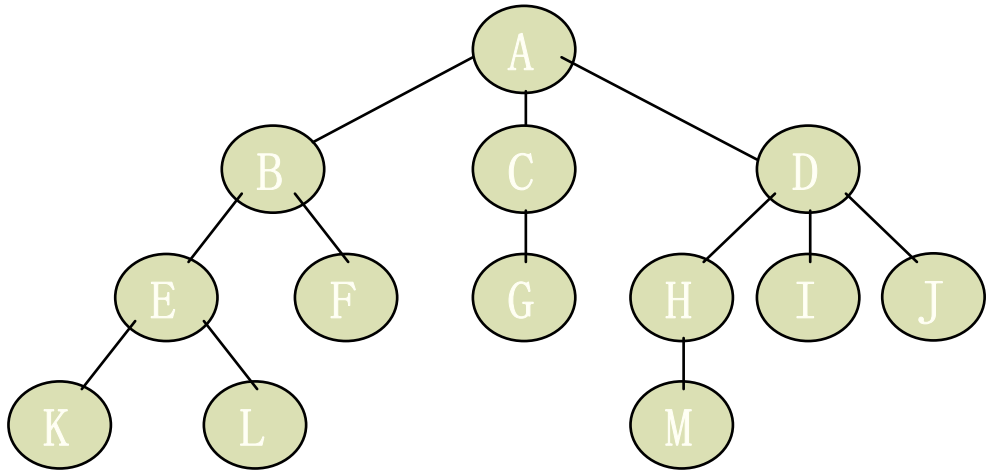
第6章 树和二叉树

- 树的定义和基本术语
- 二叉树（Binary Tree）
- 二叉树的存储结构
- 遍历二叉树（Binary Tree Traversal）
- 线索化二叉树（Threaded Binary Tree）
- 树与森林（Tree & Forest）
- 赫夫曼树（Huffman Tree）

例1．家族族谱

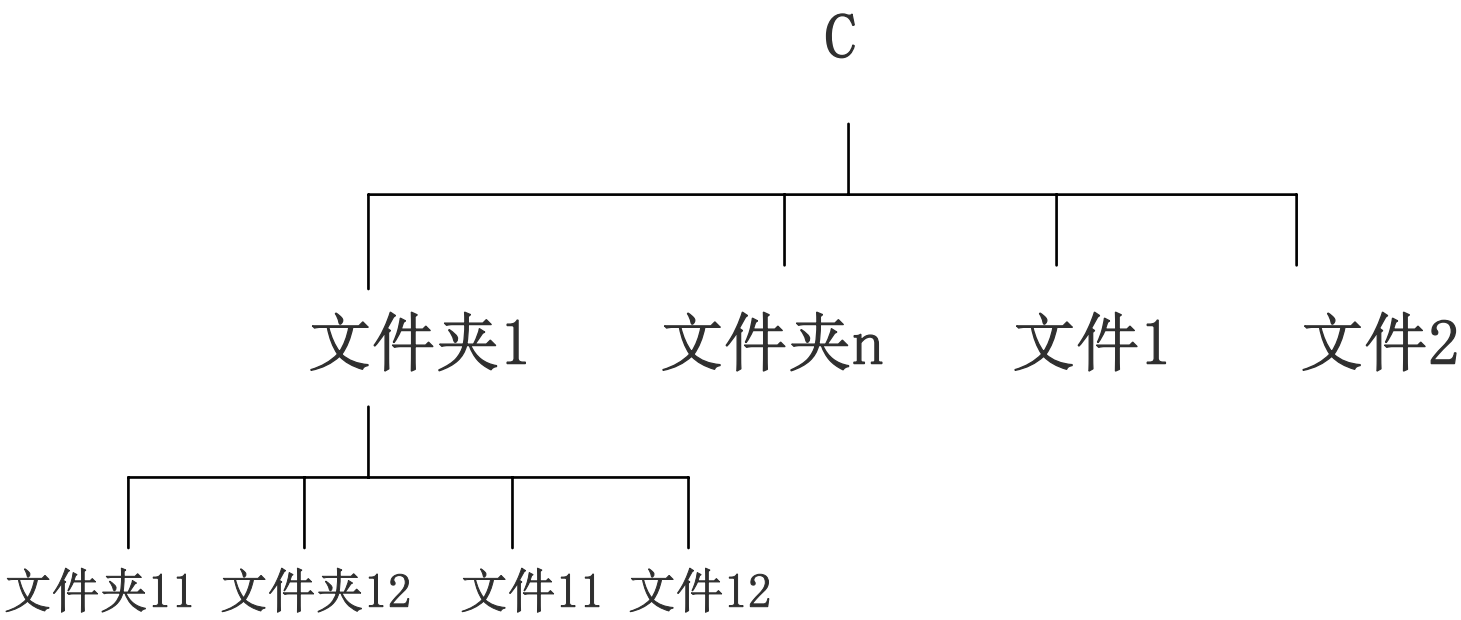
设某家庭有13个成员A、B、C、D、E、F、G、H、I、J，K，L，M他们之间的关系可下图所示的树表示：



例2．单位行政机构的组织关系

例3　计算机的文件系统

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



6.1 树的定义和基本术语

1. 树的定义

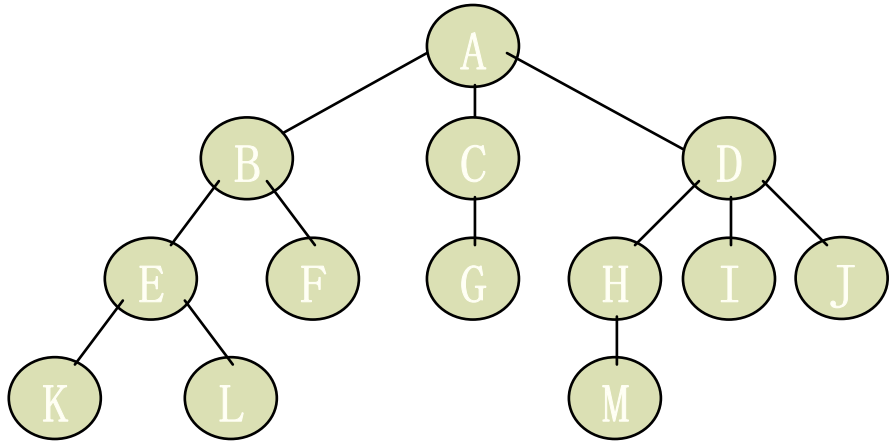
树是由n ($n \geq 0$)个结点组成的有限集合。

如果$n = 0$，称为空树；

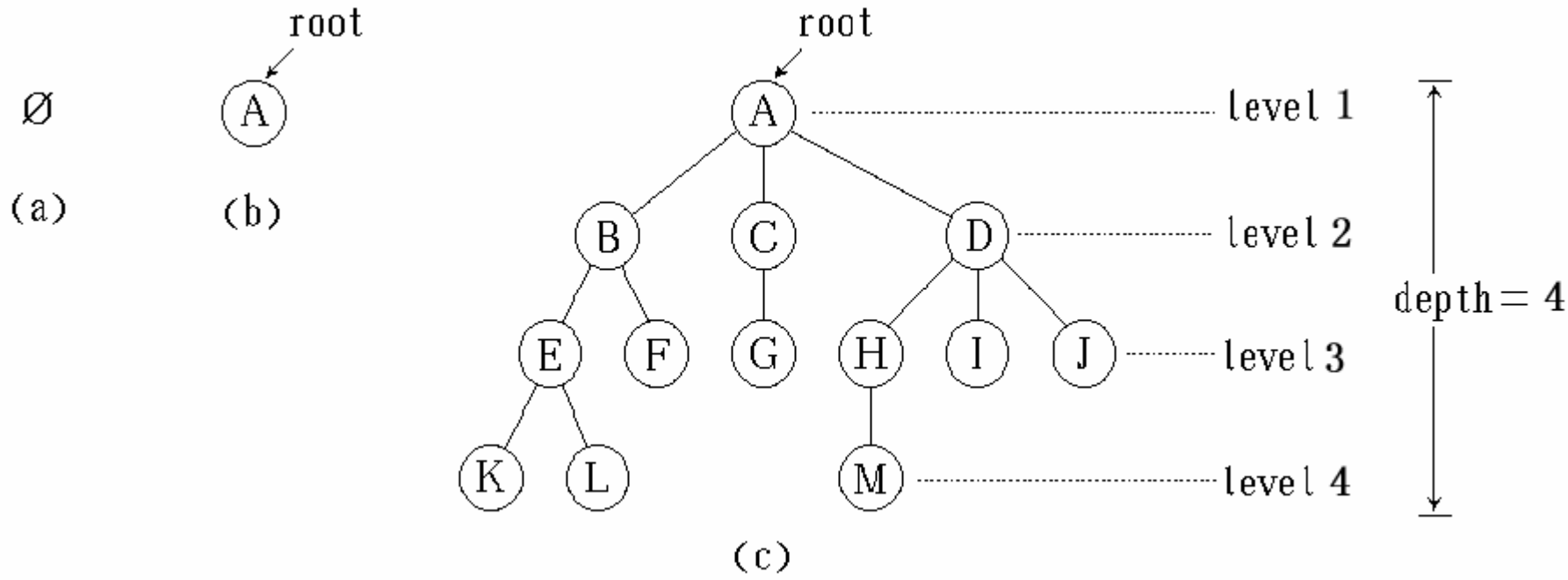
如果$n > 0$，则：

- 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 ($m \geq 0$)个互不相交的有限集合T_0，T_1，…，T_{m-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。

例：

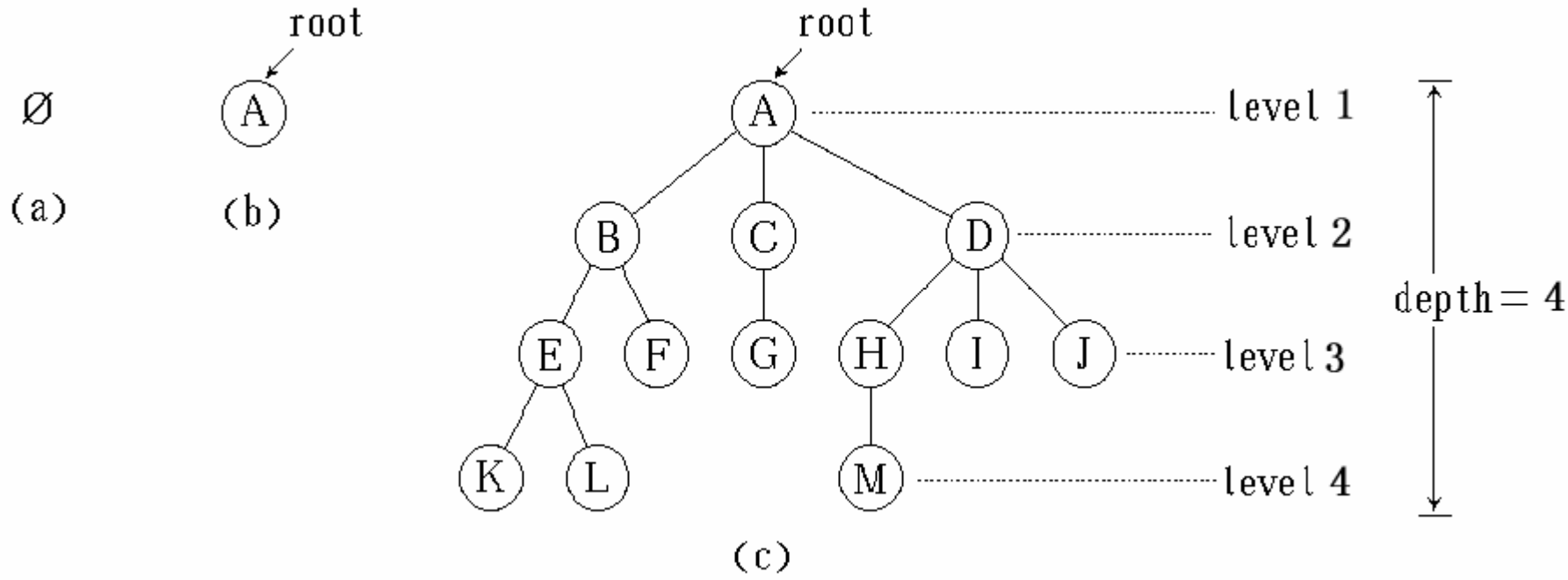


上图中一共有13个结点，其中A是根结点，其余12个结点分成3个互不相交的子集{B，E，F，K，L}，{C，G}，{D，H，I，J，M}，它们都是A的子树。

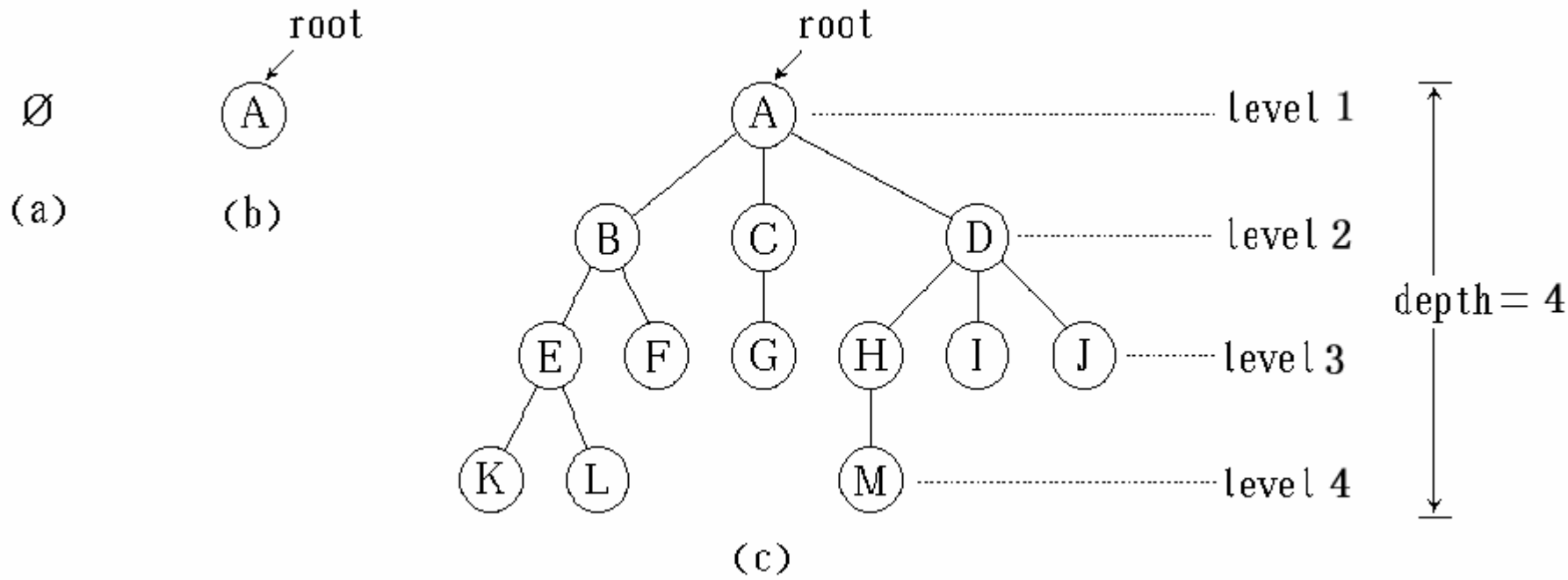


- 结点(node)
- 结点的度(degree)
- 分支(branch)结点
- 叶(leaf)结点
- 孩子(child)结点
- 双亲(parent)结点
- 兄弟(sibling)结点
- 祖先(ancestor)结点
- 子孙(descendant)结点
- 结点所处层次(level)
- 树的深度(depth)
- 树的度(degree)

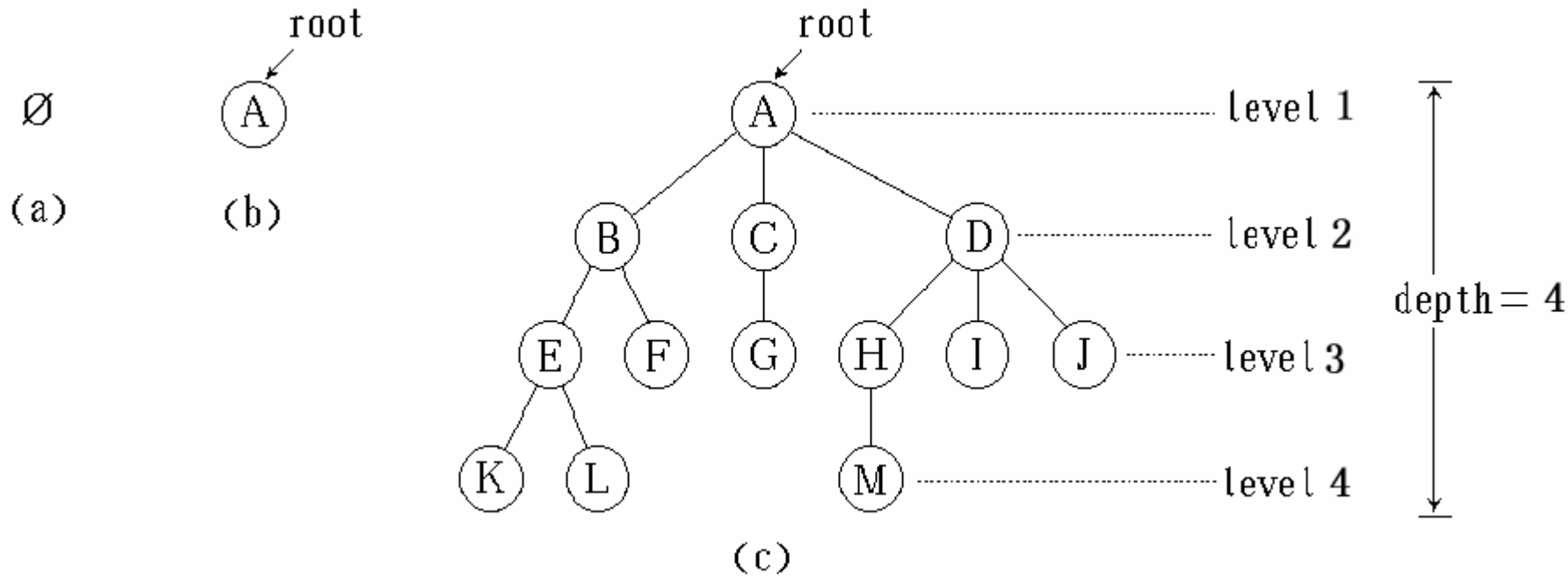
- **有序树**
- **无序树**
- **森林**



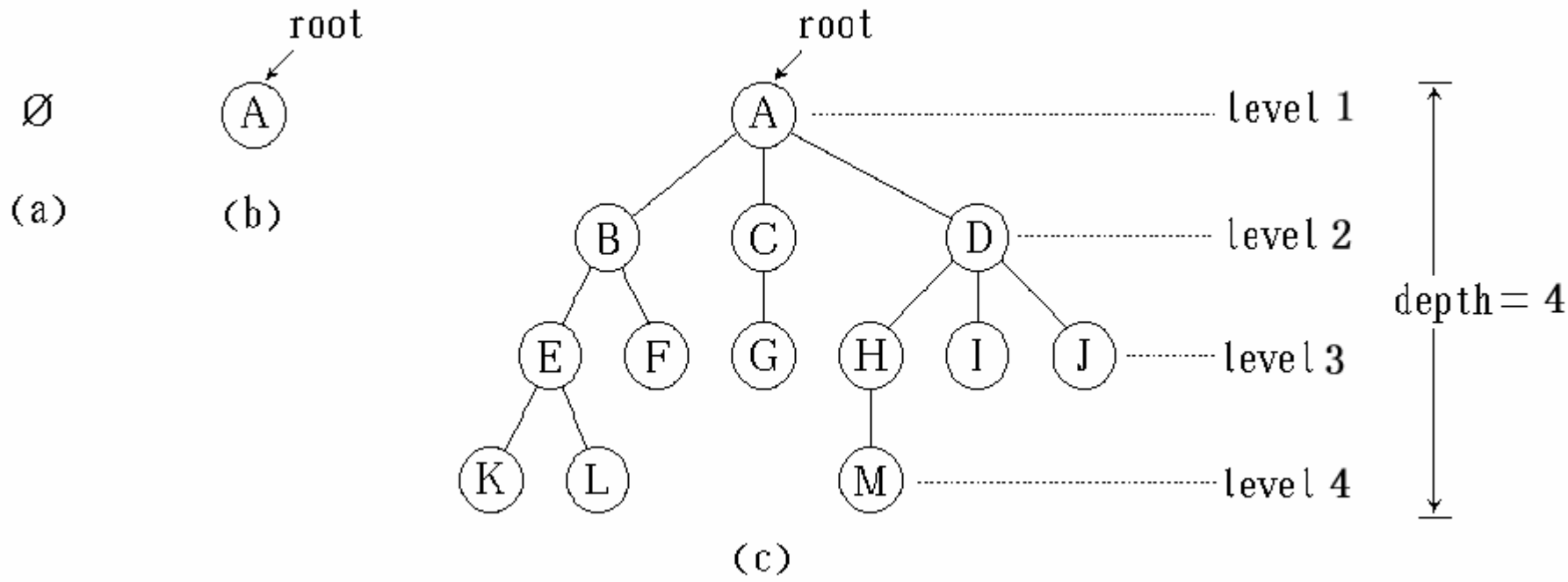
- 结点(node):包含一个数据元素及若干指向其子树的分支
- 结点的度(degree):结点所含子树的个数
- 分支(branch)结点:度不为0的结点
- 叶(leaf)结点:也叫终端结点，是度为 0 的结点
- 孩子(child)结点:结点的子树的根称为该结点的孩子
- 双亲(parent)结点: B 结点是A 结点的孩子，则A结点是B 结点的双亲



- **兄弟(sibling)结点:**同一双亲的孩子结点
- **堂兄弟结点:**双亲互为兄弟的结点
- **祖先(ancestor)结点:**从根到该结点的所经分支上的所有结点
- **子孙(descendant)结点:**以某结点为根的子树中任一结点都称为该结点的子孙
- **结点所处层次(level):**根结点的层定义为1；根的孩子为第二层结点，依此类推



- **树的深度(depth):** 树中最大的结点层
- **树的度(degree):** 树中最大的结点度。
- 路径：树中的一个结点序列，并且前一个是后一个的双亲。
- 路径的长度：该路径所经过的边的数目



- **有序树:** 子树有序的树，如：家族树；
- **无序树:** 不考虑子树的顺序；
- **森林:m(m>=0))**棵互不相交的树集合；森林和树之间的联系是：一棵树去掉根，其子树构成一个森林；一个森林增加一个根结点成为树。

树的抽象数据类型

ADT Tree {
  数据对象：
  数据关系：
  基本操作：**p119**
} ADT Tree

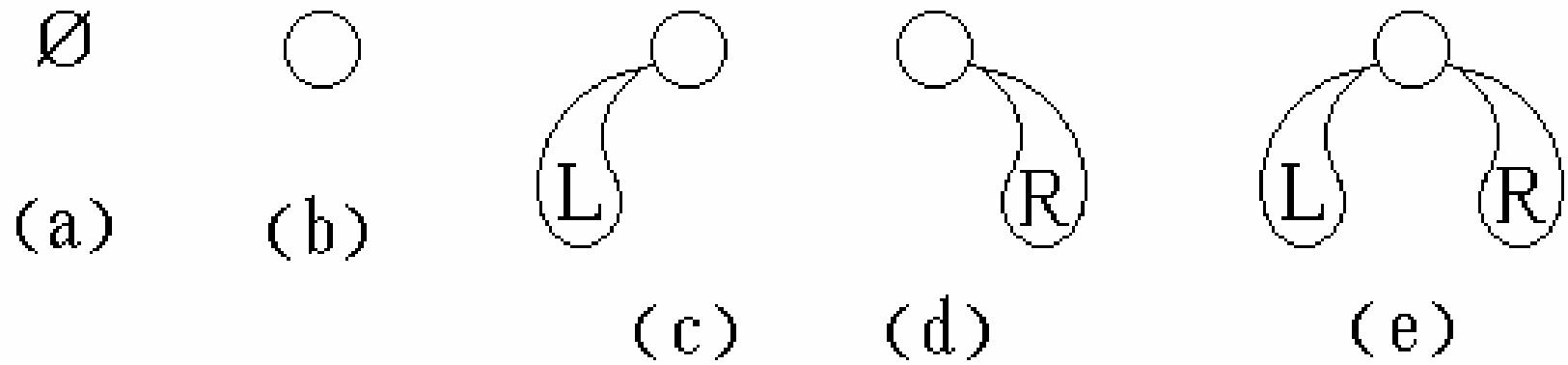
基本操作中树的建立和遍历——重点

6.2 二叉树（Binary Tree）

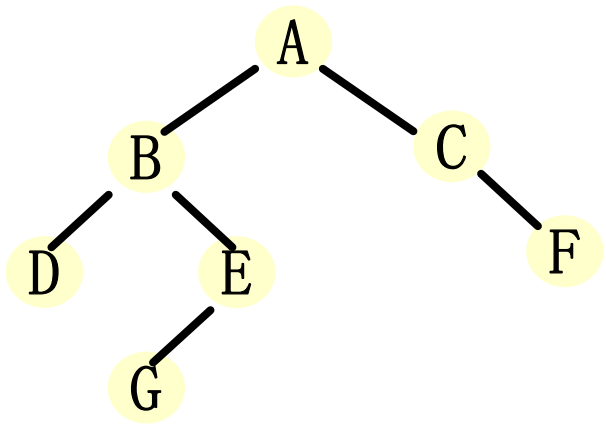
二叉树的定义

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。

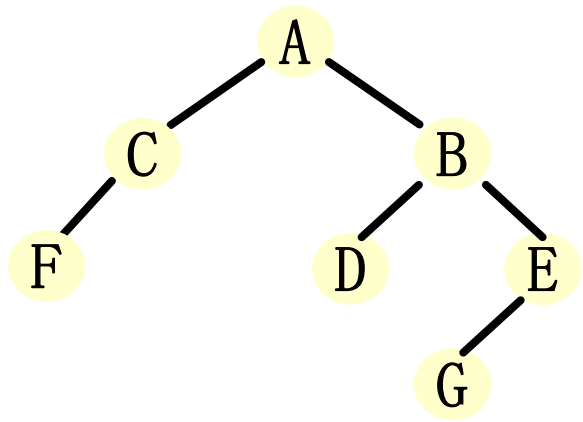
特点：1）每个结点的度≤2；2）是有序树



二叉树的五种不同形态



(a)



(b)

(a)、(b)是不同的二叉树，　(a)的左子树有四个结点，(b)的左子树有两个结点

二叉树的抽象数据类型

- 基本操作：p121～p123
- 二叉树的建立和遍历

二叉树的基本操作：

InitBiTree (&T)

初始条件：二叉树T不存在。

操作结果：构造一棵空的二叉树。

DestroyBiTree (&T)

初始条件：二叉树T存在。

操作结果：删除二叉树T 。

CreateBiTree (&T, definition)

初始条件： definition给出二叉树T的定义。

操作结果：按definition给出的定义构造二叉树T 。

BiTtreeEmpty (T)

初始条件：二叉树T存在。

操作结果：若二叉树T为空，则返回True，否则返回

False 。

Parent (T,e)

初始条件：二叉树T存在，e是T中某结点。

操作结果：若e是T的非根结点，则返回它的双亲，否则返回“空”。

LeftChild (T, e)

初始条件：二叉树T存在，e是T中某结点。

操作结果：返回e的左孩子，若e无左孩子，则返回“空”。

RightChild (T, e)

初始条件：二叉树T存在，e是T中某结点。

操作结果：返回e的左孩子，若e无左孩子，

则返回“空”。

LeftsSibling (T, e)

初始条件：二叉树T存在，e是T中某结点。

操作结果：返回e的左兄弟，若e是其双亲的左孩子或无左兄弟，则返回“空”。

RightSibling (T, e)

初始条件：二叉树T存在，e是T中某结点。

操作结果：返回e的右兄弟，若e是其双亲的右孩子或无右兄弟，则返回“空”。

Traverse (T)

初始条件：二叉树T存在。

操作结果：按照某条搜索路径遍历T，对每个结点进行一次且仅一次的访问。

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。($i \geq 1$)

[证明用数学归纳法]

证明：当i=1时，只有一个根结点，$2^{i-1}=2^0=1$，命题成立。

现在假定对所有的j，1<=j<=i，命题成立，即第j层上至多有2^{j-1}个结点。由于二叉树每个结点的度最大为2，故在第j+1层上最大结点数为第j层上最大结点数的二倍，

即$2\times 2^{j-1}=2^{(j+1)-1}$。

由j的任意性可知，二叉树的第i层上至多有2^{i-1}个结点，命题得到证明。

二叉树的性质

性质2 深度为k的二叉树最多有 2^k-1个结点。

($k \geq 1$)

[证明用求等比级数前k项和的公式]

证明：深度为k的二叉树上最大的结点数为二叉树中每层上的最大结点数之和，由性质1得到每层上的最大结点数：

$$\sum_{i=1}^{k}(\text{第}i\text{层上的最大结点数})$$

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

二叉树的性质

性质3 对任何一棵二叉树，如果其叶结点个数为 n_0，度为2的非叶结点个数为 n_2，则有

$$n_0 = n_2 + 1$$

证明：若设度为1的结点有n_1个，度为2的结点有n_2个总结点个数为n，总边数为e，则根据二叉树的定义，

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

$$e = 2n_2 + n_1 \quad (2)$$

$$n = e + 1 \quad (3)$$

由(1) (2) (3)可得：$n_0 = n_2 + 1$

同理：

三次树： $n_0=1+n_2+2n_3$

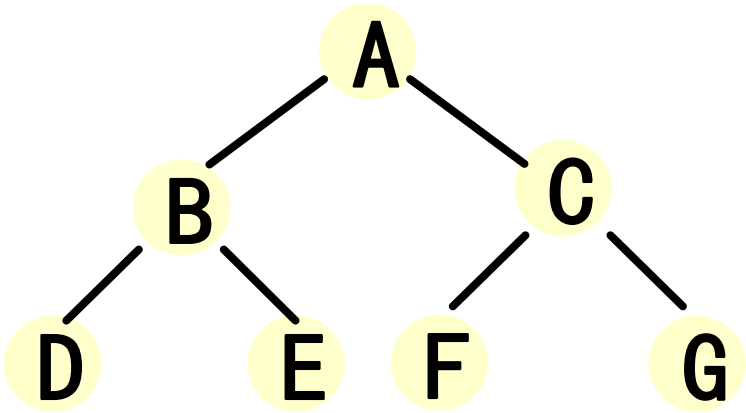
四次树： $n_0=1+n_2+2n_3+3n_4$

…

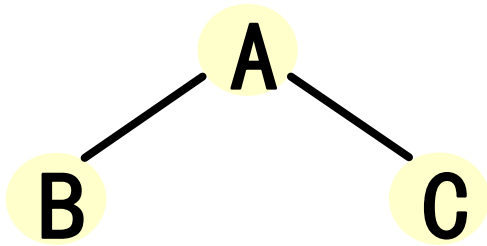
K次树： $n_0=1+n_2+2n_3+\cdots+(k-1)n_k$

定义1　满二叉树(Full Binary Tree)

一棵深度为k且有2^k-1个结点的二叉树称为满二叉树。



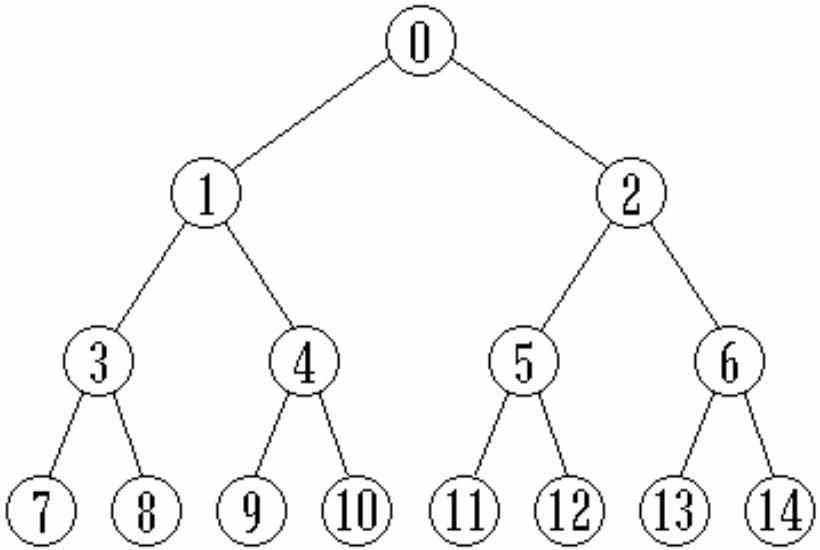
K=3的满二叉树



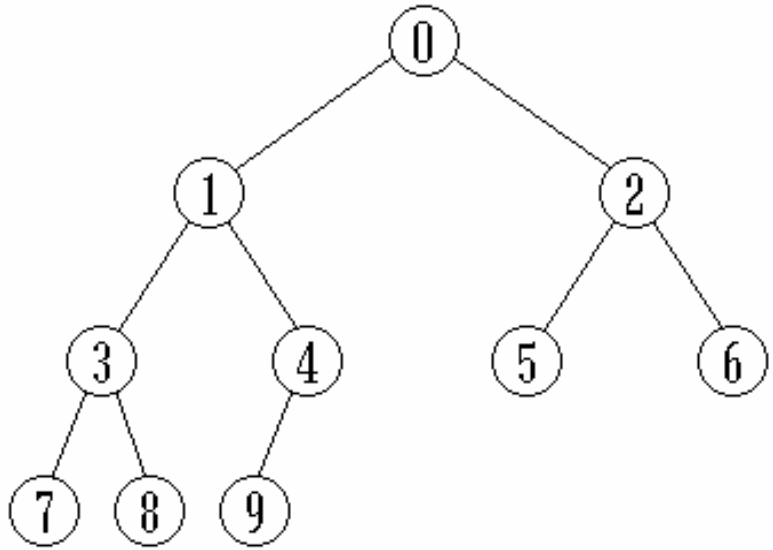
K=2的满二叉树

定义2　完全二叉树(Complete Binary Tree)

若设二叉树的深度为h，则共有h层。除第h层外，其它各层(0~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。



（a） 满二叉树



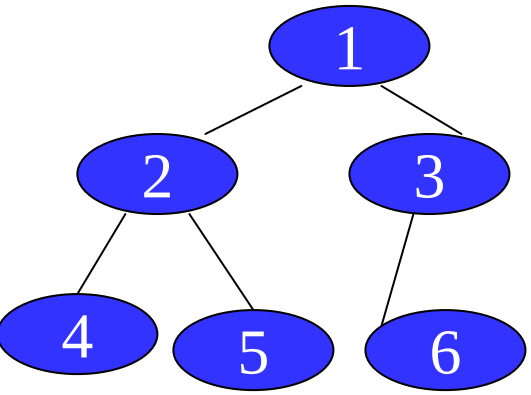
（b）完全二叉树

完全二叉树的特点是：

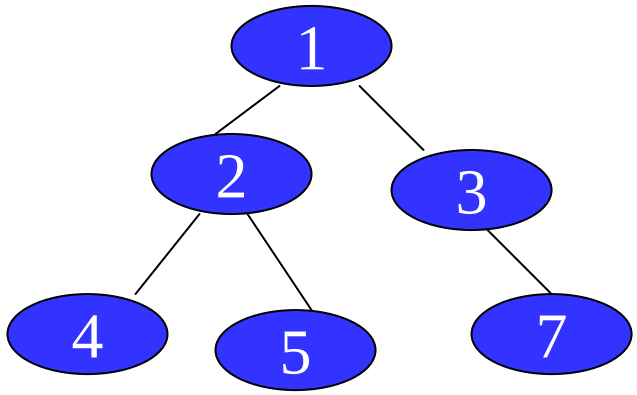
(1) 所有的叶结点都出现在第k层或k－1层。

(2) 任一结点，如果其右子树的最大层次为1，则其左子树的最大层次为1或1＋1。

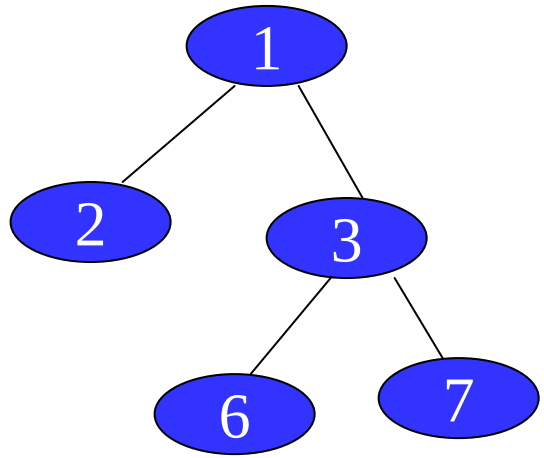
如果深度为k、由n个结点的二叉树中每个结点能够与深度为k的顺序编号的满二叉树从1到n标号的结点相对应，则称这样的二叉树为完全二叉树，满二叉树是完全二叉树的特例。



(a)完全二叉树



(b)非完全二叉树



(c)非完全二叉树

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

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

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

$$\Rightarrow \quad \text{取对数 } k-1 \leq \log_2 n < k$$

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

说明：常出现在选择题中

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：

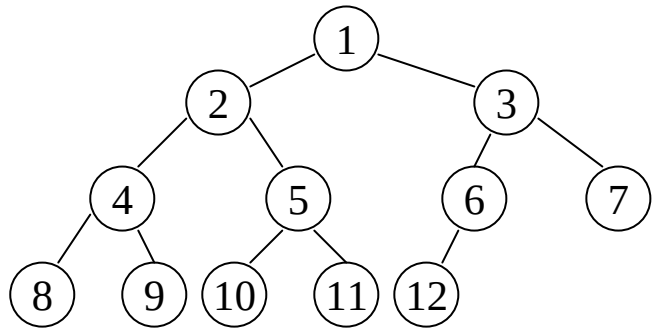
- 若i == 1，则 i 是二叉树的根，无双亲
  若i > 1，则 i 的双亲为⌊i /2⌋
- 若2*i ⩽ n，则 i 的左孩子为2*i，否则无左孩子
  若2*i+1 ⩽ n，则 i 的右孩子为2*i+1，否则无右孩子
- 若 i 为偶数，且i != n，则其右兄弟为i+1
  若 i 为奇数，且i != 1，则其左兄弟为i-1
- i 所在层次为 $\lfloor \log_2 i \rfloor + 1$

性质5：如果对一棵有n个结点的完全二叉树的结点按层序编号(从第1层到第L$\log_2 n$ 」+1层，每层从左到右),则对任一结点i(1<=i<=n),有：

（1）如果i＝1，则结点i无双亲，是二叉树的根；如果i>1，则其双亲是结点L i/2 」。

（2）如果2i>n，则结点i为叶子结点，无左孩子；否则，其左孩子是结点2i。

（3）如果2i＋1>n，则结点i无右孩子；否则，其右孩子是结点2i＋1。



完全二叉树

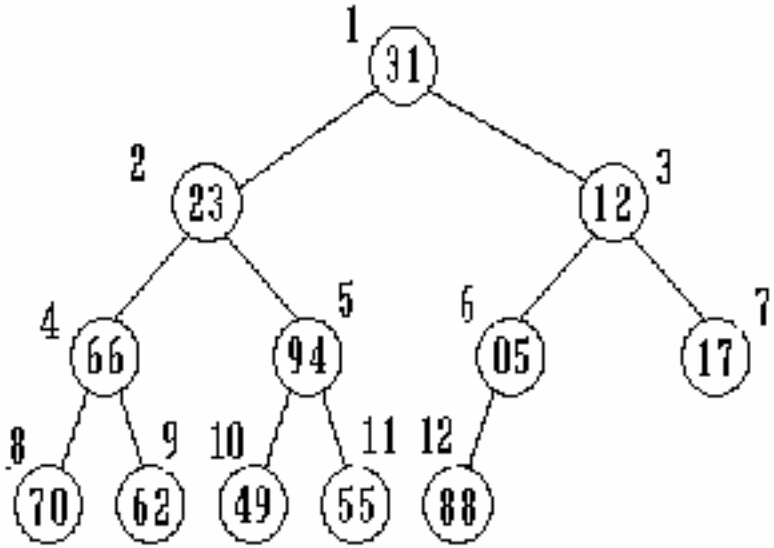
例如：i=4，**2i<n**，故其左孩子是结点**8**（**2i**）；**2i+1<n**，故其右孩子是结点**9**（**2i+1**）。**i=6**， **2i=n**，故其左孩子是结点**12**（**2i**）；但 **2i+1>n**，故其没有右孩子。 **i=7**， **2i>n**，故其没有左孩子； **2i+1>n**，故其没有右孩子。

二叉树的存储结构

- 顺序存储结构
- 链式存储结构
- 静态二叉链表和静态三叉链表

二叉树的存储结构

1. 顺序存储结构（数组表示）

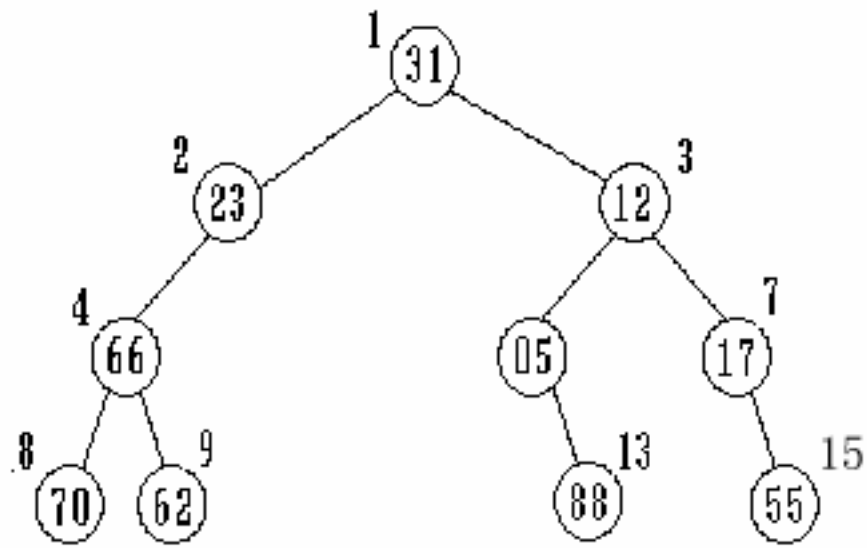


(a)

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)

完全二叉树的数组表示



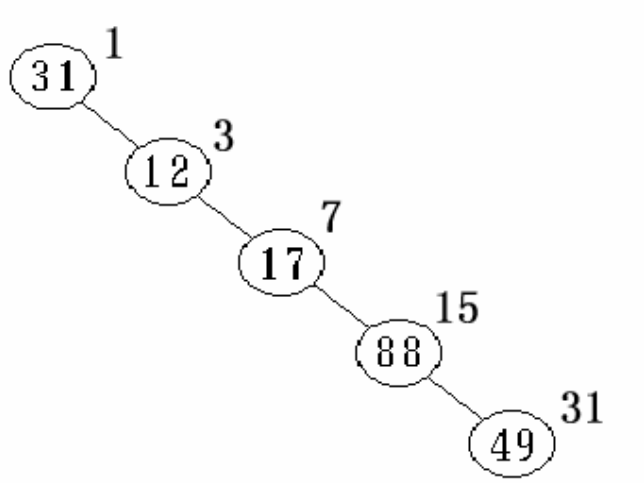
(a)

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)

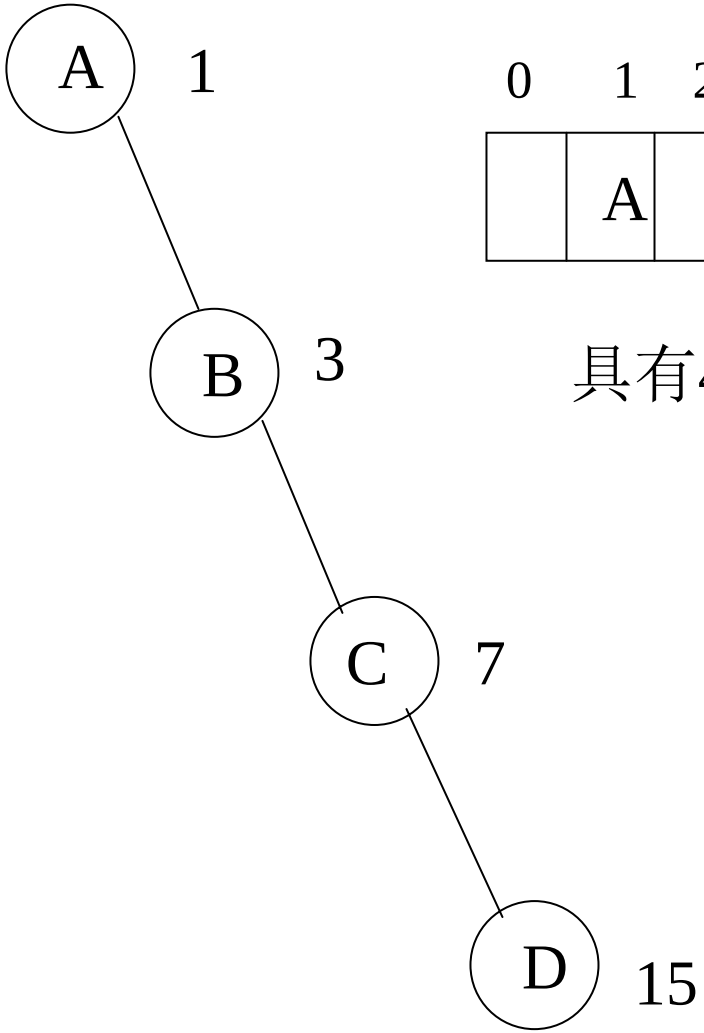
一般二叉树的数组表示

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，一般二叉树必须仿照完全二叉树那样存储，这会造成存储空间的浪费。单支树就是一个极端情况。



单支树

```
#define MAX_TREE_SIZE   100   // 最大结点数
typedef TElemType SqBiTree[MAX_TREE_SIZE];
SqBiTree   bt;
```

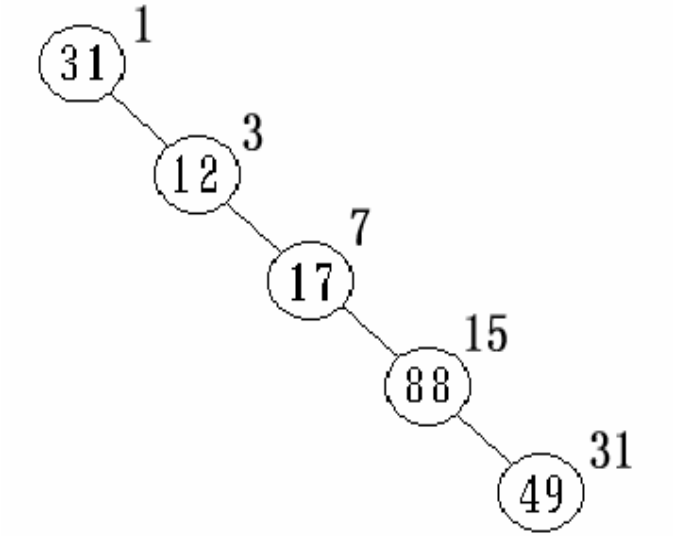


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	A		B				C								D

具有4个结点的单右支树的顺序存储

具有4个结点的单右支树

大家可以写出具有5个结点的
单右支树的顺序存储

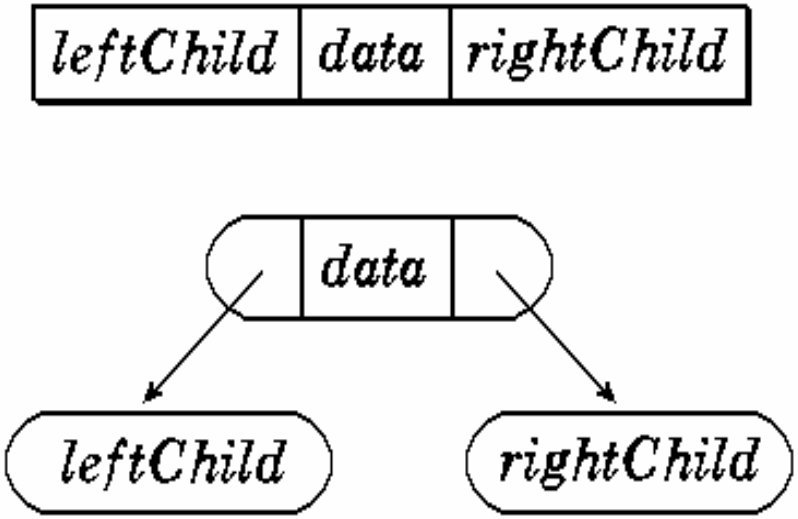



```
#define MAX_TREE_SIZE  100
typedef TElemType SqBiTree[MAX_TREE_SIZE];
SqBiTree  bt;
```

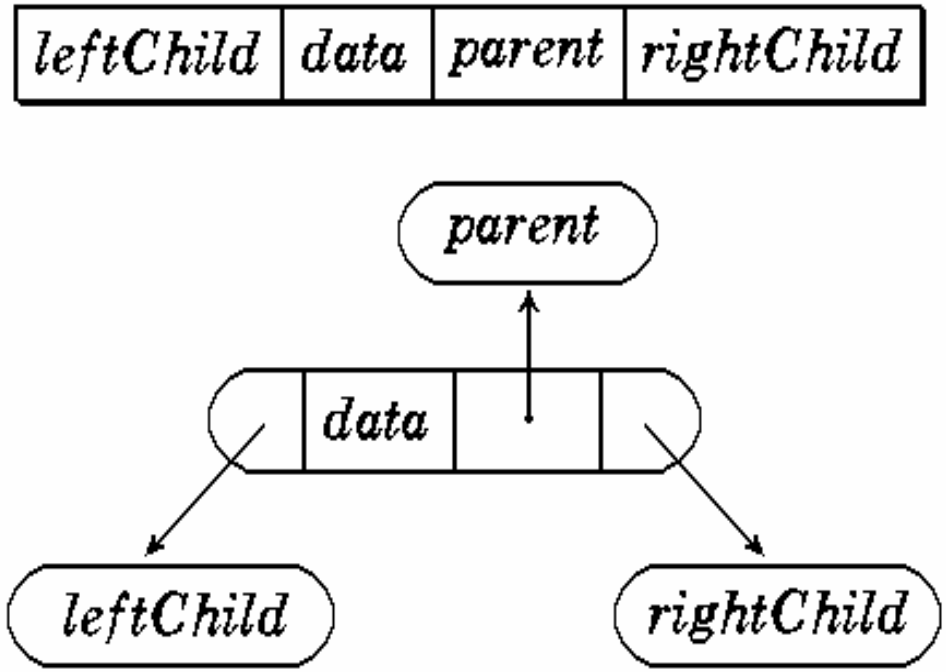
问题

1. 在最坏的情况下，一个深度为k，具有k个结点的单右支树要占2^k个存储空间。浪费存储空间。
2. 若经常需要插入与删除树中结点时，顺序存储方式不是很好！

2. 链式存储结构

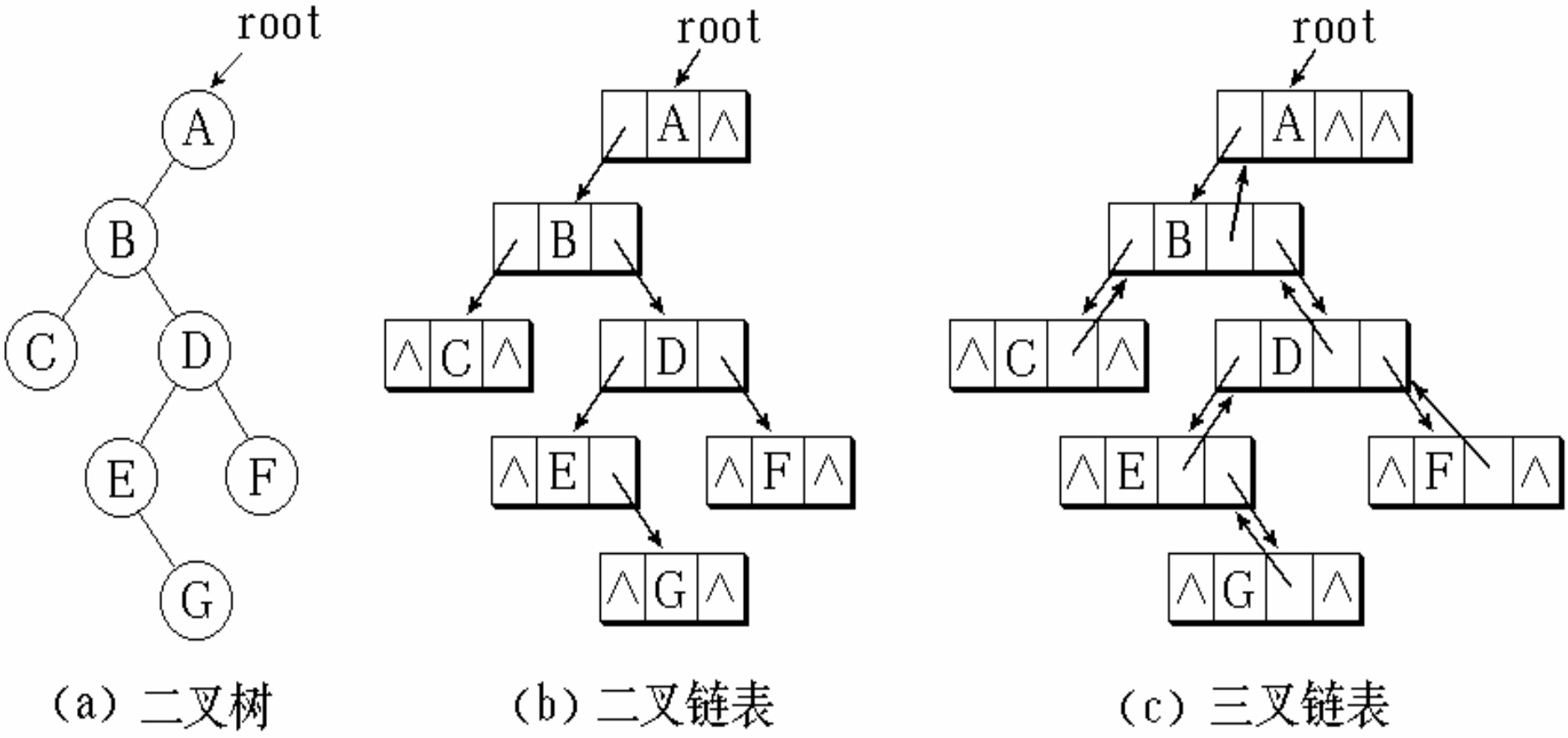


(a) 二叉链表



(b) 三叉链表

```
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

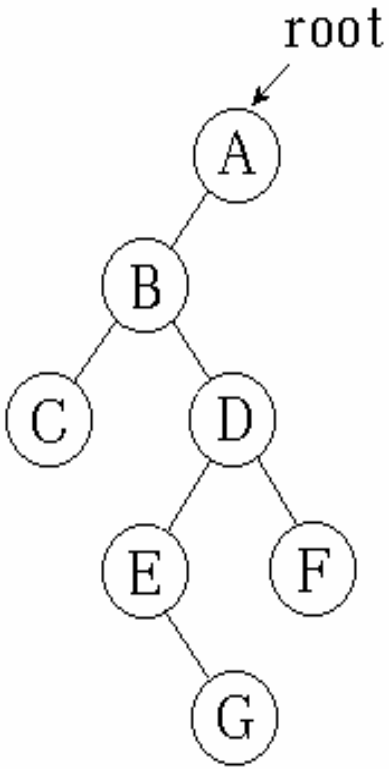
若二叉树为空，则root=NULL。

若二叉树有n个结点，则该二叉链表共有2n个指针域，其中n-1个指向结点的左右孩子，其余的n+1个指针域为空。空链域数仅稍多于总链域数的1/2。

存储效率较高。

可以将一般的树转换为二叉树，这样可以较好地进行存储和运算

3. 静态二叉链表和静态三叉链表



(a) 二叉树

	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

预先开辟空间，用数组表示

leftChild, rightChild——数组元素的下标

6.3 遍历二叉树

(Traversing Binary Tree) p128

所谓树的遍历，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。

如何遍历呢？

6.3 遍历二叉树

(Traversing Binary Tree) p128

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

中序 LDR

后序 LRD

6.3 遍历二叉树
(Traversing Binary Tree) p128

可能的遍历次序还有:

DRL　　逆先序

RDL　　逆中序

RLD　　逆后序

先序遍历（Preorder Traversal）

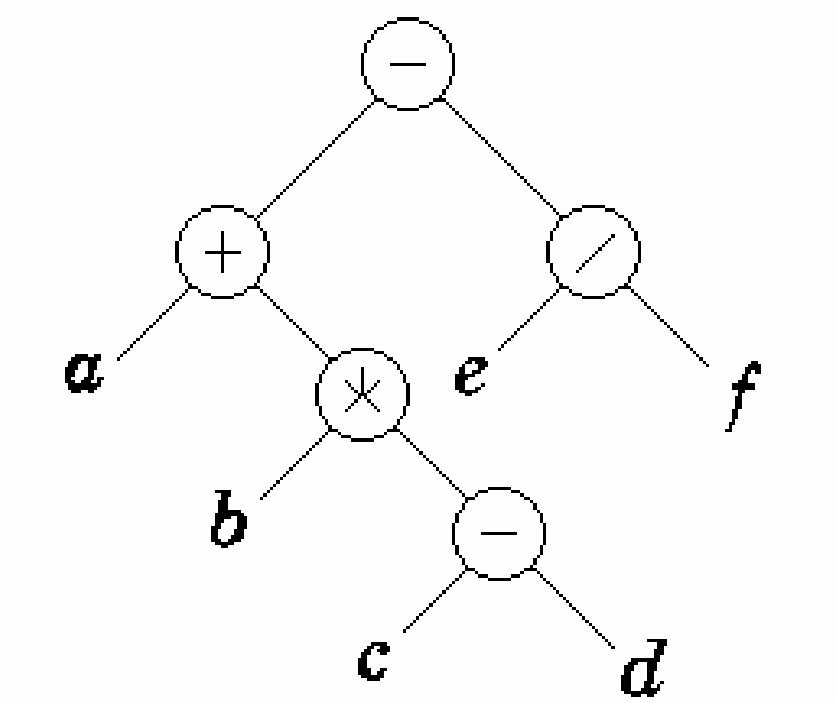
先序遍历二叉树算法的框架是：

- 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)。

遍历结果（前缀表达式）

- + a * b - c d / e f

在波兰式中，自左到右依次扫描：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。



先序遍历二叉树的递归算法

```
void PreOrderTraverse(BiTree T)
 {
    if (T){
         printf("%c",T->data);
         PreOrderTraverse(T->lchild);
         PreOrderTraverse(T->rchild);
           }
  }
```

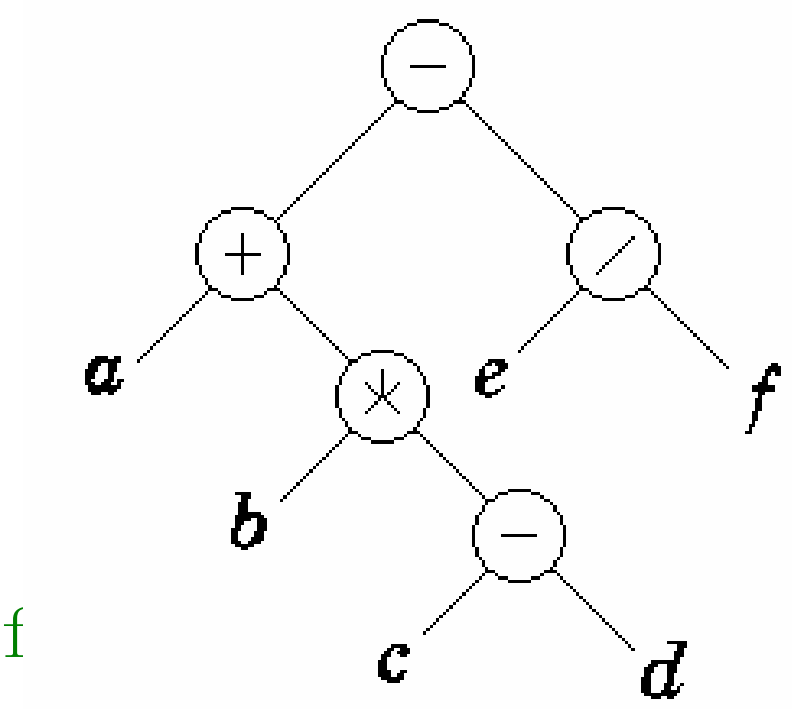
中序遍历（Inorder Traversal）

中序遍历二叉树算法的框架是：

- 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）。

遍历结果（中缀表达式）

a + b * c - d - e / f



表达式语法树

中序遍历二叉树的递归算法

```
void InOrderTraverse(BiTree T)
  {
     if (T) {
          InOrderTraverse(T->lchild);
          printf("%c",T->data);
          InOrderTraverse(T->rchild);
             }
   }
```

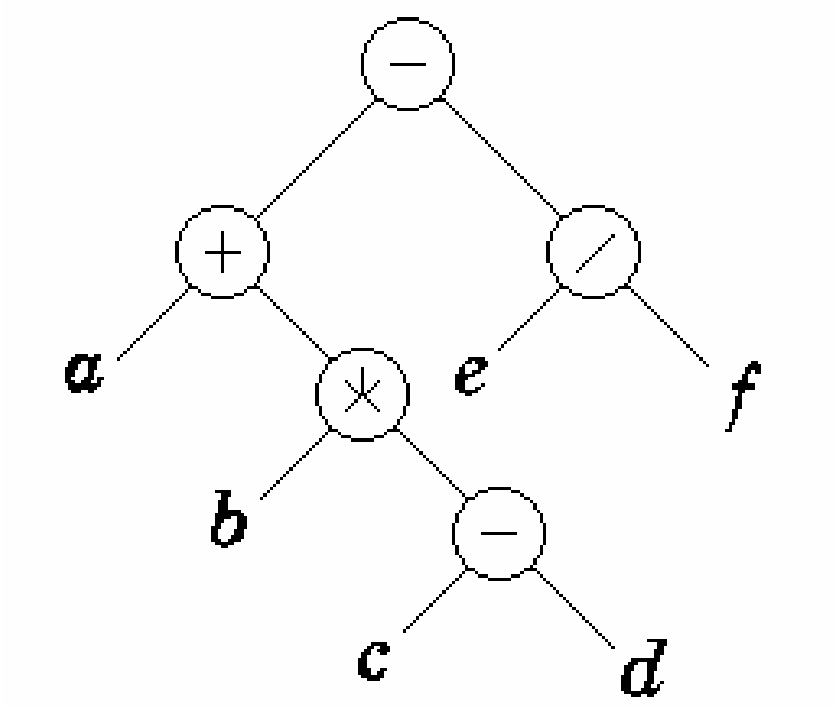
后序遍历（Postorder Traversal）

后序遍历二叉树算法的框架是：

- 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）。

遍历结果（后缀表达式）

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。

后序遍历二叉树的递归算法

```
void PostOrderTraverse(BiTree T)
{
    if (T) {
        PostOrderTraverse(T->lchild);
        PostOrderTraverse(T->rchild);
        printf("%c",T->data);
            }
 }
```

遍历二叉树的非递归算法

- 先序遍历：

  算法1，将右子树根结点入栈，（栈所需最大容量为n/2+1）；

  算法2，将根结点入栈

- 中序遍历：在遍历左子树之前，先把根结点入栈，当左子树遍历结束后，从栈中弹出，访问，再遍历右子树

- 后序遍历：

  1）设定一个指针，指向最近访问过的结点。在退栈取出根结点时，需判断：若根结点的右子树为空，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，应该遍历该根结点。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

需用到栈，顺序栈的定义如下：

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

- 先序遍历：
  - 算法1，将右子树根结点 入栈，(栈所需最大容量为n/2+1)；
  - 算法2将根结点入栈

先序遍历的非递归算法

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

先序遍历（T L R）的非递归算法2

对每个结点，在访问完后，沿其左链一直访问下来，直到左链为空，这时，所有已被访问过的结点进栈。然后结点出栈，对于每个出栈结点，即表示该结点和其左子树已被访问结束，应该访问该结点的右子树。

（1）当前指针指向根结点。

（2）打印当前结点，当前指针指向其左孩子并进栈，重复（2），直到左孩子为NULL

（3）依次退栈，将当前指针指向右孩子

（4）若栈非空或当前指针非NULL，执行（2）；否则结束。

先序遍历的非递归算法2

```
void preorder(BiTree T){
     int top=0;
     BiTree stack[20], P=T;
     do { while(P){
             printf("%c",P->data);
             stack[top]=P; top++;
             P=P->lchild;}
          if (top){
             top--; P=stack[top];
             P=P->rchild;}
        }while (top || P);
      }
```

中序遍历（ L T R）的非递归算法

对每个结点，先入栈，然后沿其左链一直访问下来，直到左链为空，这时，所有已被访问过的结点进栈。然后结点出栈，对于每个出栈结点，打印它，然后应该访问该结点的右子树。

（1）当前指针指向根结点。

（2）当前结点进栈，当前指针指向其左孩子并进栈，重复（2），直到左孩子为NULL

（3）依次退栈并打印，将当前指针指向右孩子

（4）若栈非空或当前指针非NULL，执行（2）；否则结束。

中序遍历的非递归算法1

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

中序遍历的非递归算法2（p131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; }
      }
  }
```

后序遍历的非递归算法

比前两个算法复杂。

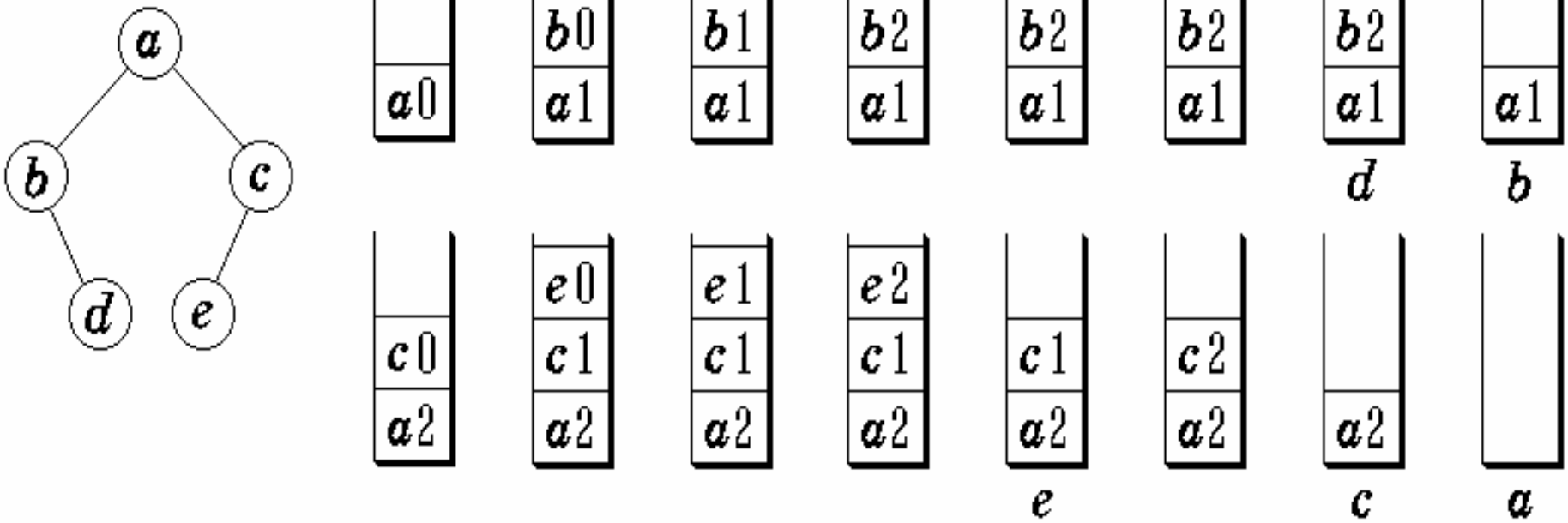
每个结点应在第三次访问时才输出。第一次访问时，让该结点入栈。然后对其左孩子进行访问，直到左链为空。第二次访问该结点时，该结点出栈，但不能输出，还需入栈，并对其右孩子进行访问。第三次访问该结点时，该结点出栈，同时输出。

- 后序遍历：

1）设定一个指针，指向 最近访问过的结点。在退栈取出根结点时，需判断：若根结点的右子树为空，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，应该遍历该根结点。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。

后序遍历时，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1*

```
void Postorder(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 (!StackEmpty(S))
                    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2*

```
void postorder(BiTree T)
{ BiTree P=T,q; int flag; SqStack S; InitStack(S);
  do {
      while (P){ *(S.top)=P;S.top++;P=P->lchild;}
      q=NULL;  flag=1;
      while ((S.top!=S.base) && flag)
            { P=*(S.top-1);
              if (P->rchild == q)
                    { printf("%c",P->data); S.top--; q=p; }
              else { P=P->rchild;   flag=0; }
            }
      }while ((S.top!=S.base));
 }
```

说明：

1）遍历的第一个和最后一个结点

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。

- 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：

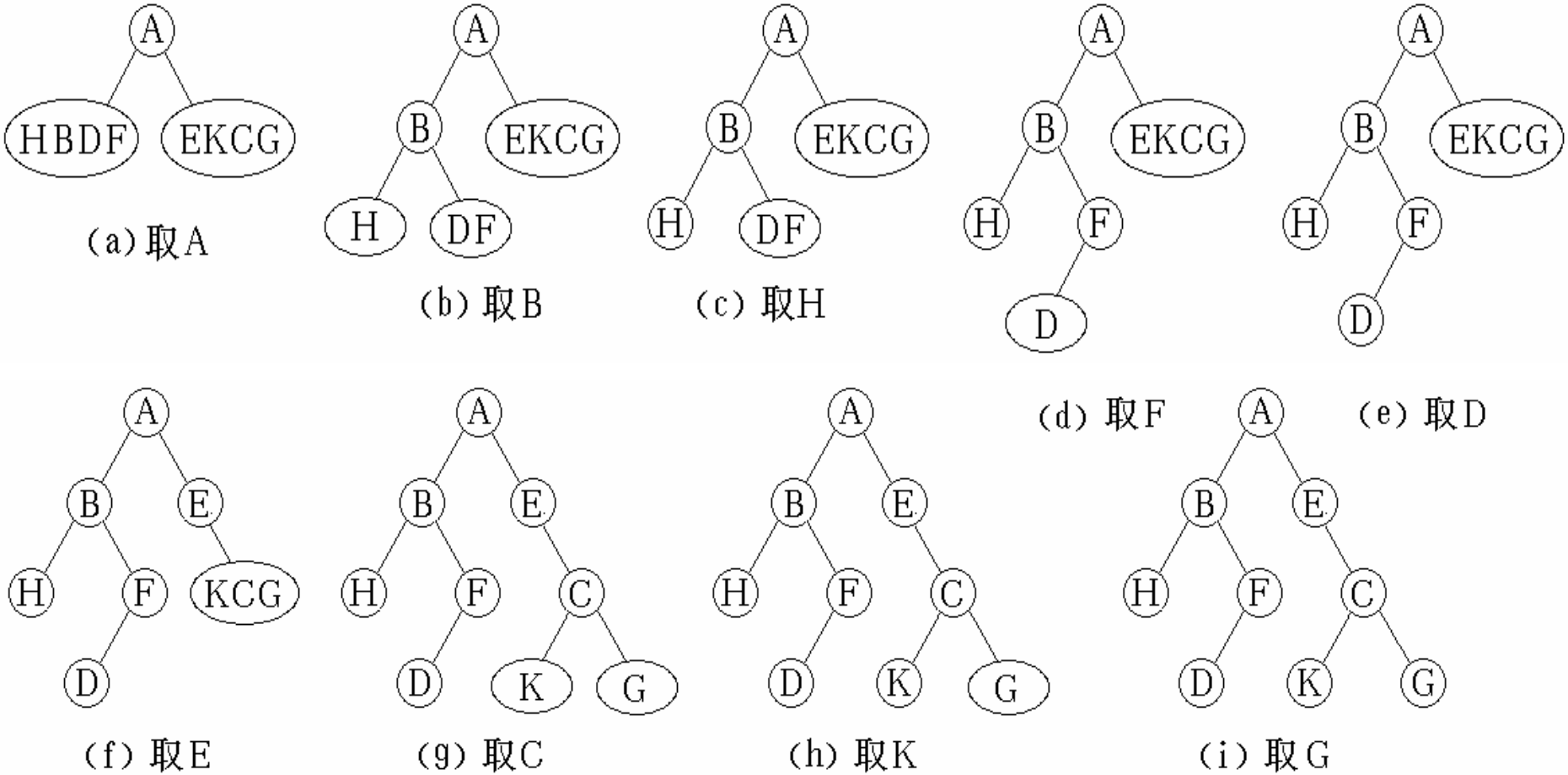
```
P=T;
while(P->rchild) P=P->rchild;
printf(P->data);
```

2）先序+中序 或中序+后序 均可唯一地确定一棵二叉树

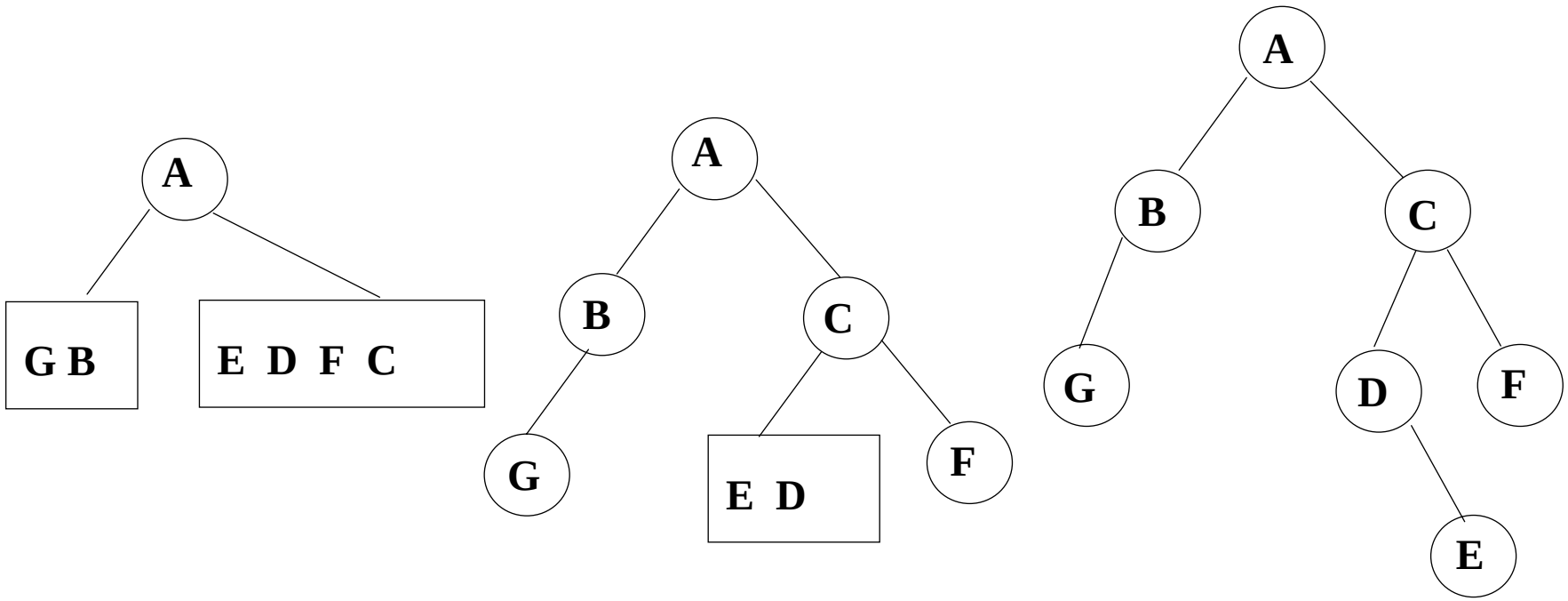
3）二叉树的二叉链表存储结构中，有____个指针域未利用，已经使用的有____个指针域，共有_____个指针域

3）二叉树的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。

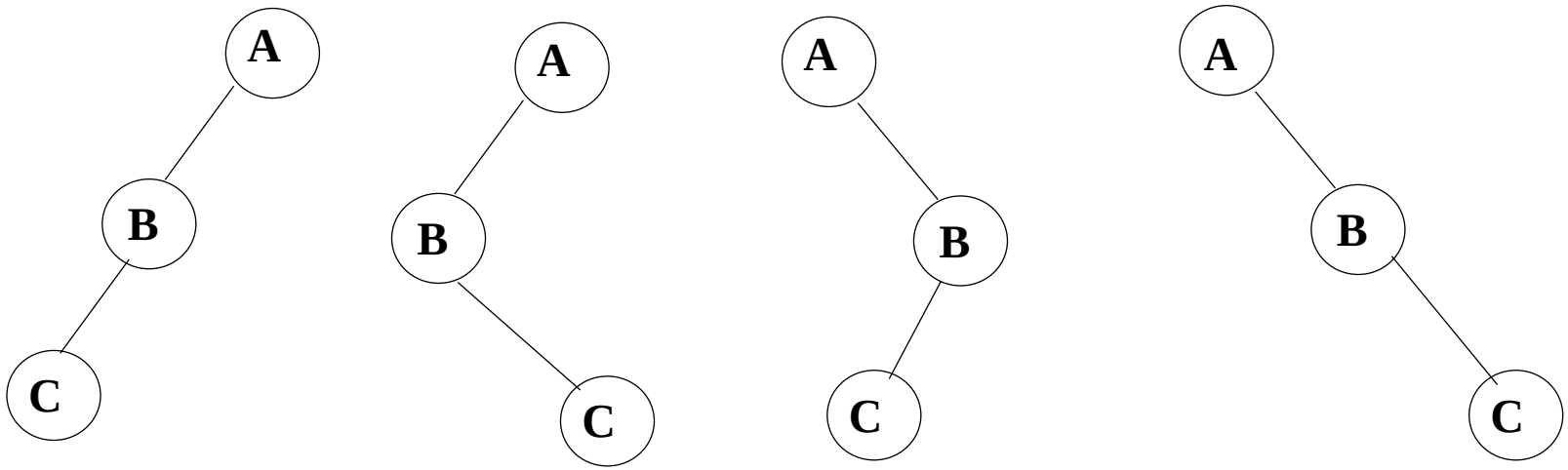
例, 先序序列 { ABHFDECKG } 和中序序列 { HBDFAEKCG }, 构造二叉树过程如下：



例，后序序列{ G B E D F C A}，中序序列{G B A D E C F}，构造二叉树过程如下：



例，先序序列{A B C}， 后序序列{C B A}，构造的二叉树不唯一。



遍历二叉树的应用

利用二叉树后序遍历计算二叉树的深度

```
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;
}
```

先序建立二叉树的递归算法(p131,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;
}
```

求二叉树结点个数

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

左右子树互换

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

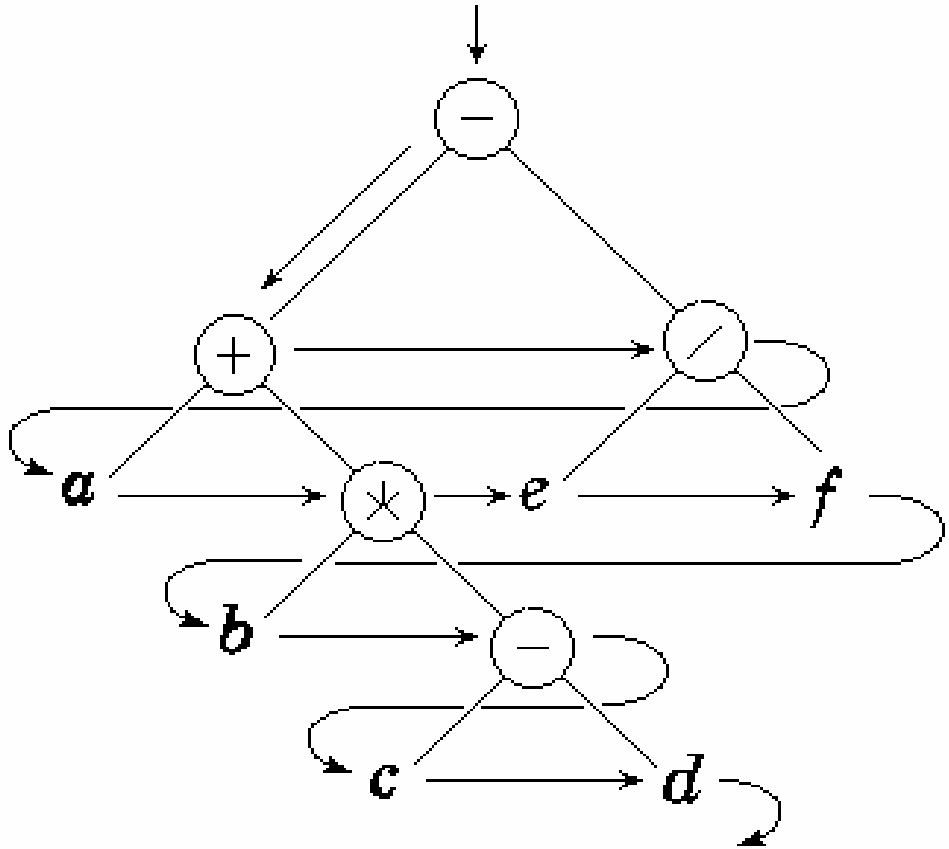
复制二叉树

```
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);
    }
}
```

按层次遍历二叉树

从根开始逐层访问，用FIFO队列实现。

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



遍历顺序

```
void LevelOrderTraverse(BiTree T)
  {  BiTree p;    SqQueue Q;    InitQueue(Q);
      if (T){ Q.base[Q.rear]=T;
              Q.rear=(Q.rear+1)%MAXQSIZE;
              while (Q.front !=Q.rear)
                { p=Q.base[Q.front];
                  printf("%c",p->data);
                  Q.front=(Q.front+1)%MAXQSIZE;
                  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;}
                  }
              }
   }
```

线索二叉树（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

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。

说明：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。

几个概念: (p132)

线索二叉树：加上线索的二叉树

线索化：对二叉树以某种次序遍历使其变为线索二叉树的过程

线索链表：以下面结点构成的二叉链表作为二叉树的存储结构，叫做线索链表

lchild	LTag	data	RTag	rchild

标志域：

LTag = 0,　lchild为左孩子指针

LTag = 1,　lchild为前驱线索

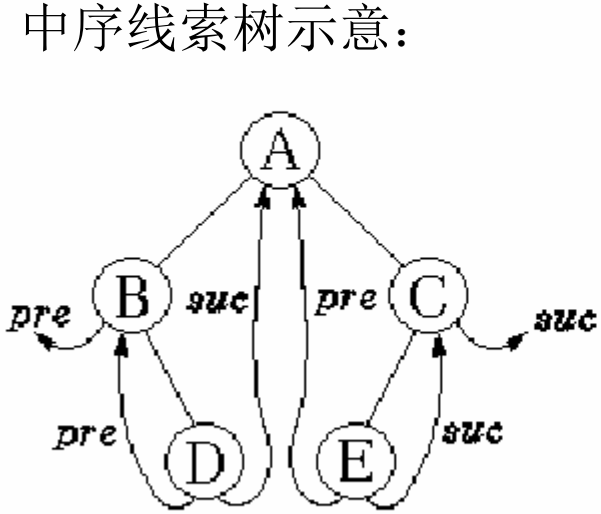
RTag = 0,　rchild为右孩子指针

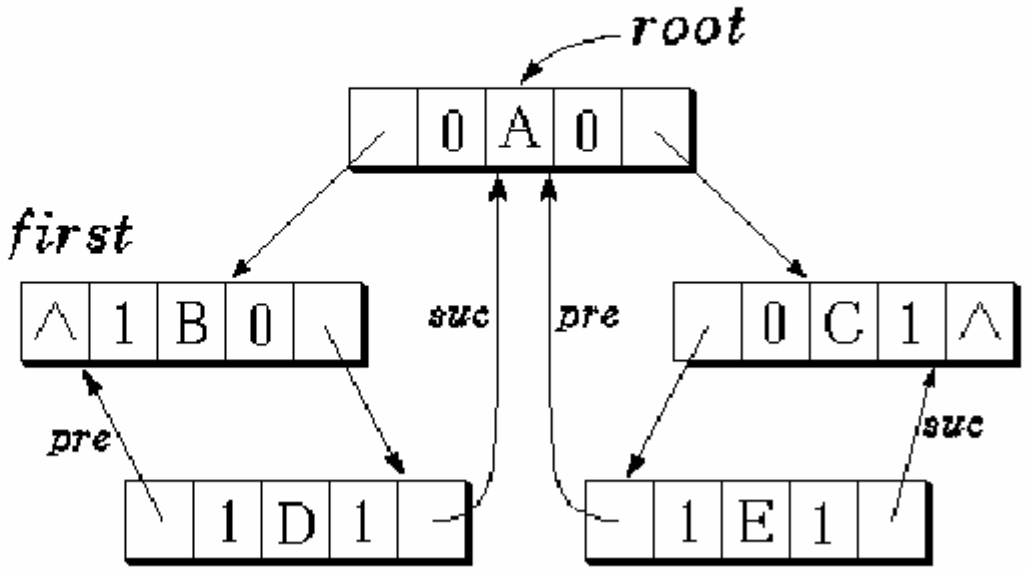
RTag = 1,　rchild为后继指针

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

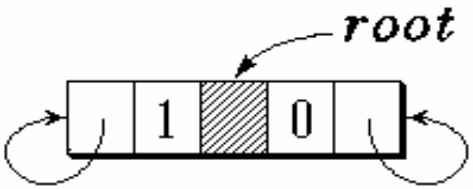
RTag = 0,　rchild为右孩子指针

RTag = 1,　rchild为后继指针

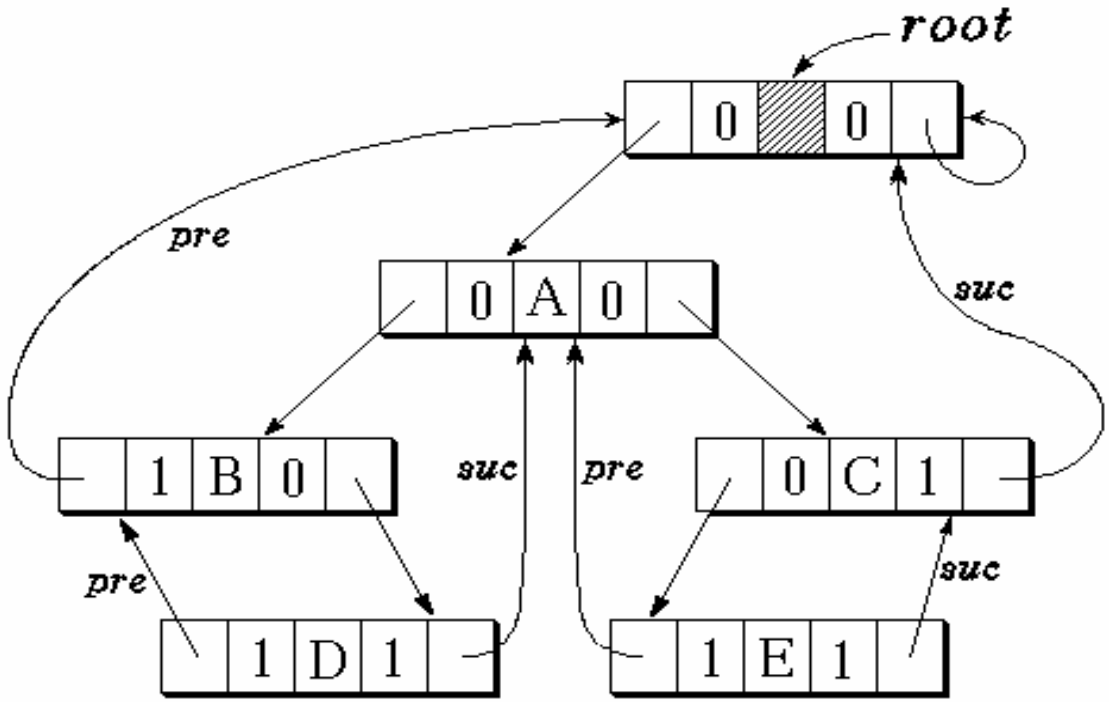
线索二叉树的存储表示　p133

```
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;
```

带表头结点的中序穿线链表



（a）空二叉链表



（b）非空二叉链表

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

- 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

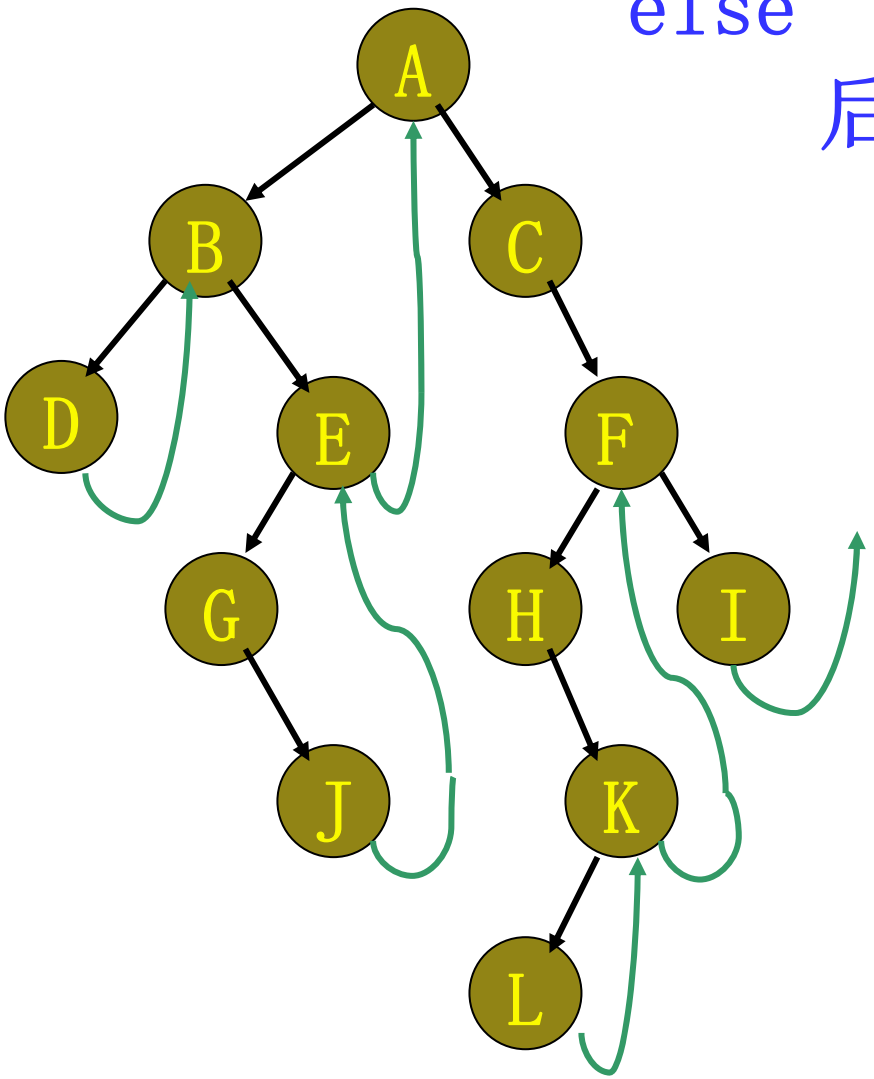
- 步骤：

  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)
    后继为 current→rchild
else  //current→RTag ==Link
    后继为当前结点右子树
        的中序下的第一个结点
```

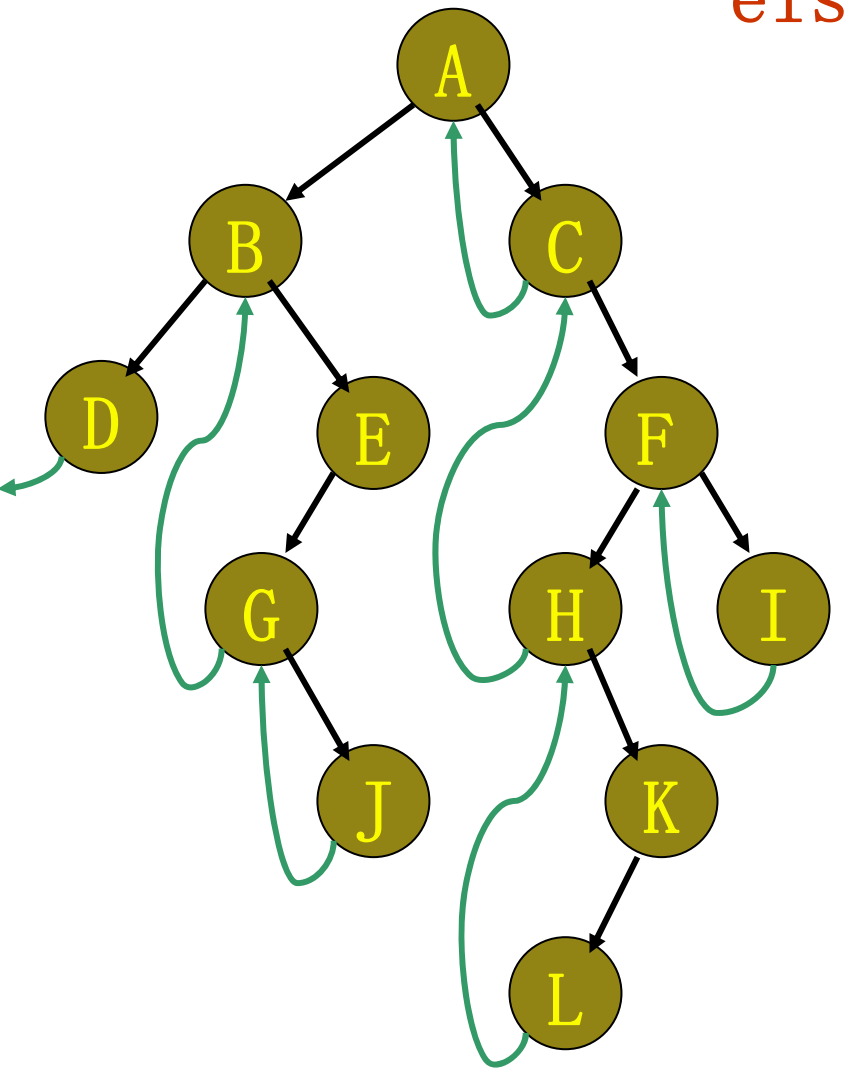


中序后继线索二叉树

DBGJEACHLKFI

寻找当前结点在中序下的前驱

```
if (current→LTag ==Thread)
        前驱为current→lchild
else //current→LTag ==Link
        前驱为当前结点左子树的
            中序下的最后一个结点
```



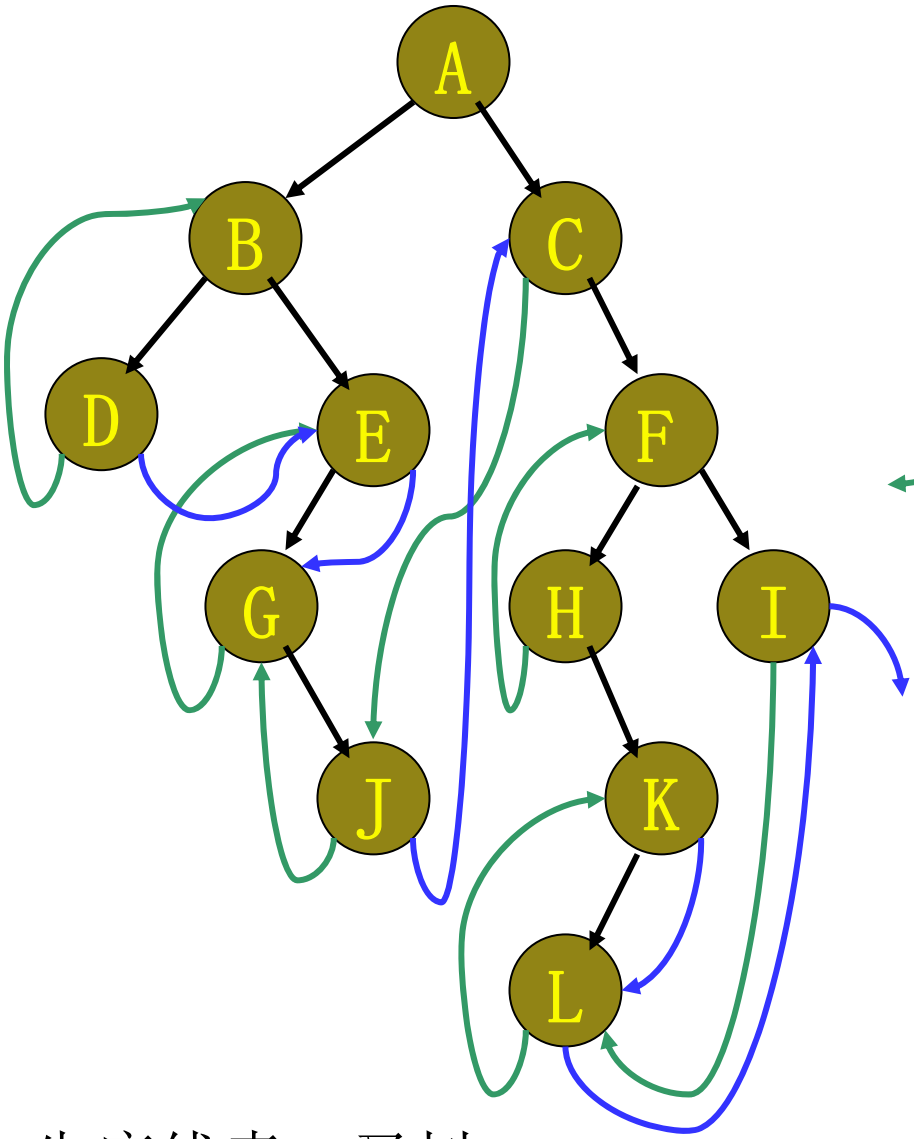
中序前驱线索二叉树

中序序列：DBGJEACHLKFI

遍历中序线索二叉树(不带头结点)

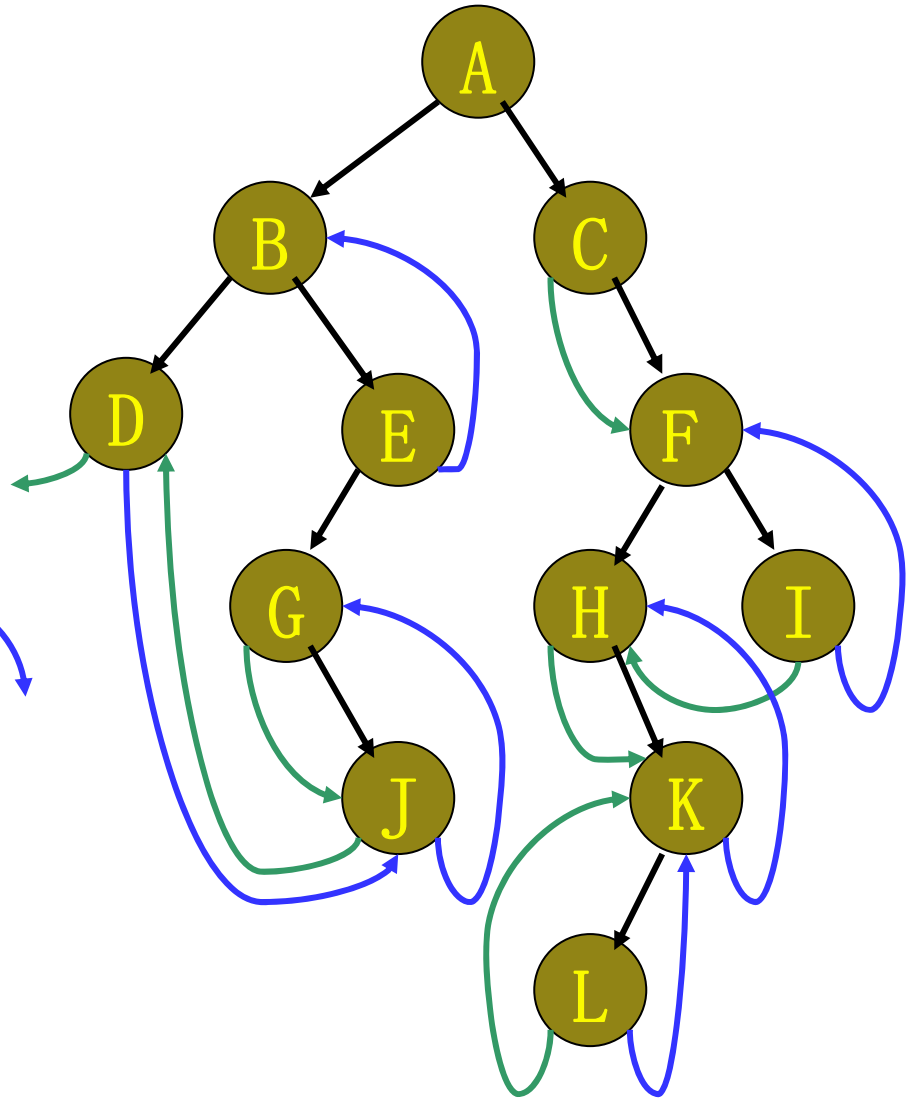
```
void inorder1_Thr(BiThrTree T)
  {  BiThrTree p=T;
     while (p->LTag==Link) p=p->lchild;
     printf("%c",p->data);
     while (p->rchild)
       { if(p->RTag==Link)
           { p=p->rchild;
             while(p->LTag==Link) p=p->lchild;}
         else p=p->rchild;
         printf("%c",p->data);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

```
void inorder2_Thr(BiThrTree T){
     BiThrTree p=T;
     while (p)
        { 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

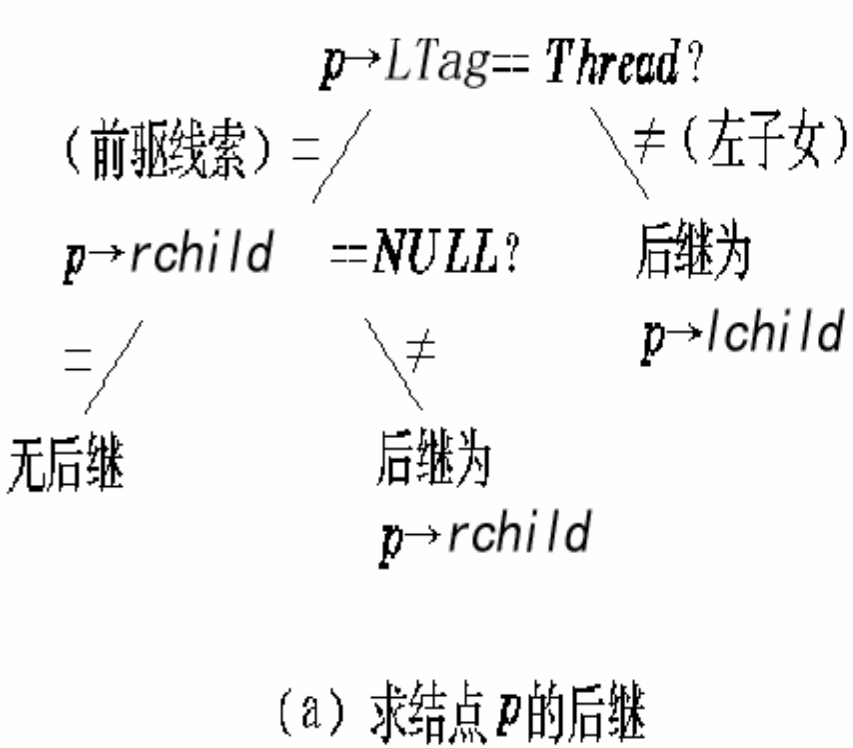
ABDEGJCFHKLI



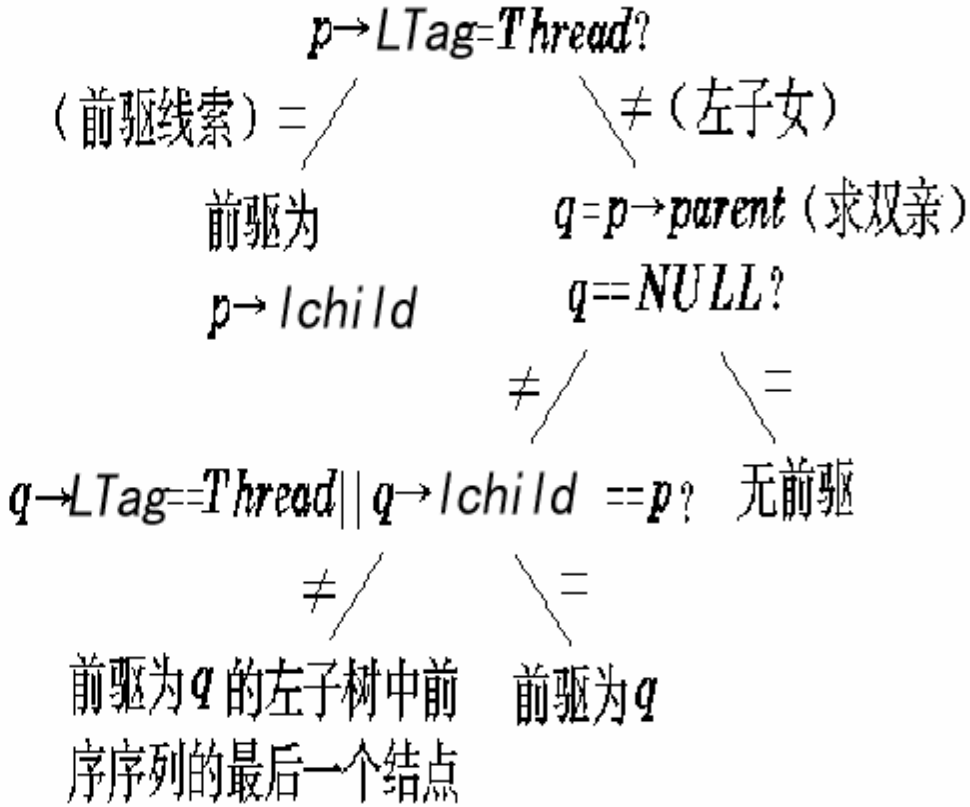
后序线索二叉树

DJGEBLKHIFCA

先序线索二叉树



(a) 求结点 p 的后继



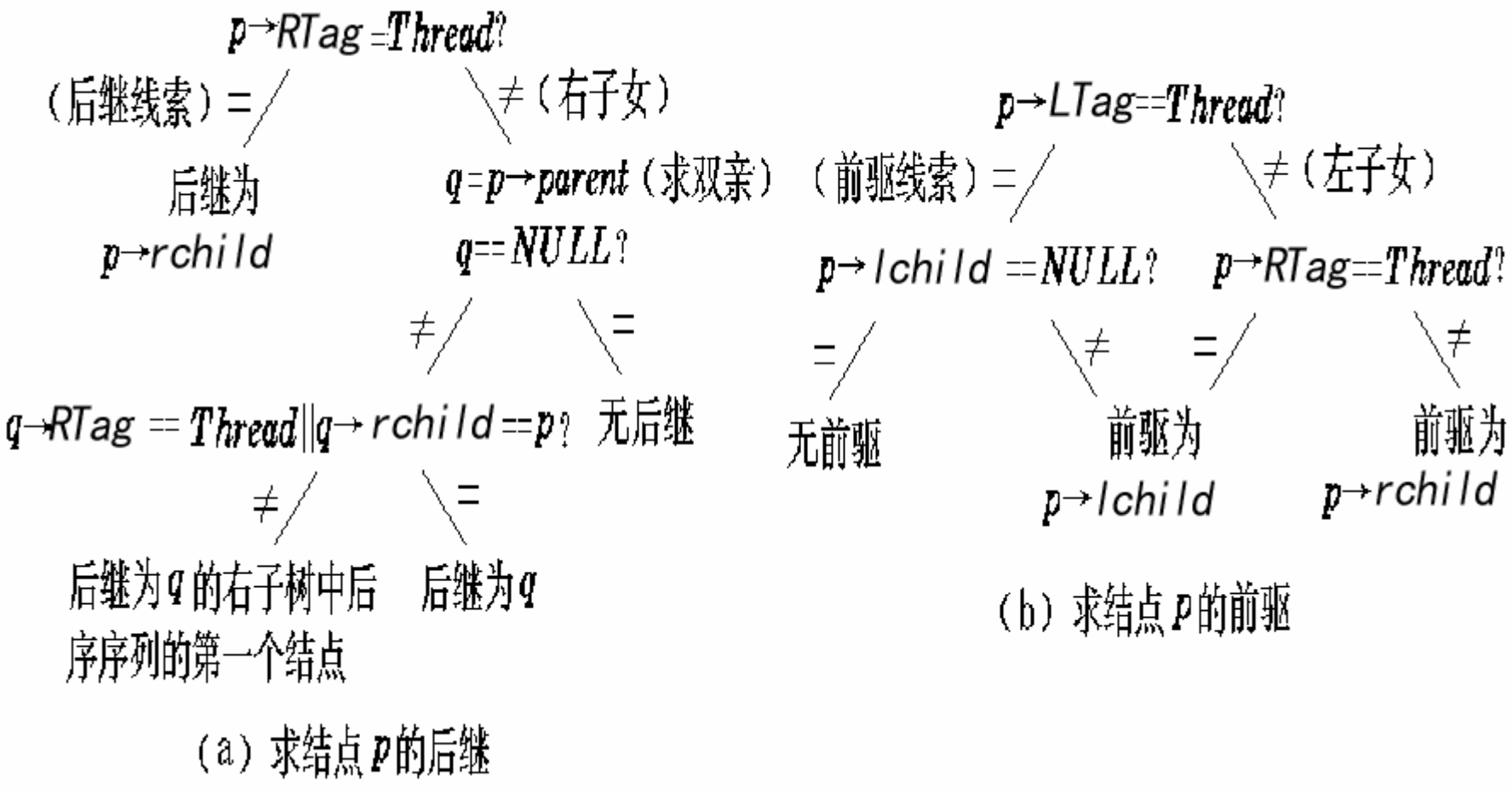
(b) 求结点 p 的前驱

在先序线索二叉树中
寻找当前结点的后继与前驱

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与前驱

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

  ——在遍历的前提下，按照先、中、后序中的一种
- 中序线索化
  - 后继线索化——处理前驱结点
    a. 如果无前驱结点，则不必加线索
    b. 如果前驱结点的右指针域为非空，也不必加线索
    c. 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。

- 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 (!P->rchild) P->RTag=Thread;
        if (pre & pre->RTag=Thread) pre->rchild=P;
        pre=P;
        InThreading(P->rchild);
      }
  }
```

- 中序线索化
  - 前驱线索化——处理后继结点

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

通过中序遍历建立中序线索化二叉树
P135算法6.7

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

P134 算法6.6 （设定一个表头结点）

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;
}
```

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;
      }
  }
```

6.4 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

◆不定长结构 d为结点的度，节省空间，但算法复杂

◆一般采用定长结构

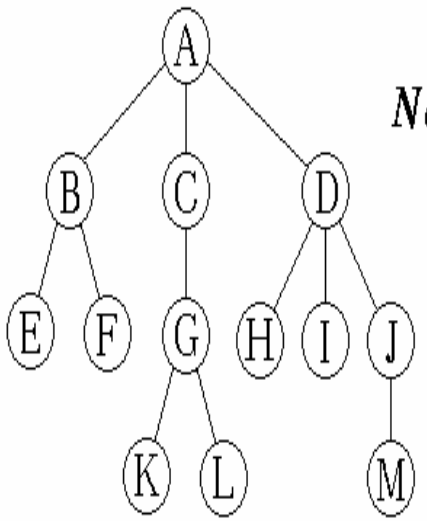
➢如有n个结点，树的度为k，则共有n*k个指针域，只有n-1个指针域被利用，而未利用的指针域为：n*k-(n-1) =n(k-1)+1，未利用率为：（ n(k-1)+1)/nk ＞ n(k-1)/nk=(k-1)/k

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。

2)常用的其他几种存储结构

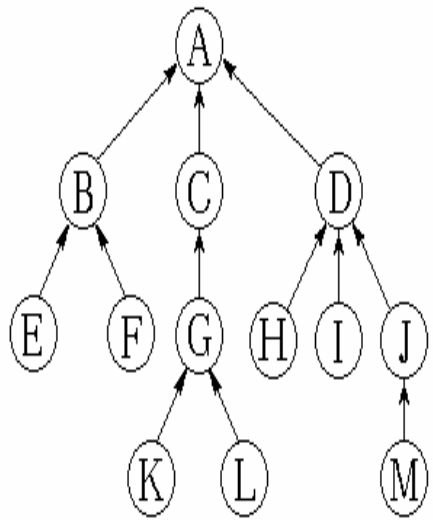
- 双亲表示法 p135
  - 用结构数组——树的顺序存储方式
  - 类型定义：p135
  - 找双亲方便，找孩子难
- 孩子链表表示法 p136
  - 顺序和链式结合的表示方法
  - 找孩子方便，找双亲难
  - 若用p137图6.14中带双亲的孩子链表表示，则找孩子找双亲都较方便
- 孩子兄弟表示法p136
  - 找孩子容易，若增加parent域，则找双亲也较方便。



(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

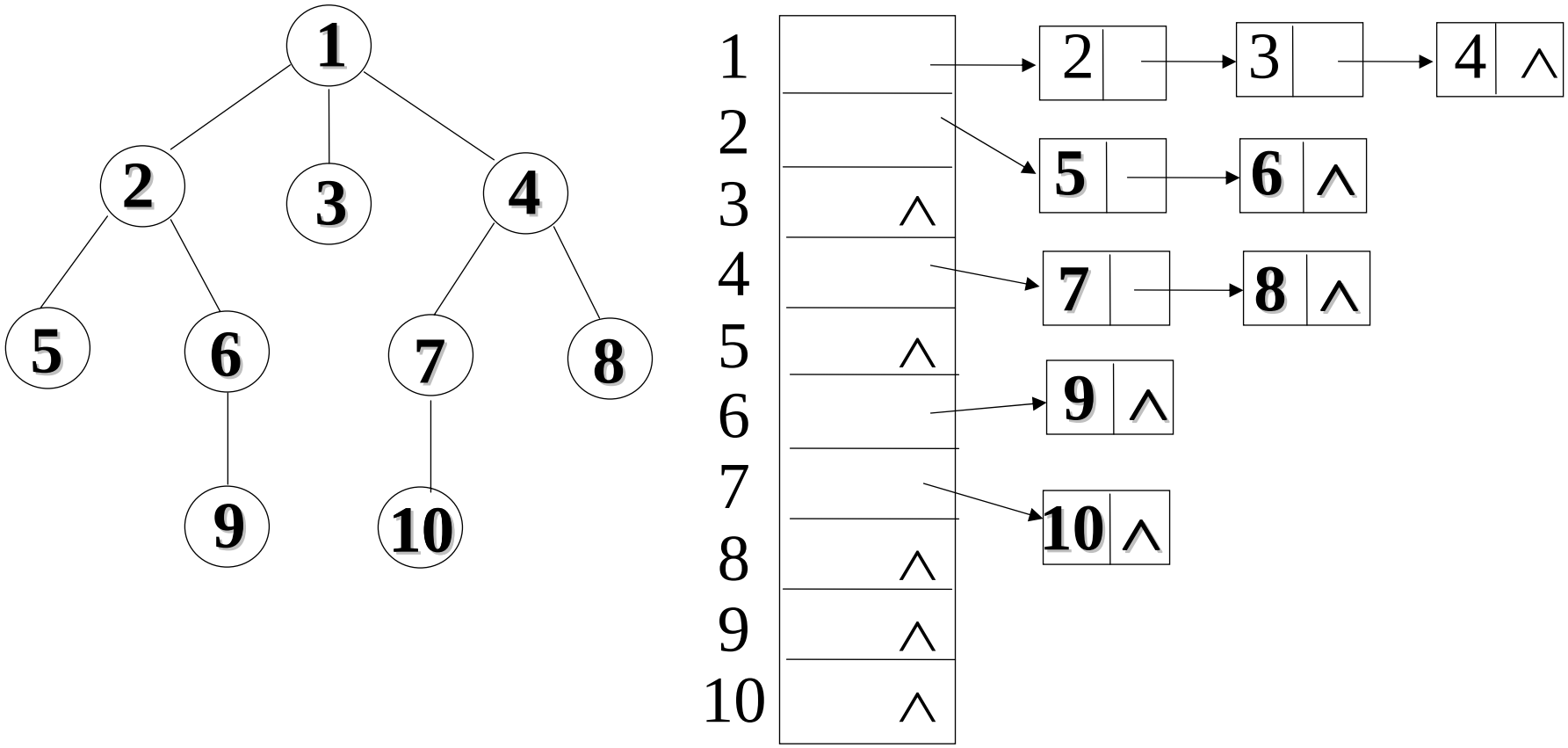


(c) 双亲表示图解

❒ 双亲表示

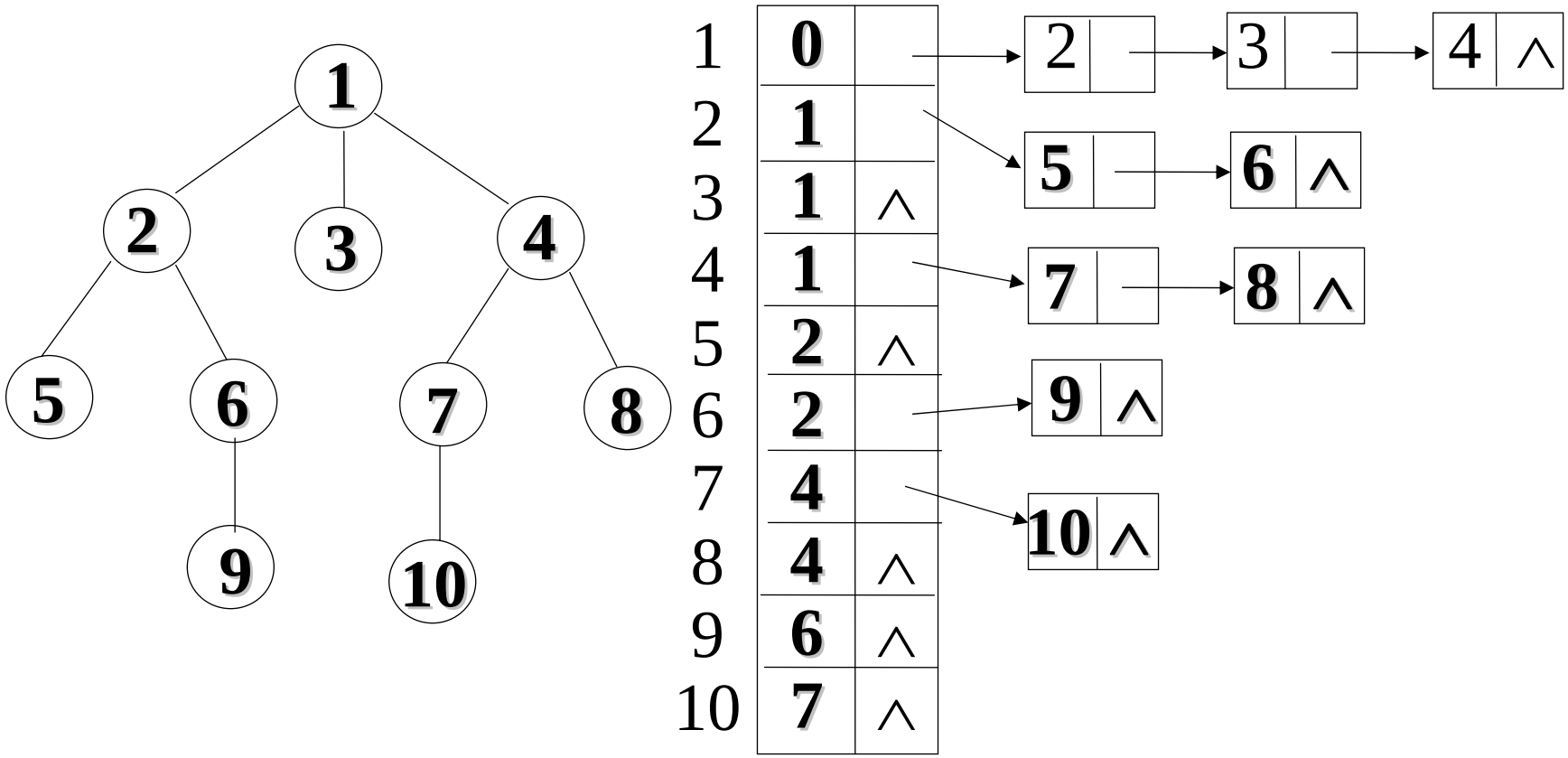
```
#define MAX_TREE_SIZE  100
typedef struct PTNode{
      TElemType data;
      int parent;
} PTNode;
typedef struct{
       PTNode nodes[MAX_TREE_SIZE];
       int r,n;  //根位置和结点数
}PTree;
```

❒ 孩子链表表示法—孩子链表



孩子表示法--单链链表

❒ 孩子链表表示法—带双亲的孩子链表

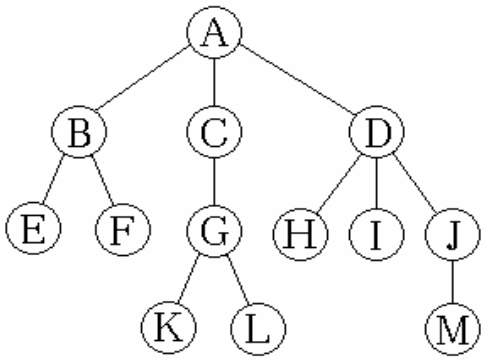


带双亲的孩子链表

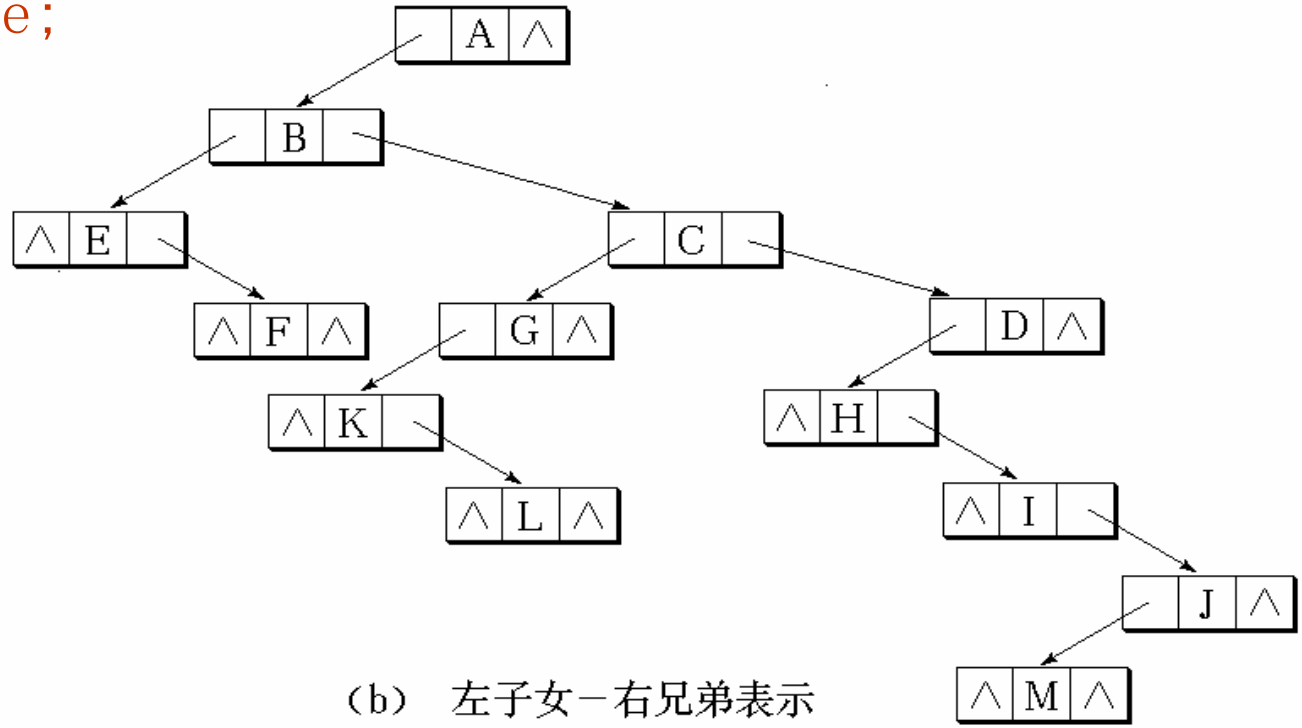
❒ 孩子兄弟表示法（二叉链表表示法）

data	*firstChild*	*nextSibling*

```
typedef struct CSNode{
ElemType  data;
struct CSNode  *firstchild, *nextsibling;
}CSNode,*CSTree;
```



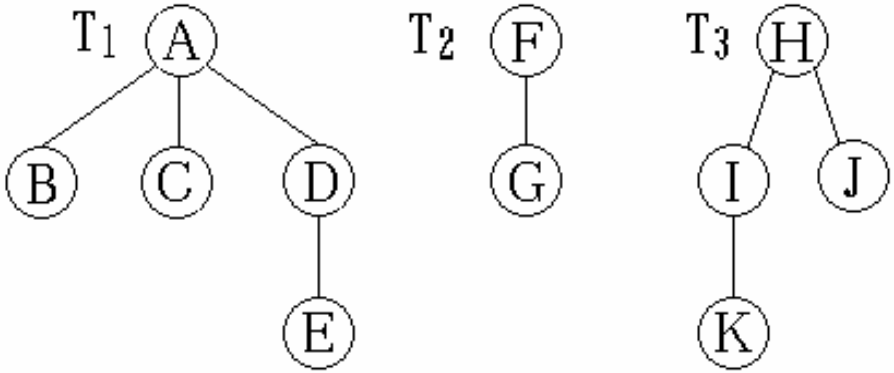
（a）树



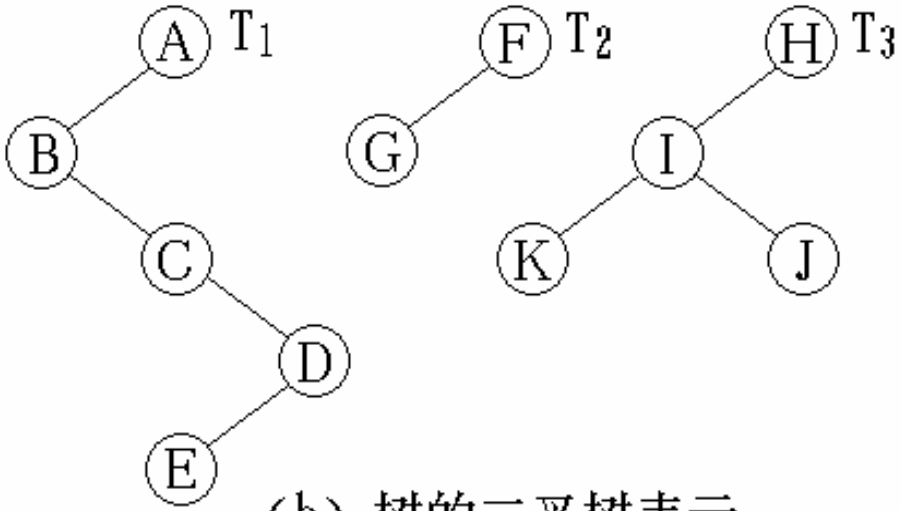
（b） 左子女－右兄弟表示

树的左孩子-右兄弟表示

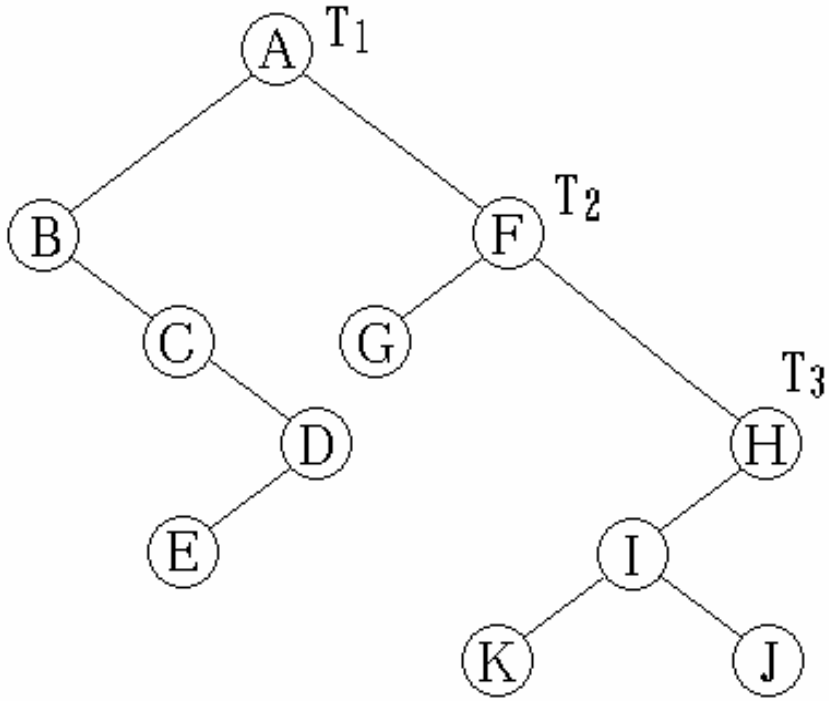
2. 森林与二叉树的转换



（a）3棵树的森林



（b）树的二叉树表示



（c）森林的二叉树表示

森林与二叉树的对应关系

(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。

- 以根结点为轴；左孩子不再旋转。

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。

(2) 森林转化成二叉树的规则

① 若F为空，即$n = 0$，则

对应的二叉树B为空二叉树。

② 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$B(T_{11}, T_{12}, \cdots, T_{1m})$，其中，$T_{11}$，$T_{12}$，…，$T_{1m}$是$root(T_1)$的子树；
- 其右子树为$B(T_2, T_3, \cdots, T_n)$，其中，T_2，T_3，…，T_n是除T_1外其它树构成的森林。

(3) 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。

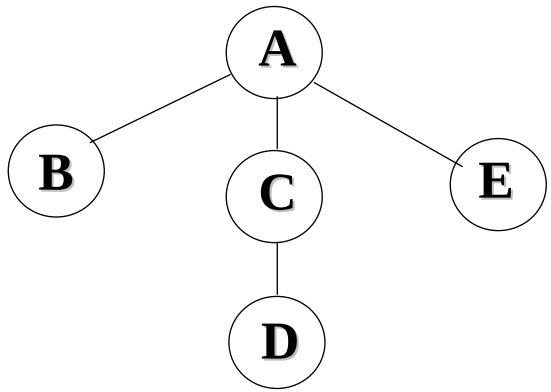
3. 树和森林的遍历

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

例：



先根序列：ABCDE

后根序列：BDCEA

3. 树和森林的遍历

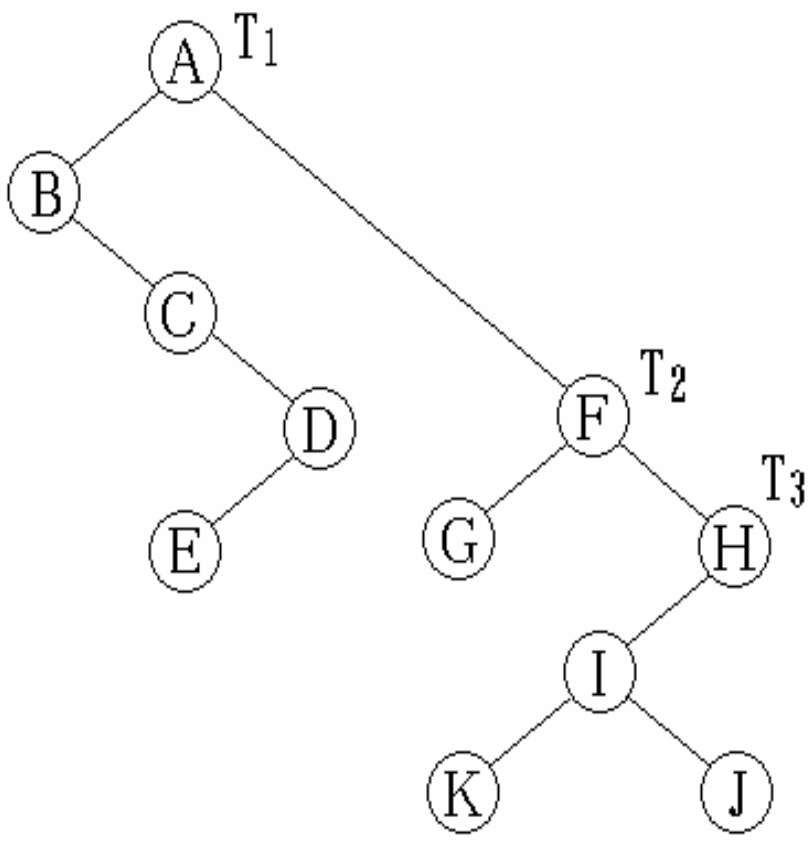
树的先根遍历 – 转换后的二叉树的先序遍历

树的后根遍历 – 转换后的二叉树的中序遍历

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

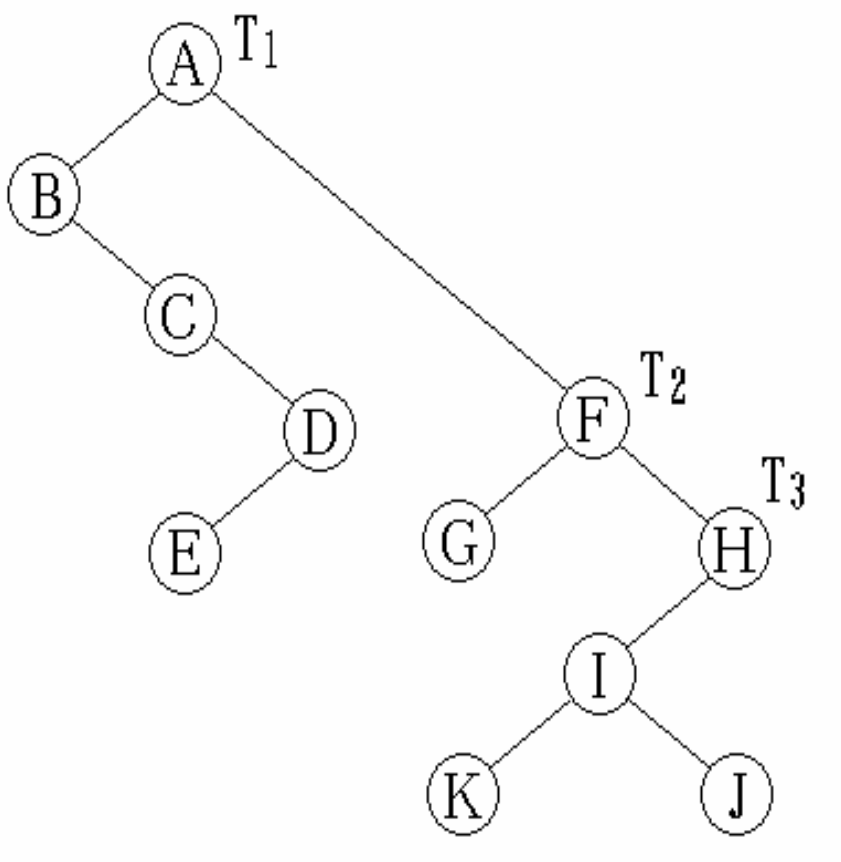


森林的二叉树表示

森林的遍历

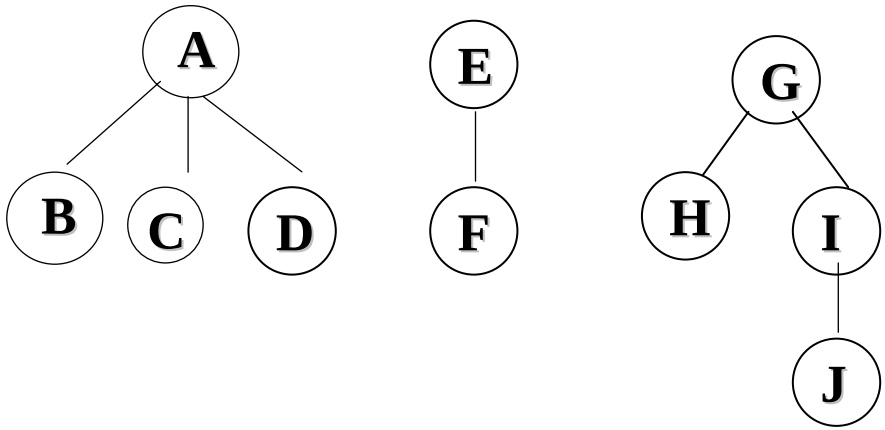
(2) 中根次序遍历的规则：

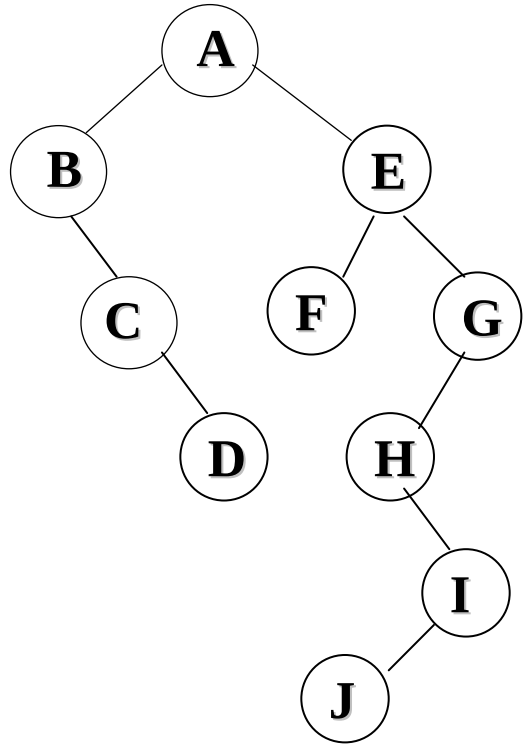
- 若森林F为空，返回；否则
  - ☞ 中根次序遍历第一棵树的子树森林；
  - ☞ 访问F的第一棵树的根结点；
  - ☞ 中根次序序遍历其它树组成的森林。



森林的二叉树表示

例：





森林的先序序列：ABCDEFGHIJ

森林的中序序列：BCDAFEHJIG

森林的遍历

先序：对应二叉树的先序遍历

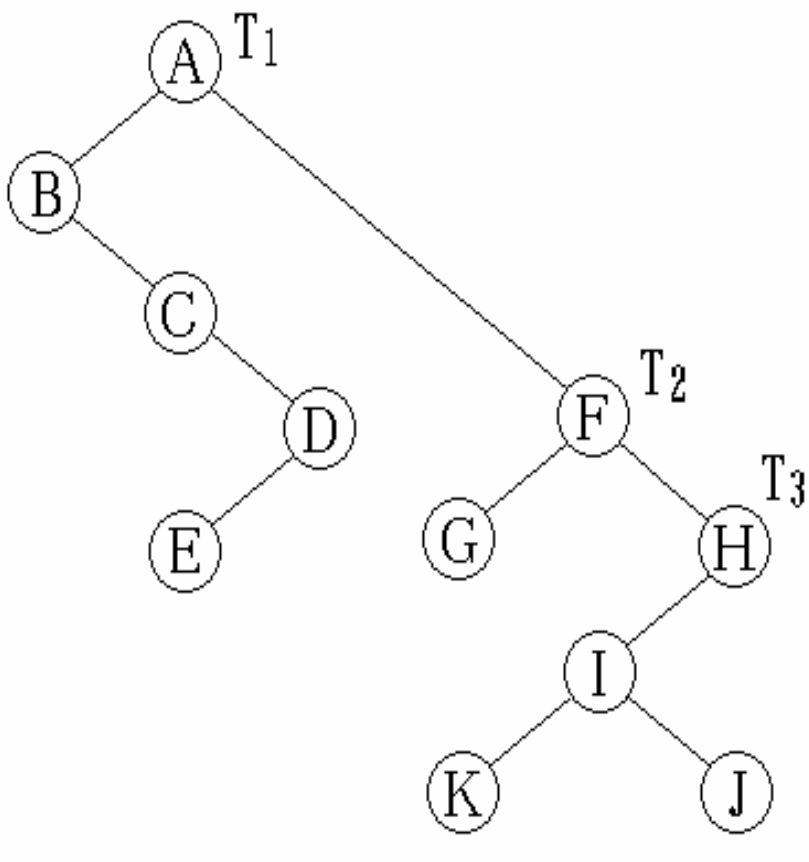
中序：对应二叉树的中序遍历

森林的遍历

(3) 后根次序遍历的规则：

- 若森林F为空，返回；否则
  - ☞ 后根次序遍历第一棵树的子树森林；
  - ☞ 后根次序遍历其它树组成的森林；
  - ☞ 访问F的第一棵树的根结点。

。

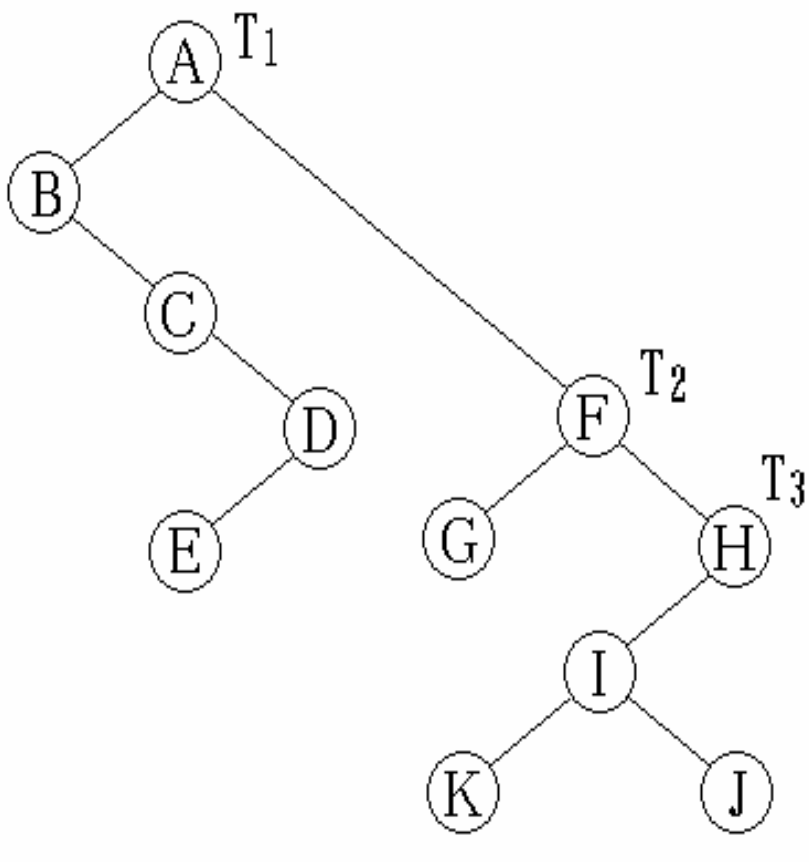


森林的二叉树表示

森林的遍历

(4) 广度优先遍历(层次序遍历)：

- 若森林F为空，返回；否则
  - ☞ 依次遍历各棵树的根结点；
  - ☞ 依次遍历各棵树根结点的所有孩子；
  - ☞ 依次遍历这些孩子结点的孩子结点。……



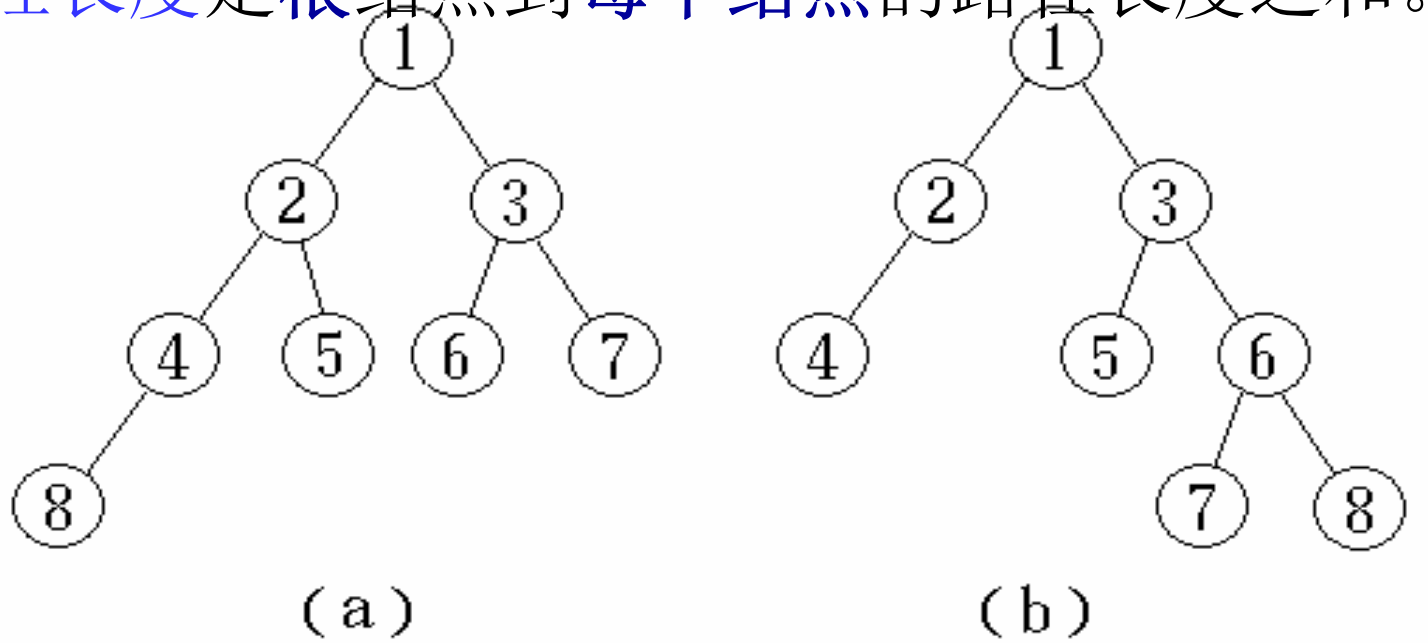
森林的二叉树表示

6.6 赫夫曼树 (Huffman Tree)及其应用p144

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。



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

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

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

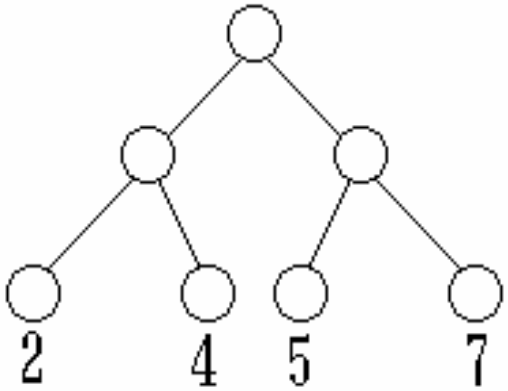
其路径长度最小者为 $$PL = \sum_{i=1}^{n} \lfloor \log_2 i \rfloor$$

带权路径长度（ Weighted Path Length, WPL ）

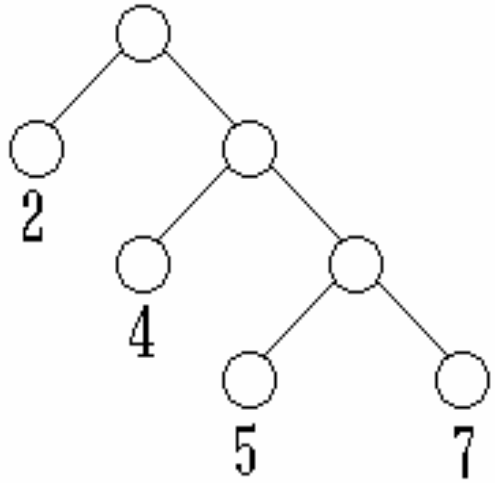
树的带权路径长度是树的各叶结点所带的权值与该结点到根的路径长度的乘积的和。

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

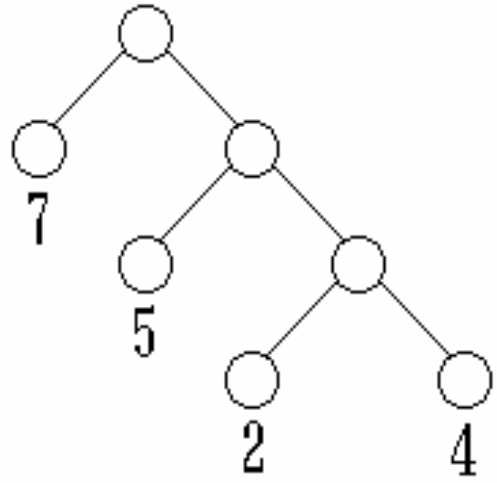
具有不同带权路径长度的扩充二叉树



(a) $WPL = 36$



(b) $WPL = 46$



(c) $WPL = 35$

赫夫曼树

带权路径长度达到最小的二叉树即为赫夫曼树（最优二叉树）。

赫夫曼算法

——如何构造一棵赫夫曼树

(1)由给定的n个权值$\{w_0, w_1, w_2, \cdots, w_{n-1}\}$，构造具有n棵二叉树的森林$F = \{T_0, T_1, T_2, \cdots, T_{n-1}\}$，其中每一棵二叉树$T_i$只有一个带有权值$w_i$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：

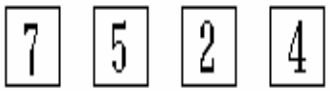
① 在F中选取两棵根结点的权值最小的二叉树，做为左、右子树构造一棵新的二叉树。置新的二叉树的根结点的权值为其左、右子树上根结点的权值之和。

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。

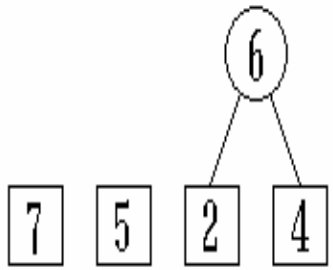
赫夫曼树的构造过程

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



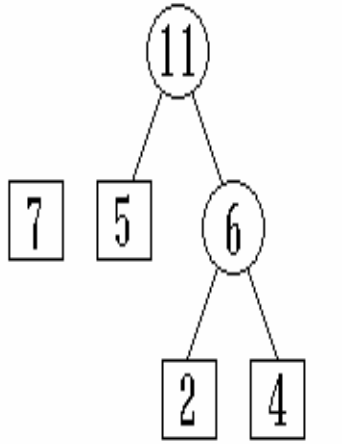
(a) 初始

F:{7}{5}{6}



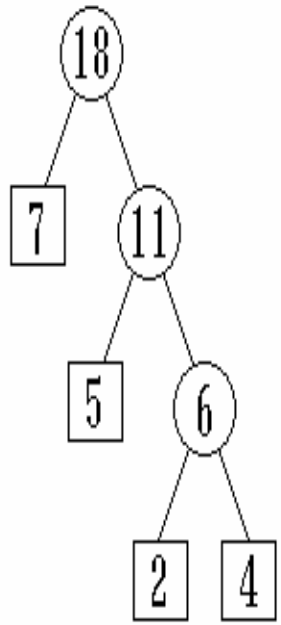
(b) 合并{2}{4}

F:{7}{11}



(c) 合并{5}{6}

F:{18}



(d) 合并{7}{11}

- 在赫夫曼树中，权值大的结点离根最近。

- 赫夫曼树的应用
  - 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　　p145图6.23
  - 用于通讯和数据传送时的赫夫曼编码

- 前缀

设S_1和S_2是两个串，如果SUBSTR(S_2 , 1 , LENGTH(S_1)) 和S_1是相等的，则称S_1是S_2的前缀。

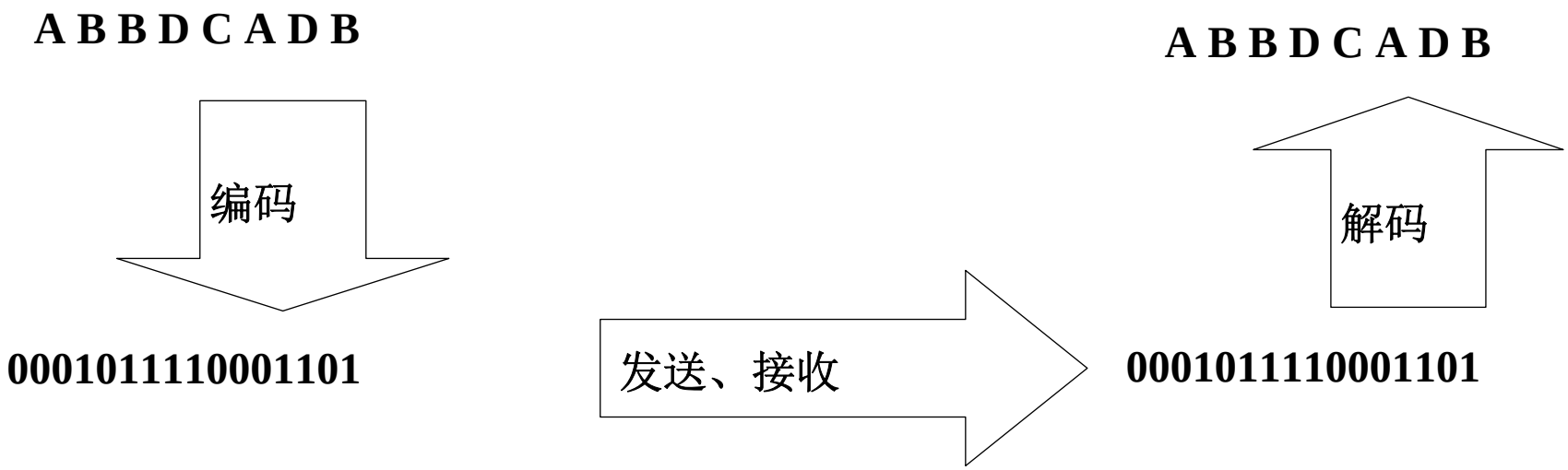
例如：“01”是“011”的前缀。

- 前缀编码

在一个编码系统中，任何一个字符的编码都不是另一个字符的编码的前缀。

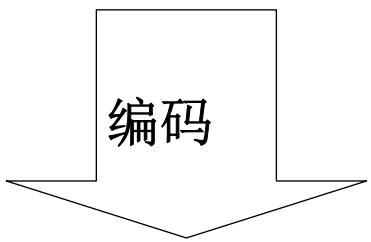
例：电报系统。发送时，对每一个字符进行编码，接收时，按编码的过程进行解码。

设**A、B、C、D**的编码分别为**00、01、10、11**。

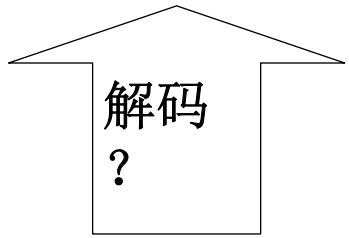


设A、B、C、D的编码分别为00、0、01、1。

A B B D C A D B



00001010010



00001010010

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}。

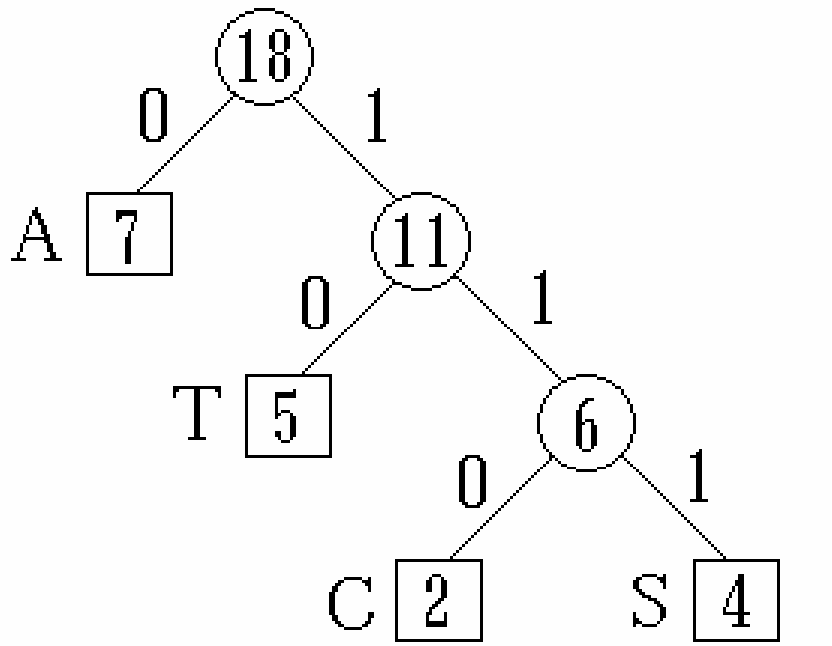
化整为｛ 2，7，4，5 ｝，以它们为各叶结点上的权值，建立赫夫曼树。左分支赋 0，右分支赋 1，得赫夫曼编码(变长编码)。

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。



赫夫曼编码树

建立赫夫曼树及求赫夫曼编码的算法 (p147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，m为结点数目
     int m=2*n-1;
     HT = (HuffmanTree)malloc((m+1)*sizeof(HTNode));
                                          //  0号单元未用
```

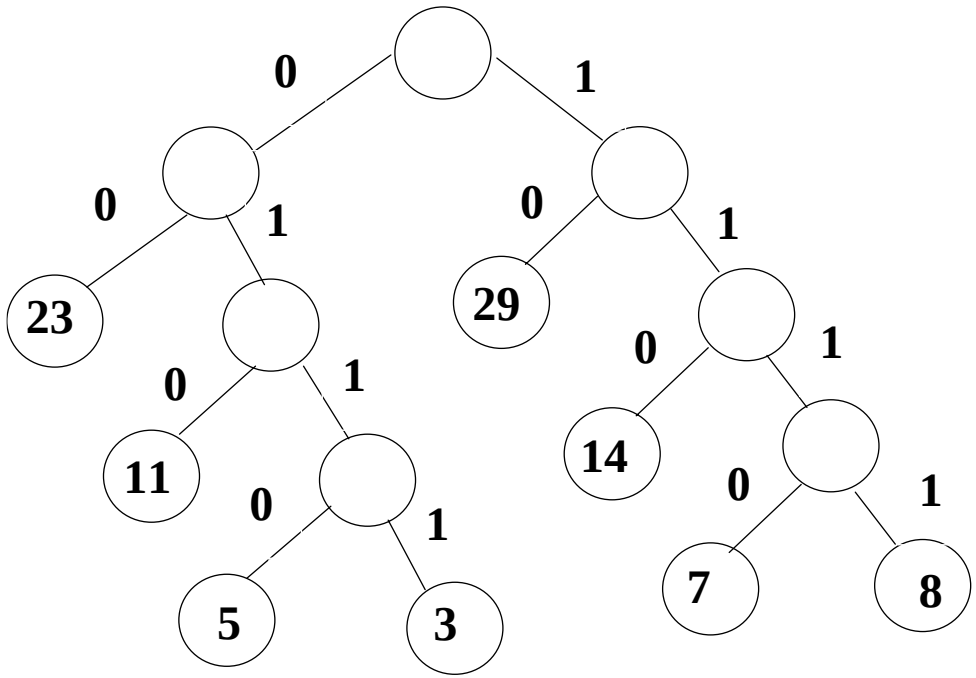
```
for (p=HT, i=1; i<=n; ++i,++p,++w)
    { p->weight = *w; p->parent=0; p->lchild=0;p->rchild=0; }
                  // *W={5,29,7,8,14,23,3,11}
                  // *p = { *w,0,0,0 };
for (; i<=m;++i,++p)
    { p->weight = 0; p->parent=0; p->lchild=0; p->rchild=0; }
                   //*p={ 0,0,0,0 };
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);
}
```

例：在一个系统的通信中只可能出现八种字符，其概率分别为0.05，0.29，0.07，0.08，0.14，0.23，0.03， 0.11 。

设权 w=（5， 29 ， 7 ， 8 ， 14， 23 ， 3，11），

n=8，则 m=15。



HT的初态

1	5	0	0	0
2	29	0	0	0
3	7	0	0	0
4	8	0	0	0
5	14	0	0	0
6	23	0	0	0
7	3	0	0	0
8	11	0	0	0
9	-	0	0	0
10	-	0	0	0
11	-	0	0	0
12	-	0	0	0
13	-	0	0	0
14	-	0	0	0
15	-	0	0	0

HT的终态

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

哈夫曼编码HC