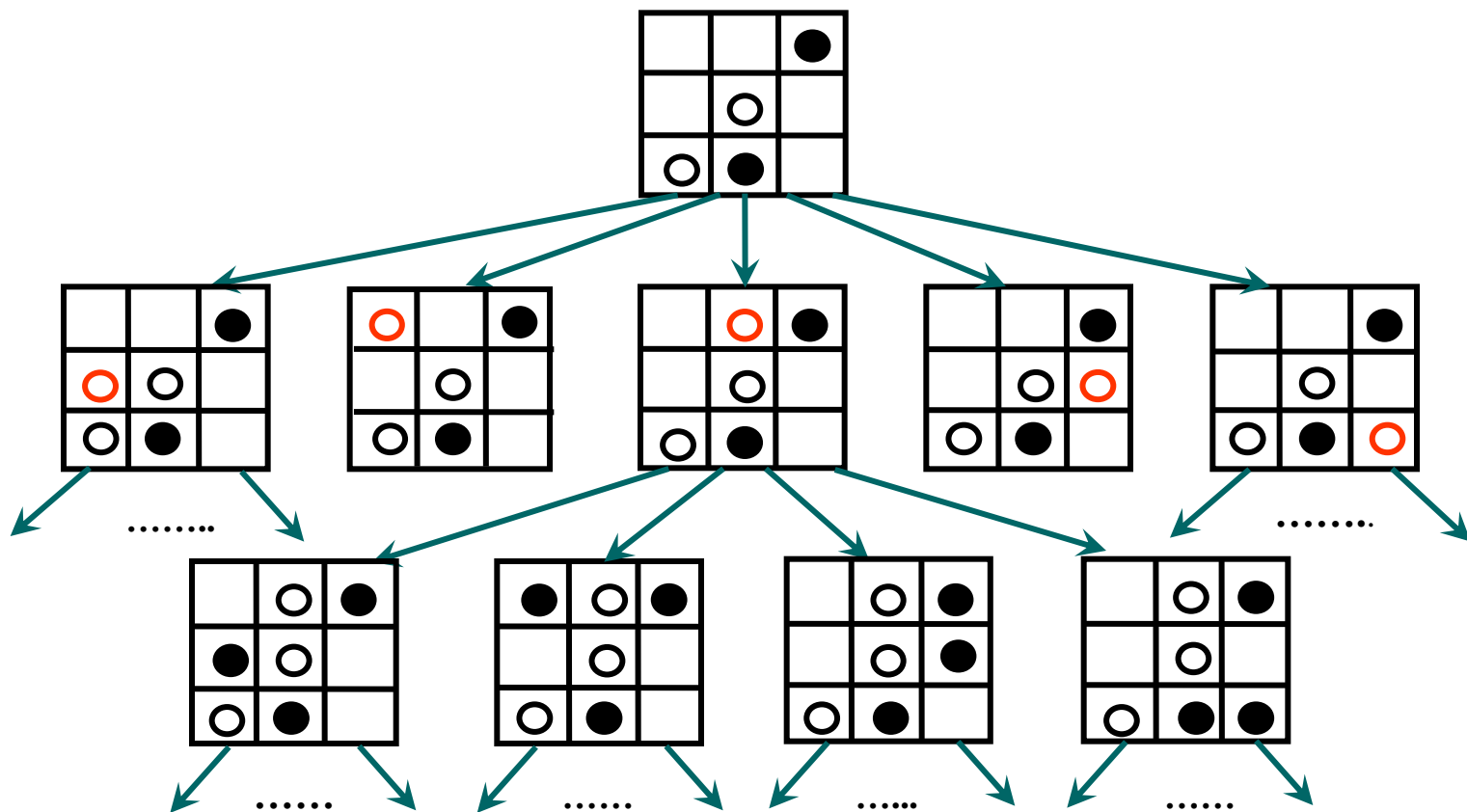


人机对弈问题



如何实现对弈？各格局之间是什么关系？
抽象出的模型是什么？

树结构

第 5 章 树和二叉树

本章的主要内容是

5.1 树的逻辑结构

5.2 树的存储结构

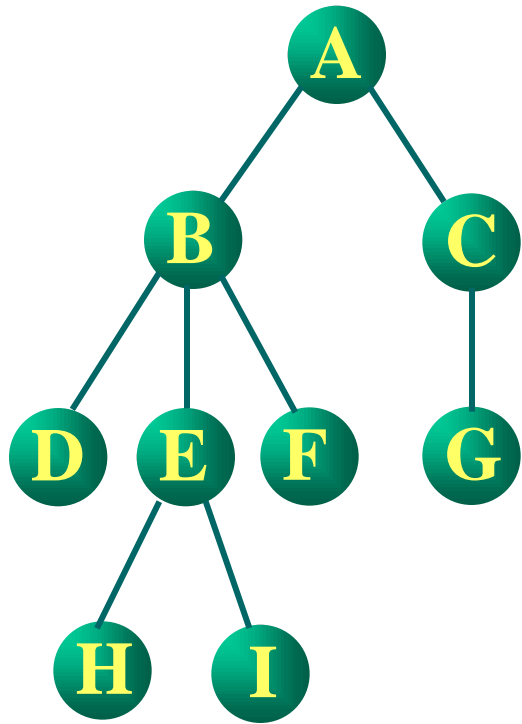
5.3 二叉树的逻辑结构

5.4 二叉树的存储结构及实现

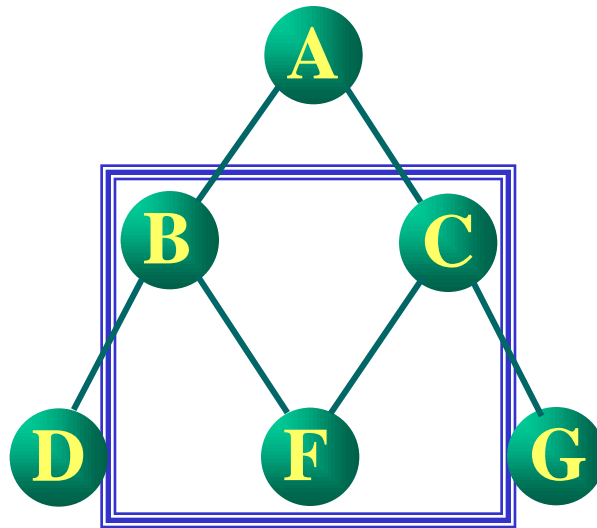
5.5 树、森林与二叉树的转换

5.6 哈夫曼树

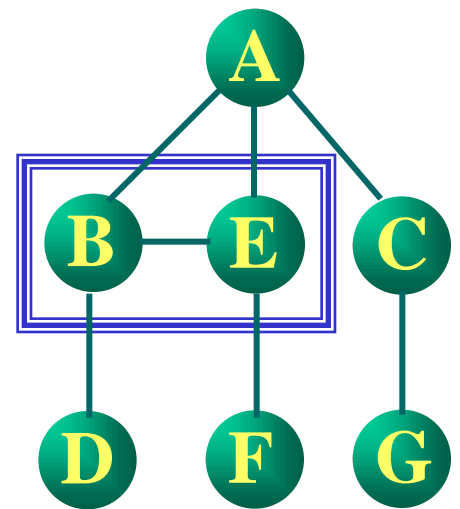
5.1 树的逻辑结构



(a) 一棵树结构



(b) 一个非树结构



(c) 一个非树结构

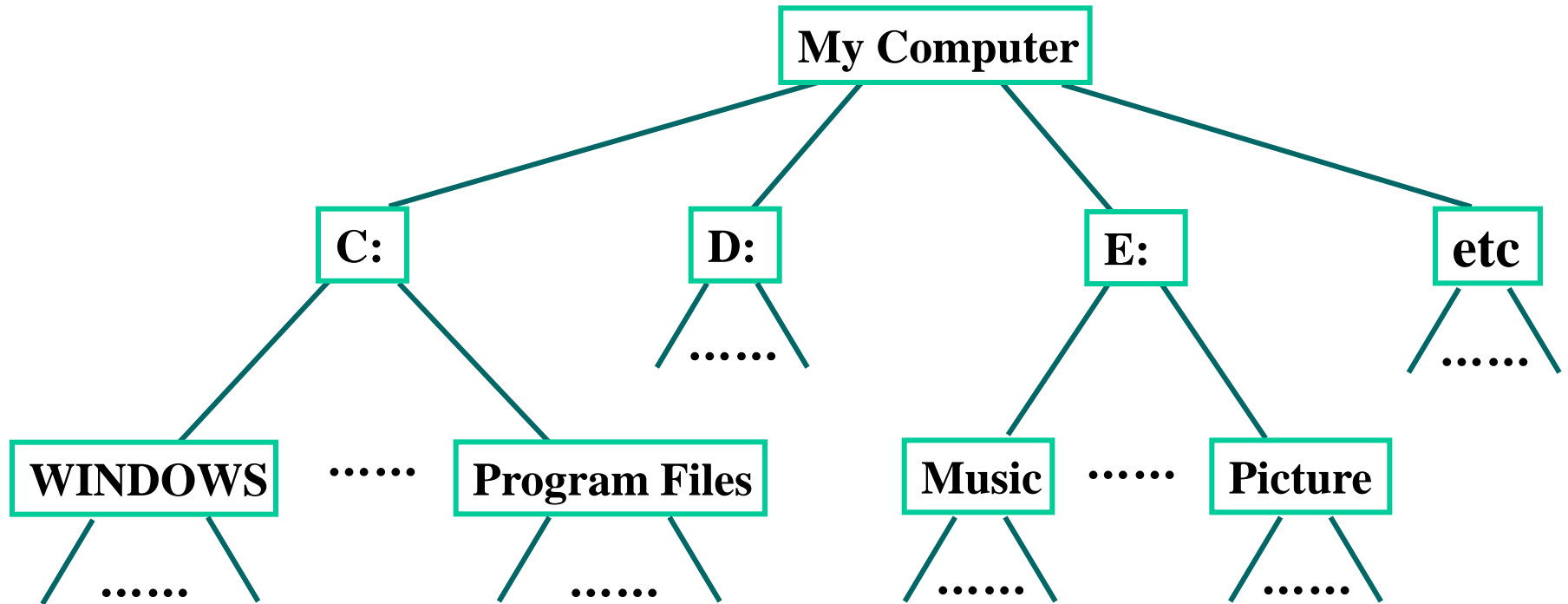
一、树的定义

树 (tree): 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:

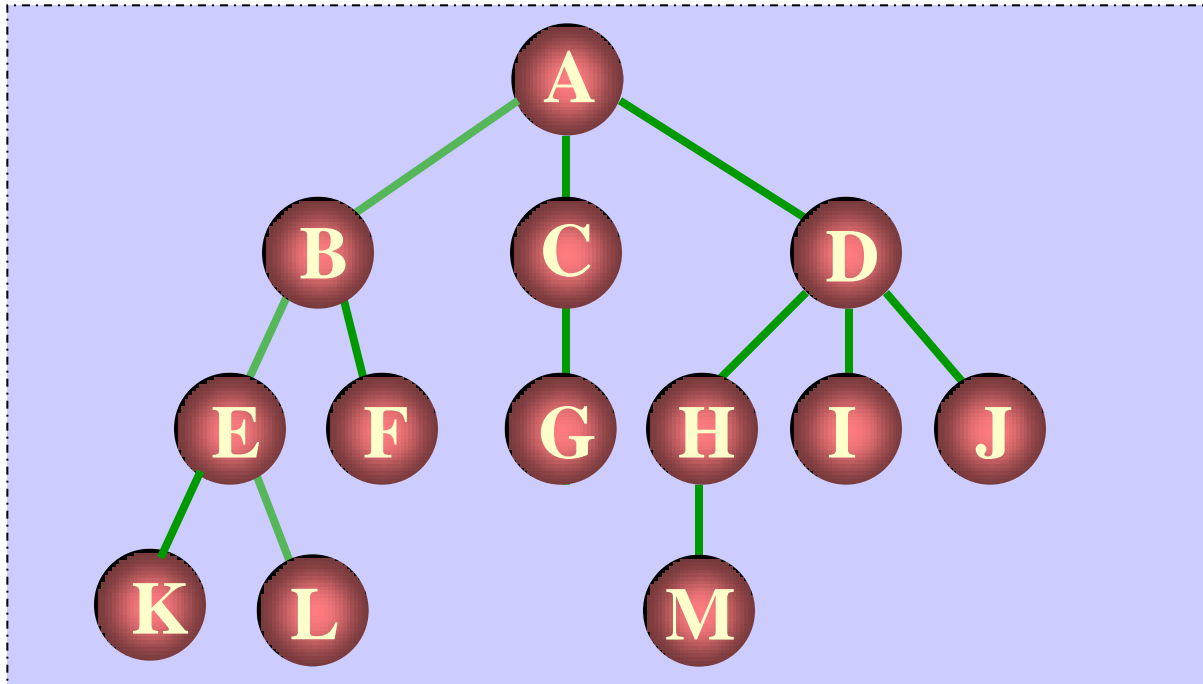
- (1) 有且仅有一个特定的称为**根 (root)**的结点;
- (2) 当 $n > 1$ 时, 除根结点之外的其余结点被分成 m ($m > 0$) 个**互不相交**的有限集合 T_1, T_2, \dots, T_m , 其中每个集合又是一棵树, 并称为这个根结点的**子树 (subtree)**。

树的定义是采用递归方法

例如：文件目录结构



二、树的基本术语



例如：

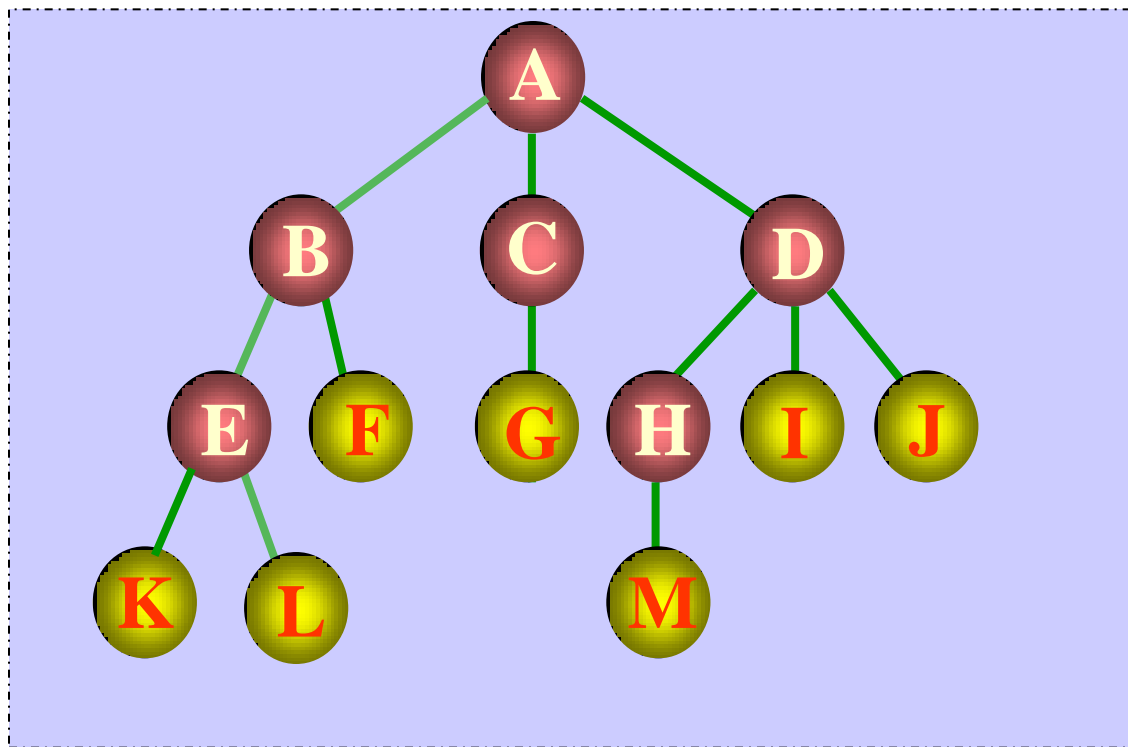
结点B的度
为2；

结点D的度
为3；

树的度为3。

结点的度 (degree)： 结点所拥有的子树的个数。

树的度： 树中各结点度的最大值。



例如：

叶子结点：

K, L, F, G, M, I, J;

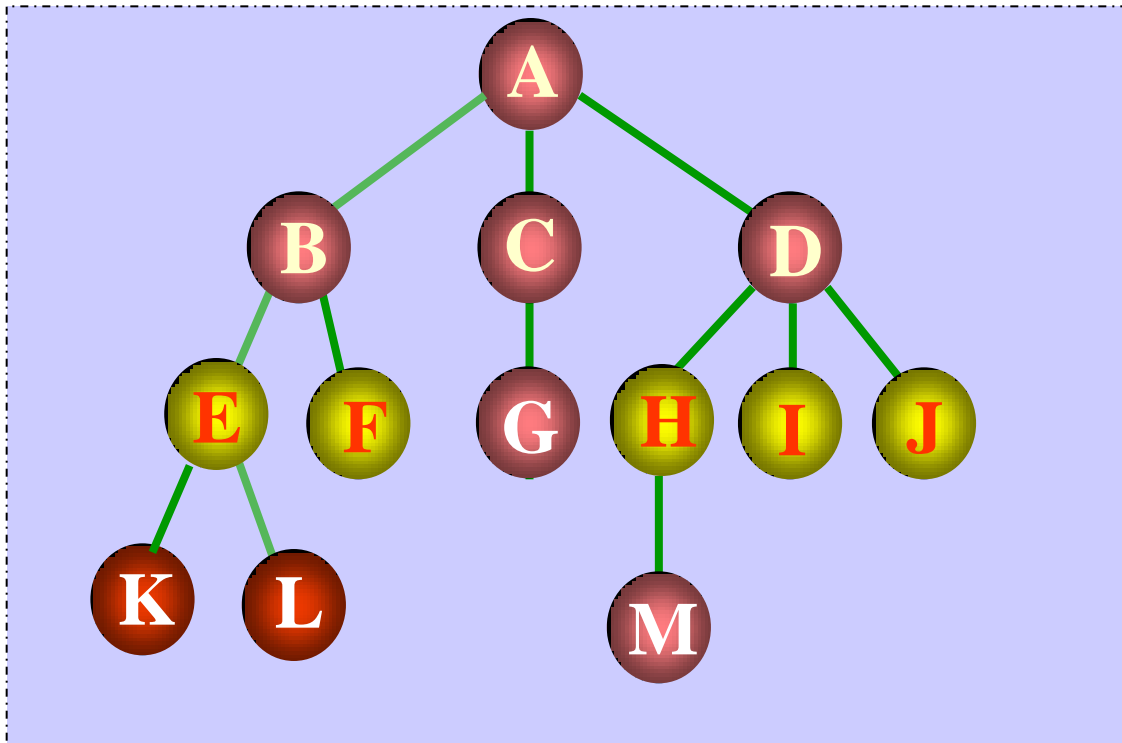
分支结点：

A, B, C, D, E。

叶子结点(leaf)：度为0的结点，也称为终端结点。

分支结点(branch)：度不为0的结点，也称为非终端结点。

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(children)**，这个结点称为它孩子结点的**双亲结点(parent)**；**兄弟：**具有同一个双亲的孩子结点互称为**兄弟结点(brother)**。

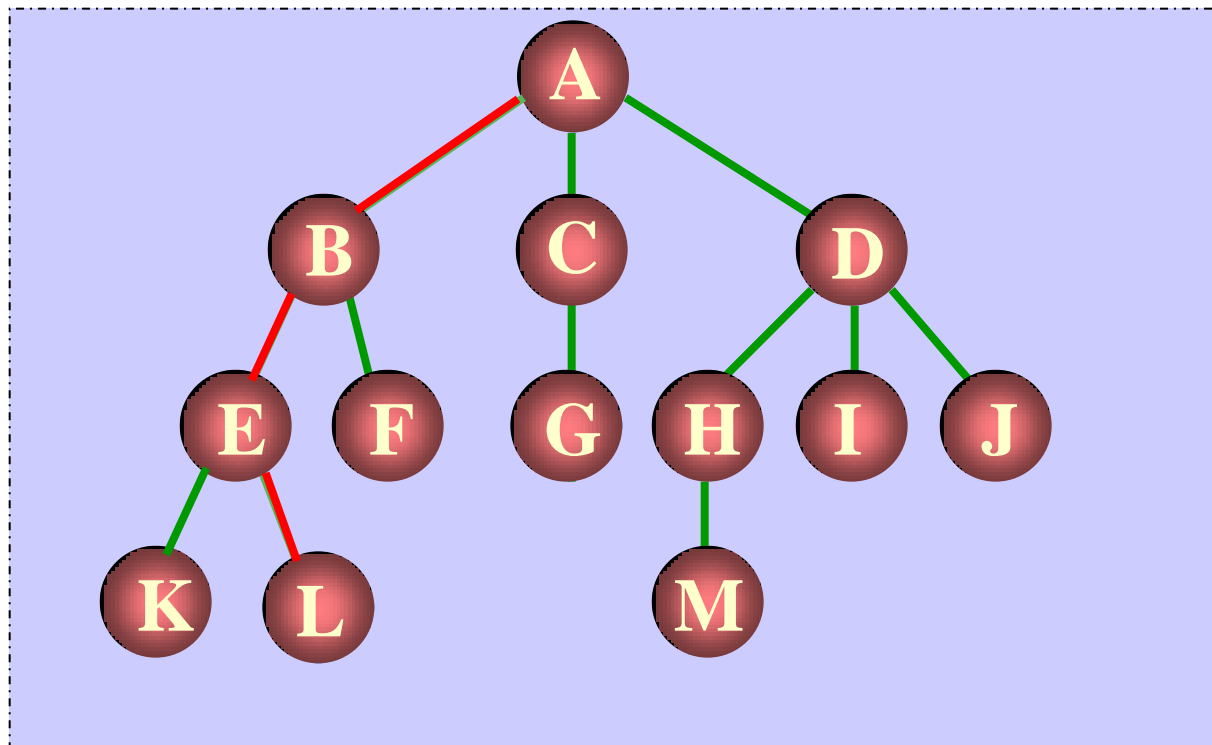


例如：

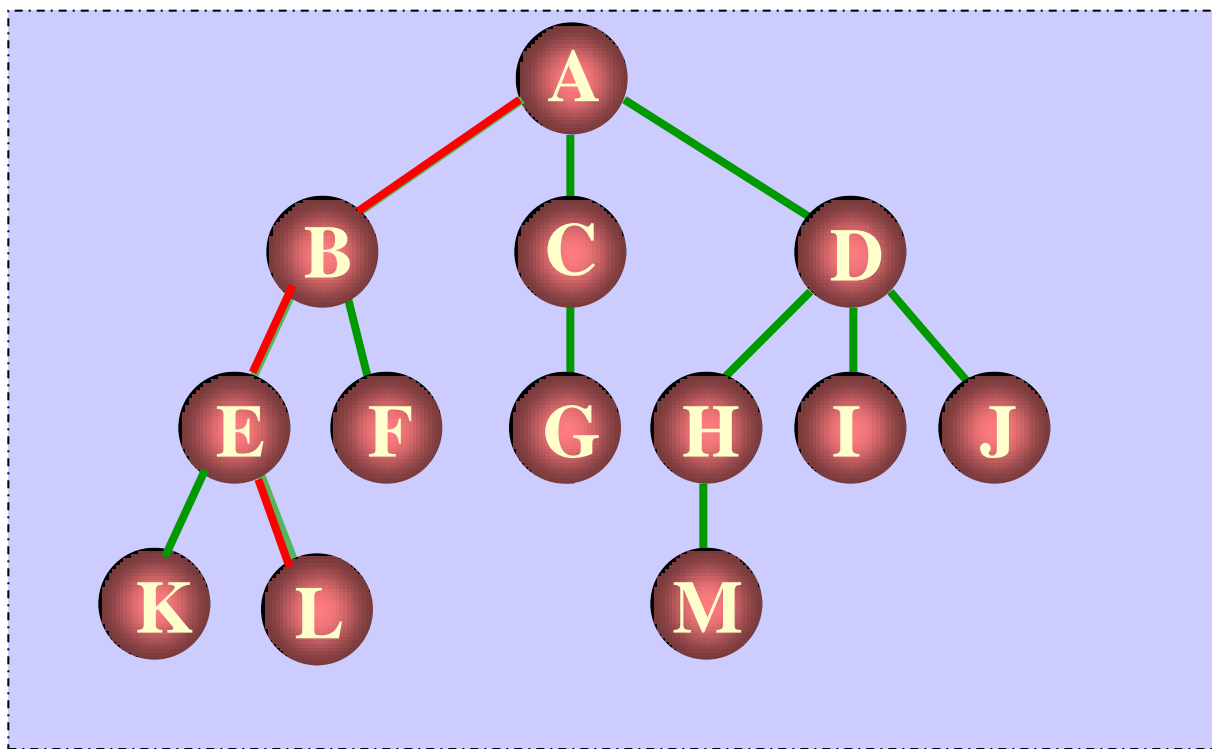
兄弟结点：

E, F; H, I, J。

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

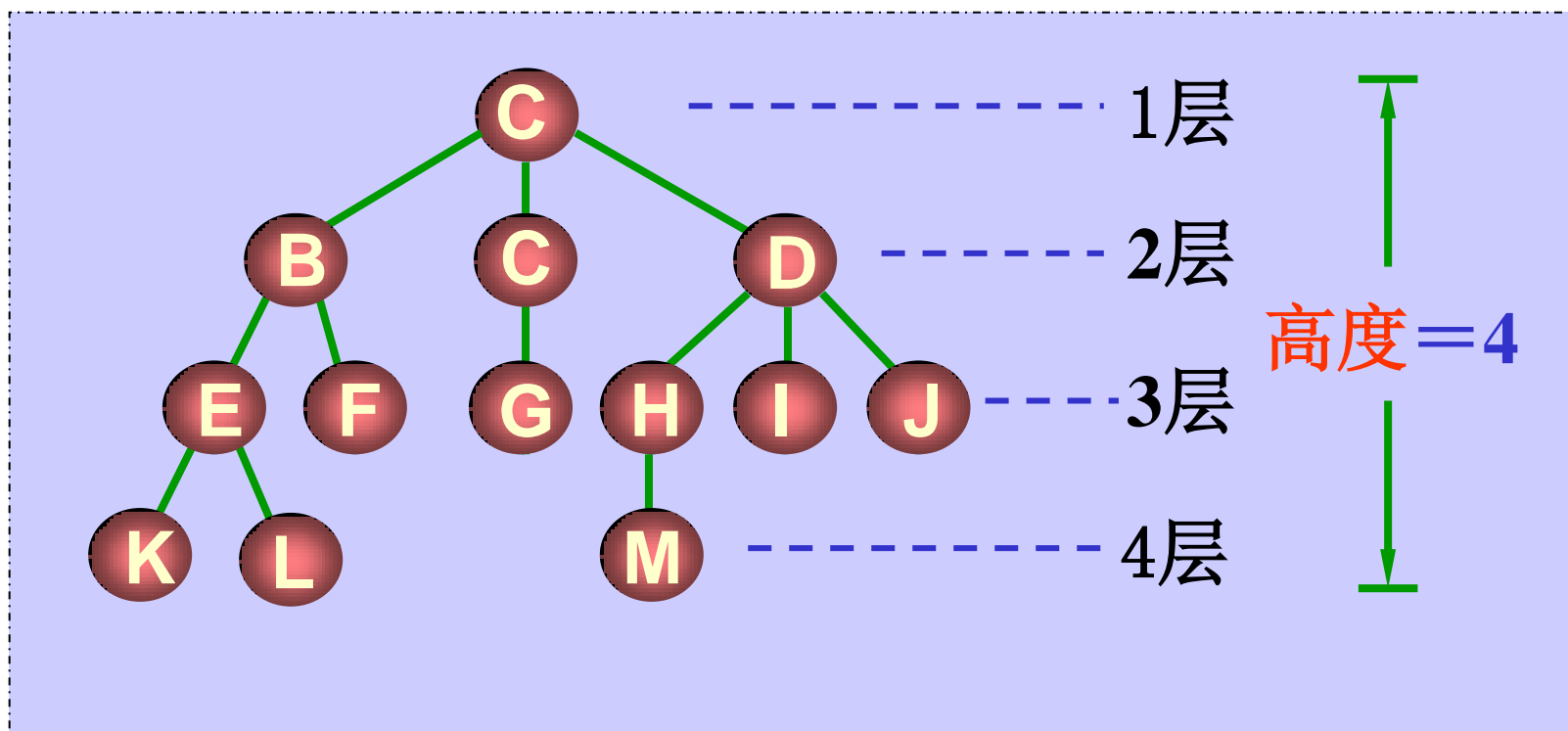


祖先、子孙：在树中，如果有一条路径从结点 x 到结点 y ，那么 x 就称为 y 的**祖先 (ancestor)**，而 y 称为 x 的**子孙 (descendant)**。

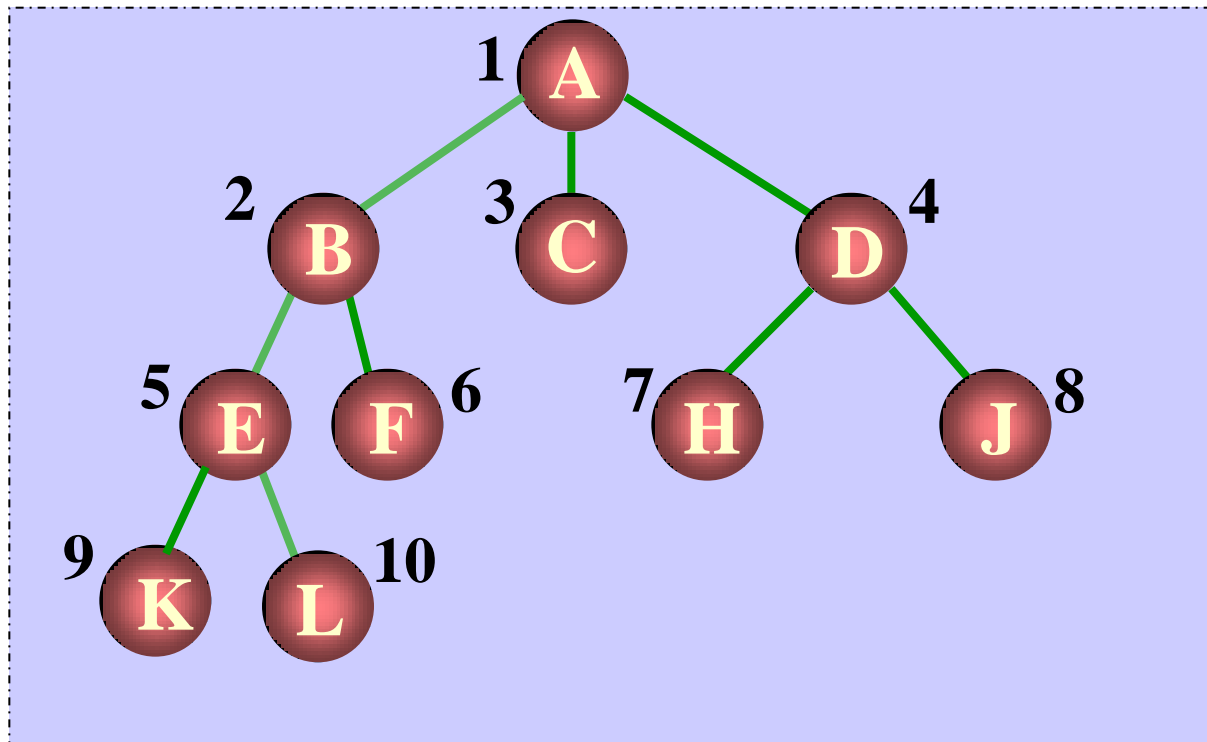


结点所在层(level)数：根结点的层数为1；对其余任何结点，若某结点在第k层，则其孩子结点在第k+1层。

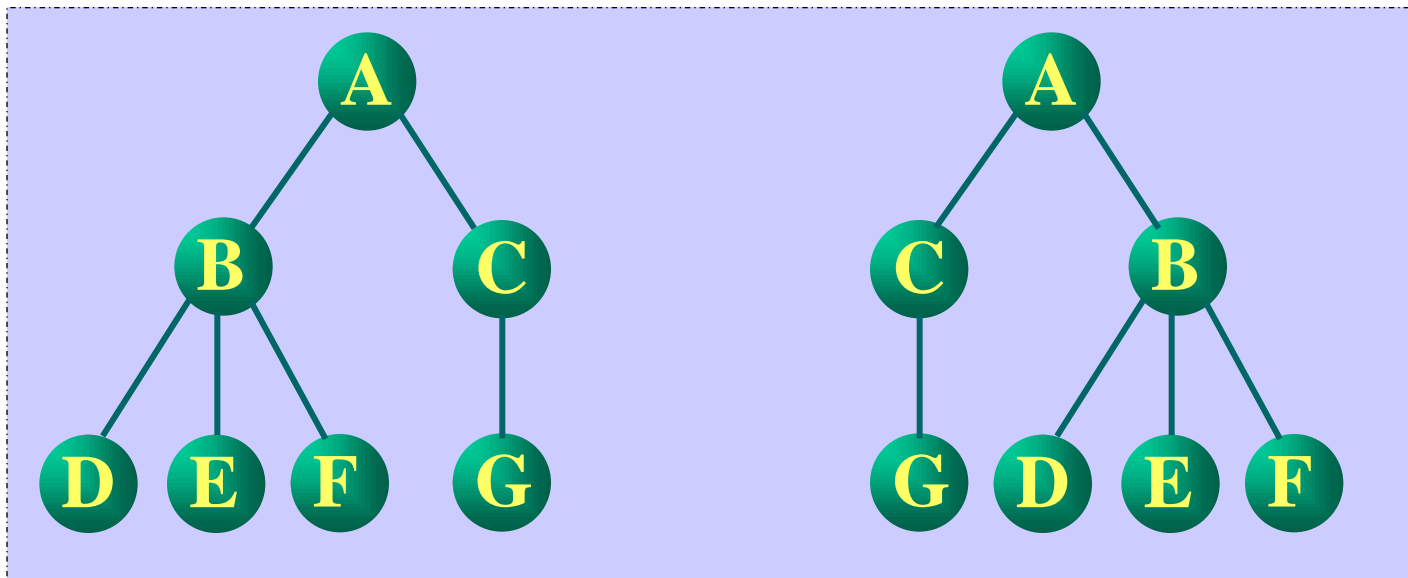
树的深度(depth)：树中所有结点的最大层数，也称**高度**



层序编号(level code): 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。

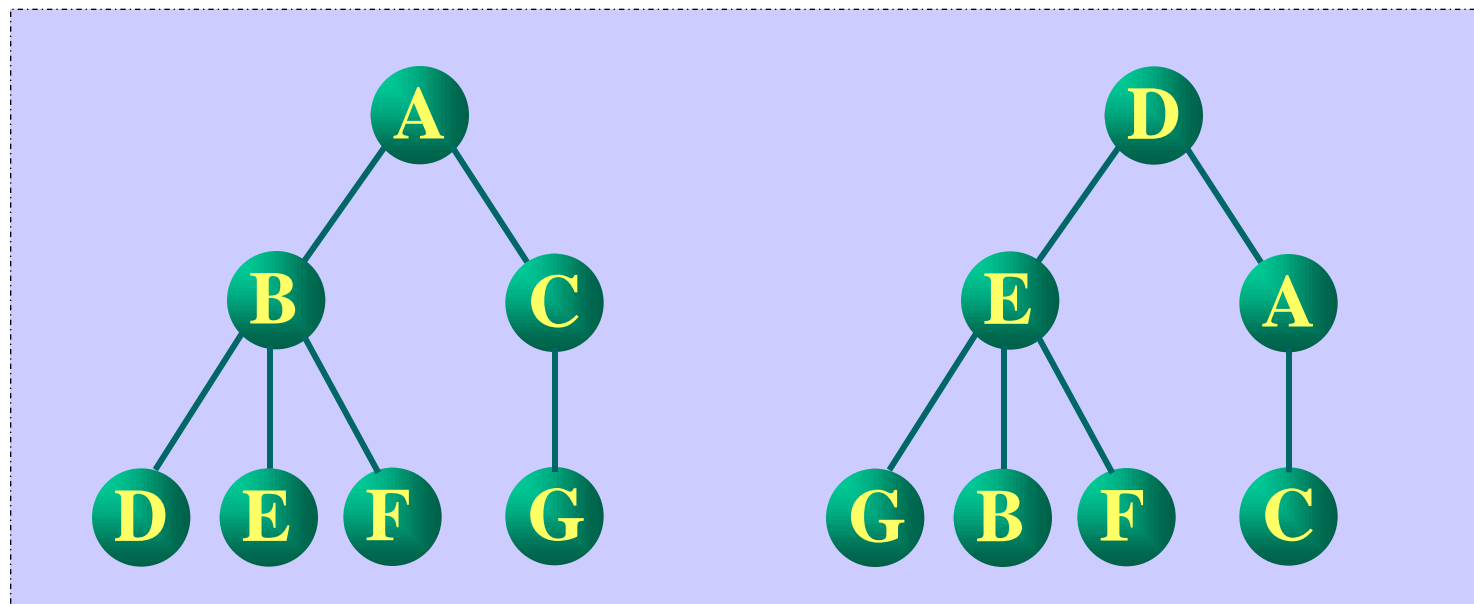


有序树、无序树：如果一棵树中结点的各子树从左到右是有次序的，称这棵树为**有序树 (ordered tree)**；反之，称为**无序树 (unordered tree)**。

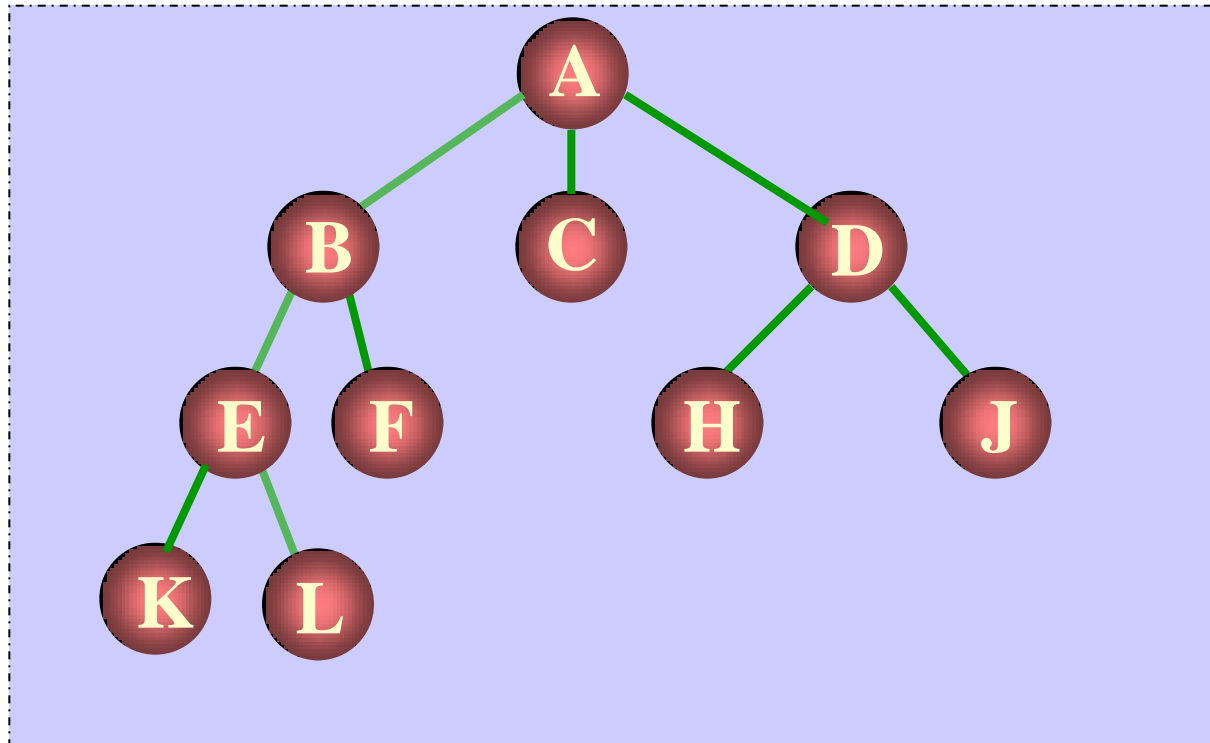


数据结构中讨论的一般都是有序树

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 (isomorphic)**。



森林(forest): m ($m \geq 0$) 棵互不相交的树的集合。



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**

三、树的遍历(traverse)

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。

① 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。

② 遍历的实质？

树结构（非线性结构）→线性结构。

③ 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。

1、前序遍历(preorder traverse)

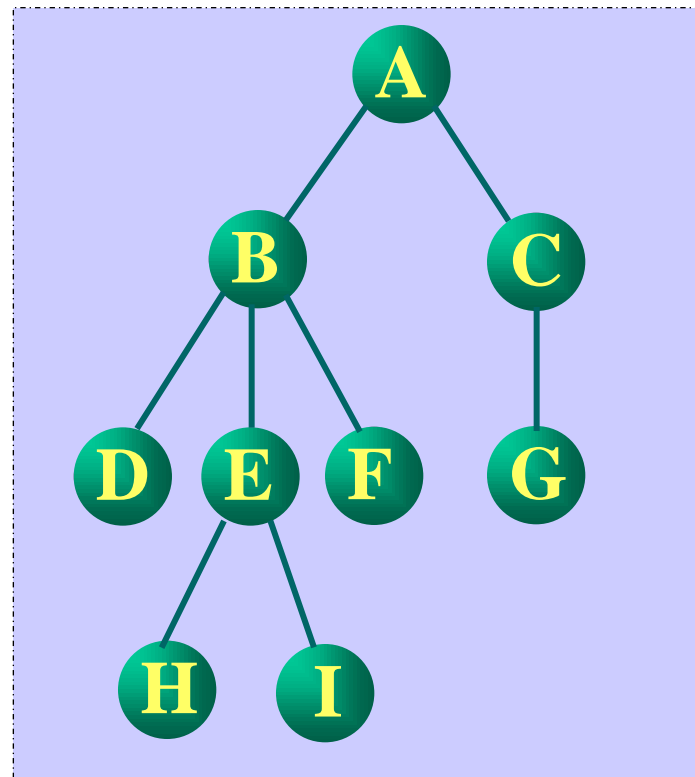
树的**前序遍历**操作定义为：

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。

例如，前序遍历序列为：

A B D E H I F C G



2、后序遍历(postorder traverse)

树的**后序遍历**操作定义为：

若树为空，则空操作返回；

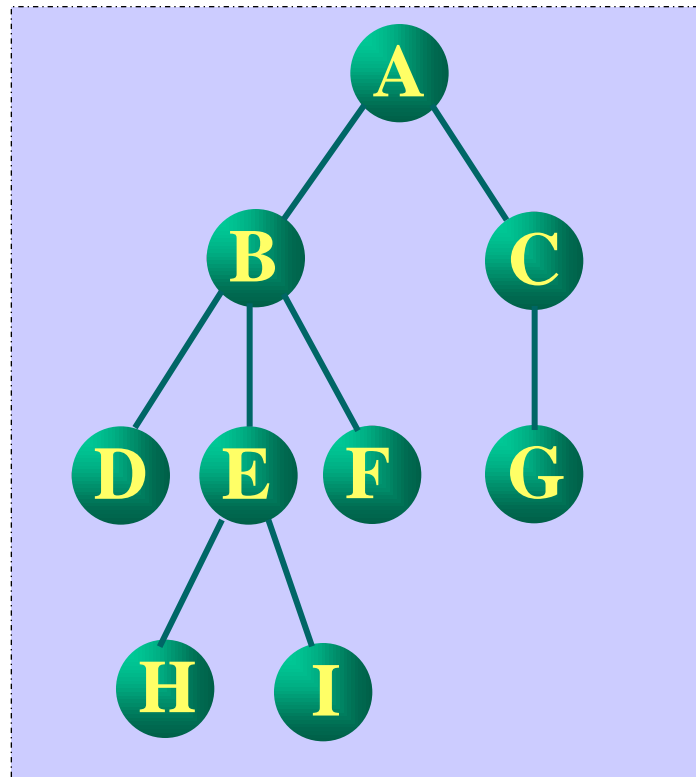
否则

(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。

例如：后序遍历序列为：

D H I E F B G C A

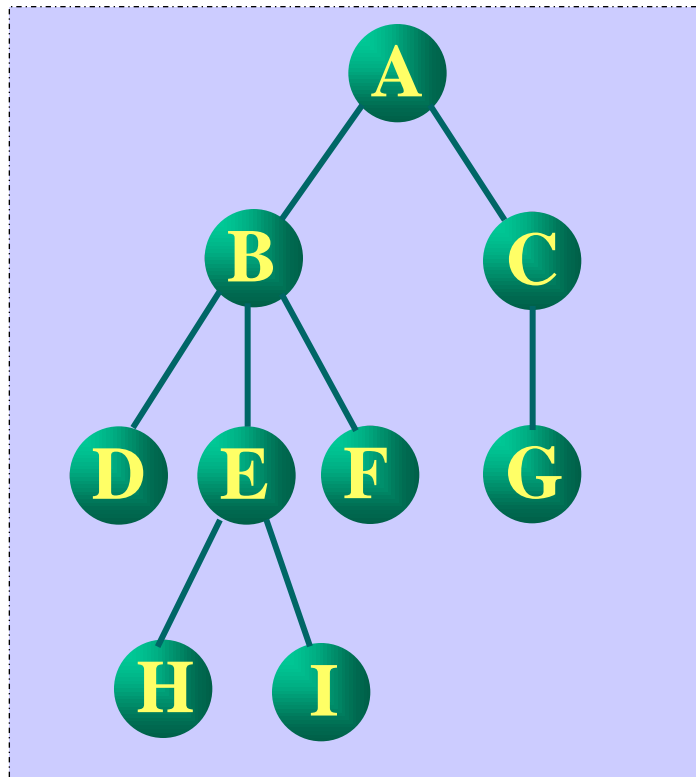


3、层序遍历(level traverse)

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。

例如：层序遍历序列：

A B C D E F G H I



5.2 树的存储结构

① 实现树的存储结构，关键问题是什么？

如何表示树中结点之间的逻辑关系。

② 树中结点之间的逻辑关系是什么？如何表示？

思考问题的出发点：如何表示结点的双亲和孩子

一、顺序存储方式

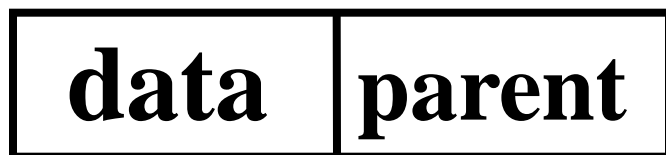
基本思想：用一维数组来存储树的各个结点（一般按**层序**存储），数组中的一个元素对应树中的一个结点，包括结点的数据信息以及该结点的双亲在数组中的下标。

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

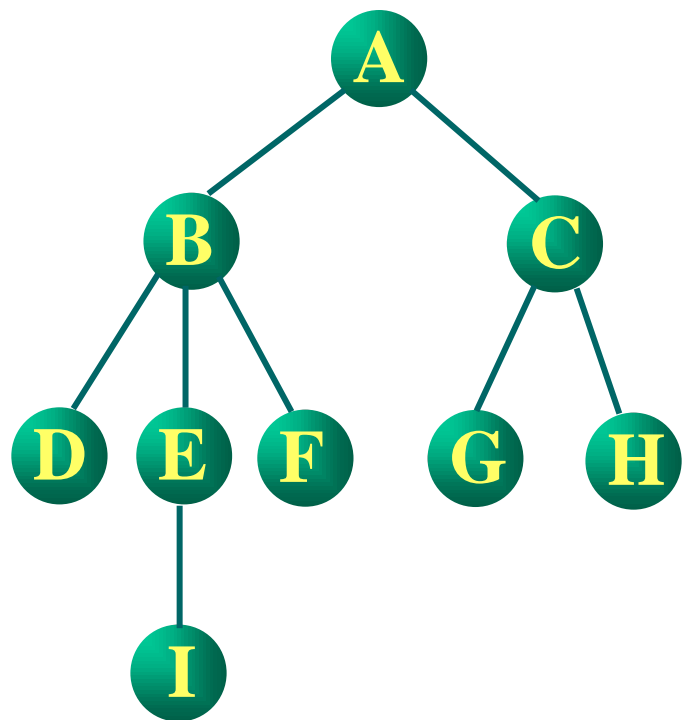
双亲表示法(parent representation)



```
typedef char dataType;  
struct PNode  
{  
    dataType 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。

双亲表示法示例:



② 如何查找双亲结点?
时间性能?

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

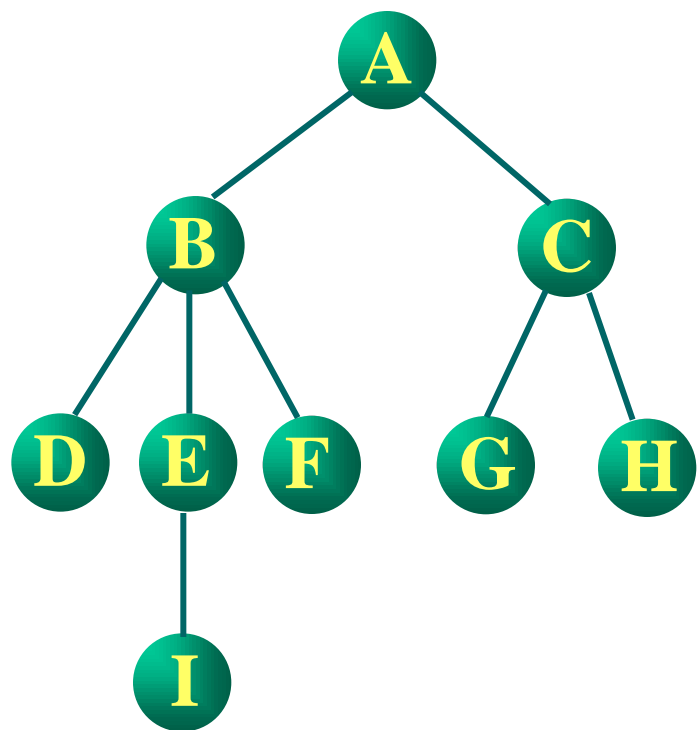
2

8

I

4

双亲表示法扩展（一）

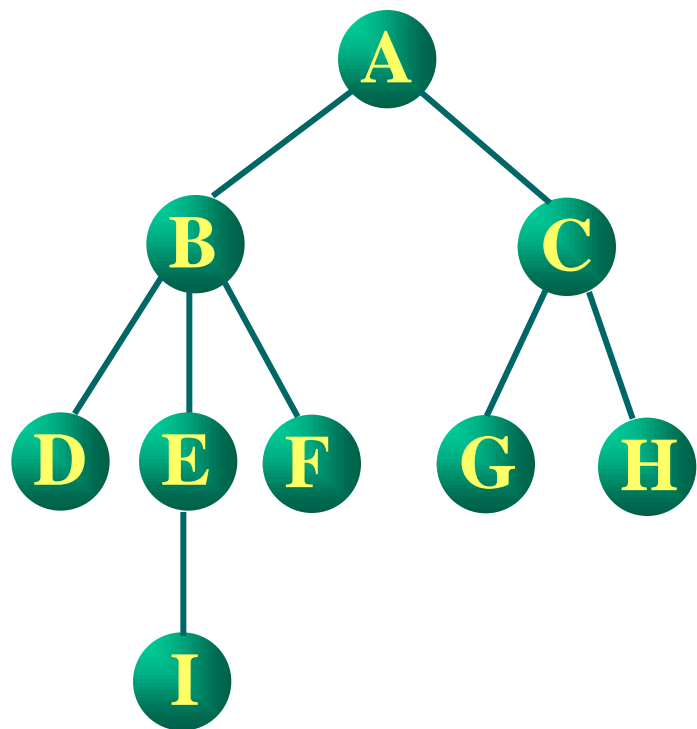


如何查找孩子结点？

时间性能？

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

双亲表示法扩展（二）



如何查找兄弟结点？

时间性能？

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

二、链式存储方式

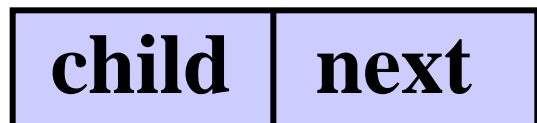
① 如何确定链表中的结点结构？

将每个结点的所有孩子放在一起，构成一个线性表。

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**。

1、孩子（链表）表示法(child representation)

孩子结点



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

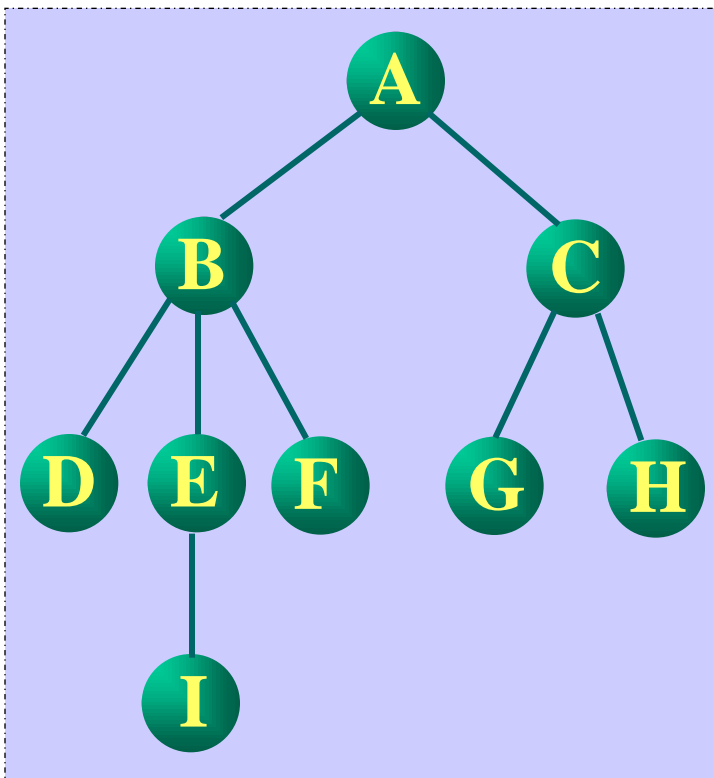
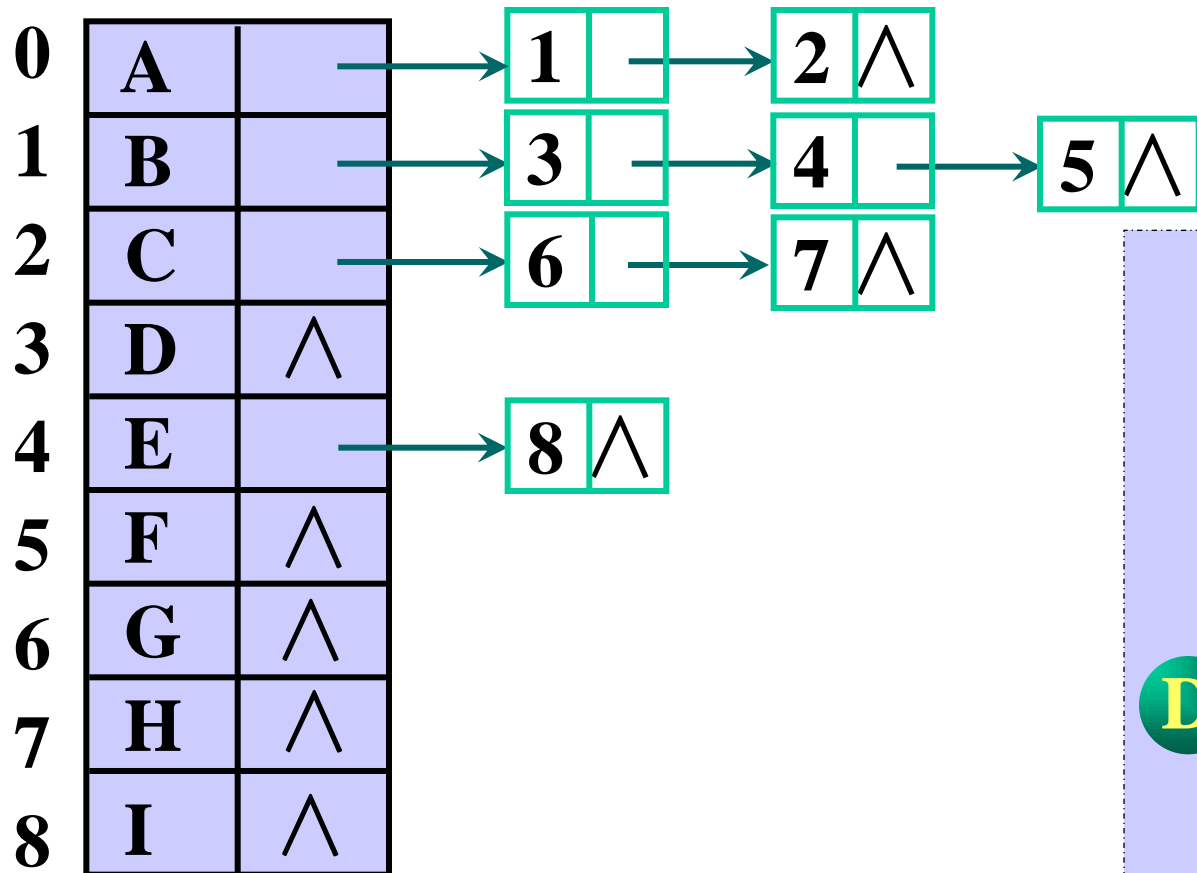
表头结点



```
typedef char dataType;
struct CBNode
{
    dataType data;
    CTNode *firstchild;
};
```

孩子（链表）表示法示例

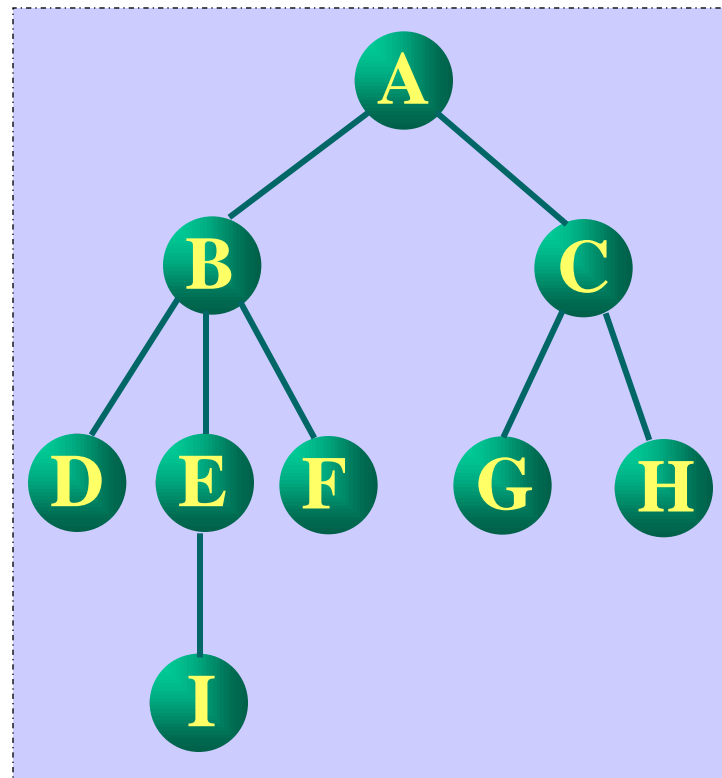
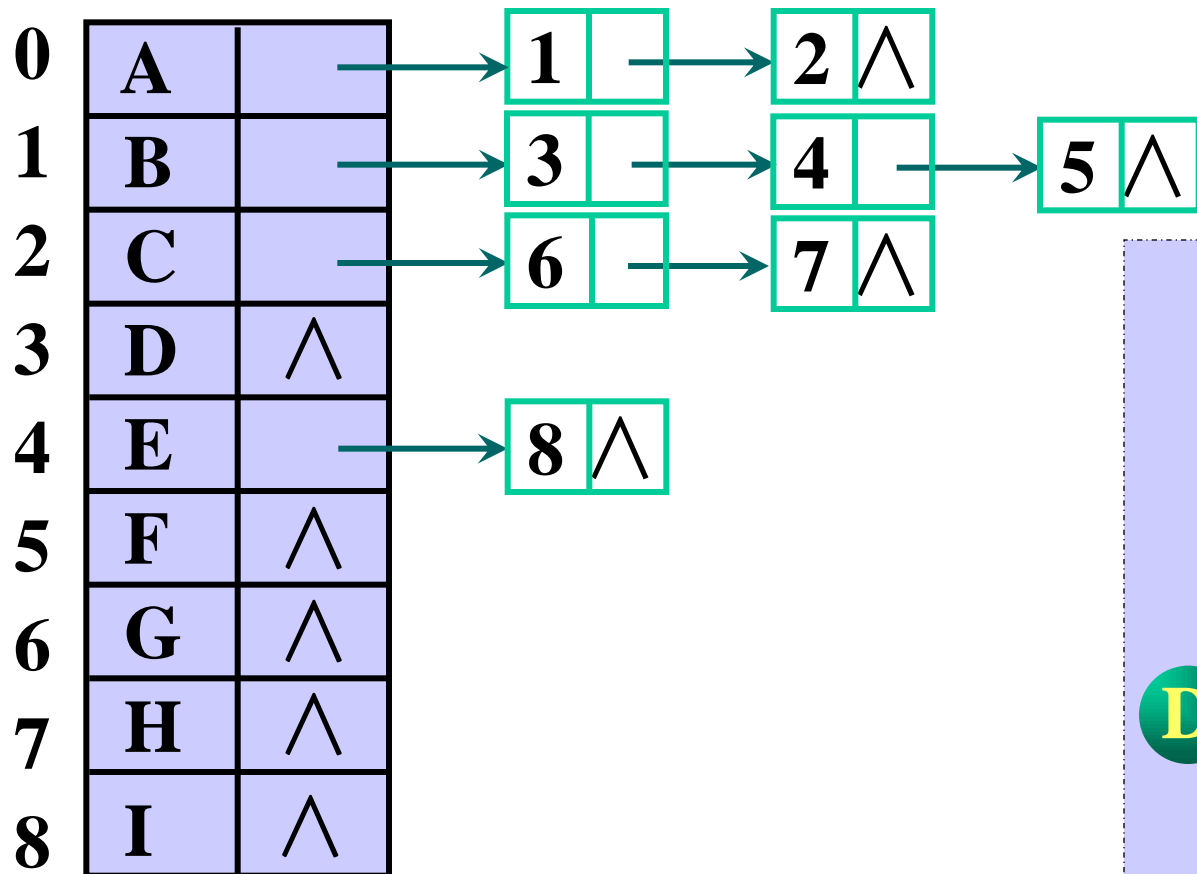
下标 data firstchild



❓ 如何查找孩子结点？时间性能？

孩子（链表）表示法示例

下标 data firstchild



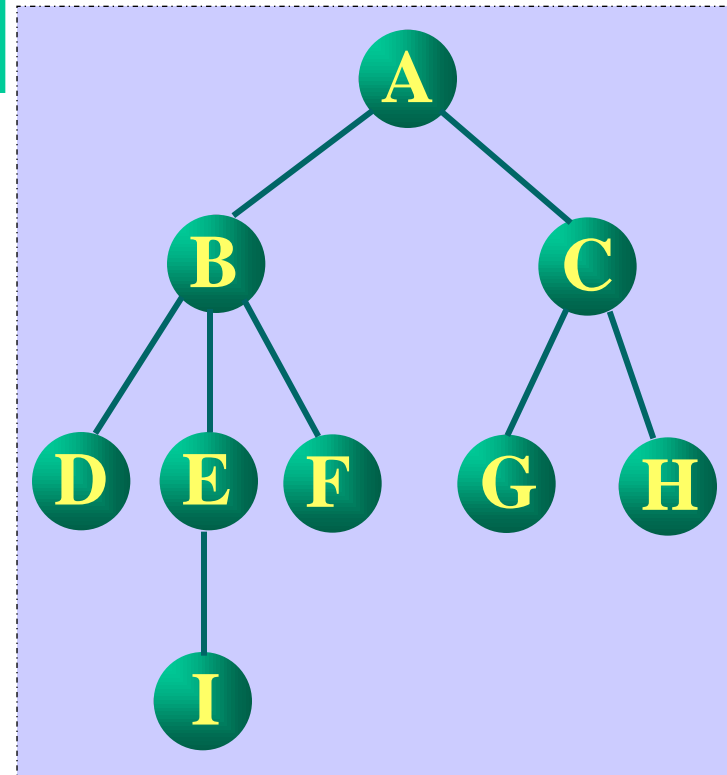
❓ 如何查找双亲结点？时间性能？

2、双亲孩子表示法(parent-child representation)

data parent firstchild

0	A	-1		→	1	-	2	∧	
1	B	0		→	3	-	4	-	→ 5
2	C	0		→	6	-	7	∧	
3	D	1	∧						
4	E	1		→	8	∧			
5	F	1	∧						
6	G	2	∧						
7	H	2	∧						
8	I	4	∧						

```
graph TD; B((B)) --- D((D)); B --- E((E)); E --- F(( ));
```

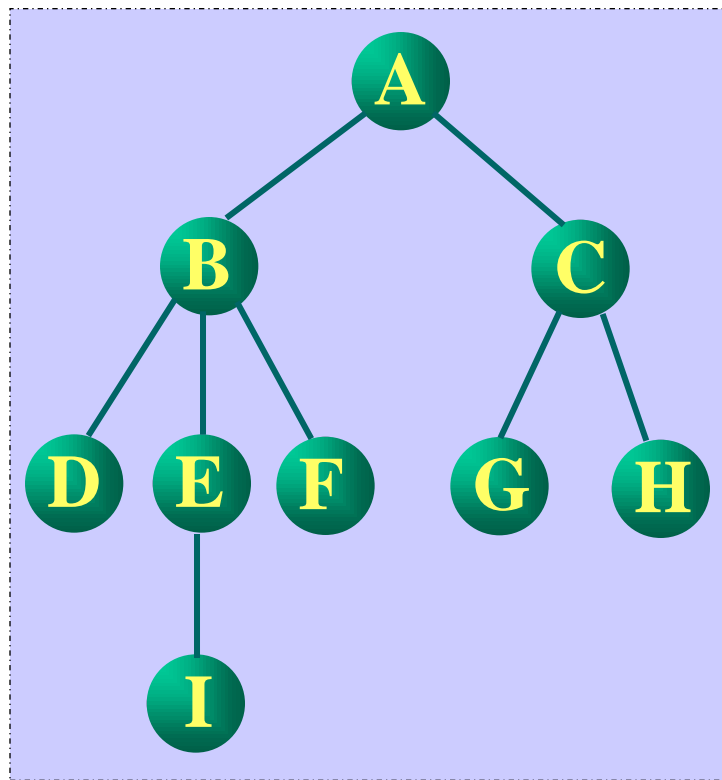


① 能否简化用链表表示?

某结点的第一个孩子是唯一
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



3、孩子兄弟表示法(child-brother representation)

结点结构	<table><tr><td>firstchild</td><td>data</td><td>rightsib</td></tr></table>	firstchild	data	rightsib
firstchild	data	rightsib		

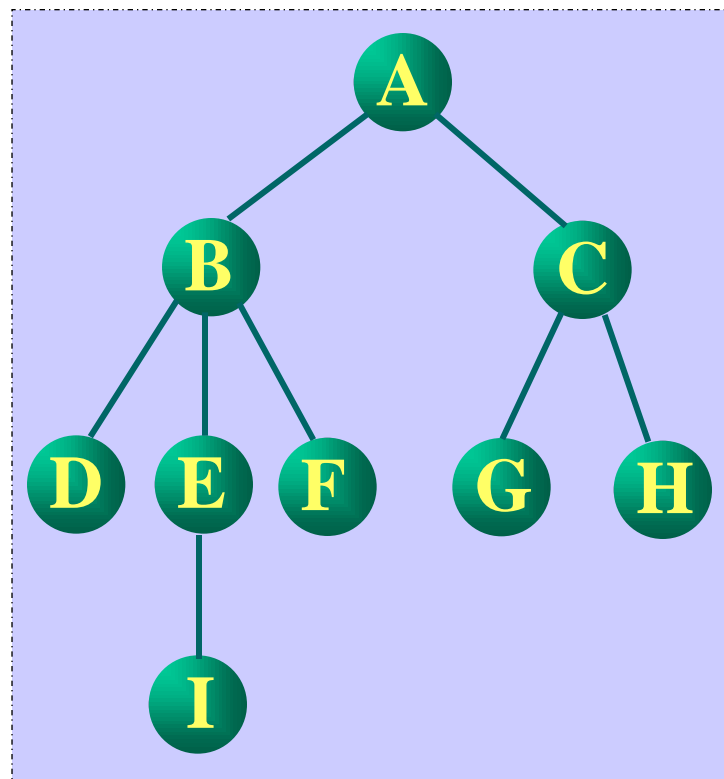
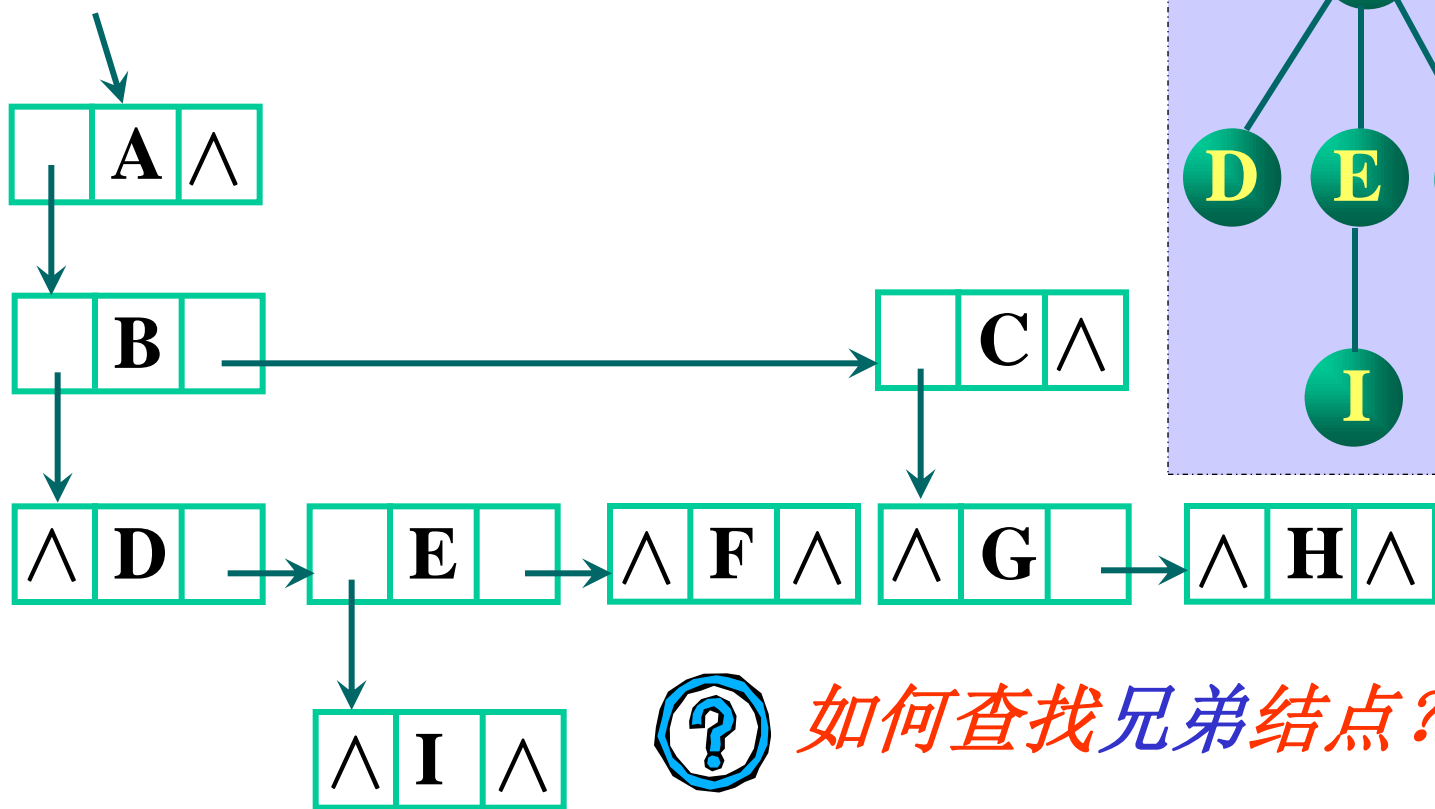
data: 数据域, 存储该结点的数据信息;

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。

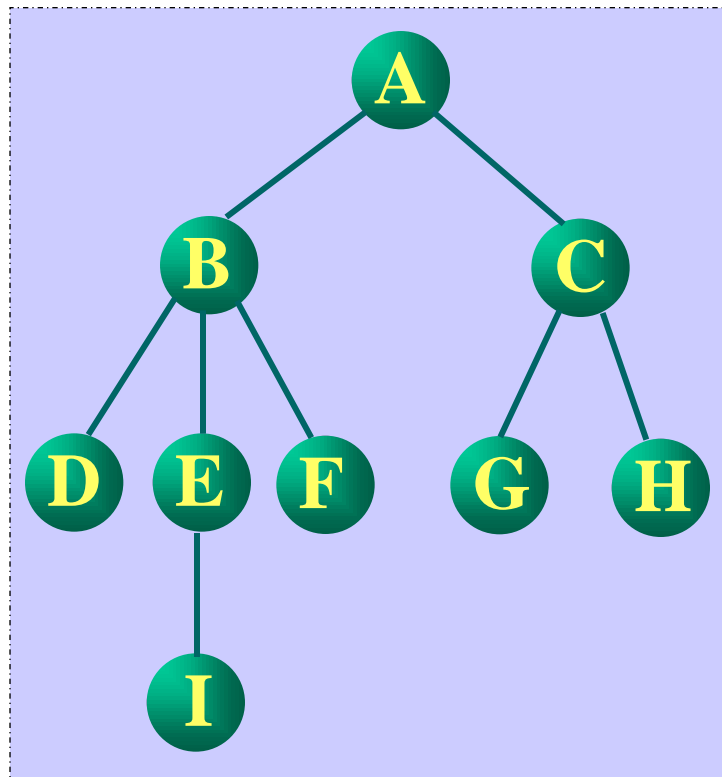
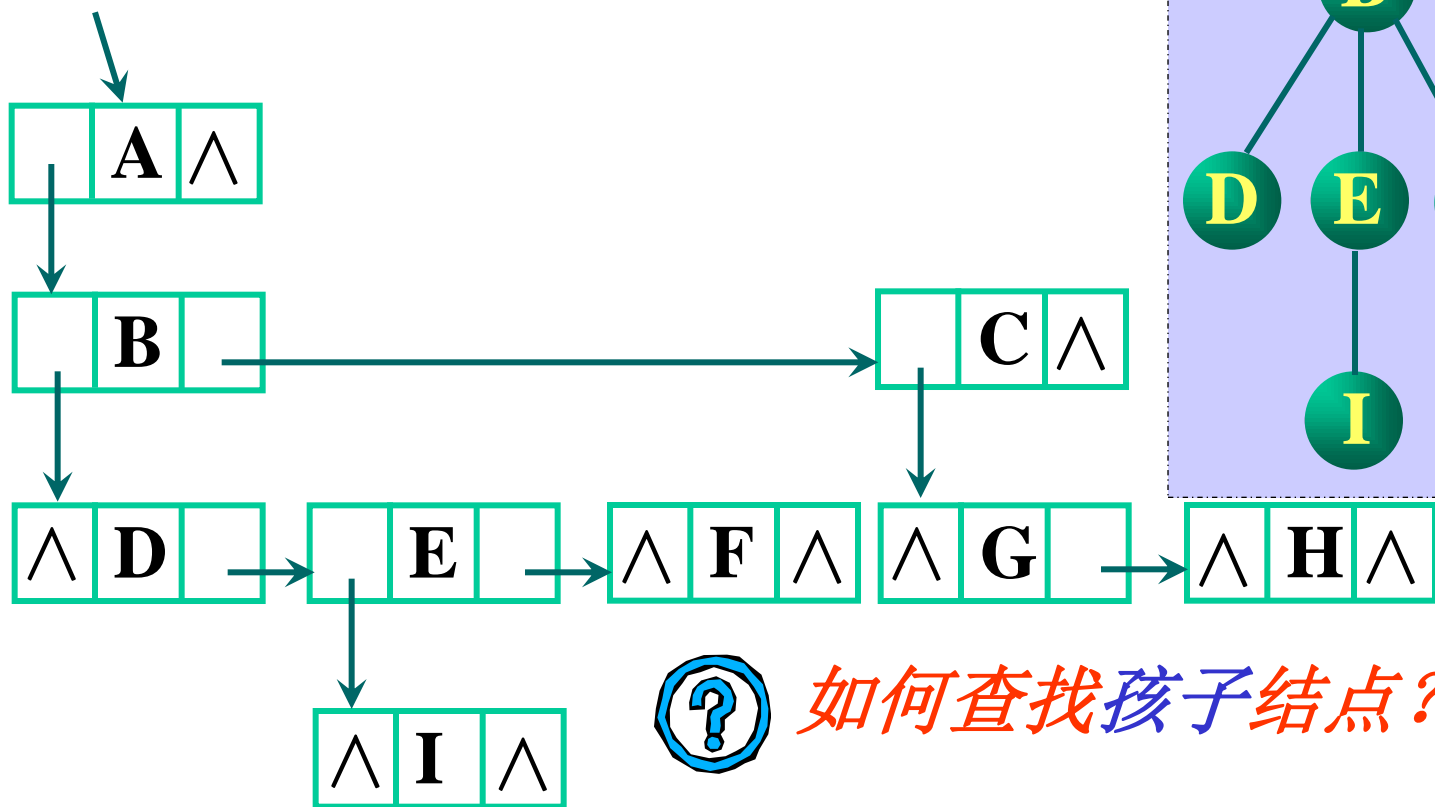
```
typedef char dataType;  
struct TNode  
{  
    dataType data;  
    TNode *firstchild, *rightsib;  
};
```

孩子兄弟表示法示例



如何查找兄弟结点？时间性能？

孩子兄弟表示法示例

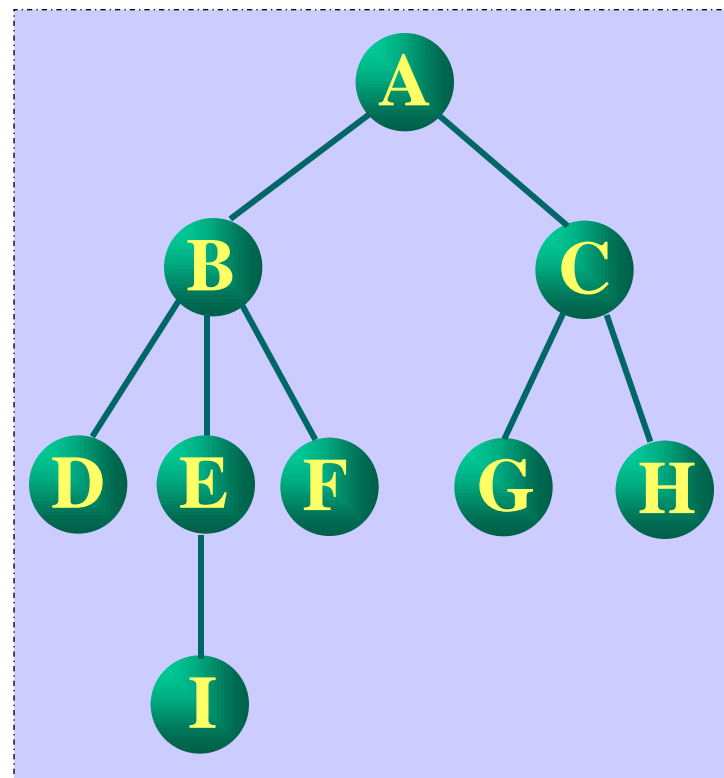
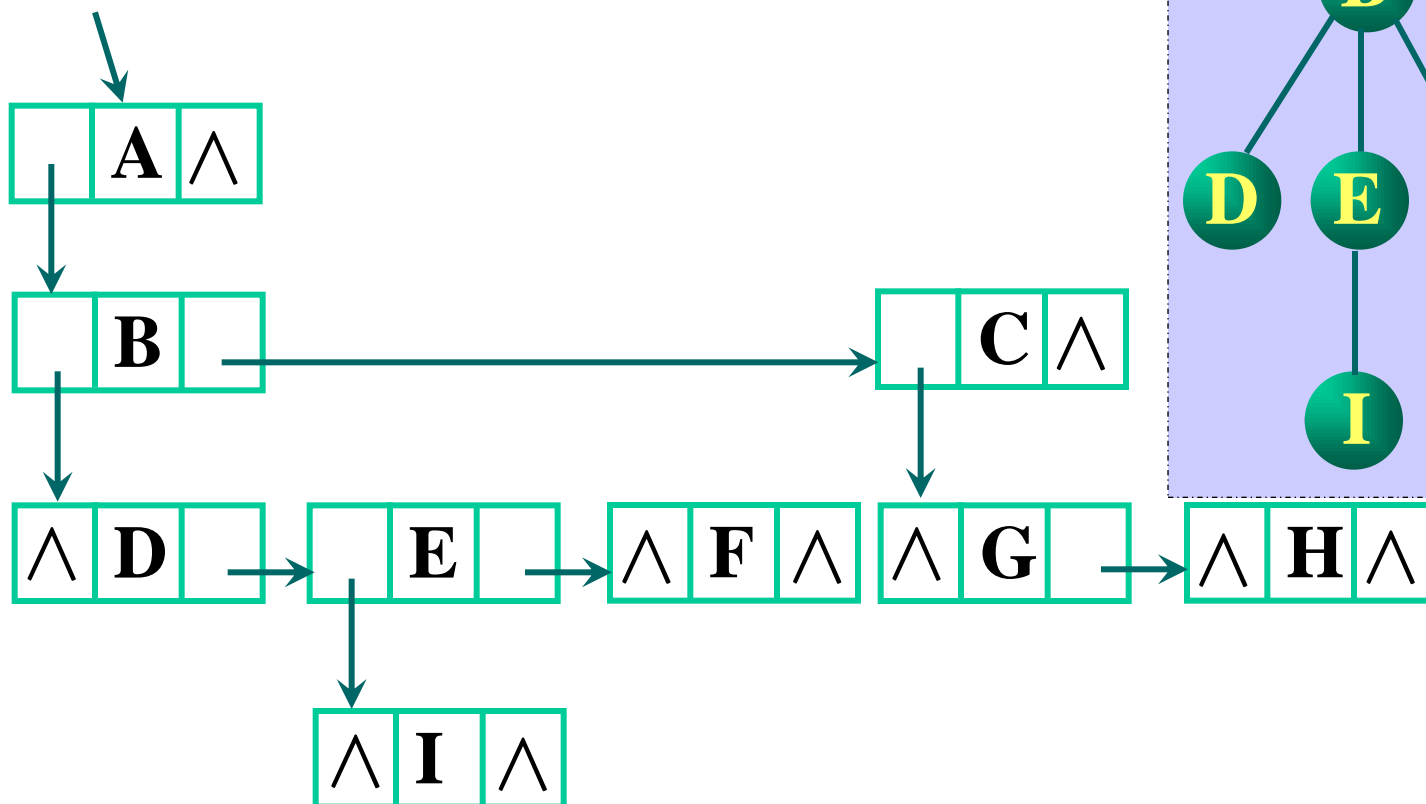


如何查找孩子结点？时间性能？

5.3 二叉树的逻辑结构



考虑一下树的孩子兄弟
表示形式?





研究二叉树的意义？

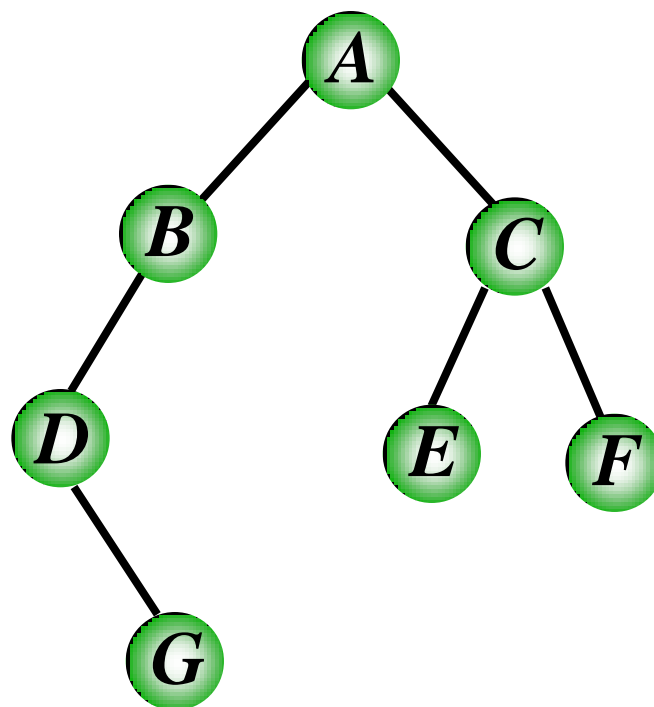
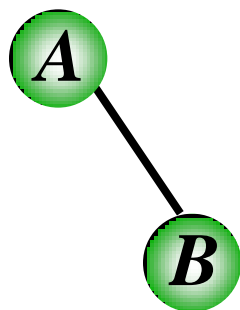
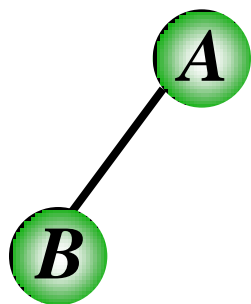
问题转化：将树转换为二叉树，从而利用二叉树解决树的有关问题。

一、二叉树的定义

二叉树是 n ($n \geq 0$) 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。

1、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。



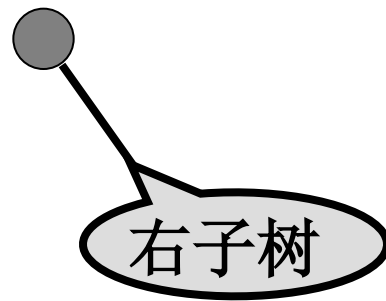
2、二叉树的基本形态

Φ

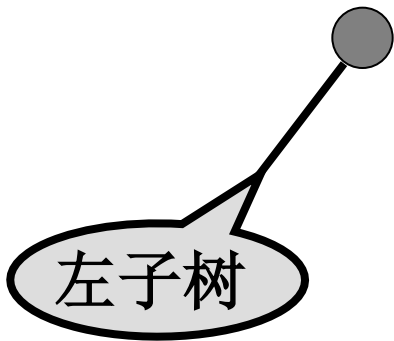
空二叉树



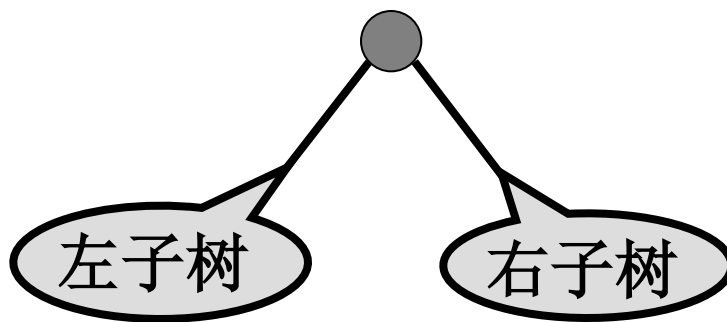
只有一个根结点



根结点只有右子树



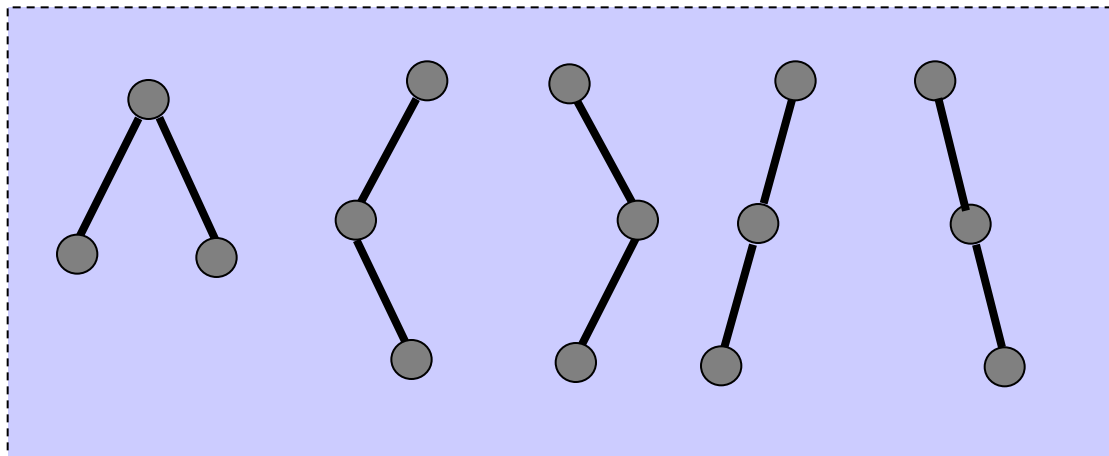
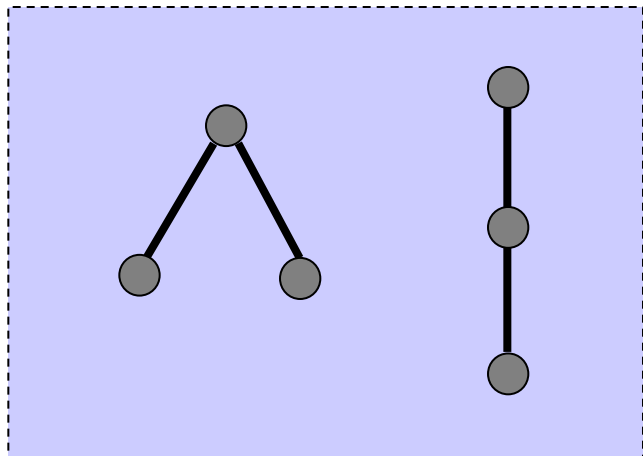
根结点只有左子树



根结点同时有左右子树

课堂练习

问题：具有3个结点的树和具有3个结点的二叉树的形态各有多少？

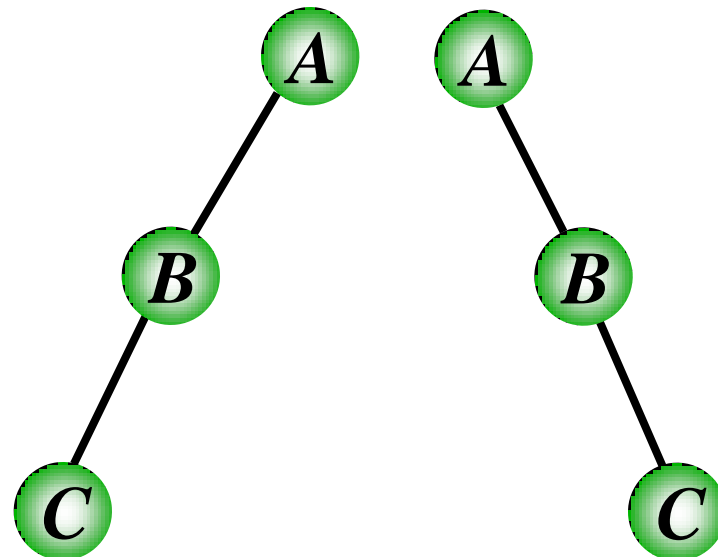


注意：二叉树和树是两种树结构。

3、特殊的二叉树

-斜树(oblique tree)

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**。

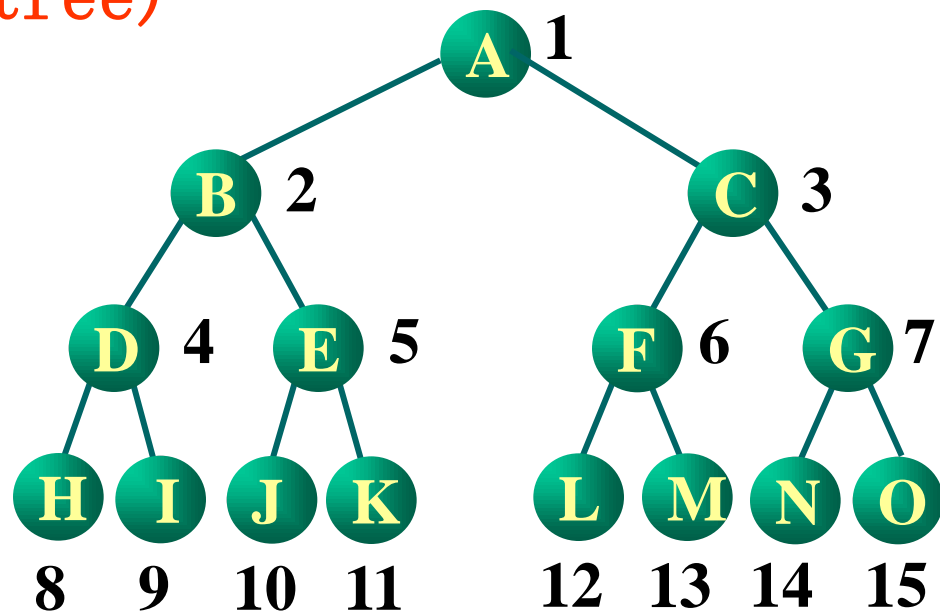


- 斜树的特点:**
1. 在斜树中, 每一层只有一个结点;
 2. 斜树的结点个数与其深度相同。

3、特殊的二叉树

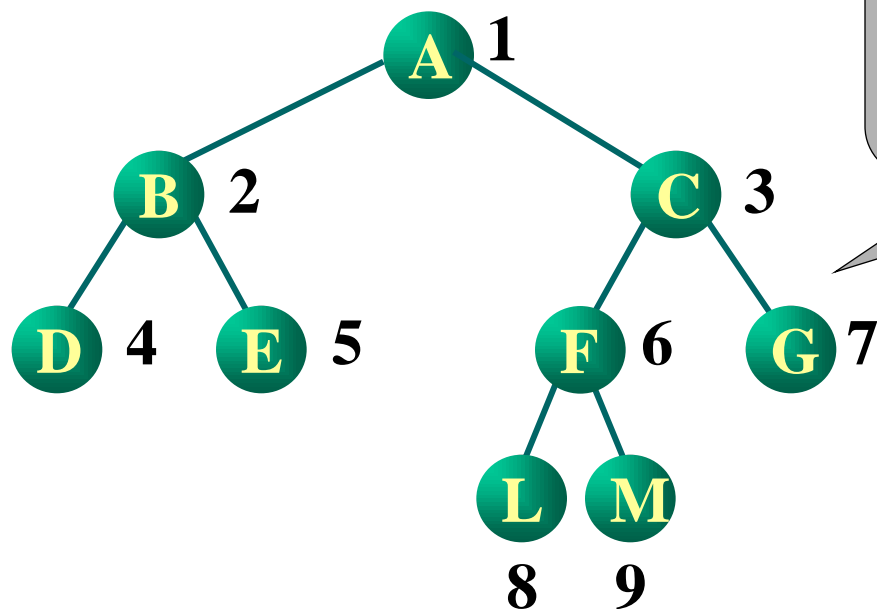
-满二叉树(full binary tree)

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。



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。

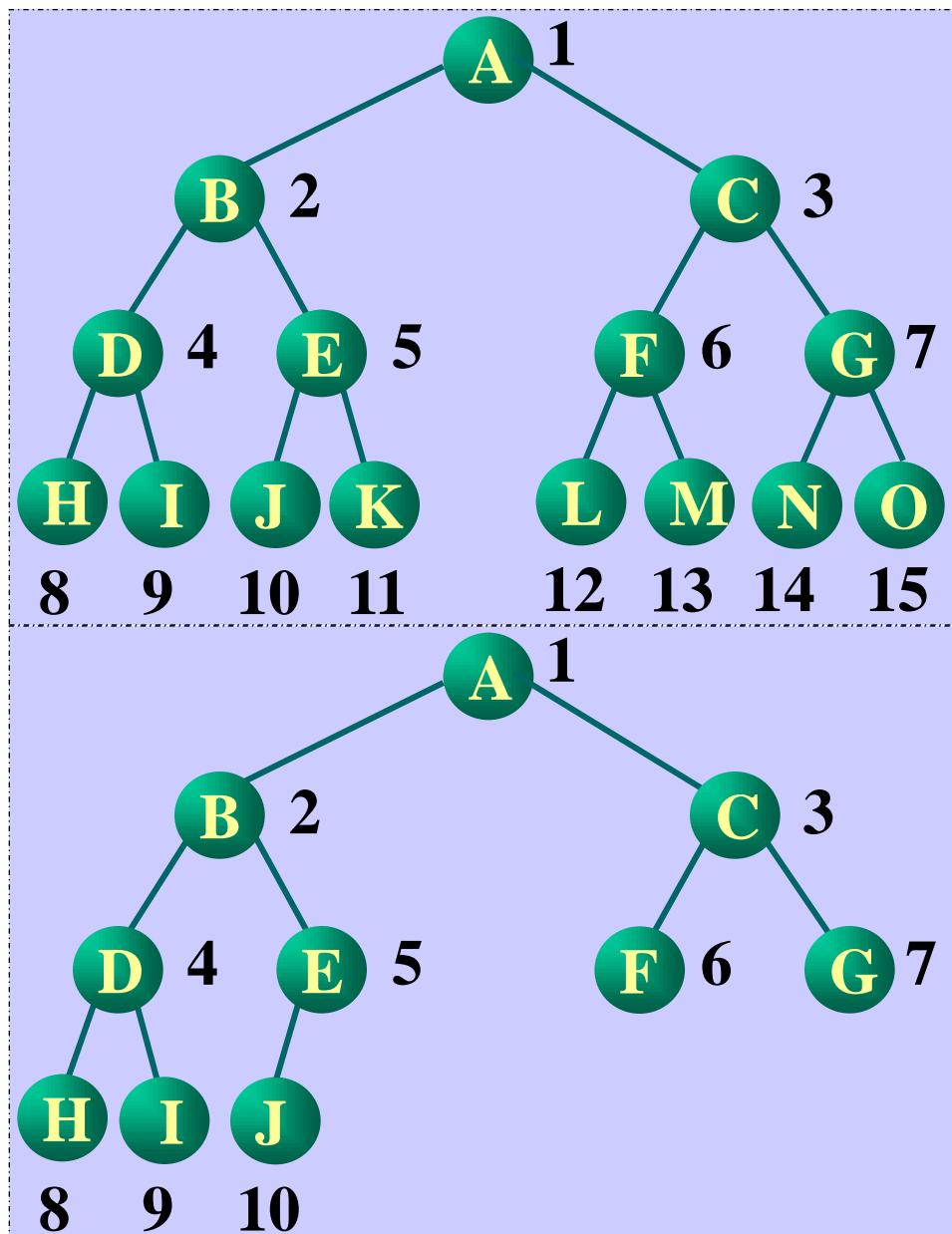
满二叉树在同样深度的二叉树中**结点**个数最多

满二叉树在同样深度的二叉树中**叶子结点**个数最多

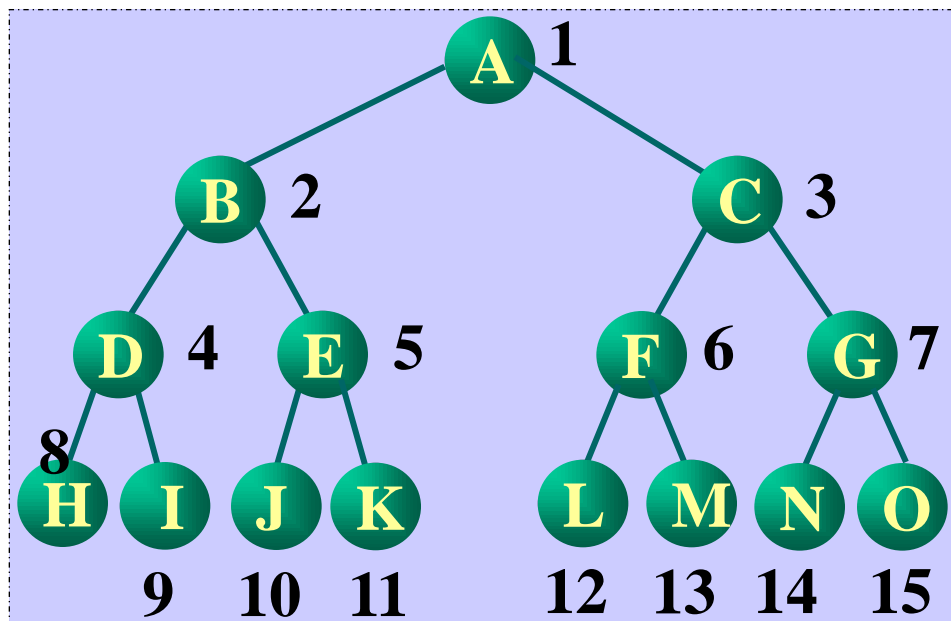
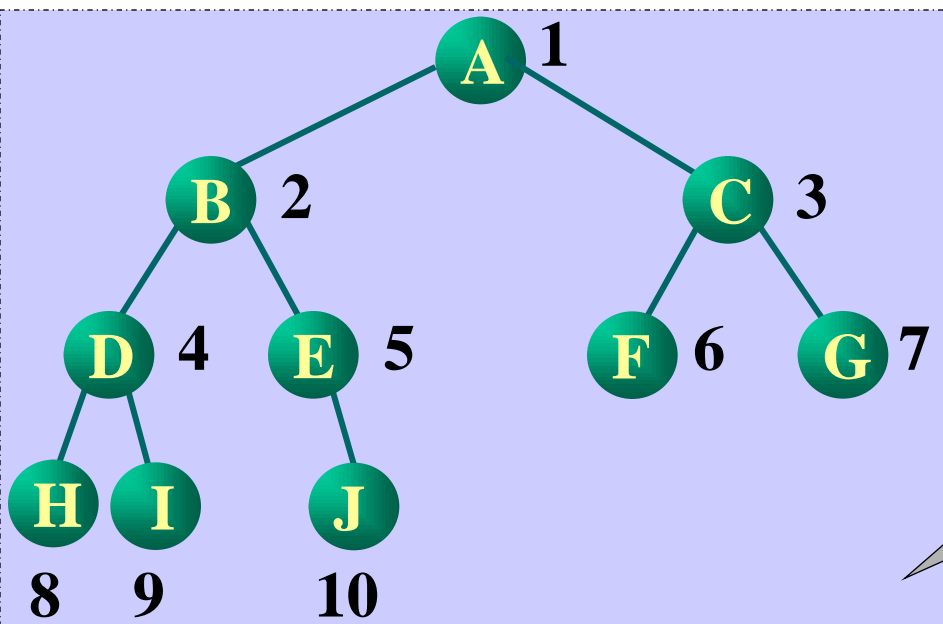
3、特殊的二叉树

-完全二叉树 (complete tree)

对一棵具有 n 个结点的二叉树按层序编号，如果编号为 i ($1 \leq i \leq n$) 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。



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。



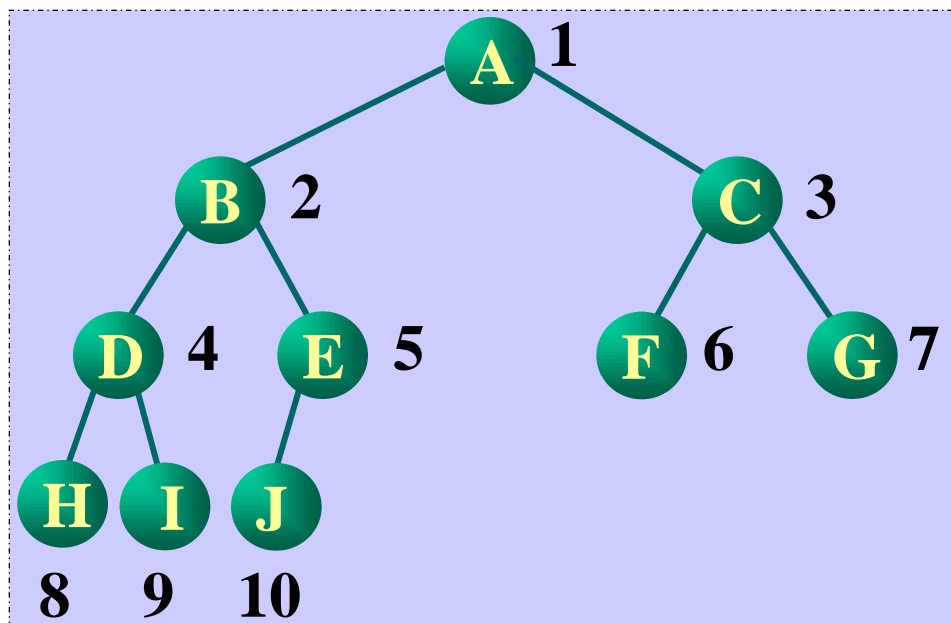
不是完全二叉树，结点10与满二叉树中的结点10不是同一个结点

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。



二、二叉树的基本性质

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

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,

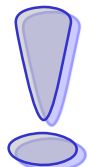
则*i*=*k*+1时, 因为第*k*+1层上的结点是第*k*层上结点的孩子, 而二叉树中每个结点最多有2个孩子, 故在第*k*+1层上最大结点个数为第*k*层上的最大结点数的二倍, 即 $2 \times 2^{k-1} = 2^k$ 。结论成立。

性质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

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

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。

性质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 为二叉树的结点总数， n_1 为二叉树中度为1的结点数，则有：

$$n = n_0 + n_1 + n_2 \dots\dots\dots (1)$$

在二叉树中，除了根结点外，其余结点都有唯一的一个分枝进入，由于这些分枝是由度为1和度为2的结点射出的，一个度为1的结点射出一个分枝，一个度为2的结点射出两个分枝，所以有：

$$n = n_1 + 2n_2 + 1 \dots\dots\dots (2)$$

由(1)(2)可以得到： $n_0 = n_2 + 1$ 。

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，所以有

$$n = n_0 + n_2$$

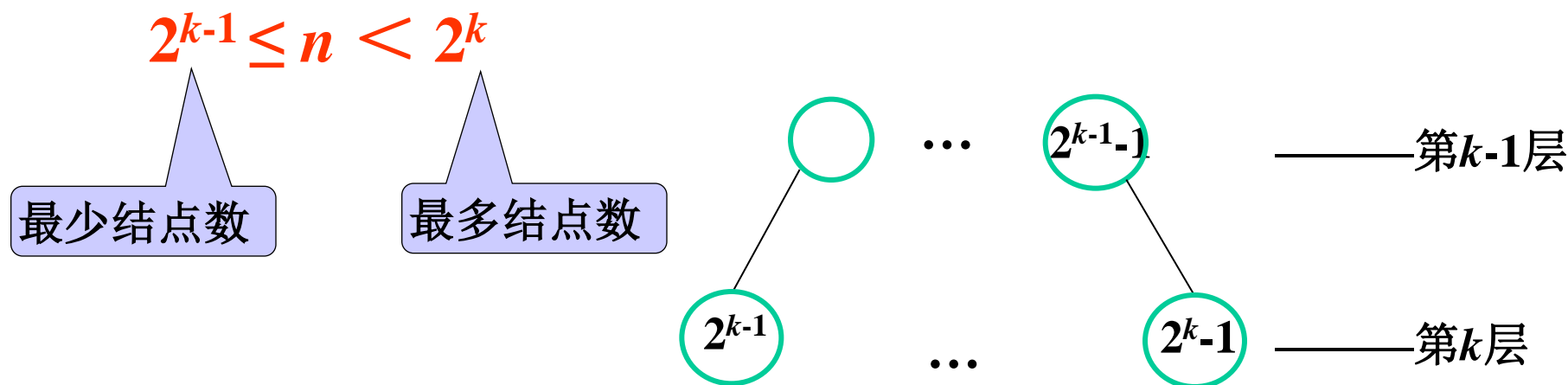
由性质3， $n_0 = n_2 + 1$

由此可以推出，叶子结点

$$n_0 = (n + 1) / 2$$

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

证明：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



证明（续）：

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

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

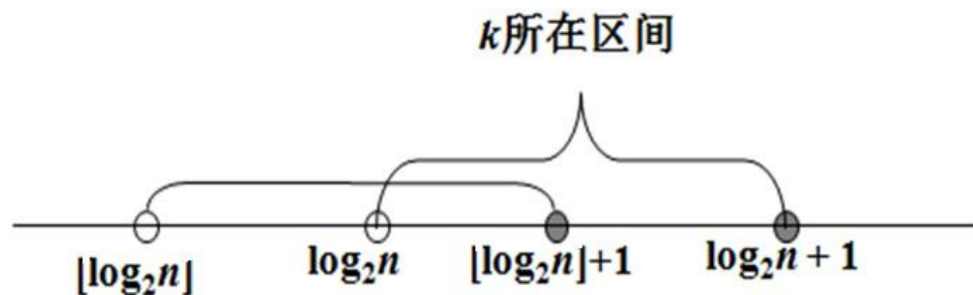
对不等式取对数，有：

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

即：

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

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



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

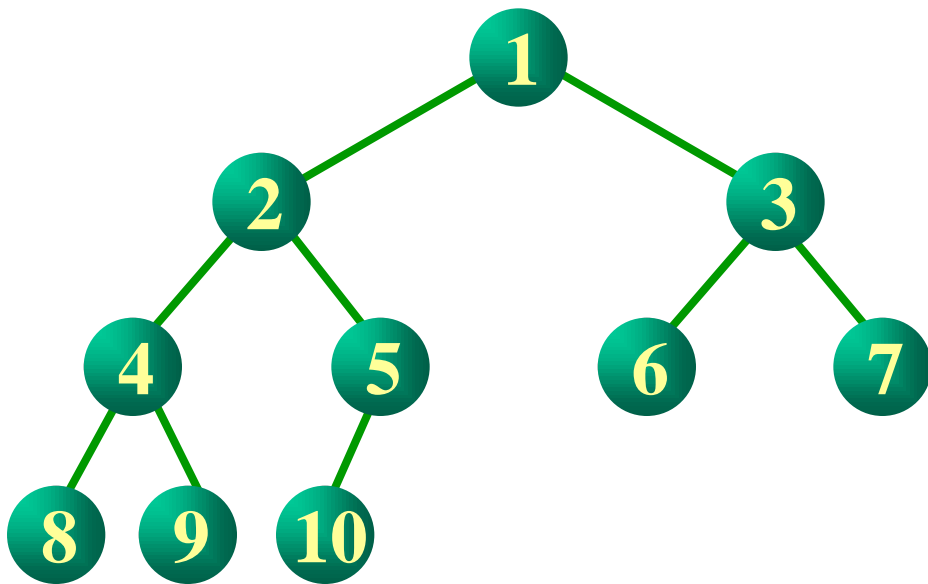
(1) 如果 $i > 1$ ，则结点 i 的双亲结点的序号为 $i/2$ ；如果 $i = 1$ ，则结点 i 是根结点，无双亲结点。

(2) 如果 $2i \leq n$ ，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 $2i + 1 \leq n$ ，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。

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

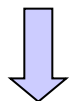
- 结点 i 的双亲结点为 $i/2$;
- 结点 i 的左孩子为 $2i$;
- 结点 i 的右孩子为 $2i+1$ 。



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——父子关系。

三、二叉树遍历

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。



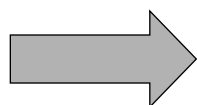
抽象操作，可以是对结点进行的各种处理，这里简化为输出结点的数据。

前序遍历
中序遍历
后序遍历
层序遍历

① **二叉树遍历操作的结果?** \Longrightarrow 非线性结构线性化

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。

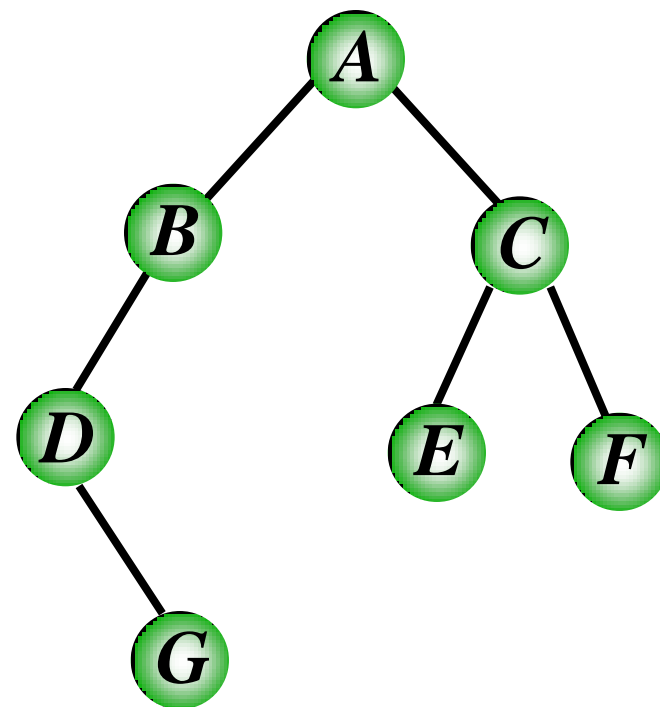
1、前序（根）遍历

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。

例如：前序遍历序列

A B D G C E F



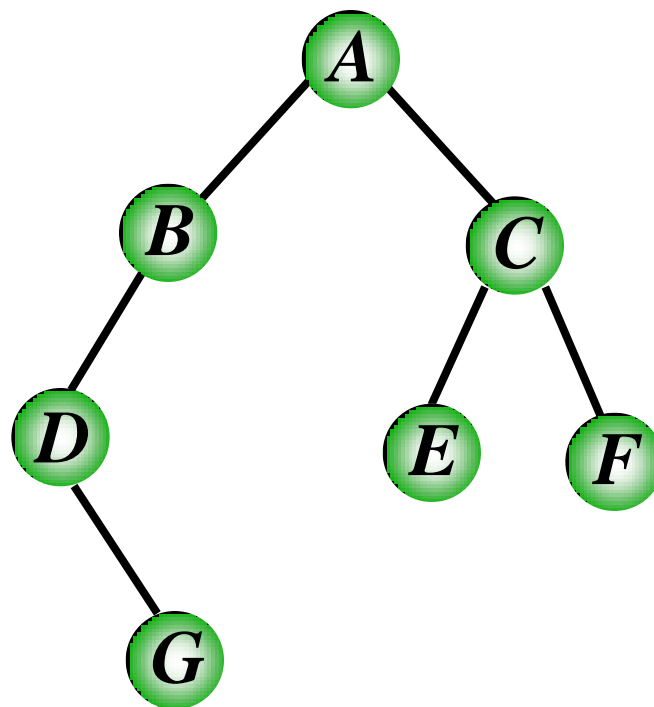
2、中序（根）遍历

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。

例如：中序遍历序列

D G B A E C F



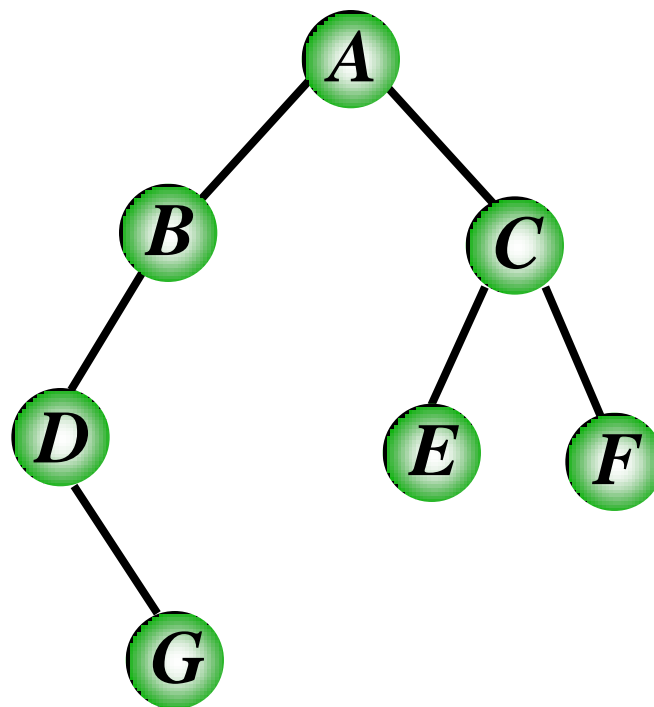
3、后序（根）遍历

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；

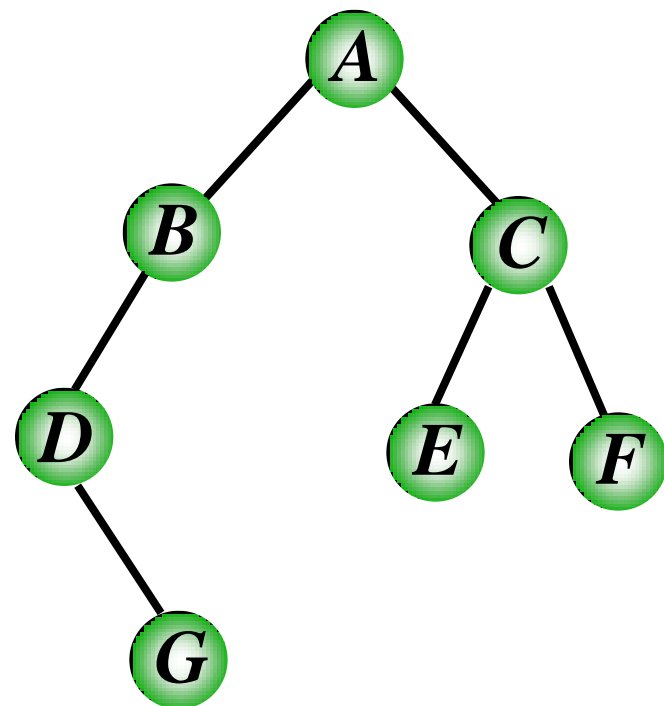
例如：后序遍历序列

G D B E F C A



4、层序遍历（广度优先策略）

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。



例如：层序遍历序列

A B C D E F G

课堂练习 (1)

前序遍历结果:

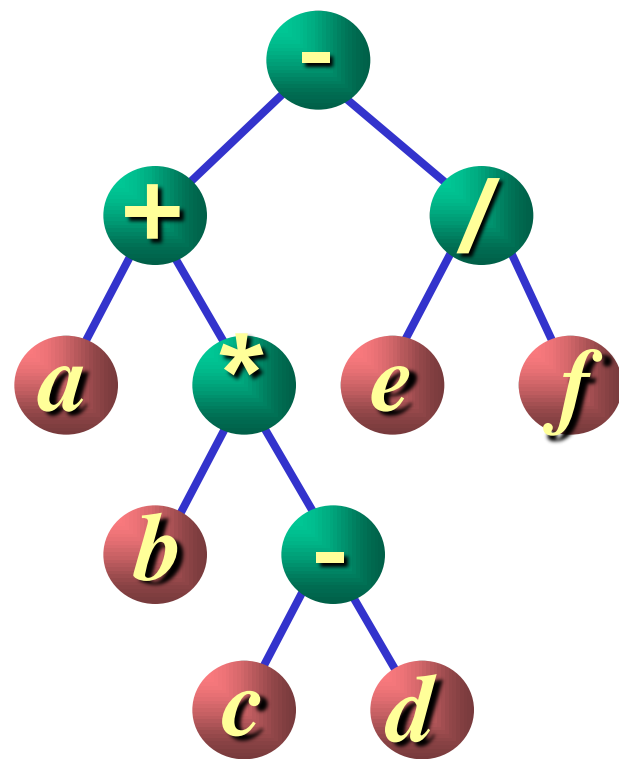
- + a * b - c d / e f

中序遍历结果:

a + b * c - d - e / f

后序遍历结果:

a b c d - * + e f / -

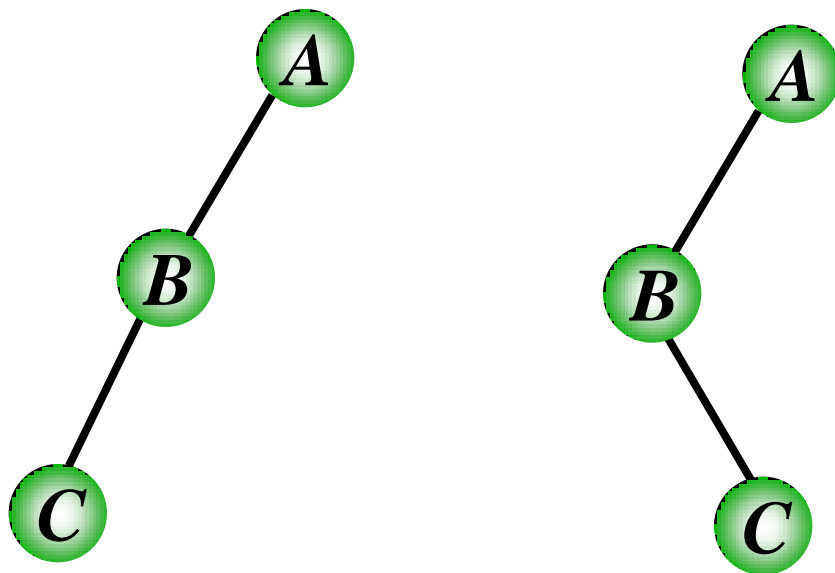


课堂练习 (2)



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：

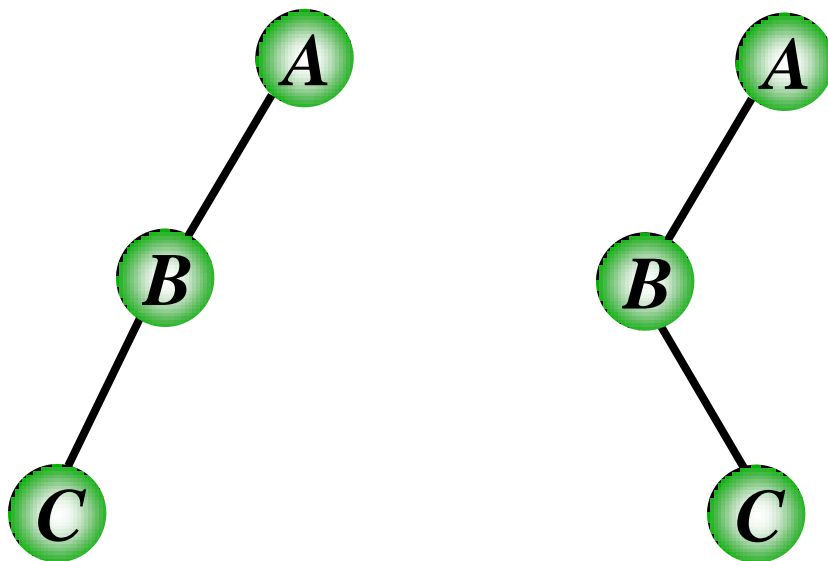


课堂练习 (3)



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：

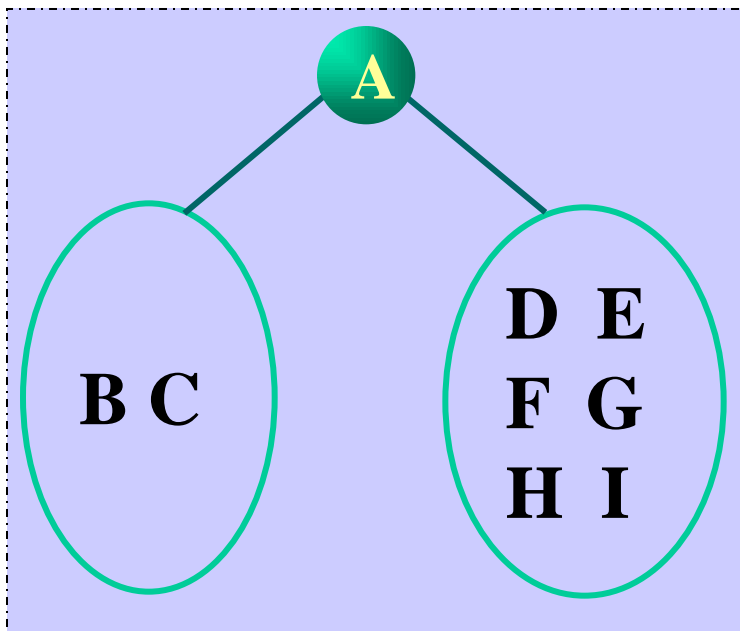


课堂练习 (4)



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？

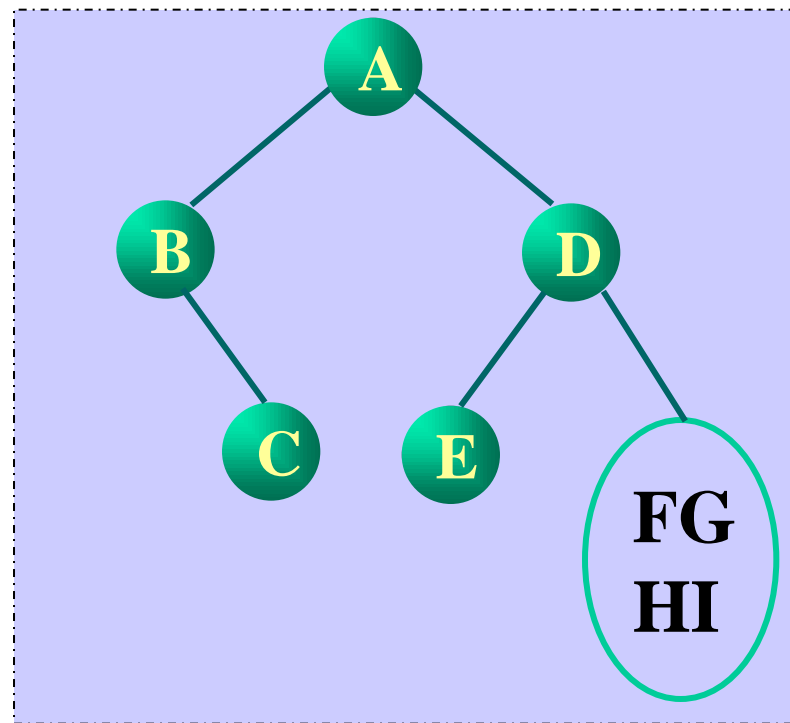
例如：已知一棵二叉树的前序遍历序列和中序遍历序列分别为**ABCDEFGHI** 和**BCAEDGHI**，如何构造该二叉树呢？

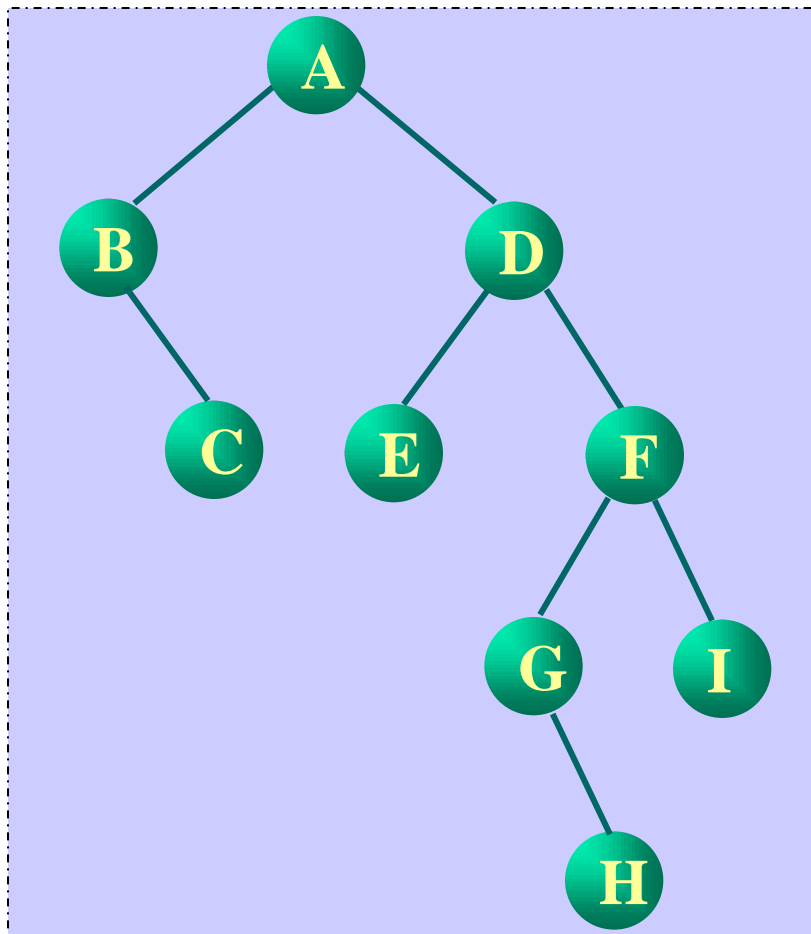


前序: A B C D E F G H I
 中序: B C A E D G H F I

前序: B C
 中序: B C

前序: D E F G H I
 中序: E D G H F I



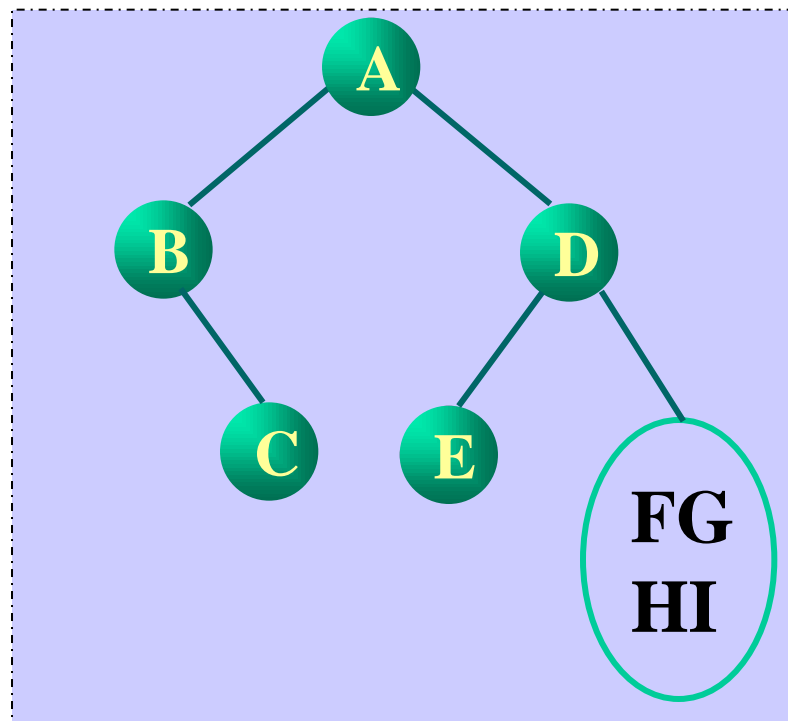


前序: **F** G H I

中序: G H **F** I

前序: **D** E F G H I

中序: E **D** G H F I



总结：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

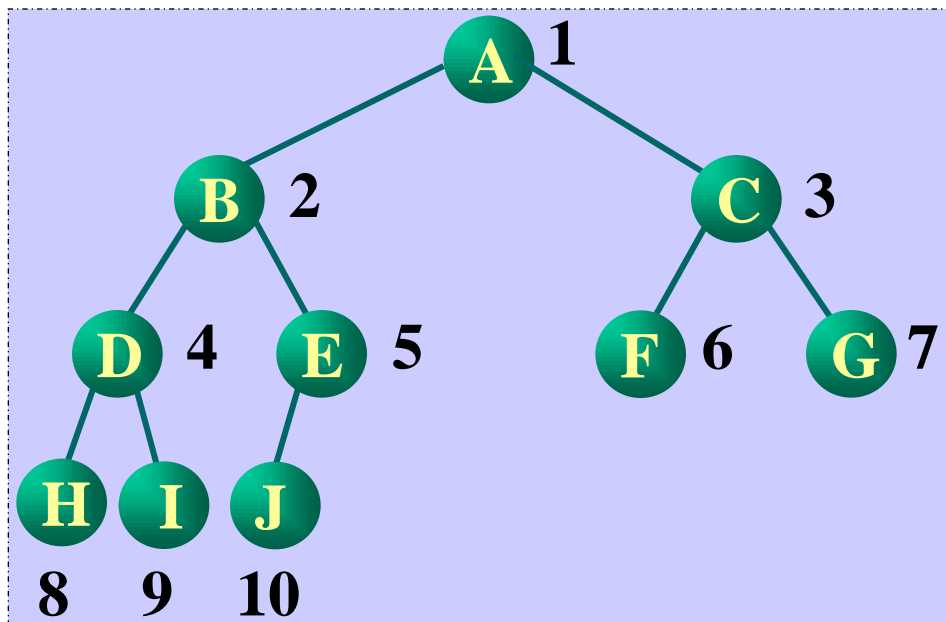
一、顺序存储结构

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。

① **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。

完全二叉树的顺序存储

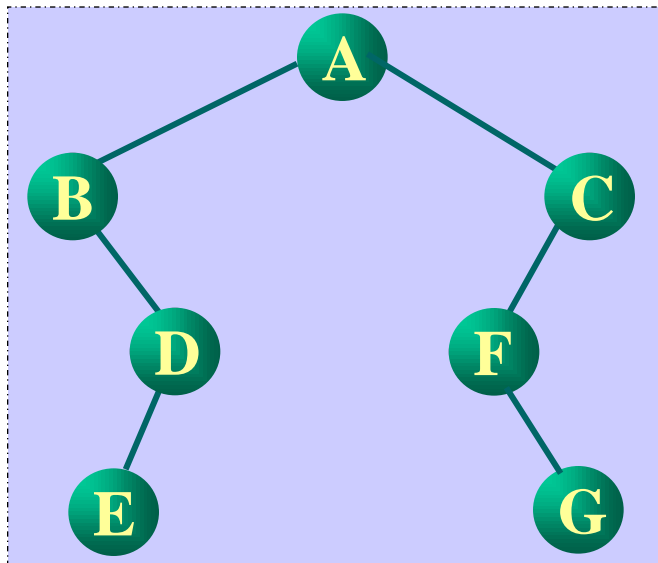


以编号
为下标

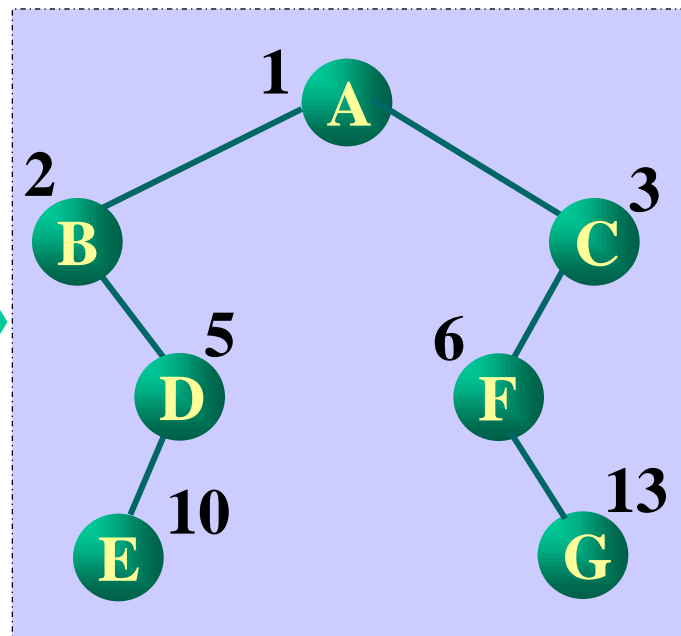
数组下标 1 2 3 4 5 6 7 8 9 10

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

二叉树的顺序存储



按照完全
二叉树编号



以编号
为下标

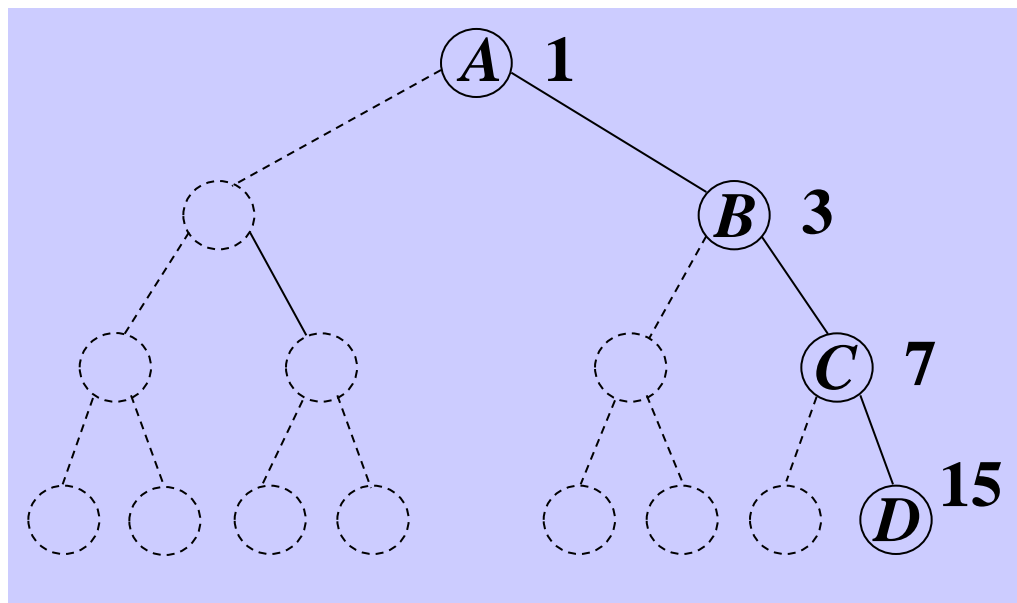
数组下标	1	2	3	4	5	6	7	8	9	10	11	12	13
	A	B	C	∧	D	E	∧	∧	∧	F	∧	∧	G



一棵斜树的顺序存储会怎样呢？

深度为 k 的右斜树， k 个结点需分配 $2^k - 1$ 个存储单元。

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。



二叉树的顺序存储结构一般仅存储**完全**二叉树

二、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：

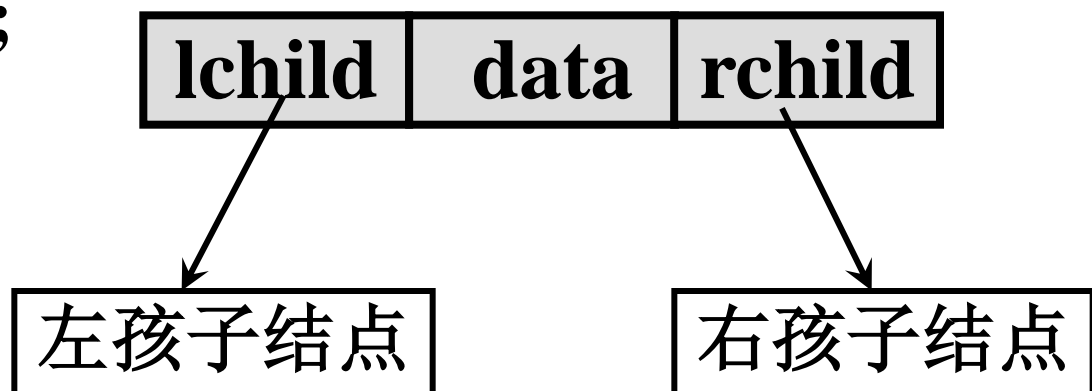


其中，**data**：数据域，存放该结点的数据信息；

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。


```
typedef char dataType;
struct BiNode
{
    dataType data;
    BiNode *lchild, *rchild;
};
```



1、二叉链表存储结构的类声明

```
class BiTree
{
public:
    BiTree() {root=create(root);}; //初始化一棵二叉树，其前序序列由键盘输入
    ~BiTree() {release(root);}; //释放二叉链表中各结点的存储空间
    BiNode *Getroot(); //获得指向根结点的指针
    void PreOrder() {PreOrder(root);}; //前序遍历二叉树
    void InOrder() {InOrder(root);}; //中序遍历二叉树
    void PostOrder() {PostOrder(root);}; //后序遍历二叉树
    void LevelOrder(); //层序遍历二叉树
private:
    BiNode *root; //指向根结点的头指针
    BiNode *Creat(BiNode *bt); //由构造函数调用
    void Release(BiNode *bt); //由析构函数调用
    void PreOrder(BiNode *bt);
    void InOrder(BiNode *bt);
    void PostOrder(BiNode *bt);
};
```

(1)构建函数： BiTree()

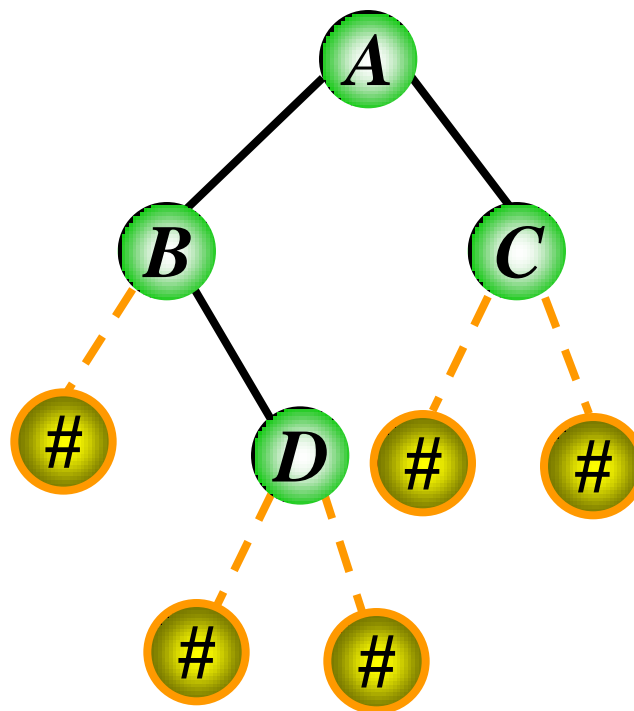
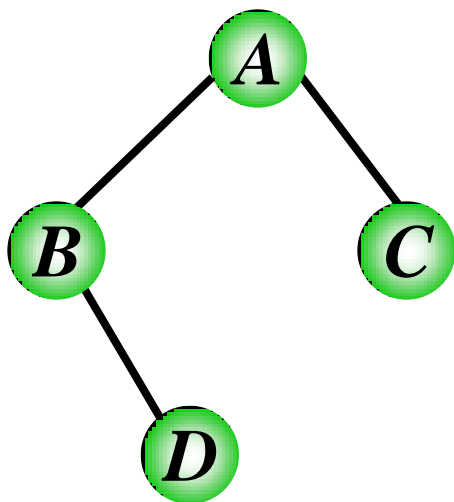
遍历是二叉树各种操作的基础，可以在遍历的过程中进行各种操作，例如建立一棵二叉树。

① 如何由一种遍历序列生成该二叉树(存储结构)?

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。

② 为什么如此处理?

唯一性



扩展二叉树的前序遍历序列: **$A B \# D \# \# C \# \#$**

二叉树的建立过程:

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**递归地**建立它的左子树和右子树。

建立二叉树的递归算法:

```
BiNode *BiTree ::Creat(BiNode *bt)
{
    char ch;
    cin>>ch;
    if (ch=='# ') bt=NULL;
    else {
        bt=new BiNode;           //生成一个结点
        bt->data=ch;
        bt->lchild=Creat(bt->lchild); //递归建立左子树
        bt->rchild=Creat(bt->rchild); //递归建立右子树
    }
    return bt;
}
```

(2)析构函数: ~BiTree()

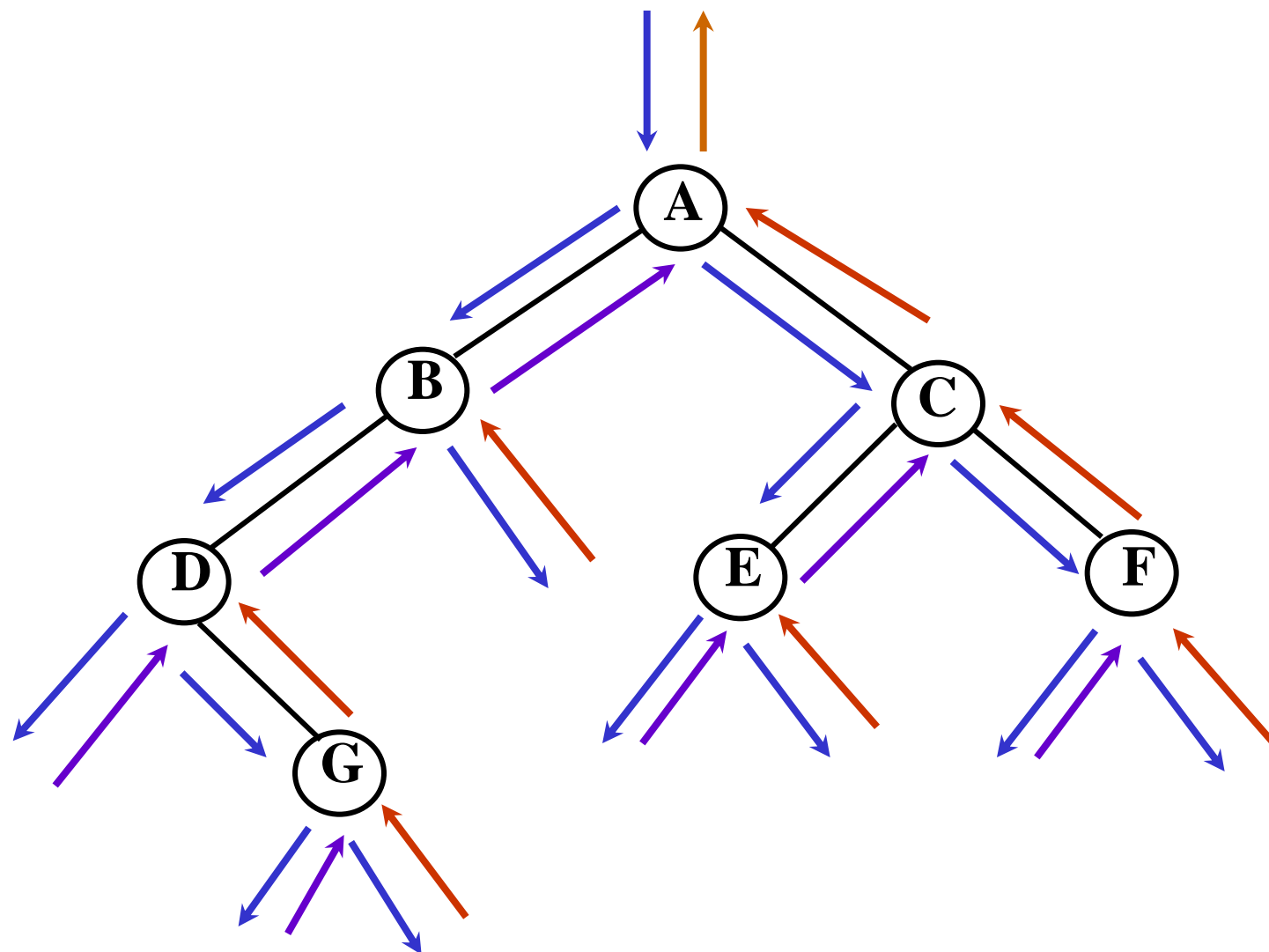
```
void BiTree::Release(BiNode* bt) { //释放二叉树占用的空间  
    if (bt != NULL){  
        Release(bt->lchild);    //释放左子树  
        Release(bt->rchild);    //释放右子树  
        delete bt;  
    }  
}
```

(3)前序遍历——递归算法

```
void BiTree::PreOrder(BiNode *bt) { //私有成员函数
    if (bt == NULL) return;
    else {
        cout<<bt->data;
        PreOrder(bt->lchild);
        PreOrder(bt->rchild);
    }
}

void BiTree::PreOrder() { //公有无参函数
    PreOrder(root);
}
```


前序遍历算法的执行轨迹



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？**

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。

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：

(1) 若 $root \neq \text{NULL}$ ，**则表明？**

当 $root$ 不空时循环

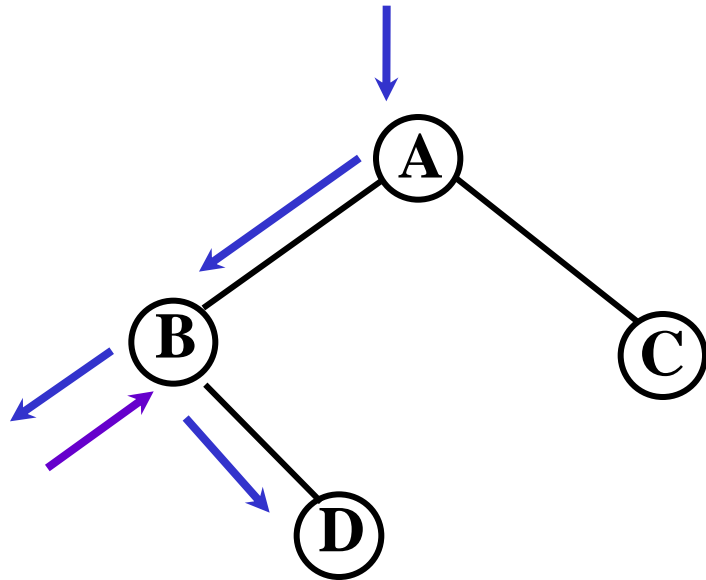
- 输出 $root \rightarrow data$;
- 将指针 $root$ 的值保存到栈中;
- 继续遍历 $root$ 的左子树

(2) 若 $root = \text{NULL}$ ，**则表明？**

如果栈 s 不空，则

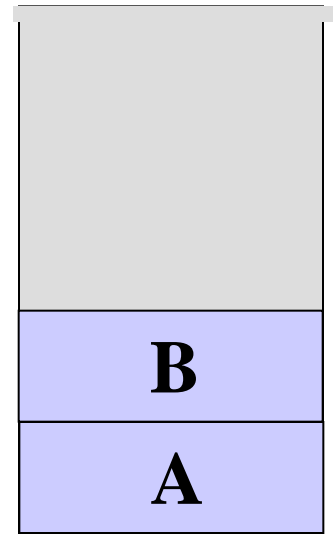
- 将栈顶元素弹出至 $root$;
- 准备遍历 $root$ 的右子树;

前序遍历的**非递归**实现

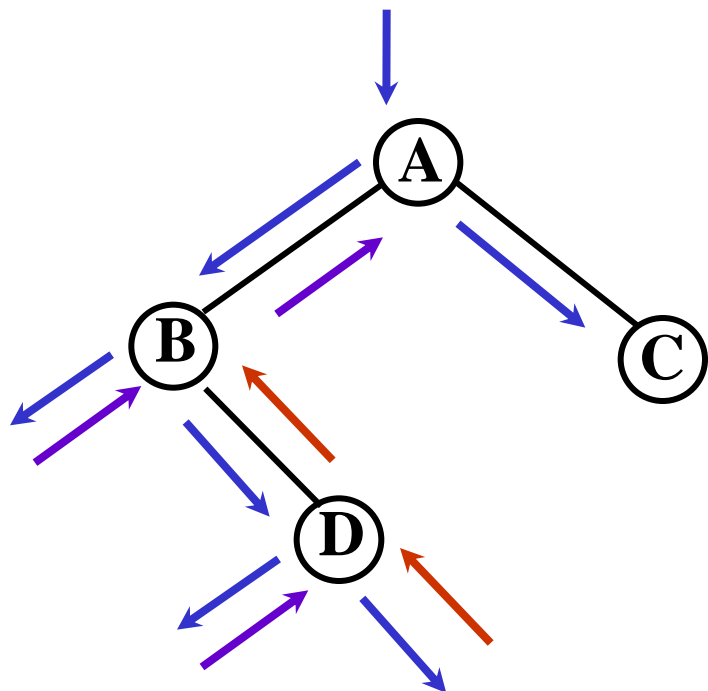


访问结点序列: **A B D**

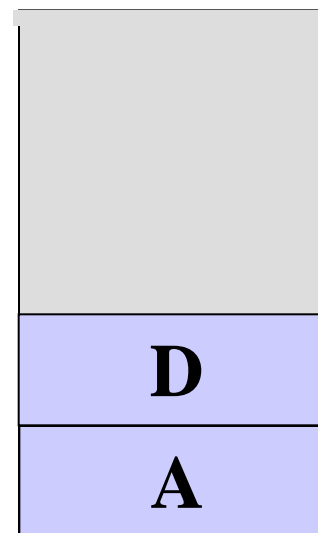
栈S内容:



前序遍历的非递归实现

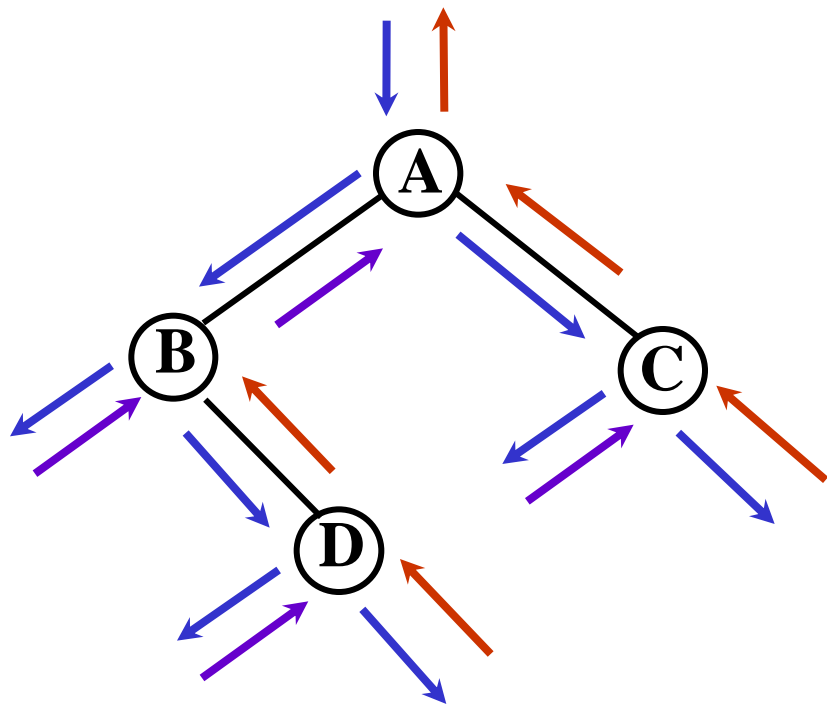


栈S内容:



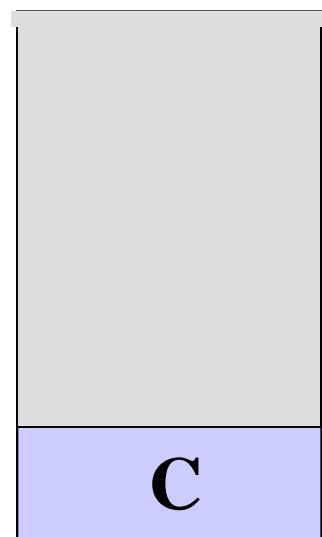
访问结点序列: **A B D**

前序遍历的非递归实现



访问结点序列: **A B D C**

栈S内容:



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（伪代码）

```
void BiTree::PreOrder(BiNode *bt) {  
    seqStack s;    //采用顺序栈，并假定不会发生上溢  
    while (bt!=NULL || !s.empty()) {  
        while (bt!= NULL) {  
            cout<<bt->data;  
            s.push(bt);  
            bt=bt->lchild;  
        }  
        if (!s.empty()) {  
            bt=s.pop();  
            bt=bt->rchild;  
        }  
    }  
}
```

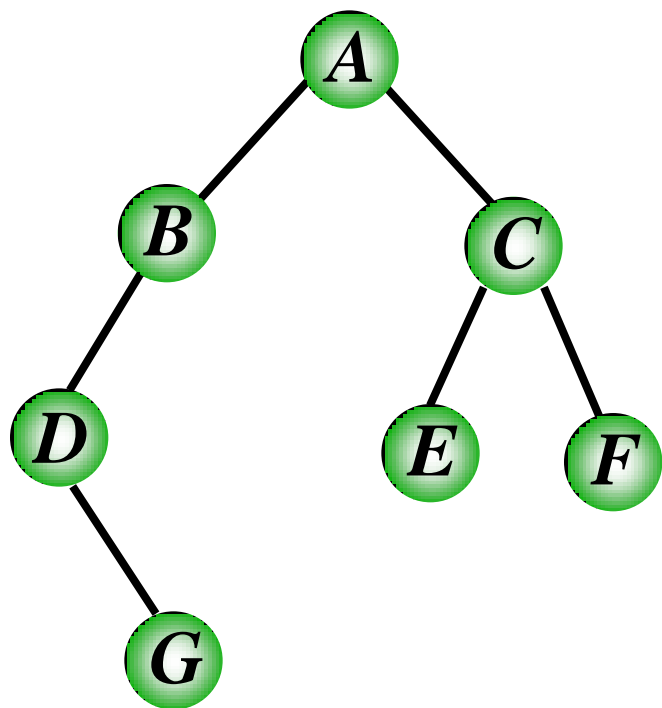
(4)中序遍历——递归算法

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


(5)后序遍历——递归算法

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

(6)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

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;

层序遍历二叉树

```
void BiTree::LevelOrder(BiNode *root) { //层序遍历二叉树
    if (root==NULL) return;
    LinkQueue *Q;
    Q.Enqueue(root);
    while (!Q.Empty()){
        BiNode *p=Q.DeQueue();
        cout<<p->data<<" ";
        if (p->lchild != NULL)    Q.Enqueue(p->lchild);
        if (p->rchild != NULL)    Q.Enqueue(p->rchild);
    }
}
```

二叉树的其他操作算法

1-打印二叉树中的叶子结点。

```
void BiTree::Print_leafnode(BiNode *bt)
{
    if (bt==NULL) return ;
    if (!bt->lchild && !bt->rchild)
        cout<<root->data;
    Print_leafnode(bt->lchild);
    Print_leafnode(bt->rchild);
}
```

```
void BiTree::Print_leafnode(){
    Print_leafnode(root);
}
```

二叉树的其他操作算法

2-计算二叉树的结点个数:

```
int Bitree::Count(BiNode *bt){  
    int left,right;  
    if (bt==NULL) return 0;  
    left=Count(bt->lchild);  
    right=Count(bt->rchild);  
    return 1+left+right;  
}
```

```
int BiTree::Count(){  
    return Count(root);  
}
```

```
int Bitree::Count(BiNode *bt){  
    if (bt==NULL) return 0;  
    return Count(bt->lchild)+Count(bt->rchild)+1;  
}
```

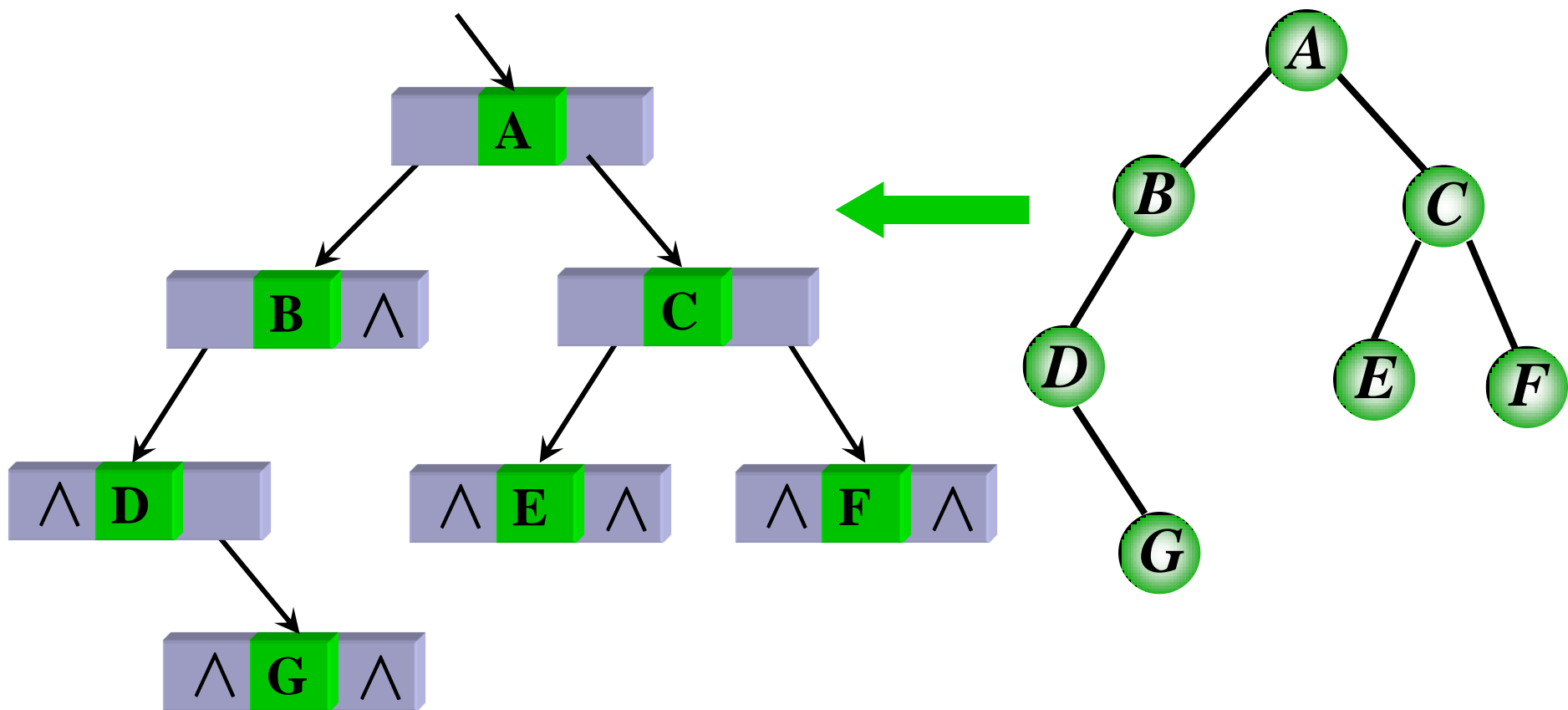
二叉树的其他操作算法

3-计算二叉树的高度:

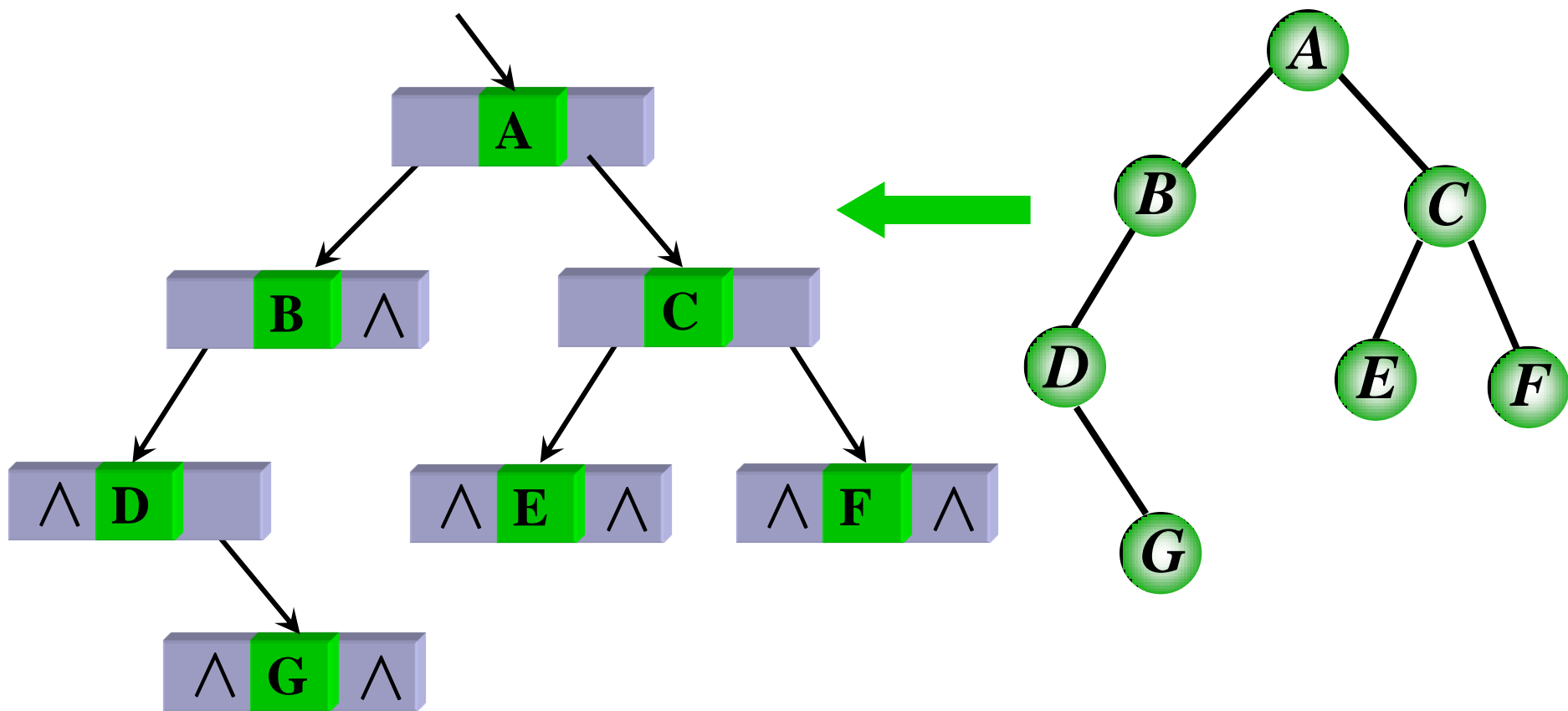
```
int Bitree::Height(BiNode *bt)
{
    int left,right;
    if (bt==NULL) return 0;
    left=Height(bt->lchild);
    right=Height(bt->rchild);
    if(left>right) return left+1; else return right+1;
}
```

```
int BiTree::Height(){
    return Height(root);
}
```

三、线索二叉树



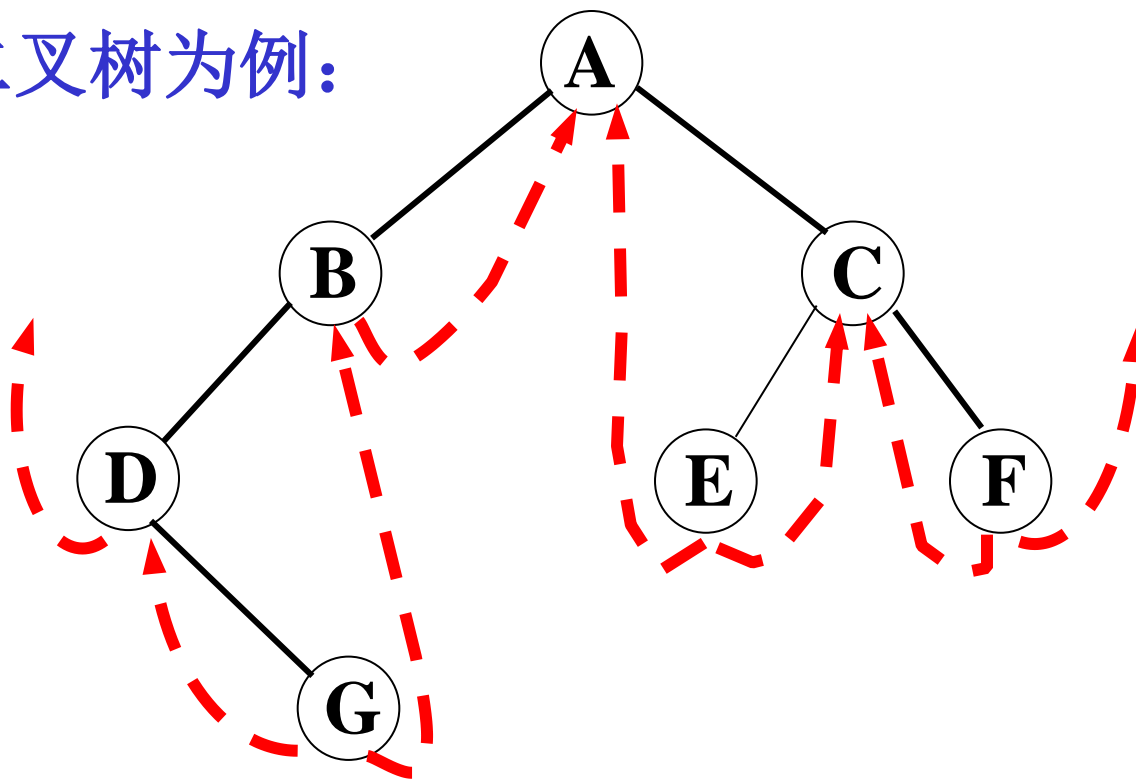
具有 n 个结点的二叉链表中，有多少个空指针？



具有 n 个结点的二叉链表中，有 $n+1$ 个空指针。
(为什么呢?)

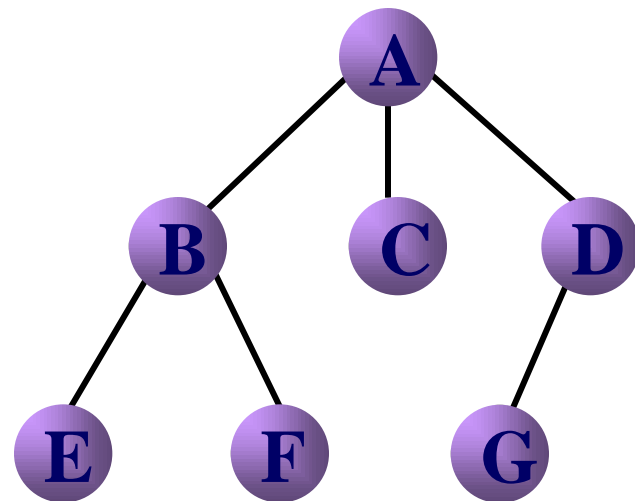
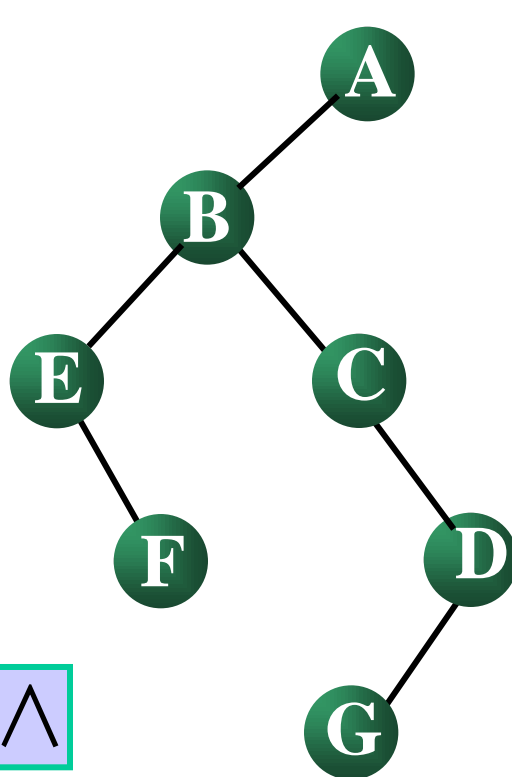
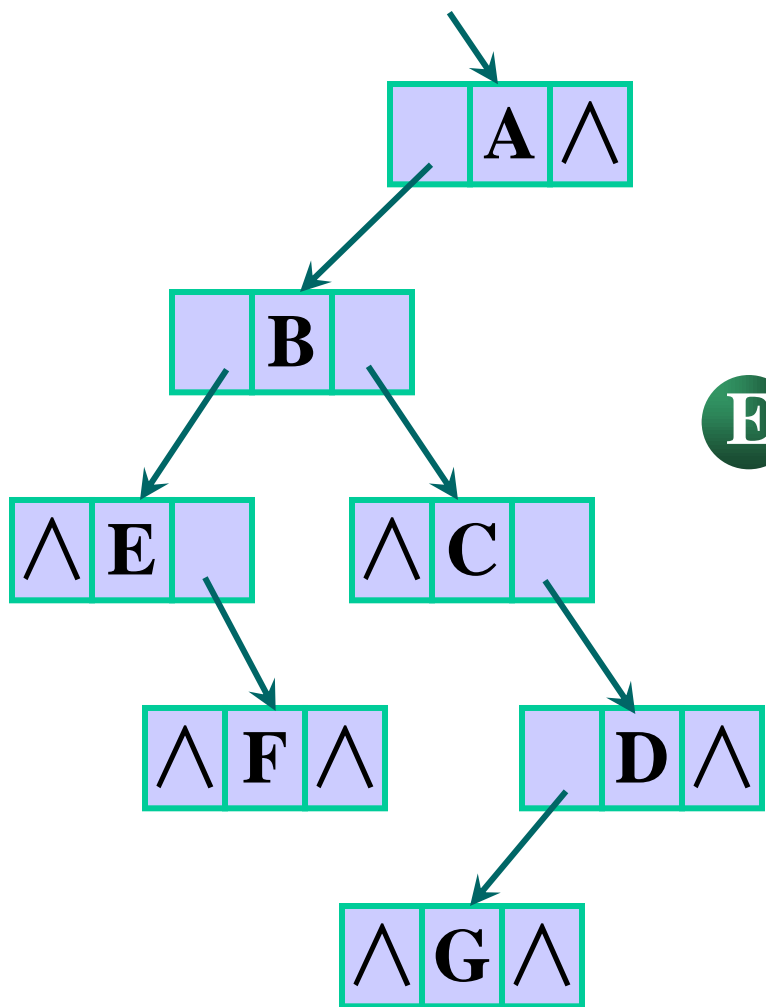
三、线索二叉树

仅以中序线索二叉树为例：



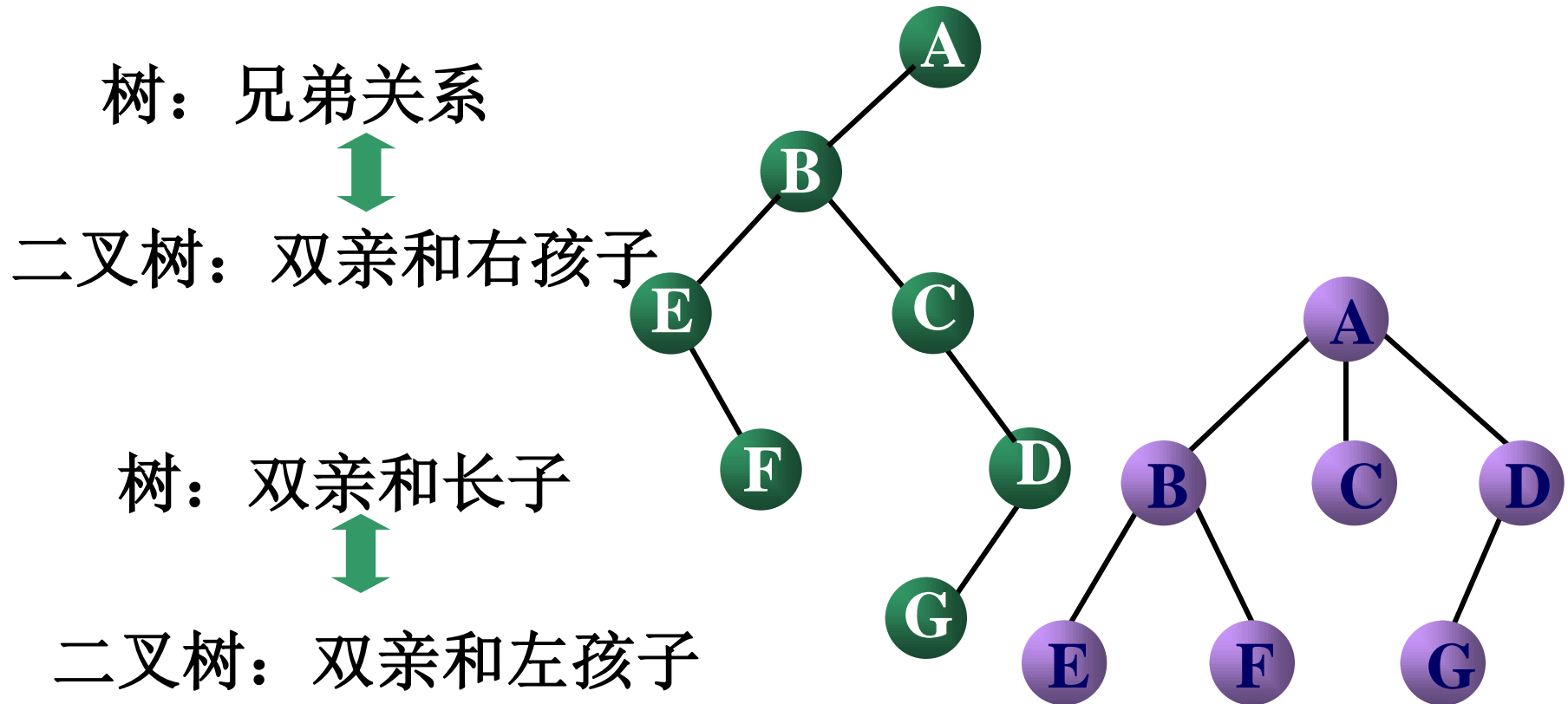
中序序列： *D G B A E C F*

5.5 树、森林与二叉树的转换

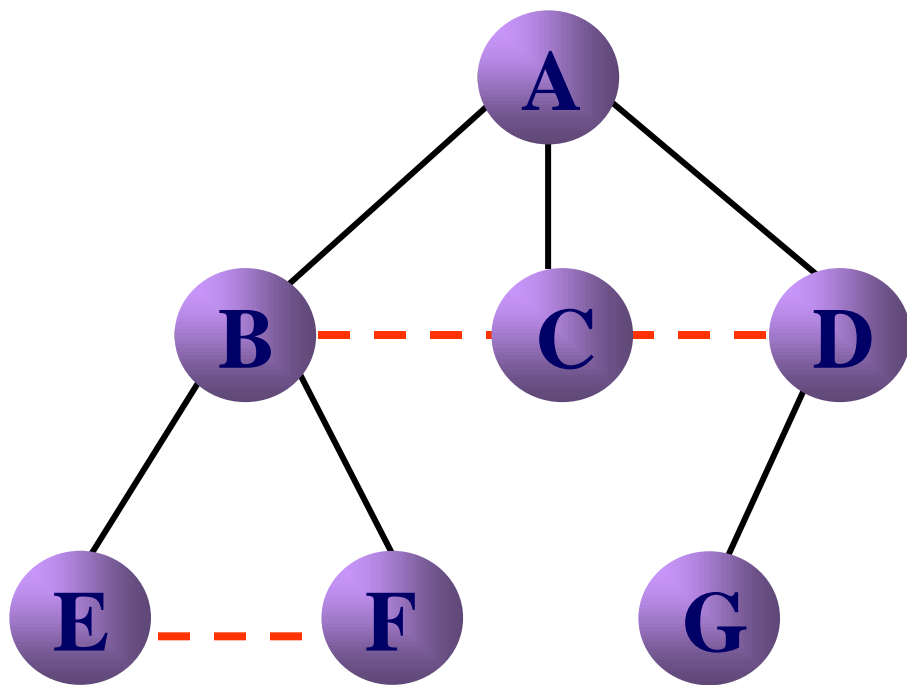


树和二叉树之间具有对应关系

树和二叉树之间的对应关系

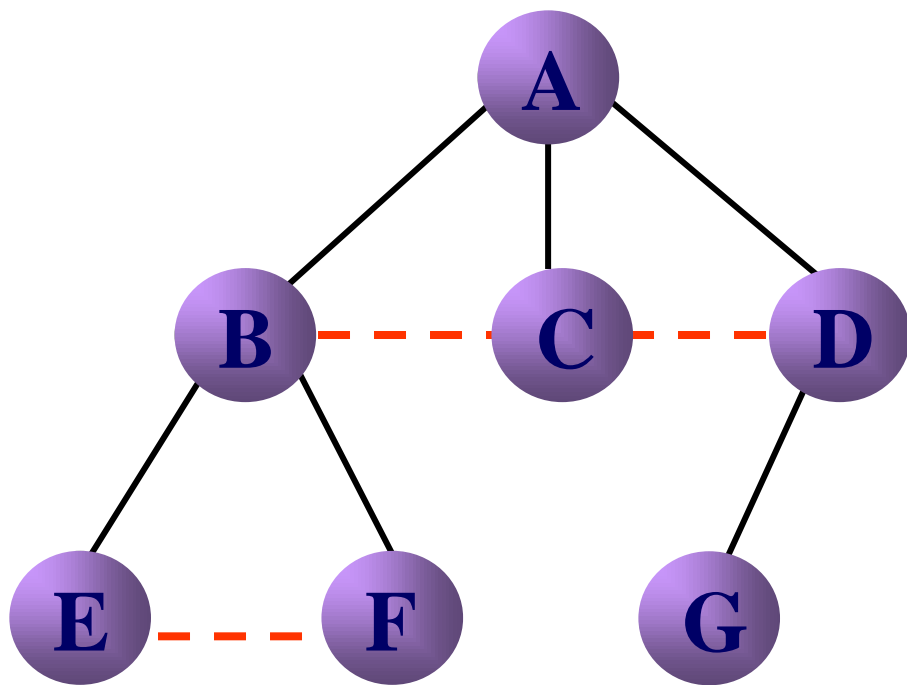


树和二叉树之间的对应关系



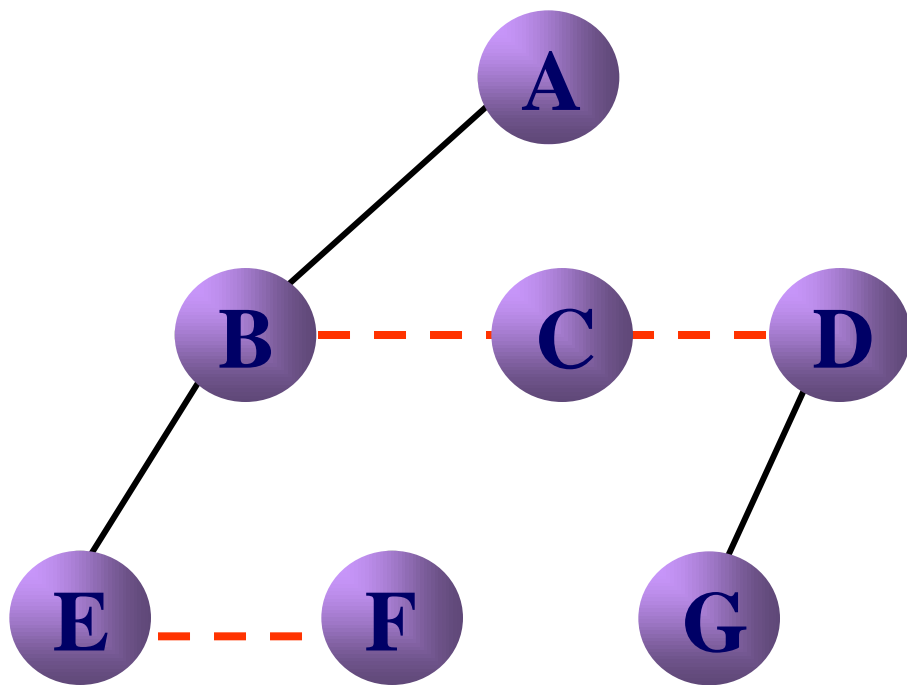
1. 兄弟加线.

树和二叉树之间的对应关系



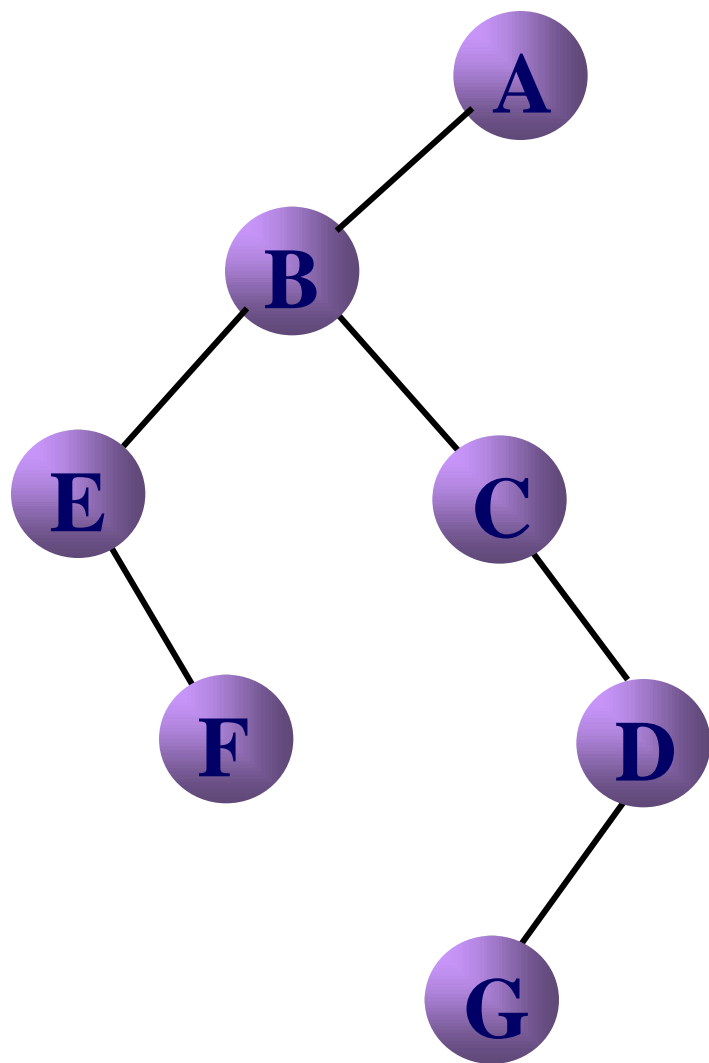
- 1.兄弟加线.
- 2.保留双亲与第一孩子连线,删去与其他孩子的连线.

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



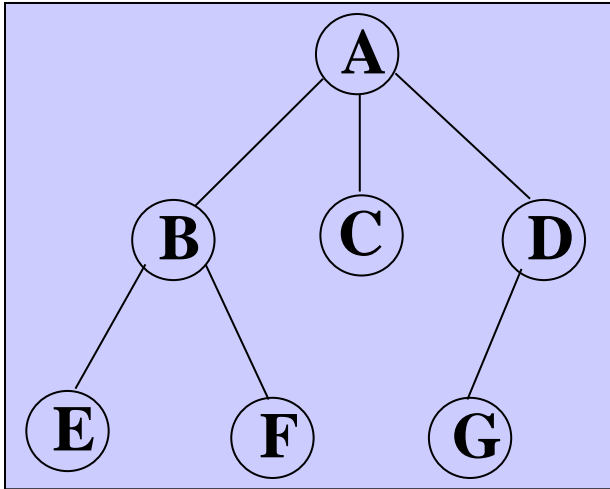
1. 兄弟加线.
2. 保留双亲与第一孩子连线, 删去与其他孩子的连线.
3. 顺时针转动, 使之层次分明.

树转换为二叉树的算法

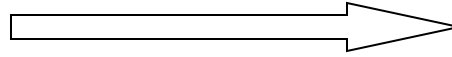
(1)加线——树中所有相邻兄弟之间加一条连线。

(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。

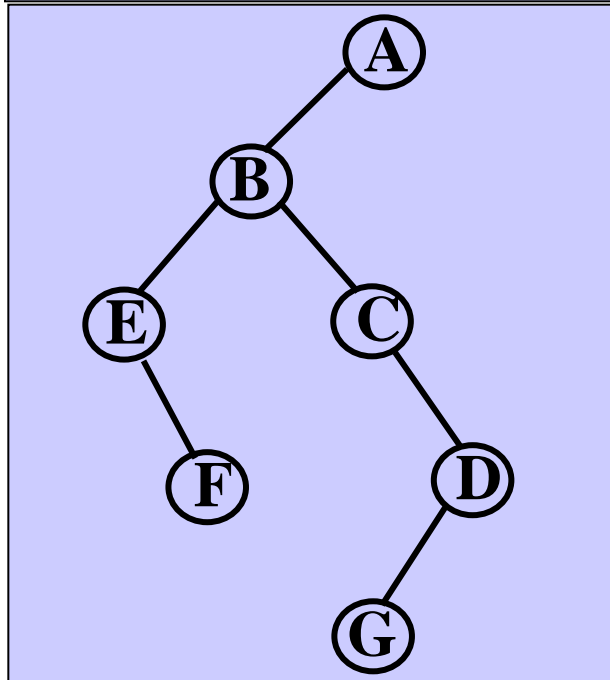


前序遍历

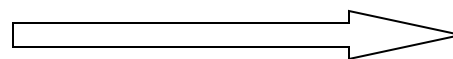


ABEFCDG

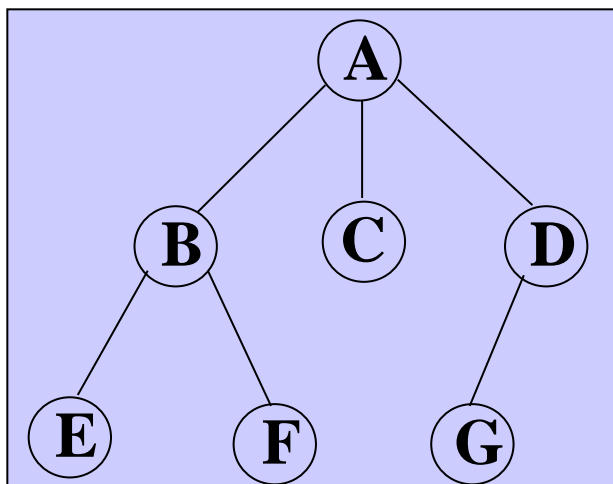
树的前序遍历等价于
二叉树的前序遍历！



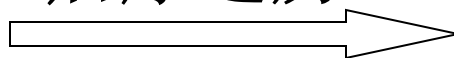
前序遍历



ABEFCDG

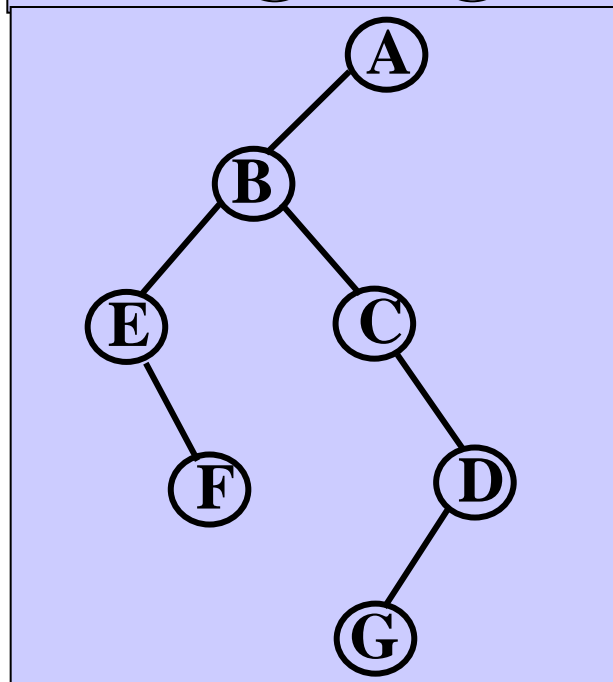


后序遍历

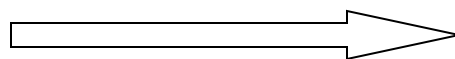


EFBCGDA

树的后序遍历等价于
二叉树的中序遍历！



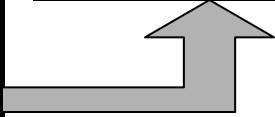
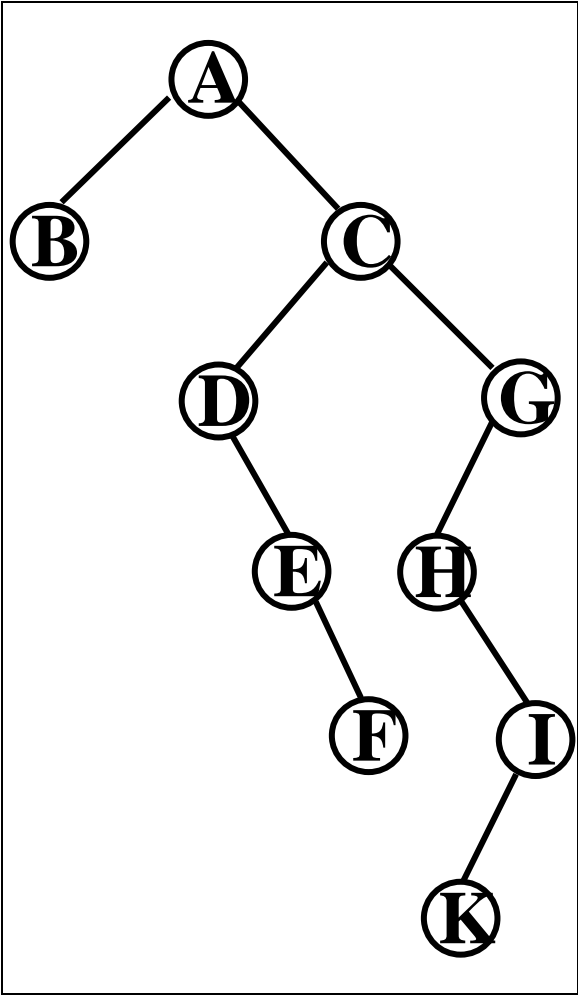
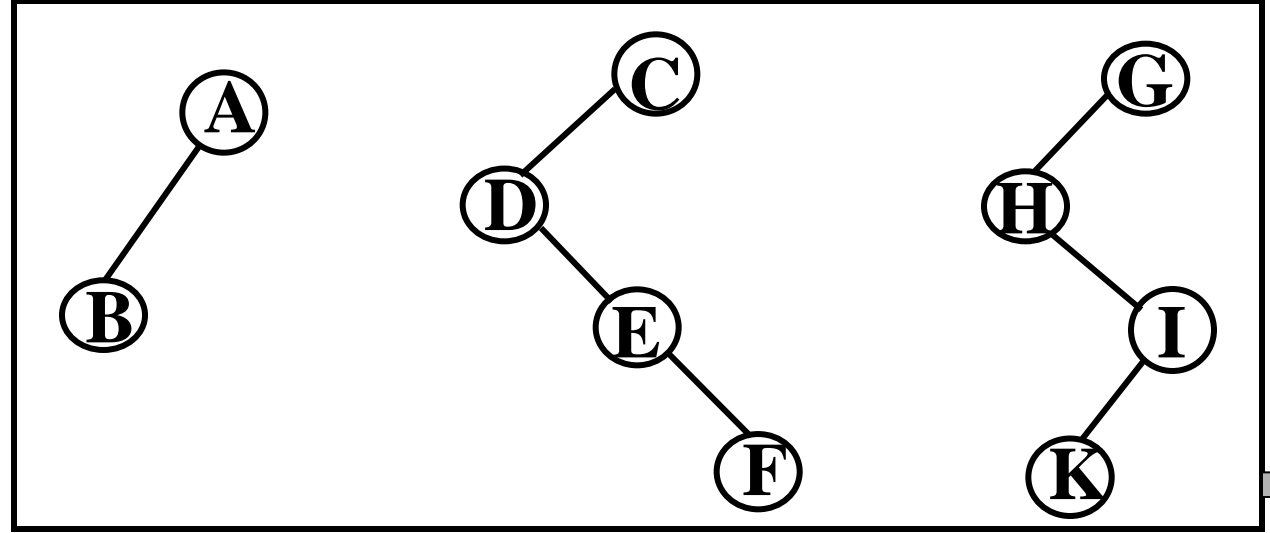
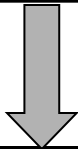
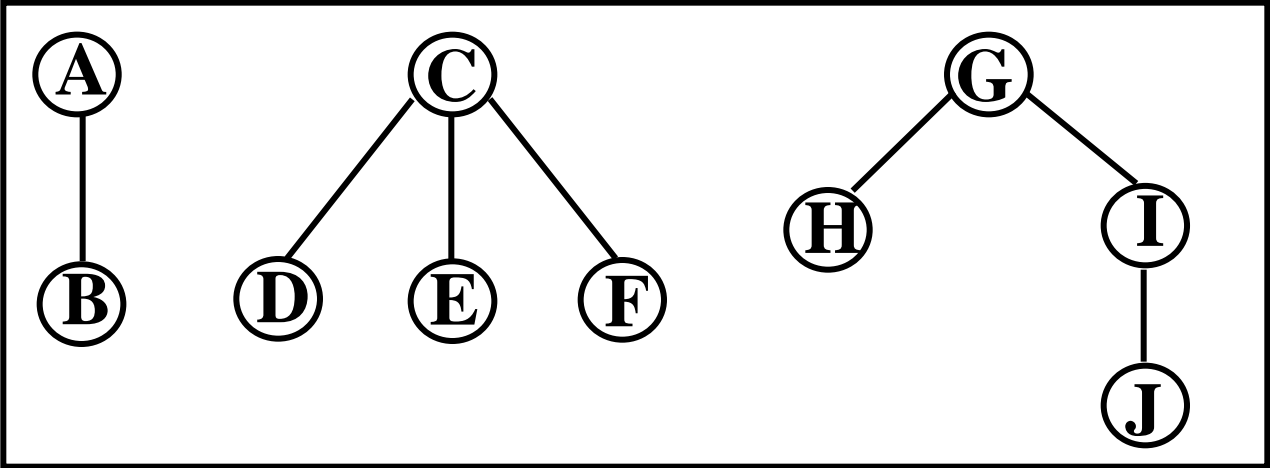
中序遍历



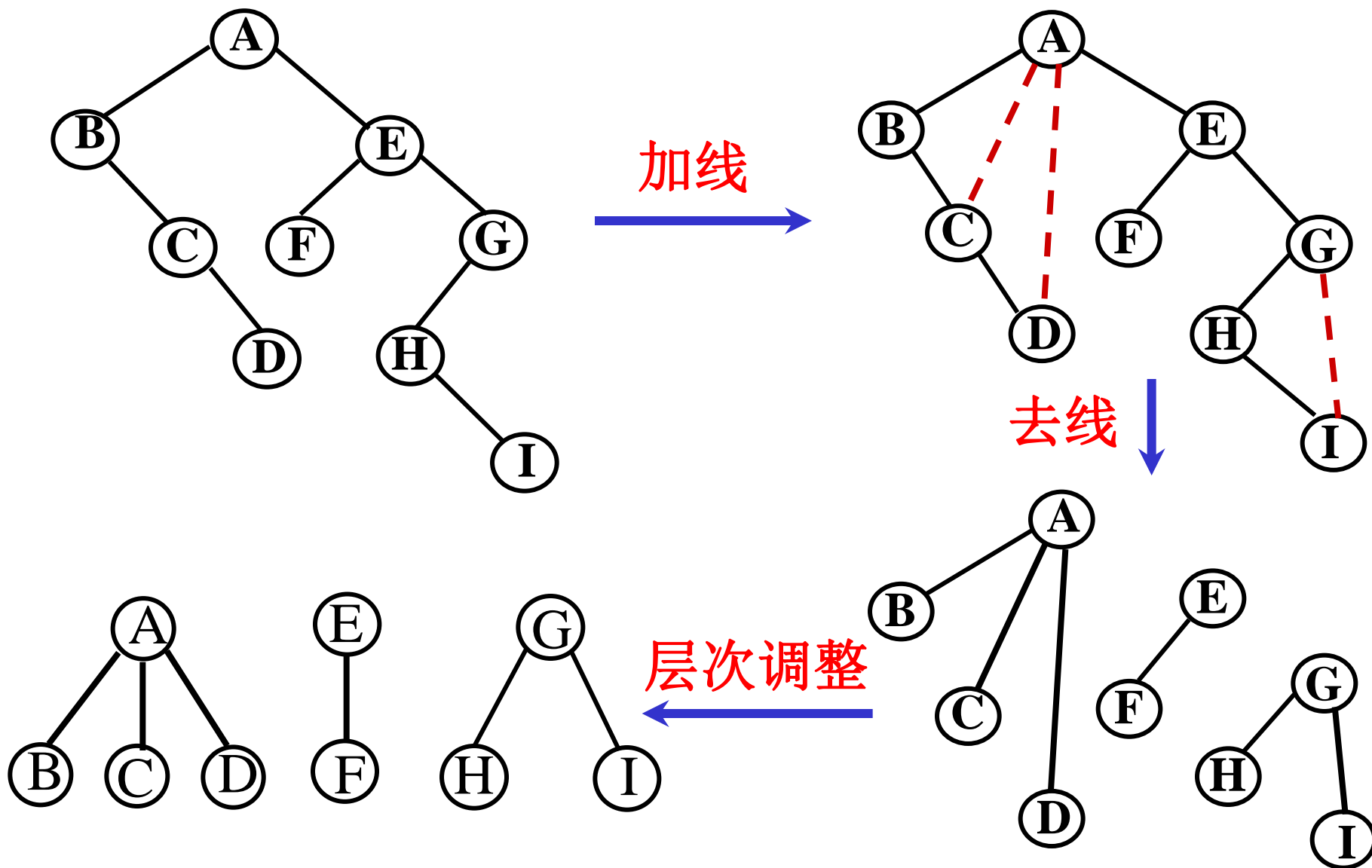
EFBCGDA

森林转换为二叉树的算法

- (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。



二叉树转换为树或森林



二叉树转换为树或森林的算法

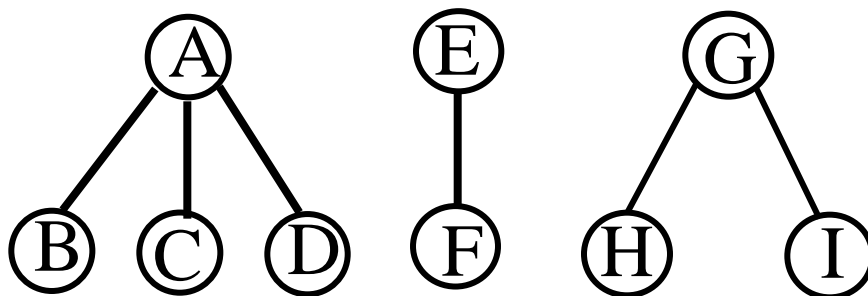
- (1) 加线——若某结点 x 是其双亲 y 的左孩子，则把结点 x 的右孩子、右孩子的右孩子、……，都与结点 y 用线连起来；
- (2) 去线——删去原二叉树中所有的双亲结点与右孩子结点的连线；
- (3) 层次调整——整理由(1)、(2)两步所得到的树或森林，使之层次分明。

森林的遍历

森林有两种遍历方法：

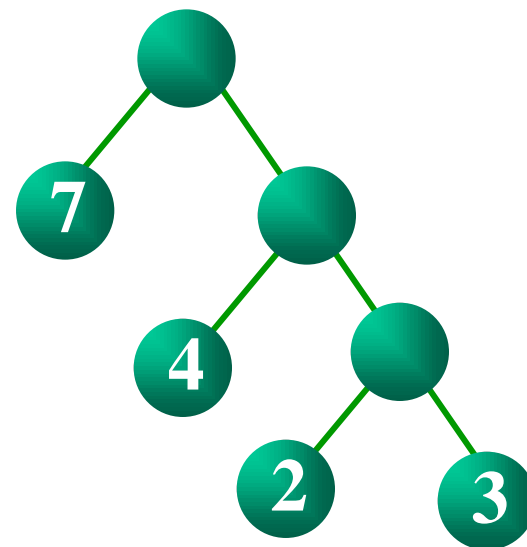
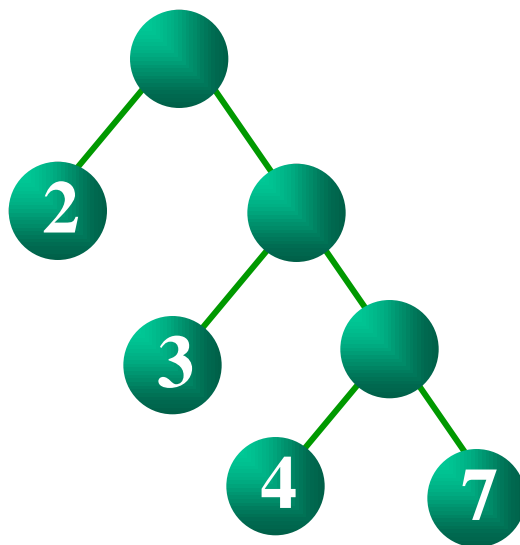
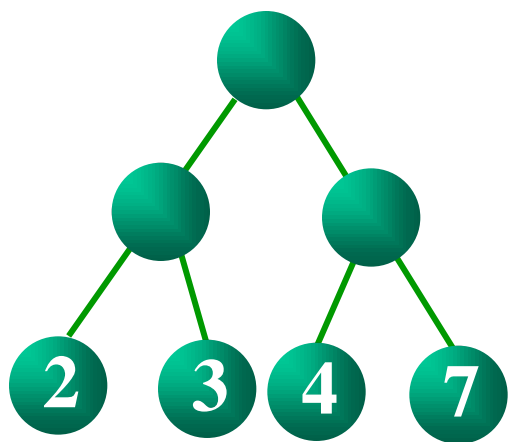
(1) **前序（根）遍历**：前序遍历森林即为前序遍历森林中的每一棵树。

(2) **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。



思考题

给定4个叶子结点，其权值分别为{2, 3, 4, 7}，可以构造多少种形状不同的二叉树？



还有吗？

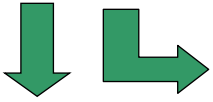
5.6 哈夫曼树及哈夫曼编码

相关概念

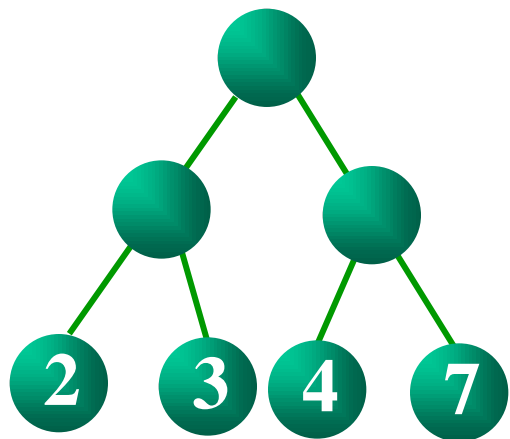
叶子结点的权值：对叶子结点赋予的一个有意义的数值量。

二叉树的带权路径长度：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：

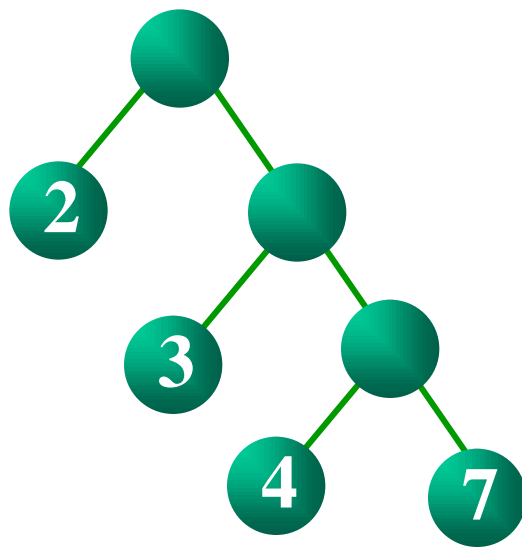
$$WPL = \sum_{k=1}^n w_k l_k$$

 从根结点到第 k 个叶子的路径长度
第 k 个叶子的权值；

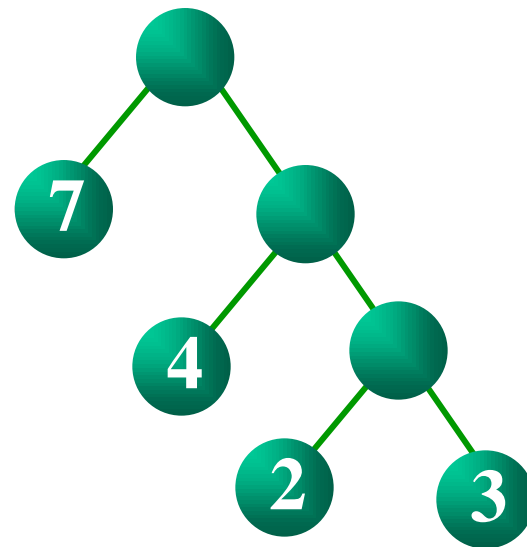
例：给定4个叶子结点，其权值分别为{2, 3, 4, 7}，可以构造出形状不同的多个二叉树。



WPL=32



WPL=41

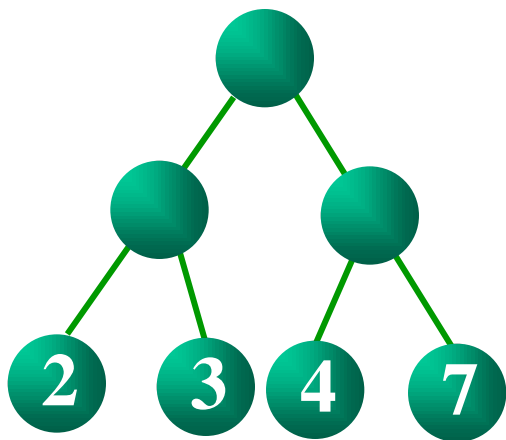


WPL=30

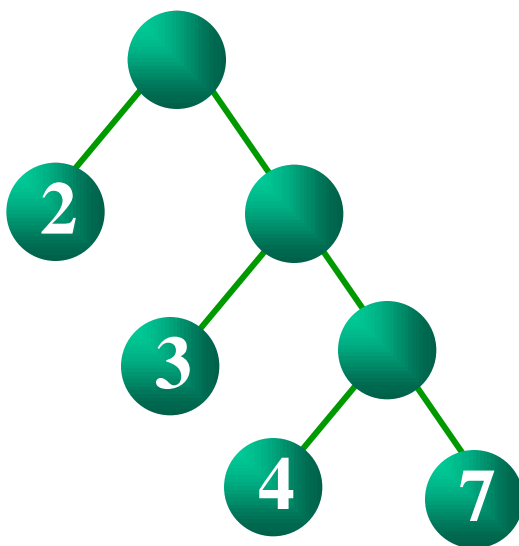
哈夫曼树：给定一组具有确定权值的叶子结点，带权路径长度最小的二叉树。

哈夫曼树的特点

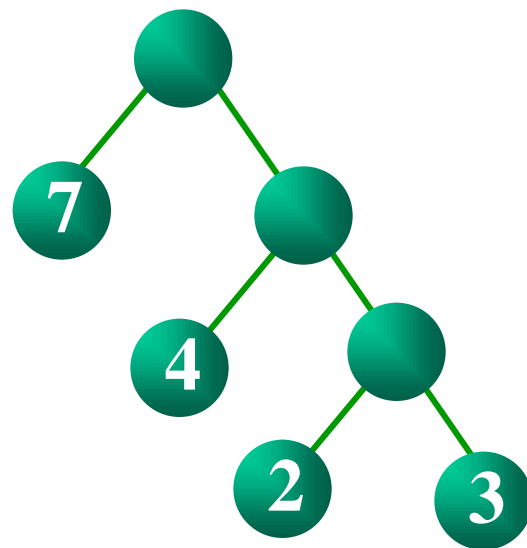
1. 权值越大的叶子结点越靠近根结点，而权值越小的叶子结点越远离根结点。
2. 只有度为0（叶子结点）和度为2（分支结点）的结点，不存在度为1的结点。



WPL=32



WPL=41



WPL=30

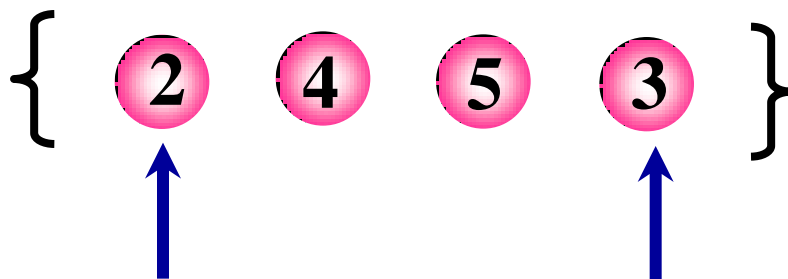
哈夫曼算法基本思想

- (1) **初始化**: 由给定的 n 个权值 $\{w_1, w_2, \dots, w_n\}$ 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 $F = \{T_1, T_2, \dots, T_n\}$;
- (2) **选取与合并**: 在 F 中选取根结点权值**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。

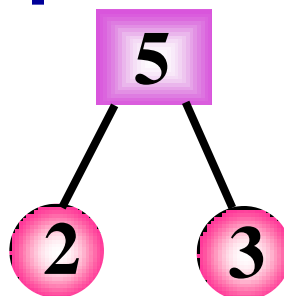
哈夫曼树的构造过程举例

$W=\{2, 3, 4, 5\}$

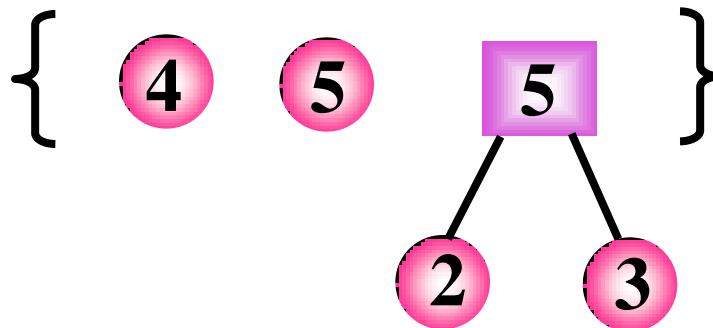
第1步：初始化



第2步：选取与合并

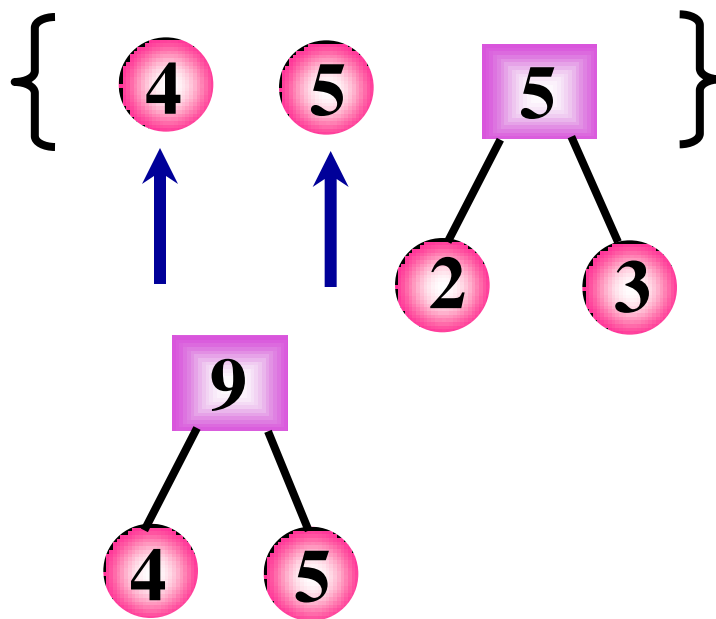


第3步：删除与加入

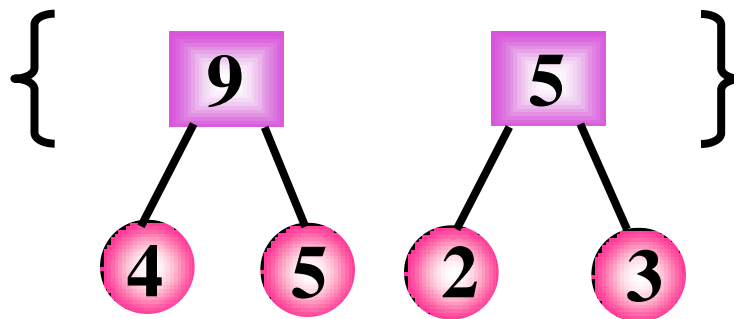


$$W = \{2, 3, 4, 5\}$$

重复第2步

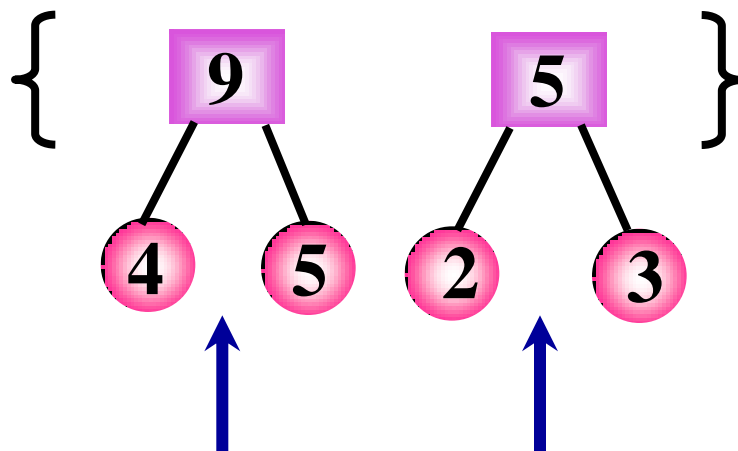


重复第3步

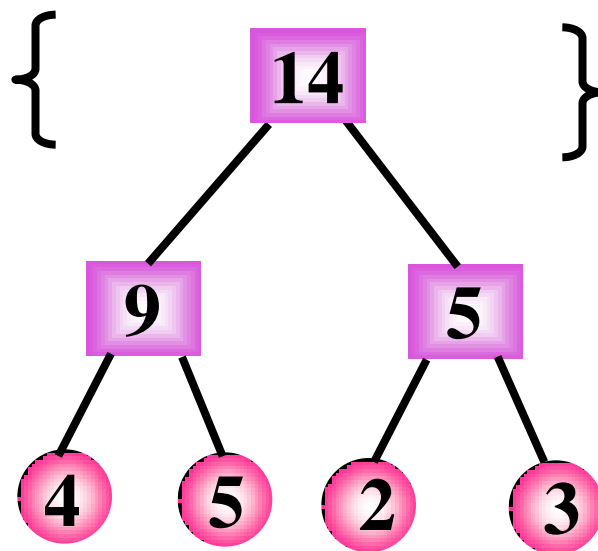


$$W = \{2, 3, 4, 5\}$$

重复第2步



重复第3步



哈夫曼算法的存储结构

为什么？

1. 设置一个数组 `huffTree[2n-1]` 保存哈夫曼树中各点的信息，数组元素的结点结构：

weight	lchild	rchild	parent
--------	--------	--------	--------

其中：

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

哈夫曼算法（伪代码）

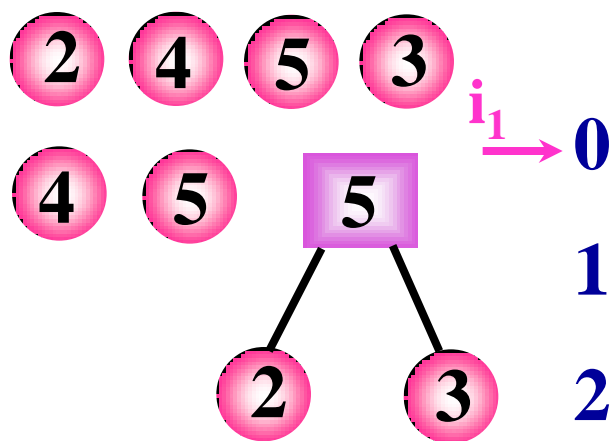
1. 数组huffTree初始化，所有元素结点的双亲、左右孩子都置为-1；
2. 数组huffTree的前n个元素的权值置给定值w[n]；
3. 进行n-1次合并
 - 3.1 在二叉树集合中选取两个权值最小的根结点，其下标分别为 i_1, i_2 ；
 - 3.2 将二叉树 i_1 、 i_2 合并为一棵新的二叉树k；

举例

2 4 5 3

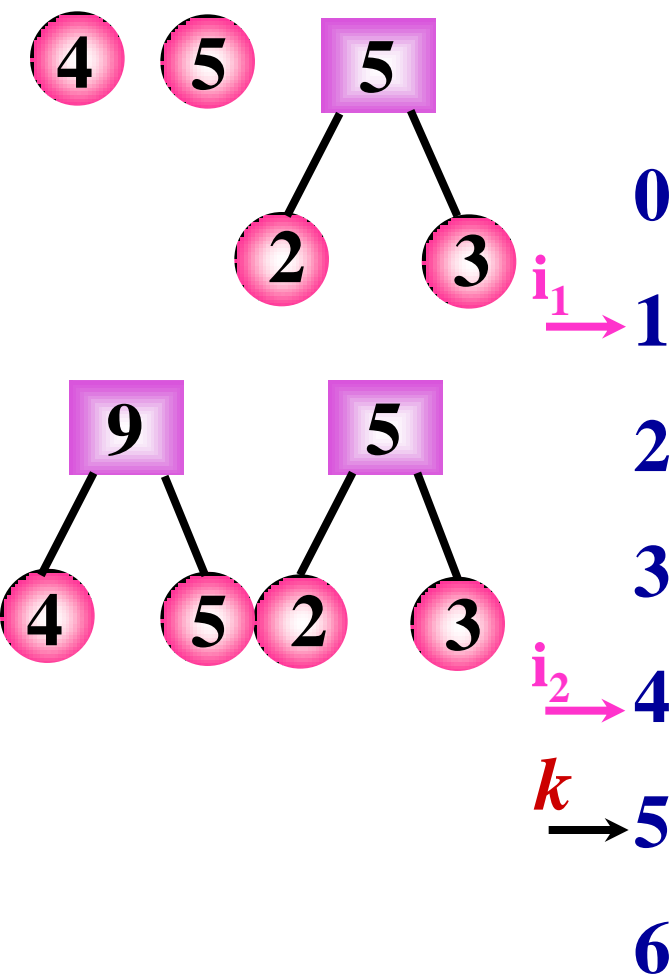
	weight	parent	lchild	rchild
0	2	-1	-1	-1
1	4	-1	-1	-1
2	5	-1	-1	-1
3	3	-1	-1	-1
4		-1	-1	-1
5		-1	-1	-1
6		-1	-1	-1

初 态



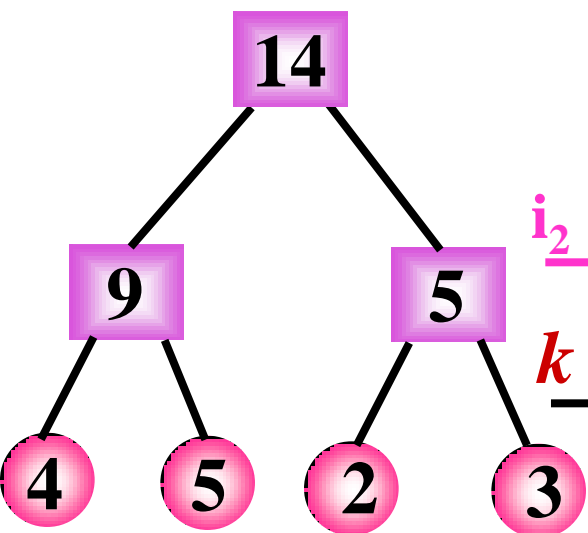
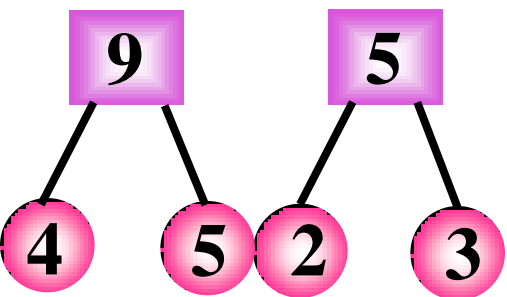
	weight	parent	lchild	rchild
0	2	4 -1	-1	-1
1	4	-1	-1	-1
2	5	-1	-1	-1
3	3	4 -1	-1	-1
4	5	-1	0 -1	3 -1
5		-1	-1	-1
6		-1	-1	-1

过程



	weight	parent	lchild	rchild
0	2	4 -1	-1	-1
1	4	5 -1	-1	-1
2	5	-1	-1	-1
3	3	4 -1	-1	-1
4	5	5 -1	0 -1	3 -1
5	9	-1	1 -1	4 -1
6		-1	-1	-1

过程



i_1 →

i_2 →

k →

weight parent lchild rchild

	weight	parent	lchild	rchild
0	2	4 -1	-1	-1
1	4	5 -1	-1	-1
2	5	6 -1	-1	-1
3	3	4 -1	-1	-1
4	5	5 -1	0 -1	3 -1
5	9	6 -1	1 -1	4 -1
6	14	-1	2 -1	5 -1

过程

哈夫曼算法（程序）：

```
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？



不等长编码的问题？

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

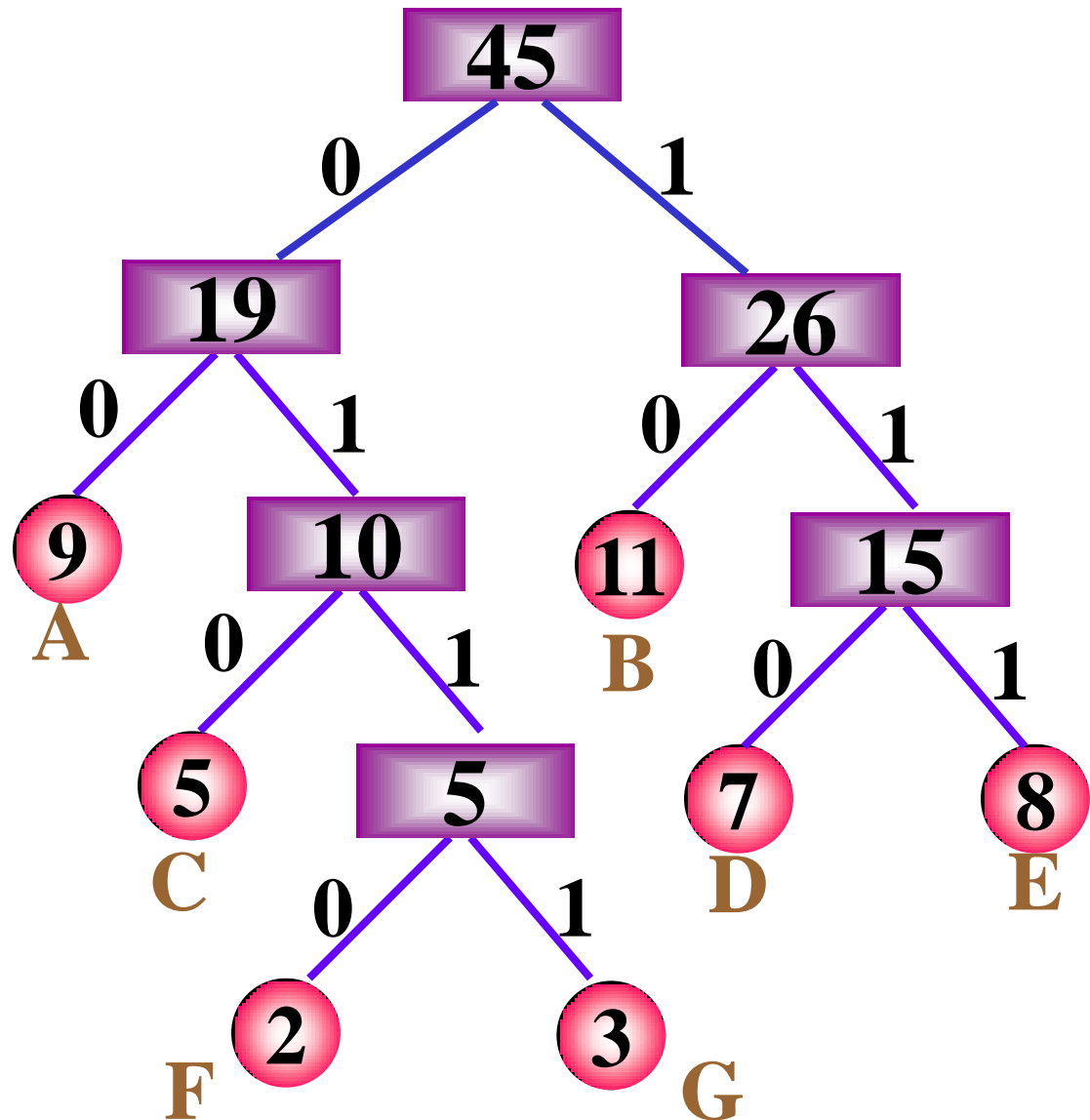
C: 010

D: 110

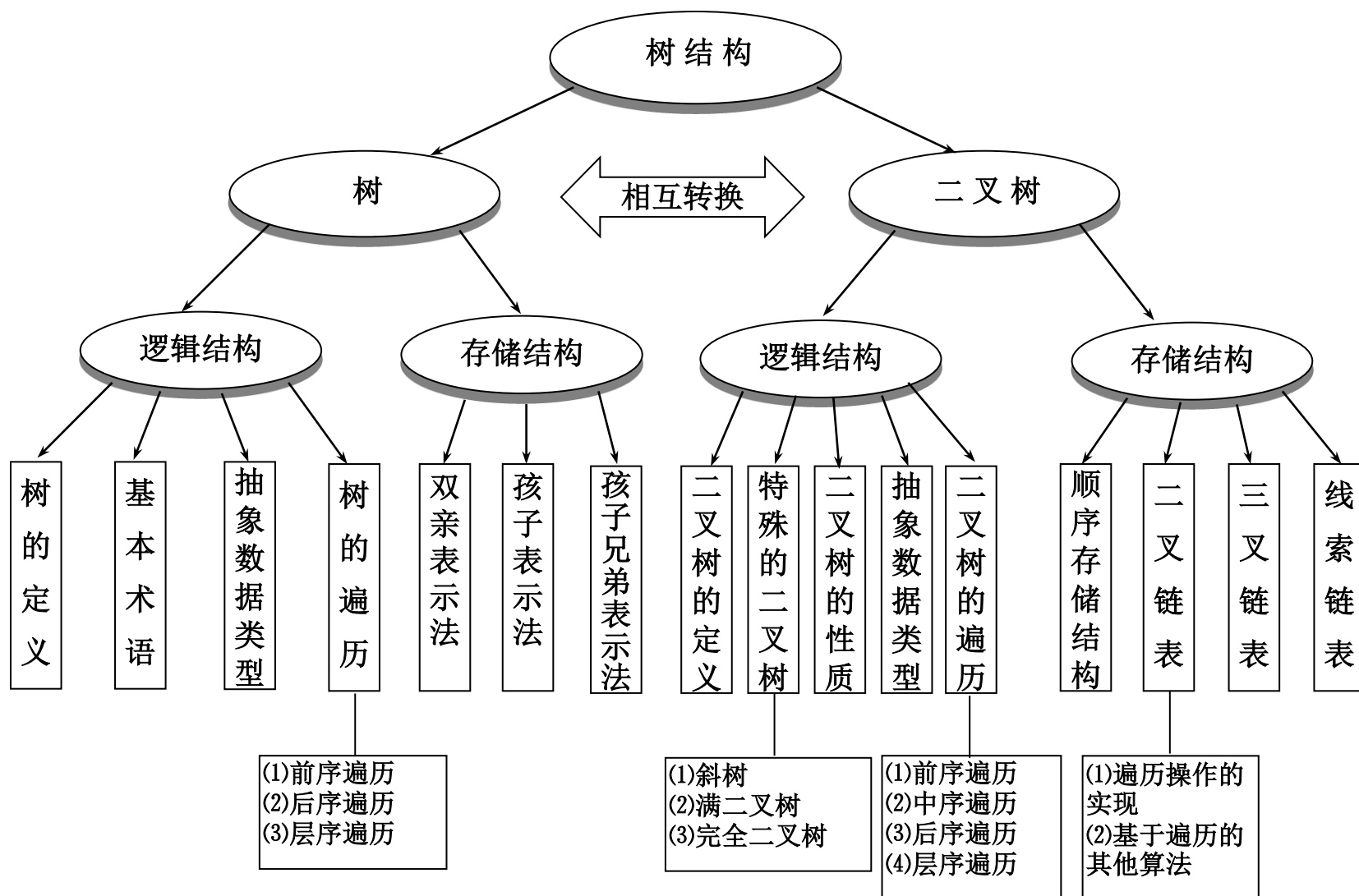
E: 111

F: 0110

G: 0111



本章小结——知识结构图



作业

习题5 (P140) : 4(4)(5)(6), 5(1)(2)(3)

已知某字符串 S 为 “*abcdeacedaeaddcedabadadaead*” ,
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。

二叉树算法设计练习

根据中序遍历算法求二叉树的结点个数。

```
void Count(BiNode *root) //n为全局量并已初始化为0
{
    if (root) {
        Count(root->lchild);
        n++;
        Count(root->rchild);
    }
}
```

二叉树算法设计练习

根据后序遍历算法求二叉树的深度。

```
int Depth(BiNode *root)
{
    if (root == NULL) return 0;
    else {
        hl= Depth(root->lchild);
        hr= Depth(root->rchild);
        return max(hl, hr)+1;
    }
}
```