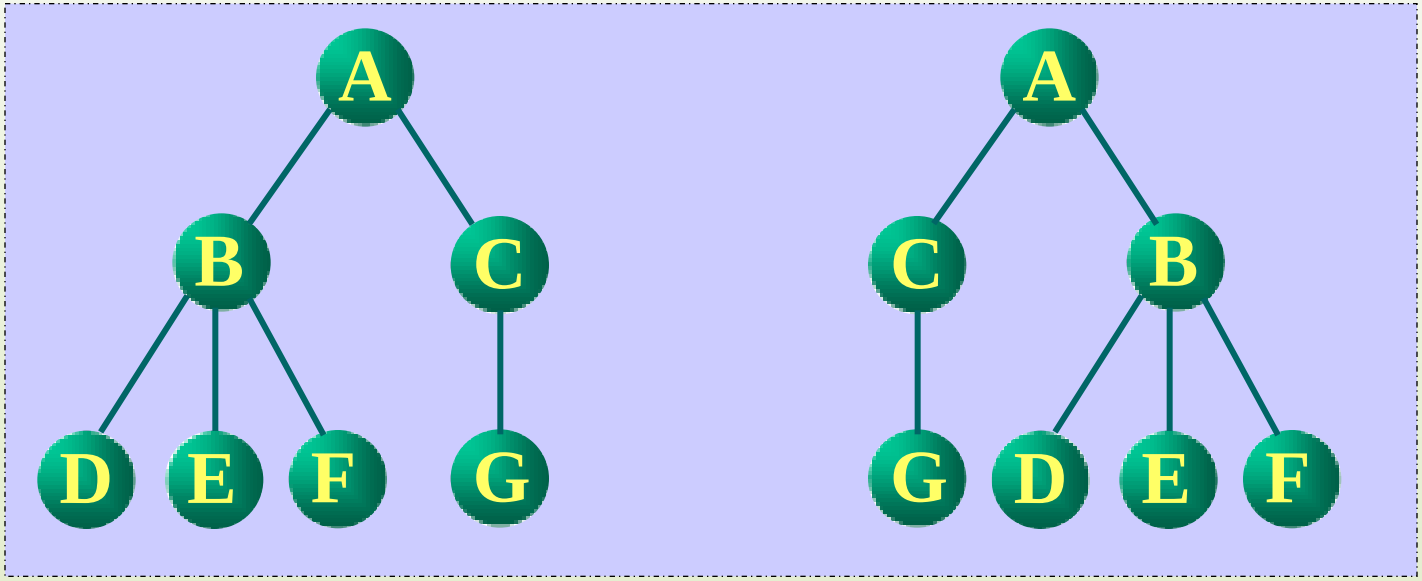
层序编号：将树中结点按照从上层到下层、同层从左到右的次序依次给他们编以从1开始的连续自然数。



# 5.1 树的逻辑结构

## 树的基本术语

有序树、无序树：如果一棵树中结点的各子树从左到右是有次序的，称这棵树为有序树；反之，称为无序树。

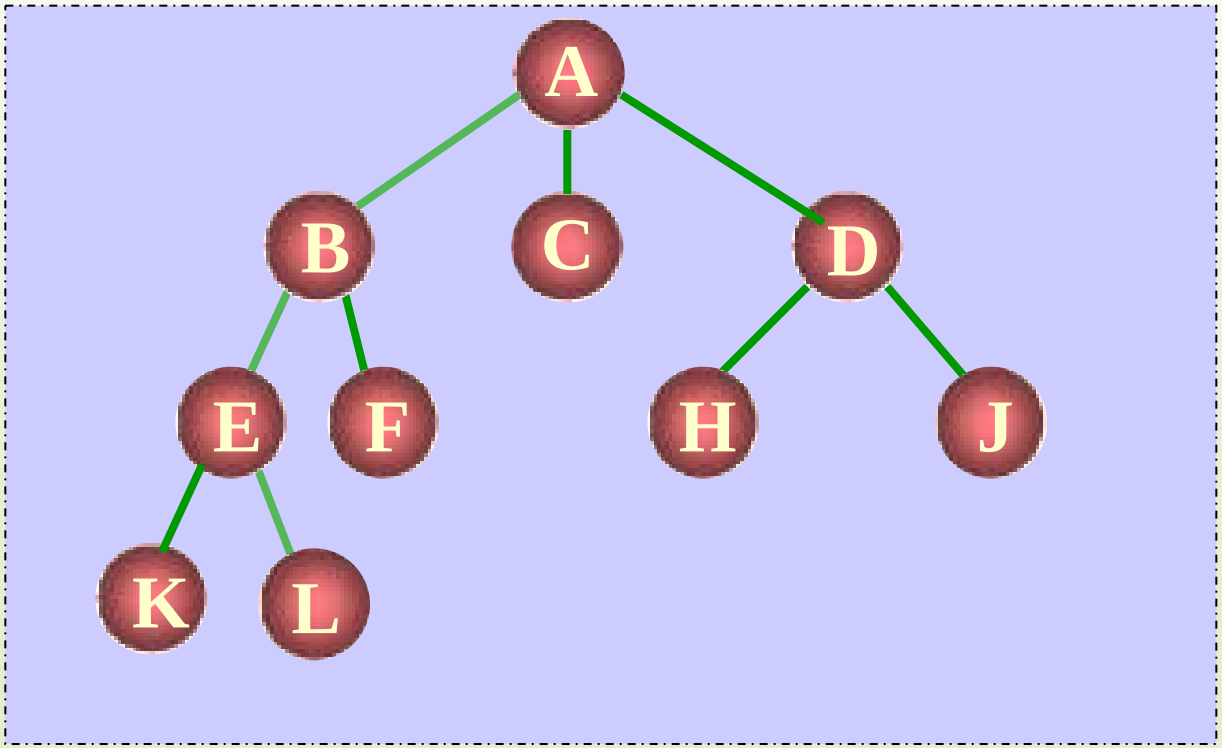



数据结构中讨论的一般都是有序树

# 5.1 树的逻辑结构

## 树的基本术语

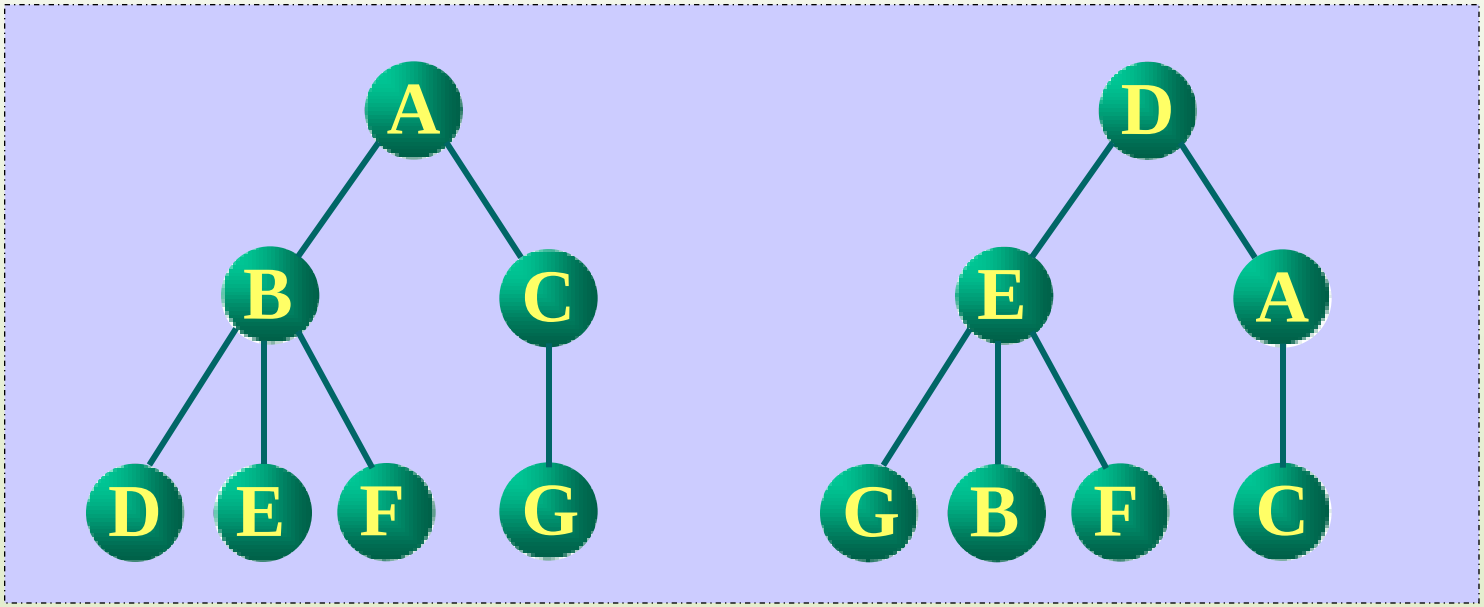
森林：$m$ ($m \geq 0$)棵互不相交的树的集合。



# 5.1 树的逻辑结构

## 树的基本术语

同构：对两棵树，若通过对结点适当地重命名，就可以使这两棵树完全相等（结点对应相等，结点对应关系也相等），则称这两棵树同构。



# 5.1 树的逻辑结构

## 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

根结点（只有一个）

无双亲

叶子结点(可以有多个)

无孩子

其它结点

一个双亲,多个孩子

一对多

# 5.1 树的逻辑结构

## 树的抽象数据类型定义

**ADT Tree**

**Data**

树是由一个根结点和若干棵子树构成，
树中结点具有相同数据类型及层次关系

**Operation**

**InitTree**

前置条件：树不存在
输入：无
功能：初始化一棵树
输出：无
后置条件：构造一个空树

# 5.1 树的逻辑结构

## 树的抽象数据类型定义

**DestroyTree**

前置条件：树已存在
输入：无
功能：销毁一棵树
输出：无
后置条件：释放该树占用的存储空间

**Root**

前置条件：树已存在
输入：无
功能：求树的根结点
输出：树的根结点的信息
后置条件：树保持不变

# 5.1 树的逻辑结构

## 树的抽象数据类型定义

**Parent**

前置条件：树已存在
输入：结点x
功能：求结点x的双亲
输出：结点x的双亲的信息
后置条件：树保持不变

**Depth**

前置条件：树已存在
输入：无
功能：求树的深度
输出：树的深度
后置条件：树保持不变

# 5.1 树的逻辑结构

## 树的抽象数据类型定义

**PreOrder**

前置条件：树已存在
输入：无
功能：前序遍历树
输出：树的前序遍历序列
后置条件：树保持不变

**PostOrder**

前置条件：树已存在
输入：无
功能：后序遍历树
输出：树的后序遍历序列
后置条件：树保持不变

**endADT**

# 5.1 树的逻辑结构

## 树的遍历操作

树的遍历：从根结点出发，按照某种次序访问树中所有结点，使得每个结点被访问一次且仅被访问一次。

如何理解访问？

抽象操作，可以是对结点进行的各种处理，这里简化为输出结点的数据。

遍历的实质？

树结构（非线性结构）→线性结构。

如何理解次序？

树通常有前序（根）遍历、后序（根）遍历和层序（次）遍历三种方式。

# 5.1 树的逻辑结构

## 前序遍历

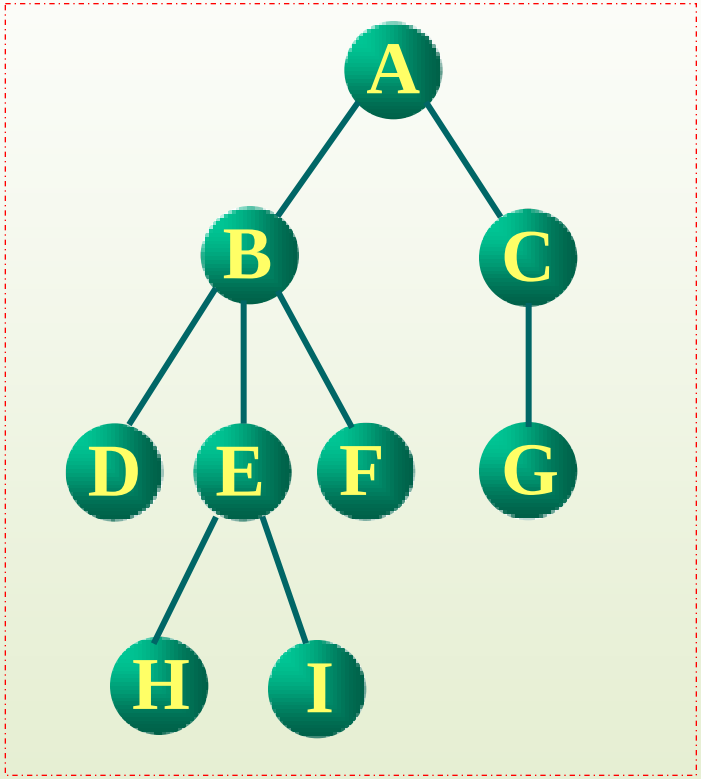
树的前序遍历操作定义为：

若树为空，则空操作返回；

否则

(1) 访问根结点；

(2) 按照从左到右的顺序前序遍历根结点的每一棵子树。




前序遍历序列：

*A B D E H I F C G*

# 5.1 树的逻辑结构

## 后序遍历

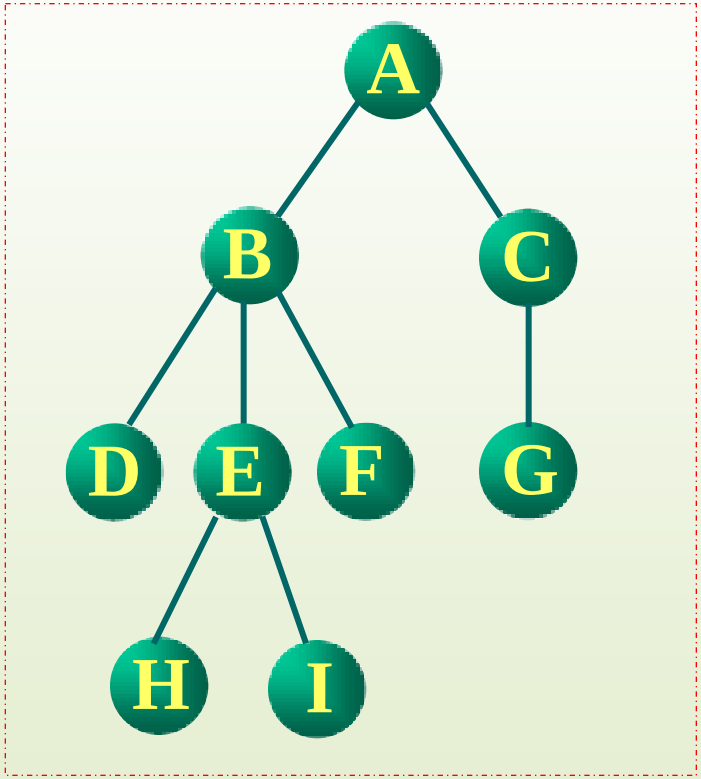
树的后序遍历操作定义为：

若树为空，则空操作返回；

否则

(1) 按照从左到右的顺序后序遍历根结点的每一棵子树；

(2) 访问根结点。



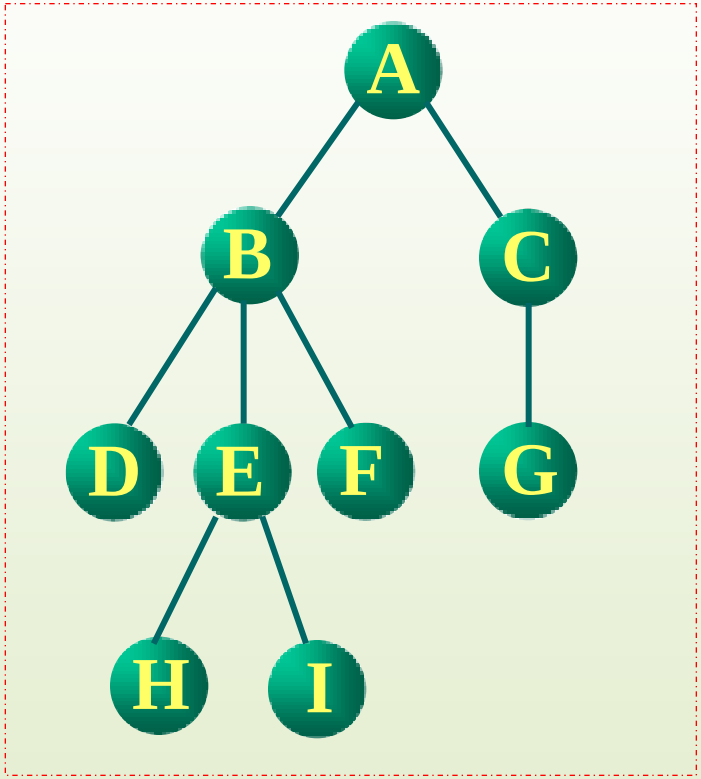

后序遍历序列：

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

# 5.1 树的逻辑结构

## 层序遍历

树的层序遍历操作定义为：从树的第一层（即根结点）开始，自上而下逐层遍历，在同一层中，按从左到右的顺序对结点逐个访问。




层序遍历序列：

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

# 5.2 树的存储结构

实现树的存储结构，关键是什么？

如何表示树中结点之间的逻辑关系。

什么是存储结构?

数据元素以及数据元素之间的逻辑关系在存储器中的表示。

树中结点之间的逻辑关系是什么？

思考问题的出发点：如何表示结点的双亲和孩子

# 5.2 树的存储结构

## 双亲表示法

基本思想：用一维数组来存储树的各个结点（一般按层序存储），数组中的一个元素对应树中的一个结点，包括结点的数据信息以及该结点的双亲在数组中的下标。

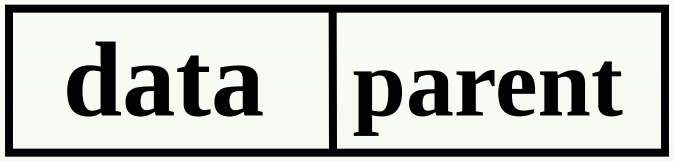
| data | parent |
|---|---|

**data**：存储树中结点的数据信息

**parent**：存储该结点的双亲在数组中的下标

# 5.2 树的存储结构

## 双亲表示法

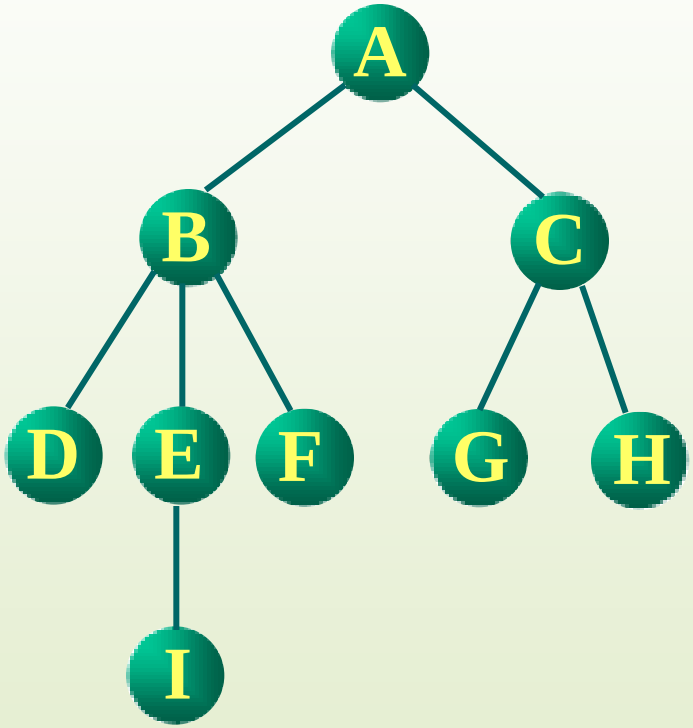



```
template <class T>
struct PNode
{
    T data;    //数据域
    int parent;  //指针域，双亲在数组中的下标
};
```

树的双亲表示法实质上是一个静态链表。

# 5.2 树的存储结构

## 双亲表示法

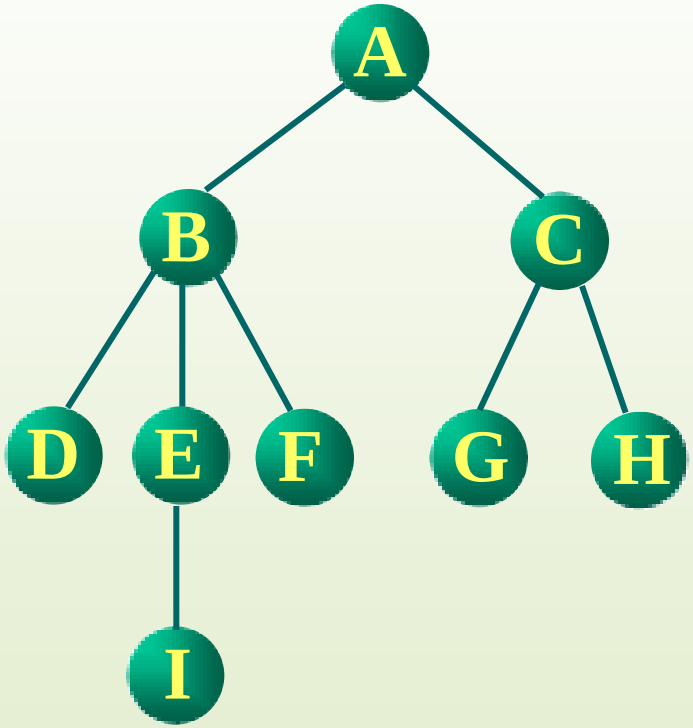



| 下标 | data | parent |
|---|---|---|
| 0 | A | −1 |
| 1 | B | 0 |
| 2 | C | 0 |
| 3 | D | 1 |
| 4 | E | 1 |
| 5 | F | 1 |
| 6 | G | 2 |
| 7 | H | 2 |
| 8 | I | 4 |

如何查找双亲结点？时间性能？

# 5.2 树的存储结构

## 双亲表示法

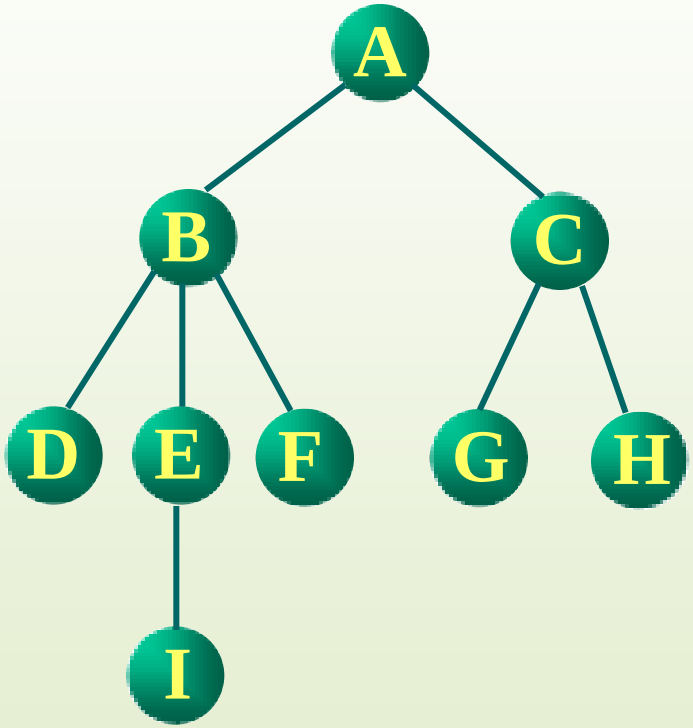



| 下标 | data | parent | firstchild |
|---|---|---|---|
| 0 | A | −1 | 1 |
| 1 | B | 0 | 3 |
| 2 | C | 0 | 6 |
| 3 | D | 1 | -1 |
| 4 | E | 1 | 8 |
| 5 | F | 1 | -1 |
| 6 | G | 2 | -1 |
| 7 | H | 2 | -1 |
| 8 | I | 4 | -1 |

如何查找孩子结点？时间性能？

# 5.2 树的存储结构

## 双亲表示法




| 下标 | data | parent | rightsib |
|---|---|---|---|
| 0 | A | −1 | −1 |
| 1 | B | 0 | 2 |
| 2 | C | 0 | −1 |
| 3 | D | 1 | 4 |
| 4 | E | 1 | 5 |
| 5 | F | 1 | −1 |
| 6 | G | 2 | 7 |
| 7 | H | 2 | −1 |
| 8 | I | 4 | −1 |

如何查找兄弟结点？时间性能？

# 5.2 树的存储结构

## 孩子表示法——多重链表表示法

如何确定链表中的结点结构？

链表中的每个结点包括一个数据域和多个指针域，每个指针域指向该结点的一个孩子结点。

方案一：指针域的个数等于树的度

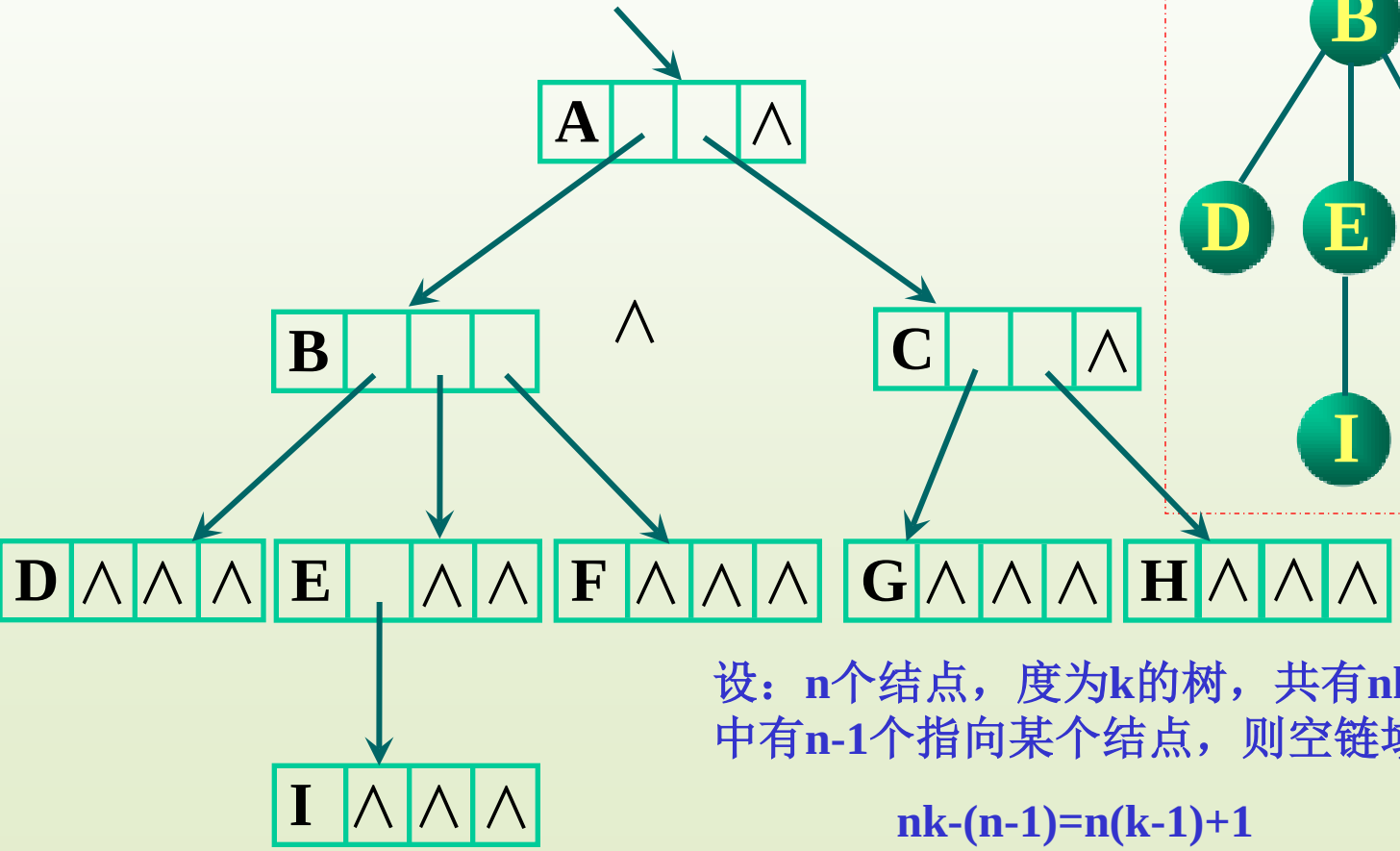
| data | child1 | child2 | …… | childd |
|---|---|---|---|---|

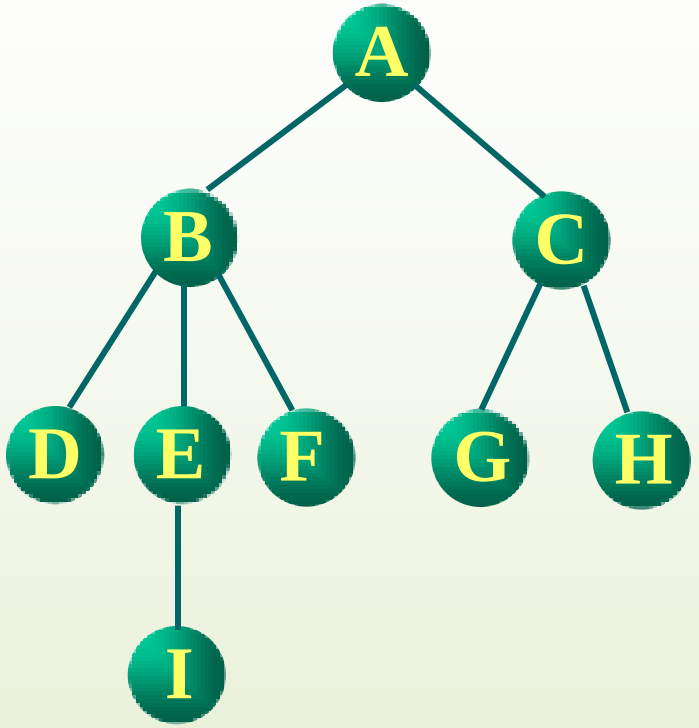
其中：data：数据域，存放该结点的数据信息；

child1~childd：指针域，指向该结点的孩子。

# 5.2 树的存储结构

缺点：浪费空间




设：n个结点，度为k的树，共有nk个链域，其中有n-1个指向某个结点，则空链域个数：

$$nk-(n-1)=n(k-1)+1$$

# 5.2 树的存储结构

## 孩子表示法——多重链表表示法

如何确定链表中的结点结构？

链表中的每个结点包括一个数据域和多个指针域，每个指针域指向该结点的一个孩子结点。

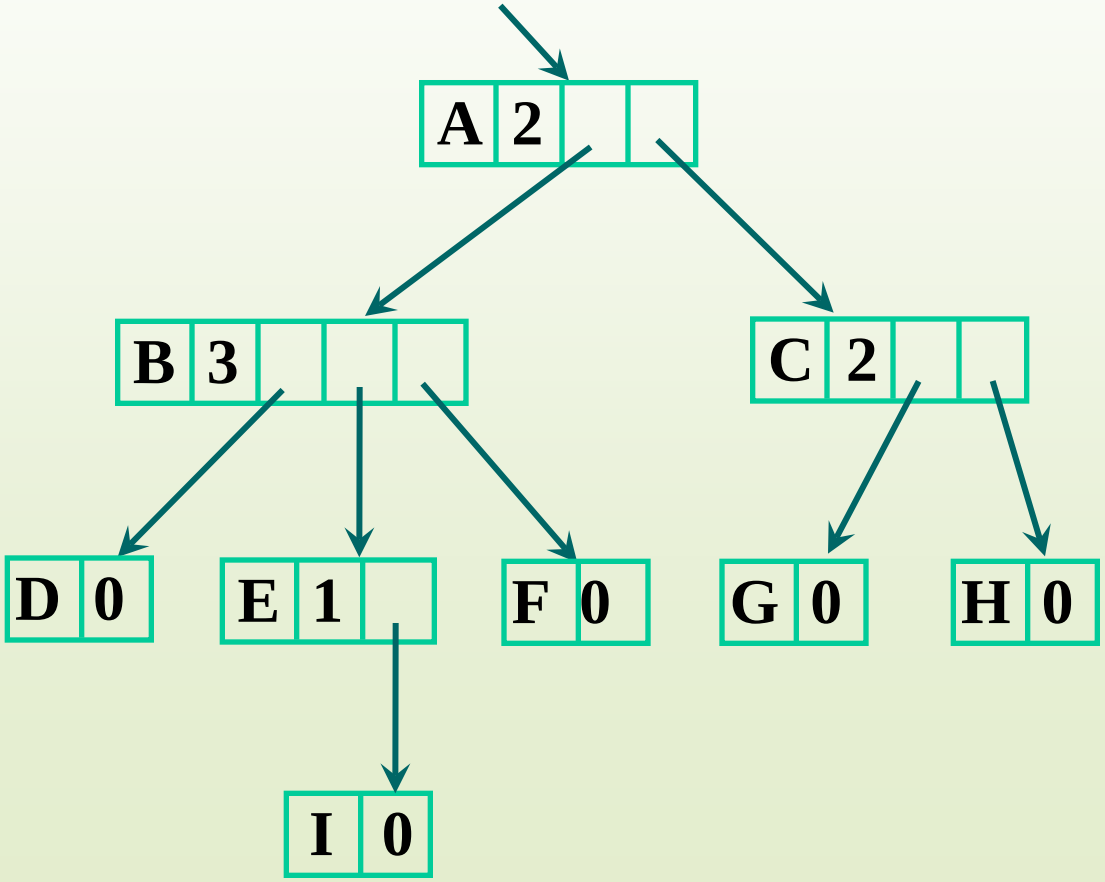
方案二： 指针域的个数等于该结点的度

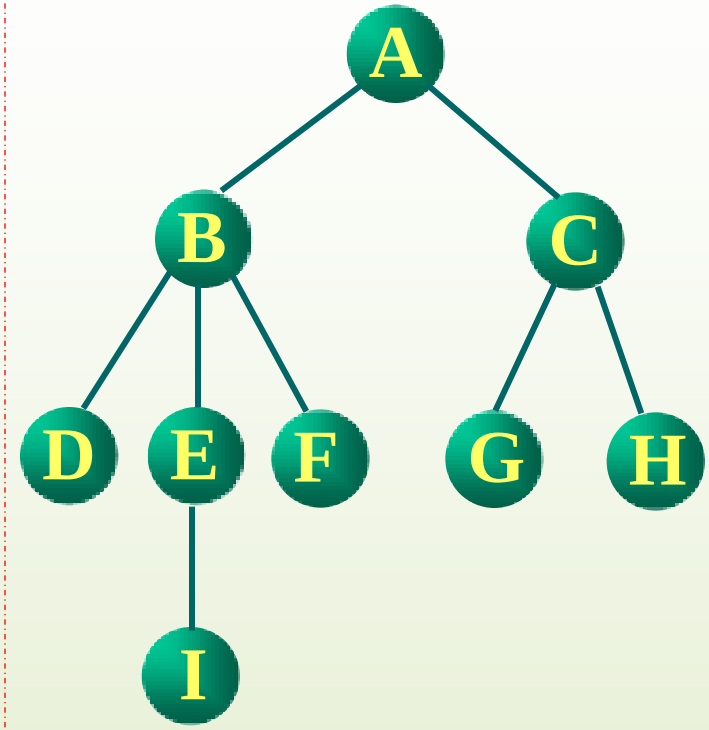
| data | degree | child1 | child2 | …… | childd |
|---|---|---|---|---|---|

其中：data：数据域，存放该结点的数据信息；
degree：度域，存放该结点的度；
child1~childd：指针域，指向该结点的孩子。

# 5.2 树的存储结构

缺点：结点结构不一致



# 5.2 树的存储结构

## 孩子表示法——孩子链表表示法

如何确定链表中的结点结构？

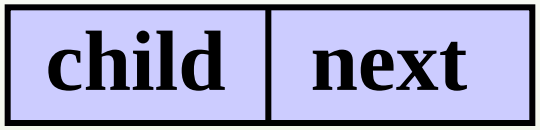
将结点的所有孩子放在一起，构成线性表。

基本思想：把每个结点的孩子排列起来，看成是一个线性表，且以单链表存储，则*n*个结点共有 *n* 个孩子链表。这 *n* 个单链表共有 *n* 个头指针，这 *n* 个头指针又组成了一个线性表，为了便于进行查找采用顺序存储。最后，将存放 *n* 个头指针的数组和存放*n*个结点的数组结合起来，构成孩子链表的表头数组。

# 5.2 树的存储结构

## 孩子表示法——孩子链表表示法

孩子结点

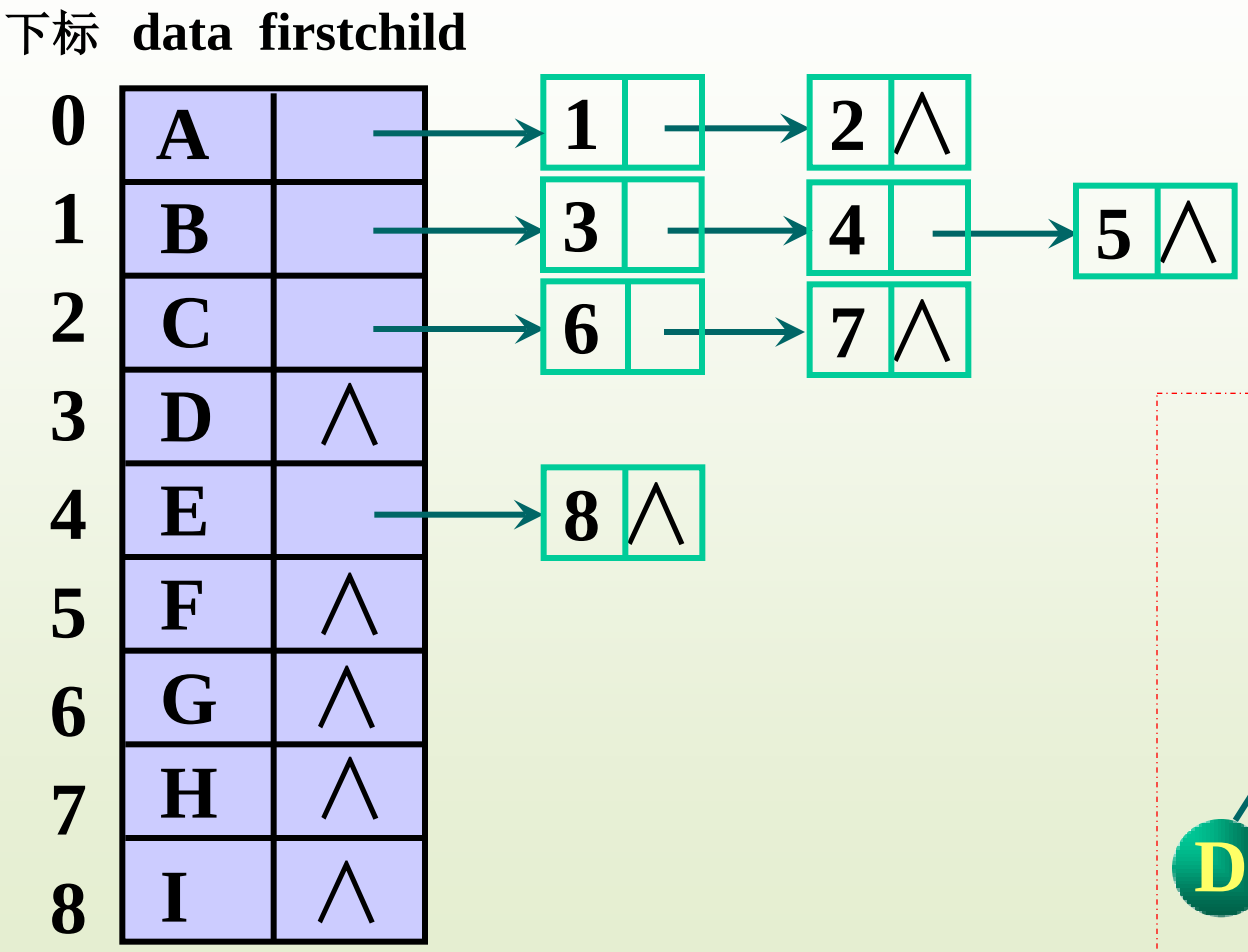


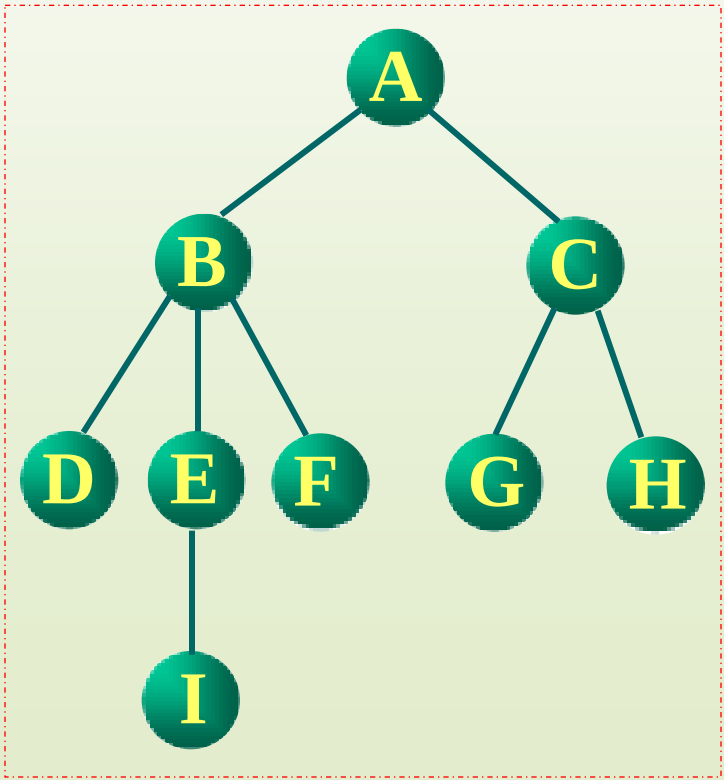

```
struct CTNode
{
    int child;
    CTNode *next;
};
```

表头结点


```
template <class T>
struct CBNode
{
   T data;
   CTNode *firstchild;
};
```

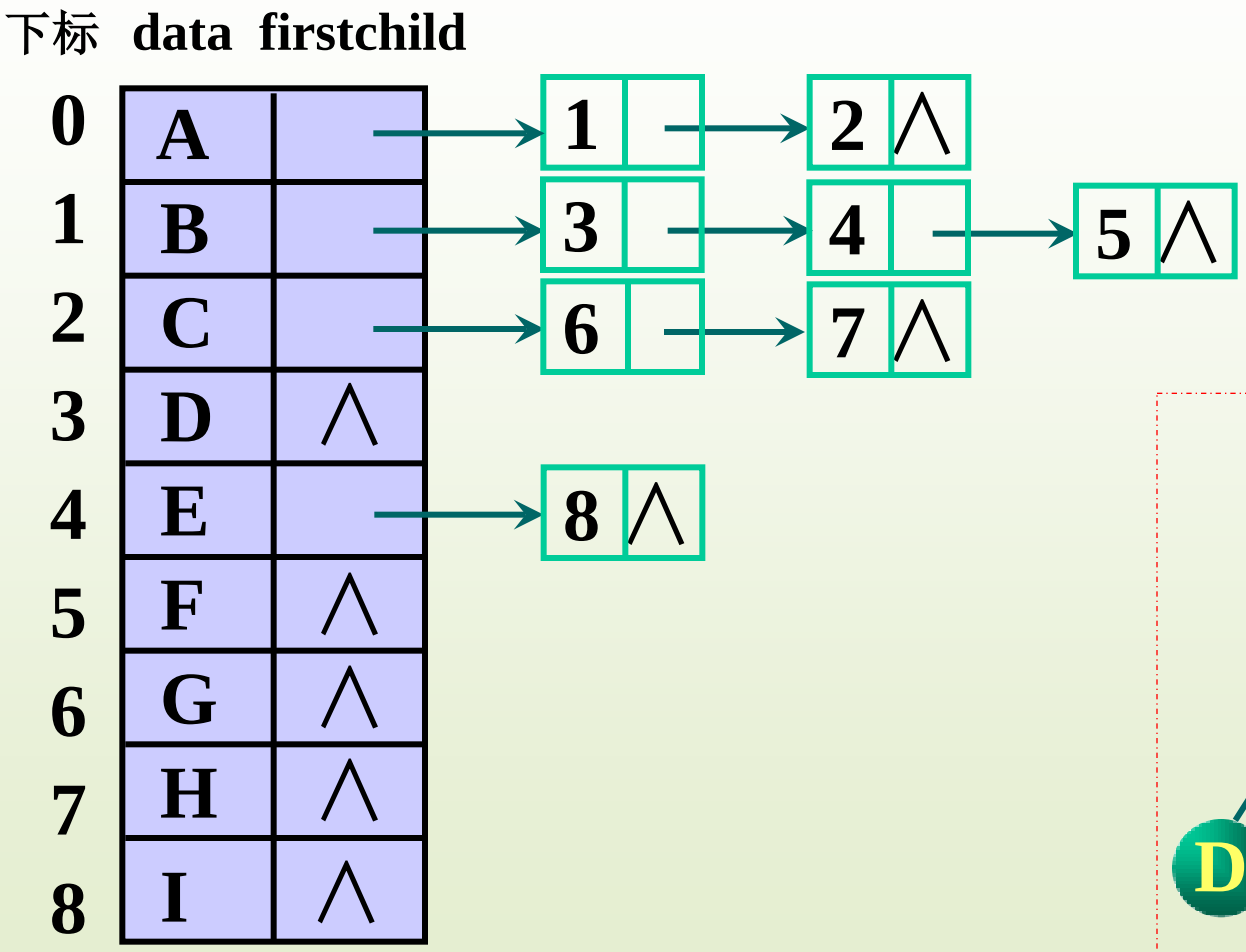
# 5.2 树的存储结构

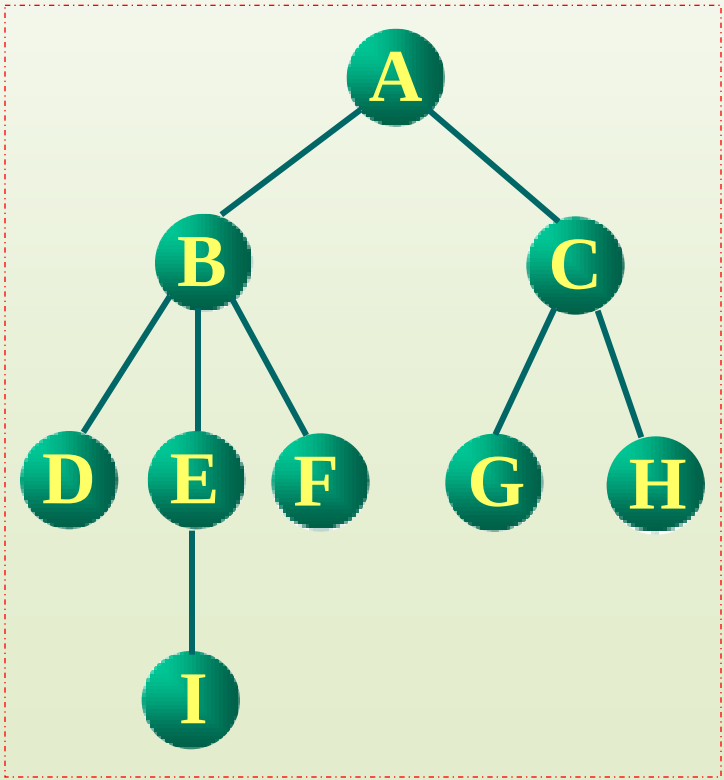



如何查找孩子结点？时间性能？

# 5.2 树的存储结构



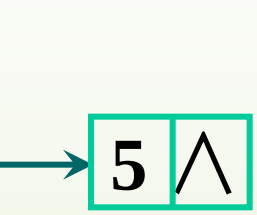

如何查找双亲结点？时间性能？

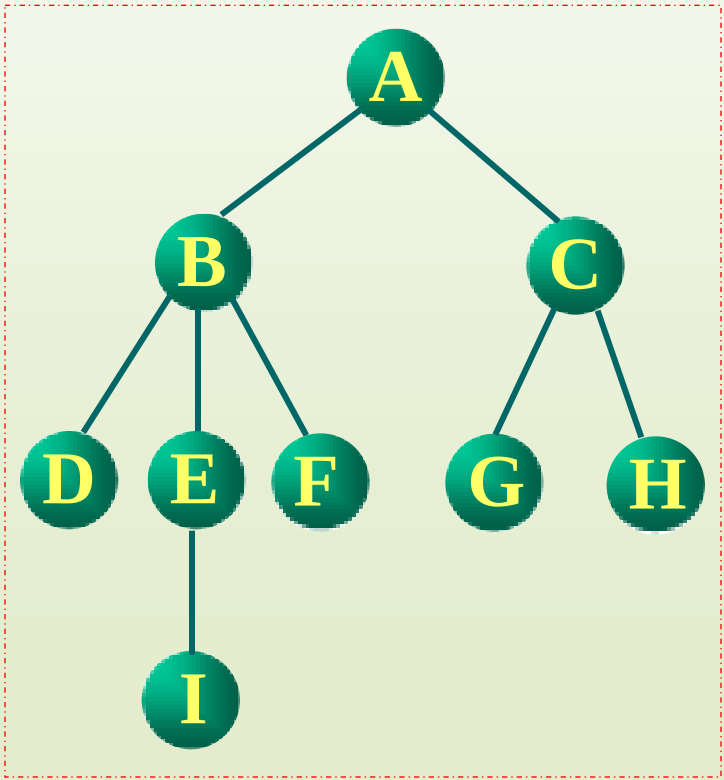
# 5.2 树的存储结构

## 双亲孩子表示法

data parent firstchild

| | data | parent | firstchild |
|---|---|---|---|
| 0 | A | −1 | |
| 1 | B | 0 | |
| 2 | C | 0 | |
| 3 | D | 1 | ∧ |
| 4 | E | 1 | |
| 5 | F | 1 | ∧ |
| 6 | G | 2 | ∧ |
| 7 | H | 2 | ∧ |
| 8 | I | 4 | ∧ |

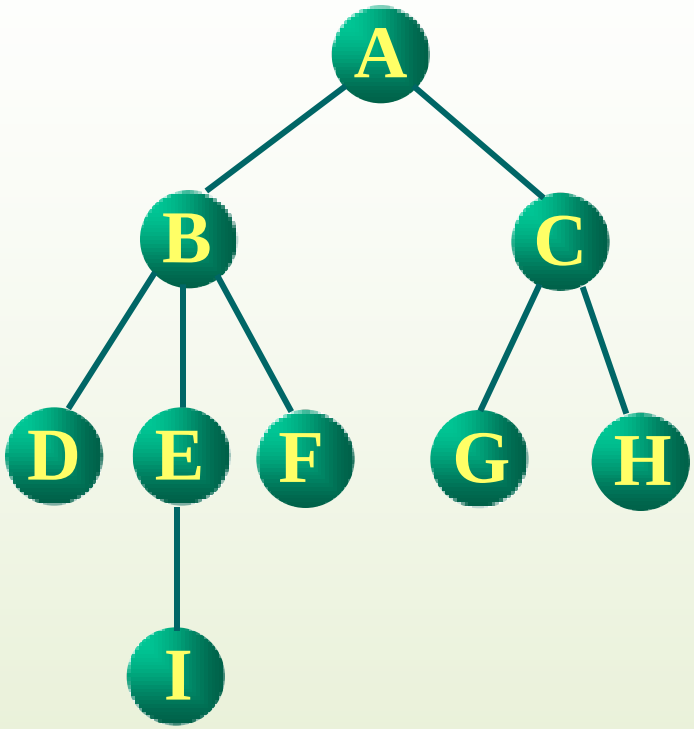


# 5.2 树的存储结构

## 孩子兄弟表示法

某结点的第一个孩子是唯一的
某结点的右兄弟是唯一的




设置两个分别指向该结点的
第一个孩子和右兄弟的指针

# 5.2 树的存储结构

## 孩子兄弟表示法

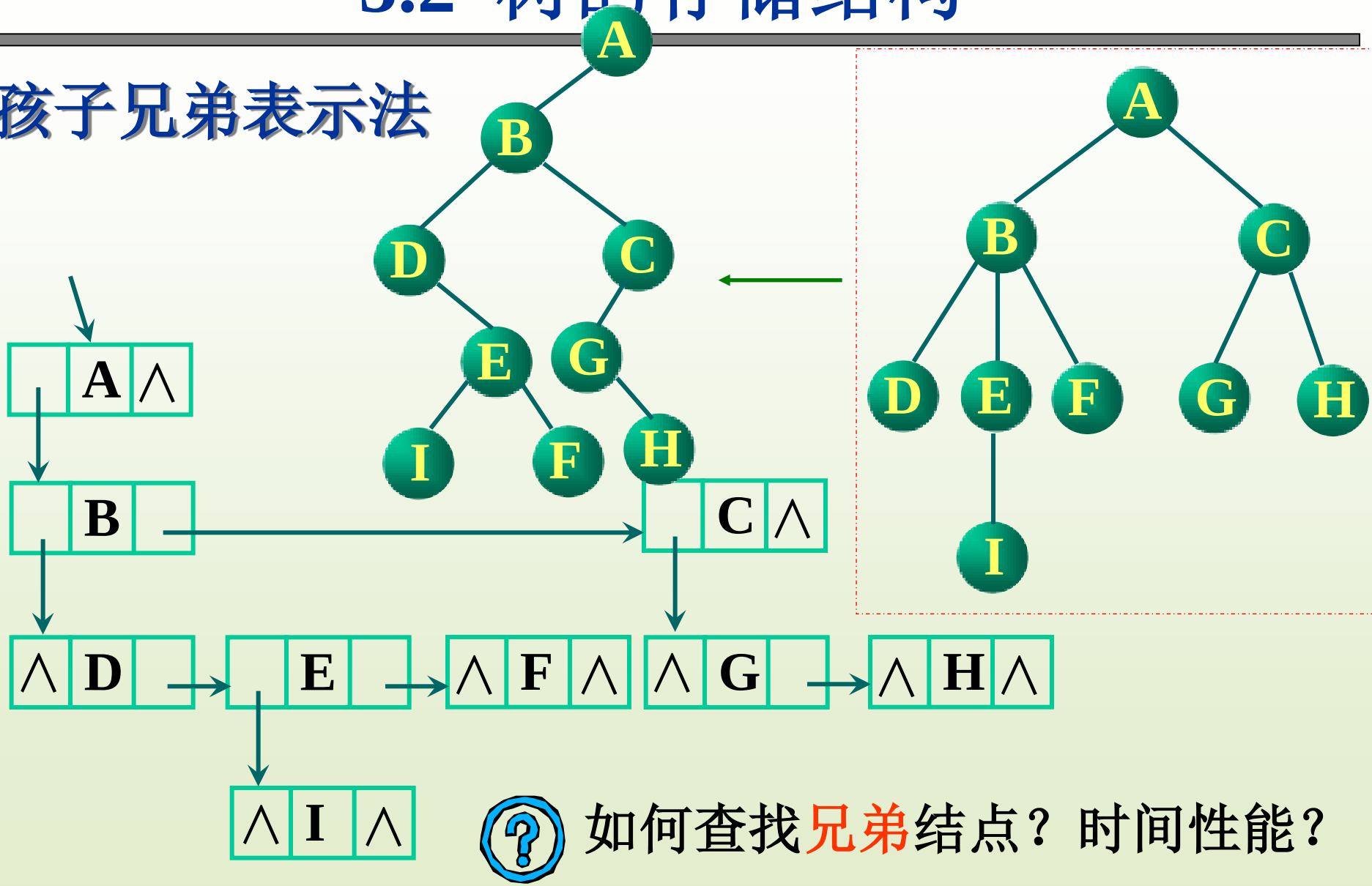
结点结构


**data：数据域，存储该结点的数据信息；**
**firstchild：指针域，指向该结点第一个孩子；**
**rightsib：指针域，指向该结点的右兄弟结点。**

```
template   <class T>
struct TNode
{
    T data;
    TNode <T> *firstchild, *rightsib;
};
```

# 5.2 树的存储结构

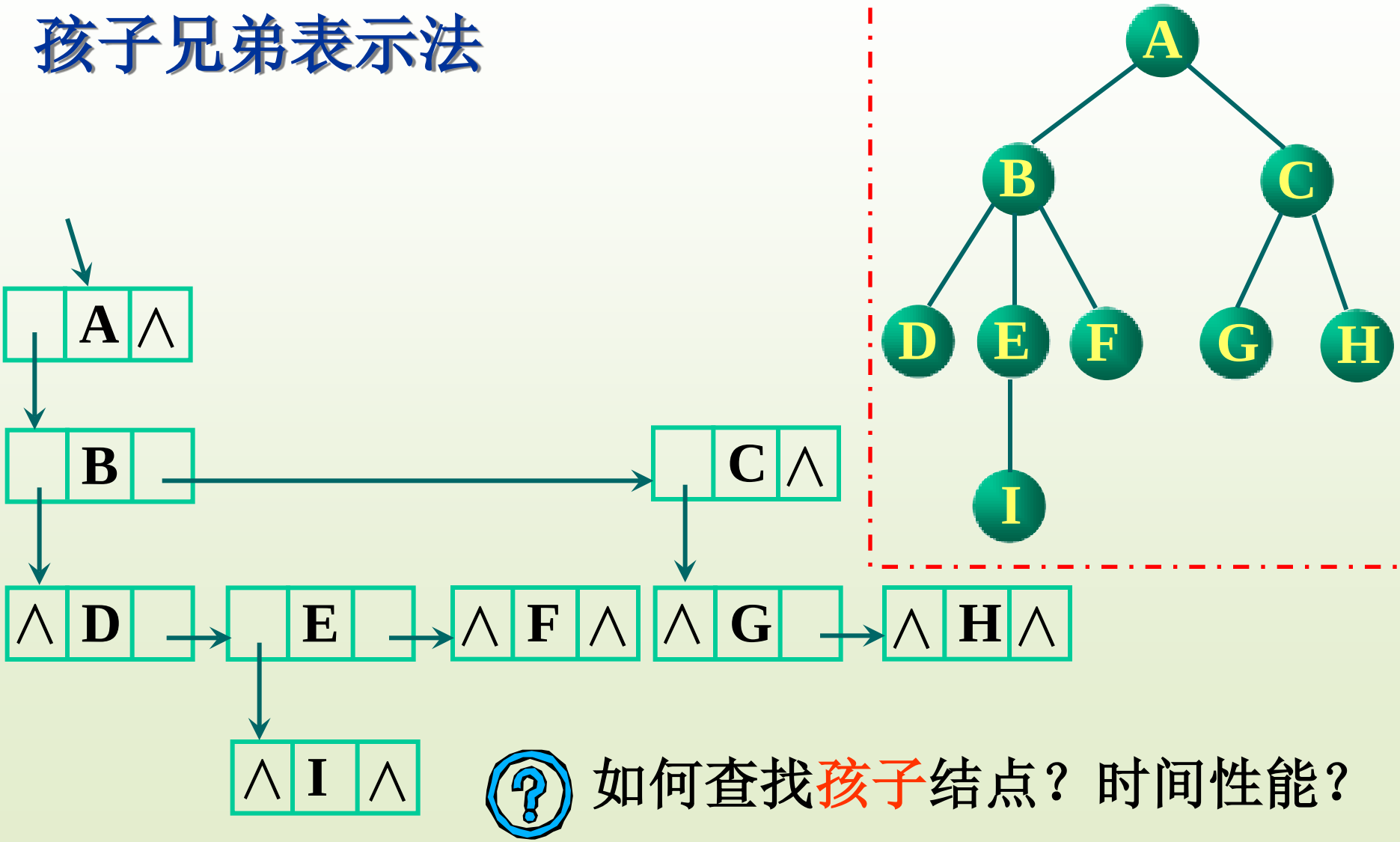
## 孩子兄弟表示法




如何查找兄弟结点？时间性能？

# 5.2 树的存储结构

## 孩子兄弟表示法




如何查找孩子结点？时间性能？

# 5.3 二叉树的逻辑结构

研究二叉树的意义？

**问题转化：将树转换为二叉树，从而利用二叉树解决树的有关问题。**

## 二叉树的定义

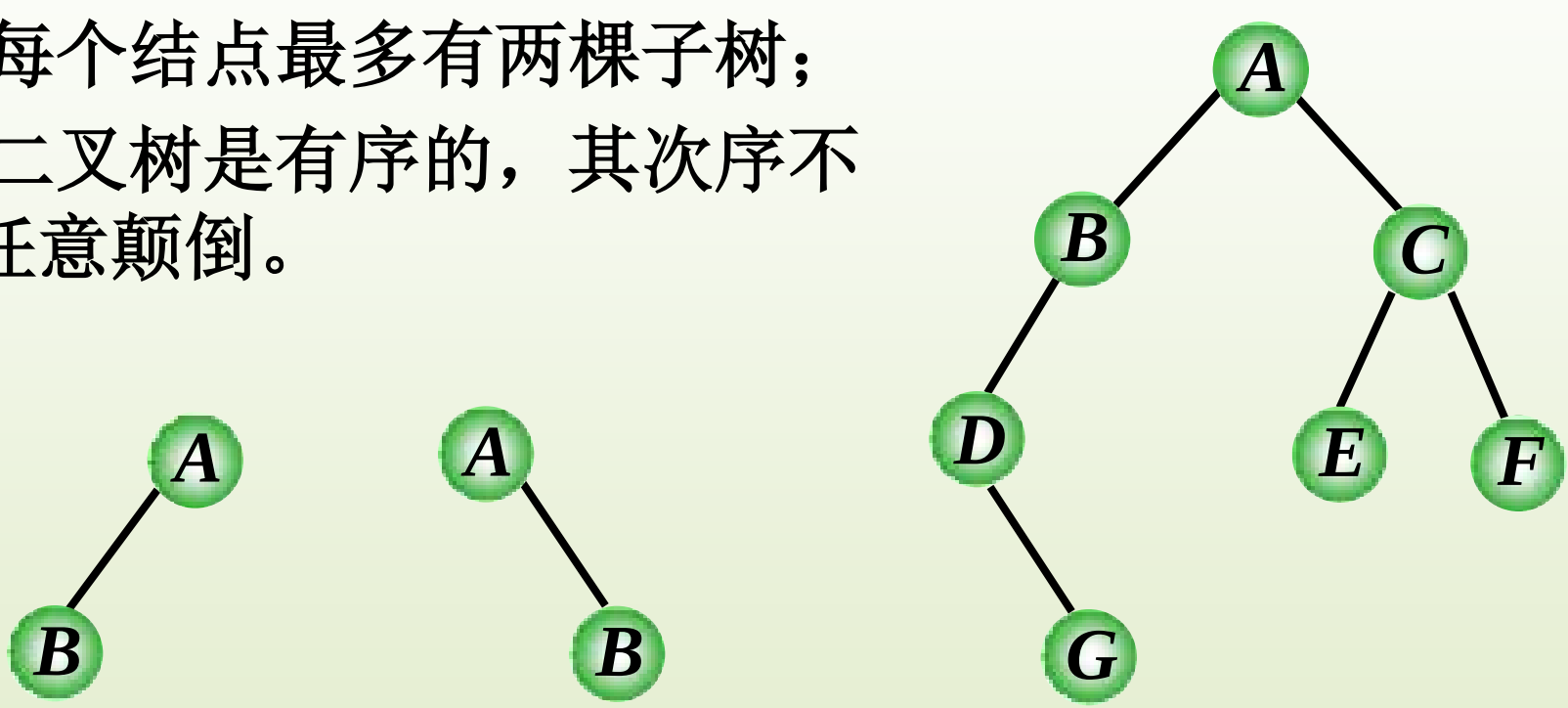
二叉树是$n$（$n\geq0$）个结点的有限集合，该集合或者为空集（称为空二叉树），或者由一个根结点和两棵互不相交的、分别称为根结点的左子树和右子树的二叉树组成。

# 5.3 二叉树的逻辑结构

## 二叉树的特点：

(1) 每个结点最多有两棵子树；

(2) 二叉树是有序的，其次序不能任意颠倒。




注意：二叉树和树是两种树结构。

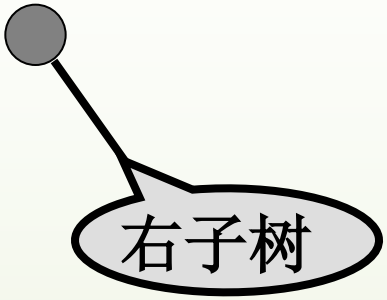
# 5.3 二叉树的逻辑结构

## 二叉树的基本形态

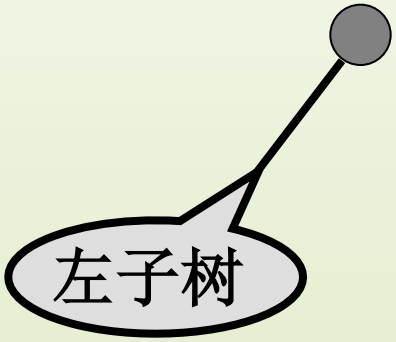
$\Phi$

空二叉树

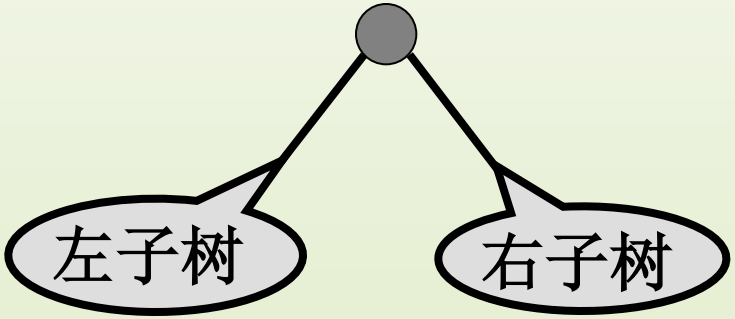
只有一个根结点




根结点只有右子树



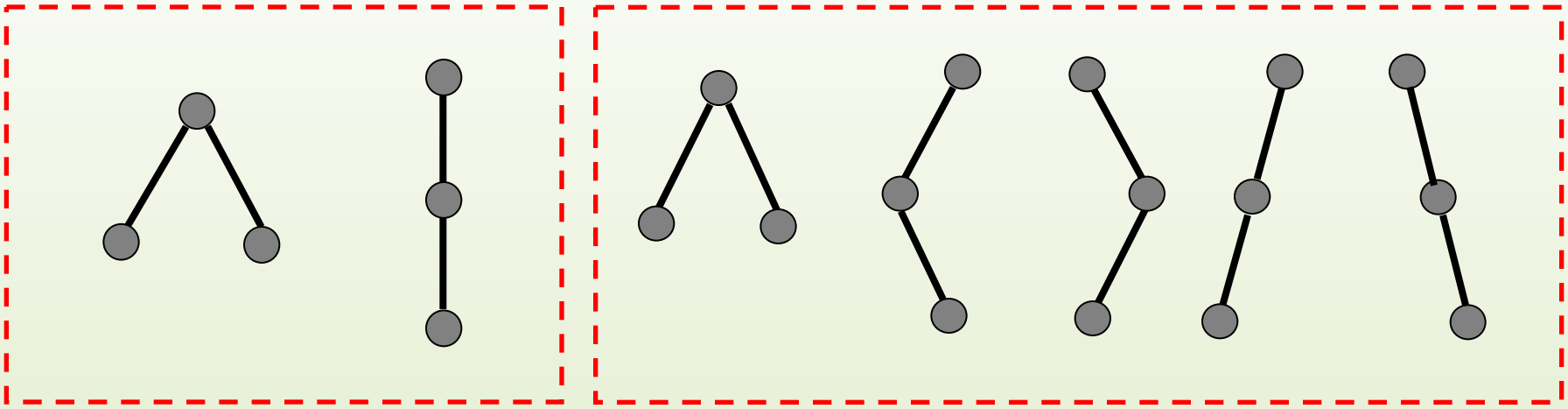

根结点只有左子树




根结点同时有左右子树

# 5.3 二叉树的逻辑结构

具有3个结点的树和具有3个结点的二叉树的形态



❖二叉树和树是两种树结构。

# 5.3 二叉树的逻辑结构

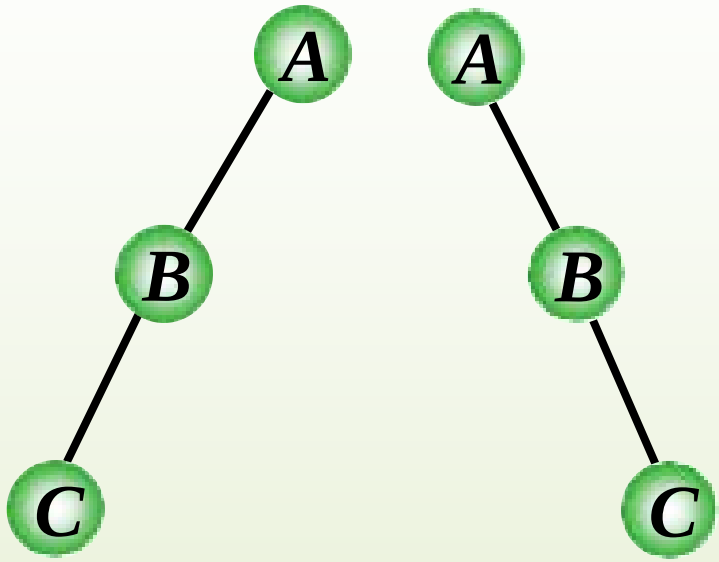
## 特殊的二叉树

### 斜树

1 . 所有结点都只有左子树的二叉树称为左斜树；

2 . 所有结点都只有右子树的二叉树称为右斜树；

3. 左斜树和右斜树统称为斜树。




斜树的特点：

1. 在斜树中，每一层只有一个结点；

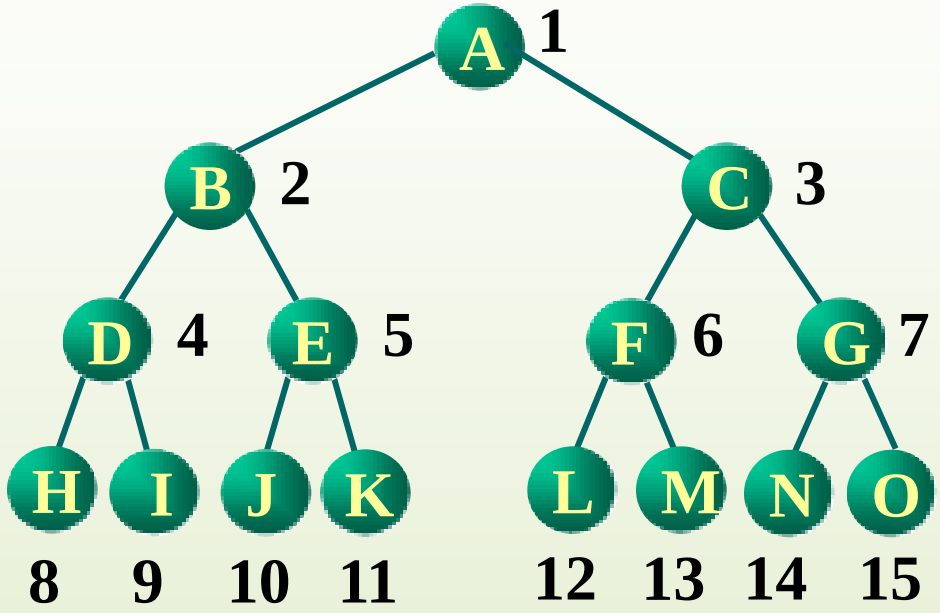
2. 斜树的结点个数与其深度相同。

# 5.3 二叉树的逻辑结构

## 特殊的二叉树

### 满二叉树

在一棵二叉树中，如果所有分支结点都存在左子树和右子树，并且所有叶子都在同一层上。




满二叉树的特点：

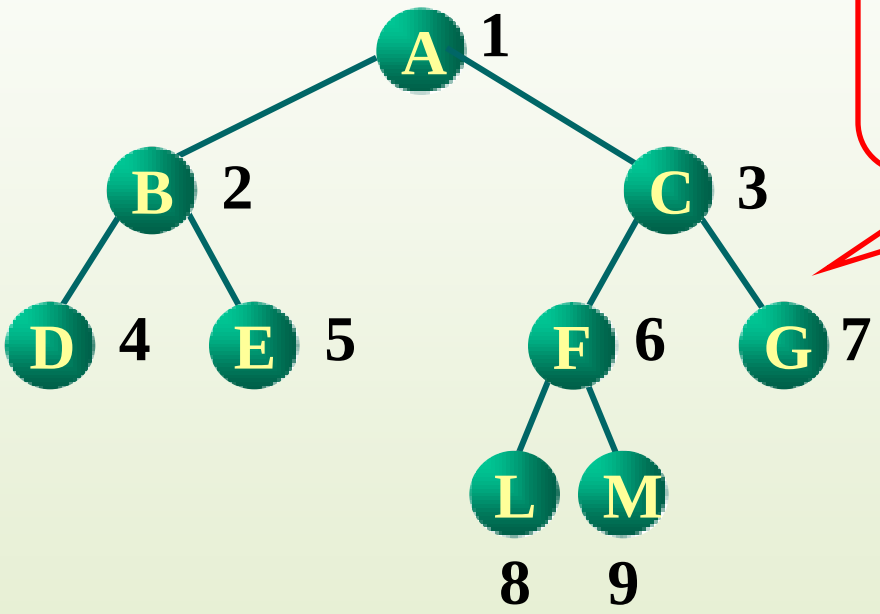
1. 叶子只能出现在最下一层；

2. 只有度为0和度为2的结点。

# 5.3 二叉树的逻辑结构

## 特殊的二叉树

### 满二叉树




不是满二叉树，虽然所有分支结点都有左右子树，但叶子不在同一层上。

满二叉树在同样深度的二叉树中结点个数最多

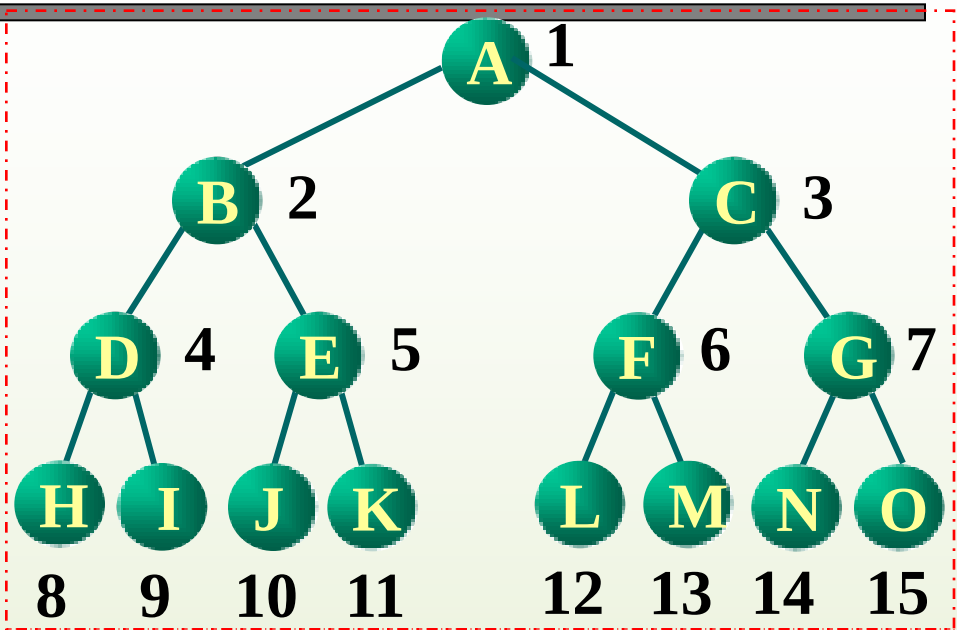
满二叉树在同样深度的二叉树中叶子结点个数最多

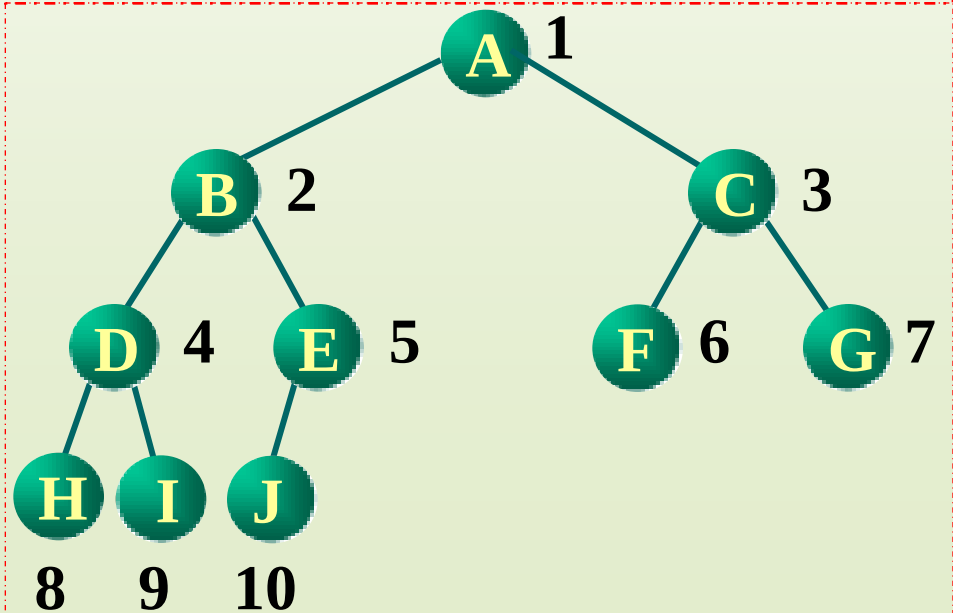
# 5.3 二叉树的逻辑结构

## 特殊的二叉树

### 完全二叉树

对一棵具有*n*个结点的二叉树按层序编号，如果编号为*i*（1≤*i*≤*n*）的结点与同样深度的满二叉树中编号为*i*的结点在二叉树中的位置完全相同。

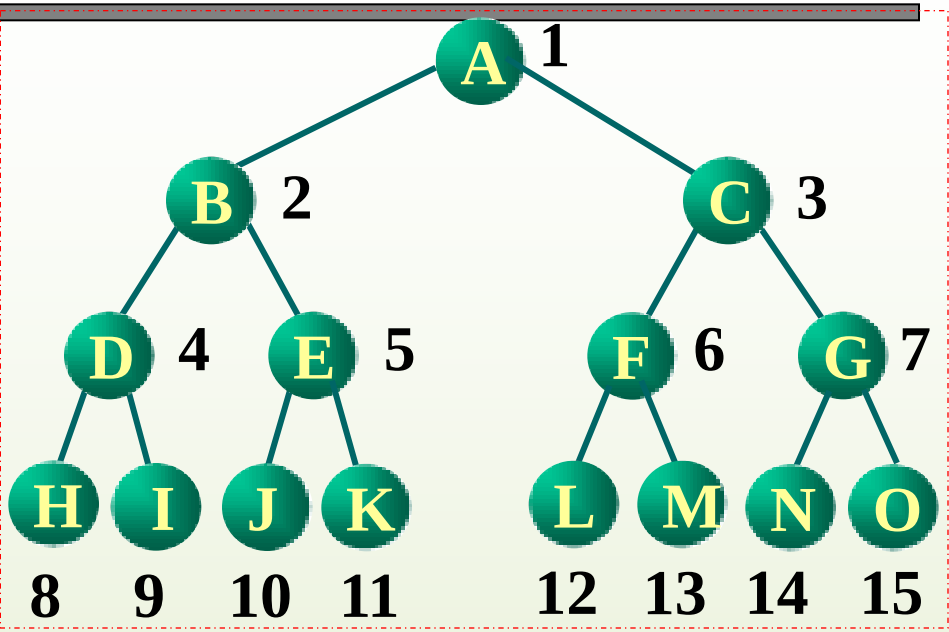


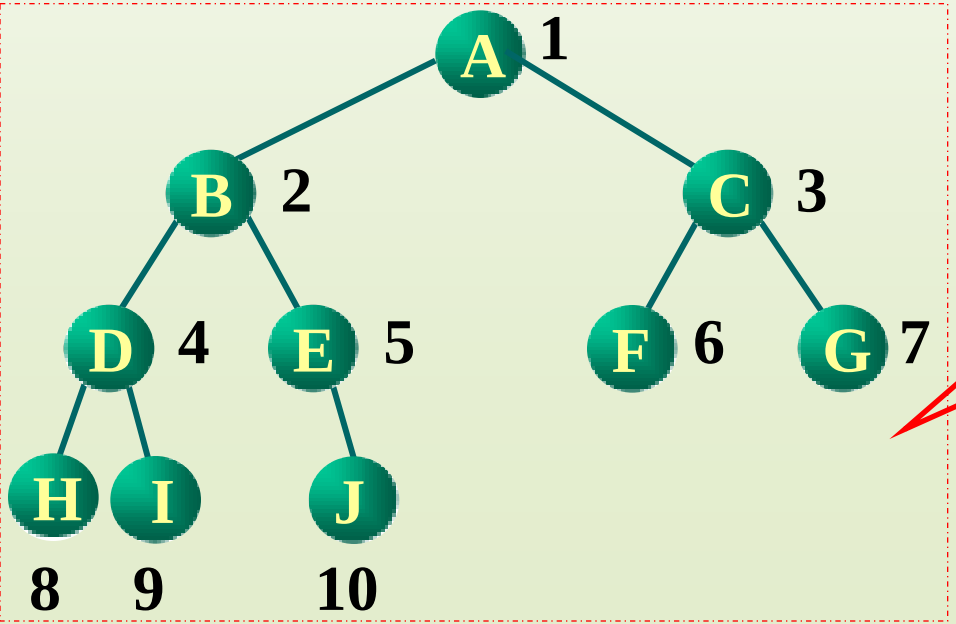



# 5.3 二叉树的逻辑结构

## 特殊的二叉树

在满二叉树中，从最后一个结点开始，连续去掉任意个结点，即是一棵完全二叉树。

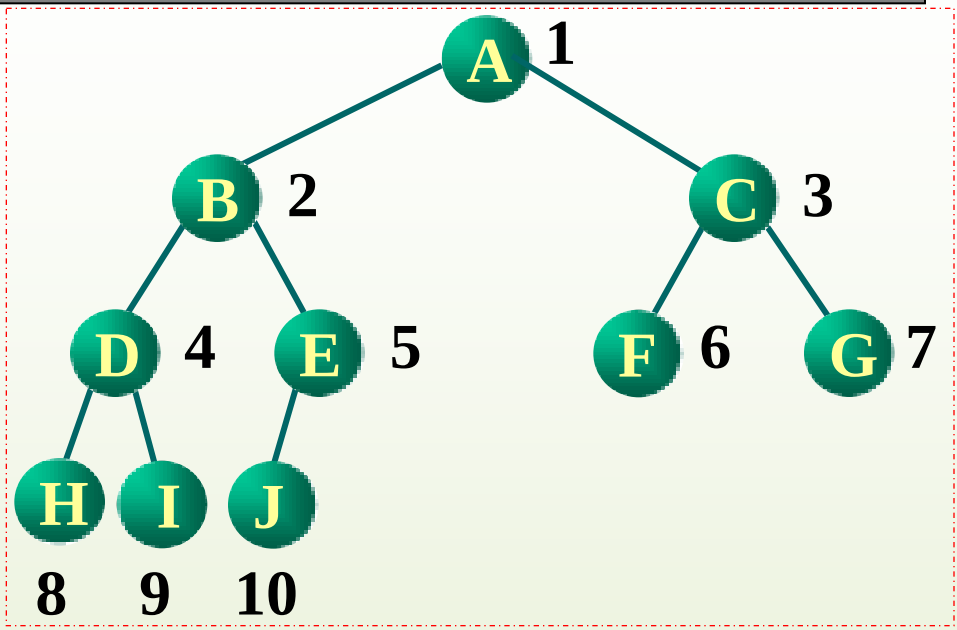



不是完全二叉树，结点10与满二叉树中的结点10不是同一个结点

# 5.3 二叉树的逻辑结构

## 特殊的二叉树




### 完全二叉树的特点

1. 叶子结点只能出现在最下两层，且最下层的叶子结点都集中在二叉树的左部；

2. 完全二叉树中如果有度为1的结点，只可能有一个，且该结点只有左孩子。

3. 深度为k的完全二叉树在k-1层上一定是满二叉树。

# 5.3 二叉树的逻辑结构

## 二叉树的基本性质

性质5-1 二叉树的第$i$层上最多有$2^{i-1}$个结点（$i \geqslant 1$）。

证明：当i=1时，第1层只有一个根结点，而

$2^{i-1}=2^0=1$，结论显然成立。

假定i=k（$1 \leq k < i$）时结论成立，即第k层上至多有$2^{k-1}$个结点， 则 i=k+1时，因为第k+1层上的结点是第k层上结点的孩子，而二叉树中每个结点最多有2个孩子，故在第k+1层上最大结点个数为第k层上的最大结点个数的二倍，即$2\times 2^{k-1}=2^k$。结论成立。

# 5.3 二叉树的逻辑结构

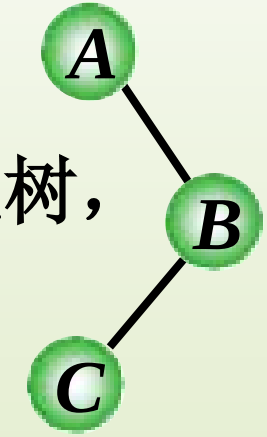
## 二叉树的基本性质

性质5-2　一棵深度为k的二叉树中，最多有$2^k-1$个结点，最少有$k$个结点。

证明：由性质1可知，深度为$k$的二叉树中结点个数最多

$= \sum_{i=1}^{k}$(第$i$层上结点的最大个数) $=2^k-1$；

每一层至少要有一个结点，因此深度为k的二叉树，至少有$k$个结点。




深度为$k$且具有$2^k-1$个结点的二叉树一定是满二叉树，

深度为$k$且具有$k$个结点的二叉树不一定是斜树。

# 5.3 二叉树的逻辑结构

## 二叉树的基本性质

性质5-3　在一棵二叉树中，如果叶子结点数为$n_0$，度为2的结点数为$n_2$，则有：$n_0 = n_2 + 1$。

证明：设$n_1$为二叉树T中度为1的结点数

∵　二叉树中所有结点的度均小于或等于2

∴　结点总数$n=n_0+n_1+n_2$　　(1)

又二叉树中，除根结点外，其余结点都只有一个分支进入，设B为分支总数，则$n=B+1$ (2)

又：分支由度为1和度为2的结点射出，

∴$B=n_1+2n_2$

于是，$n=B+1=n_1+2n_2+1$　　(3)

∴由式(1)和(3)得 $n_0=n_2+1$

同理：

三次树：　$n=1+n+2n$

# 5.3 二叉树的逻辑结构

## 二叉树的基本性质

性质5-3　在一棵二叉树中，如果叶子结点数为$n_0$，度为2的结点数为$n_2$，则有: $n_0 = n_2 + 1$。

在有$n$个结点的满二叉树中，有多少个叶子结点？

因为在满二叉树中没有度为1的结点，只有度为0的叶子结点和度为2的分支结点，所以，

$$n = n_0 + n_2$$

$$n_0 = n_2 + 1$$

即叶子结点$n_0 = (n + 1)/2$

# 5.3 二叉树的逻辑结构

## 完全二叉树的基本性质

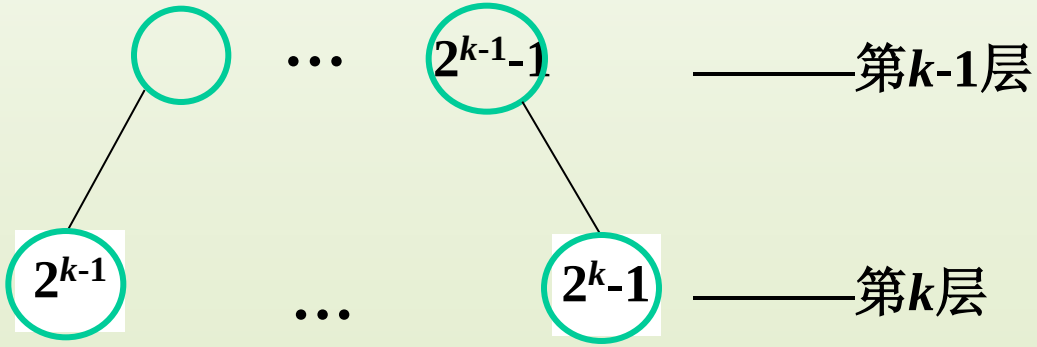
性质5-4 具有$n$个结点的完全二叉树的深度为$\lfloor \log_2 n \rfloor + 1$。

证明：假设具有$n$个结点的完全二叉树的深度为$k$，根据完全二叉树的定义和性质2，有下式成立

$$2^{k-1} \le n < 2^k$$

最少结点数

最多结点数



# 5.3 二叉树的逻辑结构

## 完全二叉树的基本性质

性质5-4 具有$n$个结点的完全二叉树的深度为$\lfloor \log_2 n \rfloor + 1$。

证明：假设具有$n$个结点的完全二叉树的深度为$k$，根据完全二叉树的定义和性质2，有下式成立

$$2^{k-1} \le n < 2^k$$

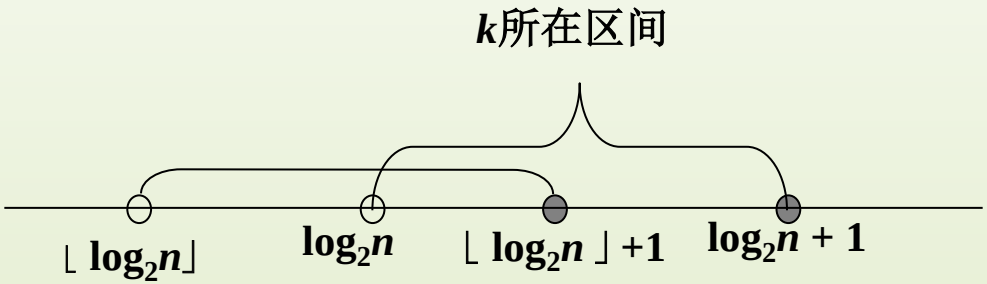
对不等式取对数，有：

$$k-1 \le \log_2 n < k$$

即：

$$\log_2 n < k \le \log_2 n + 1$$

由于$k$是整数，故必有$k = \lfloor \log_2 n \rfloor + 1$。




常出现在选择题中

说明：符号$\lfloor x \rfloor$表示不大于x的最大整数。

# 5.3 二叉树的逻辑结构

## 完全二叉树的基本性质

性质5-5　对一棵具有$n$个结点的完全二叉树中从1开始按层序编号，则对于任意的序号为$i$（$1 \le i \le n$）的结点（简称为结点$i$），有：

(1)如果$i>1$，则结点$i$的双亲结点的序号为 $i/2$；如果$i=1$，则结点$i$是根结点，无双亲结点。

(2)如果$2i \le n$，则结点$i$的左孩子的序号为$2i$；

如果$2i>n$，则结点$i$无左孩子。

(3)如果$2i+1 \le n$，则结点$i$的右孩子的序号为$2i+1$；

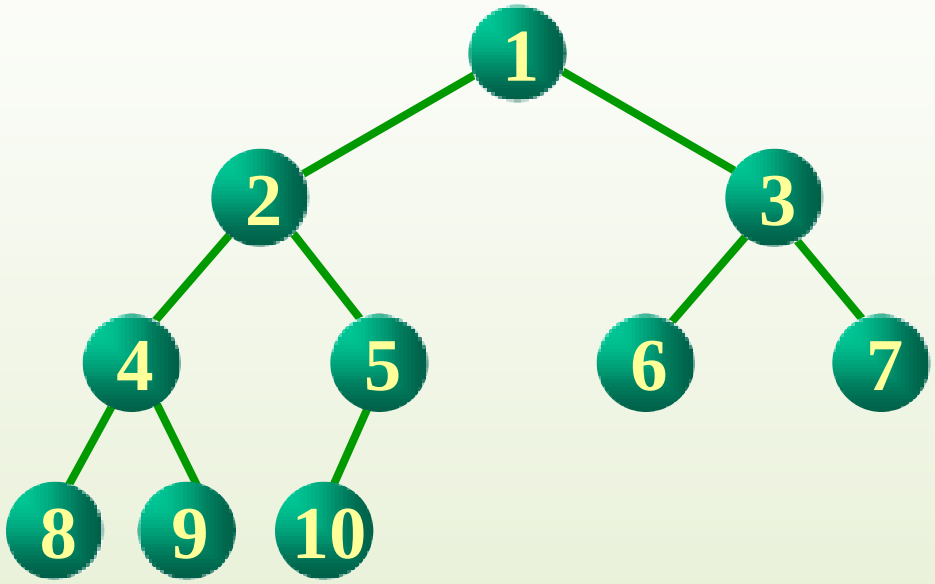
如果$2i+1>n$，则结点 $i$无右孩子。

# 5.3 二叉树的逻辑结构

## 完全二叉树的基本性质

对一棵具有$n$个结点的完全二叉树中从1开始按层序编号，则

- 结点$i$的双亲结点为 $i/2$；
- 结点$i$的左孩子为$2i$；
- 结点$i$的右孩子为$2i+1$。




性质5表明，在完全二叉树中，结点的层序编号反映了结点之间的逻辑关系。

# 5.3 二叉树的逻辑结构

## 二叉树的抽象数据类型定义

**ADT BiTree**

**Data**

由一个根结点和两棵互不相交的左右子树构成，结点具有相同数据类型及层次关系

**Operation**

**InitBiTree**

前置条件：无

输入：无

功能：初始化一棵二叉树

输出：无

后置条件：构造一个空的二叉树

# 5.3 二叉树的逻辑结构

## 二叉树的抽象数据类型定义

**DestroyBiTree**

前置条件：二叉树已存在
输入：无
功能：销毁一棵二叉树
输出：无
后置条件：释放二叉树占用的存储空间

**InsertL**

前置条件：二叉树已存在
输入：数据值x，指针parent
功能：将数据域为x的结点插入到二叉树中，作为结点parent的左孩子。如果结点parent原来有左孩子，则将结点parent原来的左孩子作为结点x的左孩子
输出：无
后置条件：如果插入成功，得到一个新的二叉树

# 5.3 二叉树的逻辑结构

## 二叉树的抽象数据类型定义

**DeleteL**

前置条件：二叉树已存在
输入：指针parent
功能：在二叉树中删除结点parent的左子树
输出：无
后置条件：如果删除成功，得到一个新的二叉树

**Search**

前置条件：二叉树已存在
输入：数据值x
功能：在二叉树中查找数据元素x
输出：指向该元素结点的指针
后置条件：二叉树不变

# 5.3 二叉树的逻辑结构

## 二叉树的抽象数据类型定义

**PreOrder**

前置条件：二叉树已存在
输入：无
功能：前序遍历二叉树
输出：二叉树中结点的一个线性排列
后置条件：二叉树不变

**InOrder**

前置条件：二叉树已存在
输入：无
功能：中序遍历二叉树
输出：二叉树中结点的一个线性排列
后置条件：二叉树不变

# 5.3 二叉树的逻辑结构

## 二叉树的抽象数据类型定义

**PostOrder**

前置条件：二叉树已存在
输入：无
功能：后序遍历二叉树
输出：二叉树中结点的一个线性排列
后置条件：二叉树不变

**LevelOrder**

前置条件：二叉树已存在
输入：无
功能：层序遍历二叉树
输出：二叉树中结点的一个线性排列
后置条件：二叉树不变

**endADT**

# 5.3 二叉树的逻辑结构

## 二叉树的遍历操作

二叉树的遍历是指从根结点出发，按照某种次序访问二叉树中的所有结点，使得每个结点被访问一次且仅被访问一次。

前序遍历
中序遍历
后序遍历
层序遍历

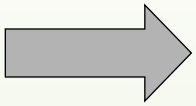
抽象操作，可以是对结点进行的各种处理，这里简化为输出结点的数据。

二叉树遍历操作的结果？⟹非线性结构线性化

# 5.3 二叉树的逻辑结构

考虑二叉树的组成：

二叉树 ┬ 根结点D
       ├ 左子树L
       └ 右子树R



二叉树的遍历方式：

**DLR、LDR、LRD、**

**DRL、RDL、RLD**

如果限定先左后右，则二叉树遍历方式有三种：

前序：**DLR**

中序：**LDR**

后序：**LRD**

层序遍历：按二叉树的层序编号的次序访问各结点。

# 5.3 二叉树的逻辑结构

## 二叉树的遍历操作

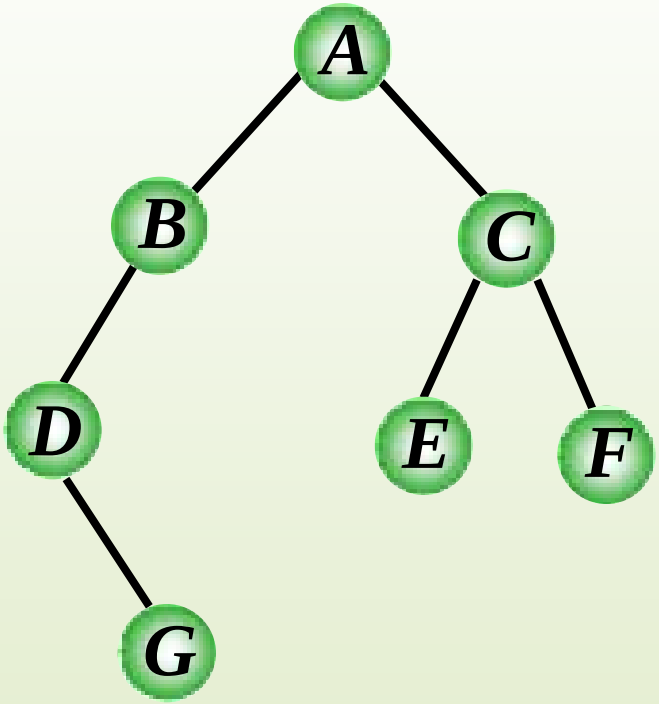
### 前序（根）遍历

若二叉树为空，则空操作返回；
否则：

①访问根结点；

②前序遍历根结点的左子树；

③前序遍历根结点的右子树。




前序遍历序列：*A B D G C E F*

# 5.3 二叉树的逻辑结构

## 二叉树的遍历操作

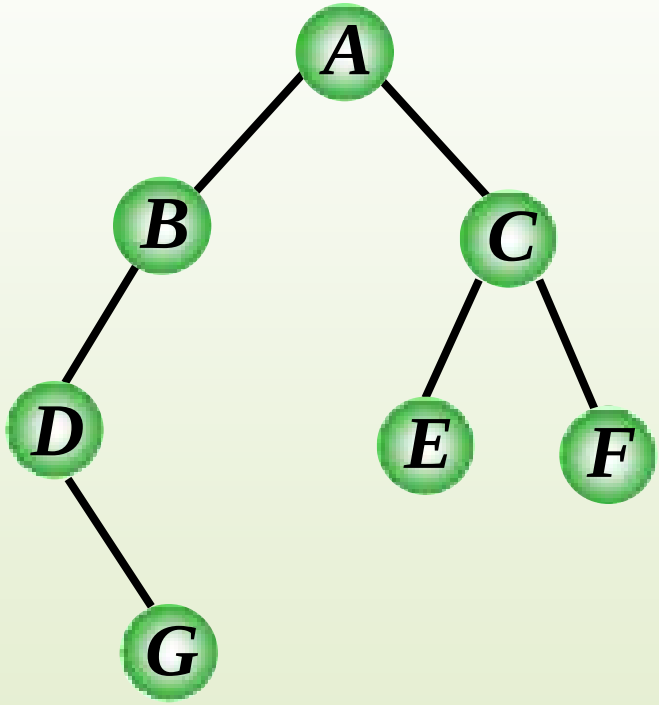
### 中序（根）遍历

若二叉树为空，则空操作返回；
否则：

①中序遍历根结点的左子树；

②访问根结点；

③中序遍历根结点的右子树。




中序遍历序列：*D G B A E C F*

# 5.3 二叉树的逻辑结构

## 二叉树的遍历操作

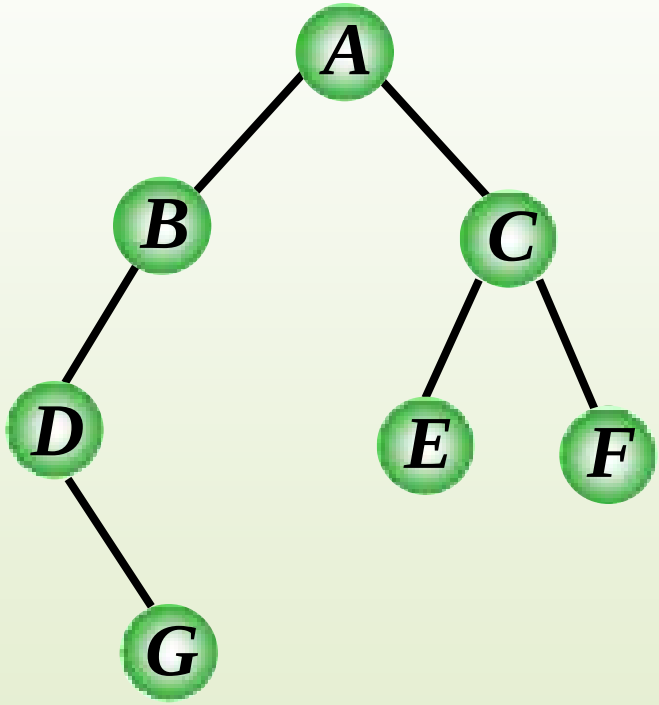
### 后序（根）遍历

若二叉树为空，则空操作返回；
否则：

①后序遍历根结点的左子树；

②后序遍历根结点的右子树。

③访问根结点；



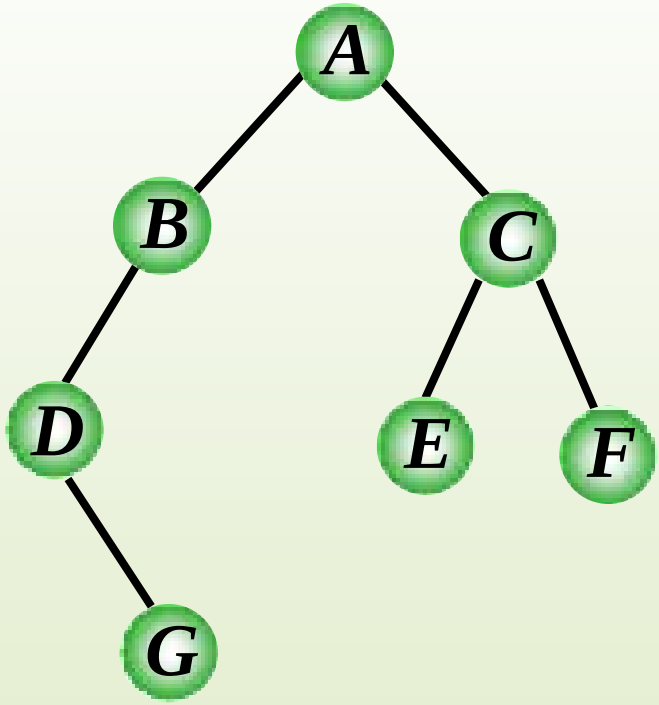

后序遍历序列：*G D B E F C A*

# 5.3 二叉树的逻辑结构

## 二叉树的遍历操作

### 层序遍历

二叉树的层次遍历是指从二叉树的第一层（即根结点）开始，从上至下逐层遍历，在同一层中，则按从左到右的顺序对结点逐个访问。




层序遍历序列：*A B C D E F G*

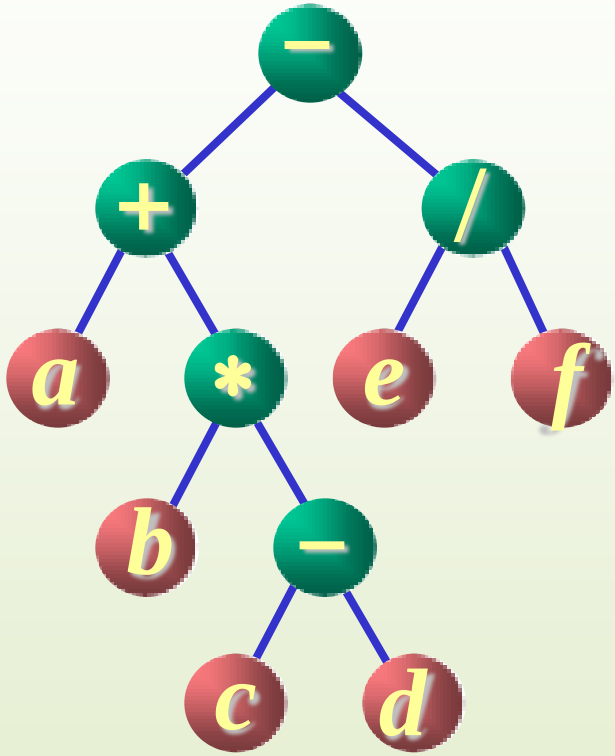
# 5.3 二叉树的逻辑结构

## 二叉树遍历操作练习

前序遍历结果：- + a * b - c d / e f

中序遍历结果：a + b * c - d - e / f

后序遍历结果：a b c d - * + e f / -

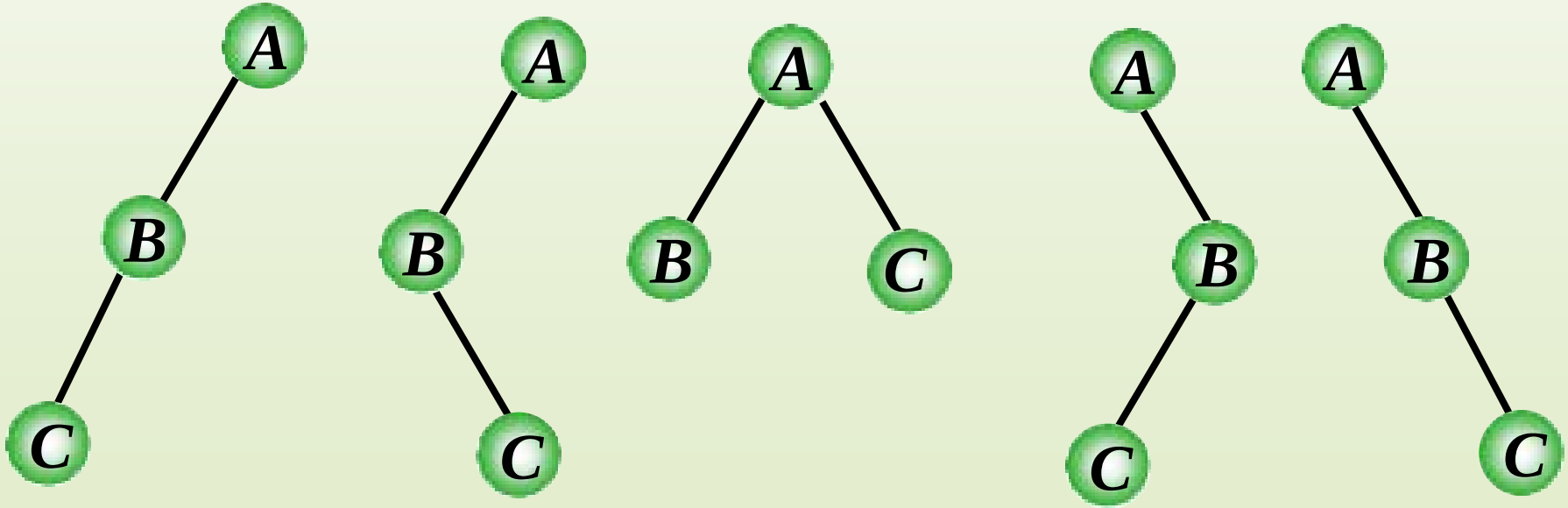


# 5.3 二叉树的逻辑结构

## 二叉树的遍历操作

若已知一棵二叉树的前序（或中序，或后序，或层序）序列，能否唯一确定这棵二叉树呢？

例：已知前序序列为ABC，则可能的二叉树有5种。

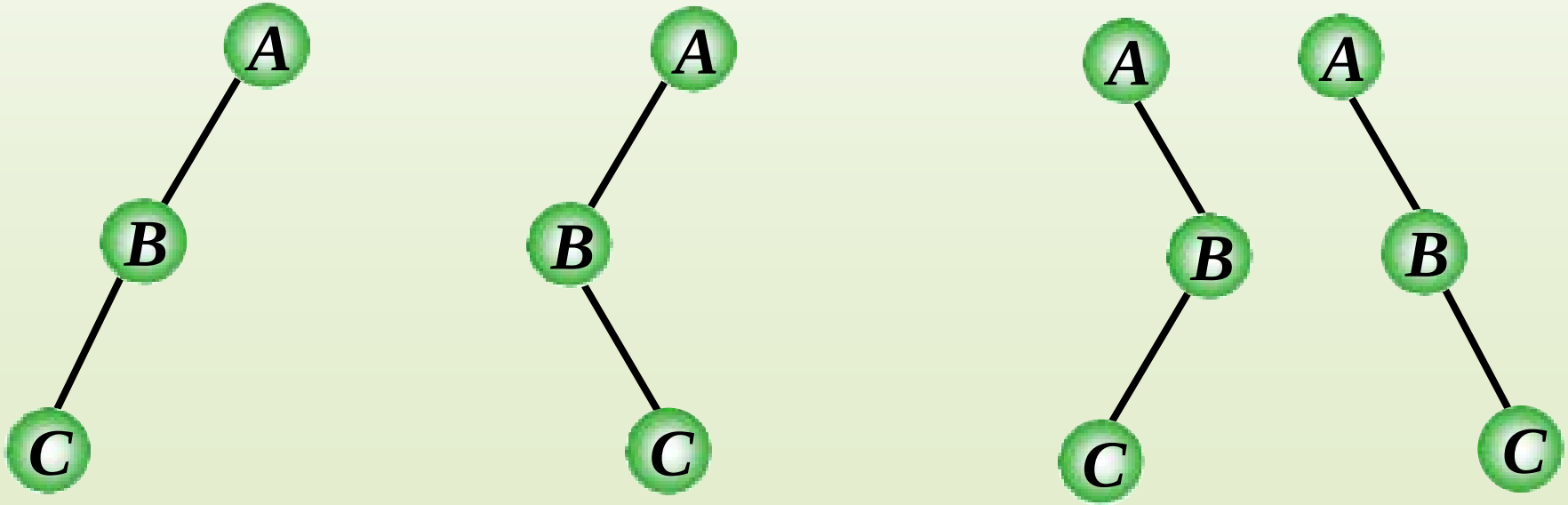


# 5.3 二叉树的逻辑结构

## 二叉树的遍历操作

若已知一棵二叉树的前序序列和后序序列，能否唯一确定这棵二叉树呢？

例：已知前序遍历序列为ABC，后序遍历序列为CBA，则下列二叉树都满足条件。



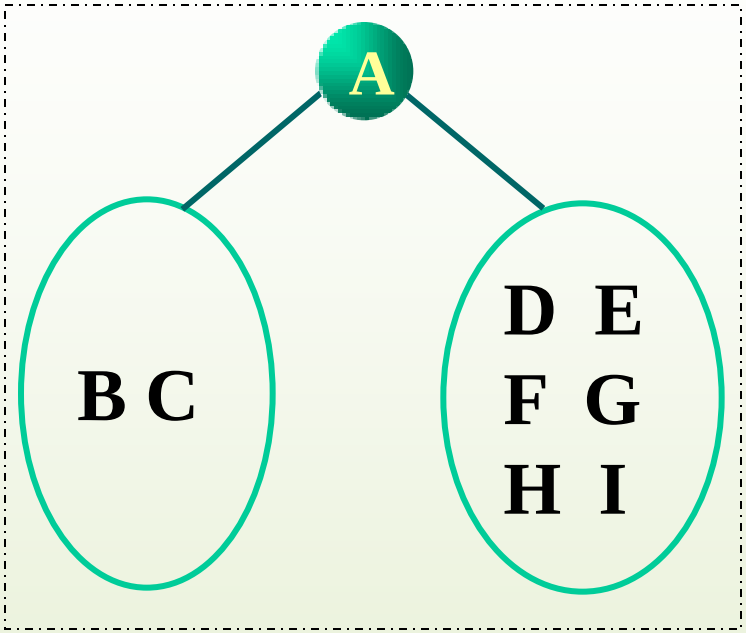
# 5.3 二叉树的逻辑结构

## 二叉树的遍历操作

若已知一棵二叉树的前序序列和中序序列，能否唯一确定这棵二叉树呢？怎样确定？

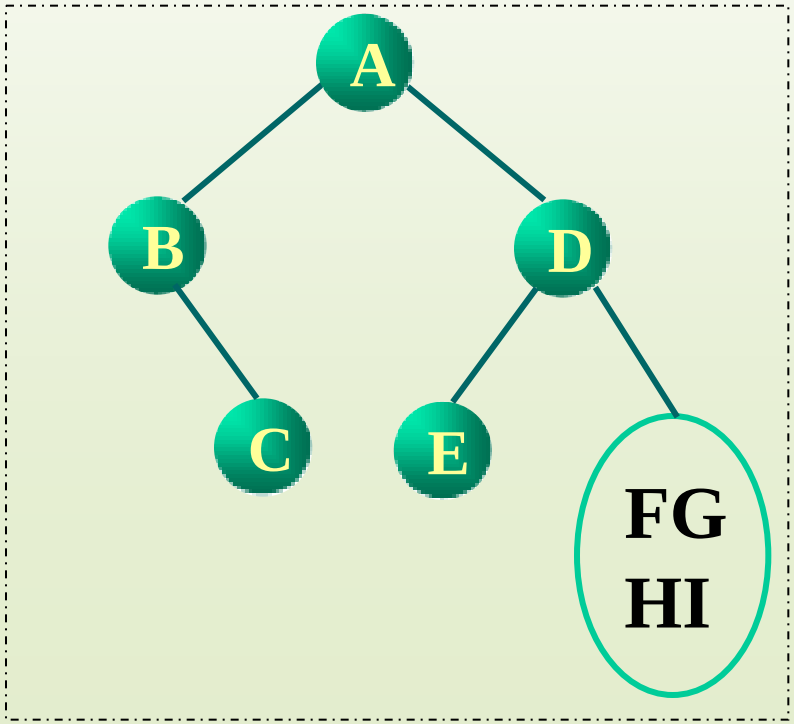
例如：已知一棵二叉树的前序遍历序列和中序遍历序列分别为**ABCDEFGHI** 和**BCAEDGHFI**，如何构造该二叉树?

# 5.3 二叉树的逻辑结构




前序：A B C D E F G H I

中序：B C A E D G H F I



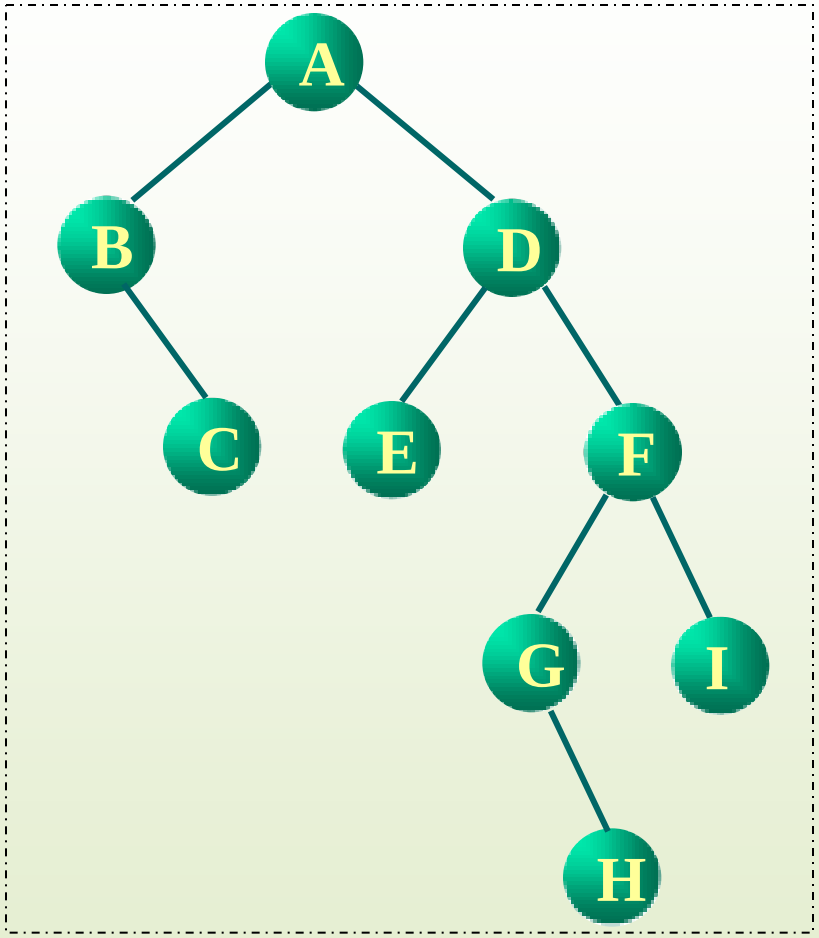

前序：B C

中序：B C

前序：D E F G H I

中序：E D G H F I

# 5.3 二叉树的逻辑结构

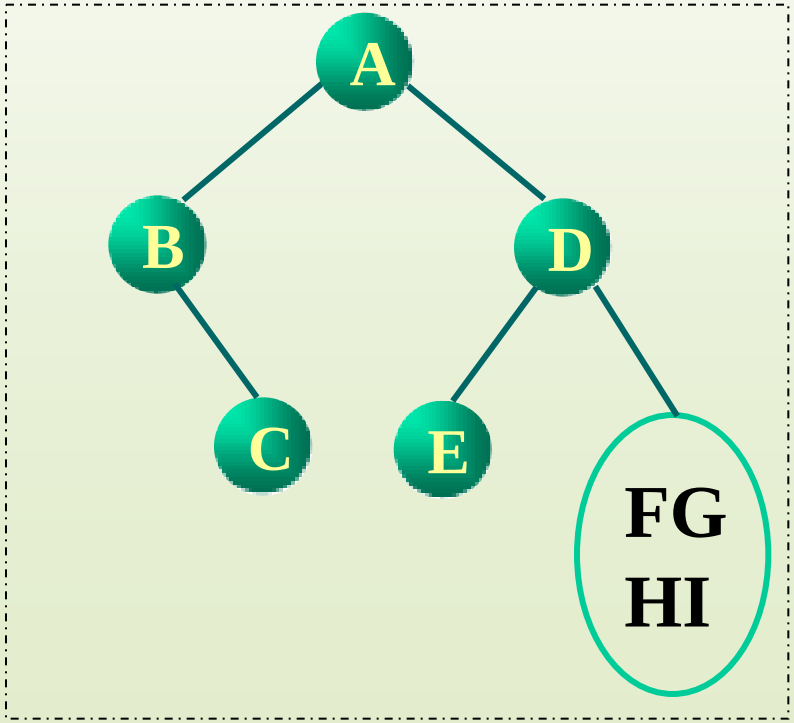



前序：D E F G H I

中序：E D G H F I

前序：F G H I

中序：G H F I



# 5.3 二叉树的逻辑结构

## 二叉树的构造操作

已知一棵二叉树的前序序列和中序序列，构造该二叉树的过程如下：

1. 根据前序序列的第一个元素建立根结点；
2. 在中序序列中找到该元素，确定根结点的左右子树的中序序列；
3. 在前序序列中确定左右子树的前序序列；
4. 由左子树的前序序列和中序序列建立左子树；
5. 由右子树的前序序列和中序序列建立右子树。

# 5.4 二叉树的存储结构及实现

## 顺序存储结构

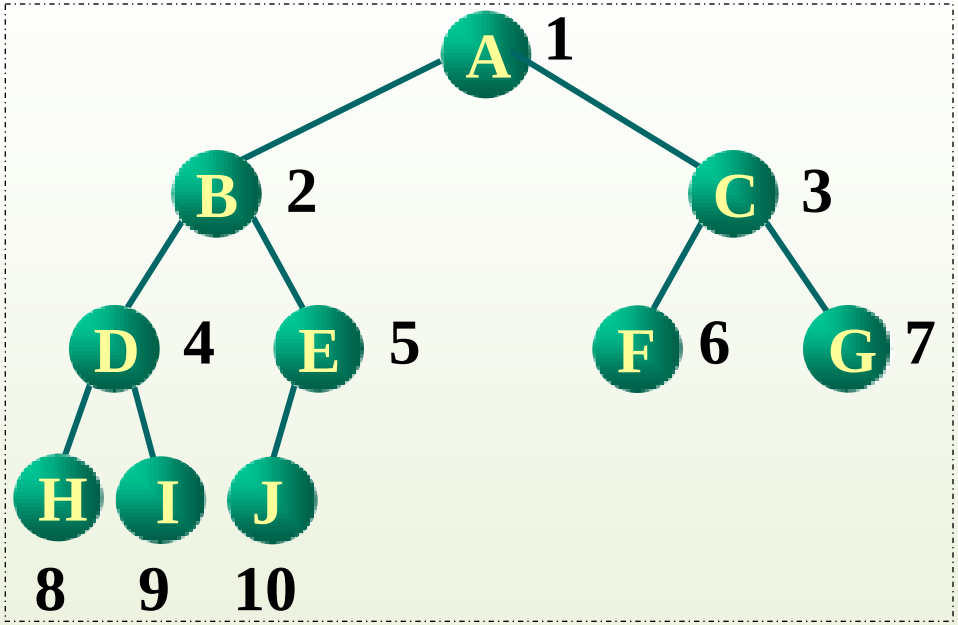
二叉树的顺序存储结构就是用一维数组存储二叉树中的结点，并且结点的存储位置（下标）应能体现结点之间的逻辑关系——父子关系。

如何利用数组下标来反映结点之间的逻辑关系?

完全二叉树和满二叉树中结点的序号可以唯一地反映出结点之间的逻辑关系 。

# 5.4 二叉树的存储结构及实现

完全二叉树的顺序存储

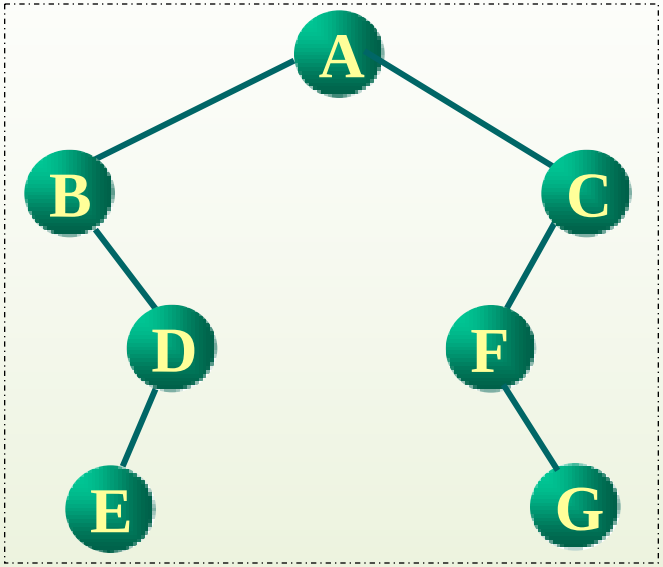



以编号为下标

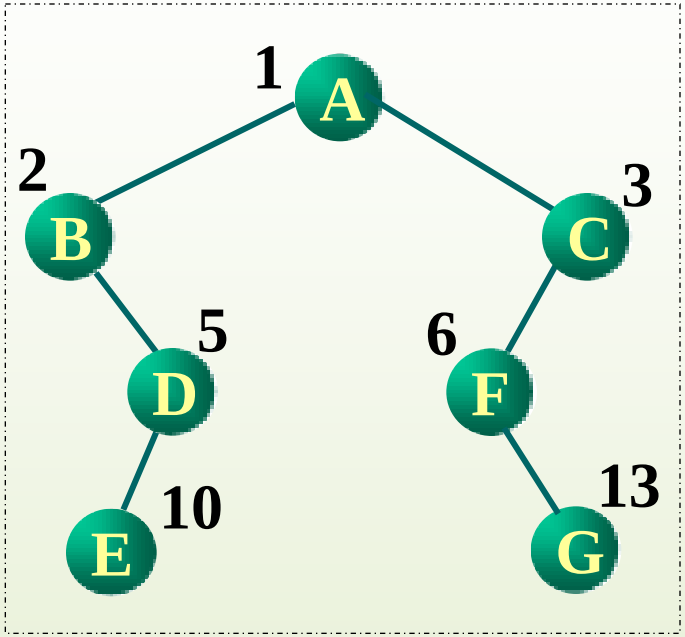
| 数组下标 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| | A | B | C | D | E | F | G | H | I | J |

# 5.4 二叉树的存储结构及实现

二叉树的顺序存储




按照完全二叉树编号




以编号为下标

| 数组下标 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | A | B | C | ∧ | D | F | ∧ | ∧ | ∧ | E | ∧ | ∧ | G |

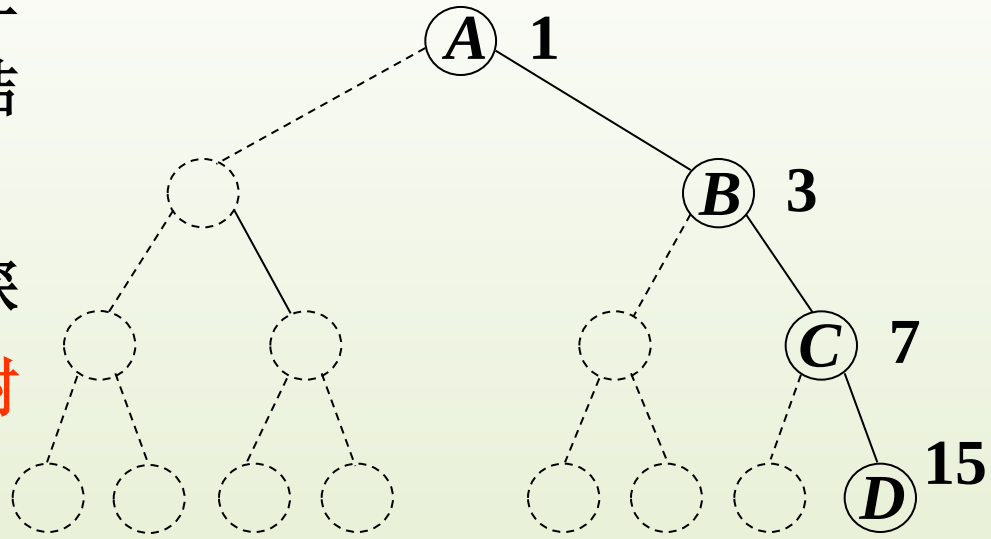
# 5.4 二叉树的存储结构及实现

一棵斜树的顺序存储会怎样呢？

一棵二叉树改造后成完全二叉树形态，需增加很多空结点，造成存储空间的浪费。

在最坏的情况下，一个深度为k且只有k个结点的右斜树确需要$2^k-1$个结点存储空间。

若k=20, 则需1M的空间。




二叉树的顺序存储结构一般仅存储完全二叉树

# 5.4 二叉树的存储结构及实现

## 二叉链表

**基本思想：**令二叉树的每个结点对应一个链表结点，链表结点除了存放与二叉树结点有关的数据信息外，还要设置指示左右孩子的指针。

**结点结构：**


其中，**data**：数据域，存放该结点的数据信息；

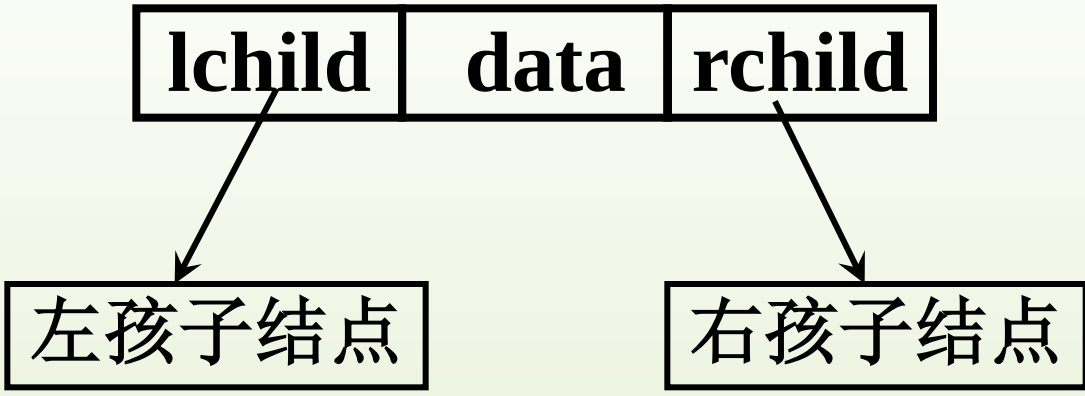
**lchild**：左指针域，存放指向左孩子的指针；

**rchild**：右指针域，存放指向右孩子的指针。

# 5.4 二叉树的存储结构及实现

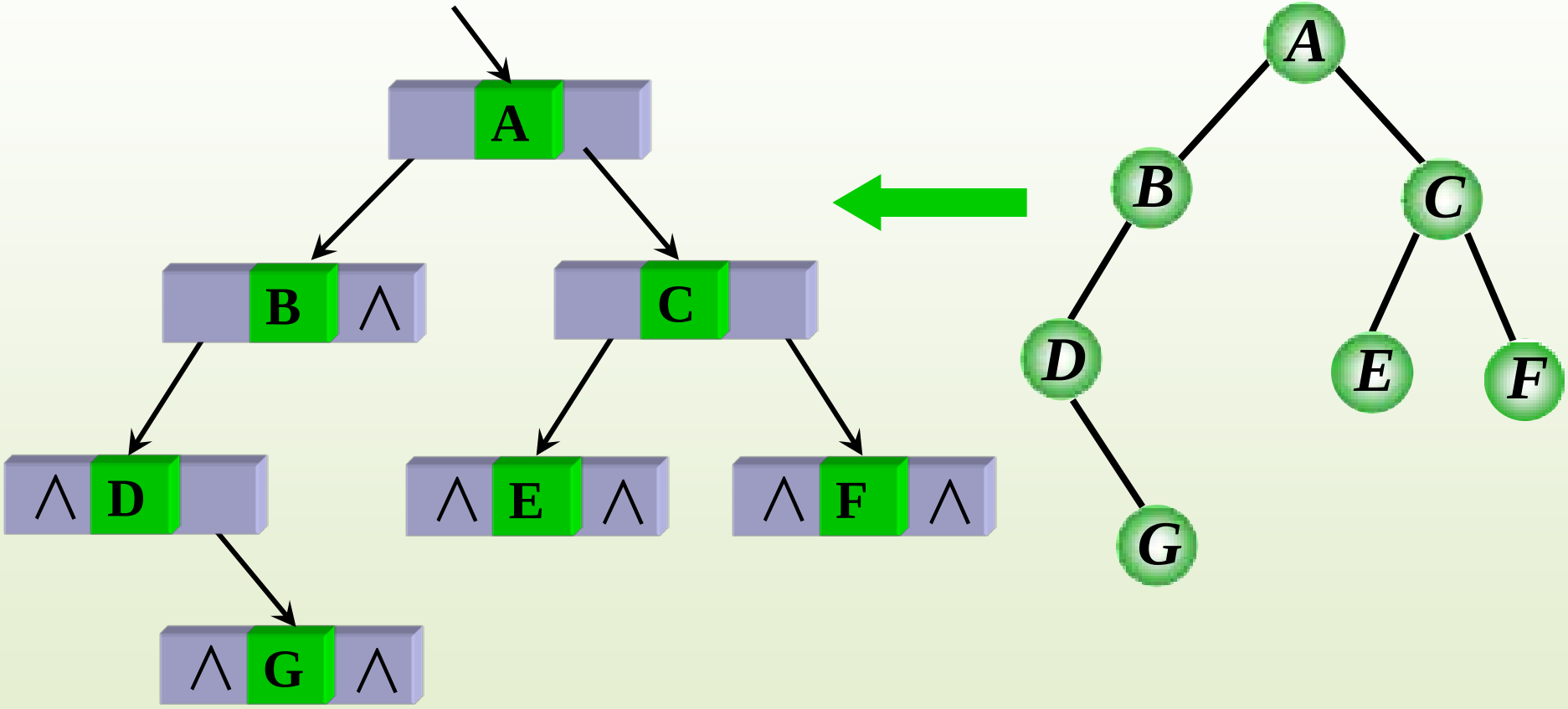
## 二叉链表

```
template <class T>
struct BiNode
{
    T data;
    BiNode<T> *lchild, *rchild;
};
```



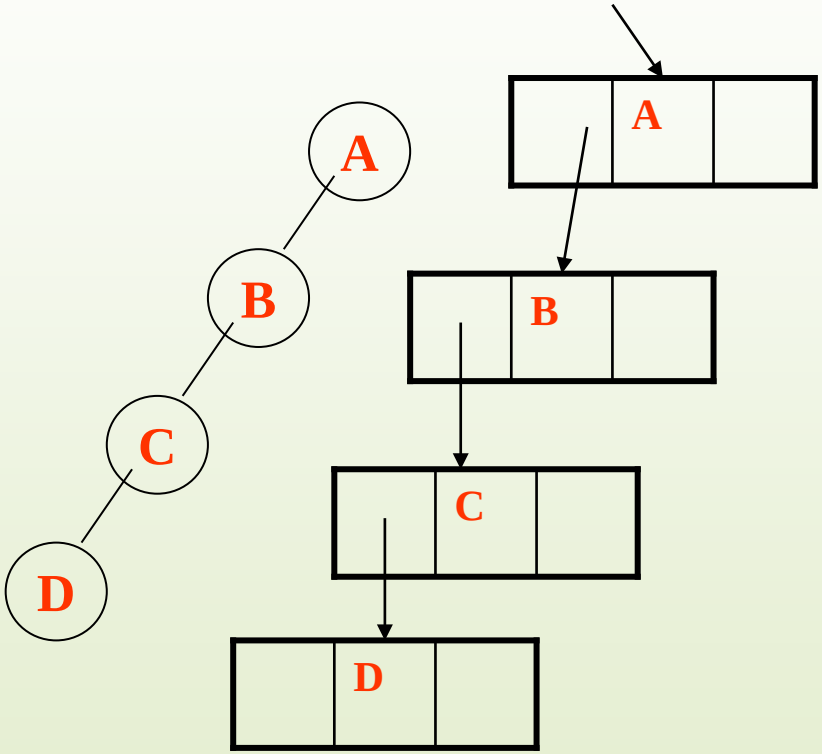
# 5.4 二叉树的存储结构及实现

## 二叉链表

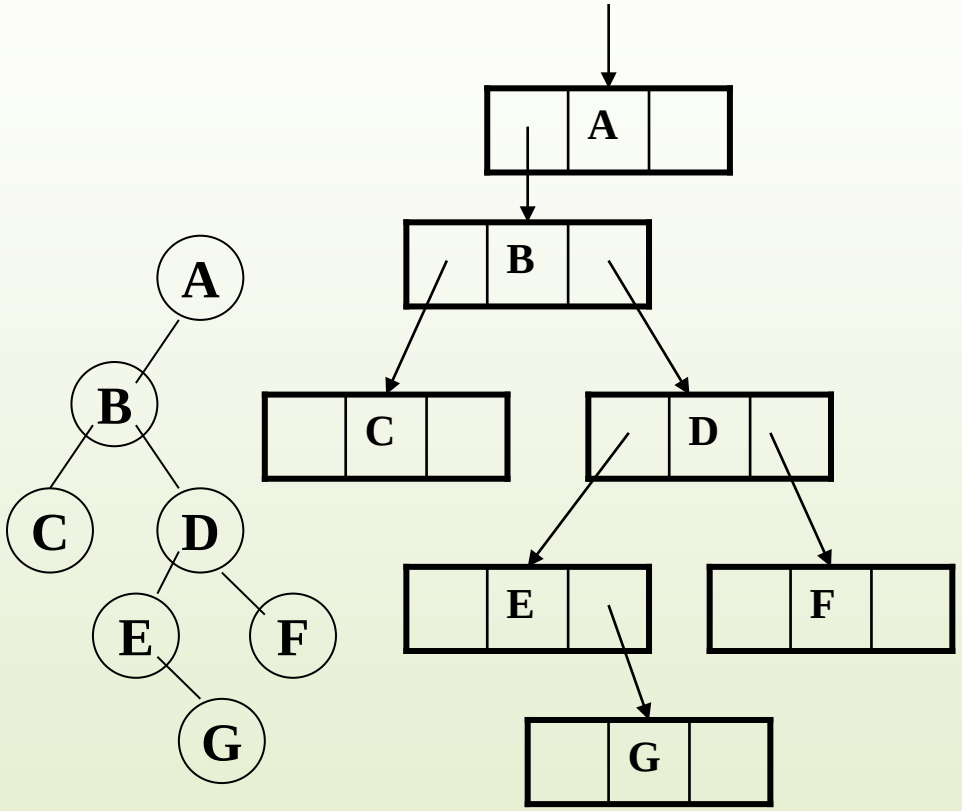


# 5.4 二叉树的存储结构及实现

## 二叉链表




左斜树的二叉链表




二叉链表

具有*n*个结点的二叉链表中，有多少个空指针？

# 5.4 二叉树的存储结构及实现

具有*n*个结点的二叉链表中，有*n*+1个空指针。

证明：

∵二叉链表中每个结点都包含2个指针域----左、右指针域

∴在n个结点的二叉链表中，共有2n个指针域

又∵ n个结点中只有根结点没有指针指向，其余n-1个节点共有n-1个指针指向，即分支个数=n-1

∴ 2n个指针域中有n-1个指针域非空

则有2n-(n-1)=n+1个空指针域

# 5.4 二叉树的存储结构及实现

二叉链表存储结构的类声明

```
template <class T>
class BiTree
{
 public:
      BiTree( ){root=Creat();}
      ~BiTree( ){Release(root);}
       void PreOrder(BiNode<T> *root);
       void InOrder(BiNode<T> *root);
       void PostOrder(BiNode<T> *root);
       void LevelOrder(BiNode<T> *root);
 private:
       BiNode<T> *root;
       BiNode<T> *Creat( );
       void Release(BiNode<T> *root);
};
```

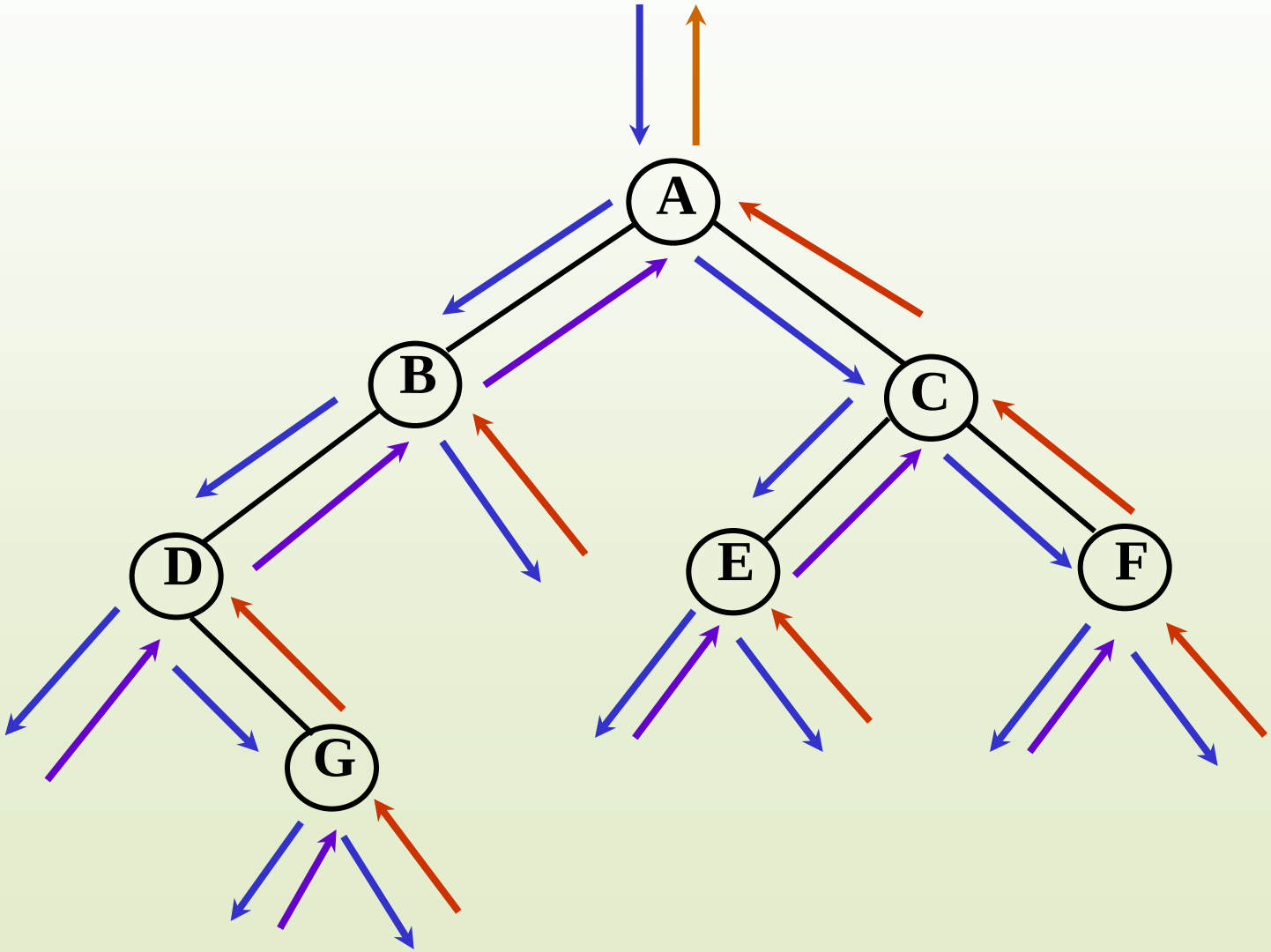
# 5.4 二叉树的存储结构及实现

## 前序遍历——递归算法

```
template   <class T>
void   BiTree<T>::PreOrder（BiNode<T> *root）
{
        if（root ==NULL） return;
        else {
            cout<<root->data;
            PreOrder（root->lchild）;
            PreOrder（root->rchild）;
        }
 }
```

# 5.4 二叉树的存储结构及实现

## 前序遍历算法的执行轨迹



# 5.4 二叉树的存储结构及实现

## 中序遍历——递归算法

```
template <class T>
void BiTree<T>::InOrder（BiNode<T> *root）
{
        if（root==NULL）return;
        else {
             InOrder（root->lchild）;
             cout<<root->data;
             InOrder（root->rchild）;
        }
}
```

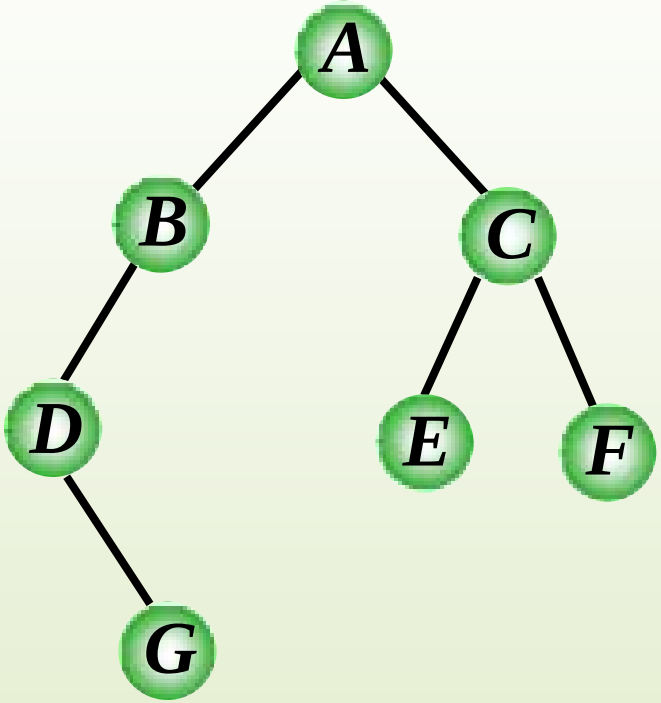
# 5.4 二叉树的存储结构及实现

## 后序遍历——递归算法

```
template <class T>
void BiTree<T>::PostOrder（BiNode<T> *root）
{
    if（root==NULL）return;
    else {
        PostOrder（root->lchild）;
        PostOrder（root->rchild）;
        cout<<root->data;
    }
}
```

# 5.4 二叉树的存储结构及实现

## 层序遍历




**A B C D E F G**

遍历序列：**A B C D E F G**

层序遍历的基本思想：

由于层序遍历结点的顺序是先访问的结点的孩子先于后访问的结点的孩子被访问，与队列操作的特点相同。所以，在进行层序遍历时，设置一个队列,将根结点入队，当队列非空时，循环执行以下三步：

（1） 从队列中取出一个结点，并访问该结点；

（2） 若该结点的左子树非空，将该结点的左子树入队；

（3） 若该结点的右子树非空，将该结点的右子树入队；

# 5.4 二叉树的存储结构及实现

## 层序遍历

1. 队列Q初始化；
2. 如果二叉树非空，将根指针入队；
3. 循环直到队列Q为空
   - 3.1 q=队列Q的队头元素出队；
   - 3.2 访问结点q的数据域；
   - 3.3 若结点q存在左孩子，则将左孩子指针入队；
   - 3.4 若结点q存在右孩子，则将右孩子指针入队；

# 5.4 二叉树的存储结构及实现

## 层序遍历

```
template <class T>
void BiTree<T>::LevelOrder(BiNode<T> *root)
{const int MaxSize = 100;
  int front = 0;
  int rear = 0;  //采用顺序队列，并假定不会发生上溢
  BiNode<T>* Q[MaxSize];
  BiNode<T>* q;
  if (root==NULL) return;
  else{
     Q[rear++] = root;
     while (front != rear)
     {
            q = Q[front++];
            cout<<q->data<<" ";
            if (q->lchild != NULL)    Q[rear++] = q->lchild;
            if (q->rchild != NULL)    Q[rear++] = q->rchild;
     }
       }
}
```

# 5.4 二叉树的存储结构及实现

## 二叉树的建立

遍历是二叉树各种操作的基础，可以在遍历的过程中进行各种操作，例如建立一棵二叉树。

如何由一种遍历序列生成该二叉树？

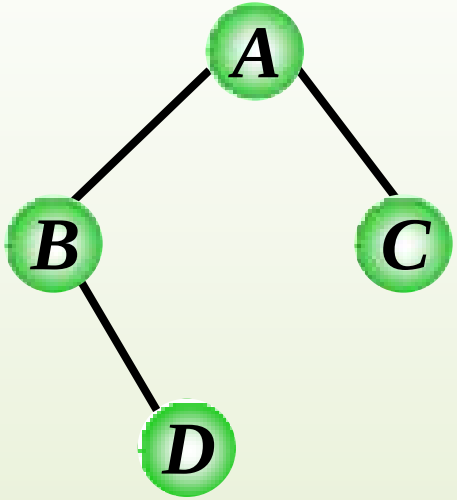
为了建立一棵二叉树，将二叉树中每个结点的空指针引出一个虚结点，其值为一特定值如“#”，以标识其为空，把这样处理后的二叉树称为原二叉树的扩展二叉树。

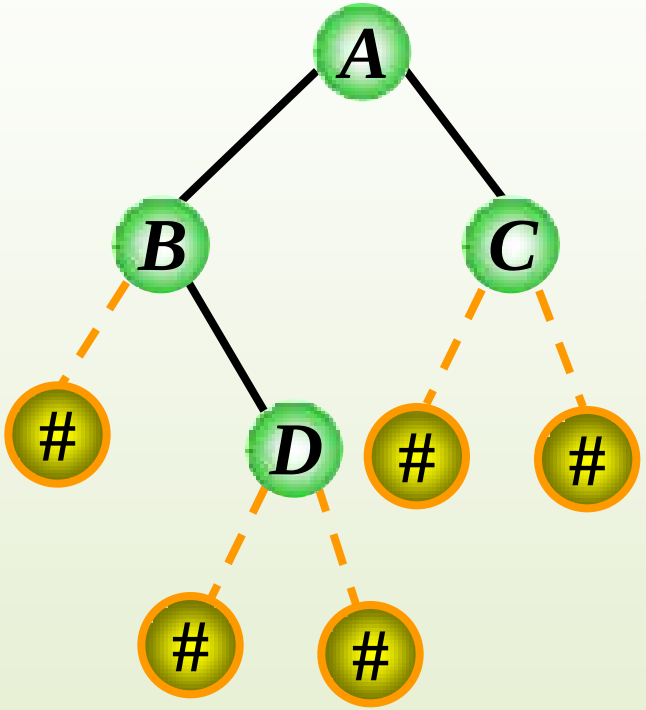
为什么如此处理?

扩展二叉树的一个遍历序列就能唯一确定一棵二叉树

# 5.4 二叉树的存储结构及实现

## 二叉树的建立




扩展二叉树的前序遍历序列:*A B # D # # C # #*

# 5.4 二叉树的存储结构及实现

## 二叉树的建立

设二叉树中的结点均为一个字符。假设扩展二叉树的前序遍历序列由键盘输入，root为指向根结点的指针，二叉链表的建立过程是：

首先输入根结点，若输入的是一个“#”字符，则表明该二叉树为空树，即root=NULL；否则输入的字符应该赋给root->data,，之后依次递归建立它的左子树和右子树 。

# 5.4 二叉树的存储结构及实现

建立二叉树递归算法

```
template<class T>
BiTree<T>::BiTree( )
{root=Creat(root);}
```

```
template <class T>
BiNode<T>  BiTree <T>::Creat( )
{
    cin>>ch;
    if (ch=='# ')     bt=NULL;
    else {
        bt=new BiNode<T>;
        bt->data=ch;
        bt->lchild=Creat();
        bt->rchild=Creat();
    }
    return bt;
}
```

# 5.4 二叉树的存储结构及实现

## 二叉树的销毁

二叉链表是动态存储分配，在析构函数中释放二叉链表的所有结点。

注意：先释放掉某结点的左、右子树才能释放这个结点。

所以，采用后序遍历，对结点的访问就是释放该结点。

# 5.4 二叉树的存储结构及实现

二叉树的销毁递归算法

```
template<class T>
BiTree<T>::~BiTree(void)
{
        Release(root);
}
```

```
template<class T>
void BiTree<T>::Release(BiNode<T>* bt)
{
  if (bt != NULL){
    Release(bt->lchild); //释放左子树
    Release(bt->rchild); //释放右子树
    delete bt;
  }
}
```

# 二叉树算法设计练习

遍历二叉树是二叉树各种操作的基础，遍历算法中对每个结点的访问操作可以是多种形式及多个操作，根据遍历算法的框架，适当修改访问操作的内容，可以派生出很多关于二叉树的应用算法。

```
void  InOrder（BiNode<T> *root）
{
        if（root==NULL）return;
        else {
             InOrder（root->lchild）;
             cout<<root->data;
             InOrder（root->rchild）;
        }
}
```

# 二叉树算法设计练习

设计算法求二叉树的结点个数。

```
template <class T>
int Count(BiNode<T> *root)
{
    if (root==NULL) return 0;
     else
     return 1+ Count(root->lchild)+ Count(root->rchild);
}
```

# 二叉树算法设计练习

设计算法按前序次序打印二叉树中的叶子结点。

```
template <class T>
void PreOrderleaf(BiNode<T> *root)
{
   if (root !=NULL) {
      if (root->lchild! =NULL&& root->rchild ! =NULL)
            cout<<root->data;
      PreOrderleaf(root->lchild);
      PreOrderleaf(root->rchild);
  }
}
```

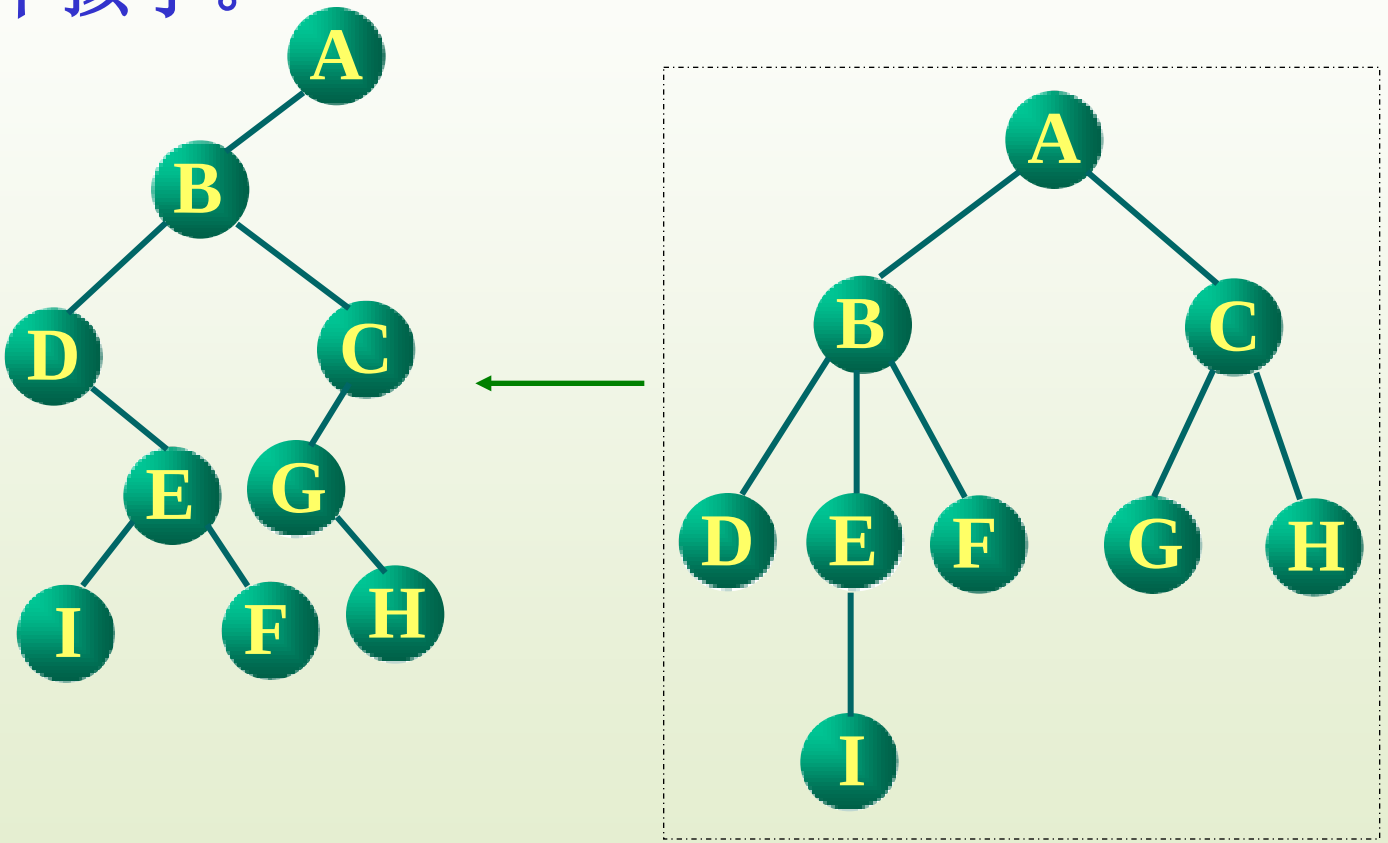
# 二叉树算法设计练习

设计算法求二叉树的深度。

```
template <class T>
int  Depth(BiNode<T> *root)
{
    if (root==NULL) return 0;
    else {
        int hl=Depth(root->lchild);
        int hr=Depth(root ->rchild);
        return hl>hr?hl+1:hr+1;
    }
}
```

# 二叉树算法设计练习

用孩子-兄弟表示法表示树，设计算法求树中结点 *x* 的第 *i* 个孩子。




| firstchild | data | rightsib |
|---|---|---|

# 二叉树算法设计练习

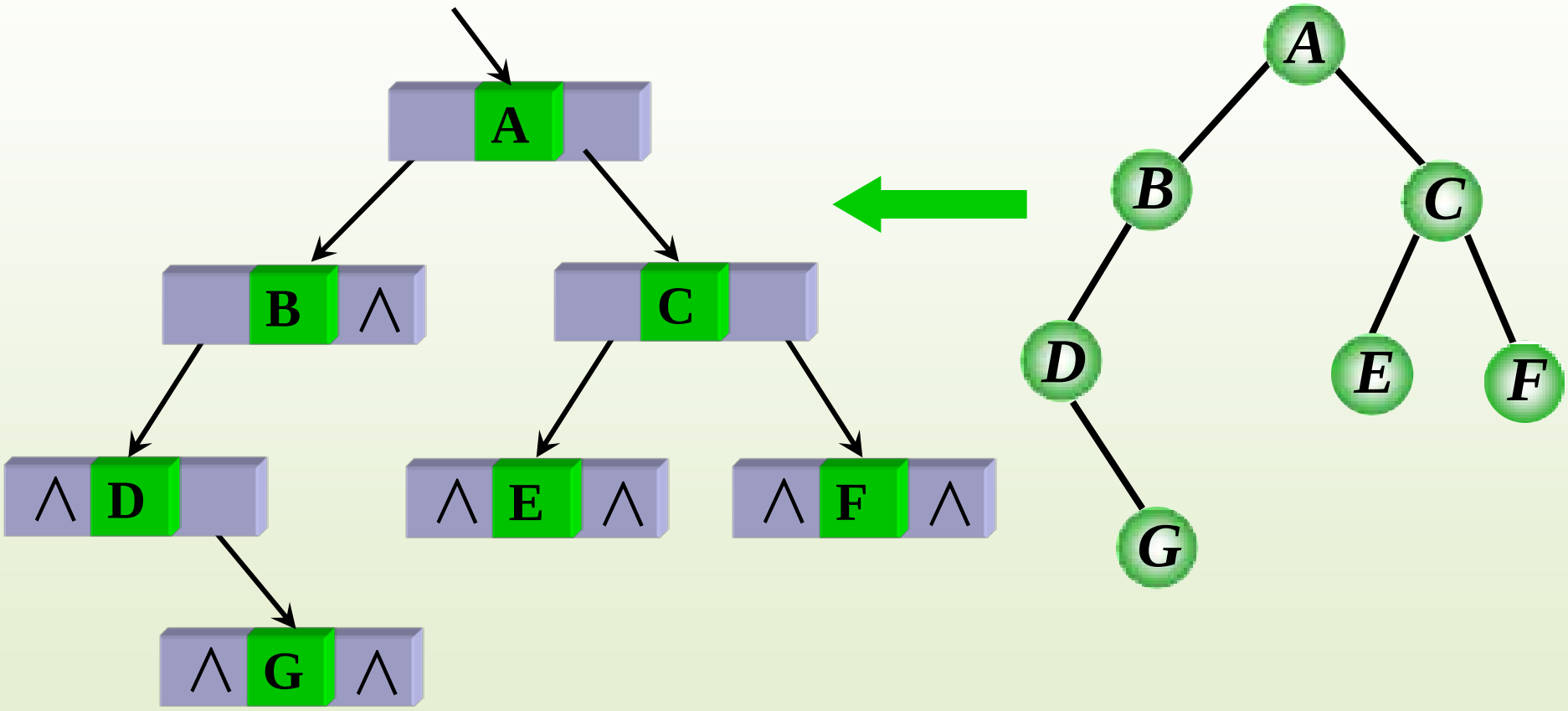
用孩子-兄弟表示法表示树，设计算法求树中结点 *x* 的第 *i* 个孩子。

```
template <class T>
TNode *Search(TNode *root, T x, int i)
{
    if (root->data= =x) {
        j=1;
        p=root->firstchild;
        while (p!=NULL && j<i)
        {
            j++;
            p=p->rightsib;
        }
        if (p) return p;
        else return NULL;
    }
    Search(root->firstchild, x, i);
    Search(root->rightsib, x, i);
}
```

```
template <class T>
struct TNode
{
  T data;
  TNode *firstchild, *rightsib;
};
```

# 5.4 二叉树的存储结构及实现

## 三叉链表




在二叉链表中，如何求某结点的双亲？

# 5.4 二叉树的存储结构及实现

## 三叉链表

在二叉链表的基础上增加了一个指向双亲的指针域。

### 结点结构

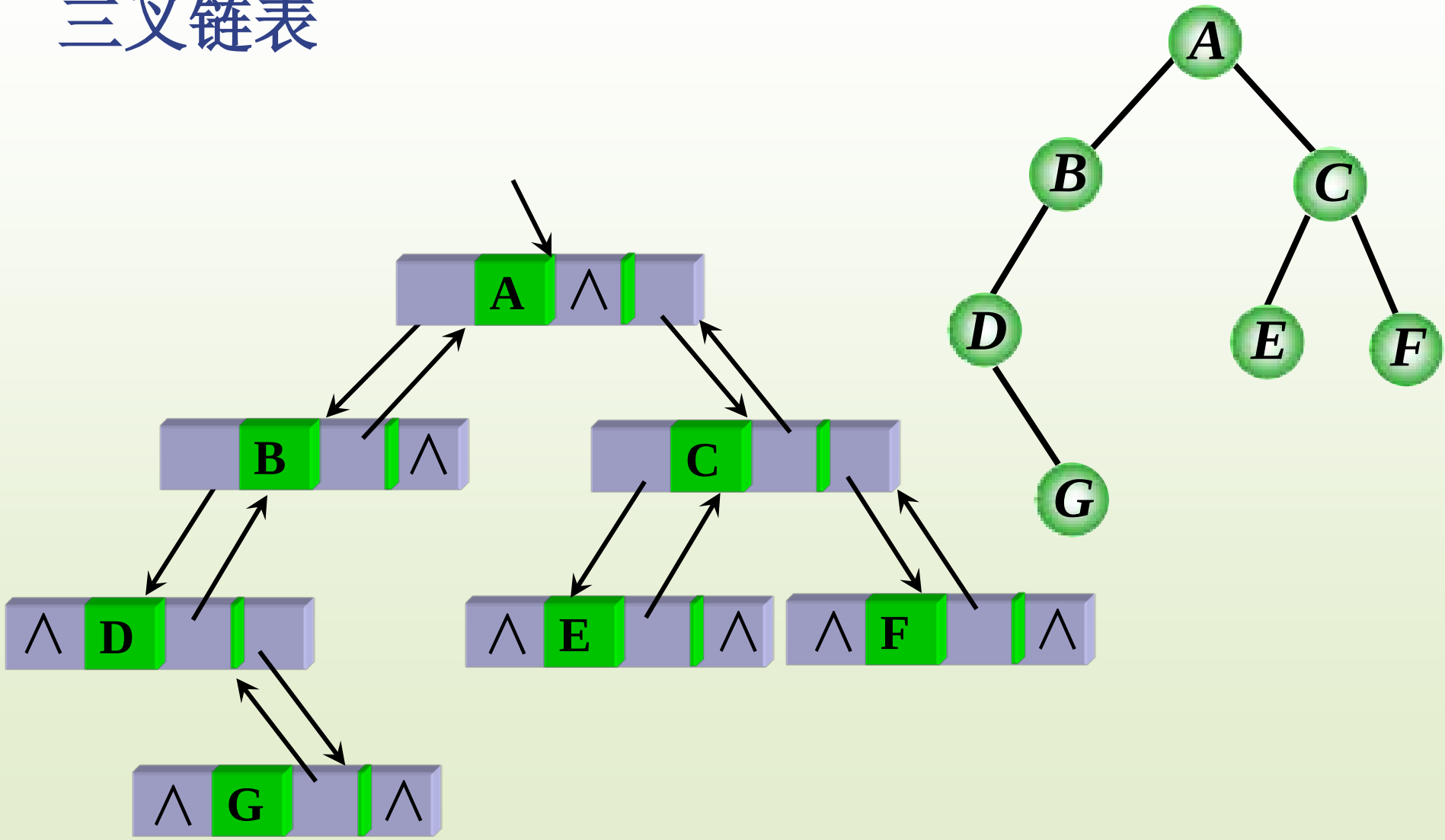
| lchild | data | parent | rchild |
|---|---|---|---|

其中：data、lchild和rchild三个域的含义同二叉链表的结点结构；

parent域为指向该结点的双亲结点的指针。

# 5.4 二叉树的存储结构及实现

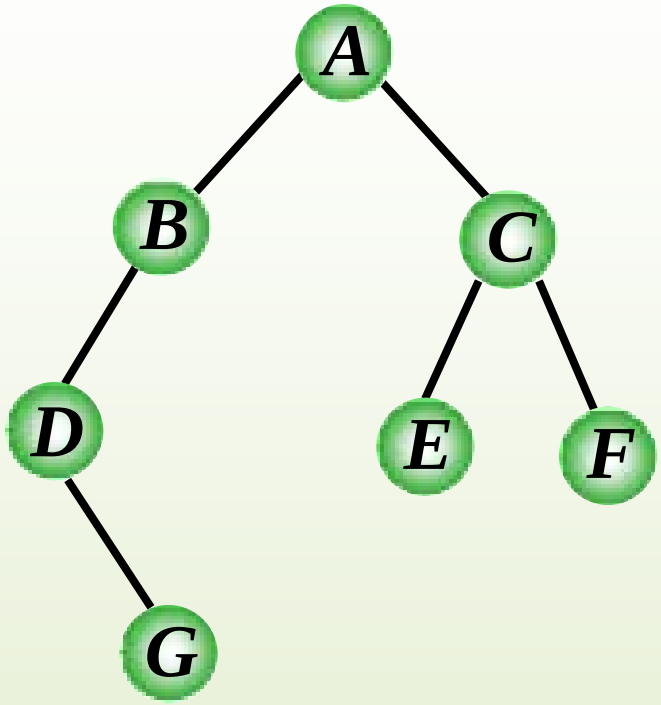
## 三叉链表



# 5.4 二叉树的存储结构及实现

## 三叉链表的静态链表形式

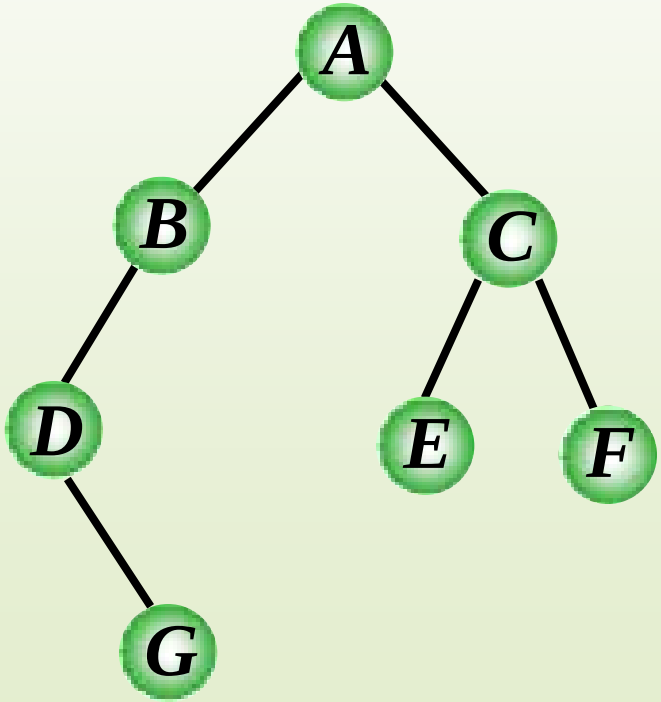
| | data | parent | lchild | rchild |
|---|---|---|---|---|
| 0 | A | -1 | 1 | 2 |
| 1 | B | 0 | 3 | -1 |
| 2 | C | 0 | 4 | 5 |
| 3 | D | 1 | -1 | 6 |
| 4 | E | 2 | -1 | -1 |
| 5 | F | 2 | -1 | -1 |
| 6 | G | 3 | -1 | -1 |



# 5.4 二叉树的存储结构及实现

## 线索链表

如何保存二叉树的某种遍历序列？




中序遍历序列：*D G B A E C F*

顺序存储

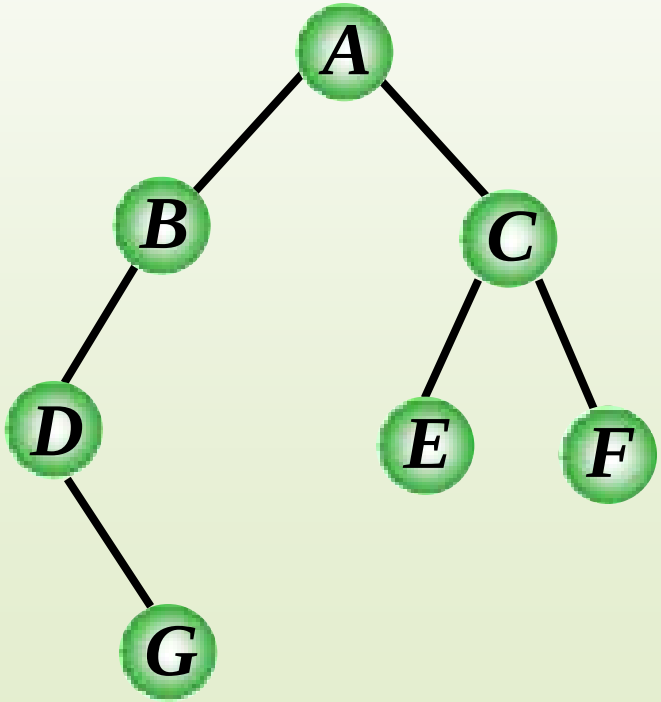
| *D* | *G* | *B* | *A* | *E* | *C* | *F* |
|---|---|---|---|---|---|---|

如果二叉树不改变，如何保存？

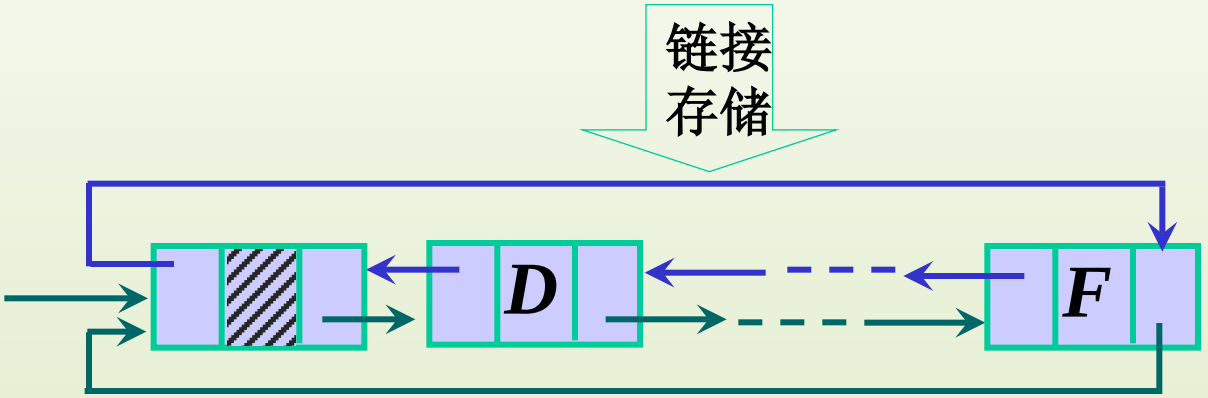
# 5.4 二叉树的存储结构及实现

## 线索链表

如何保存二叉树的某种遍历序列？




中序遍历序列：*D G B A E C F*

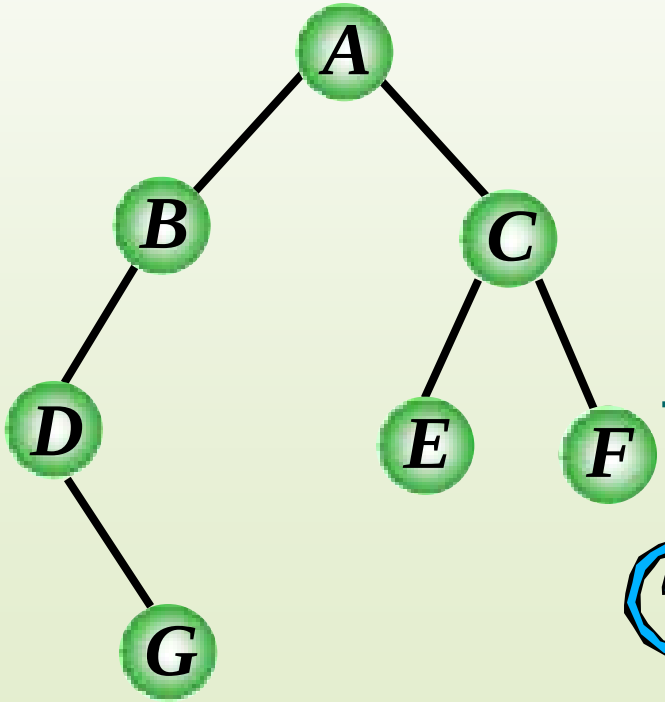



如果二叉树改变，如何保存？

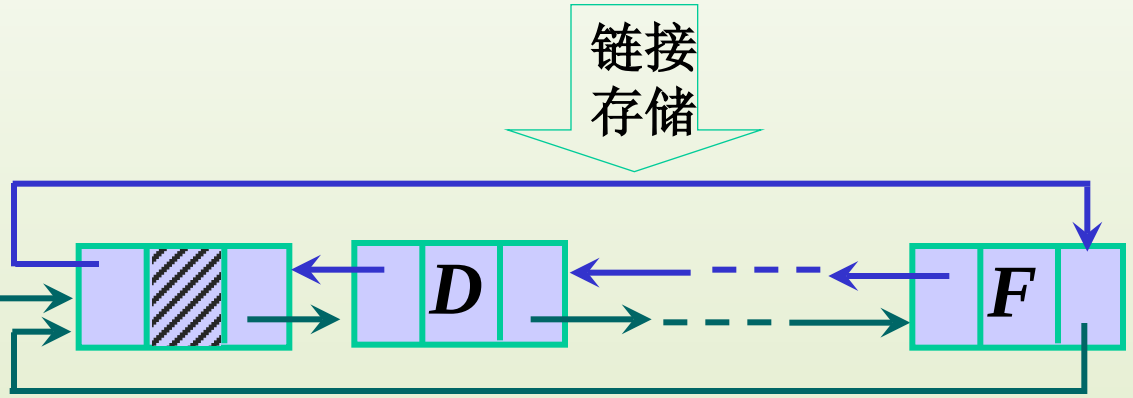
# 5.4 二叉树的存储结构及实现

## 线索链表

如何保存二叉树的某种遍历序列？




中序遍历序列：*D G B A E C F*




如何将二叉链表与中序链表结合？

# 5.4 二叉树的存储结构及实现

## 线索链表

如何保存二叉树的某种遍历序列？

将二叉链表中的空指针域指向其前驱结点和后继结点

**线索：**将二叉链表中的空指针域指向前驱结点和后继结点的指针被称为线索；

**线索化：**使二叉链表中结点的空链域存放其前驱或后继信息的过程称为线索化；

**线索二叉树：**加上线索的二叉树称为线索二叉树。

# 5.4 二叉树的存储结构及实现

## 线索链表

结点结构

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

ltag= { 0: lchild指向该结点的左孩子
1: lchild指向该结点的前驱结点

rtag= { 0: rchild指向该结点的右孩子
1: rchild指向该结点的后继结点

# 5.4 二叉树的存储结构及实现

## 线索链表

结点结构


```
enum flag {Child, Thread};
template  <class T>
struct ThrNode
{
    T data;
    ThrNode<T>  *lchild, *rchild;
    flag ltag, rtag;
};
```

# 5.4 二叉树的存储结构及实现

## 线索二叉树

二叉树的遍历方式有4种，故有4种意义下的前驱和后继，相应的有4种线索二叉树：

(1) 前序线索二叉树

(2) 中序线索二叉树

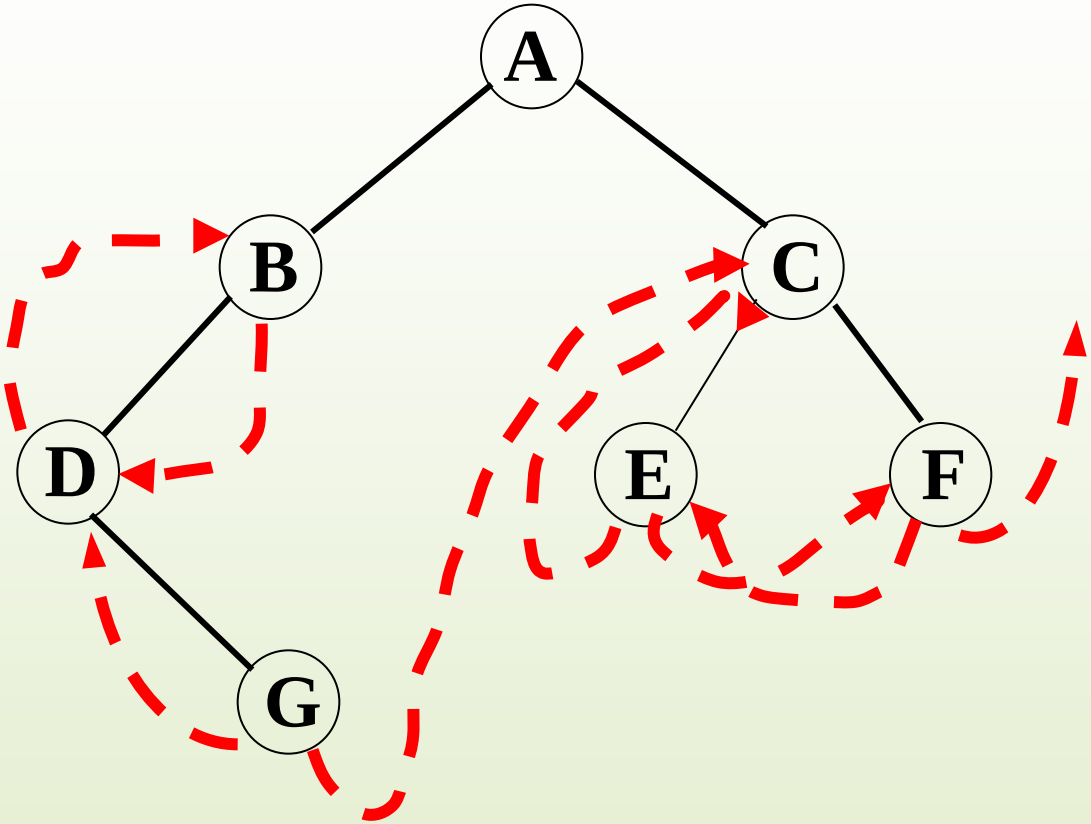
(3) 后序线索二叉树

(4) 层序线索二叉树

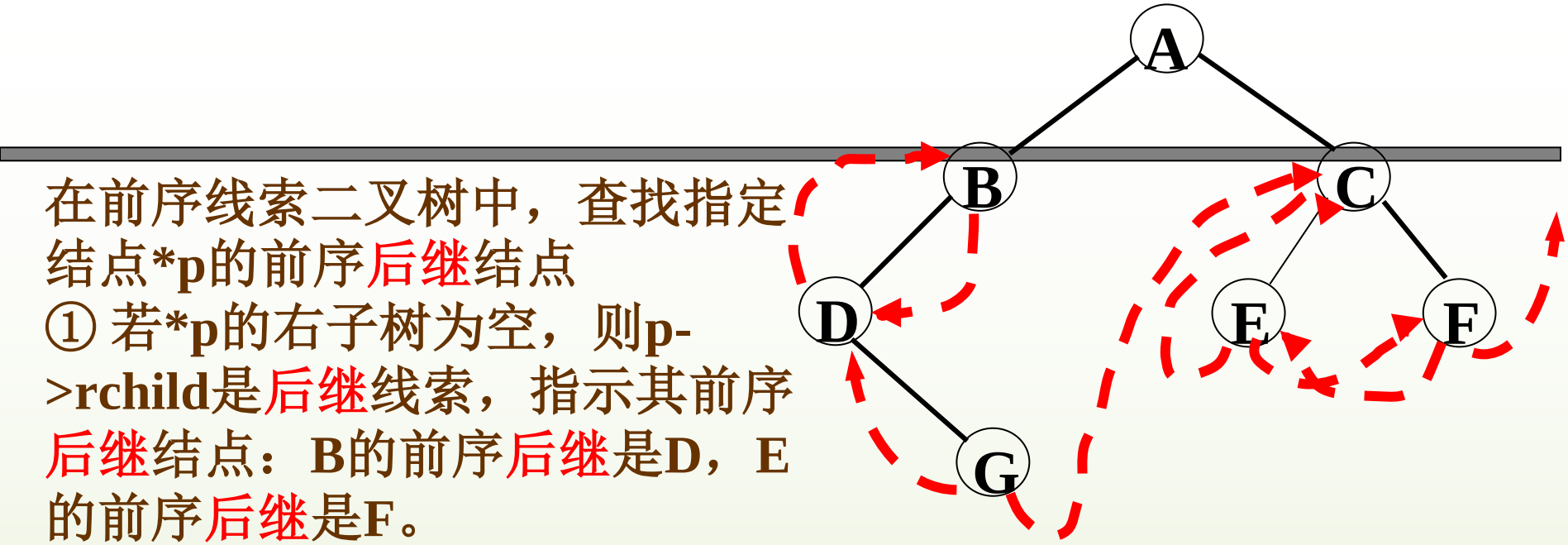
# 5.4 二叉树的存储结构及实现

## 线索二叉树

### 前序线索二叉树




前序序列： *A B D G C E F*




在前序线索二叉树中，查找指定结点*p的前序后继结点

① 若*p的右子树为空，则p->rchild是后继线索，指示其前序后继结点：B的前序后继是D，E的前序后继是F。

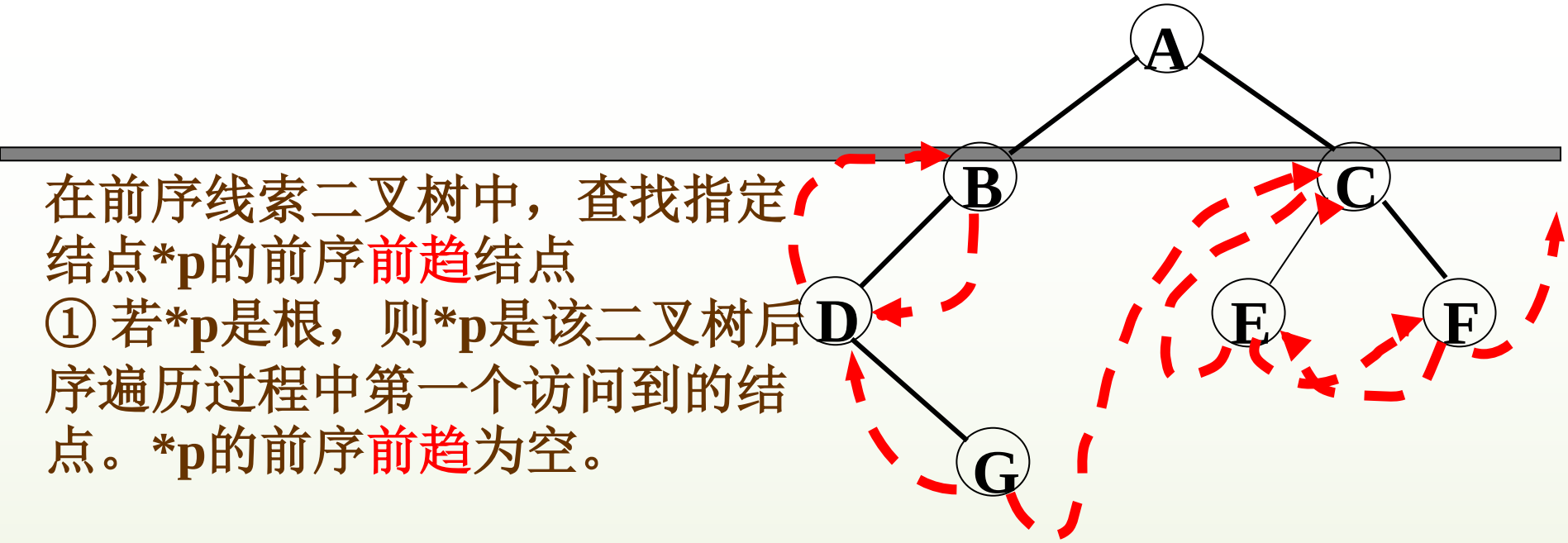
前序序列： *A B D G C E F*

② 若*p的右子树非空，则p->rchild不是后继线索。由于前序遍历时，根是在遍历其左右子树之前被访问的，故*p的前序后继必是两子树中第一个遍历结点。

- 当*p的左子树非空时，*p的左孩子必是其前序后继： A的前序后继是B;
- 当*p无左子树时，*p的前序后继必是其右孩子：D的前序后继是G 。

在前序线索二叉树中，查找指定结点*p的前序前趋结点

① 若*p是根，则*p是该二叉树后序遍历过程中第一个访问到的结点。*p的前序前趋为空。




前序序列：*A B D G C E F*

② 若*p是其双亲的右孩子，则*p的前序前趋结点就是其双亲的左子树中最后一个前序遍历到的结点，它是该子树中“最右下的叶结点”：C的后序后继是双亲A的左子树中最右下的叶结点G；

③ 若*p是其双亲的左孩子，但*p无右兄弟，*p的前序前趋结点是其双亲结点： D的前序前趋是B。

④ 若*p是其双亲的左孩子，但*p有右兄弟，则*p的前序前趋是其双亲结点： B的前序前趋是A，E的前序前趋是C。

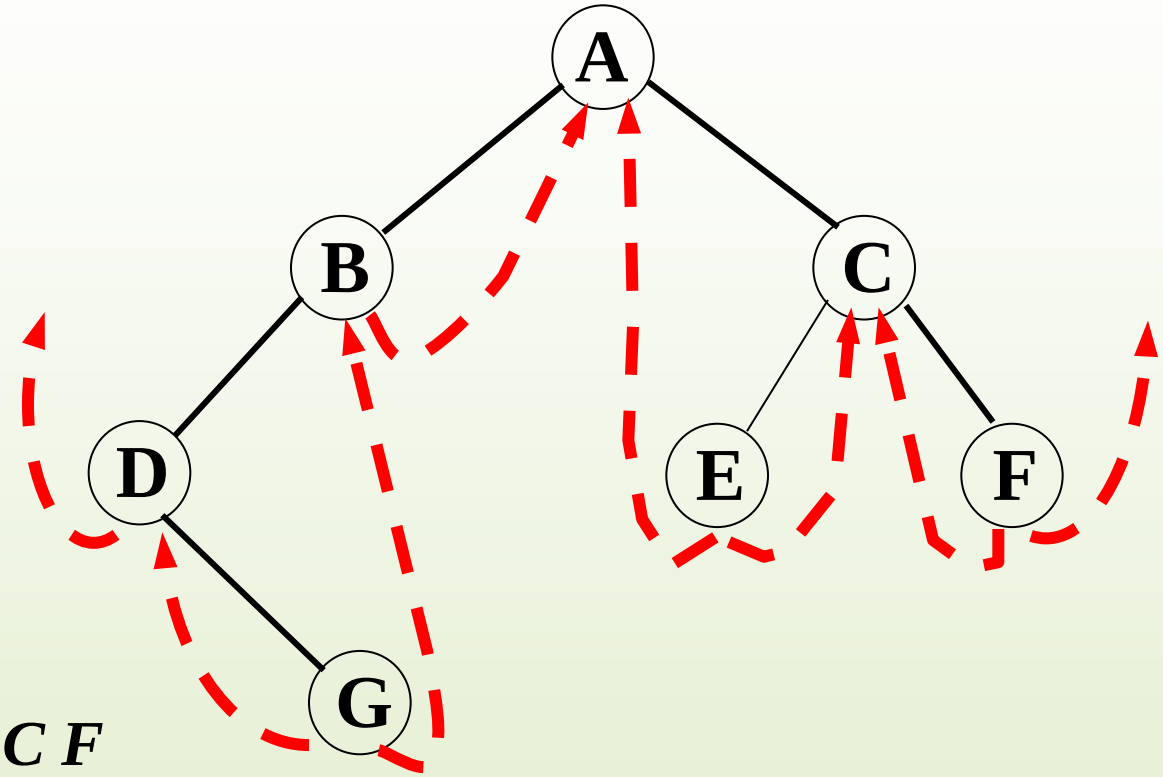
由上述讨论可知：

在前序线索二叉树中，

- 找某一点*p的前序后继，仅从*p出发就可以找到；
- 找其前序前趋必须知道*p的双亲结点。
- 当树中结点未设双亲指针时，要进行从根开始的前序遍历才能找到结点*p的前序前趋。

# 线索二叉树

## 中序线索二叉树




中序序列：*D G B A E C F*

中序线索化前驱和后继都不需要求双亲，但是都不很直接。

结点p的前驱：　（非线索）结点p的左子树中最右下结点

结点p的后继：　（非线索）结点p的右子树中最左下结点

# 在中序线索二叉树中，查找结点*p的中序后继结点

分两种情形：

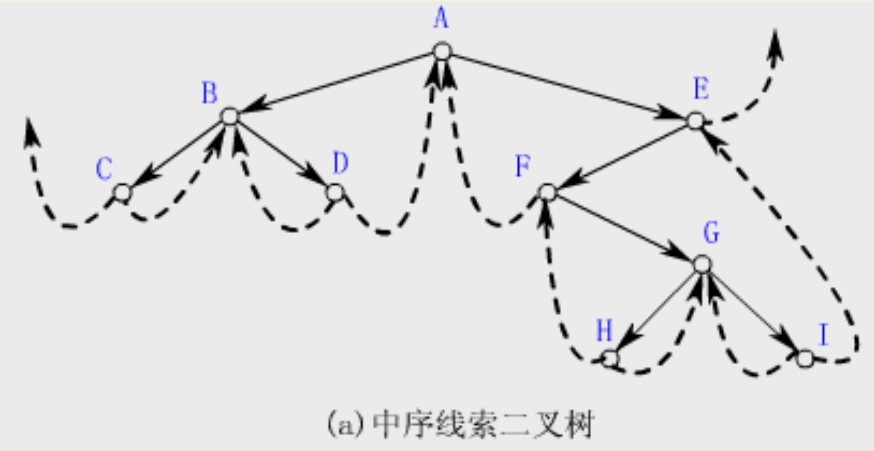
① 若*p的右子树空(即p->rtag为Thread)，则p->rchild为右线索，直接指向*p的中序后继：结点D的中序后继是A。

② 若*p的右子树非空(即p->rtag为Link)，则*p的中序后继必是其右子树中第一个中序遍历到的结点。也就是从*p的右孩子开始，沿该孩子的左链往下查找，直至找到一个没有左孩子的结点为止，该结点是*p的右子树中“最左下”的结点，即*P的中序后继结点。

A的中序后继是F，它有右孩子；

F的中序后继是H，它无右孩子；

B的中序后继是D，它是B的右孩子。



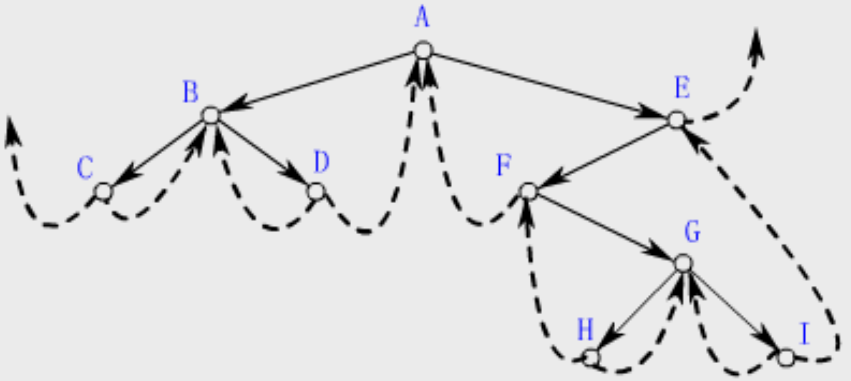
(a) 中序线索二叉树

# 在中序线索二叉树中，查找结点*p的中序前趋结点

中序是一种对称序，故在中序线索二叉树中查找结点*p的中序前趋结点与找中序后继结点的方法完全对称。具体情形如下：

① 若*p的左子树为空，则p->1child为左线索，直接指向*p的中序前趋结点：F结点的中序前趋结点是A；

② 若*p的左子树非空，则从*p的左孩子出发，沿右指针链往下查找，直到找到一个没有右孩子的结点为止。该结点是*p的左子树中“最右下”的结点，它是*p的左子树中最后一个中序遍历到的结点，即*p的中序前趋结点：结点E左子树非空，其中序前趋结点是I。



(a) 中序线索二叉树

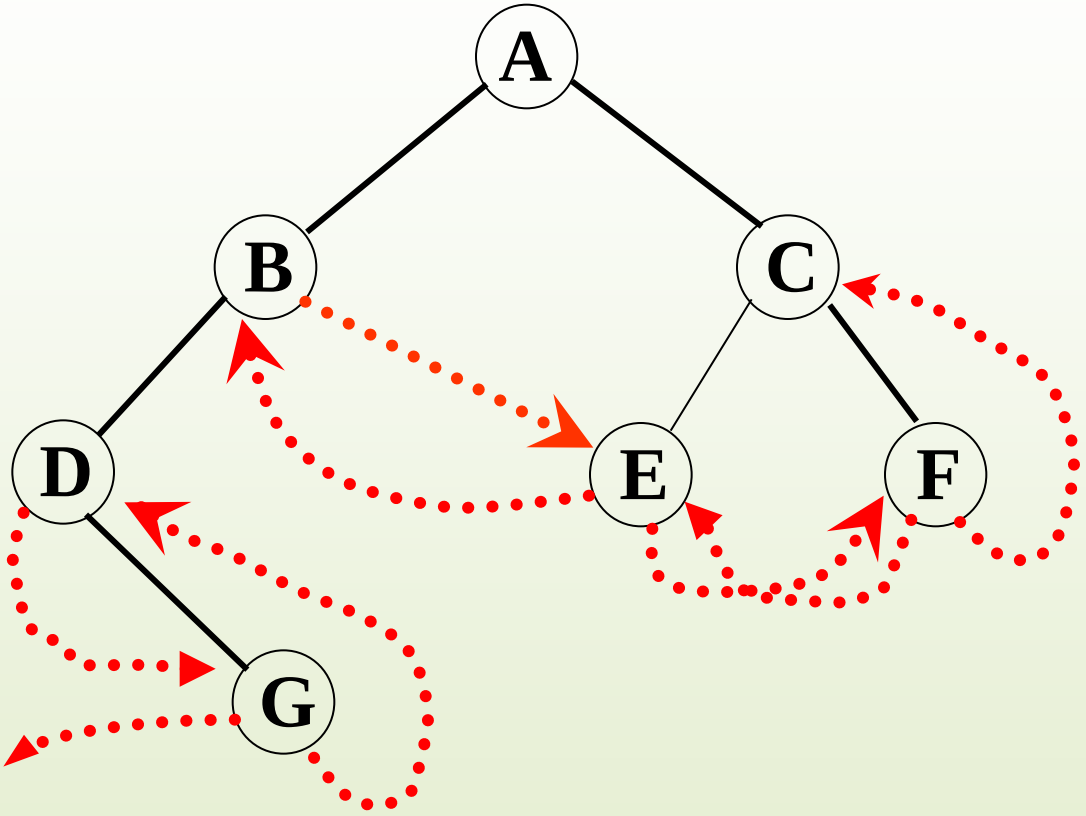
由上述讨论可知：

对于非线索二叉树，仅从*p出发无法找到其中序前趋(或中序后继)，而必须从根结点开始中序遍历，才能找到*p的中序前趋(或中序后继)。

线索二叉树中的线索使得查找中序前趋和中序后继变得简单有效。

# 线索二叉树

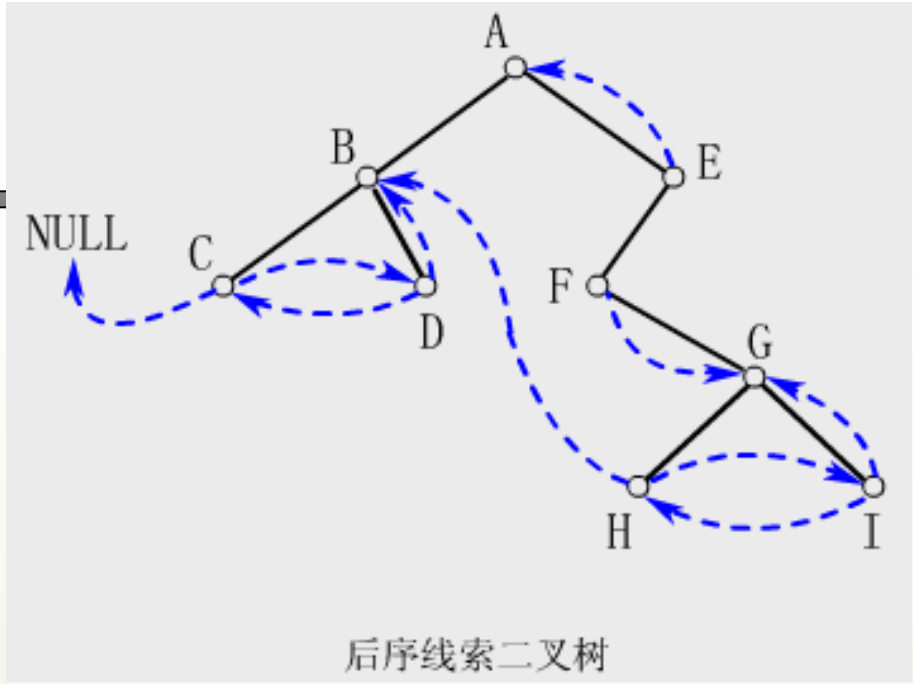
## 后序线索二叉树




后序序列： ***G D B E F C A***

在后序线索二叉树中，查找指定结点*p的后序前趋结点

① 若*p的左子树为空，则p->lchild是前趋线索，指示其后序前趋结点：H的后序前趋是B，F的后序前趋是G。



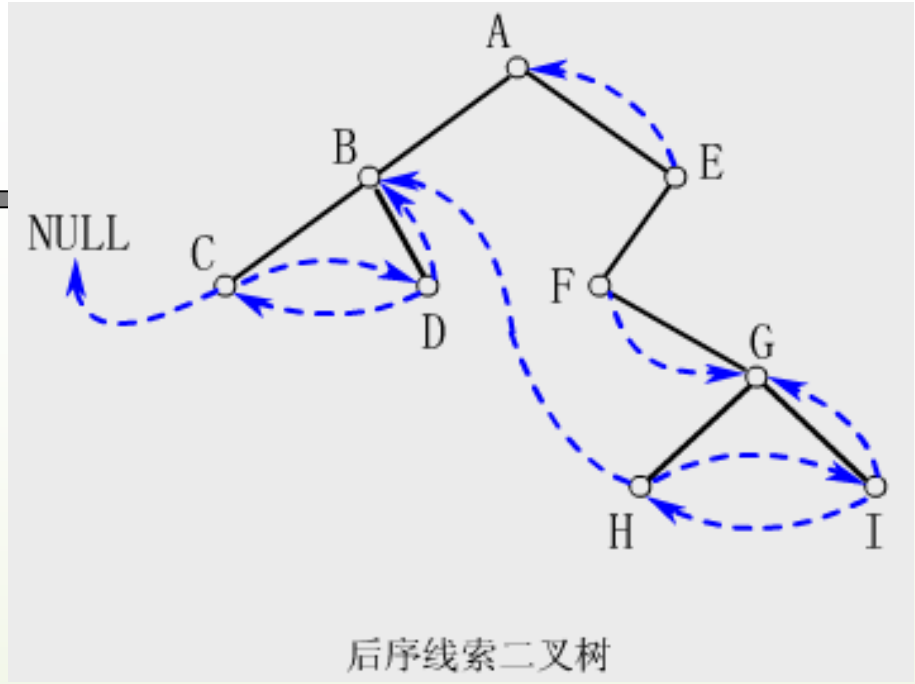
后序线索二叉树

② 若*p的左子树非空，则p->lchild不是前趋线索。由于后序遍历时，根是在遍历其左右子树之后被访问的，故*p的后序前趋必是两子树中最后一个遍历结点。

- 当*p的右子树非空时，*p的右孩子必是其后序前趋： A的后序前趋是E;
- 当*p无右子树时，*p的后序前趋必是其左孩子：E的后序前趋是F 。

在后序线索二叉树中，查找指定结点*p的后序后继结点

① 若*p是根，则*p是该二叉树后序遍历过程中最后一个访问到的结点。*p的后序后继为空



后序线索二叉树

② 若*p是其双亲的右孩子，则*p的后序后继结点就是其双亲结点：E的后序后继是A。

③ 若*p是其双亲的左孩子，但*P无右兄弟，*p的后序后继结点是其双亲结点：F的后序后继是E。

④ 若*p是其双亲的左孩子，但*p有右兄弟，则*p的后序后继是其双亲的右子树中第一个后序遍历到的结点，它是该子树中“最左下的叶结点”：B的后序后继是双亲A的右子树中最左下的叶结点H

由上述讨论中可知：在后序线索二叉树中，仅从*p出发就能找到其后序前趋结点；要找*p的后序后继结点，仅当*p的右子树为空时，才能直接由*p的右线索p->rchild得到。否则必须知道*p的双亲结点才能找到其后序后继。

因此，如果线索二叉树中的结点没有指向其双亲结点的指针，就可能要从根开始进行后序遍历才能找到结点*P的后序后继。由此，线索对查找指定结点的后序后继并无多大帮助。

- 前序线索化能依次找到后继，但是前驱需要求双亲；
- 中序线索化前驱和后继都不需要求双亲，但是都不很直接；
- 后序线索化能依次找到前驱，但是后继需要求双亲。

可见，中序线索化是最佳的选择。

中序线索链表类的声明

```
template <class T>
class InThrBiTree
{
  public:
      InThrBiTree(ThrNode<T> * root);
      ~ InThrBiTree( );
      ThrNode *Next(ThrNode<T> *p);
      void InOrder(ThrNode<T> *root);
  private:
      ThrNode<T> *root;
      void Creat(ThrNode<T> *root);
      void ThrBiTree(ThrNode<T> *root);
};
```

# 中序线索链表的建立——构造函数

分析：建立线索链表，实质上就是将二叉链表中的空指针改为指向前驱或后继的线索，而前驱或后继的信息只有在遍历该二叉树时才能得到。

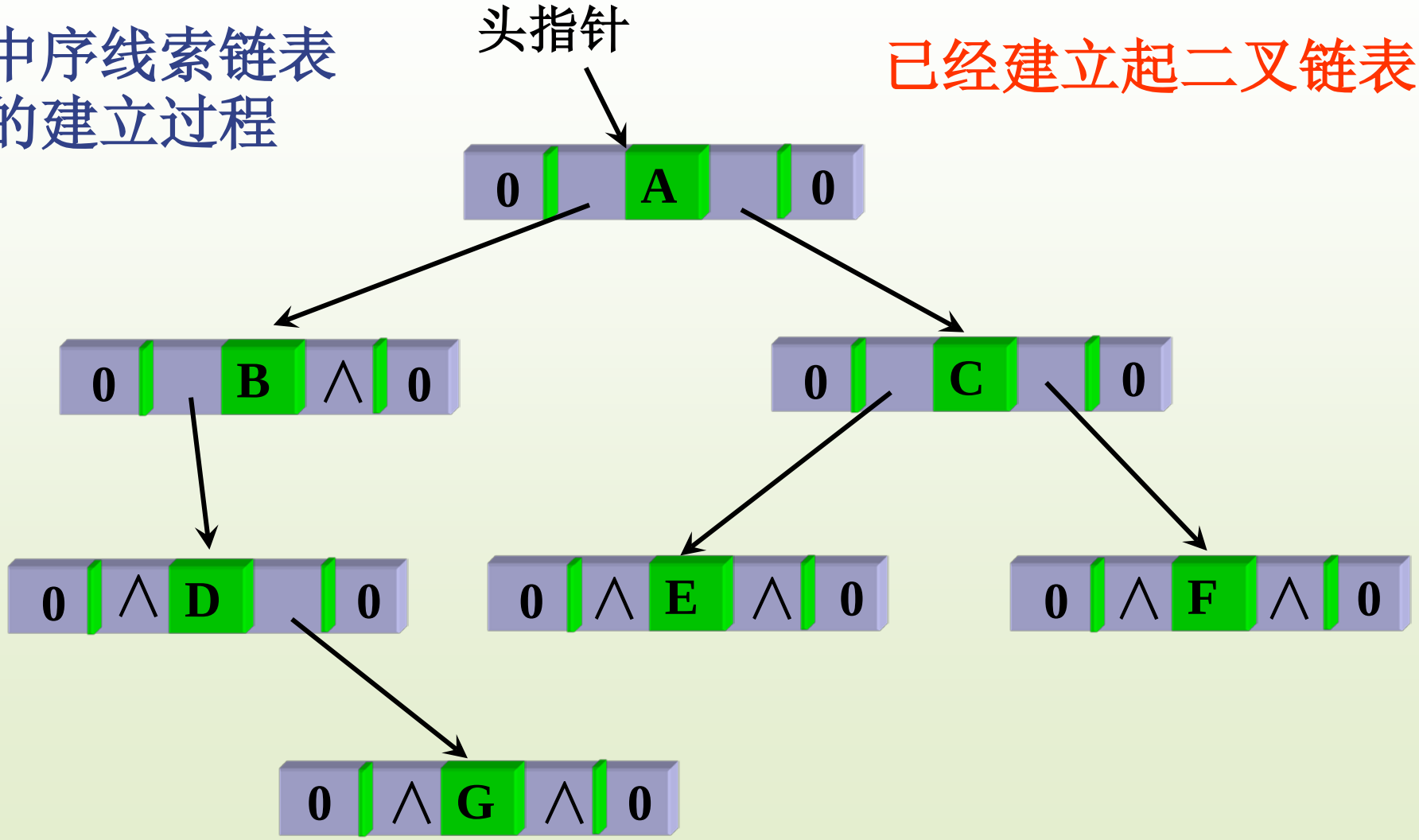
建立二叉链表

↓

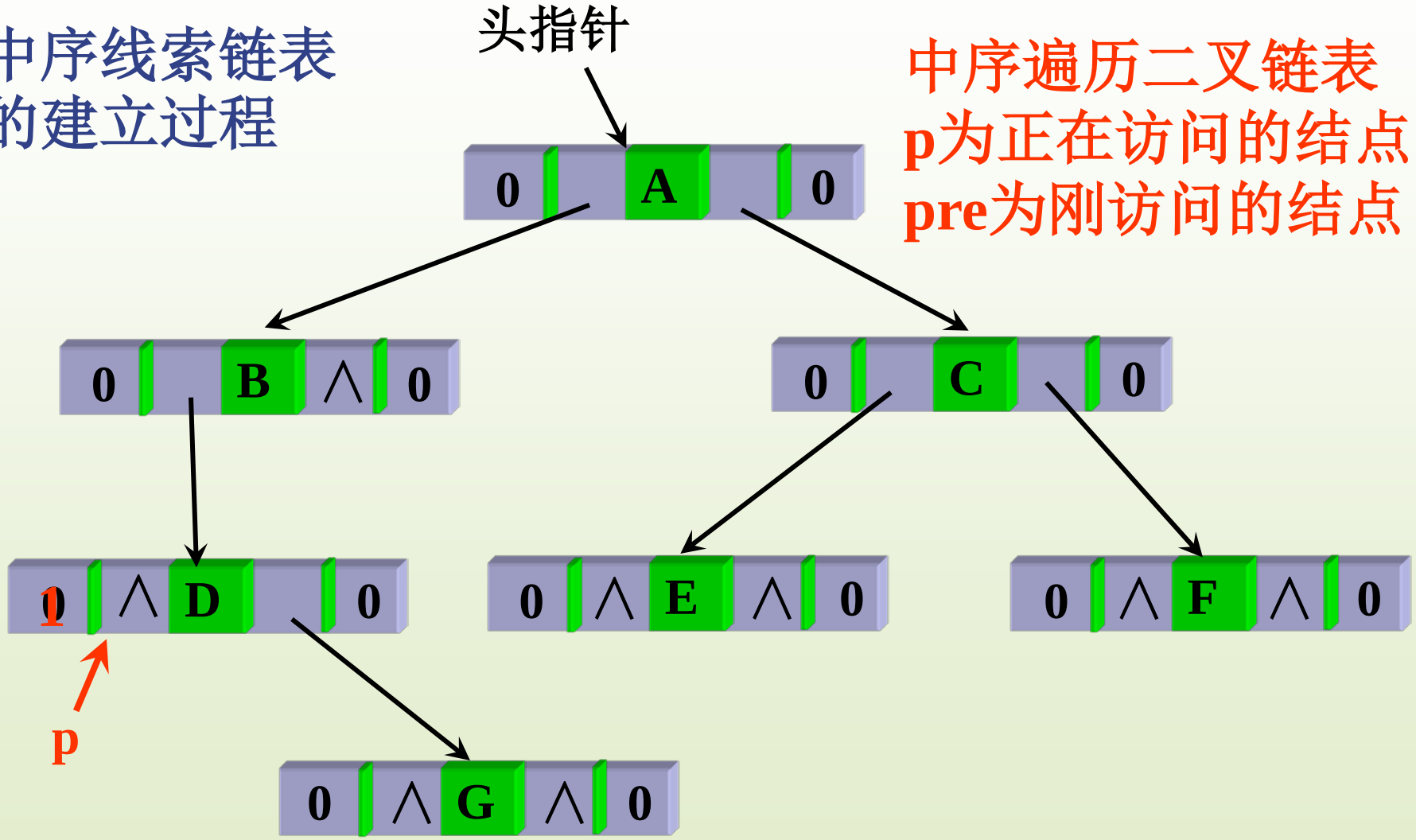
遍历二叉树，将空指针改为线索

# 中序线索链表的建立过程

已经建立起二叉链表

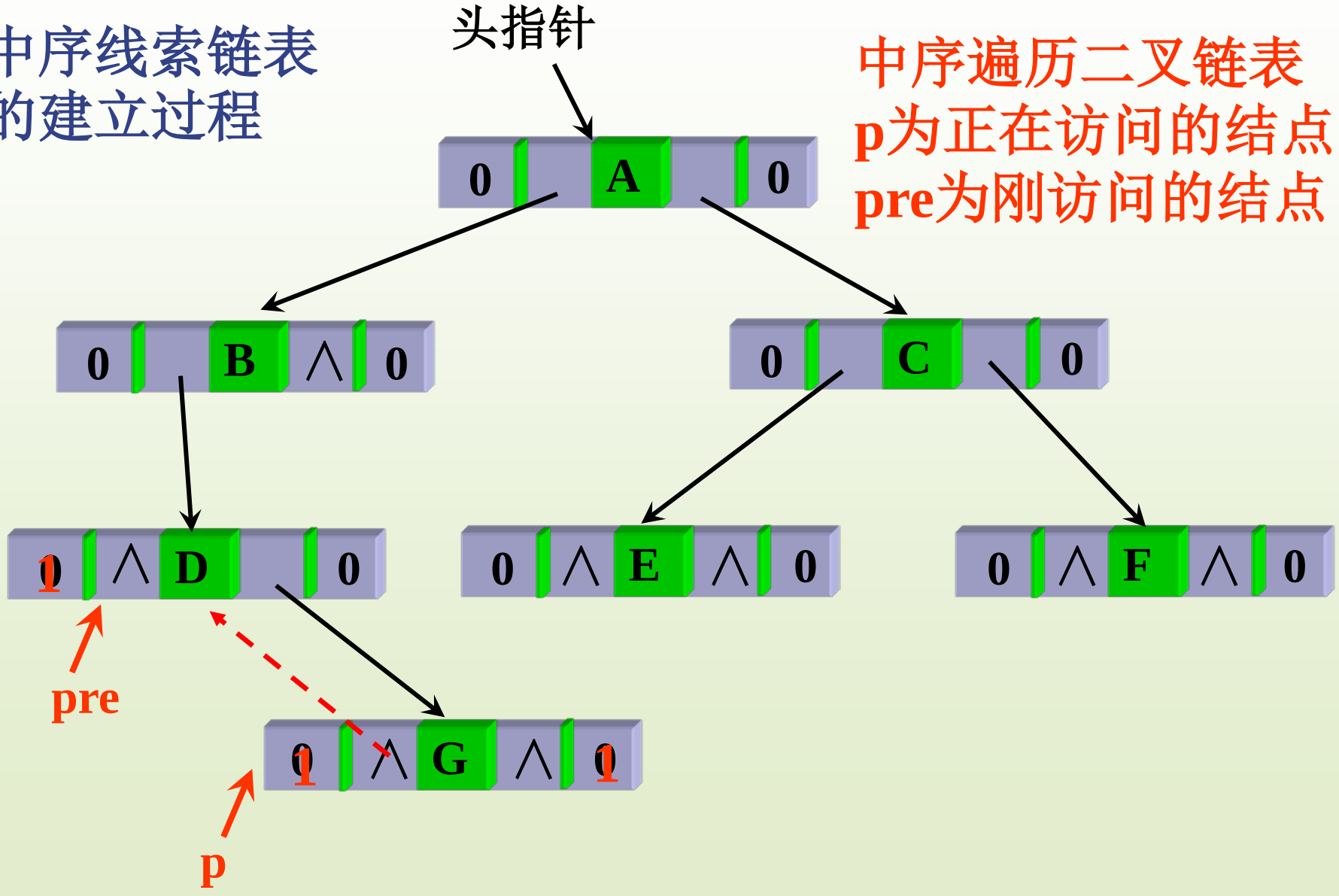


# 中序线索链表的建立过程



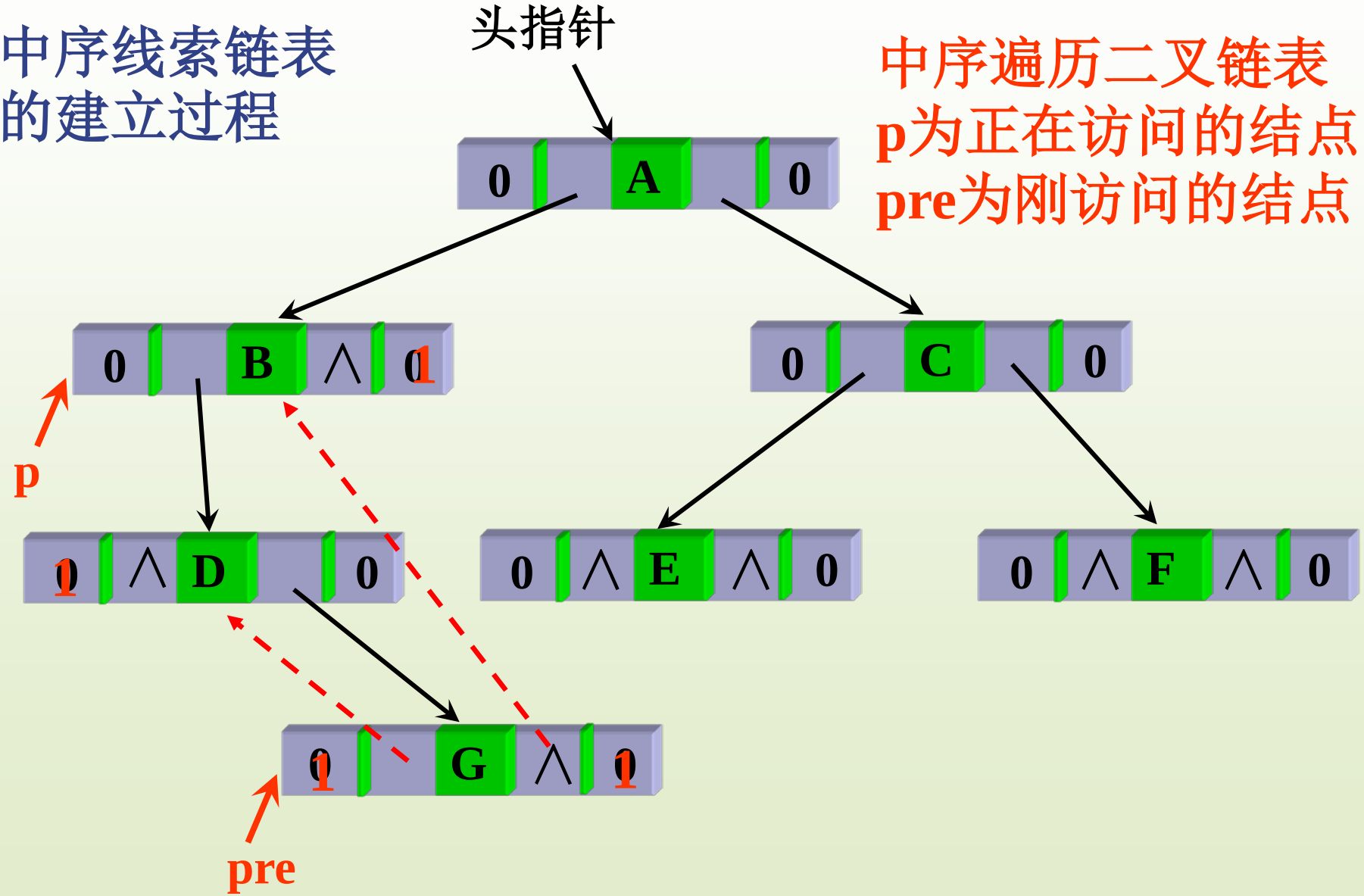

中序遍历二叉链表
p为正在访问的结点
pre为刚访问的结点

# 中序线索链表的建立过程



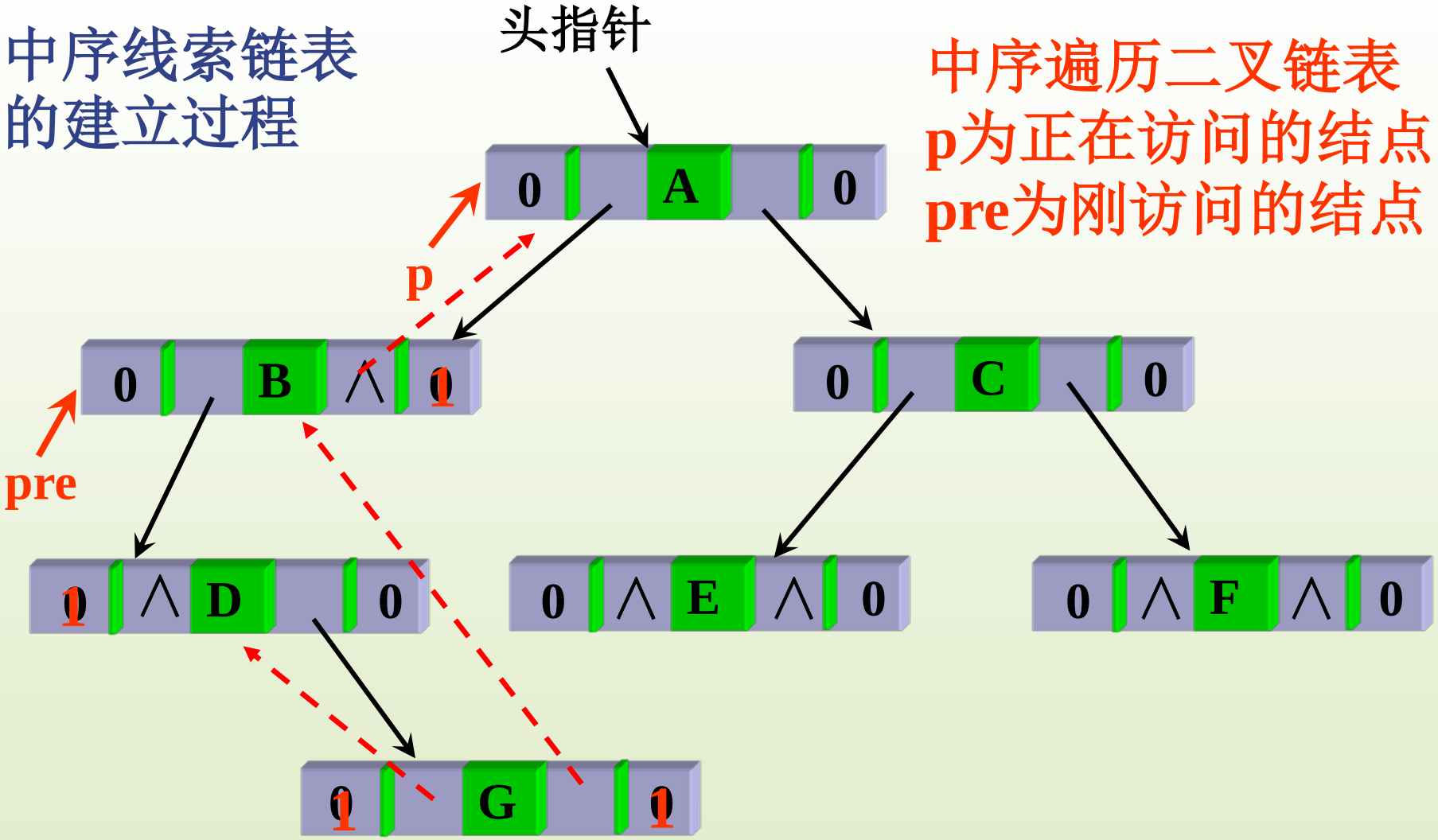

中序遍历二叉链表
p为正在访问的结点
pre为刚访问的结点

# 中序线索链表的建立过程



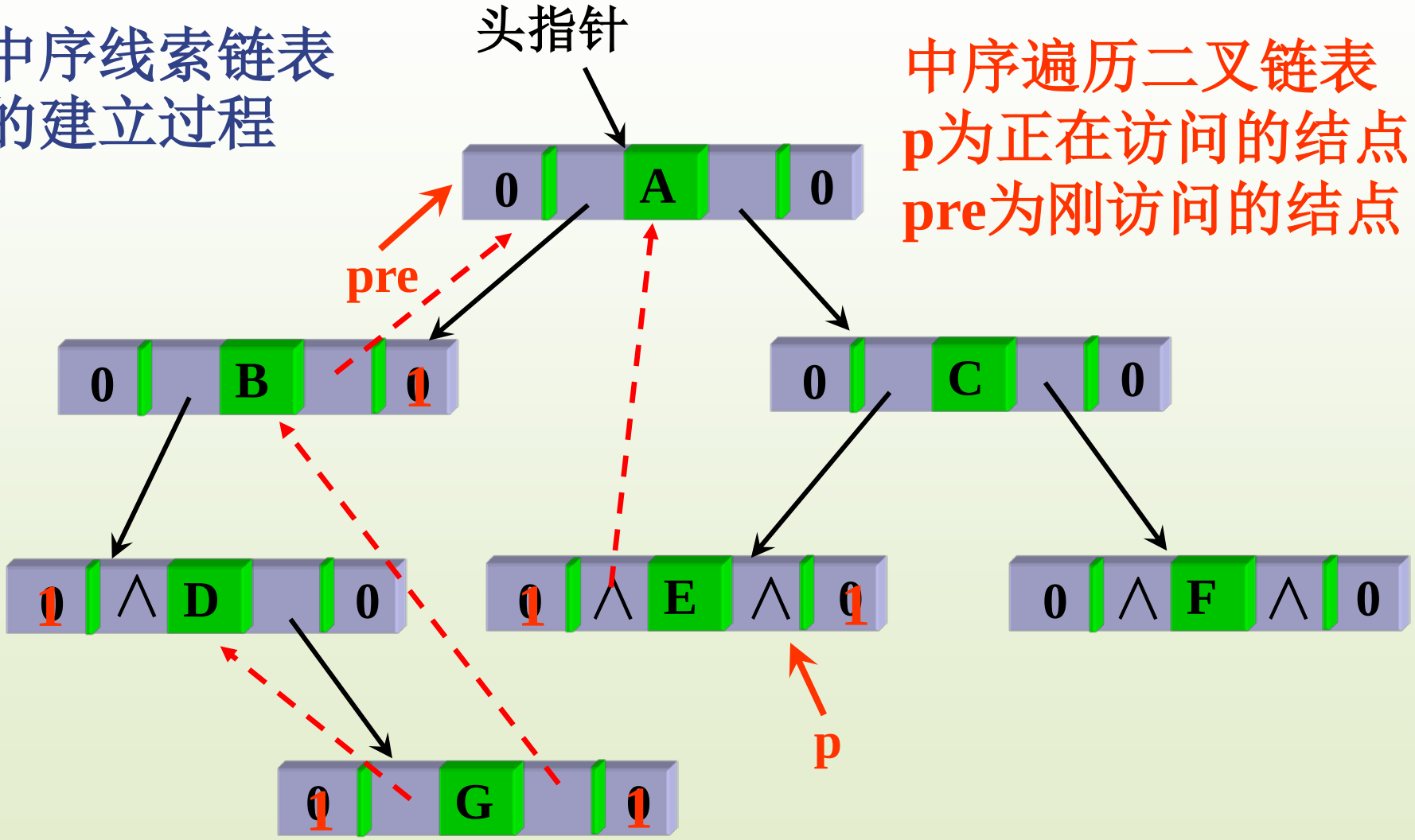

中序遍历二叉链表
p为正在访问的结点
pre为刚访问的结点

# 中序线索链表的建立过程



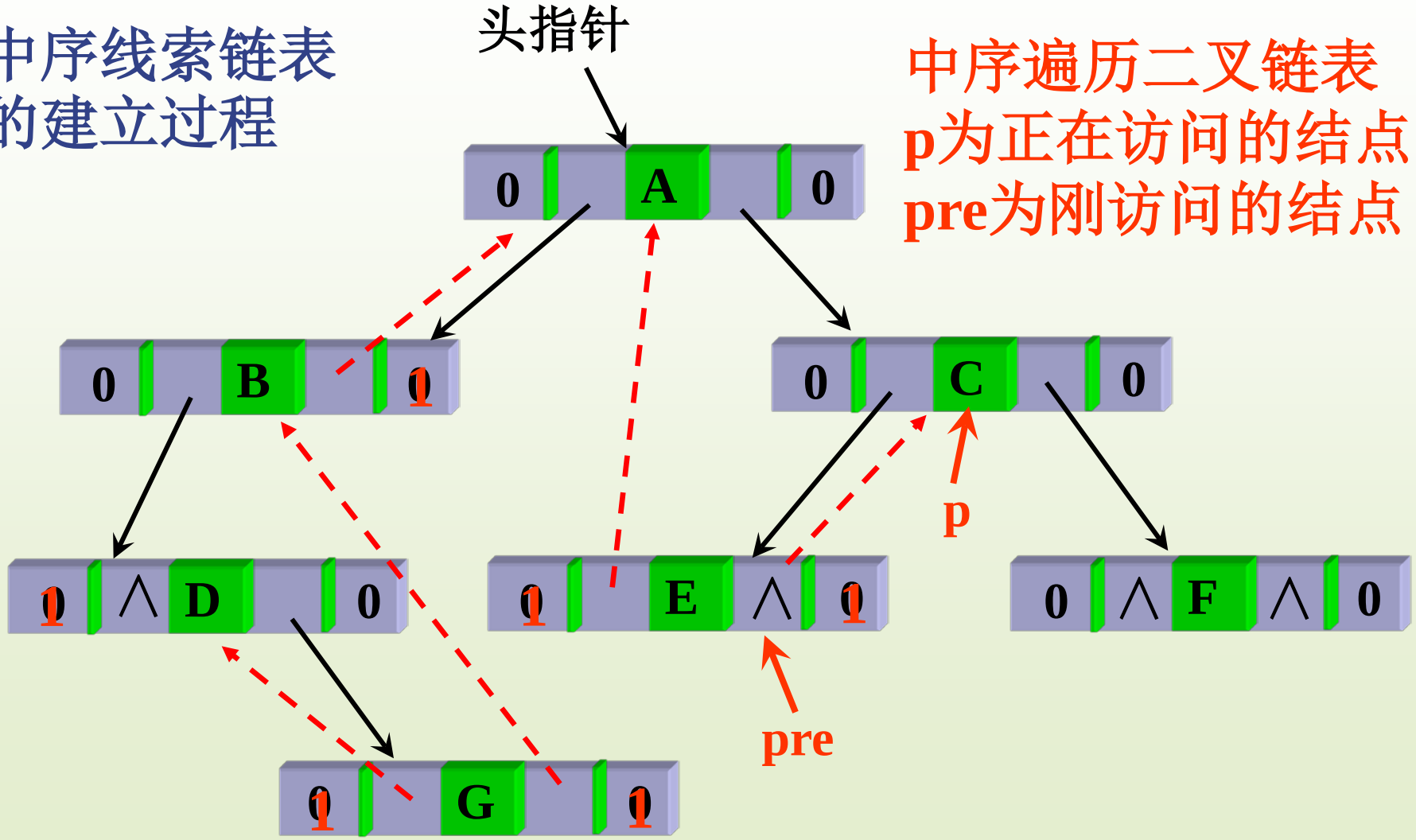

中序遍历二叉链表
p为正在访问的结点
pre为刚访问的结点

# 中序线索链表的建立过程



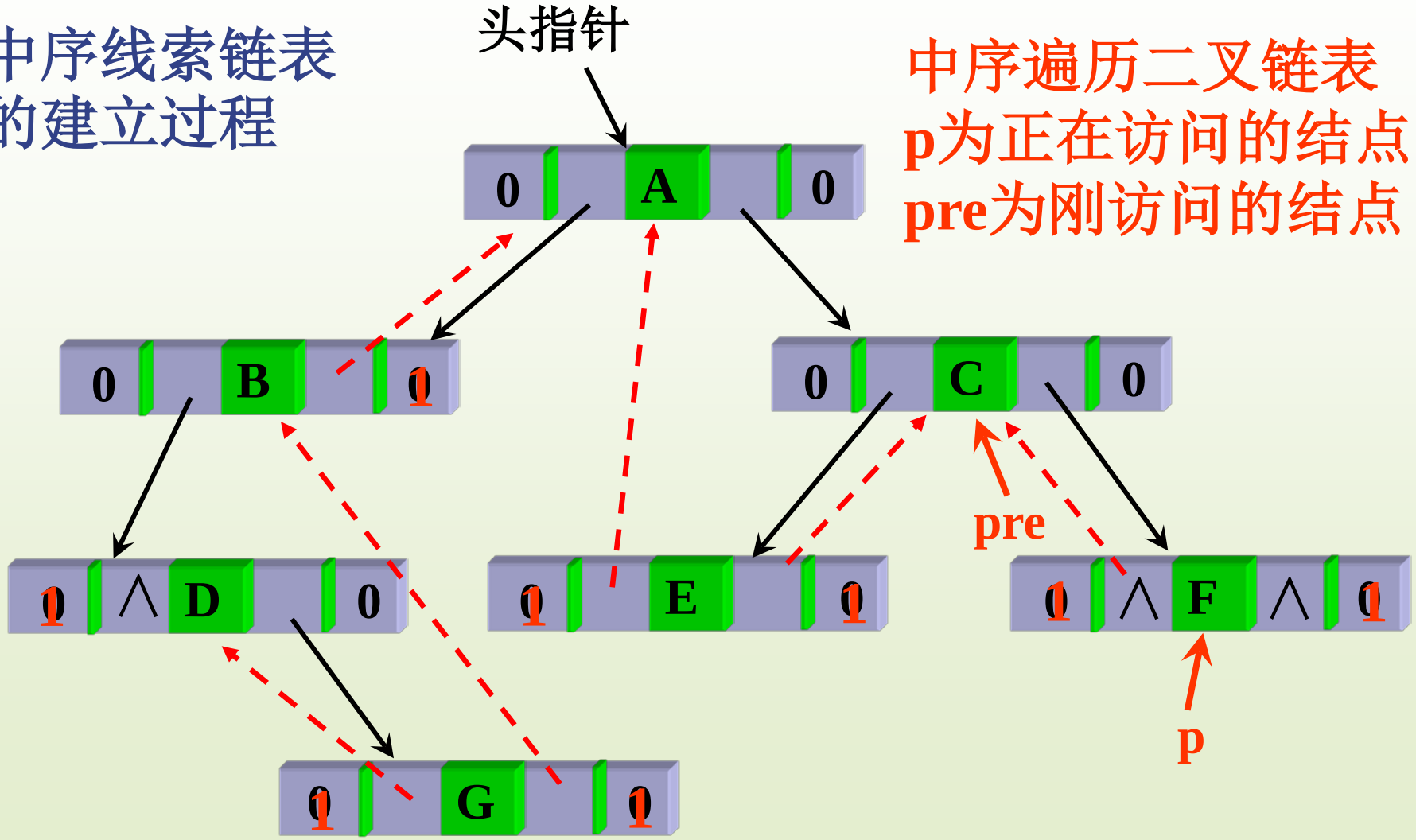

中序遍历二叉链表
p为正在访问的结点
pre为刚访问的结点

# 中序线索链表的建立过程



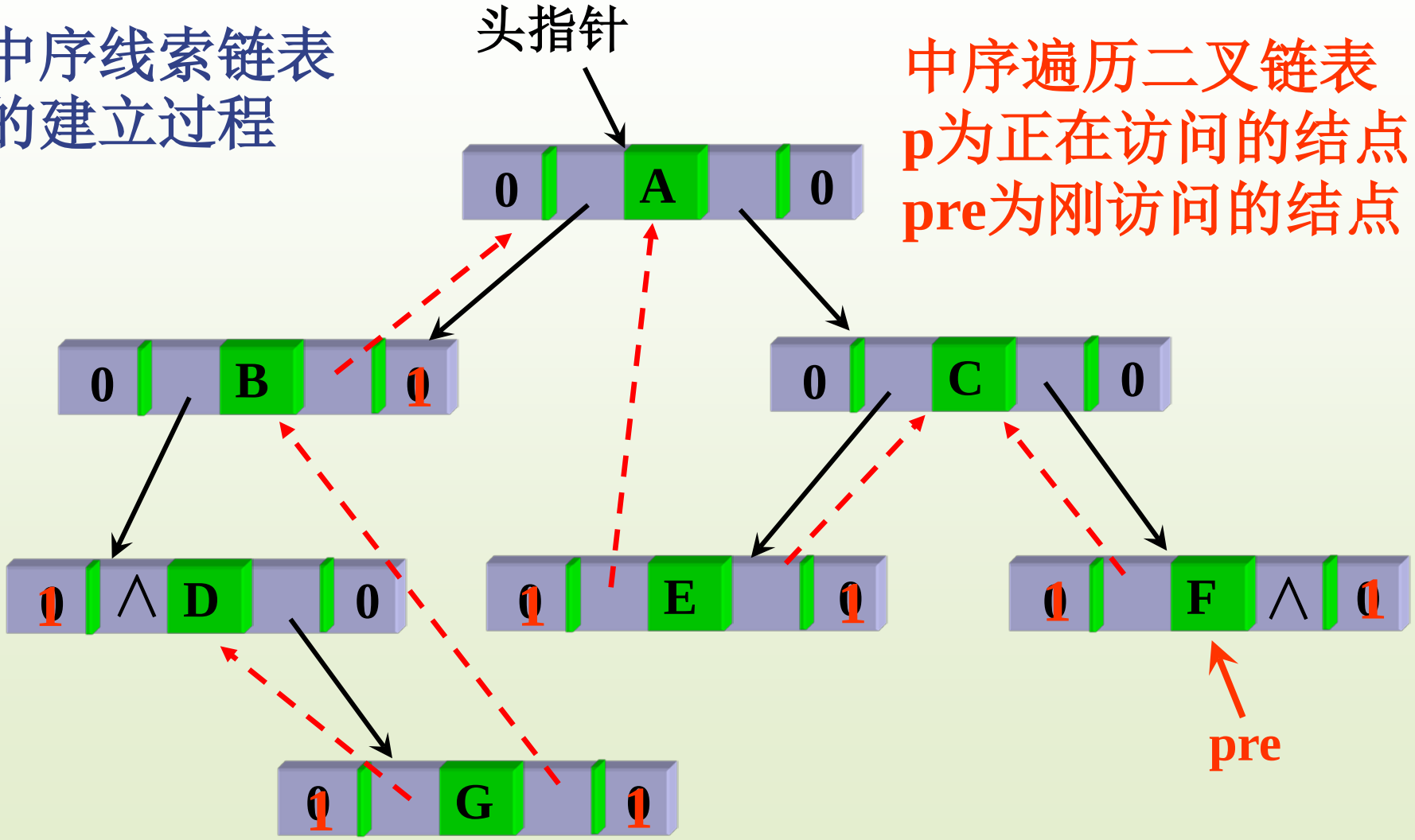

中序遍历二叉链表
p为正在访问的结点
pre为刚访问的结点

# 中序线索链表的建立过程




中序遍历二叉链表
p为正在访问的结点
pre为刚访问的结点

# 中序线索链表的建立过程




中序遍历二叉链表
p为正在访问的结点
pre为刚访问的结点

# 中序线索链表的建立

在遍历过程中，访问当前结点root的操作为：

⑴如果root的左、右指针域为空，则将相应标志置1；

⑵若root的左指针域为空，则令其指向它的前驱，这需要设指针pre始终指向刚刚访问过的结点，显然pre的初值为NULL；若pre的右指针域为空，则令其指向它的后继，即当前访问的结点root;

⑶ 令pre指向刚刚访问过的结点root;

# 中序线索链表的建立——构造函数

1. 建立二叉链表，将每个结点的左右标志置为0；
2. 遍历二叉链表，建立线索；
   2.1 如果二叉链表root为空，则空操作返回；
   2.2 对root的左子树建立线索；
   2.3 对根结点root建立线索；
      2.3.1 若root没有左孩子，则为root加上前驱线索；
      2.3.2 若root没有右孩子，则将root右标志置为1；
      2.3.3 若结点pre右标志为1，则为pre加上后继线索；
      2.3.4 令pre指向刚刚访问的结点root；
   2.4 对root的右子树建立线索。

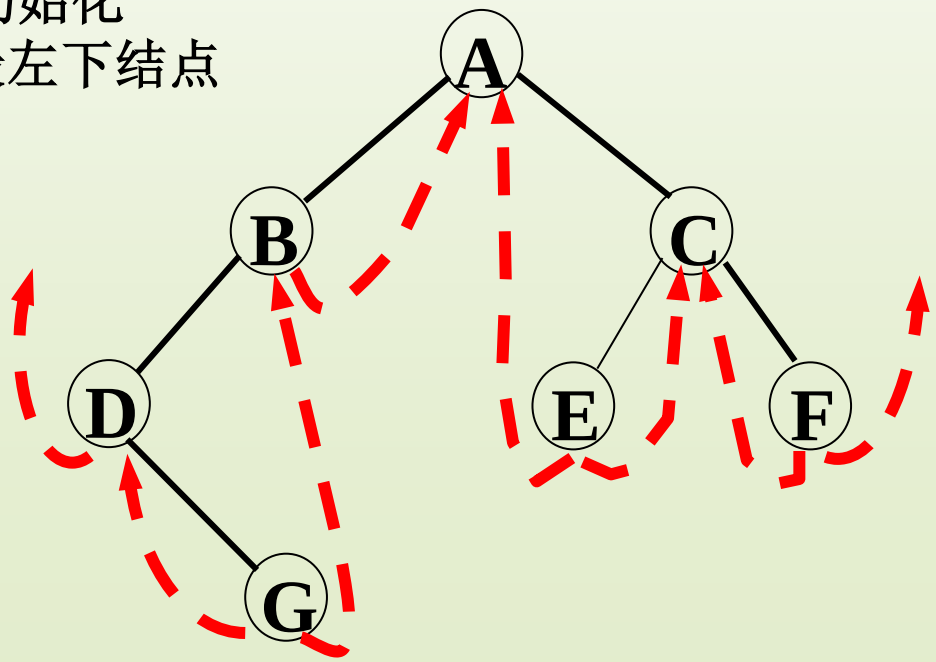
```
//构造一棵二叉树,构造函数调用
template <class T>
ThrNode<T>* InThrBiTree<T>::Creat(Creat(ThrNode<T> *root )
{
    ThrNode<T> *root;
    T ch;
    cout<<"请输入创建一棵二叉树的结点数据"<<endl;
    cin>>ch;
   if (ch=="#") root = NULL;
   else{
       root=new ThrNode<T>;     //生成一个结点
       root->data = ch;
       root->ltag = Child;    root->rtag = Child;
       root->lchild = Creat(root->lchild );  //递归建立左子树
       root->rchild = Creat(root->rchild  );  //递归建立右子树
   }
    return root;
}
```

```
//中序线索化二叉树
template <class T>
void InThrBiTree<T>::ThrBiTree(ThrNode<T> *root , ThrNode<T> *pre)
{
  if (root==NULL) return;        //递归结束条件
  ThrBiTree(root->lchild,pre);
  if (!root->lchild){            //对root的左指针进行处理
      root->ltag = Thread;
      root->lchild = pre;        //设置pre的前驱线索
  }
  if (root->rchild==NULL) root->rtag = Thread;  //对root的右指针进行处理
  if(pre != NULL){
    if (pre->rtag==Thread)  pre->rchild = root;    //设置pre的后继线索
  }
  pre = root;
  ThrBiTree(root->rchild,pre);
}
```

```
//中序线索二叉树构造函数
template <class T>
InThrBiTree<T>::InThrBiTree( )
{
    ThrNode<T>* pre = NULL;
    this->root = Creat( );
    ThrBiTree(root);
}
```

//中序线索二叉树查找结点p的后继结点

```
template <class T>
ThrNode<T>* InThrBiTree<T>::Next(ThrNode<T>* p)
{
   ThrNode<T>* q;
   if (p->rtag==Thread)   q = p->rchild;  //右标志为1，可直接得到后继结点
   else{
      q = p->rchild;           //工作指针初始化
      while (q->ltag==Child)    //查找最左下结点
           q = q->lchild;
    }
   return q;
}
```



```
//中序遍历线索二叉树
//先找到序列中的第一个结点，然后依次找结点后继直到其后继为空
template <class T>
void InThrBiTree<T>::InOrder(ThrNode<T> *root)
{
   ThrNode<T>* p = root;
   if (root==NULL)  return;     //如果线索链表为空，则空操作返回
   while (p->ltag==Child)       //查找中序遍历序列的第一个结点p并访问
         p = p->lchild;
   cout<<p->data<<" ";
   while (p->rchild!=NULL)      //当结点p存在后继，依次访问其后继结点
   {
      p = Next(p);
      cout<<p->data<<" ";
   }
    cout<<endl;
}
```

# 5.5 二叉树遍历的非递归算法

## 前序遍历——非递归算法

二叉树前序遍历的非递归算法的关键：在前序遍历过某结点的整个左子树后，如何找到该结点的右子树的根指针。——【先访问完结点】的右子树后访问

【后访问完结点】的右子树先访问

解决办法：在访问完该结点后，将该结点的指针保存在栈中，以便以后能通过它找到该结点的右子树。

# 前序遍历——非递归算法

在前序遍历中，设要遍历二叉树的根指针为root，则有两种可能：

⑴ 若root!=NULL，

则表明当前的二叉树不空，

处理：输出根结点的信息，root入栈，遍历root的左子树；

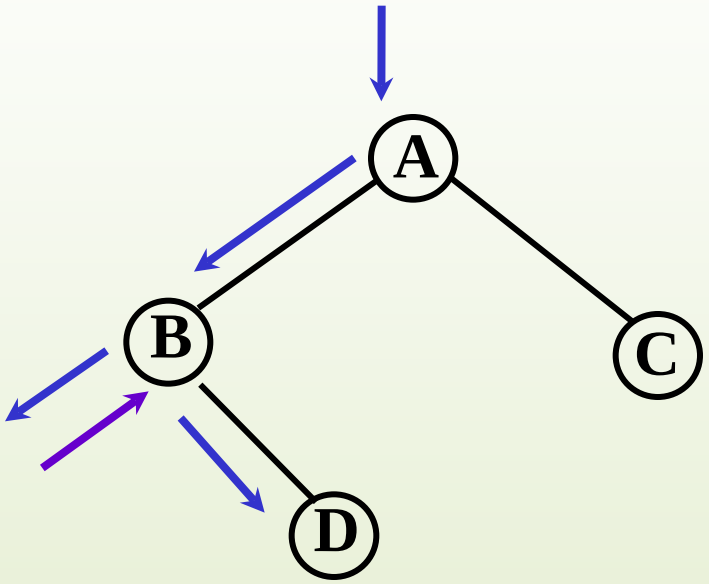
⑵ 若root==NULL，

则表明以root为根的二叉树遍历完毕，root是栈顶指针所指结点的左子树，

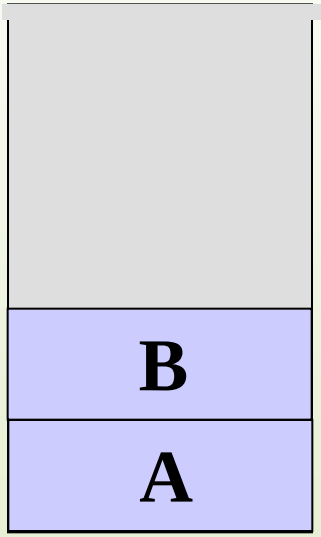
处理：

- 若栈非空，由栈顶指针所指结点找到其右子树的根指针赋给root,
- 若栈空，则遍历完毕。

# 前序遍历的非递归实现

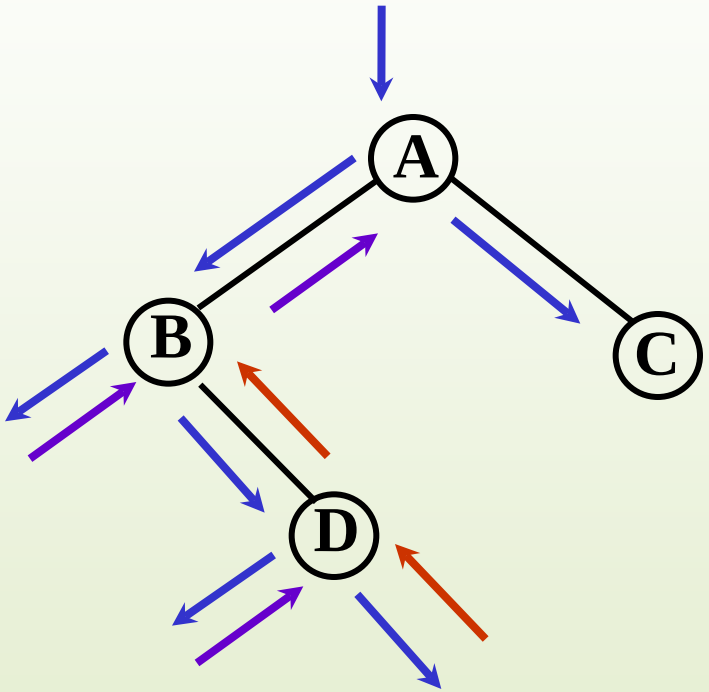



栈S内容:

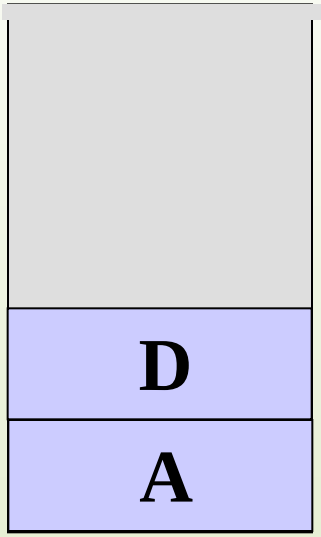



访问结点序列: A B D

# 前序遍历的非递归实现

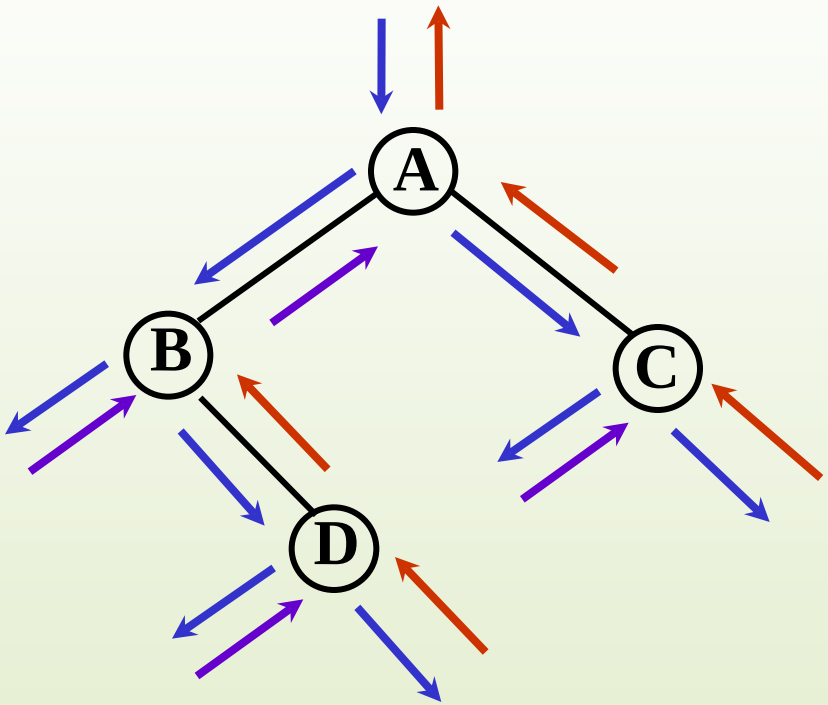



栈S内容:

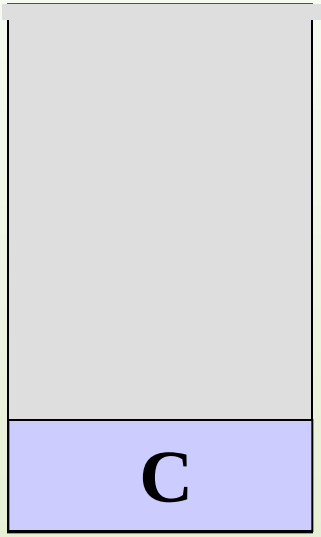



访问结点序列: A B D

# 前序遍历的非递归实现




栈S内容:




访问结点序列: A B D C

# 前序遍历——非递归算法（伪代码）

1.栈s初始化；

2.循环直到root为空且栈s为空

2.1 当root不空时循环

2.1.1 输出root->data;

2.1.2 将指针root的值保存到栈中；

2.1.3 继续遍历root的左子树

2.2 如果栈s不空，则

2.2.1 将栈顶元素弹出至root;

2.2.2 准备遍历root的右子树；

# 前序遍历——非递归算法

```
template <class T>
void BiTree<T>::PreOrder0(BiNode<T> *root)
{  const int MaxStackSize=100;
   BiNode<T> *S[MaxStackSize];
   int top= -1;   // 构造空栈, 采用顺序栈，并假定不会发生上溢
   while (root!=NULL||top!= -1)
    {  while (root!= NULL)
       {  cout<<root->data;
          S[++top]=root;;
          root=root->lchild;
       }
       if (top!= -1) {
          root=S[top--];
          root=root->rchild;
       }
    }
}
```

# 中序遍历——非递归算法

和前序算法基本上一致，但是数据访问在出栈的时候

在中序遍历中，设要遍历二叉树的根指针为root，则有两种可能：

⑴ 若root!=NULL，

则表明当前的二叉树不空，

处理：输出根结点的信息， root入栈，遍历root的左子树；

⑵ 若root==NULL，

则表明以root为根的二叉树遍历完毕，root是栈顶指针所指结点的左子树，

处理：

若栈非空，由栈顶指针所指结点找到其右子树的根指针赋给root,输出根结点的信息，

若栈空，则遍历完毕。

# 中序遍历——非递归算法

```
template <class T>
void BiTree<T>::PreOrder0(BiNode<T> *root)
{  const int MaxStackSize=100;
   BiNode<T> *S[MaxStackSize];
   int top= -1;   // 构造空栈, 采用顺序栈，并假定不会发生上溢
   while (root!=NULL||top!= -1)
    {  while (root!= NULL)
       {  cout<<root->data;
          S[++top]=root;;
          root=root->lchild;
       }
       if (top!= -1) {
          root=S[top--]; cout<<root->data;
          root=root->rchild;
       }
    }
}
```

# 后序遍历——非递归算法

后序遍历与先序遍历和中序遍历不同，在对二叉树进行后序遍历的过程中,当指针root指向某一个结点时,

•不能马上对它进行访问,而要

先遍历它的左子树——要将此结点的地址进栈保存

当其左子树遍历完毕之后,再次搜索到该结点时(该结点的地址通过退栈得到),

•还不能对它进行访问,需要

遍历它的右子树——再一次将此结点的地址进栈保存。

为了区别同一结点的两次进栈, 设置一标志flag，令：

flag=1，第1次出栈，只遍历完左子树，不能访问该结点

flag=2，第2次出栈，已遍历完右子树，可以访问该结点

# 后序遍历——非递归算法

```
template<class T>
void BiTree<T>::PostOrder0(BiNode<T> *root)
{   const int MaxStackSize=100;
    BiNodeflag<T> S[MaxStackSize];
    BiNodeflag<T> p;
    int top= -1;   //构造空栈, 采用顺序栈
    while (root!=NULL||top!= -1)
     {while (root!= NULL)
        { p.ptr=root;p.flag=1;
          S[++top]=p;
          root=root->lchild; }
       if (top!= -1) {  p=S[top--];
                        if(p.flag==1){p.flag=2;S[++top]=p;root=p.pt->rchild;}
                        else if(p.flag==2){cout<<p.pt->data;}
                      }
     }
}
```

```
template <class T>
struct BiNodeflag   //栈的结点结构
{
  int flag; //1-访问左子树  2-访问右子树
  BiNode<T> *ptr;
};
```

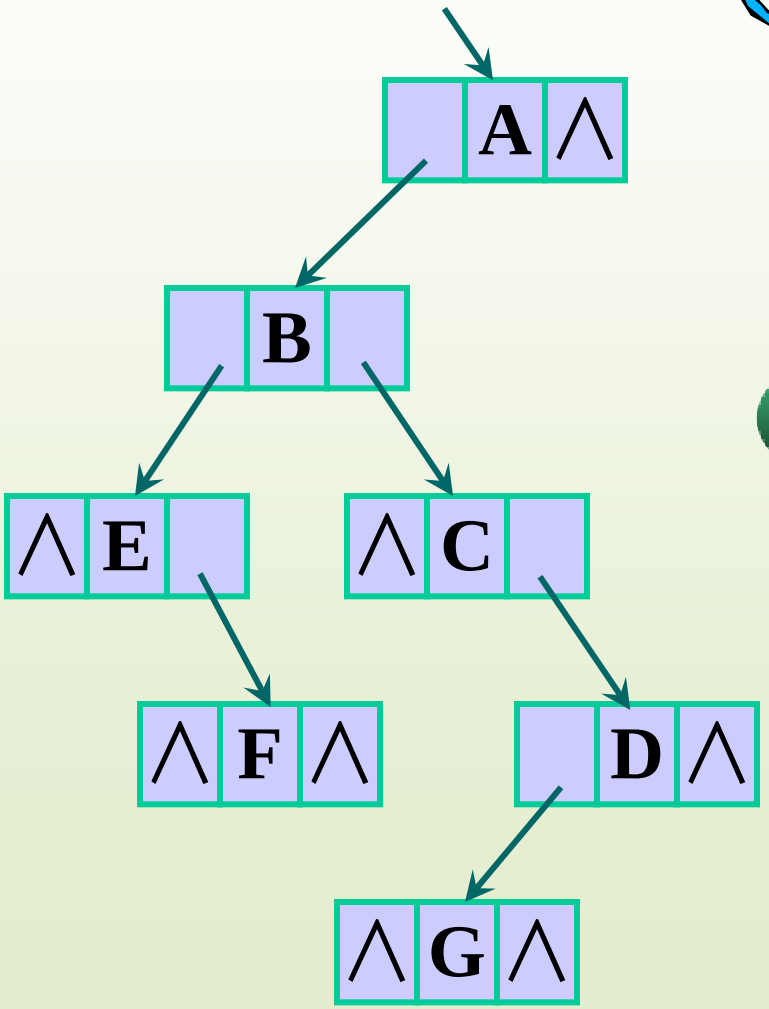
# 后序遍历——非递归算法

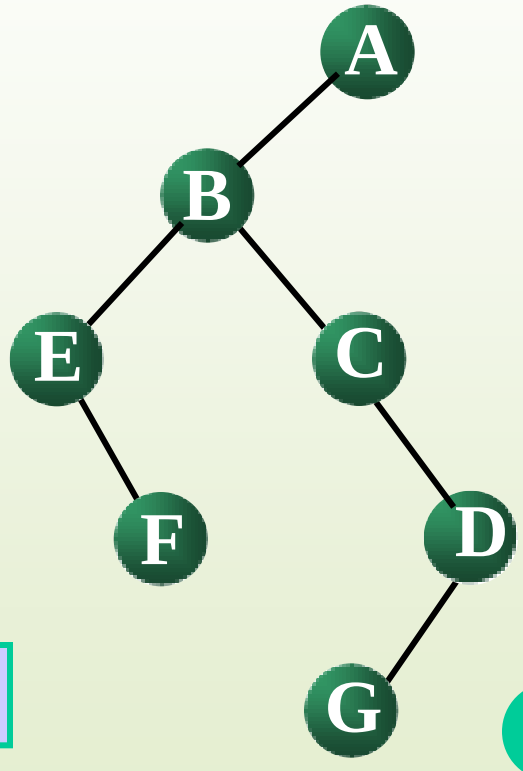
```
template<class T>
void BiTree<T>::PostOrder0(BiNode<T> *root)
{   const int MaxStackSize=100;
    BiNodeflag<T> S[MaxStackSize];
    BiNodeflag<T> p;
    int top= -1;   //构造空栈, 采用顺序栈
    while (root!=NULL||top!= -1)
     {while (root!= NULL)
        { p.ptr=root;p.flag='L';
          S[++top]=p;
          root=root->lchild; }
       if (top!= -1) {  p=S[top--];
                        if(p.flag=='L'){p.flag='R';S[++top]=p;root=p.ptr->rchild;}
                        else if(p.flag=='R'){cout<<p.ptr->data;}
                      }
     }
}
```

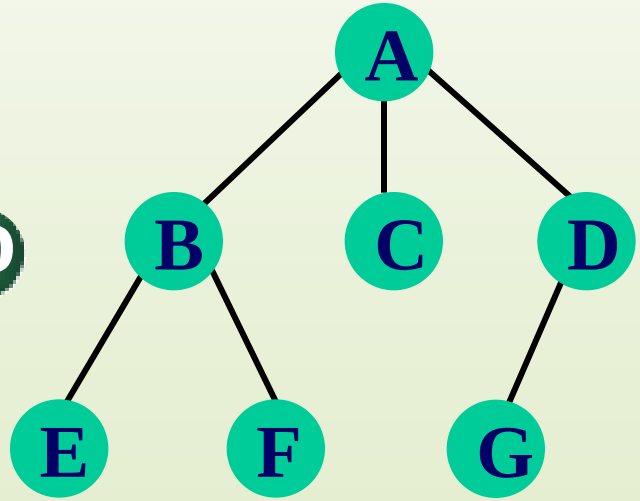
```
template <class T>
struct BiNodeflag   //栈的结点结构
{
  char flag; //L-访问左子树  R-访问右子树
  BiNode<T> *ptr;
};
```

# 5.6 树、森林与二叉树的转换

是哪些树结构的存储结构?




树和二叉树之间具有对应关系

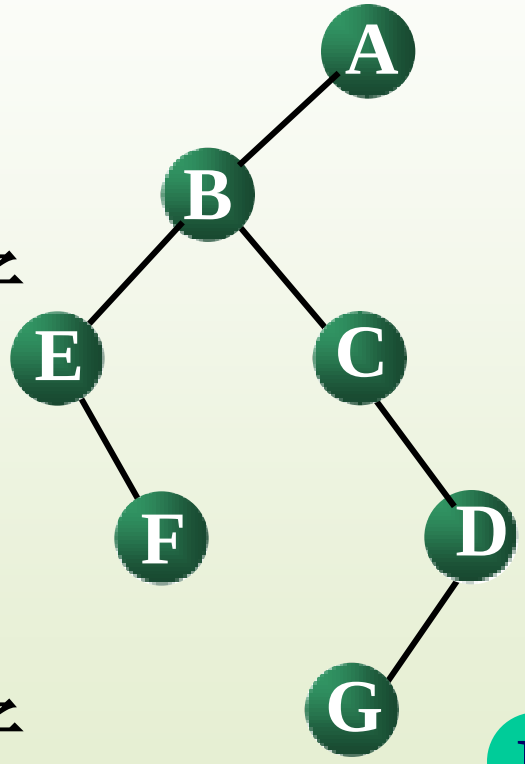
# 树和二叉树之间的对应关系

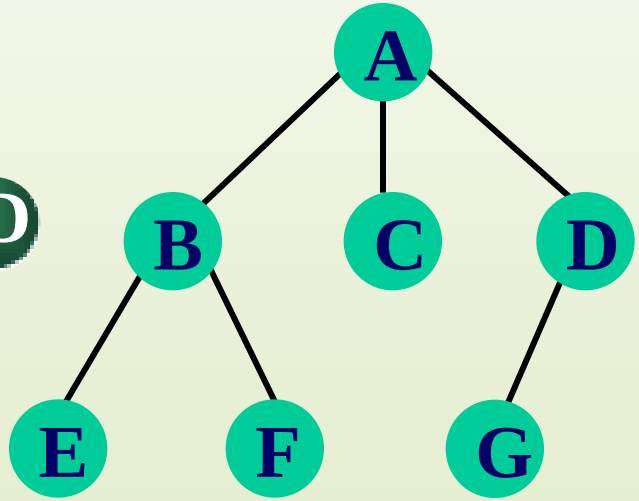
树：兄弟关系

二叉树：双亲和右孩子

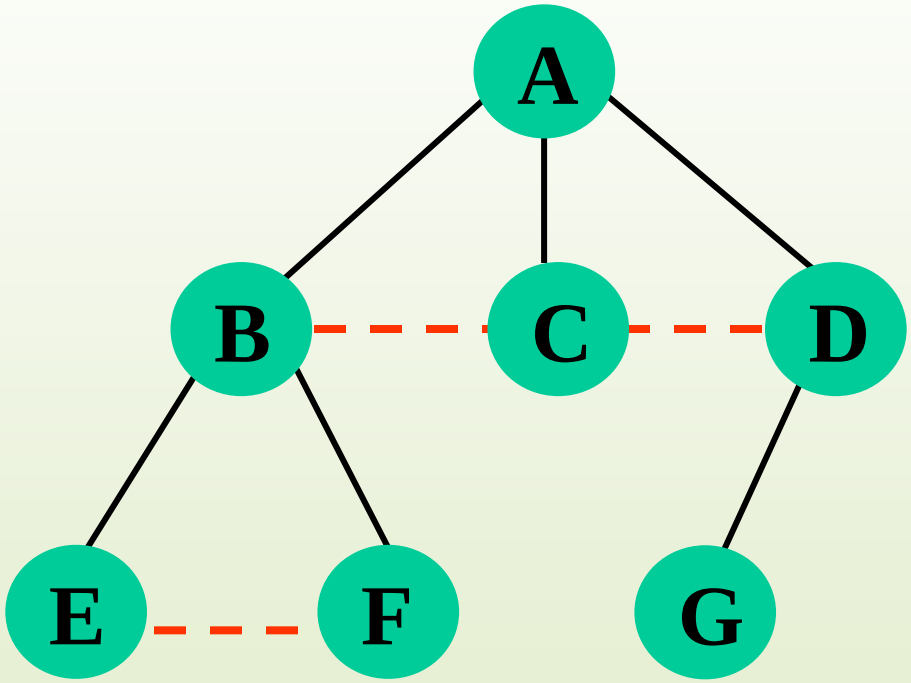
树：双亲和长子

二叉树：双亲和左孩子



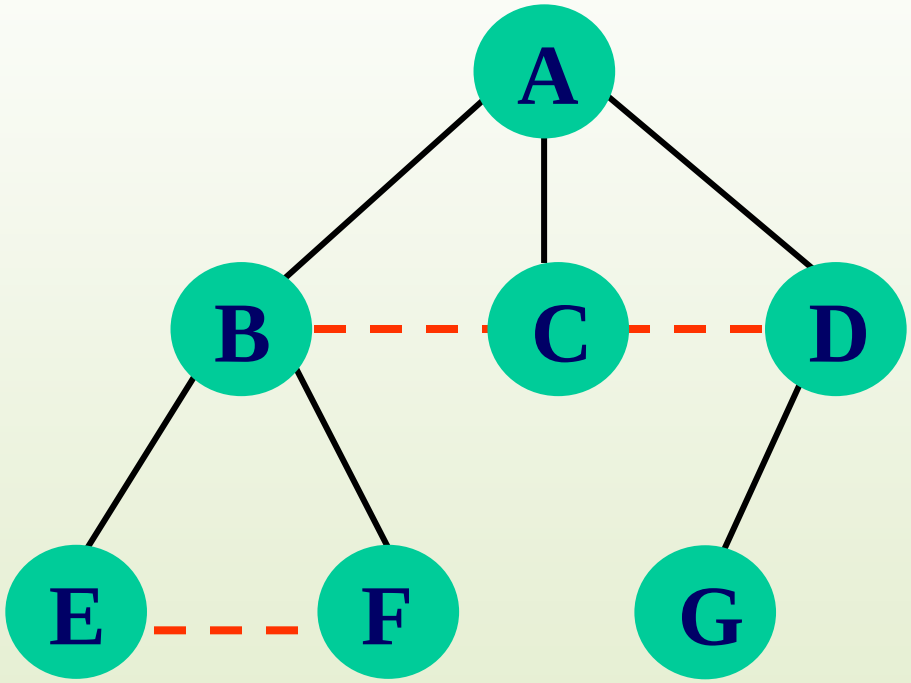



# 树和二叉树之间的对应关系




1.兄弟加线.

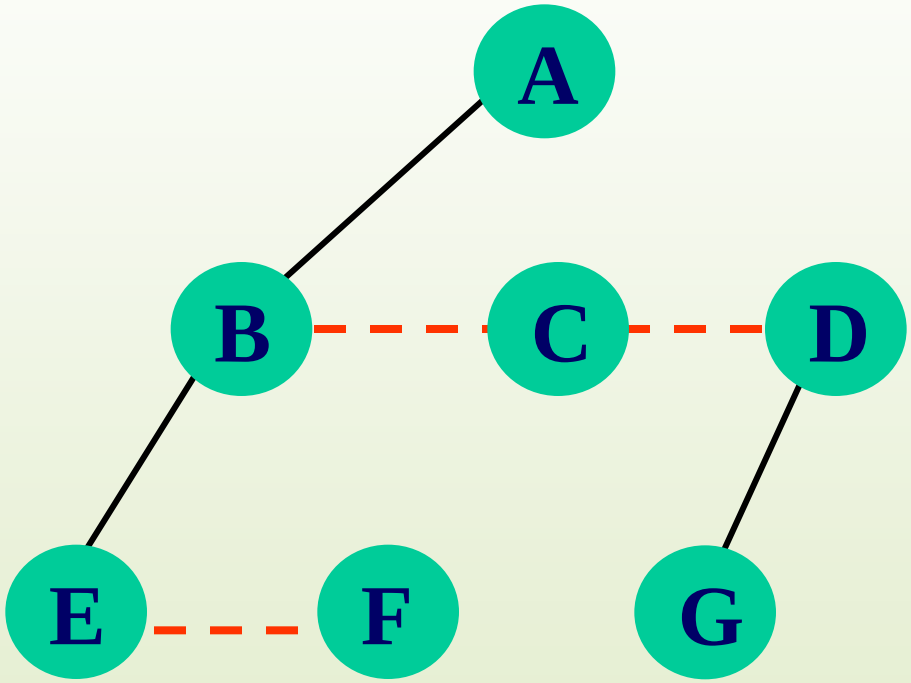
# 树和二叉树之间的对应关系




1.兄弟加线.

2.保留双亲与第一孩子连线,删去与其他孩子的连线.

# 树和二叉树之间的对应关系

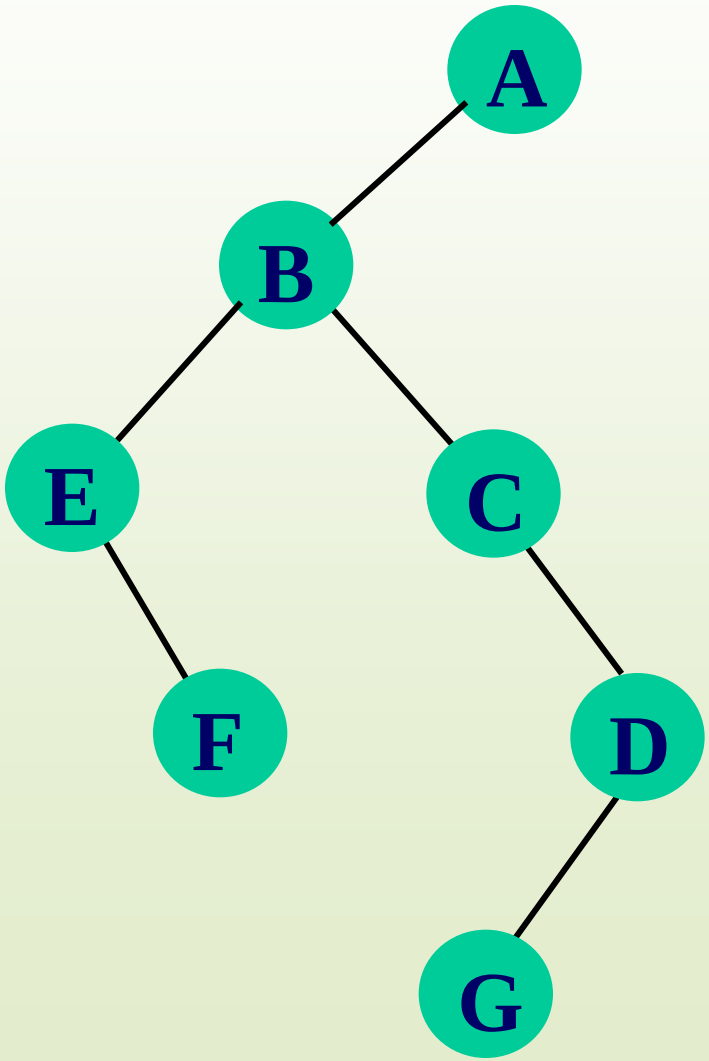



1.兄弟加线.

2.保留双亲与第一孩子连线,删去与其他孩子的连线.

3.顺时针转动,使之层次分明.

# 树和二叉树之间的对应关系




1.兄弟加线.

2.保留双亲与第一孩子连线,删去与其他孩子的连线.

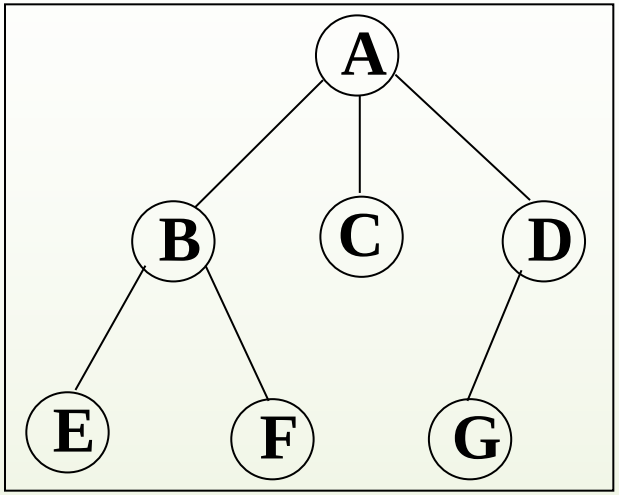
3.顺时针转动,使之层次分明.

# 树转换为二叉树

(1)加线——树中所有相邻兄弟之间加一条连线。

(2)去线——对树中的每个结点，只保留它与第一个孩子结点之间的连线，删去它与其它孩子结点之间的连线。

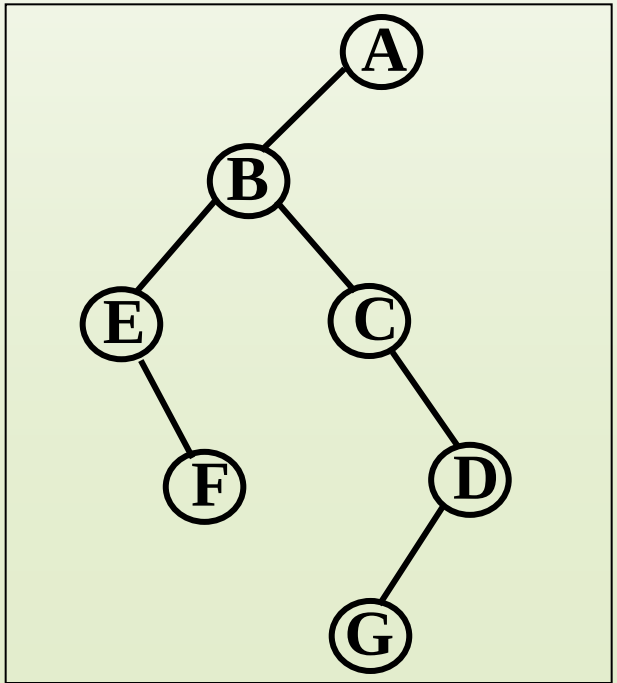
(3)层次调整——以根结点为轴心，将树顺时针转动一定的角度，使之层次分明。




前序遍历

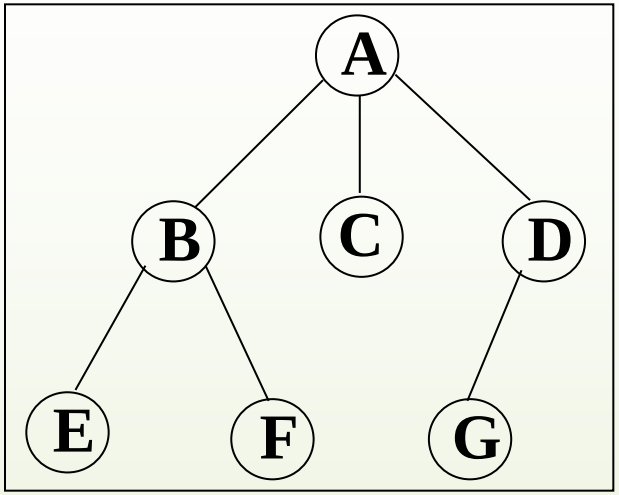
**ABEFCDG**

**树的前序遍历等价于二叉树的前序遍历！**




前序遍历

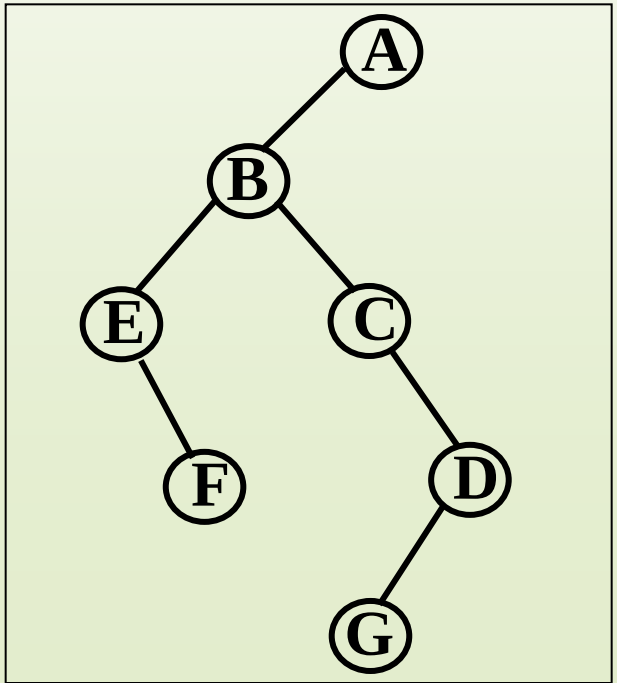
**ABEFCDG**




后序遍历

**EFBCGDA**

**树的后序遍历等价于二叉树的中序遍历！**



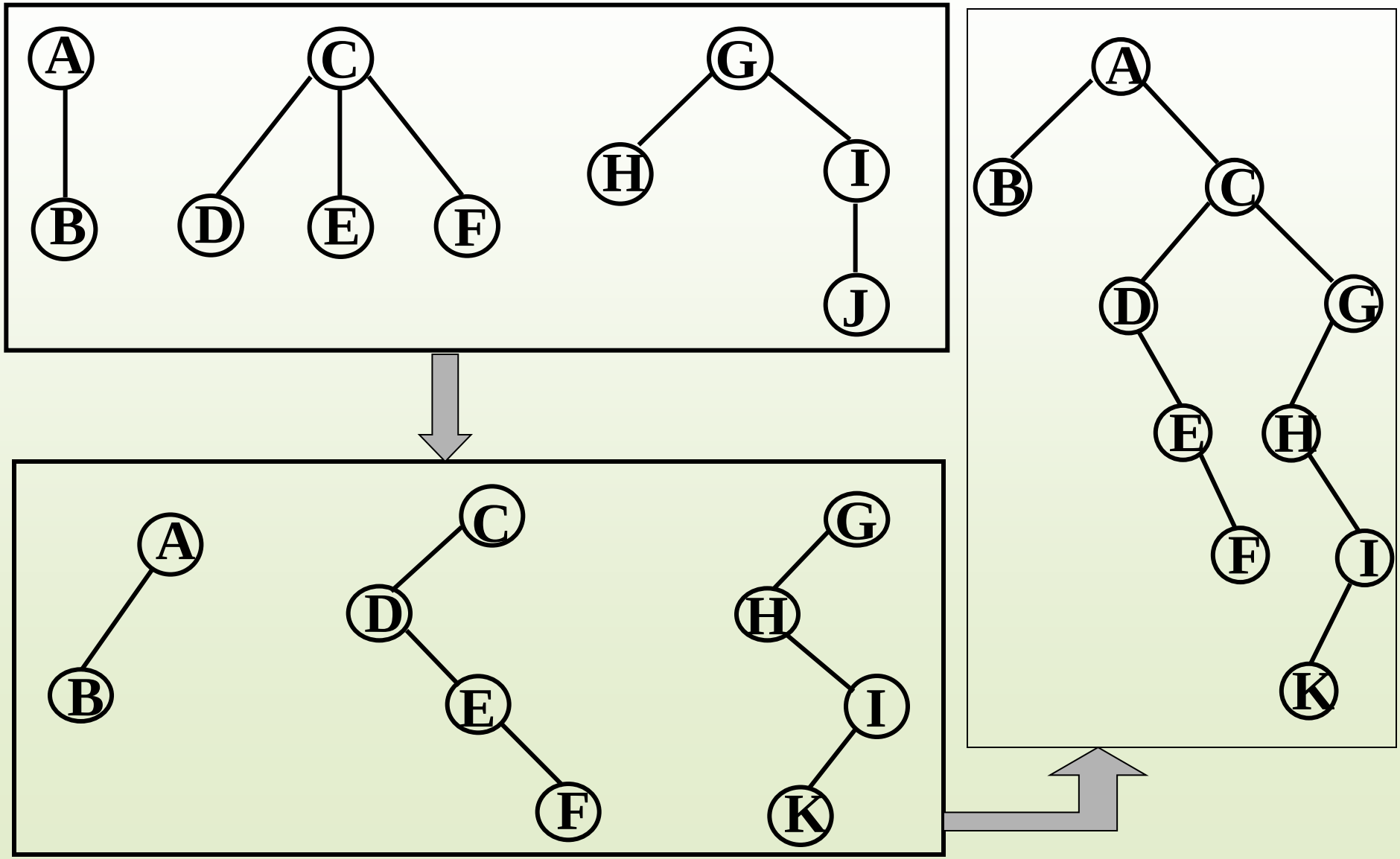

中序遍历

**EFBCGDA**

# 森林转换为二叉树

(1) 将森林中的每棵树转换成二叉树；

(2) 从第二棵二叉树开始，依次把后一棵二叉树的根结点作为前一棵二叉树根结点的右孩子，当所有二叉树连起来后，此时所得到的二叉树就是由森林转换得到的二叉树。


A
B
C
D
E
F
G
H
I
J
A
B
C
D
G
E
H
F
I
K
A
B
C
D
E
F
G
H
I
K

# 森林的遍历

森林有两种遍历方法：

⑴前序（根）遍历：前序遍历森林即为前序遍历森林中的每一棵树。

⑵后序（根）遍历：后序遍历森林即为后序遍历森林中的每一棵树。

# 5.7 应用举例

## 哈夫曼树及哈夫曼编码

**叶子结点的权值：**对叶子结点赋予的一个有意义的数值量。

**二叉树的带权路径长度（Weighted Path Length）：**设二叉树具有$n$个带权值的叶子结点，从根结点到各个叶子结点的路径长度与相应叶子结点权值的乘积之和。 记为：

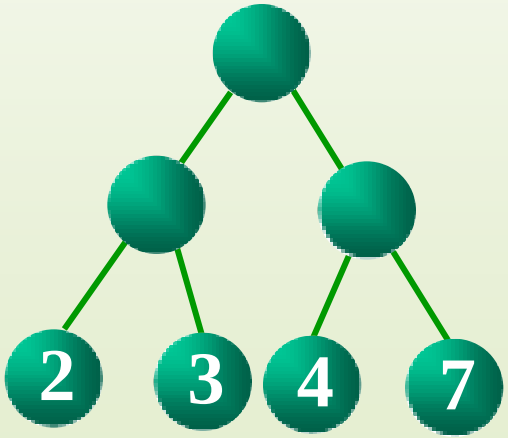
$$WPL=\sum_{k=1}^{n} w_k\, l_k$$

$l_k$：从根结点到第$k$个叶子的路径长度

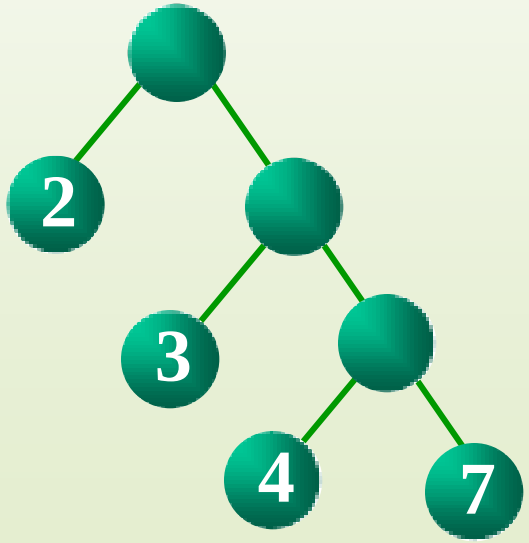
$w_k$：第$k$个叶子的权值；

**哈夫曼树(Huffman Tree)：** 给定一组具有确定权值的叶子结点，带权路径长度最小的二叉树。

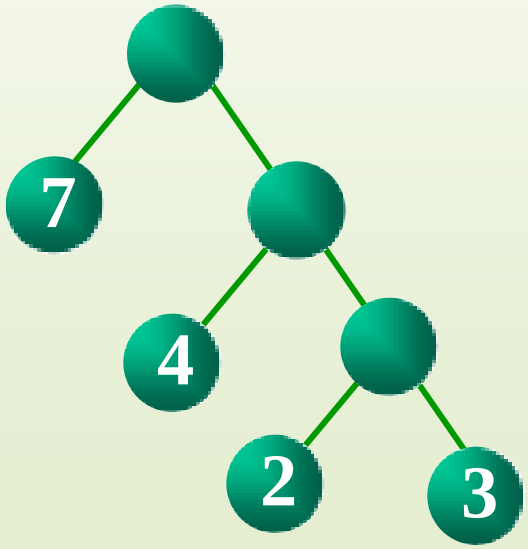
例：给定4个叶子结点，其权值分别为{2，3，4，7}，可以构造出形状不同的多个二叉树。




WPL=32




WPL=41

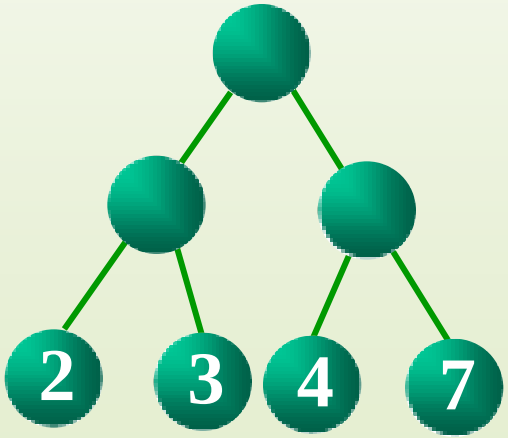



WPL=30

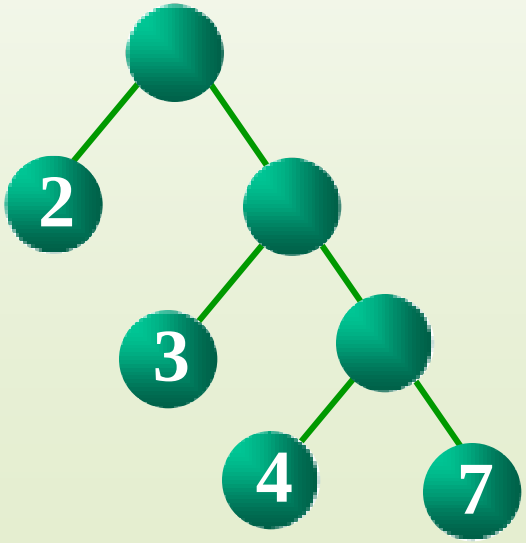
## 哈夫曼树的特点：

1. 权值越大的叶子结点越靠近根结点，而权值越小的叶子结点越远离根结点。

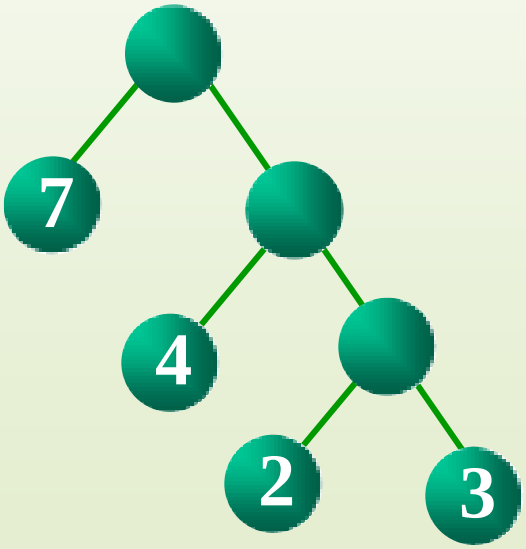
2. 只有度为0（叶子结点）和度为2（分支结点）的结点，不存在度为1的结点.




WPL=32




WPL=41




WPL=30

哈夫曼算法基本思想：

(1) 初始化：由给定的$n$个权值$\{w_1, w_2, \ldots, w_n\}$构造$n$棵只有一个根结点的二叉树，从而得到一个二叉树集合$F=\{T_1, T_2, \ldots, T_n\}$；

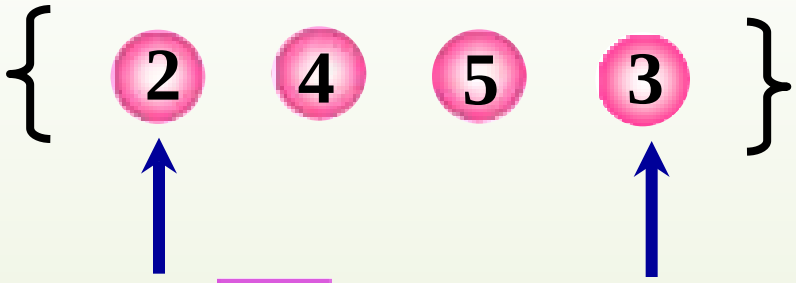
(2) 选取与合并：在$F$中选取根结点的权值最小的两棵二叉树分别作为左、右子树构造一棵新的二叉树，这棵新二叉树的根结点的权值为其左、右子树根结点的权值之和；

(3) 删除与加入：在$F$中删除作为左、右子树的两棵二叉树，并将新建立的二叉树加入到$F$中；

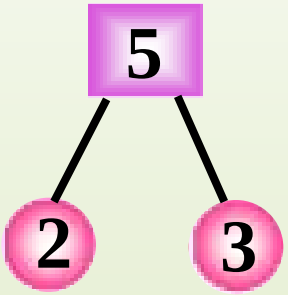
(4) 重复(2)、(3)两步，当集合$F$中只剩下一棵二叉树时，这棵二叉树便是哈夫曼树。

## $W=\{2，3，4，5\}$ 哈夫曼树的构造过程

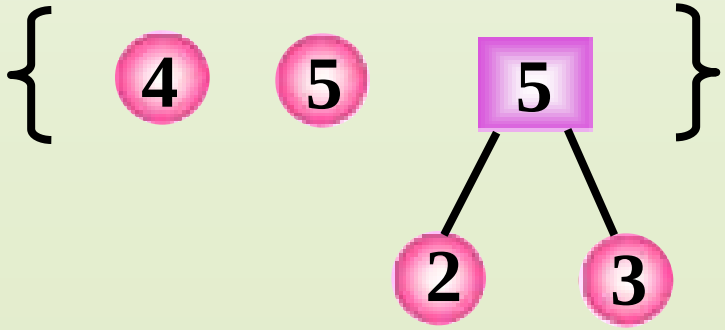
第1步：初始化




第2步：选取与合并

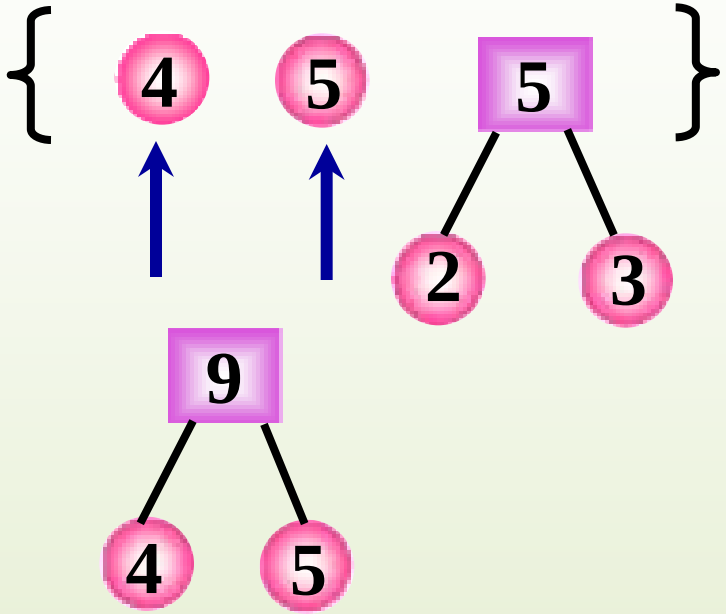



第3步：删除与加入

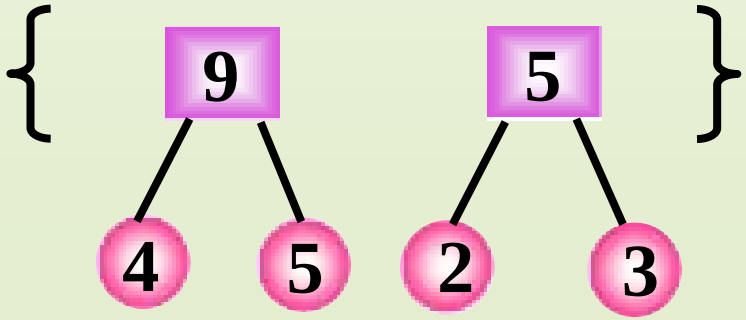


## $W$＝{2，3，4，5} 哈夫曼树的构造过程

重复第2步

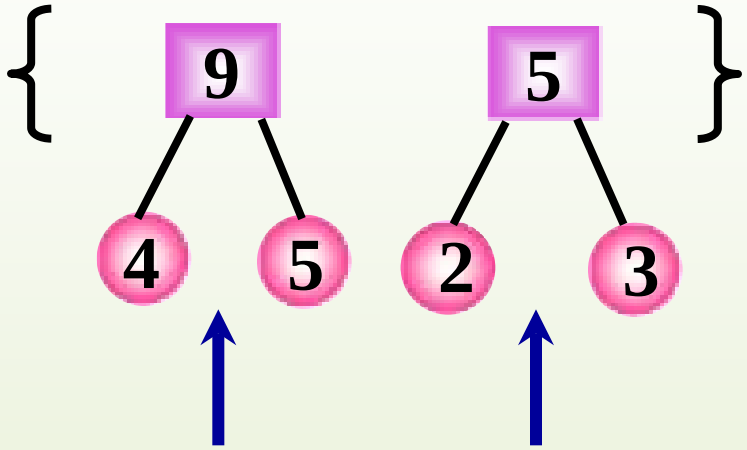



重复第3步

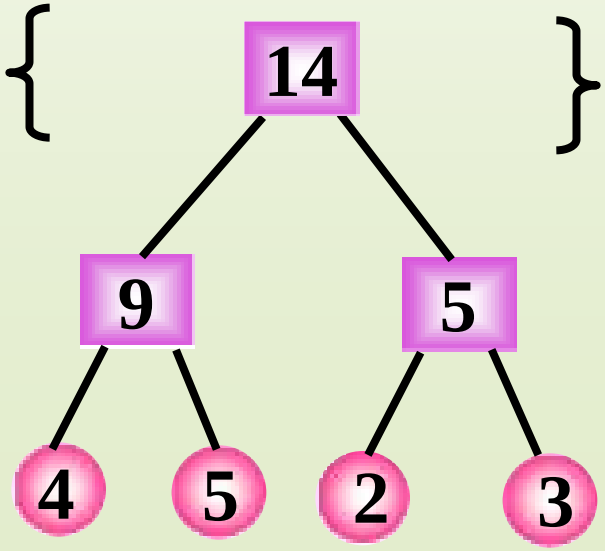


## $W$＝{2，3，4，5} 哈夫曼树的构造过程

重复第2步



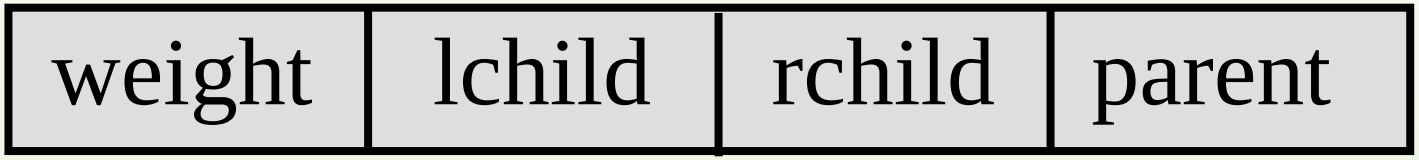

重复第3步



# 哈夫曼算法的存储结构

```
struct element
{     int weight;
      int lchild, rchild, parent;
};
```

1. 设置一个数组huffTree[ 信息，数组元素的结点结




其中：**weight**：权值域，保存该结点的权值；

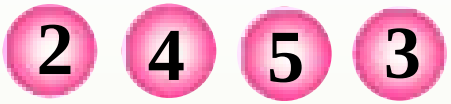
**lchild**：指针域，结点的左孩子结点在数组中的下标；

**rchild**：指针域，结点的右孩子结点在数组中的下标；

**parent**：指针域，该结点的双亲结点在数组中的下标。

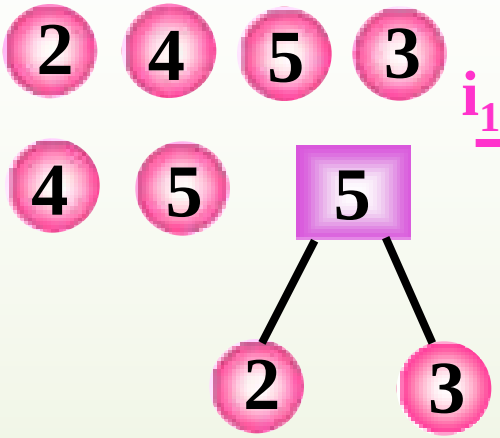
# 伪代码

1. 数组huffTree初始化，所有元素结点的双亲、左右孩子都置为−1；
2. 数组huffTree的前n个元素的权值置给定值w[n]；
3. 进行n−1次合并
   3.1 在二叉树集合中选取两个权值最小的根结点，其下标分别为$i_1$, $i_2$；
   3.2 将二叉树$i_1$、$i_2$合并为一棵新的二叉树k；




| | weight | parent | lchild | rchild |
|---|---|---|---|---|
| 0 | 2 | −1 | −1 | −1 |
| 1 | 4 | −1 | −1 | −1 |
| 2 | 5 | −1 | −1 | −1 |
| 3 | 3 | −1 | −1 | −1 |
| 4 | | −1 | −1 | −1 |
| 5 | | −1 | −1 | −1 |
| 6 | | −1 | −1 | −1 |

初 态



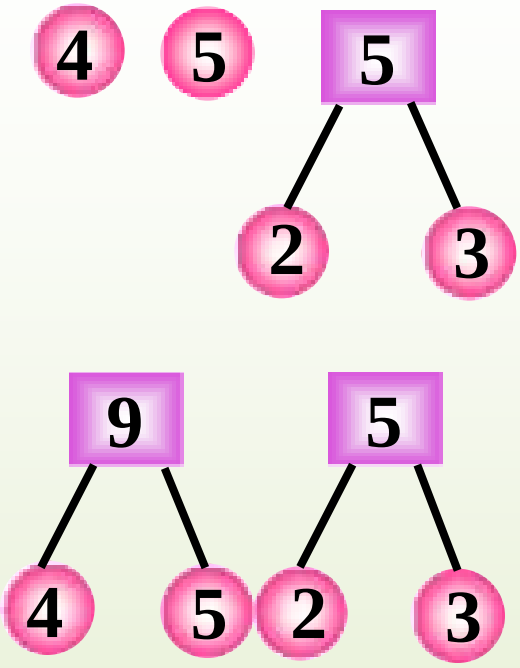

$i_1$ → 0

$i_2$ → 3

*k* → 4

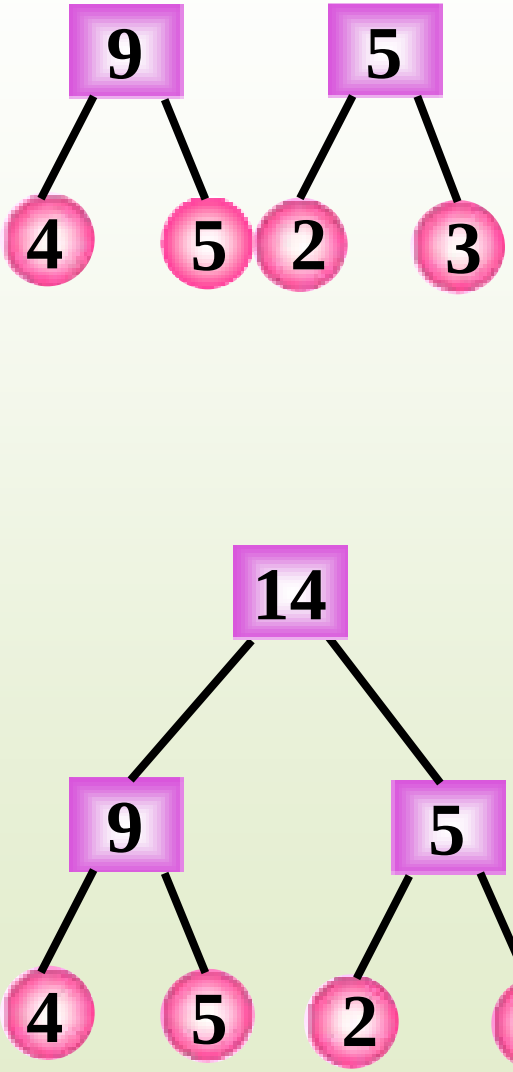
| | weight | parent | lchild | rchild |
|---|---|---|---|---|
| 0 | 2 | 4 ~~−1~~ | −1 | −1 |
| 1 | 4 | −1 | −1 | −1 |
| 2 | 5 | −1 | −1 | −1 |
| 3 | 3 | 4 ~~−1~~ | −1 | −1 |
| 4 | 5 | −1 | 0 ~~−1~~ | 3 ~~−1~~ |
| 5 | | −1 | −1 | −1 |
| 6 | | −1 | −1 | −1 |

过程




| | weight | parent | lchild | rchild |
|---|---|---|---|---|
| 0 | 2 | 4 ~~−1~~ | −1 | −1 |
| 1 ($i_1$) | 4 | 5 ~~−1~~ | −1 | −1 |
| 2 ($i_2$) | 5 | 5 ~~−1~~ | −1 | −1 |
| 3 | 3 | 4 ~~−1~~ | −1 | −1 |
| 4 | 5 | −1 | 0 ~~−1~~ | 3 ~~−1~~ |
| 5 ($k$) | 9 | −1 | 1 ~~−1~~ | 2 ~~−1~~ |
| 6 | | −1 | −1 | −1 |

过程




| | weight | parent | lchild | rchild |
|---|---|---|---|---|
| 0 | 2 | 4 ~~-1~~ | -1 | -1 |
| 1 | 4 | 5 ~~-1~~ | -1 | -1 |
| $i_1$ → 2 | 5 | 5 ~~-1~~ | -1 | -1 |
| 3 | 3 | 4 ~~-1~~ | -1 | -1 |
| 4 | 5 | 6 ~~-1~~ | 0 ~~-1~~ | 3 ~~-1~~ |
| $i_2$ → 5 | 9 | 6 ~~-1~~ | 1 ~~-1~~ | 2 ~~-1~~ |
| $k$ → 6 | 14 | -1 | 4 ~~-1~~ | 5 ~~-1~~ |

过程

```
void HuffmanTree（element huffTree[ ], int w[ ], int n ）{
    for（i=0; i<2*n−1; i++）{
        huffTree [i].parent= −1;
        huffTree [i].lchild= −1;
        huffTree [i].rchild= −1;
    }
    for（i=0; i<n; i++）
        huffTree [i].weight=w[i];
    for（k=n; k<2*n−1; k++）{
        Select（huffTree, i1, i2）;
        huffTree[k].weight=huffTree[i1].weight+huffTree[i2].weight;
        huffTree[i1].parent=k;
        huffTree[i2].parent=k;
        huffTree[k].lchild=i1;
        huffTree[k].rchild=i2;
    }
}
```

# 哈夫曼树应用——哈夫曼编码

编码：给每一个对象标记一个二进制位串来表示一组对象。

例：ASCII，指令系统

等长编码：表示一组对象的二进制位串的长度相等。

不等长编码：表示一组对象的二进制位串的长度不相等。

等长编码什么情况下空间效率高？

不等长编码什么情况下空间效率高？

- 哈夫曼树应用——哈夫曼编码

前缀编码：一组编码中任一编码都不是其它任何一个编码的前缀 。

前缀编码保证了在解码时不会有多种可能。

例：一组字符{A, B, C, D, E, F, G}出现的频率分别是{9, 11, 5, 7, 8, 2, 3}，设计最经济的编码方案。

- 哈夫曼树应用——哈夫曼编码

编码方案：

A：00

B：10

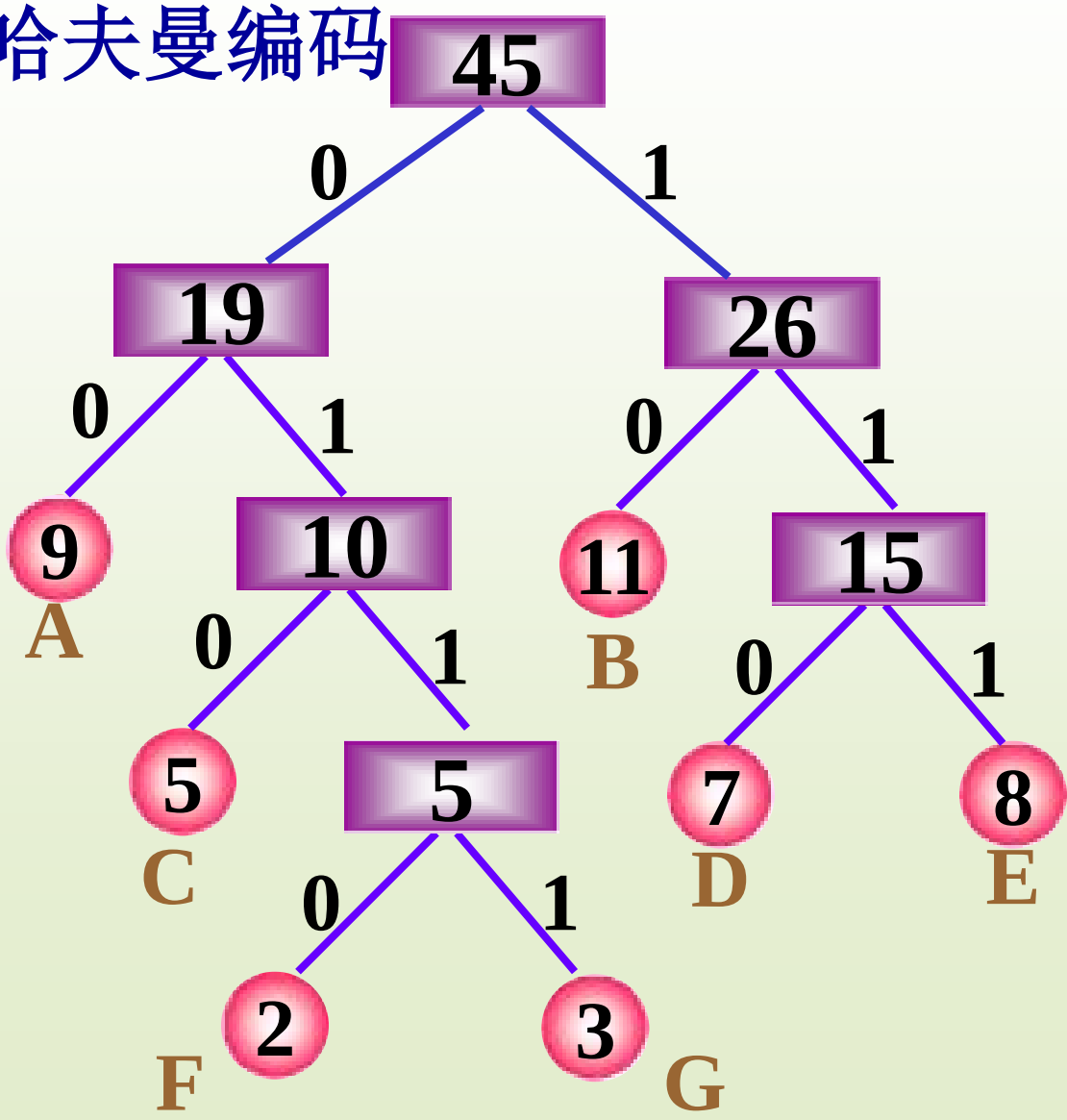
C：010

D：110

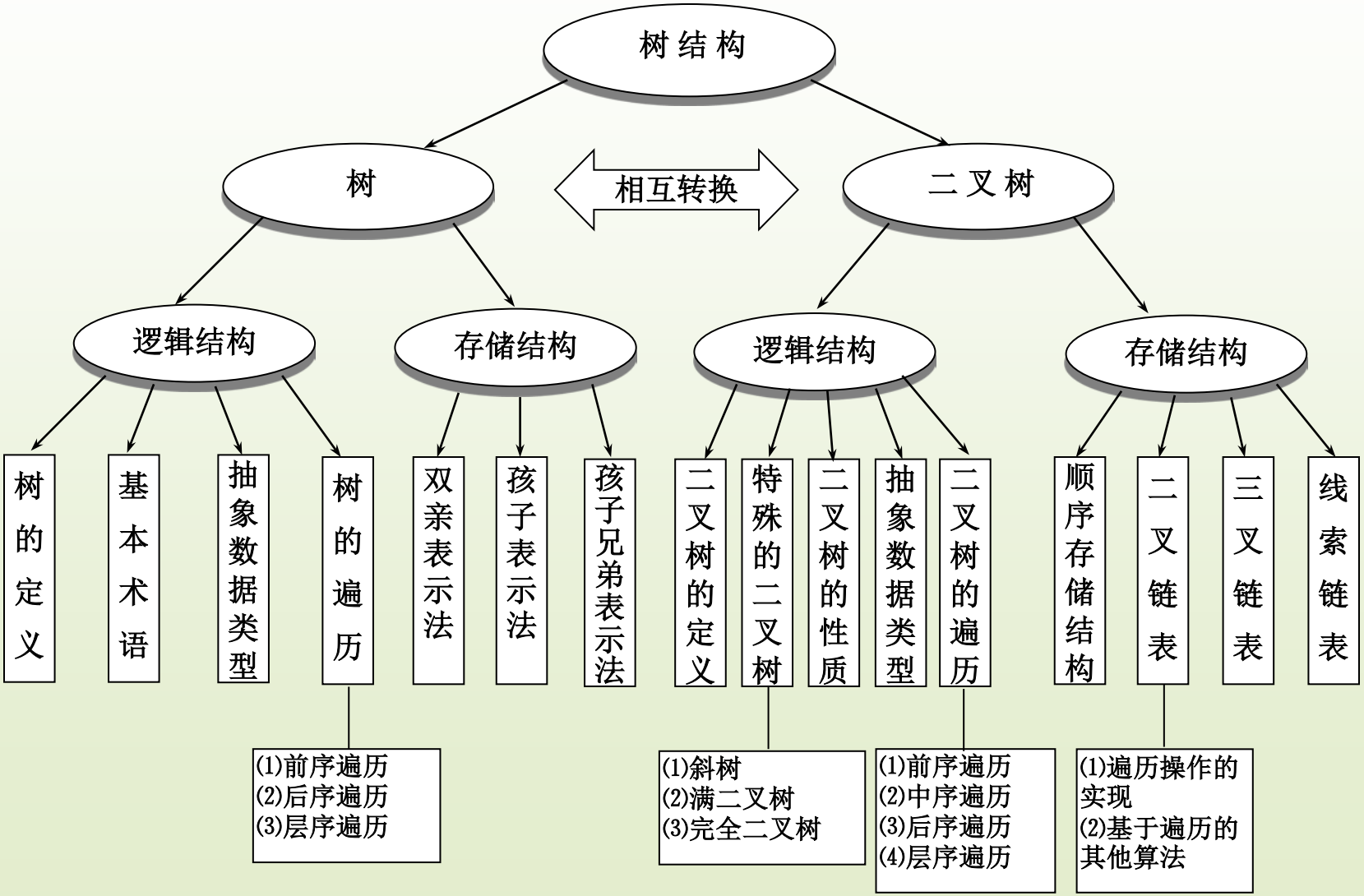
E：111

F：0110

G：0111



# 小结


树结构
树
相互转换
二叉树
逻辑结构
存储结构
逻辑结构
存储结构
树的定义
基本术语
抽象数据类型
树的遍历
双亲表示法
孩子表示法
孩子兄弟表示法
二叉树的定义
特殊的二叉树
二叉树的性质
抽象数据类型
二叉树的遍历
顺序存储结构
二叉链表
三叉链表
线索链表
(1)前序遍历
(2)后序遍历
(3)层序遍历
(1)斜树
(2)满二叉树
(3)完全二叉树
(1)前序遍历
(2)中序遍历
(3)后序遍历
(4)层序遍历
(1)遍历操作的实现
(2)基于遍历的其他算法

# 作业

1. 编写非递归算法，求二叉树中叶子结点的个数。

2. 已知某字符串*S*为“*abcdeacedaeadcedabadadaead*”，对该字符串用[0，1]进行前缀编码，问该字符串的编码至少有多少位。