第七章 树形结构

一.树形结构概论

1.与线性结构的差异。

2.地位：

计算机中最重要的非线性结构。

一方面： 计算机中

嵌套结构（菜单、子程序调用等）的组织等树形结构提供了最自然的表示。举例。

另一方面： 解决实际问题中，有效的工具

实际生活中的例子。举例

3.讨论内容

- 从逻辑角度讨论树形结构的基本概念和运算；
- 讨论树形结构的存储及运算的实现；
- 讨论简单应用情况。

4.主要讨论的结构

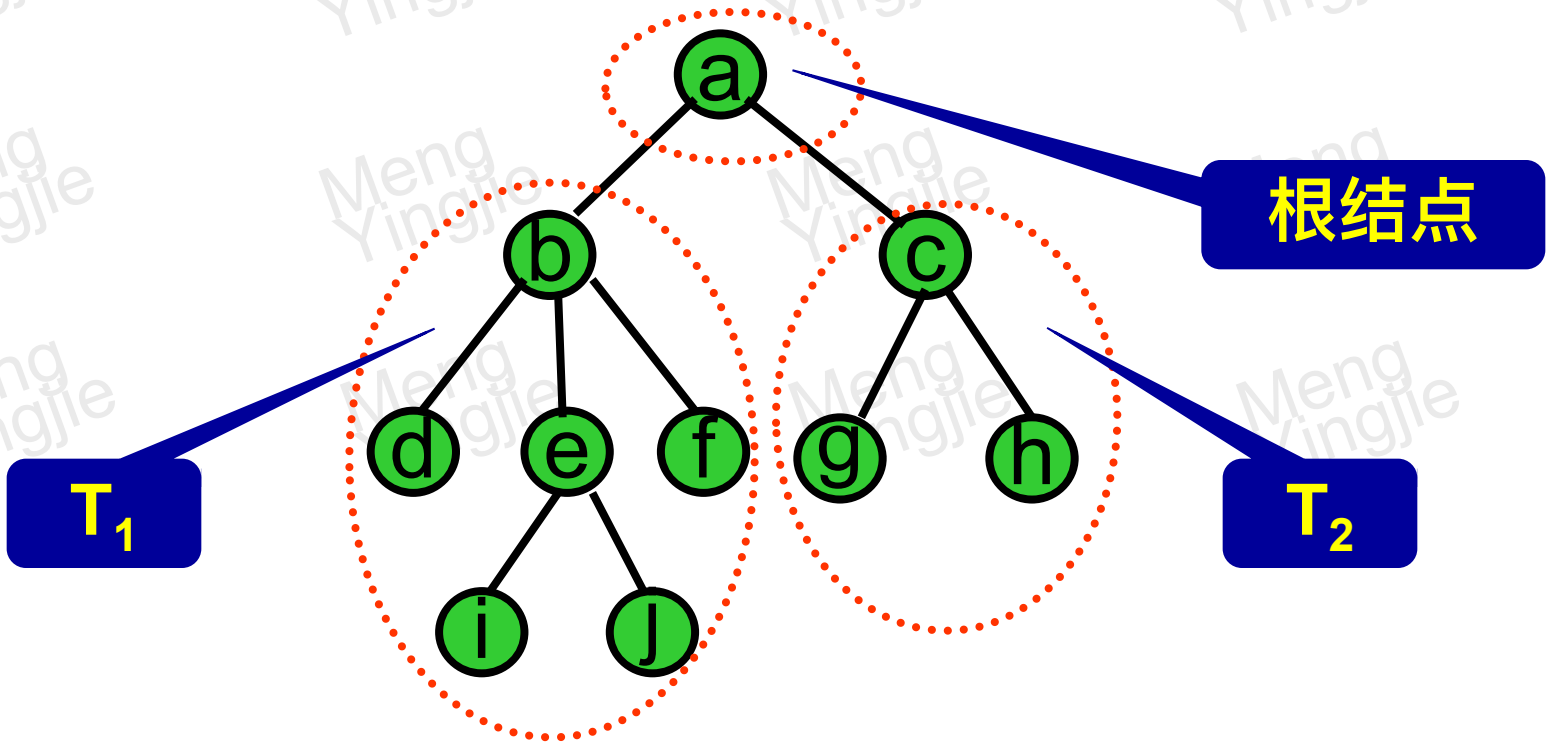
- 树
- 二叉树

二.树的基本定义

树(tree)T，是满足如下性质的有限个结点组成的非空集合：

①T中有且仅有一个称为根的结点；

②除根结点之外，其余结点分成m(m>0)个不相交的集合$T_1,T_2,\ldots,T_m$,其中每个T_i都是树，而且都称为T的子树。



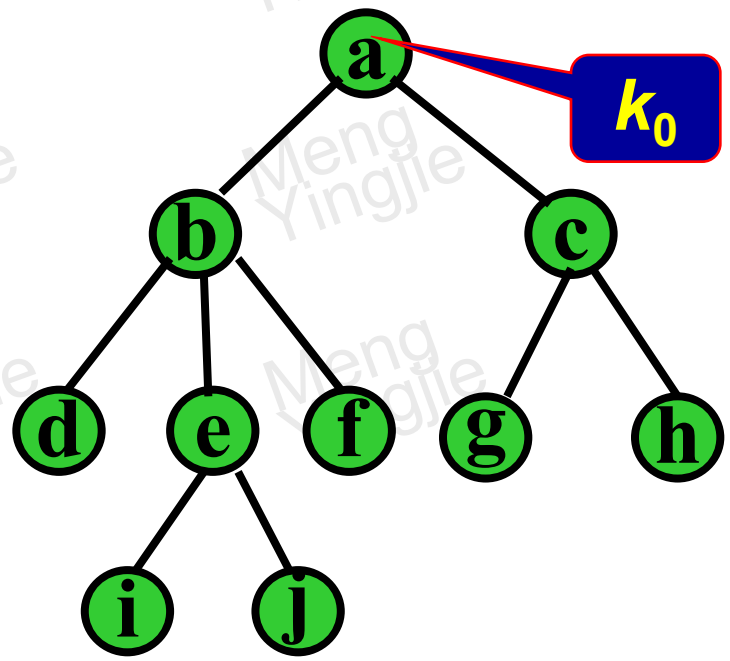
树的定义二:

树是包含n(n>0)个结点的有穷集合K,且在K上定义了一个关系N,关系N满足如下条件:

①有且仅有一个称为根的结点$k_0 \in$ K,它对于关系N来说没有前驱,k_0称为树的根。

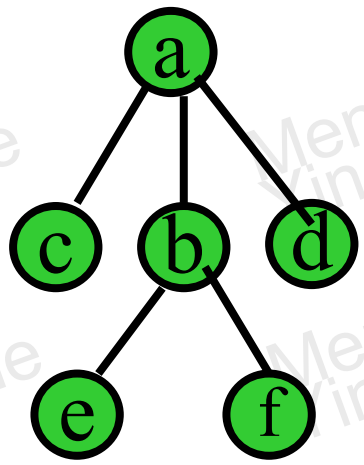
②除k_0之外,K中的每个结点对于N来说都有且仅有一个前驱。

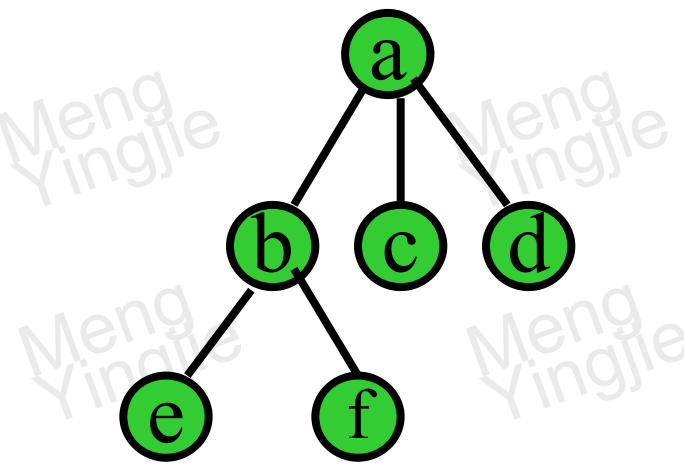
③除k_0外的任何结点$k \in$ K,都存在一个结点序列$k_0,k_1,\ldots,k_s$,使得k_0就是树根,且$k_s=k$有序对$<k_{i-1},k_i> \in$ N$(1 \le i \le s)$.这样的结点序列称作从根到结点k的一条路径。



显然树的定义属于一个递归，定义中对于子树的个数及次序并没有进行任何约束。

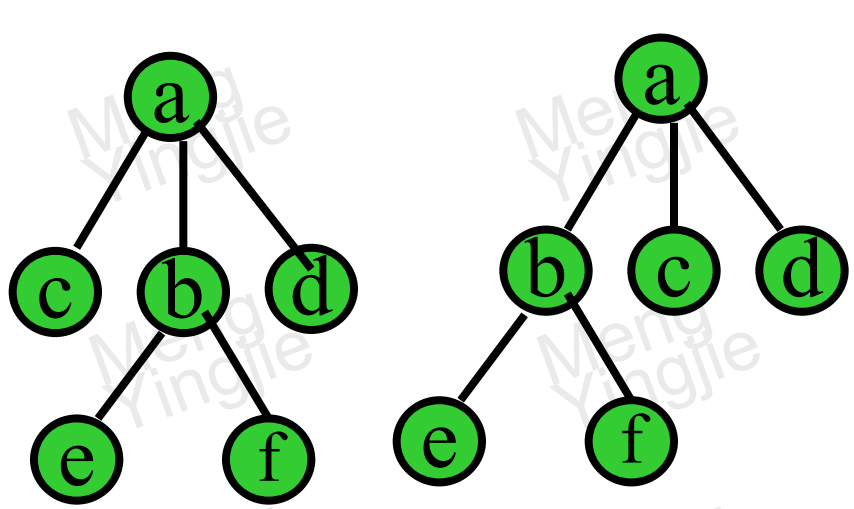
例如，对于下面的树：



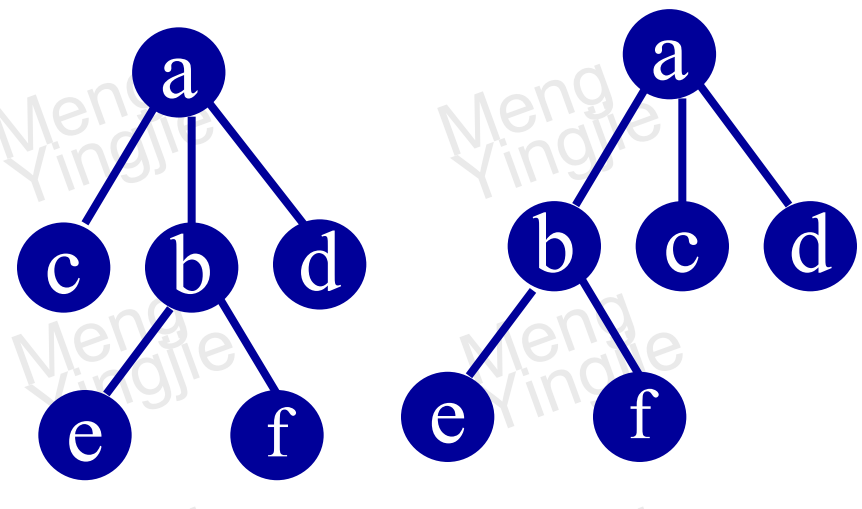


有序树:在树T中如果子树$T_1,T_2,\ldots,T_m$的相对次序是重要的(即有序的),则称T为有向有序树,简称有序树。

在有序树中可以称T_1是第一棵子树,T_2是第二棵子树，依次类推。



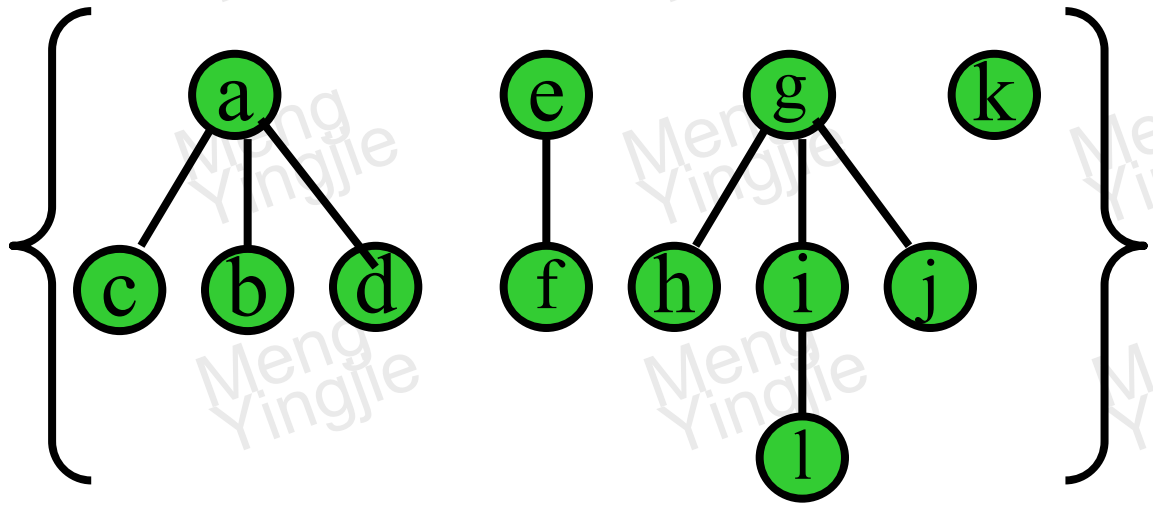
相同的树



不同的有序树

因在计算机内存中表示一棵树时，就隐含着一种确定的相对次序，因此，以后讨论的树可以认为是有序树。

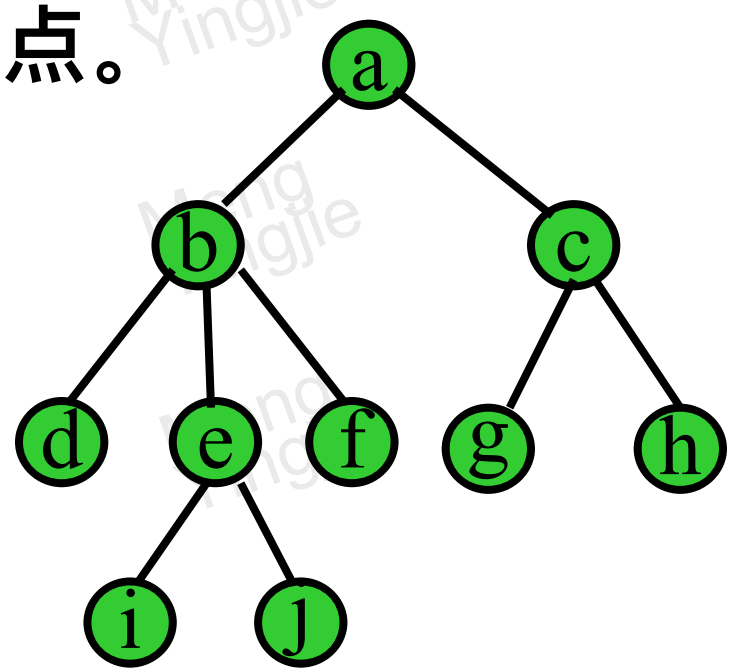
森林(或树林,forest):是零棵或多棵不相交的树的集合(通常是有序集合)。



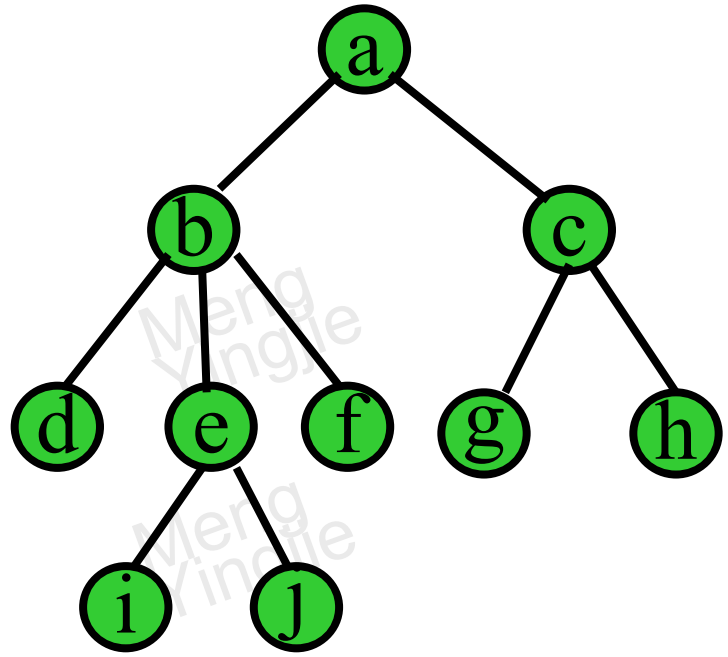
注意：这里的树与图论中的树的差别。

三.树形结构中的常用术语

- 结点的度： 子树的个数，出度。
- 终端结点： 度为零的结点，叶子。
- 非终端结点： 度非零的结点，内点。
- 树的度： 树中结点度的max值。



◆父结点、子结点：



如果< k,k' >∈ N,则称k是k'父结点(或父母、双亲),而k'则是k的子结点(或儿子、子女).

如果< k,k' >∈ N且< k,k'' >∈ N,则称k'和k''互为兄弟.

若有一条由k到达k_s的路径,则称k是k_s的祖先, k_s是k的子孙。

◆树的高度: 树中结点的最大层次数，也称深度。

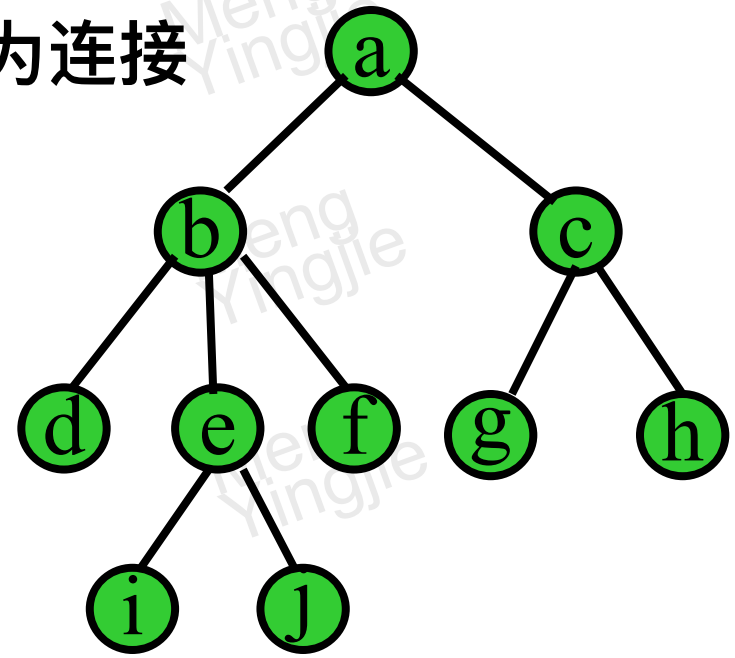
结点的层次: ①根结点在第一层。

②若结点x不是根结点,且其父结点在第i层，则x在第i+1层。

(递归定义)

◆边: 树形结构中两个结点的有序对称为连接这两个结点的边。

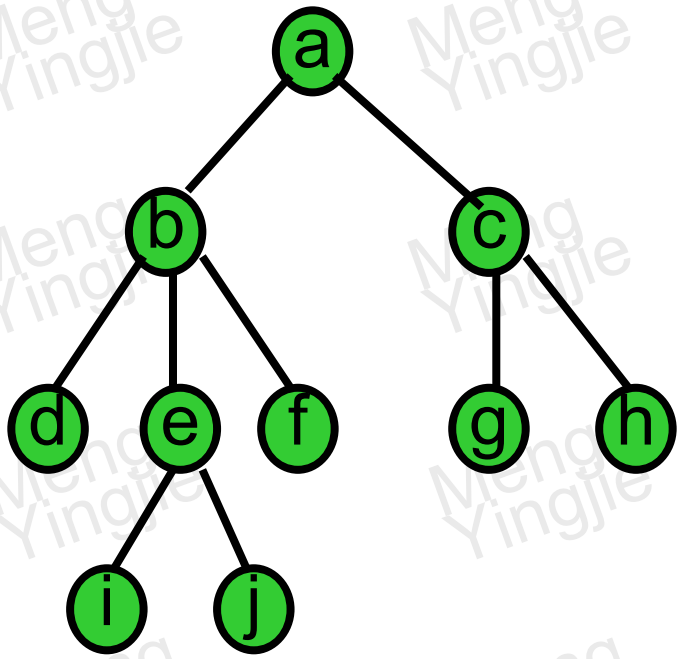
例如,<a,b>,<c,g>,<b,e>

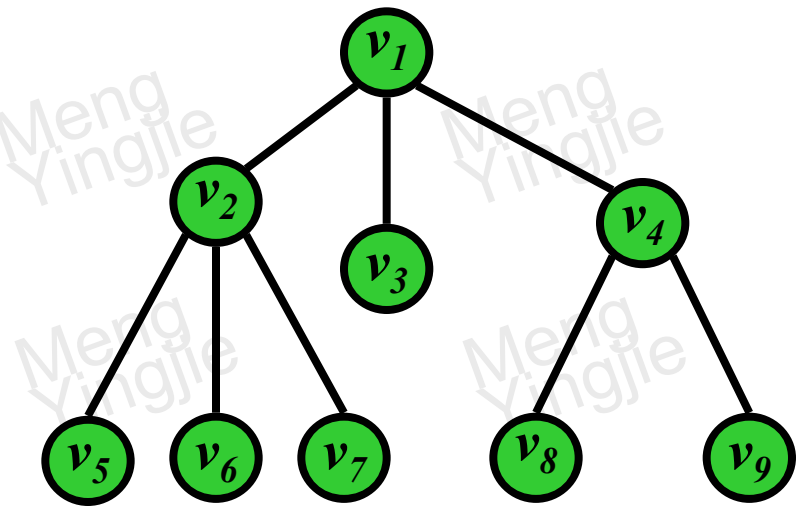


在图形化的树的表示中，边一般形式化用一个线段(隐含了有向)——本质上反映一个关系。

四.树形结构的一般表示

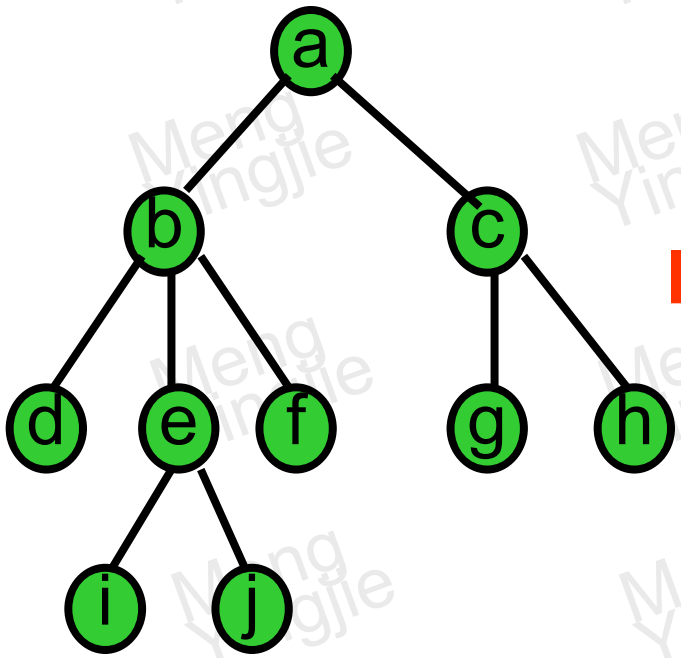
- 自然树的形态(图形)表示：



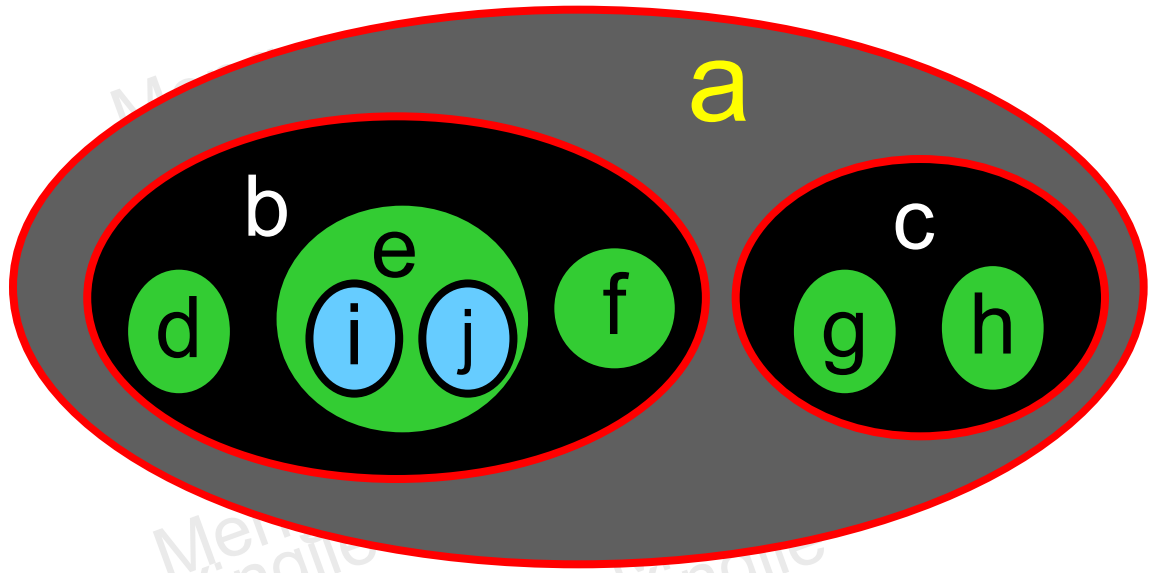


◆文氏图(Venn diagram)表示法：

以闭合的区域表示集合的图示法称文氏图.

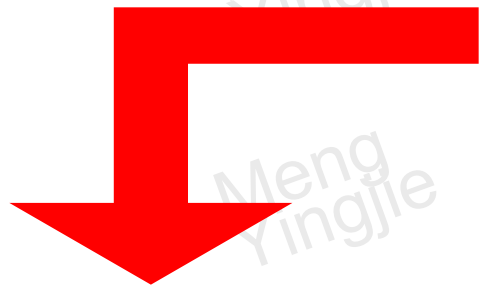


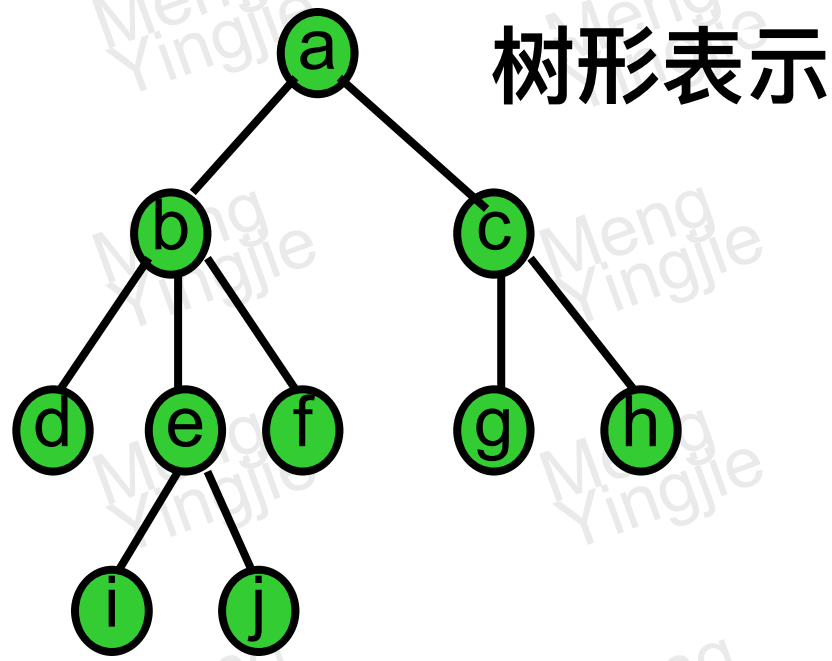
树形表示



文氏图表示

◆括弧嵌套表示法：



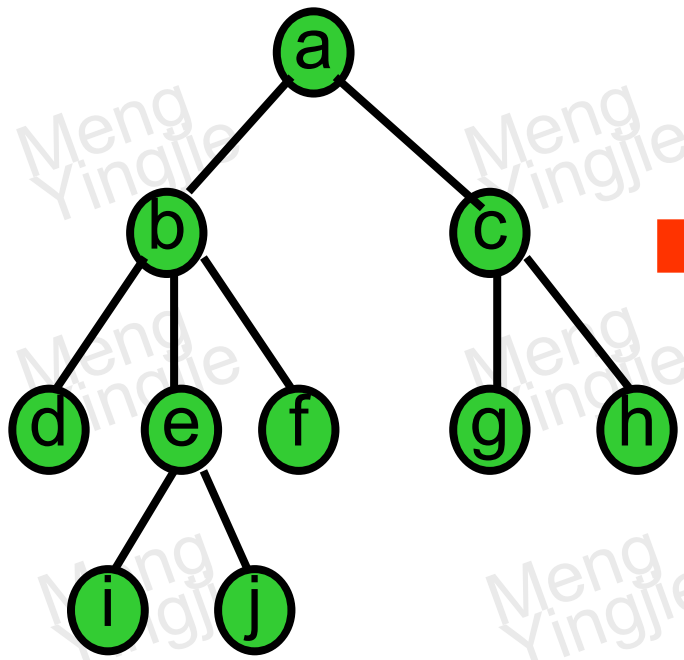


树形表示

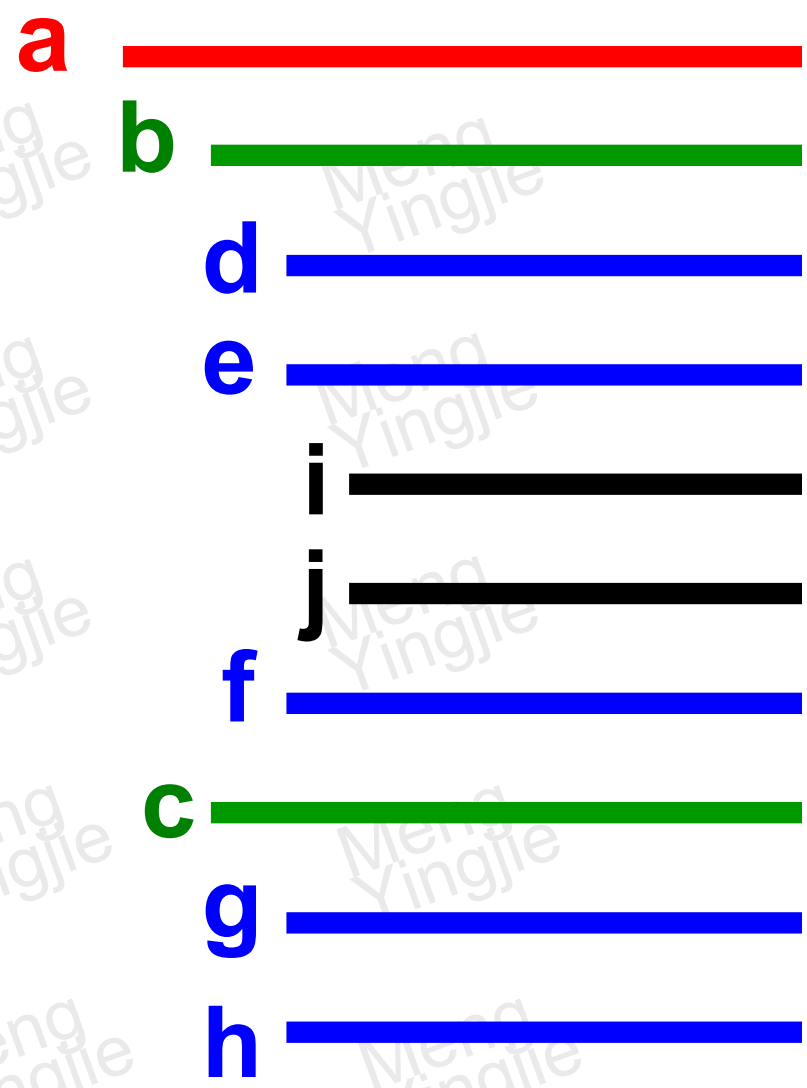
(a (b(d)(e(i)(j))(f))(c(g)(h)))

括弧嵌套表示

◆凹入表示法：



树形表示



凹入表示

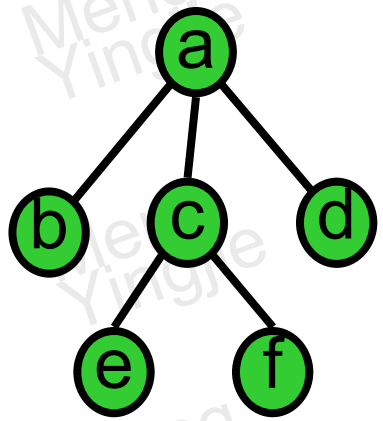
五.树形结构中的运算

1. 访问树中具有给定性质的结点。
2. 对树进行遍历。
3. 确定某结点的子结点。
4. 确定某结点的父结点。
5. 删除某结点的子树。
6. 用某棵树替换子结点。

六.树的存储

1.双亲表示法：

以一组连续的空间存储树的结点，同时在每个结点中附加一个指示器用以指出其双亲的位置。



	data	parent
1	a	0
2	b	1
3	c	1
4	d	1
5	e	3
6	f	3

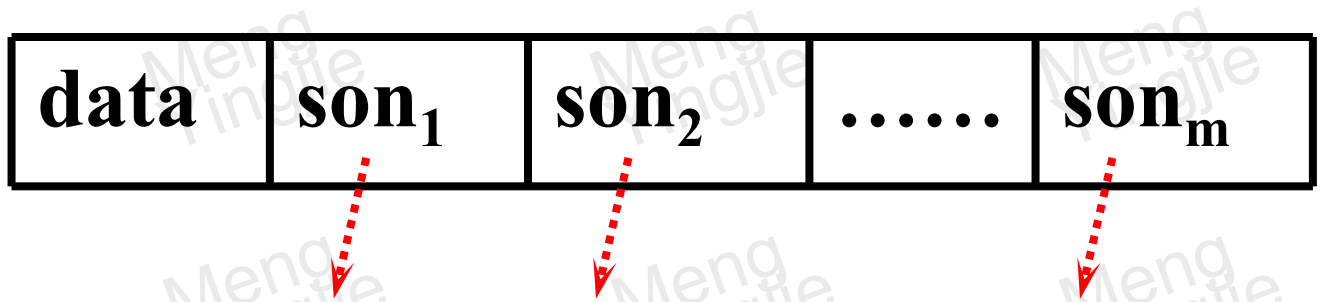
只有唯一双亲性质，属于静态结构。

2.孩子表示法：

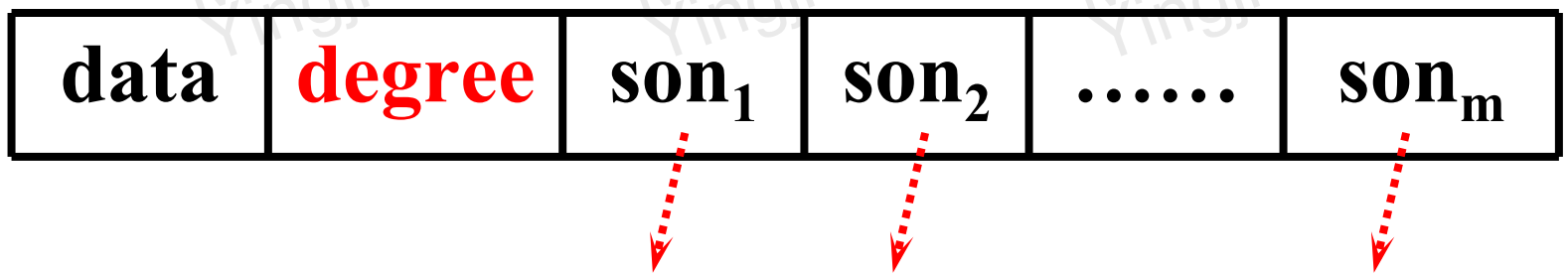
反映孩子性质。由于子树个数的不确定性，可采用：

定长/变长 结点

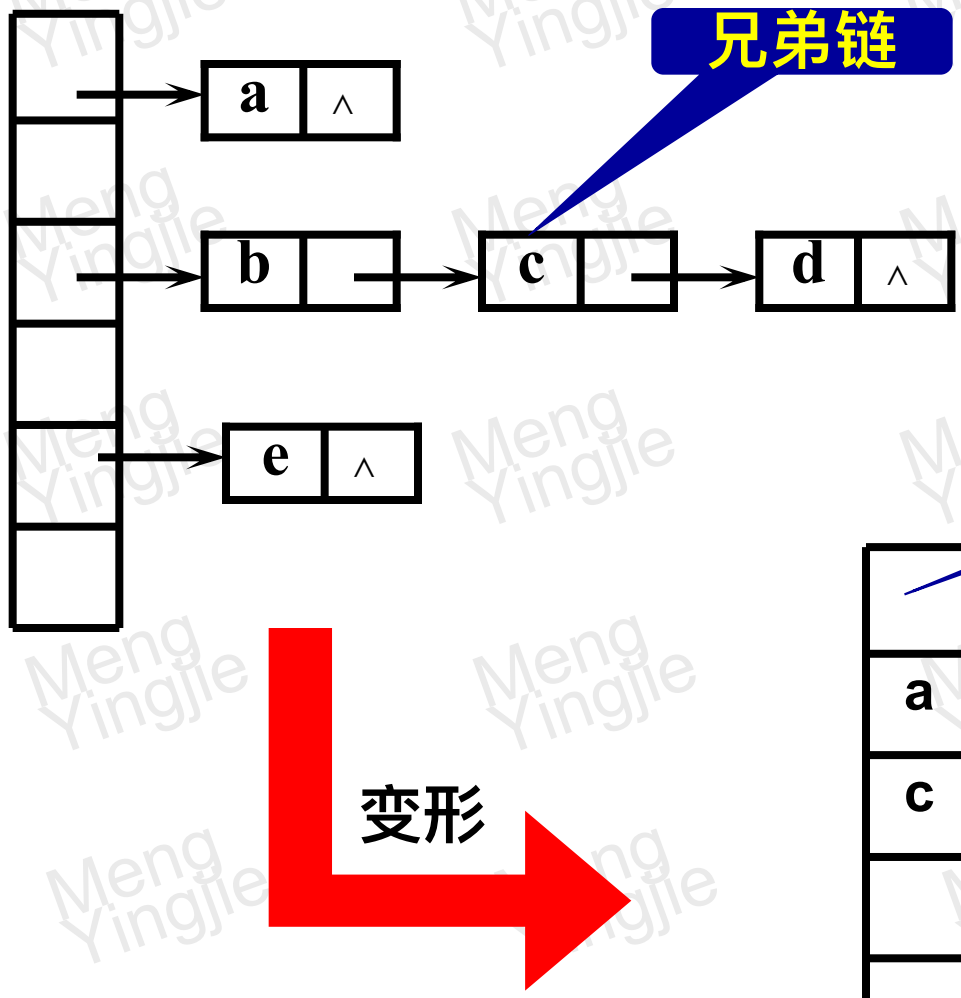
(1).定长结点：取子树个数的相对最大值作为子树项的个数。

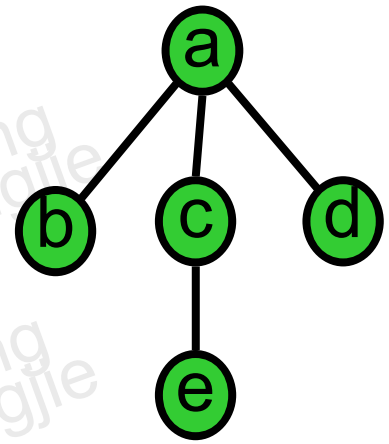


(2).变长结点：依据实际子树个数确定子树项个数。

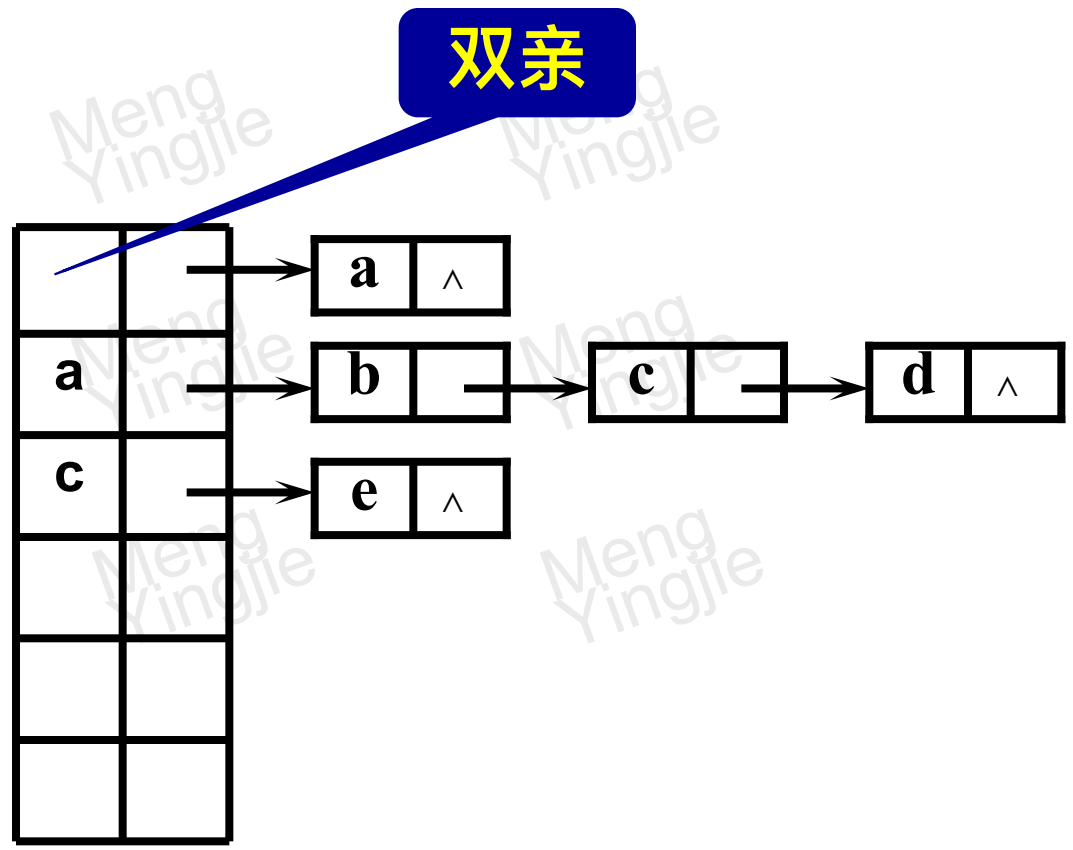


孩子表示法具体使用中的变形：



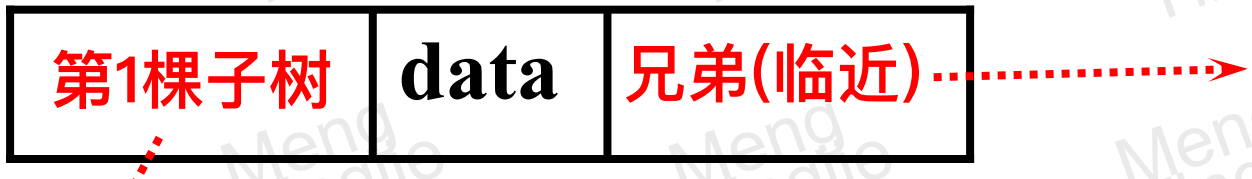


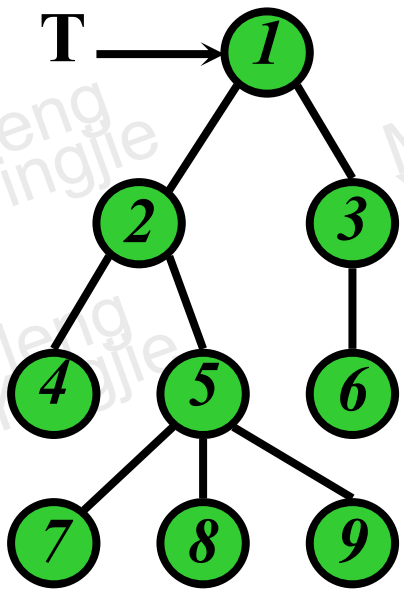
变形

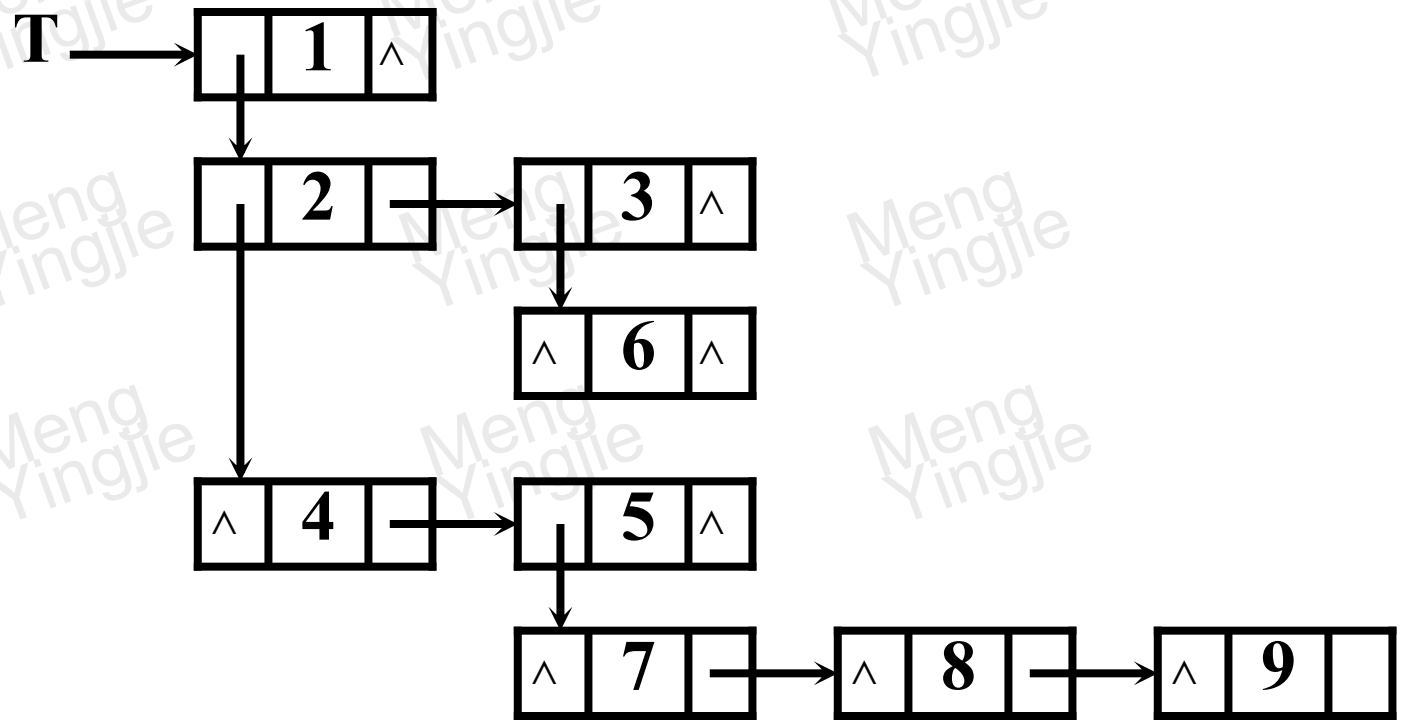


3.孩子兄弟表示法：

反映孩子及本层次结点(兄弟)性质。结点结构如下：







本节结束

一.二叉树的概念

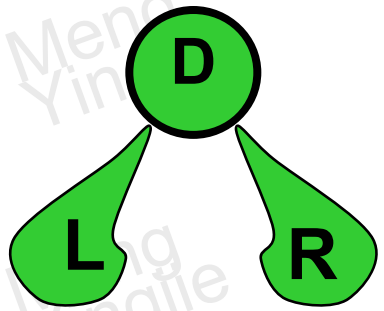
二叉树(binary tree)T是满足如下性质的结点的有限集合：

T是空集；或者T包含一个根结点且其余结点分成两个不相交的集合，并分别被称为根结点的左子树和右子树。

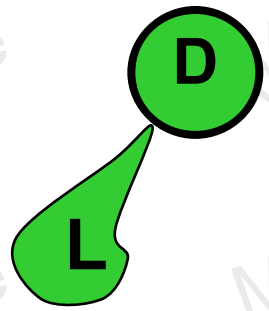
（是一个递归定义）

树与二叉树的异同

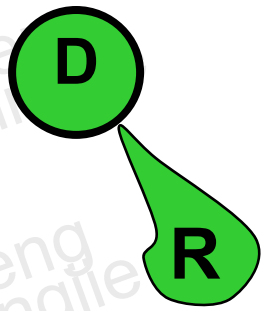
二叉树的基本形态



根和非空的左右子树

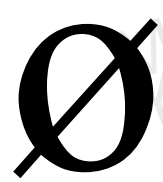


根和非空的左子树，空的右子树



根和空的左子树，非空的右子树

根和空的左右子树



空二叉树

二叉树的运算以后必须注意这五个基本形。

二叉树的定义属于递归定义，显然递归的基本项是二叉树为空的状态。

二.二叉树的特性

性质1： 在二叉树中，第i $(i\geq 1)$层的结点数最多为2^{i-1}.

证明：

性质2： 深度为k $(k\geq 1)$的二叉树结点总数最多为2^k-1.

证明：

满二叉树： 深度为k (k≥1)且有2^k-1个结点的二叉树.

(即每层结点数目都达到最大值)

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

举例说明

性质3： 任意一棵二叉树中，若终端结点数为n_0,度为2的结点数为n_2，则有：$n_0=n_2+1$

证明：

三叉树中：$n_0=2n_3+n_2+1$

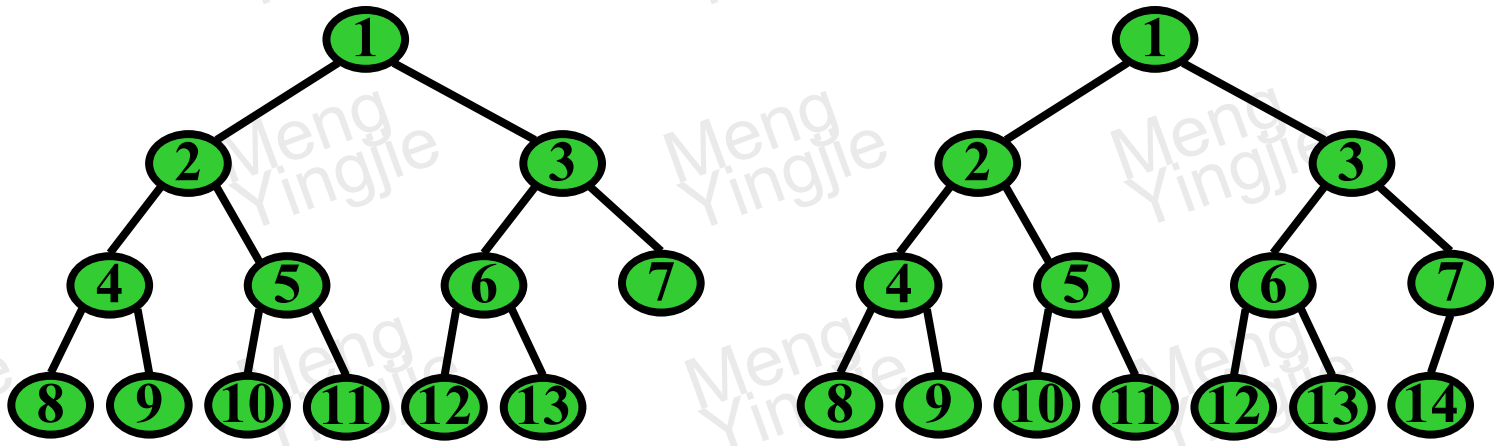
性质4： 一个有n个结点的完全二叉树其深度为$\lfloor \log_2 n \rfloor +1$.

证明：

行优先规则

性质5： 有n个结点的完全二叉树,对其结点顺序编号,则对任意结点i (1≤i≤n)有：

①若i≠1, 则 i 的双亲结点是⌊i/2⌋；若i=1, 则 i 是根，无双亲。



②若2i≤n, 则 i 的左孩子是2i；若2i＞n, 则 i 无左孩子。

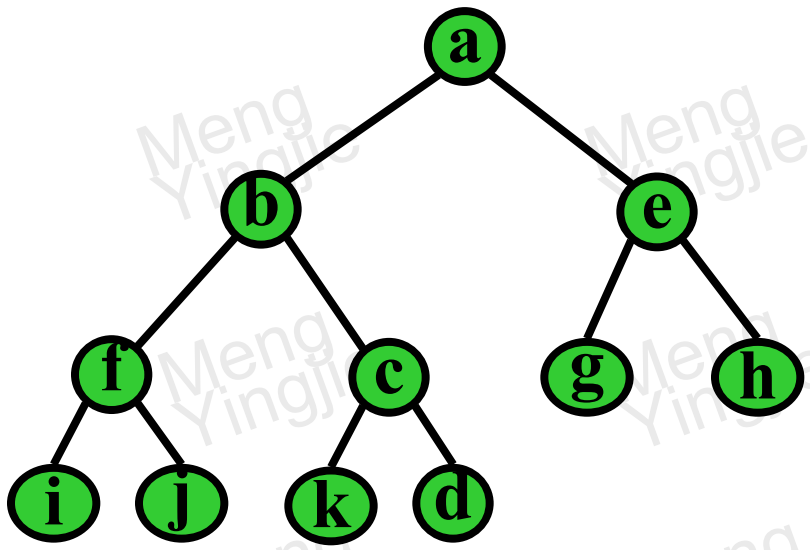
③若2i+1≤n, 则 i 的右孩子是2i+1；若2i+1＞n, 则 i 无右孩子。

（可用数学归纳法先证明②、③再导出①，此处不再赘述）

三.二叉树的存储表示

1.顺序存储(向量)

以一维数组存储二叉树的结点，尤其适合完全二叉树.



1	2	3	4	5	6	7	8	9	10	11	12	13
a	b	e	f	c	g	h	i	j	k	d		

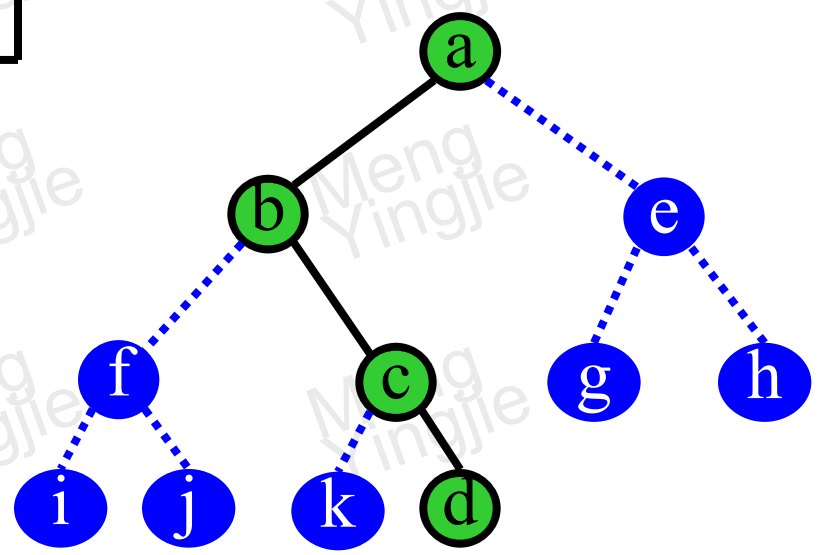
结点关系可通过性质5反映。

插入、删除实现不方便。

对于非完全二叉树的情况，仍可采用。

1	2	3	4	5	6	7	8	9	10	11	12
a	b	e	f	c	g	h	i	j	k	d	

缺点：空间浪费大(须以每层最大结点数考虑空间)



紧缩(压缩)方式：

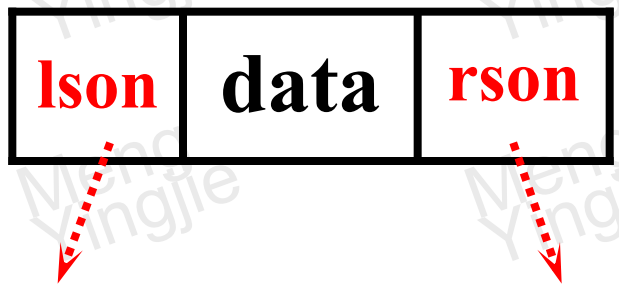
	1	2	3	4	5	6	7	8	9	10
lson	2	0	0	0						
rson	0	3	4	0						
data	a	b	c	d						

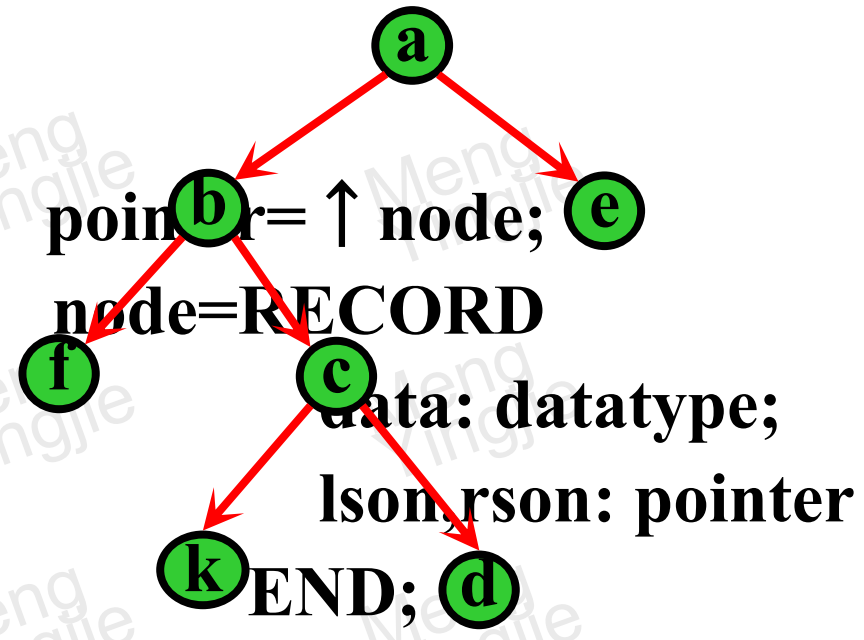
缺点:计算访问特性消失,增/删更困难。

2.链表表示

以树的孩子表示法居多。

结点结构示意图：






```
TYPE   pointer= ↑ node;
       node=RECORD
              data: datatype;
              lson,rson: pointer
            END;
Binary-Tree=pointer;
```

该表示法也称左右孩子表示法.

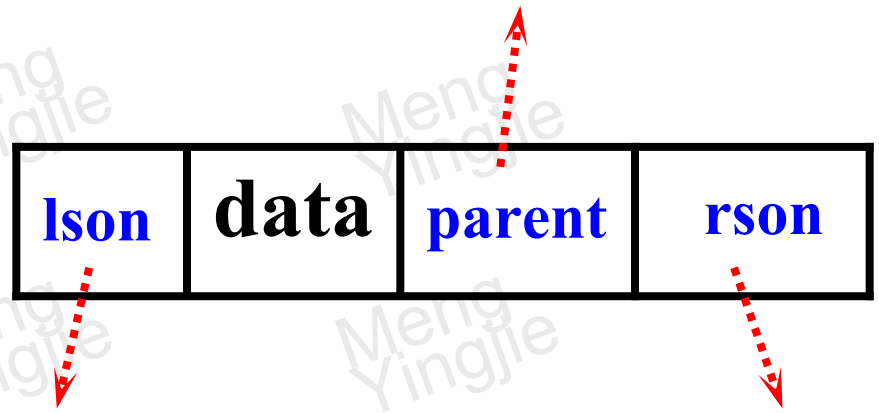
(lson-rson, llink-rlink, lchild-rchild)

(后面研究运算问题中若不加声明，默认存储形式为该形式).

性质：

3.三重(三叉)链表表示

结点结构示意图：




```
TYPE   pointer= ↑ node;
       node=RECORD
                 data: datatype;
                 lson,rson,parent: pointer
            END;
BinaryTree =pointer;
```

data	parent
lson	rson

本节结束

一.概念

遍历(traversal)：

对于给定的数据结构,系统的访问该结构中的每个结点,且每个结点仅被访问一次的操作过程称～。

系统地:按照一定的规律(次序).

访问:对元素进行的某种操作.

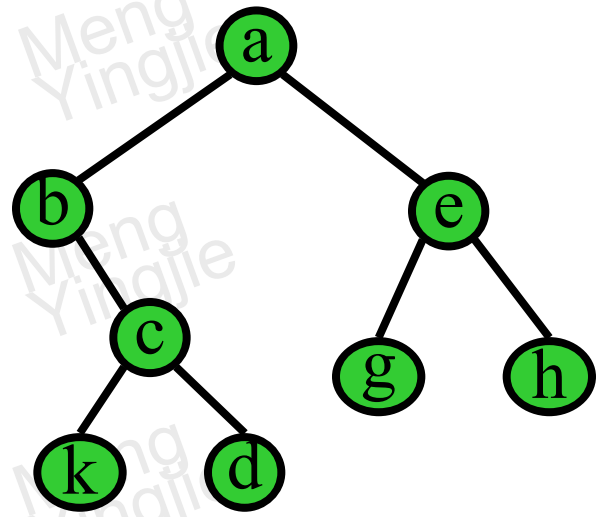
是一个操作过程.

二叉树的遍历:

对于给定的二叉树，系统地访问二叉树中的每个结点，且每个结点仅被访问一次的操作过程称～。

二.二叉树的遍历次序(order)

1.层次策略:



自上而下:

从左到右: a , b , e , c , g , h , k , d

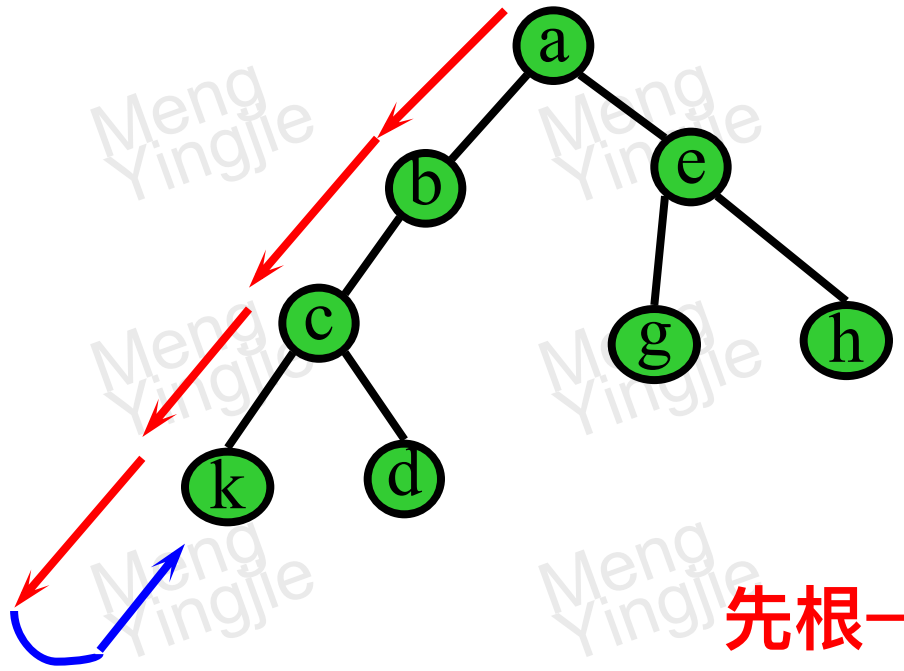
从右到左: a , e , b , h , g , c , d , k

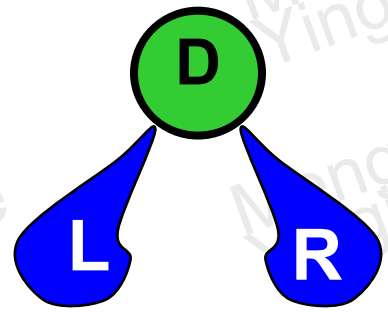
自下而上:

从左到右: k , d , c , g , h , b , e , a

从右到左: d , k , h , g , c , e , b , a

2.深度策略： 根据递归定义二叉树可以抽象为如下形式：





先根→ DLR ➡ a ,b ,c ,k , ……

DRL

中根→ LDR

后根→ LRD

RDL

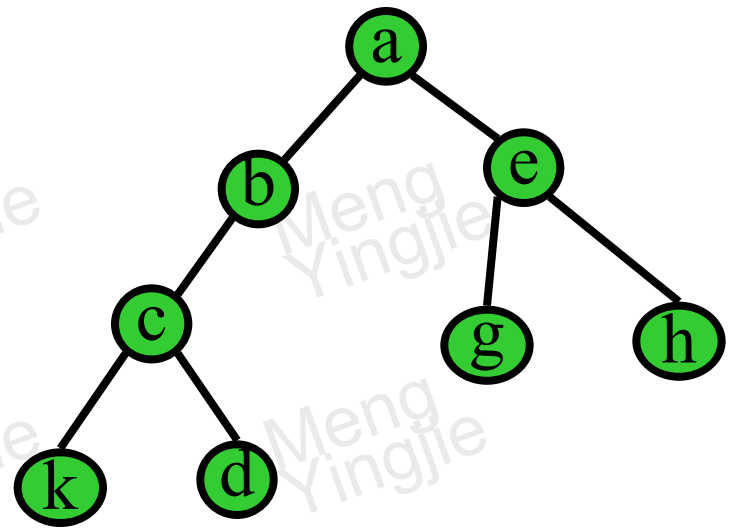
RLD

如果约定先左后右次序

三.先根(preorder)遍历

遍历方法：

访问根结点；
遍历左子树；
遍历右子树；

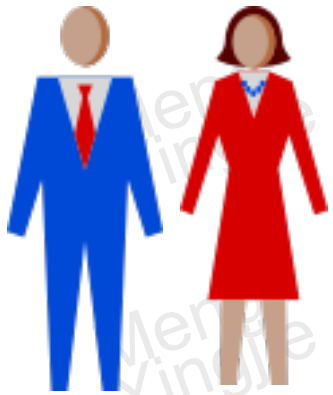


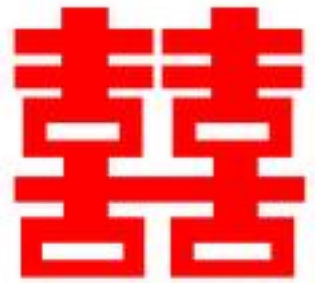
a,b,c,k,d,e,g,h

```
PROC  Preorder(VAR T: BinaryTree);
BEGIN   IF  T≠nil  THEN  【 WRITE(T↑.data);
                          Call Preorder(T↑.lson);
                           Call Preorder(T↑.rson);】
END;
```

四.中根(inorder)遍历

也称对称序、中缀、内序等，广泛使用的一种次序。现实例子：

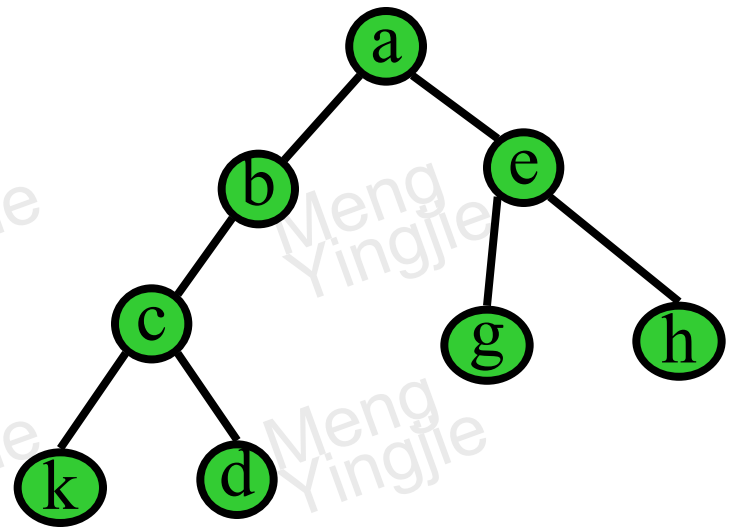





中根遍历方法：

遍历左子树；
访问根结点；
遍历右子树；

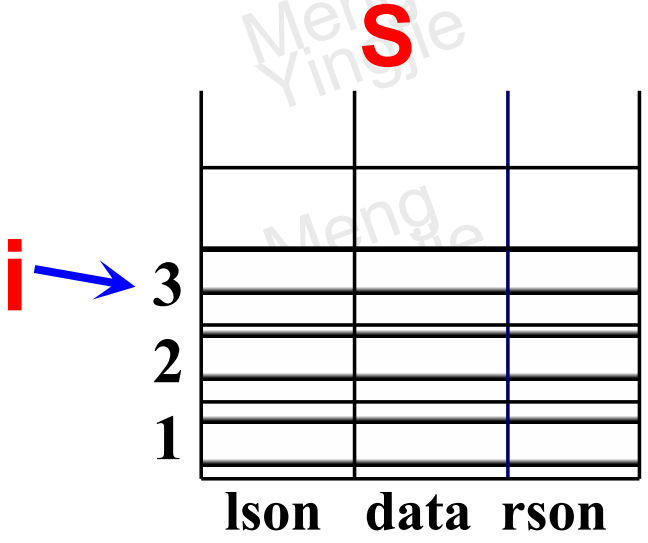


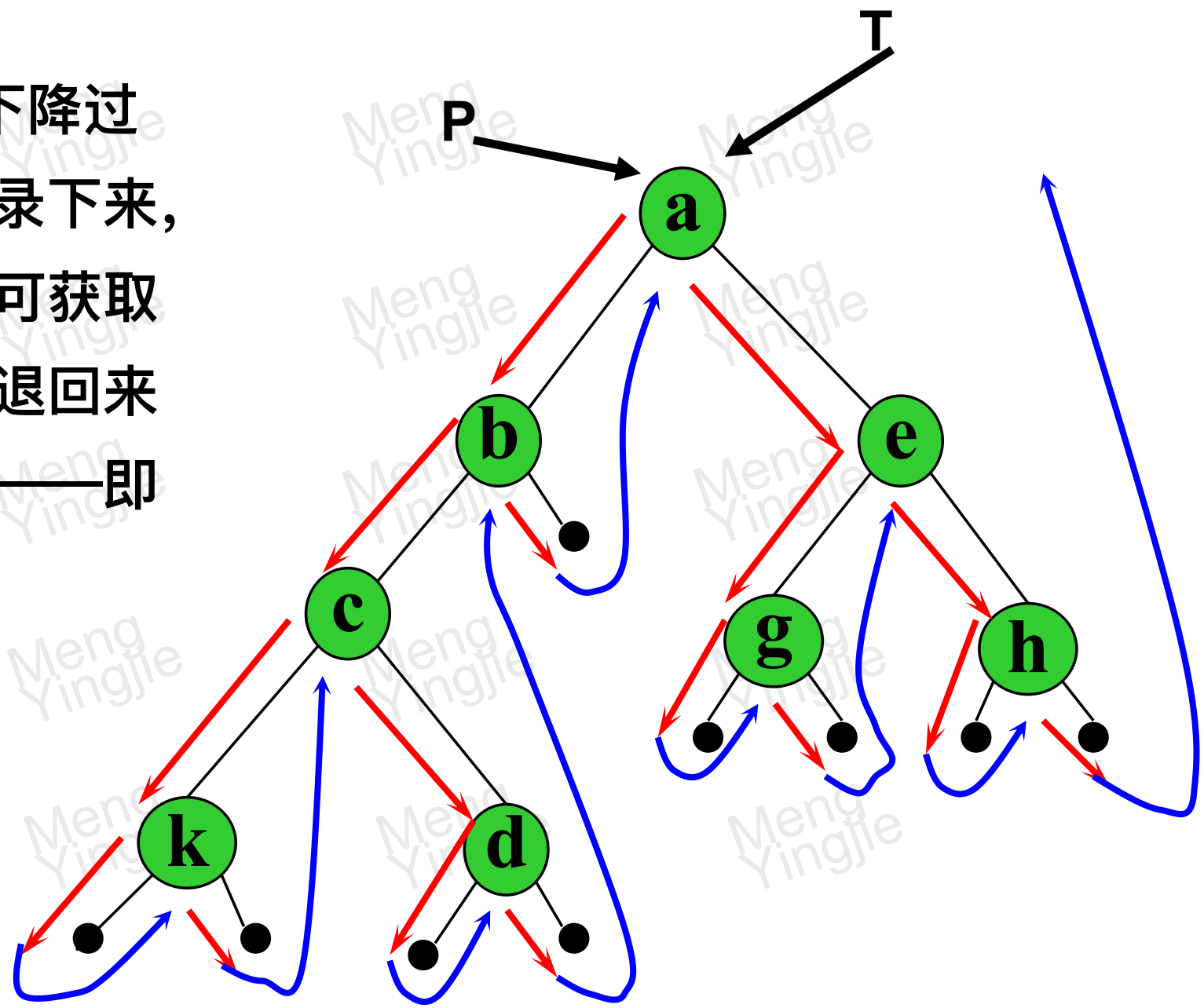
k,c,d,b,a,g,e,h

```
PROC Inorder(VAR T: BinaryTree);
BEGIN  IF T≠nil THEN 【   Call Inorder(T↑.lson);
                          write(T↑.data);
                          Call Inorder(T↑.rson);】
END;
```

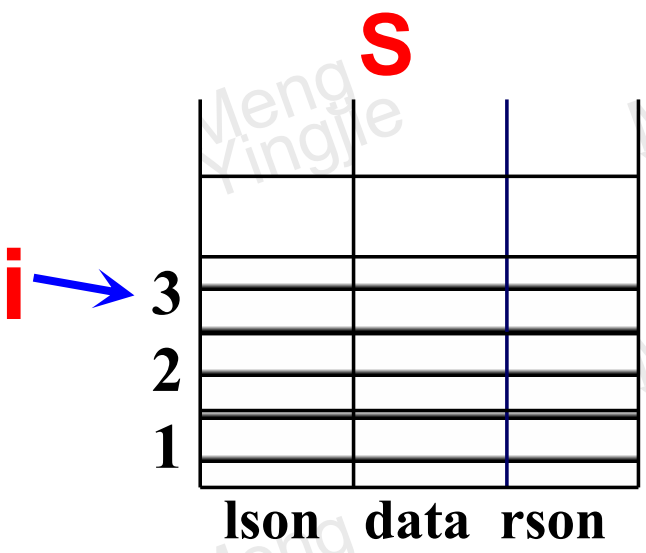
如何实现非递归过程?

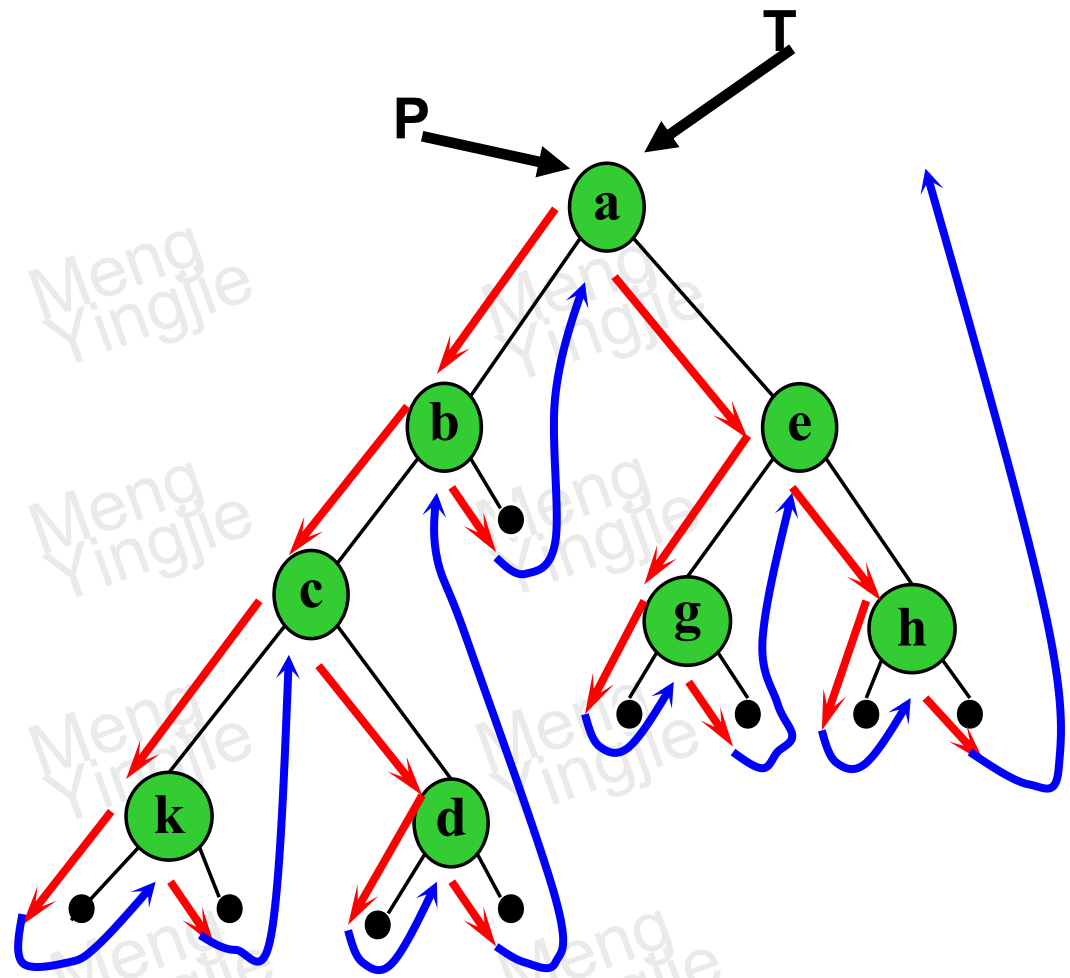
利用数组(S)将下降过程中经历的结点记录下来，再通过逆向次序即可获取正确回退结点，回退回来时进行结点的访问——即利用**堆栈**结构。





算法过程分析：





(1) 初始化:主要是两个动作：

① 搜索指针授权： p←T

② 栈的初始化： i←0

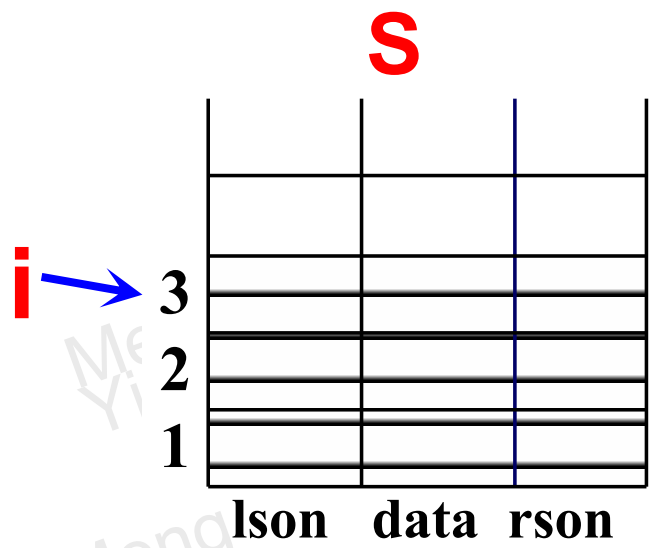
(2) 重复以下动作：

①搜索指针P不空：

保存当前P；P沿左孩子方向下降

② S中有东西时：

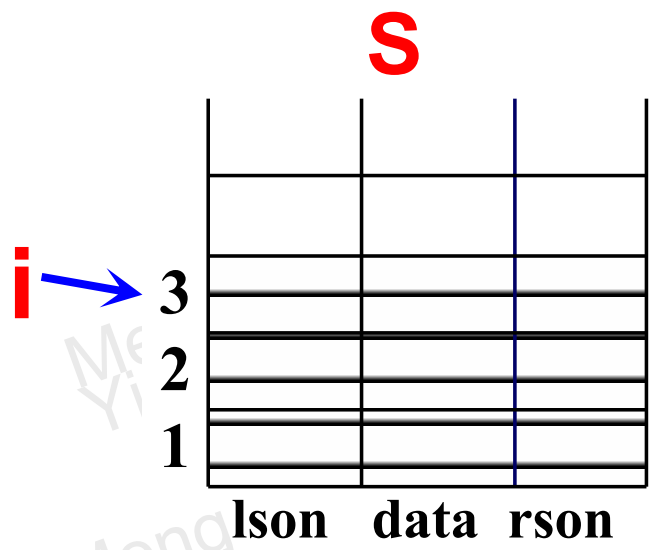
取一个结点数据；访问该结点；控制权移交其右子树



进一步细化（2）

```
REPEAT
    ① WHILE  p≠nil  DO
        保存当前P；P沿左孩子方向下降
    ② IF  i≠0  THEN
        取一个结点数据；访问该结点；控制权移交其右子树
UNTIL（i=0）AND （p=nil）
```

中序非递归过程实现(1)：




```
PROC  Inorder1(VAR T: BinaryTree);
VAR   S: ARRAY[1..max] OF BinaryTree;
BEGIN     i←0;   {i用作栈顶指针}
          p←T;   {p用作搜索指针}
          REPEAT
               WHILE p≠nil DO {遍历左子树}
               【  i←i+1;     S[i]←p; {入栈}
                   p←p↑.lson】 ;
          IF i≠0 THEN 【p← S[i];  i←i-1;  {退栈}
                        write(p↑.data); {访问根}
                        p←p↑.rson; {遍历右子树}】
          UNTIL (i=0) AND (p=nil)
END;
```

改进：没有考虑栈的尺寸

中序非递归过程实现(2)：

```
PROC  Inorder2(VAR T: BinaryTree);
VAR  S: ARRAY[1..max] OF BinaryTree
BEGIN  i←0;  tag←0;  p←T;
       WHILE (i≠0) OR (p≠nil) DO
          CASE
          tag=0: {遍历左子树}
                 IF p≠nil THEN 【 i←i+1;  S[i]←p;
                                  p←p↑.lson;  tag←0 】
                        ELSE  tag←1 ;
          tag=1: {访问根结点，遍历右子树}
                 【p← S[i];    i←i-1;    WRITE(p↑.data);
                   p←p↑.rson;   tag←0 】
          ENDCASE
END;
```

不使用栈(不需要额外空间)的遍历

也称逆转链遍历法:

在遍历的过程中当沿着结点的左或右子树“下降”时，临时改变lson或rson的值使其指向结点的双亲结点，为以后的“上升”提供路径。上升的过程中再将结点的lson和rson恢复为原来的值。

为了表明是上升还是下降，需给每个结点增加一个标志位tag:

初始状态所有结点tag为0，搜索过程中进入p的左子树时将p的tag改为1，从p的左子树推出时再将其恢复为0。

缺陷:

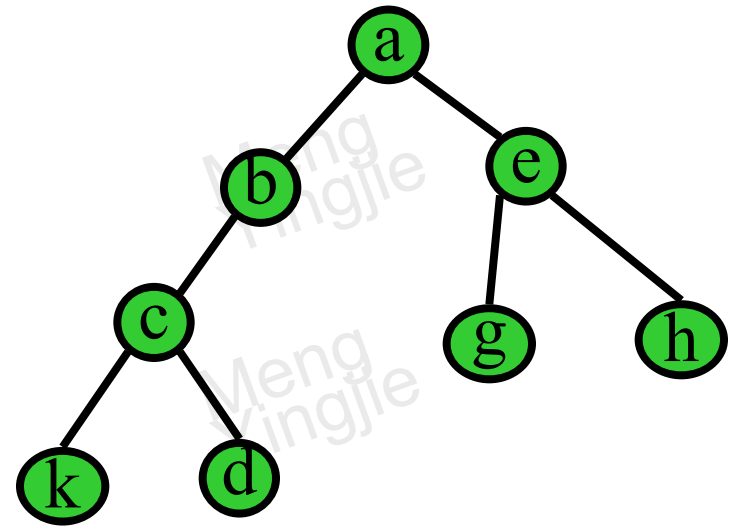
不允许多个过程同时对同一棵二叉树操作；若遍历中途操作失败，可能导致二叉树信息的丢失——被修改的指针无法恢复。

可通过线索树解决该缺陷——同样不需要额外栈空间。

五.后根(postorder)遍历

后根遍历方法：

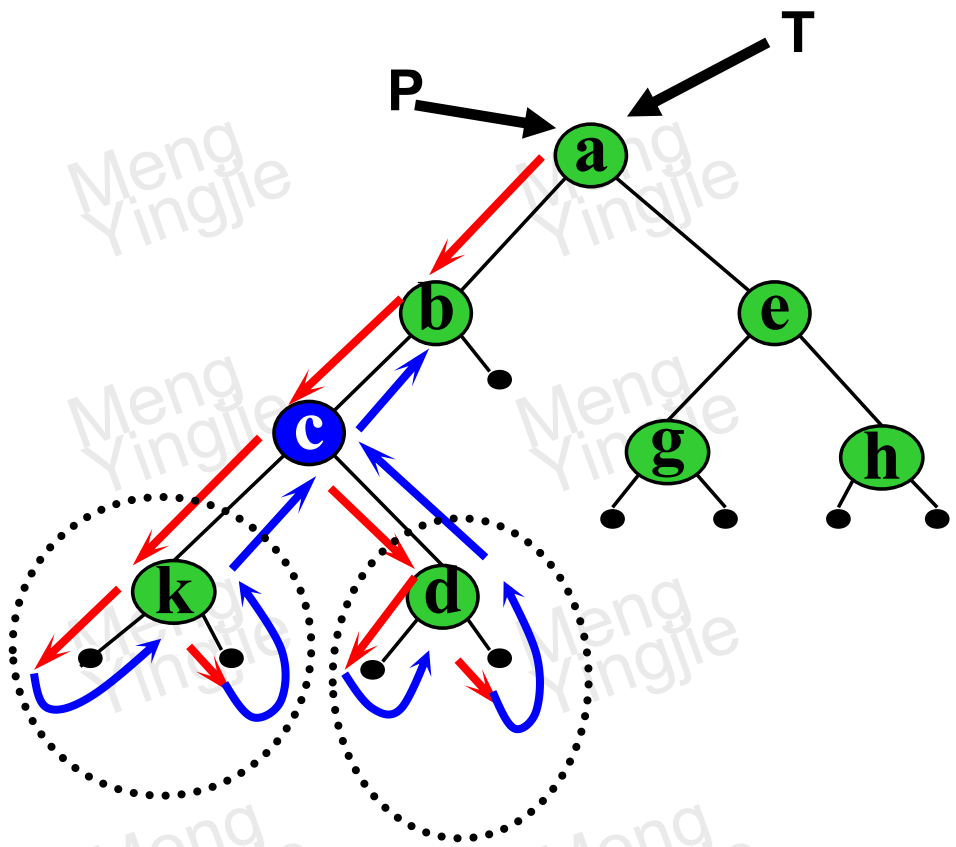
- 遍历左子树；
- 遍历右子树；
- 访问根结点；



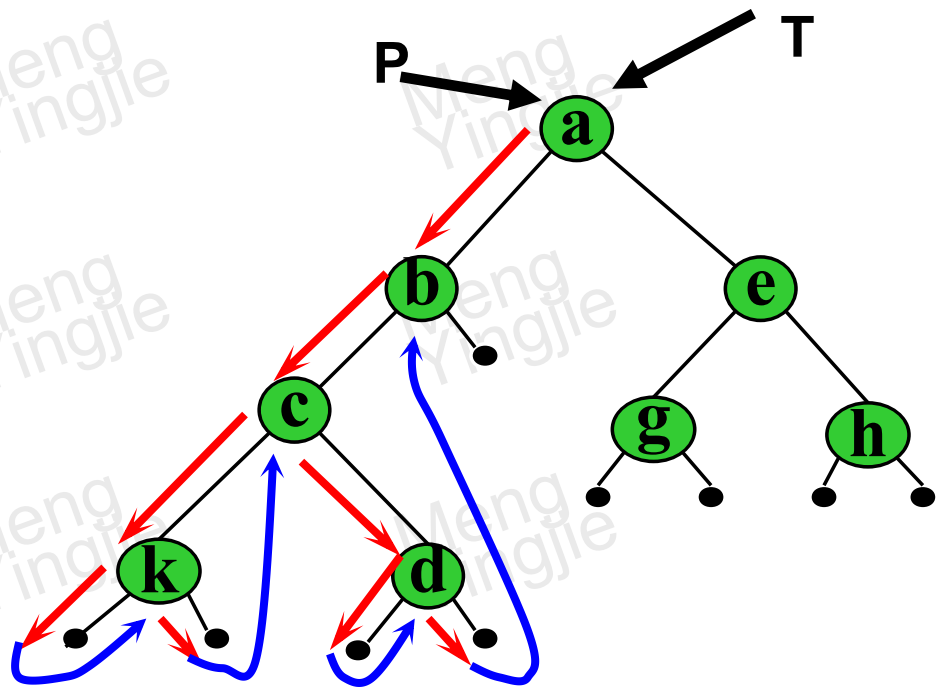
k,d,c,b,g,h,e,a

```
PROC Postorder(VAR T: BinaryTree);
BEGIN  IF T≠nil THEN 【Call Postorder (T↑.lson);
                      Call Postorder (T↑.rson);
                      write(T↑.data)】
END;
```

非递归的后根遍历:



后序搜索局部示意图



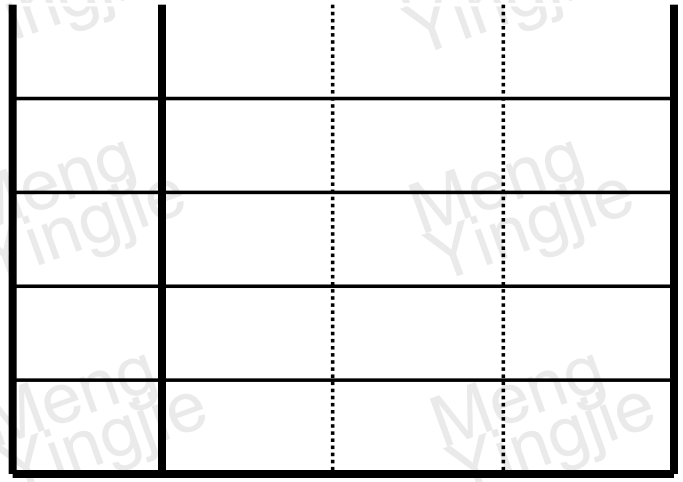
中序搜索局部示意图

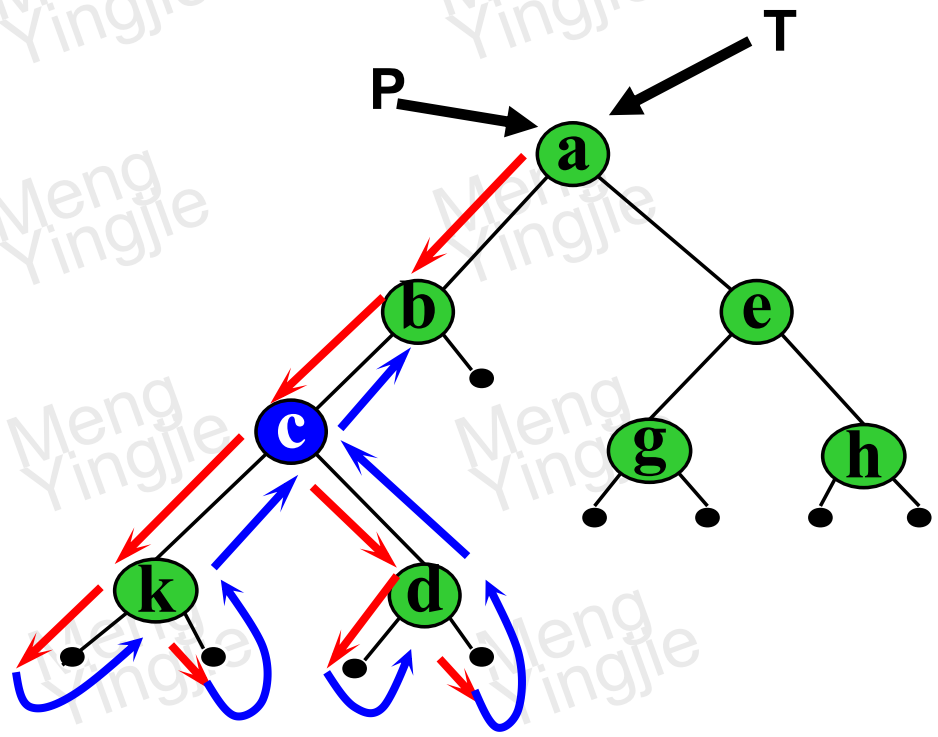
当搜索指针指向某结点(例如,C) ,不能立即访问，须先遍历左子树，因此要将此结点入栈保存，当处理完左子树后，还不能访问，还需遍历其右子树，须将该结点再次入栈保存——二次入栈。

为区别二次入栈需设立标志位tag

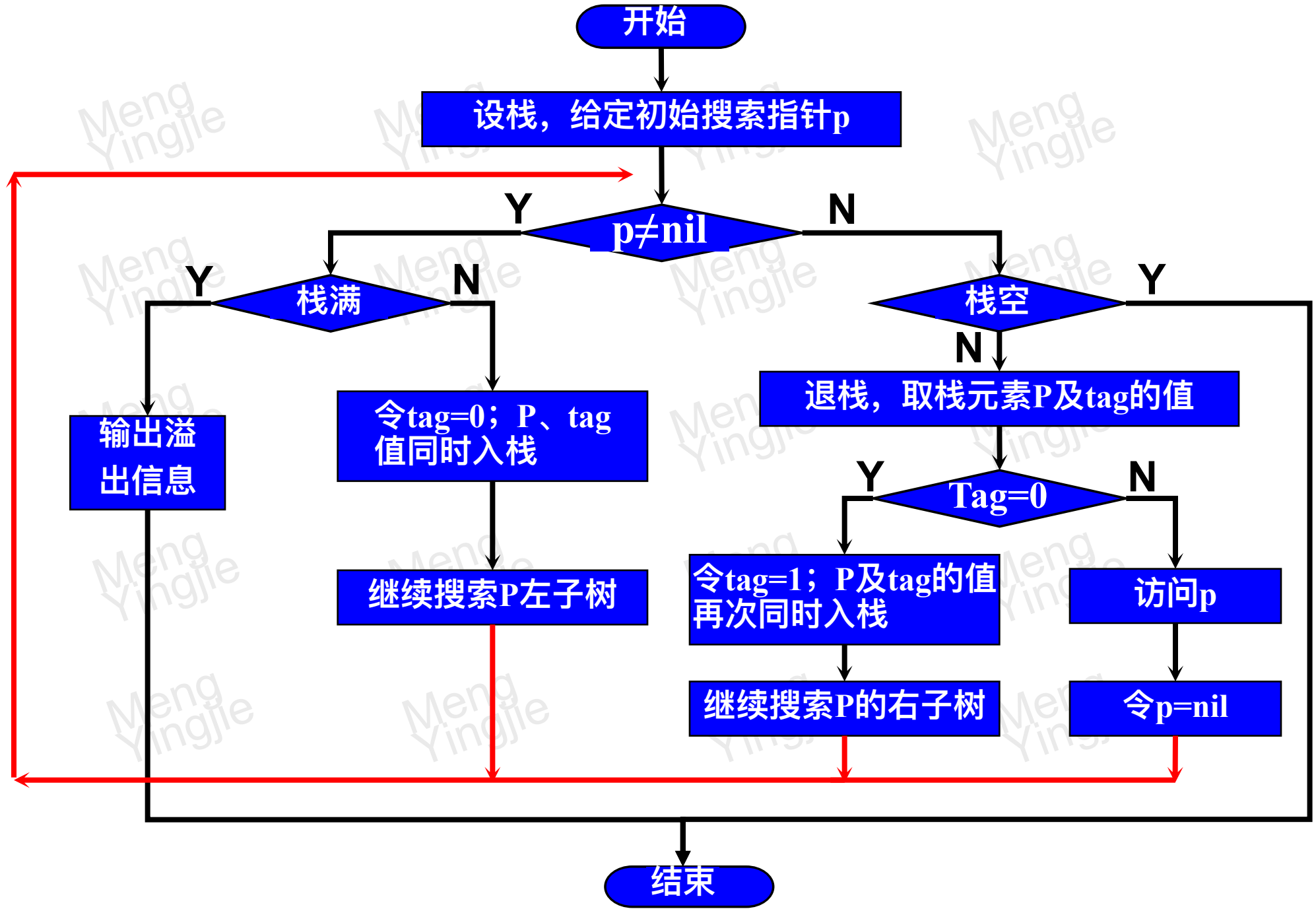
$$tag=\begin{cases}0,\text{该结点暂不访问}\\1,\text{该结点可以访问}\end{cases}$$

S





非递归的后根遍历过程:


开始
设栈，给定初始搜索指针p
p≠nil
Y
N
栈满
Y
N
输出溢出信息
令tag=0；P、tag值同时入栈
继续搜索P左子树
栈空
Y
N
退栈，取栈元素P及tag的值
Tag=0
Y
N
令tag=1；P及tag的值再次同时入栈
继续搜索P的右子树
访问p
令p=nil
结束

本节结束

一.概论

树、森林难于把握，实际使用中运用的最多的还是二叉树(或m叉树)。

用二叉树表示树是很重要的。

转换的原因：

1、二叉树有许多优良的特性，树则没有这些特性。

举例.

2、基于存储的原因：

树形结构的最佳存储方式——采用链式存储。为了处理的方便性一般用定长结点结构。设T是k叉树,结点总数为n个结点.

无效指针域(nil)个数为：k*n-(n-1)= (k-1)*n+1,或者说：

指针空间的有效空间利用率= $\frac{n-1}{k\times n}\times 100\%$

k 越大空间利用率越低:

k=2, (n-1)/2n≈50%

k=3, (n-1)/3n≈33%

k=4, (n-1)/4n≈25%

k=5, (n-1)/5n≈20%

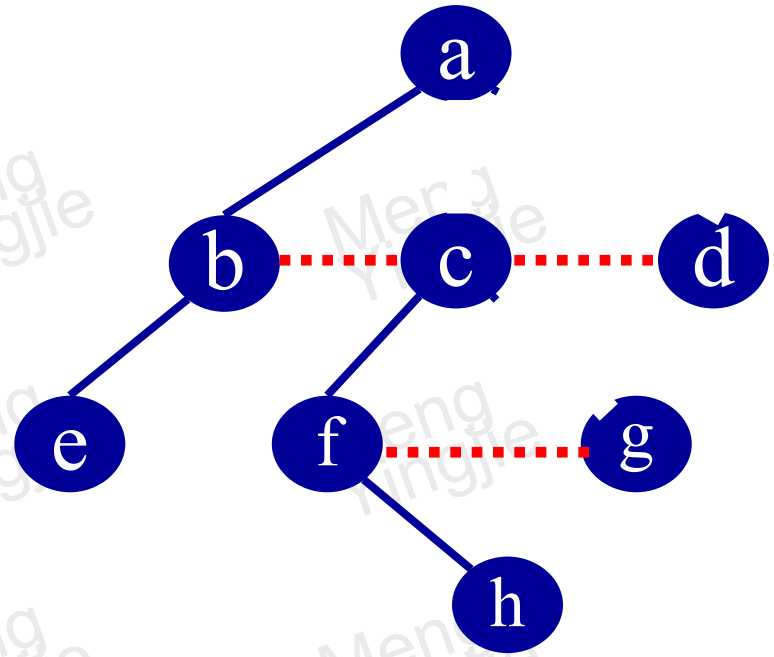
……

在动态结构组织下，二叉树空间利用率最高。

二.树转为二叉树

这里约定树为有序树。

原理：根据树的孩子兄弟存储策略演化而来)

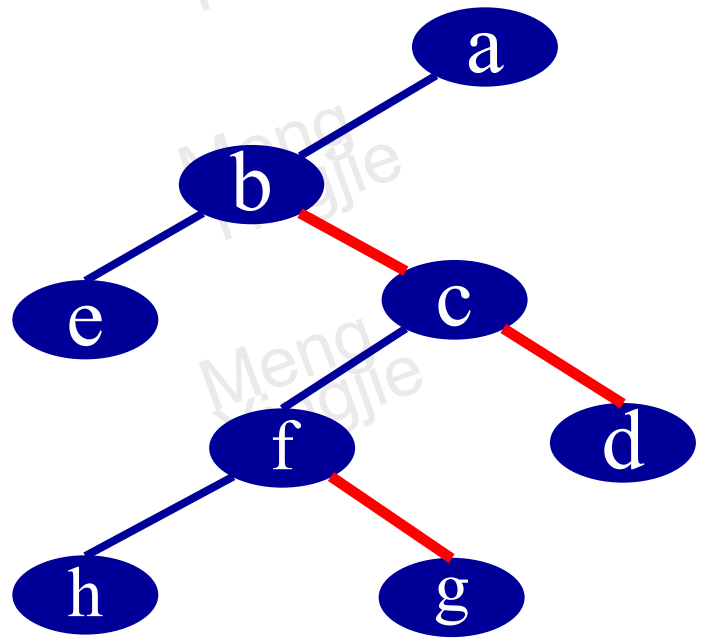


转换规则：

1.加线

2.抹线

3.调整



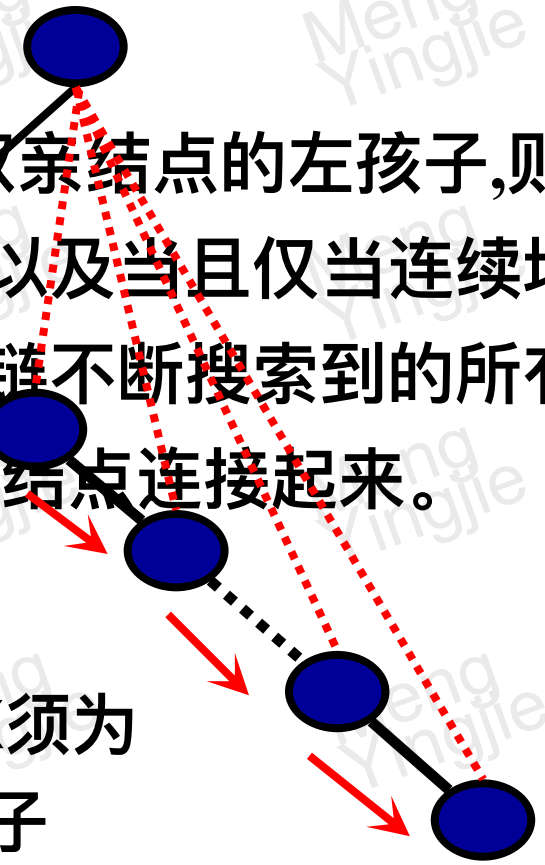
结果二叉树的特点：

三.二叉树还原为树

还原规则：

1.加线

若某结点x是双亲结点的左孩子,则将该结点的右孩子以及当且仅当连续地沿着此右孩子的右链不断搜索到的所有右孩子都与i的双亲结点连接起来。

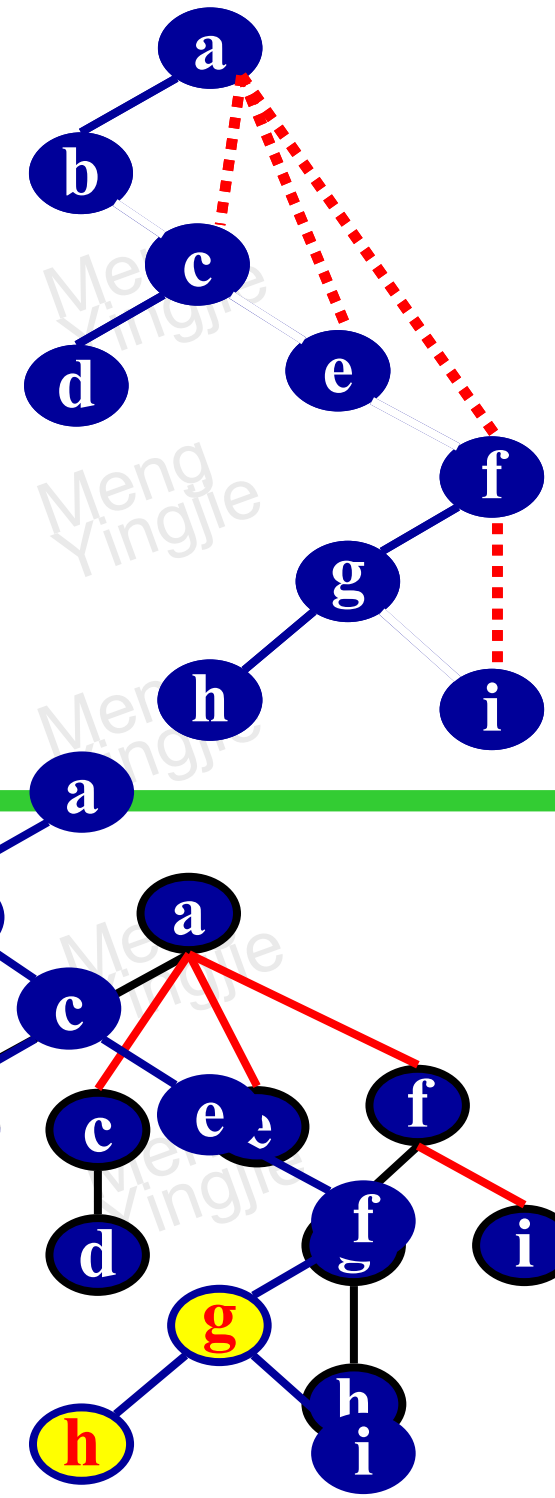


对所有X：X须为其父的左孩子

2.抹线

抹掉结点与所有右孩子之间的连线。

3.调整



四.森林转换为二叉树

转换规则:

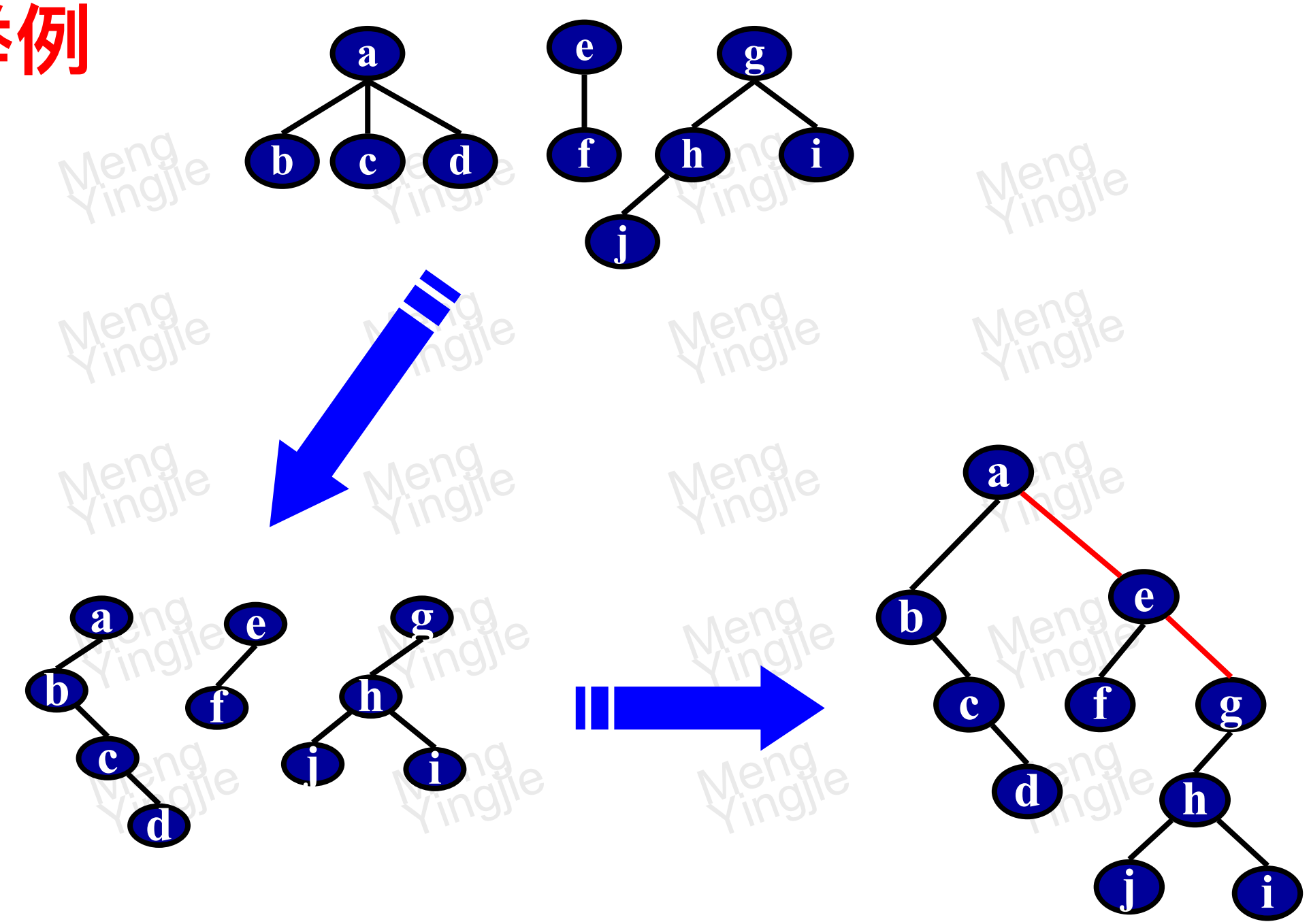
1.树转换:

将每一棵树转为二叉树。

2.二叉树连接:

依据森林中树的次序,依次将转换得来的二叉树，后一棵作为前一棵二叉树的根结点的右子树。

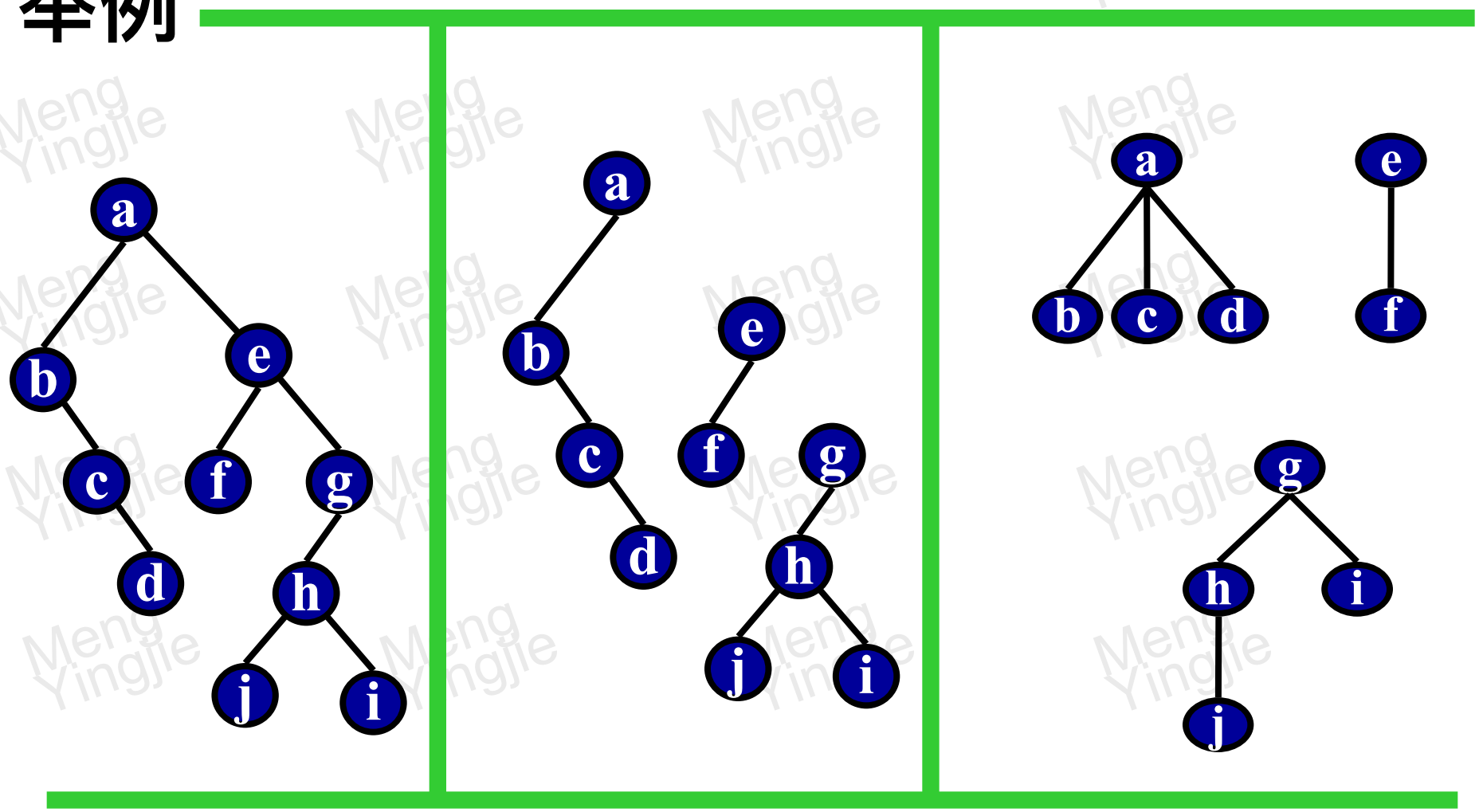
举例



五.二叉树还原为森林

还原规则： 1.抹线　　2.还原

举例



六.树的遍历

可转为二叉树后处理，然后再还原，也可以直接处理。

1.层次遍历：

2.先根遍历

- 访问树的根结点；
- 先根次序下依次遍历根结点的每个子树;

3.后根遍历

【中根遍历】

七.森林的遍历

可转为二叉树后处理，然后再还原，也可以直接处理。

1.层次遍历：须利用队列结构

2.先根遍历：

{ 访问第一棵树的根结点；
  先根次序下依次遍历第一棵树的每个子树；
  先根次序依次遍历其它的树。

3.后根遍历：

{ 后根次序下依次遍历第一棵树的每个子树；
  访问第一棵树的根结点；
  后根次序依次遍历其它的树。

本节结束

一.概论

二叉树遍历结果的应用与再利用。

对于给定的二叉树，在某种遍历规则下，可以得到唯一的**线性序列**——非线性结构中蕴涵着线性关系。

这种线性关系可以加以利用：

- 通过某些线性次序可以确定一棵二叉树
- 通过线性次序中可以快速得到结点的前驱和后继结点的特性，可以提高二叉树中某些运算实现的效率。

1.通过某些线性次序可以确定一棵二叉树

先序序列和中序序列可以唯一确定一棵二叉树。 证明

举例:

二叉树T的先序序列为(G,B,Q,A,C,P,D,E,R),中序序列为(Q,B,C,A,G,P, E,D,R). 画出T的示意图。

先序：(G,B,Q,A,C,P,D,E,R)， 中序：(Q,B,C,A,G,P, E,D,R)

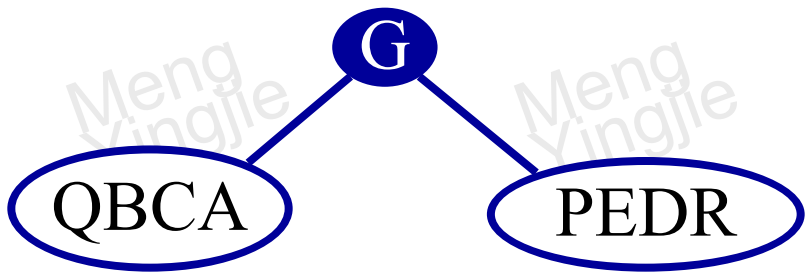
解： 由根向下画出T。

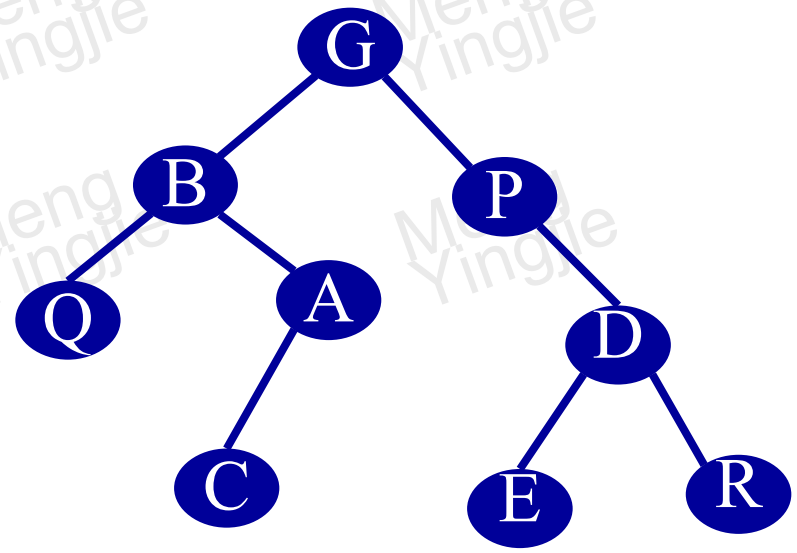
①选择前序中的第一个点得到T的根，因此G为T的根。

②点G的左子树如下得到,先用T的中序求T的左子树T_1中的点。它们在G的左边,因此T_1由QBCA构成,然后在T_1的前序（即以T的前序出现）中选择第一点(根)就的到G的左孩子,因此B为G的左孩子。

③类似,G的右子树由PEDR构成,且P为T_2的根,即P为G的右孩子。

④对每个新结点(如B,P)重复上面的过程，即可得到所需二叉树。

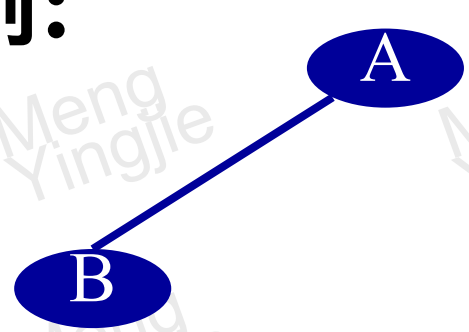




后序序列和中序序列也可以唯一确定一棵二叉树。 证明

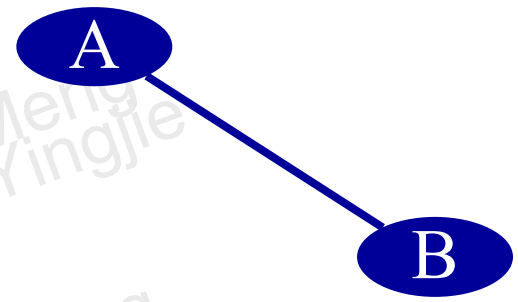
先序序列和后序序列可以唯一确定一棵二叉树?

不能，举例:



先序:A B

后序:B A



先序:A B

后序:B A

层次序列和中序序列也可以唯一确定一棵二叉树。

2.对于遍历规律下蕴涵的线性关系的利用

由于通过遍历可以得到唯一线性序列，这样对于每个结点就容易获取其前驱和后继。

但无法从二叉树直接获取结点的前驱和后继。当然可以通过遍历把信息记载下来以备使用，这样除浪费空间外，二叉树一但变化，就需要重新遍历。

能否不遍历就可获取直接前驱或直接后继?

——线索树。

二.概念

二叉树的lson-rson存储结构中有2n个指针，只有n-1个指针指示了左右孩子，而另外n+1个指针为无效指针。

目的：利用这些无效指针来表达线性关系下的直接前驱和直接后继关系，使其成为有效指针。

给二叉树加线索的过程称为线索化，表示基线性关系下前驱或后继时，这种指针称为线索(threaded)。

带线索的二叉树称为线索二叉树，简称线索树(或穿线树，threaded tree)。

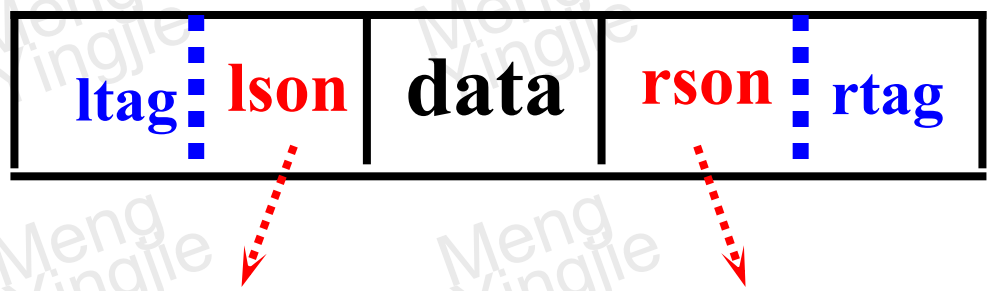
前驱与后继如何表达：

通常将空的左指针域用于表示直接前驱；

空的右指针域用于表示直接后继。

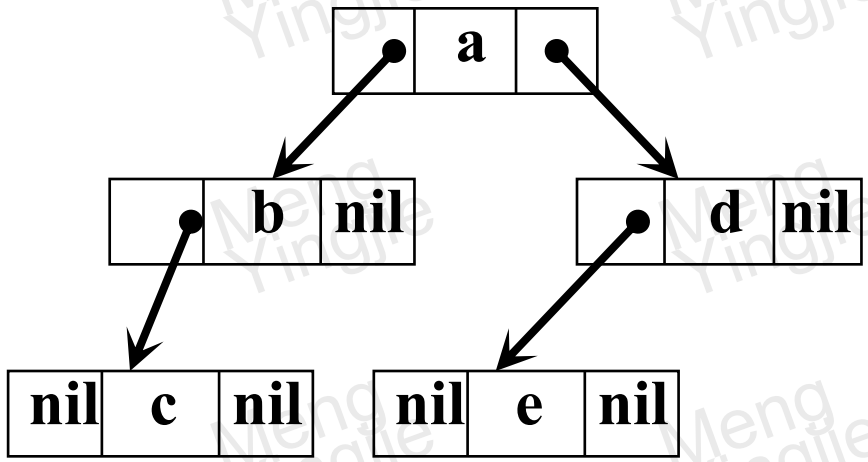
如何区别指针的两种用途?

可通过加标志位来区分。



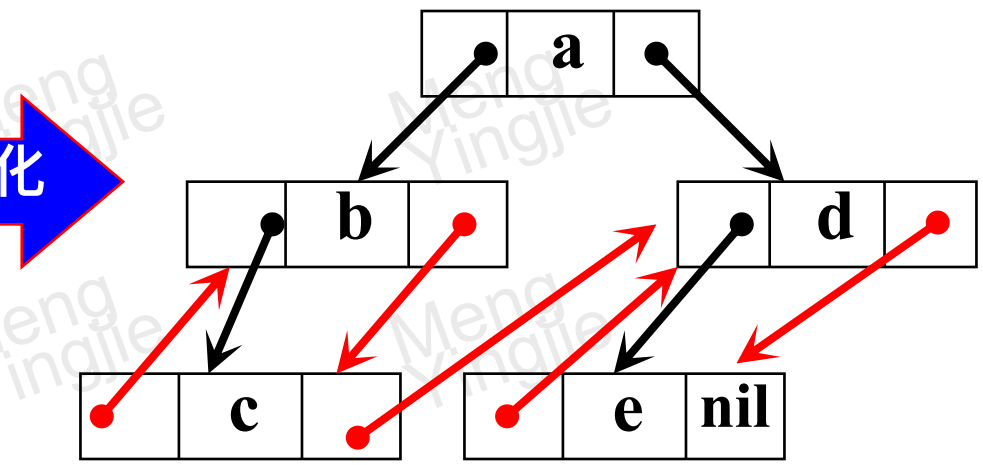
tag= { 0,表示该指针指向孩子。
1,表示该指针为线索，指向前驱或后继。

举例1-1： 前序线性序列：abcde

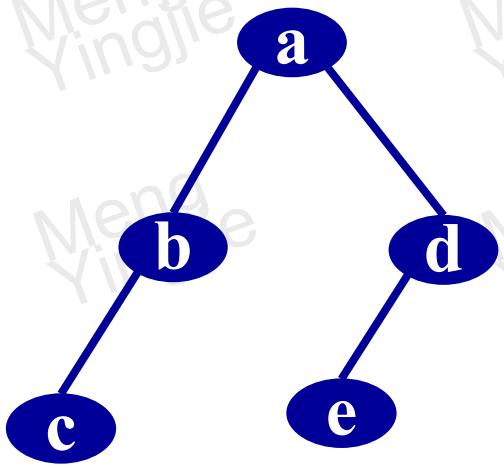


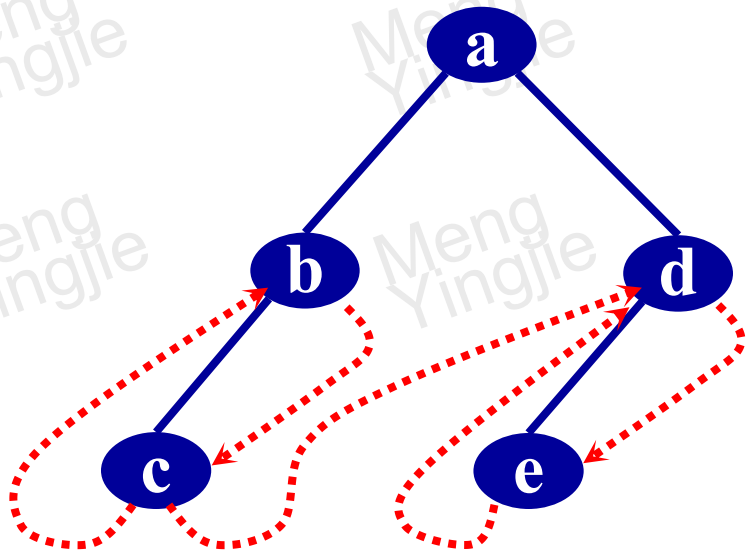
前序线索化

前序线索树

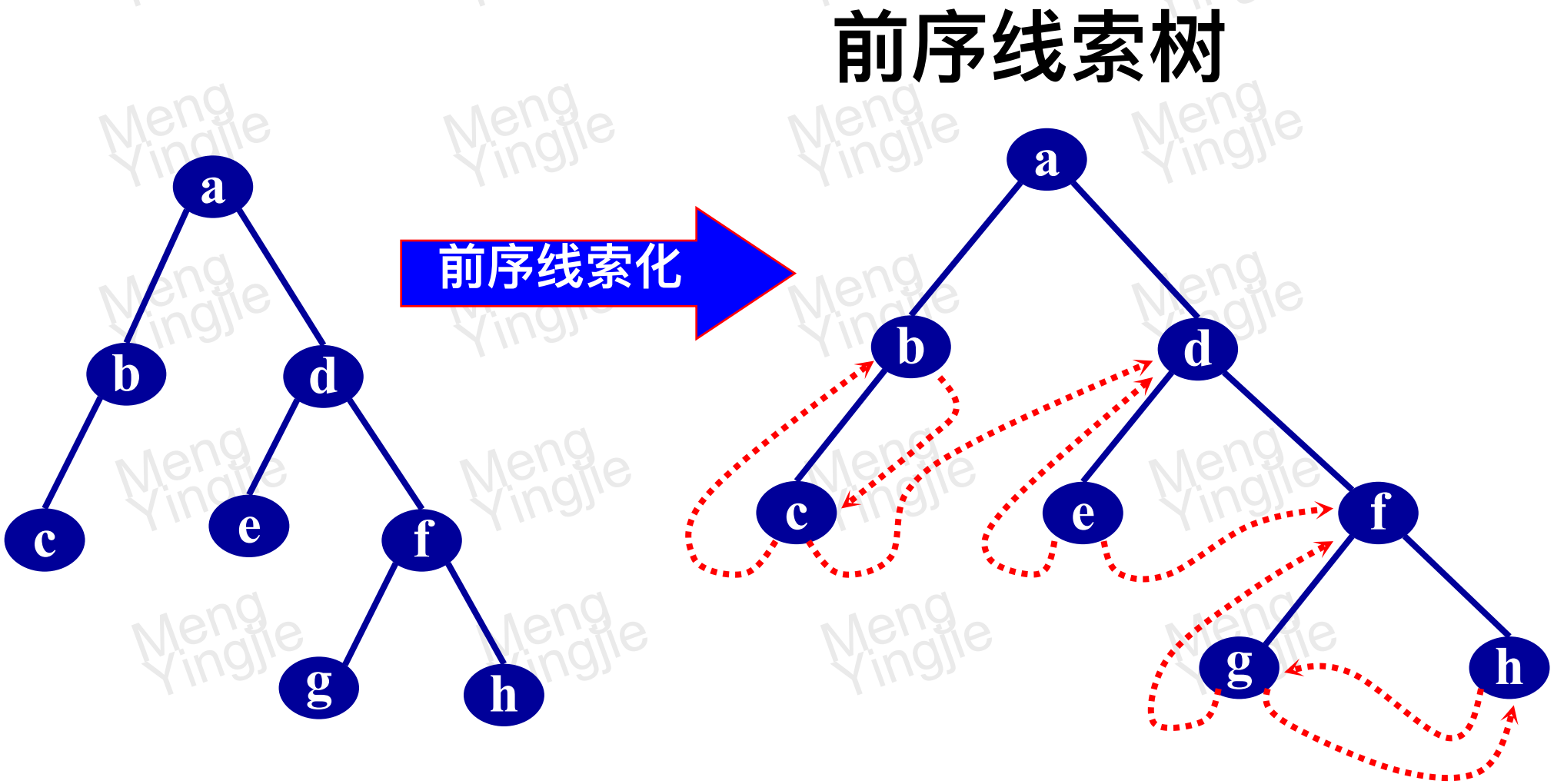


简化表示

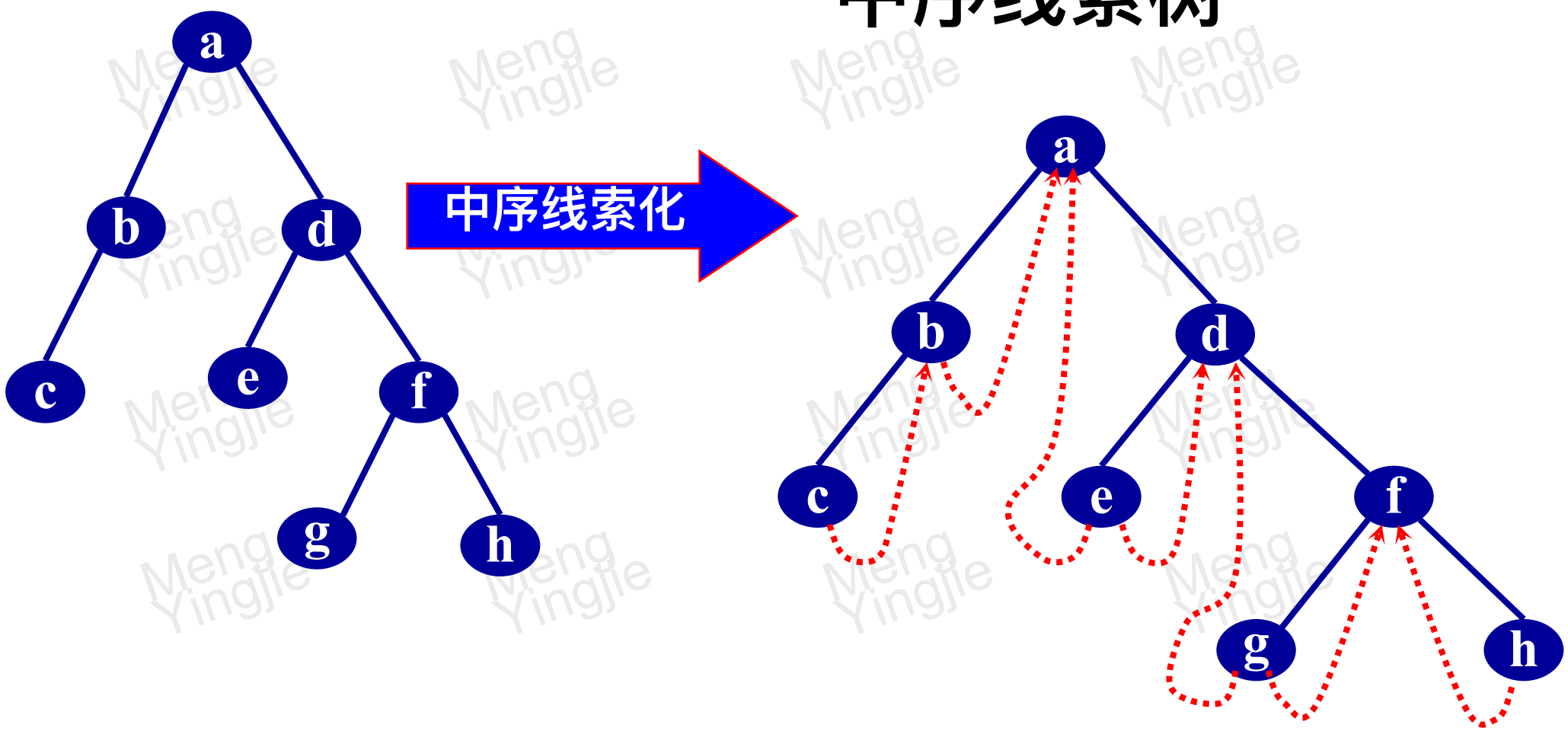




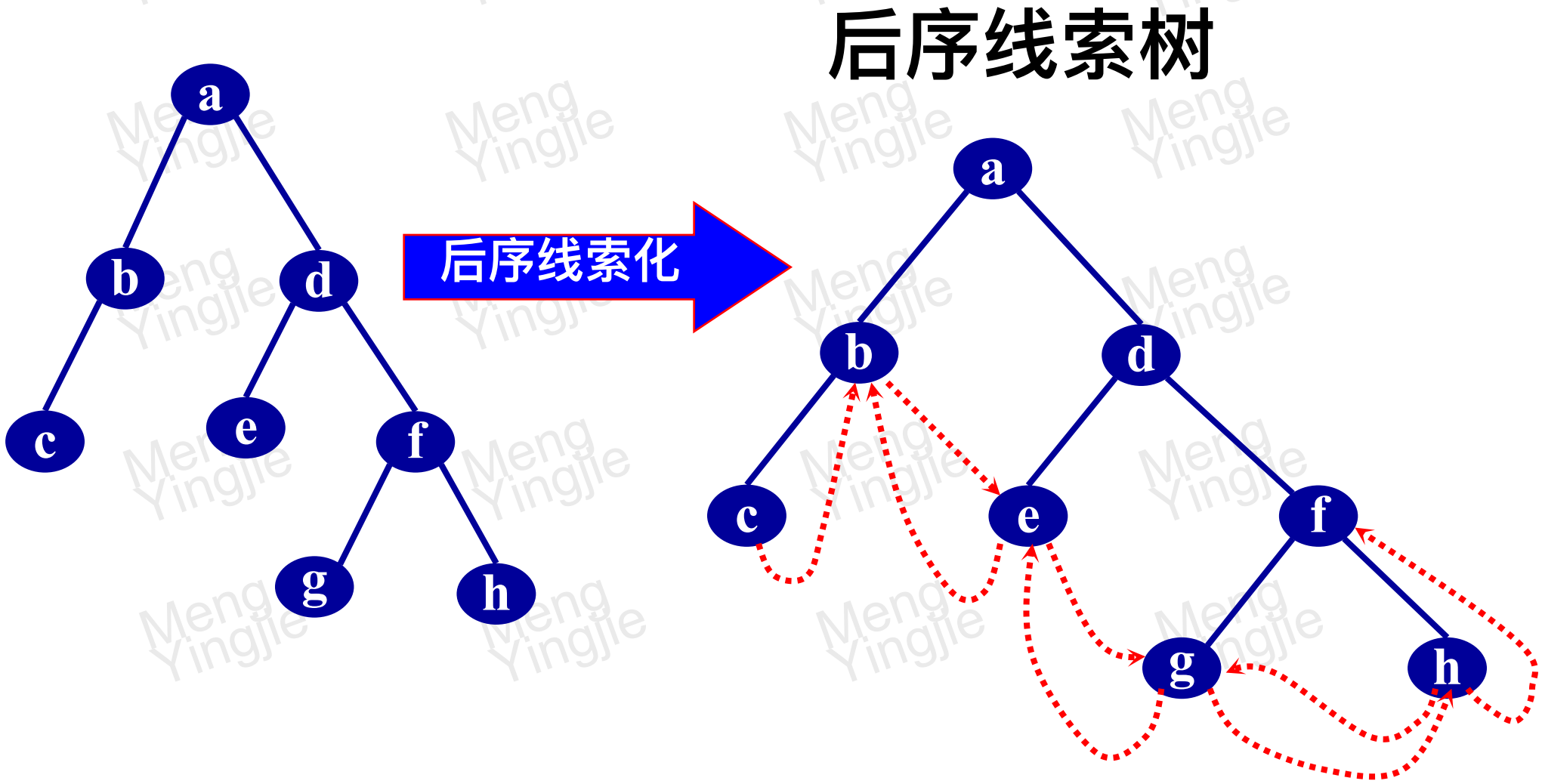
举例1-2：前序线性序列：abcdefgh



举例2：中序线性序列：cbaedgfh

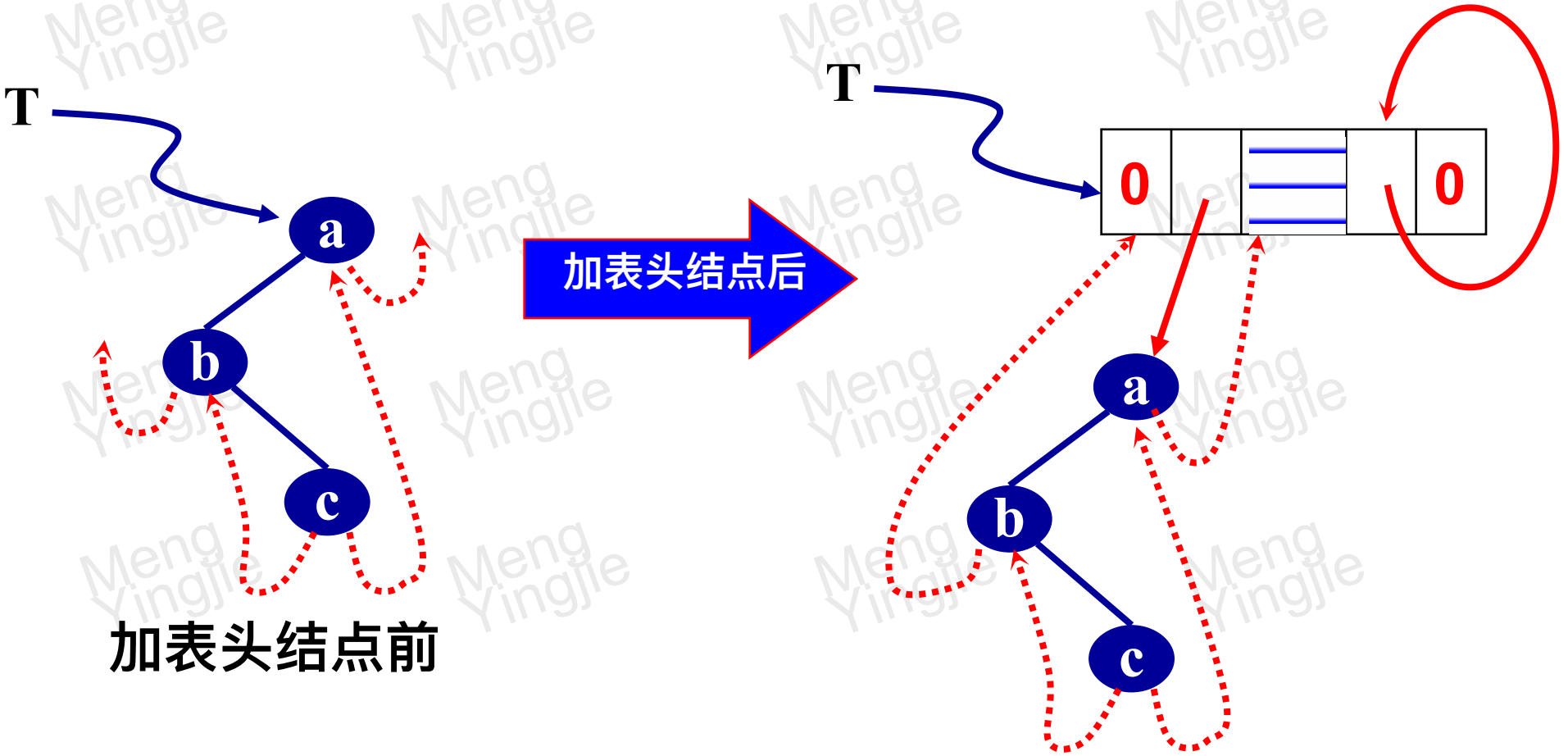


举例3：后序线性序列：cbeghfda



线索树的改进：

可以将链表中的表头结点推广到线索树中。即可以给线索树增加一个表头结点。以中序线索树为例：



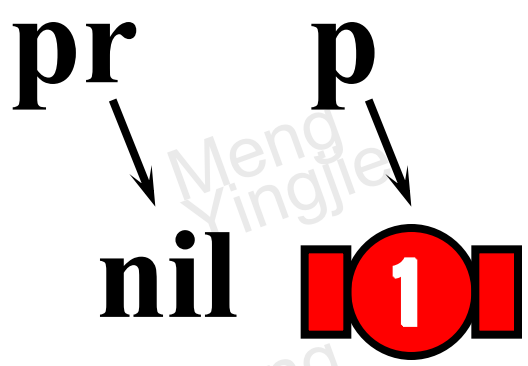
三.按中序线索化二叉树

先不考虑带表头结点情况。需要在遍历的过程中将空的指针修改为线索，因此可将遍历算法中的访问过程改为加线索过程即可得到线索化的算法，但需要使用非递归过程。

递归算法无法随即获取当前处理结点的前一结点，即直接前驱，故需用非递归方式实现。

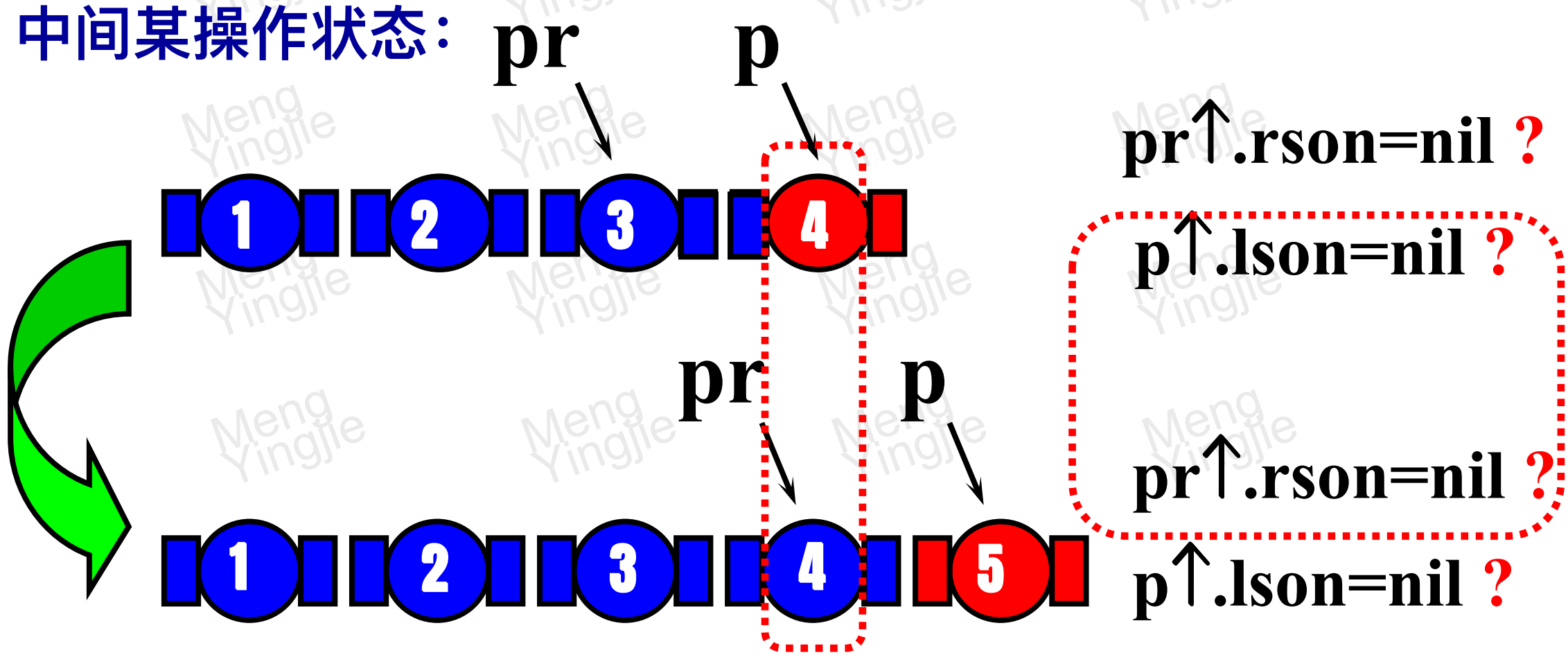
具体理由分析。。。。

过程分析：设 Pr 为待访问结点P的**直接前驱结点**。初始状态如下：



首结点无前驱，不用加线索；

中间某操作状态：



按中序线索化算法过程

```
PROC  threaded(VAR T: BinaryTree);
VAR  S: ARRAY[1..max] OF BinaryTree;
BEGIN   i←0;   p←T;   pr←nil;
   REPEAT
      WHILE p≠nil DO 【 i←i+1;  S[i]←p;   p←p↑.lson】;
      IF i≠0 THEN 【p← S[i];   i←i-1;
                                  IF pr≠nil
                                  THEN 【IF pr↑.rson=nil THEN pr↑.rson ← - p;
                                        IF p↑.lson=nil THEN p↑.lson← - pr; 】;
                                  pr ← p;
                                  p ←p↑.rson】;
   UNTIL (i=0) AND (p=nil)
END;
```

访问P

四.中序线索树中寻找结点后继

寻找结点P的后继结点，结果存放于Q。

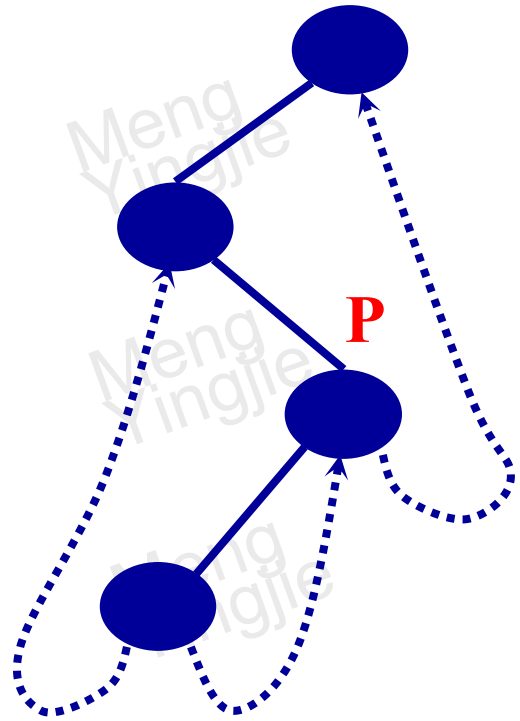
分析：根据中序P的后继应当在其右链当中

1.若P右子树为空(依据条件： P↑.rtag=1)

此时， P↑.rson即为其直接后继

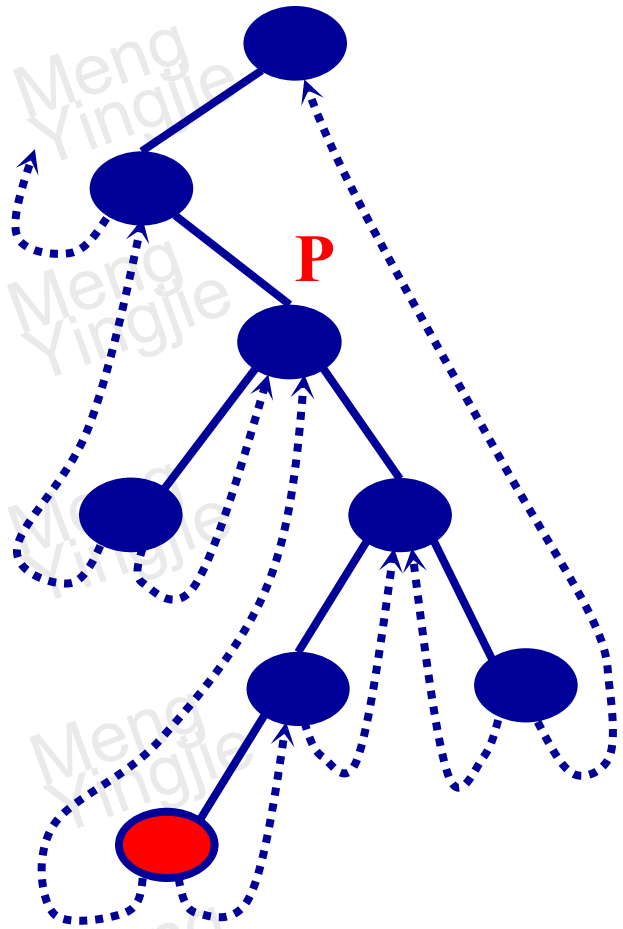
处理动作：

P↑.rson => Q



2.若P右子树非空（依据条件： P↑.rtag=0 ）

表明P有真正的右子树，此时，P的直接后继应当在其右子树的沿着左链搜索到的第一个 P↑.ltag=1 的结点。



处理过程：

(1) 搜索初始化

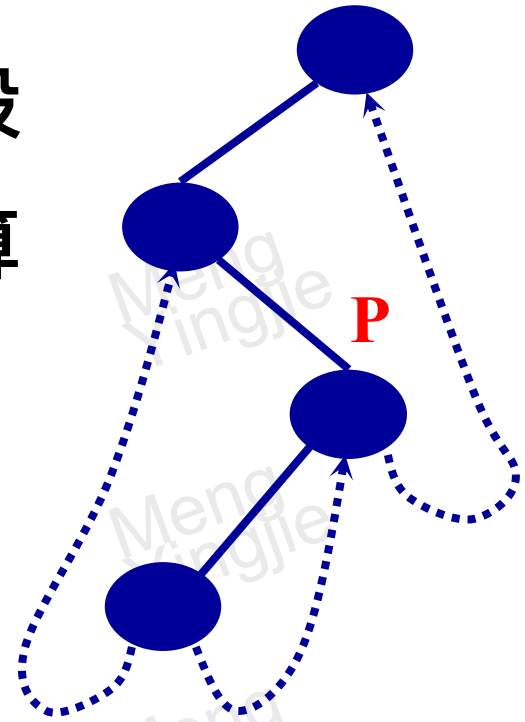
q ← p↑.rson;

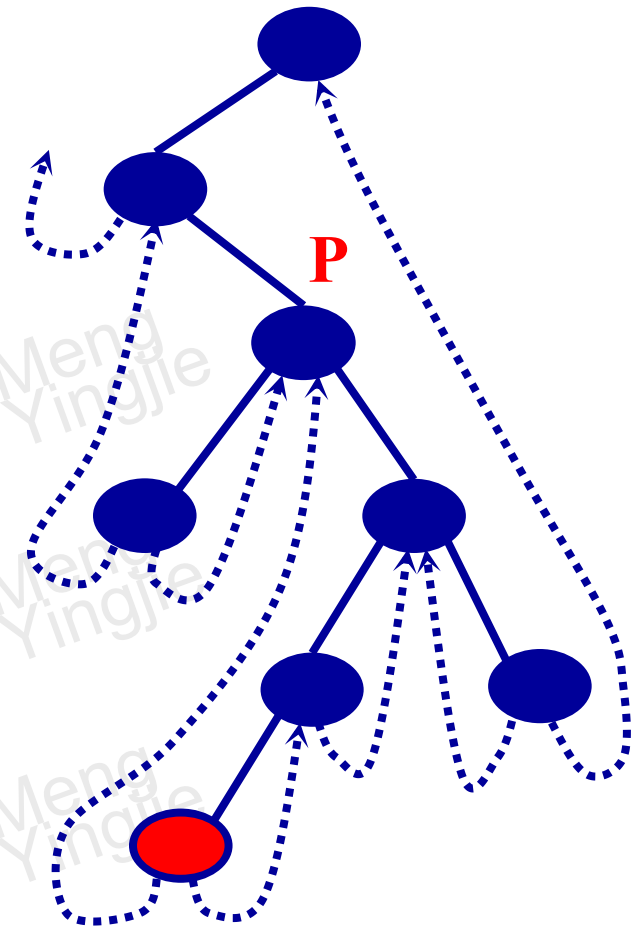
(2)搜索

WHILE q↑.ltag=0 DO

q<=q↑.lson

综合以上两种情况，可设计寻找结点P的后继结点的算法，结果存放于Q。

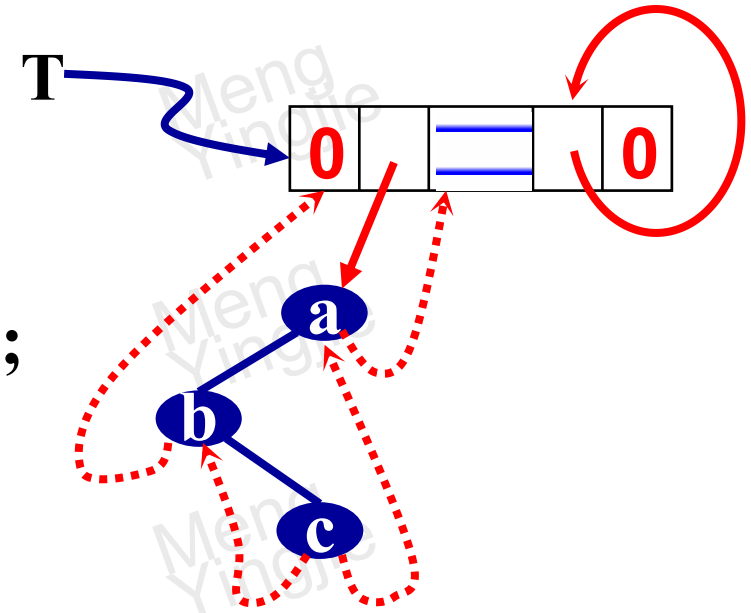





```
PROC   INSUC(VAR P,Q: BinaryTree);
BEGIN   q←p↑.rson;
        IF p↑.rtag=0 {说明有真正的右子树}
                THEN   WHILE q↑.ltag=0  DO q←q↑.lson
END;
```

类似可以写出寻找前驱的算法(左子树的右链中寻找)。

改进：增加表头结点后的算法（寻找结点P的后继结点，结果存放于Q）




```
PROC  INSUC_Head(VAR P,Q: BinaryTree);
BEGIN {排除空树}
      IF p↑.lson=p THEN 【q←p;   exit】 ;
     {查找，后继应当在其右链中}
       q←p↑.rson;
       IF p↑.rtag=0 {说明有真正的右子树}
     THEN  WHILE  q↑.ltag=0  DO q←q↑.lson
END;
```

五.按照中序规律遍历中序线索树

通过遍历我们引入了线索,反过来我们再讨论带线索的二叉树的遍历。

基本方法：

(1)寻找二叉树T全局第一个被访问的结点P

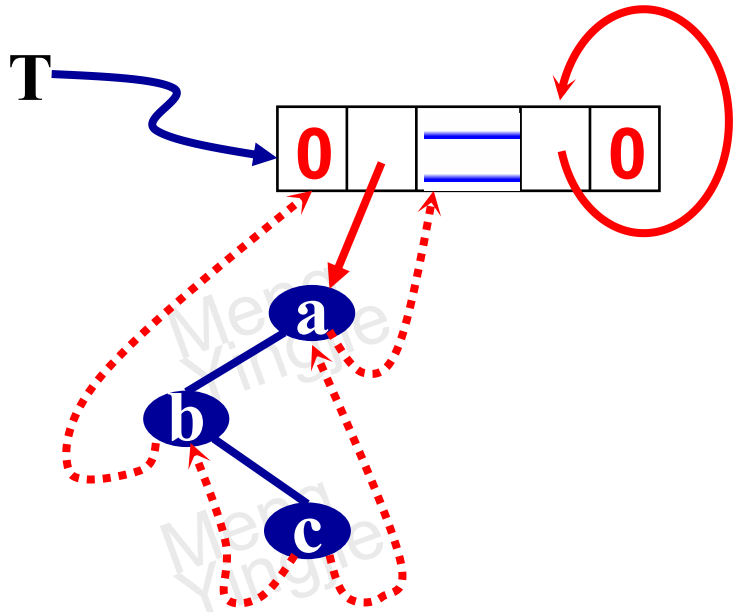
(2)访问P

(3)寻找P的直接后接结点Q，Q=>P,

(4)重复(2)–(3)直到P=nil

关键点：利用寻找直接后继的算法。

中序遍历中序线索二叉树（带表头结点的）的算法设计:




```
PROC   TINORD(VAR T: BinaryTree);
    {T为带表头的中序线索树}
BEGIN  CALL  INSUC(T,P);
       WHILE p≠T  DO 【write(p↑.data);  call  INSUC(P,P) 】
END;

PROC   INSUC(VAR P,Q: BinaryTree);
BEGIN   IF p↑.lson=p THEN 【q←p;   exit 】 ;
        q←p↑.rson;
         IF p↑.rtag=0 THEN  WHILE q↑.ltag=0  DO q←q↑.lson
END;
```

六.中序线索树中插入结点

既然在线索树中存在线性次序，插入一个结点后只要保证线索正确即可。

约定插入方法：将要插入的结点T作为结点S的右孩子。

（分析：相当于在单链表中S之后插入一个T结点，依据中序次序自然T只能是S的右孩子）

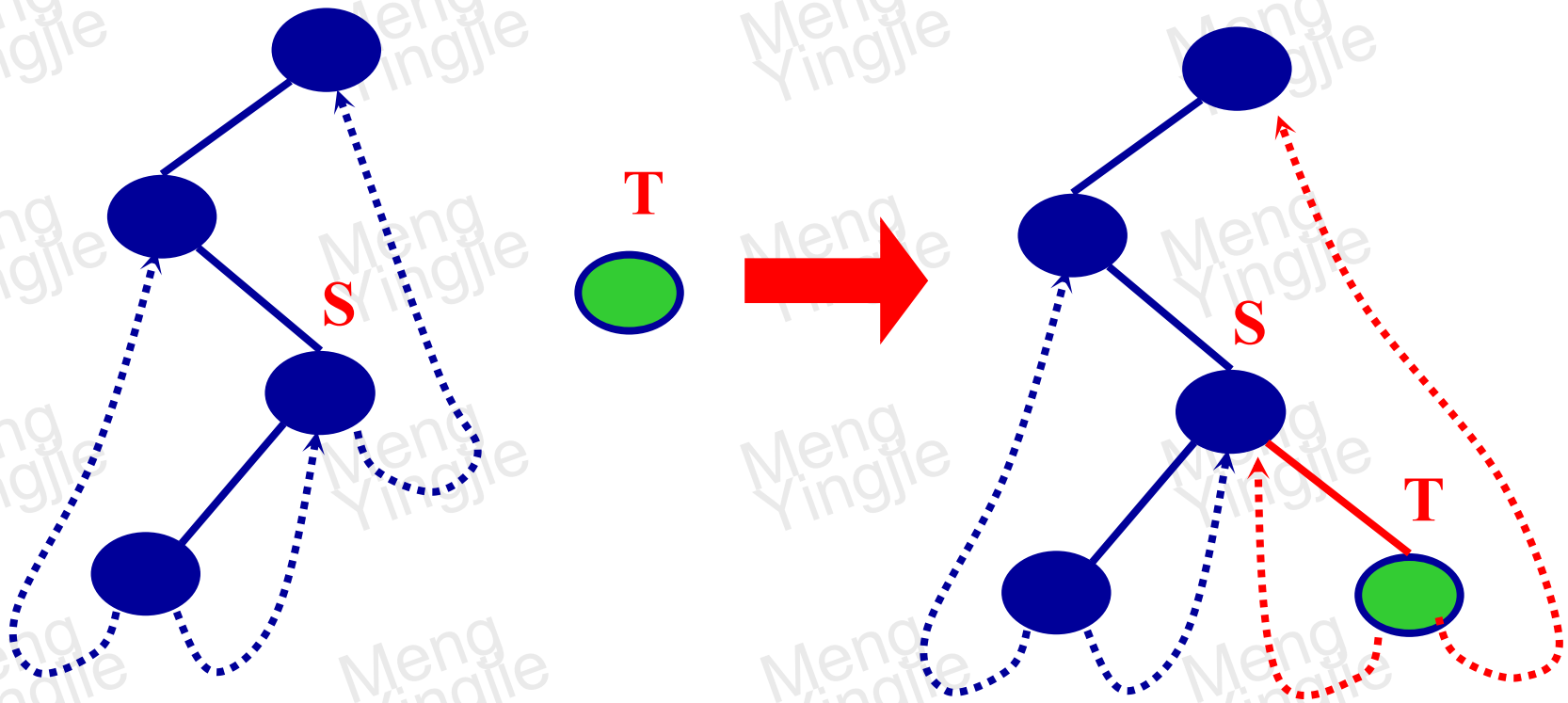
分情况讨论：

1.若S右子树为空，此时，S的右链应为线索

2.若S右子树非空，此时，S的右链应为真正的右孩子

1.若S右子树为空，依据条件： S↑.rtag=1

直接插入：修改S、T的相应指针即可。



动作: 从上图可以看出须进行3个数据项的变更：

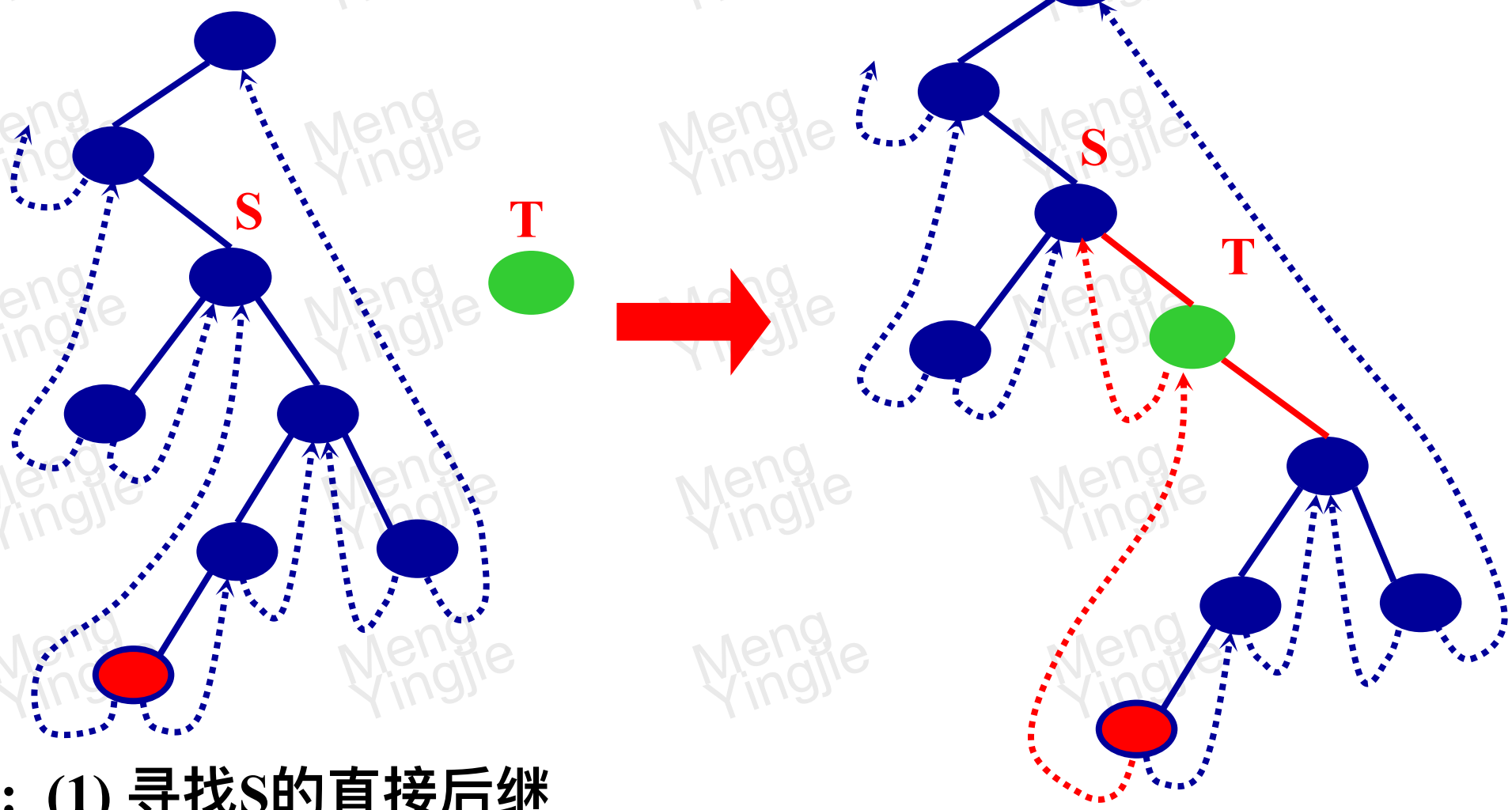
线索移动,相当于T右指针线索化 (1) S↑.rson => T↑.rson， S↑.rtag=> T↑.rtag

加线索,相当于左指针线索化 (2) S =>T↑.lson ， 1=> T↑.ltag

真孩子链接 (3) T =>S↑.rson ； 0=> S↑.rtag

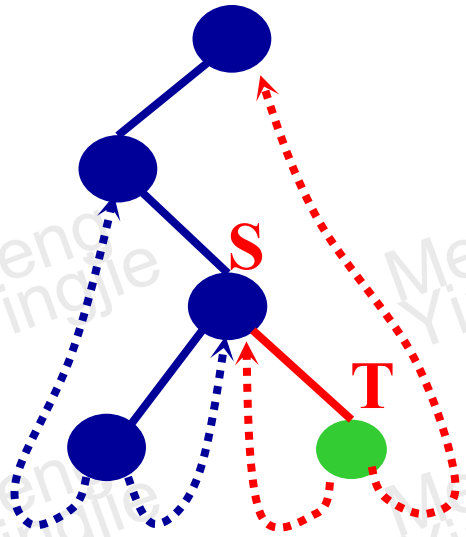
2.若S右子树非空，**依据条件**： S↑.rtag=0

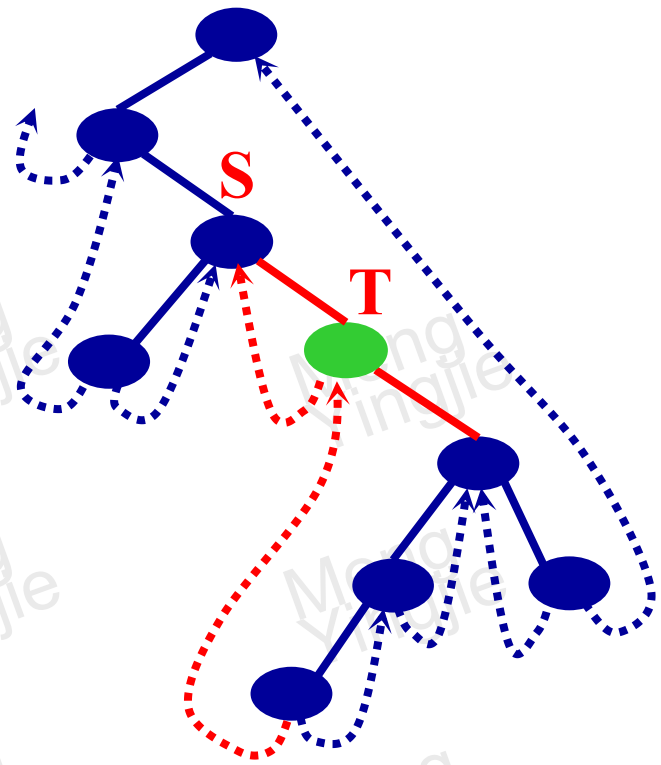
不仅要修改S、T的指针还要修改S的直接后继结点的指针。涉及 3 个结点。



过程：(1) 寻找S的直接后继

(2) 修改相应指针，进行4个数据项的修改






```
PROC  insrson(VAR T,S: BinaryTree);
BEGIN  T↑.rson←S↑.rson;    T↑.rtag←S↑.rtag;
       T↑.ltag←1;          T↑.lson←S;
       S↑.rtag←0;          S↑.rson←T;
       IF T↑.rtag=0 THEN 【CALL INSUC(T,Q);  Q↑.lson←T 】
END;
```

这里只讨论了插入的一种情况，其它情况可类似讨论。

本节结束

树形结构是一种应用非常广泛的非线性结构，除组织目录(即检索)外，在许多问题中常常采用树形结构作为中间结构以求解问题、确定对策。

主要讨论：

- 遍历的方法的应用;
- 二叉排序树;
- 霍夫曼树;
- 分类与判定树等

一.遍历的运用

利用遍历方法可以求解许多问题。只需要对遍历算法稍加改变就可以，以下仅举几个例子。

1 .交换二叉树的左右子树。

```
PROC  exchange (VAR T: BinaryTree);
BEGIN
   IF  T≠nil  THEN  【swop(T↑.lson,T↑.rson);  {交换T的左右子树}
                    call  exchange(T↑.lson);
                    call  exchange(T↑.rson)】
END;
```

2.求二叉树的高度:

(1)采用一般过程进行设计

```
PROC   high(VAR T: BinaryTree;h:integer);
BEGIN
   IF T=nil THEN h←0
          ELSE 【call high(T↑.lson,h1);
                call high(T↑.rson,h2);
                h←1+max(h1,h2)】
END;
```

(2)采用函数过程进行设计

——用函数过程最恰当

```
FUNC   high(VAR T: BinaryTree) : integer;
BEGIN
   IF  T=nil  THEN  high←0
              ELSE  high← 1+max(high(T↑.lson), high(T↑.rson))
END;
```

3.统计二叉树的叶子结点个数:

(1)采用一般过程进行设计

```
PROC  counleaf(VAR T: BinaryTree;  count:integer);
BEGIN
      CASE
      T=nil :  count←0;
      T↑.lson=nil AND T↑.rson=nil) : count←1;
      ELSE 【    call  countleaf(T↑.lson,c1);
                 call  countleaf(T↑.rson,c2);
                 count←c1+c2  】
      ENDCASE
END;
```

(2)采用函数过程进行设计

```
FUNC  Counleaf(VAR T: BinaryTree):integer;
BEGIN
      CASE
      T=nil：  Counleaf←0
      T↑.lson=nil AND T↑.rson=nil：  Counleaf←1;
      ELSE
          Counleaf←countleaf(T↑.lson)+countleaf(T↑.rson)
      ENDCASE
END;
```

二.二叉排序树(binary sort tree)

是线性有序表的二叉树表示，也是检索中最常用的工具之一。

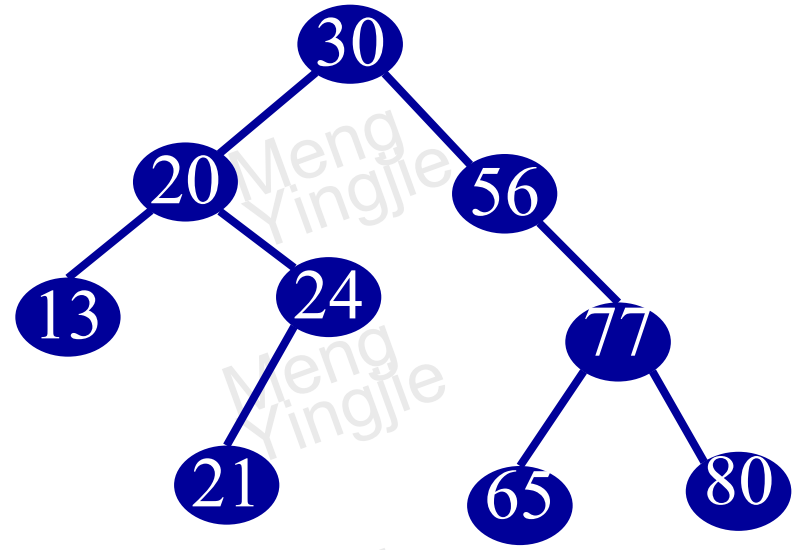
1.定义：

二叉排序树或者是空二叉树，或者是满足如下性质的二叉树：

- 若它的左子树非空，则左子树上所有结点的值均小于根结点的值；
- 若它的右子树非空，则右子树上所有结点的值均大于根结点的值；

左子树、右子树本身又是一棵二叉排序树。

2.特点：



非线性结构表示了一个线性有序表。

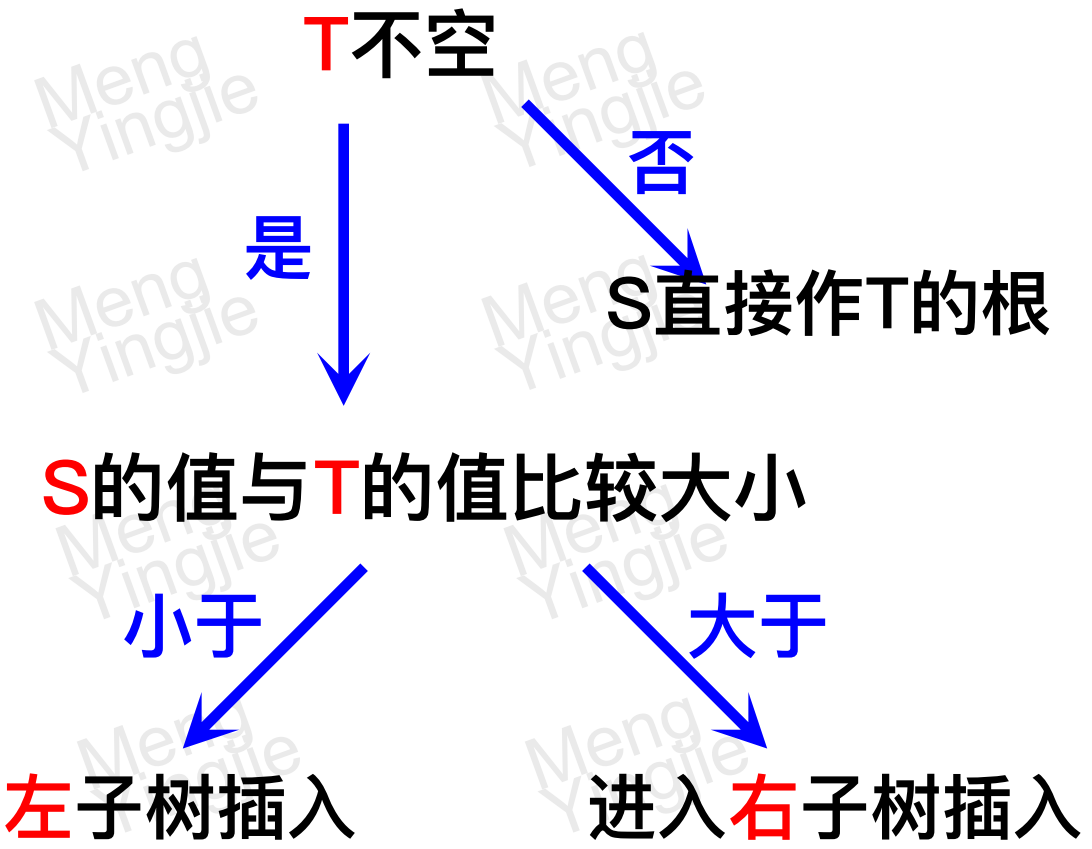
根据二叉排序树的定义，按中序遍历该二叉树，所得到的中序序列是一个线性有序序列（升序或降序）。

从存储结构上来讲它比向量要多占用存储空间，但从其构造过程可以发现其优点。

3.构造二叉排序树：

依据前面二叉排序树的定义我们先讨论在二叉排序树中插入一个结点的基本方法。假设：将S插入到二叉排序树T中。

常规情况：



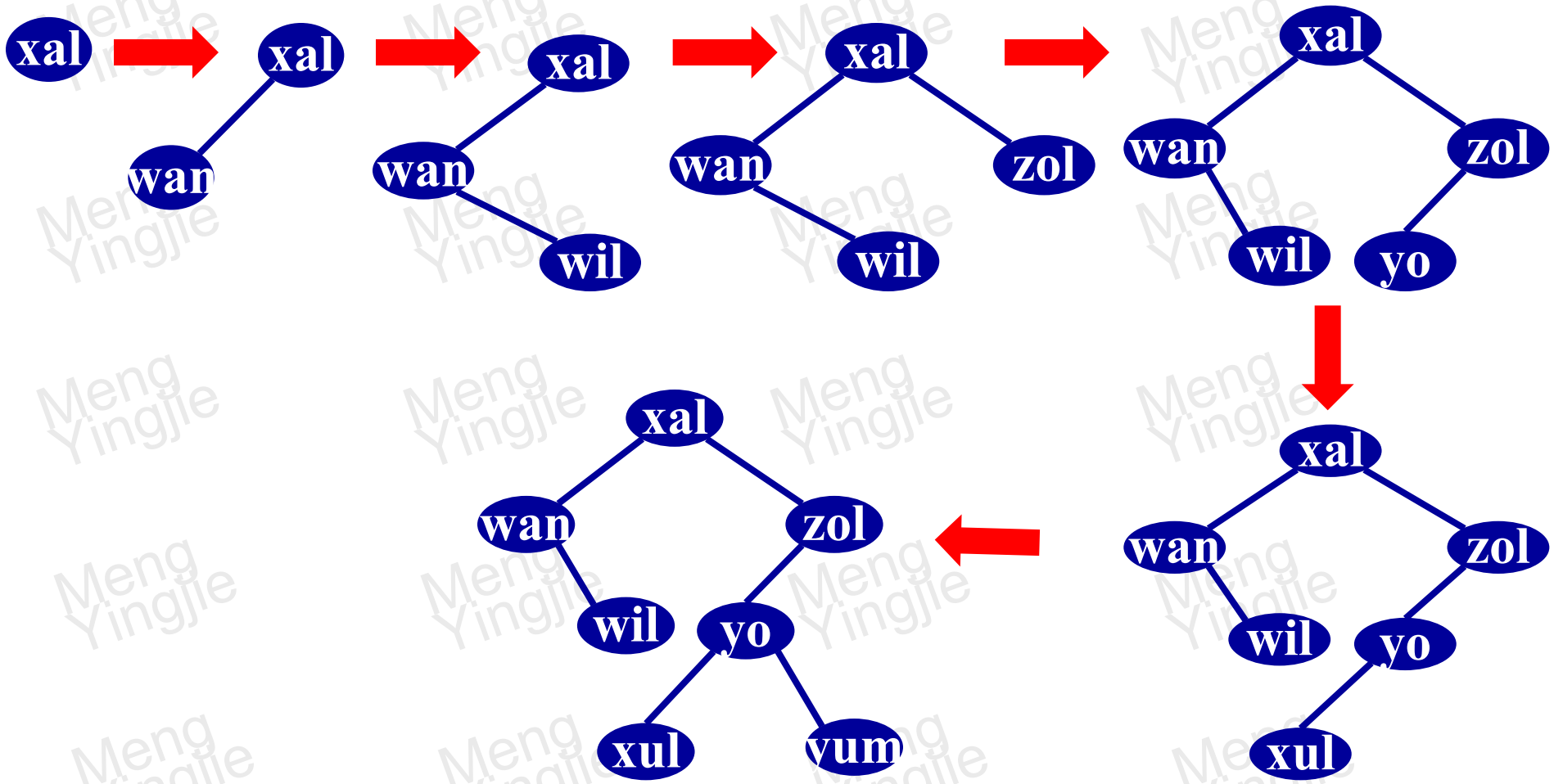
边界情况：

左/右子树可以利用二叉树定义的递归性进行引用！即方法雷同.

基于上面的分析，我们给出将一个结点S插入到二叉排序树T中的过程：

```
PROC  InsertNode(VAR T,S: BinaryTree);
BEGIN
  IF T=nil THEN T←S
  ELSE IF (S↑.data < T↑.data)
       THEN call InsertNode(T↑.lson,S)
       ELSE call InsertNode(T↑.rson,S)
END;
```

例：有一个关键字集合K={xal,wan,wil,zol,yo,xul,yum,wen,wim}对其构造一棵二叉排序树。



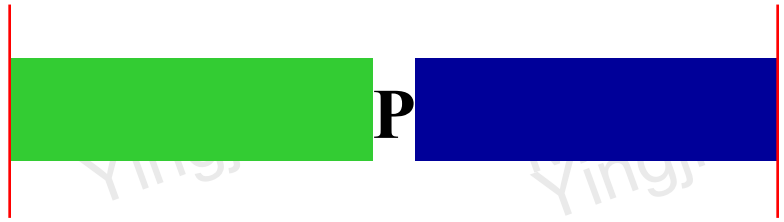
特点：沿深度方向每次增加了一个叶结点，不需要移动元素

从键盘输入m个值，建立一棵二叉排序树T.

```
PROC  SortTree(VAR T: BinaryTree);
BEGIN
      T←nil;
      FOR  i=1  TO  m  DO
           【 NEW(S);
           read( S↑.data);
           S↑.lson←nil;
           S↑.rson←nil;
           call  InsertNode(T,s)  】
END;
```

4.二叉排序树中删除结点：

先考虑线性有序序列删除情况：例如，删除P：

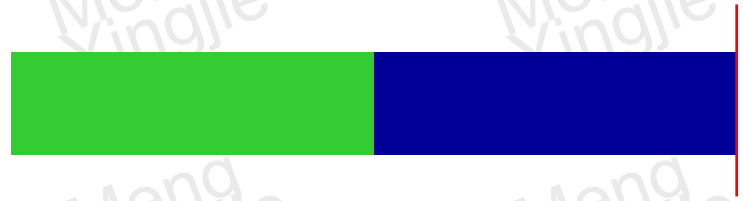


将P删除后，为了保证序列的连续性需移动元素，可用两种方法解决：

①将P之后元素前移：

相当于用P的中序后继代替P

②将P之前元素后移



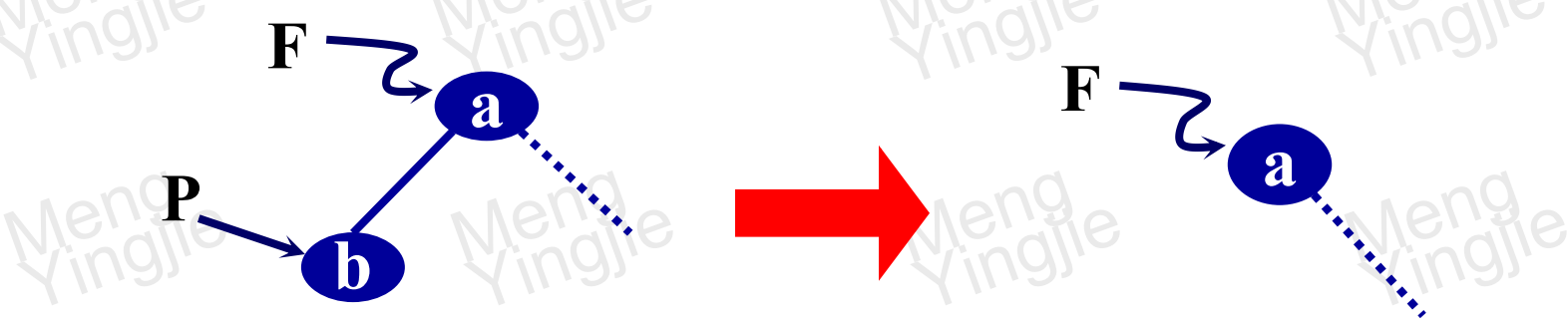
相当于用P的中序前驱代替P

关于结点后继已在线索树部分讨论过，此处讨论：用P的中序前驱代替P的情况。

设P为删除结点,其双亲结点为F，我们讨论用P的中序前驱代替P的情况。

围绕二叉树的五种基本形态讨论，概括起来可有3种情况：

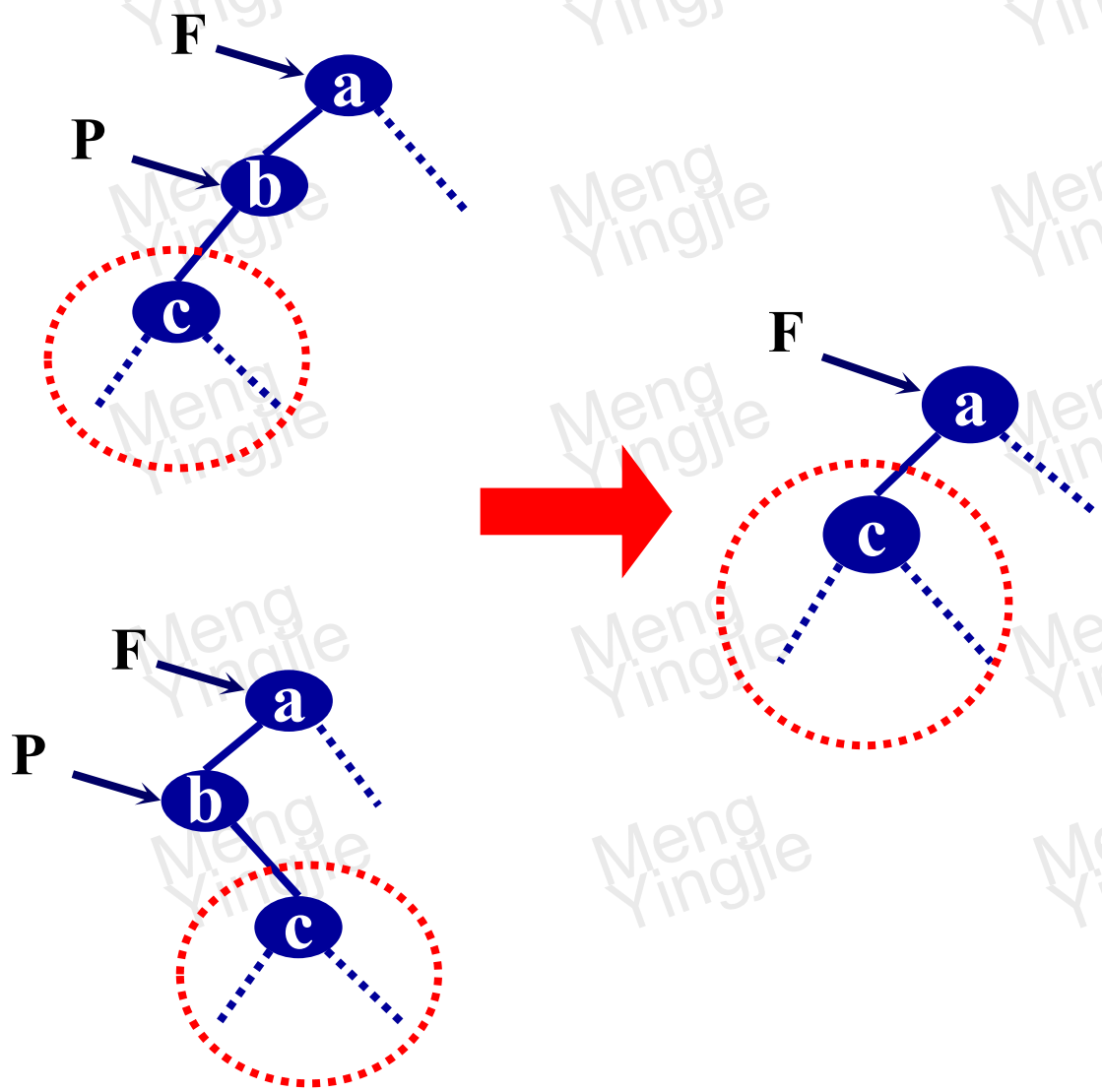
①P无左右孩子，只需修改双亲指针即可。



动作：

nil => F↑.lson

②P只有左孩子或右孩子，只需让P的左或右孩子代替P即可。



p↑.lson≠nil, 动作：

p↑.lson => F↑.lson

p↑.rson≠nil,动作：

p↑.rson => F↑.lson

③P左孩子、右孩子均不空，需要寻找P的前驱结点S(在P的左链的右链中)代替P,并修改相应指针修改即可。

a.如何寻找P的前驱结点S(与线索树部分类似)

S的左子树可能为空，也可能不空，但其右孩子一定为空，为了能够顺利摘去S，还需要在搜索S的过程中为其增加一个后拖变量Q。

为了处理S的左子树搜索时需设置后拖变量Q

初始：q←p;　S←p↑.lson;

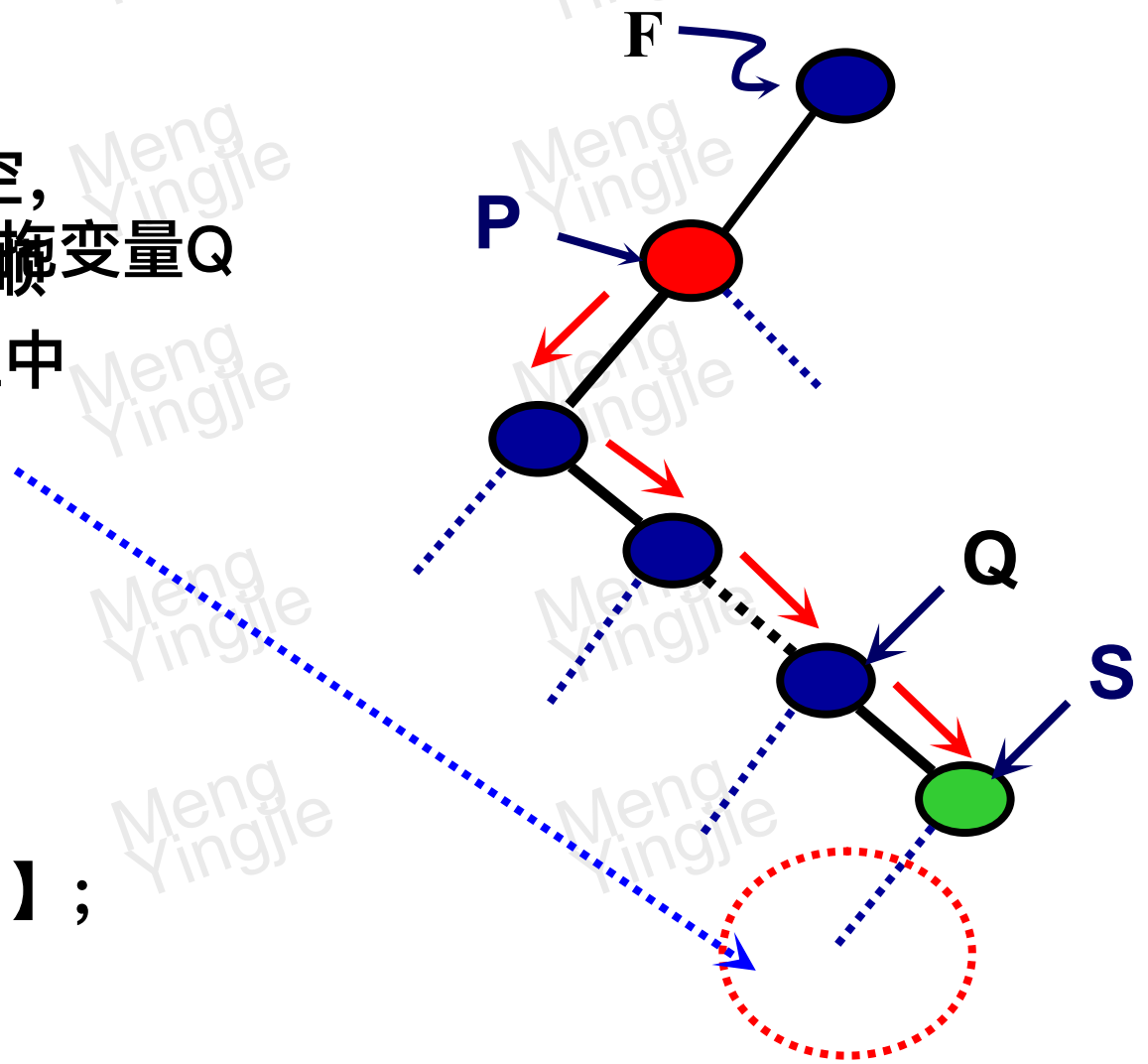
搜索：

While (S↑.rson≠nil) DO

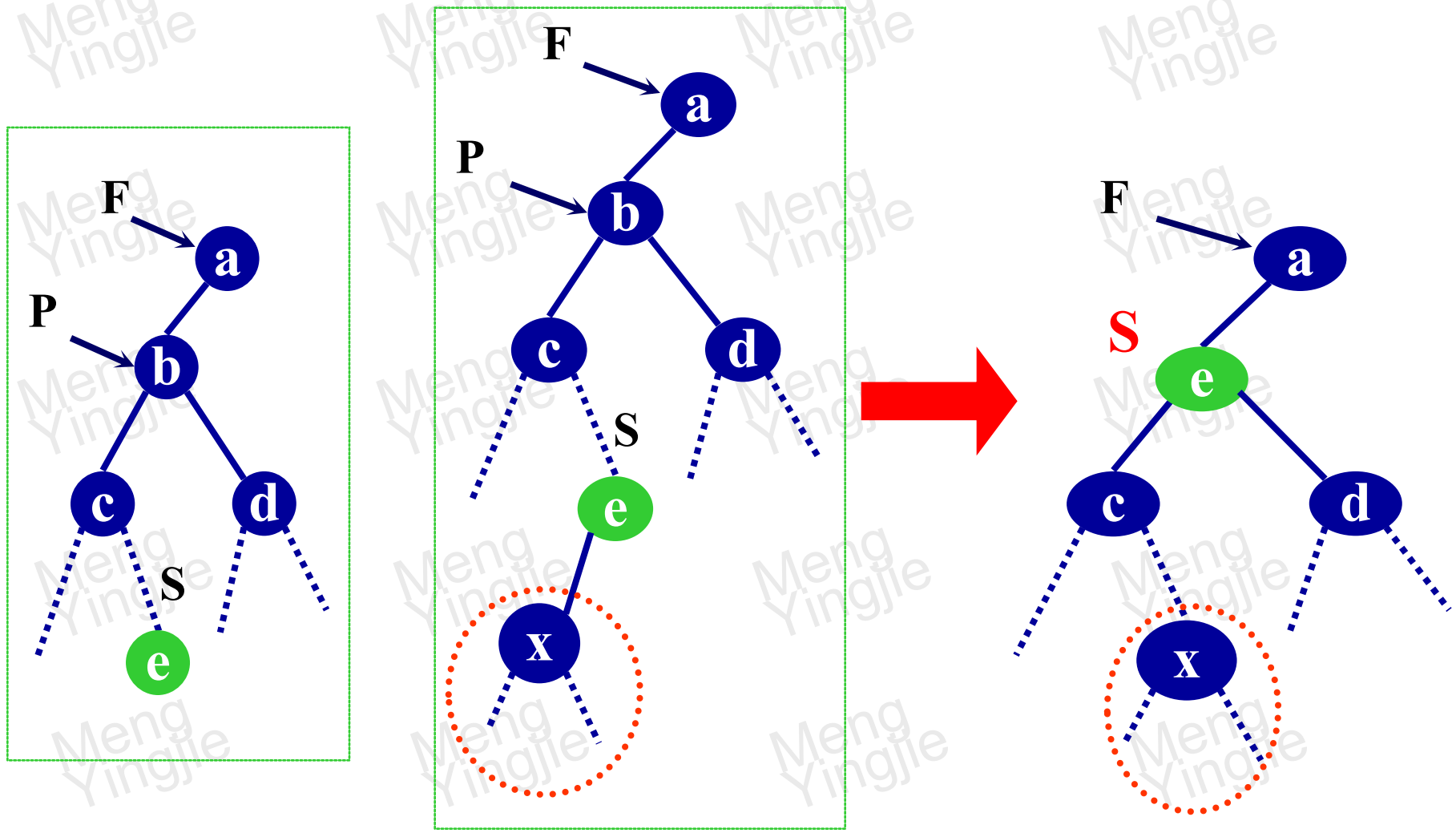
【 q ← S;　S ← S↑.rson 】;

S左子树进行挂接:

Q↑.rson ← S↑.lson



b.找到P的前驱结点S并修改相应指针后，用S代替P



整个过程注意：搜索S时，S需要带一个后拖变量始终记载S的父结点。

设S为要代替删除点P的结点。根据以上分析的三种情况删除过程的算法设计如下：

```
PROC  DeleteSortTree(VAR F,P: BinaryTree; );
BEGIN
    CASE
    P↑.lson=nil :   S←P↑.rson;
    P↑.rson=nil :   S←P↑.lson;
    ELSE 【q←p;    S←p↑.lson;
          while (S↑.rson≠nil) DO 【q←S;  S←S↑.rson 】;
          S↑.rson← P↑.rson;
          IF p≠q THEN 【q↑.rson←S↑.lson;  S↑.lson ←P↑.lson 】】
     ENDCASE;
     IF  P=F↑.lson THEN F↑.lson ←S
                   ELSE F↑.rson ←S
END;
```

删除也可按其它规则，只要保证二叉排序树性质即可。

三.霍夫曼树(最优二叉树)

背景:

符号表的组织(例如，编码，字典等组织)一般都采用树形结构，从使用效率来讲，就要求对符号表的检索达到最小平均长度，即构造的树形结构具有某种最小性，Huffman于1952年提出了一种能够解决这种问题的方法。

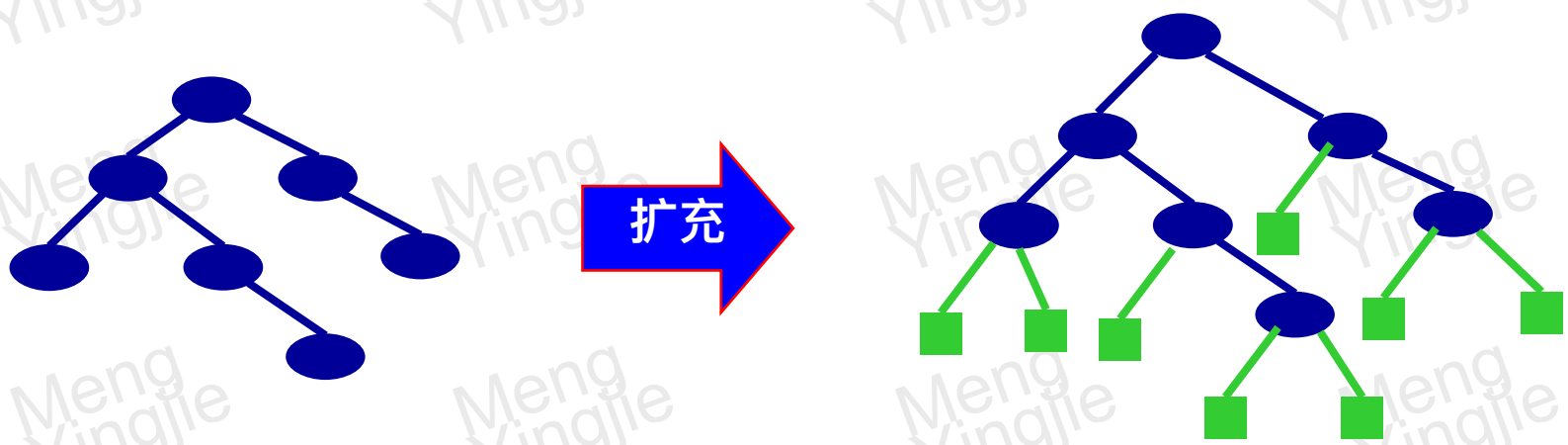
利用这种方法可实现非等长度下的最优编码，这种编码在数字图象数据处理中常用于压缩图象的数据量。

1.一般概念

两个结点之间的路径长度：从一个结点到另外一个结点之间的边(分支)数。

树的路径长度：从根到其它每个结点的路径长度之和。

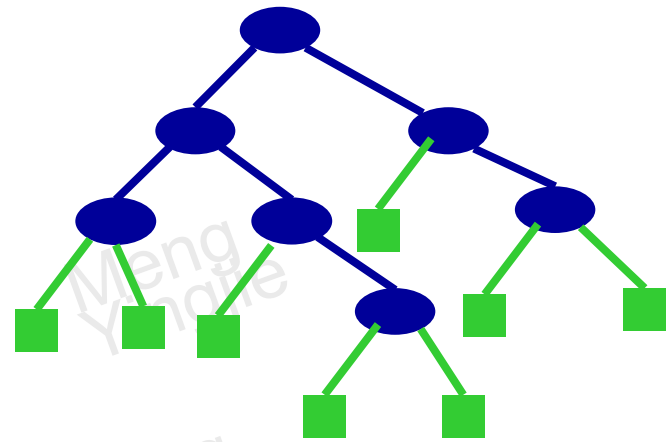
扩充二叉树(加长二叉树，2-树):对原二叉树进行扩充，使原二叉树的每个结点都带有左右两个孩子所得到的二叉树。



因此，我们可以讨论二叉树的内/外的路径长度之和问题,即，要使其达到最小化，二叉树应该是什么样的形态?

内结点:原二叉树结点。**外结点**:扩充结点。

2–树应用举例:在代数系统中,只有二元运算的情况下,内结点均为运算符,外结点均为操作数。在二分树检索中的失败结点。



扩充二叉树

扩充二叉树内路径长度:根结点到所有内结点的路径长度之和。

扩充二叉树外路径长度:根结点到所有外结点的路径长度之和。

实际应用中最有意义的是要求出路径长度最短的二叉树。

内外路径长度的关系：

设E(expansion)为2-树中的外路径长度，I(inside)为内路径长度，n为内结点个数。则有：$E=I+2n$ ，可用归纳法证明。

对二叉树而言，从根出发，
路径长度为0的结点只有1个；
路径长度为1的结点只有2个；
路径长度为2的结点只有4个；依次类推：
路径长度为k的结点只有2^k个，

所以，有n个内结点的2-树的内路径长度I_n至少为以下序列的前n项各数之和：

2个　4个　8个

0, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 4,…

该序列就是：$\lfloor\log_2 1\rfloor, \lfloor\log_2 2\rfloor, \lfloor\log_2 3\rfloor, \lfloor\log_2 4\rfloor, \ldots\ldots$

因对于 $2^t \le k < 2^{t+1}$,存在$\lfloor\log_2 1\rfloor=t$,所以可以得到：$I_n \ge \sum_{k=1}^{n} \lfloor\log_2 k\rfloor$

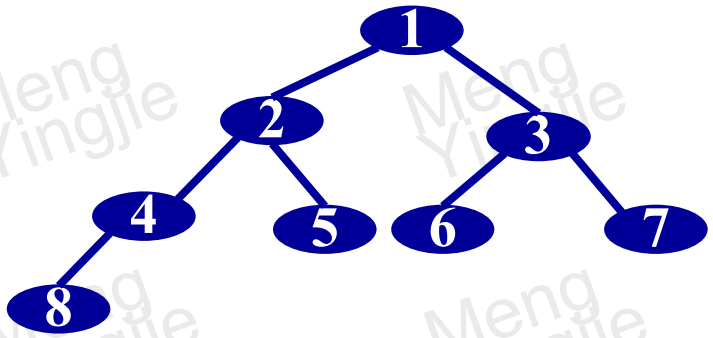
根据 $I_n \geq \sum_{k=1}^{n} \lfloor \log_2 k \rfloor$ 可得出最小内、外路径长度为：

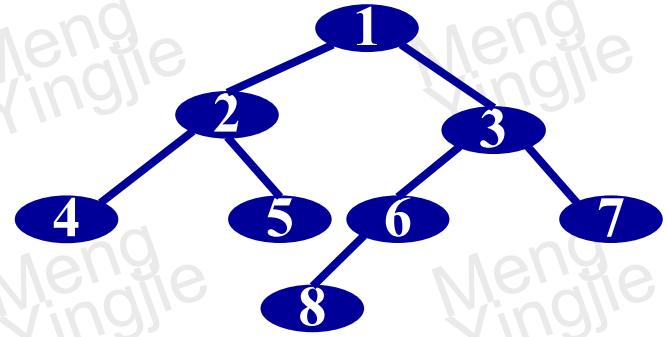
$$I(n)_{min} = \sum_{k=1}^{n} \lfloor \log_2 k \rfloor$$

$$E(n)_{min} = \sum_{k=1}^{n} \lfloor \log_2 k \rfloor + 2n$$

也就是说给定n个内结点，将其构造为顺序二叉树时，它具有最小的内、外路径长度。但具有最小内、外路径长度的二叉树不一定是顺序二叉树(也就是说不唯一)

例如，下面两个二叉树的内路径长度都为13。





2.加权(weighted)路径长度

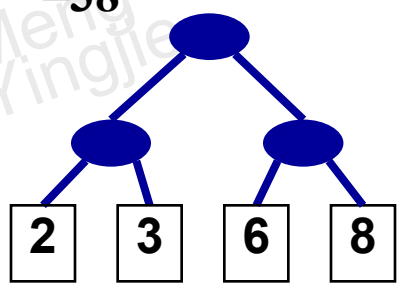
设T为n个外结点的扩充二叉树，且给每个外点$k(1\le k\le n)$赋予一个(非负)权值w_k，则加权路径长度WPL定义为：

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

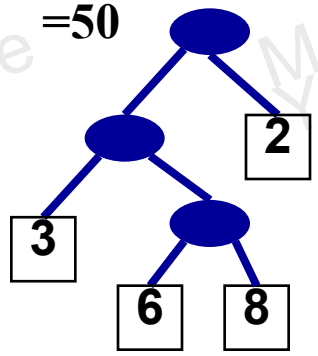
问题：如何得到具有WPL最小的二叉树?

等权值情况下，根据前面讨论(可看成外点权值为1)，顺序二叉树时最小，但在不等权值情况下，则不一定：

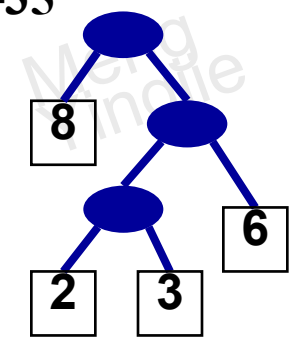
WPL=2(2+3+6+8)
=38



WPL=1*2+2*3+3(6+8)
=50



WPL=3(2+3)+2*6+1*8
=35



3.霍夫曼算法及霍夫曼树

Huffman最早提出了构造具有最小加权路径长度最小的二叉树的算法，此算法称Huffman算法，按此算法构造的具有最小加权路径长度的二叉树称Huffman树(或最优二叉树)。

Huffman算法的基本思想：

①给定一组权值集合$\{w_1, w_2, \ldots, w_n\}$.据此构成n棵二叉树组成的森林F。

注：F中的每棵二叉树只有一个带权值为$W_i(1\leq i\leq n)$的根结点。

②将$F=\{T_1, T_2, \ldots, T_n\}$按根结点的值由小到大进行排序。

③取出T_1和T_2组成一棵二叉树T；再将T插入到F中，并使F依据根结点的值有序。

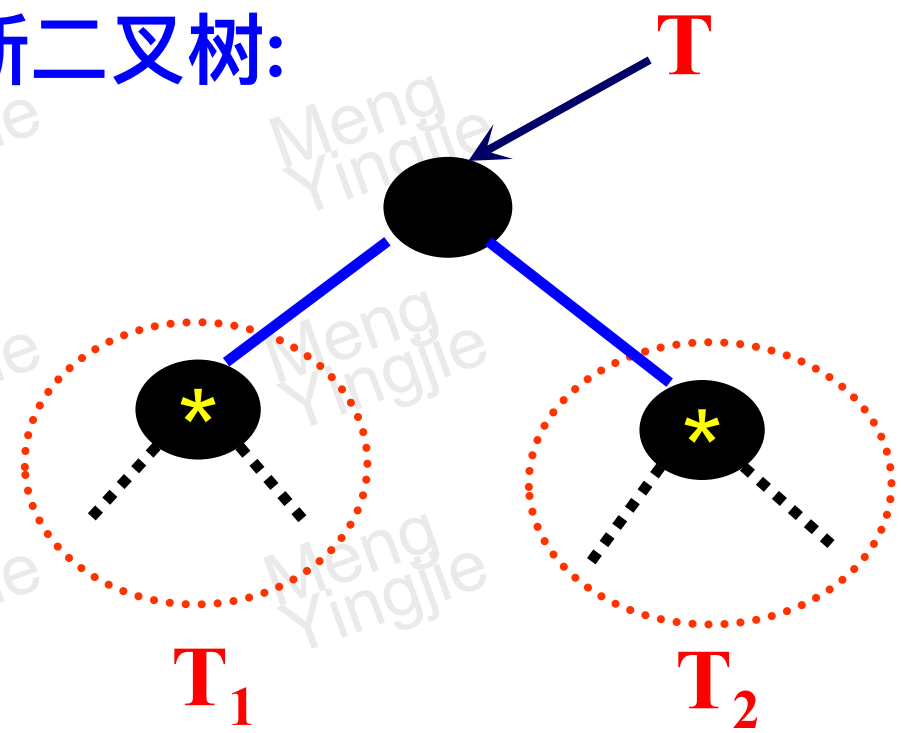
④反复执行③直到F={T}为止。

Huffman算法中的须要注意的事项：

第③步：取两个二叉树生成一个新二叉树:

T的根节点值由T_1、T_2和成：

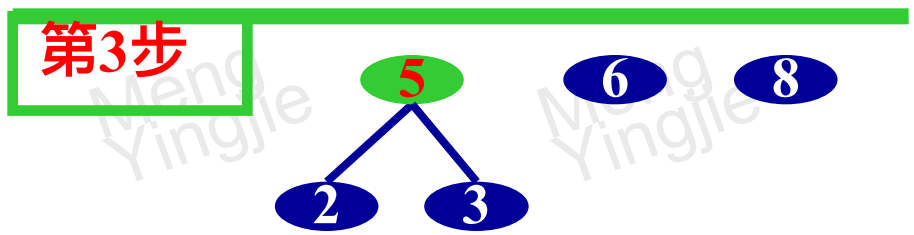
$T\uparrow.data \leftarrow T_1\uparrow.data + T_2\uparrow.data$

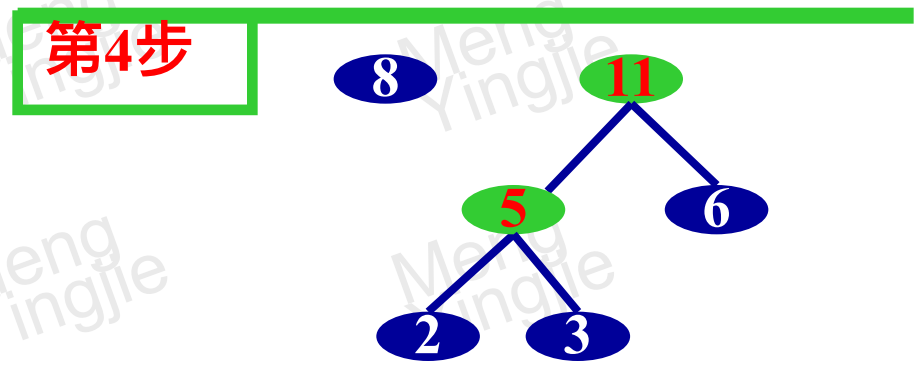


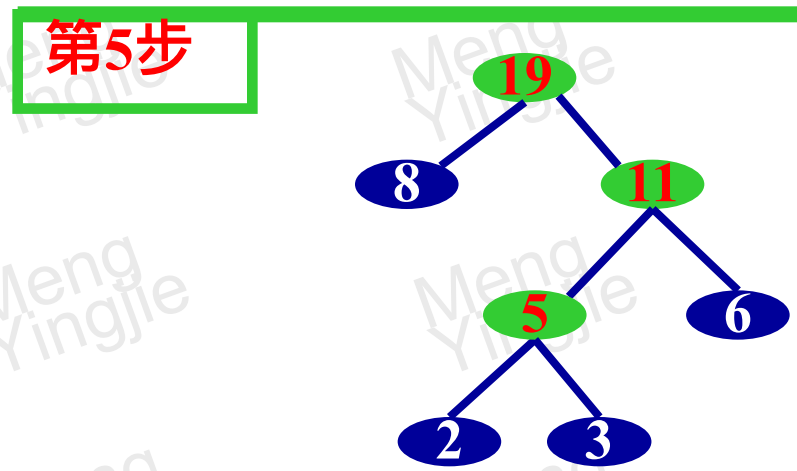
对于左右子树的分配，就路径长度值的大小计算来说，显然T_1、T_2谁作为左、右子树都无所谓，但落实到具体算法的设计，则必须明确，即，只能是以下取其一：

- T1作为左子树，T2作为右子树；
- T2作为左子树，T1作为右子树

举例： {3，2，6，8}构造Huffman树。







显然，权值越大的结点离根结点越近，即：加权路径越短。

4.霍夫曼算法的设计

存储结构的选取：记录数组

R

	1	2	…	n	n+1	…	2n-1
data	w_1	w_2	…	w_n			
lson	0	0	…	0			
rson	0	0	…	0			

辅助结构，用于存储排序结果：记录数组

A

	1	2	3	…	n
data					
adr					

存放每棵二叉树的根结点的值.

每棵二叉树的根结点在R中的下标值.

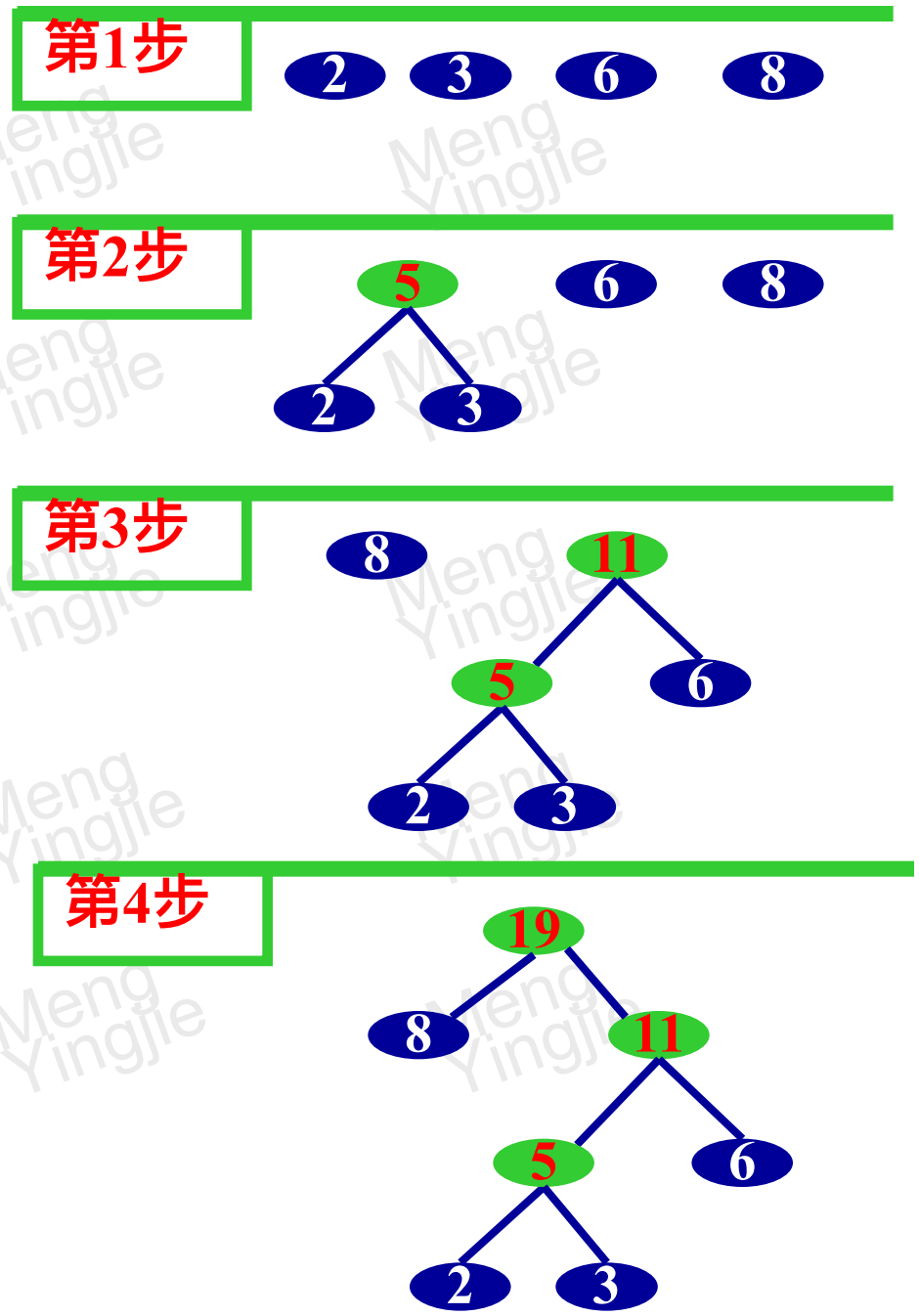
举例： {2，3，6，8}构造Huffman树。

R

	1	2	3	4	5	6	7
data	2	3	6	8	5	11	19
lson	0	0	0	0	1	5	4
rson	0	0	0	0	2	3	6

A

	1	2	3	4
data	8	11		
adr	4	6		



算法过程：

R

data	w_1	w_2	…	w_n			
lson	0	0	…	0			
rson	0	0	…	0			
	1	2	…	n	n+1	…	2n-1

```
PROC  Huffman(R,n);
BEGIN
    FOR i←1 TO n DO 【A[i].data←R[i].data;A[i].adr←i】 ;
   i←0;
  WHILE n-i≥2 DO
     【call insort(A,n-i);  {对数组A的前n-i个元素依data项进行稳定排序}
       i←i+1;
       R[n+i].data←A[1].data+A[2].data;    {生成T的根结点的值}
       R[n+i].lson←A[1].adr;               {生成T的左子树}
       R[n+i].rson←A[2].adr;               {生成T的右子树}
       A[1].data←R[n+i].data;      {将生成的二叉树作为A第1元素}
       A[1].adr←n+i;
       A[2].data←A[n-i+1].data;    {添补A第2元素空位}
       A[2].adr←A[n-i+1].adr 】
END;
```

A

data					
adr					
	1	2	3	…	n

5.霍夫曼树的应用

①最佳组织结构

将一个测试判断过程看成二叉树,依据每个分类的结果预测概率值，通过构造Huffman树,可形成最佳判断过程。

例1-1.第1章讨论的成绩分类。

例1-2.营销机构中,可根据各营销点市场份额构造Huffman树形成最小总营销体系。

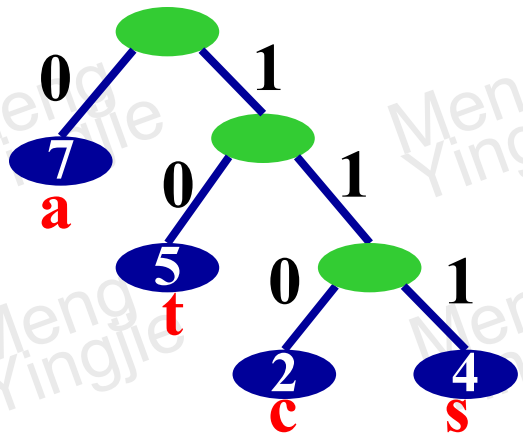
②通讯及数据传输中的二进制编码

例,有下面正文，需对其编码，要求总编码长度最短。

cast cats sat at a tasa

符号集D=(c,a,s,t),对应的每个字符出现次数为WD=(2,7,4,5)

依据WD=(2,7,4,5)构造Huffman树，结果如下：



对该二叉树编码，约定左0，右1，则：

字符编码 a: 0

t: 10

s: 111

c: 110

110011110

出现次数最多的字符编码最短。

特点： a. 给出的文本有最短的编码。

b. 若符号$d_i \neq d_j$，则d_i的编码不可能为d_j的编码的词头(前缀),即使两个字符相等,其编码也不相同，因此两个字符之间就不需要分隔符.(因为根结点到每个外点的二进制位序列不同)

但两个词之间仍需要空白分隔.

缺点：编码不等长，因此译码困难。

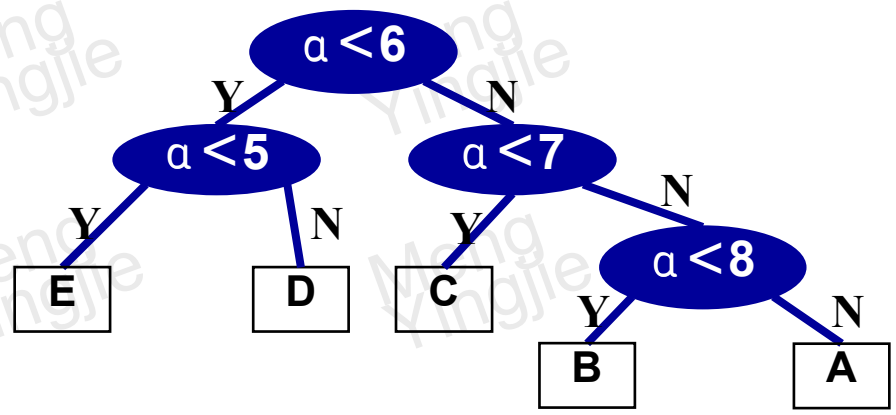
四.分类与判定树

分类是一种常见运算,树形结构一类重要的应用就是描述分类过程,其作用是将输入数据按预定的标准划分成不同的种类。例如产品质量的分类。

等级	E	D	C	B	A
检测值	α＜5	5≤α＜6	6≤α＜7	7≤α＜8	8≤α
百分比	0.2	0.2	0.3	0.2	0.1

用于描述分类过程的树形结构称为判定树。

多为二叉树结构，也有树结构的。对于二叉树型的可看成是扩充二叉树，外结点为分类结果，内结点为分类判断(即都是作一次比较)。



举例：

八枚硬币问题：假定有八枚硬币a,b,c,d,e,f,g,h,其中有一枚是伪造的。真、伪硬币的区别仅是重量不同，可能重，也可能轻。要求以最小的比较次数通过天平找出伪币。

以H表示重,L表示轻,经过三次乘量就可找出。现分析一种情况。

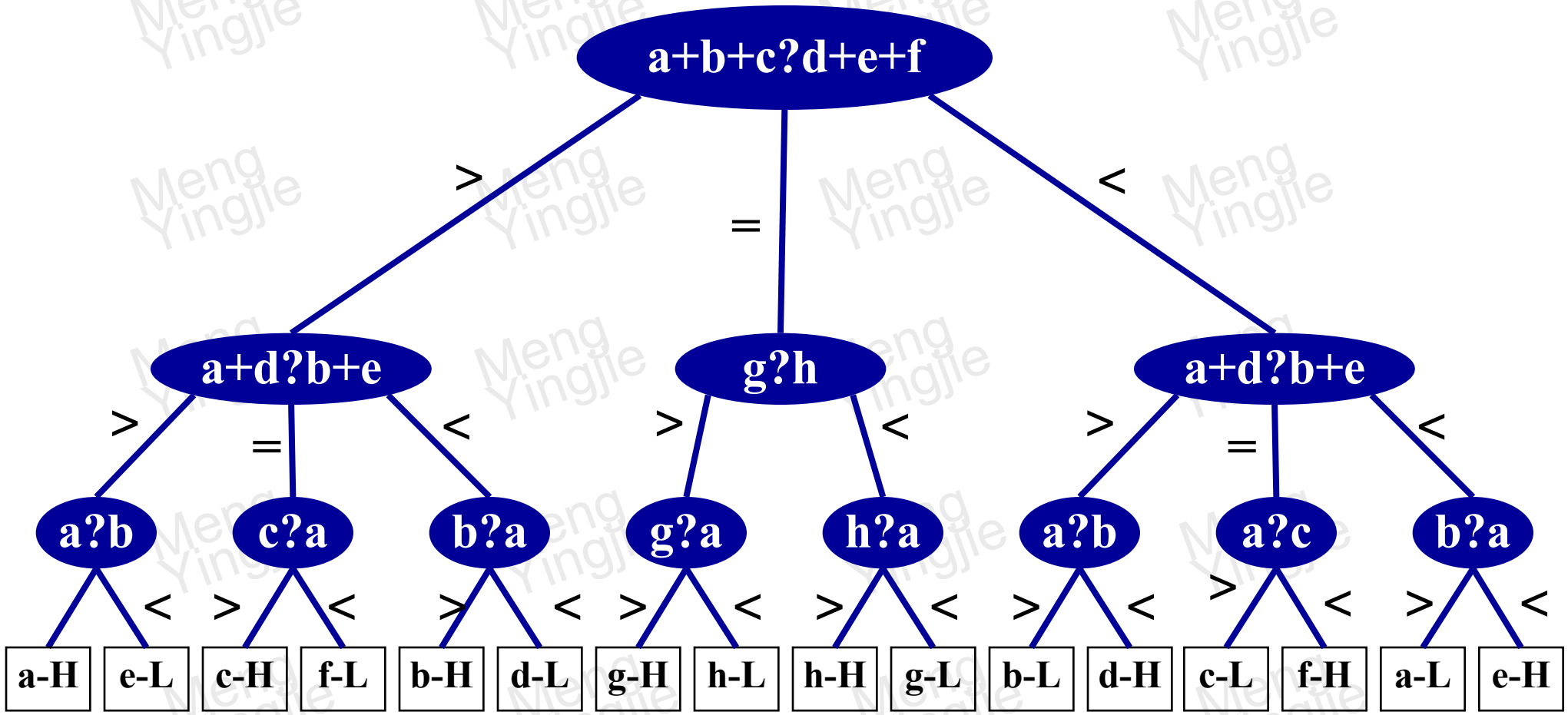
(1).各取三枚硬币进行称量，如果(a+b+c)<(d+e+f)，则伪币必在这六枚中，而g,h是真币。

(2).再在上述六枚硬币的不等式两边各取一枚作为一方，如一方取a,d，另一方取b,e，再进行称量。如果(a+d)<(b+e)，说明c,f是真币；同时由于d和b交换位置后，不等式依然成立，所以d,b也是真币。

(3).取一枚真币(如b)与a比较,若a轻则a是伪币;若相等则e重,e是伪币。

此问题共有16种可能性，但乘量只需三次。判定树见下图。

八枚硬币问题的判定树：


a+b+c?d+e+f
>
=
<
a+d?b+e
g?h
a+d?b+e
a?b
c?a
b?a
g?a
h?a
a?b
a?c
b?a
a-H
e-L
c-H
f-L
b-H
d-L
g-H
h-L
h-H
g-L
b-L
d-H
c-L
f-H
a-L
e-H

本节结束

1、若一棵二叉树先序遍历和中序遍历序列分别为ABDEHCFGI, DBEHAFCIG,试画出该二叉树。

2、有一份报文，使用5个字符:a,b,c,d,e,它们出现的频率依次是4，7，5，2，9，给出每个字符的huffman编码。

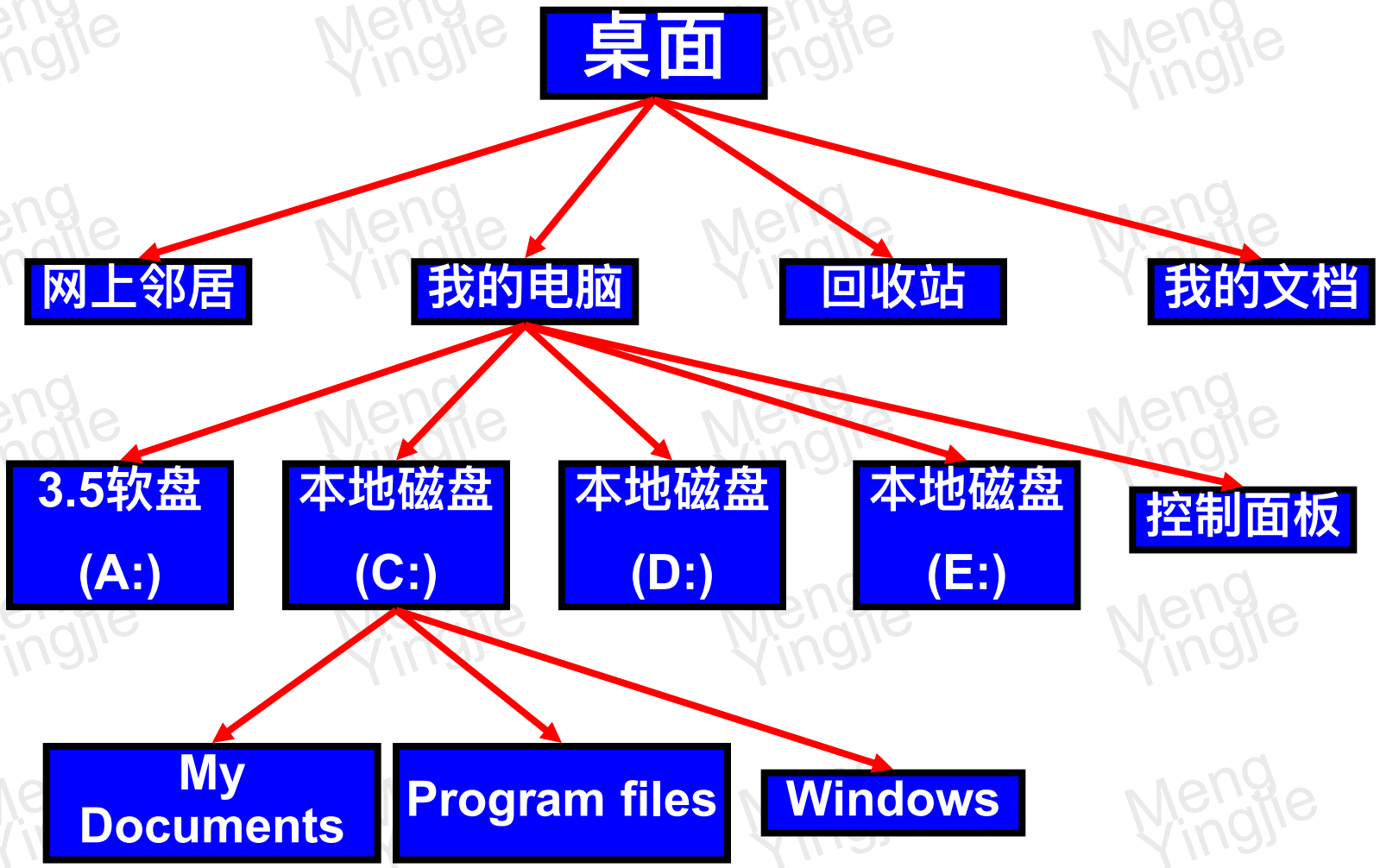
3、试编写一个统计二叉树内结点的算法。

4、设计一个先序遍历的非递归算法。

2008.5.15(周四)数据结构课堂交作业

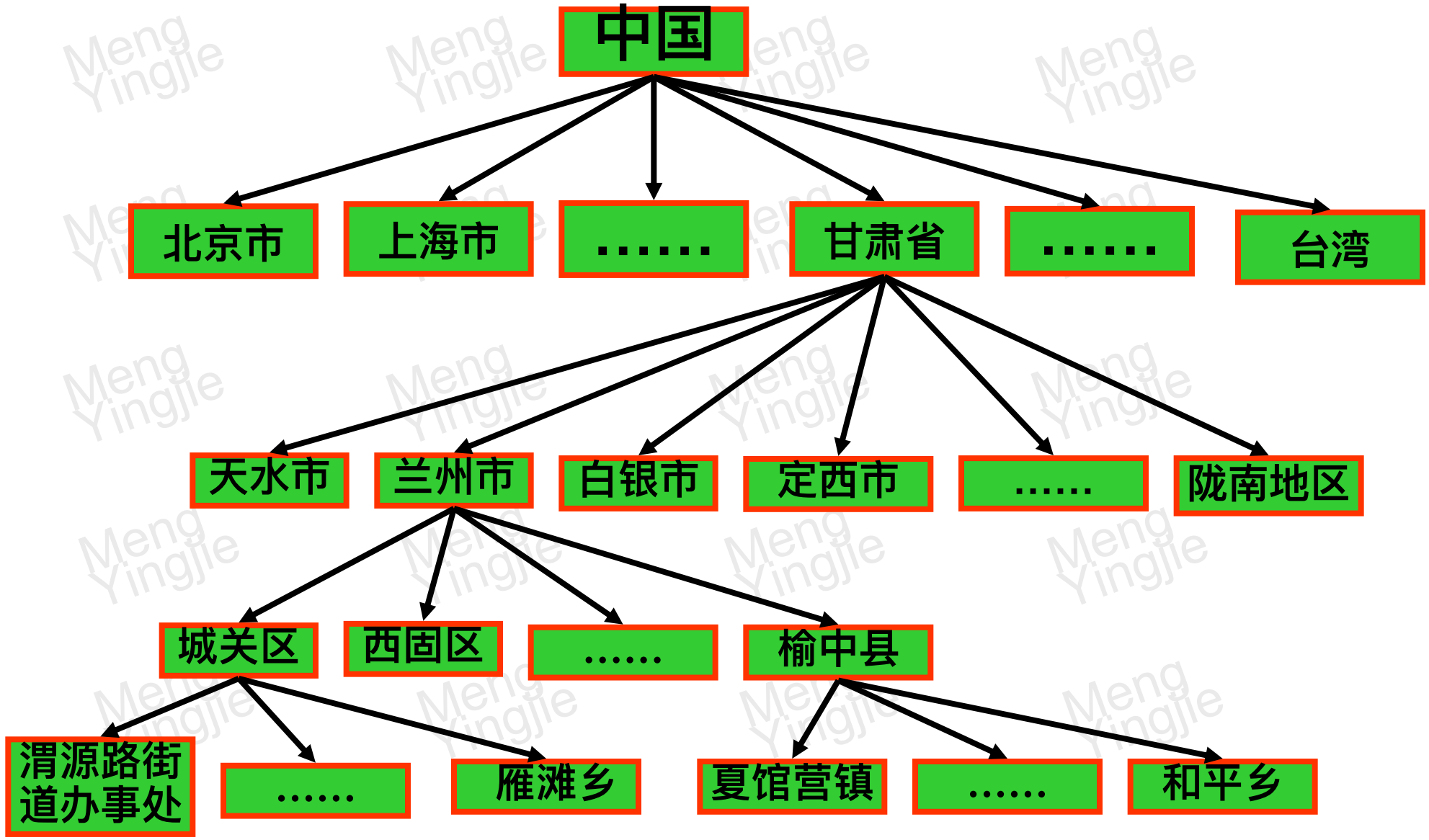
本章结束

一个Windows系统菜单调用关系图

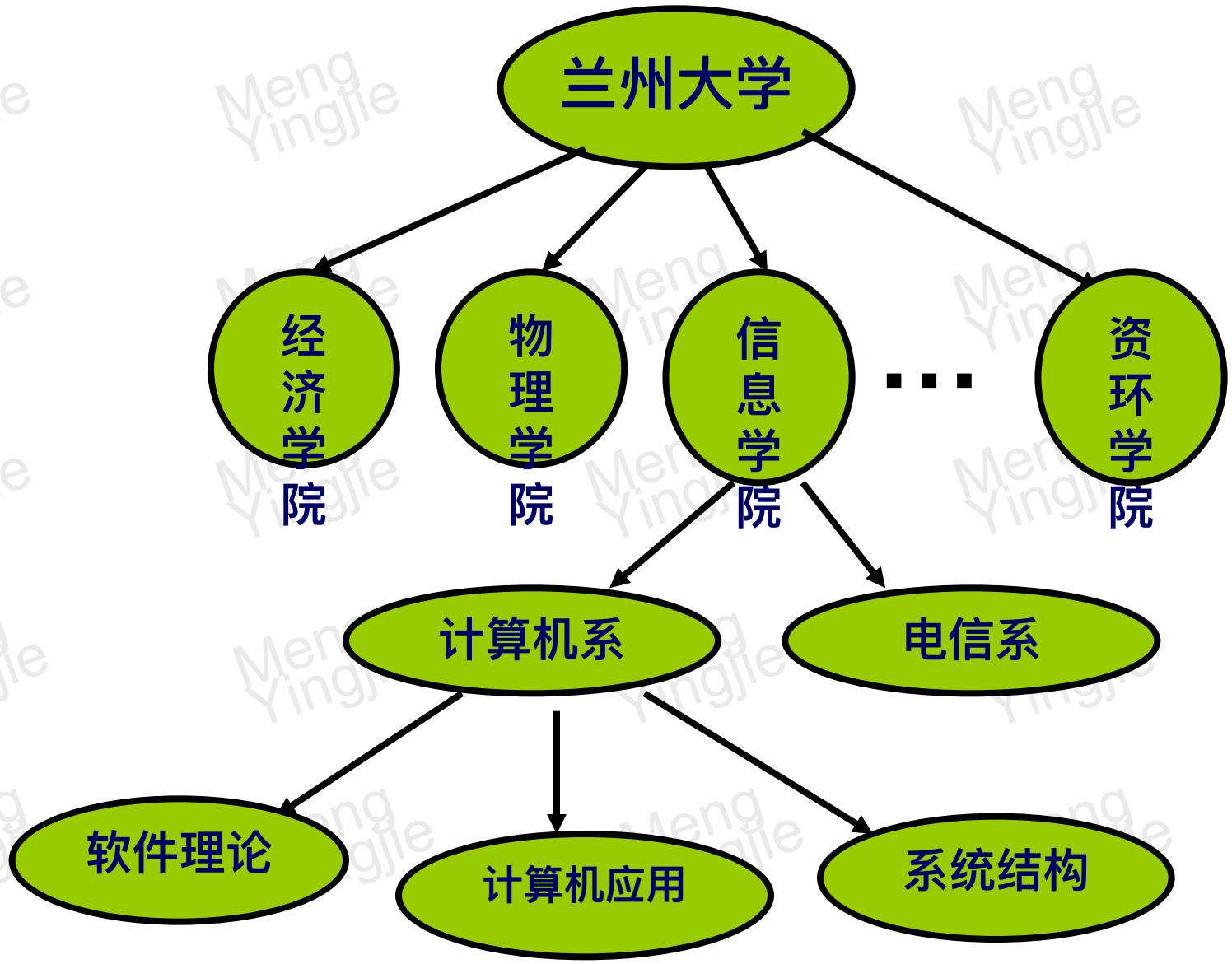


程序调用，文件数据的组织，互连网的地址组织等。

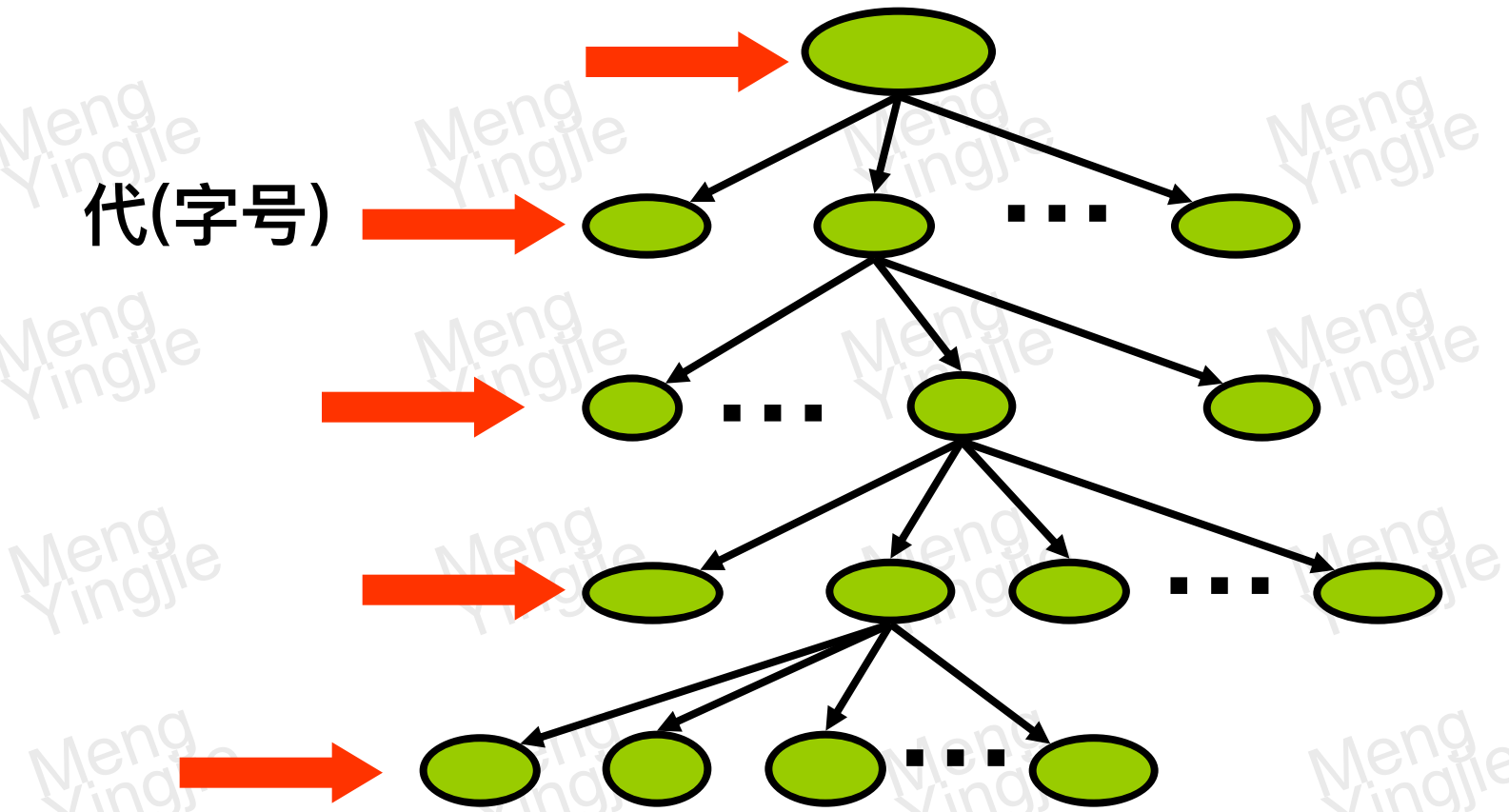
一个国家的行政机关组织简图



一个组织或企事业的组织机构简图



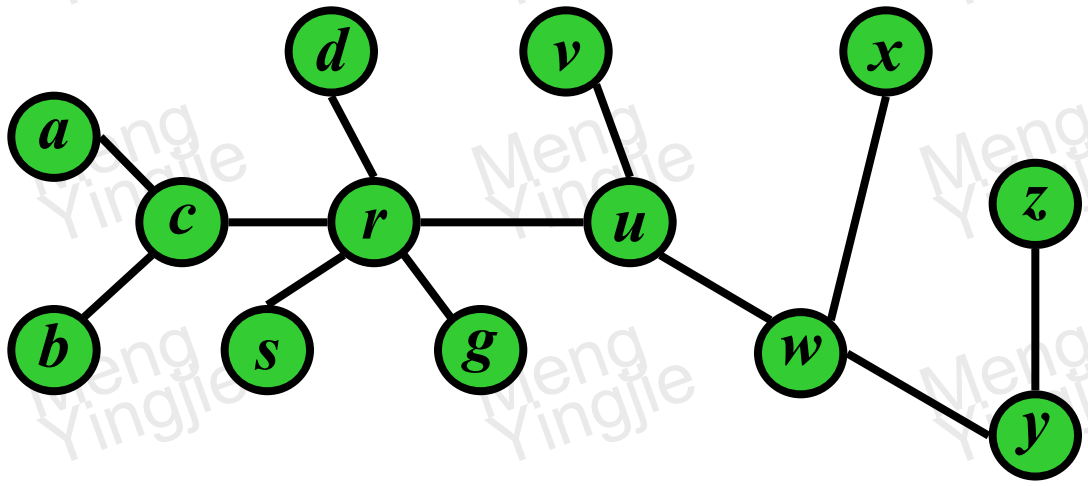
家谱图，门派演变图

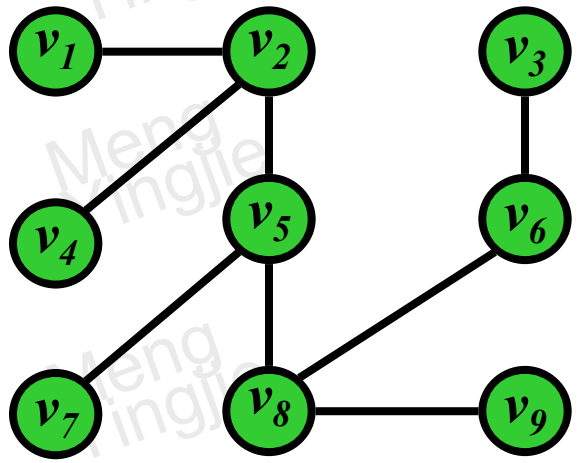


再例如，文章的组织、语法的句子(词)等。

图论中有关树的概念：

树(或树图)：连通且没有回路的无向图。



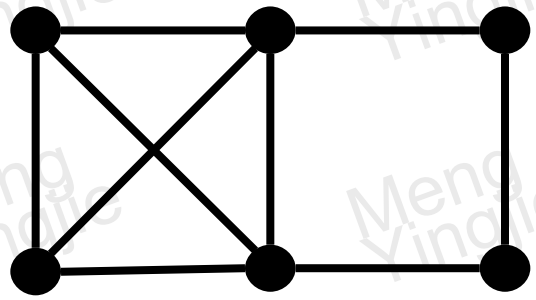


给定图，以下关于树的定义是等价的：

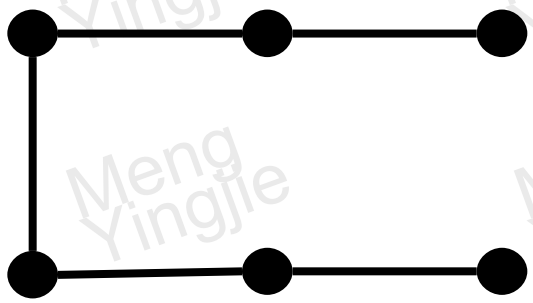
- 无回路的连通图。
- 无回路且e=v-1,其中e为边数,v为结点数。
- 无回路，但增加一条新边，得到一个且仅一个回路。
- 连通但删除任一边后便不连通。
- 每对结点之间有一条且仅有一条路径。

生成树(或支撑树,spanning tree):

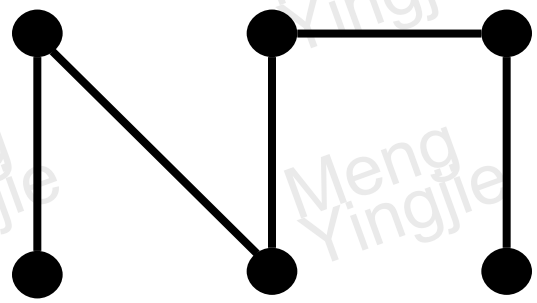
连通图G的子图T称G的生成树,如果T是树,且包含了G的所有顶点。下图给出了连通图G及G的生成树T_1 ,T_2 ,T_3



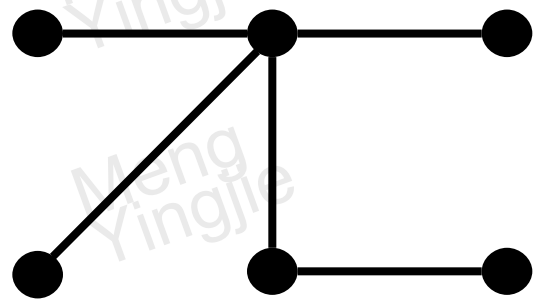
图G



生成树T_1



生成树T_2

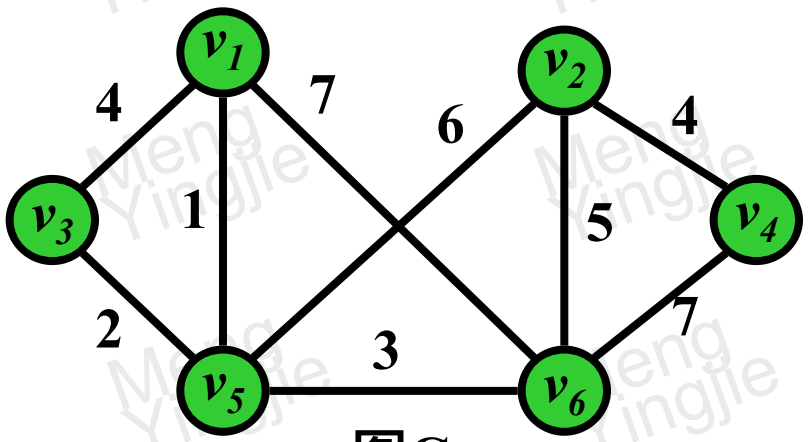


生成树T_3

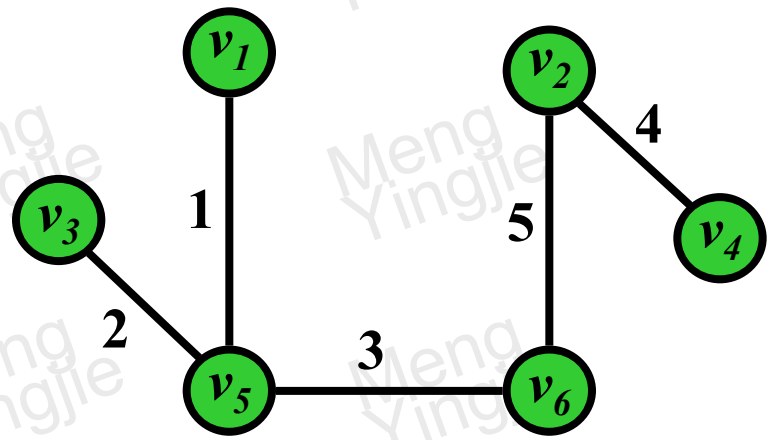
最小生成树(或最小支撑树,minimum spanning tree):

设G为连通的赋权图，即G的每一条边被指定了一个称为边的权的非负数，则G的每个生成树被指定了一个T中每条边的权之和的总权。

G的最小生成树就是总权最小的生成树。



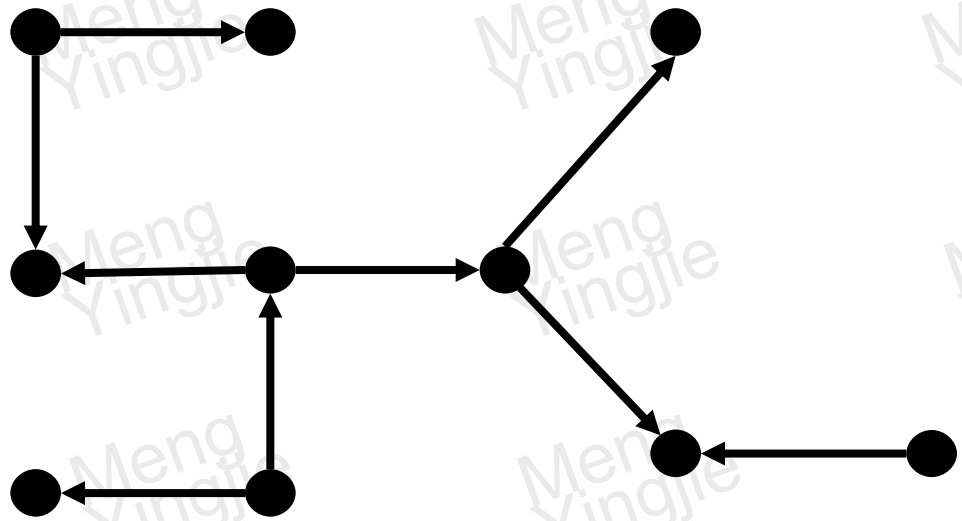
图G



最小生成树

有向树(directed tree)：

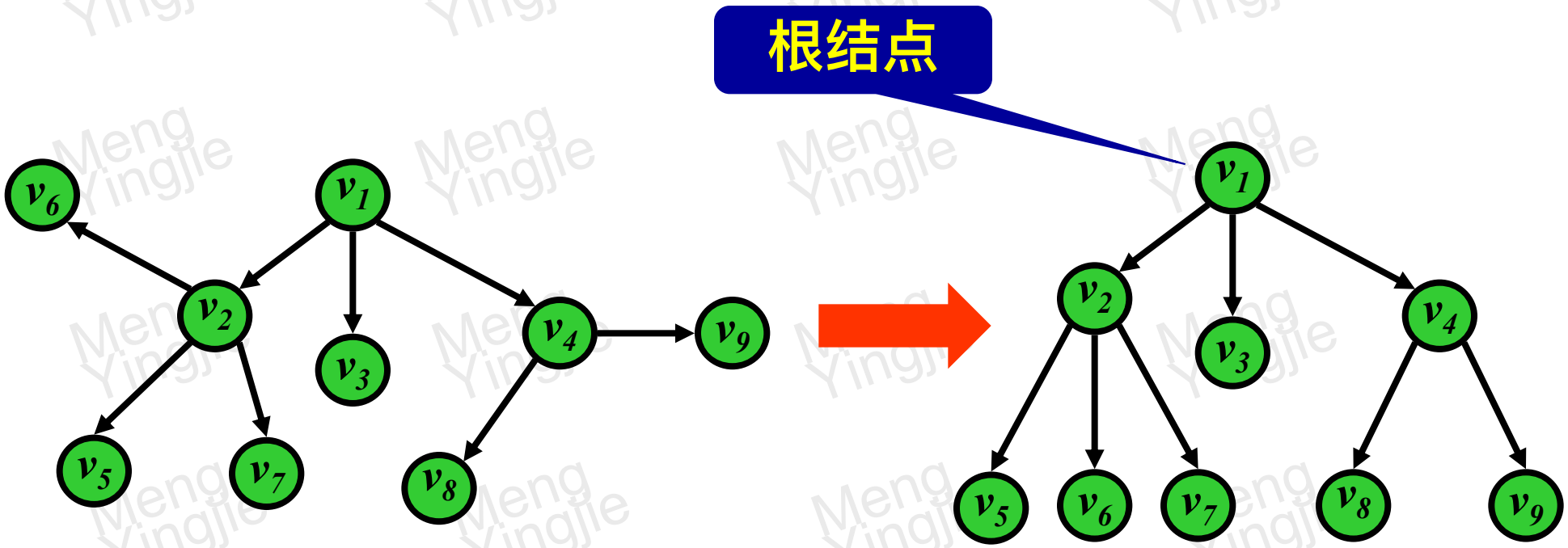
如果一个有向图在不考虑边的方向时是一棵树，那么这个有向图称为有向树。如下图。



根树(或有根树,root tree)：

数据结构的树

一个有向树，如果恰有一个结点的入度为0,其余所有结点的入度都为1,则称该有向树为根树。入度为0的结点称为根。



树与二叉树的异同 ——概念：

树T是满足如下性质的有限个结点组成的非空集合：T中有且仅有一个称为根的结点；除根结点之外，其余结点分成m(m>0)个不相交的的集合T_1, T_2, … ,T_m,其中每个T_i都是树，而且都称为T的子树。

二叉树T是满足如下性质的结点的有限集合：T是空集;或者T包含一个根结点且其余结点分成两个不相交的集合，并分别称为根结点的左子树和右子树。

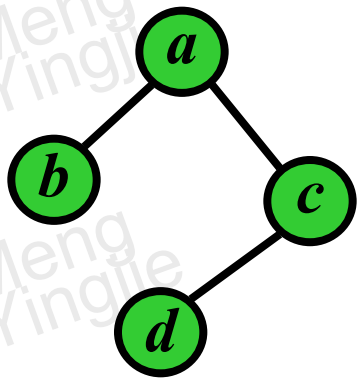
树与二叉树的异同 ——本质：

尽管概念上有许多联系，但二叉树不是树的特例。它们本质上是两个不同的研究对象。

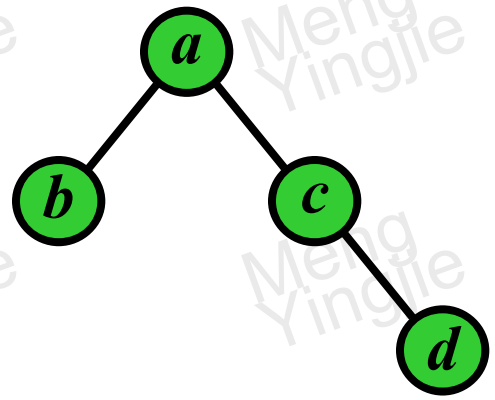
最主要差别：二叉树子树的个数的确定性以及次序的有序性(要区分左右),即便只有一棵子树也要区分左右，特别是这点对于有序树来说也不成立。下面举例说明：

附录7.2.1__树与二叉树的差异

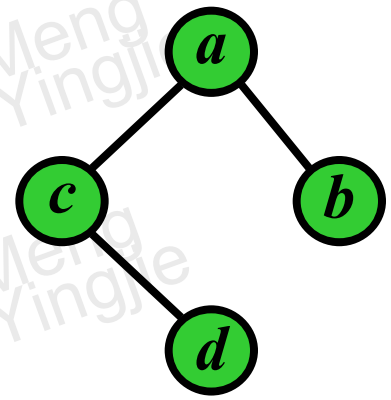
举例：



T_1



T_2



T_3

- 树：T_1 ,T_2 ,T_3为三个相同的树。
- 有序树：T_1 ,T_2 为相同的有序树，但它们与T_3是不同的有序树。
- 二叉树：T_1 ,T_2 ,T_3它们是三个不同的二叉树。

在二叉树中，第i (i≥1)层的结点数最多为2^{i-1}.

证明：用数学归纳法证明。

当i=1时，只有根结点，$2^{i-1}=2^{1-1}=2^0=1$。成立。

设对于所有的j(1≤j<i)成立,即第j层上至多含有2^{j-1}个结点。

根据假设，第i-1层上至多含有2^{i-2}个结点。

由于二叉树中每个结点至多有2个孩子，故第i层上的结点数应当为第i-1层上的最大结点数的2倍，即

$2\times 2^{i-2}=2^{i-1}$。

故命题成立。

深度为k (k≥1)的二叉数结点总数最多为2^k-1.

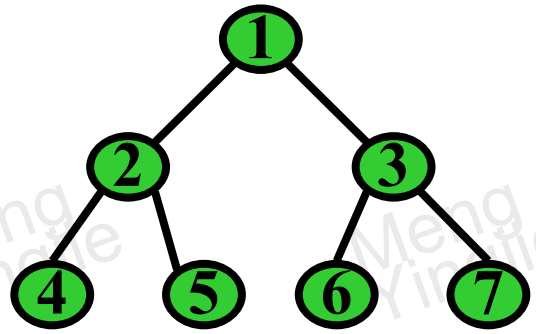
证明： 深度为k的二叉树最多含有的结点数应是二叉树中每层上最多含有的结点数之和。

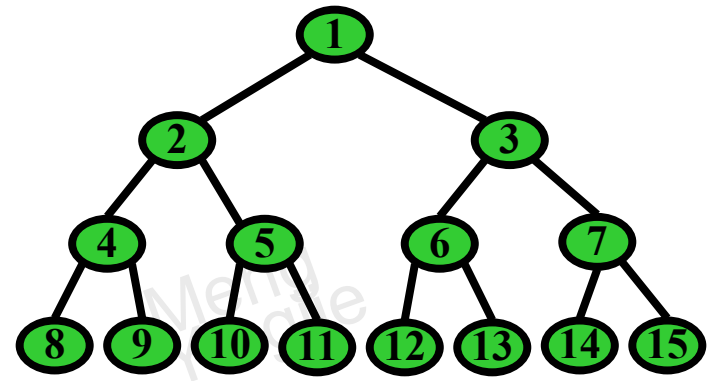
由性质1可得：

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

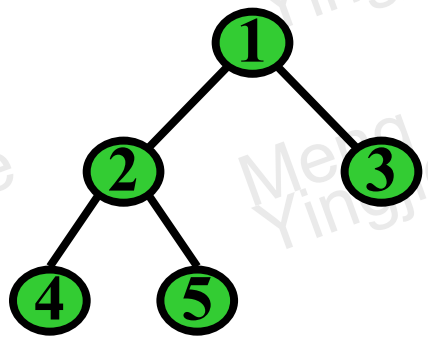
故命题成立。

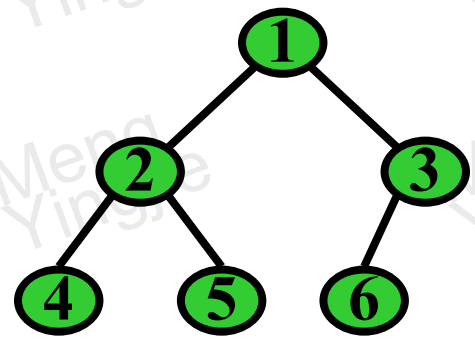
满二叉树



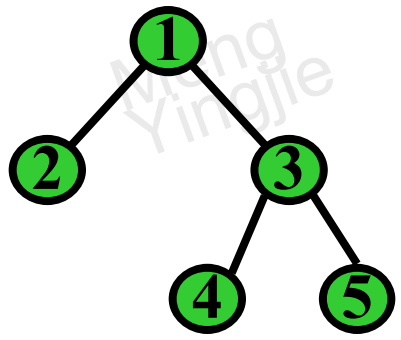


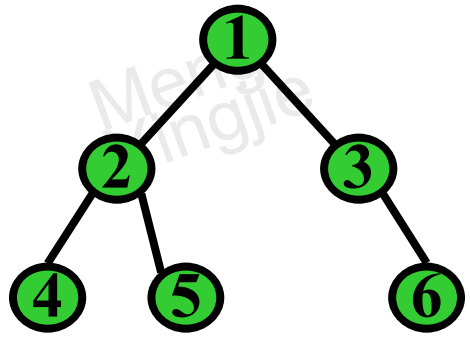
完全二叉树(或顺序二叉树)

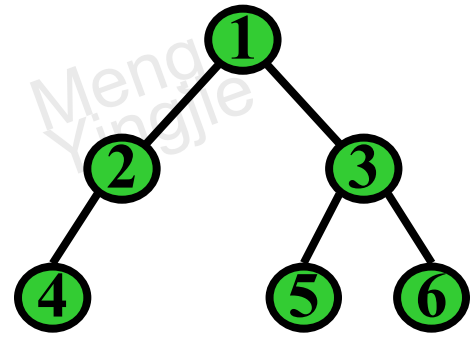


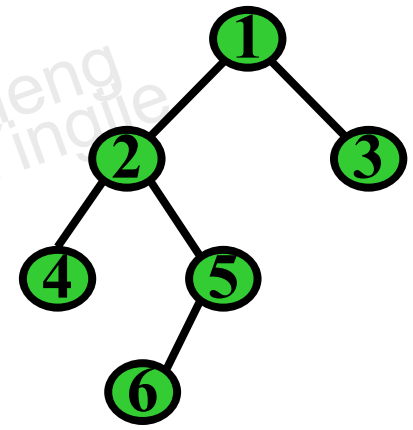


非完全二叉树









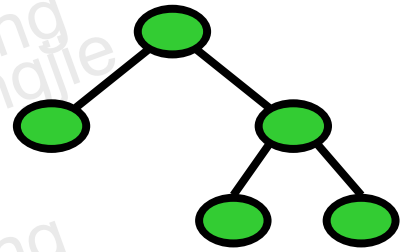
满二叉树与完全二叉树的其它定义：

(以下定义并非原文，仅从含义上说明)

1. 许卓群等.《数据结构》.高等教育出版社.

满二叉树:二叉树中若所有结点的度或者为0,或者为2,则此二叉树称~.

完全二叉树:概念相同。



2.张铭等.《数据结构与算法》.高等教育出版社.(北京大学，许卓群作者群)

满二叉树：二叉树中若所有结点的度或者为0,或者为2,则此二叉树称~.

完全二叉树:概念相同。

3.国防科大.王广芳等.电子计算机软件《数据结构》.湖南科学出版社.

满二叉树：概念相同。

完全二叉树:二叉树中若所有结点的度或者为0,或者为2,则此二叉树称~.

任意一棵二叉树中,若终端结点数为n_0,度为2的结点数为n_2，则有：$n_0=n_2+1$.

证明: 设二叉树中结点总数为n, 度为1的结点数为n_1.

(1).因二叉树中所有结点的度数都≤2，因此：

$$n=n_0+n_1+n_2 \quad \text{.....(a)}$$

(2).设B为二叉树中的分支(即边)数。在二叉树中除根结点外,每个结点都有一个分支进入。

$$B = n-1 \quad ===> \quad n = B+1 \quad \text{.....(b)}$$

(3).二叉树中的分支都是由度为1和度为2的结点发出，故

$$B=1\times n_1+2\times n_2 \quad \text{.....(c)}$$

(4).由(b)、(c)可得出：

$$n=1+n_1+2n_2 \quad \text{.....(d)}$$

(5).由(a)、(d)可得出：

$$n_0= n_2 + 1$$

证毕。

一个有n个结点的完全二叉树其深度为$\lfloor \log_2 n \rfloor+1$.

证明： 设k为该二叉数的深度。

根据(性质2:)深度为k (k≥1)的二叉数结点总数最多为2^k-1以及完全二叉树的定义可知：

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

因此，对不等式取对数运算可得：

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

因k为整数，所以：$k-1=\lfloor \log_2 n \rfloor+1$ ，即：

$$k=\lfloor \log_2 n \rfloor+1.$$

证毕。

符号：$\lfloor x \rfloor$、$\lceil x \rceil$、$[x]$

$\lfloor 3.14 \rfloor=3$, $\lfloor 3.64 \rfloor=3$

$\lceil 3.14 \rceil=4$, $\lceil 3.64 \rceil=4$

$[3.14]=3$, $[3.64]=4$

先序序列和中序序列可以唯一确定一棵二叉树.

证明：采用归纳法证明。设二叉树有n个结点。

(1).当n=1时，前序序列和中序序列均只有1个元素，且相同，即为树根，由此唯一确定了一棵只有一个根结点的二叉树。

(2).假设m≤n-1时都成立。

以下只需证明m=n时也成立即可。

(3)设二叉树的先序序列为：$(a_1, a_2, \ldots, a_n)$

中序序列为：$(b_1, b_2, \ldots, b_n)$

因前序序列由前序遍历二叉树所得，因此a_1即为根结点；
又中序序列由中序遍历二叉树所得，则在中序序列中必定能找到和a_1值相同得结点，设该结点为b_j（$1\le j\le n$），由此可得两个集合：

$(b_1, b_2, \ldots, b_{j-1})$ 为二叉树的左子树中序序列；

$(b_{j+1}, b_{j+2}, \ldots, b_n)$为二叉树的右子树中序序列；

①若$j=1$，即b_1为根结点,此时二叉树左子树为空,$(a_2,\ldots,a_n)$为右子树的前序序列,$(b_2,\ldots,b_n)$为右子树的中序序列。右子树中结点数目为n-1,由(2)可知,这两个序列可以唯一确定右子树,因此也就唯一确定了二叉树。

②若$j=n$，即b_n为根结点，此时二叉树右子树为空，同① 理由,子序列$(a_2,\ldots,a_n)$和$(b_1,\ldots,b_{n-1})$ 唯一确定左子树。

③若$2\le j\le n-1$，则子序列$(a_2,\ldots,a_j)$和$(b_1,\ldots,b_{j-1})$ 唯一确定左子树;
子序列$(a_{j+1},\ldots,a_n)$和$(b_{j+1},\ldots,b_n)$ 唯一确定右子树;

由①、②、③可知唯一根和唯一的左、右子树只能构成唯一的二叉树。

证毕

设pre[1..n]存放前序序列,ind[1..n]存放中序序列.试设计通过pre[i..j]和ind[u..v]构造二叉树的过程，根结点指针为S。(假设i,j,u,v均为正整数，且i≤j,u≤v).

```
PROC bintree(i,j,u,v:integer; VAR S: BinaryTree);
BEGIN   s←nil;
   IF j-i≠v-u THEN 【write(‘ 两序中元素个数不等，出错‘ ); EXIT】;
   IF j-i≥0
   THEN 【new(s);    s↑.data=pre[i];
          k←u;      {k为中序中根结点的位置号}
          while (ind[k]≠pre[i]) AND (k≤v) DO  k←k+1;
          IF k＞v THEN 【write(‘ 前序中结点在中序中找不到,出错‘ ); EXIT】;
          m←i+(k-u);
          {m为左子树在前序序列下遍历的最后一个结点在pre中的位置号}
          IF k=u THEN s↑.lson←nil
                 Else bintree(i+1,m,u,k-1,s↑.lson) ; {构造左子树}
          IF  k=v THEN s↑.rson←nil
                 Else bintree(m+1,j,k+1,v,s↑.rson)  {构造右子树} 】
END;
```

后序序列和中序序列可以唯一确定一棵二叉树.

证明：采用归纳法证明。设二叉树有n个结点。

(1).当n=1时，后序序列和中序序列均只有1个元素，且相同，即为树根，由此唯一确定了一棵只有一个根结点的二叉树。

(2).假设m≤n-1时都成立。

以下只需证明m=n时也成立即可。

(3)设二叉树的后序序列为:(a_1 , a_2 , … , a_n)

中序序列为:(b_1 , b_2 , … , b_n)

因后序序列由后序遍历二叉树所得，因此a_n即为根结点；

又中序序列由中序遍历二叉树所得，则在中序序列中必定能找到和a_n值相同得结点，设该结点为b_j（$1 \le j \le n$），由此可得两个集合：

$(b_1, b_2, \ldots, b_{j-1})$ 为二叉树的左子树中序序列；

$(b_{j+1}, b_{j+2}, \ldots, b_n)$为二叉树的右子树中序序列；

①若j=1，即b_1为根结点,此时二叉树左子树为空,$(a_1,\ldots,a_{n-1})$为右子树的后序序列,$(b_2,\ldots,b_n)$为右子树的中序序列。右子树中结点数目为n-1,由(2)可知,这两个序列可以唯一确定右子树,因此也就唯一确定了二叉树。

②若j=n，即b_n为根结点，此时二叉树右子树为空，同① 理由,子序列$(a_1,\ldots,a_{n-1})$和$(b_1,\ldots,b_{n-1})$ 唯一确定左子树。

③若$2 \le j \le n-1$，则子序列$(a_1,\ldots,a_{j-1})$和$(b_1,\ldots,b_{j-1})$ 唯一确定左子树；

子序列$(a_j,\ldots,a_{n-1})$和$(b_{j+1},\ldots,b_n)$ 唯一确定右子树；

由①、②、③可知唯一根和唯一的左、右子树只能构成唯一的二叉树。

证毕

设pot[1..n]存放后序序列,ind[1..n]存放中序序列.试设计通过pot[i..j]和ind[u..v]构造二叉树的过程，根结点指针为S。(假设i,j,u,v均为正整数，且i≤j,u≤v).

```
PROC bintree(i,j,u,v:integer; VAR S: BinaryTree);
BEGIN   s←nil;
   IF j-i≠v-u THEN 【write(‘ 两序中元素个数不等，出错‘ ); EXIT】;
   IF j-i≥0
   THEN 【new(s);    s↑.data=pot[j];
          k←u;      {k为中序中根结点的位置号}
          while (ind[k]≠pot[j]) AND (k≤v) DO  k←k+1;
          IF k＞v THEN 【write(‘ 前序中结点在中序中找不到,出错‘ ); EXIT】;
          m←i+(k-u);
          {m为左子树在后序序列下遍历的最后一个结点在pre中的位置号}
          IF k=u THEN s↑.lson←nil
                 Else bintree(i,m-1,u,k-1,s↑.lson) ; {构造左子树}
          IF  k=v THEN s↑.rson←nil
                 Else bintree(m,j-1,k+1,v,s↑.rson)  {构造右子树} 】
END;
```

设E(expansion)为2-树中的外路径长度，I(inside)为内路径长度，n为内结点个数。则有：E=I+2n

证明：采用归纳法证明:

(1).当n=1时，I=0，E=2,等式成立。

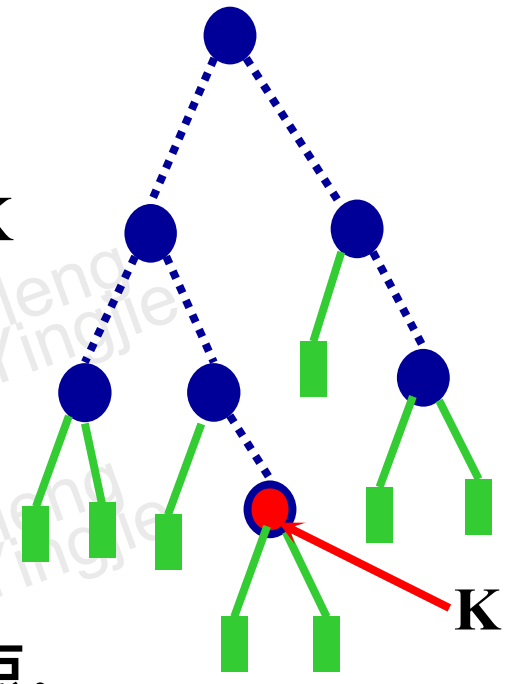
(2).设对于n时,等式也成立即:

$E_n=I_n+2n$ ……(a)

(3).现考虑对于n+1个内结点时的情况:

(3).现考虑对于n+1个内结点时的情况：

我们删去它的一个作为原二叉树树叶的路径长度为K的内部结点，使其成为具有n个内部结点的2-树。



因删去了一个路径长度为K的内部结点，内路径长度变为 $I_n=I_{n+1}-K$ ……(b)

由于删除导致减少了2个路径长度为K+1的外部结点，增加了1个路径长度为K+1的外部结点，外路径长度变为

$E_n=I_{n+1}-2(K+1)+K=E_{n+1}-K-2$ ……(c)

将(a)、(b)带入(c)，可得：

$I_n+2n=E_{n+1}-K-2$

$I_{n+1}-K+2n=E_{n+1}-K-2$

$E_{n+1}=I_{n+1}+2(n+1)$

证毕

附录结束