第6章 树与二叉树

- 树的定义与基本概念
- 二叉树
- 二叉树遍历
- 二叉树的计数
- 线索二叉树
- 树与森林

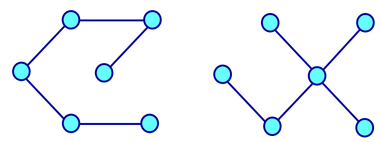
树和森林的概念

- 两种树：自由树与有根树。
- 自由树：

  一棵自由树 T_f 可定义为一个二元组

  $T_f = (V, E)$

  其中 $V = \{v_1, ..., v_n\}$ 是由 n $(n > 0)$ 个元素组成的有限非空集合，称为顶点集合。$E = \{(v_i, v_j) \mid v_i, v_j \in V, 1 \le i, j \le n\}$ 是 $n-1$ 个序对的集合，称为边集合，E 中的元素 (v_i, v_j) 称为边或分支。



自由树

- 有根树：

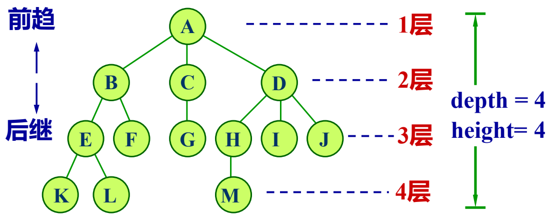
  一棵有根树 T，简称为树，它是 n $(n \ge 0)$ 个结点的有限集合。当 $n = 0$ 时，T 称为空树；否则，T 是非空树，记作

$$T = \begin{cases} \Phi, & n = 0 \\ \{r, T_1, T_2, ..., T_m\}, & n > 0 \end{cases}$$

- r 是一个特定的称为根(root)的结点，它只有直接后继，但没有直接前驱；
- 根以外的其他结点划分为 m $(m \ge 0)$ 个互不相交的有限集合 $T_1, T_2, …, T_m$，每个集合又是一棵树，并且称之为根的子树。

树的特点

- 树是分层结构，又是递归结构。每棵子树的根结点有且仅有一个直接前趋，但可以有 0 个或多个直接后继。

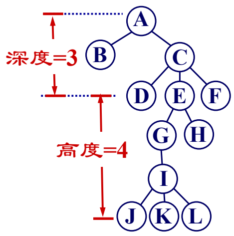


术语

- **结点（node）**：包含数据元素的值及相关指针的存储单位。
- **结点的度（degree）**：结点所拥有的子树棵数。
- **叶结点（leaf）**：度为0的结点，又称终端结点。
- **分支结点（branch）**：除叶结点外的其他结点，又称为非终端结点或非叶结点。
- **子女（child）**：若结点 x 有子树，则子树的根结点即为结点 x 的子女。
- **双亲（parent）**：又称为父结点。若结点 x 有子女，它即为子女的双亲。

- **兄弟（sibling）**：同一双亲的子女互称为兄弟。
- **祖先（ancestor）**：从根结点到该结点所经分支上的所有结点。
- **子孙（descendant）**：某一结点的子女，以及这些子女的子女都是该结点的子孙。
- **结点间的路径（path）**：树中任一结点v_i经过一系列结点$v_1, v_2, ..., v_k$到v_j，其中$(v_i, v_1), (v_1, v_2), ..., (v_k, v_j)$是树中的分支，则称$v_i, v_1, v_2, ..., v_k, v_j$是$v_i$与$v_j$间的路径。
- **结点的深度（depth）**：结点所处层次，即从根到该结点的路径上的分支数加一。根结点在第1层。

- 结点的深度和结点的高度是不同的。**结点的深度**即结点所处层次，是从根向下逐层计算的；**结点的高度**是从下向上逐层计算的：叶结点的高度为1，其他结点的高度是取它的所有子女结点最大高度加一。
- 树的深度与高度相等。**树的深度**按离根最远的叶结点算，**树的高度**按根结点算，都是6。
- 非叶结点包括根结点。

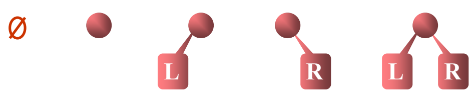


- **树的度（degree）**：树中结点的度的最大值。
- **有序树（ordered tree）**：树中根结点的各棵子树$T_1, T_2, \ldots$是有次序的，即为有序树。其中，T_1叫做根的第1棵子树，T_2叫做根的第2棵子树，…。
- **无序树**：树中结点的各棵子树之间的次序是不重要的，可以互相交换位置。
- **森林（forest）**：m（m≥0）棵树的集合。在数据结构中，删去一棵非空树的根结点，树就变成森林（不排除空的森林）；反之，若增加一个根结点，让森林中每一棵树的根结点都变成它的子女，森林就成为一棵树。

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。
- 这个定义是递归的。



二叉树的五种不同形态

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]

- $i = 1$时，根结点只有1个，$2^{1-1} = 2^0 = 1$；
- 若设 $i = k$ 时性质成立，即该层最多有 2^{k-1} 个结点，则当 $i = k+1$ 时，由于第 k 层每个结点最多可有 2 个子女，第 $k+1$ 层最多结点个数可有 $2 \cdot 2^{k-1} = 2^k$ 个，故性质成立。

性质2

高度为 h 的二叉树最多有 2^h-1个结点。$(h\geqslant 1)$

[证明用求等比级数前k项和的公式]

高度为 h 的二叉树有 h 层，各层最多结点个数相加，得到等比级数，求和得：

$$2^0+2^1+2^2+\ldots+2^{h-1}=2^h-1$$

- 空树的高度为 0，只有根结点的树的高度为 1。

性质3

对任何一棵二叉树，如果其叶结点有 n_0 个，度为2的非叶结点有 n_2 个， 则有

$$n_0=n_2+1$$

证明：

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

$n=n_0+n_1+n_2 \qquad e=2n_2+n_1=n-1$

因此，有 $2n_2+n_1=n_0+n_1+n_2-1$

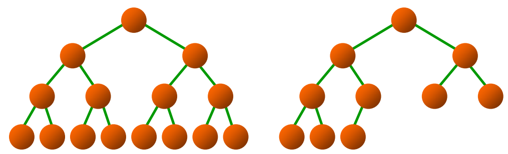
$n_2=n_0-1 \Longrightarrow n_0=n_2+1$

- 引申：可用于判断二叉树各类结点个数。

定义1 满二叉树 (Full Binary Tree)

定义2 完全二叉树 (Complete Binary Tree)

- 若设二叉树的高度为 h，则共有 h 层。除第 h 层外，其它各层 $(1\sim h-1)$ 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。



性质4

具有 $n\ (n\geqslant 0)$ 个结点的完全二叉树的高度为

$\lceil \log_2(n+1) \rceil$

证明：设完全二叉树的高度为 h，则有

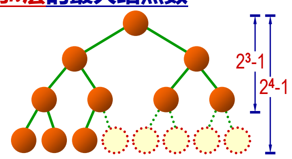
$$\underbrace{2^{h-1}-1}_{\text{上面}h-1\text{层结点数}} < n \leqslant \underbrace{2^h-1}_{\text{包括第}h\text{层的最大结点数}}$$

变形 $2^{h-1}<n+1\leqslant 2^h$

取对数

$h-1<\log_2(n+1)\leqslant h$

有 $h=\lceil \log_2(n+1) \rceil$



- 求高度的另一公式为 $\lfloor \log_2 n \rfloor+1$，它的推导如下：

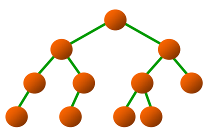
  由 $2^{h-1}-1<n\leqslant 2^h-1$

  得 $2^{h-1}-1\leqslant n-1<2^h-1$

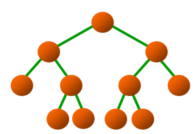
  即 $2^{h-1}\leqslant n<2^h$ 取对数，有 $h-1\leqslant \log_2 n<h$

  最后得 $h=\lfloor \log_2 n \rfloor+1$。
- 注意，此式对于 $n=0$ 不适用。
- 若设完全二叉树中叶结点有 n_0 个，则该二叉树总的结点数为 $n=2n_0$，或 $n=2n_0-1$。
- 若完全二叉树的结点数为奇数，没有度为1 的结点；为偶数，有一个度为1的结点。

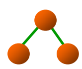
其他几种典型的二叉树



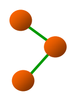
理想平衡树或丰满树



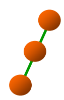
严格二叉树或正则二叉树



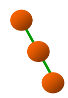
双枝树



单枝树



左斜单枝树

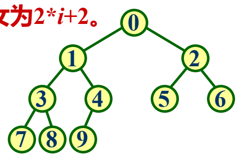


右斜单支树

性质5

如将一棵有 n 个结点的完全二叉树自顶向下，同一层自左向右连续给结点编号: 0, 1, 2, …, n-1，则有以下关系:

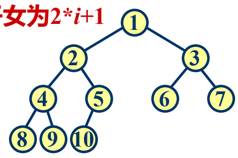
- 若 $i = 0$, 则 i 无双亲;
  若 $i > 0$, 则 i 的双亲为 $\lfloor (i-1)/2 \rfloor$。
- 若 $2*i+1 < n$, 则 i 的左子女为 $2*i+1$;
  若 $2*i+2 < n$, 则 i 的右子女为 $2*i+2$。
- 若 i 为偶数, 且 i != 0, 则其左兄弟为 i-1;
  若 i 为 奇数, 且 i != $n-1$, 则其右兄弟为 $i+1$。



注意:

如果完全二叉树各层次结点从 1 开始编号: 1, 2, 3, …, n，则有以下关系:

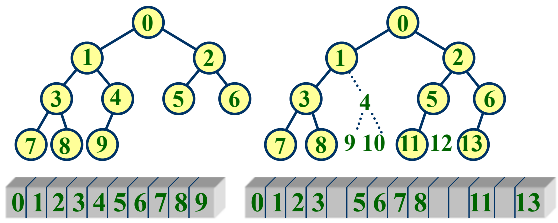
- 若 $i = 1$, 则 i 无双亲;
  若 $i > 1$, 则 i 的双亲为 $\lfloor i / 2 \rfloor$。
- 若 $2*i <= n$, 则 i 的左子女为 $2*i$;
  若 $2*i+1 <= n$, 则 i 的右子女为 $2*i+1$
- 若 i 为奇数, 且 i != 1, 则其左兄弟为 $i-1$;
  若 i 为偶数, 且 i != n, 则其右兄弟为 $i+1$。



完全二叉树的顺序存储表示

- 对于一棵完全二叉树，可将所有结点按其编号，顺序存储到一维存储数组的对应位置。
- 例如，编号为 0 的结点存放到数组的 0 号位置，编号为 1 的结点存放到数组的 1 号位置，编号为 i 的结点存放在数组的第 i 号位置。
- 可以按照完全二叉树的性质 5，很方便地寻找某个结点的左、右子女、双亲和兄弟。
- 对于一般二叉树，必须仿照完全二叉树对结点编号，即使有缺失也要编号，再按完全二叉树存储。

二叉树的顺序表示示例



完全二叉树
的顺序表示

一般二叉树
的顺序表示

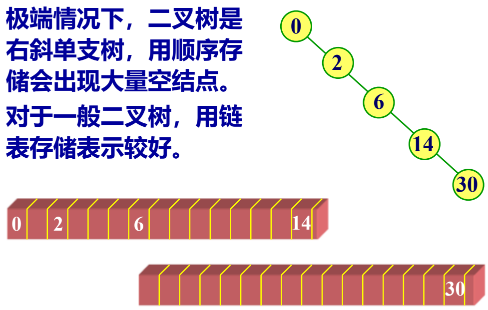
二叉树顺序存储表示的结构定义

```
#define maxSize 128
typedef char TElemType;          //元素数据类型
typedef struct {
    TElemType data[maxSize];     //存储数组
    int n;                       //当前结点个数
} SqBTree;
```

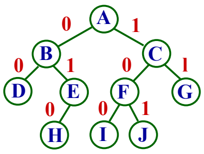
- 对于完全二叉树，因结点编号连续，数据存储密集，适于用顺序表示。但对于一般二叉树，由于性状不规则，结点编号不连续，数组存放较乱。

极端情形: 只有右单支的二叉树

- 极端情况下，二叉树是右斜单支树，用顺序存储会出现大量空结点。
- 对于一般二叉树，用链表存储表示较好。



- 一棵二叉树，如图所示。分支用三元组{p, c, d}描述，p 是双亲数据，c 是子女数据，d 是分支方向，= 0 是左分支，= 1 是右分支。



- 上图有 9 个分支，按层从根开始，各分支为
  {'A', 'B', '0'}，{'A', 'C', '1'}，{'B','D', '0'}，
  {'B', 'E', '1'}，{'C', 'F', '0'}，{'C', 'G', '1'}，
  {'E', 'H', '0'}，{'F', 'I', '0'}，{'F', 'J', '1'}
- 现在要求按层顺序输入各分支，建立二叉树。

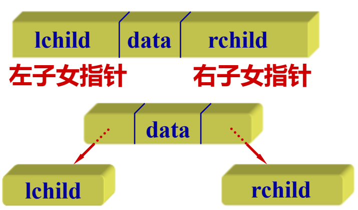
- 第一个分支的双亲一定是根，它应存放在顺序表示数组的 0 号位置，设它为 j。
- 其子女根据 d 的值，若为左子女则其值应存放在数组的第 $2*j+1$ 的位置，若为右子女则其值应存放在数组的第 $2*j+2$ 位置。
- 除第一个分支外，后续每个分支都要在已存放的结点数据中查找双亲 p 的位置，找到后设其位置在 j，就可以插入这个分支。
- 因为有可能建立的二叉树不是完全二叉树，算法用变量 k 控制最后插入的结点位置。


```
#include "SqBTree.h"
int createSqBTree ( SqBTree& BT, char a[ ][3], int n ) {
//从字符数组 a 中按层输入 n 条分支，建立二叉树BT
//的顺序存储表示BT，函数返回实际占用存储个数
   int i, j, k = 0;
   for ( i = 0; i < maxSize; i++ ) BT.data[i] = '#'; //初始化
   BT.data[k] = a[0][0];        //根是第一个分支的双亲
   for ( i = 0; i < n; i++ ) {         //逐个分支处理
      for ( j = 0; j <= k; j++ )       //查各分支双亲位置
         if ( a[i][0] == BT.data[j] )          //查到
            if ( a[i][2] == '0' ) {     //插入左分支
```



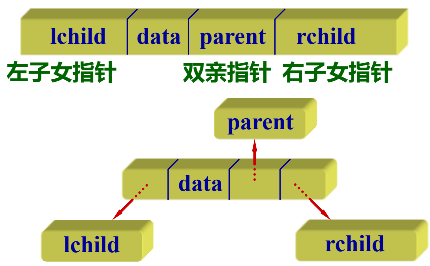
```
                 BT.data[2*j+1] = a[i][1];  k = 2*j+1;
                 break;
              }
              else {                  //插入右分支
                 BT.data[2*j+2] = a[i][1];  k = 2*j+2;
                 break;
              }
     }
     BT.n = n;  return k;
}
```

二叉树的二叉链表表示



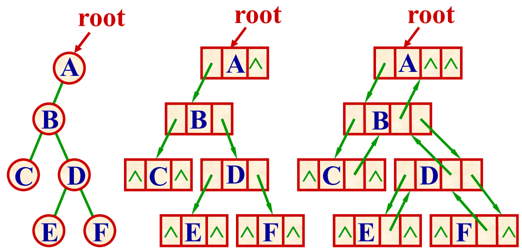
- 使用二叉链表，找子女的时间复杂度为$O(1)$，找双亲的时间复杂度为$O(\log_2 i)$～$O(i)$，其中，i 是该结点编号。

二叉树的三叉链表表示



- 使用三叉链表，找子女、双亲的时间都是$O(1)$。

二叉树链表表示的示例



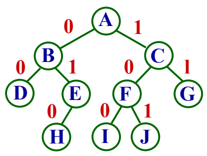
二叉树　二叉链表　三叉链表

二叉树的结构定义

```
typedef char TElemType;          //树结点数据类型
typedef struct node {            //树结点定义
   TElemType data;               //结点数据
   struct node *lchild, *rchild; //左、右子女指针
} BiTNode, *BinTree;             //树定义
```

- 例：按层输入二叉树的各个分支，建立这棵二叉树。各分支用 {p, c, d} 描述，其中，p 是双亲的数据，c 是子女数据，d 是方向，d = ‘0’是左分支，d = ‘1’是右分支。

- 图中的 9 个分支分别为：
  {'A', 'B', '0'}, {'A', 'C', '1'},
  {'B','D', '0'}, {'B', 'E', '1'},
  {'C', 'F', '0'}, {'C', 'G', '1'},
  {'E', 'H', '0'}, {'F', 'I', '0'},
  {'F', 'J', '1'}



- 因为按层输入，第一个分支的双亲 p 一定是根。
- 插入各分支时，也要在已建部分查找双亲。为此建立一个辅助数组，存放已建结点的结点地址。
- 查到双亲后，建立子女结点和链接。因为是按层输入，双亲一定已经建立。


```
#include "BinTree.h"
#define queSize 64
void createBinTree ( BinTree& BT, char a[ ][3], int n ) {
//从字符数组 a 中输入 n 条分支，建立二叉树 BT 的
//二叉链表，要求各分支按层输入
   int i, j, k = 0;  BiTNode *s, *p;
   BT = ( BiTNode *) malloc ( sizeof (BiTNode) );
   BT->data = a[0][0];                  //建根结点
   BT->lchild = NULL;  BT->rchild = NULL;
   BinTree Q[queSize];  Q[0] = BT;     //Q存已建结点
   for ( i = 0; i < n; i++ )            //逐个分支处理
     for ( j = 0; j <= k; j++ )         //查找双亲结点
```



```
       if ( a[i][0] == Q[j]->data ) {      //查到双亲
          p = ( BiTNode *) malloc ( sizeof (BiTNode));
          p->data = a[i][1];               //建子女结点
          p->lchild = p->rchild= NULL;
          s = Q[j];  Q[++k] = p;           //子女保存
          if ( a[i][2] == '0' ) s->lchild = p;
          else s->rchild = p;              //链接
          break;
       }
}
```

- 二叉树的输出在“二叉树遍历的应用”给出。

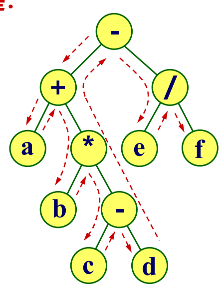
二叉树遍历

- 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。设
  - 访问根结点记作 V，
  - 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　VLR　镜像　VRL
  - 中序　LVR　镜像　RVL
  - 后序　LRV　镜像　RLV
- 遍历就是把树结点按某种次序排列成线性序列。

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 (V);
  - 中序遍历右子树 (R)。



遍历结果

a + b * c - d - e / f

二叉树递归的中序遍历算法

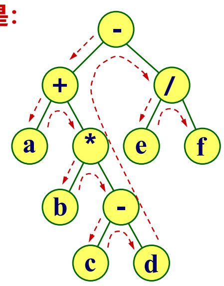
```
void InOrder ( BiTNode *T ) {
    if ( T != NULL ) {
        InOrder ( T->lchild );
        visit ( T->data );
        InOrder ( T->rchild );
    }
}
```

- visit()是输出数据值的操作，在实际使用时可用相应的操作来替换。

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 (V);
  - 先序遍历左子树 (L);
  - 先序遍历右子树 (R)。



遍历结果

- + a * b - c d / e f

二叉树递归的先序遍历算法

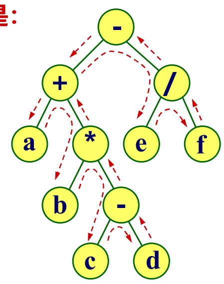
```
void PreOrder (BiTNode *T) {
    if (T != NULL) {
        visit (T->data);
        PreOrder (T->lchild);
        PreOrder (T->rchild);
    }
}
```

- 与中序遍历算法相比，visit() 操作放在两个子树递归先序遍历的最前面。

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 (V)。



遍历结果

a b c d - * + e f / -

二叉树递归的后序遍历算法

```
void PostOrder (BiTNode *T) {
    if (T != NULL) {
        PostOrder (T->lchild);
        PostOrder (T->rchild);
        visit (T->data);
    }
}
```

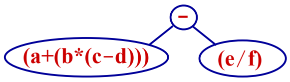
- 与中序遍历算法相比，visit() 操作放在两个子树递归先序遍历的最后面。

从 a+b*(c−d)−e/f 生成表达式树

(1) 先根据运算符的优先级对表达式加括号

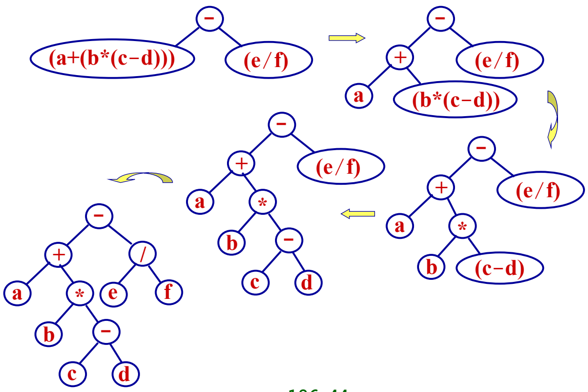
((a+(b*(c−d)))−(e/f))

(2) 脱一层括号，(a+(b*(c−d))) − (e/f)，取两个括号中间的“−”为根，将表达式分为两部分，左子树是(a+(b*(c−d)))，右子树为(e/f)：



(3) 对左子树递归执行步骤(2)；//若树空则不再递归

(4) 对右子树递归执行步骤(2)；//若树空则不再递归



应用二叉树遍历的事例

交换二叉树所有分支的左右子女

```
int Exchange ( BiTNode *t ) {
   if ( t != NULL && ( t->lchild != NULL | |
         t->rchild != NULL ) ) {
      BiTNode *s = t->lchild;  t->lchild = t->rchild;
      t->rchild = s;               //交换*t的两个子女
      Exchange ( t->lchild );      //交换左子树的子女
      Exchange ( t->rchild );      //交换右子树的子女
   }
}
```

- 算法采用先序遍历实现。采用中序遍历行不行?

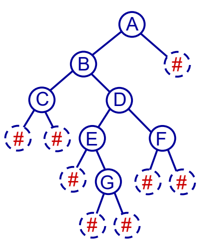
求二叉树高度的递归算法

```
int  Height ( BiTNode *t ) {
   if ( t == NULL ) return 0;
   else {
      int m = Height ( t->lchild );
      int n = Height ( t->rchild );
      return (m > n) ? m+1 : n+1;
   }
}
```

- 空树的高度为 0；非空树情形，先计算根结点左右子树的高度，取大者再加一得到二叉树高度。

利用二叉树先序遍历序列建立二叉树

- 对于如图所示的二叉树，约定以输入序列中不可能出现的值作为空结点的值以结束递归，例如用“#”或用“0”表示字符序列或正整数序列空结点。
- 按照先序遍历所得到的先序序列为A B C # # D E # G # # F # # #。



- 算法的基本思想是：每读入一个值，就为它建立一个结点，作为子树的根结点，其地址通过函数的引用型参数 T 直接链接到作为实际参数的指针中。然后，分别对根的左、右子树递归地建立子树，直到读入“#”建立空子树递归结束。
- 若读入值为“;”停止建立二叉树。算法描述如下

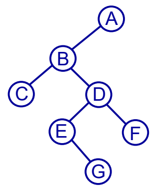
```
#include "BinTree.h"
void createBinTree_Pre ( BiTNode *& T,
      TElemType pre[ ], int& n ) {
//以递归方式建立二叉树 T，pre[ ]是输入序列，
//以‘;’结束，空结点的标识为‘#’。引用参数 n 初
//始调用前赋值0，退出后 n 是输入统计
```

```
TElemType ch = pre[n++];  //读入结点数据
if ( ch == ';' ) return;        //处理结束，返回
if ( ch != '#' ) {              //建立非空子树
    T = (BiTNode *) malloc (sizeof (BiTNode));
    T->data = ch;               //建立根结点
    createBinTree_Pre ( T->lchild, pre, n );
                                //递归建立左子树
    createBinTree_Pre ( T->rchild, pre, n );
                                //递归建立右子树
}
else T = NULL;                  //否则建立空子树
};
```

126-49

用括号方式输出二叉树

- 右图所示的二叉树的括号表示：
  A (B (C, D (E (, G), F)),)
- 若二叉树为空，输出空格“ ”；
- 若二叉树非空，则先输出根结点的数据，再判断根是否叶？
  - ➢若是叶结点，则空操作；
  - ➢若不是叶结点，则 ① 输出左括号“(”，② 递归输出左子树，③ 输出逗号“,”，④ 递归输出右子树，⑤ 输出右括号“)”。



126-50

```
void printBinTree ( BiTNode *t ) {
  if ( t != NULL ) {
    printf ( "%c", t->data );      //输出根
    if ( t->lchild != NULL | | t->rchild != NULL ) {
      printf ( "(" );              //输出左括号
      PrintBinTree ( t->lchild );  //递归输出左子树
      printf ( "," );              //输出逗号
      PrintBinTree ( t->rchild );  //递归输出右子树
      printf ( ")" );              //输出右括号
    }
  }
  else printf ( " " );             //空树，输出空格
}
```

126-51

二叉树遍历的非递归算法

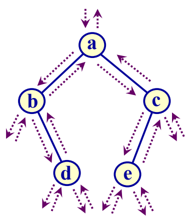
- 无论先序、中序、后序遍历，都可以归属为一种单一的设计模式，叫做欧拉巡回遍历，基于它可以很容易地写出通用的非递归算法。
- 二叉树 T 的欧拉巡回遍历方式，就是环绕着 T 走动。其作法是，从根结点开始向它的左子女结点移动，此时把二叉树 T 的各边当作“墙”，在走动的过程中，永远让墙保持在左边。
- 在欧拉巡回中 T 的每个结点 v 都会遇到三次：
  ① 在对v的左子树做欧拉巡回遍历之前
  ② 在对v的左子树做欧拉巡回遍历之后

126-52

③ 在对v的右子树做欧拉巡回遍历之后

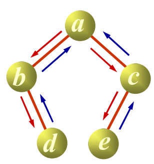
- 以v为根的子树的欧拉巡回遍历算法描述如下：

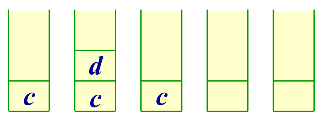
```
void eulerTour (T, v) {
   visit (v);      //“先序”
   if ( v->lchild != NULL )
      eulerTour (T, v->lchild);
   visit (v);      //“中序”
   if ( v->rchild != NULL )
      eulerTour (T, v->rchild);
   visit (v);      //“后序”
}
```



126-53

利用栈的先序遍历的非递归算法





访问	访问	退栈	退栈	访问
a	*b*	*d*	*c*	*e*
进栈	进栈	访问	访问	左进
c	*d*	*d*	*c*	空
左进	左进	左进	左进	栈空
b	空	空	*e*	结束

54

```
#define stackSize 30
void preOrder_iter ( BinTree BT ) {
//利用栈实现二叉树BT的先序遍历
   BiTNode *S[stackSize];  int top = -1;  //建栈
   BiTNode *p = BT;
   do {
      while ( p != NULL ) {
         printf ( "%c ", p->data );          //访问结点
         if ( p->rchild != NULL ) S[++top] = p->rchild;
         p = p->lchild;                      //"左下"
      }
      if ( top != -1 ) p = S[top--];         //"左上右下"
```

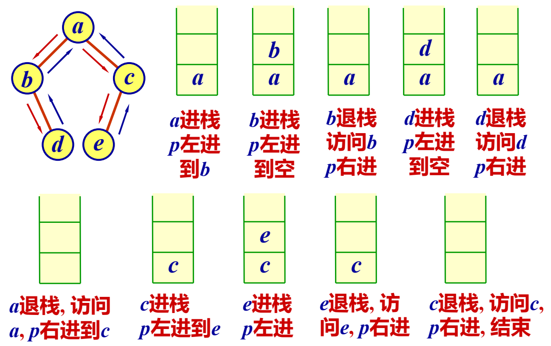
126-55

```
   } while ( p != NULL || top != -1 );
}
```

- 算法的时间复杂度取决于程序中的while循环。由于二叉树的每个结点都要访问，若设二叉树的结点数为 n，则算法的时间复杂度为$O(n)$。
- 算法的空间复杂度取决于二叉树的高度，最好情形为$O(\log_2 n)$，最坏情形为$O(n)$。

126-56

利用栈的中序遍历的非递归算法



126-57

```
#define stackSize 30
void inOrder_iter ( BinTree BT ) {
//利用栈实现二叉树BT的中序遍历
   BiTNode *S[stackSize];  int top = -1;
   BiTNode *p = BT;    //p是遍历指针，从根开始
   do {
      while ( p != NULL )    //遍历指针进到左子女
         { S[++top] = p;  p = p->lchild; }
      if ( top != -1 ) {     //栈不空时退栈
         p = S[top--];       //退栈, 访问
         printf ( "%c ", p->data );
         p = p->rchild;      //遍历指针进到右子女结点
```

126-58

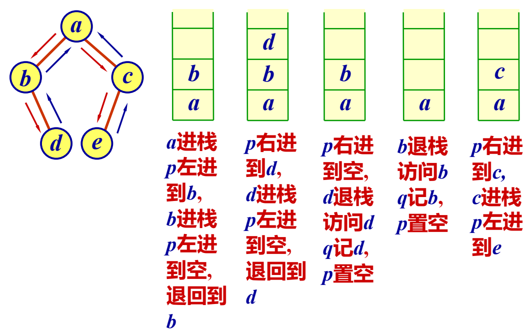
```
      }
   } while ( p != NULL || top != -1 );
}
```

① 中序遍历时，先把根结点放一放，遍历左子树，这导致内层有一个循环，边向左子女方向走，边把结点记忆到栈中，直到左子女为空。

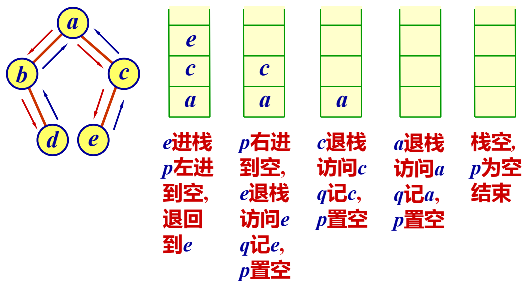
② 接下来，访问栈顶结点，然后进到该结点的右子女，对此结点的右子树再执行①，因此外层有一个大循环，此循环的结束条件是栈为空，同时遍历指针 p 也为空。如访问根后栈为空，但右子树（根为p）不为空，循环还要继续。

126-59

利用栈的后序遍历的非递归算法



126-60



- 若*p的右子树为空或*p的右子女已访问过，则退栈，q记忆退出结点，访问它，置p为空；


```
#define stackSize 30
void postOrder_iter ( BinTree BT ) {
//利用栈实现二叉树BT的后序遍历
   BiTNode *S[stackSize];  int top = -1;
   BiTNode *p = BT, *pre = NULL;      //pre是p前趋
   do {
     while ( p != NULL )                //左子树进栈
       { S[++top] = p;  p = p->lchild; }
     if ( top != -1 ) {
       p = S[top];               //用p记忆栈顶元素
       if ( p->rchild != NULL && p->rchild != pre )
          p = p->rchild;       //p有右子女且未访问过
       else {
```



```
          printf ( "%c ", p->data );      //访问
          pre = p;  p = NULL;   //记忆刚访问过的结点
          top--;            //转遍历右子树或访问根结点
       }
     }
   } while ( p != NULL || top != -1 );
}
```

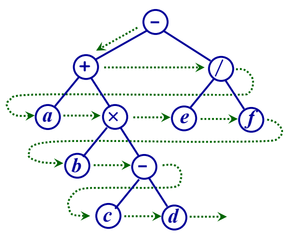
- 传统的后序遍历非递归算法需要对每一个进栈的元素设置一个标志 tag，进栈时让 tag = L，在退栈时如果退栈元素的 tag = L，则表示从左子树退出，还需要访问右子树；如果 tag = R， 则表示从右子树退出，可以访问根。本算法不需 tag 了。

二叉树的层次序遍历

- 层次序遍历从二叉树的根结点开始，自上向下，自左向右分层依次访问树中的各个结点。如图所示，按照层次序遍历，结点的访问次序为

  $- + / a \times e f b - c d$

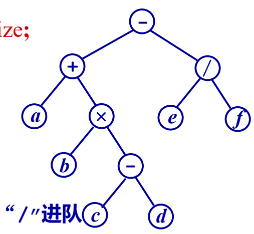
- 按层次顺序访问二叉树的处理需要利用一个队列。在访问二叉树的某一层结点时，把下一层结点指针预先记忆在队列中，利用队列安排逐层访问的次序。




```
#define queueSize 30
void levelOrder ( BinTree BT ) {
//利用队列实现二叉树BT的层次序遍历。
   BiTNode *Q[queueSize];  int rear = 0, front = 0;
   BiTNode *p = BT;  Q[rear++] = p;     //根进队
   while ( rear != front ) {           //队列不空时
     p = Q[front];  front = (front+1) % queueSize;
     printf ( "%c ", p->data );       //退队访问
     if ( p->lchild != NULL ) {       //若有左子女，进队
       Q[rear] = p->lchild;
       rear = (rear+1) % queueSize;
     }
```



```
     if ( p->rchild != NULL ) {    //若有右子女，进队
       Q[rear] = p->rchild;
       rear = (rear+1) % queueSize;
     }
   }
}
```



Q [– | | |] 根 “–”进队

Q [+ | / | |] “–”出队， “+” “/”进队

Q [/ | a | × |] “+”出队， “a” “×”进队

Q [a | × | e | f] “/”出队， “e” “f”进队

Q | × | e | f | | “a”出队，无进队

Q | e | f | b | – | “×”出队，“b” “–”进队

Q | f | b | – | | “e”出队，无进队

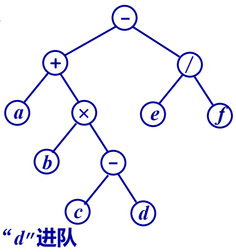
Q | b | – | | | “f”出队，无进队

Q | – | | | | “b”出队，无进队

Q | c | d | | | “–”出队，“c” “d”进队

Q | d | | | | “c”出队，无进队

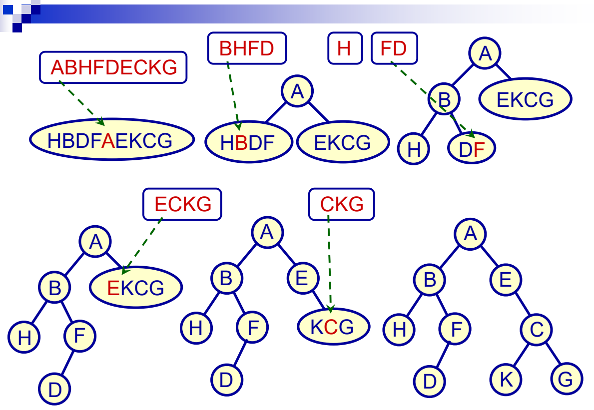
Q | | | | | “d”出队，无进队



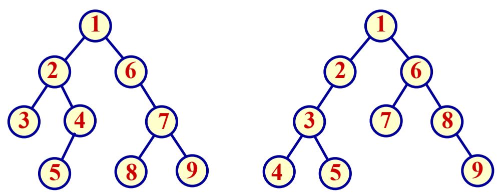
二叉树的计数

- 由二叉树的先序序列和中序序列可唯一地确定一棵二叉树。
- 复习二叉树先序序列隐含的性质：先序序列第一个元素一定是二叉树的根。其后紧跟的是根的左子女（若根的左子树非空），或者是根的右子女（若根的左子树为空）。
- 复习二叉树中序序列隐含的性质：二叉树的根可把其中序序列分为两个子序列，左子序列是根的左子树的中序序列，右子序列是根的右子树的中序序列。

- 顺便说明二叉树后序序列隐含的性质：与先序序列正好相反，后序序列最后一个元素一定是二叉树的根，其紧前一个元素是根的右子女（若根的右子树非空），或者是根的左子女（若 的右子树为空）。子树则类推。
- 二叉树的中序序列与先序序列搭配起来，就可以恢复该二叉树。例如，一棵二叉树的先序序列为ABHFDECKG，中序序列为HBDFAEKCG。
- 先序序列第一个 ‘A’一定是根，它把中序序列一分为二： “HBDF”和 “EKCG”，这就得到二叉树的左子树和右子树的中序序列，再看先序序列，…

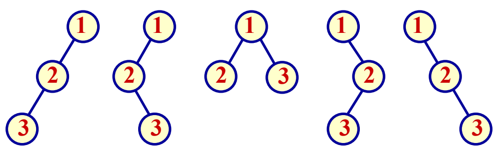


- 如果先序序列固定不变，给出不同的中序序列，可得到不同的二叉树。



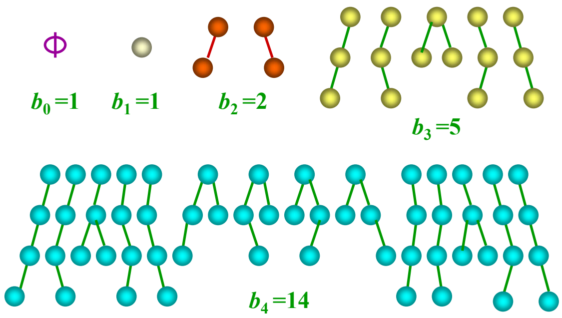
- 问题是：固定先序排列，选择所有可能的中序排列，可以构造出多少种不同的二叉树？

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



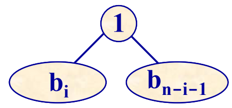
- 那么，如何推广到一般情形呢？首先，只有一个结点的不同二叉树只有一个；有 2 个结点的不同二叉树只有 2 种，其他情况呢？

有0个, 1个, 2个, 3个结点的不同二叉树如下



计算具有 n 个结点的不同二叉树的棵数

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



- *Catalan*函数

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

- 例

$$b_3 = \frac{1}{3+1} C_6^3 = \frac{1}{4}\frac{6\cdot 5\cdot 4}{3\cdot 2\cdot 1} = 5$$

$$b_4 = \frac{1}{4+1} C_8^4 = \frac{1}{5}\frac{8\cdot 7\cdot 6\cdot 5}{4\cdot 3\cdot 2\cdot 1} = 14$$

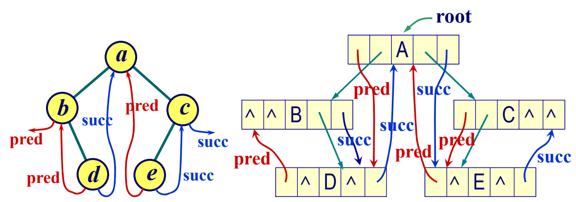
线索二叉树 (Threaded Binary Tree)

- 又称为穿线树。
- 通过二叉树遍历，可将二叉树中所有结点的数据排列在一个线性序列中，可以找到某数据在这种排列下它的前趋和后继。
- 希望不必每次都通过遍历找出这样的线性序列。只要事先做预处理，将某种遍历顺序下的前趋、后继关系记在树的存储结构中，以后就可以高效地找出某结点的前趋、后继。
- 为此，在二叉树存储结点中增加线索信息。

线索 (Thread)

pred	lchild	data	rchild	succ

- 增加前趋Pred指针和后继Succ指针的二叉树



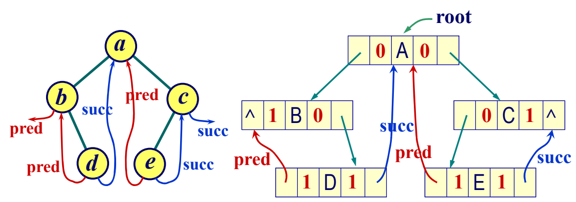
- 这种设计的缺点是每个结点增加两个指针，当结点数很大时存储消耗较大。
- 改造树结点，将 pred 指针和 succ 指针压缩到 lchild 和 rchild 的空闲指针中，并增设两个标志 ltag 和 rtag，指明指针是指示子女还是前趋 / 后继。后者称为线索。

lchild	ltag	data	rtag	rchild

- ltag (或rtag) = 0，表示相应指针指示左子女（或右子女结点）；
- ltag (或rtag) = 1, 表示相应指针为前趋（或后继）线索。

中序线索二叉树及其链表表示

lchild	ltag	data	rtag	rchild



线索二叉树的结构定义

```
typedef  int TTElemType;

typedef struct node {
    int ltag, rtag;
    struct node *lchild, *rchild;
    TTElemType data;
} ThreadNode, *ThreadTree;
```

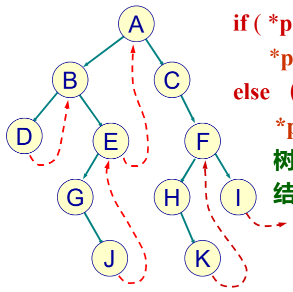
- 注意，线索二叉树在树结点中增加了ltag和rtag，改变了二叉树的结构。

在中序线索二叉树中寻找第一个结点

```
ThreadNode * First ( ThreadNode *t ) {
//函数返回以 *t 为根的线索二叉树中的中序序列下
//的第一个结点
   ThreadNode *p = t;
   while ( p->ltag == 0 ) p = p->lchild;
    return p;                  //最左下的结点
}
```

- 中序线索二叉树中从根 *t 开始，沿左子女链走到底，即 *t 为根二叉树的中序第一个结点。

寻找指定结点 *p 在中序下的后继



if (*p的右指针是后继线索)
  *p 的后继为 p->rchild。
else （右指针是右子女指针）
  *p 的后继为结点 *p 右子树 q 中的中序下的第一个结点。

在中序线索二叉树中
寻找指定结点的中序后继

```
ThreadNode * Next ( ThreadNode * p ) {
//函数返回在线索二叉树中指定结点 *p 在中序下的
//后继结点
  if ( p->rtag == 0 ) return First (p->rchild);
      //p->rtag == 0, 后继为右子树中序第一个结点
  else  return p->rchild;
      //p->rtag == 1, 直接返回后继线索
}
```



```
void Inorder ( ThreadNode * t ) {
//以 t 为根的线索二叉树的中序遍历
   ThreadNode * p;
   for ( p = First (t);  p != NULL;  p = Next (p) )
       printf ( "%d", p->data );
   printf ( "\n" );
}
```

- 这个遍历算法没有递归，也没有使用栈。
- 对于一棵普通的二叉树，可以通过中序遍历实现它的全线索化。在算法中，增加了 pre 指针，跟踪遍历指针 t，以便它们链接。

通过中序遍历建立中序线索二叉树

```
void inThread (ThreadNode *t,  ThreadNode *& pre) {
//预设了一个 pre 指针，指示 t 的中序前趋，在主
//程序中预置为NULL
  if ( t != NULL ) {
     InThread ( t->lchild,  pre );   //对 *t 左子树线索化
     if ( t->lchild == NULL )     //对 *t 的前趋线索化
        { t->lchild = pre;  t->ltag = 1; }
     if ( pre != NULL && pre->rchild == NULL )
        { pre->rchild = t;  pre->rtag = 1; }
                          //对前趋 *pre 的后继线索化
```

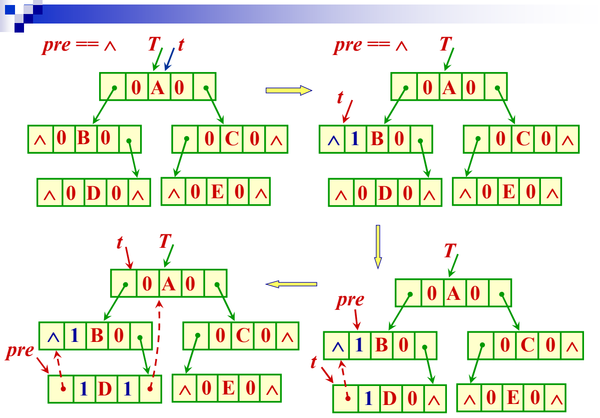
```
      pre = t;              //前趋跟上,当前指针向前遍历
      InThreaded ( p->rchild, pre ); //对 *t 右子树线索化
   }
}

void createInThread ( ThreadTree T ) {
    ThreadNode *pre = NULL;    //前趋指针
    if ( T != NULL ) {             //树非空, 线索化
        InThread ( T, pre );        //中序遍历线索二叉树
        pre->rchild = NULL;
        pre->rtag = 1;         //后处理, 中序最后一个结点
    }
}
```

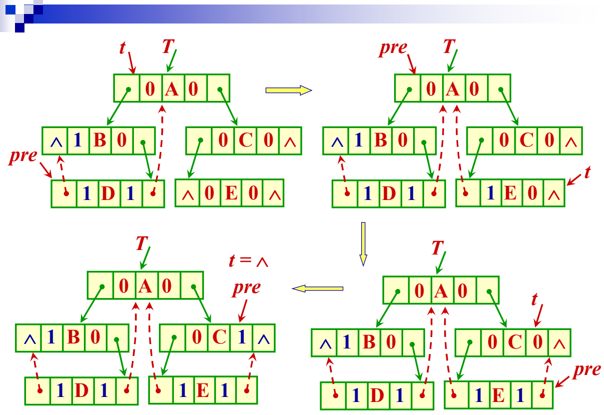
126-85

- 使用函数 InThread，可把以 *t 为根的子树一次中序线索化，但中序下最后一个结点的后继线索没有加上，指针 pre 在退出时正在指示这一结点。
- 主程序createInThread，初始时令前趋指针 pre = NULL，然后调用函数 InThread，实现对二叉树的中序线索化。在从 InThread 返回后，做一个后处理，对中序下的最后一个结点加后继线索。
- 从函数createInThread返回后，就完成了一棵中序线索二叉树的全线索化。
- 下图是对一棵二叉树做中序全线索化的示例。

126-86

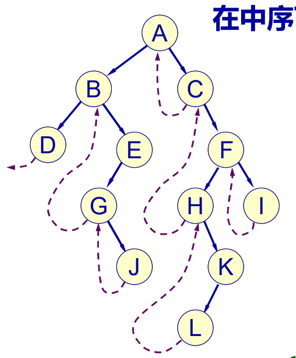


126-87



126-88

在中序线索二叉树中寻找指定结点 *p 在中序下的前趋



if (p 有前趋线索)
　　则前趋为 p->lchild
else　//p 有左子女
　　则前趋为结点 p 的左子树中中序下的最后一个结点

126-89

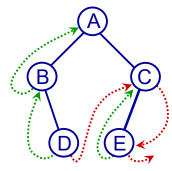
```
ThreadNode * Last ( ThreadNode *t ) {
//函数返回线索二叉树中序序列下的最后一个结点
   for ( ThreadNode *p = t; p->rtag == 0;
        p = p->rchild );    //最右下结点
   return p;
}

ThreadNode * Prior ( ThreadNode * p ) {
//函数返回线索二叉树中结点 *p 在中序下的前趋
   if ( p->ltag == 1 ) return p->lchild;
   else return Last (p->lchild );
}
```

126-90

先序线索二叉树

- 先序线索二叉树是通过先序遍历建立起先序线索而得到的线索二叉树。
- 树中的线索记录了在先序次序下结点之间的前趋、后继关系。
- 在先序线索二叉树上寻找指定结点先序下的后继的思路：
  - 如果结点有左子女，左子女是先序下的后继；如果没有左子女，则右子女（或右线索）是先序下的后继。

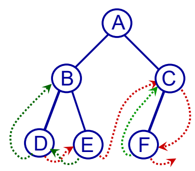


先序序列 ABDCE


```
ThreadNode *Next ( ThreadNode * p ) {
//函数返回在先序线索二叉树中*p的后继
    if ( p->ltag == 0 ) return p->lchild;
    else return p->rchild;
}
```

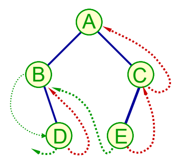
- 在先序线索二叉树中寻找指定结点*p 的前趋比较复杂，其求解思路为：
  - 如果结点 *p 有前趋线索，则可直接找到先序下的前趋结点，否则
  - 首先寻找结点*p 的双亲*q：

- 如果*q 不存在，则 *p 为根，无前趋；
- 如果*p 是*q 的左子女，则*q 是*p 的前趋；
- 如果*p 是*q 的右子女，则前趋是*q 左子树中先序下的最后一个结点。



先序序列 ABDECF

后序线索二叉树



后序序列

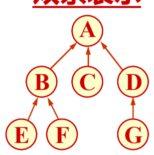
D B E C A

- 后序线索二叉树与先序线索二叉树是对称的，只要把左、右互换即可。故不再详细讨论。
- 左图是一棵后序线索二叉树的示例，后序次序下的第一个结点是 D，它有后继线索，则其后继 B 可通过后继线索得到。B 没有后继线索，则其后继需要通过它的双亲 A 找其右子树中后序次序下的第一个结点 E。

树与森林

- 树的存储表示

双亲表示



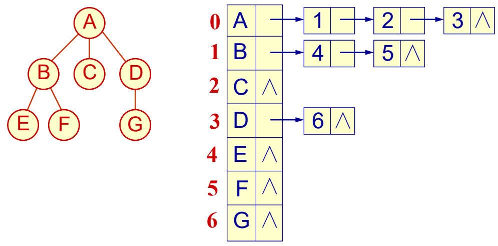
	0	1	2	3	4	5	6
data	A	B	C	D	E	F	G
parent	-1	0	0	0	1	1	3

- 为了操作实现的方便，有时会规定结点的存放顺序。例如，可以按树的先序次序存放树中的各个结点，或按树的层次次序安排所有结点。


```
#define maxSize 50              //树中最多结点个数
#define stackSize 20

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;
```

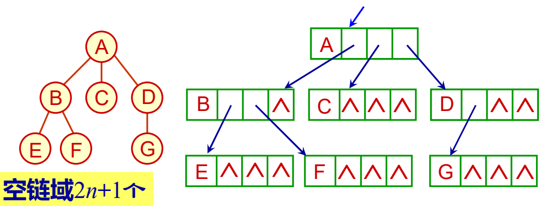
子女链表表示



- 无序树情形链表中各结点顺序任意，有序树必须自左向右链接各个子女结点。

子女指针表示

- 一个合理的想法是在结点中存放指向每一个子女结点的指针。但由于各个结点的子女数不同，每个结点设置数目不等的指针，将很难管理。
- 为此，设置等长的结点，每个结点包含的指针个数相等，等于树的度（degree）。
- 这保证结点有足够的指针指向它的所有子女结点。但可能产生很多空闲指针，造成存储浪费。



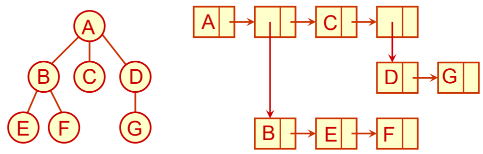
空链域$2n+1$个

等数量的链域

data	$child_1$	$child_2$	$child_3$	·····	$child_d$

广义表表示

- 下图给出的树的广义表描述 A(B(E, F), C, D(G))。
- 子表的头结点是子树的根，其后链接它的所有子女。如果子女是叶结点（单元素或原子），结点中直接赋值，否则结点中是子表的头指针。



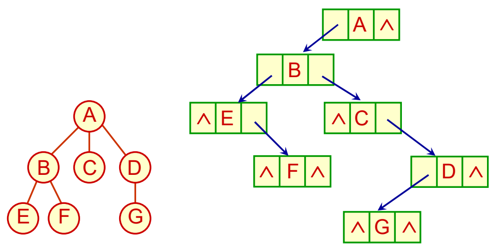
子女-兄弟链表表示

- 也称为树的二叉树表示。结点构造为：

data	lchild	rsibling

- lchild 指向该结点的第一个子女结点。无序树的情形，可任意指定一个结点为第一个子女。
- rsibling 指向该结点的下一个兄弟。任一结点在存储时总是有顺序的。
- 若想找某结点的所有子女，可先找lchild，再反复用rsibling 沿链扫描。

- 树的子女-兄弟链表表示是双链表结构，与二叉树结点的定义类似，但语义是不同的。操作的含义自然也不同。



子女-兄弟链表表示的树的结构定义

```
typedef char TElemType;
typedef struct node {
    TElemType data;
    struct node *lchild, *rsibling;
} CSNode, *CSTree;
```

- 树结构是递归的，可用递归函数实现相应操作。
- 在用子女-兄弟链表表示的树中，寻找指定结点 *p 的第一个子女、下一个兄弟的操作很简单，时间复杂度为 $O(1)$，但寻找双亲操作则比较复杂。


```
CSNode * firstChild ( CSNode *p ) {
//在树中找结点 *p 的第一个子女
   if ( p != NULL ) return p->lchild;
   else return NULL;
}
CSNode * nextSibling ( CSNode *p ) {
//在树中找结点 *p 的下一个兄弟
   if ( p != NULL ) return  p->rsibling;
   else return NULL;
}
```

- 如果一个结点的lchild为空，它一定是树的叶结点


```
CSNode *Parent ( CSNode *T, CSNode *p ) {
//在以 T 为根的树中找结点 *p 的双亲
   if ( T == NULL | | p == NULL )  return NULL;
   CSNode *q = T->lchild, *s;   //*q为根的第一个子女
   while ( q != NULL ) {
      if ( q == p ) return T;          //找到双亲
      if ( ( s = Parent (q, p) ) != NULL) return s;
         //递归到以 *q 为根的子树中查找 *p 的双亲
      q = q->rsibling;                 //查下一个兄弟
   }
   return NULL;                        //未找到双亲
}
```


建立树的子女-兄弟链表的算法

```
#include "CSTree.h"
#define size 30
void createCSTree_Degree ( CSNode *& t,
      TElemType e[ ], int degree[ ], int n ) {
//根据树结点的层次序列e[n]和各结点的度degree[n]
//构造树的子女—兄弟链表，参数n是树结点个数
   if ( n == 0 ) { t = NULL;  return; };
   int d, i, k = 0;
   CSTree *Q = (CSTree *) malloc ( size*sizeof
        (CSTree) );            //队列 Q 存放结点地址
   for ( i = 0; i < n; i++ ) {  //建所有树结点
```



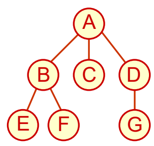
```
      Q[i] = (CSNode *) malloc ( sizeof (CSNode) );
      Q[i]->data = e[i];
      Q[i]->lchild = Q[i]->rsibling = NULL;
   }
   for ( i = 0; i < n; i++ ) {         //链接所有结点
      d = degree[i];
      if ( d > 0 ) {
         k++;  Q[i]->lchild = Q[k];
         while ( --d > 0 )         //所有兄弟挨在一起
            { Q[k]->rsibling = Q[k+1];  k++; }
      }
   }
```



```
      t = Q[0];  free (Q);
   }
```

- 如图所示的树，其层次序列和各结点的度如下

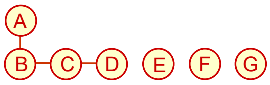
level	A	B	C	D	E	F	G
degree	3	2	0	1	0	0	0



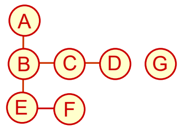
- 创建树的所有结点

A B C D E F G

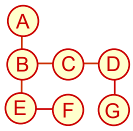
- 对根 A，链接它的所有子女B、C、D 到子女-兄弟链表



- 对 B，链接它的所有子女 E、F 到子女-兄弟链表

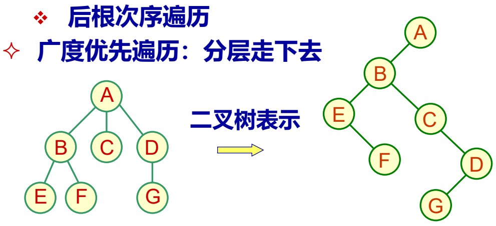


- 对 C，它没有子女，链表不变；
- 对 D，链接子女 G 到子女-兄弟链表
- 对 E，它没有子女，链表不变；
- 对 F，它没有子女，链表不变；
- 对 G，它没有子女，链表不变.



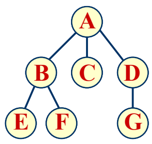
树的遍历

- 树的遍历有两类：
  - ✧ 深度优先遍历：沿某条通路走下去
    - ❖ 先根次序遍历
    - ❖ 后根次序遍历
  - ✧ 广度优先遍历：分层走下去



树的先根次序遍历

- 当树非空时
  - ▫ 访问根结点
  - ▫ 依次先根遍历根的各棵子树



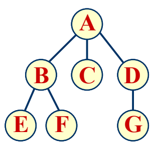
- 树先根遍历 ABEFCDG
  对应二叉树先序遍历 ABEFCDG
- 树的先根遍历结果与其对应二叉树表示的先序遍历结果相同
- 树的先根遍历可以借助对应二叉树的先序遍历算法实现

树的先根次序遍历的递归算法

```
void PreOrder ( CSNode *t ) {
//以指针 t 为根, 先根次序遍历
   if ( t != NULL ) {
        visit ( t->data );                  //访问根结点
        CSNode *p = t->lchild;              //第一棵子树
        while ( p != NULL ) {
            PreOrder (p);             //递归先根遍历子树
            p = p->rsibling;
        }
    }
}
```


树的后根次序遍历

- 当树非空时
  - ▫ 依次后根遍历根的各棵子树
  - ▫ 访问根结点



- 树后根遍历 EFBCGDA
  对应二叉树中序遍历 EFBCGDA
- 树的后根遍历结果与其对应二叉树表示的中序遍历结果相同
- 树的后根遍历可以借助对应二叉树的中序遍历算法实现

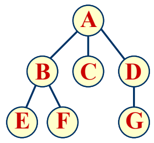
树的后根次序遍历的递归算法

```
void PostOrder ( CSNode *t ) {
//以指针 t 为根, 按后根次序遍历树
   if ( t != NULL ) {
       CSNode *p = t->lchild;
       while ( p != NULL ) {
           PostOrder ( p );
           p = p->rsibling;
       }
       visit ( t->data );        //最后访问根结点
    }
}
```

广度优先(层次次序)遍历

- 按广度优先次序遍历树的结果

  ABCDEFG



- 广度优先遍历算法

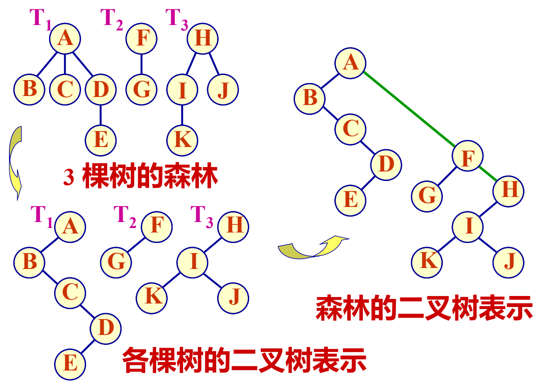
```
void LevelOrder ( CSTree T ) {
//分层遍历树，算法用到一个队列
    Queue Q;   InitQueue(Q);
    CSNode *p;
    if ( T != NULL ) {          //树不空
        EnQueue(Q, T);          //根结点进队列
```



```
        while ( ! QueueEmpty(Q) ) {
            DeQueue(Q, p);   visit(p->data);
                //队列中取一个并访问之
            p = p->lchild;
                //待访问结点的子女结点进队列
            while ( p != NULL ) {
                EnQueue ( Q, p );
                p = p->rsibling;
            }
        }
    }
}
```


森林与二叉树的转换

- 将一般树化为二叉树表示就是用树的子女-兄弟表示来存储树的结构。
- 森林与二叉树表示的转换可以借助树的二叉树表示来实现。



森林转化成二叉树的规则

❶ 若 F 为空，即 n = 0，则对应的二叉树 B 为空树。

❷ 若 F 不空，则

- ➢ 二叉树 B 的根是 F 第一棵树 T_1 的根；
- ➢ 其左子树为 B ($T_{11}, T_{12}, \ldots, T_{1m}$)，其中，$T_{11}, T_{12}, \ldots, T_{1m}$ 是 T_1 的根的子树；
- ➢ 其右子树为 B ($T_2, T_3, \ldots, T_n$)，其中，$T_2, T_3, \ldots, T_n$ 是除 T_1 外其它树构成的森林。

二叉树转换为森林的规则

❶ 如果 B 为空，则对应的森林 F 也为空。

❷ 如果 B 非空，则

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

森林的遍历

- 森林的遍历也分为深度优先遍历（包括先根次序遍历和中根次序遍历）和广度优先遍历。

深度优先遍历

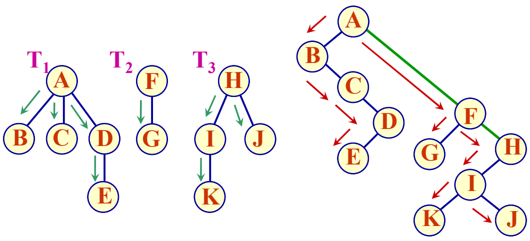
- 给定森林 F，若 $F = \varnothing$，则遍历结束。否则
- 若$F = \{\{T_1 = \{ r_1, T_{11}, \ldots, T_{1k} \}, T_2, \ldots, T_m\}$，则可以导出先根遍历、中根遍历两种方法。其中，$r_1$ 是第一棵树的根结点，$\{T_{11}, \ldots, T_{1k}\}$是第一棵树的子树森林，$\{T_2, \ldots, T_m\}$是除去第一棵树之后剩余的树构成的森林。

126-121

森林的先根次序遍历

- 若森林F = Ø，返回；否则
  ① 访问森林的根（也是第一棵树的根）r_1；
  ② 先根遍历森林第一棵树的根的子树森林 $\{T_{11}, \ldots, T_{1k}\}$；
  ③ 先根遍历森林中除第一棵树外其他树组成的森林$\{T_2, \ldots, T_m\}$。
- 注意，② 是一个循环，对于每一个T_{1i}，执行树的先根次序遍历；③ 是一个递归过程，重新执行 T_2 为第一棵树的森林的先根次序遍历。

126-122



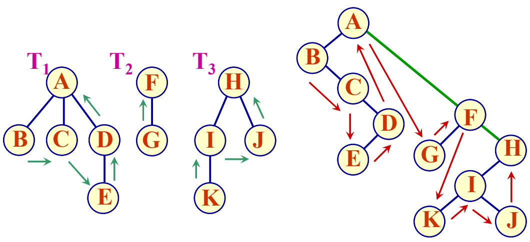
- 森林的先根次序遍历的结果序列
  ABCDE FG HIKJ
- 这相当于对应二叉树的先序遍历结果。

126-123

森林的中根次序遍历

- 若森林 $F = \varnothing$，返回；否则
  ① 中根遍历森林 F 第一棵树的根结点的子树森林$\{T_{11}, \ldots, T_{1k}\}$；
  ② 访问森林的根结点 r_1；
  ③ 中根遍历森林中除第一棵树外其他树组成的森林$\{T_2, \ldots, T_m\}$。
- 注意，与先根次序遍历相比，访问根结点的时机不同。所以在③的情形，对以T_2为第一棵树的森林遍历时，重复执行①②③的操作。

126-124

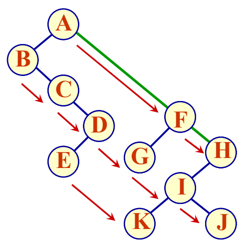


- 森林的中根次序遍历的结果序列
  BCEDA GF KIJH
- 这相当于对应二叉树中序遍历的结果。

126-125

广度优先遍历（层次序遍历）

- 若森林 F 为空，返回；否则
  ① 依次遍历各棵树的根结点；
  ② 依次遍历各棵树根结点的所有子女；
  ③ 依次遍历这些子女结点的子女结点；
  ④ ……



AFH BCDGIJ EK

126-126