第六章 树和二叉树

线性结构

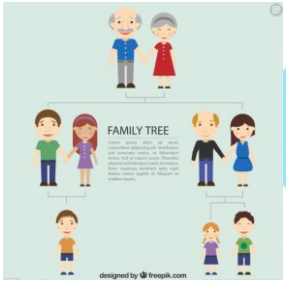
1．数据元素的集合中必存在唯一的一个"第一个元素"；
2．数据元素的集合中必存在唯一的一个"最后的元素"；
3．除最后元素之外，其它数据元素均有唯一的"后继"；
4．除第一元素之外，其它数据元素均有唯一的"前驱"。

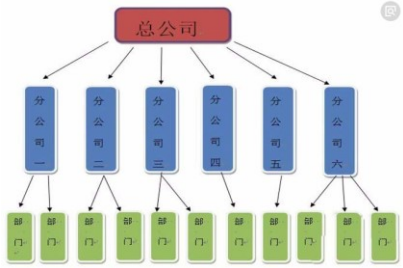
非线性结构

- 所谓**非线性结构**是指，在该结构中至少存在一个数据元素，有两个或两个以上的直接前驱（或直接后继）元素。相对应于线性结构，非线性结构的逻辑特征是一个结点元素可能对应多个直接前驱和多个后继。
- 树型结构和图型就是其中十分重要的非线性结构，可以用来描述客观世界中广泛存在的层次结构和网状结构的关系，如家族谱、城市交通等。

树型结构是一类非常重要的非线性结构。树型结构是以分支关系定义的层次结构。

树在计算机领域中也有着广泛的应用，例如在编译程序中，用树来表示源程序的语法结构；在数据库系统中，可用树来组织信息；在分析算法的行为时，可用树来描述其执行过程等等。





主 要 内 容

树的概念
二叉树的概念
二叉树的遍历
二叉树的应用
线索二叉树
树与二叉树
哈夫曼树及哈夫曼编码
二叉查找树
平衡二叉查找树
B-树

6.1 树的定义

6.1.1 定义

树(Tree)是n($n\geqq0$)个结点的有限集合T，若n=0时称为空树，否则：

(1) 有且只有一个特殊的称为树的根(Root)结点；

(2) 若n>1时，其余的结点被分为m(m>0)个互不相交的非空子集T_1，T_2，T_3…T_m，其中每个子集本身又是一棵树，称其为根的子树(Subtree)。

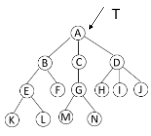
没有空子树

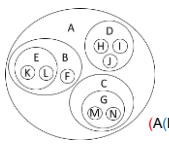
树的表示形式

(1) 倒悬树。是最常用的表示形式，如图(a)。

(2) 嵌套集合。是一些集合的集体，对于任何两个集合，或者不相交，或者一个集合包含另一个集合。图(b)是树的嵌套集合形式。

(3) 广义表形式。图(c)是树的广义表形式。

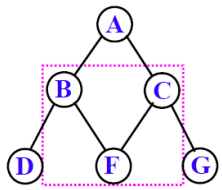


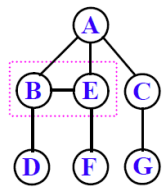


(A(B(E(K,L),F),C(G(M,N)),D(H,I,J)))

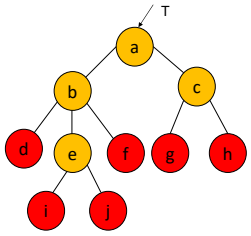
(a) 倒悬树　　(b) 嵌套集合形式　　(c) 广义表

非树的结构





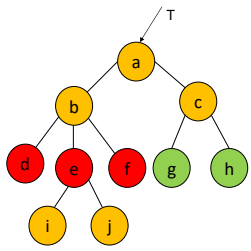
结点的度： 结点的子树个数



结点a的度为2
结点b的度为3
结点d的度为0

树的度： 树中结点度的最大值
分支结点 ： 度不为0的结点
叶结点： 度为0的结点
子女结点： 某结点子树的根结点
双亲结点： 某个结点是其子树之根的双亲

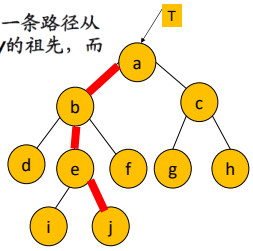
兄弟结点： 具有同一双亲的所有结点



结点d,结点e和
结点f互为兄弟

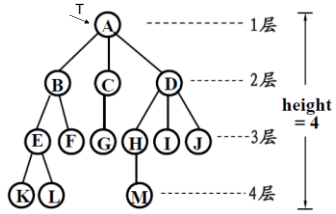
路(径)和路(径) 长度：如果树的结点序列 $n_1, n_2, \ldots, n_k$有如下关系：结点n_i是n_{i+1}的双亲（$1<=i<k$），则把$n_1, n_2, \ldots, n_k$称为一条由n_1至n_k的路径；路径上经过的边的个数称为路径长度。

祖先、子孙：在树中，如果有一条路径从结点x到结点y，那么x就称为y的祖先，而y称为x的子孙。

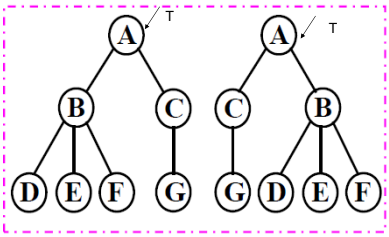


结点的层数：根结点的层数为1；对其余任何结点，若某结点在第k层，则其孩子结点在第k+1层。

树的深度：树中所有结点的最大层数，也称高度。



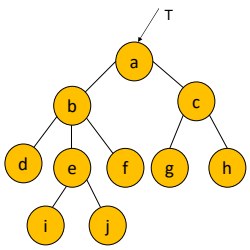
- 有序树　　子树的次序不能互换
- 无序树　　子树的次序可以互换
- 森林　　互不相交的树的集合



6. 1. 2 树的基本操作

1. 求指定结点所在树的根结点
2. 求指定结点的双亲结点
3. 求指定结点的某一孩子结点
4. 求指定结点的最右边兄弟结点
5. 将一棵树插入到另一树的指定结点下作为它的子树
6. 删除指定结点的某一子树
7. 树的遍历

6.1.3 树的存储设计



需要考虑的因素

数据元素的值

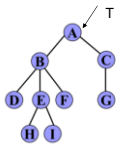
父子关系

兄弟关系

子孙关系

1.双亲表示法

根据树的定义，树中的每个结点都有唯一的一个双亲结点，因此，可用一组连续的存储空间(一维数组)存储树中的各个结点，数组中的一个元素表示树中的一个结点，数组元素为结构体类型，其中包括结点本身的信息以及结点的双亲结点在数组中的序号，树的这种存储方法称为双亲表示法。

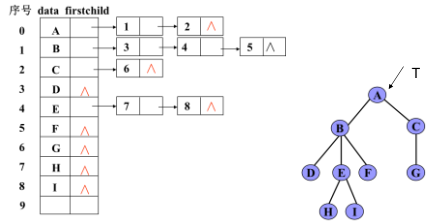


序号	data	parent
0	A	-1
1	B	0
2	C	0
3	D	1
4	E	1
5	F	1
6	G	2
7	H	4
8	I	4

这种表示法在求某结点的孩子结点的操作时，则需要查询整个数组。

2. 孩子表示法

孩子表示法主要描述的是结点与孩子的关系。由于每个结点的孩子个数不定，所以利用链式存储结构更加适宜。



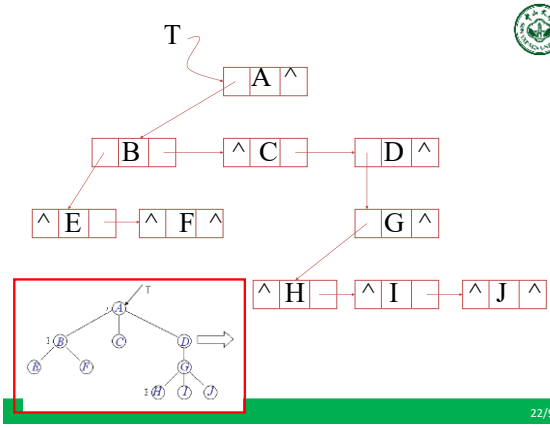
在孩子表示法中查找双亲比较困难，查找孩子却十分方便，故适用于对孩子操作多的应用。

3. 孩子兄弟表示法

孩子兄弟表示法也是一种链式存储结构。它通过描述每个结点的一个孩子和兄弟信息来反映结点之间的层次关系，其结点结构为：

firstchild	item	nextsibling

其中，firstchild为指向该结点第一个孩子的指针，nextsibling为指向该结点的下一个兄弟，item是数据元素内容。



6.2 二叉树 (Binary Trees)

6.2.1 定义

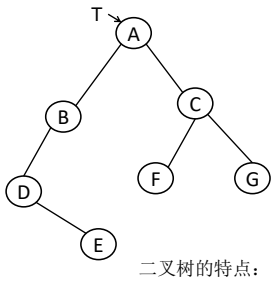
二叉树一个是n（$n\geq 0$）个结点的有限集合，该集合或者为空（称为空二叉树）；或者是由一个根结点和两棵互不相交的、分别称为左子树和右子树的二叉树组成。

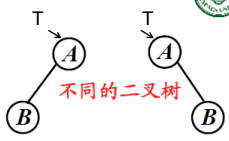
结构特点：

✓ 每个结点最多只有两棵子树，即结点的度不大于**2**。

✓ 子树有左右之别，子树的次序(位置)不能颠倒。

✓ 即使某结点只有一棵子树，也有左右之分。



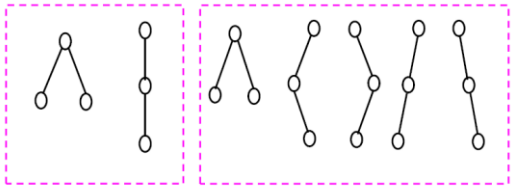


二叉树的特点：

左（右）子树可以是空二叉树。

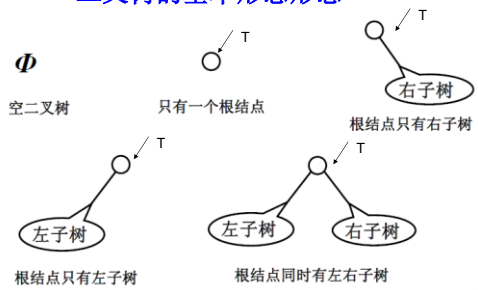
二叉树中结点的度是只含有的非空子树的数目。

具有3个结点的树和二叉树的不同构的形态：



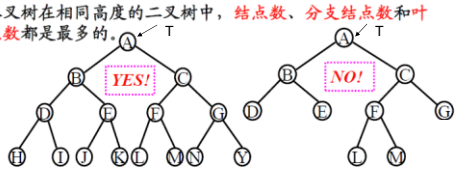
树的不同构形态　　二叉树的不同构形态

• 二叉树的基本形态形态



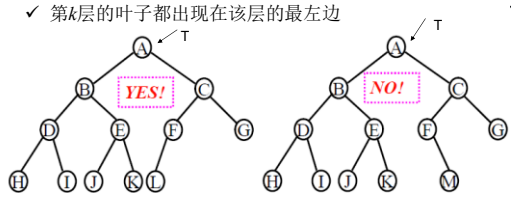
特殊的二叉树----满二叉树

- 定义：高度为K且有2^K-1个结点的二叉树称为满二叉树。
- 结构特点：
  - 分支结点都有两棵子树
  - 叶子结点都在最后一层
- 满二叉树在相同高度的二叉树中，结点数、分支结点数和叶结点数都是最多的。



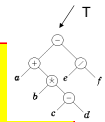
特殊的二叉树----完全二叉树

- 定义：满足下列性质的二叉树（假设树的高度为k）称为完全二叉树。
  - ✓ 所有叶子结点都出现在第k层或k-1层
  - ✓ 第k层的叶子都出现在该层的最左边



6.2.2 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：$i = 1$ 时，有$2^{i-1} = 2^0 = 1$，成立

假定 $i = k$ 时性质成立；

当 $i = k+1$ 时，第$k+1$的结点至多是第k层结点的两倍，即总的结点个数至多为$2 \times 2^{k-1} = 2^k$

故命题成立。

性质2 高度为k的二叉树最多有 2^k-1个结点。($k \geq 1$)

证明：仅当每一层都含有最大结点数时，二叉树的结点数最多，利用性质1可得二叉树的结点数至多为：

$2^0 + 2^1 + 2^2 + 2^3 + \ldots + 2^{k-1} = 2^k - 1$

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

$$n_0 = n_2 + 1$$

证明：

1、结点总数为度为0的结点加上度为1的结点再加上度为2的结点：

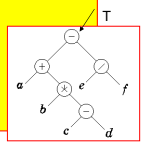
$n = n_0 + n_1 + n_2$

2、另一方面，二叉树中度为1的结点有一个孩子，度为2的结点有二个孩子，根结点不是任何结点的孩子，因此，结点总数为：

$n = n_1 + 2n_2 + 1$

3、两式相减，得到：

$n_0 = n_2 + 1$



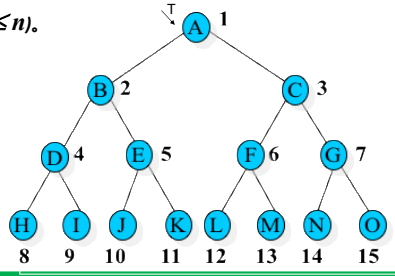
性质4 具有n个结点的完全二叉树的高度为$\lfloor \log_2 n \rfloor + 1$

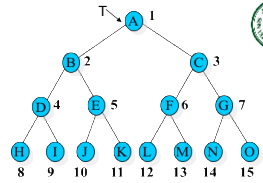
证明：设完全二叉树的高度为h，则有

$2^{h-1} - 1 < n \le 2^h - 1$　　$2^{h-1} <= n < 2^h$

取对数 $h - 1 < \log_2(n) < h$

性质5 如果将一棵有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1, 2, …, n-1,n，然后按此结点编号将树中各结点顺序地存放于一个一维数组中，并简称编号为i的结点为结点i ($1 \le i \le n$)。





则有以下关系：

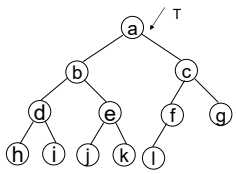
- 若i == 1, 则 i 无双亲
  若i > 1, 则 i 的双亲为$\lfloor i/2 \rfloor$
- 若2*i <= n, 则 i 的左子女为2*i；否则，i无左子女，必定是叶结点，二叉树中i>$\lfloor n/2 \rfloor$的结点必定是叶结点
  若2*i+1 <= n, 则 i 的右子女为2*i+1，否则，i无右子女
- 若 i 为奇数, 且i不为1，则其左兄弟为i-1，否则无左兄弟；
  若 i 为偶数, 且小于 n，则其右兄弟为i+1，否则无右兄弟
- i 所在层次为$\lfloor \log_2(i) \rfloor + 1$

6.2.3 二叉树的存储设计

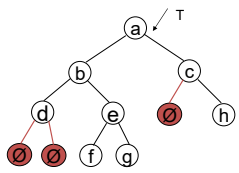
二叉树也可以采用两种存储方式：连续存储结构和链式存储结构。

1、连续设计存储

- 类似于树的顺序存储表示：父亲数组表示。
- 完全二叉树，可以直接用一维数组进行二叉树的存储。，对于一般的二叉树，将其每个结点与完全二叉树上的结点相对照，存储在一维数组中。



(a) 完全二叉树



(b) 非完全二叉树

1	2	3	4	5	6	7	8	9	10	11	12
a	b	c	d	e	f	g	h	i	j	k	l

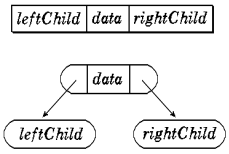
(c) 完全二叉树的顺序存储形式

1	2	3	4	5	6	7	8	9	10	11
a	b	c	d	e	Ø	h	Ø	Ø	f	g

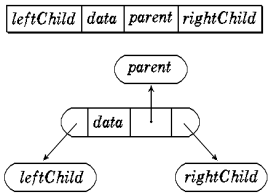
(d) 非完全二叉树的顺序存储形式

2、链接存储设计

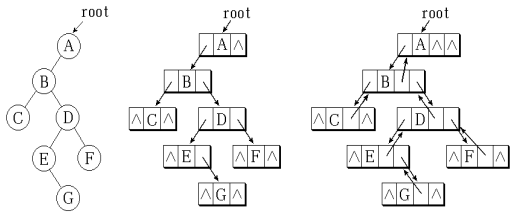
设计不同的结点结构可构成不同的链式存储结构。



（a）二叉链表



（b）三叉链表



（a）二叉树　（b）二叉链表　（c）三叉链表


```
typedef int datatype;
typedef struct node
{ datatype data;
  struct node *Lchild,*Rchild;
} bitree;
```

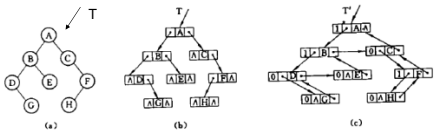

3. 循环链方式的二叉树存储设计

二叉链表的优点是: 结构简单、直观, 基本操作易于实现;
缺点是:

（1）空间利用率低。 含n 个结点的二叉树共有2n 个指针字段, 其中的n + 1 个字段的值为空;

（2）寻找给定结点的父/ 兄结点, 遍历二叉树等基本操作的时、空开销大。

循环链的基本思想是 : 利用结点的指针字段把二叉树中的每组父子结点链成一个有向环 , 以保证迅速地找到结点的父结点和左 、 右子结点。



左_右链方式的存储结点均由四个字段组成：tag(取0或1)，child(指向子结点的指针)，right (右指针字段)和info(信息字段），则该结构的定义如下：

设T是一棵二叉树(采用左_右链方式表示),

T'是T采用本文方法存储表示的二叉树。指针P指向二叉树中的结点。

若二叉树T为空,则令T'为空，否则定义T'如下：

若结点p是T的根结点，则在T'中令p->right =null

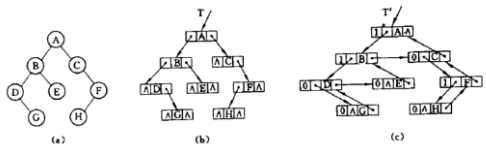
若结点p是T的结点，则在T'中令p->child =null

若结点p有且仅有一个子结点q，则在T'中令p->child =q, q->right =p,

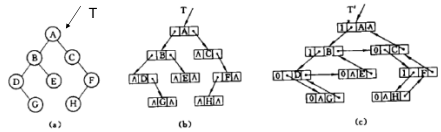
若结点p即有左子结点q，又有右子结点r，则在T'中令

p->child =q, q->right =r， r->right =p;

若结点p有左子结点q，则在T'中令 p->tag =1, 否则p->tag =0。



T'的结点与 T 的结点一一对应; T'本身 是一个强连通的有向图。T'中任一结点的指针经由其子结点或父结点移动至多三次就可回到出发点。



容易验证T'具有如下重要的基本性质：

（1）p是二叉树的根结点，当且仅当p->right =null

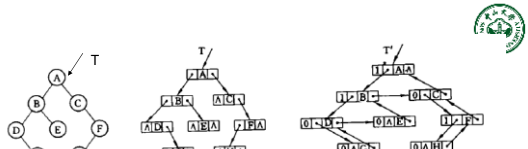
（2）p是二叉树的叶结点，当且仅当p->child =null

（3）p在是二叉树有左子结点，当且仅当p->tag =1,进一步p的左子结点是p->child

（4）p在是二叉树有右子结点，当且仅当(p->tag =1,且p->child->right≠p)或(p->tag =0,且p->child->≠null)

进一步若p->tag =1 则p的右子结点是p->child->right，否则是p->child

（5）p有父结点，当且仅当p->right ≠ null



利用性质（1）或（2）可判断结点是否是根或叶结点；利用性质（3）或（4）可判断结点是否有左或右孩子并可找出该孩子。利用性质（5）可判断结点是否有父结点,可找出其父结点并能断定。该结点是其父结点的左或右孩子。

6.3 二叉树的遍历

二叉树的遍历是根据某种规则，按照一定的顺序访问树中的每一个结点，使得每个结点被访问且仅被访问一次。这个过程称为二叉树的遍历。

所谓访问是指对结点做某种处理。如：输出信息、修改结点的值等。

通过一次完整的遍历，可使二叉树中结点信息由非线性排列变为某种意义上的线性序列。也就是说，遍历操作使非线性结构线性化。

6.3 二叉树的遍历

二叉树是一种非线性结构，每个结点都可能有左、右两棵子树，因此，需要寻找一种规律，使二叉树上的结点能排列在一个线性队列上，从而便于遍历。

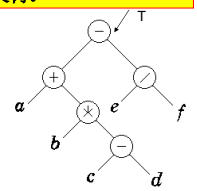
二叉树的基本组成：根结点、左子树、右子树。若能依次遍历这三部分，就是遍历了二叉树。

按照子树和根的访问顺序，可以分为：

先序遍历

中序遍历

后序遍历



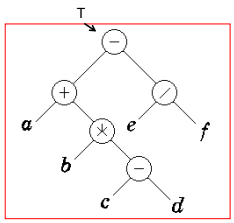
6.3.1 先序遍历

先序遍历二叉树递归算法：

- **若二叉树为空，遍历结束；**
- **否则**
  - **访问根结点 (V)；**
  - **先序遍历根结点的左子树 (L)；**
  - **先序遍历根结点的右子树 (R)。**

遍历结果

-+a*b-cd/ef




```
void PreorderTraverse(BTNode *T)
  { if (T!=NULL)
      { visit(T->data) ;     /* 访问根结点 */
        PreorderTraverse(T->Lchild) ;
        PreorderTraverse(T->Rchild) ;
      }
  }
```

说明：visit()函数是访问结点的数据域，其要求视具体问题而定。树采用二叉链表的存储结构，用指针变量T来指向。

非递归算法

设T是指向二叉树根结点的指针变量，非递归算法是：

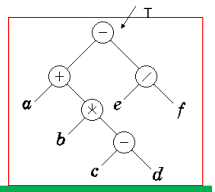
若二叉树为空，则返回；否则，令p=T；

⑴ 访问p所指向的结点；

⑵ q=p->Rchild，若q不为空，则q进栈；

⑶ p=p->Lchild，若p不为空，转(1)，否则转(4)；

⑷ 退栈到p，转(1)，直到栈空为止。




```
#define  MAX_NODE  50
void  PreorderTraverse( BTNode  *T)
  {  BTNode  *Stack[MAX_NODE] ,*p=T, *q ;
     int  top=0 ;
     if  (T==NULL)  printf(" Binary Tree is Empty!\n") ;
     else {  do
               {  visit( p-> data ) ;   q=p->Rchild ;
                  if  ( q!=NULL )  stack[++top]=q ;
                  p=p->Lchild ;
                  if (p==NULL) { p=stack[top] ;  top-- ; }
               }
             while (p!=NULL) ;
          }
  }
```

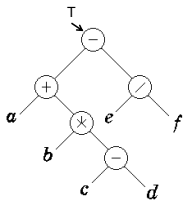

6.3.2 中序遍历

中序遍历二叉树的递归算法：

- 若二叉树为空，则空操作；
- 否则
  - 中序遍历根结点的左子树 (L)；
  - 访问根结点 (V)；
  - 中序遍历根结点的右子树 (R)。

遍历结果

a + b * c - d - e / f



表达式语法树

中序遍历的递归算法

```
void  InorderTraverse(BTNode  *T)
  {  if  (T!=NULL)
        {  InorderTraverse(T->Lchild) ;
           visit(T->data) ;      /*  访问根结点  */
           InorderTraverse(T->Rchild) ;
        }
  }
```


中序遍历的非递归算法

设T是指向二叉树根结点的指针变量，非递归算法是：

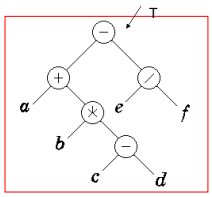
若二叉树为空，则返回；否则，令p=T

⑴ 若p不为空，p进栈，p=p->Lchild；

⑵ 否则(即p为空)，退栈到p，访问p所指向的结点；

⑶ p=p->Rchild，转(1)；

直到栈空为止。

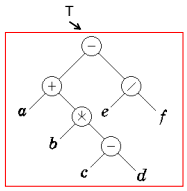



```
#define MAX_NODE  50
void  InorderTraverse( BTNode  *T)
{  BTNode  *Stack[MAX_NODE] ,*p=T ;
   int  top=0 , bool=1 ;
   if  (T==NULL)  printf(" Binary Tree is Empty!\n") ;
  else  { do
            { while (p!=NULL)
                 {  stack[++top]=p ;  p=p->Lchild ;  }
              if  (top==0)  bool=0 ;
              else  {  p=stack[top] ;  top-- ;
                       visit( p->data ) ;  p=p->Rchild ; }
            }  while (bool!=0) ;
        }
}
```

6.3.3 后序遍历

后序遍历二叉树递归算法：

- 若二叉树为空，则空操作；
- 否则
  - 后序遍历根结点的左子树(L)；
  - 后序遍历根结点的右子树(R)；
  - 访问根结点(V)。



遍历结果

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

后序遍历的递归算法

```
void PostorderTraverse(BTNode *T)
  { if (T!=NULL)
      { PostorderTraverse(T->Lchild) ;
        PostorderTraverse(T->Rchild) ;
        visit(T->data) ;     /* 访问根结点 */
      }
  }
```

遍历二叉树的算法中基本操作是访问结点，因此，无论是哪种次序的遍历，对有n个结点的二叉树，其时间复杂度均为$O(n)$。

非递归算法

在后序遍历中，根结点是最后被访问的。因此，在遍历过程中，当搜索指针指向某一根结点时，不能立即访问，而要先遍历其左子树，此时根结点进栈。当其左子树遍历完后再搜索到该根结点时，还是不能访问，还需遍历其右子树。所以，此根结点还需再次进栈，当其右子树遍历完后再退栈到该根结点时，才能被访问。

因此，设立一个状态标志变量tag：

$$tag=\begin{cases}0: & \text{结点暂不能访问}\\ 1: & \text{结点可以被访问}\end{cases}$$

其次，设两个堆栈S_1、S_2，S_1保存结点，S_2保存结点的状态标志变量tag。S_1和S_2共用一个栈顶指针。

设T是指向根结点的指针变量，非递归算法是：

若二叉树为空，则返回；否则，令p=T；

(1) 第一次经过根结点p，不访问：

p进栈S1， tag 赋值0，进栈S2，p=p->Lchild。

(2) 若p不为空，转(1)，否则，取状态标志值tag：

(3) 若tag=0：对栈S1，不访问，不出栈；修改S2栈顶元素值(tag赋值1)，取S1栈顶元素的右子树，即p=S1[top]->Rchild，转(1)；

(4) 若tag=1：S1退栈，访问该结点；

直到栈空为止。

算法实现：

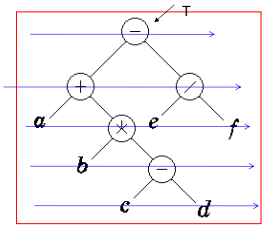
```
#define MAX_NODE 50
void PostorderTraverse( BTNode *T)
  { BTNode *S1[MAX_NODE] ,*p=T ;
    int S2[MAX_NODE] , top=0 , bool=1 ;
    if (T==NULL) printf("Binary Tree is Empty!\n") ;
    else { do
           { while (p!=NULL)
               { S1[++top]=p ; S2[top]=0 ;
                 p=p->Lchild ;
               }
             if (top==0) bool=0 ;
```



```
           else if (S2[top]==0)
                  { p=S1[top]->Rchild ; S2[top]=1 ; }
                else
                  { p=S1[top] ; top-- ;
                    visit( p->data ) ; p=NULL ;
                    /* 使循环继续进行而不至于死循环 */
                  }
           } while (bool!=0) ;
      }
  }
```

6.3.4 层次遍历二叉树

层次遍历二叉树，是从根结点开始遍历，按层次次序“自上而下，从左至右”访问树中的各结点。



遍历结果

- + / a * e f b - c d

为保证是按层次遍历，必须设置一个队列，初始化时为空。

设T是指向根结点的指针变量，层次遍历非递归算法是：

若二叉树为空，则返回；否则，令p=T，p入队；

(1) 队首元素出队到p；

(2)访问p所指向的结点；

(3)将p所指向的结点的左、右子结点依次入队。直到队空为止。


```
#define MAX_NODE  50
void  LevelorderTraverse( BTNode  *T)
  { BTNode  *Queue[MAX_NODE] ,*p=T ;
    int  front=0 , rear=0 ;
    if  (p!=NULL)
      { Queue[++rear]=p;   /*  根结点入队 */
        while (front<rear)
          { p=Queue[++front]; visit( p->data );
            if (p->Lchild!=NULL)
                Queue[++rear]=p;   /*  左结点入队 */
            if (p->Rchild!=NULL)
                Queue[++rear]=p;   /*  左结点入队 */
          }
      }
  }
```


6. 4 二叉树遍历的应用

“遍历”是二叉树最重要的基本操作，是各种其它操作的基础，二叉树的许多其它操作都可以通过遍历来实现。如建立二叉树的存储结构、求二叉树的结点数、求二叉树的深度等。

6.4.1 二叉树的二叉链表创建

(1) 按满二叉树方式建立

按满二叉树的方式对结点进行编号建立链式二叉树。对每个结点，输入*i*、*ch*。

- *i*： 结点编号，按从小到大的顺序输入
- *ch*： 结点内容，假设是字符

在建立过程中借助一个一维数组*S*[*n*]，编号为*i*的结点保存在*S*[*i*]中。

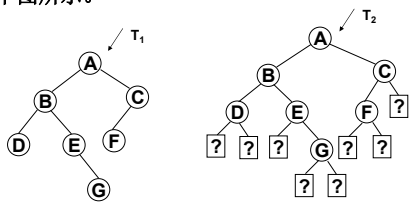

```
typedef struct BTNode
  { char  data ;
    struct BTNode *Lchild , *Rchild ;
  }BTNode ;

BTNode  *Create_BTree(void)
  /*  建立链式二叉树，返回指向根结点的指针变量 */
  { BTNode  *T , *p , *s[MAX_NODE] ;
    char ch ; int i , j ;
    while (1)
      { scanf(“%d”, &i) ;
        if  (i==0)  break ;   /*  以编号0作为输入结束 */
```

```
        else
        {  ch=getchar() ;
           p=(BTNode *)malloc(sizeof(BTNode)) ;
           p->data=ch ;
           p->Lchild=p->Rchild=NULL ; s[i]=p ;
           if (i==1)  T=p ;
           else
              { j=i/2 ;   /*   j是i的双亲结点编号 */
                if (i%2==0)  s[j]->Lchild=p ;
                  else  s[j]->Rchild=p ;
              }
        }
    }
    return(T) ;
}
```


(2) 按先序遍历方式建立

对一棵二叉树进行“扩充”，就可以得到有该二叉树所扩充的二叉树。有两棵二叉树T_1及其扩充的二叉树T_2如下图所示。

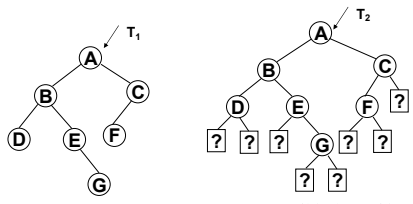


(a) 二叉树T_1　　(b) T_1的扩充二叉树T_2

二叉树T_1及其扩充二叉树T_2

二叉树的扩充方法是：在二叉树中结点的每一个空链域处增加一个扩充的结点(总是叶子结点，用方框“□”表示)。对于二叉树的结点值：

- 是char类型：扩充结点值为“？”；
- 是int类型：扩充结点值为0或-1；



(a) 二叉树T_1　　(b) T_1的扩充二叉树T_2

二叉树T_1及其扩充二叉树T_2

当创建图所示的二叉树时，输入的字符序列应当是：

ABD??E?G??CF???

二叉树的先序创建的递归算法：读入一棵二叉树对应的扩充二叉树的先序遍历的结点值序列。每读入一个结点值就进行分析：

- 若是扩充结点值：令根指针为NULL；
- 若是(正常)结点值：动态地为根指针分配一个结点，将该值赋给根结点，然后递归地创建根的左子树和右子树。


```
BTNode  *Preorder_Create_BTree(BTNode  *T)
   /*  建立链式二叉树，返回指向根结点的指针变量 */
  {  char ch ; ch=getchar() ;
     if  (ch==NULLKY)
        {   T=NULL;  return(T) ;  }
     else
        {  T=(BTNode *)malloc(sizeof(BTNode)) ;
           T->data=ch ;
           Preorder_Create_BTree(T->Lchild) ;
           Preorder_Create_BTree(T->Rchild) ;
           return(T) ;
        }
  }
```

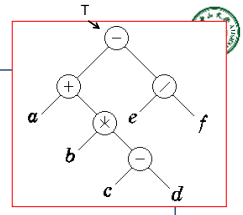
6.4.2 求二叉树的叶子结点数

可以直接利用先序遍历二叉树算法求二叉树的叶子结点数。只要将先序遍历二叉树算法中vist()函数简单地进行修改就可以。

```
void  PreorderTraverse(BTNode  *T)
{ if (T!=NULL)
    { visit(T->data) ;     /* 访问根结点 */
      PreorderTraverse(T->Lchild) ;
      PreorderTraverse(T->Rchild) ;
    }
}
```


6.4.3 求二叉树的高度

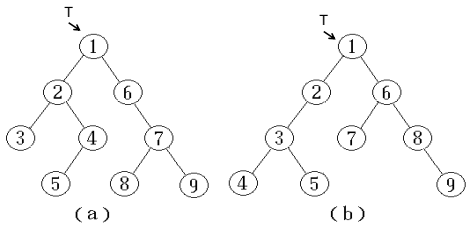
```
void depth(biter *t)
{
  int dep1, dep2;
  if (t = = NULL) return(0);
  else
    {
      dep1 = depth(t->Lchild);
      dep2 = depth(t->Rchild);
      if (dep1 > dep2) return(dep1 + 1);
      else return(dep2 + 1); printf(“)”);
    }
}
```



6.4.4 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。

例:先序序列{ ABHFDECKG }和中序序列{ HBDFAEKCG }

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。



先序序列： 1，2，3，4，5，6，7，8，9
中序序列a： 3，2，5，4，1，6，8，7，9
中序序列b： 4，3，5，2，1，7，6，8，9

6.5 线索二叉树

遍历二叉树是按一定的规则将树中的结点排列成一个线性序列，即是对非线性结构的线性化操作。如何找到遍历过程中动态得到的每个结点的直接前驱和直接后继(第一个和最后一个除外)？如何保存这些信息？

设一棵二叉树有n个结点，则有n-1条边(指针连线)， 而n个结点共有2n个指针域(Lchild和Rchild)，显然有n+1个空闲指针域未用。则可以利用这些空闲的指针域来存放结点的直接前驱和直接后继信息。

对结点的指针域做如下规定：

- ◆ 若结点有左孩子，则Lchild指向其左孩子，否则，指向其直接前驱；
- ◆ 若结点有右孩子，则Rchild指向其右孩子，否则，指向其直接后继；

为避免混淆，对结点结构加以改进，增加两个标志域。

Ltag	Lchild	data	Rchild	Rtag

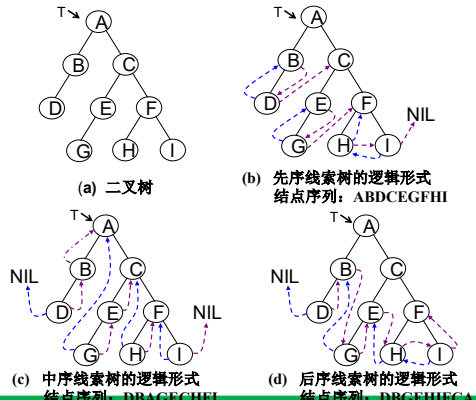
线索二叉树的结点结构

Ltag= { 0：Lchild域指示结点的左孩子
        1：Lchild域指示结点的前驱

Rtag= { 0：Rchild域指示结点的右孩子
        1：Rchild域指示结点的后继

用这种结点结构构成的二叉树的存储结构，叫做线索链表；指向结点前驱和后继的指针叫做**线索**；按照某种次序遍历，加上线索的二叉树称之为**线索二叉树**。

```
typedef struct BiThrNode
  {  ElemType  data;
     struct BiTreeNode *Lchild , *Rchild ;
     int  Ltag , Rtag ;
  }BiThrNode ;
```

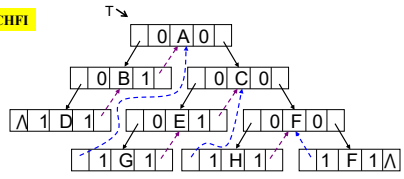
(a) 二叉树

(b) 先序线索树的逻辑形式
结点序列：ABDCEGFHI

(c) 中序线索树的逻辑形式
结点序列：DBAGECHFI

(d) 后序线索树的逻辑形式
结点序列：DBGEHIFCA

DBAGECHFI



(e) 中序线索树的链表结构

说明：画线索二叉树时，实线表示指针，指向其左、右孩子；虚线表示线索，指向其直接前驱或直接后继。

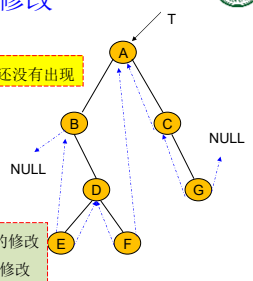
6.5.1 线索化二叉树

二叉树的线索化指的是依照某种遍历次序使二叉树成为线索二叉树的过程。

线索化的过程就是在遍历过程中修改空指针使其指向直接前驱或直接后继的过程。

线索指针的修改

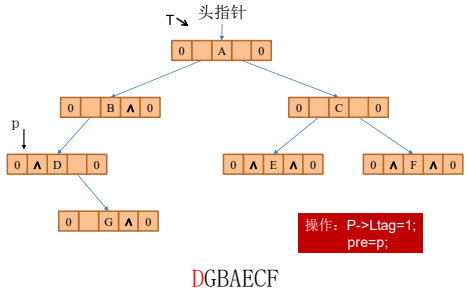
- 对每个正在访问的结点P，其后继线索还没有出现
- 对每个正在访问的结点P进行前驱线索的修改
- 对pre（P的前驱结点）的后继线索进行修改



中序遍历序列：BEDFACG

中序二叉树的线索化

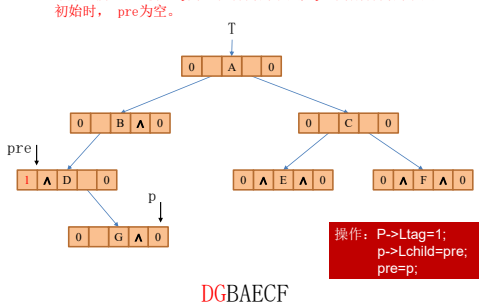
中序遍历二叉链表：p为正在访问的节点，pre为刚访问的节点，初始时， pre为空。



DGBAECF

中序二叉树的线索化

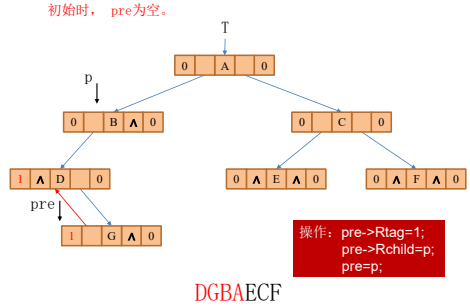
中序遍历二叉链表：p为正在访问的节点，pre为刚访问的节点，初始时， pre为空。



DGBAECF

中序二叉树的线索化

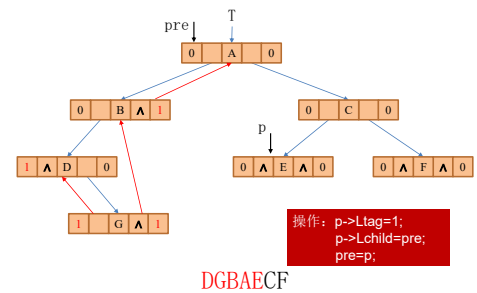
中序遍历二叉链表：p为正在访问的节点，pre为刚访问的节点，初始时， pre为空。



DGBAECF

中序二叉树的线索化

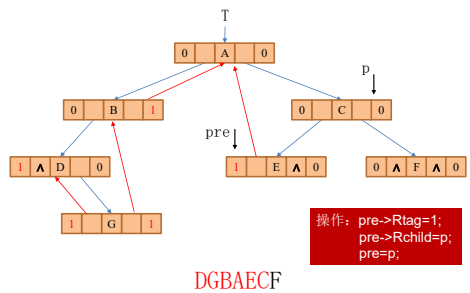
中序遍历二叉链表：p为正在访问的节点，pre为刚访问的节点，初始时， pre为空。



DGBAECF

中序二叉树的线索化

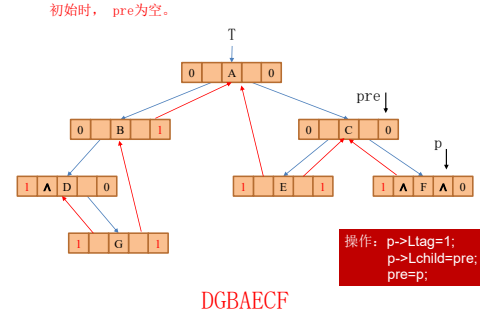
中序遍历二叉链表：p为正在访问的节点，pre为刚访问的节点，初始时， pre为空。



DGBAECF

中序二叉树的线索化

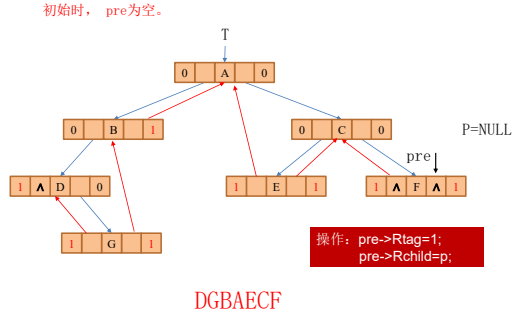
中序遍历二叉链表：p为正在访问的节点，pre为刚访问的节点，初始时， pre为空。



DGBAECF

中序二叉树的线索化

中序遍历二叉链表：p为正在访问的节点，pre为刚访问的节点，初始时， pre为空。

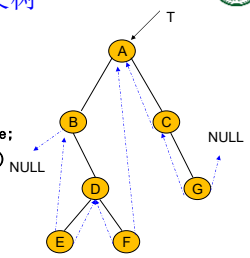


DGBAECF

线索化二叉树

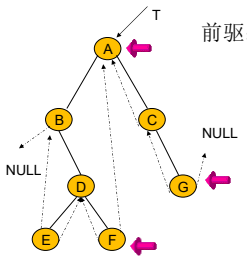
初始时，pre为空

```
INORDERThreading(bitree *t)
{  if (t)
    { INORDERThreading (t->lchild);
      if (t->ltag==1)  t->lchild=pre;
      if (pre!=NULL)&&(pre->rtag==1)
          pre->rchild=t;
      pre=t;
      INORDERThreading (t->rchild);
    }
}
```



中序遍历序列：BEDFACG

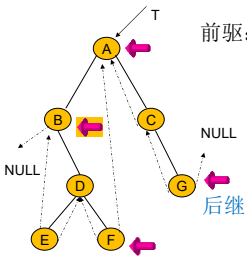
6.5.2 中序线索二叉树上前驱后继的查找



前驱：if (p->ltag==1) pre=p->lchild;
else
pre=p->lchild;
while (pre->rtag==0)
pre=pre->rchild;

中序遍历序列：BEDFACG

中序线索二叉树上前驱后继的查找

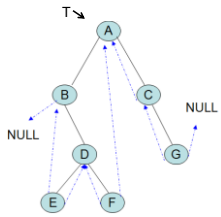


中序遍历序列：BEDFACG

前驱：if (p->ltag==1) pre=p->lchild;
else
pre=p->lchild;
while (pre->rtag==0)
pre=pre->rchild;

后继：if (p->rtag==1) next=p->rchild;
else {
next=p->rchild;
while (next->ltag==0)
next=next->lchild; }

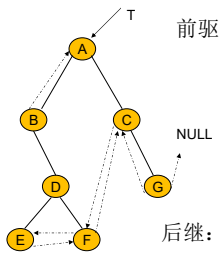
中序线索二叉树的遍历



中序遍历序列：BEDFACG

```
Inorder(bitree T)
{  //查找二叉树的第一个结点
   p=t;
   while (p->ltag==0) p=p->lchild;
   visit(p);
   //查找结点的后继
   while (p->rchild!=NULL)
     {   if (p->rtag==1)
           { p=p->rchild; visit(p); }
         else {  next=p->rchild;
                 while (next->ltag==0)
                  next=next->lchild;
                  visit(next); p=next; }
     }
}
```


6.5.3 前序线索二叉树上前驱后继的查找

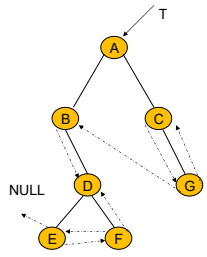


前序遍历序列：ABDEFCG

前驱：if (p->ltag==1) pre=p->lchild;
else
？？？

后继：if (p->rtag==1) next=p->rchild;
else
if (p->ltag==0) next=p->lchild
else next=p->rchild

6.5.4 后序线索二叉树上前驱后继的查找



后序遍历序列：EFDBGCA