第6章 树和二叉树

- 树的定义和基本术语
- 二叉树
- 二叉树的存储结构
- 遍历二叉树
- 线索化二叉树
- 树和森林
- 赫夫曼树
- 二叉树的计数

6.1 树的定义和基本术语

■1. 树的定义

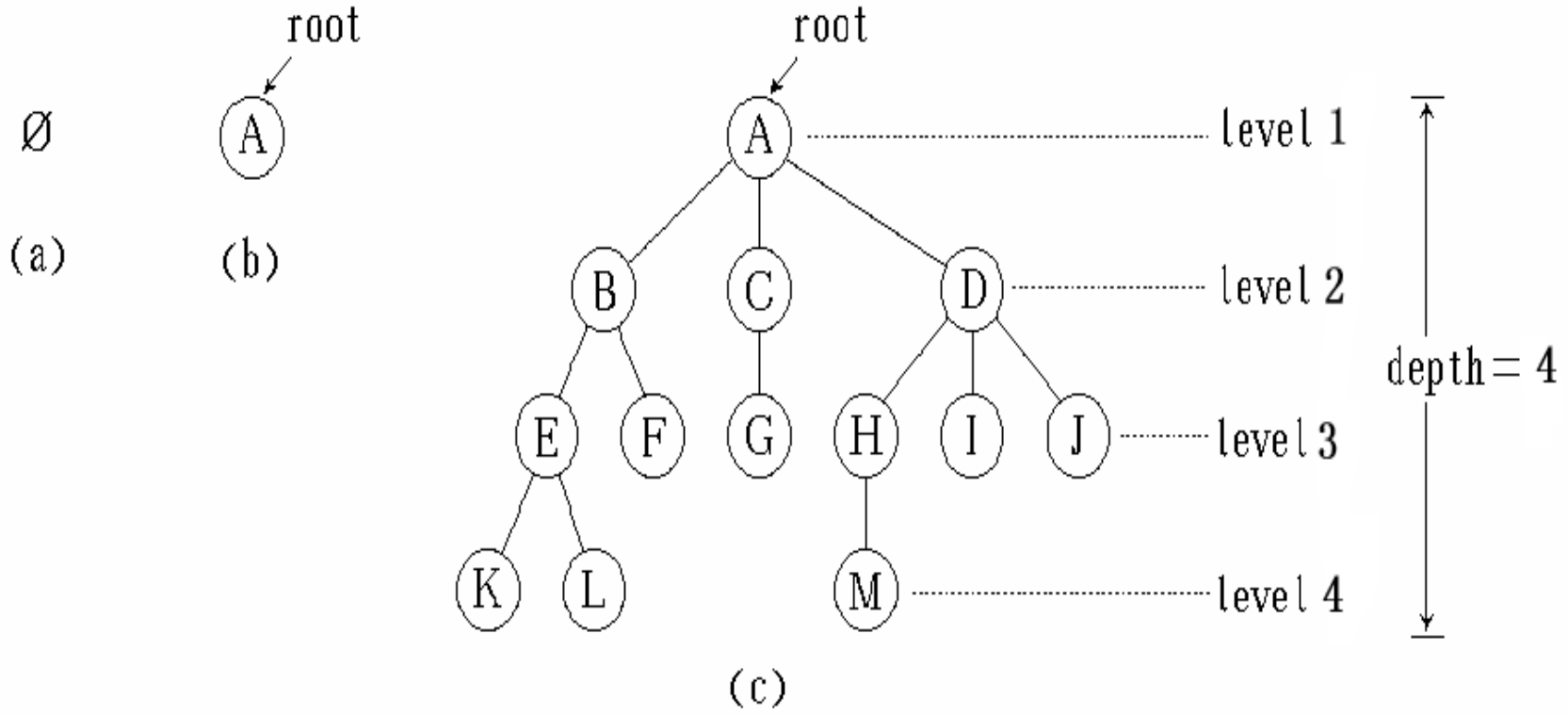
树是由n (n≥0)个结点组成的有限集合。

如果n = 0，称为空树；

如果n > 0，则：

有一个特定的称之为根(root)的结点，它只有后继，但没有前驱；

除根以外的其它结点划分为m (m ≥ 0)个互不相交的有限集合T0，T1，…，Tm-1，每个集合本身又是一棵树，并且称之为根的子树(SubTree)。每棵子树的根结点有且仅有一个直接前驱，但可以有0个或多个后继。


root
Ø
A
(a)
(b)
root
A
B
C
D
E
F
G
H
I
J
K
L
M
level 1
level 2
level 3
level 4
depth = 4
(c)

■2．树的基本术语

1）度（次数、级）

结点：包含一个数据元素及若干指向其子树的分支。

结点的度：一个结点所拥有的子树的个数

树的度：树内各结点的度的最大值

叶子（终端结点）：度为0的结点

分支（非终端）结点：度不为0的结点

2）描述上下及左右的关系

孩子：一个结点的子树的根

双亲：B结点是A结点的孩子，则A结点是B结点的双亲

兄弟：同一个双亲的孩子之间互称兄弟

祖先：结点的祖先是从根到该结点所经分支上的所有结点

子孙：以某结点为根的子树中任意结点都称为该结点的子孙

■2．树的基本术语

3）层次

结点的层次：根结点的层定义为1，其孩子结点的层为第二层，依次类推。

树的深度（高度）：树中最大的结点层

4）路径：树中的结点序列，前一个结点是后一个结点的双亲或孩子。

路径的长度：该路径经过的边的数目。

5）有序树：子树有序的树，如家族树

6）无序树：不考虑子树的顺序

7）森林：m(m>=0)棵互不相交的树的集合。树去掉根结点，其子树就构成一个森林。

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；

- 若D仅含一个数据元素，则R为空集，
- 否则R={H}，H是如下二元关系：

（1）在D中存在唯一的称为根的数据元素root，它在关系H下无前驱；

（2）若D-{root}≠∅，则存在D-{root}的一个划分$D_1, D_2, \ldots, D_m (m>0)$，对任意j≠k$(1 \le j, k \le m)$有$D_j \cap D_k = \emptyset$，且对任意的i(1≤i≤m)，存在唯一数据元素$x_i \in D_i$，有<root，$x_i$>∈ H；

树的抽象数据类型

（3）对应于D-{root}的划分，H-{<root,x_1>,…,<root,x_m>,}有唯一的一个划分$H_1, H_2, \ldots, H_m (m>0)$，对任意$j \neq k (1 \leq j,k \leq m)$有$H_j \cap H_k = \varnothing$，且对任意的$i (1 \leq i \leq m)$，$H_i$是$D_i$上的二元关系，$(D_i, \{H_i\})$是一棵符合本定义的树，称为根root的子树。

基本操作P:

① InitTree(&T);
② DestroyTree(&T);
③ CreateTree(&T,definition);
④ ClearTree(&T);
⑤ TreeEmpty(T);

⑥ TreeDepth(T);
⑦ Root(T);
⑧ Value(T,cur_e);
⑨ Assign(T,cur_e,value);//
⑩ Parent(T,cur_e);//返回cur_e的双亲结点
⑪ LeftChild(T,cur_e);//返回cur_e的最左孩子
⑫ RightSibling(T,cur_e);//返回cur_e的右兄弟
⑬ InsertChild(&T,&p,i,c);//插入c为T中p所指结点的第i棵子树
⑭ DeleteChild(&T,&p,i);//删除T中p所指结点的第i棵子树
⑮ TraverseTree(T,visit());//遍历,对每个结点执行visit函数

}ADT Tree

- **树的建立和遍历是重点**

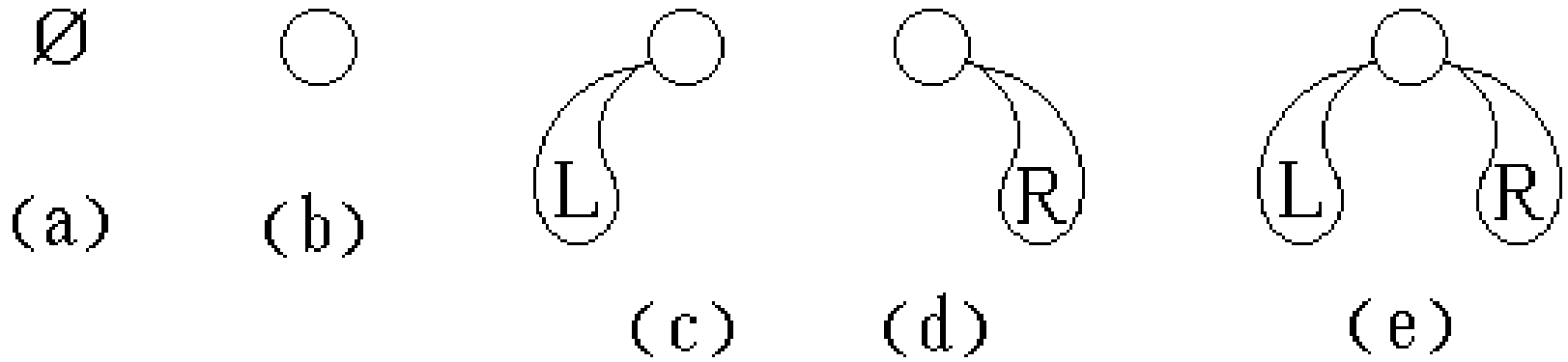
6.2 二叉树 (Binary Tree)

- **二叉树的定义:**

  一棵二叉树是结点的一个有限集合，该集合或者为空，或者是由一个根结点加上两棵分别称为左子树和右子树的、互不相交的二叉树组成。

  特点：1）每个结点的度≤2;

  2）是有序树



二叉树的五种不同形态

二叉树的抽象数据类型

ADT BinaryTree{

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=∅，则R=∅，称BinaryTree为空二叉树；
- 若D ≠ ∅，则R={H}，H是如下二元关系：

（1）在D中存在唯一的称为根的数据元素root，它在关系H下无前驱；

（2）若D-{root}≠ ∅，则存在D-{root}={D_1, D_2}，且$D_1 \cap D_2$=∅；

（3）若$D_1 \neq \emptyset$，则D_1中存在唯一的元素x_1，<root,x_1>∈ H,且存在D_1上的关系H_1 ⊂H;若$D_2 \neq \emptyset$，则D_2中存在唯一的元素x_2，<root,x_2>∈ H,且存在D_2上的关系H_2 ⊂H; H ={<root,x_1>, <root,x_2>,H_1,H_2};

（4）(D_1,{H_1})是一棵符合本定义的二叉树，称为根的左子树，(D_2,{H_2})是一棵符合本定义的二叉树，称为根的右子树。

基本操作P:

① InitBiTree(&T);
② DestroyBiTree(&T);
③ CreateBiTree(&T,definition);
④ ClearBiTree(&T);
⑤ BiTreeEmpty(T);

⑥ BiTreeDepth(T);
⑦ Root(T);//返回T的根
⑧ Value(T,e);//e是T中某结点，返回e的值
⑨ Assign(T,&e,value);//结点e赋值为value
⑩ Parent(T,e);//返回e的双亲结点
⑪ LeftChild(T,e);//返回e的左孩子
⑫ RightChild(T,e);//返回e的右孩子
⑬ LeftSibling(T,e);//返回e的左兄弟
⑭ RightSibling(T,e);//返回e的右兄弟
⑮ InsertChild(&T,&p,LR,c);// LR为0或1,插入c为p的左或右子树
⑯ DeleteChild(&T,&p,LR);//删除T中p所指结点的左或右子树

⑰ PreOrderTraverse(T,visit());//先序遍历,对每个结点执行visit函数

⑱ InOrderTraverse(T,visit());//中序遍历PostOrderTraverse(T,visit());//后序遍历

⑲ LevelOrderTraverse(T,visit());//层序遍历

}ADT BinaryTree

二叉树的性质

- **性质1**

  若二叉树的层次从1开始，则在二叉树的第 i 层最多有 2^{i-1} 个结点。(i ≥ 1)

  [证明用数学归纳法]

- **性质2**

  深度为k的二叉树最多有 2^k-1个结点。(k≥1)

  [证明用求等比级数前k项和的公式]

- **性质3**

  对任何一棵二叉树，如果其叶结点个数为 n0，度为2的非叶结点个数为 n2，则有

  n0＝n2＋1

■ 证明:

若设度为0的结点有n0个，度为1的结点有n1个，度为2的结点有n2个,总结点个数为n，总边数为e，则根据二叉树的定义，

n = n0 + n1 + n2 (1)

e = 2n2 + n1 (2)

n = e + 1 (3)

由（1）（2）（3）可得: n0 = n2 + 1

■ 同理:

三次树: n0=1+n2+2n3

四次树: n0=1+n2+2n3+3n4

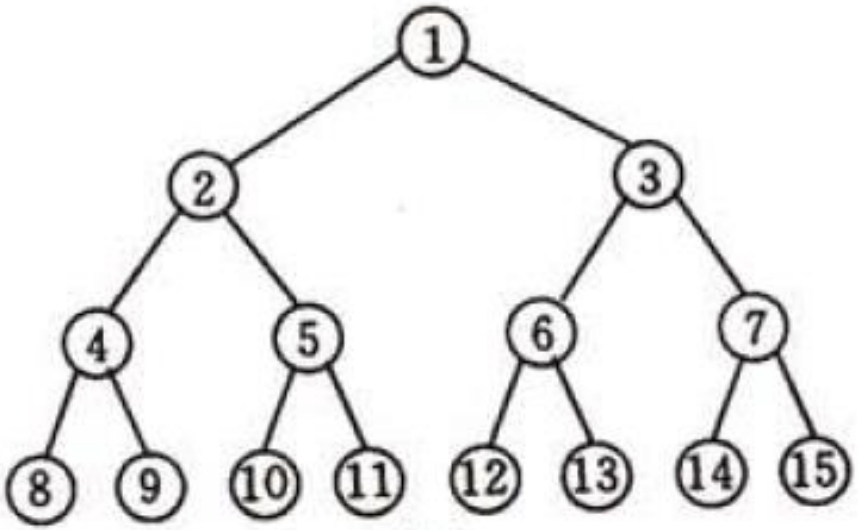
…

K次树: n0=1+n2+2n3+…+(k-1)nk

满二叉树

- **定义1 满二叉树**

  深度为k且有2^k-1个结点的二叉树是满二叉树
  （根结点的深度为1）

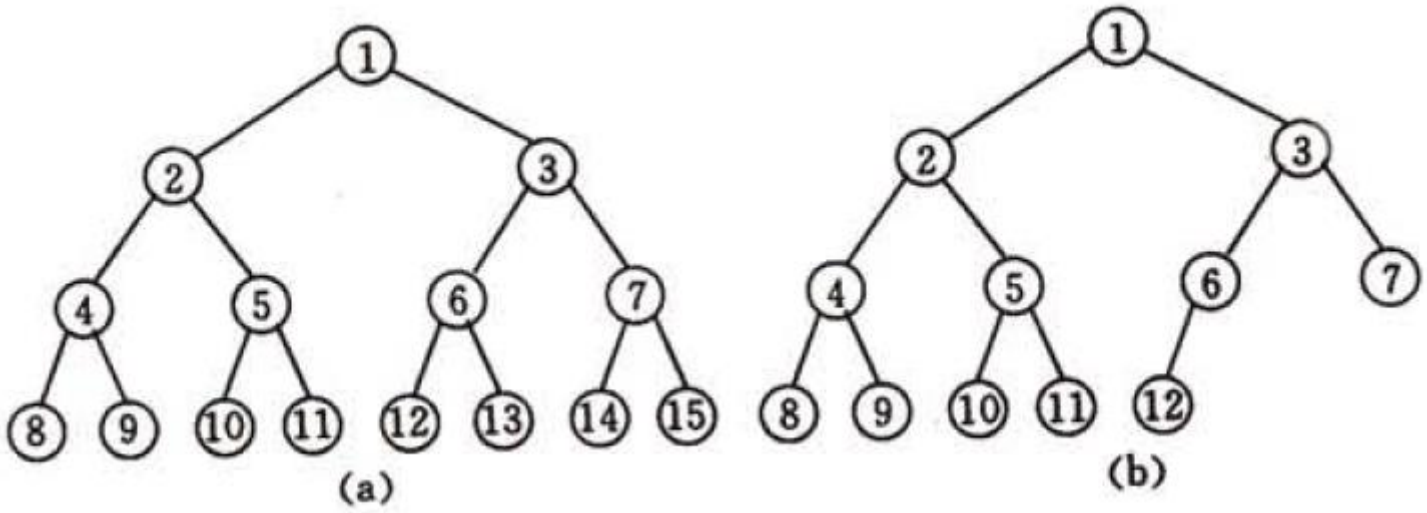


(a)

完全二叉树

- **定义2 完全二叉树(Complete Binary Tree)**

  深度为k,有n个结点的二叉树，当且仅当其每一个结点都与深度为k的满二叉树中编号从1到n的结点一一对应时，称之为完全二叉树。



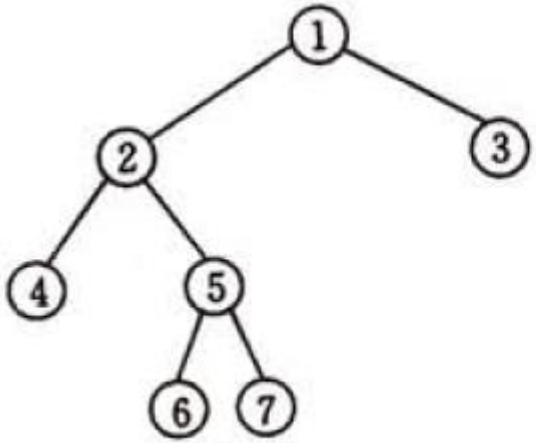
■ 完全二叉树的一些性质：

a）若设二叉树的深度为h，除第h层外，其它各层(1~h-1)的结点数都达到最大个数，第h层从右向左连续缺若干结点，这就是完全二叉树。

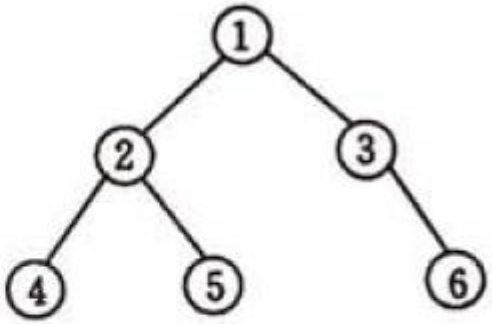
b）叶子结点均位于最下面二层。

c）任一结点，若其右子树的最大层数是L，其左子树的最大层数是L或L+1。

例：　(c)、(d)均不是完全二叉树



(c)



(d)

■性质4　具有n个结点的完全二叉树的深度为

$\lfloor log_2^n \rfloor$ +1

■证明：

设完全二叉树的深度为k，则有

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

⇒ $2^{k-1} \leq n < 2^k$

⇒ 取对数 $k-1 \leq log_2^n < k$

⇒ 因为k为整数，所以 $k = \lfloor log_2^n \rfloor + 1$

■性质5

如果将一棵有n个结点的完全二叉树的结点按层序(自顶向下，同一层自左向右）连续编号1，2，…，n，然后按此结点编号将树中各结点顺序地存放于一个一维数组中，并简称编号为i的结点为结点i (1 ≤ i ≤ n)。则有以下关系：

① 若i = 1，则 i 是二叉树的根，无双亲；若i > 1，则 i 的双亲为⌊i /2⌋。

② 若2*i ≤ n，则 i 的左孩子为2*i，否则无左孩子； 若2*i+1 ≤ n，则 i 的右孩子为2*i+1，否则无右孩子。

③ 若 i 为偶数，且i != n，则其右兄弟为i+1；若 i 为奇数，且i != 1，则其左兄弟为i-1。

④ i 所在层次为 $\lfloor log_2^i \rfloor$ +1 。

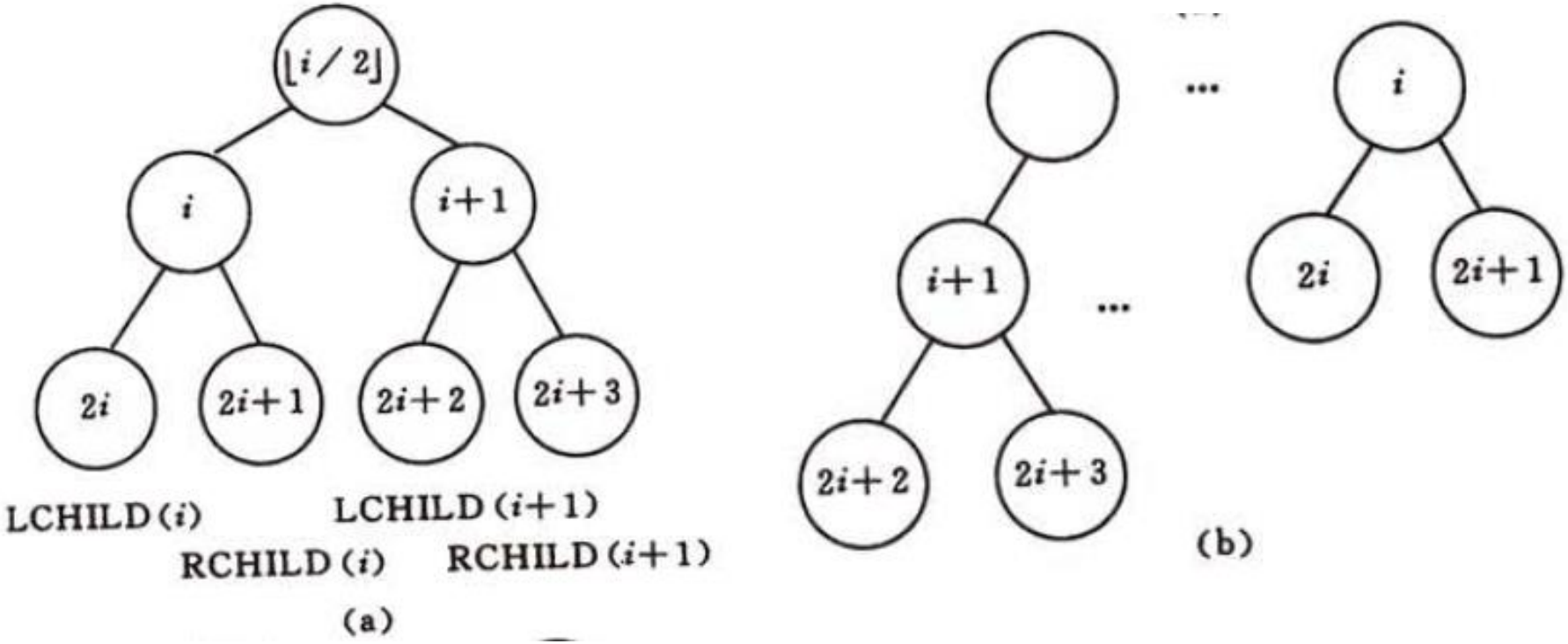


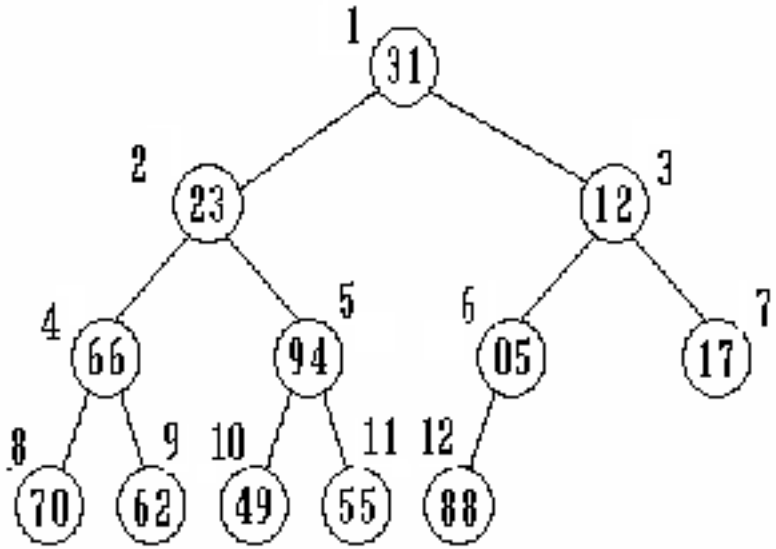
图 6.5　完全二叉树中结点 i 和 $i+1$ 的左、右孩子

(a) 结点 i 和 $i+1$ 在同一层上；

(b) 结点 i 和 $i+1$ 不在同一层上

6.2.3二叉树的存储结构

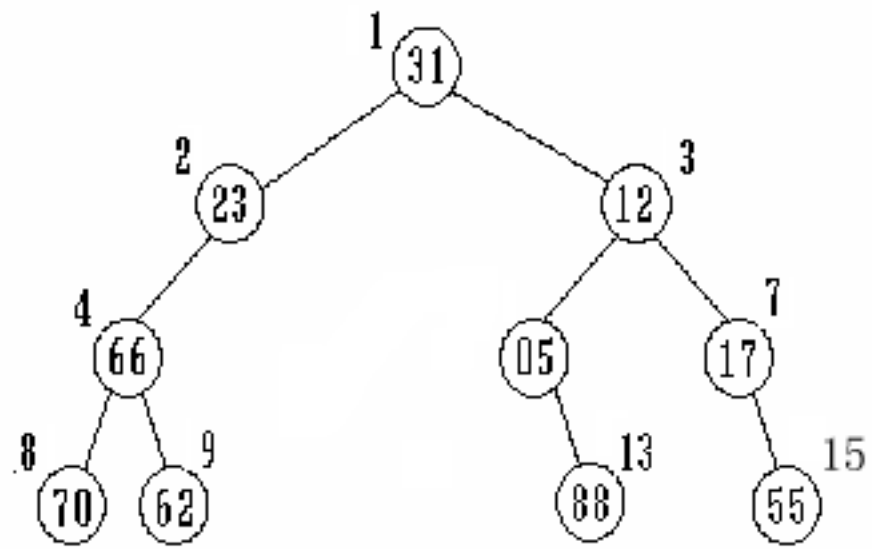
1.顺序存储结构



(a)

1	2	3	4	5	6	7	8	9	10	11	12
31	23	12	66	94	05	17	70	62	49	55	88

(b)



(a)

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
31	23	12	66		05	17	70	62				88		55

(b)

完全二叉树的数组表示　一般二叉树的数组表示

- **二叉树的类型定义（数组表示）：**

```
#define MAX_TREE_SIZE  100
typedef TElemType SqBiTree[MAX_TREE_SIZE];
SqBiTree  bt;
```

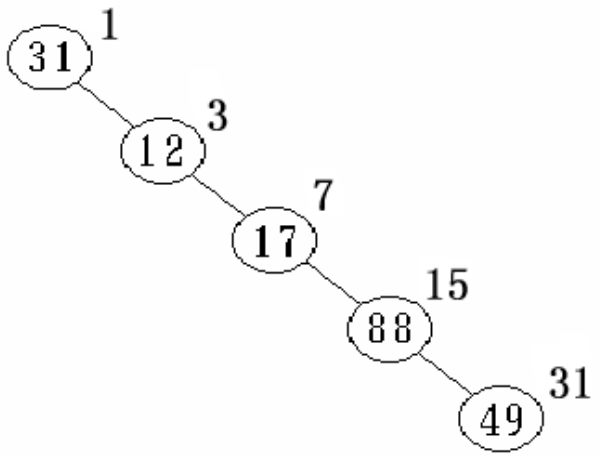
- **任意二叉树对照完全二叉树那样存储**

  根据完全二叉树的特性，结点在向量中的相对位置蕴含着结点之间的关系。

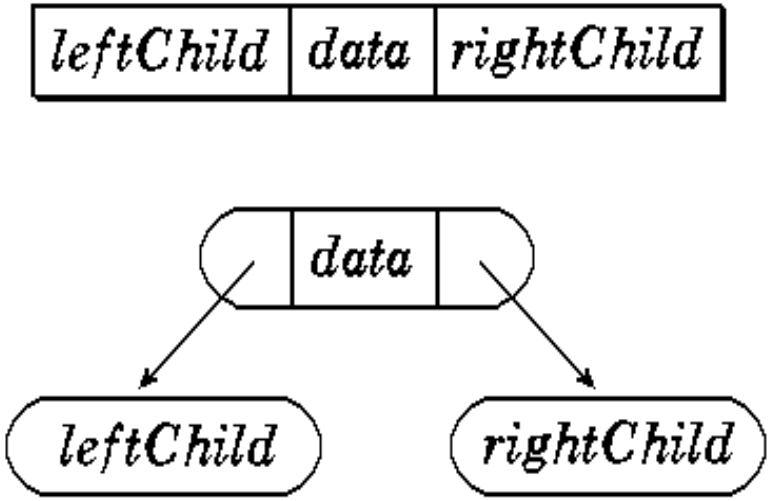
- **缺点：**

  会造成存储空间的浪费

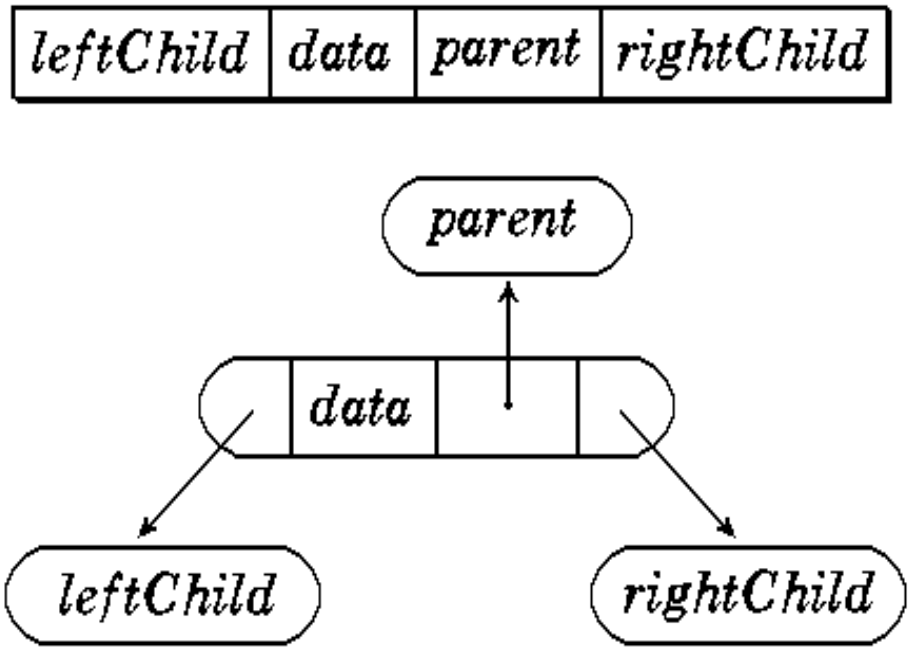
  右图单支树就是一个极端情况



2.链式存储结构



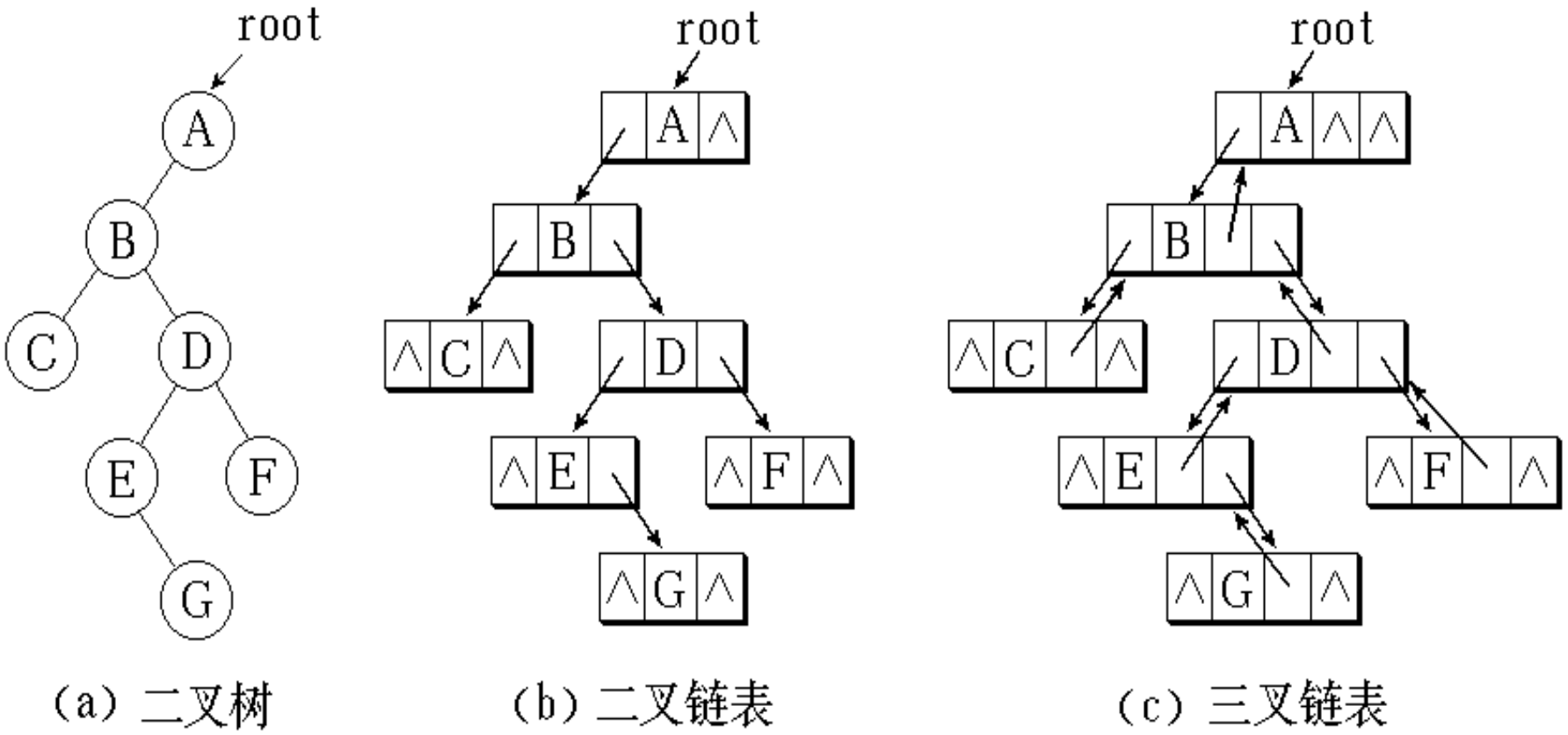
(a) 二叉链表



(b) 三叉链表

- 类型定义:

```
typedef struct BiTNode{   //二叉链表的定义
   TElemType    data;
   Struct BiTNode *lchild,*rchild;
}BiTNode, *BiTree;
```



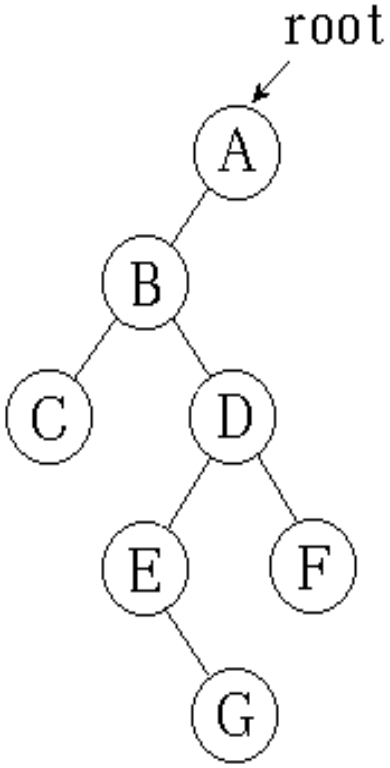
二叉树链表表示的示例

例如：用c++定义的二叉链表类

```
class BiTNode{
public:
        TElemType    data;
        BiTNode *lchild, *rchild;
        BiTNode(TElemType data0){     data=data0;
                lchild=rchild=NULL;          }
        int isleaf(){ return lchild==NULL &&
    rchild==NULL;}
        void addChild(BiTNode *child,int isleft){
                (isleft? lchild: rchild) =child;
    }
}
```

3.静态二叉链表和静态三叉链表

	data	*parent*	*leftChild*	*rightChild*
0	A	−1	1	−1
1	B	0	2	3
2	C	1	−1	−1
3	D	1	4	5
4	E	3	−1	6
5	F	3	−1	−1
6	G	4	−1	−1



- 静态链表的类型定义:

```
typedef struct {
    TElemType    data;
    int parent, leftChild, rightChild;
} Node;      #define N 100
typedef  Node  Bitree[N];
Bitree btree;  //声明一个静态链表变量
```

6.3 遍历二叉树

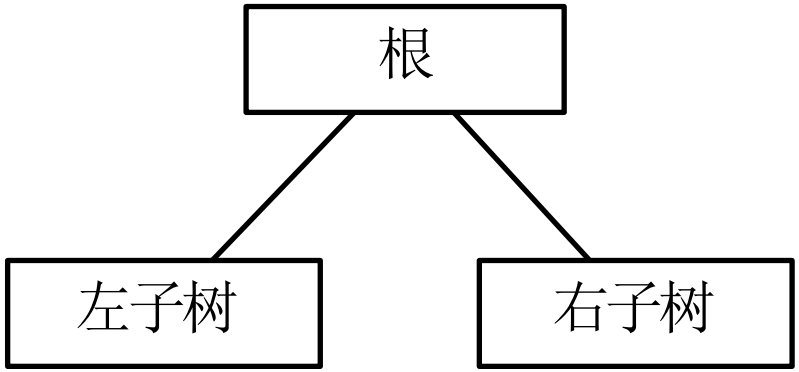
- 树的遍历，就是按某种次序访问树中的结点，要求每个结点访问一次且仅访问一次。
- 遍历的结果：产生一个关于结点的线性序列。

设 访问根结点记作 D

遍历根的左子树记作 L

遍历根的右子树记作 R

则可能的遍历次序有



先序	DLR	DRL	逆先序
中序	LDR	RDL	逆中序
后序	LRD	RLD	逆后序

先序遍历 (Preorder Traversal)

- **先序遍历二叉树算法**

  若二叉树为空，则空操作；否则

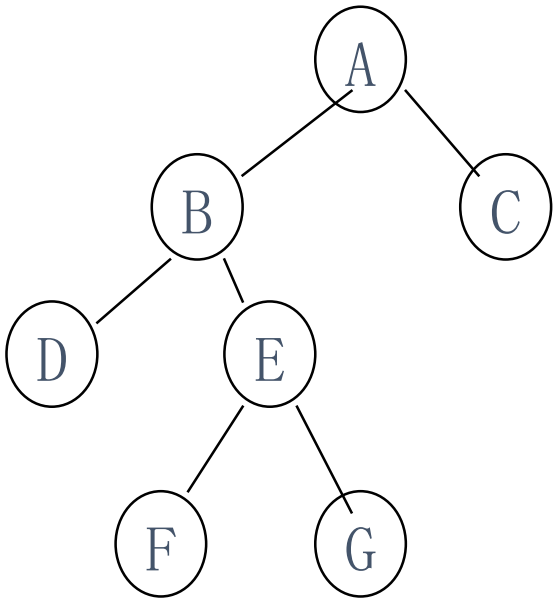
  - 访问根结点 (D);
  - 先序遍历左子树 (L);
  - 先序遍历右子树 (R)。

- **先序遍历的递归函数**

```
void PreOrder(BiTree T){
  if (T){ //T不为空
    printf("%c",T->data);//访问根结点
    PreOrder(T->lchild);//先序遍历左子树
    PreOrder(T->rchild);//先序遍历右子树
  }
}
```

■ 先序遍历二叉树的遍历过程：

1. 访问根A
2. 先序遍历A的左子树{B}
   - 2.1访问根B
   - 2.2先序遍历B的左子树{D}
     - 2.2.1 访问根D
     - 2.2.2 先序遍历D的左子树∅
     - 2.2.3 先序遍历D的右子树∅
   - 2.3先序遍历B的右子树{E}
     - 2.3.1访问根E
     - 2.3.2先序遍历E的左子树{F}
       - 访问根F；先序遍历F的左子树∅；先序遍历F的右子树∅；
     - 2.3.2先序遍历E的右子树{G}
       - 访问根G；先序遍历G的左子树∅；先序遍历G的右子树∅
3. 先序遍历A的右子树{C}
   - 3.1访问根C
   - 3.2先序遍历C的左子树∅
   - 3.3先序遍历C的右子树∅



遍历结果：
ABDEFGC

- 先序遍历二叉树的递归算法(模板)

```
Status PreOrder(BiTree T,Status
(*Visit)(TElemType e)){
 if (T){
  if (Visit(T->data))
   if(PreOrder(T->lchild,Visit))
    if(PreOrder(T->rchild,Visit))
        return OK;
  return ERROR;
 }else return OK;}
```

- 访问函数

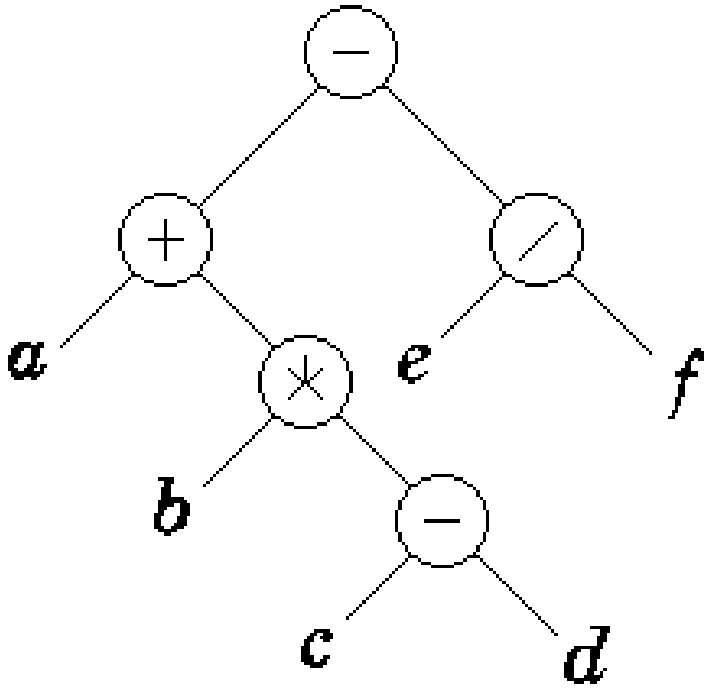
```
Status Print(TElemType e){
  printf("%c",e); return OK;
}
```

- 调用: **BiTree t; PreOrder (t,Print);**

先序遍历结果（前缀表达式，波兰式）

-+a*b-cd/ef

在**波兰式**中，自左向右依次扫描：连续出现2个操作数时，将其前面的运算符退出，对该两操作数进行这两个操作数前面的运算符的运算，运算的中间结果进栈，然后再进行上述的操作。



表达式语法树

中序遍历 (Inorder Traversal)

- **中序遍历二叉树算法**

  若二叉树为空，则空操作；

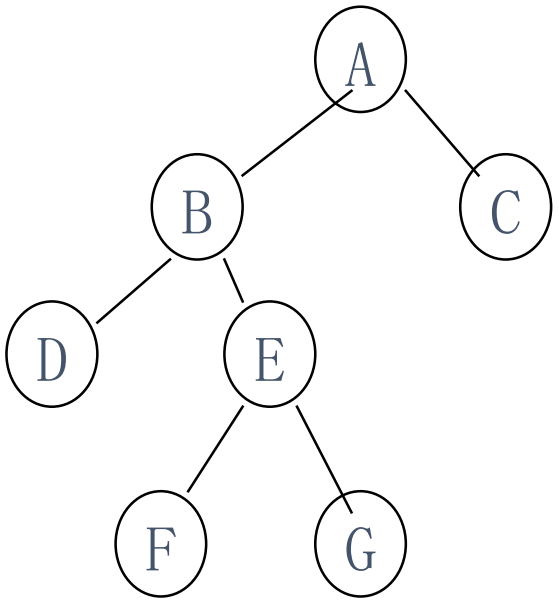
  否则

  - 中序遍历左子树 (L)；
  - 访问根结点 (D)；
  - 中序遍历右子树 (R)。

- **中序遍历的递归函数**

```
void InOrder(BiTree T){
  if (T){ //T不为空
    InOrder(T->lchild);//中序遍历左子树
    printf("%c",T->data);//访问根结点
    InOrder(T->rchild);//中序遍历右子树
  }}
```

■ 中序遍历二叉树的遍历过程：



1. 中序遍历A的左子树{B}
   - 2.1中序遍历B的左子树{D}
     - 2.1.1 中序遍历D的左子树Ø
     - 2.1.2 访问根D
     - 2.1.3 中序遍历D的右子树Ø
   - 2.2访问根B
   - 2.3中序遍历B的右子树{E}
     - 2.3.1中序遍历E的左子树{F}
       - 中序遍历F的左子树Ø；访问根F；中序遍历F的右子树Ø；
     - 2.3.2访问根E
     - 2.3.3中序遍历E的右子树{G}
       - 中序遍历G的左子树Ø；访问根G；中序遍历G的右子树Ø
2. 访问根A
3. 中序遍历A的右子树{C}
   - 3.1中序遍历C的左子树Ø
   - 3.2访问根C
   - 3.3中序遍历C的右子树Ø

遍历结果：
DBFEGAC

- 中序遍历二叉树的递归算法(模板)

```
Status InOrder(BiTree T,Status
(*Visit)(TElemType e)){
 if (T){
   if (InOrder(T->lchild,Visit) )
    if(Visit(T->data))
      if(InOrder(T->rchild,Visit)
           return OK;
   return ERROR;
 }else return OK;}
```

- 访问函数

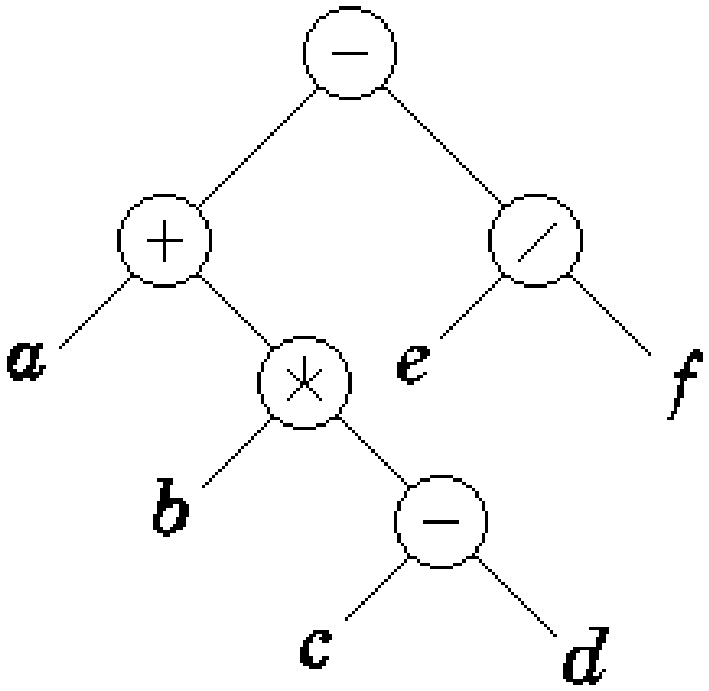
```
Status Print(TElemType e){
   printf("%c",e); return OK;
}
```

- 调用: **BiTree t; InOrder (t,Print);**

中序遍历结果（中缀表达式）

a + b * c - d - e / f

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



表达式语法树

后序遍历 (Postorder Traversal)

- **后序遍历二叉树算法**

  若二叉树为空，则空操作；

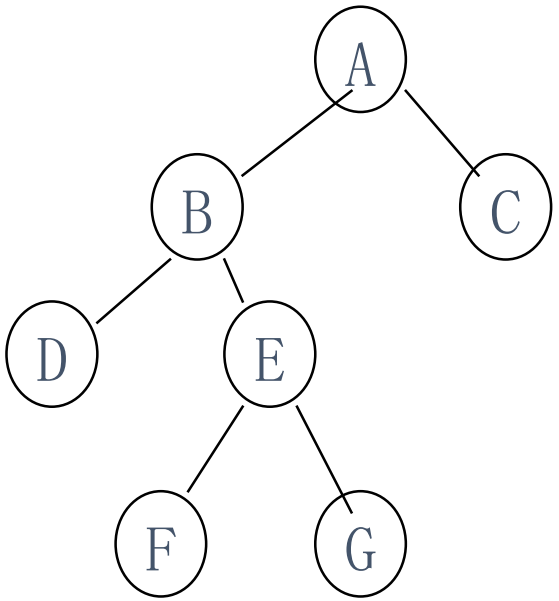
  否则

  - 后序遍历左子树 (L)；
  - 后序遍历右子树 (R)；
  - 访问根结点 (D)；

- **中序遍历的递归函数**

```
void PostOrder(BiTree T){
  if (T){ //T不为空
    PostOrder(T->lchild);//后序遍历左子树
    PostOrder(T->rchild);//后序遍历右子树
    printf("%c",T->data);//访问根结点
    }}
```

■后序遍历二叉树的遍历过程：



1.后序遍历A的左子树{B}

- 2.1后序遍历B的左子树{D}
  - 2.1.1 后序遍历D的左子树∅
  - 2.1.2 后序遍历D的右子树∅
  - 2.1.3 访问根D
- 2.2后序遍历B的右子树{E}
  - 2.3.1后序遍历E的左子树{F}
    - 后序遍历F的左子树∅；后序遍历F的右子树∅；访问根F；
  - 2.3.2后序遍历E的右子树{G}
    - 后序遍历G的左子树∅；先序遍历G的右子树∅；访问根G；
  - 2.3.3访问根E
- 2.3访问根B

2.后序遍历A的右子树{C}

- 3.1后序遍历C的左子树∅
- 3.2后序遍历C的右子树∅
- 3.3访问根C

3.访问根A

遍历结果：
DFGEBCA

- 后序遍历二叉树的递归算法(模板)

```
Status PostOrder(BiTree T,Status
(*Visit)(TElemType e)){
 if (T){
   if (PostOrder(T->lchild,Visit) )
    if(PostOrder(T->rchild,Visit))
      if(Visit(T->data))
           return OK;
   return ERROR;
 }else return OK;}
```

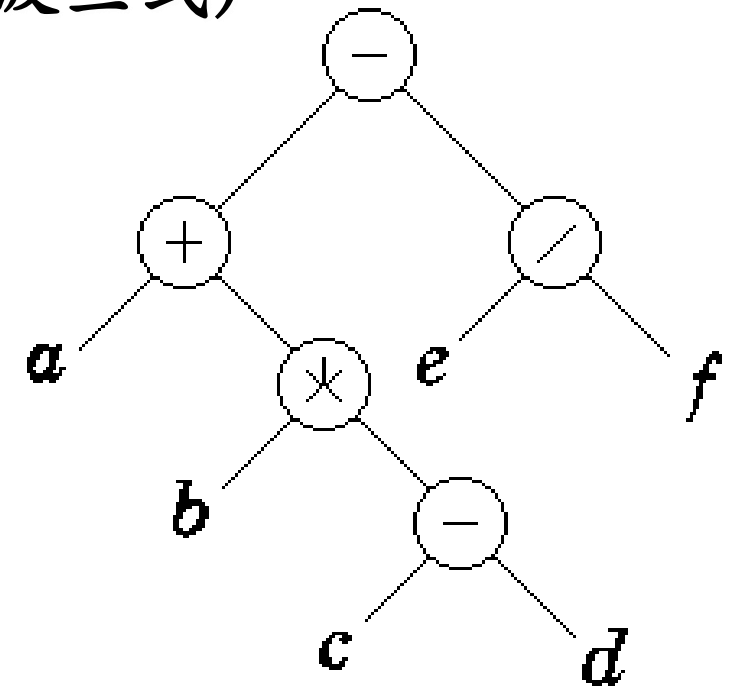
- 访问函数

```
Status Print(TElemType e){
   printf("%c",e); return OK;
}
```

- 调用: **BiTree t; PostOrder (t,Print);**

遍历结果（后缀表达式，逆波兰式）

a b c d - * + e f / -



在逆波兰式中，自左到右依次扫描：是操作数，则依次进栈；遇到运算符。则退出两个操作数，对该两操作数进行该运算符的运算，运算的中间结果进栈；然后再继续重复上述的操作。

重构二叉树

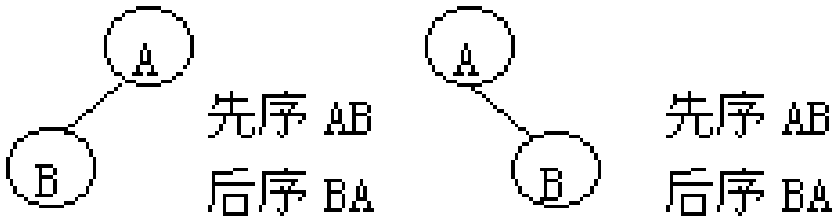
- 根据遍历序列能否构造二叉树?

下列二种情况可以唯一确定一棵二叉树

(1)给出先序遍历结果和中序遍历结果,

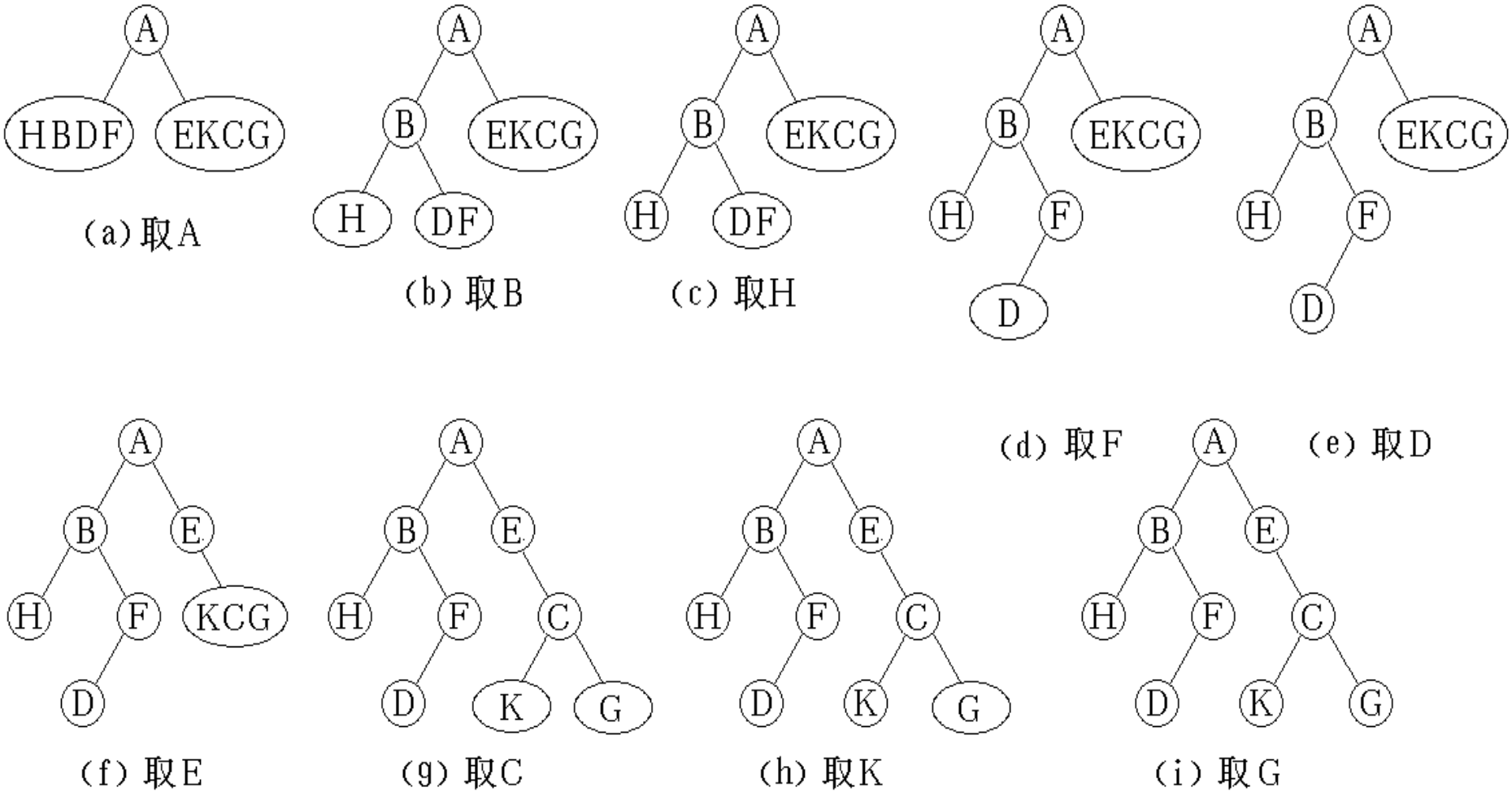
(2)给出后序遍历结果和中序遍历结果;

只凭先序和后序遍历结果无法唯一确定一棵二叉树。



■由二叉树的先序序列和中序序列确定二叉树。

例，先序序列 { ABHFDECKG } 和中序序列 { HBDFAEKCG }，构造二叉树过程如下：



先序+中序构造二叉树的算法

1、从先序遍历序列中取出第一个结点，该结点必为根结点。然后在中序遍历中找出根结点，其前的序列为左子树，其后的序列为右子树

2、对左右子树的先序遍历和中序遍历序列再重复第一步，直到得出所有叶结点为止。

- 算法依据：

  根据先序定义：第一个结点（假定是A）必为根结点。

  根据中序定义：A前的结点为左子树， A后的结点为右子树。

  因此由二叉树的先序序列和中序序列可唯一地确定一棵二叉树。

遍历二叉树的非递归算法

- **递归算法简洁----推荐使用**

  当前的编译系统进行了优化

- **一些特殊情况下用栈模拟递归程序**
  - 理解编译栈的工作原理
  - 理解先中后序遍历的回溯特点
  - 有些应用环境资源限制，不适合用递归

非递归的先序遍历

- **算法1：将右子树根结点入栈**
  - 遇到一个结点，访问该结点，将其右子树根结点入栈，然后遍历其左子树
  - 遍历完左子树，从栈顶弹出一个结点，遍历该子树
  - 遍历每个子树均通过上述两步完成
- **算法2：将根结点入栈**
  - 遇到一个结点，访问该结点，并将其入栈，然后遍历其左子树
  - 遍历完左子树，从栈顶弹出一个结点，遍历其右子树
  - 遍历每个左子树和右子树均通过上述两步完成

非递归的先序遍历算法1

- 要点：栈中预先存一个NULL，存右子树根结

```
void preorder(BiTree T){
   SqStack S;    BiTree P=T;
   InitStack(S);
   Push(S,NULL);//压在栈底作“监视哨”
   while (P){
     printf("%c",P->data);
     if (P->rchild) Push(S,P->rchild);
      if (P->lchild) P=P->lchild;
      else Pop(S,P);
  }
 }
```

非递归的先序遍历算法2

- 要点：遇到每一个结点，都存入栈中

```
void preorder(BiTree T){
   stack<BiTree> S; BiTree P=T;
   do { while(P){
            printf("%c",P->data);
            S.push(P);//入栈
            P=P->lchild;}
        if (!S.empty()){//栈不空
            P=S.top();S.pop();//出栈
            P=P->rchild;}
       }while (!S.empty() || P);
  }
```

非递归的中序遍历

- **算法思想：**
  - 先把根结点入栈，然后遍历其左子树；
  - 遍历完左子树后，从栈顶弹出一个结点，访问之，再遍历其右子树
  - 遍历每个左子树和右子树均通过上述两步完成

中序遍历的非递归代码1

```
void inorder(BiTree T){
   SqStack S;    BiTree P=T;
   InitStack(S);
   do{  while(P){
        *(S.top) = P;  S.top++;//入栈
             P=P->lchild; }
        if (S.top!=S.base){//栈不空
             S.top--;  P=*(S.top);//出栈
             printf("%c",P->data);
             P=P->rchild; }
   }while((S.top!=S.base) || P);
}
```

中序遍历的非递归代码2

- 算法6.3

```
void inorder(BiTree T)
 { SqStack S;    BiTree P=T;
   InitStack(S);
   while( P || !Empty(S))
      { if (P){ Push(S,P);
                P=P->lchild; }
        else  { Pop(S,P);
                printf("%c",P->data);
                P=P->rchild; }
      }
 }
```

非递归的后序遍历

- **算法1：设定一个指针，指向最近访问过的结点。**
  - 遇到结点，先入栈，再遍历左子树；
  - 遍历完左子树后，出栈取出一个结点，判断该结点的右子树是否已遍历完毕：

    1）若该结点的右子树为空，或它的右子树非空，但已遍历完毕，即它的右子树根结点恰好是最近一次访问过的结点时，应该访问该根结点。

    反之，该根结点应重新入栈，先遍历它的右子树。

    2）还可同时设定一个标记，指示该根结点是第一次还是第二次入栈。

后序遍历的非递归算法1

代码1：q保存刚刚访问过的结点

```
void Postorder1(BiTree T)
{  BiTree p=T, q=NULL;
   SqStack S;InitStack(S);  Push(S,p);
   while (!StackEmpty(S))
   { if(p && p!=q) {
        Push(S,p); p=p->lchild; }
     else { Pop(S,p);
        if (p->rchild && p->rchild!=q)
           { Push(S,p); p=p->rchild; }
        else { printf("%c",p->data);
           q=p;}
      } } }
```

后序遍历的非递归算法1

代码2：设标记flag,flag=1表示栈顶元素的左子树访问完毕；flag=0表示开始遍历右子树

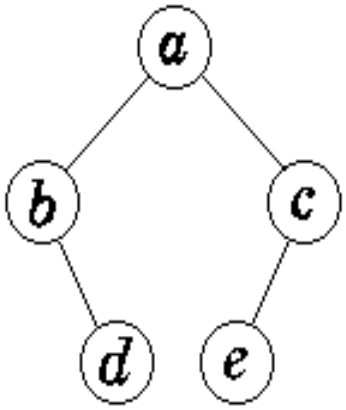
```
void postorder2(BiTree T)
{ BiTree P=T,q; int flag; SqStack S;
 InitStack(S);
  do {  while (P){
          Push(P);P=P->lchild;}
       q=NULL;  flag=1;
       while (!StackEmpty(S) && flag)
       {  GetTop(S,P);
          if (P->rchild == q)
          { printf("%c",P->data);q=p;Pop(S); }
           else { P=P->rchild; flag=0; }      }
   }while (!StackEmpty(S)); }
```

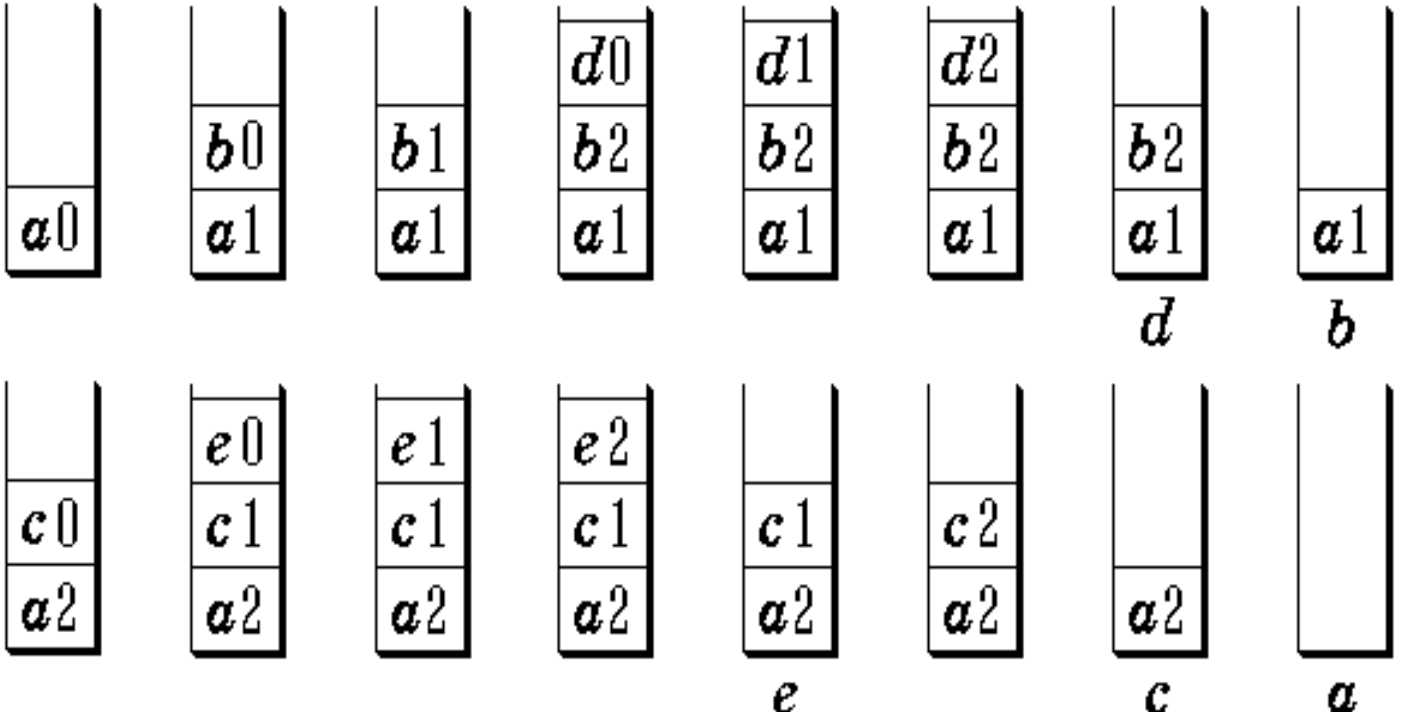
- 上述代码中需用到栈。
- 顺序栈的定义如下：

```
typedef BiTNode* SElemType;
typedef struct{
   SElemType *base;
   SElemType *top;
   int stacksize;
}SqStack;
```

- 注意：栈里存的是二叉链表的结点指针

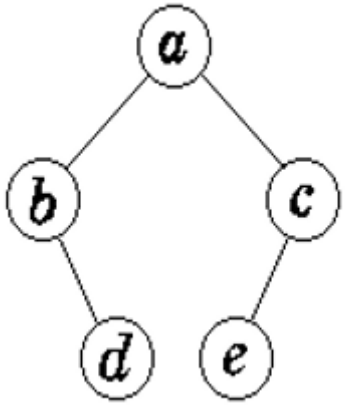
- **算法2：栈中存放每个结点，以及该结点是第几次入栈。**
  - 每遇到一个结点，先把它推入栈中，让PopTim=0。在遍历其左子树前，改结点的PopTim=1，将其左孩子推入栈中。在遍历完左子树后，还不能访问该结点，必须继续遍历右子树，此时改结点的PopTim=2，并把其右孩子推入栈中。在遍历完右子树后，结点才退栈访问。





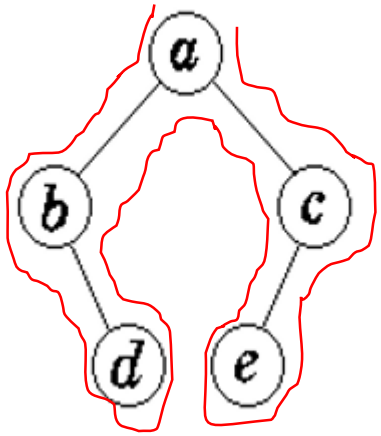
后序遍历的非递归算法2

```
struct SElemType{BiTree p;int tag;}
void postorder(BiTree T){
    SElemType elem; BiTree P=T;
    stack<SElemType> S;
    while(!S.empty() || P){
    while(P){ //
       elem={P,0}; S.push(elem);//第1次入栈
         P=P->lchild; }
      elem=S.top(); S.pop(); P=elem.p;
    if (elem.tag==0){ //
       elem.tag=1;S.push(elem);//第2次入栈
         P=P->rchild;}else{
    printf("%c",P->data); P=NULL;}}}
```



二叉树的非递归遍历模板

```
struct SElemType{BiTree p;int tag;}
void inorder(BiTree T){
SElemType elem; BiTree P=T; stack<SElemType>
S;
while(!S.empty() || P){
  while(P){ /*1调用访问函数，是先序遍历*/
     elem={P,0}; S.push(elem);//第一次入栈
     P=P->lchild; }
  elem=S.top(); S.pop(); P=elem.p;
  if (elem.tag==0){/*2调用访问函数，中序遍历*/
     elem.tag=1;S.push(elem);//第二次入栈
     P=P->rchild;}else {
     /*3调用访问函数，后序遍历*/P=NULL; }}}
```



先序:

a b d c e

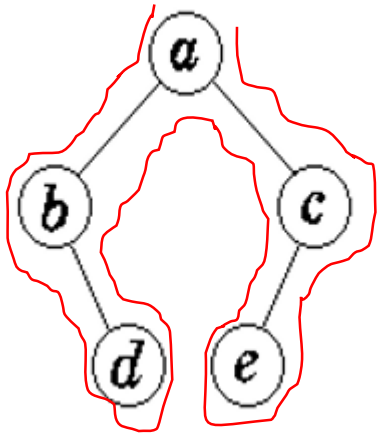
中序:

b d a e c

后序:

d b e c a

a0	b0 a0	b1 a0	d0 b1 a0	d1 b1 a0	b1 a0	a0	a1	c0 a1	e0 c0 a1	e1 c0 a1	c0 a1	c1 a1	a1	



先序:

a b d c e

中序:

b d a e c

后序:

d b e c a

a0	b0 a1	b1 a1	d0 b2 a1	d1 b2 a1	d2 b2 a1	b2 a1	a1	c0 a2	e0 c1 a2	e1 c1 a2	e2 c1 a2	c1 a2	c2 a2	a2	

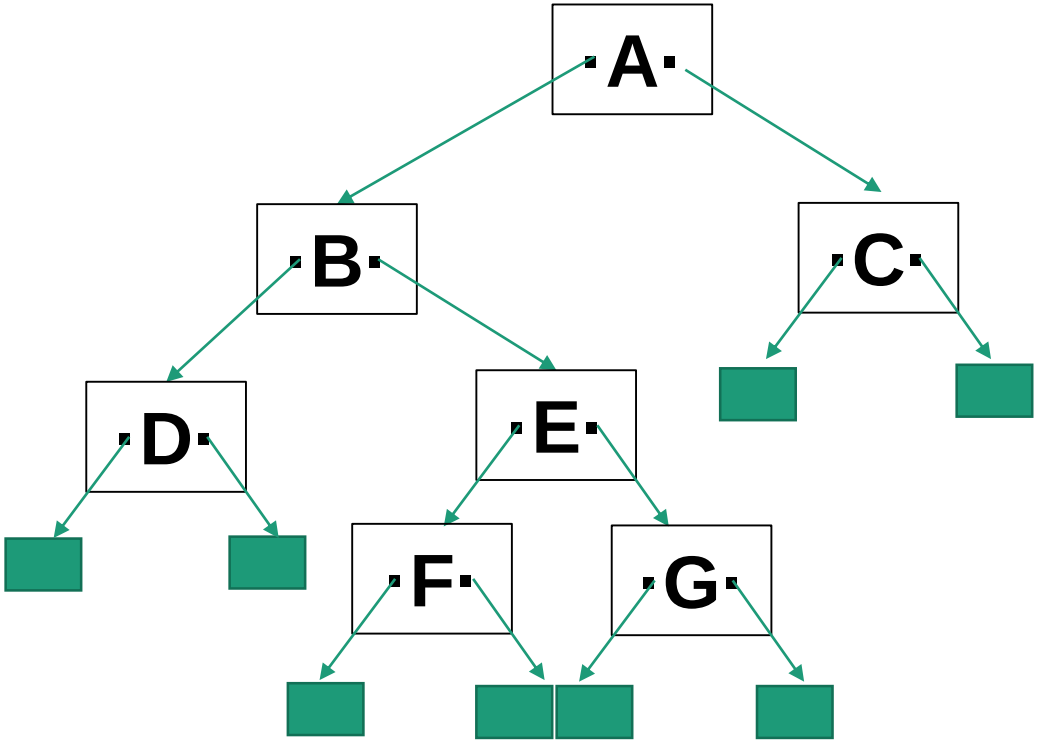
先序建立二叉树的递归算法

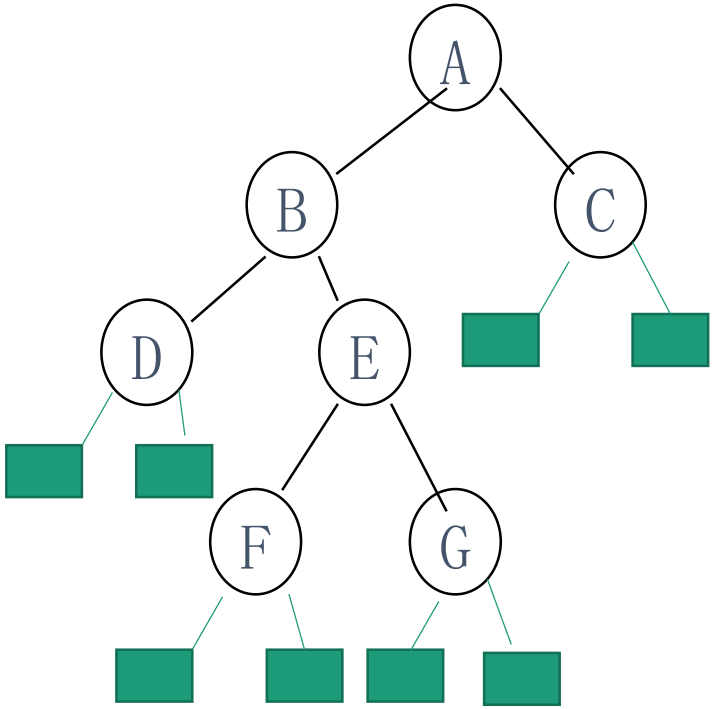
- 算法6.4

```
Status CreateBiTree(BiTree &T)
 {char ch;  scanf("%c",&ch);
  if (ch=='#') T=NULL;
  else{
   if(!(T=(BiTNode*)malloc(sizeof(BiTNode))
)) exit(OVERFLOW);
        T->data = ch;
        CreateBiTree(T->lchild);
        CreateBiTree(T->rchild);
      }
  return OK;
}
```

先序建立二叉树的递归算法(算法6.4)

输入序列：A B D # # E F # # G # # C # #





二叉树的显示输出

```
void PrintBiTree(BiTree T,int n)
{
   int i;  char ch=' ';
   if (T) {
        PrintBiTree(T->rchild,n+1);
        for (i=1;i<=n;++i)
{printf("%5c",ch);}
        printf("%c\n", T->data);
        PrintBiTree(T->lchild,n+1);
   }
}
```

遍历的第一个结点

- **先序：**

  根结点；

- **中序：**

  沿着左链走，找到一个没有左孩子的点；

- **后序：**

  从根结点出发，沿着左链走，找到一个既没有左孩子又没有右孩子的结点。

遍历的最后一个结点

- **中序：**

  从根结点出发，沿着右链走，找到一个没有右孩子的结点；

- **后序：**

  根结点。

- **先序：**

  从根结点出发，沿着右链走，找到一个没有右孩子的结点；如果该结点有左孩子，再沿着其左孩子的右链走，以此类推，直到找到一个没有孩子的结点。

练习：

1）分别写出中序遍历第一个结点和最后一个结点的代码

2）_____序列+_____序列 或_____序列+_____序列均可唯一地确定一棵二叉树

3）对于有n个节点的二叉树，其二叉链表存储结构中，有____个指针域未利用，已经使用的有____个指针域，共有_____个指针域

作业1

1．已知一棵二叉树的中序遍历为DBGEACF，后序遍历为DGEBFCA，请画出该二叉树，并写出其先序遍历序列。

2.一棵有510个结点的完全二叉树的高度为多少？（独根树高度为1）

3.一棵有512个结点的完全二叉树，叶子结点数为多少？

4．已知一棵有m个结点的完全二叉树，叶子结点数为_____。

5.在一棵非空二叉树中，若度为0的结点的个数n，度为2的结点个数为m，则有n=_____

6．写出求二叉树叶子结点个数的递归程序。

7．先序遍历的非递归算法1，栈所需的最大容量是多少？画出该二叉树的形态。假定结点个数为n。

中序遍历的第一个和最后一个结点

- 求中序的第一个结点的算法：

```
P=T;
while (P->lchild)  P=P->lchild;
printf(P->data);
```

- 求中序的最后一个结点的算法：

```
P=T;
while(P->rchild) P=P->rchild;
printf(P->data);
```

2）先序+中序 或中序+后序 均可唯一地确定一棵二叉树

3）对于有n个节点的二叉树，其二叉链表存储结构中，有n+1个指针域未利用，已经使用的有n-1个指针域，共有2n个指针域。

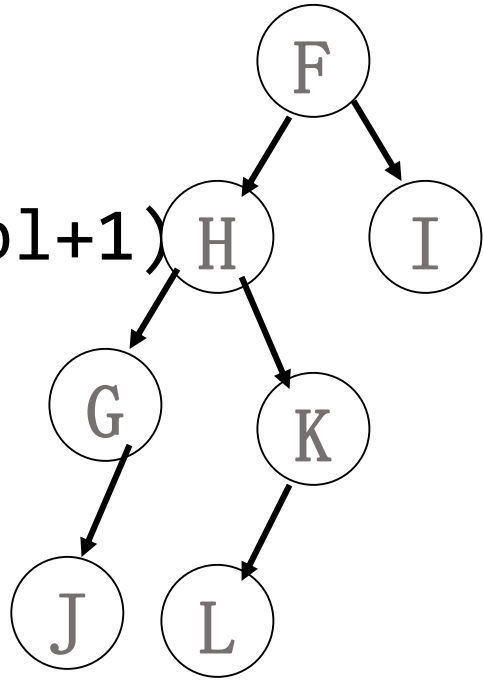
遍历二叉树的应用

- 计算二叉树的深度
- 求二叉树的结点个数
- 左右子树互换
- 复制二叉树

计算二叉树的深度

- 根结点的深度=max{左子树深度，右子树深度}+1
- 利用后序遍历

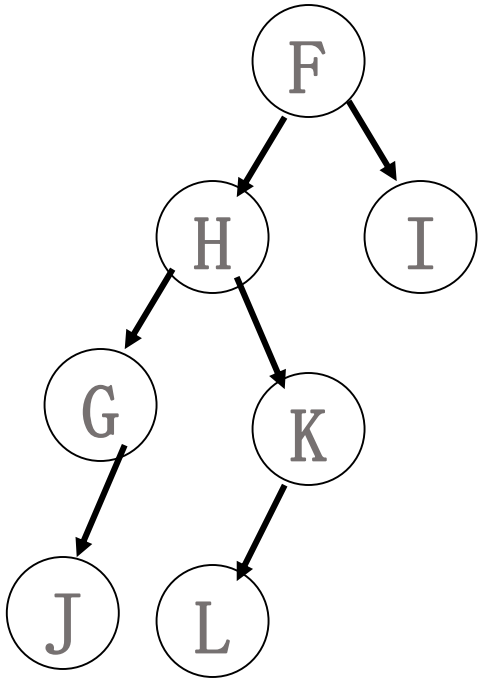
```
int Depth(BiTree T){
   int depl,depr;
   if (T){
    depl=Depth(T->lchild);
    depr=Depth(T->rchild);
    if (depl>=depr) return (depl+1);
    else return (depr+1);
   }
   return 0;
}
```



求二叉树结点个数

- 结点个数=左子树结点个数+右子树结点个数+1
- 后序遍历

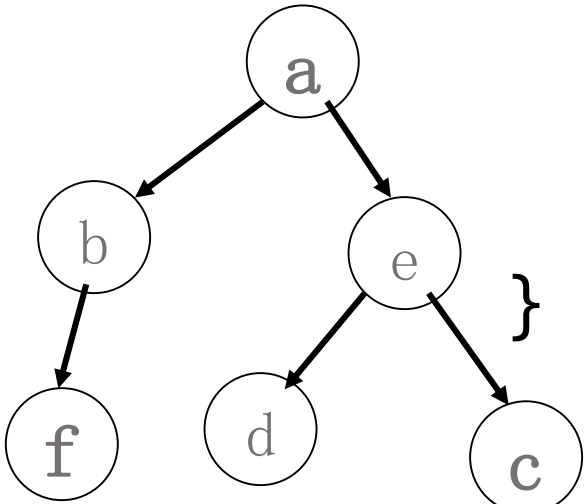
```
int Size(BiTree T)
{
    if (T==NULL) return 0;
    else return 1 +
        Size (T->lchild ) +
        Size ( T->rchild);
}
```

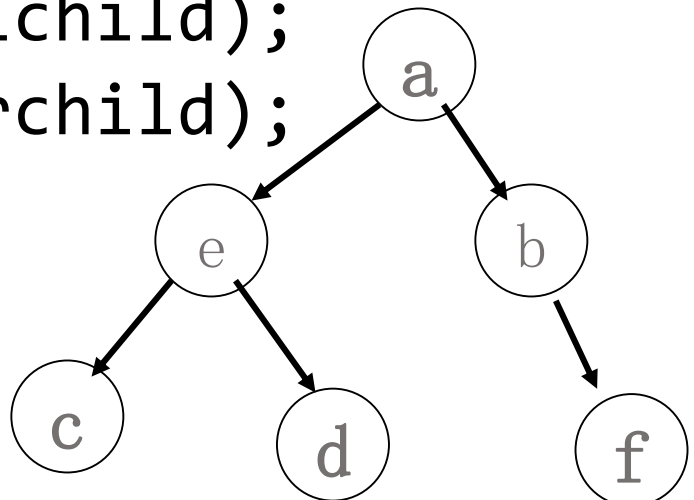


左右子树互换

- 任一结点的左右子树均互换

```
void Exchange(BiTree &T)
{  BiTree S;
   if (T) {
            S=T->lchild;
            T->lchild=T->rchild;
            T->rchild=S;
            Exchange(T->lchild);
            Exchange(T->rchild);
   }
}
```





复制二叉树

```
void CopyTree(BiTree T,BiTree &T1){
  if(T){

T1=(BiTree)malloc(sizeof(BiTNode));
      if (!T1){ printf("Overflow\n");
                exit(1); }
    T1->data=T->data;
    T1->lchild=T1->rchild=NULL;
    CopyTree(T->lchild,T1->lchild);
    CopyTree(T->rchild,T1->rchild);
    }
}
```

层次遍历（宽度遍历）

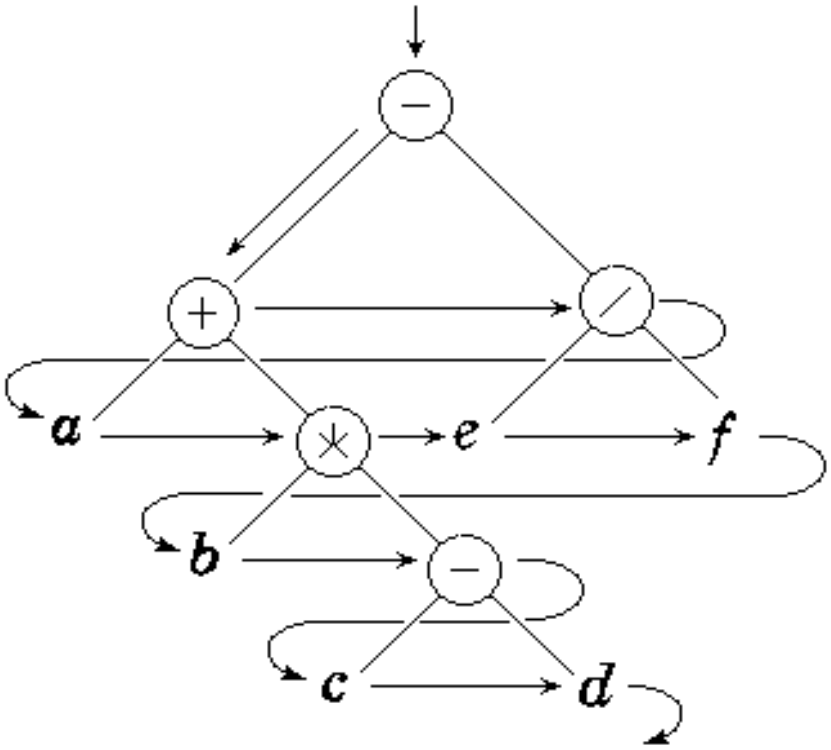
- 从根结点开始逐层访问，用FIFO队列实现。
- 队列的数据结构：

```
typedef BiTNode* QElemType;
typedef struct{
   QElemType  *base;
   int front,rear;
}SqQueue;
```

- 右图遍历顺序：

  从上到下，从左到右

  -+/a*efb-cd



```
//用自定义队列实现
void LevelOrderTraverse(BiTree T){  BiTree p;
SqQueue Q; InitQueue(Q);
     if (T){
        Q.base[Q.rear]=T;
        Q.rear=(Q.rear+1)%MAXQSIZE;//入队
        while (Q.front !=Q.rear){
            p=Q.base[Q.front];
           Q.front=(Q.front+1)%MAXQSIZE;//出队
            printf("%c",p->data);
            if (p->lchild){
                 Q.base[Q.rear]=p->lchild;
                Q.rear=(Q.rear+1)%MAXQSIZE; }
            if (p->rchild){
                 Q.base[Q.rear]=p->rchild;
                 Q.rear=(Q.rear+1)%MAXQSIZE;}
         }/*while*/ }/*if*/   }
```

```
//用c++STL中的queue来实现 #include <queue.h>
void LevelOrderTraverse(BiTree T)//层次遍历
  {  BiTree p;   queue<BiTree> Q;
     if (!T) return;
     Q.push(T);//入队
     while (!Q.empty()){
         p=Q.front();Q.pop();//出队
         printf("%c",p->data);
          if (p->lchild)
               Q.push(p->lchild); //入队
         if (p->rchild)
               Q.push(p->rchild);//入队
      } //while
}
```

6.4 线索化二叉树（穿线树、线索树）(Threaded Binary Tree)

- **线索（Thread）：指向结点前驱和后继的指针**
  - 若结点有左孩子，则lchild指示其左孩子，否则lchild中存储该结点的前驱结点的指针；
  - 若结点有右孩子，则rchild指示其右孩子，否则rchild中存储指向该结点的后继结点的指针
  - 在线索树中的前驱和后继是指按某种次序遍历所得到的序列中的前驱和后继。
- **实质：**

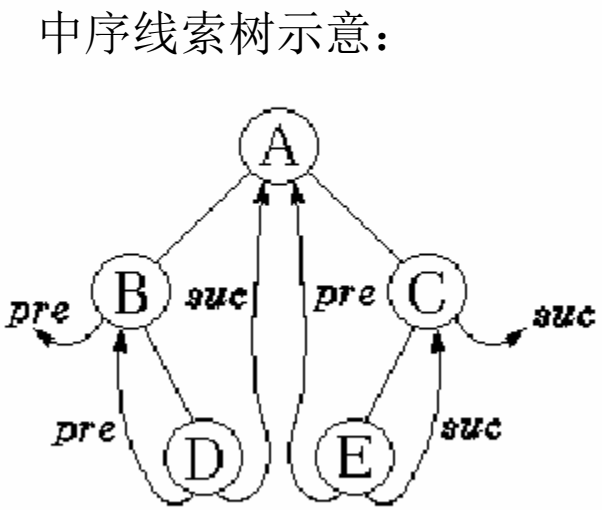
  对一个非线性结构进行线性化操作，使每个结点（除第一和最后一个外）在这些线性序列中有且仅有一个直接前驱和直接后继。
- **概念：**

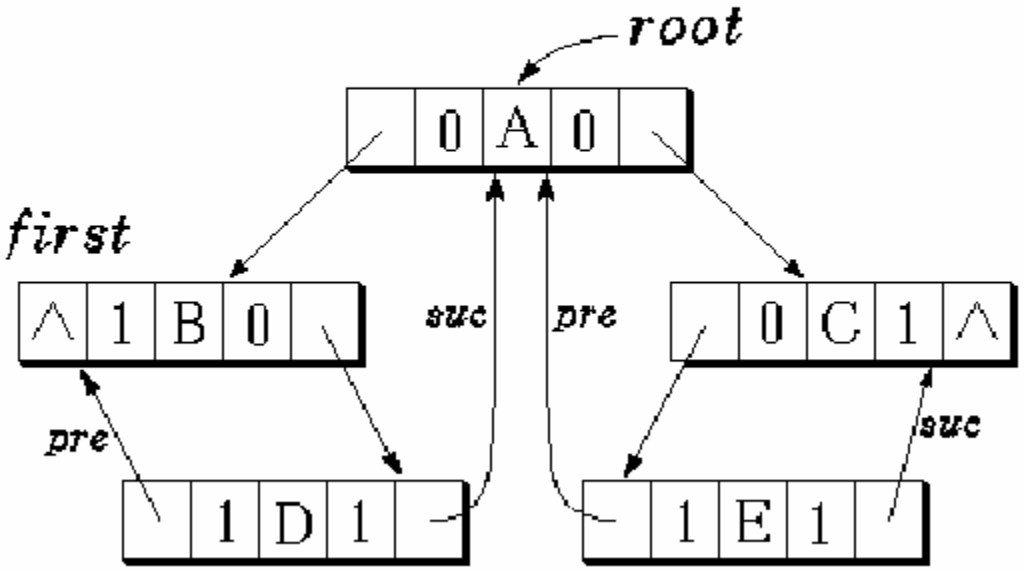
  线索链表、线索、线索化、线索二叉树

线索二叉树及其线索链表的表示

lchild	LTag	data	RTag	rchild

中序线索树示意：





标志域：

ltag = 0, lchild为左孩子指针

ltag = 1, lchild为前驱线索

rtag = 0, rchild为右孩子指针

rtag = 1, rchild为后继指针

线索二叉树的类型定义

- 类型定义:

```
typedef enum{Link,Thread}PointerTag;
//Link==0: 指针，指向孩子结点
//Thread==1: 线索，指向前驱或后继结点
typedef struct BiThrNode{
   TElemType data;
   struct BiThrNode *lchild,*rchild;
   PointerTag LTag,RTag;
}BiThrNode, *BiThrTree;
BiThrTree T;
```

遍历线索二叉树

- **从遍历的第一个结点来看**

  先序序列中第一个结点必为根结点；

  中、后序序列中第一个结点的左孩子定为空

- **从遍历的最后一个结点来看**

  先、中序序列中最后一个结点的右孩子必为空；

  后序序列中最后一个结点一定为根结点

- **线索树的作用**

  对于遍历操作，线索树优于非线索树

  遍历线索树不用设栈

- **步骤**

  1) 找遍历的第一个结点

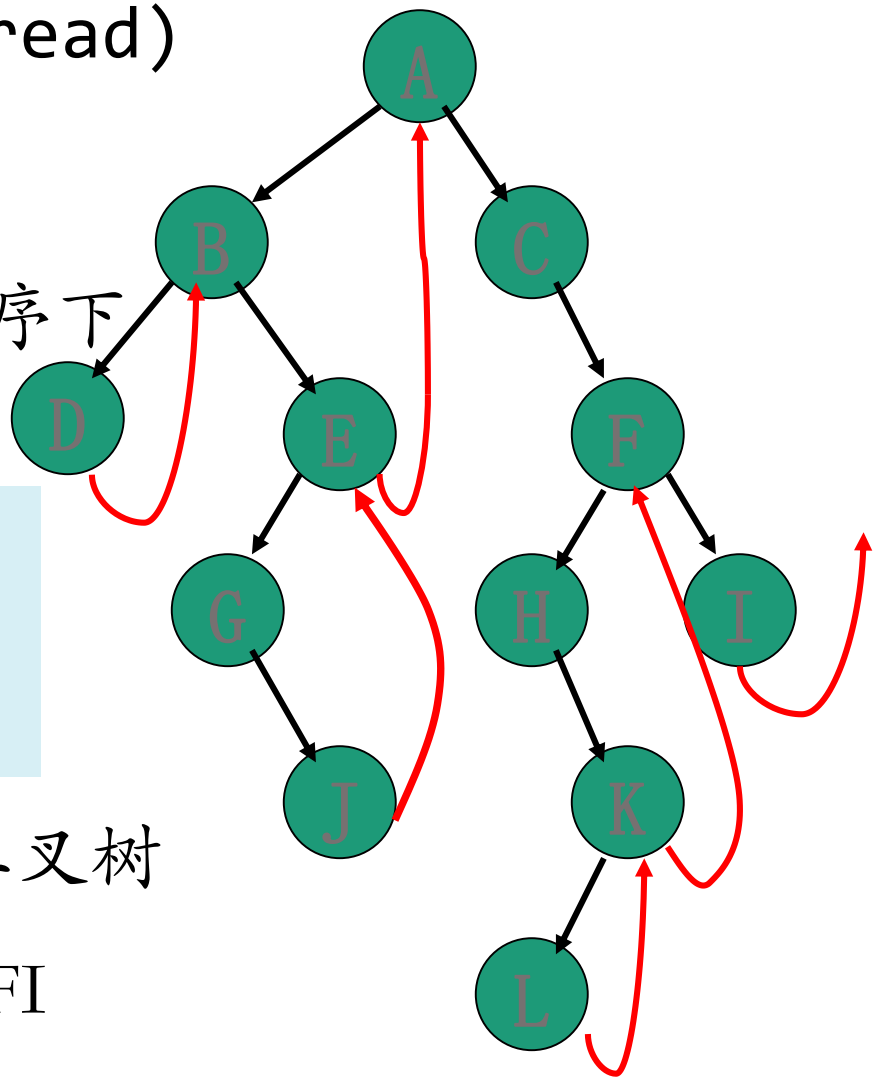
  2) 不断地找遍历到的结点的后继结点，直到树中各结点都遍历到为止，结束。

寻找当前结点在中序下的后继

```
if (current.RTag ==Thread)
后继=current.rchild
else
```

后继=当前结点右子树的中序下的第一个结点

```
p=current->rchild;
while(p->LTag==Link)
        p=p->lchild;
```



中序后继线索二叉树

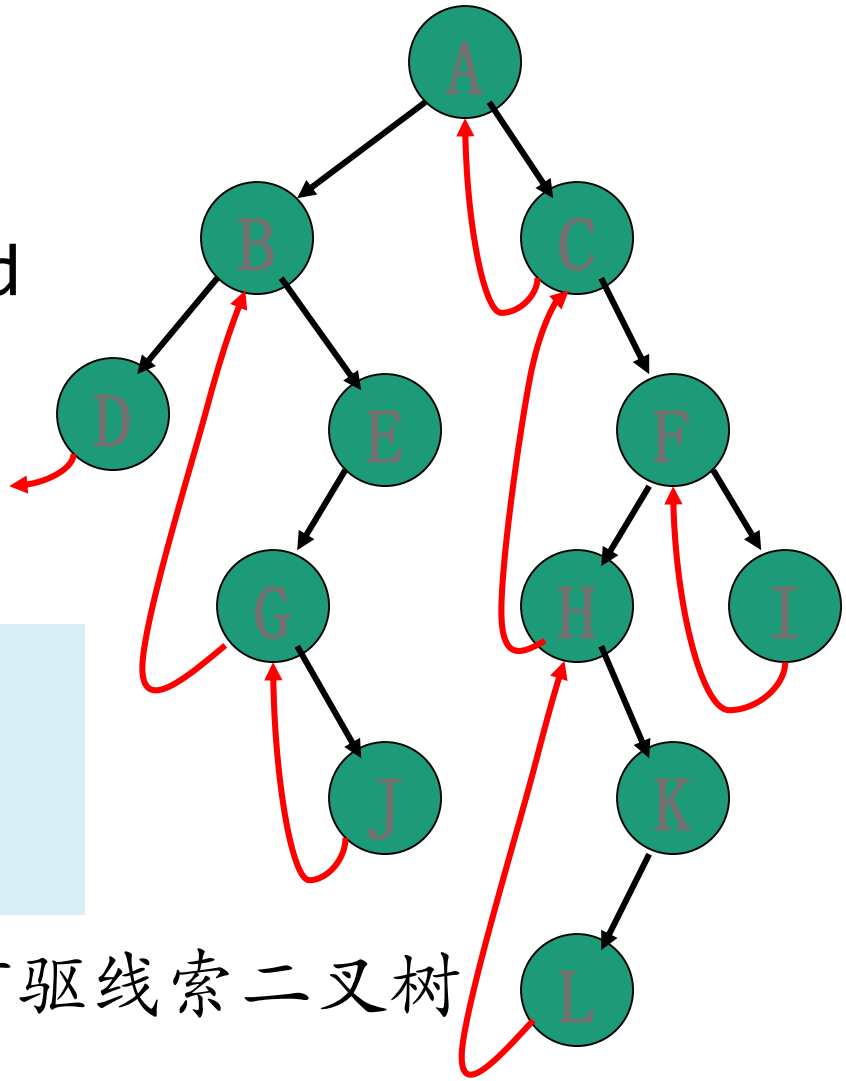
DBGJEACHLKFI

寻找当前结点在中序下的前驱

```
if (current->LTag
==Thread)
前驱=current->lchild
else
```

前驱=当前结点左子树的中序下的最后一个结点

```
p=current->lchild;
while(p->RTag==Link)
        p=p->rchild;
```



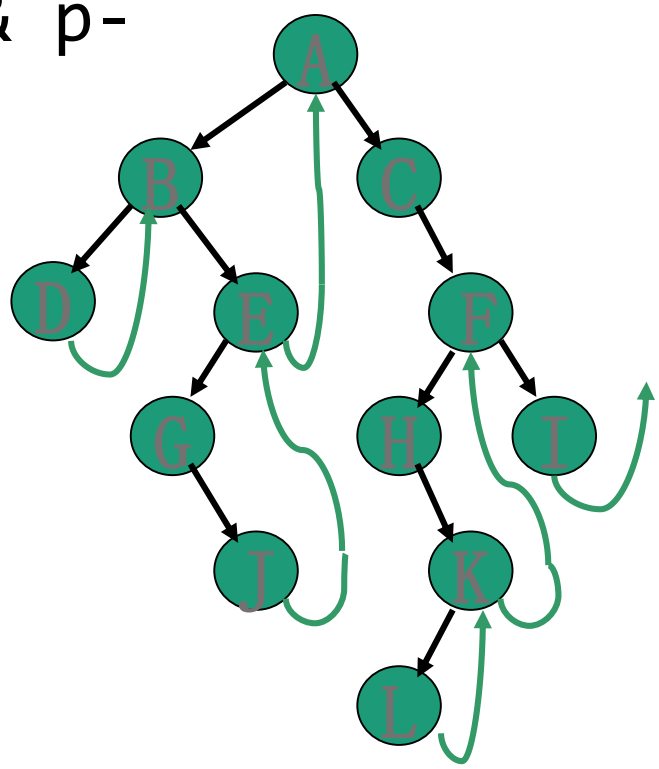
中序前驱线索二叉树

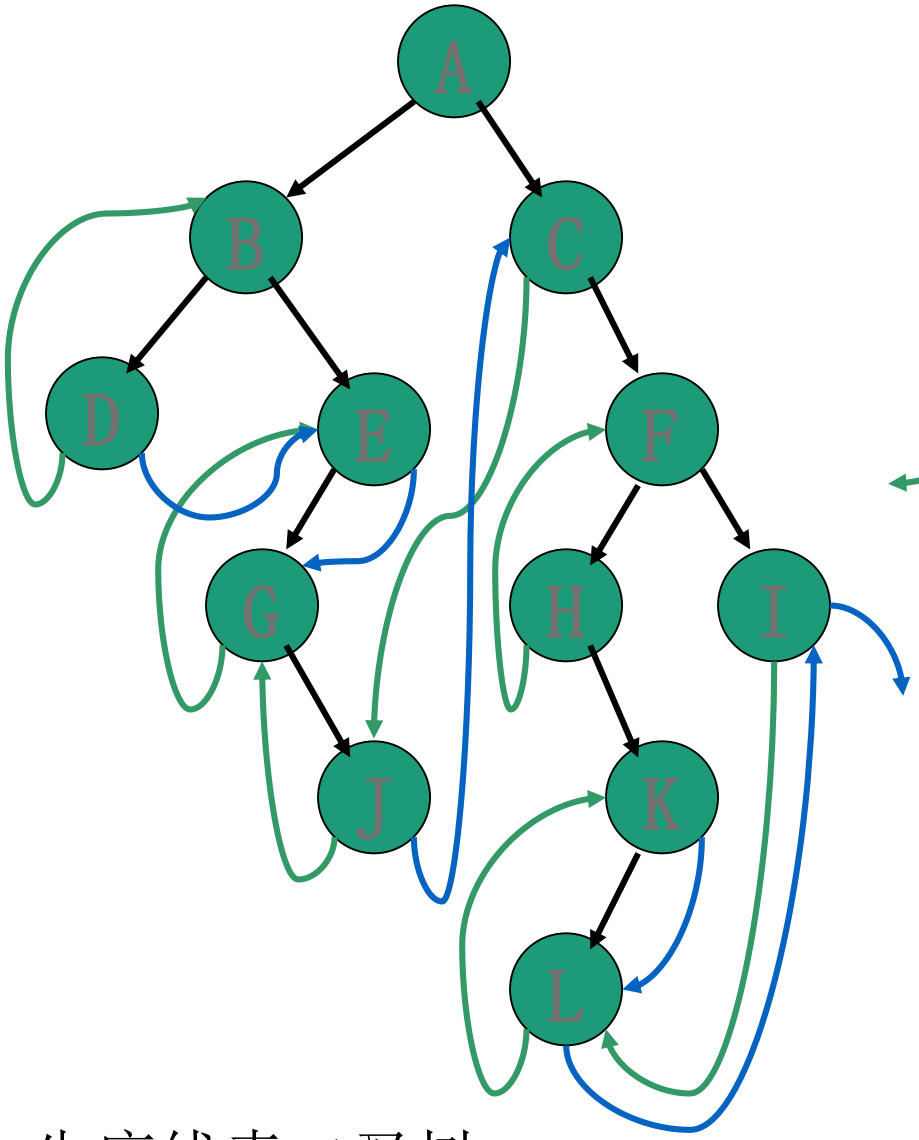
中序序列:
DBGJEACHLKFI

遍历中序线索二叉树(不带头结点)

```
void inorder1_Thr(BiThrTree T){
    BiThrTree p=T;
  while (p->LTag==Link) p=p->lchild;
  printf("%c",p->data);
  while (p->rchild){
       if(p->RTag==Link){//有右子树
           p=p->rchild;
        while(p->LTag==Link) p=p->lchild;
        }else//无右子树
        p=p->rchild;
     printf("%c",p->data);
   }
 }
```

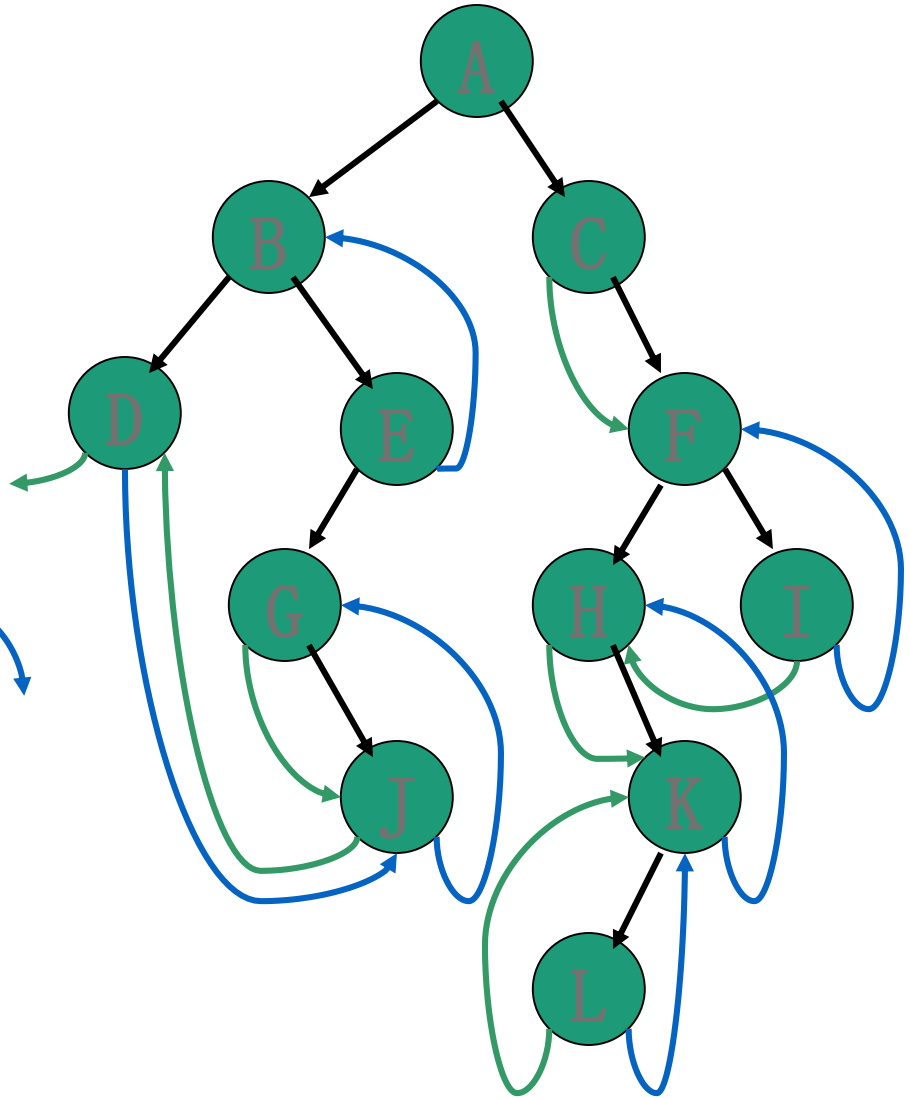
```
void inorder2_Thr(BiThrTree T){
    BiThrTree p=T;
    while (p){//不是中序遍历的最后一个结点
   while(p->LTag==Link) p=p->lchild;
     printf("%c",p->data);
     while(p->RTag==Thread && p-
>rchild){
      p=p->rchild;
        printf("%c",p->data);
        }
       p=p->rchild;
    }
}
```





先序线索二叉树

ABDEGJCFHKLI

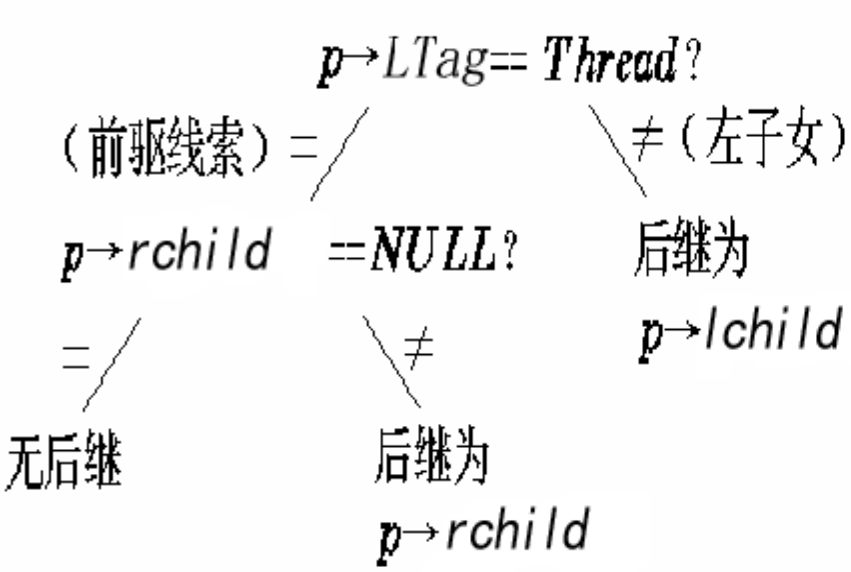


后序线索二叉树

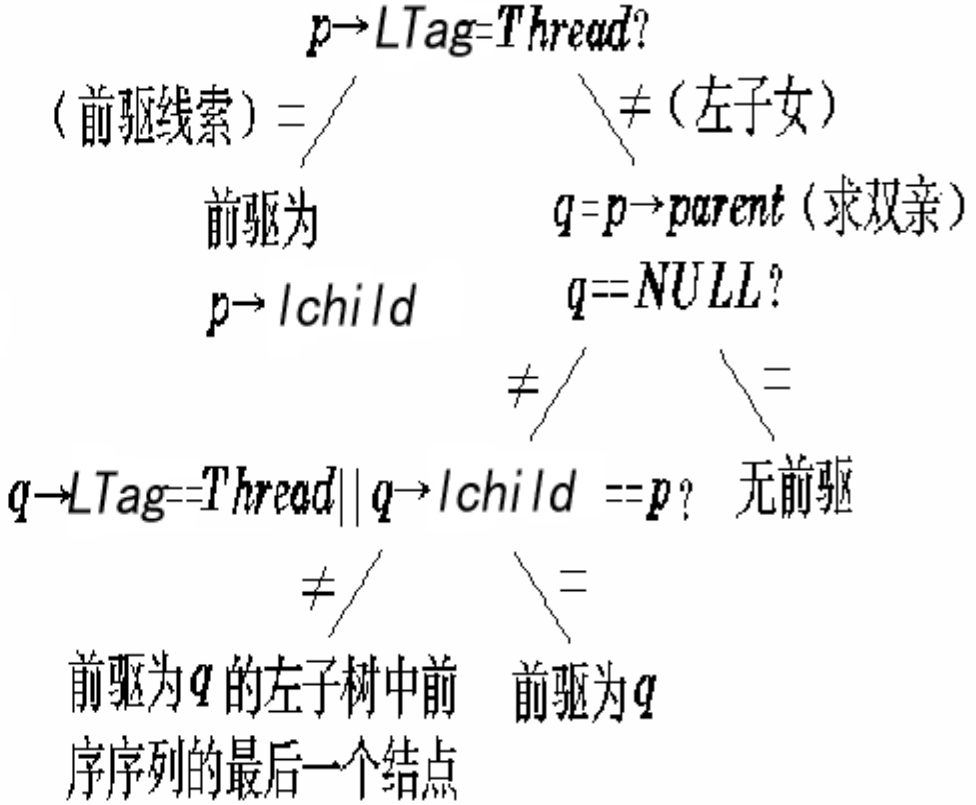
DJGEBLKHIFCA

先序线索二叉树

- 在先序线索二叉树中寻找当前结点的后继与前驱



（a）求结点 p 的后继



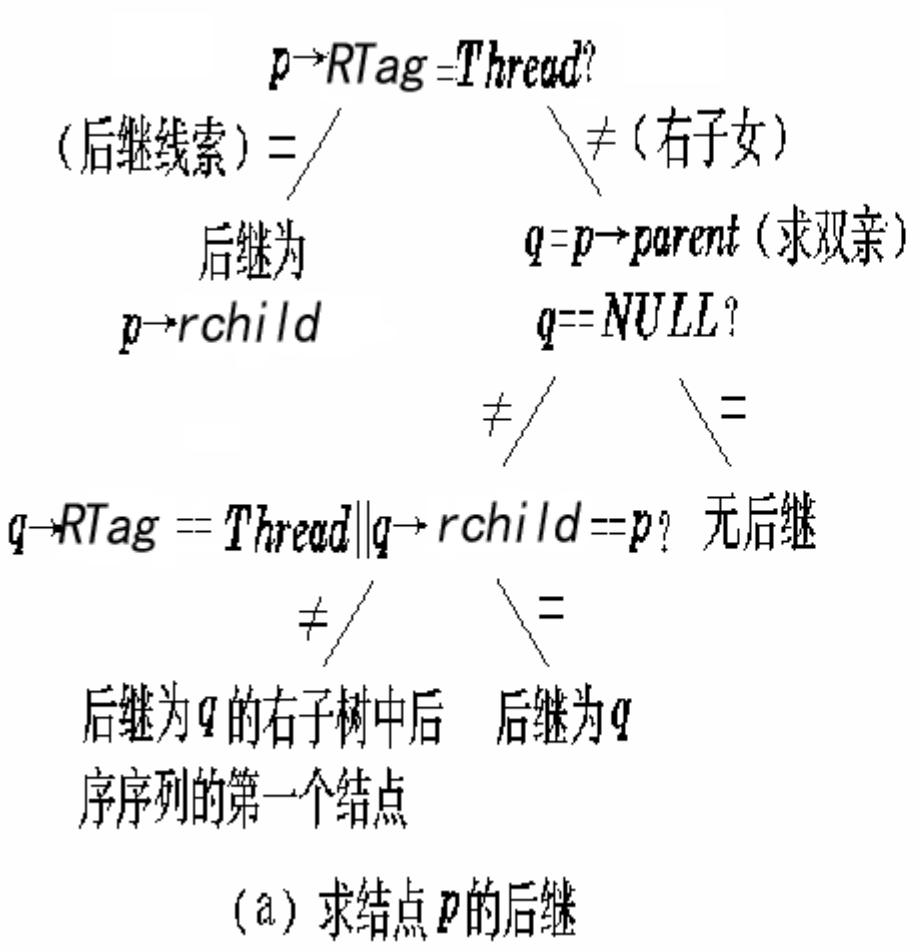
（b）求结点 p 的前驱

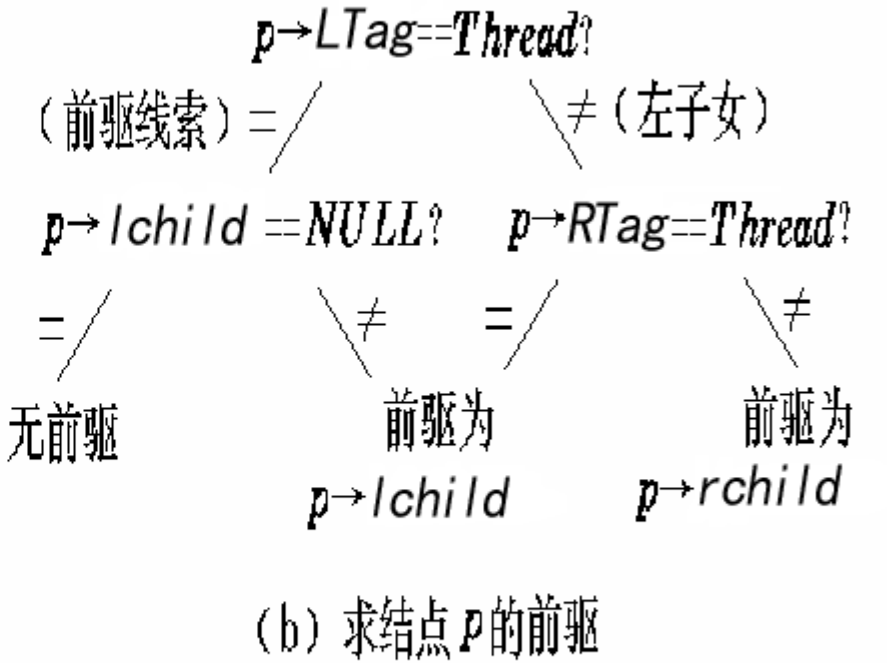
- 遍历先序线索二叉树(不带头结点)

```
void preorder_Thr(BiThrTree T){
      BiThrTree p=T;
      printf("%c",p->data);
      while (p->rchild){
          if (p->LTag==Link)
        p=p->lchild;
       else
        p=p->rchild;
       printf("%c",p->data);
      }
}
```

后序线索二叉树

- 在后序线索化二叉树中寻找当前结点的后继





- 遍历后序线索二叉树(不带头结点)

```
void postorder_Thr(TriThrTree T){
  TriThrTree f,p=T;
   do{
     while (p->LTag==Link)
        p=p->lchild;
        if (p->RTag==Link)
        p=p->rchild;
   } while (p->LTag!=Thread ||p-
RTag!=Thread);
    printf("%c",p->data);
}
```

```
while (p!=T)
  { if (p->RTag==Link)
      { f=p->parent;
    if (f->RTag==Thread || p==f-
>rchild) p=f;
    else{ p=f->rchild;
          do{ while(p->LTag==Link)
p=p->lchild;
       if (p->RTag==Link) p=p->rchild;
            }while (p->LTag!=Thread
|| p->RTag!=Thread);
         }   }
    else p=p->rchild;
    printf("%c",p->data);
  }}
```

二叉树的线索化

- **将未线索过的二叉树给予线索**

  在某种（先、中、后、层序）遍历的过程中，修改空指针的值为指向前驱或后继的线索。

- **中序线索化**

  后继线索化——处理前驱结点

  - a．如果无前驱结点或前驱结点的右指针域为非空，也不必加线索
  - b．如果前驱结点的右指针域为空，则把当前结点的指针值赋给前驱结点的右指针域。

  前驱线索化——处理当前结点

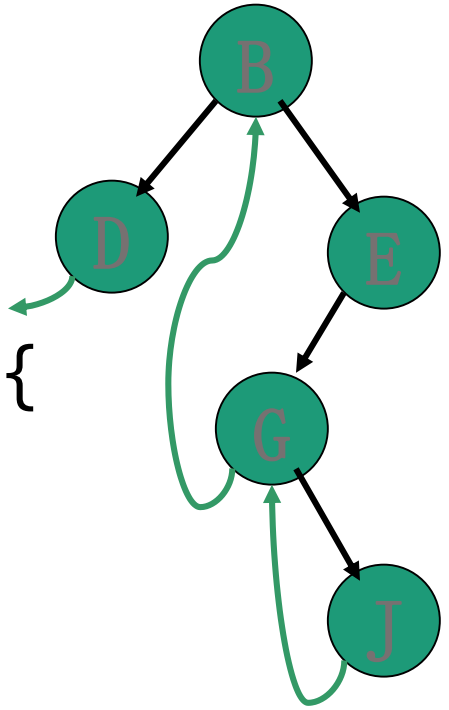
  - 如果当前结点的左指针域为空，则把前驱结点的指针值赋给当前结点的左指针域。

中序线索化

- 中序线索化递归程序

```
void InThreading(BiThrTree P){
  if (P){
    InThreading(P->lchild);
    后继线索化(处理前驱结点)
    前驱线索化 (处理当前结点)
    pre=P;//pre是前驱结点, P是当前结点
    InThreading(P->rchild);
  }
}
```

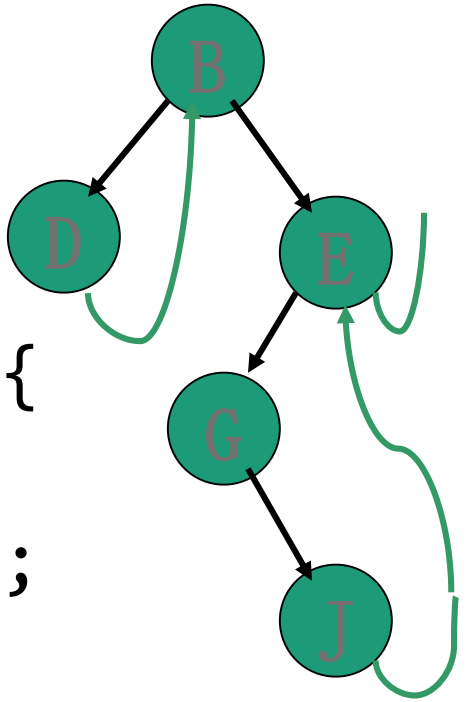
中序线索化



- 后继线索化——处理前驱结点

```
void InThreading(BiThrTree P){
  if (P){
    InThreading(P->lchild);
    if (pre && !pre->rchild){
      pre->RTag=Thread; pre->rchild=P;
    } //后继线索化
    前驱线索化 。。。见下页
    pre=P;
    InThreading(P->rchild);
  }
}
```

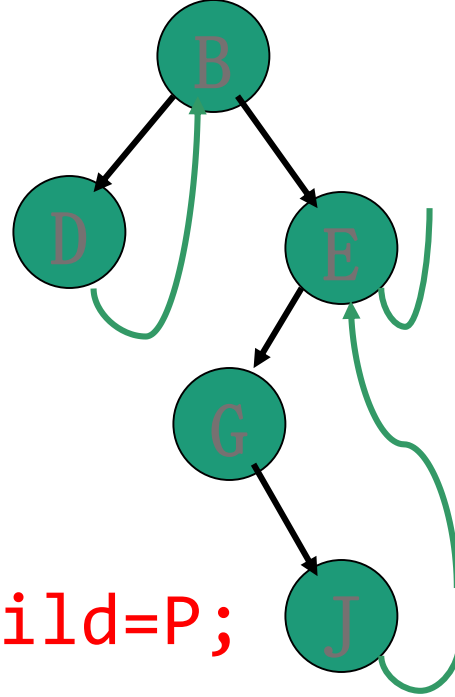
中序线索化



- 前驱线索化——处理后继（当前）结点

```
void InThreading(BiThrTree P){
  if (P){
      InThreading (P->lchild);
      后继线索化。。。见上页
      if (!P->lchild){
         P->LTag=Thread; P->lchild=pre;
      } //前驱线索化
      pre=P;
      InThreading(P->rchild);
  }
}
```

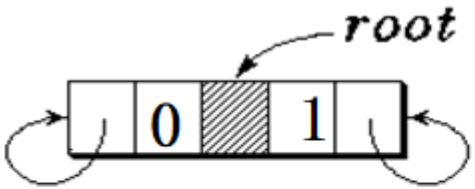
中序线索化（算法6.7）



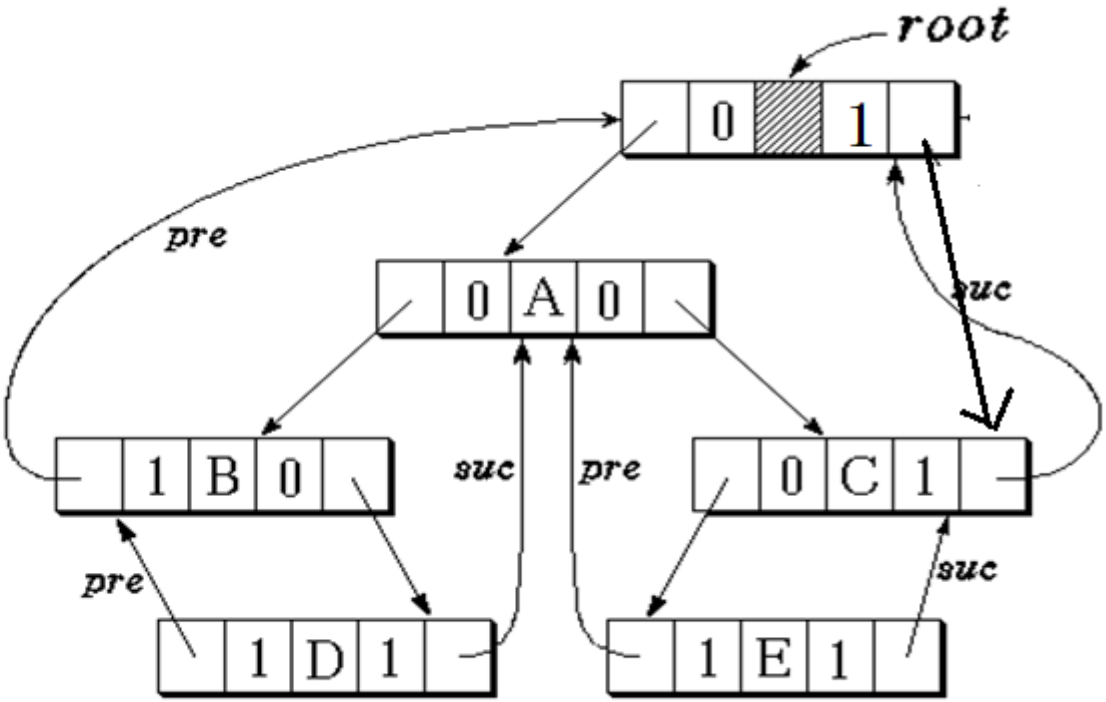

```
void InThreading(BiThrTree P){
  if (P){
      InThreading (P->lchild);
      if (pre && !pre->rchild){
      pre->RTag=Thread; pre->rchild=P;
      } //后继线索化
      if (!P->lchild){
         P->LTag=Thread; P->lchild=pre;
      } //前驱线索化
      pre=P;
      InThreading(P->rchild);
  }
}
```

中序线索化

- 在二叉树的线索链表上加一个头结点，另其lchild域的指针指向根结点；其rchild域的指针指向中序遍历的最后一个结点。双向线索链表
- 二叉树中序序列中的第一个结点的lchild域的指针和最后一个结点的rchild域的指针均指向头结点。



（a）空二叉链表



（b）非空二叉链表

算法6.6 （带表头结点）

注：指针pre始终指向当前访问结点的前驱结点

```
int InOrderThreading(BiThrTree &Thrt,
BiThrTree T){
    if (!(Thrt =
(BiThrTree)malloc(sizeof(BiThrNode))))
exit(OVERFLOW);
    Thrt->LTag = Link; Thrt->RTag=Thread;
    Thrt->rchild=Thrt;
    if (!T) Thrt->lchild = Thrt;
    else {Thrt->lchild=T;  pre=Thrt;
         InThreading(T);
         pre->rchild = Thrt;    pre-
>RTag=Thread;     Thrt->rchild = pre;
    }return OK; }
```

带头结点的中序线索化

- 算法6.6 ：指针pre始终指向当前访问结点的前驱结点

```
int InOrderThreading(BiThrTree &Thrt,
BiThrTree T){
   if (!(Thrt =
(BiThrTree)malloc(sizeof(BiThrNode))))
exit(OVERFLOW);
   Thrt->LTag = Link; Thrt->RTag=Thread;
   Thrt->rchild=Thrt;
   if (!T) Thrt->lchild = Thrt;
   else {Thrt->lchild=T;   pre=Thrt;
        InThreading(T);
        pre->rchild = Thrt;     pre-
>RTag=Thread;    Thrt->rchild = pre;
   }return OK;}
```

中序遍历带头结点的中序线索树

- 算法6.5

```
Status InOrderTraverse_Thr(BiThrTree T,
Status (*Visit)(TElemType e){
   p=T->lchild;
   while (p!=T){ //是否指向头结点
        while(p->LTag==Link) p=p->lchild;
        if (!Visit(p->data) return ERROR;
        while(p->RTag==Thread && p-
>rchild !=T){    p=p->rchild;
              Visit(p->data);//访问后继结点
        }
   }return OK;
}//InOrderTraverse_Thr
```

中序线索化

```
void InThreading2 (BiThrTree P){
  if(P){ InThreading2 (P->lchild);
       if (!P->lchild)
            { P->LTag=Thread; P-
>lchild=pre;}
       if (!P->rchild) P->RTag=Thread;
       if (pre && pre->RTag==Thread)
pre->rchild=P;
       pre=P;
       InThreading2 (P->rchild);
    }
  }
```

先序线索化

```
void PreThreading(BiThrTree P){
    if (P){
     if (!P->lchild){
        P->LTag=Thread;P->lchild=pre;}
       if (!P->rchild) P->RTag=Thread;
     if (pre && pre->RTag==Thread) pre-
>rchild=P;
       pre=P;
       if (P->LTag==Link) PreThreading(P-
>lchild);
       if (P->RTag==Link) PreThreading(P-
>rchild);
      }    }
```

后序线索化

```
void PostThreading(TriThrTree P){
    if (P){
      PostThreading(P->lchild);
       PostThreading(P->rchild);
       if (!P->lchild){
          P->LTag=Thread; P->lchild=pre; }
          if (!P->rchild) P->RTag=Thread;
       if (pre && pre->RTag==Thread) pre-
>rchild=P;
       pre=P;
     }
   }
```

6.5 树和森林

1. 树的存储结构

1) 多重链表（标准存储结构）

定长结构 （n为树的度）指针利用率不高

data	$child_1$	$child_2$	$child_3$	·····	$child_n$

一般采用定长结构

不定长结构 d为结点的度，节省空间，但算法复杂

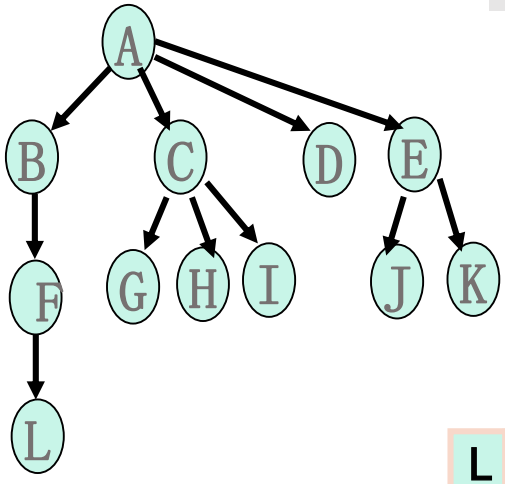
data	*d*	$child_2$	$child_3$	·····	$child_d$

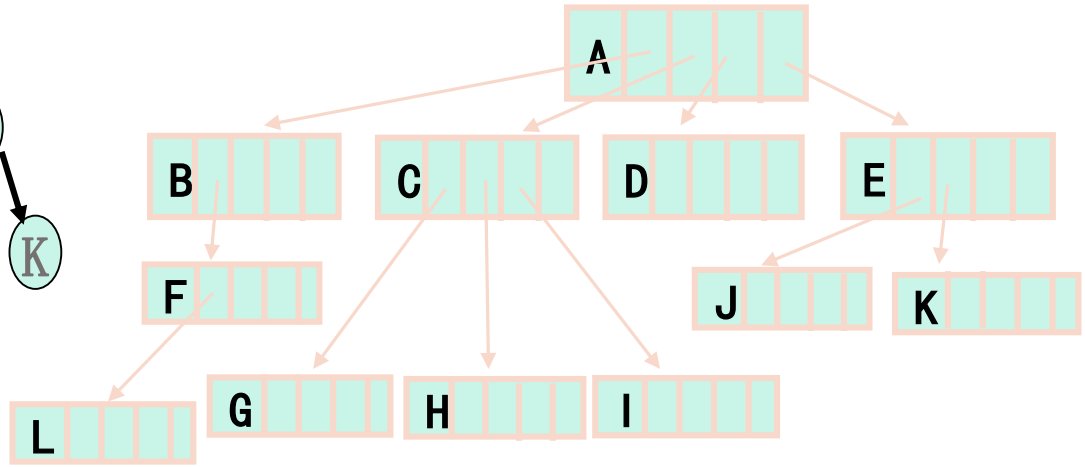
如有n个结点，树的度为k，则共有n*k个指针域，只有n-1个指针域被利用，而未利用的指针域为：n*k-(n-1) =n(k-1)+1，未利用率为：(n(k-1)+1)/nk > n(k-1)/nk=(k-1)/k

二次树：1/2 ；三次树：2/3 ；四次树：3/4

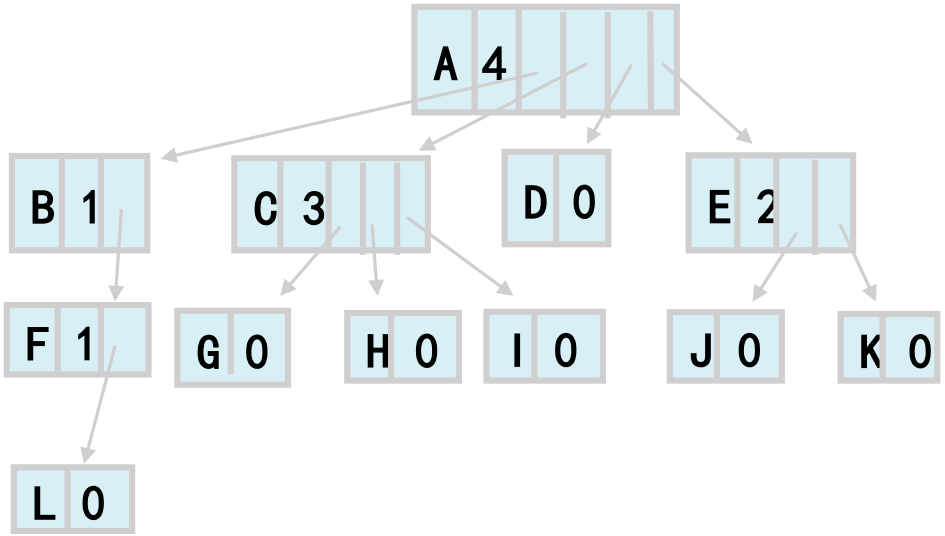
树的度越高，未利用率越高，由于二叉树的利用率较其他树高，因此用二叉树。

定长结构





不定长结构：
（动态分配）
子节点数目改变时，存储空间需重新分配



■ 类型定义:

A. 定长结构:

```
#define N 4
typedef struct dNode{
    ElemType data;
    struct dNode *child[N];
}dNode,*dTree;
```

B. 不定长结构:

```
typedef struct dNode{
    ElemType data;
    int degree;
    struct dNode **child;
}dNode,*dTree;
```

2)常用的其他几种存储结构

- **双亲表示法**

  用结构数组——树的顺序存储方式

  找双亲方便，找孩子难

- **孩子链表表示法**

  顺序和链式结合的表示方法

  找孩子方便，找双亲难

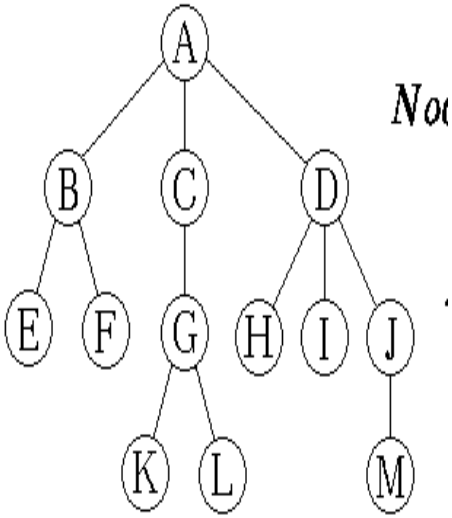
  若用带双亲的孩子链表表示，则找孩子找双亲都较方便

- **孩子兄弟表示法**

  找孩子容易，若增加parent域，则找双亲也较方便。

双亲表示类型定义

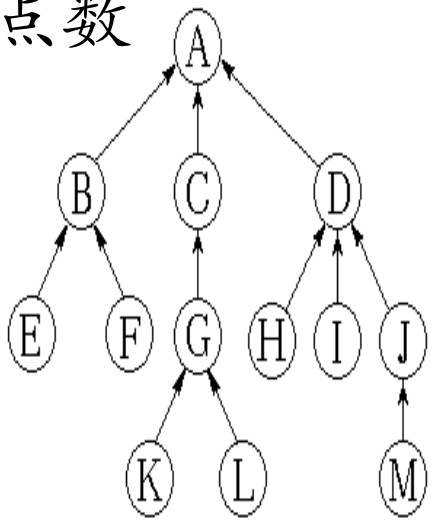
```
#define MAX_TREE_SIZE  100
typedef struct PTNode{
   TElemType data;
   int parent;
} PTNode;
typedef struct{
        PTNode nodes[MAX_TREE_SIZE];
        int r,n;  //根位置和结点数
}PTree;
```



(a) 树

NodeList	1	2	3	4	5	6	7	8	9	10	11	12	13
data	A	B	E	F	C	G	K	L	D	H	I	J	M
parent	0	1	2	2	1	5	6	6	1	9	9	9	12

(b) 双亲表示数组

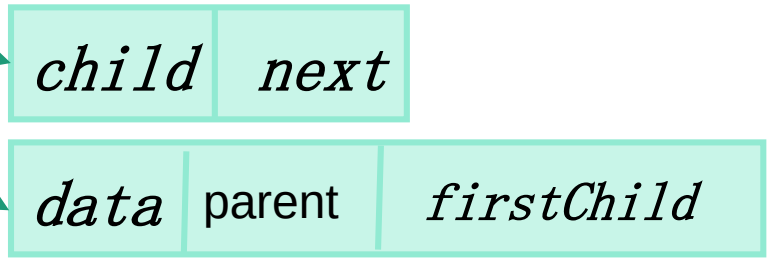


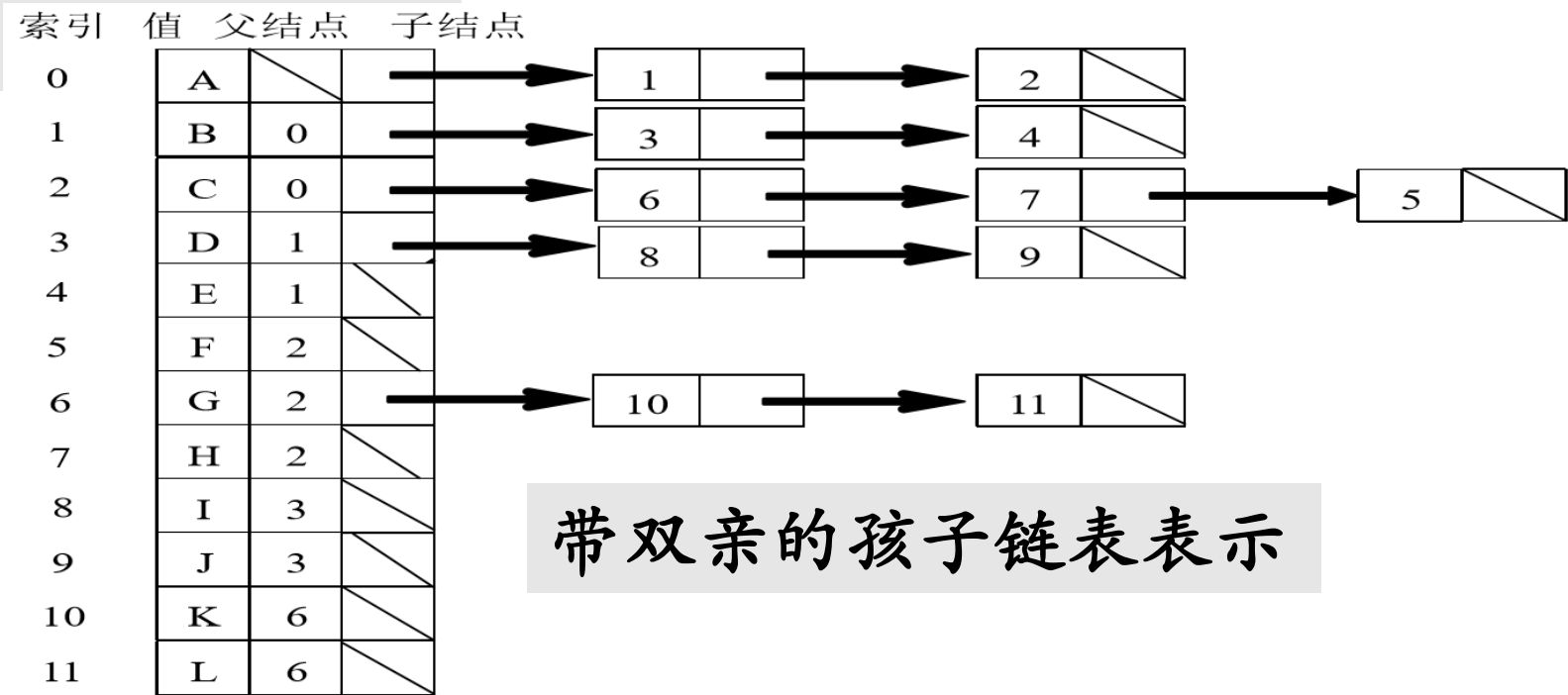
(c) 双亲表示图解

■孩子链表表示法()

```
typedef struct CTNode{
int  child;
struct CTNode  *next;
}*ChildPtr;
typedef struct {
ElemType  data;
int parent;
ChildPtr  firstchild;
}CTBox;
```

```
typedef struct{
    CTBox nodes[MAX_TREE_SIZE];
    int n,r;
}CTree;
```



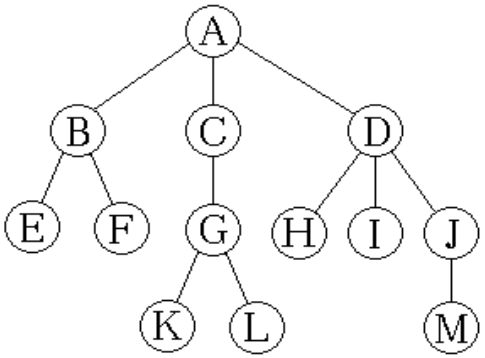


带双亲的孩子链表表示

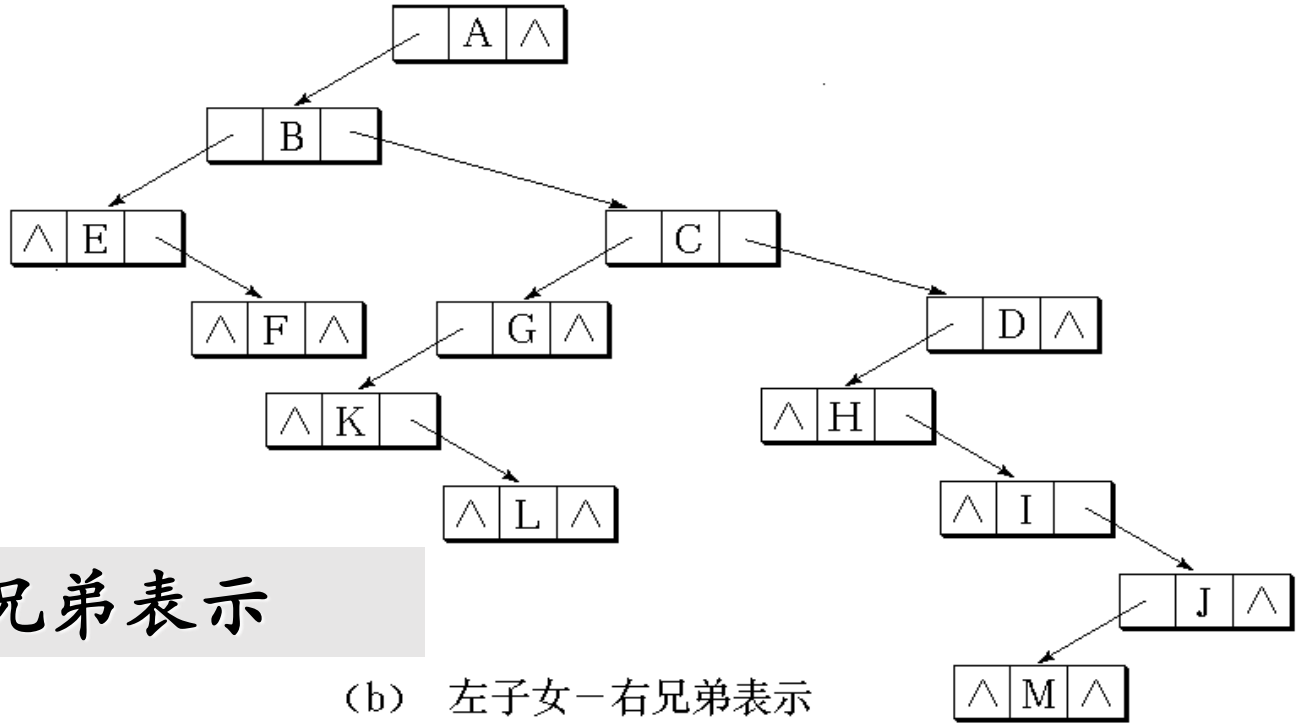
■孩子兄弟表示法（二叉链表表示法）

data	*firstChild*	*nextSibling*

```
typedef struct CSNode{
ElemType  data;
struct CSNode  *firstChild, *nextSibling;
}CSNode,*CSTree;
```



（a）树



（b） 左子女－右兄弟表示

树的左孩子-右兄弟表示

2.森林、树与二叉树的转换

- **森林:**

  m个（m≥0）棵互不相交的树的集合。

- **树:**

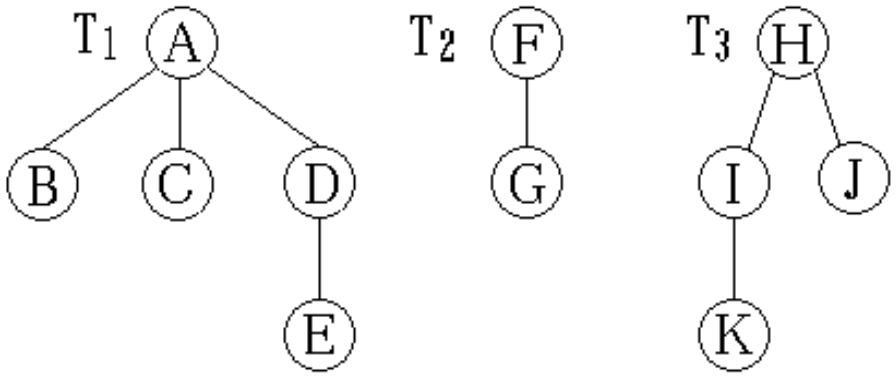
  Tree=(root,F)，其中：root是树的根结点，F是森林,是其子树的集合，F=(T1,T2,… Tm)，其中Ti=(ri,Fi)称作根root的第i棵子树;

  当m≠0时，树根和子树森林存在下列关系：

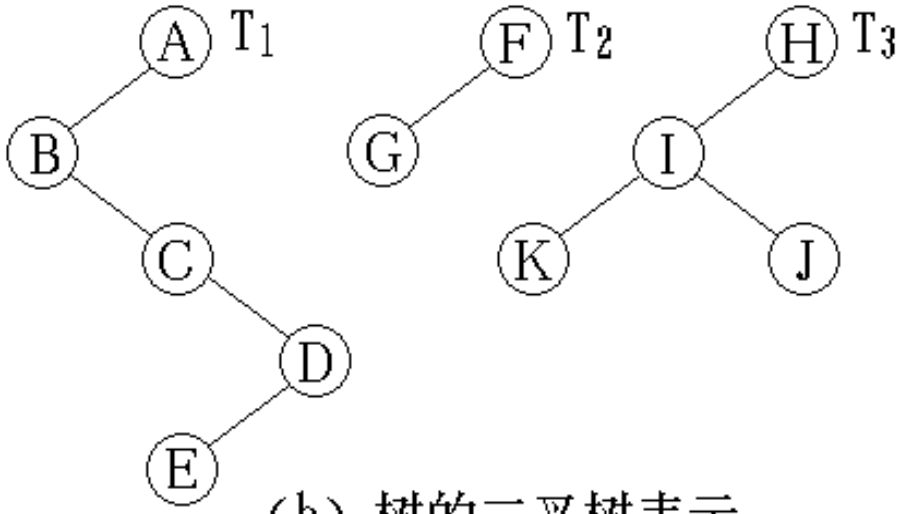
  $$RF=\{<root,ri>|i=1,2,\ldots,m,\ m>0\}$$

- **森林和树存在相互递归的关系**

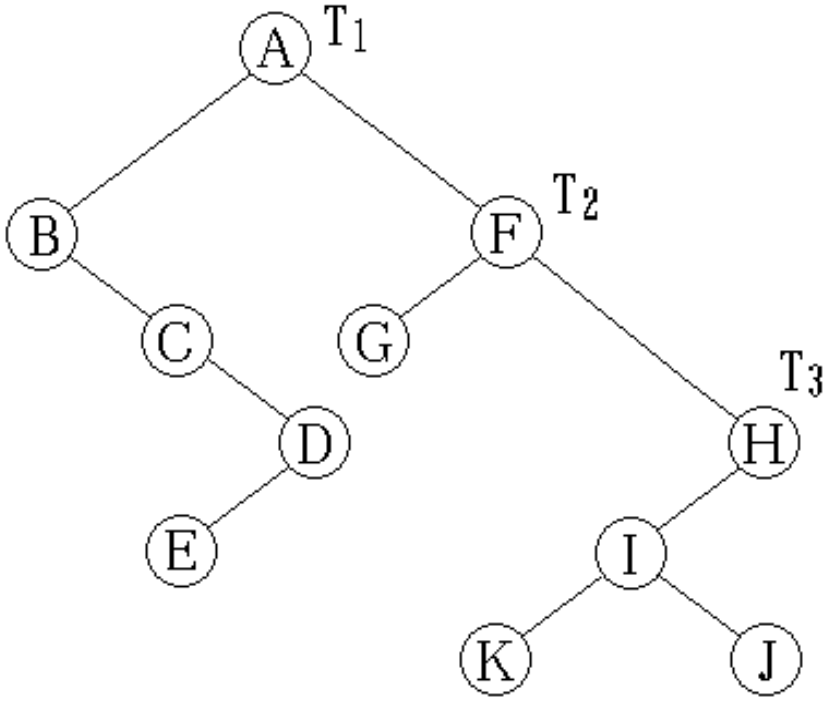
■森林与二叉树的对应关系



（a）3棵树的森林



（b）树的二叉树表示



（c）森林的二叉树表示

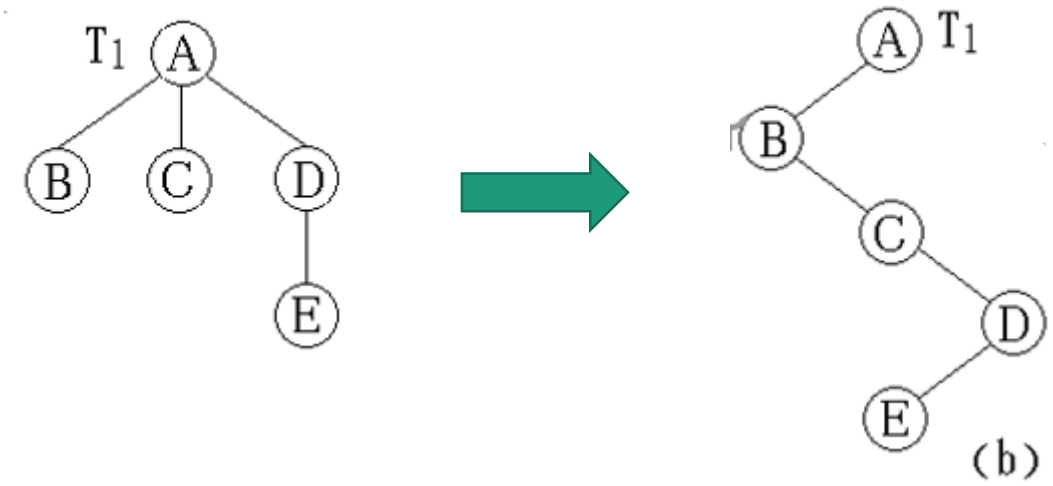
(1) 树转化成二叉树的简单方法

① 在同胞兄弟之间加连线;

② 保留结点与第一个孩子之间的连线，去掉其余连线;

③ 顺时针旋转45度。

④ 以根结点为轴；左孩子(只有一个孩子)不再旋转。

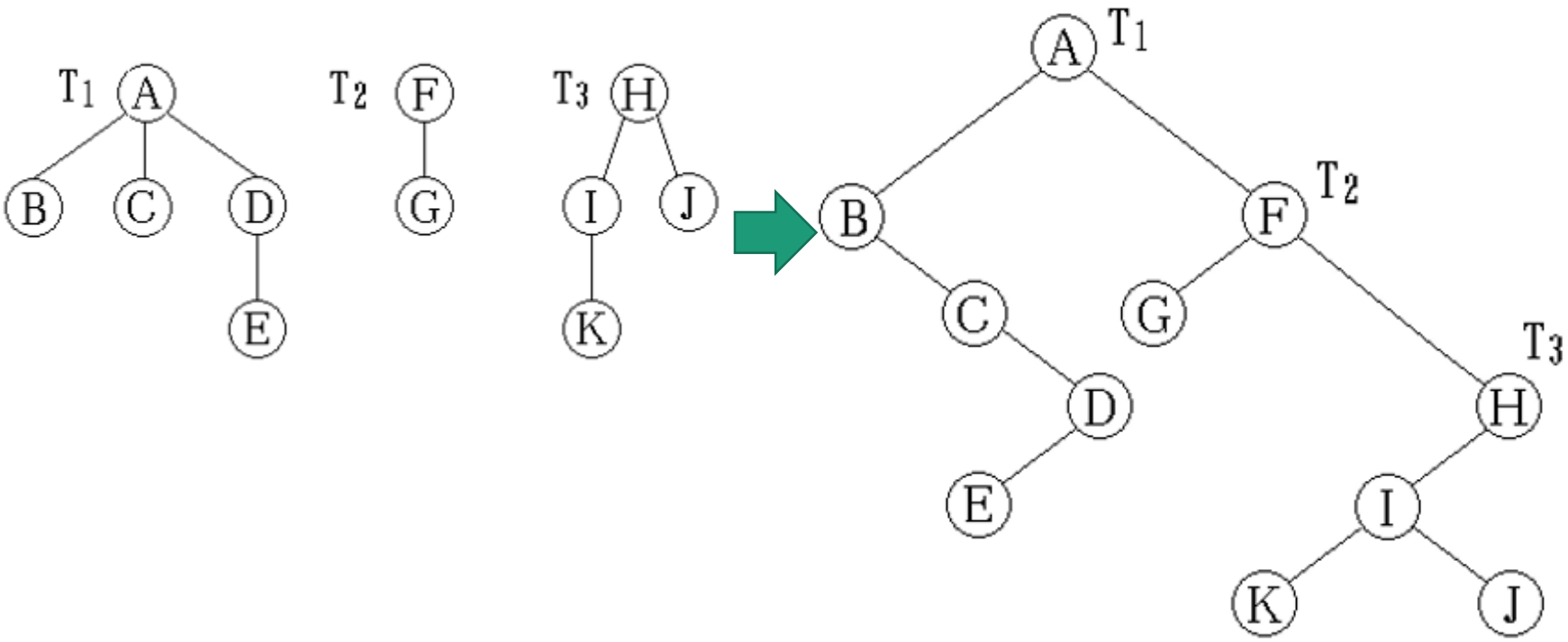


（2）森林转化成二叉树

① 将森林中的每棵树转换成对应的二叉树；

② 将森林中已经转换成的二叉树的各个根视为兄弟，各兄弟之间自第一棵树根到最后一棵树根之间加连线；

③ 以第一棵树的根为轴，顺时针旋转45度。



(3) 森林转化成二叉树的规则

若F为空，即n = 0，则对应的二叉树B为空二叉树。

若F不空，则对应二叉树B的根root (B)是F中第一棵树T_1的根root (T_1)；其左子树为B (T_{11}，T_{12}，…，T_{1m})，其中，T_{11}，T_{12}，…，T_{1m}是root (T_1)的子树；其右子树为B (T_2，T_3，…，T_n)，其中，T_2，T_3，…，T_n是除T_1外其它树构成的森林。

(4) 二叉树转换为森林的规则

① 如果B为空，则对应的森林F也为空。

② 如果B非空，则F中第一棵树T_1的根为root；T_1的根的子树森林$\{T_{11}, T_{12}, \cdots, T_{1m}\}$是由root的左子树LB转换而来，F中除了$T_1$之外其余的树组成的森林$\{T_2, T_3, \cdots, T_n\}$是由root的右子树RB转换而成的森林。

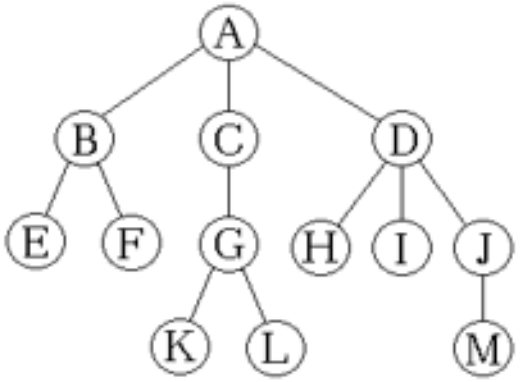
3．树和森林的遍历

- **树的遍历（先根遍历、后根遍历、层次遍历）**

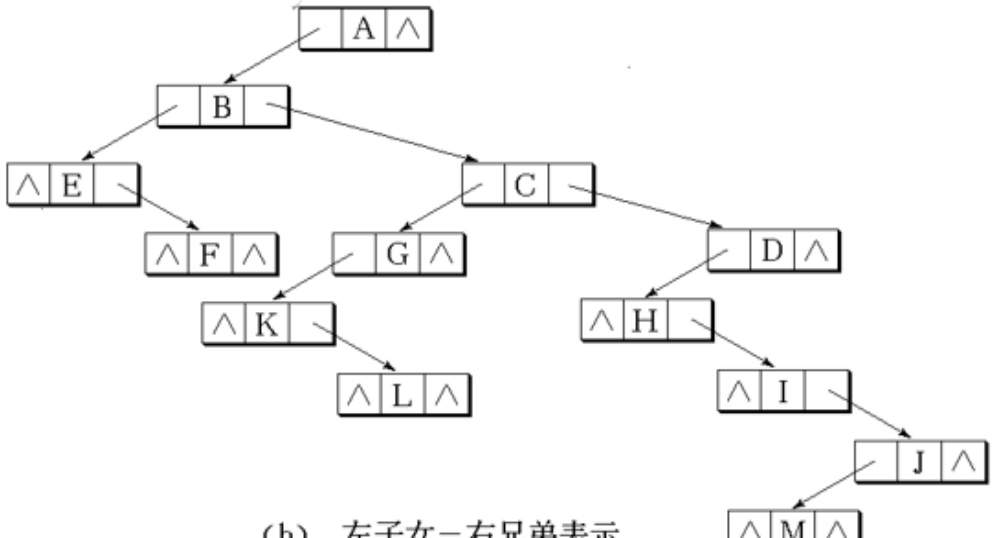
  （1）先根遍历：先根访问树的根结点，然后依次先根遍历根的每棵子树

  （2）后根遍历：先依次后根遍历每棵子树，然后访问根结点

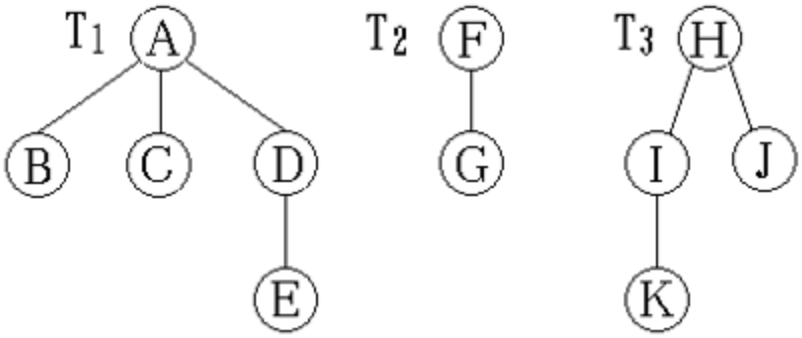
- **树的先根遍历对应转换后的二叉树的先序遍历**
- **树的后根遍历对应转换后的二叉树的中序遍历**



（a）树



■森林的遍历



(1) 先根次序遍历的规则：

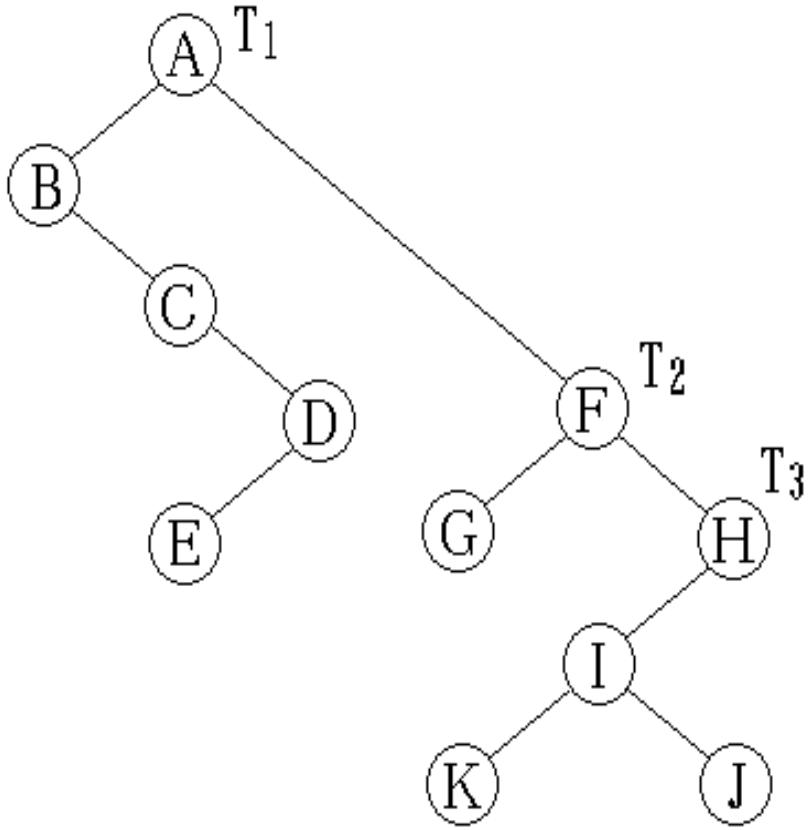
若森林F为空，返回；

否则

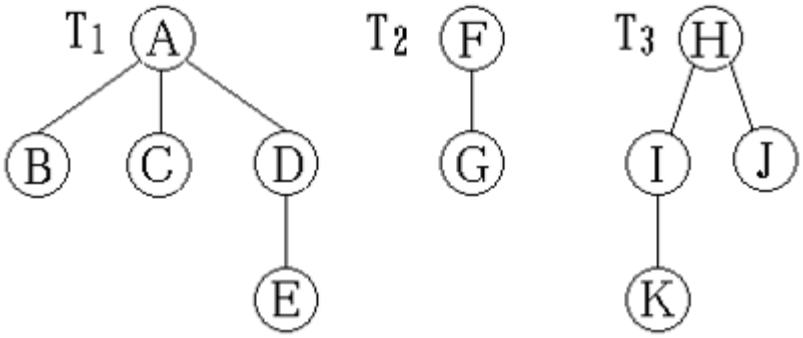
① 访问F的第一棵树的根结点；

② 先根次序遍历第一棵树的子树森林；

③ 先根次序遍历其它树组成的森林。



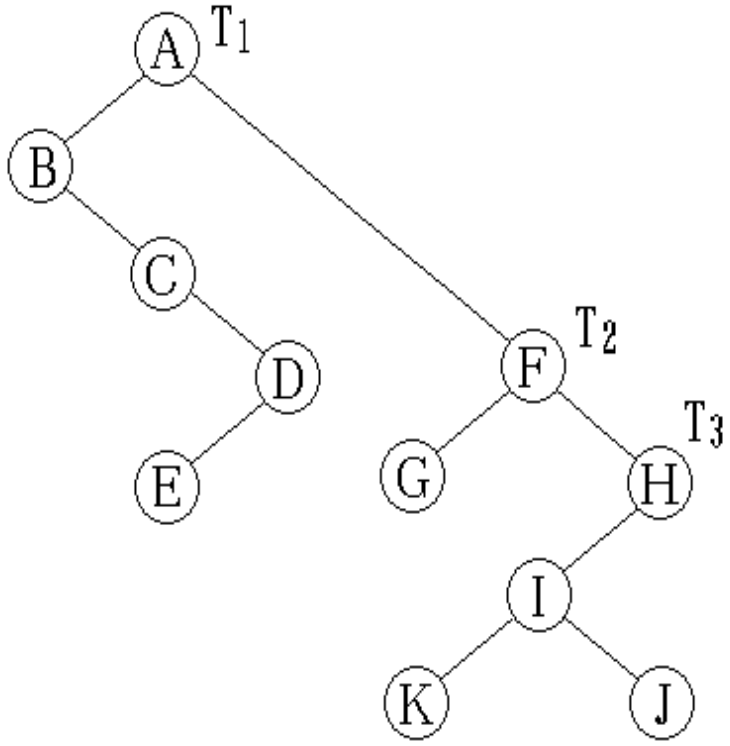
森林的二叉树表示

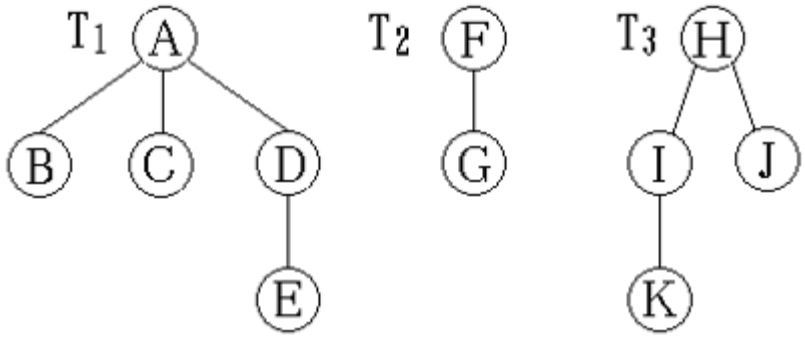


(2) 中根次序遍历的规则:

若森林F为空，返回;
否则

① 中根次序遍历第一棵树的子树森林;
② 访问F的第一棵树的根结点;
③ 中根次序序遍历其它树组成的森林。

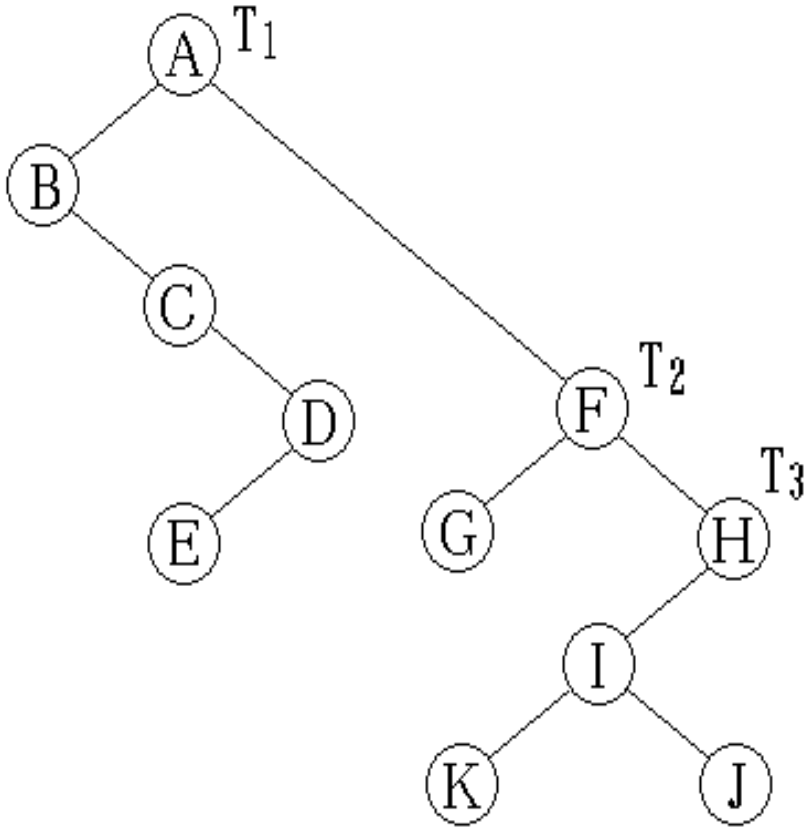


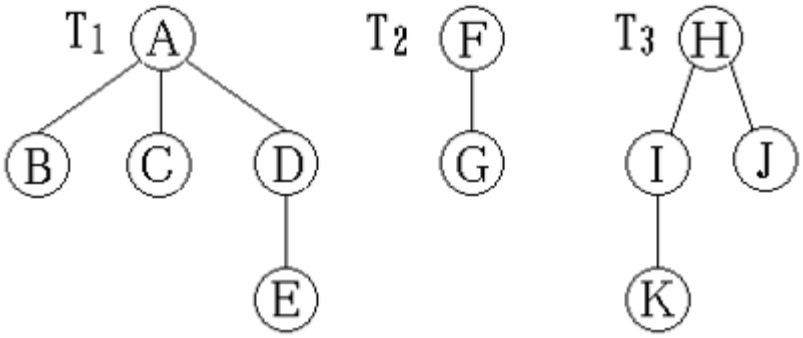


(3) 后根次序遍历的规则：

若森林F为空，返回；
否则

① 后根次序遍历第一棵树的子树森林；
② 后根次序遍历其它树组成的森林；
③ 访问F的第一棵树的根结点。





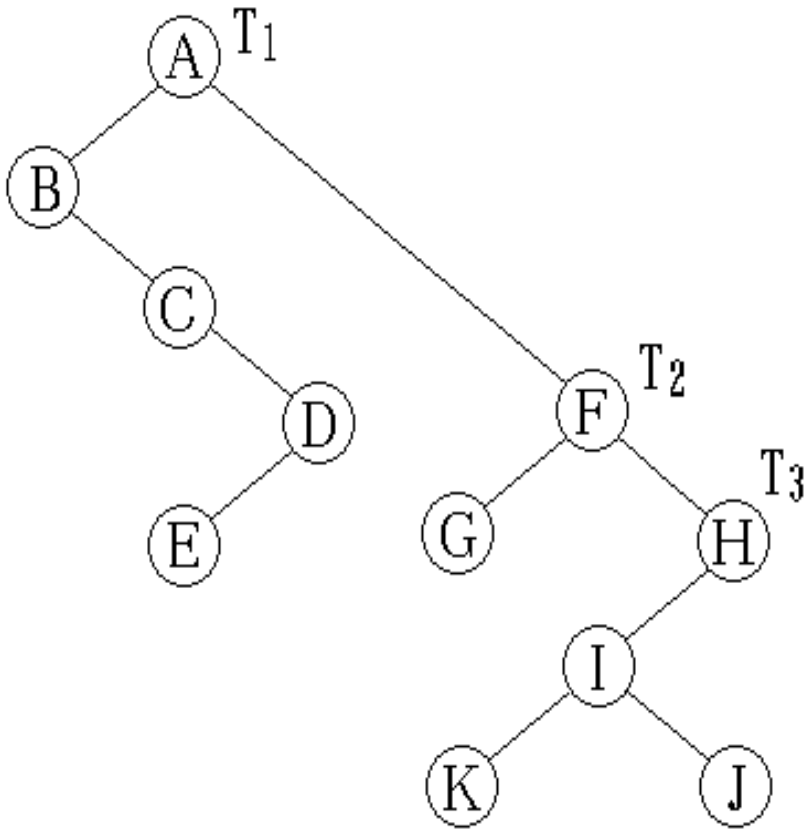
(4) 广度优先遍历(层次序遍历)：

若森林F为空，返回；否则

① 依次遍历各棵树的根结点；

② 依次遍历各棵树根结点的所有孩子；

③ 依次遍历这些孩子结点的孩子结点。



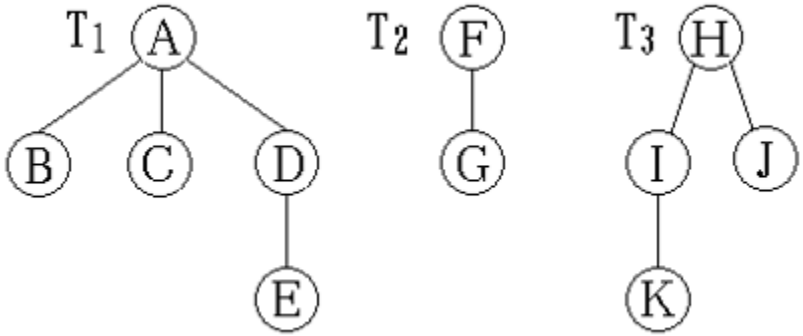
森林的二叉树表示

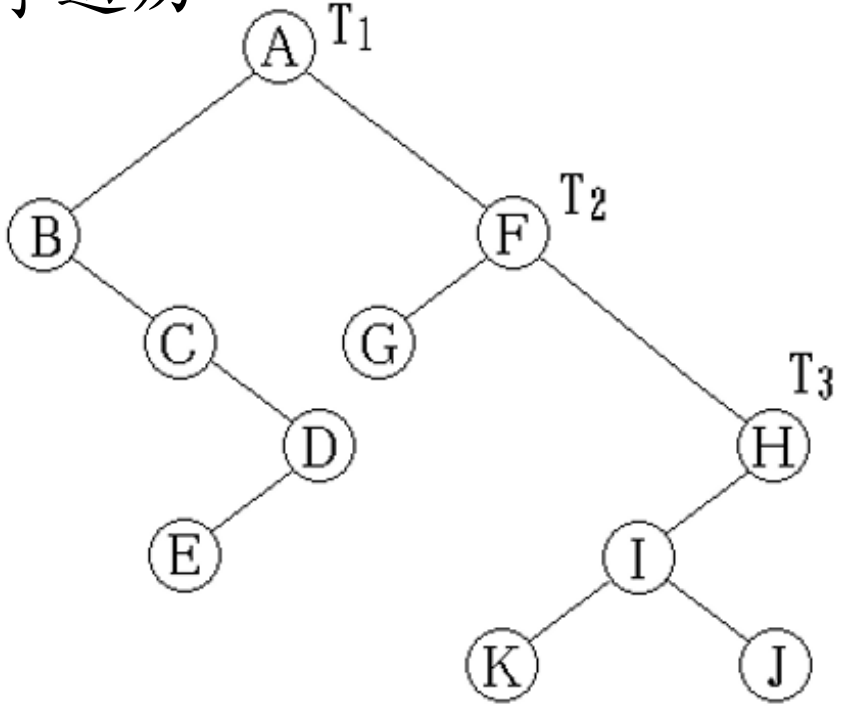
（5）森林的遍历与二叉树的遍历

先根：对应二叉树的先序遍历

中根：对应二叉树的中序遍历

后根：对应二叉树的后序遍历





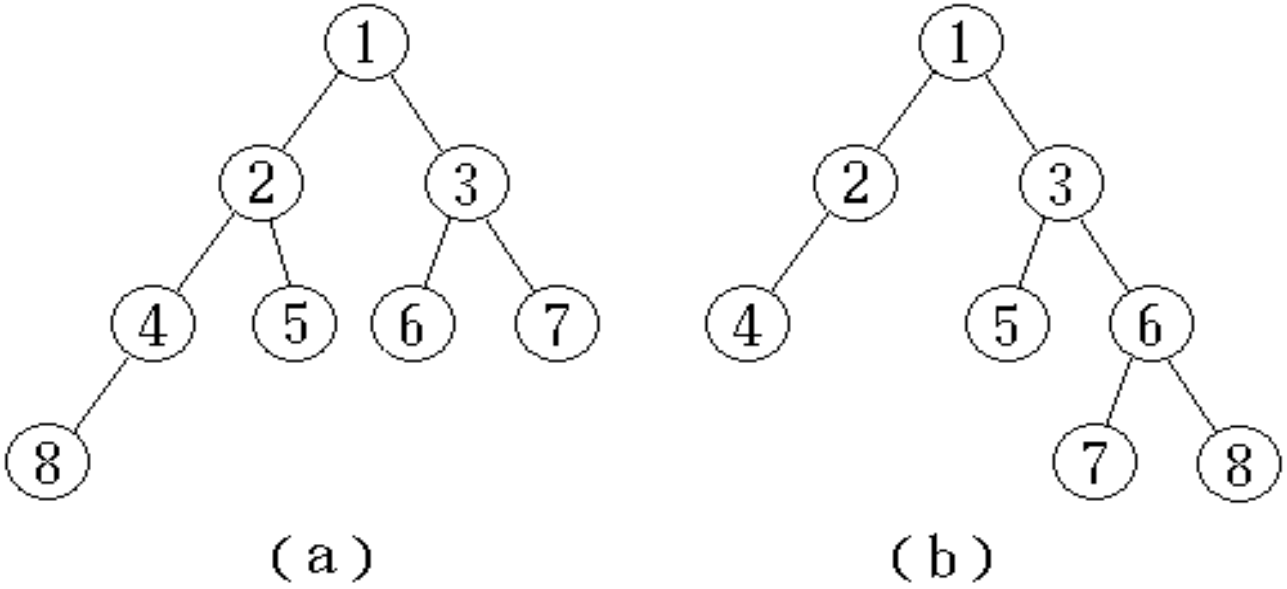
6.6 赫夫曼树及其应用

1.路径长度（Path Length）

两个结点之间的路径长度是连接两结点的路径上的分支数。

树的路径长度是根结点到每个结点的路径长度之和。

例：具有不同路径长度的二叉树



- n个结点的二叉树的路径长度不小于下述数列前n项的和，即

$$PL=\sum_{i=1}^{n}\lfloor \log_2 i \rfloor$$

$$=0+1+1+2+2+2+2+3+3+\cdots$$

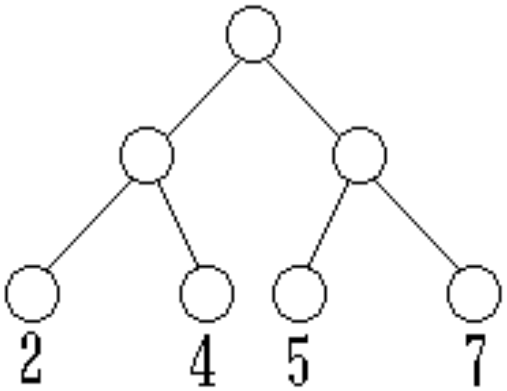
- 其路径长度最小者为 $PL=\sum_{i=1}^{n}\lfloor \log_2 i \rfloor$

2.带权路径长度（ Weighted Path Length, WPL ）

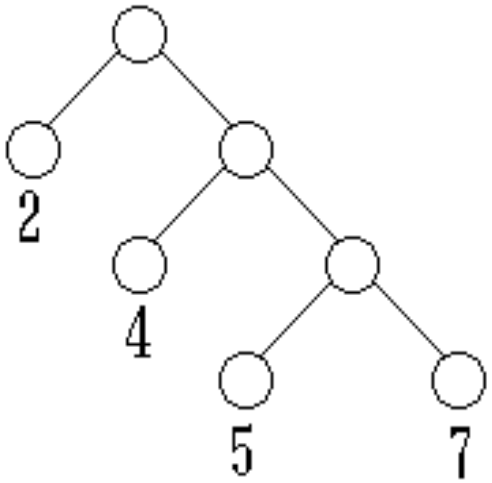
树的带权路径长度是树的各叶结点所带的权值与该结点到根的路径长度的乘积的和。

$$WPL=\sum_{i=1}^{n} w_i * l_i$$

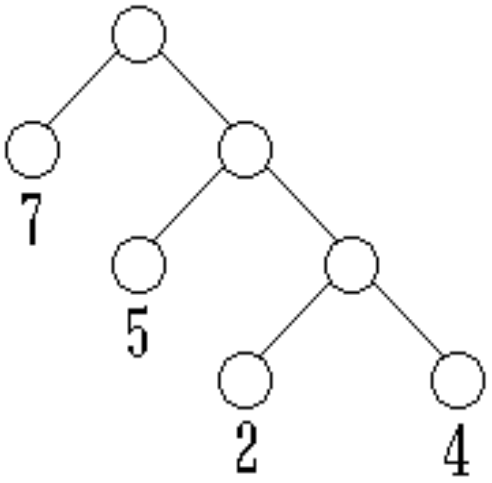
■具有不同带权路径长度的扩充二叉树



(a) $WPL = 36$



(b) $WPL = 46$



(c) $WPL = 35$

3．赫夫曼树（最优二叉树）

带权路径长度达到最小的二叉树即为赫夫曼树。

在赫夫曼树中，权值大的结点离根最近。

■如何构造一棵赫夫曼树?

赫夫曼算法:

(1)由给定的n个权值{w0, w1, w2, …, wn-1},构造具有n棵二叉树的森林F = {T0, T1, T2, …, Tn-1},其中每一棵二叉树Ti只有一个带有权值wi的根结点,其左、右子树均为空。

(2)重复以下步骤,直到F中仅剩下一棵树为止:

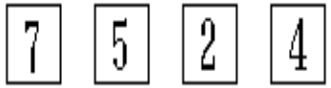
① 在F中选取两棵根结点的权值最小的二叉树,做为左、右子树构造一棵新的二叉树。置新的二叉树的根结点的权值为其左、右子树上根结点的权值之和。

② 在F中删去这两棵二叉树。

③ 把新的二叉树(追)加入到F中。

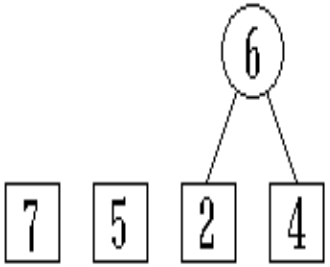
■赫夫曼树的构造过程

F:{7}{5}{2}{4}



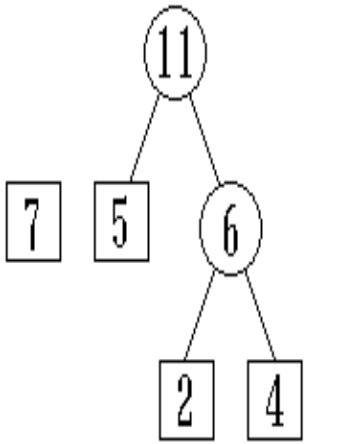
(a) 初始

F:{7}{5}{6}



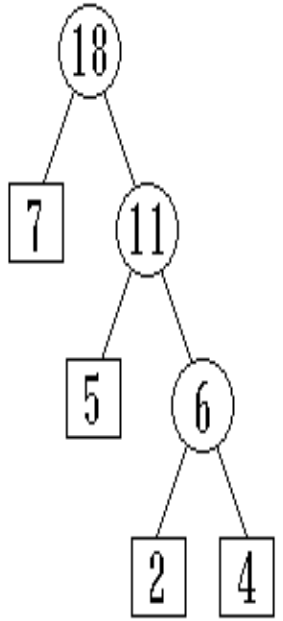
(b) 合并{2}{4}

F:{7}{11}



(c) 合并{5}{6}

F:{18}



(d) 合并{7}{11}

4．赫夫曼树的应用

1)在解决某些判定问题时，利用赫夫曼树可以得到最佳判定算法；

2)用于通讯和数据传送时的赫夫曼编码

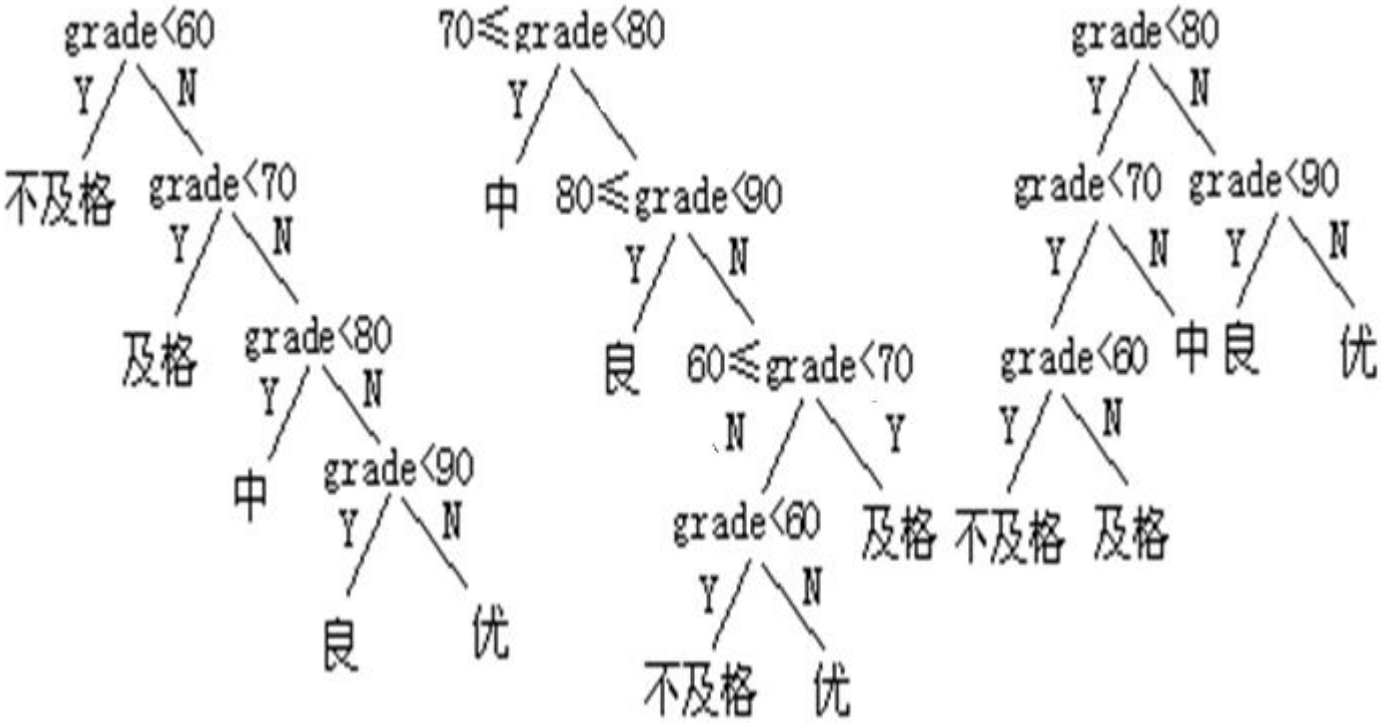
- huffman编码适合于字符频率不等，差别较大的情况
- 不同的频率分布，会有不同的压缩比率
- 大多数商业压缩程序都是采用几种编码方式以应付各种类型的文件。例如：Zip压缩就是LZ77与Huffman结合。

3)归并法外排序，合并顺串。

例如：编制一个将百分数的成绩转化为优、良、中、及格和不及格五级制表示的程序。假设成绩分布规律如下：

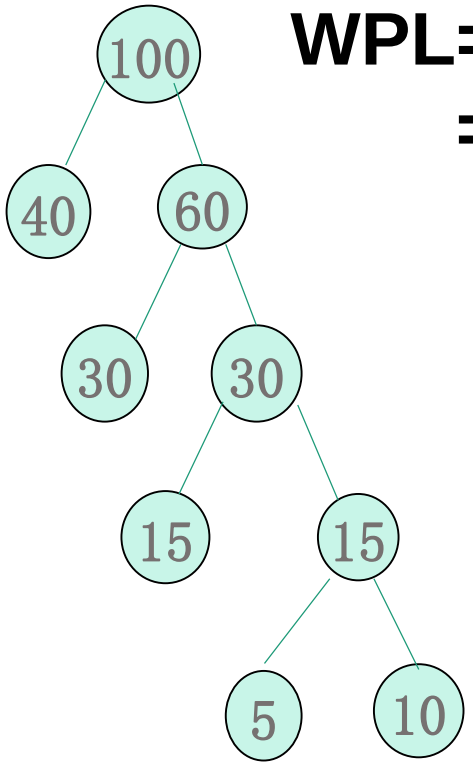
0-59	60-69	70-79	80-89	90-100
0.05	0.15	0.40	0.30	0.10

假设有10000个学生成绩， a树需比较31500次，b树需比较20500次，C树需比较22000次。



例.根据成绩分布求哈夫曼树，得到最佳判定算法

0-59	6-69	70-79	80-89	90-100
0.05	0.15	0.40	0.30	0.10



WPL=(5+10)*4+15*3+30*2+40*1
=205

赫夫曼编码

- 主要用途是实现数据压缩。

  设给出一段报文:

  CAST CAST SAT AT A TASA

  字符集合是 { C, A, S, T }, 各个字符出现的频度(次数)是 W={ 2, 7, 4, 5 }。

  若给每个字符以等长编码

  A : 00 T : 10 C : 01 S : 11

  则总编码长度为 (2+7+4+5) * 2 = 36.

  若按各个字符出现的概率不同而给予不等长编码, 可望减少总编码长度。

  因各字符出现的概率为{ 2/18, 7/18, 4/18, 5/18}。

化整为 { 2，7，4，5 }，以它们为各叶结点上的权值，建立赫夫曼树。左分支赋 0，右分支赋 1，得赫夫曼编码(变长编码)。

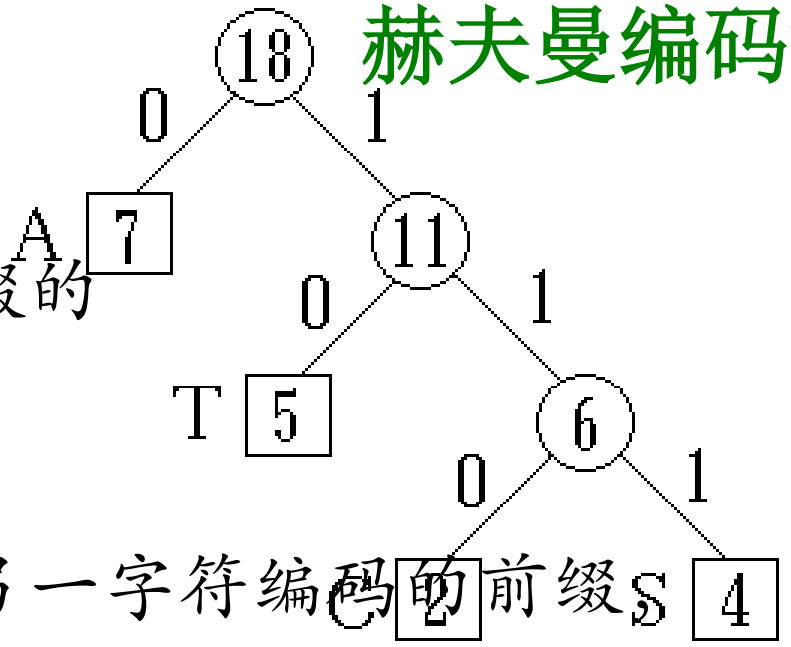
A : 0　　T : 10　　C : 110　　S : 111

它的总编码长度：7*1+5*2+(2+4)*3 = 35。

比等长编码的情形要短。

总编码长度正好等于赫夫曼树的带权路径长度WPL。

赫夫曼编码是一种无前缀的编码。解码时不会混淆。



5. 前缀编码

任一字符的编码都不是另一字符编码的前缀称前缀编码。

建立赫夫曼树及求赫夫曼编码

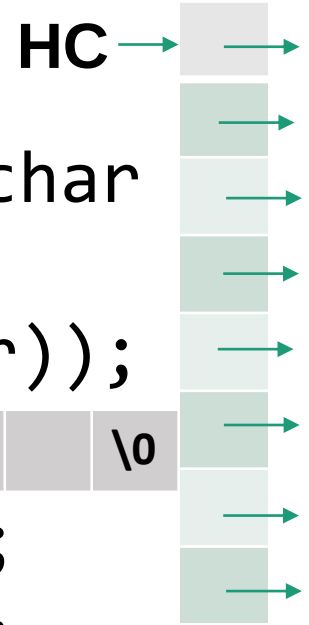
- 算法6.12

```
typedef struct {
   unsigned int weight;
   unsigned int parent,lchild,rchild;
} HTNode, *HuffmanTree;
typedef char  **HuffmanCode;
void HuffmanCoding(HuffmanTree
&HT,HuffmanCode &HC,int *w, int n){
HuffmanTree p;  char *cd;int i,s1,s2,start;
unsigned int c,f;
if (n<=1) return;// n为字符数目          int
m=2*n-1;/*m为结点数目*/
HT=(HuffmanTree)malloc((m+1)*sizeof(HTNode)
); //  0号单元未用
```

```
for (p=HT+1, i=1; i<=n; ++i,++p,++w){
  p->weight = *w; p->parent=0;
  p->lchild = 0;p->rchild=0; }
  // *w={5,29,7,8,14,23,3,11}
  // *p = { *w,0,0,0 };
 for (; i<=m;++i,++p){
   p->weight = 0; p->parent=0; p-
>lchild=0; p->rchild=0; }
//*p={ 0,0,0,0 };
 for (i=n+1; i<=m;++i) //  建赫夫曼树
 { Select(HT,i-1,s1,s2);
   HT[s1].parent=i; HT[s2].parent = i;
   HT[i].lchild = s1;HT[i].rchild= s2;
  HT[i].weight = HT[s1].weight +
HT[s2].weight;     }
```

■//从叶子到根逆向求赫夫曼编码

```
  HC =
(HuffmanCode)malloc((n+1)*sizeof(char
*)); //  0号单元未用
  cd = (char*)malloc(n*sizeof(char));
  cd[n-1]='\0';
  for (i=1;i<=n;++i){ start = n-1;
    for (c=i,f=HT[c].parent; f!=0;
c=f,f=HT[f].parent)
      if (HT[f].lchild ==c) cd[- -
start]='0';  else cd[--start]='1';
    HC[i]=(char *)malloc((n-
start)*sizeof(char));
    strcpy(HC[i],&cd[start]);
    printf("%s\n",HC[i]);   }
  free(cd); }
```



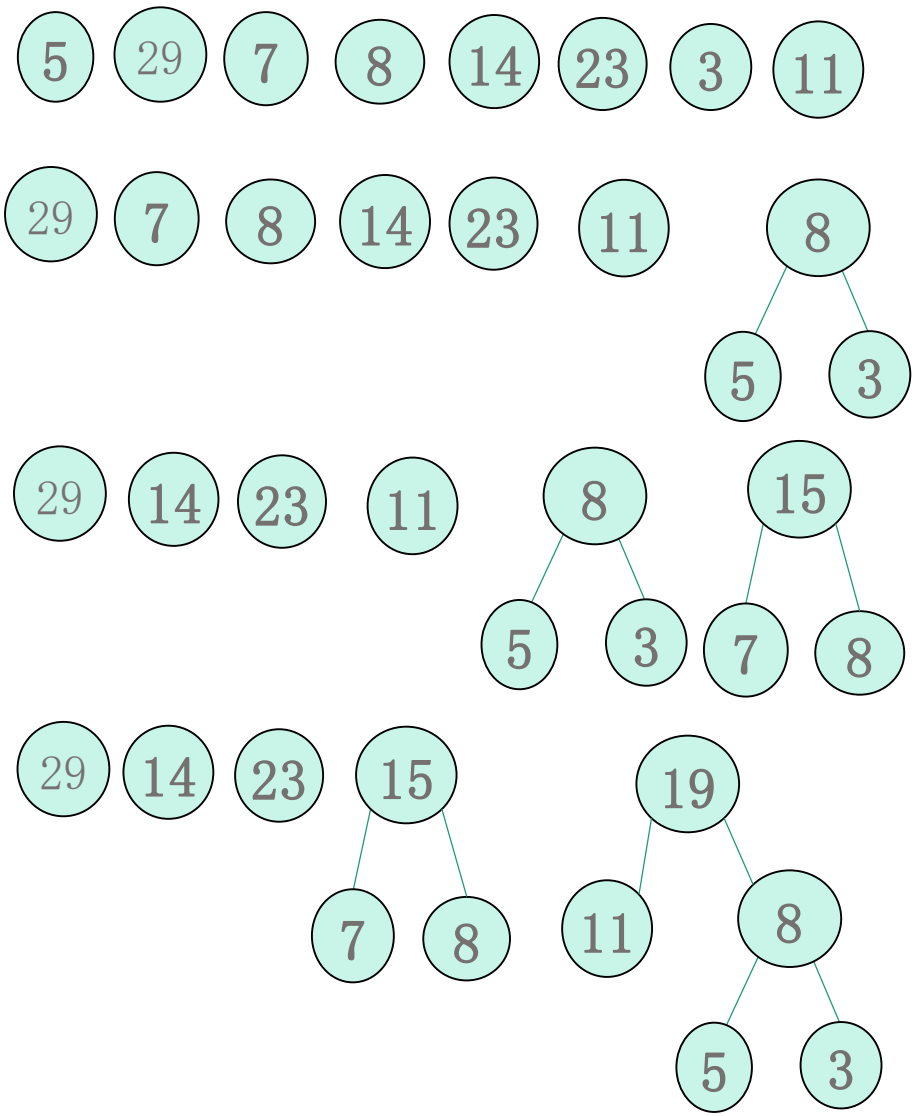
- //无栈非递归遍历，求赫夫曼编码

```
  HC = (HuffmanCode)malloc((n + 1)*
sizeof(char*));
  cd = (char*)malloc(n * sizeof(char));
  int p1 = m; int cdlen = 0;
  for (i = 1;i<=m;++i) HT[i].weight = 0;
  while (p1) {
    if (HT[p1].weight == 0) {//向左
      HT[p1].weight = 1;
      if (HT[p1].lchild != 0) {
  p1=HT[p1].lchild;cd[cdlen++]= '0';}
      else if (HT[p1].rchild == 0) {
  HC[p1]=(char*)malloc((cdlen+1)*sizeof(ch
ar));       cd[cdlen] = '\0';
strcpy(HC[p1],cd);}  }/*end 向左*/  else
```

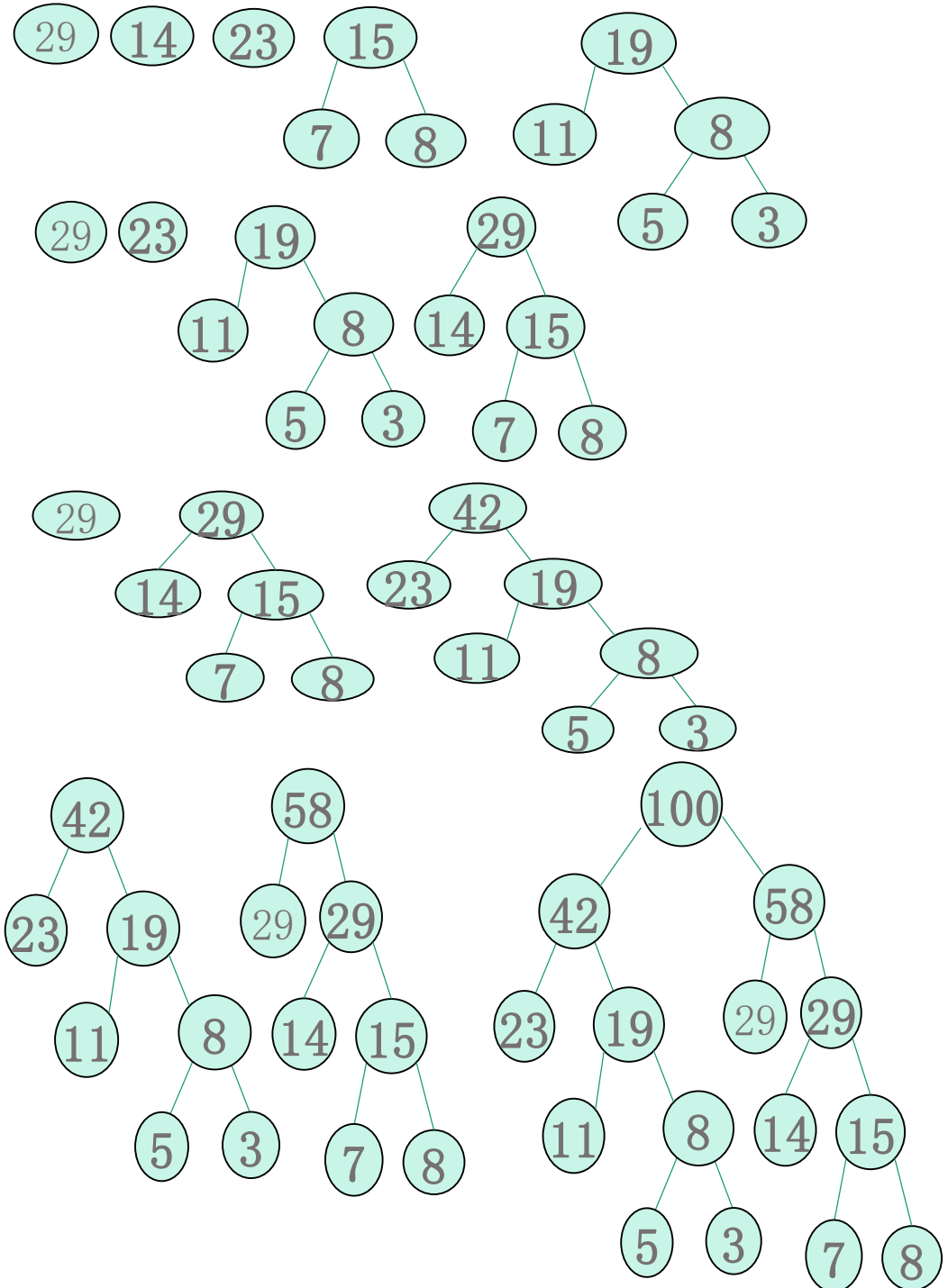
- //无栈非递归遍历，求赫夫曼编码

```
        if (HT[p1].weight == 1) {
            HT[p1].weight = 2;
            if (HT[p1].rchild != 0) {
              p1 = HT[p1].rchild;
              cd[cdlen++] = '1';
            }
         }
       else {
            HT[p1].weight = 0;
            p1 = HT[p1].parent; --cdlen;
            }
       }//while
```

■例.已知某系统在通信联络中只可能出现8种字符，其概率分别为0.05，0.29，0.07，0.08，0.14，0.23，0.03，0.11，试设计哈夫曼编码。

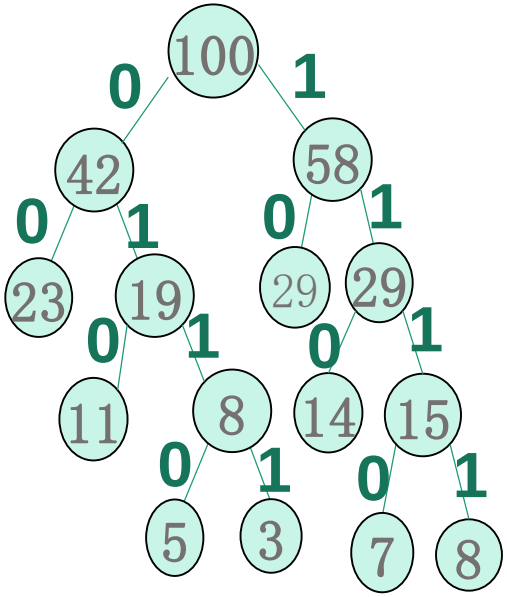


	weight	parent	lchild	rchild
1	5	0 9	0	0
2	29	0	0	0
3	7	0 10	0	0
4	8	0 10	0	0
5	14	0	0	0
6	23	0	0	0
7	3	0 9	0	0
8	11	0 11	0	0
9	8	0 11	0 1	0 7
10	15	0	0 3	0 4
11	19	0	0 8	0 9
12		0	0	0
13		0	0	0
14		0	0	0
15		0	0	0



	weight	parent	lchild	rchld
1	5	0 9	0	0
2	29	0 14	0	0
3	7	0 10	0	0
4	8	0 10	0	0
5	14	0 12	0	0
6	23	0 13	0	0
7	3	0 9	0	0
8	11	0 11	0	0
9	8	0 11	0 1	0 7
10	15	0 12	0 3	0 4
11	19	0 13	0 8	0 9
12	29	0 14	0 5	0 10
13	42	0 15	0 6	0 11
14	58	0 15	0 2	0 12
15	100	0	0 13	0 14

5, 29, 7, 8, 14, 23, 3, 11



HC

1	0	1	1	0	\0
2	1	0	\0		
3	1	1	1	0	\0
4	1	1	1	1	\0
5	1	1	0	\0	
6	0	0	\0		
7	0	1	1	1	\0
8	0	1	0	\0	

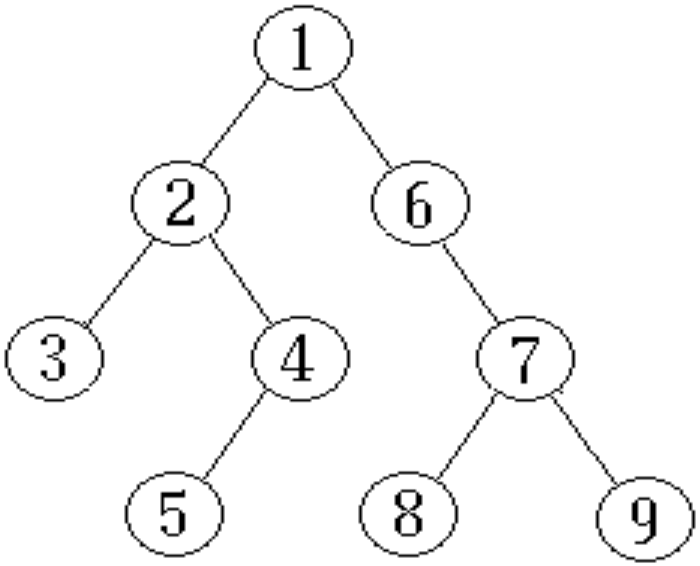
cd

			0	1	1	0	\0

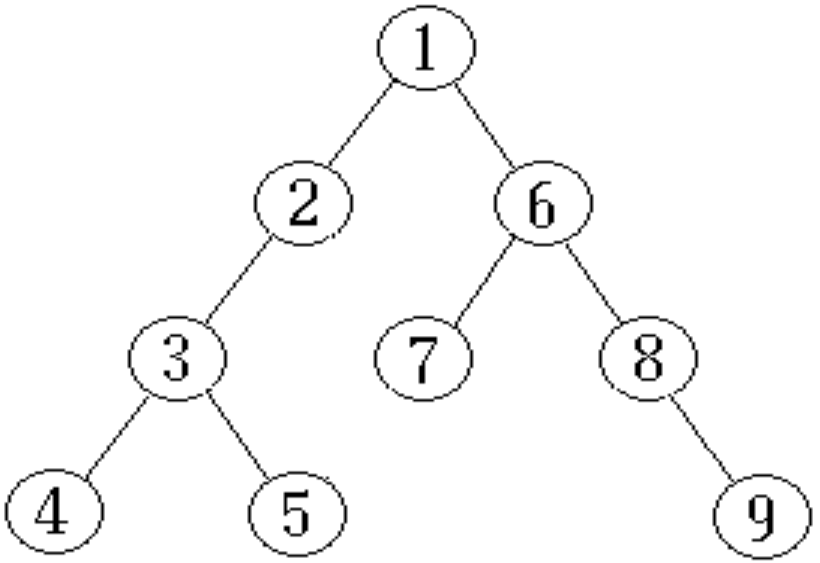
start

6.7 二叉树的计数*

- 如果先序序列固定不变，给出不同的中序序列，可得到不同的二叉树。



（a）

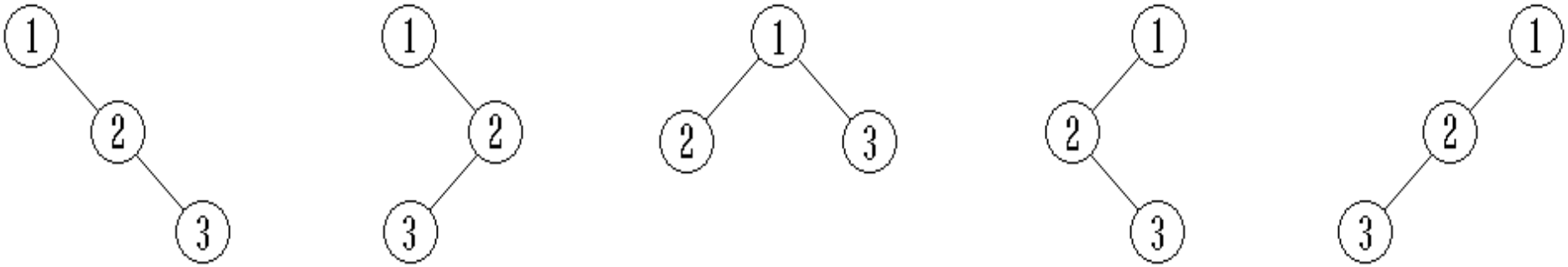


（b）

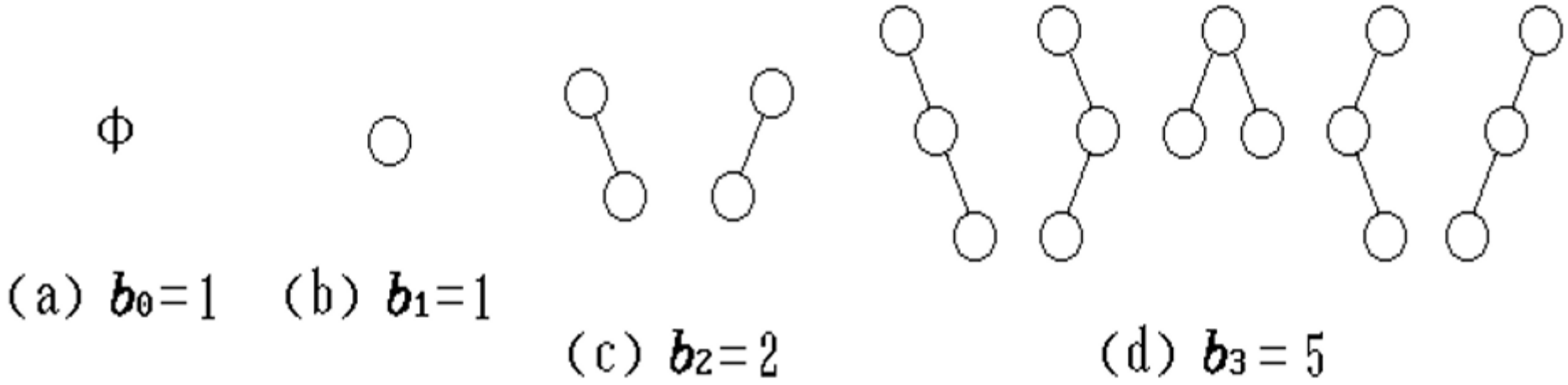
- 问题：有n个数据值，可能构造多少种不同的二叉树?

  我们可以固定先序排列，选择所有可能的中序排列。

- 例如，有 3 个数据 { 1，2，3 }，它们的先序排列均为 123，可得5种不同的二叉树。中序序列可能是 123，132，213，231，321。

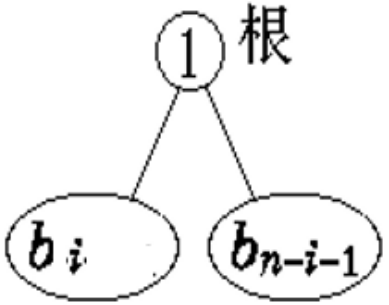


- 有0个，1个，2个，3个结点的不同二叉树如下：



- 计算具有n个结点的不同二叉树的棵数

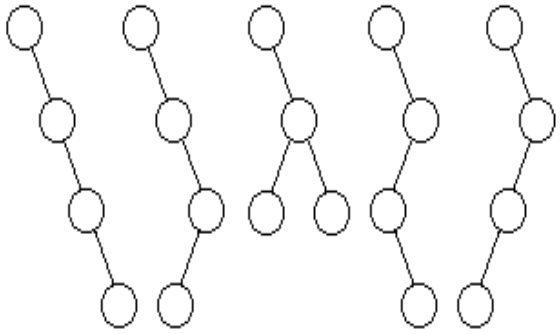
Catalan函数



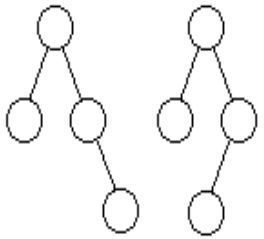
(e) $b_n = \sum_{i=0}^{n-1} b_i \cdot b_{n-i-1}$

$$b_n = \frac{1}{n+1} C_{2n}^{n} = \frac{1}{n+1} \frac{(2n)!}{n! \cdot n!}$$

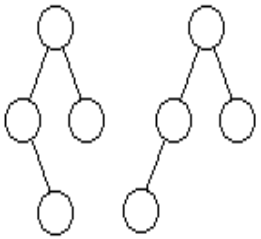
- 具有4个结点的不同二叉树



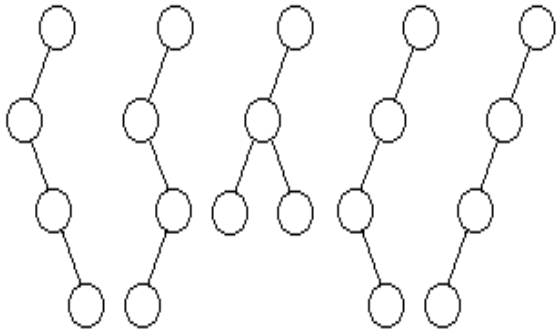
(a) $i=0$



(b) $i=1$



(c) $i=2$



(d) $i=3$