第 5 章　树与二叉树

5.1　树的基本概念

5.1.1　树的概念

1. 树的定义

树是树状结构的简称，它是一种重要的非线性结构。树的定义是：

树是 n（n≥0）个结点的有限结合，当 n = 0 时称为空树，在任一非空树（n＞0）中，它有且仅有一个称为根的结点，其余的结点可分为 m 棵（m≥0）互不相交的子树（称为根的子树），每棵子树又同样是一棵树。

树的定义是分层的、递归的，树是一种递归的数据结构，递归结束于叶结点。

2. 树的逻辑表示

树的逻辑表示有如图 5-1 所示的 5 种。

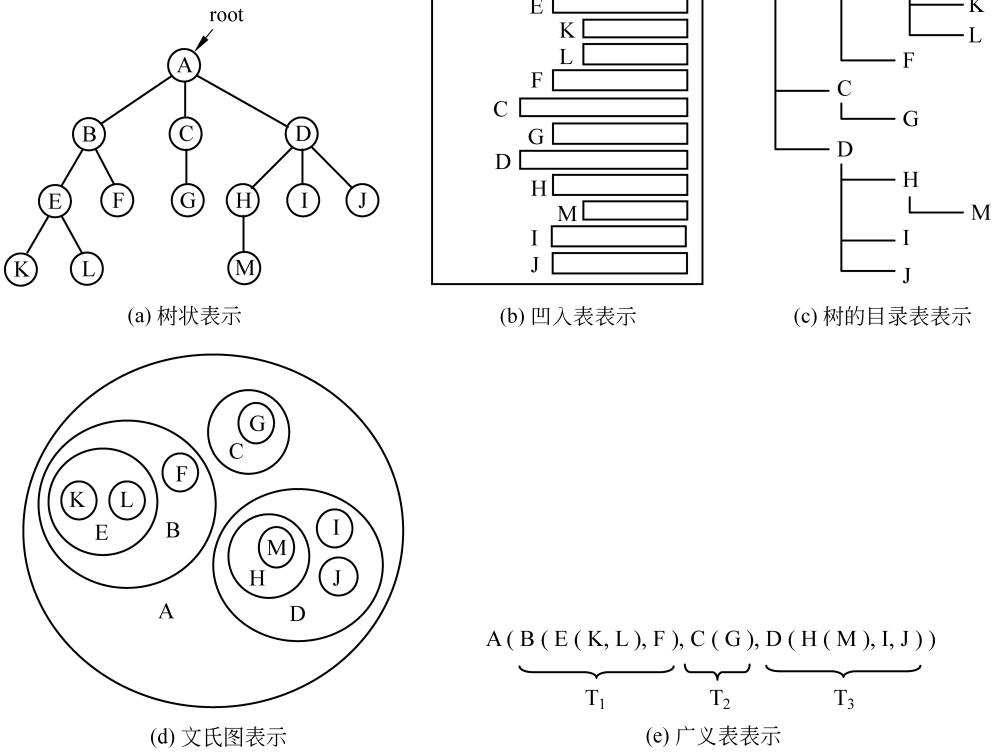


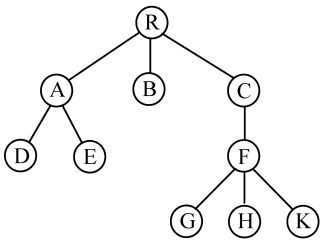
图 5-1　树的逻辑表示

5.1.2　树的双亲存储表示

1. 双亲存储表示的结构定义

用一个一维数组存储树中的结点，同时在每个结点中附设一个指针指向该结点的双亲结点在数组中的位置，如图 5-2 所示，其类型定义（保存于 PTree.h 头文件中）如下。

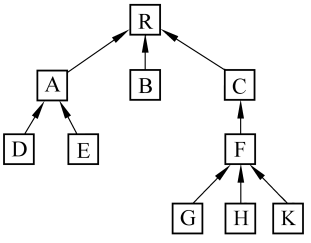
```
typedef char TElemType;                    //结点数据的类型
typedef struct node {                      //树结点类型定义
     TElemType data;                       //结点数据
     int parent;                           //结点双亲指针
} PTNode;
typedef struct {                           //树类型定义
     PTNode tnode[maxSize];                //双亲指针表示
     int n;                                //现有结点数
} PTree;
```



(a) 一棵树

	data	parent
0	R	−1
1	A	0
2	B	0
3	C	0
4	D	1
5	E	1
6	F	3
7	G	6
8	H	6
9	K	6

(b) 双亲表示



(c) 双亲表示图示

图 5-2　树的双亲表示法

树的所有结点在双亲指针数组中的排列，可以按照先根遍历次序，也可以按照层次遍历次序。无论哪一种存放方式，根结点都在数组的最前端，它的双亲指针为−1。然而查找某个给定结点的子女和兄弟的运算因存放顺序不同而有所不同。

2. 树的双亲存储表示相关的算法

5-1　一棵树采用双亲表示存储，设计一个算法，实现树的初始化运算 InitTree。

【解答】 用双亲表示的树的初始化运算就是把每个结点的双亲指针置为“−1”，实际上构造了一个每棵树只有根结点的森林。算法的实现如下。

```
void InitTree(PTree& T, int sz) {
//算法调用方式 InitTree(T, sz)。输入：用双亲表示存储的树 T，树中结点个数 sz;
//输出：将 T 置空
     T.n = sz;
     for(int i = 0; i < sz; i++) T.tnode[i].parent = -1;
}
```

若双亲指针数组的元素个数为 n，算法的时间复杂度为 O(n)，空间复杂度为 O(1)。

5-2　一棵树采用双亲表示存储，设计一个算法，查找结点 i 的第一个子女 FirstChild。

【解答】 在双亲表示中结点 i 的所有子女都排列在结点 i 之后。为查找指定结点 i 的第一个子女，从 i 开始扫描双亲指针数组，当首次遇到某结点 j 的双亲是 i，该结点 j 即是结点 i 的第一个子女。算法的实现如下。

```
int FirstChild(PTree& T, int i) {
//算法调用方式 int k = FirstChild(T, i)。输入：用双亲表示存储的树 T，指定树结点
//下标 i；输出：若结点 i 有子女，则函数返回第一个子女的下标，否则返回-1
     for(int j = i+1; j < T.n; j++)
          if(T.tnode[j].parent == i) break;
     if(j < T.n) return j;                          //找到结点 i 的第一个子女 j
     else return -1;
}
```

若双亲指针数组的元素个数为 n，算法的时间复杂度为 O(n)，空间复杂度为 O(1)。

5-3　一棵树采用双亲表示法存储，设计一个算法，查找结点 i 的子女结点 j 的下一个兄弟 NextSibling。

【解答】 为查找结点 i 的子女 j 的下一个兄弟，从结点 j 开始扫描双亲指针数组，当发现某结点 k 的双亲为 i 立即停止查找，此结点即为所求；若数组扫描到最后没有找到双亲为 i 的结点，则不存在 i 的下一个子女。算法的实现如下。

```
int NextSibling(PTree& T, int i, int j) {
//算法调用方式 int k = NextSibling(T, i, j)。输入：用双亲表示存储的树 T，指定
//树结点下标 i，它的某个子女结点 j；输出：若结点 i 第 j 个子女有下一个兄弟，则函数
//返回下一个兄弟的下标；若没有下一个兄弟，函数返回-1
     for(int k = j+1; k < T.n; k++)
         if(T.tnode[k].parent == i) break;
     if(k < T.n) return k;                    //找到下一个兄弟结点
     else return -1;
}
```

若双亲指针数组的元素个数为 n，算法的时间复杂度为 O(n)，空间复杂度为 O(1)。

5-4　一棵树采用双亲表示存储，设计一个算法，查找结点 i 的双亲 FindParent。

【解答】 结点 i 的双亲可以通过该结点的双亲指针直接找到。算法的实现如下。

```
int FindParent(PTree& T, int i) {
//算法调用方式 int k = FindParent(T, i)。输入：用双亲表示存储的树 T，指定树结
//点下标 i；输出：若结点 i 有双亲，则函数返回双亲结点的下标，否则返回-1
     if(i < T.n && i > 0) return T.tnode[i].parent;
     else return -1;
}
```

算法的时间复杂度和空间复杂度均为 O(1)。

5-5　若一棵树采用双亲表示存储，设计一个算法，输入树的广义表表示，例如"A(B(E(K, L), F), C(G), D(H(M), I, J)) #"，创建用双亲表示存储的树（最后的"#"是控制广义表串结束的符号）。

【解答】 若一棵树的广义表表示为 A(B(E(K, L), F), C(G), D(H(M), I, J))，广义表的名字即树根，名字后紧跟的括号内是它的子孙，以此类推。因为在双亲数组中是按元素出现先后存放的，实际上是一个先根次序。为此需要用到一个栈，在遇到“(”时保存子树根结点的地址，以便括号内的元素链接到它们的双亲；在遇到“)”时退掉根结点。为了控制处理结束，约定在广义表的最后加上“#”作为结束。算法的实现如下。

```
#define stackSize 20                                          //栈的最大容量
void CreatePTree(PTree& T, TElemType G[]) {
//算法调用方式 CreatePTree(T, G)i。输入：已初始化的空树 T，树的广义表表示 G;
//输出：引用参数 T 返回创建出来的树的双亲存储表示，根在 0 号位置，所有结点在
//双亲指针数组中按先根次序排列
     TElemType S[stackSize];  int top = -1;                   //设置栈并置空
     int i, j, k = -1;  TElemType ch;                         //k 是存指针，i 是取指针
     for(i = 0; G[i] != '#'; i++) {                           //循环取广义表字符
          ch = G[i];
          switch(ch) {                                        //按不同字符处理
          case '(': S[++top] = k;  break;
          case ')': j = S[top--];  break;
          case ',': break;
          default : T.tnode[++k].data = ch;
                if(top == -1) T.tnode[k].parent = -1;
                else { j = S[top];  T.tnode[k].parent = j; }
          }
     }
     T.n = k+1;
}
```

若树的广义表表示字符串中有 n 个字符，算法的时间复杂度和空间复杂度均为 O(n)。

5-6　若一棵树采用双亲表示存储，设计一个算法，以广义表表示的字符序列输出这棵树。

【解答】 算法采用递归方式输出，首先输出根，然后取根的第一个子女 j，若有则先输出一个左括号“(”以示有子树，再递归输出以 j 为根的子树；子树处理完退回后再取 j 的下一个兄弟 j，若有则先输出一个逗号“,”再递归输出 i 的下一棵子树；若没有则表示以 i 为根的树输出完，最后输出一个右括号“)”结束。算法的实现如下。

```
void printPTree_Gen(PTree& T, int i) {
//算法调用方式 printPTree_Gen(T, i)。输入：用双亲表示存储的树 T，指定树结点下
//标 i(若从根开始 i 给 0)；输出：算法按广义表表示输出树 T
     printf("%c", T.tnode[i].data);                           //输出根的数据
     int j;
     if((j = FirstChild(T, i)) != -1) printf("(");            //有子女,输出左括号
     while(j != -1) {
          printPTree_Gen(T, j);                               //递归输出第一棵子树
          j = NextSibling(T, i, j);                           //下一棵子树
          if(j != -1) printf(",");
          else printf(")");
```

```
        }
    }
```

若树中结点个数为 n，树的深度为 d，算法的时间复杂度为 O(n×d)，空间复杂度为 O(d)。

5-7　一棵树采用双亲表示存储，设计一个算法，计算树的深度。

【解答】 对双亲数组中的每一个树结点，分别计算沿 parent 指针链走过的结点数，此即该结点的深度，统计所有结点的深度的最大值就可求得树的深度。算法的实现如下。

```
int Depth_PTree(PTree& T) {
//算法调用方式 int d = Depth_PTree(T)。输入：用双亲表示存储的树 T；输出：函数
//返回树 T 的深度
     int i, j, depth, maxDepth = 0;
     for(i = 0; i < T.n; i++) {
          depth = 0;
          for(j = i; j >= 0; j = T.tnode[j].parent) depth++;
                                                          //求每一结点的深度
          if(depth > maxDepth) maxDepth = depth;          //求深度的最大值
     }
     return maxDepth;
}
```

若树中结点个数为 n，树的深度为 d，算法的时间复杂度为 O(n×d)，空间复杂度为 O(1)。

5-8　一棵树采用双亲表示存储，树中根结点存于 T.tnode[0]。设计一个算法，计算 p 所指结点和 q 所指结点的最近公共祖先结点。

【解答】 首先计算 p、q 所指结点的深度，再将较深的结点沿双亲方向上溯，直到与另一方结点处于同一深度。然后双方同时上溯，直到重合到一个结点，此结点即为所求。算法的实现如下。

```
int CommonAncestry(PTree& T, int p, int q) {
//算法调用方式 int k = CommonAncestry(T, p, q)。输入：用双亲表示存储的树 T，树
//中的两个结点 p 和 q；输出：函数返回 p 和 q 的最近公共祖先结点 k
     int i = p, j = q, dp = 0, dq = 0;
     while(i != -1) { i = T.tnode[i].parent;  dp++;}    //计算 p 结点深度
     while(j != -1) { j = T.tnode[j].parent;  dq++;}    //计算 q 结点深度
     for(i = p; dp > dq; i = T.tnode[i].parent, dp--); //双方深度找平
     for(j = q; dp < dq; j = T.tnode[j].parent, dq--); //以上两句仅一条执行
     while(i != j)                                      //找共同祖先 i = j
          { i = T.tnode[i].parent ;  j = T.tnode[j].parent ; }
     return i;
}
```

若树中结点个数为 n，树的深度为 d，算法的时间复杂度为 O(n×d)，空间复杂度为 O(1)。

5-9　一棵树采用双亲表示存储，设计一个算法，统计该树中叶结点个数。

【解答】 若树中有 n 个结点，设置一个辅助数组 count[n]，记录各结点的度，初始均为 0。算法扫描一遍双亲指针数组，若双亲指针等于 k，则 count[k]加 1（除根结点的双亲指针为−1 外）。最后 count[] = 0 的即为叶结点。统计叶结点个数即可。算法的实现如下。

```
int leafCount(PTree& T) {
       int count[maxSize], i, k = 0;
       for(i = 0; i < T.n; i++) count[i] = 0;
       for(i = 0; i < T.n; i++)                              //统计各结点的度
          if(T.tnode[i].parent != -1) count[T.tnode[i].parent]++;
       for(i = 0; i < T.n; i++)
          if(count[i] == 0) k++;                             //统计度为 0 的结点数
       return k;                                             //返回叶结点数
}
```

若树中结点个数为 n，算法的时间复杂度和空间复杂度均为 O(n)。

5-10　一棵树采用双亲表示法存储，设计一个算法，计算该树的度。

【解答】 与题 5-9 类似，使用一个辅助数组 count[]统计各结点的度，再统计各结点的度的最大值，从而得到树的度。算法的实现如下。

```
int Degree(PTree& T) {
//算法调用方式 int k = Degree (T)。输入：用双亲表示存储的树 T;
//输出：函数返回树的度数
     int count[maxSize], i, d = 0;
     for(i = 0; i < T.n; i++) count[i] = 0;
     for(i = 0; i < T.n; i++)                                //统计各结点的度
         if(T.tnode[i].parent != -1) count[T.tnode[i].parent]++;
     for(i = 0; i < T.n; i++)
         if(count[i] > d) d = count[i];                  //统计各结点度的最大值
     return d;                                               //返回叶结点数
}
```

若树中结点个数为 n，算法的时间复杂度和空间复杂度均为 O(n)。

5.1.3　树的子女链表存储表示

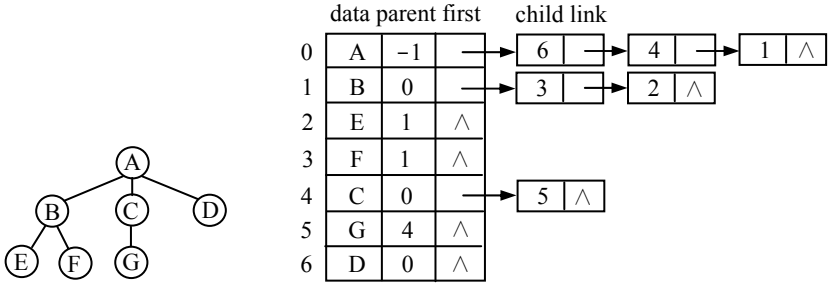
1. 子女链表存储表示的结构定义

这种存储表示方法为树中每个结点设置一个子女链表，并将这些结点的数据和对应子女链表的头指针放在一个向量中。例如，对于图 5-3（a）所示的树，其子女链表表示如图 5-3（b）所示。在各结点的子女链表中，child 是子女结点在向量中的下标（序号），link 是该子女的下一兄弟的链接指针。

由此可得树的子女链表存储表示的类型定义如下（存放于头文件 ChildList.h 中）。

```
#define maxNodes 30
typedef char TElemType;              //结点数据类型（大写字母表示）
typedef struct vnode {               //树结点的结构定义
TElemType data;                      //结点数据
     int parent;                     //双亲结点在结点表中位置
     struct vnode *first;            //子女链表头指针
} VNode;
typedef struct enode {               //子女链表结点的结构定义
```

```
    int child;                          //子女结点在结点表中位置
    struct enode *link;                 //下一结点链接指针
} ENode;
typedef struct {                        //子女链表存储表示
    VNode NodeList[maxNodes];           //结点表
    int n;                              //当前结点个数
} ChildList;
```



(a) 一棵树　　(b) 带双亲指针的子女链表

图 5-3　树的子女链表表示法

一般约定根结点放在向量的 0 号位置，其他各个结点在向量中或者按照先根次序，或者按照层次次序排列。此外，在子女链表中各个子女的先后位置在无序树的情形下任意，而在有序树中，必须按各子树的自左到右的次序依次链接。在很多应用中，把子女链表和双亲表示结合在一起，这样处理亲子关系比较方便，如图 5-3 所示。

2. 子女链表存储表示相关的算法

5-11　一棵树采用子女链表存储表示存储，设计一个算法，实现树的初始化运算 InitTree。

【解答】 初始化就是把子女链表置空。因此可通过一个循环，把所有结点的子女链头指针置空，把它们的双亲指针置为-1。算法的实现如下。

```
void InitCTree(ChildList& T);          //初始化函数
//算法调用方式 InitCTree(T)。输入：用子女链表存储的树 T；输出：把所有结点的
//子女链头指针置空，把它们的双亲指针置为-1
    for(int i = 0; i < maxNodes; i++)
        { T.NodesList[i].first = NULL;  T.NodesList[i].parent = -1; }
    T.n = 0;
}
```

若结点表的最大结点数为 n，算法的时间复杂度为 O(n)，空间复杂度为 O(1)。

5-12　若用大写字母标识树的结点，则可用带标号的广义表形式表示一棵树，其语法图如图 5-4 所示。例如，图 5-3（a）中的树可用广义表表示为“A(B(E, F), C(G), D)”，设计一个算法，由这种广义表表示的字符序列构造树的子女链表表示。

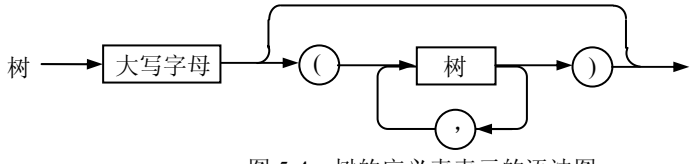


图 5-4　树的广义表表示的语法图

【解答】 根据语法图，一棵树首先有一个用大写字母标识的根，然后用一对括号给出根的子树，各子树之间用逗号间隔。子树又是树，可以继续用此语法图识别。下面给出的算法是非递归实现，其中用到了一个栈，记录子树的根：遇到左括号，根进栈；遇到右括号，根退栈。叶结点不进栈。算法的实现如下。

```
#define size 25
bool createCTree(ChildList& T, char A[], int n) {
//算法调用方式 bool succ = createCTree(T, A, n)。输入：已初始化的用子女链表
//存储的树 T，存放输入数据序列的数组 A，数组元素个数 n；输出：若建树成功，函数
//返回 true，引用参数 T 返回已创建的树；若输入有误，函数返回 false
     char ch = A[0];  ENode *p;  ENode *r[size];
     for(int j = 0; j < size; j++) r[j] = NULL;     //各结点子女链尾指针
     int S[size];  int top = -1, pos = 0, i = 1;
     T.NodeList[0].data = ch;                  //根存入结点表，位置 pos = 0
     if(A[i] == '(') {                         //子树非空
          S[++top] = pos;                      //根进栈
          while(top != -1) {                   //栈空,创建树完成，不空，创建子树
             ch = A[++i];
             switch(ch) {
             case '(' : S[++top] = pos;  break;          //当前树结点进栈
             case ')' : top--;  break;                   //树结点退栈
             case ',' : break;                           //处理下一个子女
             default:
                T.NodeList[++pos].data = ch;             //创建结点
                T.NodeList[pos].parent = S[top];         //创建双亲指针
                p =(ENode *) malloc(sizeof(ENode));
                p->child = pos;  p->link = NULL;         //创建子女结点
                if(T.NodeList[S[top]].first == NULL)
                   T.NodeList[S[top]].first = p;
                else r[S[top]]->link = p;
                r[S[top]] = p;
             }
          }
          T.n = pos+1;  return true;
     }
     else { printf("输入有误! \n");  return false; };
}
```

若输入数据数组的元素个数为 n，创建成功树的深度为 d，算法的时间复杂度为 O(n+d)，空间复杂度为 O(d)。

5-13　若树以子女链表表示存储，设计一个算法，以广义表表示的字符序列输出这棵树。

【解答】 算法采用递归方式输出，首先输出根，然后判断根的子女链是否为空；若非空则在一对括号之间递归输出根的子树，中间用逗号隔开。算法的实现如下。

```
void printCTree(ChildList& T, int i) {
```

```
//算法调用方式 printCTree(T, i)。输入：用子女链表存储的树 T，子树的
//根结点号 i；输出：递归地输出树 T 的广义表表示
     printf("%c", T.NodeList[i].data);
     if(T.NodeList[i].first != NULL) {                    //非叶子结点
          printf("(");
          ENode *p = T.NodeList[i].first;
          while(p != NULL) {
               printCTree(T, p->child);                   //递归输出子树
               if(p->link != NULL) printf(",");           //最后子女后不加逗号
               p = p->link;
          }
          printf(")");
     }
}
```

若树中结点个数为 n，递归算法的时间复杂度和空间复杂度均为 O(n)。

5-14　若树以子女链表表示存储，设计一个算法，以凹入表形式输出这棵树。

【解答】 算法使用了一个层次标志 i 控制输出各层次时向右移格的数目。算法先输出子树根结点 e 的值，再对它的子女依次递归输出。算法的实现如下。

```
void printCTree_1(ChildList& T, int e, int i) {
//算法调用方式 printCTree_1(T, e, i)。输入：用子女链表存储的树 T，当前输出子树
//的根在结点表中的下标 e，层次号 i；输出：按凹入表形式分层输出树的所有结点
     int j;  ENode *p;  char ch;
     for(j = 0; j < 5*(i-1); j++) printf(" ");      //留出空格以表现出层次
     ch =(e == 0) ? '#' : T.NodeList[T.NodeList[e].parent].data;
     printf("%c(%c)\n", T.NodeList[e].data, ch);    //输出结点数据，换行
     for(p = T.NodeList[e].first; p != NULL; p = p->link)
         printCTree_1(T, p->child, i+1);            //顺序输出各个子树
}
```

若树中结点个数为 n，递归算法的时间复杂度和空间复杂度均为 O(n)。

5-15　若树以子女链表表示存储，设计一个算法，对于结点表中第 i 个结点，计算以它为根的子树中的结点个数。

【解答】 采用递归算法：若当前结点为叶结点，则返回 1，这是递归的结束条件；若不是叶结点，则扫描该结点的子女链表，统计每个子女为根的子树中的结点个数，把它们累加起来，再加 1，得到该结点为根的子树的结点个数。算法的实现如下。

```
int count_CTree(ChildList& T, int i) {
//算法调用方式 int k = count_CTree(T, i)。输入：用子女链表存储的树 T，当前处理
//子树的根结点 i；输出：函数返回树 T 中以 i 为根子树的结点个数，主程序调用时
//参数 i 赋给 0 值，代表全树的根
     if(T.NodeList[i].first == NULL) return 1;          //叶结点
     else {
          int k = 0;
          for(ENode *q = T.NodeList[i].first; q != NULL; q = q->link)
              k += count_CTree(T, q->child);
```

```
            return k+1;
        }
    }
```

若树中结点个数为 n，递归算法的时间复杂度和空间复杂度均为 O(n)。

5-16　若树以子女链表表示存储，设计一个算法，计算树的度。

【解答】在树的结点表中，检查每一个结点，统计结点的子女链表中子女结点的个数，作为该结点的度，取所有结点的最大值，就可得到树的度。算法的实现如下。

```
int degree_CTree(ChildList& T) {
//算法调用方式 int k = degree_CTree(T)。输入：用子女链表存储的树 T;
//输出：函数返回树 T 的度
     int i, d, maxd = 0;  ENode *p;
     for(i = 0; i < T.n; i++) {
         d = 0;
         for(p = T.NodeList[i].first; p != NULL; p = p->link) d++;
         if(d > maxd) maxd = d;              //存储子树度最大值
     }
     return maxd;
}
```

若树的分支数为 e，算法的时间复杂度为 O(e)，空间复杂度为 O(1)。

5-17　若树以子女链表表示存储，设计一个算法，计算树的高度。

【解答】采用递归算法，对于叶结点，直接返回 1，表示以该结点为根的子树只有一个结点；否则扫描第 i 个结点，计算子女链表中每个子女的高度，取其最大值，再加 1，即可得到该结点的高度。算法的实现如下。

```
int height_CTree(ChildList& T, int i) {
//算法调用方式 int k = height_CTree(T)。输入：用子女链表存储的树 T;
//输出：函数返回子树 i 的高度。主程序调用时 i 给 0（根结点的序号为 0）
     if(T.NodeList[i].first == NULL) return 1;      //叶结点高度为 1
     int d, maxd = 0;  ENode *p;
     for(p = T.NodeList[i].first; p != NULL; p = p->link) {
          d = height_CTree(T, p->child);
          if(d > maxd) maxd = d;                    //存储所有子树最大高度
     }
     return maxd+1;                                 //树的高度为最大高度加 1
}
```

若树的分支数为 e，树的高度为 h，算法的时间复杂度为 O(e)，空间复杂度为 O(h)。

5-18　若树以子女链表表示存储，设计一个算法，求指定结点 i 的第一个子女。

【解答】若结点表第 i 个结点的 first 域不空，则 first 指针所指结点 child 域中存放的即为结点 i 的第一个子女。算法的实现如下。

```
int FirstChild(ChildList& T, int i) {
//算法调用方式 int j = FirstChild(T, i)。输入：用子女链表存储的树 T, 指定结点 i
//(0≤i≤maxNodes); 输出：函数返回树 T 中结点 i 的第一个子女的结点号
```