第6章 树和二叉树

一、选择题

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1.D | 2.B | 3.C | 4.D | 5.D | 6.A | 7.1C | 7.2A | 7.3C | 7.4A | 7.5C | 8.B |
| 9.C | 10.D | 11.B | 12.E | 13.D | 14.D | 15.C | 16.B | 17.C | 18.C | 19.B | 20.D |
| 21.A | 22.A | 23.C | 24.C | 25.C | 26.C | 27.C | 28.C | 29.B | 30.C | 31.D | 32.B |
| 33.A | 34.D | 35.B | 36.B | 37.C | 38.B | 39.B | 40.B | 41.1F | 41.2B | 42.C | 43.B |
| 44.C | 45.C | 46.B | 47.D | 48.B | 49.C | 50.A | 51.C | 52.C | 53.C | 54.D | 55.C |
| 56.B | 57.A | 58.D | 59.D | 60.B | 61.1B | 61.2A | 61.3G | 62.B | 63.B | 64.D | 65.D |
| 66.1C | 66.2D | 66.3F | 66.4H | 66.5I |  |  |  |  |  |  |  |

部分答案解释如下。

12. 由二叉树结点的公式：n=n0+n1+n2=n0+n1+(n0-1)=2n0+n1-1， 因为n=1001,所以1002=2n0+n1,在完全二叉树树中，n1只能取0或1,在本题中只能取0，故n=501，因此选E。

42.前序序列是“根左右”，后序序列是“左右根”，若要这两个序列相反，只有单支树，所以本题的A和B均对，单支树的特点是只有一个叶子结点，故C是最合适的，选C。A或B都不全。由本题可解答44题。

47. 左子树为空的二叉树的根结点的左线索为空（无前驱），先序序列的最后结点的右线索为空（无后继），共2个空链域。

52．线索二叉树是利用二叉树的空链域加上线索，n个结点的二叉树有n+1个空链域。

二、判断题

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1.× | 2.× | 3.× | 4. √ | 5. √ | 6. √ | 7.√ | 8.× | 9. √ | 10.× | 11.× | 12.× |
| 13.× | 14.√ | 15.× | 16.× | 17.√ | 18.√ | 19.× | 20.√ | 21.× | 22.√ | 23.× | 24.× |
| 25.√ | 26.× | 27.× | 28.× | 29.√ | 30.× | 31.× | 32.√ | 33.× | 34.× | 35.× | 36.√ |
| 37.√ | 38.× | 39.× | 40.× | 41.(3) | 42.√ | 43.√ | 44.× | 45.√ | 46.× | 47.× | 48.× |
| 49.√ | 50.√ |  |  |  |  |  |  |  |  |  |  |

部分答案解释如下。

6．只有在确定何序（前序、中序、后序或层次）遍历后，遍历结果才唯一。

19．任何结点至多只有左子树的二叉树的遍历就不需要栈。

24. 只对完全二叉树适用，编号为i的结点的左儿子的编号为2i(2i<=n)，右儿子是2i+1（2i+1<=n）

37. 其中序前驱是其左子树上按中序遍历的最右边的结点（叶子或无右子女），该结点无右孩子。

38 . 新插入的结点都是叶子结点。

42. 在二叉树上，对有左右子女的结点，其中序前驱是其左子树上按中序遍历的最右边的结点（该结点的后继指针指向祖先），中序后继是其右子树上按中序遍历的最左边的结点（该结点的前驱指针指向祖先）。

44．非空二叉树中序遍历第一个结点无前驱，最后一个结点无后继，这两个结点的前驱线索和后继线索为空指针。

三.填空题

1.(1)根结点(2)左子树(3)右子树 2.(1)双亲链表表示法(2)孩子链表表示法(3)孩子兄弟表示法

3．p->lchild==null && p->rchlid==null 4.(1) ++a\*b3\*4-cd (2)18 5.平衡因子

6. 9 7. 12 8.(1)2k-1 (2)2k-1 9.(1)2H-1 (2)2H-1 (3)H=⎣log2N⎦+1

10. 用顺序存储二叉树时，要按完全二叉树的形式存储，非完全二叉树存储时，要加“虚结点”。设编号为i和j的结点在顺序存储中的下标为s 和t ,则结点i和j在同一层上的条件是⎣log2s⎦=⎣log2t⎦。

11. ⎣log2i⎦=⎣log2j⎦ 12.(1)0 (2)(n-1)/2 (3)(n+1)/2 (4) ⎣log2n⎦ +1 13.n

14. N2+1 15.(1) 2K+1-1 (2) k+1 16. ⎣N/2⎦ 17. 2k-2 18. 64

19. 99 20. 11 21.(1) n1-1 (2)n2+n3

22.(1)2k-2+1（第k层1个结点，总结点个数是2H-1，其双亲是2H-1/2=2k-2）(2) ⎣log2i⎦+1 23.69

24. 4 25.3h-1 26. ⎣n/2⎦ 27. ⎡log2k⎤+1

28.(1)完全二叉树 (2)单枝树，树中任一结点（除最后一个结点是叶子外）,只有左子女或只有右子女。

29.N+1 30.(1) 128(第七层满，加第八层１个)　　(2) 7

31. 0至多个。任意二叉树，度为１的结点个数没限制。只有完全二叉树，度为１的结点个数才至多为1。

32．21 33.(1)2 (2) n-1 (3) 1 (4) n (5) 1 (6) n-1

34.(1) FEGHDCB (2)BEF（该二叉树转换成森林，含三棵树，其第一棵树的先根次序是BEF）

35.(1)先序（2）中序 36. (1)EACBDGF （2）2 37.任何结点至多只有右子女的二叉树。

38.(1)a (2) dbe (3) hfcg 39.(1) .D.G.B.A.E.H.C.F. (2) ...GD.B...HE..FCA

40.DGEBFCA 41．(1)5 （2）略 42.二叉排序树 43.二叉树 44.前序

45.(1)先根次序（2）中根次序 46.双亲的右子树中最左下的叶子结点 47.2 48.(n+1)/2

49.31（x的后继是经x的双亲y的右子树中最左下的叶结点） 50.(1)前驱 (2)后继

51.(1)1 (2)y^.lchild (3)0 (4)x (5)1 (6) y (7)x(编者注：本题按中序线索化)

52.带权路径长度最小的二叉树，又称最优二叉树 53.69 54.(1)6 (2)261

55.(1)80 (2)001（不唯一）56.2n0-1

57.本题①是表达式求值，②是在二叉排序树中删除值为x的结点。首先查找x，若没有x，则结束。否则分成四种情况讨论：x结点有左右子树；只有左子树；只有右子树和本身是叶子。

(1)Postoder\_eval(t^.Lchild) (2) Postorder\_eval(t^.Rchild) (3)ERROR(无此运算符)(4)A

(5)tempA^.Lchild (6)tempA=NULL (7)q^.Rchild (8)q (9)tempA^.Rchild (10)tempA^.Item<r^.Item

58.(1) **IF** t=NIL THEN num:=0 **ELSE** num:=num(t^.l)+num(t^.r)+1

(2) **IF** (t=NIL) AND (m≤n) OR (t<>NIL) AND (m>n) THEN all:=false

**ELSE** BEGIN chk(t^.l,2\*m);chk (t^.r,2\*m+1);END

59. (1)p->rchild (2)p->lchild (3)p->lchild (4)ADDQ(Q,p->lchild) (5)ADDQ(Q,p->rchild)

60.(1)t->rchild!=null (2)t->rchild!=null (3)N0++ (4)count(t->lchild) (5)count(t->rchild)

61.(1)p (2)0 (3)height(p->lchild) (4)0 (5)height(p->rchild) (6)lh+1 (7)rh+1 (8)0

62.(1)p<>NIL (2)addx(p) (3)addx(tree) (4)r^.rchild

63.(1)stack[tp]=t (2) p=stack[tp--] (3)p (4)++tp

64.① 本算法将二叉树的左右子树交换

　② (1)new (s) //初始化，申请结点 (2) s^.next=NIL　 //s是带头结点的链栈

(3)s^.next^.data　//取栈顶元素 (4)s^.next:= p^.next //栈顶指针下移

(5)dispose(p)　　 //回收空间 (6)p^.next:=s^.next　　//将新结点入链栈

(7)push(s,p^.rchild) //先沿树的左分支向下，将p的右子女入栈保存

(8)NOT empty(s) (9) finishe:=true //已完成 (10)finish=true （或s^.next=NIL）

65.(1)new(t) (2)2\*i≤n　 (3)t^.lchild,2\*i (4)2\*i+1≤n (5)t^.rchild,2\*i+1 (6)1

66.(1)Push(s,p) (2)K=2 (3)p->data=ch (4)BT=p (5) ins>>ch

67.(1)result; (2)p:=p^.link; (3) q:=q^.pre （（２）（３）顺序可变）

68.(1)top++ (2) stack[top]=p->rchild (3)top++ (4)stack[top]=p->lchild

69．(1)(i<=j) AND (x<=y) (2)A[i]<>B[k] (3)k-x

(4)creatBT(i+1,i+L,x,k-1,s^.lchild) (5) creatBT(i+L+1,j,k+1,y,s^.rchild)

70. (1)push(s,bt) （2）pop(s) （3）push(s,p^.rchild) // p的右子树进栈

71．(1) p=p->lchild // 沿左子树向下 （2）p=p->rchild

72．(1)0 （2）hl>hr (3)hr=hl

73. (1)top>0 (2)t\*2 // 沿左分枝向下 （3）top-1 // 退栈

74．(1)p:=p^.lchild （2）（3）p:=S.data[s.top]^.rchild (4)s.top=0

75. (1)\*ppos // 根结点 （2）rpos=ipos (3)rpos–ipos (4)ipos (5)ppos+1

76. (1)top>0 (2)stack[top]:=nd^.right (3)nd^.left<>NIL (4)top:=top+1 (左子树非空)

77. (1) p<>thr // 未循环结束 （2）p->ltag=0 (3)p->lchild

(4)p->rtag=1 && p->rchild!=thr (5) p=p->rchild (6)p=p->rchild

78. 若p^.rtag=1,则p^.rchild 为后继，否则p的后继是p的右子树中最左下的结点

(1)q=p^.rchild (2)q^.ltag=0 (3) q^.lchild

79．（1）tree->lchild (2)null (3)pre->rchild

(4)pre->rtag=1 (5) pre->right=tree; (6) tree->right (注（4）和（5）顺序可换)

80．（1）node->rflag==0 (2)\*x=bt (3) \*x=node->right

四．应用题

1．树的孩子兄弟链表表示法和二叉树二叉链表表示法，本质是一样的，只是解释不同，也就是说树（树是森林的特例，即森林中只有一棵树的特殊情况）可用二叉树唯一表示，并可使用二叉树的一些算法去解决树和森林中的问题。

树和二叉树的区别有三：一是二叉树的度至多为2，树无此限制；二是二叉树有左右子树之分，即使在只有一个分枝的情况下， 也必须指出是左子树还是右子树，树无此限制；三是二叉树允许为空，树一般不允许为空（个别书上允许为空）。

2.树和二叉树逻辑上都是树形结构，区别有以上题1所述三点。二叉树不是树的特例。

3．线性表属于约束最强的线性结构，在非空线性表中，只有一个“第一个”元素，也只有一个“最后一个”元素；除第一个元素外，每个元素有唯一前驱；除最后一个元素外，每个元素有唯一后继。树是一种层次结构，有且只有一个根结点，每个结点可以有多个子女，但只有一个双亲（根无双亲），从这个意义上说存在一（双亲）对多（子女）的关系。广义表中的元素既可以是原子，也可以是子表，子表可以为它表共享。从表中套表意义上说，广义表也是层次结构。从逻辑上讲，树和广义表均属非线性结构。但在以下意义上，又蜕变为线性结构。如度为1的树，以及广义表中的元素都是原

子时。另外，广义表从元素之间的关系可看成前驱和后继，也符

合线性表，但这时元素有原子，也有子表，即元素并不属于同一

数据对象。

4．方法有二。一是对该算术表达式（二叉树）进行后序遍历，

得到表达式的后序遍历序列，再按后缀表达式求值；二是递归

求出左子树表达式的值，再递归求出右子树表达式的值，最后

按根结点运算符（+、-、\*、/ 等）进行最后求值。

5．该算术表达式转化的二叉树如右图所示。 第5题图

6．n（n>0）个结点的d度树共有nd个链域，除根结点外，每个结点均有一个指针所指，故该树的空链域有nd-(n-1)=n(d-1)+1个。

7．证明：设二叉树度为0和2的结点数及总的结点数分别为n0，n2 和n，则n=n0+n2 … (1)

再设二叉树的分支数为B,　除根结点外，每个结点都有一个分支所指，则 n=B+1… … …(2)

度为零的结点是叶子，没有分支，而度为2的结点有两个分支，因此（2）式可写为

n=2\*n2+1 ……………(3)

由（1）、（3）得n2=n0-1,代入（1），并由（1）和（2）得B=2\*(n0-1)。 证毕。

8．(1)kh-1(h为层数)

（2）因为该树每层上均有Kh-1个结点，从根开始编号为1，则结点i的从右向左数第２个孩子的结点编号为ki。设n 为结点i的子女，则关系式(i-1)k+2<=n<=ik+1成立，因i是整数，故结点n的双亲i的编号为⎣n-2)/k⎦+1。

(3) 结点n(n>1)的前一结点编号为n-1（其最右边子女编号是(n-1)\*k+1），故结点 n的第 i个孩子的编号是(n-1)\*k+1+i。

(4) 根据以上分析，结点n有右兄弟的条件是，它不是双亲的从右数的第一子女，即 (n-1)%k!=0，其右兄弟编号是n+1。

9．最低高度二叉树的特点是，除最下层结点个数不满外，其余各层的结点数都应达到各层的最大值。设n个结点的二叉树的最低高度是h，则n应满足2h-1<n<=2h-1关系式。解此不等式，并考虑h是整数，则有h=⎣logn⎦+1，即任一结点个数为n 的二叉树的高度至少为O(logn)。

10．2n-1(本题等价于高度为n的满二叉树有多少叶子结点，从根结点到各叶子结点的单枝树是不同的二叉树。)

11．235。由于本题求二叉树的结点数最多是多少，第7层共有27-1=64个结点，已知有10个叶子，其余54个结点均为分支结点。它在第八层上有108个叶子结点。所以该二叉树的结点数最多可达(27-1+108)=235。(注意；本题并未明说完全二叉树的高度，但根据题意，只能8层。)

12．1023（=210-1）

13．证明：设度为1和2 的结点数是n1和n2，则二叉树结点数n为n=m+n1+n2………… (1)

由于二叉树根结点没有分枝所指，度为1和2的结点各有1个和2个分枝，度为0 的结点没有分枝，故二叉树的结点数n与分枝数B有如下关系

n=B+1=n1+2\*n2+1……………………….(2)

由（1）和（2），得n2=m-1。即n个结点的二叉树，若叶子结点数是m，则非叶子结点中有（m-1）个度为2，其余度为1。

14．根据顺序存储的完全二叉树的性质，编号为i的结点的双亲的编号是⎣i/2⎦，故A[i]和A[j]的最近公共祖先可如下求出：

**while**(i/2!=j/2)

**if**(i>j) i=i/2; **else** j=j/2;

退出**while**后，若i/2=0,则最近公共祖先为根结点，否则最近公共祖先是i/2。

15．N个结点的K叉树，最大高度N（只有一个叶结点的任意k叉树）。设最小高度为H，第i(1<=i<=H)层的结点数Ki-1，则N=1+k+k2+…+ kH-1，由此得H=⎣logK(N(K-1)+1)⎦

16. 结点个数在20到40的满二叉树且结点数是素数的数是31，其叶子数是16。

17．设分枝结点和叶子结点数分别是为nk和n0，因此有n=n0+nk (1)

另外从树的分枝数B与结点的关系有 n=B+1=K\*nk +1 (2)

由（1）和（2）有 n0=n-nk=(n(K-1)+1)/K

18.用顺序存储结构存储n个结点的完全二叉树。编号为i的结点，其双亲编号是⎣i/2⎦(i=1时无双亲)，其左子女是2i(若2i<=n,否则i无左子女)，右子女是2i+1(若2i+1<=n,否则无右子女)。

19. 根据完全二叉树的性质，最后一个结点（编号为n）的双亲结点的编号是⎣n/2⎦，这是最后一个分枝结点，在它之后是第一个终端（叶子）结点，故序号最小的叶子结点的下标是⎣n/2⎦+1。

20. 按前序遍历对顶点编号，即根结点从１开始，对前序遍历序列的结点从小到大编号。

21. 设树的结点数为n，分枝数为B，则下面二式成立

n=n0+n1+n2+…+nm (1)

n=B+1= n1+2n2+…+mnm (2)

由(1)和(2)得叶子结点数n0=1+

22. ⎡log2n⎤ +1 　　　　　　　　 23.15

24. 该结论不成立。对于任一a€A,可在B中找到最近祖先f。a在f的左子树上。对于从f到根结点路径上所有b€B，有可能f在b的右子树上，因而a也就在b的右子树上，这时a＞b，因此a＜b不成立。同理可以证明b<c不成立。而对于任何a∈A,c∈C均有a<c。

25. n个结点的m次树，共有n\*m个指针。除根结点外，其余n-1个结点均有指针所指，故空指针数为n\*m-(n-1)=n\*(m-1)+1。证毕。

26. 证明 设度为1和2 及叶子结点数分别为n0，n1和n2，则二叉树结点数n为n=n0+n1+n2 (1)

再看二叉树的分支数，除根结点外，其余结点都有一个分支进入，设B为分支总数，则n=B+1。度为1和2的结点各有1个和2个分支，度为0 的结点没有分支，故n=n1+2n2+1 (2)

由（1）和（2），得n0= n2+1。

27. 参见题26．

28. 设完全二叉树中叶子结点数为n，则根据完全二叉树的性质，度为2的结点数是n-1，而完全二叉树中，度为1的结点数至多为1，所以具有n个叶子结点的完全二叉树结点数是n+(n-1)+1=2n或2n-1（有或无度为1的结点）。由于具有2n(或2n-1)个结点的完全二叉树的深度是⎣log2(2n)⎦+1( ⎣log2(2n-1)⎦+1，即⎡log2n⎤+1,故n个叶结点的非满的完全二叉树的高度是⎡log2n⎤+1。（最下层结点数>=2）。

29. (1)根据二叉树度为2 结点个数等于叶子结点个数减1的性质，故具有n个叶子结点且非叶子结点均有左左子树的二叉树的结点数是2n-1。

(2)证明：当i=1时，2-（1-1）=20=1，公式成立。设当i=n-1时公式成立，证明当i=n时公式仍成立。

设某叶子结点的层号为t，当将该结点变为内部结点，从而再增加两个叶子结点时，这两个叶子结点的层号都是t+1，对于公式的变化，是减少了一个原来的叶子结点，增加了两个新叶子结点，反映到公式中，因为2-(t-1)=2-(t+1-1)+2-(t+1-1)，所以结果不变，这就证明当i=n时公式仍成立。证毕．

30．结点数的最大值2h-1（满二叉树），最小值2h-1（第一层根结点，其余每层均两个结点）。

31．（1）k(u-1)+1+i (2) ⎣(v-2)/k⎦+1 (参见第8题推导)

32．该二叉树是按前序遍历顺序编号，以根结点为编号1，前序遍历的顺序是“根左右”。

33．（1）设n=1，则e=0+2\*1=2（只有一个根结点时，有两个外部结点），公式成立。

设有n个结点时，公式成立，即

En=In+2n (1)

现在要证明，当有n+1个结点时公式成立。

增加一个内部结点，设路径长度为l，则

In+1=In+l (2)

该内部结点，其实是从一个外部结点变来的，即这时相当于也增加了一个外部结点（原外部结点变成内部结点时，增加两个外部结点），则

En+1=En+l+2 (3)

由（1）和（2），则（3）推导为

En+1=In+2n+l+2=In+1-l+2n+l+2

=In+1+2(n+1）

故命题成立

（2）成功查找的平均比较次数s=I/n

不成功查找的平均比较次数u=（E-n）/（n+1）=（I+n）/n+1

由以上二式，有s=（1+1/n）\*u-1 。

|  |
| --- |
| 1 |

|  |
| --- |
| 15 |

|  |
| --- |
| 3 |

|  |
| --- |
|  |

|  |
| --- |
| 2 |

|  |
| --- |
| 11 |

|  |
| --- |
| 14 |

|  |
| --- |
| 12 |

|  |
| --- |
| 13 |

|  |
| --- |
| 10 |

|  |
| --- |
| 6 |

|  |
| --- |
| 5 |

|  |
| --- |
| 4 |

|  |
| --- |
| 9 |

|  |
| --- |
| 8 |

|  |
| --- |
| 7 |

|  |
| --- |
| 16 |

|  |
| --- |
| 0 |

34．

该有向图只有一个顶点入度为0，其余顶点入度均为1，

它不是有向树。 35．题图

36．参见26题

37．由于二叉树前序遍历序列和中序遍历序列可唯一确定一棵二叉树，因此，若入栈序列为1，2，3，…，n，相当于前序遍历序列是1，2，3，…，n，出栈序列就是该前序遍历对应的二叉树的中序序列的数目。因为中序遍历的实质就是一个结点进栈和出栈的过程，二叉树的形态确定了结点进栈和出栈的顺序，也就确定了结点的中序序列。

下图以入栈序列1，2，3，（解释为二叉树的前序序列）为例，说明不同形态的二叉树在中序遍历时栈的状态和访问结点次序的关系：

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | 1 |  |  | | --- | | 3 |  |  | | --- | | 2 | | |  | | --- | | 1 |  |  | | --- | | 3 |  |  | | --- | | 2 | | |  | | --- | | 1 |  |  | | --- | | 3 |  |  | | --- | | 2 | | |  | | --- | | 2 |  |  | | --- | | 1 |  |  | | --- | | 3 | | |  | | --- | | 2 |  |  | | --- | | 1 |  |  | | --- | | 3 | |
| 栈状态 访问  空  1  1 2  1 2 3  1 2 3  1 2  空 1 | 栈状态 访问  空  1  1 2  1 2  1 3  1 3  空 1 | 栈状态 访问  空  1  1 2  1 2  空 1  3  空 3 | 栈状态 访问  空  1  空 1  2  2 3  2 3  空 2 | 栈状态 访问  空  1  空 1  2  空 2  3  空 3 |

38．给定二叉树结点的前序序列和对称序（中序）序列，可以唯一确定该二叉树。因为前序序列的第一个元素是根结点，该元素将二叉树中序序列分成两部分，左边（设l个元素）表示左子树，若左边无元素，则说明左子树为空；右边（设r个元素）是右子树，若为空，则右子树为空。根据前序遍历中“根­—左子树—右子树”的顺序，则由从第二元素开始的l个结点序列和中序序列根左边的l个结点序列构造左子树，由前序序列最后r个元素序列与中序序列根右边的r个元素序列构造右子树。

由二叉树的前序序列和后序序列不能唯一确定一棵二叉树，因无法确定左右子树两部分。例如，任何结点只有左子树的二叉树和任何结点只有右子树的二叉树，其前序序列相同，后序序列相同，但却是两棵不同的二叉树。

39．前序遍历是“根左右”，中序遍历是“左根右”，后序遍历是“左右根”。三种遍历中只是访问“根”结点的时机不同，对左右子树均是按左右顺序来遍历的，因此所有叶子都按相同的相对位置出现。

40．在第38题，已经说明由二叉树的前序序列和中序序列可以确定一棵二叉树，现在来证明由二叉树的中序序列和后序序列，也可以唯一确定一棵二叉树。

当n=1时，只有一个根结点，由中序序列和后序序列可以确定这棵二叉树。

设当n=m-1时结论成立，现证明当n=m时结论成立。

设中序序列为S1，S2，…，Sm,后序序列是P1,P2,…，Pm。因后序序列最后一个元素Pm是根，则在中序序列中可找到与Pm相等的结点（设二叉树中各结点互不相同）Si(1≤i≤m)，因中序序列是由中序遍历而得，所以Si是根结点，S1，S2，…，Si-1是左子树的中序序列，而Si+1,Si+2,…,Sm是右子树的中序序列。

若i=1，则S1是根，这时二叉树的左子树为空，右子树的结点数是m-1,则{S2，S3，…，Sm}和{P1，P2，…，Pm-1}可以唯一确定右子树，从而也确定了二叉树。

若i=m,则Sm是根，这时二叉树的右子树为空，左子树的结点数是m-1，则{S1，S2，…，Sm-1}和{P1，P2，…，Pm-1}唯一确定左子树，从而也确定了二叉树。

最后，当1<i<m时，Si把中序序列分成{S1，S2，…，Si-1}和{Si+1,Si+2,…，Sm}。由于后序遍历是“左子树—右子树—根结点”，所以{P1,P2,…,Pi-1}和{Pi,Pi+1,…Pm-1}是二叉树的左子树和右子树的后序遍历序列。因而由{S1，S2，…，Si-1}和{P1，P2，…，Pi-1}

|  |
| --- |
| A |

|  |
| --- |
| B |

|  |
| --- |
| D |

|  |
| --- |
| C |

|  |
| --- |
| F |

|  |
| --- |
| E |

|  |
| --- |
| G |

|  |
| --- |
| B |

|  |
| --- |
| D |

|  |
| --- |
| C |

|  |
| --- |
| F |

|  |
| --- |
| E |

|  |
| --- |
| H |

|  |
| --- |
| A |

|  |
| --- |
| G |

可唯一确定二叉树的左子树，由{Si+1,Si+2,…，Sm}和

{Pi,Pi+1,…，Pm-1}可唯一确定二叉树的右子树 。

由中序序列DBEAFGC和后序序列DEBGFCA构

造的二叉树如右图：

41．证明请参见第40题和第38题 第40题图 第41题图

由前序序列ABDGECFH和中序序列DGBEAFHC构造的二叉树如图：

42．参见第38题

43．先序遍历二叉树的顺序是“根—左子树—右子树”，中序遍历“左子树—根—右子树”，后序遍历顺序是：“左子树—右子树―根＂，根据以上原则，本题解答如下：

1. 若先序序列与后序序列相同，则或为空树，或为只有根结点的二叉树
2. 若中序序列与后序序列相同，则或为空树，或为任一结点至多只有左子树的二叉树．
3. 若先序序列与中序序列相同，则或为空树，或为任一结点至多只有右子树的二叉树．
4. 若中序序列与层次遍历序列相同，则或为空树，

|  |
| --- |
| H |

|  |
| --- |
| G |

|  |
| --- |
| D |

|  |
| --- |
| A |

|  |
| --- |
| C |

|  |
| --- |
| J |

|  |
| --- |
| I |

|  |
| --- |
| B |

|  |
| --- |
| F |

|  |
| --- |
| E |

|  |
| --- |
| M |

|  |
| --- |
| P |

|  |
| --- |
| O |

|  |
| --- |
| N |

|  |
| --- |
| KO |

|  |
| --- |
| L |

或为任一结点至多只有右子树的二叉树

由中序序列DBEAFIHCG和后序序列DEBHIFGCA确定的二叉树略。

44. 森林转为二叉树的三步：

（１）连线（将兄弟结点相连，各树的根看作兄弟）；

（２）切线（保留最左边子女为独生子女，将其它子女分枝切掉）；

（３）旋转（以最左边树的根为轴，顺时针向下旋转45度）。

其实经过（１）和（２），已转为二叉树，

执行（３）只是为了与平时的二叉树的画法一致。

45．（1）①tree[p]·l→p ② tree[p]·r→p ③ p=0

（2）框（A）移至Ⅲ处，成为前序遍历。 44题图

46.



|  |
| --- |
| A |

|  |
| --- |
| BM |

|  |
| --- |
| F |

|  |
| --- |
| D |

|  |
| --- |
| (3) |

|  |
| --- |
| C |

|  |
| --- |
| EM |

|  |
| --- |
| H |

|  |
| --- |
| G |

(1) (2)

47.

|  |
| --- |
| K |

|  |
| --- |
| J |

|  |
| --- |
| F |

|  |
| --- |
| C |

|  |
| --- |
| (2) |

|  |
| --- |
| E |

|  |
| --- |
| A |

|  |
| --- |
| B |

|  |
| --- |
| L |

|  |
| --- |
| D |

|  |
| --- |
| G |

|  |
| --- |
| H |

|  |
| --- |
| I |

|  |
| --- |
| (1) |

|  |
| --- |
| A |

|  |
| --- |
| K |

|  |
| --- |
| C |

|  |
| --- |
| B |

|  |
| --- |
| H |

|  |
| --- |
| D |

|  |
| --- |
| G |

|  |
| --- |
| E |

|  |
| --- |
| L |

|  |
| --- |
| F |

|  |
| --- |
| J |

|  |
| --- |
| I |

|  |
| --- |
| A |

|  |
| --- |
| I |

|  |
| --- |
| D |

|  |
| --- |
| B |

|  |
| --- |
| E |

|  |
| --- |
| C |

|  |
| --- |
| H |

|  |
| --- |
| F |

|  |
| --- |
| G |

48． （1）

（2）设二叉树的前序遍历序列为P1，P2，…，Pm，中序遍历序列为S1,S2,…，Sm。因为前序遍历是“根左右”，中序遍历是“左根右”，则前序遍历序列中第一个结点是根结点（P1）。到中序序列中查询到Si=P1,根据中序遍历时根结点将中序序列分成两部分的原则，有：

若i=1，即S1=P1，则这时的二叉树没有左子树; 否则，S1,S2,…,Si-1是左子树的中序遍历序列,用该序列和前序序列P2,P3，…，Pi去构造该二叉树的左子树。

若i=m,即Sm=P1,则这时的二叉树没有右子树; 否则,Si+1,Si+2,…，Sm是右子树的中序遍历序列，用该序列和前序序列中Pi+1,Pi+2,…,Pm,去构造该二叉树的右子树。算法描述请参见下面算法设计第56题。

（3）若前序序列是abcd，并非由这四个字母的任意组合（4!=24）都能构造出二叉树。因为以abcd为输入序列，通过栈只能得到1/(n+1)\*2n!/(n!\*n!)=14 种，即以abcd为前序序列的二叉树的数目是14。任取以abcd作为中序遍历序列，并不全能与前序的abcd序列构成二叉树。例如：若取中序列dcab就不能。

该14棵二叉树的形态及中序序列略。

49．不能。因DABC并不是ABCD的合法出栈序列，参照第37、48（3）的解释。

50．先序序列是“根左右” 后序序列是“左右根”，可见对任意结点，若至多只有左子女或至多只有右子女，均可使前序序列与后序序列相反，图示如下：

|  |
| --- |
| 51题图 |

|  |
| --- |
| F |

|  |
| --- |
| E |

|  |
| --- |
| C |

|  |
| --- |
| I |

|  |
| --- |
| A |

|  |
| --- |
| B |

|  |
| --- |
| G |

|  |
| --- |
| D |

|  |
| --- |
| H |

|  |
| --- |
| 50题 |

51．

52．按层次遍历，第一个结点（若树不空）为根，该结点在中序序列中把序列分成左右两部分—左子树和右子树。若左子树不空，层次序列中第二个结点左子树的根；若左子树为空，则层次序列中第二个结点右子树的根。对右子树也作类似的分析。层次序列的特点是：从左到右每个结点或是当前情况下子树的根或是叶子。

|  |
| --- |
| (52)题图 |

|  |
| --- |
| A |

|  |
| --- |
| B |

|  |
| --- |
| D |

|  |
| --- |
| G |

|  |
| --- |
| E |

|  |
| --- |
| C |

|  |
| --- |
| F |

|  |
| --- |
| I |

|  |
| --- |
| H |

|  |
| --- |
| J |

|  |
| --- |
| I |

|  |
| --- |
| (52)题(1) |

|  |
| --- |
| A |

|  |
| --- |
| D |

|  |
| --- |
| E |

|  |
| --- |
| B |

|  |
| --- |
| F |

|  |
| --- |
| C |

|  |
| --- |
| G |

|  |
| --- |
| H |

|  |
| --- |
| (52题(1)对应森林) |

|  |
| --- |
| E |

|  |
| --- |
| C |

|  |
| --- |
| A |

|  |
| --- |
| B |

|  |
| --- |
| D |

|  |
| --- |
| I |

|  |
| --- |
| G |

|  |
| --- |
| H |

|  |
| --- |
| F |

53．森林的先序序列和后序序列对应其转换的二叉树的先序序列和中序序列，应先据此构造二叉树，再构造出森林。

|  |
| --- |
| F |

|  |
| --- |
| A |

|  |
| --- |
| E |

|  |
| --- |
| B |

|  |
| --- |
| D |

|  |
| --- |
| C |

|  |
| --- |
| J |

|  |
| --- |
| H |

|  |
| --- |
| G |

|  |
| --- |
| I |

|  |
| --- |
| K |

|  |
| --- |
| O |

|  |
| --- |
| N |

|  |
| --- |
| L |

|  |
| --- |
| MG |

54．

53题森林

|  |
| --- |
| A |

(54)题图

|  |
| --- |
| A |

|  |
| --- |
| B |

|  |
| --- |
| C |

|  |
| --- |
| D |

|  |
| --- |
| E |

|  |
| --- |
| B |

|  |
| --- |
| D |

|  |
| --- |
| C |

|  |
| --- |
| E |

55．HIDJKEBLFGCA

56.

|  |
| --- |
| G |

|  |
| --- |
| 56 (2) |

|  |
| --- |
| A |

|  |
| --- |
| B |

|  |
| --- |
| C |

|  |
| --- |
| E |

|  |
| --- |
| D |

|  |
| --- |
| H |

|  |
| --- |
| K |

|  |
| --- |
| I |

|  |
| --- |
| F |

|  |
| --- |
| J |

|  |
| --- |
| L |

|  |
| --- |
| 56 (3) |

|  |
| --- |
| 1 |

|  |
| --- |
| 3 |

|  |
| --- |
| 2 |

|  |
| --- |
| 4 |

|  |
| --- |
| 8 |

|  |
| --- |
| 5 |

|  |
| --- |
| 6 |

|  |
| --- |
| 7 |

|  |
| --- |
| 56 (1) |

|  |
| --- |
| A |

|  |
| --- |
| B |

|  |
| --- |
| D |

|  |
| --- |
| K |

|  |
| --- |
| F |

|  |
| --- |
| C |

|  |
| --- |
| J |

|  |
| --- |
| E |

|  |
| --- |
| I |

|  |
| --- |
| H |

|  |
| --- |
| G |

|  |
| --- |
| 56题图 |

|  |
| --- |
| A |

|  |
| --- |
| B |

|  |
| --- |
| C |

|  |
| --- |
| E |

|  |
| --- |
| D |

|  |
| --- |
| G |

|  |
| --- |
| I |

|  |
| --- |
| H |

|  |
| --- |
| F |

|  |
| --- |
| J |

|  |
| --- |
| K |

57．M叉树的前序和后序遍历分别与它转换成的二叉树的先序和中序遍历对应。

58．前序遍历是“根左右”，中序遍历是“左根右”，后序遍历是“左右根”。若将“根”去掉，三种遍历就剩“左右”。三种遍历中的差别就是访问根结点的时机不同。二叉树是递归定义的，对左右子树均是按左右顺序来遍历的，因此所有叶子结点间的先后关系都是相同的。

59．本题的核心是三种遍历的顺序：“根左右”-“左根右”-“左右根”，但对本题的解答必须先定义结点间相互关系的“左右”。本解答中将N是M的左子女，当作N在M 的左边，而N是M的右子女，当作N在M 的右边。若定义P是M和N的最近公共祖先，N在P的左子树中，M在P的右子树中，称N在M 的左边，那时的答案是不一样的。

|  |  |  |  |
| --- | --- | --- | --- |
|  | 先根遍历时n先被访问 | 中根遍历时n先被访问 | 后根遍历时n先被访问 |
| N在M的左边 |  | √ | √ |
| N在M的右边 |  |  | √ |
| N是M的祖先 | √ |  |  |
| N是M的子孙 |  |  | √ |

60．HIDJKEBLFGCA

61．

|  |
| --- |
| I |

|  |
| --- |
| C |

|  |
| --- |
| A |

|  |
| --- |
| B |

|  |
| --- |
| G |

|  |
| --- |
| E |

|  |
| --- |
| D |

|  |
| --- |
| F |

|  |
| --- |
| K |

|  |
| --- |
| J |

|  |
| --- |
| L |

|  |
| --- |
| H |

62．后序遍历的顺序是“左子树—右子树—根结点”。因此，二叉树最左下的叶子结点是遍历的第一个结点。下面的语句段说明了这一过程（设p是二叉树根结点的指针）。

**if**(p!=null)

{**while** (p->lchild!=null || p->rchild!=null)

{**while**(p->lchild!=null) p=p->lchild;

**if**(p->rchild!=null) p=p->rchild; } }

**return**(p); //返回后序序列第一个结点的指针；

63. 采用前序和后序两个序列来判断二叉树上结点n1必定是结点n2的祖先。

在前序序列中某结点的祖先都排在其前。若结点n1是n2的祖先,则n1必定在n2之前。而在后序序列中，某结点的祖先排在其后，即若结点n1是n2的祖先，则n1必在n2之后。根据这条规则来判断若结点n1在前序序列中在n2之前，在后序序列中又在n2之后，则它必是结点n2的祖先。

64.(1) （2） 前序序列：ABCEDFHGIJ （3）后序线索树

中序序列：ECBHFDJIGA

后序序列：ECHFJIGDBA

65. 最后一个递归调用语句所保留的参数没有意义。这类递归因其在算法最后，通常被称为“尾递归”， 可不用栈且将其（递归）消除。 如中序遍历递归算法中，最后的递归语句inorder (bt->rchild)可改为下列语句段：

bt=bt->rchild;

**while** (bt->rchild!=null)

{inorder (bt ->lchild); printf(“%c”,bt ->data);//访问根结点，假定结点数据域为字符

bt=bt ->rchild; }

66.在二叉链表表示的二叉树中， 引入线索的目的主要是便于查找结点的前驱和后继。因为若知道各结点的后继，二叉树的遍历就变成非常简单。二叉链表结构查结点的左右子女非常方便，但其前驱和后继是在遍历中形成的。为了将非线性结构二叉树的结点排成线性序列， 利用结点的空链域，左链为空时用作前驱指针，右链为空时作为后继指针。再引入左右标记ltag 和rtag ，规定ltag=0，lchild 指向左子女,ltag=1时,lchild指向前驱；当rtag=0时 ,rchild指向右子女，rtag=1时，rchild 指向后继。这样，在线索二叉树（特别是中序线索二叉树）上遍历就消除了递归，也不使用栈（后序线索二叉树查后继仍需

要栈。）

67. 

|  |
| --- |
| 67题(2) |

|  |
| --- |
| M |

|  |
| --- |
| K |

|  |
| --- |
| L |

|  |
| --- |
| N |

|  |
| --- |
| P |

|  |
| --- |
| O |

|  |
| --- |
| I |

|  |
| --- |
| G |

|  |
| --- |
| E |

|  |
| --- |
| F |

|  |
| --- |
| H |

|  |
| --- |
| J |

|  |
| --- |
| (3)后根遍历森林，结点序列为:  BDCAIFJGHELOPMNK |

|  |
| --- |
| C |

|  |
| --- |
| A |

|  |
| --- |
| B |

|  |
| --- |
| D |

67题(1)

68. （1）前序序列：ABDEHCFG

（2）中序序列：DHEBAFCG

（3）后序序列：HEDBFGCA



69.（1）

（2）BiTree INORDER-PRIOR(N,X) //在中序线索二叉树上查找结点N的前驱结点X

{**if**(n->ltag==1){X=N->lchild; **return** (X);}

**else** {p=N->lchild;

**while** (p->rtag==0) p=p->rchild;

X=p;**return**(p);} }

70.



|  |
| --- |
| G |

|  |
| --- |
| B |

|  |
| --- |
| D |

|  |
| --- |
| A |

|  |
| --- |
| I |

|  |
| --- |
| F |

|  |
| --- |
| H |

|  |
| --- |
| E |

|  |
| --- |
| C |

|  |
| --- |
| 70题（3） |

71.

|  |
| --- |
| 前序序列：ＡＢＣＥＤＦＨＧＩJ  中序序列： E C B H F D J I G A  后序序列： ＥＣＨＦＪＩＧＤＢＡ    71题（2） |

71题（3）

72. 后序线索树中结点的后继，要么是其右线索（当结点的rtag=1时），要么是其双亲结点右子树中最左下的叶子结点。所以，只有当二叉树只有根或树中任一结点均无右子树时，进行遍历才不用栈，其遍历程序段如下：

**while**（p->ltag==0）p==p->lchild; //找最左下叶子结点

**while**（p->rchild!=null）{visit(p->data); //访问结点；

p=p->rchild;} //沿线索向上

对前序线索二叉树，当二叉树非空时，若其结点有左子女（ltag=0），左子女是后继；否则，若结点有右子女（rtag=0），则右子女是后继；若无右子女（rtag=1），右子女是线索，也指向后继。所以，对任何前序线索二叉树进行前序遍历均不需使用栈。

73．左右子树均不空的二叉树先序线索化后，空指针域为1个（最后访问结点的右链为空）。

74．**if**(p->ltag==0) **return**(p->lchild);//左子女不空，左子女为直接后继结点

**else** **return**(p->rchild); //左子女空，右子女（或右线索）为后继

75. 后序线索树中结点的后继（根结点无后继），要么是其右线索（当结点的rtag=1时），要么是其双亲结点右子树中最左下的叶子结点。对中序线索二叉树某结点，若其左标记等于1,则左子女为线索，指向直接前驱；否则，其前驱是其左子树上按中序遍历的最后一个结点。

76. 树的后根遍历（对应二叉树的中序遍历）全线索链表

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Data | A | B | C | D | E | F | G | H | I | J | K |
| Ltag | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 |
| Fch | 2 | null | 5 | 7 | 8 | 5 | 11 | 2 | 8 | 9 | 3 |
| Rtag | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| Nsib | null | 3 | 4 | 1 | 6 | 3 | 4 | 9 | 10 | 5 | 7 |

 77.

虽然哈夫曼树的带权路径长度是唯一的，但形态不唯一。本题中各字母编码如下：c1:0110 c2:10 c3:0010 c4:0111 c5:000 c6:010 c7:11 c8:0011

78. 字符A，B，C，D出现的次数为9,1,5,3。其哈夫曼编码如下A:1，B:000，C:01，D:001

79．（2）wpl=(2+3)\*5+6\*4+(9+14+15)\*3+(16+17)\*2=229



|  |
| --- |
| 3 |

|  |
| --- |
| 5 |

|  |
| --- |
| 9 |

|  |
| --- |
| 0 |

|  |
| --- |
| 0 |

|  |
| --- |
| 0 |

|  |
| --- |
| 1 |

|  |
| --- |
| 1 |

|  |
| --- |
| 1 |

|  |
| --- |
| 78题哈夫曼编码树 |

(3) 编码为：15:111, 3:10101, 14:110, 2:10100, 6:1011, 9:100, 16:00, 17:01

(4) 常用哈夫曼树为通讯用的字符编码，本题中集合的数值解释为字符发生的频率（次数）。由哈夫曼树构造出哈夫曼编码。译码时，进行编码的“匹配”，即从左往右扫描对方发来的“编码串”，用字符编码去匹配，得到原来的元素（本题中的数）。

80．首先确定是否需加“虚权值”（即权值为0）,对m个权值，建k叉树，若（m-1）%(k-1)=0,则不需要加“虚权值”，否则，第一次归并时需（m-1）%(k-1)+1个权值归并。建立k叉树的过程如下：

（1）将m个权值看作m棵只有根结点的k叉树的集合F={T1，T2，…，Tm}。

（2）从F中选k（若需加虚权值，则第一次选（m-1）%(k-1)+1）个权值最小的树作子树，构成一棵k叉树，k叉树根结点的权值为所选的k个树根结点权值之和，在F中

删除这k棵子树，并将新k叉树加入到F中。

（3）从F中选k个权值最小的树作子树，构成一棵k叉树，其根

结点权值等于所选的k棵树根结点权值之和，在F中删除这k棵树，

并将新得到的树加到F中。

(4) 重复（3），直到F中只有一棵树为止，这就是最优的k叉树。

对本题10个权值，构造最优三叉树。

因（10-1）%(3-1)=1，

所以第一次用2个权值合并。

最小加权路径长度：

（1+4）\*4+(9+16)\*3+(25+36+49+64+81)\*2+100\*1=705

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 字符 | weight | Parent | lch | rch |
| 1 | A | 3 | 0 | 0 | 0 |
| 2 | B | 12 | 0 | 0 | 0 |
| 3 | C | 7 | 0 | 0 | 0 |
| 4 | D | 4 | 0 | 0 | 0 |
| 5 | E | 2 | 0 | 0 | 0 |
| 6 | F | 8 | 0 | 0 | 0 |
| 7 | G | 11 | 0 | 0 | 0 |
| 8 |  |  | 0 | 0 | 0 |
| 9 |  |  | 0 | 0 | 0 |
| 10 |  |  | 0 | 0 | 0 |
| 11 |  |  | 0 | 0 | 0 |
| 12 |  |  | 0 | 0 | 0 |
| 13 |  |  | 0 | 0 | 0 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 字符 | weight | parent | lch | rch |
| 1 | A | 3 | 8 | 0 | 0 |
| 2 | B | 12 | 12 | 0 | 0 |
| 3 | C | 7 | 10 | 0 | 0 |
| 4 | D | 4 | 9 | 0 | 0 |
| 5 | E | 2 | 8 | 0 | 0 |
| 6 | F | 8 | 10 | 0 | 0 |
| 7 | G | 11 | 11 | 0 | 0 |
| 8 |  | 5 | 9 | 5 | 1 |
| 9 |  | 9 | 11 | 4 | 8 |
| 10 |  | 15 | 12 | 3 | 6 |
| 11 |  | 20 | 13 | 9 | 7 |
| 12 |  | 27 | 13 | 2 | 10 |
| 13 |  | 47 | 0 | 11 | 12 |

81．初态图 终态图

82．前缀码是一编码不是任何其它编码前缀的编码。例如，0和01就不是前缀码，因为编码0是编码01的前缀。仅从编码来看，0和01 是前缀码，但因历史的原因，它不被称为前缀码，而是把一编码不是另一编码前缀的编码称为前缀码。

利用二叉树可以构造前缀码，例如，以A，B，C，D为叶子可构成二叉树，将左分枝解释为0，右分枝解释成1，从根结点到叶子结点的0、1串就是叶子的前缀码。用哈夫曼树可构造出最优二叉树，使编码长度最短。

83．哈夫曼树只有度为0的叶子结点和度为2的分枝结点，设数量分别为n0和n2，则树的结点数n为n=n0+n2。 另根据二叉树性质：任意二叉树中度为0的结点数n0和度为2的结点数n2间的关系是n2=n0-1,代入上式，则n=n0+n2=2n0-1。

84．（1）T树的最大深度Kmax=6（除根外，每层均是两个结点）

T树的最小深度Kmin=4（具有6个叶子的完全二叉树是其中的一种形态）

（2）非叶子结点数是5。（n2=n0-1） （3）哈夫曼树见下图，其带权路径长度wpl=51



85．（1）错误，循环结束条件top=0不能满足，因为在top>1情况下，执行top:=top-1

(2)错误 （3）错误 （4）正确 （5）结点的深度与其右孩子深度相同，比左孩子深度少1。

五．算法设计题

1．[题目分析]以二叉树表示算术表达式，根结点用于存储运算符。若能先分别求出左子树和右子树表示的子表达式的值，最后就可以根据根结点的运算符的要求，计算出表达式的最后结果。

**typedef** **struct** node

{ElemType data; **float** val;

**char** optr; //只取‘+’， ‘-’， ‘\*’，‘/’

**struct** node \*lchild，\*rchild }BiNode,\*BiTree;

**float** PostEval(BiTree bt) // 以后序遍历算法求以二叉树表示的算术表达式的值

{**float** lv,rv;

**if**(bt!=null)

{lv=PostEval(bt->lchild); // 求左子树表示的子表达式的值

rv=PostEval(bt->rchild); // 求右子树表示的子表达式的值

**switch**(bt->optr)

{**case** ‘+’: value=lv+rv; **break**;

**case** ‘-’: value=lv-rv;**break**;

**case** ‘\*’: value=lv\*rv;**break**;

**case** ‘/’: value=lv/rv;

} } **return**(value); }

2.[题目分析] 本题是将符号算术表达式用二叉树表示的逆问题，即将二叉树表示的表达式还原成原表达式。二叉树的中序遍历序列与原算术表达式基本相同，差别仅在于二叉树表示中消除了括号。将中序序列加上括号就恢复原貌。当根结点运算符优先级高于左子树（或右子树）根结点运算符时，就需要加括号。

**int** Precede(**char** optr1,optr2)

// 比较运算符级别高低，optr1级别高于optr2时返回1，相等时返回0，低于时返回-1

{**switch**(optr1)

{**case**‘+’:**case**‘-’:**if**(optr2==‘+’||optr2==‘-’)**return**(0);**else** **return**(-1);

**case**‘\*’:**case**‘/’:**if**(optr1==‘\*’||optr2==‘/’)**return**(0);**else** **return**(1);

} }

**void** InorderExp (BiTree bt)

//输出二叉树表示的算术表达式，设二叉树的数据域是运算符或变量名

{**int** bracket;

**if**(bt)

{**if**(bt->lchild!=null)

{bracket=Precede(bt->data,bt->lchild->data)//比较双亲与左子女运算符优先级

**if**(bracket==1) printf(‘(’)；

InorderExp(bt->lchild); //输出左子女表示的算术表达式

**if**(bracket==1)printf(‘)’)； //加上右括号

}

printf(bt->data); //输出根结点

**if**(bt->rchild!=null) //输出右子树表示的算术表达式

{bracket=Precede(bt->data,bt->rchild->data)

**if** (bracket==1)printf(“(”)； //右子女级别低，加括号

InorderExp (bt->rchild);

**if**(bracket==1)printf(“)”)；

} }

}//结束Inorder Exp

3．[题目分析]首先通过对二叉树后序遍历形成后缀表达式，这可通过全局变量的字符数组存放后缀表达式；接着对后缀表达式求值，借助于一栈存放运算结果。从左到右扫描后缀表达式，遇操作数就压入栈中，遇运算符就从栈中弹出两个操作数，作运算符要求的运算，并把运算结果压入栈中，如此下去，直到后缀表达式结束，这时栈中只有一个数，这就是表达式的值。

**char** ar[maxsize];//maxsize是后缀表达式所能达到的最大长度

**int** i=1;

**void** PostOrder(BiTree t )//后序遍历二叉树t,以得到后缀表达式

{**if**(t)

{PostOrder(t->lchild); PostOrder(b->rchild)；ar[i++]=b->data; }

}//结束PostOrder

**void** EXPVALUE()

//对二叉树表示的算术表达式，进行后缀表达式的求值

{ar[i]=‘\0’; //给后缀表达式加上结束标记

**char** value[]; //存放操作数及部分运算结果

i=1; ch=ar[i++];

**while**(ch!=‘\0’)

{**switch**(ch)

{**case** ch in op: opnd1=pop(value);opnd2=pop(value); //处理运算符

push(operate(opnd2,ch,opnd1));**break**;

**default**: push(value,ch); //处理操作数，压入栈中

}

ch=ar[i++]; //读入后缀表达式

} printf(value[1]); //栈中只剩下一个操作数，即运算结束。

} //结束EXPVALUE

[算法讨论] 根据题意，操作数是单字母变量，存放运算结果的栈也用了字符数组。实际上，操作数既可能是变量，也可以是常量。运算中，两个操作数（opnd1 和opnd2）也不会直接运算，即两个操作数要从字符转换成数（如‘3’是字符，而数值3=‘3’-‘0’）。数在压入字符栈也必须转换，算法中的operate也是一个需要编写的函数，可参见上面算法设计题1，其细节不再深入讨论。

4．[题目分析]当森林（树）以孩子兄弟表示法存储时，若结点没有孩子（fch=null）,则它必是叶子，总的叶子结点个数是孩子子树（fch）上的叶子数和兄弟（nsib）子树上叶结点个数之和。

**typedef** **struct** node

{ElemType data;//数据域

**struct** node \*fch, \*nsib;//孩子与兄弟域 }\*Tree;

**int** Leaves (Tree t)

//计算以孩子-兄弟表示法存储的森林的叶子数

{**if**(t)

**if**(t->fch==null) //若结点无孩子，则该结点必是叶子

**return**(1+Leaves(t->nsib)); //返回叶子结点和其兄弟子树中的叶子结点数

**else** **return** (Leaves(t->fch)+Leaves(t->nsib)); //孩子子树和兄弟子树中叶子数之和

}//结束Leaves

5．[题目分析]由指示结点i 左儿子和右儿子的两个一维数组L[i]和R[i]，很容易建立指示结点i 的双亲的一维数组T[i]，根据T数组，判断结点U是否是结点V后代的算法，转为判断结点V是否是结点U的祖先的问题。

**int** Generation (**int** U,V,N,L[],R[],T[])

//L[]和R[]是含有N个元素且指示二叉树结点i左儿子和右儿子的一维数组，

//本算法据此建立结点i的双亲数组T，并判断结点U是否是结点V的后代。

{**for**(i=1;i<=N;i++) T[i]=0; //T数组初始化

**for** (i=1;i<=N;i++) //根据L和R填写T

**if**(L[i]!=0) T[L[i]]=i; //若结点i的左子女是L，则结点L的双亲是结点i

**for**(i=1;i<=N;i++)

**if** (R[i]!=0) T[R[i]]=i; //i的右子女是r，则r的双亲是i

**int** parent=U; //判断U是否是V的后代

**while** (parent!=V && parent!=0) parent=T[parent];

**if** (parent==V){printf(“结点U是结点V的后代”);**return**(1);}

**else**{ printf(“结点U不是结点V 的后代”);**return**(0);}

}结束Generation

6．[题目分析]二叉树是递归定义的，以递归方式建立最简单。判定是否是完全二叉树，可以使用队列，在遍历中利用完全二叉树“若某结点无左子女就不应有右子女”的原则进行判断。

BiTree Creat() //建立二叉树的二叉链表形式的存储结构

{ElemType x；BiTree bt;

scanf(“%d”,&x); //本题假定结点数据域为整型

**if**(x==0) bt=null;

**else** **if**(x>0)

{bt=(BiNode \*)malloc(**sizeof**(BiNode));

bt->data=x; bt->lchild=creat(); bt->rchild=creat();

}

**else** error(“输入错误”)；

**return**(bt);

}//结束 BiTree

**int** JudgeComplete(BiTree bt) //判断二叉树是否是完全二叉树,如是，返回1，否则，返回0

{**int** tag=0; BiTree p=bt, Q[]; // Q是队列，元素是二叉树结点指针，容量足够大

**if**(p==null) **return** (1);

QueueInit(Q); QueueIn(Q,p); //初始化队列，根结点指针入队

**while** (!QueueEmpty(Q))

{p=QueueOut(Q); //出队

**if** (p->lchild && !tag) QueueIn(Q,p->lchild); //左子女入队

**else** {**if** (p->lchild) **return** 0; //前边已有结点为空，本结点不空

**else** tag=1; //首次出现结点为空

**if** (p->rchild && !tag) QueueIn(Q,p->rchild); //右子女入队

**else** **if** (p->rchild) **return** 0; **else** tag=1;

} //**while**

**return** 1; } //JudgeComplete

[算法讨论]完全二叉树证明还有其它方法。判断时易犯的错误是证明其左子树和右子数都是完全二叉树，由此推出整棵二叉树必是完全二叉树的错误结论。

7．BiTree Creat(ElemType A[],**int** i)

//n个结点的完全二叉树存于一维数组A中，本算法据此建立以二叉链表表示的完全二叉树

{BiTree tree;

**if** (i<=n){tree=(BiTree)malloc(**sizeof**(BiNode)); tree->data=A[i];

**if**(2\*i>n) tree->lchild=null；**else** tree->lchild=Creat(A,2\*i)；

**if**(2\*i+1>n) tree->rchild=null；**else** tree->rchild=Creat(A,2\*i+1)； }

**return** (tree)； }//Creat

[算法讨论]初始调用时,i=1。

8. [题目分析]二叉树高度可递归计算如下：若二叉树为空，则高度为零，否则，二叉树的高度等于左右子树高度的大者加1。这里二叉树为空的标记不是null而是0。设根结点层号为1，则每个结点的层号等于其双亲层号加1。

现将二叉树的存储结构定义如下:

**typedef** **struct** node

{**int** L[];//编号为i的结点的左儿子

**int** R[];//编号为i的结点的右儿子

**int** D[];//编号为i的结点的层号

**int** i; //存储结点的顺序号（下标）

}tnode;

（1）**int** Height(tnode t,**int** i)//求二叉树高度，调用时i=1

{**int** lh,rh;

**if** (i==0) **return** (0);

**else**{lh=Height(t,t.L[i]); rh=Height(t,t.R[i]);

**if**(lh>rh) **return**(lh+1); **else** **return**(rh+1);

}

}//结束Height

（2）**int** Level（tnode t）//求二叉树各结点的层号，已知编号为1的结点是根，且层号为1

{t.D[1]=1;

**for**(i=1;i<=n;i++) {depth=t.D[i]； //取出根结点层号

**if**(t.L[i]!=0) t.D[t.L[i]]=depth+1; //i结点左儿子层号

**if**(t.R[i]!=0) t.D[t.R[i]]=depth+1; }//i结点右儿子层号

}结束level

9.[题目分析]二叉树采用顺序存储结构（一维数组）是按完全二叉树的形状存储的，不是完全二叉树的二叉树顺序存储时，要加“虚结点”。数组中的第一个元素是根结点。本题中采用队列结构。

**typedef** **struct**

{BiTree bt; //二叉树结点指针

**int** num; }tnode // num是结点在一维数组中的编号

tnode Q[maxsize]; //循环队列,容量足够大

**void** creat(BiTree T,ElemType BT[ ])

//深度h的二叉树存于一维数组BT[1:2h-1]中，本算法生成该二叉树的二叉链表存储结构

{tnode tq; //tq是队列元素

**int** len=2h-1; //数组长度

T=(BiTree)malloc(**sizeof**(BiNode)); //申请结点

T->data=BT[1]; //根结点数据

tq.bt=T; tq.num=1;

Q[1]=tq; //根入队列

front=0;rear=1; //循环队列头、尾指针

**while**(front!=rear) //当队列不空时循环

{front=(front+1) % maxsize ;

tq=Q[front] ; p=tq.bt; i=tq.num ; //出队，取出结点及编号

**if** (BT[2\*i]==‘#’||2\*i>len) p->lchild=null; //左子树为空，‘#’表示虚结点

**else** //建立左子女结点并入队列

{p->lchild=(BiTree) malloc(**sizeof**(BiNode)); //申请结点空间

p->lchild🡪data=BT[2\*i]; // 左子女数据

tq.bt=p->lchild; tq.num=2\*i; rear=(rear+1) % maxsize ;//计算队尾位置

Q[rear]=tq; //左子女结点及其编号入队

}

**if**(BT[2\*i+1]==‘#’|| 2\*i+1>len) p->rchild=null; //右子树为空

**else** //建立右子女结点并入队列

{p->rchild=(BiTree)malloc(**sizeof** (BiNode); //申请结点空间

p->rchild->data=BT[2\*i+1]; tq.bt=p->rchild; tq.num=2\*i+1;

rear=(rear+1)%maxsize; Q[rear]=tq; //计算队尾位置,右子女及其编号入队

}

} //**while**

}//结束creat

[算法讨论] 本题中的虚结点用‘#’表示。应根据二叉树的结点数据的类型而定。

10．[题目分析]本题静态链表中结点是按动态二叉链表的前序遍历顺序存放的。首先对动态二叉链表的二叉树进行前序遍历，填写静态链表的“下标”和data域，再对动态二叉链表的二叉树进行层次遍历，设队列Q，填写静态链表的lchild域和rchild域。

**typedef** **struct** node //静态链表结点结构

{ElemType data; //结点数据

**int** row,lchild,rchild ; //下标，左右子女

}component；

component st[]; //st容量足够大

**struct** node {BiTree t; **int** idx; }qbt;

**static** **int** num=0;

**void** PreOrder(BiTree bt);

// 前序遍历二叉树，填写静态链表的“下标”和data域

{**if** (bt)

{st[++num].data=bt->data; st[num].row=num;

PreOrder(bt->lchild); PreOrder(bt->rchild);

} }

**int** Locate(ElemType x)

//在静态链表中查二叉树结点的下标

{**for** (i=1;i<=num;i++) **if** (st[i].data==x) **return** (i);

}

**void** DynaToST (BiTree bt) //将二叉树的动态二叉链表结构转为静态链表结构

{**int** i=0; //i为静态链表st的下标

**if** (bt!=null)

{QueueInit(Q); //Q是队列，容量足够大，队列元素是qbt

qbt.t=bt; qbt.idx=1; QueueIn(Q,qbt); st[1].data=bt->data;

**while**(!QueueEmpty(Q))

{qbt=QueueOut(Q); //出队列

p=qbt.t; i=qbt.idx; //二叉树结点及在静态链表中的下标

**if** (p->lchild!=null) //若左子女存在,查其在静态链表中的下标, 填lchild域值

{lch=Locate(p->lchild->data)；st[i].lchild=lch;

qbt.t=p->lchild; qbt.idx=lch; QueueIn(Q,qbt); }

**else** st[i].lchild=0; //无左子女，其lchild域填0

**if** (p->rchild!=null) //若右子女存在,查其在静态链表中的下标，填rchild域值

{rch=Locate(p->->rchild->data)；st[i].rchild=rch;

qbt.t=p->rchild; qbt.idx=rch; QueueIn(Q,qbt); }

**else** st[i].rchild=0; //无左子女，其lchild域填0

}//**while**

}//结束DynaToST

11. [题目分析] 由于以双亲表示法作树的存储结构，找结点的双亲容易。因此我们可求出每一结点的层次，取其最大层次就是树有深度。对每一结点，找其双亲，双亲的双亲，直至（根）结点双亲为0为止。

**int** Depth(Ptree t) //求以双亲表示法为存储结构的树的深度，Ptree的定义参看教材

{**int** maxdepth=0;

**for**(i=1;i<=t.n;i++)

{temp=0; f=i;

**while**(f>0) {temp++; f=t.nodes[f].parent; } // 深度加1,并取新的双亲

**if**(temp>maxdepth) maxdepth=temp; //最大深度更新

}

**return**(maxdepth);//返回树的深度

} //结束Depth

12. [题目分析] 二叉树是递归定义的，其运算最好采取递归方式

**int** Height(btre bt)//求二叉树bt的深度

{**int** hl,hr;

**if** (bt==null) **return**(0);

**else** {hl=Height(bt->lch); hr=Height(bt->rch);

**if**(hl>hr) **return** (hl+1); **else** **return**(hr+1);

} }

13．[题目分析] 求二叉树高度的算法见上题。求最大宽度可采用层次遍历的方法，记下各层结点数，每层遍历完毕，若结点数大于原先最大宽度，则修改最大宽度。

**int** Width(BiTree bt)//求二叉树bt的最大宽度

{**if** (bt==null) **return** (0); //空二叉树宽度为0

**else**

{BiTree Q[];//Q是队列，元素为二叉树结点指针，容量足够大

front=1;rear=1;last=1;//front队头指针,rear队尾指针,last同层最右结点在队列中的位置

temp=0; maxw=0; //temp记局部宽度, maxw记最大宽度

Q[rear]=bt; //根结点入队列

**while**(front<=last)

{p=Q[front++]; temp++; //同层元素数加1

**if** (p->lchild!=null) Q[++rear]=p->lchild; //左子女入队

**if** (p->rchild!=null) Q[++rear]=p->rchild; //右子女入队

**if** (front>last) //一层结束，

{last=rear;

**if**(temp>maxw) maxw=temp;//last指向下层最右元素, 更新当前最大宽度

temp=0;

}//**if**

}//**while**

**return** (maxw);

}//结束width

14．[题目分析]由孩子兄弟链表表示的树，求高度的递归模型是：若树为空，高度为零；若第一子女为空，高度为1和兄弟子树的高度的大者；否则，高度为第一子女树高度加1和兄弟子树高度的大者。其非递归算法使用队列，逐层遍历树，取得树的高度。

**int** Height(CSTree bt) //递归求以孩子兄弟链表表示的树的深度

{**int** hc,hs;

**if** (bt==null) **return** (0);

**else** **if** (!bt->firstchild) **return** (1+height(bt->nextsibling);//子女空，查兄弟的深度

**else** // 结点既有第一子女又有兄弟，高度取子女高度+1和兄弟子树高度的大者

{hc=height(bt->firstchild)； //第一子女树高

hs=height(bt->nextsibling)；//兄弟树高

**if**(hc+1>hs)**return**(hc+1); **else** **return** (hs);

}

}//结束height

**int** height(CSTree t) //非递归遍历求以孩子兄弟链表表示的树的深度

{**if**(t==null) **return**(0);

**else**{**int** front=1,rear=1; //front,rear是队头队尾元素的指针

**int** last=1,h=0; //last指向树中同层结点中最后一个结点，h是树的高度

Q[rear]=t; //Q是以树中结点为元素的队列

**while**(front<=last)

{t=Q[front++]; //队头出列

**while**(t!=null) //层次遍历

{**if** (t->firstchild) Q[++rear]=t->firstchild; //第一子女入队

t=t->nextsibling; //同层兄弟指针后移

}

**if**(front>last) //本层结束，深度加1（初始深度为0）

{h++;last=rear;} //last再移到指向当前层最右一个结点

}//**while**(front<=last)

}//**else**

}//Height

15.[题目分析]后序遍历最后访问根结点，即在递归算法中，根是压在栈底的。采用后序非递归算法，栈中存放二叉树结点的指针，当访问到某结点时，栈中所有元素均为该结点的祖先。本题要找p和q 的最近共同祖先结点r ,不失一般性，设p在q的左边。后序遍历必然先遍历到结点p，栈中元素均为p的祖先。将栈拷入另一辅助栈中。再继续遍历到结点q时，将栈中元素从栈顶开始逐个到辅助栈中去匹配，第一个匹配（即相等）的元素就是结点p 和q的最近公共祖先。

**typedef** **struct**

{BiTree t;**int** tag;//tag=0 表示结点的左子女已被访问，tag=1表示结点的右子女已被访问

}stack;

stack s[],s1[];//栈，容量够大

BiTree Ancestor(BiTree ROOT,p,q,r)//求二叉树上结点p和q的最近的共同祖先结点r。

{top=0; bt=ROOT;

**while**(bt!=null ||top>0)

{**while**(bt!=null && bt!=p && bt!=q) //结点入栈

{s[++top].t=bt; s[top].tag=0; bt=bt->lchild;} //沿左分枝向下

**if**(bt==p) //不失一般性，假定p在q的左侧,遇结点p时，栈中元素均为p的祖先结点

{**for**(i=1;i<=top;i++) s1[i]=s[i]; top1=top; }//将栈s的元素转入辅助栈s1 保存

**if**(bt==q) //找到q 结点。

**for**(i=top;i>0;i--)//；将栈中元素的树结点到s1去匹配

{pp=s[i].t;

**for** (j=top1;j>0;j--)

**if**(s1[j].t==pp) {printf(“p 和q的最近共同的祖先已找到”)；**return** (pp);}

｝

**while**(top!=0 && s[top].tag==1) top--; //退栈

**if** (top!=0)｛s[top].tag=1;bt=s[top].t->rchild;｝ //沿右分枝向下遍历

}//结束**while**(bt!=null ||top>0)

**return**(null);//ｑ、p无公共祖先

｝//结束Ancestor

16.[题目分析]二叉树顺序存储，是按完全二叉树的格式存储，利用完全二叉树双亲结点与子女结点编号间的关系，求下标为i和j的两结点的双亲，双亲的双亲，等等，直至找到最近的公共祖先。

**void** Ancestor(ElemType A[],**int** n,i,j)

//二叉树顺序存储在数组A[1..n]中，本算法求下标分别为i和j的结点的最近公共祖先结点的值。

{**while**(i!=j)

**if**(i>j) i=i/2; //下标为i的结点的双亲结点的下标

**else** j=j/2; //下标为j的结点的双亲结点的下标

printf(“所查结点的最近公共祖先的下标是%d，值是%d”,i,A[i]);//设元素类型整型。

}// Ancestor

17.[题目分析]用二叉树表示出父子，夫妻和兄弟三种关系，可以用根结点表示父（祖先），根结点的左子女表示妻，妻的右子女表示子。这种二叉树可以看成类似树的孩子兄弟链表表示法；根结点是父，根无右子女，左子女表示妻，妻的右子女（右子女的右子女等）均可看成兄弟（即父的所有儿子），兄弟结点又成为新的父，其左子女是兄弟（父的儿子）妻，妻的右子女（右子女的右子女等）又为儿子的儿子等等。首先递归查找某父亲结点，若查找成功，则其左子女是妻，妻的右子女及右子女的右子女等均为父亲的儿子。

BiTree Search(BiTree t,ElemType father)//在二叉树上查找值为father的结点

{**if**(t==null) **return** (null); //二叉树上无father结点

**else** **if**(t->data==father) **return**(t); //查找成功

p=Search(t->lchild,father); p=Search(t->rchild,father); }

}//结束Search

**void** PrintSons(BiTree t,ElemType p) //在二叉树上查找结点值为p的所有的儿子

{p=Serach(t,p); //在二叉树t上查找父结点p

**if**(p!=null) //存在父结点

{q=p->lchild; q=q->rchild; //先指向其妻结点，再找到第一个儿子

**while**(q!=null) {printf(q->data); q=q->rchild;} //输出父的所有儿子

}

}//结束PrintSons

18.[题目分析]后序遍历最后访问根结点，当访问到值为x的结点时，栈中所有元素均为该结点的祖先。

**void** Search(BiTree bt,ElemType x) //在二叉树bt中，查找值为x的结点，并打印其所有祖先

{**typedef** **struct**

{BiTree t; **int** tag; }stack;//tag=0表示左子女被访问，tag=1表示右子女被访问

stack s[]; //栈容量足够大

top=0;

**while**(bt!=null||top>0)

{**while**(bt!=null && bt->data!=x) //结点入栈

{s[++top].t=bt; s[top].tag=0; bt=bt->lchild;} //沿左分枝向下

**if**(bt->data==x){ printf(“所查结点的所有祖先结点的值为:\n”); //找到x

**for**(i=1;i<=top;i++) printf(s[i].t->data); **return**; } //输出祖先值后结束

**while**(top!=0 && s[top].tag==1) top--; //退栈（空遍历）

**if**(top!=0) {s[top].tag=1;bt=s[top].t->rchild;} //沿右分枝向下遍历

}// **while**(bt!=null||top>0)

}结束search

因为查找的过程就是后序遍历的过程，使用的栈的深度不超过树的深度，算法复杂度为O(logn)。

19.[题目分析] 先序遍历二叉树的非递归算法，要求进栈元素少，意味着空指针不进栈。

**void** PreOrder(Bitree bt)//对二叉数bt进行非递归遍历

{**int** top=0; Bitree s[]; //top是栈s的栈顶指针，栈中元素是树结点指针，栈容量足够大

**while**(bt!=null || top>0)

{**while**(bt!=null)

{printf(bt->data); //访问根结点

**if**(bt->rchlid) s[++top]=bt->rchild; //若有右子女，则右子女进栈

bt=bt->lchild; }

**if** (top>0) bt=s[top--];

}

本题中的二叉树中需进栈的元素有 C,H,K,F。

20.[题目分析]二叉树的顺序存储是按完全二叉树的顺序存储格式，双亲与子女结点下标间有确定关系。对顺序存储结构的二叉树进行遍历，与二叉链表类似。在顺序存储结构下，判二叉树为空时，用结点下标大于n（完全二叉树）或0（对一般二叉树的“虚结点”）。本题是完全二叉树。

**void** PreOrder(ElemType bt,**int** n)//对以顺序结构存储的完全二叉树bt进行前序遍历

{**int** top=0,s[]; //top是栈s的栈顶指针，栈容量足够大

**while**(i<=n||top>0)

{**while**(i<=n)

{ printf(bt[i]); //访问根结点；

**if** (2\*i+1<=n) s[++top]=2\*i+1; //右子女的下标位置进栈

i=2\*i; } //沿左子女向下

**if**(top>0) i=s[top--]; } //取出栈顶元素

}//结束PreOrder

21.[题目分析] 本题使用的存储结构是一种双亲表示法，对每个结点，直接给出其双亲（的下标），而且用正或负表示该结点是双亲的右子女或左子女。该类结构不适于直接进行前序遍历（即若直接前序遍历，算法要很复杂），较好的办法是将这类结构转为结点及其左右子女的顺序存储结构，即

Tree2=ARRAY[1..max] OF RECORD data: **char**; //结点数据

lc,rc: integer; END;//结点的左右子女在数组中的下标

**void** Change (Tree t, Tree2 bt, **int** \*root) //先将t转为如上定义类型的变量bt;

{**for**(i=1;i<=max;i++) {bt[i].lc=bt[i].rc=0;} //先将结点的左右子女初始化为0

**for**(i=1;i<=max;i++) //填入结点数据，和结点左右子女的信息

{bt[i].data=t[i].data;

**if**(t[i].parent<0) bt[-t[i].parent].lc=i; //左子女

**else** **if**(t[i].parent>0) bt[t[i].parent].rc=i; //右子女

**else** \*root=i; //root记住根结点

} }//change

**void** PreOrder(Tree2 bt) //对二叉树进行前序遍历

{**int** \*root,top=0; **int** s[]; //s是栈

change(t,bt,root); **int** i=\*root;

**while**(i!=0||top>0)

{**while** (i!=0)

{printf (bt[i].data)；**if**(bt[i].rc!=0) s[++top]=bt[i].rc; //右子女进栈

i=bt[i].lc;

}

**if** (top>0) i=s[top--];

} }//结束preorder

[算法讨论]本题的前序递归算法如下

**void** PreOrder(**int** root)//root是二叉树根结点在顺序存储中的下标，本算法前序遍历二叉树bt

{**if**(root!=0){printf(bt[root].data);//访问根结点

PreOrder(bt[root].lc);//前序遍历左子树

PreOrder(bt[root].rc);//前序遍历右子树

} }//结束preorder，初始调用时，root是根结点的下标

这类问题的求解方法值得注意。当给定数据存储结构不合适时，可由已给结构来构造好的数据结构，以便于运算。象上面第5题也是这样，先根据L和R数组，构造一个结点的双亲的数组T。

22．[题目分析]二叉树先序序列的最后一个结点，若二叉树有右子树，则是右子树中最右下的叶子结点；若无右子树，仅有左子树，则是左子树最右下的叶子结点；若二叉树无左右子树，则返回根结点。

BiTree LastNode(BiTree bt)//返回二叉树bt先序序列的最后一个结点的指针

{BiTree p=bt;

**if**(bt==null) **return**(null);

**else** **while**(p)

**if** (p->rchild) p=p->rchild; //若右子树不空，沿右子树向下

**else** **if** (p->lchild) p=p->lchild; //右子树空，左子树不空，沿左子树向下

**else** **return**(p); //p即为所求

}//结束lastnode

23.[题目分析]高度为K的二叉树，按顺序方式存储，要占用2K –1个存储单元，与实际结点个数n关系不大，对不是完全二叉树的二叉树，要增加“虚结点”，使其在形态上成为完全二叉树。

**int** m=2K –1; //全局变量

**void** PreOrder(ElemType bt[],i )

//递归遍历以顺序方式存储的二叉树bt， i是根结点下标（初始调用时为1）。

{**if** (i<=m) //设虚结点以0表示

{printf(bt[i])； //访问根结点

**if**(2\*i<=m && bt[2\*i]!=0) PreOrder(bt,2\*i); //先序遍历左子树

**if**(2\*i+1<=m && bt[2\*i+1]!=0) PreOrder(bt,2\*i+1);// 先序遍历右子树

} }//结束PreOrder

二叉树中最大序号的叶子结点,是在顺序存储方式下编号最大的结点

**void** Ancesstor(ElemType bt[]) //打印最大序号叶子结点的全部祖先

{c=m; **while**(bt[c]==0) c--; //找最大序号叶子结点,该结点存储时在最后

f=c/2; //c的双亲结点f

**while**(f!=0) //从结点c的双亲结点直到根结点，路径上所有结点均为祖先结点

{printf(bt[f]); f=f/2; }//逆序输出，最老的祖先最后输出

} //结束

24. **void** InOrder(BiTree bt)

{BiTree s[],p=bt; //s是元素为二叉树结点指针的栈，容量足够大

**int** top=0;

**while**(p || top>0)

{**while**(p) {s[++top]=p; bt=p->lchild;} //中序遍历左子树

**if**(top>0){p=s[top--]; printf(p->data); p=p->rchild;} //退栈，访问，转右子树

} }

25.[题目分析] 二叉树用顺序方式存储，其遍历方法与用二叉链表方式存储类似。判空时，在二叉链表方式下用结点指针是否等于null，在顺序存储方式下， 一是下标是否是“虚结点”，二是下标值是否超过最大值（高为H的二叉树要有2H-1个存储单元，与实际结点个数关系不大）。当然，顺序存储方式下，要告诉根结点的下标。

**void** InOrder(**int** i) //对顺序存储的二叉树进行中序遍历，i是根结点的下标

{**if**(i!=0)

{InOrder(ar[i].Lc); //中序遍历左子树

printf(ar[i].data); //访问根结点

InOrder(ar[i].Rc); //中序遍历左子树

} } // 结束InOrder

26.[题目分析] 借助队列和栈，最后弹出栈中元素实现对二叉树按自下至上，自右至左的层次遍历

**void** InvertLevel(biTree bt) // 对二叉树按自下至上，自右至左的进行层次遍历

{**if**(bt!=null)

{StackInit(s); //栈初始化，栈中存放二叉树结点的指针

QueueInit(Q); //队列初始化。队列中存放二叉树结点的指针

QueueIn(Q,bt);

**while**(!QueueEmpty(Q)) //从上而下层次遍历

{p=QueueOut(Q); push(s,p); //出队, 入栈

**if**(p->lchild) QueueIn(Q,p->lchild); //若左子女不空，则入队列

**if**(p->rchild) QueueIn(Q,p->rchild);} //若右子女不空,则入队列

**while**(!StackEmpty(s)) {p=pop(s); printf(p->data);} //自下而上,从右到左的层次遍历

}//**if**(bt!=null)

} //结束InvertLevel

27. **int** Level(BiTree bt) //层次遍历二叉树，并统计度为1的结点的个数

{**int** num=0; //num统计度为1的结点的个数

**if**(bt){QueueInit(Q); QueueIn(Q,bt)；//Q是以二叉树结点指针为元素的队列

**while**(!QueueEmpty(Q))

{p=QueueOut(Q); printf(p->data); //出队,访问结点

**if**(p->lchild && !p->rchild ||!p->lchild && p->rchild)num++;//度为1的结点

**if**(p->lchild) QueueIn(Q,p->lchild); //非空左子女入队

**if**(p->rchild) QueueIn(Q,p->rchild); //非空右子女入队

} }//**if**(bt)

**return**(num); }//返回度为1的结点的个数

28. **void** exchange(BiTree bt)//将二叉树bt所有结点的左右子树交换

{**if**(bt){BiTree s；

s=bt->lchild; bt->lchild=bt->rchild; bt->rchild=s; //左右子女交换

exchange(bt->lchild); //交换左子树上所有结点的左右子树

exchange(bt->rchild); //交换右子树上所有结点的左右子树

} }

[算法讨论]将上述算法中两个递归调用语句放在前面，将交换语句放在最后，则是以后序遍历方式交换所有结点的左右子树。中序遍历不适合本题。下面是本题（1）要求的非递归算法

（1）**void** exchange(BiTree t) //交换二叉树中各结点的左右孩子的非递归算法

{**int** top=0; BiTree s[],p; //s是二叉树的结点指针的栈，容量足够大

**if**(bt)

{s[++top]=t;

**while**(top>0)

{t=s[top--];

**if**(t->lchild||t->rchild){p=t->lchild;t->lchild=t->rchild;t->rchild=p;}//交换左右

**if**(t->lchild) s[++top]=t->lchild; //左子女入栈

**if**(t->rchild) s[++top]=t->rchild; //右子女入栈

}//**while**(top>0)

}//**if**(bt) }// 结束exchange

29.[题目分析]对一般二叉树，仅根据一个先序、中序、后序遍历，不能确定另一个遍历序列。但对于满二叉树，任一结点的左右子树均含有数量相等的结点，根据此性质，可将任一遍历序列转为另一遍历序列（即任一遍历序列均可确定一棵二叉树）。

**void** PreToPost(ElemType pre[] ,post[],**int** l1,h1,l2,h2)

//将满二叉树的先序序列转为后序序列，l1,h1,l2,h2是序列初始和最后结点的下标。

{**if**(h1>=l1)

{post[h2]=pre[l1]; //根结点

half=(h1-l1)/2; //左或右子树的结点数

PreToPost(pre,post,l1+1,l1+half,l2,l2+half-1) //将左子树先序序列转为后序序列

PreToPost(pre,post,l1+half+1,h1,l2+half,h2-1) //将右子树先序序列转为后序序列

} }//PreToPost

30.BiTree IntoPost(ElemType in[],post[],**int** l1,h1,l2,h2)

//in和post是二叉树的中序序列和后序序列,l1,h1,l2,h2分别是两序列第一和最后结点的下标

{BiTree bt=(BiTree)malloc(**sizeof**(BiNode));//申请结点

bt->data=post[h2];//后序遍历序列最后一个元素是根结点数据

**for**(i=l1;i<=h1;i++) **if**(in[i]==post[h2])**break**;//在中序序列中查找根结点

**if**(i==l1) bt->lchild=null; //处理左子树

**else** bt->lchild=IntoPost(in,post,l1,i-1,l2,l2+i-l1-1)；

**if**(i==h1) bt->rchild=null; //处理右子树

**else** bt->rchild=IntoPost(in,post,i+1,h1,l2+i-l1,h2-1);

**return**(bt); }

31.**TYPE** bitreptr=^binode;

binode=RECORD data:ElemType; lchild,rchlid:bitreptr END;

**PROC** PreOrder(bt:bitreptr); //非递归前序遍列二叉树

**VAR** S:ARRAY[1..max] OF bitreptr; //max是栈容量，足够大

inits(S);//栈初始化

**WHILE** (bt<>NIL) OR (**NOT** empty(S)) **DO**

[**WHILE** (bt<>NIL )**DO**

[write(bt↑data); push(S,bt->rchild); bt:=bt↑.lchild;]//访问结点,右子女进栈

**WHILE** (**NOT** empty(S) **AND** top(S)=NIL) bt:=pop(S);// 退栈

**IF** **NOT** empty(S) **THEN** bt:=pop(S);

] **ENDP**;

[算法讨论]若不要求使用top(S)，以上算法还可简化。

32.[题目分析]二叉树采取顺序结构存储，是按完全二叉树格式存储的。对非完全二叉树要补上“虚结点”。由于不是完全二叉树，在顺序结构存储中对叶子结点的判定是根据其左右子女为0。叶子和双亲结点下标间的关系满足完全二叉树的性质。

**int** Leaves(**int** h) //求深度为h以顺序结构存储的二叉树的叶子结点数

{**int** BT[]; **int** len=2h-1, count=0; //BT是二叉树结点值一维数组，容量为2h

**for** (i=1;i<=len;i++) //数组元素从下标1开始存放

**if** (BT[i]!=0) //假定二叉树结点值是整数，“虚结点”用0填充

**if**(i\*2)>len) count++; //第i个结点没子女，肯定是叶子

**else** **if**(BT[2\*i]==0 && 2\*i+1<=len && BT[2\*i+1]==0) count++; //无左右子女的结点是叶子

**return** (count)

} //结束Leaves

33.[题目分析] 计算每层中结点值大于50的结点个数，应按层次遍历。设一队列Q，用front和rear分别指向队头和队尾元素，last指向各层最右结点的位置。存放值大于50的结点结构为

**typedef** **struct** {**int** level,value,idx; }node;//元素所在层号、值和本层中的序号

node a[],s;

**void** ValueGT50(BiTree bt)//查各层中结点值大于50的结点个数，输出其值及序号

{**if**(bt!=null)

{**int** front=0,last=1,rear=1,level=1,i=0,num=0;//num记>50的结点个数

BiTree Q[];Q[1]=bt;//根结点入队

**while**(front<=last)

{bt=Q[++front];

**if**(bt->data>50){s.level=level; s.idx=++i; s.value=bt->data; a[++num]=s;}

**if**(bt->lchild!=null) Q[++rear]=bt->lchild;//左子女入队列

**if**(bt->rchild!=null) Q[++rear]=bt->rchild;//右子女入队列

**if**(front==last) {last=rear; level++; i=0;} //本层最后一个结点已处理完

} //初始化下层：last指向下层最右结点，层号加1，下层>50的序号初始为0

}//**while**

**for**(i=1;i<=num;i++) //输出data域数值大于50的结点的层号、data域的数值和序号

printf(“层号=%3d,本层序号=%3d，值=%3d”，a[i].level,a[i].idx,a[i].value);

}//算法ValueGT50结束

34. **PROC** print(bt:BiTree；xy:integer)

//将二叉树逆时针旋转90度打印，xy是根结点基准坐标，调用时xy=40

**IF** bt<>NIL **THEN** [ print(bt↑.rchild,xy+5); //中序遍历右子树

writeln(bt->data:xy); //访问根结点

print(bt↑.lchild,xy+5);] //中序遍历左子树

**ENDP**;

35.BiTree creat()//生成并中序输出二叉排序树

{scanf(“%c”,&ch) //ch是二叉排序树结点值的类型变量，假定是字符变量

BiTree bst=null,f=null;

**while**(ch!=‘#’) //‘#’是输入结束标记

{s=(BiTree)malloc(**sizeof**(BiNode)); //申请结点

s->data=ch; s->lchild=s->rchild=null;

**if** (bst==null) bst=s; //根结点

**else** //查找插入结点

{p=bst;

**while**(p)

**if** (ch>p->data) {f=p; p=p->rchild;} //沿右分枝查,f是双亲

**else** {f=p; p=p->lchild;} //沿左分枝查

**if**(f->data<ch) f->rchild=s; **else** f->lchild=s;}//将s结点插入树中

scanf(“%c”,&ch); //读入下一数据

} //**while** (ch!=‘#’)

**return**(bst); } //结束 creat

**void** InOrder(BiTree bst) //bst是二叉排序树，中序遍历输出二叉排序树

{**if**(bst)

{InOrder (bst->lchild); printf(bst->data); InOrder(bst->rchild); }

}//结束InOrder

36.[题目分析]二叉树结点p所对应子树的第一个后序遍历结点q的求法如下：若p有左子树，则q是p的左子树上最左下的叶子结点；若p无左子树，仅有右子树，则q是p的右子树上最左下的叶子结点。

BiTree PostFirst(p)

{BiTree q=p;

**if** (!q) **return**(null); **else**

**while**(q->lchild || q->rchild); //找最左下的叶子结点

**if**(q->lchild) q=q->lchild; //优先沿左分枝向下去查“最左下”的叶子结点

**else** q=q->rchild; //沿右分枝去查“最左下”的叶子结点

**return**(q);

}

[算法讨论]题目“求p所对应子树的第一个后序遍历结点”，蕴涵p是子树的根。若p是叶子结点，求其后继要通过双亲。

37.(1)由先序序列A[1..n]和中序序列B[1..n],可以确定一棵二叉树,详见本章四第38题.

（2）**void** PreInCreat( ElemTypeA[],B[]，**int** l1,h1,l2,h2)

//由二叉树前序序列A[1..n]和中序序列B[1..n]建立二叉树,l1,h1,和l2,h2分别为先序序列和

//中序序列第一和最后结点的下标.初始调用时l1=l2=1,h1=h2=n 。

{**typedef** **struct** {**int** l1,h1,l2,h2; BiTree t; }node;

BiTree bt;

**int** top=0,i; node s[],p; //s为栈,容量足够大

bt=(BiTree)malloc(**sizeof**(BiNode)); //申请结点空间

p.l1=l1; p.h1=h1; p.l2=l2; p.h2=h2; p.t=bt; s[++top]=p; //初始化

**while**(top>0)

{p=s[top--]; bt=p.t; l1=p.l1; h1=p.h1; l2=p.l2 ;h2=p.h2;//取出栈顶数据

**for**(i=l2;i<=h2;i++) **if**(B[i]==A[l1]) **break**; //到中序序列中查根结点的值

bt->data=A[l1]; //A[l1]为根结点的值

**if**(i==l2) bt->lchild=null; //bt无左子树

**else** //将建立左子树的数据入栈

{bt->lchild=(BiTree)malloc(**sizeof**(BiNode)); p.t=bt->lchild;

p.l1=l1+1; p.h1=l1+i-l2; p.l2=l2; p.h2=i-1; s[++top]=p; }

**if**(i==h2) bt->rchild=null; //bt无右子树

**else** {bt->rchild=(BiTree)malloc(**sizeof**(BiNode)); p.t=bt->rchild;

p.l1=l1+i-l2+1; p.h1=h1; p.l2=i+1; p.h2=h2; s[++top]=p; }//右子树数据入栈

}//**while**

}结束PreInCreat

(3)当二叉树为单支树时，栈深n；，当二叉树左右子树高相等时，栈深logn。时间复杂度O(n)。

38.[题目分析]知二叉树中序序列与后序序列，第30题以递归算法建立了二叉树，本题是非递归算法。

**void** InPostCreat(ElemType IN[],POST[],**int** l1,h1,l2,h2)

//由二叉树的中序序列IN[]和后序序列POST[]建立二叉树，l1,h1和l2,h2分别是中序序列和

//后序序列第一和最后元素的下标，初始调用时，l1=l2=1,h1=h2=n。

{**typedef** **struct** {**int** l1,h1,l2,h2; BiTree t; }node;

node s[],p;//s为栈，容量足够大

BiTree bt=(BiTree)malloc(**sizeof**(BiNode)); **int** top=0,i;

p.l1=l1; p.h1=h1; p.l2=l2; p.h2=h2; p.t=bt; s[++top]=p;//初始化

**while**(top>0)

{p=s[top--]; bt=p.t; l1=p.l1; h1=p.h1; l2=p.l2; h2=p.h2;//取出栈顶数据

**for**(i=l1;i<=h1;i++) **if**(IN[i]==POST[h2]) **break**;//在中序序列中查等于POST[h2]的结点

bt->data=POST[h2]; //根结点的值

**if**(i==l1) bt->lchild=null; //bt无左子树

**else** //将建立左子树的数据入栈

{bt->lchild=(BiTree)malloc(**sizeof**(BiNode)); p.t=bt->lchild;

p.l1=l1; p.h1=i-1; p.l2=l2; p.h2=l2+i-l1-1; s[++top]=p; }

**if**(i==h1) bt->rchild=null; //bt无右子树

**else** {bt->rchild=(BiTree)malloc(**sizeof**(BiNode)); p.t=bt->rchild;

p.l1=i+1; p.h1=h1; p.l2=l2+i-l1; p.h2=h2-1; s[++top]=p; }//右子树数据入栈

}// **while**(top>0)

}结束InPostCreat

39．BiTree Copy(BiTree t)//复制二叉树t

{BiTree bt;

**if** (t==null) bt=null;

**else**{bt=(BiTree)malloc(**sizeof**(BiNode)); bt->data=t->data;

bt->lchild=Copy(t->lchild);

bt->rchild=Copy(t->rchild);

}

**return**(bt); }//结束Copy

40.[题目分析]森林在先根次序遍历时，首先遍历第一棵子树的根，接着是第一棵子树的结点；之后是第二棵树，……，最后一棵树。本题中E[i]是H[i]所指结点的次数，次数就是结点的分支个数B，而分支数B与树的结点数N的关系是N=B+1（除根结点外，任何一个结点都有一个分支所指）。所以，从E[i]的第一个单元开始，将值累加，当累加到第i个单元，其值正好等于i-1时，就是第一棵树。接着，用相同方法，将其它树分开，进行到第n个单元，将所有树分开。例如，上面应用题第47题(2)的森林可按本题图示如下。从左往右将次数加到下标8（=B+1）时，正好结束第一棵树。

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| H[i] | A | B | D | G | L | H | E | I | C | F | J | K |
| E[i] | 3 | 2 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

**void** Forest(ElemType H[],**int** E[],int,n)

// H[i]是森林F在先根次序下结点的地址排列，E[i]是H[i]所指结点的次数，本算法计算森林

//F的树形个数，并计算森林F的最后一个树形的根结点地址

{**int** i=1,sum=0,j=0,m=0; //sum记一棵树的分支数，j记树的棵数，m记一棵树的结点数

**int** tree[]; //tree记每棵树先序遍历最后一个结点在H[i]中的地址

**while** (i<=n) //n是森林中结点个数，题目已给出

{sum+=E[i]; m++;

**if** (sum+1==m && i<=n) //记树先序最后结点的地址，为下步初始化

{sum=0; m=0; tree[++j]=i;}

i++;

}//**while**

**if** (j==1)**return** (1); //只有一棵树时，第一个结点是根

**else** **return**(tree[j-1]+1)

}//**for**est

41.[题目分析]叶子结点只有在遍历中才能知道，这里使用中序递归遍历。设置前驱结点指针pre，初始为空。第一个叶子结点由指针head指向，遍历到叶子结点时，就将它前驱的rchild指针指向它，最后叶子结点的rchild为空。

LinkedList head,pre=null; //全局变量

LinkedList InOrder(BiTree bt)

//中序遍历二叉树bt，将叶子结点从左到右链成一个单链表，表头指针为head

{**if**(bt){InOrder(bt->lchild); //中序遍历左子树

**if**(bt->lchild==null && bt->rchild==null) //叶子结点

**if**(pre==null) {head=bt; pre=bt;} //处理第一个叶子结点

**else**{pre->rchild=bt; pre=bt; } //将叶子结点链入链表

InOrder(bt->rchild); //中序遍历左子树

pre->rchild=null; //设置链表尾

}

**return**(head); } //InOrder

时间复杂度为O(n),辅助变量使用head和pre,栈空间复杂度O(n)

42. 层次遍历，参见上面算法第33题。中序遍历序列和后序序列可确定一棵二叉树，详见上面应用题第40题。先序序列和后序序列无法确定一棵二叉树，因为无法将二叉树分成左右子树。如先序序列AB和后序序列BA，A是根，但B可以是左子女，也可以是右子女。

43.[题目分析]此树看作度为2的有序树，先将根结点入队列。当队列不空，重复以下动作：结点出队；若结点有两个子女，先将第二（右）子女入队列，并转向第一子女；若结点有一个子女，则入队列；如此下去，直到碰到叶子结点或只有一个子女的结点，再重复上述动作，直至队列为空。定义树结构如下：

**typedef** **struct** node

{ElemType data; **struct** node \*firstchild, \*secondchild; }\*Tree;

**void** TreeTravers(Tree t) //按字母顺序输出树中各结点的值

{Tree p,Q[]; //Q为队列，元素是树结点指针，容量是够大

**if** (t)

{QueueInit(Q); QueueIn(Q,t); //根结点入队

**while**(!QueueEmpty(Q))

{p=QueueOut(Q); //出队

**while** (p->firstchild && p->secondchild) //当有双子女时

{QueueIn(Q,p->secondchild);printf(p->data);//访问结点

p=p->firstchild;}// 沿第一子女向下

**if** (p->firstchild){printf(p->data); QueueIn(Q,p->firstchild)} //一个子女树入队

**if** (!p->firstchild && !p->secondchild) printf(p->data);//访问叶子结点

}//**while**(!QueueEmpty(Q))

}//**if**

}//算法结束

44.[题目分析] 由定义，结点的平衡因子bf等于结点的左子树高度与右子树高度之差，设计一遍历算法，在遍历结点时，求结点的左子树和右子树的高度，然后得到结点的平衡因子。

**int** Height(BiTree bt)//求二叉树bt的深度

{**int** hl,hr;

**if** (bt==null) **return**(0);

**else** {hl=Height(bt->lchild); hr=Height(bt->rchild);

**if**(hl>hr) **return** (hl+1); **else** **return**(hr+1);

} }// Height

**void** Balance(BiTree bt)

//计算二叉树bt各结点的平衡因子

{**if** (bt)

{Balance(bt->lchild); //后序遍历左子树

Balance(bt->rchild); //后序遍历右子树

hl=Height(bt->lchild); hr=Height(bt->rchild);//求左右子树的高度

bt->bf=hl-hr; //结点的平衡因子bf

} }//算法结束

45.[题目分析]本题应采用层次遍历方式。若树不空，则二叉树根结点入队，然后当队列不空且队列长不超过n，重复如下操作：出队，若出队元素不为空，则记住其下标，且将其左右子女入队列；若出队元素为空，当作虚结点，也将其“虚子女”入队列。为节省空间，就用树T的顺序存储结构A[1..n]作队列，队头指针front，队尾指针rear，元素最大下标last.

**void** Traverse(BiTree bt ,**int** n)

// 求二叉树bt的顺序存储结构A[1..n]，下标超过n报错，给出实际的最大下标

{BiTree A[], p;

**if**(bt!=null)

{**int** front=0,rear=1,last=1; A[1]=bt;

**while**(front<=rear)

{p=A[++front]; **if**(p) last=front; // 出队;用last记住最后一个结点的下标

rear=2\*front;//计算结点（包括虚结点）“左子女”下标

**if** (p) //二叉树的实际结点

{**if**(rear>n) printf(“%c结点无左子女”); **else** A[rear]=p->lchild;

**if**(rear+1>n) printf(“%c结点无右子女”); **else** A[rear+1]=p->rchild;

}

**else** //p是虚结点

{ **if**(rear<=n) A[rear]=null; **if**(rear+1<=n) A[rear+1]=null; }

}// **while**(front<=rear)

printf(“实际的最大下标是%d”,last);

}//**if**(bt!=null) }//Traverse

46. [题目分析]两棵空二叉树或仅有根结点的二叉树相似；对非空二叉树，可判左右子树是否相似，采用递归算法。

**int** Similar(BiTree p,q) //判断二叉树p和q是否相似

{**if**(p==null && q==null) **return** (1);

**else** **if**(!p && q || p && !q) **return** (0);

**else** **return**(Similar(p->lchild,q->lchild) && Similar(p->rchild,q->rchild))

}//结束Similar

47.[题目分析] 根据树的双亲表示法创建树的孩子兄弟链表表示法,首先给出根结点在双亲表示法中的下标，但找根结点的子女要遍历双亲表示法的整个静态链表，根结点的第一个子女是孩子兄弟表示法中的孩子，其它子女结点作兄弟。对双亲表示法中的任一结点，均递归建立其孩子兄弟链表子树。

CSTree PtreeToCstree (PTree pt,**int** root)

//本算法将双亲表示法的树pt转为孩子兄弟链表表示的树，root是根结点在双亲表示法中的下标。

{CSTree child , sibling; **int** firstchild;

CSTree cst=(CSTree)malloc(**sizeof**(CSNode)); //申请结点空间

cst->data=pt.nodes[root].data; cst->firstchild=null; cst->nextsibling=null;//根结点

firstchild=1;

**for**(i=1;i<=pt.n;i++) //查找root的孩子

**if**(pt.nodes[i].parent==root)

{child=PtreetoCstree(pt,i);

**if**(firstchild) {cst->firstchild=child; firstchild=0;sibling=cst->firstchild;}

**else** //child不是root的第一个孩子，作兄弟处理

{sibling->nextsibling=child; sibling=sibling->nextsibling;}

}//**if**

}//end **for**

**return** cst; }//结束PtreetoCstree

48.[题目分析] 删除以元素值x为根的子树，只要能删除其左右子树，就可以释放值为x的根结点，因此宜采用后序遍历。删除值为x结点，意味着应将其父结点的左(右)子女指针置空，用层次遍历易于找到某结点的父结点。本题要求删除树中每一个元素值为 x的结点的子树，因此要遍历完整棵二叉树。

**void** DeleteXTree(BiTree bt) //删除以bt为根的子树

{DeleteXTree(bt->lchild); DeleteXTree(bt->rchild);//删除bt的左子树、右子树

free(bt); }// DeleteXTree //释放被删结点所占的存储空间

**void** Search(B:Tree bt,ElemType x)

//在二叉树上查找所有以x为元素值的结点，并删除以其为根的子树

{BiTree Q[];//Q是存放二叉树结点指针的队列，容量足够大

**if**(bt)

{**if**(bt->data==x) {DeleteXTree(bt); exit(0);}//若根结点的值为x，则删除整棵树

{QueueInit(Q); QueueIn(Q,bt);

**while**(!QueueEmpty(Q))

{p=QueueOut(Q);

**if**(p->lchild) // 若左子女非空

**if**(p->lchild->data==x) //左子女结点值为 x,应删除当前结点的左子树

{DeleteXTree(p->lchild); p->lchild=null;} //父结点的左子女置空

**else** Enqueue (Q,p->lchild);// 左子女入队列

**if**(p->rchild) // 若右子女非空

**if**(p->rchild->data==x) //右子女结点值为 x,应删除当前结点的右子树

{DeleteXTree(p->rchild); p->rchild=null;} //父结点的右子女置空

**else** Enqueue (Q,p->rchild);// 右子女入队列

}//**while**

}//**if**(bt) }//search

49.[题目分析]在二叉树上建立三叉链表，若二叉树已用二叉链表表示，则可象48题那样，给每个结点加上指向双亲的指针（根结点的双亲指针为空）。至于删除元素值为x的结点以及以x为根的子树，与48题完全一样，请参照48题。下面给出建立用三叉链表表示的二叉树的算法。二叉树按完全二叉树格式输入，对非完全二叉树，要补上“虚结点”。按完全二叉树双亲和子女存储下标间的关系，完成双亲和子女间指针的链接。‘#’是输入结束标志，‘$’是虚结点标志。

BiTree creat()/ /生成三叉链表的二叉树（题目给出PASCAL定义，下面的用类C书写）

{BiTree p,Q[]，root; //Q是二叉树结点指针的一维数组，容量足够大

**char** ch; **int** rear=0; //一维数组最后元素的下标

scanf(&ch);

**while**(ch!=‘#’)

{p=null;

**if**(ch!=‘$’){p=(BiTree)malloc(**sizeof**(nodetp));

p->data=ch; p->lchild=p->rchild=null; }

Q[++rear]=p; //元素或虚结点

**if**(p){**if**(rear==1) {root=p;root->parent=null; } //根结点

**else**{Q[rear]->parent=Q[rear/2]; /双亲结点和子女结点用指针链上

**if** (rear%2==0) Q[rear/2]->lchild=Q[rear]; **else** Q[rear/2]->rchild=Q[rear];

}

scanf(“%c”,&ch);

}//**while**

**return**(root); }//结束creat

50.**int** BTLC(BiTree T,**int** \*c)//对二叉树T的结点计数

{**if**(T)

{\*c++;

BTLC(T->lchild,&c); //统计左子树结点

BTLC(T->rchild,&c); //统计右子树结点

} }//结束Count,调用时\*c=0

51. **int** Count(CSTree t)//统计以孩子兄弟链表表示的树的叶子结点个数

{**if**(t==null) **return**(0);

**else** **if**(t->firstlchild==null) //左子女为空，结点必为叶子

**return**(1+Count(t->nextsibling))； //（叶子）+兄弟子树上的叶子结点

**else** **return**(Count(t->firstchild)+Count(t->nextsibling));//子女子树+兄弟子树

}//Count

52.**void** Count(BiTree bt,**int** \*n0,\*n) //统计二叉树bt上叶子结点数n0和非叶子结点数n

{**if**(bt)

{**if** (bt->lchild==null && bt->rchild==null) \*n0++；//叶子结点

**else** \*n++; //非叶结点

Count(bt->lchild,&n0,&n);

Count(bt->rchild,&n0,&n);

} }//结束 Count

53．**int** Count (BiTree bt) // 非递归遍历求二叉树上的叶子结点个数

{**int** num=0;

BiTree s[]; //s是栈，栈中元素是二叉树结点指针，栈容量足够大

whlie(bt!=null || top>0)

{**while**(bt!=null) {push(s,bt)；bt=bt->lchild；} //沿左分支向下

**if**(!StackEmpty(s))

{bt=pop(s)；**if**(bt->lchild==null && bt->rchild==null) num++；//叶子结点

bt=bt->rchild；

}

} **return**(num);

}//结束Count

54.[题目分析]对二叉树的某层上的结点进行运算，采用队列结构按层次遍历最适宜。

**int** LeafKlevel(BiTree bt, **int** k) //求二叉树bt 的第k(k>1) 层上叶子结点个数

{**if**(bt==null || k<1) **return**(0);

BiTree p=bt,Q[]; //Q是队列，元素是二叉树结点指针,容量足够大

**int** front=0,rear=1,leaf=0; //front 和rear是队头和队尾指针, leaf是叶子结点数

**int** last=1,level=1; Q[1]=p; //last是二叉树同层最右结点的指针，level 是二叉树的层数

**while**(front<=rear)

{p=Q[++front];

**if**(level==k && !p->lchild && !p->rchild) leaf++; //叶子结点

**if**(p->lchild) Q[++rear]=p->lchild; //左子女入队

**if**(p->rchild) Q[++rear]=p->rchild; //右子女入队

**if**(front==last) {level++; //二叉树同层最右结点已处理,层数增1

last=rear; } //last移到指向下层最右一元素

**if**(level>k) **return** (leaf); //层数大于k 后退出运行

}//**while** }//结束LeafKLevel

55.[题目分析]按题目要求，每个结点的编号大于其左右孩子的编号，结点左孩子的编号小于右孩子的编号。由此看出，从小到大按“左右根”顺序，这正是后序遍序的顺序，故对二叉树进行后序遍历，在遍历中对结点进行编号，现将二叉树结点结构定义如下：

**typedef** **struct** node

{ElemType data; **int** num; **struct** node \*lchild, \*rchild; }Bnode,\*Btree；

**void** PostOrder(Btree t)

//对二叉树从1开始编号，结点编号大于其左右子女结点编号，结点的左子女编号小于其右子女编号

{**typedef** **struct** {Btree t; **int** tag; }node;

Btree p=t; node sn,s[]; //s为栈，容量足够大

**int**  k=0,top=0; //k为结点编号，top为栈顶指针

**while**(p!=null || top>0)

{**while**(p) {sn.t=p; sn.tag=0; s[++top]=sn; p=p->lchild;} //沿左分枝向下

**while**(top>0 && s[top].tag==1){sn=s[top--];sn.t->num=++k;}//左右孩子已遍历,结点赋编号

**if** (top>0) {s[top].tag=1; p=s[top].t->rchild;}

}//**while**(p!=null || top>0)

}结束PostOrder

56. [题目分析]非递归算法参考上面第37题。下面给出递归算法。

**void** PreInCreat(BiTree root,ElemType pre[],in[],**int** l1,h1,l2,h2)

//根据二叉树前序序列pre和中序序列in建立二叉树。l1,h1,l2,h2是序列第一和最后元素下标。

{root=(BiTree)malloc(**sizeof**(BiNode)); //申请结点

root->data=pre[l1]; //pre[l1]是根

**for**(i=l2;i<=h2;i++) **if**(in[i]==pre[l1]) **break**; //在中序序列中，根结点将树分成左右子树

**if**(i==l2) root->lchild=null; //无左子树

**else** PreInCreat(root->lchild,pre,in,l1+1,l1+(i-l2),l2,i-1)； //递归建立左子树

**if**(i==h2) root->rchild=null; //无右子树

**else** PreInCreat((root)->rchild,pre,in,l1+(i-l2)+1,h1,i+1,h2) //递归建立右子树

}//结束PreInCreat

57（1）略 (2) 根据中序和后序序列，建立二叉树的递归算法见上面第30题，非递归算法见第38题。

58.[题目分析]采用后序非递归遍历二叉树，栈中保留从根结点到当前结点的路径上的所有结点。

**void** PrintPath(BiTree bt,p) //打印从根结点bt到结点p之间路径上的所有结点

{BiTree q=bt,s[]; //s是元素为二叉树结点指针的栈，容量足够大

**int** top=0; tag[];//tag是数组，元素值为0或1，访问左、右子树的标志，tag和s同步

**if** (q==p) {printf(q->data); **return**;} //根结点就是所找结点

**while**(q!=null || top>0)

{**while**(q!=null) //左子女入栈，并置标记

**if**（q==p） //找到结点p，栈中元素均为结点p 的祖先

{printf(“从根结点到p结点的路径为\n”);

**for**(i=1;i<=top;i++) printf(s[i]->data); printf(q->data); **return**;

}

**else** {s[++top]=q; tag[top]=0; q=q—>lchild;} //沿左分支向下

**while**(top>0 && tag[top]==1)) top--；//本题不要求输出遍历序列，这里只退栈

**if** (top>0) {q=s[top]; q=q->rchild; tag[top]=1; } //沿右分支向下

}//**while**(q!=null || top>0)

}//结束算法PrintPath

59.[题目分析]上题是打印从根结点到某结点p的路径上所有祖先结点，本题是打印由根结点到叶子结点的所有路径。只要在上题基础上把q是否等于p的判断改为q是否是叶子结点即可。其语句段如下：

**if**(q->lchild==null&&q->rchild==null) //q为叶子结点

{printf(“从根结点到p结点的路径为\n”);

**for**(i=1;i<=top;i++) printf(s[i]->data);//输出从根到叶子路径上，叶子q的所有祖先

printf(q->data); }

60.[题目分析]因为后序遍历栈中保留当前结点的祖先的信息，用一变量保存栈的最高栈顶指针，每当退栈时，栈顶指针高于保存最高栈顶指针的值时，则将该栈倒入辅助栈中，辅助栈始终保存最长路径长度上的结点，直至后序遍历完毕，则辅助栈中内容即为所求。

**void** LongestPath(BiTree bt)//求二叉树中的第一条最长路径长度

{BiTree p=bt,l[],s[]; //l, s是栈，元素是二叉树结点指针，l中保留当前最长路径中的结点

**int** i，top=0,tag[],longest=0;

**while**(p || top>0)

{ **while**(p) {s[++top]=p；tag[top]=0; p=p->Lc;} //沿左分枝向下

**if**(tag[top]==1) //当前结点的右分枝已遍历

{**if**(!s[top]->Lc && !s[top]->Rc) //只有到叶子结点时，才查看路径长度

**if**(top>longest) {**for**(i=1;i<=top;i++) l[i]=s[i]; longest=top; top--;}

//保留当前最长路径到l栈，记住最高栈顶指针，退栈

}

**else** **if**(top>0) {tag[top]=1; p=s[top].Rc;} //沿右子分枝向下

}//**while**(p!=null||top>0)

}//结束LongestPath

61.[题目分析]在线索二叉树上插入结点,破坏了与被插入结点的线索。因此,插入结点时,必须修复线索。在结点y的右侧插入结点x，因为是后序线索树，要区分结点y有无左子树的情况。

**void** TreeInsert(BiTree t,y,x)//在二叉树t的结点y的右侧，插入结点x

{**if**(y->ltag==0) //y有左子女

{p=y->lchild; **if** (p->rtag==1) p->rchild=x; //x是y的左子女的后序后继

x->ltag=1; x->lchild=p; //x的左线索是y的左子女

}

**else** //y无左子女

{x->ltag=1; x->lchild=y->lchild;//y的左线索成为x的左线索

**if**(y->lchild->rtag==1) //若y的后序前驱的右标记为1

y->lchild->rchild=x; //则将y的后序前驱的后继改为x

}

x->rtag=1; x->rchild=y; y->rtag=0; y->rchild=x; //x作y的右子树

}//结束 TreeInsert

62.[题目分析]在中序全线索化T树的P结点上，插入以X为根的中序全线索化二叉树，要对X有无左右子树进行讨论，并修改X左(右)子树上最左结点和最右结点的线索。在中序线索树上查找某结点p的前驱的规律是：若p->ltag=1，则p->lchild就指向前驱，否则，其前驱是p的左子树上按中序遍历的最后一个结点；查找某结点p的后继的规律是：若p->rtag=1，则p->rchild就指向后继，否则，其后继是p的右子树上按中序遍历的第一个结点。

**int** TreeThrInsert(BiThrTree T,P,X)

//在中序全线索二叉树T的结点P上，插入以X为根的中序全线索二叉树，返回插入结果信息。

{**if**(P->ltag==0 && P->rtag==0) {printf(“P有左右子女，插入失败\n”)； **return**(0); }

**if**(P->ltag==0) //P有左子女，将X插为P的右子女

{**if**(X->ltag==1) X->lchild=P; //若X无左子树， X的左线索(前驱)是P

**else** //寻找X的左子树上最左（下）边的结点

{q=X->lchild;

**while**(q->ltag==0) q=q->lchild;

q->lchild=P;

}

**if**(X->rtag==1) //修改X的右线索

X->rchild=P->rchild; //将P的右线索改为X的右线索

**else** //找X右子树最右面的结点

{q=X->rchild; **while**(q->rtag==0) q=q->rchild;

q->rchild=P->rchild;

}

P->rtag=0;P->rchild=X; //将X作为P的右子树

} //结束将X插入为P的右子树

**else** //P有右子女，将X插入为P的左子女

{**if**(X->ltag==1) //X无左子女

X->lchild=P->lchild; //将P的左线索改为X的左线索

**else** //X有左子女，找X左子树上最左边的结点

{q=X->lchild;

**while**(q->ltag==0) q=q->lchild;

q->lchild=P->lchild;

}

**if**(X->rtag==1) X->rchild=P; //若X无右子树，则X的右线索是P

**else** //X有右子树，查其右子树中最右边的结点，将该结点的后继修改为P

{q=X->rchild;

**while**(q->rtag==0) q=q->rchild;

q->rchild=P;

}

P->ltag=0; //最后将P的左标记置0

P->lchild=X; //P的左子女链接到X

} //结束将X插入为P的左子女

} //结束Tree Thrfnsert

63.[题目分析]在中序线索树中，非递归查找数据域为A的结点（设该结点存在，其指针为P）并将数据域为x的Q结点插入到左子树中。若P无左子女，则Q成为P的左子女,原P的左线索成为Q的左线索，Q的右线索为P；若P有左子树，设P左子树中最右结点的右线索是结点Q，结点Q的右线索是P。

**void** InThrInsert(BiThrTree T,Q; ElemType A)

//在中序线索二叉树T中,查找其数据域为A的结点，并在该结点的左子树上插入结点Q

{BiThrTree P=T;

**while**(P)

{**while**(P->LT==0 && P->data!=A) P=P->LL; //沿左子树向下

**if** (P->data==A) **break**; //找到数据域为A的结点,退出循环

**while**(P->RT==1) P=P->RL; //还没找到数据域为A的结点沿右线索找后继

P=P->RL; //沿右子树向下

}

**if**(P->LT==1) //P没有左子树，Q结点插入作P的左子女

{Q->LL=P->LL; Q->LT=1； //将P的左线索作为Q的左线索

}

**else** //P有左子树，应修改P的左子树最右结点的线索

{Q->LL=P->LL;Q->LT=0; //Q成为P的左子女

s=Q->LL; //s指向原P的左子女

**while**(s->RT==0) s=s->RL; //查找P的左子树最右边的结点

s->RL=Q; //原P左子树上最右结点的右线索是新插入结点Q

}

P->LT=0;P->LL=Q; //修改P的标记和指针

Q->RT=1;Q->RL=P; //将Q链为P的左子女,其中序后继是P;

}//结束InThrInsert

64.[题目分析] “双链”就利用二叉树结点的左右指针，重新定义左指针为指向前驱的指针，右指针是指向后继的指针，链表在遍历中建立，下面采用中序遍历二叉树。

BiTree head=null,pre; //全局变量链表头指针head,pre

**void** CreatLeafList(BiTree bt) //将BiTree 树中所有叶子结点链成带头结点的双链表，

{**if**（bt） //若bt不空

{CreatLeafList（bt->lchild）; //中序遍历左子树

**if**(bt->lchild==null && bt->rchild==null) //叶子结点

**if**(head==null)//第一个叶子结点

{head=(BiTree)malloc(**sizeof**(BiNode)); //生成头结点

head->lchild=null; head->rchild=bt; //头结点的左链为空,右链指向第一个叶子结点

bt->lchild=head; pre=bt; //第一个叶子结点左链指向头结点，pre指向当前叶子结点

}

**else** //已不是第一个叶子结点

{pre->rchild=bt; bt->lchild=pre; pre=bt;} //当前叶子结点链入双链表

CreatLeafList(bt->rchild); //中序遍历右子树

pre->rchild=null; //最后一个叶子结点的右链置空（链表结束标记）

}//**if**(bt) }//结束CreatLeafList;

65.[题目分析]求中序全线索树任意结点p的前序后继，其规则如下：若p有左子女，则左子女就是其前序后继；若p无左子女而有右子女，则p的右子女就是p的前序后继；若p无左右子女，这时沿p的右线索往上，直到p的右标志为0（非线索），这时若p的右子女为空，则表示这是中序遍历最后一个结点，故指定结点无前序后继，否则，该结点就是指定结点的前序后继。程序段如下：

**if**(p->ltag==0 && p->lchild!=null) **return**(p->lchild); //p的左子女是p的前序后继

**else** **if**(p->rtag==0) && p->rchild!=null) **return**(p->rchild);//p右子女是其前序后继

**else** //p无左右子女，应沿右线索向上（找其前序后继），直到某结点右标记为0

{**while** (p->rtag==1) p=p->rchild;

**if** (p->rchild) **return**(p->rchild);**else** **return**(null); }//指定结点的前序后继

[算法讨论]请注意题目“中序序列第一结点的左标志和最后结点的右标志皆为０（非线索），对应指针皆为空”的说明。若无这一说明，只要结点的左标记为0，其左子女就是其前序后继。最后，当p无子女，沿右线索向上找其前序后继时，若最后结点的右标志为０，但对应指针为空，p也无前序后继。

66.[题目分析] 不借助辅助堆栈实现中序遍历，必须解决如何查找后继的问题。使用线索树就行。为此，将结点结构修改为(ltag,lchild,data,rchild,rtag)。各字段的含义在上面已多次使用，不再介绍。设二叉树已中序线索化。下面首先编写一查中序后继的函数，接着是中序遍历的非递归算法。

BiTree After(BiThrTree t) //查中序线索二叉树上结点t的后继

{**if** (t->rtag==1) **return**(t->rchild);

p=t->rchild;

**while**(p->ltag==0) p=p->lchild; //p右子树中最左下的结点是p的中序后继

**return**(p); } //**if**

**void** InOrder(BiThrTree bt)

//非递归中序遍历带头结点的中序线索二叉树bt

{p=bt->lchild; //p指向原二叉树的根结点

**if** (p!=bt) //二叉树非空

{**while** (p->ltag==0) p=p->lchild; //找中序遍历的第一个结点

**while** (p!=bt) //没回到头结点，就一直找后继并遍历

{visit(\*p); p=After(p); }

}//**if** }结束算法InOrder

67.[题目分析]在中序穿线树中找结点的双亲，最简单情况是顺线索就可找到。例如，结点的左子女的右线索和右子女的左线索都指向双亲。但对于有左右子女的结点来说，则要利用中序穿线树中线索“向上”指向祖先的特点：若结点p是结点q右子树中中序遍历最左下的结点，p的左线索指向q；若结点p是结点q左子树上中序遍历最右下的结点，p的右线索指向是q。反过来，通过祖先找子女就容易了。另外，若结点q的后继是中序穿线树的头结点，则应特殊考虑。

**void** FFA(BiThrTree t,p,q)//在中序穿线树t上，求结点p的双亲结点q

{q=p; //暂存

**while**(q->RTAG==0) q=q->RLINK; //找p的中序最右下的结点

q=q->RLINK; //顺右线索找到q的后继（p的祖先结点）

**if** (q==t) q=t->LLINK; //若后继是头结点，则转到根结点

**if** (q==p) {printf(“根结点无双亲\n”)；**return**; }

**if** (q->LLINK==p) **return**(q); **else** q=q->LLINK; //准备到左子树中找p

**while** (q->RLINK!=p) q=q->RLINK；**return**(q); } //找最右结点的过程中回找到p

}//结束FFA

[算法讨论]本题也可以先求结点p最左下结点的前驱线索，请读者自己写出算法。

68.[题目分析]带头结点的中序线索树，其头结点的lchild指向二叉树的根结点，头结点的rchild域指向中序遍历的最后一个结点。而二叉树按中序遍历的第一个结点的lchild和最后一个结点的rchild指向头结点。故从头结点找到根结点后，顺“后继”访问二叉树。在中序线索树中，找前序的后继，已在第65题进行了详细的讨论，这里不再赘述。中序线索树在上面的“四、应用题”已画过多个，这里也不重复。

**void** PreorderInThreat(BiTrhTree tbt)

//前序遍历一中序全线索二叉树tbt，tbt是头结点指针

{bt=tbt->lchild;

**while**(bt)

{**while**(bt->ltag==0){printf(bt->data); bt=bt->lchild;}//沿左分枝向下

printf(bt->data); //遍历其左标志为1的结点，准备右转

**while**(bt->rtag==1 && bt->rchild!=tbt) bt=bt->rchild;//沿右链向上

**if** (bt->rchild!=tbt) bt=bt->rchild;//沿右分枝向下

}

}//结束PreorderInThreat

时间复杂度O(n)。

69.[题目分析]线索化是在遍历中完成的，因此，对于二叉树进行前序、中序、后序遍历，在“访问根结点”处进行加线索的改造，就可实现前序，中序和后序的线索化

BiThrTree pre=null;//设置前驱

**void** PreOrderThreat(BiThrTree BT)

//对以线索链表为存储结构的二叉树BT进行前序线索化

{**if** (BT!=null)

{**if** (BT->lchild==null){BT->ltag=1; BT->lchild=pre;}//设置左线索

**if** (pre!=null && pre->rtag==1) pre->rchild=BT; //设置前驱的右线索；

**if** (BT->rchild==null) BT->rtag=1; //为建立右链作准备

pre=BT;//前驱后移

**if** (BT->ltag==0) PreOrderThreat(BT->lchild); //左子树前序线索化

PreOrderThreat(BT->rchild); //右子树前序线索化

}//**if** (BT!=null) }结束PreOrderThreat

70. BiThrTree pre==null;

**void** InOrderThreat(BiThrTree T)//对二叉树进行中序线索化

{**if** (T)

{InOrderThreat(T->lchild); //左子树中序线索化

**if** (T->lchild==null) {T->ltag=1; T->lchild=pre; } //左线索为pre;

**if** (pre!=null && pre->rtag==1) pre->rchild=T;} //给前驱加后继线索

**if** (T->rchild==null) T->rtag=1; //置右标记，为右线索作准备

pre=BT;//前驱指针后移

InOrderThreat(T->rchild); //右子树中序线索化

} }//结束InOrderThreat

71. **void** InOrderThreat(BiThrTree thrt)

//thrt是指向中序全线索化头结点的指针，本算法中序遍历该二叉树

{p=thrt->lchild; //p指向二叉树的根结点，当二叉树为空时，p指向thrt

whild(p!=thrt)

{**while**（p->ltag==0) p=p->lchild;//沿左子女向下

visit(\*p);//访问左子树为空的结点

**while**(p->rtag==1 && p->rchild!=thrt){p=p->rchild;visit(\*p);}//沿右线索访问后继结点

p=p->rchild;//转向右子树

} }//结束InOrderThread

72.[题目分析]若使新插入的叶子结点S成T右子树中序序列的第一个结点，则应在T的右子树中最左面的结点（设为p）处插入，使S成为结点p的左子女。则S的前驱是T，后继是p.

**void** ThrTreeInsert(BiThrTree T,S)

//在中序线索二叉树T的右子树上插入结点S，使S成为T右子树中序遍历第一个结点

{p=T->rchild; //用p去指向T的右子树中最左面的结点

**while**(p->ltag==0) p=p->lchild;

S->ltag=1;S->rtag=1; //S是叶子，其左右标记均为1

S->lchild=T;S->rchild=p;//S的前驱是根结点T，后继是结点p

p->lchild=S;p->ltag=0; //将p的左子女指向S ，并修改左标志为0

}//结束 ThrTreeInsert

73．BiThrTree InOrder(BiThrTree T,ElemType x)

//先在带头结点的中序线索二叉树T中查找给定值为x的结点，假定值为x的结点存在

{p=T->lchild;//设p指向二叉树的根结点

**while**(p!=T)

{**while**(p->ltag==0 && p-data!=x) p=p->lc;

**if**(p->data==x)**return**(p);

**while**(p->rtag==1 && p->rc!=T) {p=p->rc; **if**(p->data== x) **return**(p);}

p=p->rc; }

}//结束InOrder

BiThrTree AfterXNode(BiThrTree T)//在中序线索二叉树T中，求给定值为 x的结点的后继结点

{BiThrTree p=InOrde(T,x); //首先在T 树上查找给定值为x 的结点，由p指向

**if**(p->rtag==1) **return**(p->rc); //若p 的左标志为1，则p的rc指针指向其后继

**else** {q=p->rc; **while**(q->ltag==0)q=q->lc; **return**(q); }

//结点p的右子树中最左面的结点是结点p的中序后继

} }//结束AfterXnode

74．[题目分析]后序遍历是“左-右-根”，因此，若结点有右子女，则右子女是其后序前驱，否则，左子女（或左线索）指向其后序前驱。

BiThrTree PostSucc (BiThrTree T,p)//在后序线索二叉树T中，查找指定结点p的直接前驱q

{**if**(p->Rtag==0) q=p->Rchild;//若p有右子女，则右子女为其前驱

**else** q=p->Lchild; //若p无右子女，左子女或左线索就是p的后序前驱

**return** (q);

}//结束PostSucc

75．BiThrTree InSucc(BiThrTree T,p) //在对称序穿线树T中，查找给定结点p的中序后继

{**if**（p->rtag==1）q=p->rchild; //若p的右标志为1，用其右指针指向后继

**else** {q=p->rchild; **while**(q->ltag==0) q=q->lchild; }//p的后继为其右子树中最左下的结点

**return** (q);

}//结束InSucc

76．[题目分析]在后序序列中，若结点p有右子女，则右子女是其前驱，若无右子女而有左子女，则左子女是其前驱。若结点p左右子女均无，设其中序左线索指向某祖先结点f（p是f右子树中按中序遍历的第一个结点），若f有左子女，则其左子女是结点p在后序下的前驱；若f无左子女，则顺其前驱找双亲的双亲，一直继续到双亲有左子女（这时左子女是p的前驱）。还有一种情况，若p是中序遍历的第一个结点，结点p在中序和后序下均无前驱。

BiThrTree InPostPre (BiThrTree t,p)

//在中序线索二叉树t中，求指定结点p在后序下的前驱结点q

{BiThrTree q;

**if** (p->rtag==0) q=p->rchild; //若p有右子女，则右子女是其后序前驱

**else** **if** (p->ltag==0) q=p->lchild; //若p无右子女而有左子女，左子女是其后序前驱。

**else** **if**(p->lchild==null) q=null;//p是中序序列第一结点，无后序前驱

**else** //顺左线索向上找p的祖先，若存在，再找祖先的左子女

{**while**(p->ltag==1 && p->lchild!=null) p=p->lchild;

**if**(p->ltag==0) q=p->lchild; //p结点的祖先的左子女是其后序前驱

**else** q=null; //仅右单枝树（p是叶子），已上到根结点，p结点无后序前驱

}

**return**(q); }//结束InPostPre

77．[题目分析]在中序穿线二叉树中，求某结点p的后序后继q，需要知道p的双亲结点f的信息。（中序线索树的性质；若p是f的左子女，则f是p的最右子孙的右线索；若p是f的右子女，则f是p的最左子孙的左线索。）找到f后，若p是f的右子女，则p的后继是f；若p是f的左子女，且f无右子女，则p的后继也是f；若p是f的左子女，且f有右子女，则f的右子树中最左边的叶子结点是p的后继。因此，算法思路是，先找p的双亲f，根据p是f的左/右子女再确定p的后序后继。

BiThrTree InThrPostSucc(BiThrTree r,p)

//在中序线索二叉树r上，求结点p（假定存在）的后序后继结点q）

{**if**(p==r)**return**(null) //若p为根结点，后序后继为空

T=p

**while**(T->LT==1) T=T->LL; //找p的最左子孙的左线索

q=T->LL; //q是 p最左子孙的左线索,是p的祖先

**if**(q->RL==p) **return**(q); //p是q的右子女，则q是p后序后继。

T=p;

**while**(T->RT==1) T=T->RL; //找p的最右子孙的右线索

q=T->RL; //q是p最右子孙的右线索

**if**(q->LL=p) //若p是q的左子女

**if**(q->RT==0) **return**(q);//若p是q的左子女且q无右子女，则p的后序后继是q

**else** //p的双亲q有右子树，则q的右子树上最左下的叶子结点是p的后继

{q=q->RL;

**while**(q->LT==1||q->RT==1) //找q右子树中最左下的叶子结点

{**while**(q->LT==1) q=q->LL; //向左下

**if**(q->RT==1) q=q->RL; //若q无左子女但有右子女，则向右下，直到叶子结点

}

**return**(q); //q是的p后继

}

} //结束InThrPostSucc

[算法讨论] 请注意本题条件：标记为0时是线索，而为1时是指向子女。

78．[题目分析]第77题已讨论了在中序线索树中查找结点p的后序后继问题，本题要求在中序线索树上进行后序遍历。因后序遍历是“左右根”，最后访问根结点，即只有从右子树返回时才能访问根结点，为此设一标志**return**flag,当其为1时表示从右侧返回，可以访问根结点。为了找当前结点的后继，需知道双亲结点的信息，在中序线索树中，某结点最左子孙的左线索和最右子孙的右线索均指向其双亲，因此设立两个函数LeftMost和RightMost求结点的最左和最右子孙，为了判定是否是从右子树返回，再设一函数IsRightChild。

BiThrTree LeftMost(BiThrTree t) //求结点t最左子孙的左线索

{BiThrTree p=t;

**while**(p->ltag==0) p=p->lchild; //沿左分枝向下

**if** (p->lchild!=null) **return**(p->lchild); **else** **return**(null);

}//LeftMost

BiThrTree RightMost（BiThrTree t）//求结点t最右子孙的右线索

{BiThrTree p=t;

**while**(p->rtag==0) p=p->rchild; //沿右分枝向下

**if** (p->rchild!=null) **return** (p->rchild); **else** **return**(null);

}//RightMost

**int** IsRightChild(BiThrTree t,father) //若t是father 的右孩子，返回1,否则返回0

{father=LeftMost(t);

**if**(father &&f ather->rchild==t) **return**(1); **else** **return**(0);

}//Is RightChild;

**void** PostOrderInThr (BiThrTree bt) //后序遍历中序线索二叉树bt

{BiThrTree father, p=bt;

**int** flag;

**while**(p!=null)

{**while**(p->ltag==0 ) p=p->lchild; // 沿左分枝向下

**if**(p->rtag==0) flag=0;//左孩子为线索，右孩子为链，相当从左返回

**else** flag=1; //p为叶子，相当从右返回

**while**(flag==1)

{visit(\*p);//访问结点

**if**(IsRightChild(p,father)) {p=father; flag=1;} //修改p指向双亲

**else** //p是左子女，用最右子孙的右线索找双亲

{p=RightMost(p);

**if**(p&&p->rtag==1) flag=1; **else** flag=0;

}

}// **while**(flag==1)

**if**(flag==0 && p!=null) p=p->rchild; //转向当前结点右分枝

} }//结束PostOrderInThr

79．（1）哈夫曼树的构造过程

① 根据给定的n个权值{W1,W2,W3,…,Wn}构成n棵二叉树的集合F={T1,T2,…，Tn},其中每棵二叉树Ti只有权值为Wi的根结点，其左右子树均为空。

② 在F中选取两棵根结点的权值最小的树作左右子树构造一棵新二叉树，新二叉树根结点的权值为其左右子树上根结点的权值之和。

③ 在F中删除这两棵树，同时将新得到的二叉树加入F中。

④ 重复②和③，直到F中只剩一棵树为止。这棵树便是哈夫曼树。

（2）含有n个叶子结点的哈夫曼树共有2n-1个结点，采用静态链表作为存储结构，设置大小为2n-1的数组。现将哈夫曼树的结点及树的结构定义如下：

**typedef** **struct**{**float** weight ; //权值

**int** parent,lc,rc;//双亲、左、右子女 }node ;

**typedef** node HufmTree[2\*n-1];

**void** Huffman(**int** n,**float** w[n],HufmTree T)

//构造n个叶子结点的哈夫曼树T，n个权值己放在W[n]数组中

{**int** i,j,p1,p2 //p1,p2为最小值和次最小值的坐标

**float** small1,small2; //small1和small2为权值的最小值和次小值

**for**(i=0;i<2\*n-1;i++) //置初值，结点的权、左、右子女，双亲

{T[i].parent=-1; T[i].lc=-1; T[i].rc=-1;

**if**(i<n) T[i].weight=w[i]; **else** T[i].weight=0;

}

**for** (i=n ;i<2\*n-1;i++) //构造新二叉树

{p1=p2=0;small1=small2=maxint; //初值

**for**(j=0;j<i;j++)

**if**(T[j].weight<small1 && T[j].parent==-1) //最小值

{p2=p1; small2=small1; p1=j; small1=T[j].weight;}

**else** **if**(T[j].weight<small2 && T[j].parent==-1) //次小值

{p2=j;small2=T[j].weight;}

T[i].weight=T[p1].weight+T[p2].weight; //合并成一棵新二叉树

T[i].lc=p1; T[i].rc=p2; //置双亲的左右子女

T[p1].parent=i; T[p2].parent=i; //置左、右子女的双亲

}//**for**(i=0;i<2\*n-1;i++) }//结束huffman

求哈夫曼编码的算法

**typedef** **struct** {**char** bit[n]; **int** start;}codetype;

**void** HuffmanCode(CodeType code[n],HufmTree T) //哈夫曼树T已求出，现求其哈夫曼编码

{**int** i,j,c,p;

CodeType cd;

**for**(i=0;i<n;i++)

{cd.start=n;c=i;p=T[i].parent;

**while**(p!=-1)

{cd.start--;

**if**(T[p].lc==c) cd.bit[cd.start]=‘0’//左分枝生成代码‘0’

**else** cd.bit[cd.start]=‘1’; // 右分枝生成代码‘1’

c=p; p=T[p].parent; //双亲变为新子女，再求双亲的双亲

}

code[i]=cd; //成组赋值，求出一个叶子结点的哈夫曼编码

}//**for** }//结束HuffmanCode

80．[题目分析] 二叉树的层次遍历序列的第一个结点是二叉树的根。实际上，层次遍历序列中的每个结点都是“局部根”。确定根后，到二叉树的中序序列中，查到该结点，该结点将二叉树分为“左根右”三部分。若左、右子树均有，则层次序列根结点的后面应是左右子树的根；若中序序列中只有左子树或只有右子树，则在层次序列的根结点后也只有左子树的根或右子树的根。这样，定义一个全局变量指针R，指向层次序列待处理元素。算法中先处理根结点，将根结点和左右子女的信息入队列。然后，在队列不空的条件下，循环处理二叉树的结点。队列中元素的数据结构定义如下：

**typedef** **struct**

{ **int** lvl; //层次序列指针，总是指向当前“根结点”在层次序列中的位置

**int** l,h; //中序序列的下上界

**int** f; //层次序列中当前“根结点”的双亲结点的指针

**int** lr; // 1—双亲的左子树 2—双亲的右子树

}qnode;

BiTree Creat(datatype in[],level[],**int** n)

//由二叉树的层次序列level[n]和中序序列in[n]生成二叉树。 n是二叉树的结点数

{**if** (n<1) {printf(“参数错误\n”); exit(0);}

qnode s,Q[]; //Q是元素为qnode类型的队列，容量足够大

init(Q); **int** R=0; //R是层次序列指针，指向当前待处理的结点

BiTree p=(BiTree)malloc(**sizeof**(BiNode)); //生成根结点

p->data=level[0]; p->lchild=null; p->rchild=null; //填写该结点数据

**for** (i=0; i<n; i++) //在中序序列中查找根结点，然后，左右子女信息入队列

**if** (in[i]==level[0]) **break**;

**if** (i==0) //根结点无左子树，遍历序列的1—n-1是右子树

{p->lchild=null;

s.lvl=++R; s.l=i+1; s.h=n-1; s.f=p; s.lr=2; enqueue(Q,s);

}

**else** **if** (i==n-1) //根结点无右子树，遍历序列的1—n-1是左子树

{p->rchild=null;

s.lvl=++R; s.l=1; s.h=i-1; s.f=p; s.lr=1; enqueue(Q,s);

}

**else** //根结点有左子树和右子树

{s.lvl=++R; s.l=0; s.h=i-1; s.f=p; s.lr=1;enqueue(Q,s);//左子树有关信息入队列

s.lvl=++R; s.l=i+1;s.h=n-1;s.f=p; s.lr=2;enqueue(Q,s);//右子树有关信息入队列

}

**while** (!empty(Q)) //当队列不空，进行循环，构造二叉树的左右子树

{ s=delqueue(Q); father=s.f;

**for** (i=s.l; i<=s.h; i++)

**if** (in[i]==level[s.lvl]) **break**;

p=(bitreptr)malloc(**sizeof**(binode)); //申请结点空间

p->data=level[s.lvl]; p->lchild=null; p->rchild=null; //填写该结点数据

**if** (s.lr==1) father->lchild=p;

**else** father->rchild=p； //让双亲的子女指针指向该结点

**if** (i==s.l)

{p->lchild=null; //处理无左子女

s.lvl=++R; s.l=i+1; s.f=p; s.lr=2; enqueue(Q,s);

}

**else** **if** (i==s.h)

{p->rchild=null; //处理无右子女

s.lvl=++R; s.h=i-1; s.f=p; s.lr=1; enqueue(Q,s);

}

**else**{s.lvl=++R; s.h=i-1; s.f=p; s.lr=1; enqueue(Q,s);//左子树有关信息入队列

s.lvl=++R; s.l=i+1; s.f=p; s.lr=2; enqueue(Q,s); //右子树有关信息入队列

}

}//结束**while** (!empty(Q))

**return**(p);

}//算法结束