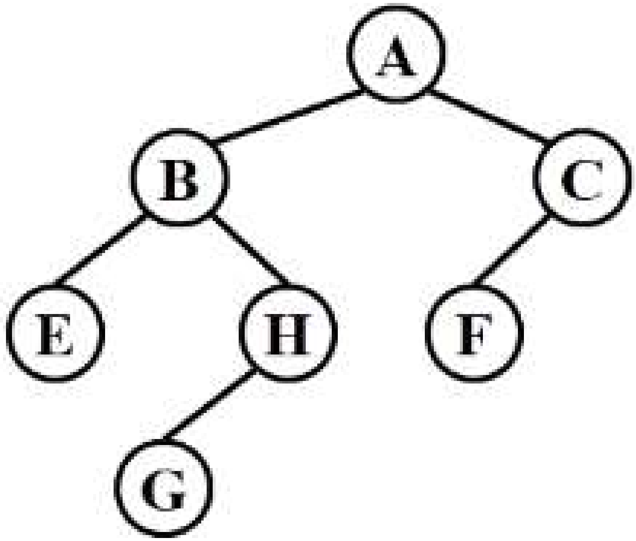
二叉树的遍历



算法思想6.1
前序遍历：
若BT非空，则：
1.访问根结点；
2.前序遍历左子树；
3.前序遍历右子树；

前序遍历(NLR)序列： A B E H G C F

中序遍历(LNR)序列： E B G H A F C

后序遍历(LRN)序列： E G H B F C A

二叉树的遍历

前序遍历二叉树的递归算法

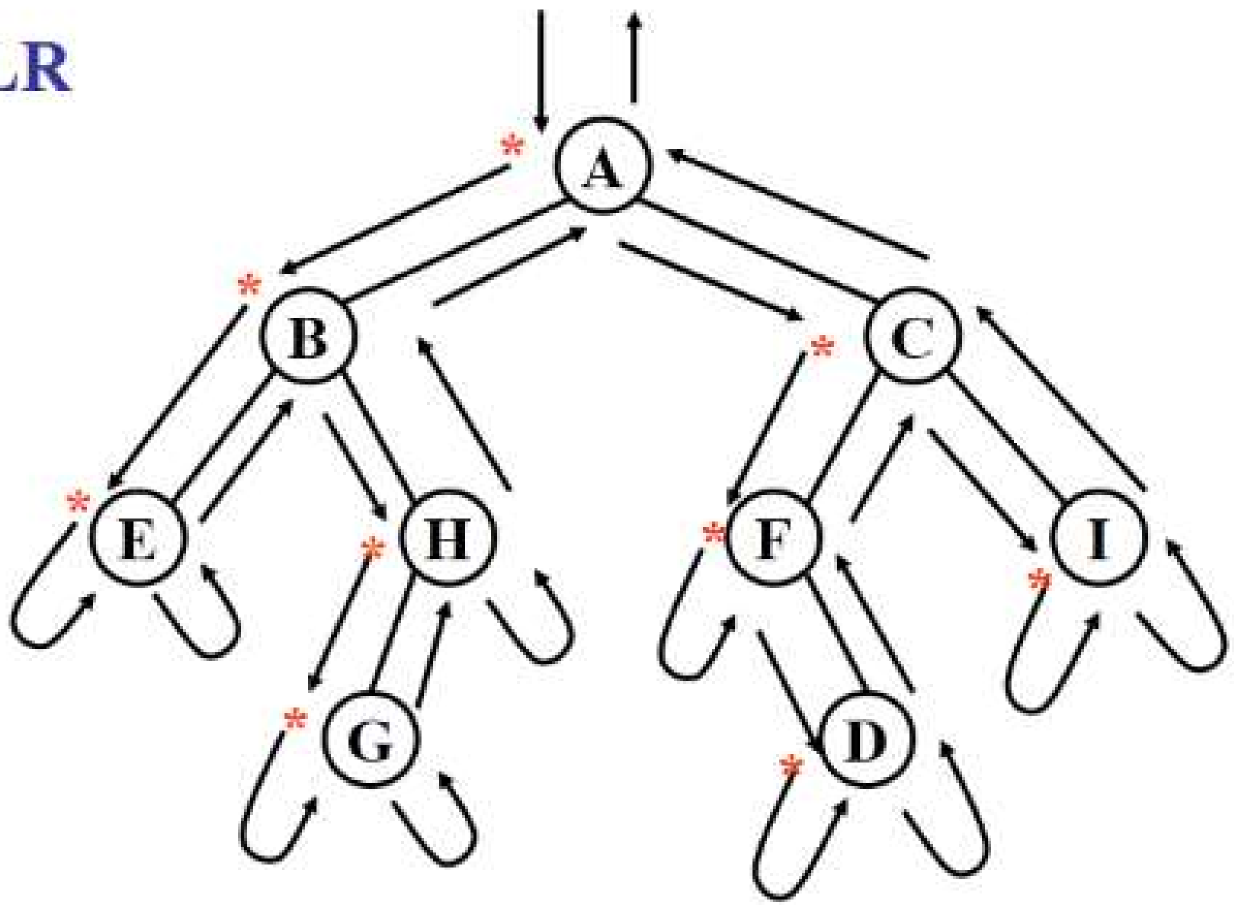
算法 6.1

```
Void PreOrderTraverse(BiTree BT){
   if (BT){
      visit(BT);
      PreOrderTraverse(BT->lchild);
      PreOrderTraverse(BT->rchild);
   }//
} // end of PreOrderTraverse
```

遍历二叉树的非递归算法：

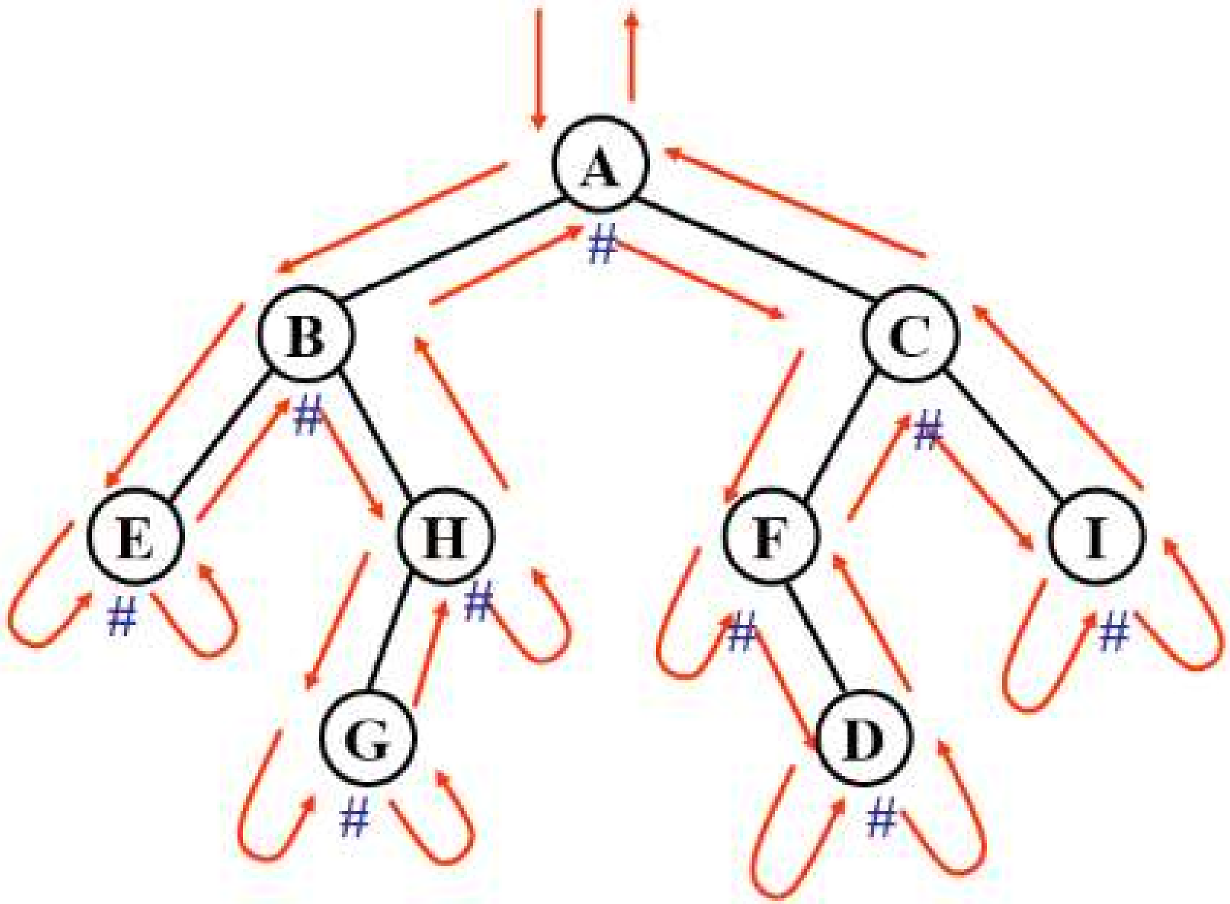
我们先观察一下三种遍历行走的路线

前序遍历NLR



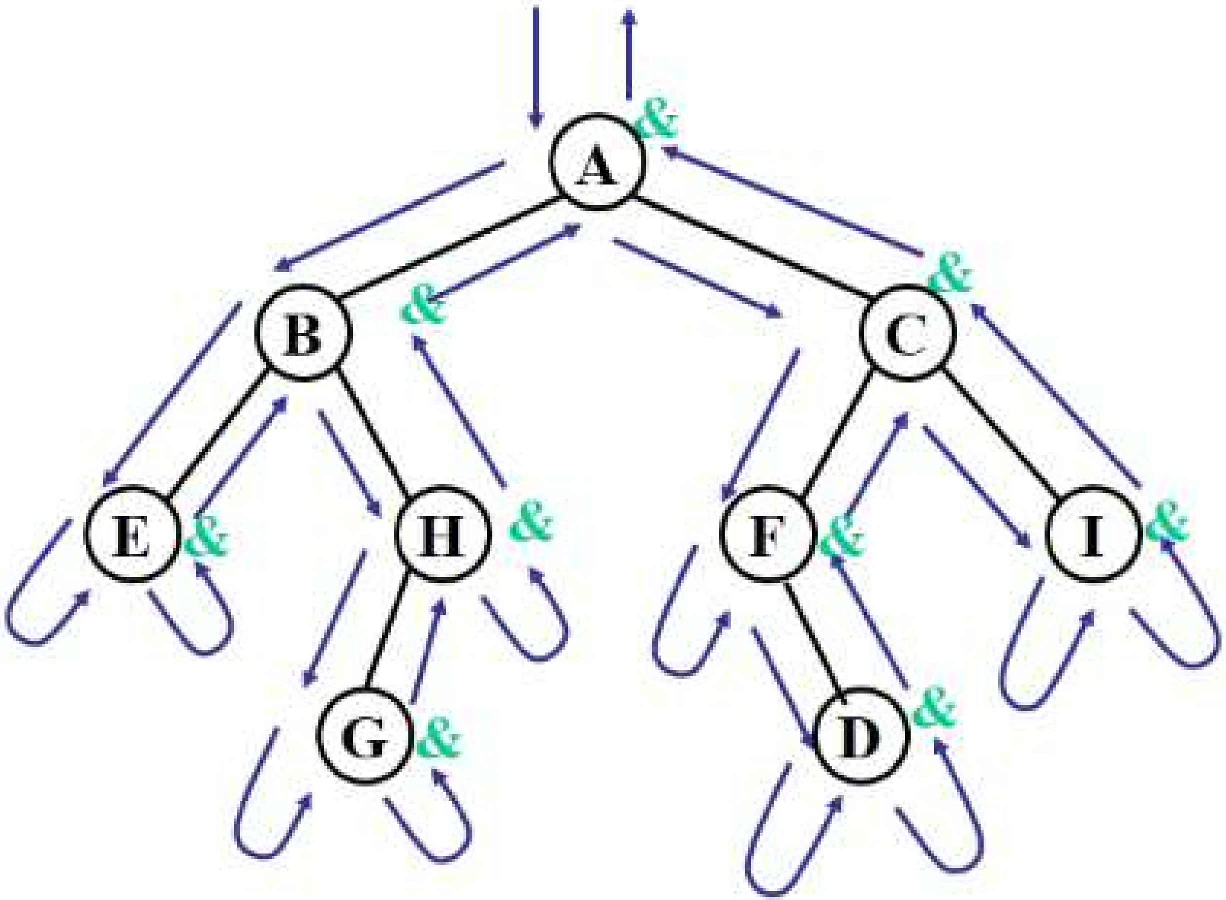
二叉树的遍历

中序遍历LNR



二叉树的遍历

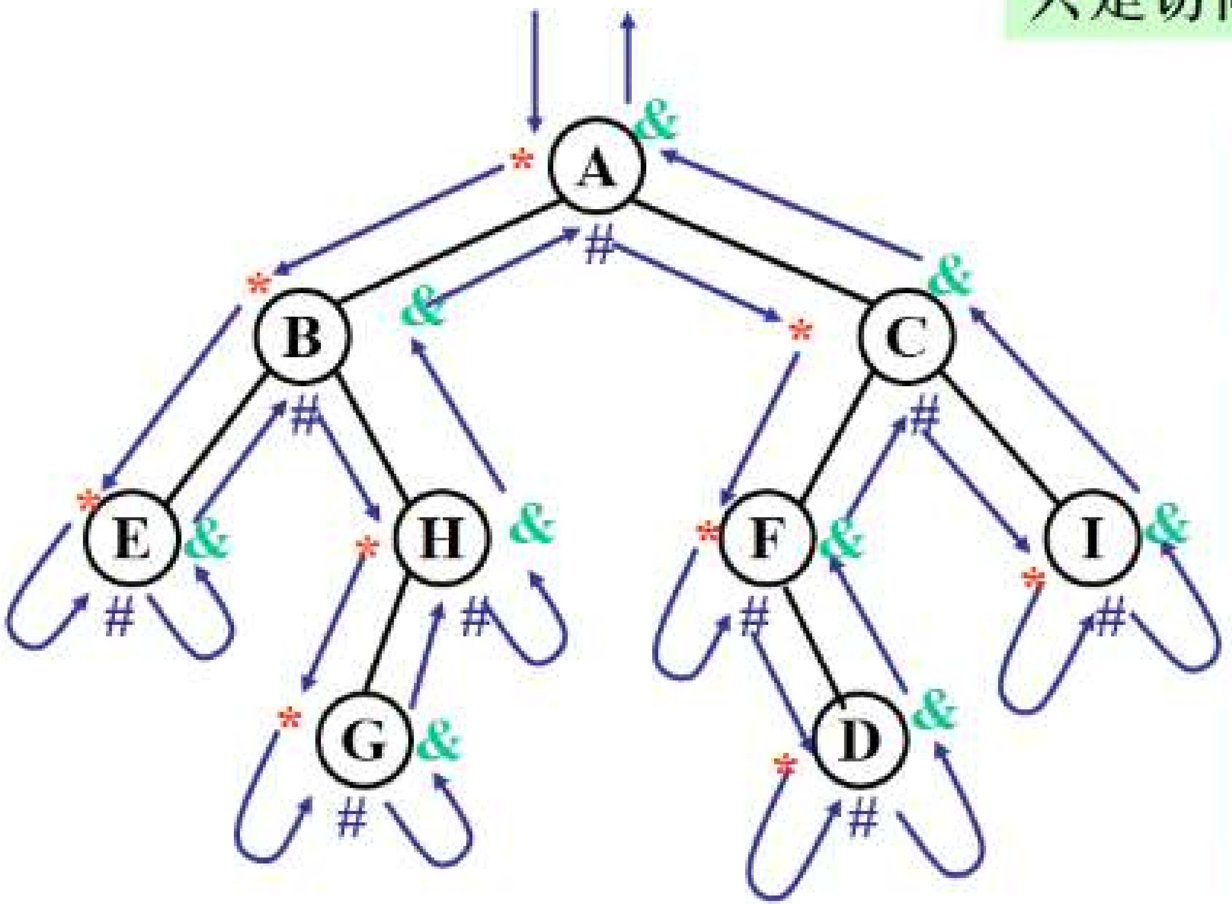
后序遍历LRN



二叉树的遍历

三种遍历的访问位置对比：

三种遍历的路线完全一样，只是访问时间不同；



前序：第一次经过*时访问

中序：第二次经过#时访问

后序：第三次经过&时访问

二叉树的遍历

遍历线路的核心规则是：先左后右；

我们用一个栈stack记录走过的位置，以便返回；

stack 中数据元素的类型： *BiTNode(或BiTree)

函数：BiTree P;

push(S,p);

pop(S,p);

Boolean StackEmpty(S);

下面给出基于逻辑

结构的算法描述

如何进行前序遍历？

非递归中序遍历二叉树的算法思想

建立栈 stack;

1. P指向根；
2. 当p不空 且 stack 不空时反复做：

   若 p不空 ,p 入 栈; p指向左子女；

   否则：

   - 出栈顶元素到p中；
   - 访问p；
   - p指向右子女；

4. 结束

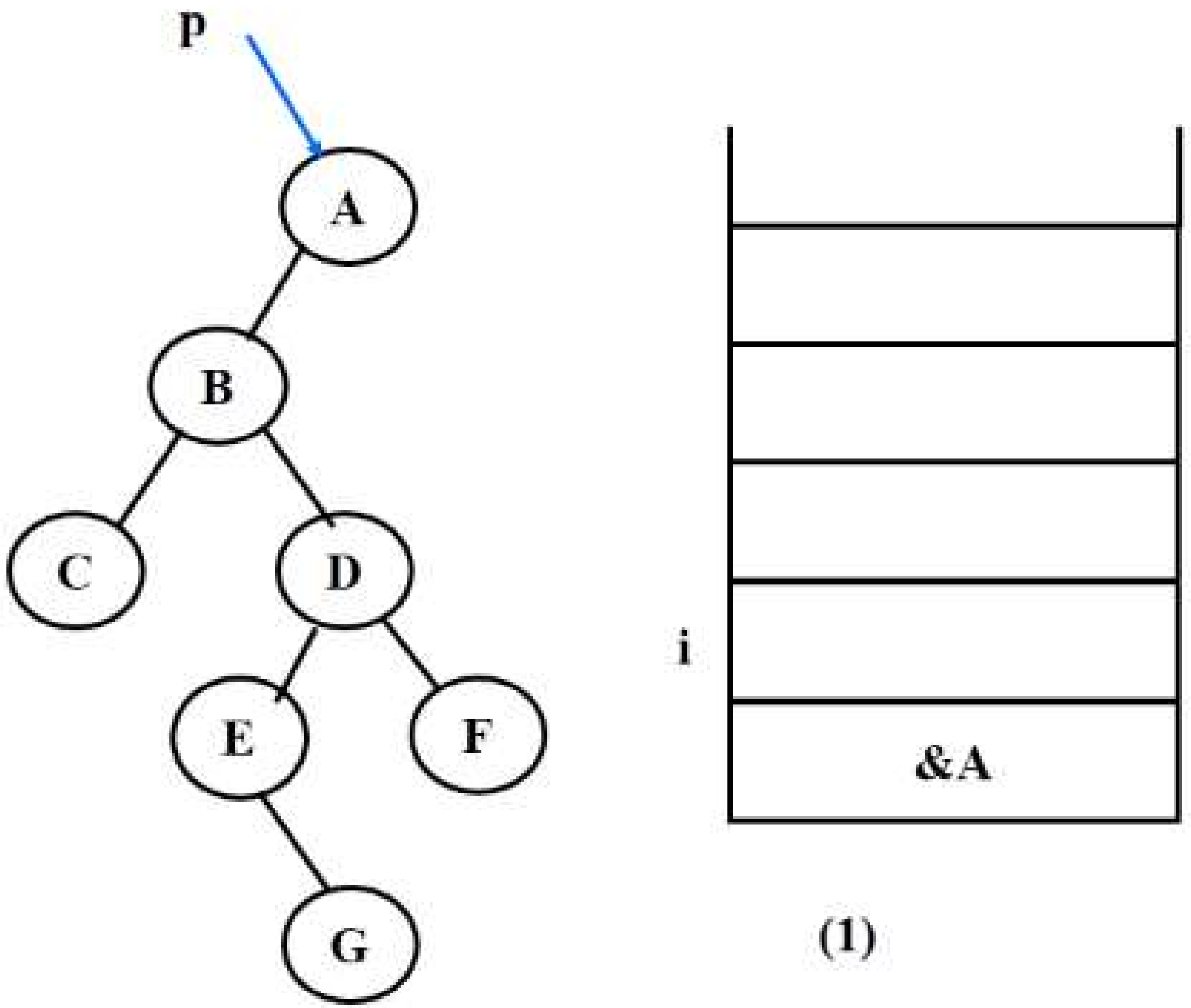
二叉树的遍历

非递归中序遍历BT的算法：

```
Void InorderTraverse(BiTree BT) {
  //采用二叉链表存储结构，中序遍历二叉树T的非递归算法.
  InitStack(S); p=BT;
  while(p||!StackEmpty(S)) {
    if(p) { push(S, p); p=p->lchild; }//根指针进栈，遍历左子树
    else { //根指针退栈，访问根结点，遍历右子树
          pop(S, p); visit(p));
          p=p->rchild;
    }//else
} //InOrderTraverse
```

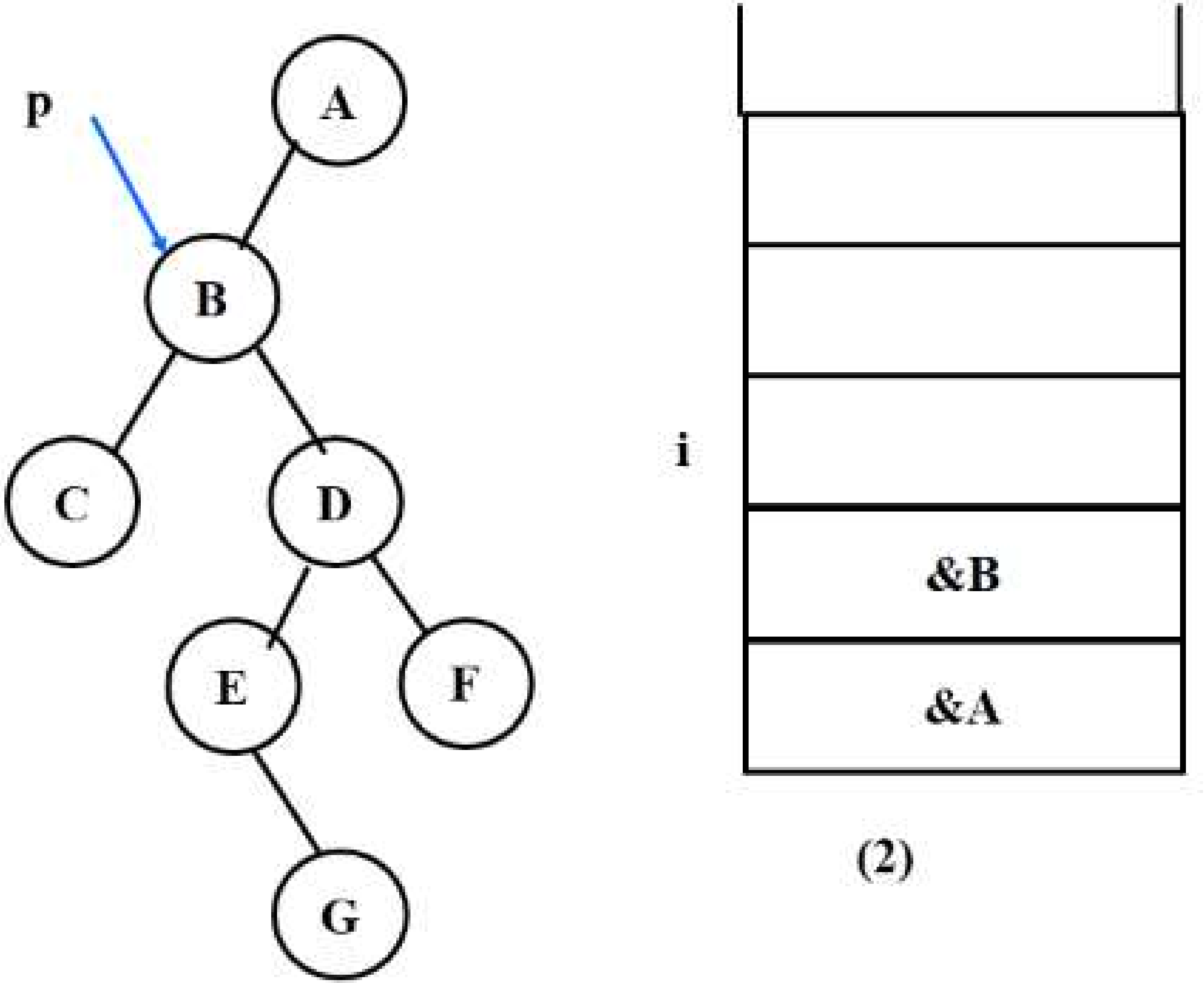
二叉树的遍历

例



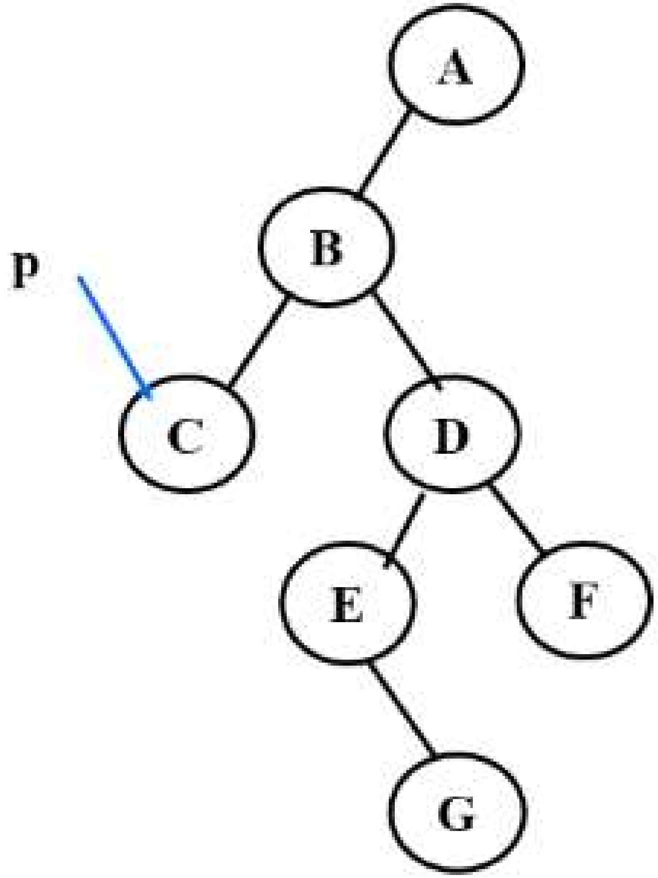
二叉树的遍历

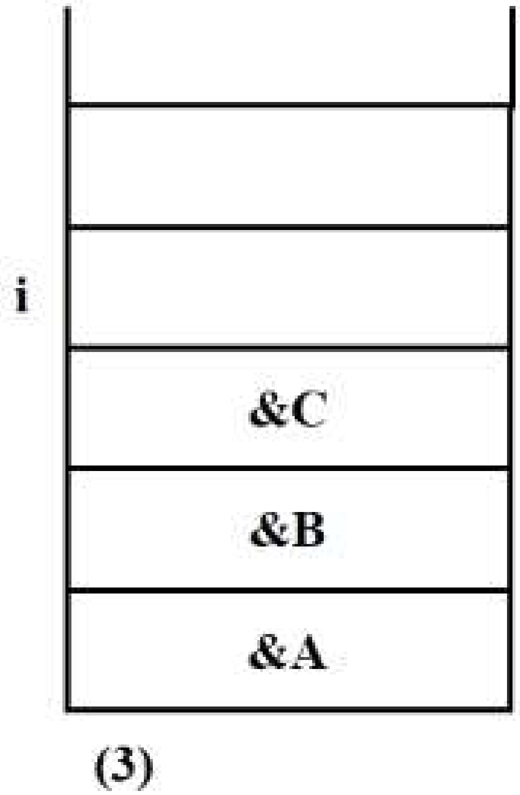
例



二叉树的遍历

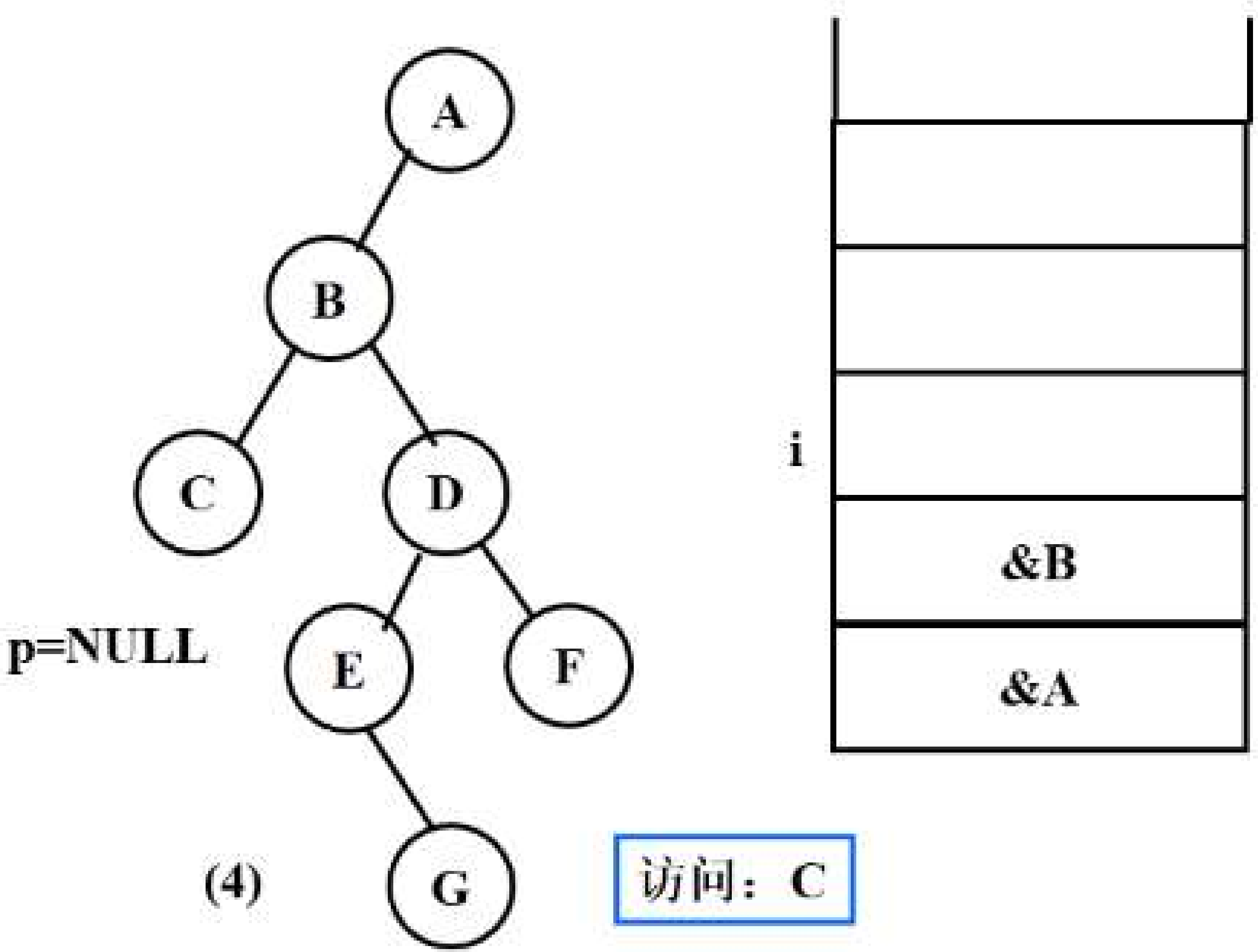
例





二叉树的遍历

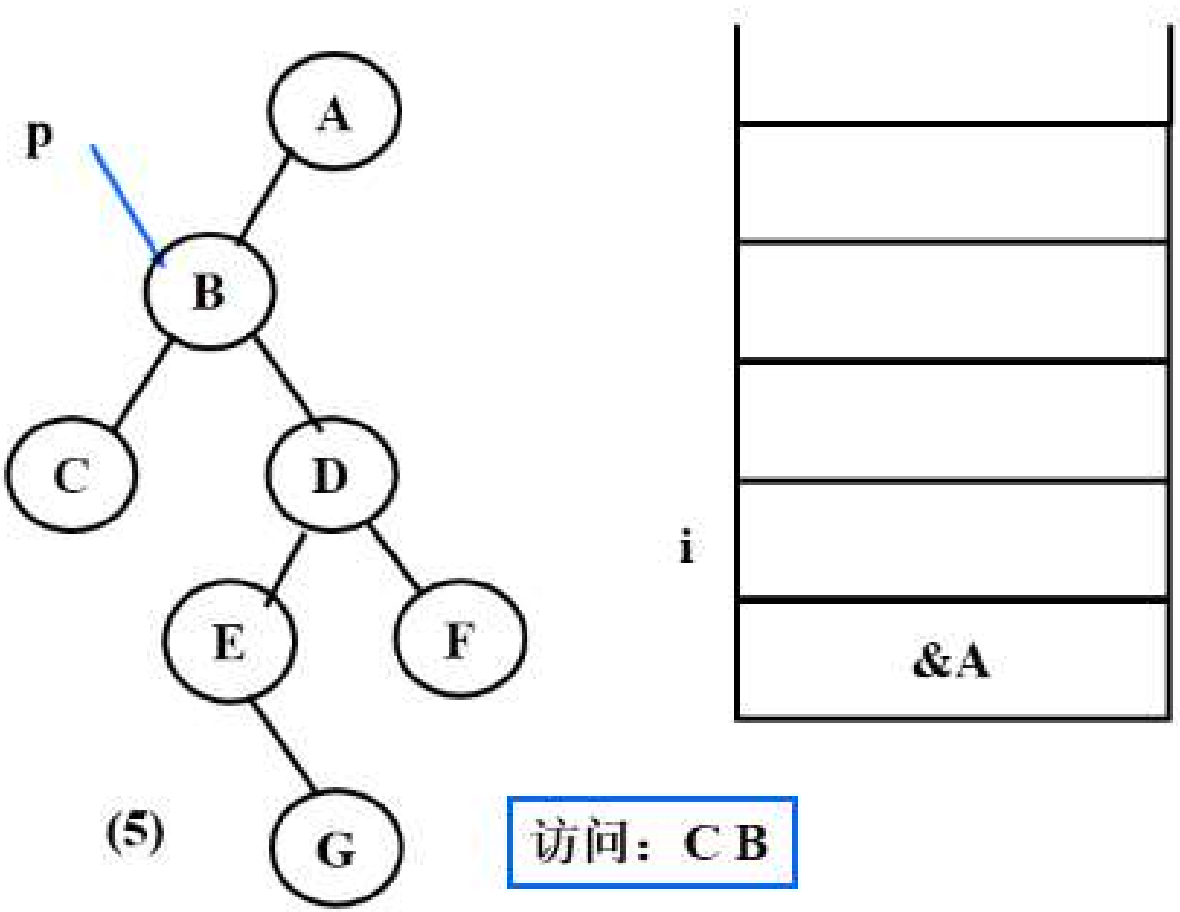
例



访问：C

二叉树的遍历

例

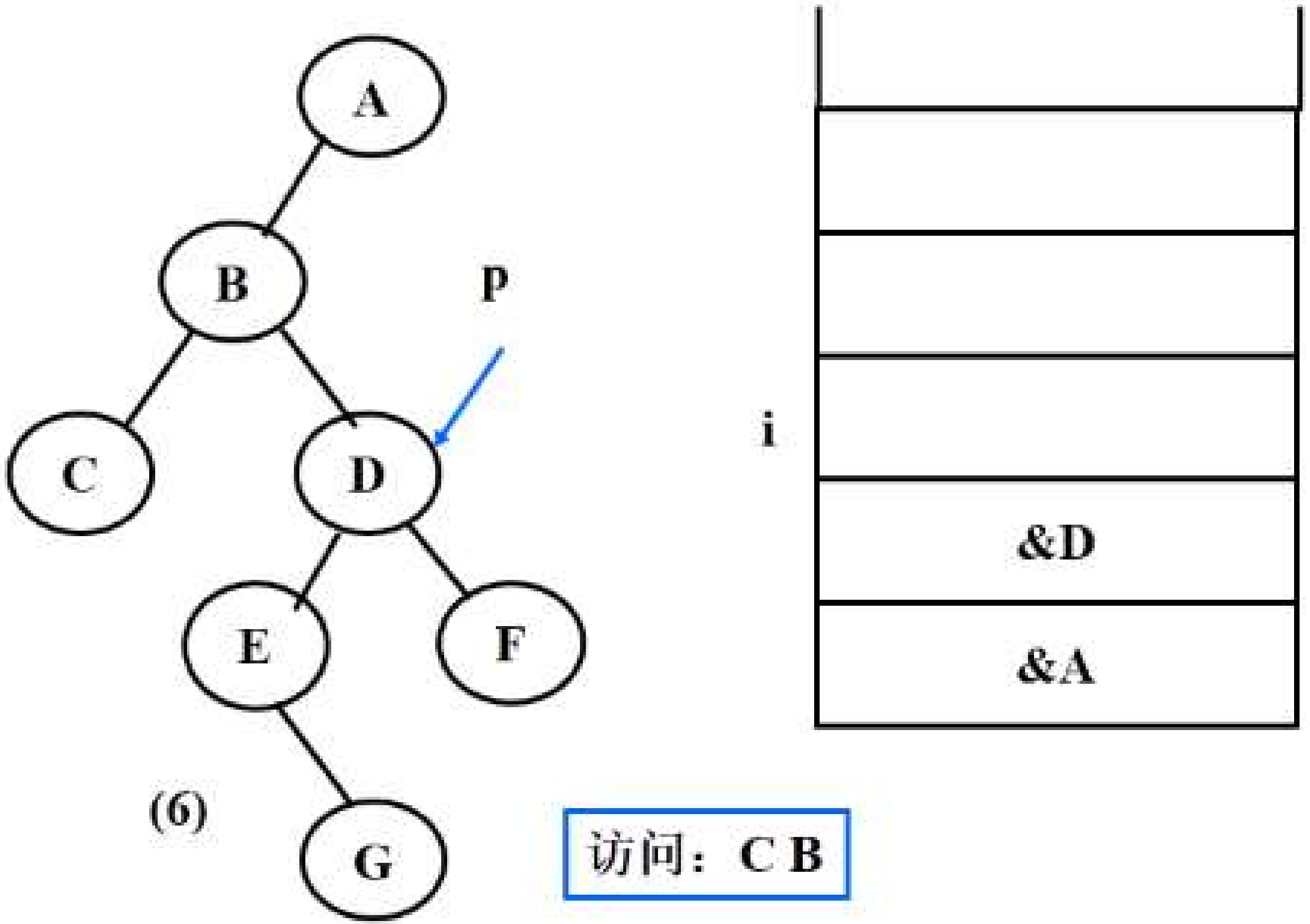


(5)

访问：C B

二叉树的遍历

例

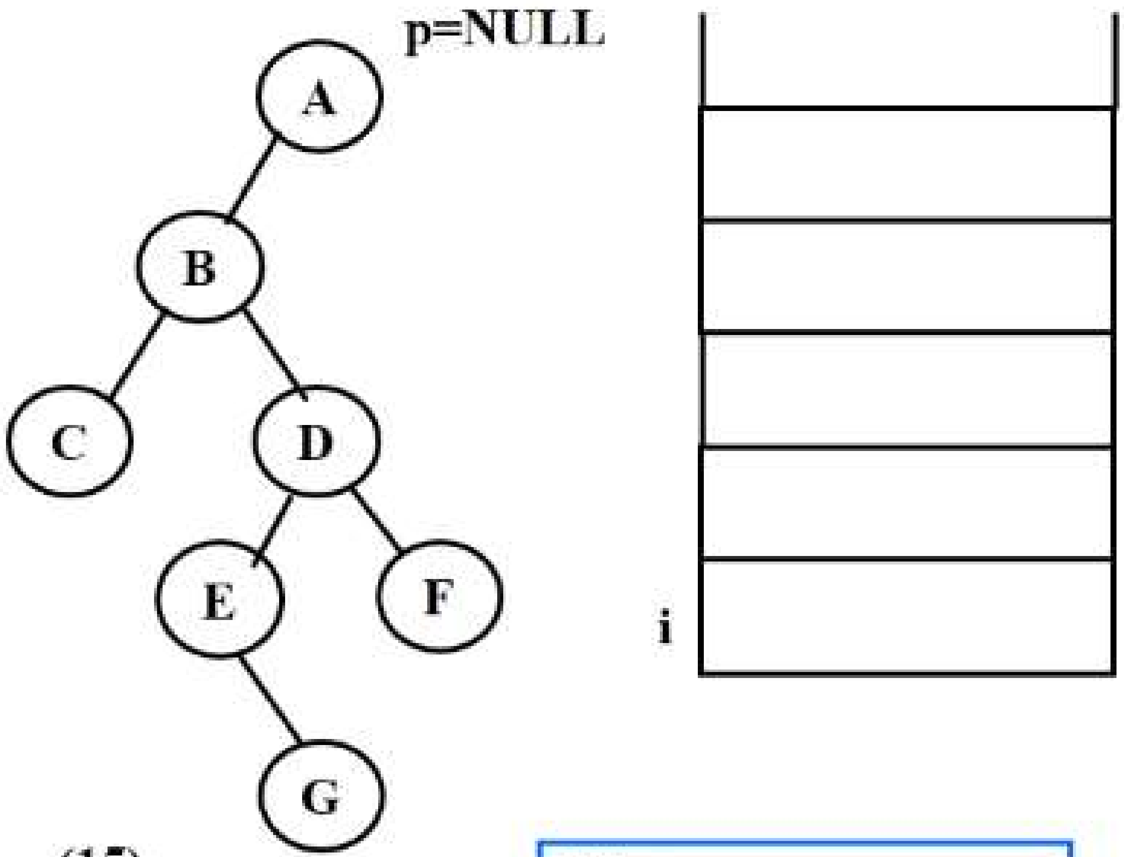


(6)

访问：C B

二叉树的遍历

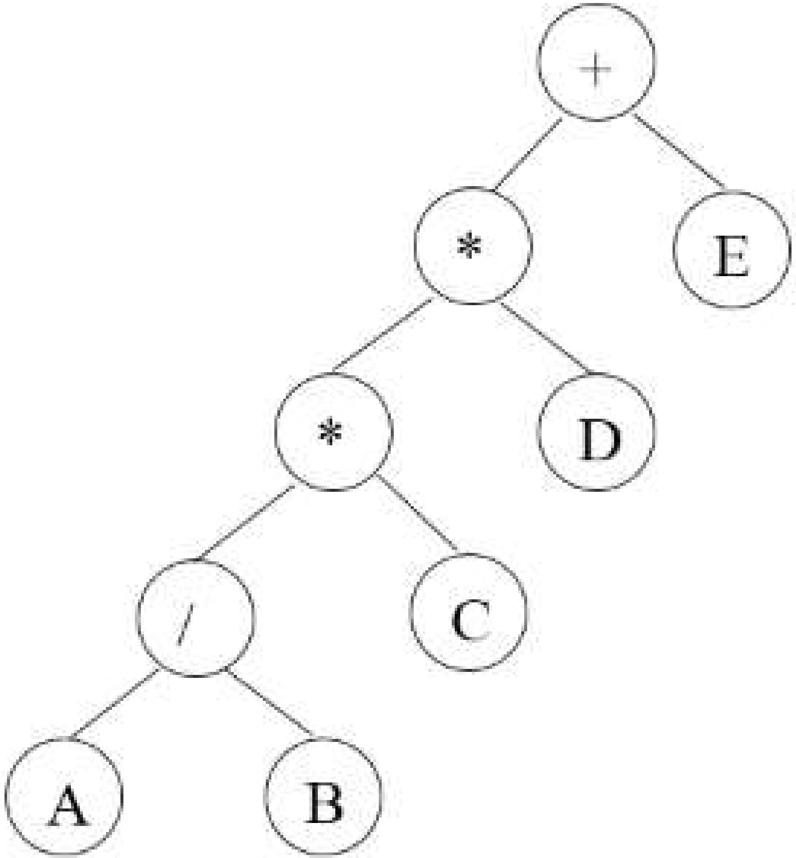
例



(15)

访问：C B E G D F A

二叉树遍历应用



先序遍历

+ * * / A B C D E

前缀表示

中序遍历

A / B * C * D + E

中缀表示

后序遍历

A B / C * D * E +

后缀表示

层序遍历

+ * E * D / C A B

二、线索二叉树(Threaded Binary Tree)

目的：利用二叉树的空指针保存遍历序列的前驱和后继。

- 用空的左指针指向某一遍历序列的前驱.
- 用空的右指针指向某一遍历序列的后继.

这两种指针称为线索(Thread)。

为了区分线索与真实指针，给结点增加两个域Ltag和Rtag

单选题 3分

n个结点的二叉树,有2n个指针,只用了n-1个,有（）个是空指。

A n+1

B n

C n-1

D 2n

二、线索二叉树(Threaded Binary Tree)

lchild	Ltag	data	Rtag	rchild

Ltag=0: lchild 指向结点的左子女;

Ltag=1: lchild 指向某一遍历序列前驱;

Rtag=0: rchild 指向结点的右子女;

Rtag=1: rchild 指向某一遍历序列后继;

二、线索二叉树(Threaded Binary Tree)

```
Typedef  enum{Link,Thread}  PointerTag;

Typedef  struct BiThrNode {

   ElemType    data;

   struct  BiThrNode   *lchild, *rchild;

   PointerTag    Ltag, Rtag;

} BiThrNode, *BiThrTree;
```

lchild	Ltag	data	Rtag	rchild

二、线索二叉树(Threaded Binary Tree)

加了线索的二叉树是线索二叉树；

给二叉树加线索的过程是线索化（穿线）；

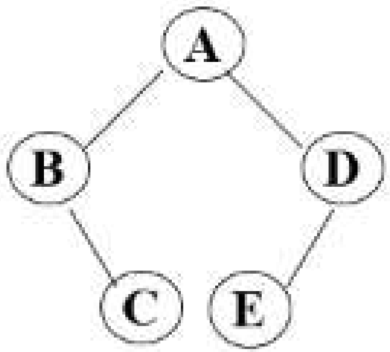
按前序遍历序列穿线的二叉树是前序线索二叉树；

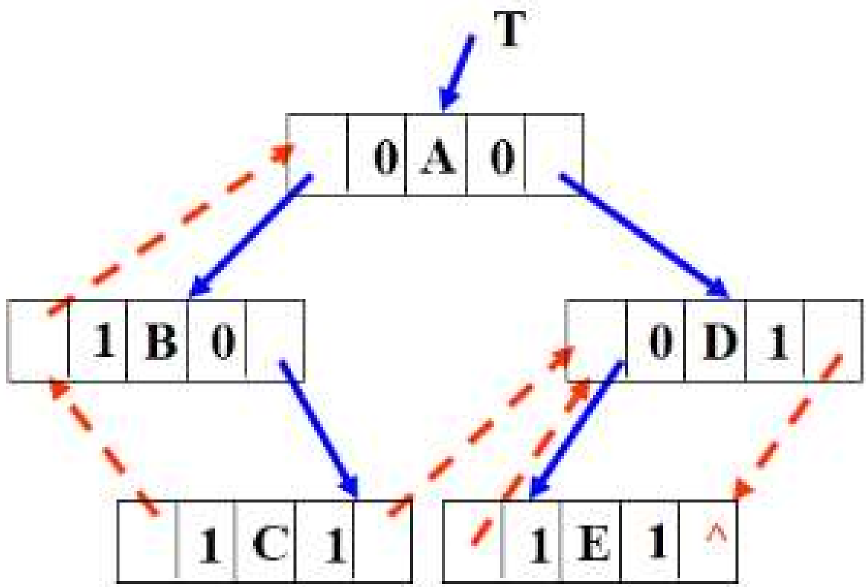
按中序遍历序列穿线的二叉树是中序线索二叉树；

按后序遍历序列穿线的二叉树是后序线索二叉树；

中序线索二叉树简称线索二叉树；

二、 线索二叉树(Threaded Binary Tree)

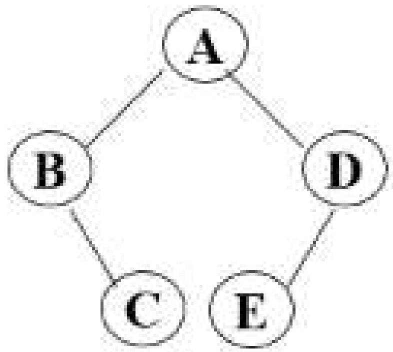


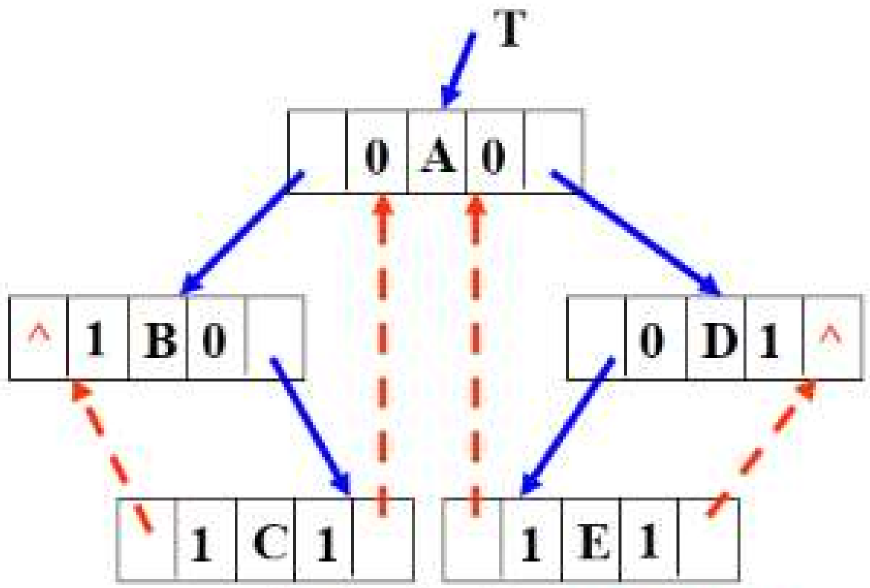


先序序列：ABCDE

先序线索二叉树

二、 线索二叉树(Threaded Binary Tree)



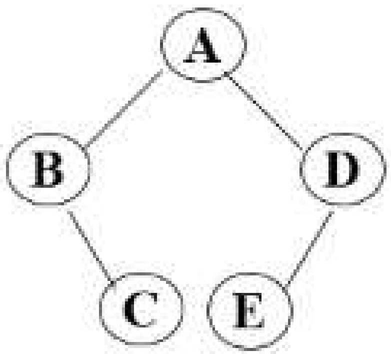


中序序列：BCAED

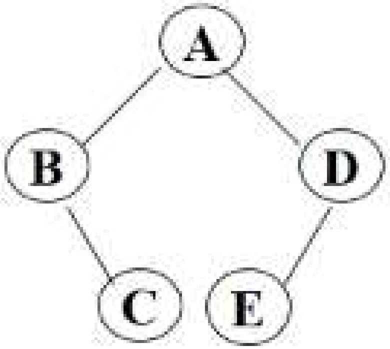
中序线索二叉树

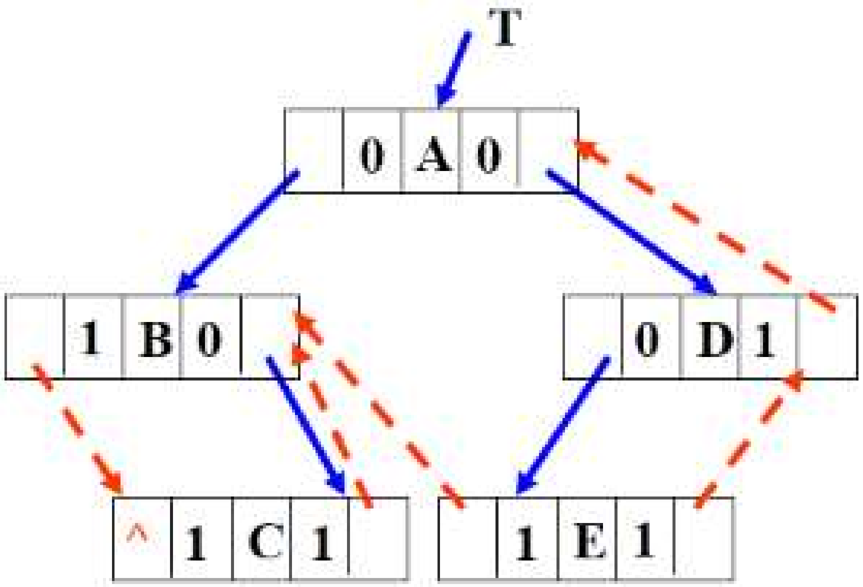
主观题 7分

请完成下面二叉树的线索化过程。



二、 线索二叉树(Threaded Binary Tree)





后序序列：CBEDA

后序线索二叉树

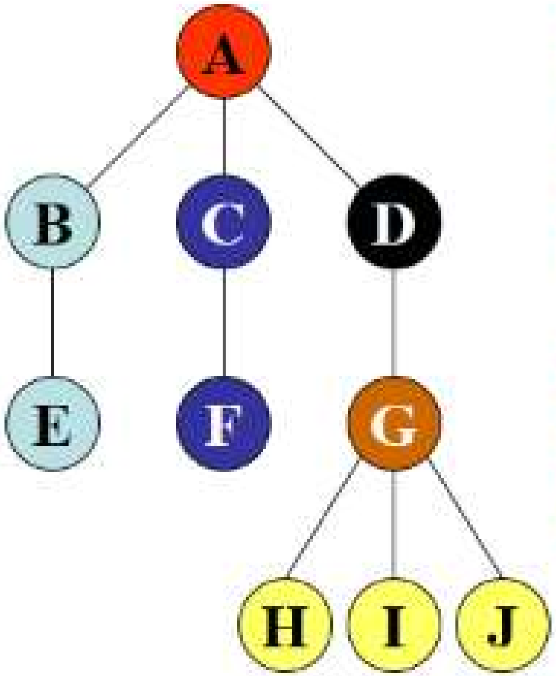
一、 树的存储结构

树的双亲表示法说明：

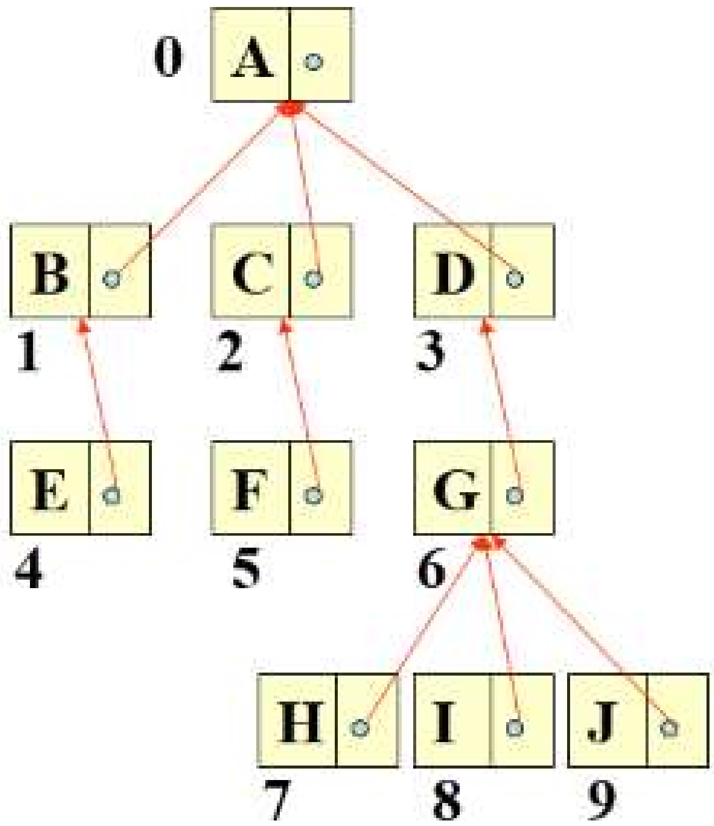
```
#define  MAX-TREE-SIZE   100
typedef struct  PTNode{
      ElementType  data;
      int  parent;  // 该结点的双亲的下标
} PTNode;
typedef   struct  {
      PTNode   nodes[MAX-TREE-SIZE];
      int   n;  //树的结点数
} PTree;
```

一、 树的存储结构

例 用双亲法存储树



0	A	-1
1	B	0
2	C	0
3	D	0
4	E	1
5	F	2
6	G	3
7	H	6
8	I	6
9	J	6
10		
11		



一、 树的存储结构

2、 孩子（子女）表示法

结点结构

Data	child1	child2	child3	child4	…………	childk

对不同的结点,k(度)是不同的,因此应取最大数,即树的度;这种方法不可取;所以最自然的方法是为每个结点建立一个单链表，该单链表存储它的所有孩子，所有结点组成一个数组，称表头数组。表头数组中每一项称孩子链表头指针

CTBox:
(头结点)

data	firstchild

单链表中每一项称孩子结点(也称表结点):

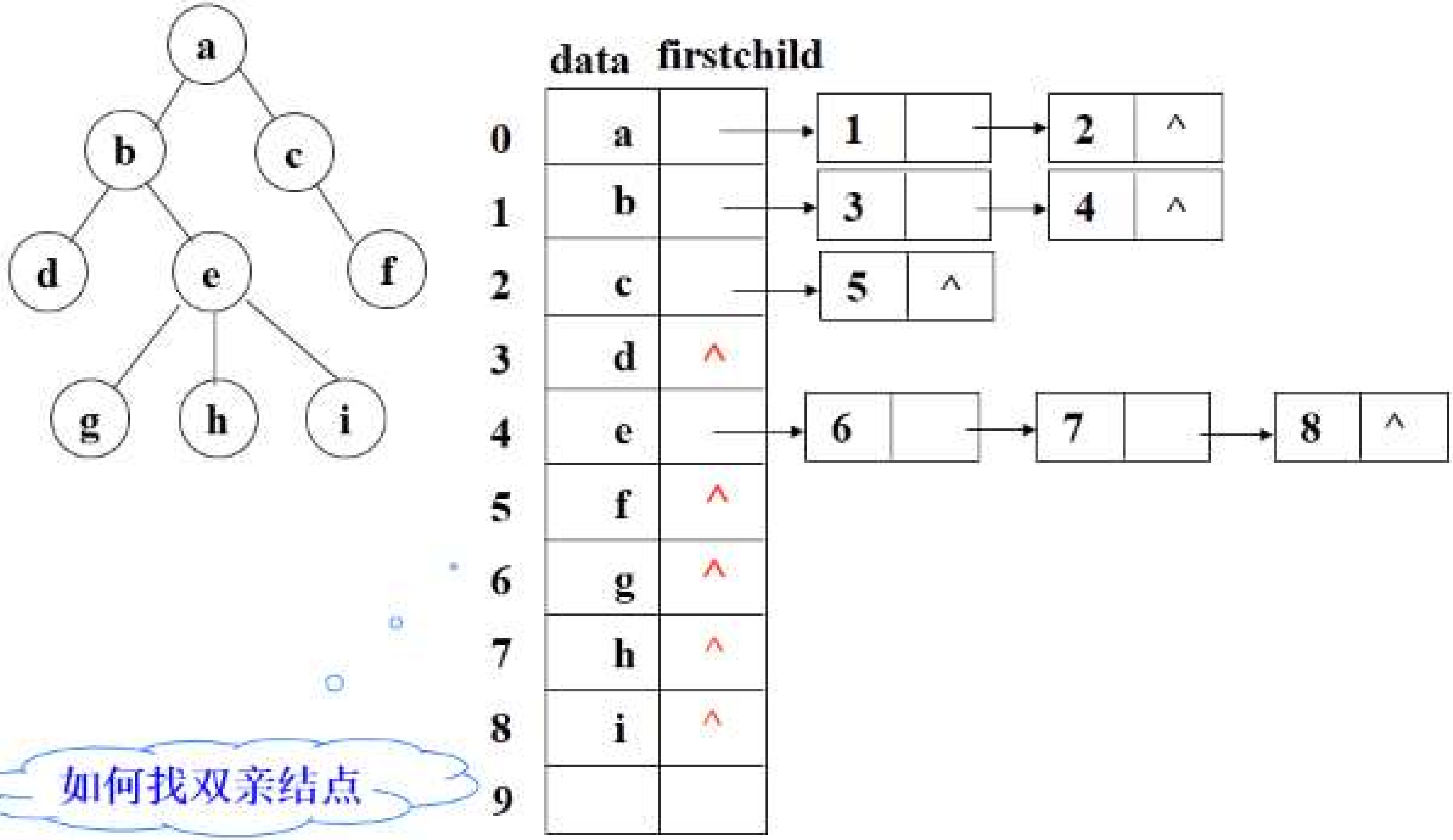
CTNode:

child	next

树的孩子链表存储表示说明：

```
typedef struct CTNode { //孩子结点（表结点）
 int   child;
 struct CTNode  *next;
} *ChildPtr;
tyPedef struct {           //头结点
 TElemType   data;
 ChildPtr       firstchild;
}CTBox;
typedef struct { //孩子链表头指针
CTBox nodes[MAX_TREE_SIZE];
int   n, r;      //结点数和根的位置;
}CTree;
```

2、孩子表示法(子女表示法)



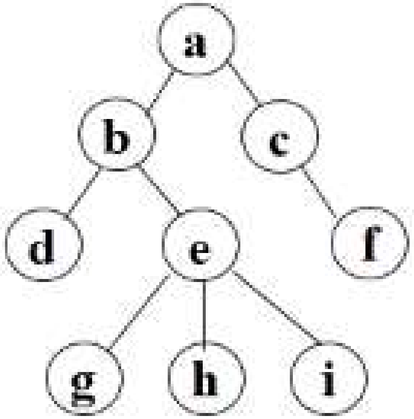
如何找双亲结点

带双亲的孩子链表存储表示:

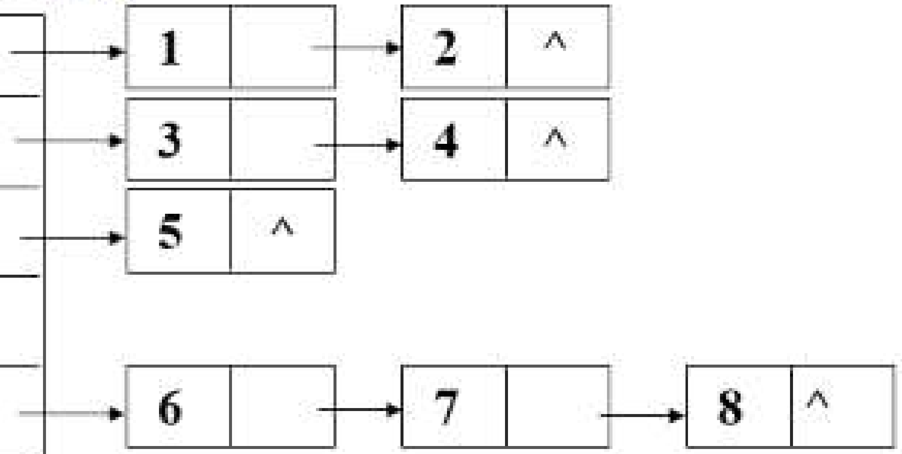
```
typedef struct CTNode { //孩子结点（表结点）
  int   child;
  struct CTNode *next;
} *ChildPtr;
typedef struct{            //头结点
  TElemType  data;     int  parent;
  ChildPtr   firstchild;
}CTBox;
typedef struct { //孩子链表头指针
CTBox  nodes[MAX_TREE_SIZE];
int   n， r;       //结点数和根的位置;
}CTree;
```

2、孩子表示法(子女表示法)

- 带双亲的孩子链表



	data	parent	firstchild
0	a	-1	
1	b	0	
2	c	0	
3	d	1	^
4	e	1	
5	f	2	^
6	g	4	^
7	h	4	^
8	i	4	^



一、 树的存储结构

3、 孩子兄弟表示法(也称二叉树表示法或二叉链表表示法)

结点结构（CSNode）：

data	**firstchild**	**nextsibling**

firstchild：指向该结点的第一个孩子

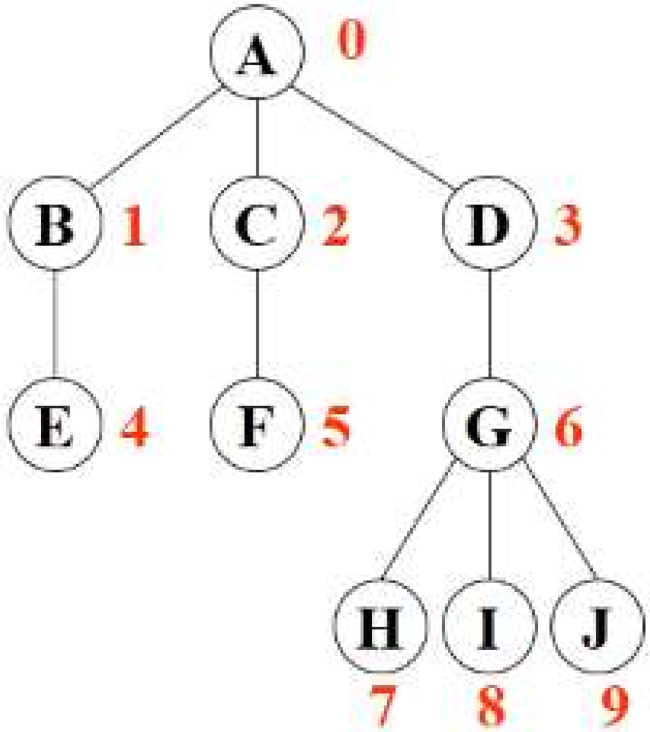
nextsibling：指向该结点的下一个兄弟

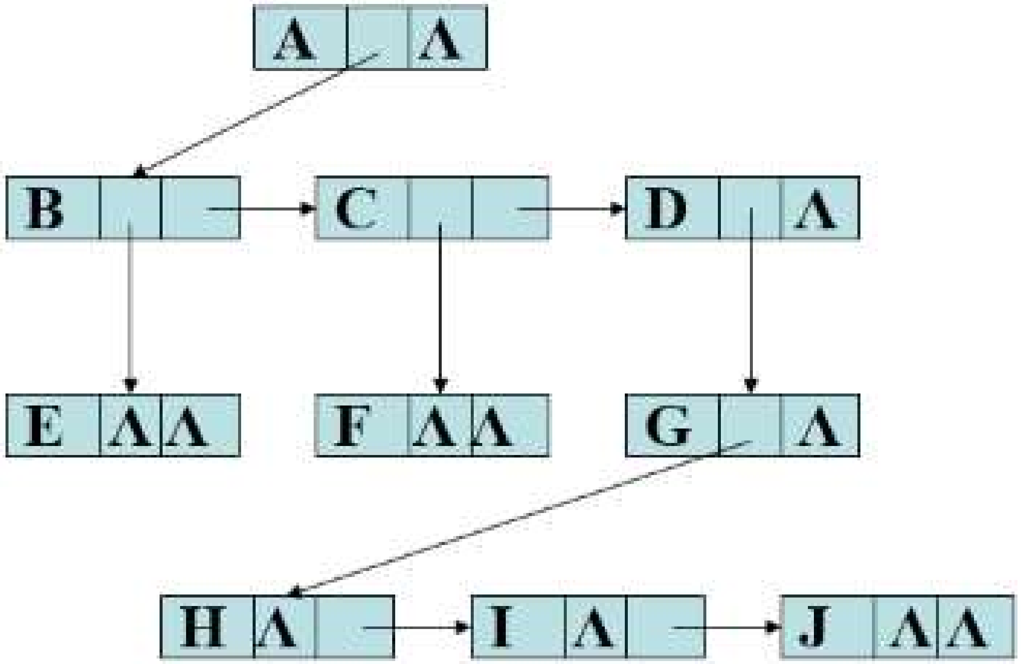
树的孩子链表存储表示

```
typedef struct CSNode {
  TElemType   data;
  struct  CSNode  * firstchild，   * nextsihling;
  }CSNode，*CSTree;
```

一、 树的存储结构

例 用孩子兄弟法存储树





二、 树与森林的遍历

树的遍历: 按根的次序区分有两种遍历次序

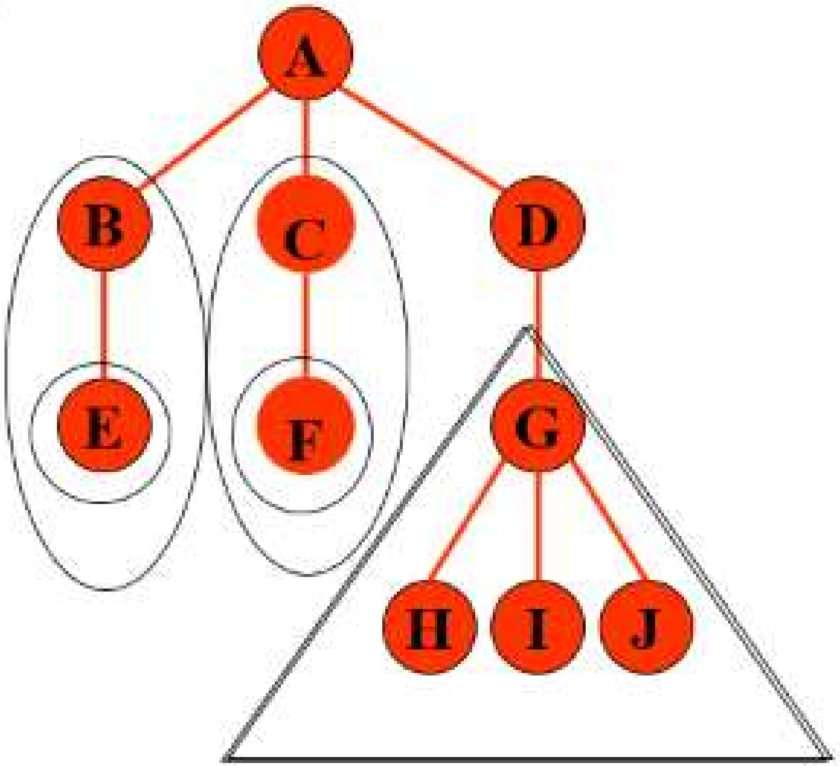
(1) 先根遍历:

若树非空，则

- 访问根结点;
- 从左到右先根遍历根的每棵子树;

二、 树与森林的遍历

例 先根遍历树



先根遍历序列:

A B E C F D G H I J

二、 树与森林的遍历

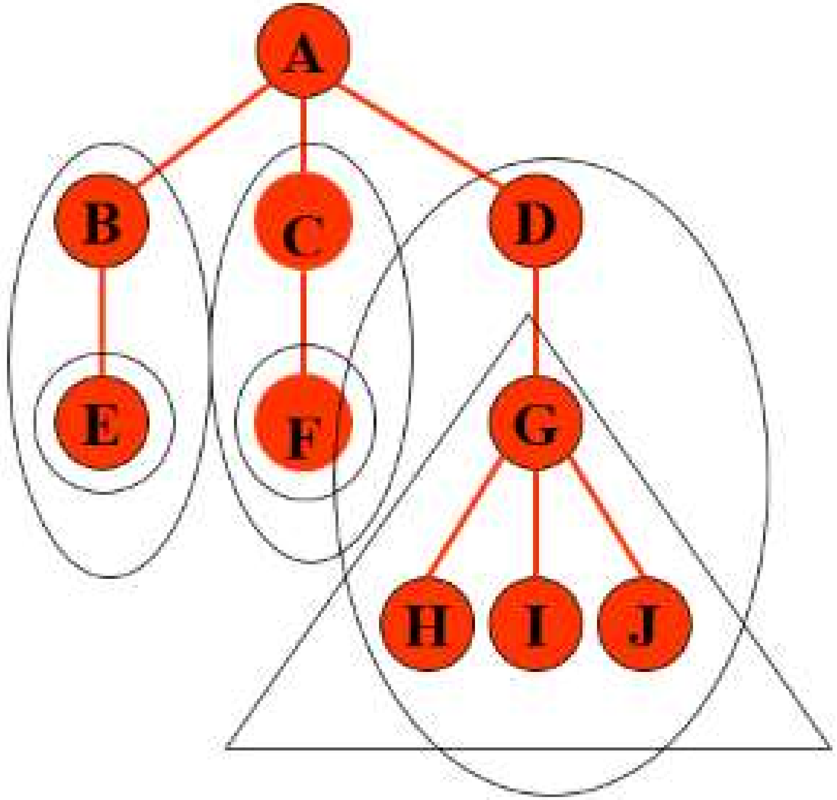
（2）后根遍历:

若树非空，则

- 从左到右后根次序遍历根的每棵子树;
- 访问根结点;

二、 树与森林的遍历

例 后根遍历树



后根遍历序列：

E B F C H I J G D A

二、 树与森林的遍历

森林的遍历:

森林的遍历是基于树的遍历完成的,对应有两种遍历次序:

(1) 先序遍历:

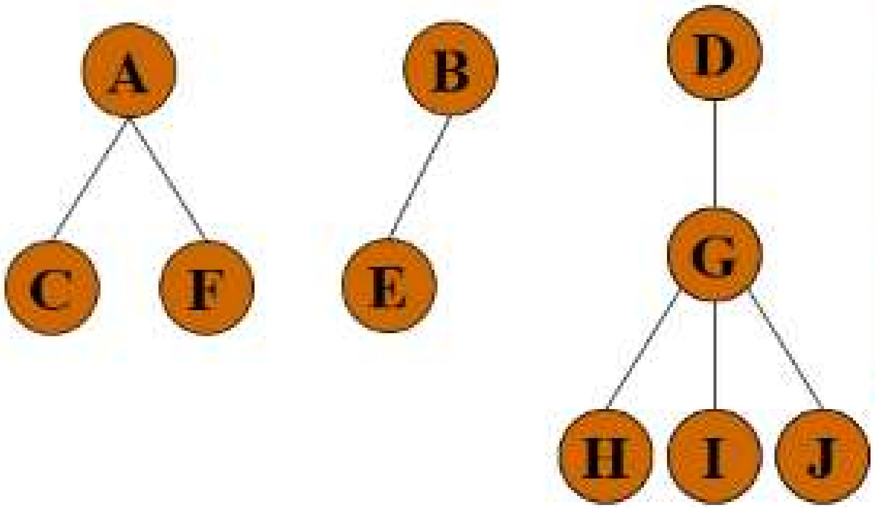
- 访问第一棵树的根;
- 先序遍历第一棵树中的根结点的子树森林;
- 先序遍历其余的树所构成的森林;

(2) 中序遍历:

- 中序遍历第一棵树的子树;
- 访问第一棵树的根;
- 中序遍历其余的树所构成的森林;

二、 树与森林的遍历

先序遍历森林



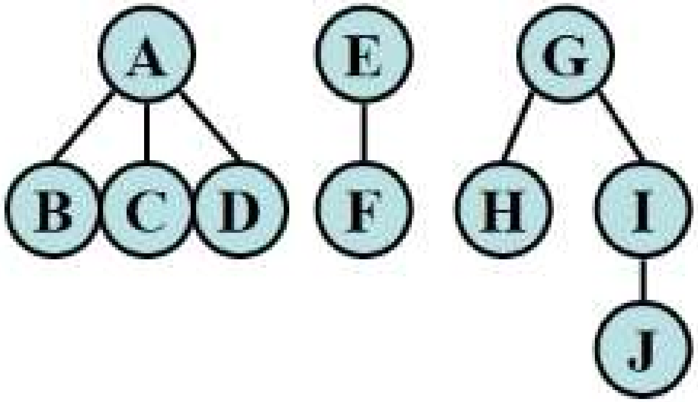
先序遍历：

- 访问第一棵树的根;
- 先序遍历第一棵树中的根结点的子树森林;
- 先序遍历其余的树所构成的森林;

先序遍历序列： A C F B E D G H I J

二、 树与森林的遍历

中序遍历森林



中序遍历:

- 中序遍历第一棵树的子树;
- 访问第一棵树的根;
- 中序遍历其余的树所构成的森林;

中序遍历序列：**BCDAFEHJIG**

三、 森林与二叉树的转换

在森林与二叉树 之间存在一一对应的关系。

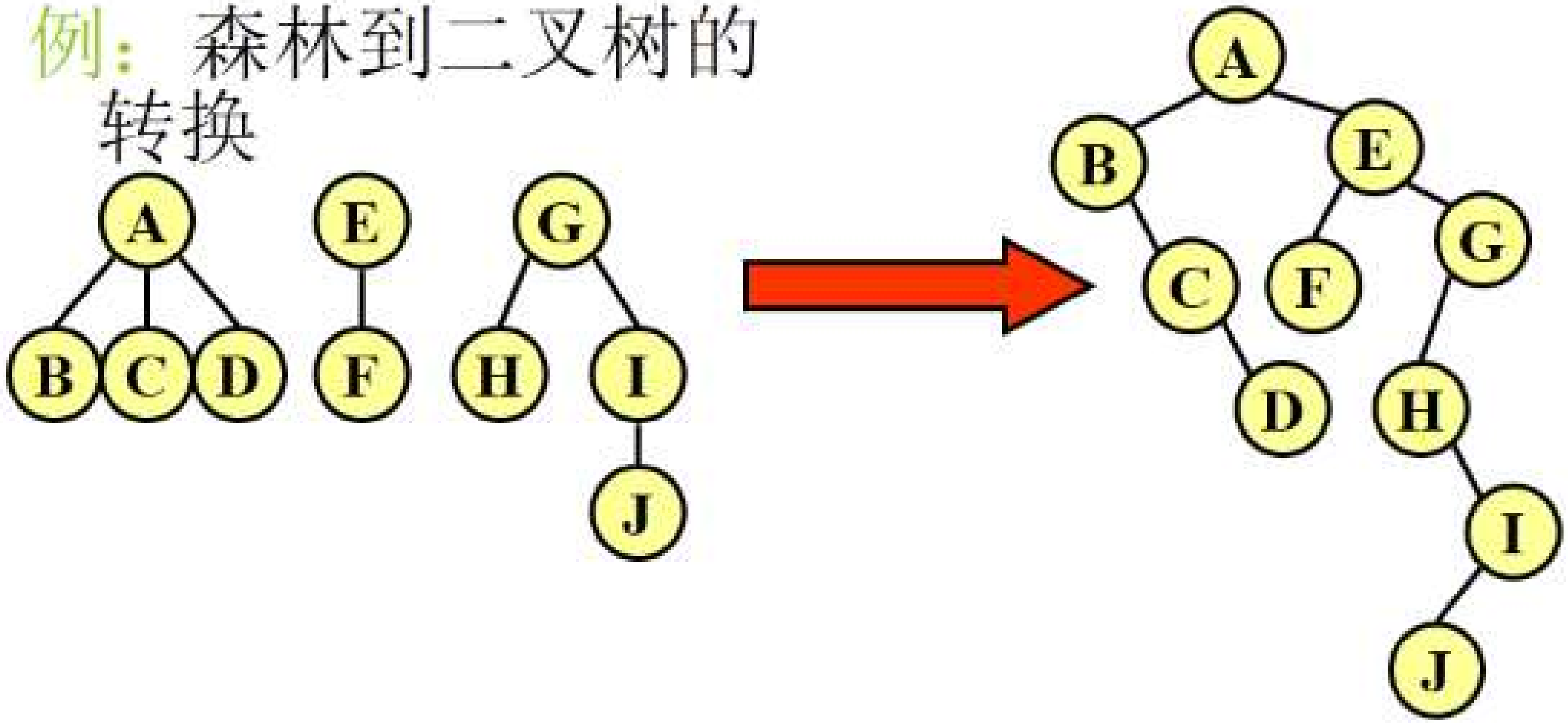
1). 森林=>二叉树的转换

自然转换法:

凡是兄弟用线连起来，然后去掉双亲到子女的连线，但保留双亲到其第一子女的连线，最后转45°。

三、 森林与二叉树的转换

例：森林到二叉树的转换



前序序列：ABCDEFGHIJ ＝前序序列：ABCDEFGHIJ

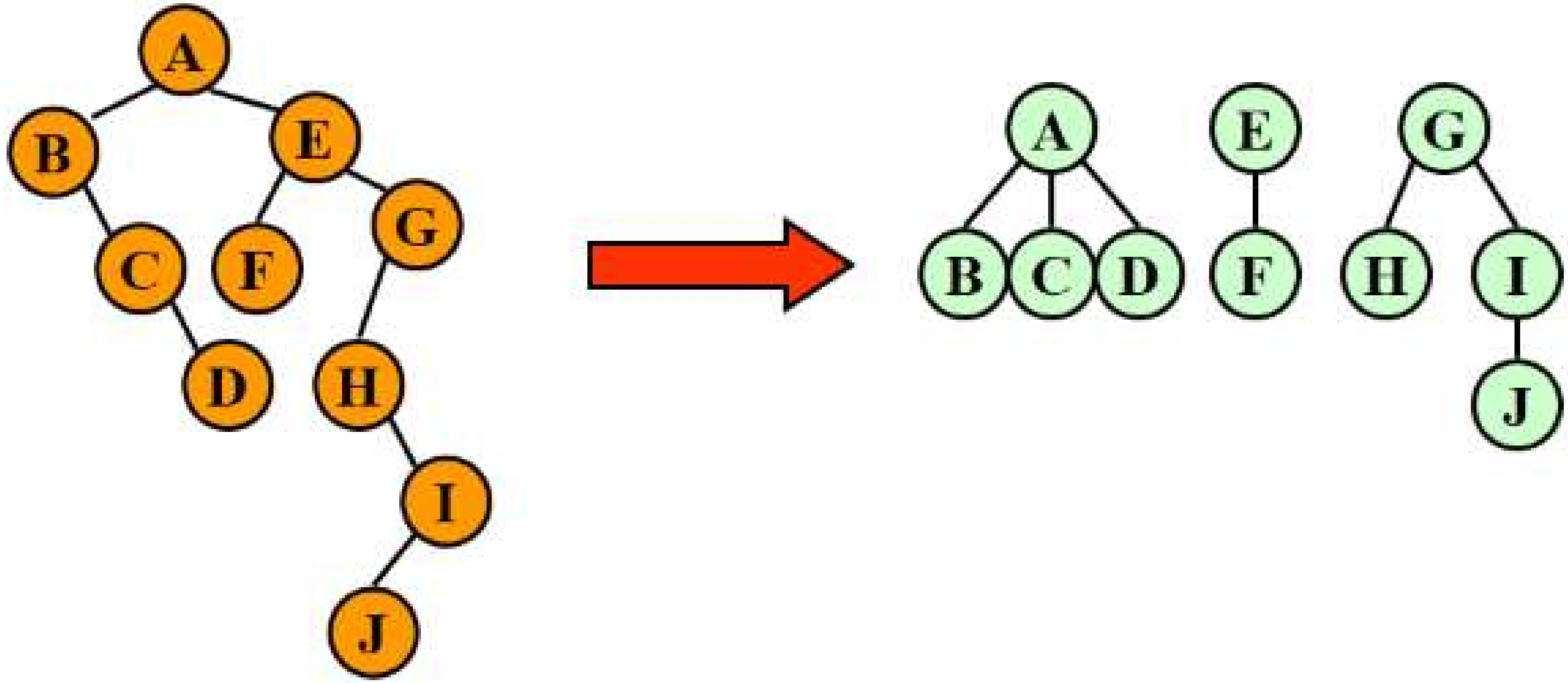
中序序列：BCDAFEHJIG ＝中序序列：BCDAFEHJIG

三、 森林与二叉树的转换

2) 二叉树=>森林的转换

自然转换法:

若某结点是其双亲的左孩子，则该结点的右孩子、右孩子的右孩子 …, 都与该结点的双亲连接起来,最后去掉所有双亲到右孩子的连线.


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

6.6 Huffman树及其应用

- 最优二叉树（**Huffman**）
- 如何构造最优二叉树
- 如何求哈夫曼编码

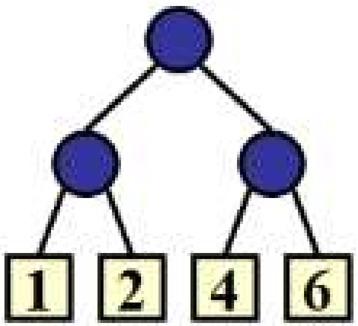
一、哈夫曼树(最优树)的定义

- 结点的路径长度：从根结点到该结点的路径上分支的数目。
- 树的路径长度：树中每个结点的路径长度之和。
- (结点)带权路径长度:结点的路径长度*结点的权=l_i*w_i
- 树的带权路径长度：树中所有叶结点的带权路径长度之和WPL(T)= $\sum_{k=1}^{n} w_k l_k$

一、哈夫曼树(最优树)的定义

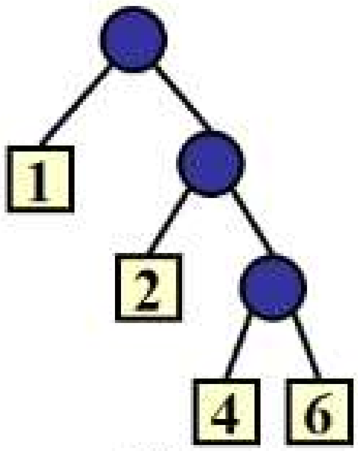
设K={k_1, k_2, …k_n}，它们的权**W**={w_1, w_2, …, w_n}，构造以k_1, k_2, …k_n为叶结点的二叉树，使得**WPL**=$\sum_{k=1}^{n} w_k l_k$为最小，则称之为“最优二叉树”。

例　W={1,2,4,6}，可构造出如下的二叉树：



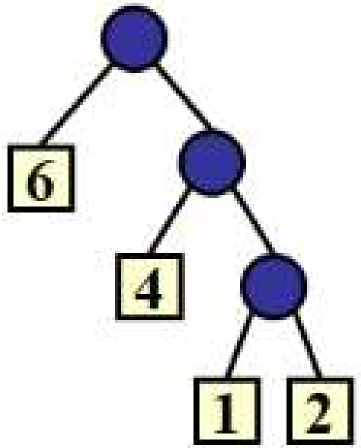
(1)

WPL =(1+2+4+6)*2=26



(2)

WPL =1+2*2+(4+6)*3=35



(3)

WPL =6+4*2+(1+2)*3=23

根据定义求Huffman树的方法是：对给定的n个叶子结点（外部结点），构造出全部二叉树并求出其WPL，然后找出WPL最小的树。

当n较大时，显然这种方法是不可取的。