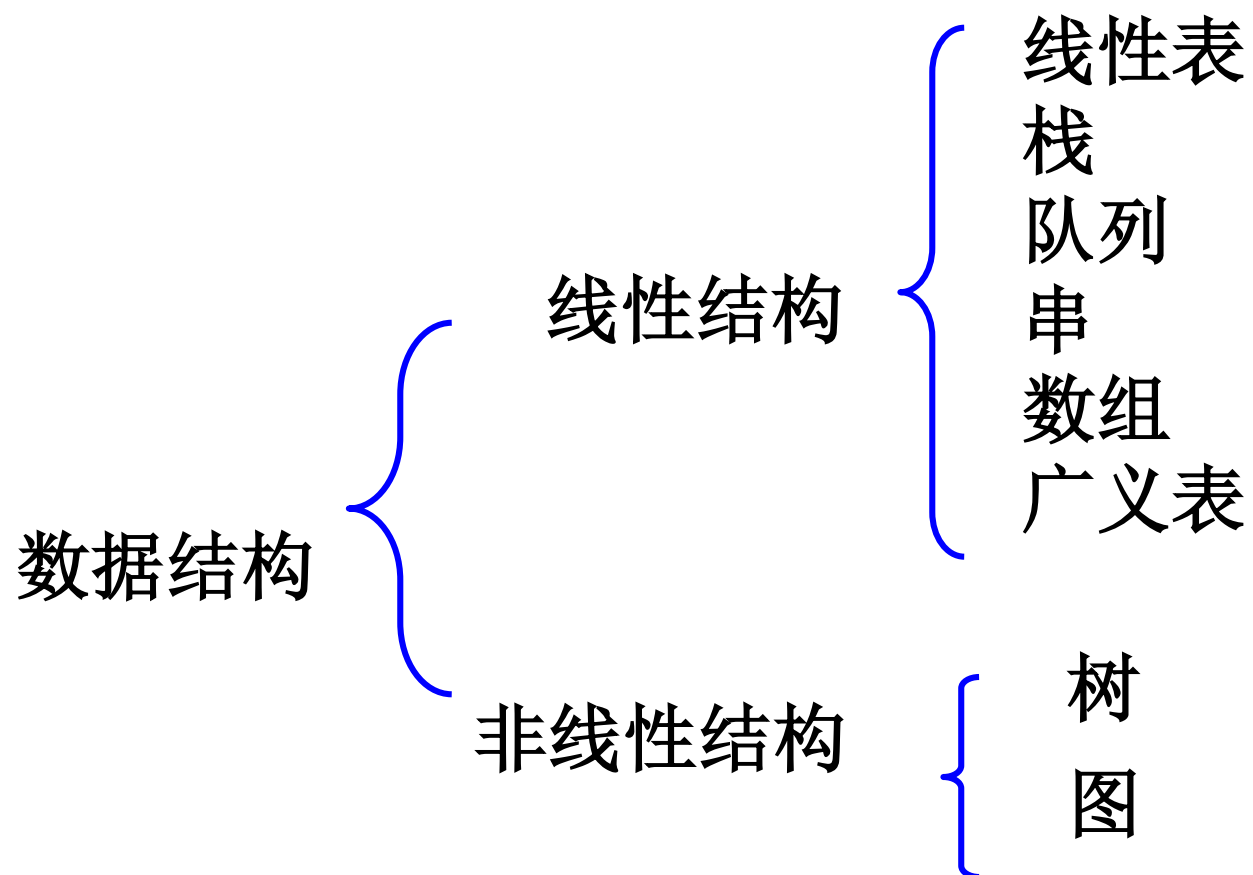






# 数据结构类型



本章是“数据结构”课程的重点



# 本章学习要点





# 本章重点与难点

## ■ 重点：

- (1) 二叉树的定义、存储结构、遍历及应用；
- (2) 线索二叉树的定义、存储结构及相应的操作；
- (3) 树和森林与二叉树之间的相互转化方法；
- (4) 哈夫曼树的建立方法和哈夫曼编码。

## ■ 难点：

- (1) 二叉树的构造 、应用；
- (2) 线索二叉树的遍历和相应的操作。



# 第六章 树

6.1 树的有关概念

6.2 二叉树

6.3 二叉树的遍历

6.4 遍历的应用

6.5 线索二叉树

6.6 树和森林

6.7 树及应用



# 第六章 树

## 6.1 树的有关概念

## 6.2 二叉树

## 6.3 二叉树的遍历

## 6.4 遍历的应用

## 6.5 线索二叉树

## 6.6 树和森林

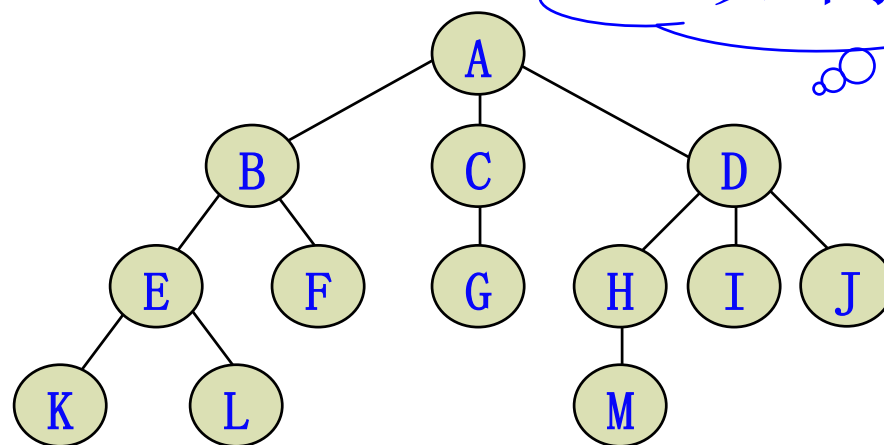
## 6.7 树及应用



## 6.1 树的有关概念

- 树的定义:

如何定义树?





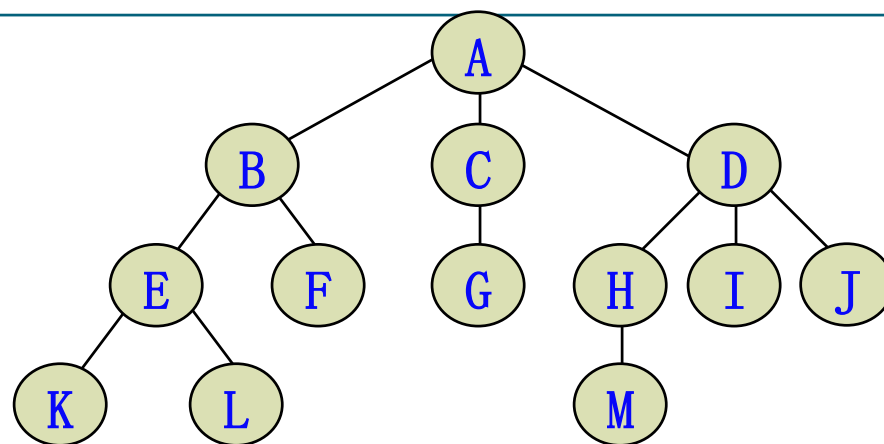
## 6.1 树的有关概念

- 树的定义:

树形结构是一种重要的非线性结构。树是 $n$  ( $n \geq 0$ ) 个结点的有限集合，当 $n=0$ 时，称其为**空树**。在任一棵非空树中：

(1) 有且仅有一个称为**根的结点**。

(2) 当 $n > 1$ 时。其余结点可分为 $m$ 个**互不相交的集合**，而且这些集合中的每一集合都本身又是一棵树，称为根的子树。



**说明：**树是递归结构，在树的定义中又用到了树的概念





## 6.1 树的有关概念

- 树的构造性定义：
  - 一个结点 $x$  组成的集合 $\{x\}$ 是**一棵树**，这个结点 $x$  称为**这棵树的根**（root）。
  - 假设 $x$  是一个结点， $T_1, T_2, \dots, T_k$ 是 $k$ 棵互不相交的树，可以构造一棵新树：令 $x$  为根，并有 $k$  条边由 $x$  指向树 $T_1, T_2, \dots, T_k$ 。这些边也叫做**分支**， $T_1, T_2, \dots, T_k$ 称作根为 $x$ 的树之**子树**（SubTree）。



## 6.1 树的有关概念

定义的共同点：

- 1、相同类型的元素构成的集合；
- 2、特定的结点---根；
- 3、除了根之外，组成  $k$  个划分，且互不相交；
- 4、每一个划分又是一棵树---递归；

几点说明：

- ① 递归定义，但不会产生循环定义；
- ② 构造性定义便于树型结构的建立；
- ③ 一株树的每个结点都是这株树的某株子树的根。



## 6.1 树的有关概念

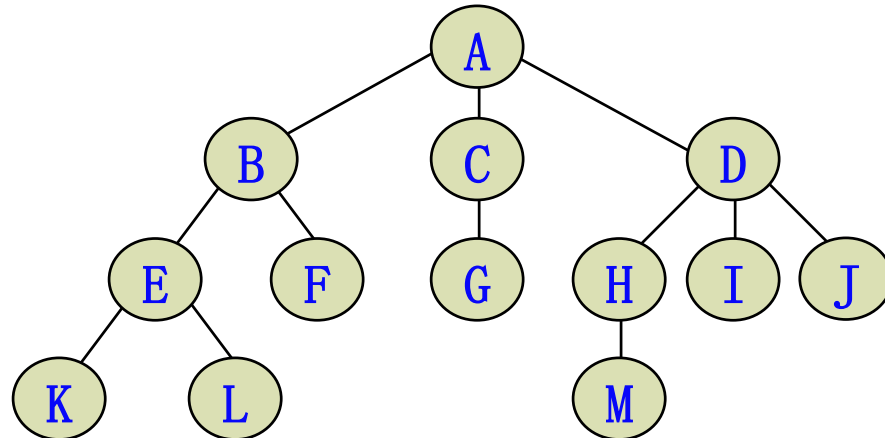
- 树的概念:

例如: 集合  $T = \{A, B, C, D, E, F, G, H, I, J, K, L, M\}$

A是根, 其余结点可以划分为3个互不相交的集合:

$T_1 = \{B, E, F, K, L\}$  ,  $T_2 = \{C, G\}$  ,  $T_3 = \{D, H, I, J, M\}$

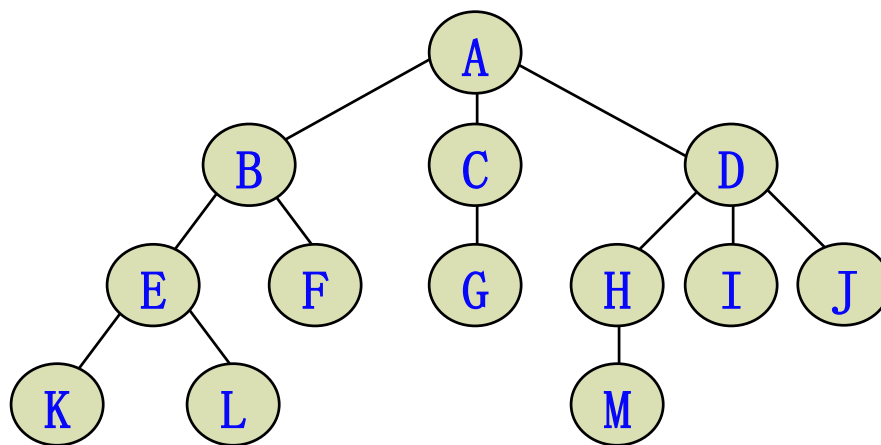
这些集合中的每一集合都本身又是一棵树, 它们是A的子树。





## 6.1 树的有关概念

- 树的逻辑结构特点：
  - 树是一种分枝结构，树中只有根结点没有前驱；其余结点都**有且仅一个**前驱；都存在**唯一**一条从根到该结点的路径。
  - 每个结点可以有零个或多个后继；





## 6.1 树的有关概念

### 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)

### 非线性结构—树

根结点  
(无前驱)

多个叶子结点  
(无后继)

其它数据元素  
(一个前驱、多个后继)

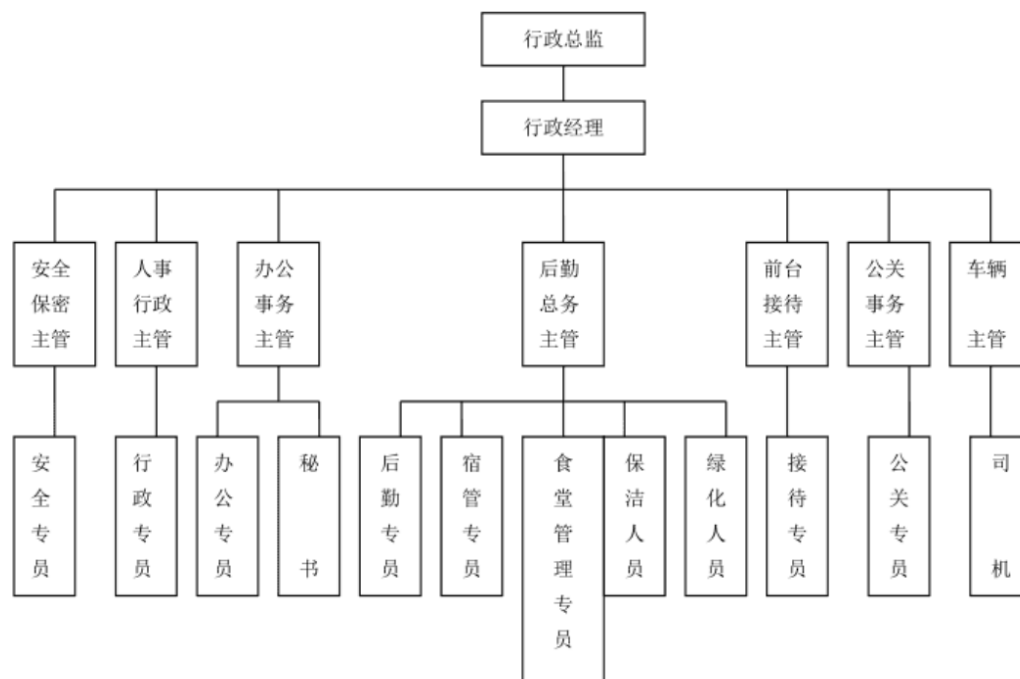


## 6.1 树的有关概念

- 树的应用:

1) 树可表示具有分枝结构关系的对象

**例** 单位行政机构的组织关系





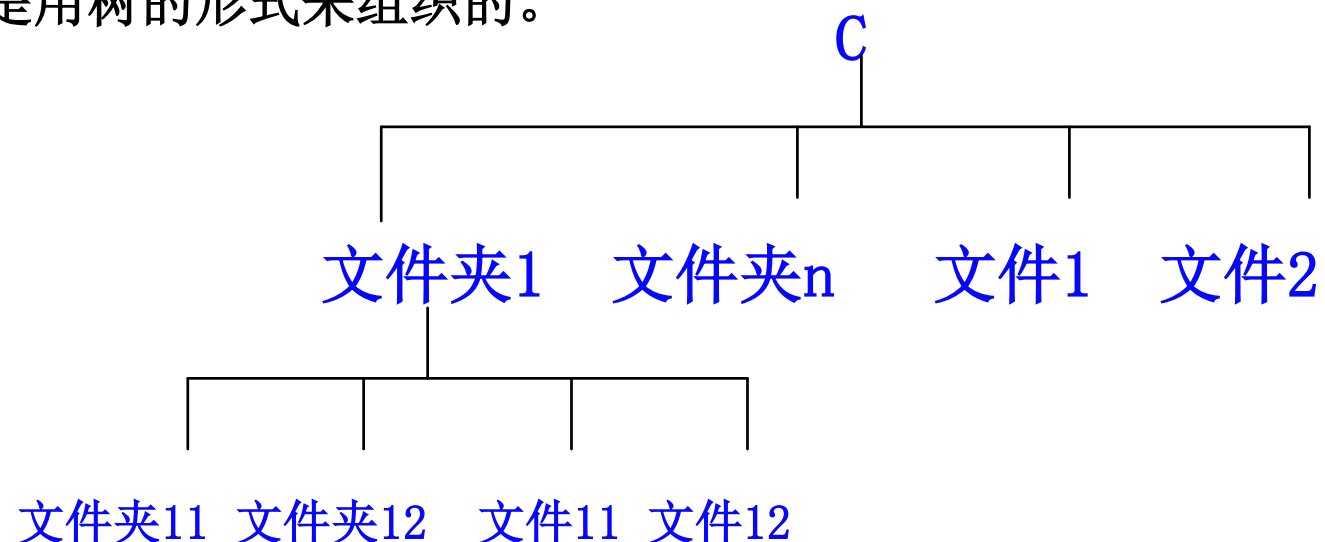
## 6.1 树的有关概念

- 树的应用：

- 2) 树是常用的数据组织形式

- 有些应用中数据元素之间并不存在分支结构关系，但是为了便于管理和使用数据，将它们用树的形式来组织。

**例** **计算机的文件系统：**不论是DOS文件系统还是window文件系统，所有的文件是用树的形式来组织的。

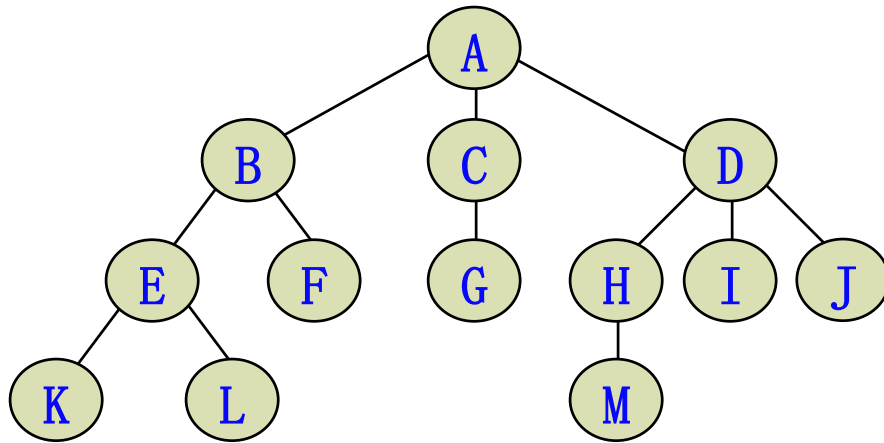




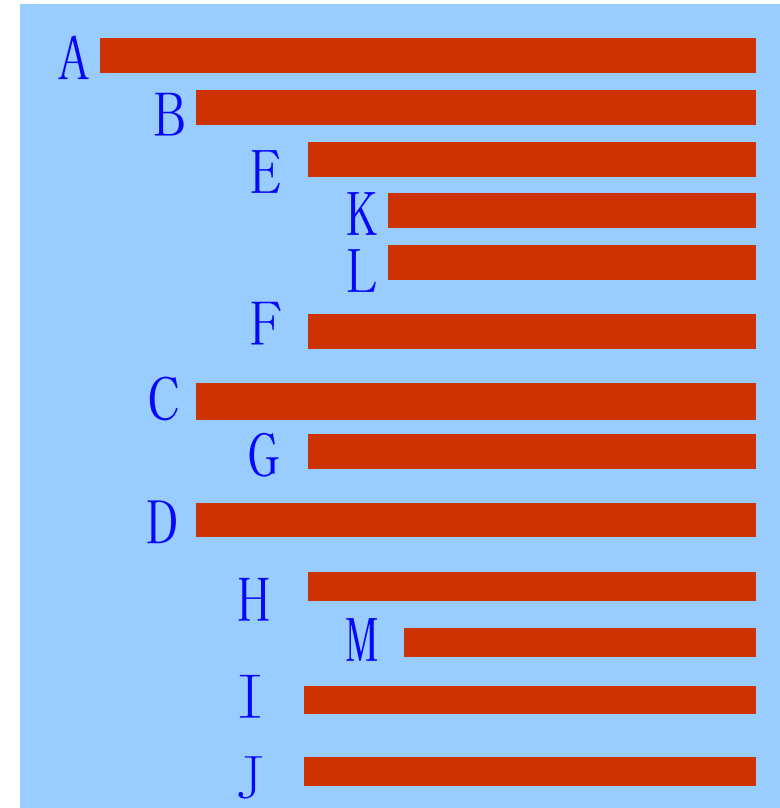
## 6.1 树的有关概念

- 树的表示:

### (1) 树形表示法



### (2) 凹入表示法



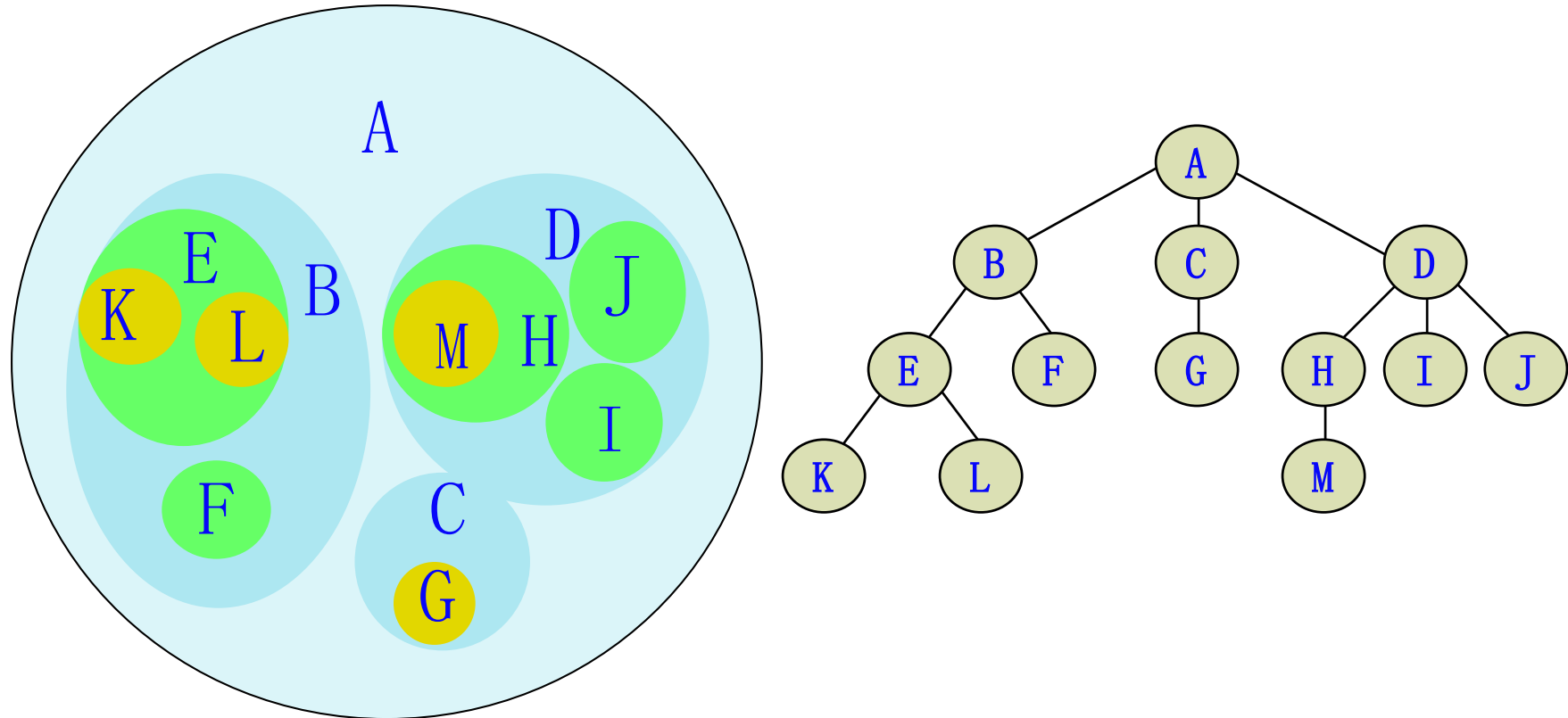




## 6.1 树的有关概念

- 树的表示:

- (3) 嵌套集合表示法 (文氏图)

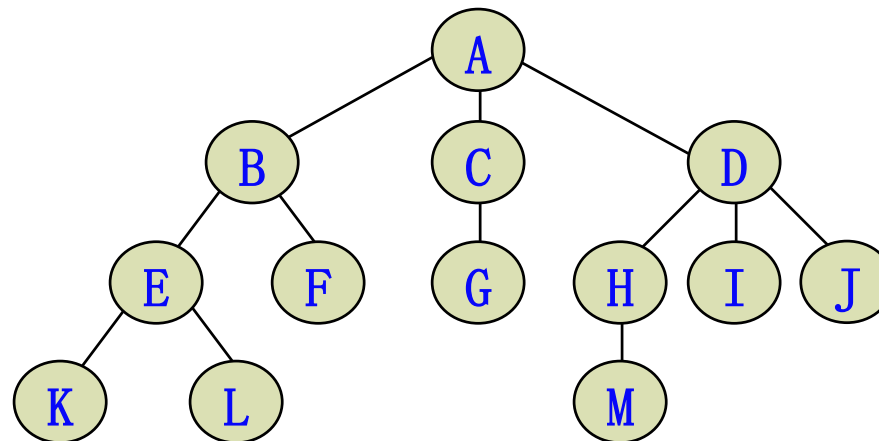




## 6.1 树的有关概念

- 树的表示:

### (4) 广义表表示法



(A) 第一层

(A(B, C, D)) 第二层

(A(B(E, F), C(G), D(H, I, J))) 第三层

(A(B(E(K, L), F), C(G), D(H(M), I, J))) 第四层

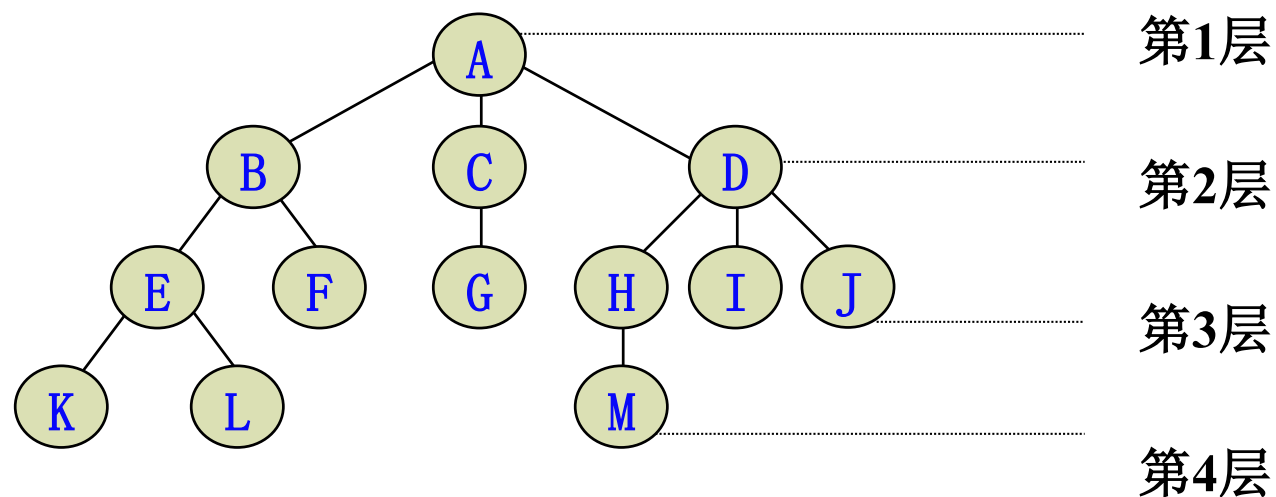
表示方法的多样化说明树结构应用的重要性



## 6.1 树的有关概念

- 树的有关术语:

- 结点: 包含一个数据元素和若干指向子树的分支 (边)
  - 结点的度: 结点子树的个数;
  - 叶子节点 (终端节点): 度为0的节点
  - 分支节点 (非终端节点): 度不为0的节点
  - 结点的层: 根结点的层定义为1, 其它依此类推;





## 6.1 树的有关概念

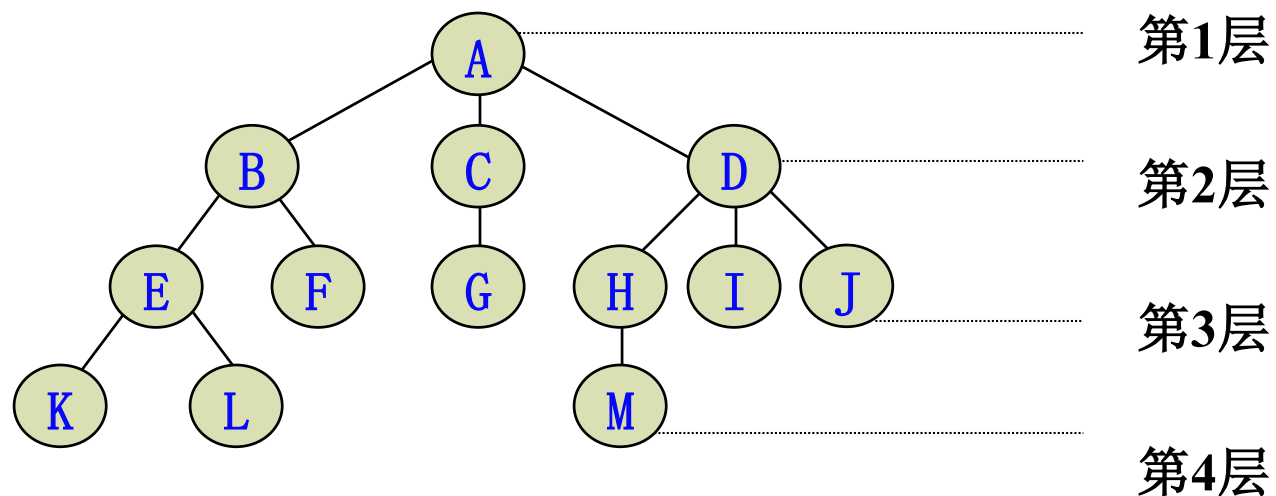
- 树的有关术语:

 **树的深度（高）**：树中最大的结点层；

 **树的度**：树中最大的结点度；

树的度为**3**

树的深度为**4**



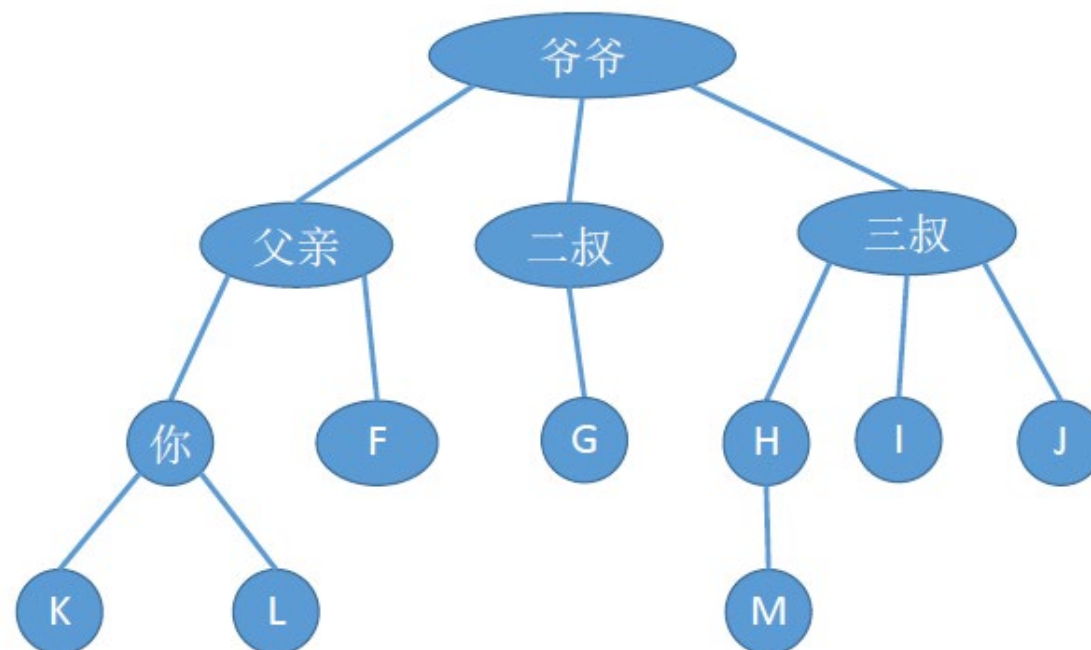


## 6.1 树的有关概念

- 树的有关术语:

- 结点的子树称为该结点的**孩子**，该结点称为孩子的**双亲（父亲）**，同一个双亲的孩子称为**兄弟**；
- 祖先**：从根到该结点的所经分支上的所有结点
- 子孙**：以某结点为根的子树中任一结点都称为该结点的子孙

堂兄弟节点？



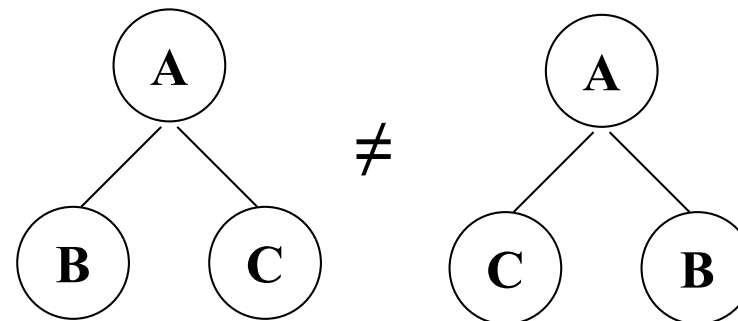


## 6.1 树的有关概念

- 树的有关术语:

-  **有序树**: 子树有序的树, 如: 家族树;

-  **无序树**: 不考虑子树的顺序;



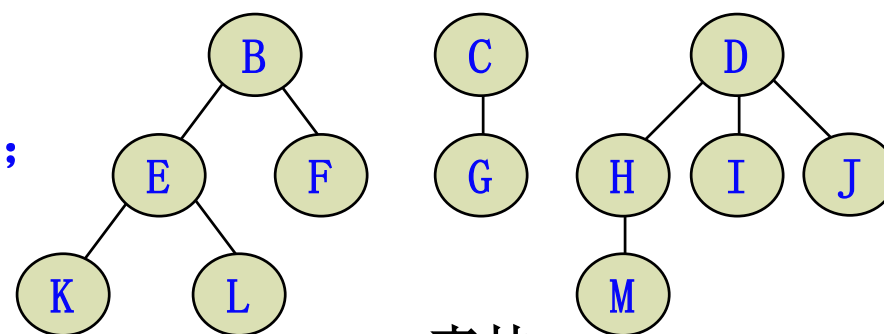
树

-  **森林**: 是 $m \geq 0$ 棵互不相交的树集合。

- **树和森林的关系**:

- (1) 一棵树去掉根, 其子树构成一个森林;

- (2) 一个森林增加一个根结点成为树。



森林



## 6.1 树的有关概念

- 树的抽象数据类型:

ADT Tree{

数据对象 D: D是具有相同特性的数据元素的集合。

数据关系 R:

若D为空集, 则称为空树。

否则:

(1) 在D中存在唯一的称为根的数据元素root;

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



## 6.1 树的有关概念

- 树的抽象数据类型:

ADT Tree{

数据对象 D:

数据关系 R:

基本操作 P:

查	找	类
插	入	类
删	除	类
		}





## 6.1 树的有关概念

- 树的抽象数据类型:

查找类:

Root(T) // 求树的根结点

Value(T, cur\_e) // 求当前结点的元素值

Parent(T, cur\_e) // 求当前结点的双亲结点

LeftChild(T, cur\_e) // 求当前结点的最左孩子

RightSibling(T, cur\_e) // 求当前结点的右兄弟

TreeEmpty(T) // 判定树是否为空树

TreeDepth(T) // 求树的深度

TraverseTree( T, Visit() ) // 遍历



## 6.1 树的有关概念

- 树的概念:

插入类:

`InitTree(&T)` // 初始化置空树

`CreateTree(&T, definition)` // 按定义构造树

`Assign(T, cur_e, value)` // 给当前结点赋值

`InsertChild(&T, &p, i, c)` // 将以c为根的树插入为结点p的第i棵子树



## 6.1 树的有关概念

- 树的概念:

删除类:

`ClearTree(&T)` // 将树清空

`DestroyTree(&T)` // 销毁树的结构

`DeleteChild(&T, &p, i)` // 删除结点p的第i棵子树



# 第六章 树

6.1 树的有关概念

**6.2 二叉树**

6.3 二叉树的遍历

6.4 遍历的应用

6.5 线索二叉树

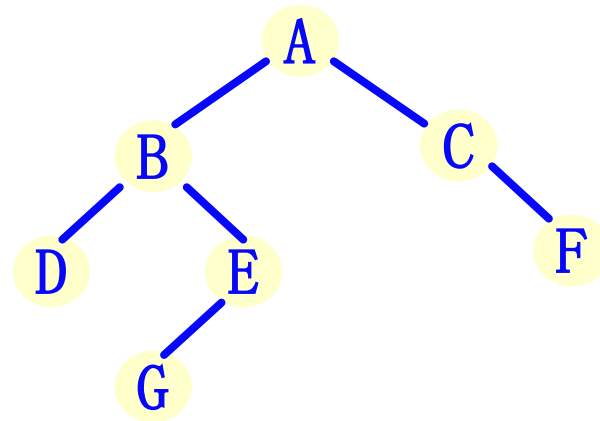
6.6 树和森林

6.7 树及应用



## 6.2 二叉树

树是一种分枝结构的对象，在树的概念中，对每一个结点孩子的个数没有限制，因此树的形态多种多样，本节我们主要讨论一种特殊的树型结构——**二叉树**。





## 6.2 二叉树

6.2.1 二叉树的概念

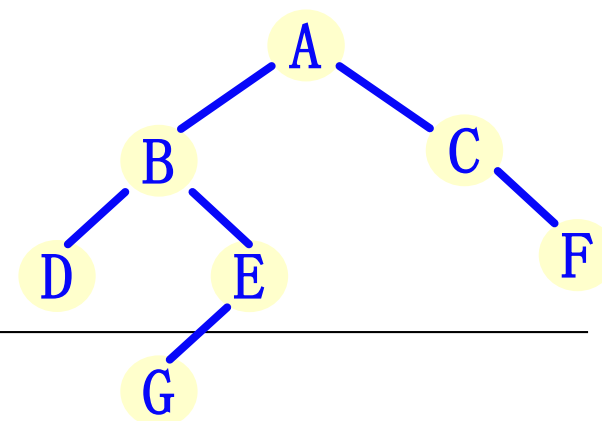
6.2.2 二叉树的性质

6.2.3 二叉树的存储结构



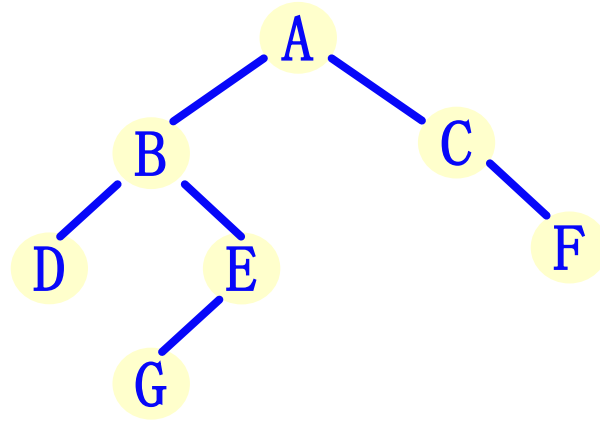
## 6.2.1 二叉树的概念

- **概念**：二叉树或为空树，或由根及两颗**不相交的左、右子树**构成，并且**左、右子树**本身也是二叉树。
- **特点**：
  - 二叉树中每个结点最多有两棵子树；即**二叉树每个结点的度小于等于2**；
  - 左、右子树不能颠倒——**有序树**；
  - 二叉树是**递归结构**，在二叉树的定义中又用到了二叉树的概念；

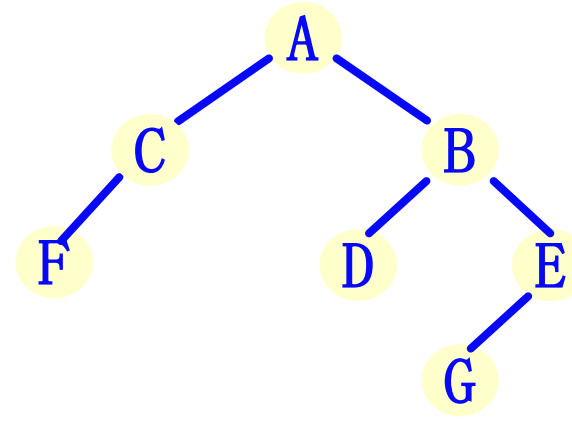




## 6.2.1 二叉树的概念



(a)



(b)

二叉树是有左右之分的



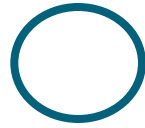


## 6.2.1 二叉树的概念

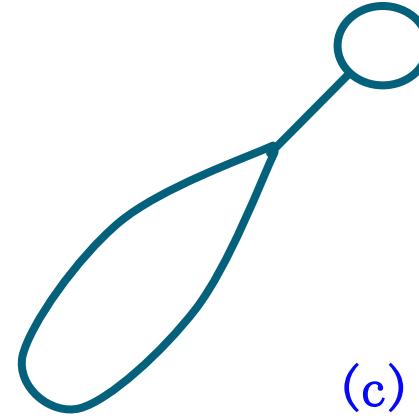
- 二叉树的基本形态

$\Phi$

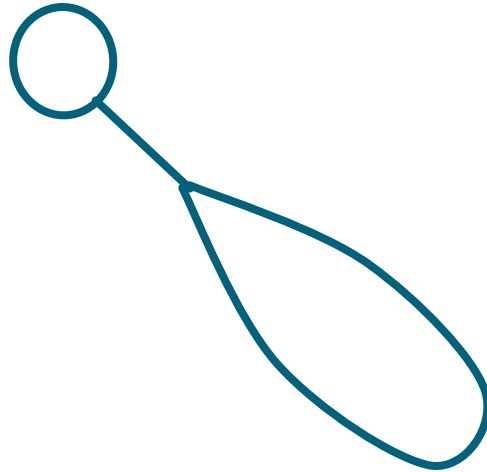
(a) 空树



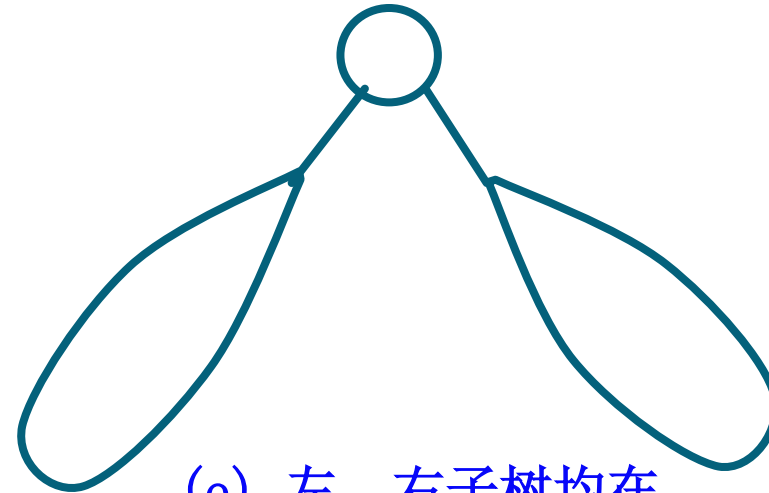
(b) 仅有根



(c) 右子树空



(d) 左子树空

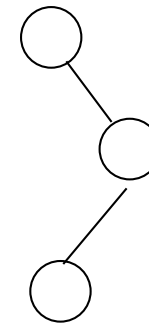
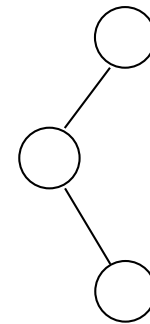
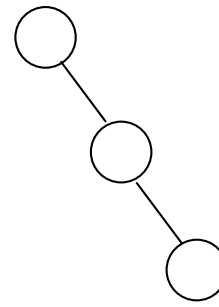
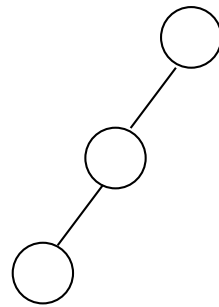
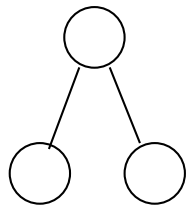
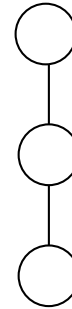
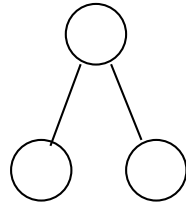


(e) 左、右子树均在



## 6.2.1 二叉树的概念

**问题：**具有三个结点的树和二叉树各有多少棵？





## 6.2.2 二叉树的性质

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

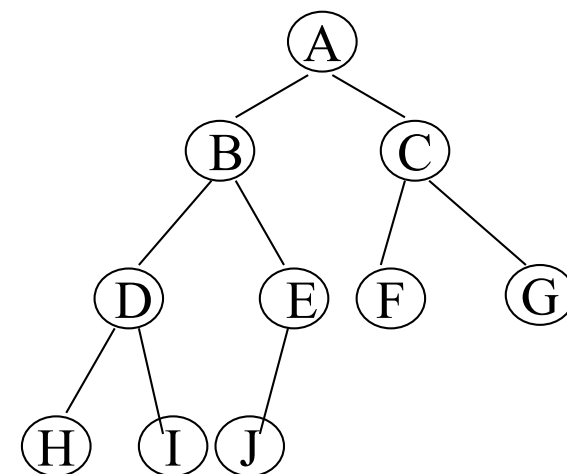
**证明：**用数学归纳法就可以证明。

**性质2** 深度为 $k$ 的二叉树**最多**有 $2^k - 1$ 个结点。

**证明：**最多结点数为各层结点个数相加，即

$$1 + 2 + 4 + \cdots + 2^{k-1} = 2^k - 1$$

**k层二叉树**

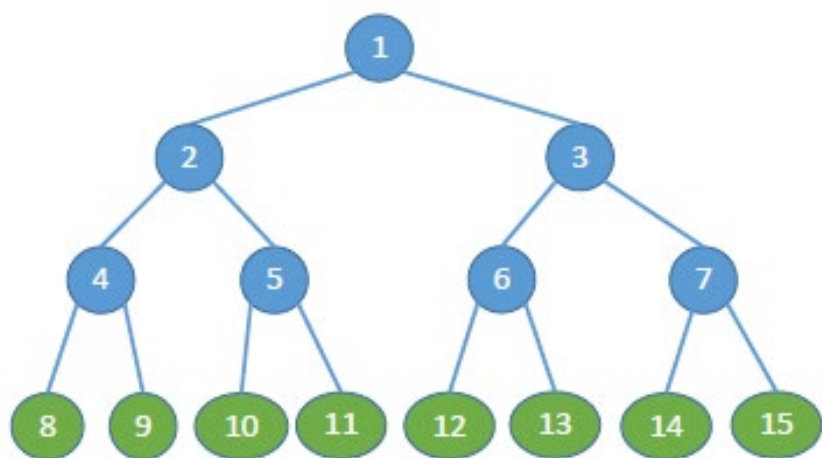




## 6.2.2 二叉树的性质

- 两种特殊的二叉树

**满二叉树：**如果深度为 $k$ 的二叉树，有 $2^k-1$ 个结点则称为满二叉树；



特点：

- ①只有最后一层有叶子结点
- ②不存在度为1的结点
- ③按层序从1开始编号，结点 $i$ 的左孩子为 $2i$ ，右孩子为 $2i+1$ ；结点 $i$ 的父节点为 $\lfloor i/2 \rfloor$

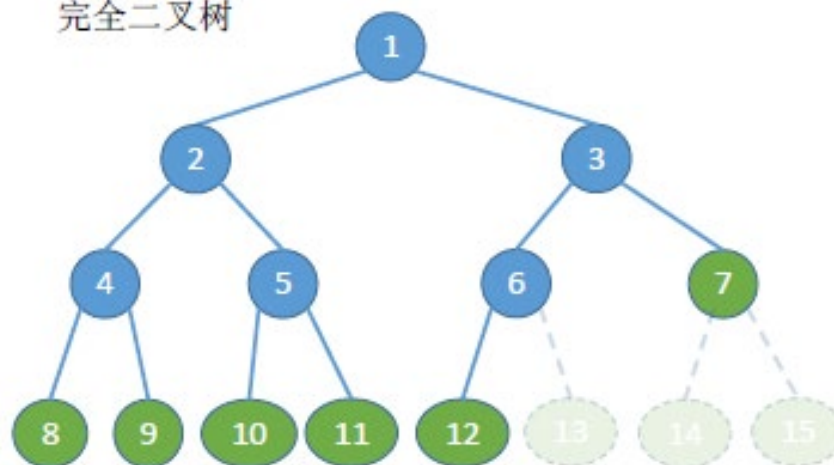


## 6.2.2 二叉树的性质

- 两种特殊的二叉树

**完全二叉树：**二叉树中所含的  $n$  个结点和满二叉树中编号为1至 $n$ 的结点一一对应。

完全二叉树



特点：

- ①只有最后两层可能有叶子结点
- ②最多只有一个度为1的结点
- ③按层序从1开始编号，结点 $i$ 的左孩子为 $2i$ ，右孩子为 $2i+1$ ；结点 $i$ 的父节点为 $\lfloor i/2 \rfloor$
- ④  $i \leq \lfloor n/2 \rfloor$ 为分支结点， $i > \lfloor n/2 \rfloor$ 为叶子结点

**结论：**满二叉树一定是完全二叉树，反之不一定



## 6.2.2 二叉树的性质

**性质3** 具有 $n$ 个结点的**完全二叉树的深度**为 $\lfloor \log_2 n \rfloor + 1$

**证明：** 设所求完全二叉树的深度为 $k$

由性质2和完全二叉树的定义知：

$k-1$ 层的最多结点数  $\longleftarrow 2^{k-1}-1 < n \leq 2^k-1 \longrightarrow k$ 层的最多结点数

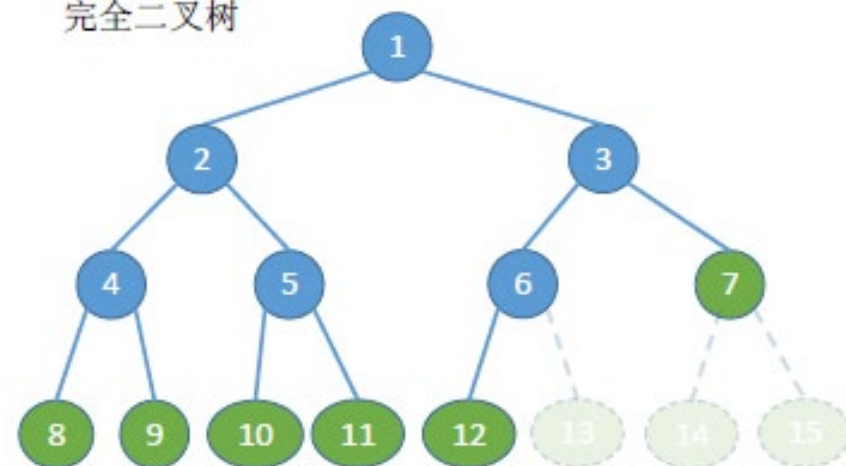
由此可以推出： $2^{k-1} \leq n < 2^k$

取对数得： $k-1 \leq \log_2 n < k$

由于 $k$ 为整数，故有 $k-1 = \lfloor \log_2 n \rfloor$

即： $k = \lfloor \log_2 n \rfloor + 1$

完全二叉树



**性质2：**深度为 $k$ 的二叉树最多有 $2^k-1$ 个结点



## 6.2.2 二叉树的性质

**性质4** 对任意二叉树T，如果度数为0结点数为 $n_0$ ，度数为1结点数为 $n_1$ ，度数为2结点数为 $n_2$ ，则 $n_0=n_2+1$ 。

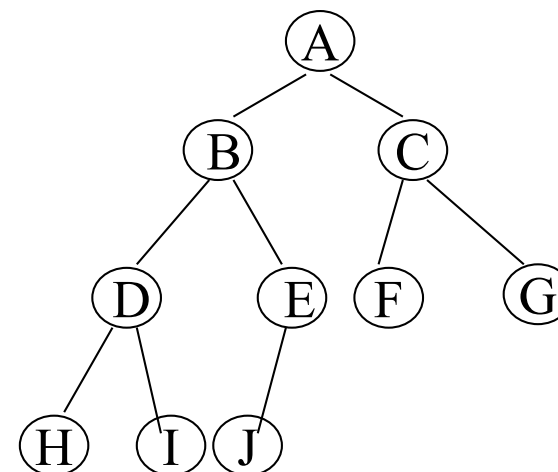
证明：二叉树T的**结点总数**  $n=n_0+n_1+n_2$  (1)

设二叉树中的**分支总数**为**b**，有

射入分支： $b=n-1$  (2)

射出分支： $b=n_1+2*n_2$  (3)

由(1) (2) (3)得 求得： $n_0=n_2+1$

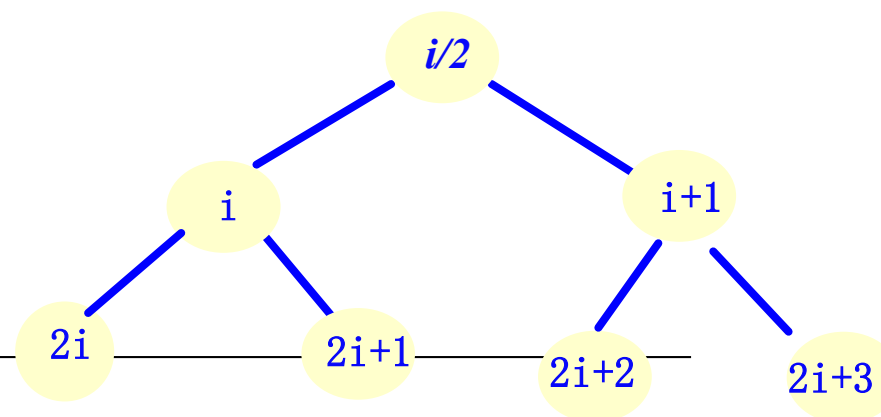




## 6.2.2 二叉树的性质

**性质5:** 若对含  $n$  个结点的完全二叉树从上到下且从左至右进行 1 至  $n$  的编号，则对完全二叉树中任意一个编号为  $i$  的结点：

- (1) 若  $i=1$ ，则该结点是二叉树的根，无双亲，否则，编号为  $\lfloor i/2 \rfloor$  的结点为其双亲结点；
- (2) 若  $2i > n$ ，则该结点无左孩子，否则，编号为  $2i$  的结点为其左孩子结点；
- (3) 若  $2i+1 > n$ ，则该结点无右孩子结点，否则，编号为  $2i+1$  的结点为其右孩子结点。







## 6.2.2 二叉树的性质

**性质5:** 若对含  $n$  个结点的完全二叉树从上到下且从左至右进行 1 至  $n$  的编号，则对完全二叉树中任意一个编号为  $i$  的结点：

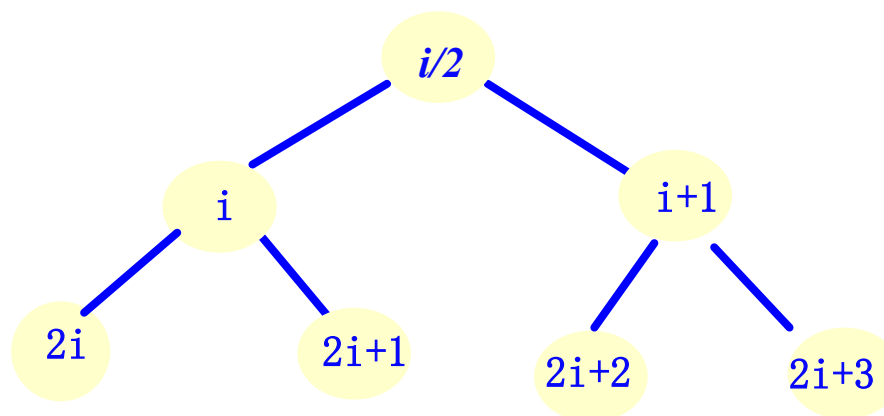
1. 当  $i=1$  时，无双亲，左孩子为  $2$  ( $2i$ )，右孩子为  $3$  ( $2i+1$ )

2. 当  $i>1$  时：

2.1 若  $i$  为第  $j$  层的第一个元素， $i=2^{j-1}$

2.2 若  $i$  为第  $j$  层的第二个元素， $i=i+1$

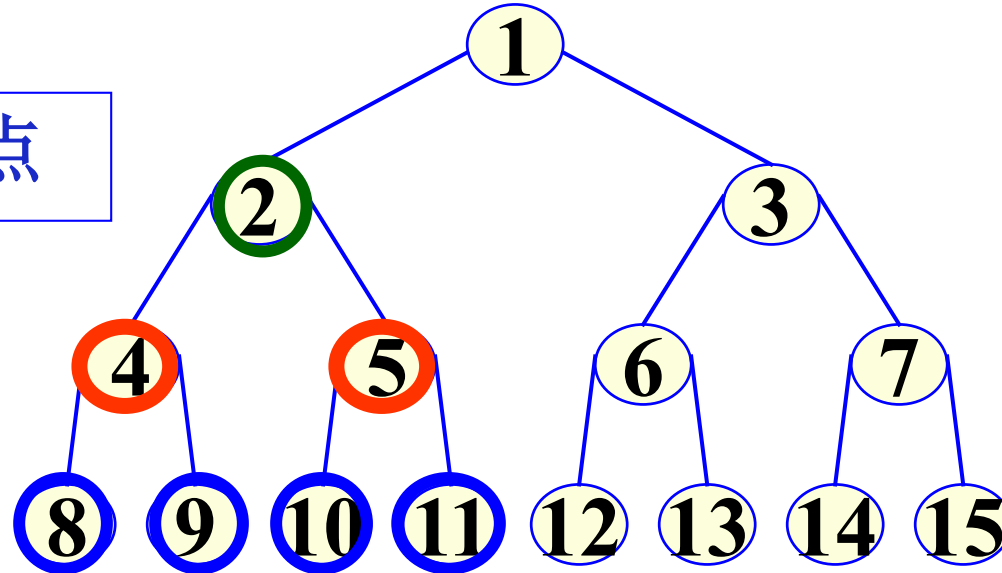
以此类推。。。





## 6.2.2 二叉树的性质

$i=1$  只有根结点



编号  $i=4$   
双亲为  $\lfloor i/2 \rfloor = 2$   
左子树为  $2i=8$   
右子树为  $2i+1=9$

编号  $i=5$   
双亲为  $\lfloor i/2 \rfloor = 2$   
左子树为  $2i=10$   
右子树为  $2i+1=11$

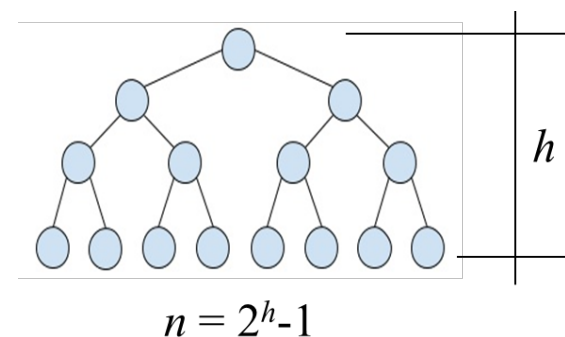
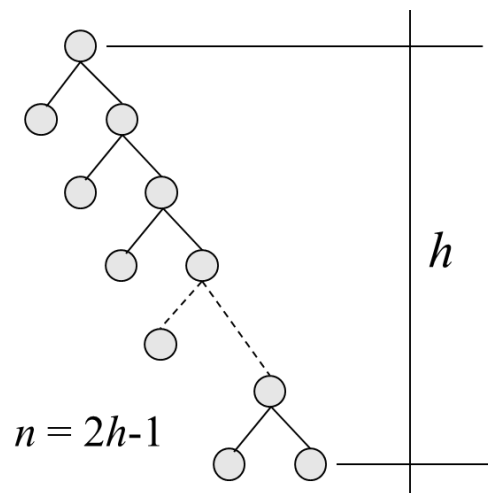
$i=8, n=15$   
 $2i > n$   
无左子树

通过性质5把非线性结构转化成了线性结构



## 6.2.2 二叉树的性质

【练习1】 设高为 $h$ 的二叉树只有度为0和度为2的结点，则此类二叉树的结点数至少为\_\_\_\_，至多为\_\_\_\_。



一棵有124个叶子结点 ( $n_0$ ) 的完全二叉树,  
最多有\_\_[\[填空1\]](#)\_\_个结点 ( $n$ ) ?



## 6.2.2 二叉树的性质

【练习2】一棵有124个叶子结点 ( $n_0$ ) 的完全二叉树，  
最多有 ? 个结点 ( $n$ ) ?

因为:  $n_0 = n_2 + 1$  (性质4)  
 $n = n_0 + n_1 + n_2$  (结点总数)

所以有:  $n = n_1 + 2n_0 - 1$

在完全二叉树中，

$n_1$  不是 0 就是 1

只有  $n_1 = 1$  时， $n$  取最大值为  $2n_0$



## 6.2.2 二叉树的性质

【练习3】证明任一棵满二叉树T中的分支数  $B$  满足：

$$B = 2(n_0 - 1) \quad , \quad \text{其中 } n_0 \text{ 为叶子结点数。}$$

证明：

满二叉树中不存在度为1的节点，设度为2的结点数为 $n_2$

则：  $n = n_0 + n_2$

又：  $n = B + 1$

所以有：  $B = n_0 + n_2 - 1$  ， 而  $n_0 = n_2 + 1, n_2 = n_0 - 1$

$$B = n_0 + n_0 - 1 - 1 = 2(n_0 - 1)$$



## 6.2.2 二叉树的性质

【练习4】具有  $n$  个结点的满二叉树，其叶子结点的个数为多少？

方法1：结点总数： $n=n_0+n_1+n_2$ ;  
但对满二叉树，除有 $n_0=n_2+1$ 外，还有 $n_1=0$ ;  
故有： $n=n_0+n_0-1$   
 $n_0=(n+1)/2$

方法2：设满二叉树的高度为  $h$ ;  
则根据二叉树的性质，叶子结点数为 $2^{h-1}$ ;  
二叉树总结点数 $n=2^h-1$ ;  
可导出： $2^{h-1}=(n+1)/2$ ;

【思考】 $n$ 个结点的完全二叉树，其叶子结点的个数为多少？



## 6.2.3 二叉树的存储结构

- 二叉树的顺序存储表示
- 二叉树的链式存储表示







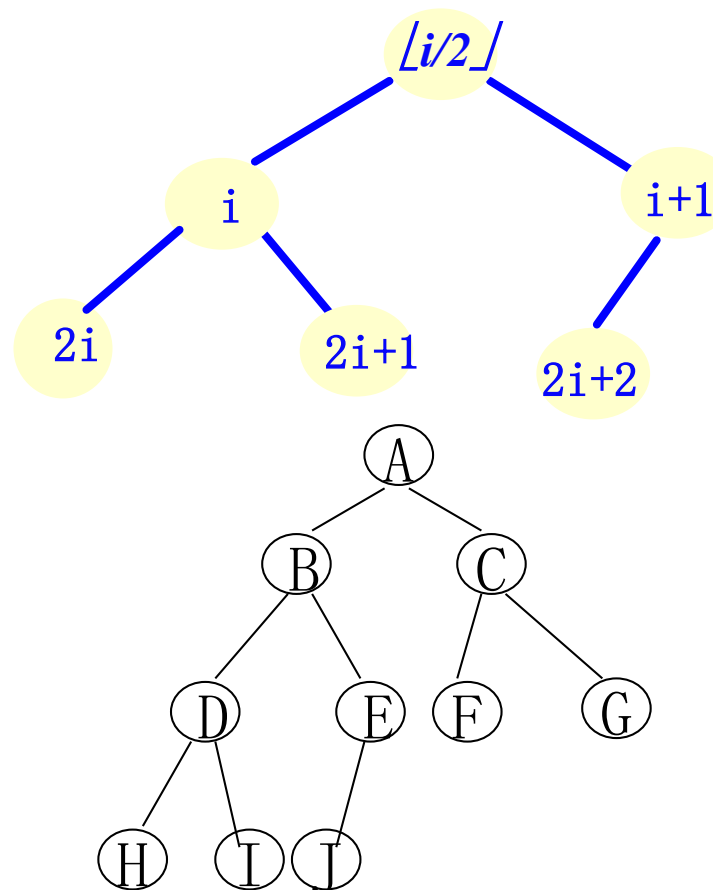
## 6.2.3 二叉树的存储结构

- 二叉树的顺序存储表示

(1) 完全（或满）二叉树

采用一维数组，按层次顺序依次存储二叉树的每一个结点。  
如下图所示：

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



利用性质5实现线性结构和非线性结构的灵活转换。

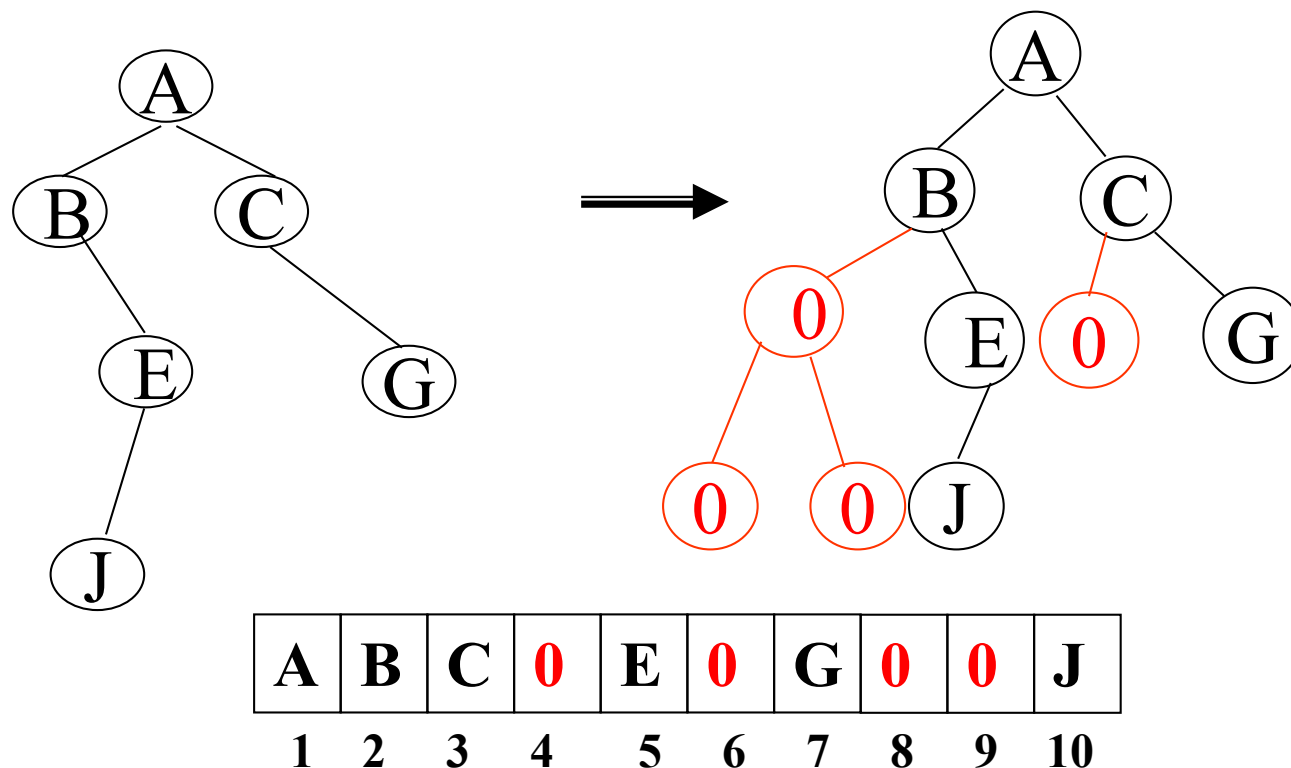


## 6.2.3 二叉树的存储结构

- 二叉树的顺序存储表示

- (2) 一般二叉树

通过虚设部分结点，使其变成相应的完全二叉树。

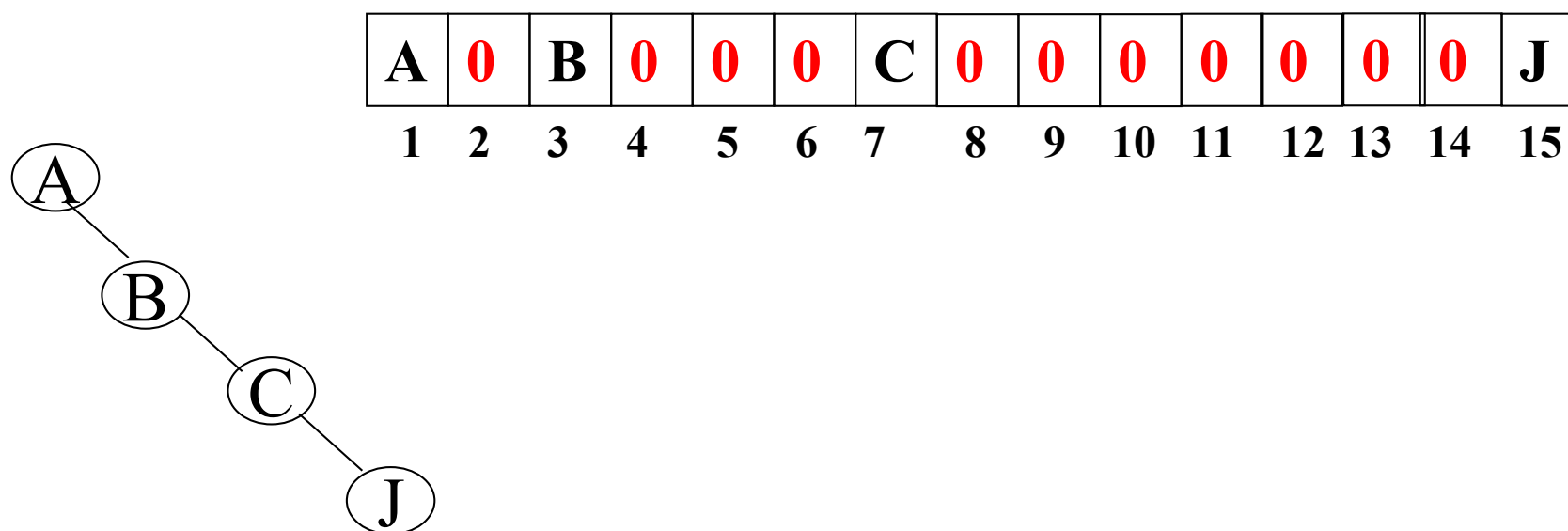




## 6.2.3 二叉树的存储结构

- 二叉树的顺序存储表示

### (3) 特殊的二叉树



**说明：**顺序存储方式对于畸形二叉树，浪费较大空间



## 6.2.3 二叉树的存储结构

- 二叉树的链式存储表示

二叉链表存储:

二叉链表中每个结点包含三个域: 数据域、左指针域、右指针域

lch	data	rch
-----	------	-----

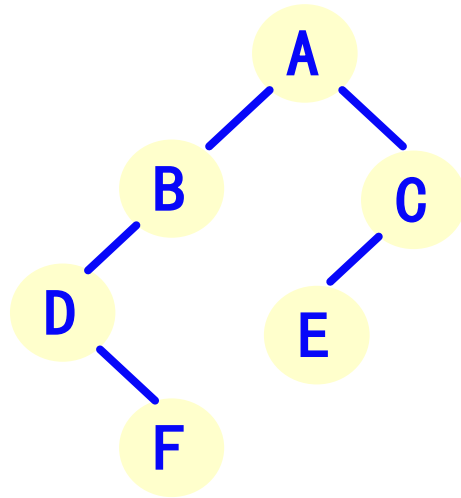
C 语言的类型描述如下:

```
typedef Struct BinTNode
{
    DataType data;
    Struct BinTNode *lch, *rch;
} BinTNode, *BinTree;
```

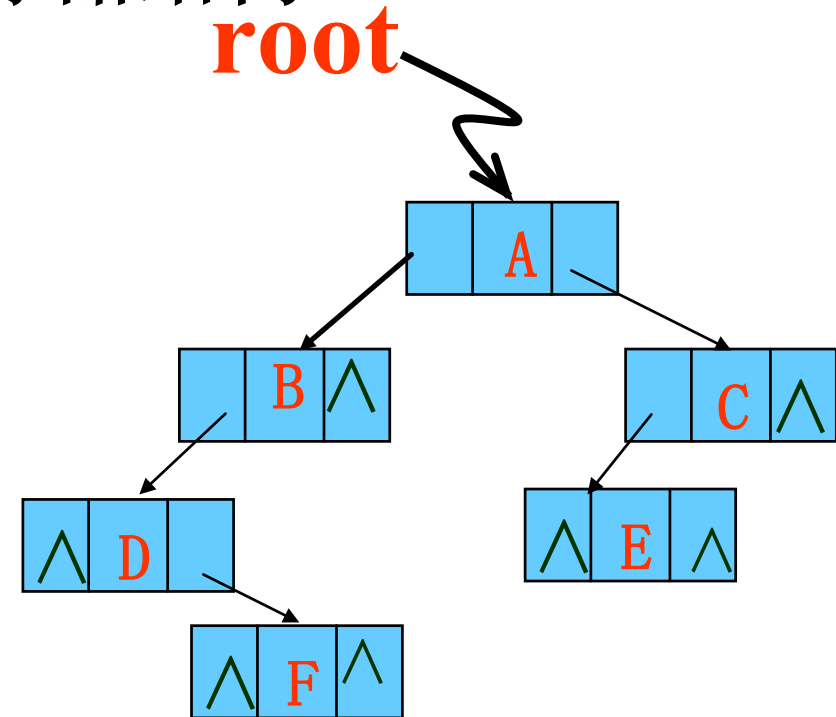


## 6.2.3 二叉树的存储结构

- 二叉树的链式存储表示



二叉树



二叉链表图示

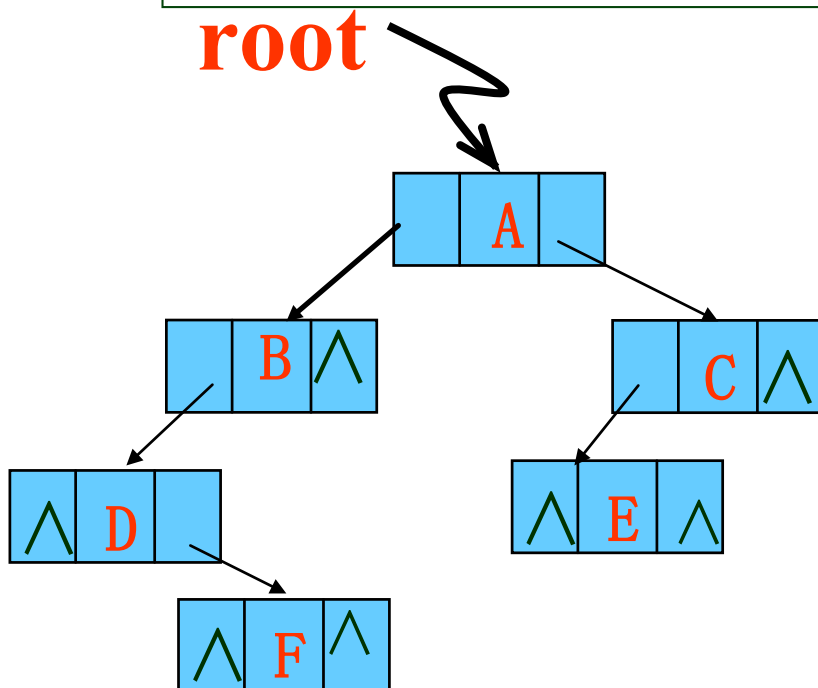
n 个结点的二叉树中，  
有多少个空链接域？



## 6.2.3 二叉树的存储结构

- 二叉树的链式存储表示

**性质6:**  $n$  个结点的二叉树中, 共有  $n+1$  个空指针域。



二叉链表图示

证:  $n$  个结点总的指针域数  $2n$

除了根结点外, 其余  $n-1$  个结点

都是由指针域指出的结点;

所以, 剩余的结点数即

**空指针域个数为:**

$$2n - (n-1) = n+1$$



## 6.2.3 二叉树的存储结构

二叉链表的缺点是很难找到结点的双亲

- 二叉树的链式存储表示---三叉链表

三叉链表（带双亲指针的二叉链表）：三叉链表中每个结点包含四个域：数据域、左指针域、右指针域、双亲指针域

结点结构：

lch	data	rch	parent
-----	------	-----	--------

C 语言的类型描述如下：

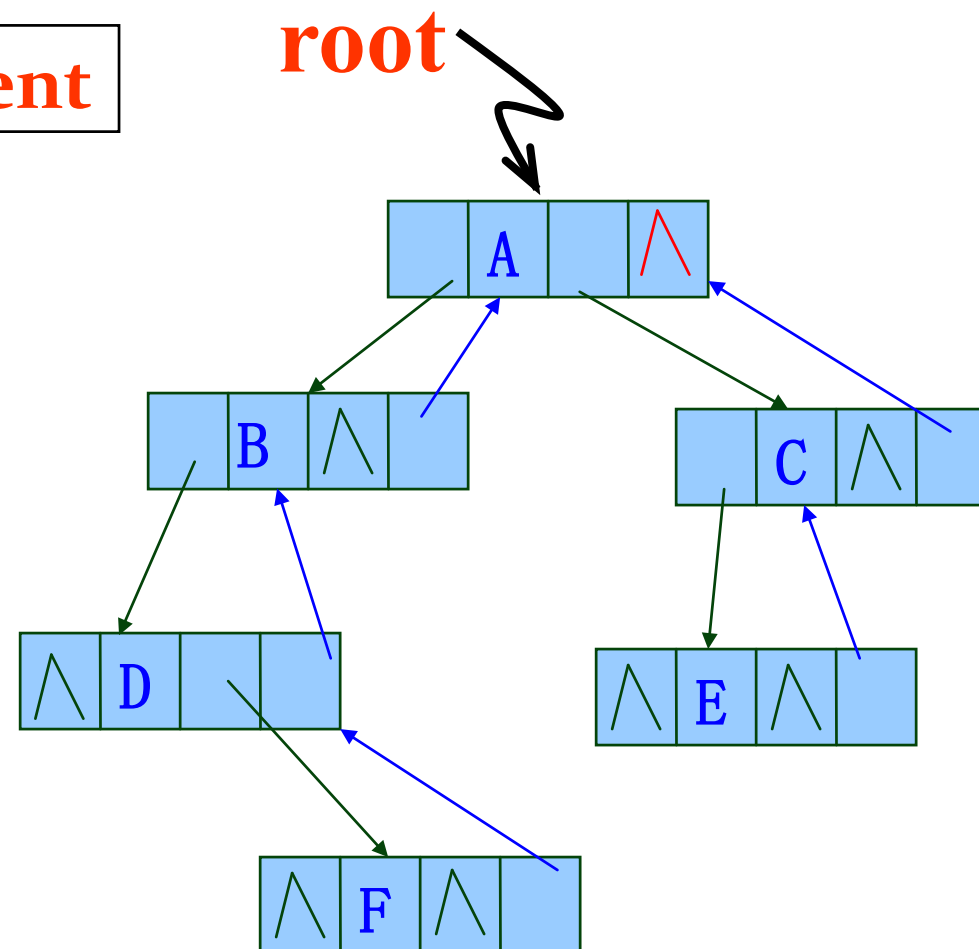
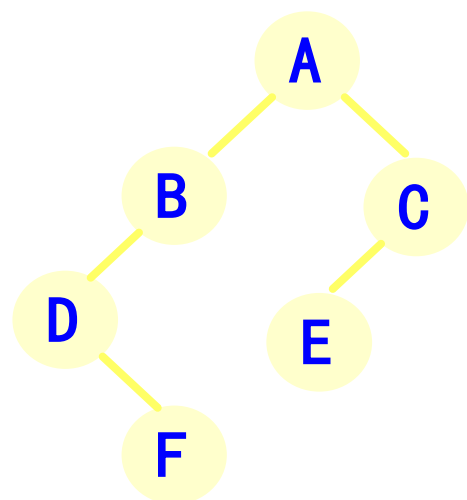
```
typedef Struct node
{
    DataType data;
    Struct node *lch, *rch, *parent;
} *BinTree;
```



## 6.2.3 二叉树的存储结构

- 二叉树的链式存储表示---**三叉链表**

lch	data	rch	parent
-----	------	-----	--------







## 6.3 二叉树的遍历

6.3.1 二叉树的遍历方法

6.3.2 遍历的递归算法

6.3.3 遍历的非递归算法





## 6.3.1 二叉树的遍历方法

- **遍历**:
  - 按某种搜索路径**访问**二叉树的每个结点，而且每个结点仅被访问一次。
- **访问**:
  - 访问是指对结点进行各种操作的简称，包括输出、查找、修改等等操作。
- **遍历**是各种数据结构最基本的操作，许多其它的操作可以在遍历基础上实现。



## 6.3.1 二叉树的遍历方法

- “遍历” 是任何类型均有的操作：
  - 线性结构的遍历：只有一条搜索路径 (因为每个结点均只有一个后继)；
  - 非线性结构的遍历：二叉树是非线性结构，则存在如何遍历；即按什么样的搜索路径遍历的问题。

如何访问二叉树的每个结点，  
而且每个结点仅被访问一次？





## 6.3.1 二叉树的遍历方法

- 对“二叉树”而言，可以有2种遍历方式：
  - 按层次遍历；
    - 从左往右、从上到下
    - 实现：顺序存储、链式存储



## 6.3.1 二叉树的遍历方法

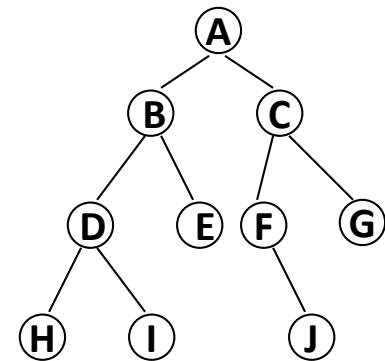
- 层序遍历的实现

按层遍历引入了**队列**作为辅助工具。

算法思想为：

- (1) 将二叉树根入队列；
- (2) 将队头元素出队列，并判断此元素是否有左右孩子，若有，则将其左右孩子入列，否则转 (3)；
- (3) 重复步骤 (2)，直到队列为空。

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





## 6.3.1 二叉树的遍历方法

- 按层遍历

```
Struct node {  
    Struct node *lchild;  
    Struct node *rchild;  
    datatype data;  
};  
Typedef struct node * BTREE;
```

```
Struct QUEUE {  
    Struct node data[maxlength];  
    int front;  
    int rear;  
};
```

```
Void LeverList(BTREE T)  
{  
    QUEUE Q;  
    BTREE p=T;  
    MakeNull(Q);  
    if(T) { EnQueue(p, Q);  
        while(!Empty(Q))  
        { p=DeQueue(Q);  
            visit(p->data);  
            if(p->lchild)  
                EnQueue(p->lchild);  
            if(p->rchild)  
                EnQueue(p->rchild);  
        }  
    }  
}
```



## 6.3.1 二叉树的遍历方法

- 对“二叉树”而言，可以有2种遍历方式：
  - 按层次遍历；
    - 从左往右、从上到下
  - 按左子树、右子树、根三个基本单元遍历
    - 有几种顺序？



## 6.3.1 二叉树的遍历方法

二叉树由根、左子树、右子树三部分组成

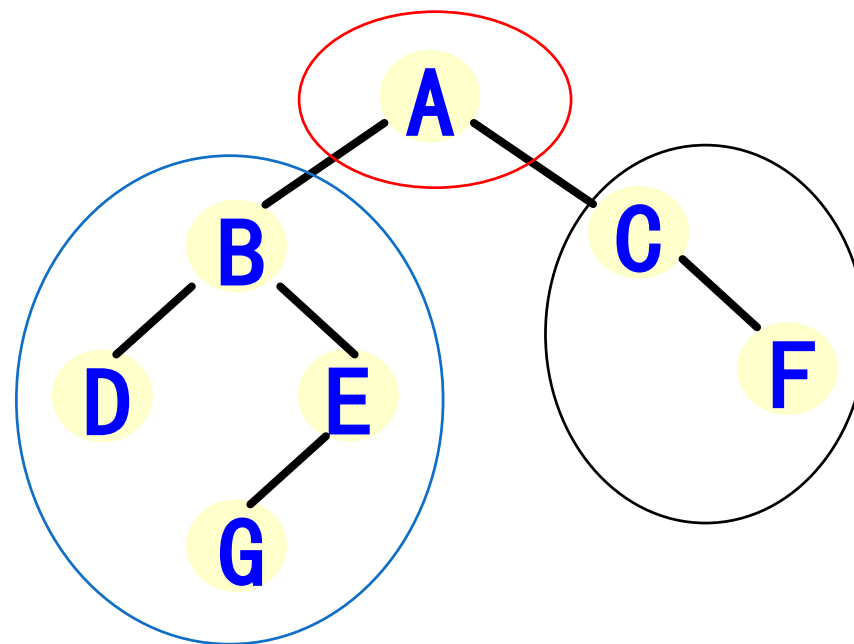
二叉树的遍历可以分解为：访问根，遍历左子树和遍历右子树

令：L：遍历左子树  
T：访问根结点  
R：遍历右子树

有六种遍历方法：

T L R, L T R, L R T,

T R L, R T L, R L T



约定先左后右, 有三种遍历方法：T L R、L T R、L R T，分别称为先序（先根）遍历、中序（中根）遍历、后序（后根）遍历



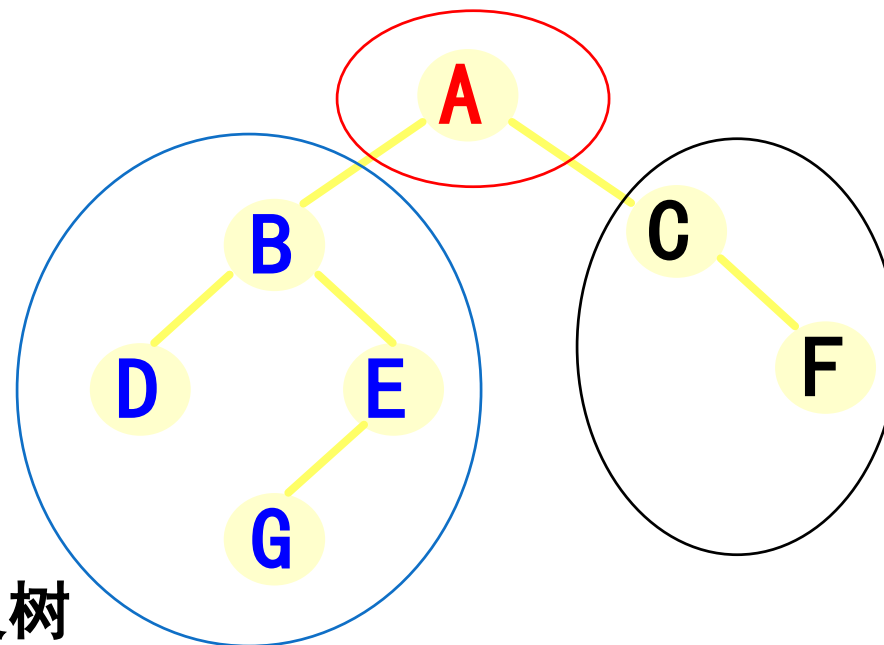


## 6.3.1 二叉树的遍历方法

- 中序遍历 ( L T R )

若二叉树非空

- (1) 中序遍历左子树
- (2) 访问根结点
- (3) 中序遍历右子树



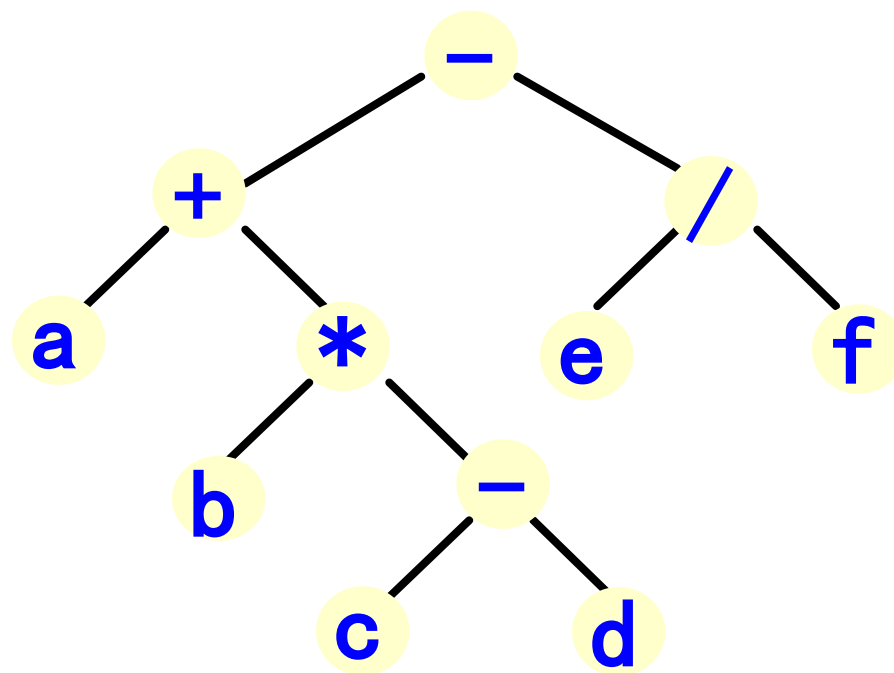
**例** 中序遍历右图所示的二叉树

中序遍历序列: D ,B ,G ,E ,A ,C ,F



## 6.3.1 二叉树的遍历方法

**练习** 表达式  $a+b*(c-d)-e/f$  用二叉树表示如下，求中序遍历序列：



中序  $a+b*c-d-e/f$       --- 中缀表示

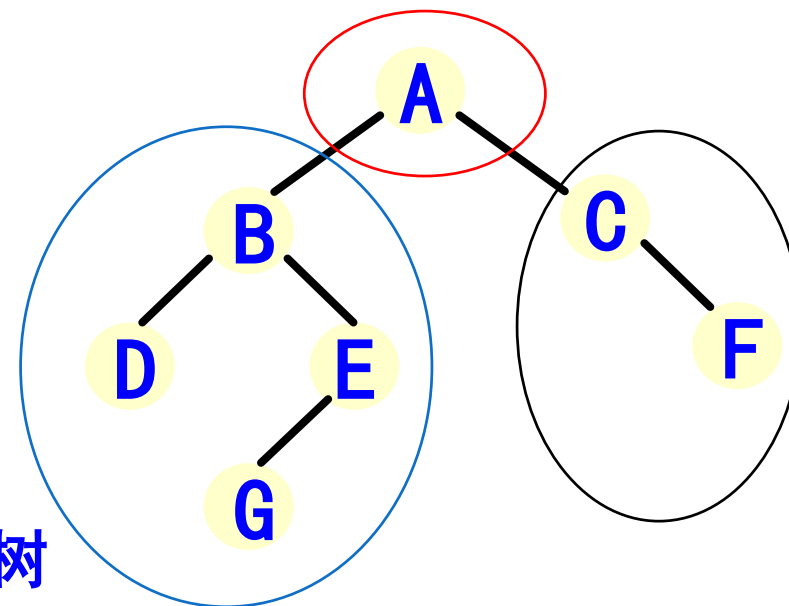


## 6.3.1 二叉树的遍历方法

- 先序遍历 (T L R)

若二叉树非空

- (1) 访问根结点;
- (2) 先序遍历左子树;
- (3) 先序遍历右子树;



**例** 先序遍历右图所示的二叉树

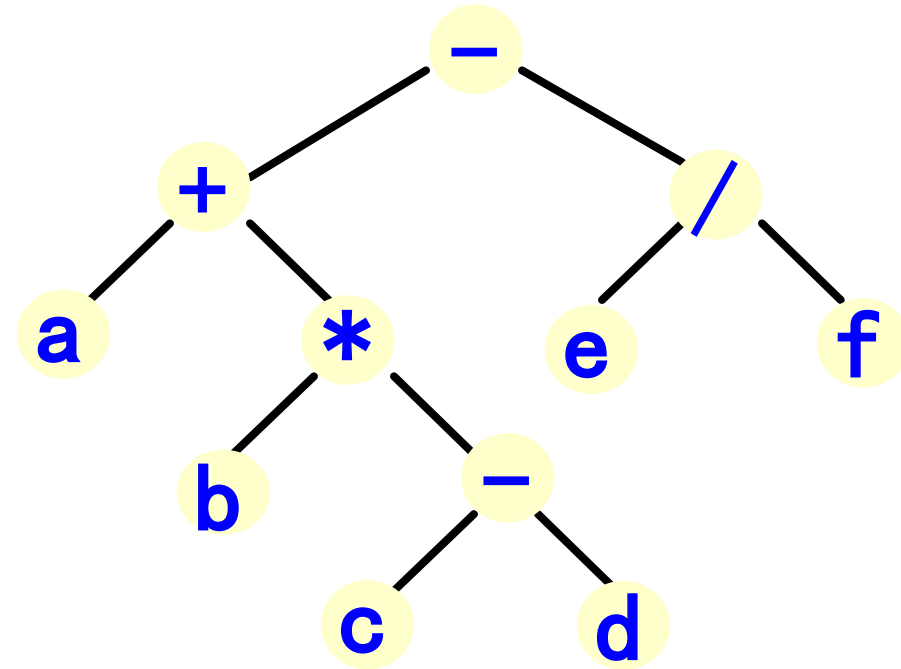
- (1) 访问根结点A
- (2) 先序遍历左子树：即按 T L R 的顺序遍历左子树
- (3) 先序遍历右子树：即按 T L R 的顺序遍历右子树

先序遍历序列：A, B, D, E, G, C, F



## 6.3.1 二叉树的遍历方法

**练习** 先序遍历下图所示的二叉树



先序 - + a \* b - c d / e f

前缀表示 (波兰式)

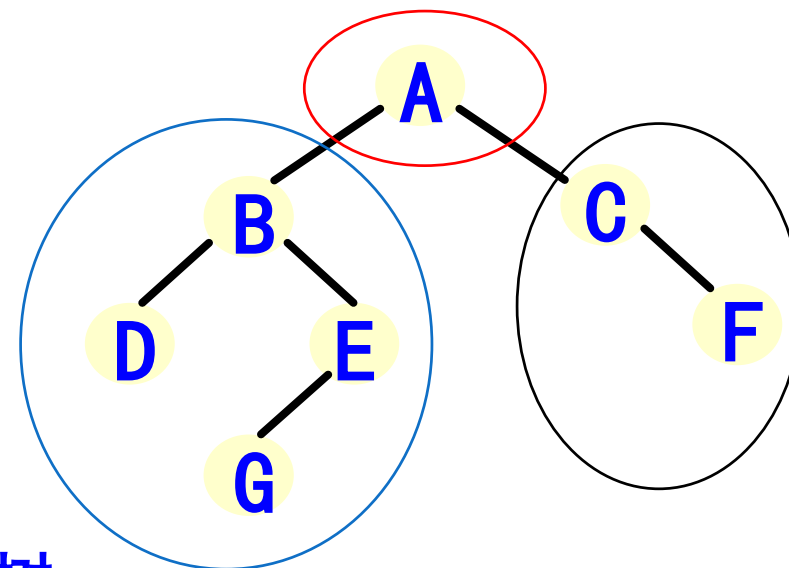


## 6.3.1 二叉树的遍历方法

- 后序遍历 ( L T R )

若二叉树非空

- (1) 后序遍历左子树
- (2) 后序遍历右子树
- (3) 访问根结点



**例** 后序遍历右图所示的二叉树

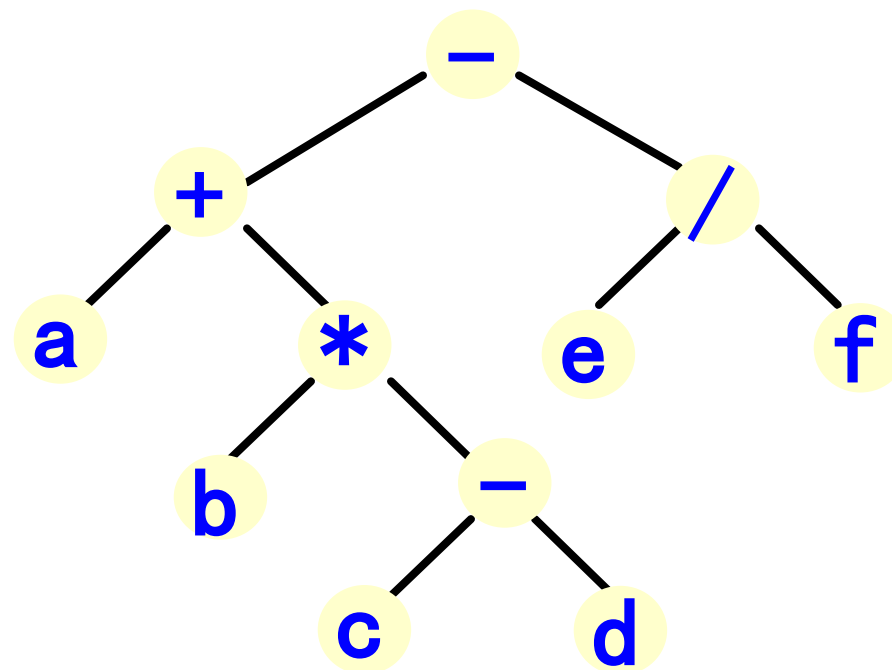
- (1) 后序遍历左子树：即按 L R T 的顺序遍历左子树
- (2) 后序遍历右子树：即按 L R T 的顺序遍历右子树
- (3) 访问根结点A

后序遍历序列： D, G, E, B, F, C, A



## 6.3.1 二叉树的遍历方法

**练习** 后序遍历下图所示的二叉树



后序 a b c d - \* + e f / -

后缀表示（逆波兰式）



## 6.3 二叉树的遍历

### 6.3.1 二叉树的遍历方法

### 6.3.2 遍历的递归算法

### 6.3.3 遍历的非递归算法





## 6.3.2 遍历的递归算法

- 先序遍历 (TLR) 的定义

若二叉树非空

- (1) 访问根结点;
- (2) 先序遍历左子树
- (3) 先序遍历右子树;

上面先序遍历的定义等价于:

若二叉树为空, 结束 —— 基本项 (也叫终止项)

若二叉树非空 —— 递归项

- (1) 访问根结点;
- (2) 先序遍历左子树;
- (3) 先序遍历右子树;





## 6.3.2 遍历的递归算法

- 先序遍历递归算法

```
void prev (BinTree T)
```

```
{ if (T)
```

```
{ visit(T->data);
```

```
prev(T->lch);
```

```
prev(T->rch);
```

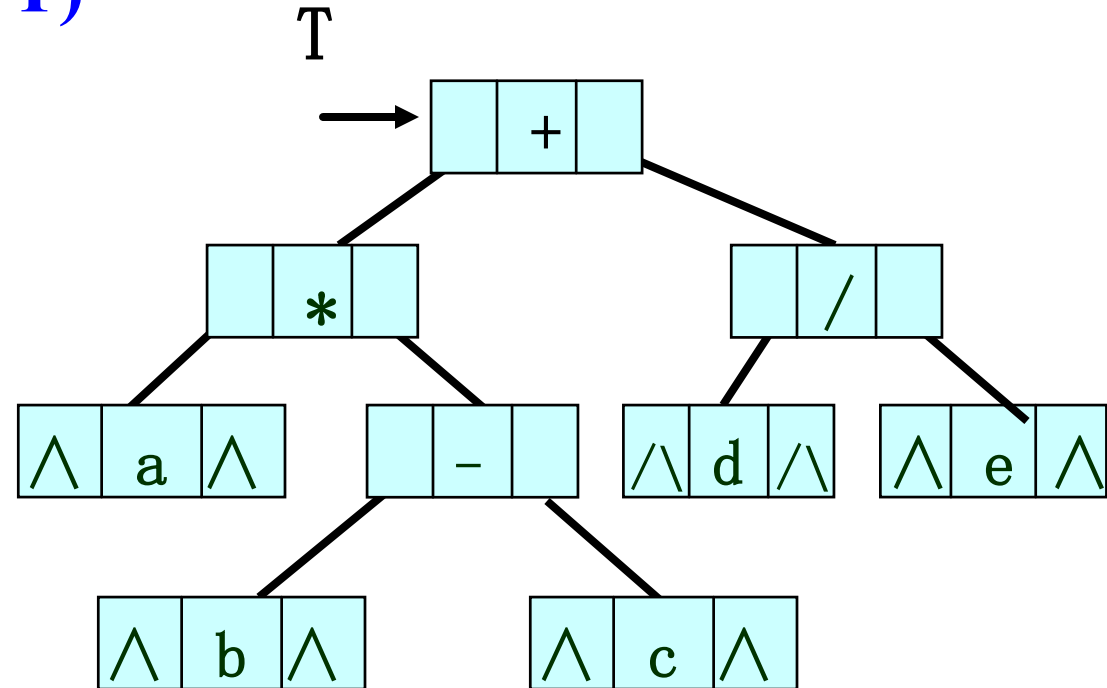
```
}
```

```
}
```

先序序列为

$+ * a - b c / d e$

称为前缀表达式



$$a * (b - c) + d / e$$



## 6.3.2 遍历的递归算法

- 中序遍历递归算法

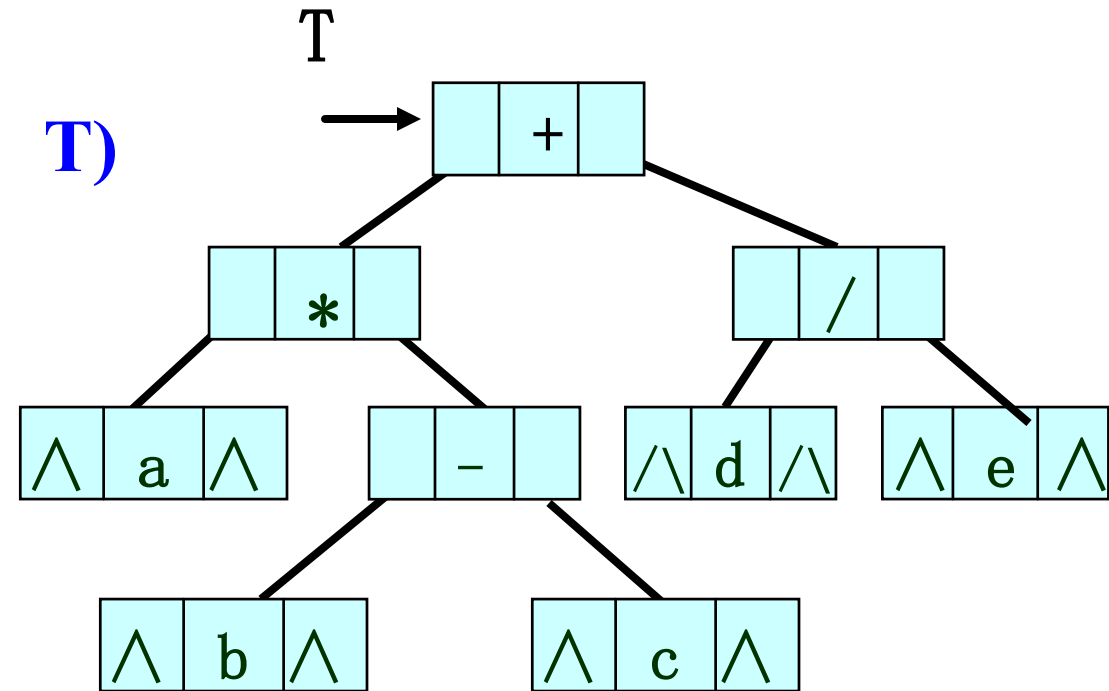
```

void Mid (BinTree T)
{ if (T)
  {Mid(T->lch);
   visit( T->data);
   Mid(T->rch);
  }
}
  
```

中序序列为

$a * b - c + d / e$

称为中缀表达式



$a * (b - c) + d / e$

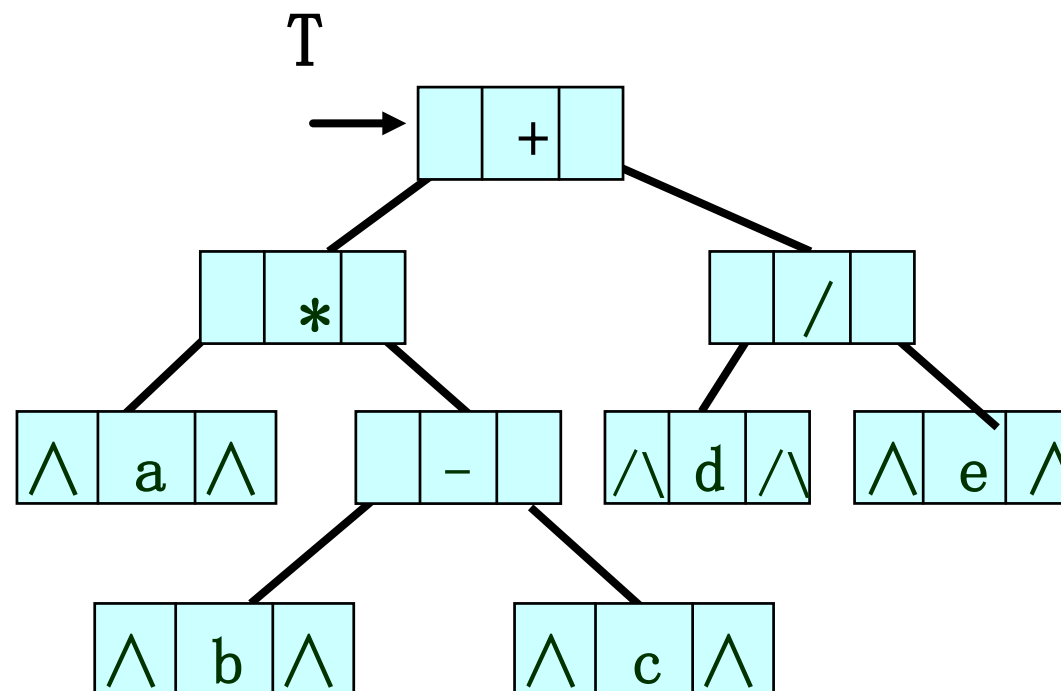


## 6.3.2 遍历的递归算法

- 后序遍历递归算法

```
void Post(BinTree T)
{ if (T)
  { Post(T->lch);
    Post(T->rch);
    visit( T->data);
  }
}
```

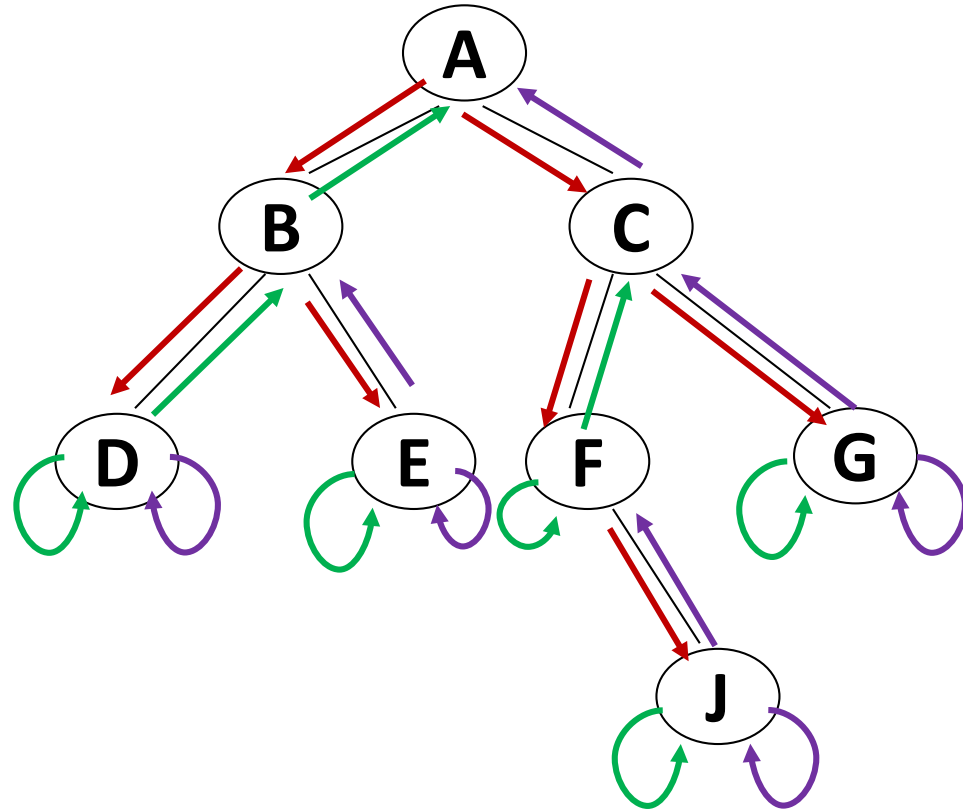
后序序列为 **a b c - \* d e / +**  
称为**后缀表达式**



$a * (b - c) + d / e$



## 6.3.1 二叉树的遍历方法



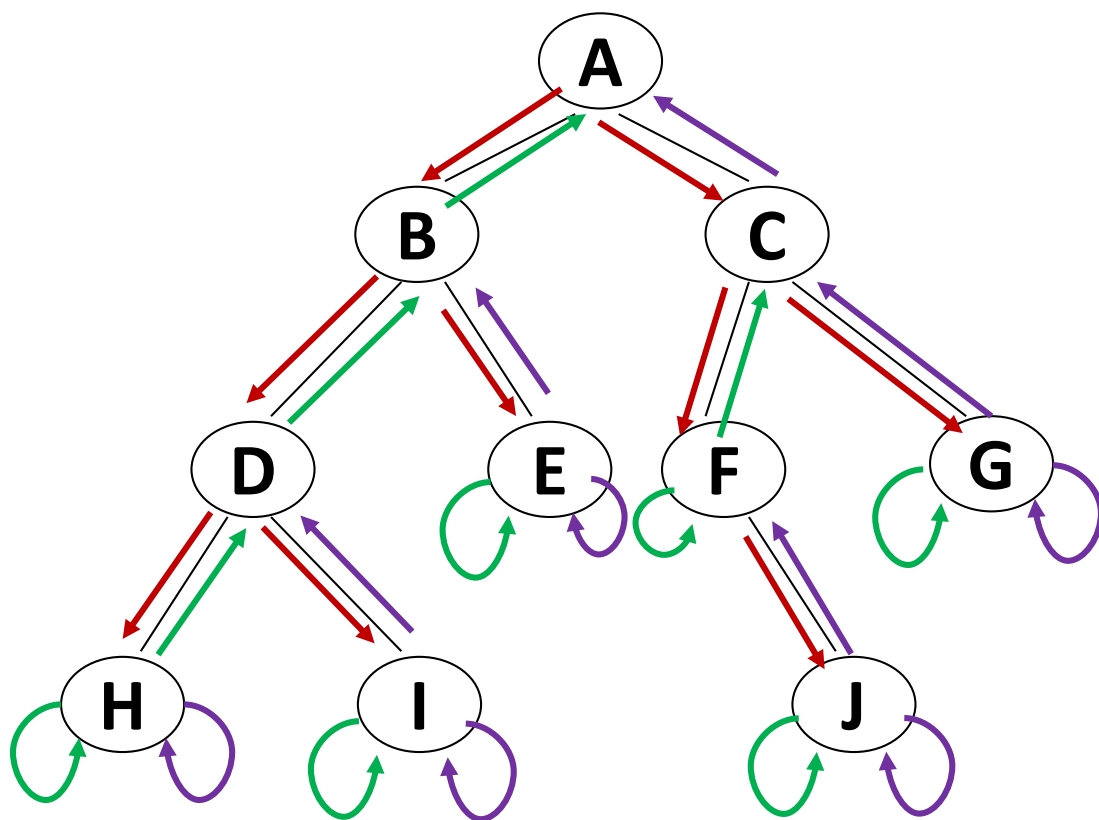
```
1 void prev (BinTree T)
2 { if (T)
3   { visit(T->data);
4     prev(T->lch);
5     prev(T->rch);
6   }
7 }
```






递归工作栈



## 6.3.1 二叉树的遍历方法

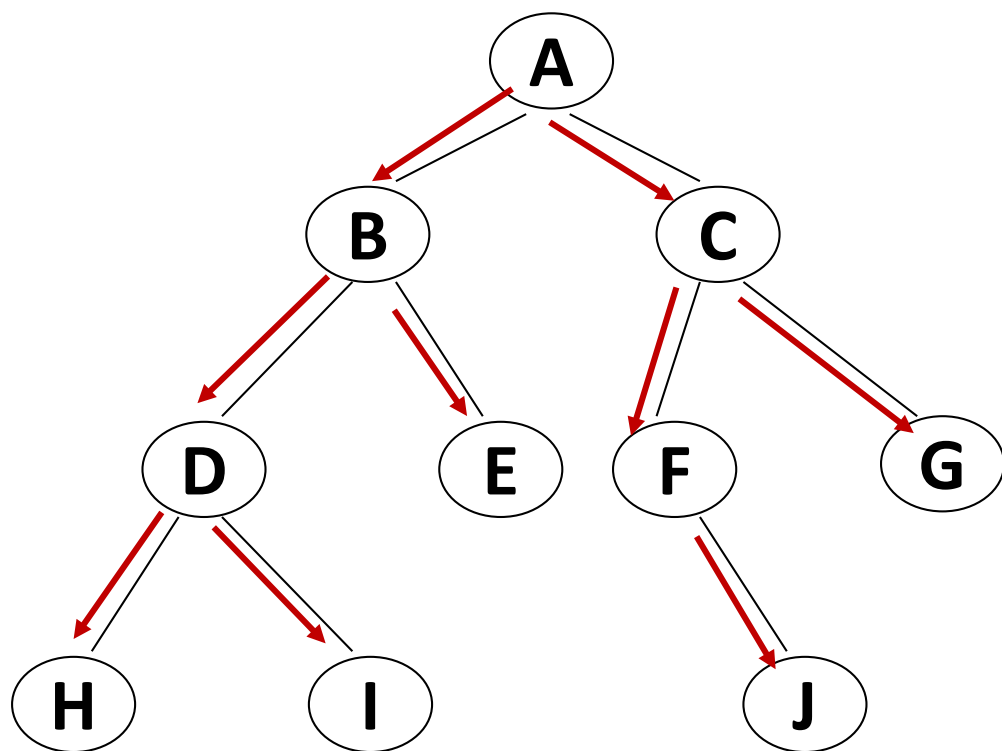


脑补空结点，从根节点出发，画一条路：  
如果左边还有没走的路，优先往左边走  
走到路的尽头（空结点）就往回走  
如果左边没路了，就往右边走  
如果左、右都没路了，则往上面走




第一次路过   
第二次路过   
第三次路过 



## 6.3.1 二叉树的遍历方法



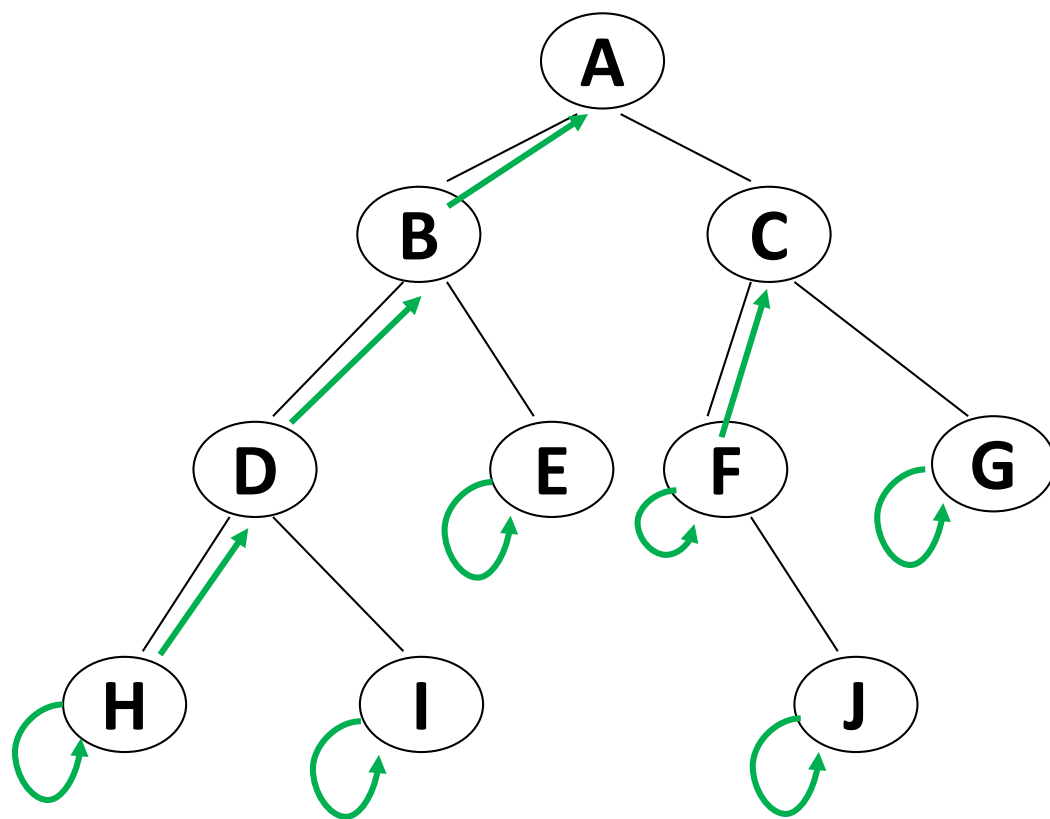
脑补空结点，从根节点出发，画一条路：  
如果左边还有没走的路，优先往左边走  
走到路的尽头（空结点）就往回走  
如果左边没路了，就往右边走  
如果左、右都没路了，则往上面走

第一次路过   
第二次路过   
第三次路过 




先序遍历——第一次路过时访问结点



## 6.3.1 二叉树的遍历方法



脑补空结点，从根节点出发，画一条路：  
如果左边还有没走的路，优先往左边走  
走到路的尽头（空结点）就往回走  
如果左边没路了，就往右边走  
如果左、右都没路了，则往上面走

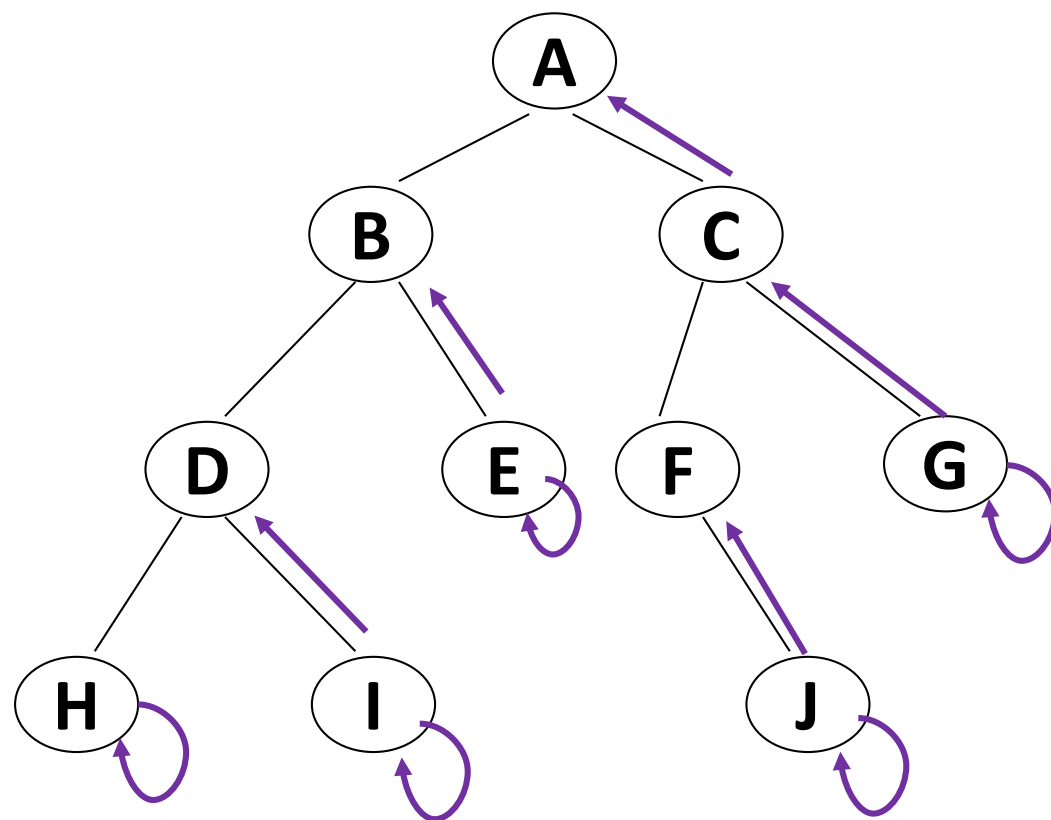
第一次路过   
第二次路过   
第三次路过 

先序遍历——第一次路过时访问结点




中序遍历——第二次路过时访问结点



## 6.3.1 二叉树的遍历方法



脑补空结点，从根节点出发，画一条路：  
如果左边还有没走的路，优先往左边走  
走到路的尽头（空结点）就往回走  
如果左边没路了，就往右边走  
如果左、右都没路了，则往上面走

第一次路过   
第二次路过   
第三次路过 

先序遍历——第一次路过时访问结点

中序遍历——第二次路过时访问结点

后序遍历——第三次路过时访问结点





## 6.3 二叉树的遍历

6.3.1 二叉树的遍历方法

6.3.2 遍历的递归算法

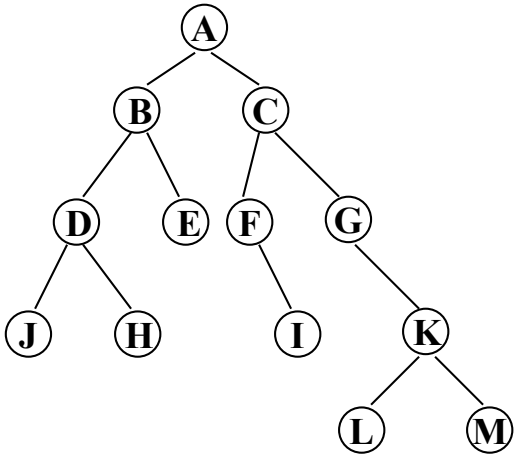
6.3.3 遍历的非递归算法





### 6.3.3 遍历的非递归算法

例



先序: **A B D J H E C F I G K L M**  
中序: **J D H B E A F I C G L K M**  
后序: **J H D E B I F L M K G C A**

No.	指针T	栈	输出
1	A →	#	
2	B →	#A	
3	D →	#AB	
4	J →	#ABD	
5	^	#ABDJ →	J
6	^	#ABD →	D
7	H →	#AB	
8	^	#ABH →	H
9	^	#AB →	B
10	E →	#A	
11	^	#AE →	E
12	^	#A →	A
13	C →	#	
14	F →	#C	
15	^	#CF →	F
16	I →	#C	
17	^	#CI →	I
18	^	#C →	C
19	G →	#	
20	^	#G →	G
21	K →	#	
22	L →	#K	
23	^	#KL →	L
24	^	#K →	K
25	M →	#	
26	^	#M →	M
27	^	#	结束

数据结构:

设栈S:  
用以保留  
当前结点;

算法:  
Loop:  
{  
    if (T 非空)  
        { T进栈;  
          左一步;}  
    else  
        { T指向栈顶元素  
          并退栈;  
          输出;  
          右一步;}  
};



### 6.3.3 遍历的非递归算法

二叉树的中序遍历  
的非递归过程

```
Void NInOrder( BT )
BTREE BT;
{ STACK S ; BTREE T ;
  MakeNull( S ) ;
  T = BT ;
  while ( !IsEmpty( T ) || ! Empty ( S ) )
    if ( !IsEmpty ( T ) )
      { Push( T ,S );
        T = Lchild ( T ) ; }
    else
      { T = TOP ( S ) ; POP ( S ) ;
        visit( Data( T ) ) ;
        T = Rchild ( T ) ; }
  }
```

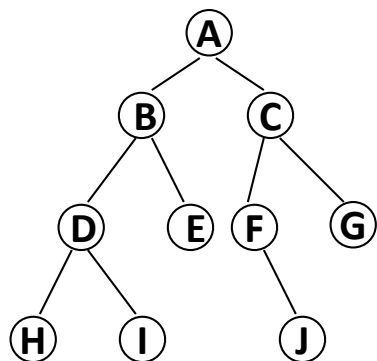
进栈; 左走一步

退栈; 右走一步



## 6.3.3 遍历的非递归算法

先序遍历非递归算法    中序遍历非递归算法    后序遍历非递归算法



```

Loop:
{
  if (T 非空)
  { 输出;
    T进栈;
    左一步;}
  else
  {T指向栈顶元
    素并退栈;
    右一步;}
};
  
```

```

Loop:
{
  if (T 非空)
  { T进栈;
    左一步;}
  else
  {T指向栈顶元
    素并退栈;
    输出;
    右一步;}
};
  
```

```

Loop:
{
  if (T 非空)
  { T进栈;
    左一步;}
  else
  { 当栈顶指针所指结点的
    右子树不存在或已访问,
    T指向栈顶元素并退栈,
    输出;
    否则右一步;}
};
  
```



## 6.3.3 遍历的非递归算法

### 二叉树的三叉链表存储表示

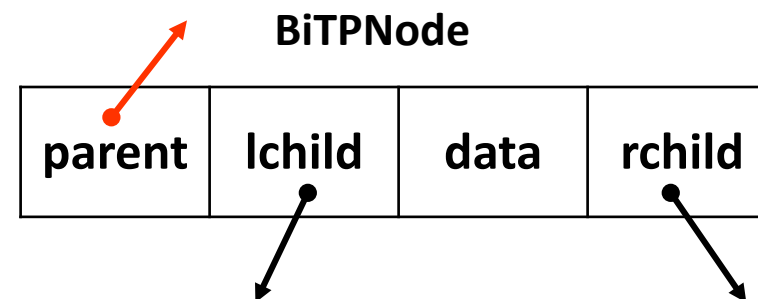
```
typedef struct BiTPNode
```

```
{
```

```
    ElementType data;
```

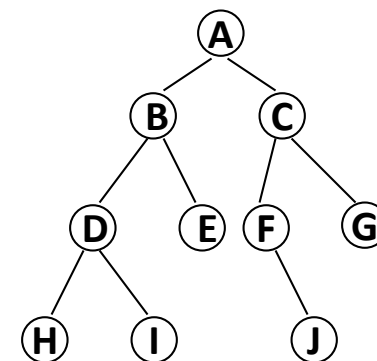
```
    struct BiTPNode *parent, *lchild, *rchild;
```

```
} *BiPTree;
```



很显然:

- ❑ 相对二叉链表表示的二叉树，除了找父结点的操作变得很容易外，其它基本操作没有什么变化。
- ❑ 对二叉树的先序/中序/后序的非递归遍历，不需要再使用栈。





## 不用栈非递归遍历

```
void InOrder(TriTree PT, void (*visit)(TElemType))
```

```
{ TriTree p=PT, pr;
```

```
  while(p)
```

```
  { if (p->lchild) p = p->lchild; //找最左结点
```

```
    else { visit(p->data); //访问最左节点
```

```
      if (p->rchild) p = p->rchild; //若有右子树，找右子树最左结点
```

```
      else { pr = p; //否则返回其父结点
```

```
        p = p->parent;
```

```
        while (p && (p->lchild != pr || !p->rchild))
```

```
        { if (p->lchild == pr) visit(p->data);
```

```
          pr = p; //父结点已被访问，故返回上一级
```

```
          p = p->parent;
```

```
        } if (p){ visit(p->data);
```

```
          p = p->rchild;
```

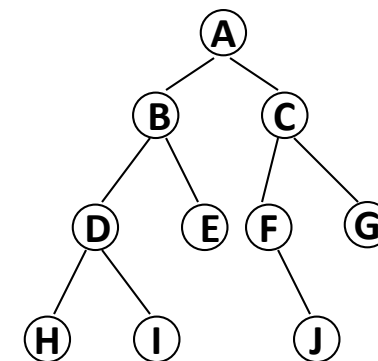
```
        }
```

```
      }
```

```
    }
```

```
  }
```

```
}
```



//该while  
循环沿双亲链  
一直查找，若  
无右孩子则访  
问，直至找到  
第一个有右孩  
子的结点为止  
(但不访问该  
结点，留给下  
步if语句访问)

//若其不  
是从左子树  
回溯来的，  
或左结点的  
父结点并没  
有右孩子

//访问父结点，并转  
到右孩子（经上步  
while处理，可以确定  
此时p有右孩子）