第6章 树和二叉树

树型结构是一类非常重要的非线性结构。直观地，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。

本章将详细讨论树和二叉树数据结构，主要介绍树和二叉树的概念、术语，二叉树的遍历算法。树和二叉树的各种存储结构以及建立在各种存储结构上的操作及应用等。

6.1 树的基本概念

6.1.1 树的定义和基本术语

1 树的定义

树(Tree)是n(n≧0)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)个互不相交的子集$T_1, T_2, T_3 \ldots T_m$，其中每个子集本身又是一棵树，称其为根的子树(Subtree)。

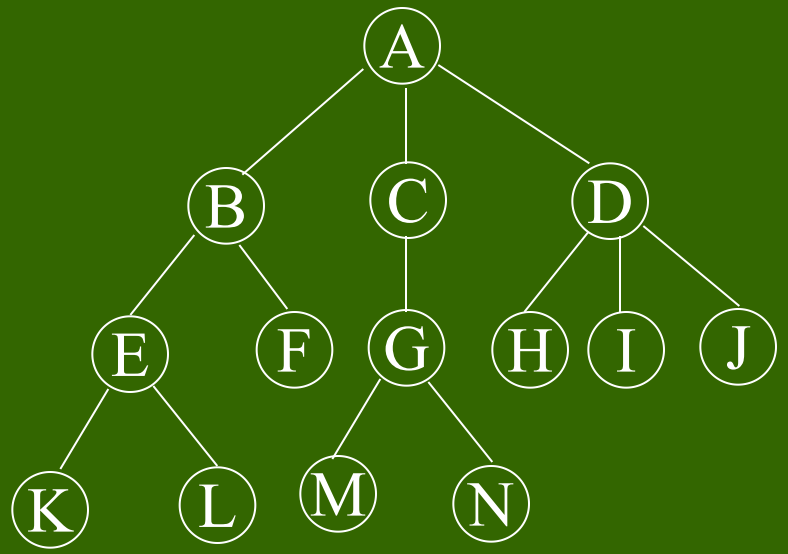
这是树的递归定义，即用树来定义树，而只有一个结点的树必定仅由根组成，如图6-1(a)所示。

2 树的基本术语

(1) **结点(node)**：一个数据元素及其若干指向其子树的分支。

(2) **结点的度(degree)**、**树的度**：结点所拥有的子树的棵数称为**结点的度**。树中结点度的最大值称为**树的度**。

(a) 只有根结点



(b) 一般的树

图6-1 树的示例形式

如图6-1(b)中结点A的度是3，结点B的度是2，结点M的度是0，树的度是3。

(3) 叶子(leaf)结点、非叶子结点：树中度为0的结点称为叶子结点(或终端结点)。相对应地，度不为0的结点称为非叶子结点(或非终端结点或分支结点)。除根结点外，分支结点又称为内部结点。

如图6-1(b)中结点H、I、J、K、L、M、N是叶子结点，而所有其它结点都是分支结点。

(4) 孩子结点、双亲结点、兄弟结点

一个结点的子树的根称为该结点的孩子结点(child)或子结点；相应地，该结点是其孩子结点的双亲结点(parent)或父结点。

如图6-1(b)中结点B 、C、D是结点A的子结点，而结点A是结点B 、C、D的父结点；类似地结点E 、F是结点B的子结点，结点B是结点E 、F的父结点。

同一双亲结点的所有子结点互称为兄弟结点。

如图6-1(b)中结点B 、C、D是兄弟结点；结点E 、F是兄弟结点。

(5) 层次、堂兄弟结点

规定树中根结点的层次为1，其余结点的层次等于其双亲结点的层次加1。

若某结点在第l(l≧1)层，则其子结点在第l+1层。

双亲结点在同一层上的所有结点互称为堂兄弟结点。如图6-1(b)中结点E、F、G、H、I、J。

(6) 结点的层次路径、祖先、子孙

从根结点开始，到达某结点p所经过的所有结点称为结点p的层次路径(有且只有一条)。

结点p的层次路径上的所有结点（p除外）称为p的祖先(ancestor) 。

以某一结点为根的子树中的任意结点称为该结点的子孙结点(descent)。

(7) 树的深度(depth)：树中结点的最大层次值，又称为树的高度，如图6-1(b)中树的高度为4。

(8) 有序树和无序树：对于一棵树，若其中每一个结点的子树（若有）具有一定的次序，则该树称为有序树，否则称为无序树。

(9) 森林(forest)：是m(m≧0)棵互不相交的树的集合。显然，若将一棵树的根结点删除，剩余的子树就构成了森林。

3 树的表示形式

(1) 倒悬树。是最常用的表示形式，如图6-1(b)。

(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6-2(a)是图6-1(b)树的嵌套集合形式。

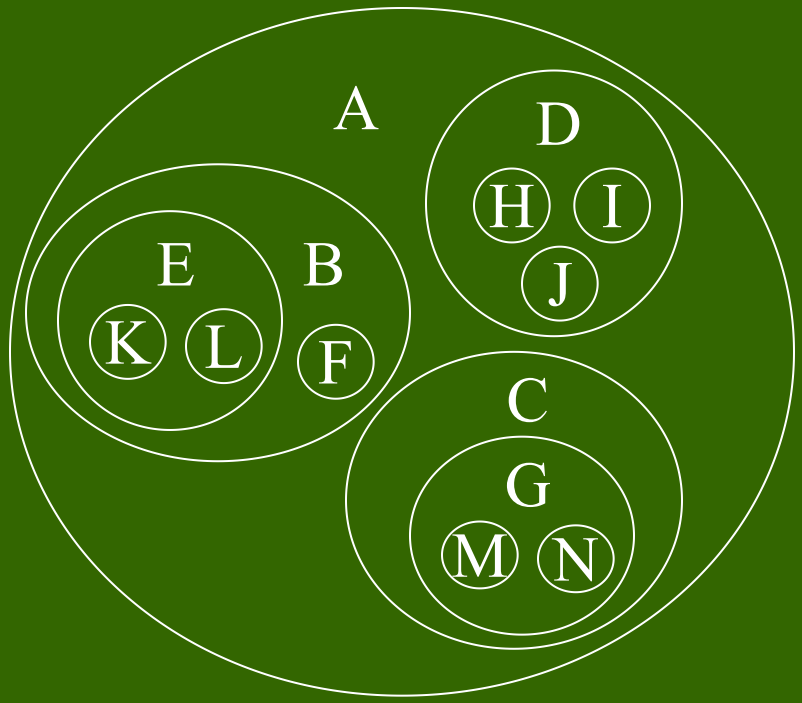
(3) 广义表形式。图6-2(b)是树的广义表形式。

(4) 凹入法表示形式。

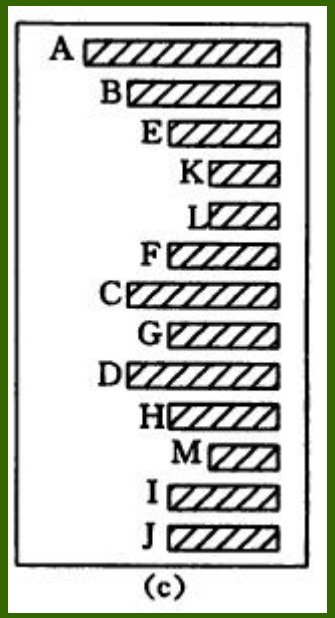
树的表示方法的多样化说明了树结构的重要性。

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

(b) 广义表形式



(a) 嵌套集合形式



(c) 凹入法形式

图6-2 树的表示形式

6.1.2 树的抽象数据类型定义

ADT Tree{

数据对象D：D是具有相同数据类型的数据元素的集合。

数据关系R：若D为空集，则称为空树；

……

基本操作：

……

} ADT Tree

详见$p_{118\sim119}$。

数据关系 R:若 D 为空集,则称为空树;

若 D 仅含一个数据元素,则 R 为空集,否则 R = {H},H 是如下二元关系:

(1) 在 D 中存在惟一的称为根的数据元素 root,它在关系 H 下无前驱;

(2) 若 D - {root} ≠ Φ,则存在 D - {root} 的一个划分 $D_1, D_2, \cdots, D_m (m>0)$,对任意 $j \neq k (1 \leqslant j,k \leqslant m)$ 有 $D_j \cap D_k = \Phi$,且对任意的 $i (1 \leqslant i \leqslant m)$,惟一存在数据元素 $x_i \in D_i$,有 $<root, x_i> \in H$;

(3) 对应于 D - {root} 的划分,$H - \{<root, x_1>, \cdots, <root, x_m>\}$ 有惟一的一个划分 $H_1, H_2, \cdots, H_m (m>0)$,对任意 $j \neq k (1 \leqslant j,k \leqslant m)$ 有 $H_j \cap H_k = \Phi$,且对任意 $i (1 \leqslant i \leqslant m)$,$H_i$ 是 D_i 上的二元关系,$(D_i, \{H_i\})$ 是一棵符合本定义的树,称为根 root 的子树。

6.2 二叉树

6.2.1 二叉树的定义

1 二叉树的定义

二叉树(Binary tree)是n(n≥0)个结点的有限集合。若n=0时称为空树，否则：

⑴ 有且只有一个特殊的结点称为树的根(Root)结点；

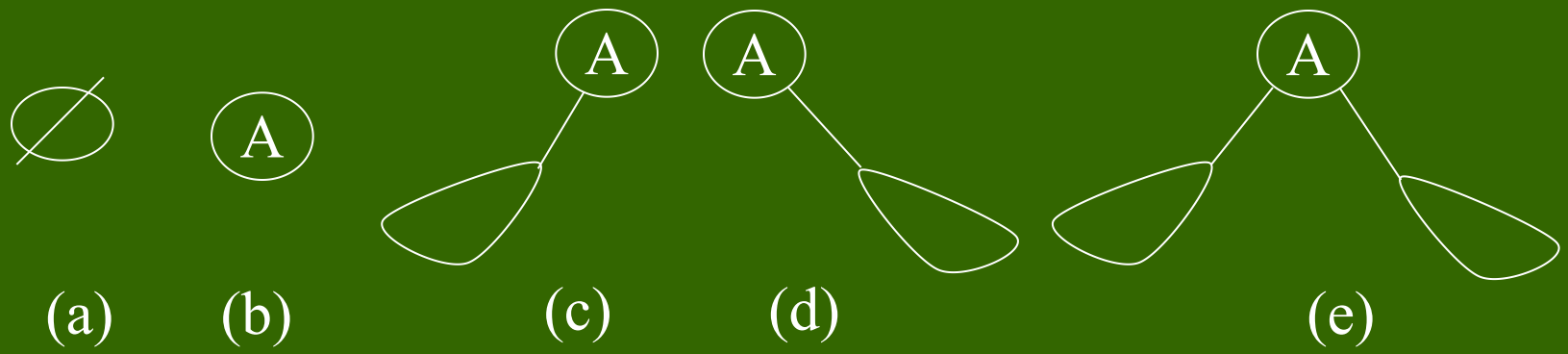
⑵ 若n>1时，其余的结点被分成为二个互不相交的子集T_1,T_2，分别称之为左、右子树，并且左、右子树又都是二叉树。

由此可知，二叉树的定义是递归的。

二叉树在树结构中起着非常重要的作用。因为二叉树结构简单，存储效率高，树的操作算法相对简单，且任何树都很容易转化成二叉树结构。上节中引入的有关树的术语也都适用于二叉树。

2　二叉树的基本形态

二叉树有5种基本形态，如图6-3所示。



(a) 空二叉树　(b) 单结点二叉树　(c) 右子树为空

(d) 左子树为空　(e) 左、右子树都不空

图6-3　二叉树的基本形态

6.2.2 二叉树的性质

性质1：在非空二叉树中，第i层上至多有2^{i-1}个结点$(i \geqslant 1)$。

证明：用数学归纳法证明。

当i=1时：只有一个根结点，$2^{1-1}=2^0=1$，命题成立。

现假设对i>1时，处在第i-1层上至多有$2^{(i-1)-1}$个结点。

由归纳假设知，第i-1层上至多有2^{i-2}个结点。由于二叉树每个结点的度最大为2，故在第i层上最大结点数为第i-1层上最大结点数的2倍。

即　　$2\times 2^{i-2}=2^{i-1}$　　证毕

性质2：深度为k的二叉树至多有2^k-1个结点（$k \geqslant 1$）。

证明：深度为k的二叉树的最大的结点数为二叉树中每层上的最大结点数之和。

由性质1知，二叉树的第1层、第2层… 第k层上的结点数至多有： 2^0、2^1 …2^{k-1} 。

∴ 总的结点数至多有： $2^0+2^1+\ldots+2^{k-1}=2^k-1$ 证毕

性质3：对任何一棵二叉树，若其叶子结点数为n_0，度为2的结点数为n_2，则$n_0=n_2+1$。

证明：设二叉树中度为1的结点数为n_1，二叉树中总结点数为N，因为二叉树中所有结点均小于或等于2，则有：$N=n_0+n_1+n_2$

再看二叉树中的分支数：

除根结点外，其余每个结点都有唯一的一个进入分支，而所有这些分支都是由度为1和2的结点射出的。设B为二叉树中的分支总数，则有：　　N=B+1

∴　　$B=n_1+2\times n_2$

∴　　$N=B+1=n_1+2\times n_2+1$

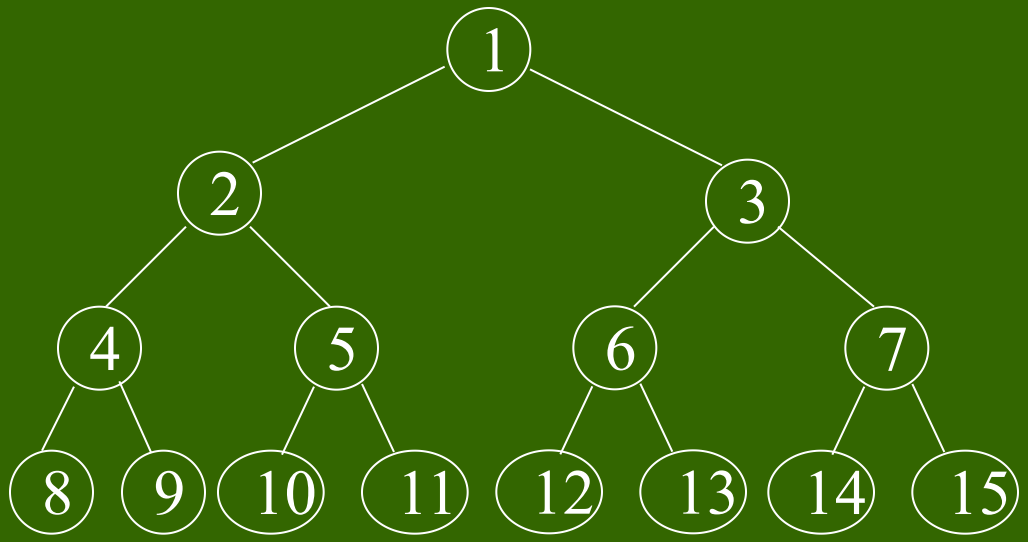
∴　　$n_0+n_1+n_2=n_1+2\times n_2+1$

即　　$n_0=n_2+1$　　证毕

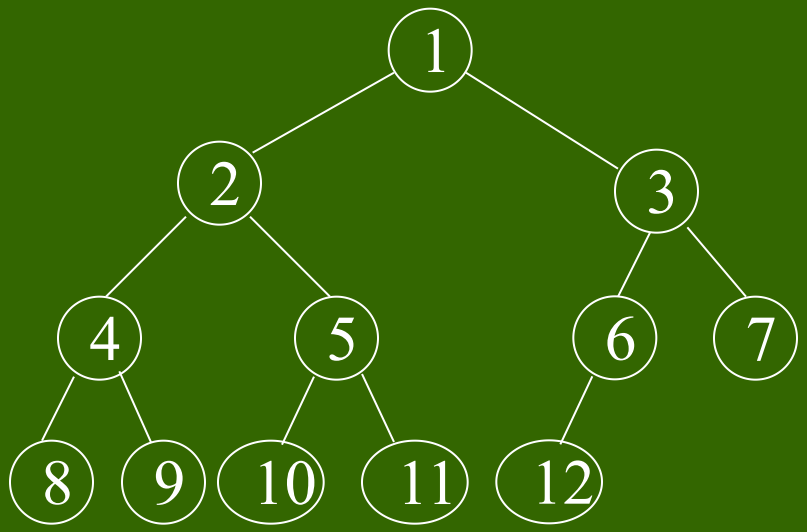
满二叉树和完全二叉树

一棵深度为k且有2^k-1个结点的二叉树称为满二叉树(Full Binary Tree)。

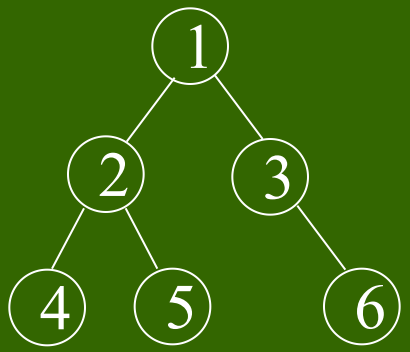
如图6-4(a) 就是一棵深度为4的满二叉树。

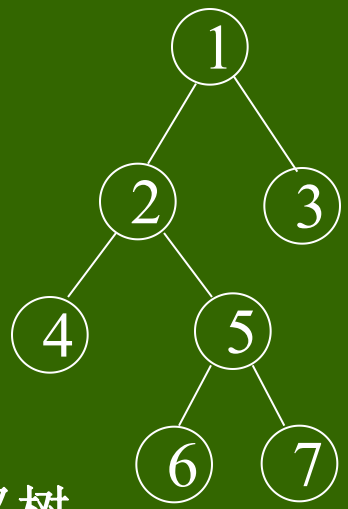


(a) 满二叉树



(b) 完全二叉树





(c) 非完全二叉树

图6-4 特殊形态的二叉树

满二叉树的特点：

- 基本特点是每一层上的结点数总是最大结点数。
- 满二叉树的所有的支结点都有左、右子树。
- 可对满二叉树的结点进行连续编号，规定从根结点开始，按“自上而下、自左至右”的原则进行。

完全二叉树（Complete Binary Tree）：如果深度为k，有n个结点的二叉树，当且仅当其每一个结点都与深度为k的满二叉树中编号从1到n的结点一一对应，该二叉树称为完全二叉树。

或深度为k的满二叉树中编号从1到n的前n个结点构成了一棵深度为k的完全二叉树。

其中 $2^{k-1} \leqq n \leqq 2^k-1$ 。

完全二叉树是满二叉树的一部分，而满二叉树是完全二叉树的特例。

完全二叉树的特点：

若完全二叉树的深度为k，则所有的叶子结点都出现在第k层或k-1层。对于任一结点，如果其右子树的最大层次为*l*，则其左子树的最大层次为*l*或*l*+1。

性质4： n个结点的完全二叉树深度为：$\lfloor \log_2 n \rfloor + 1$。

其中符号：$\lfloor x \rfloor$表示不大于x的最大整数。

$\lceil x \rceil$表示不小于x的最小整数。

证明： 假设完全二叉树的深度为k，则根据性质2及完全二叉树的定义有：

$2^{k-1}-1<n\leqslant 2^{k}-1$ 或 $2^{k-1}\leqslant n<2^{k}$

取对数得：$k-1<\log_2 n<k$ 因为k是整数。

∴ $k=\lfloor \log_2 n \rfloor+1$ 证毕

性质5：若对一棵有n个结点的完全二叉树(深度为$\lfloor \log_2 n \rfloor+1$)的结点按层（从第1层到第$\lfloor \log_2 n \rfloor+1$层)序自左至右进行编号,则对于编号为i（$1\leqslant i\leqslant n$)的结点：

(1) 若i=1：则结点i是二叉树的根，无双亲结点；否则，若i>1，则其双亲结点编号是$\lfloor i/2 \rfloor$。

(2) 如果2i>n：则结点i为叶子结点，无左孩子；否则，其左孩子结点编号是2i。

(3) 如果2i+1>n：则结点i无右孩子；否则，其右孩子结点编号是2i+1。

证明：用数学归纳法证明。首先证明(2)和(3)，然后由(2)和(3)导出(1)。

当i=1时，由完全二叉树的定义知，结点i的左孩子的编号是2，右孩子的编号是3。

若2>n，则二叉树中不存在编号为2的结点，说明结点i的左孩子不存在。

若3>n，则二叉树中不存在编号为3的结点，说明结点i的右孩子不存在。

现假设对于编号为j(1≤j≤i)的结点，(2)和(3)成立。即：

- 当2j≤n：结点j的左孩子编号是2j；当2j>n时，结点j的左孩子结点不存在。

◆ 当2j+1≦n：结点j的右孩子编号是2j+1；当2j+1>n时，结点j的右孩子结点不存在。

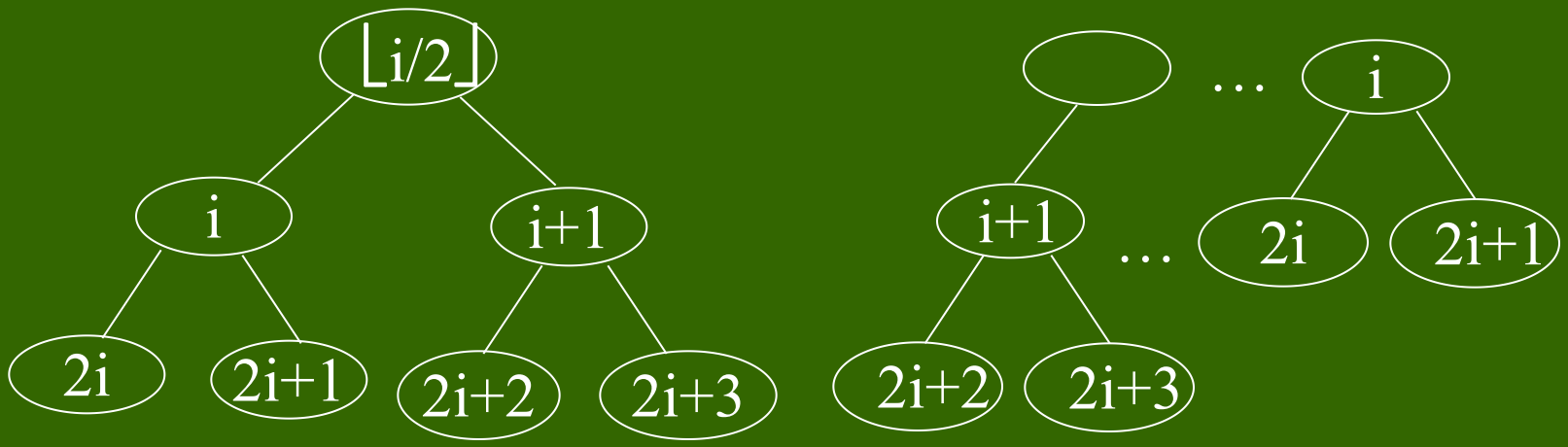
当i=j+1时，由完全二叉树的定义知，若结点i的左孩子结点存在，则其左孩子结点的编号一定等于编号为j的右孩子的编号加1，即结点i的左孩子的编号为：

$$(2j+1)+1=2(j+1)=2i$$

如图6-5所示，且有2i≦n。相反，若2i>n，则左孩子结点不存在。同样地，若结点i的右孩子结点存在，则其右孩子的编号为：2i+1，且有2i+1≦n。相反，若2i+1>n，则右孩子结点不存在。结论(2)和(3)得证。

再由(2)和(3)来证明(1)。

当i=1时，显然编号为1的是根结点，无双亲结点。



(a) i和i+1结点在同一层　　(b) i和i+1结点不在同一层

图6-5 完全二叉树中结点i和i+1的左右孩子

当i>1时，设编号为i的结点的双亲结点的编号为m，若编号为i的结点是其双亲结点的左孩子，则由(2)有：

i=2m，即m=⌊i/2⌋；

若编号为i的结点是其双亲结点的右孩子，则由(3)有：

i=2m+1，即m=⌊(i-1) /2⌋；

∴ 当i>1时，其双亲结点的编号为⌊i/2⌋。　　证毕

6.2.3 二叉树的存储结构

1 顺序存储结构

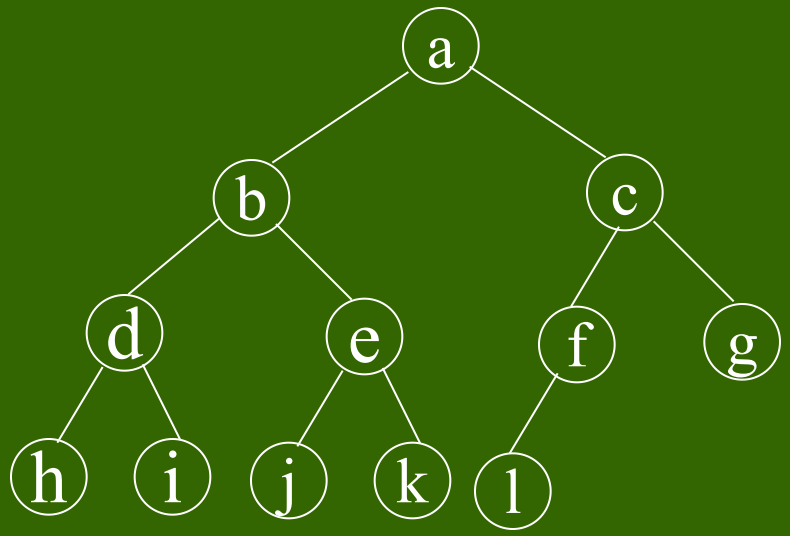
二叉树存储结构的类型定义：

```
#define MAX_SIZE  100
 typedef telemtype sqbitree[MAX_SIZE];
```

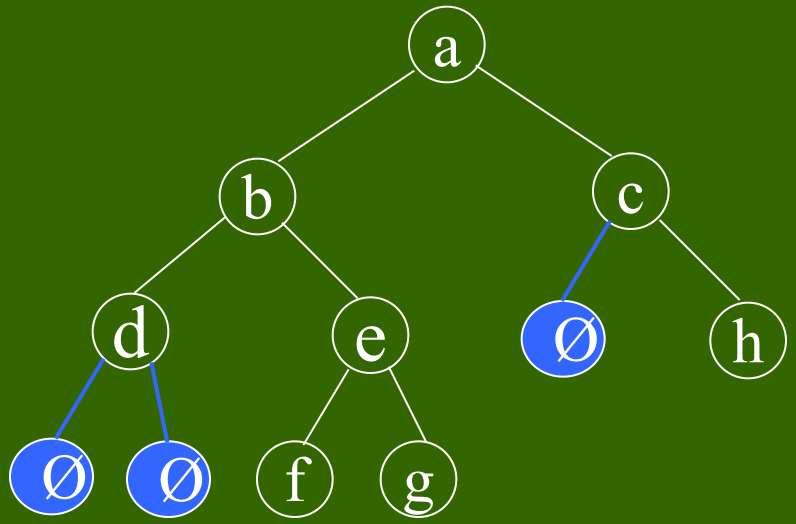
用一组地址连续的存储单元依次“自上而下、自左至右”存储完全二叉树的数据元素。

对于完全二叉树上编号为i的结点元素存储在一维数组的下标值为i-1的分量中，如图6-6(c)所示。

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，如图6-6(d)所示。



(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

图6-6 二叉树及其顺序存储形式

最坏的情况下，一个深度为k且只有k个结点的单支树需要长度为2^k-1的一维数组。

2 链式存储结构

设计不同的结点结构可构成不同的链式存储结构。

(1) 结点的类型及其定义

① 二叉链表结点。有三个域：一个数据域，两个分别指向左右子结点的指针域，如图6-7(a)所示。

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

② 三叉链表结点。除二叉链表的三个域外，再增加一个指针域，用来指向结点的父结点，如图6-7(b)所示。

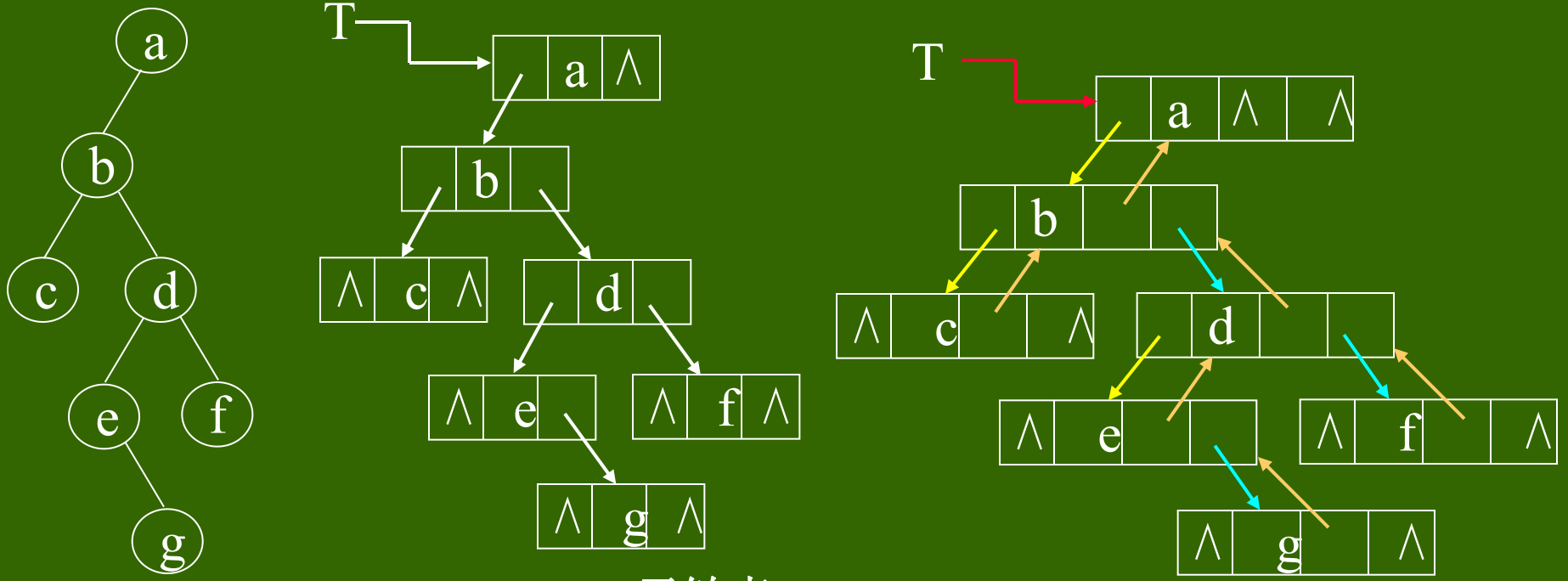
```
typedef struct BTNode_3
   {  ElemType  data ;
      struct BTNode_3  *Lchild , *Rchild , *parent ;
   }BTNode_3 ;
```

(a) 二叉链表结点　　(b) 三叉链表结点

图6-7　链表结点结构形式

(2) 二叉树的链式存储形式

例有一棵一般的二叉树，如图6-8(a)所示。以二叉链表和三叉链表方式存储的结构图分别如图6-8(b)、6-8(c)所示。



(a) 二叉树　　(b) 二叉链表　　(c) 三叉链表

图6-8　二叉树及其链式存储结构

6.3 遍历二叉树及其应用

遍历二叉树(Traversing Binary Tree)：是指按指定的规律对二叉树中的每个结点访问一次且仅访问一次。

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。

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。

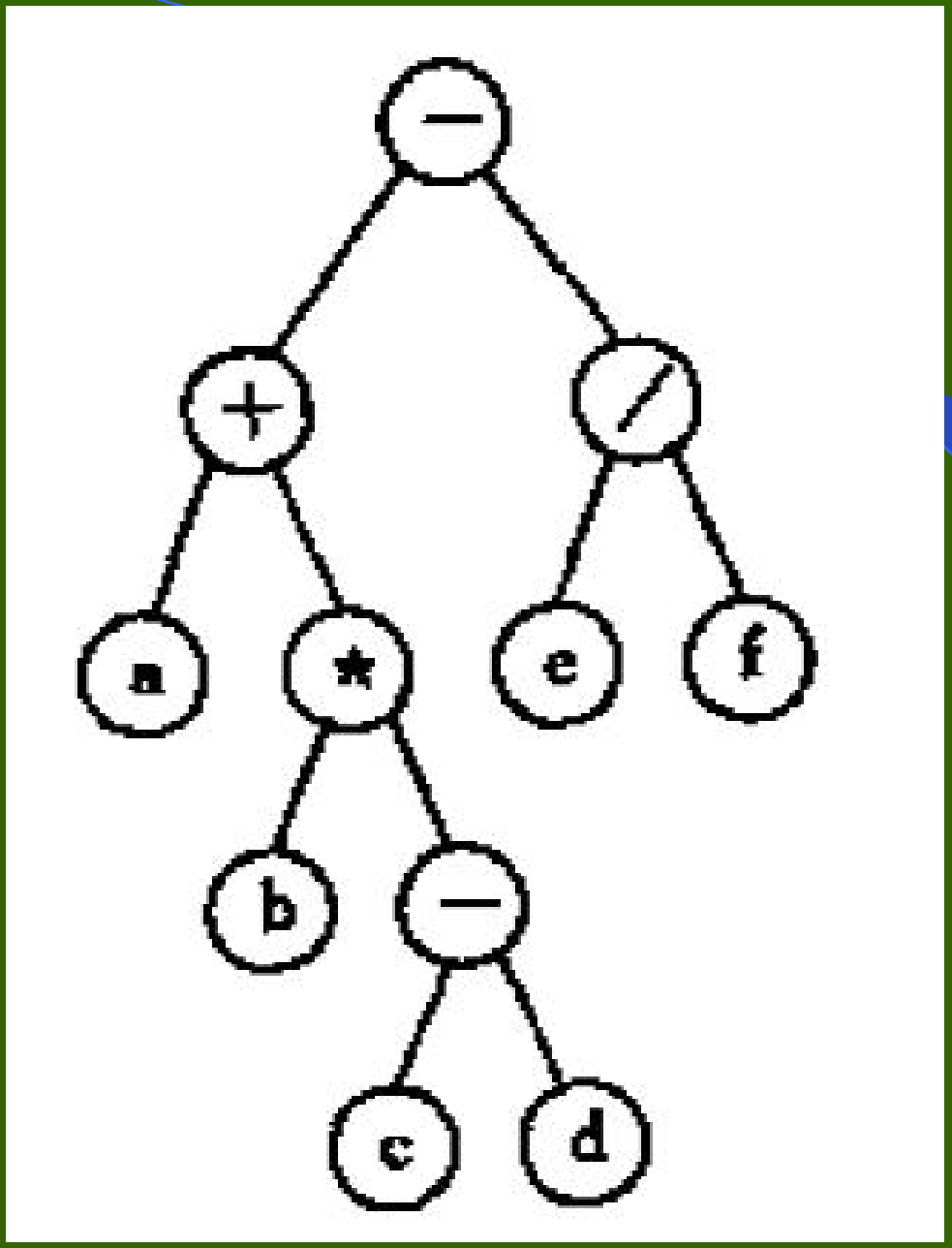
若以L、D、R分别表示遍历左子树、遍历根结点和遍历右子树，则有六种遍历方案：DLR、LDR、LRD、DRL、RDL、RLD。若规定先左后右，则只有前三种情况三种情况，分别是：

DLR——先(根)序遍历。

LDR——中(根)序遍历。

LRD——后(根)序遍历。

对于二叉树的遍历，分别讨论递归遍历算法和非递归遍历算法。递归遍历算法具有非常清晰的结构，但初学者往往难以接受或怀疑，不敢使用。实际上，递归算法是由系统通过使用堆栈来实现控制的。而非递归算法中的控制是由设计者定义和使用堆栈来实现的。


−
+
/
a
*
e
f
b
−
c
d

6.3.1 先序遍历二叉树

1 递归算法

算法的递归定义是：

若二叉树为空，则遍历结束；否则

(1) 访问根结点；

(2) 先序遍历左子树(递归调用本算法)；

(3) 先序遍历右子树(递归调用本算法)。

先序遍历的递归算法

```
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。

2 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。

算法实现：

```
#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 中序遍历二叉树

1 递归算法

算法的递归定义是：

若二叉树为空，则遍历结束；否则

(1) 中序遍历左子树(递归调用本算法)；

(2) 访问根结点；

(3) 中序遍历右子树(递归调用本算法)。

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) ;
         }
   }      /*图6-8(a) 的二叉树，输出的次序是：  cbegdfa */
```

2 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。

算法实现：

```
#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 后序遍历二叉树

1 递归算法

算法的递归定义是：

若二叉树为空，则遍历结束；否则

⑴ 后序遍历左子树(递归调用本算法)；

⑵ 后序遍历右子树(递归调用本算法)；

⑶ 访问根结点。

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  */
         }
   }     /*图6-8(a)  的二叉树，输出的次序是：  cgefdba    */
```

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) 。

如图6-9所示的二叉树表示表达式：(a+b*(c-d)-e/f)

按不同的次序遍历此二叉树，将访问的结点按先后次序排列起来的次序是：

其先序序列为：　-+a*b-cd/ef

其中序序列为：　a+b*c-d-e/f

其后序序列为：　abcd-*+ef/-

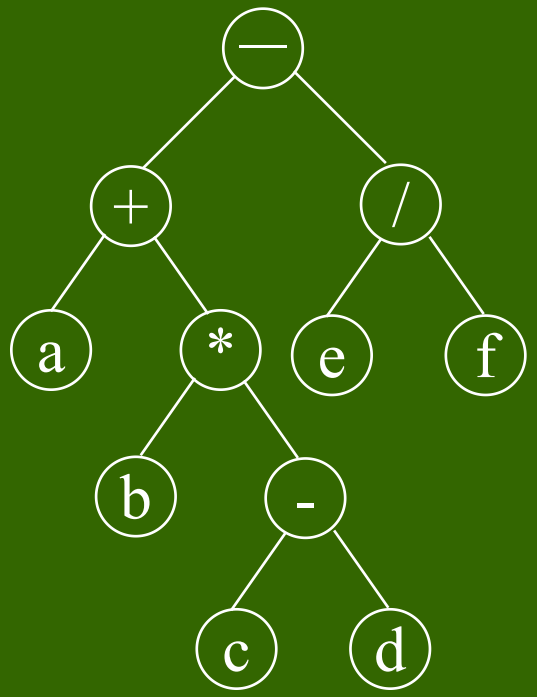


图6-9　表达式 (a+b*(c-d)-e/f)二叉树

2 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；

⑴ 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 。

⑵ 若p不为空，转(1)，否则，取状态标志值tag：

⑶ 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)；

⑷ 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。

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；

⑴ 队首元素出队到p；

⑵访问p所指向的结点；

⑶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  */
               }
         }
```

6.3.5 二叉树遍历算法的应用

“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。

1 二叉树的二叉链表创建

(1) 按满二叉树方式建立 (补充)

在此补充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。

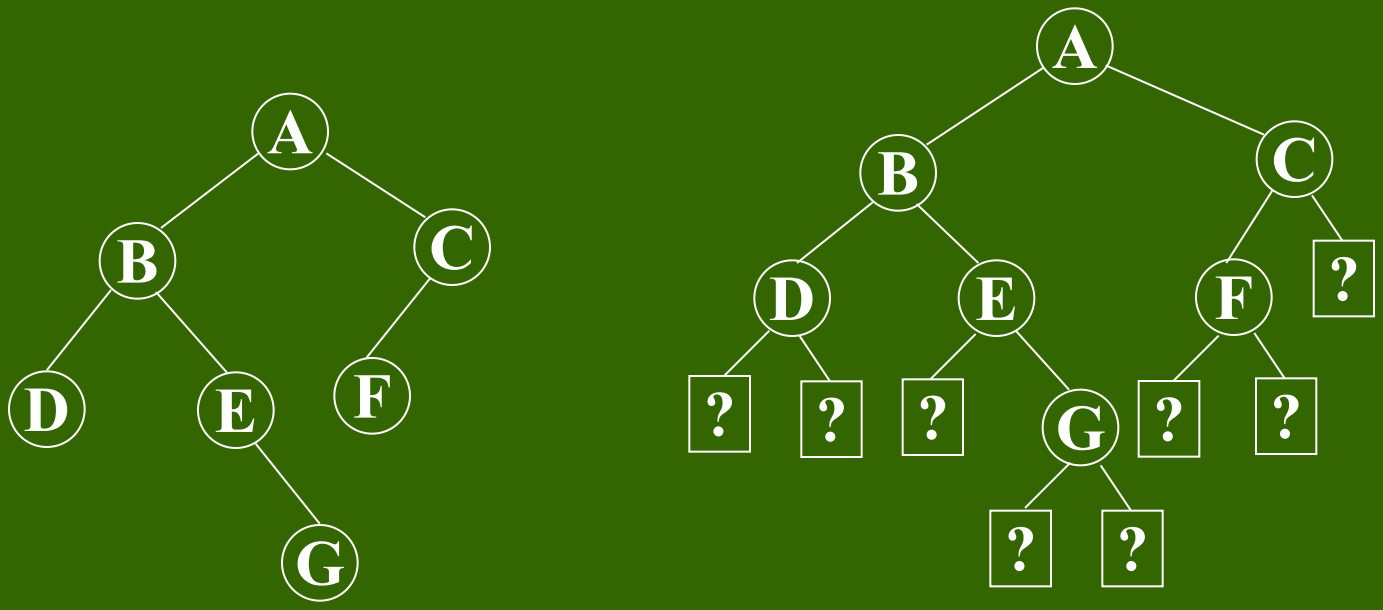
算法实现：

```
#define MAX_NODE  50
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”，就可以得到由该二叉树所扩充的二叉树。两棵二叉树T_1及其扩充的二叉树T_2如图6-10所示。



(a) 二叉树T_1　　(b) T_1的扩充二叉树T_2

图6-10　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；

下面的算法是二叉树的前序创建的递归算法，读入一棵二叉树对应的扩充二叉树的前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。

算法实现：

```
#define NULLKY '?'
#define MAX_NODE  50
typedef struct BTNode
   {  char  data ;
      struct BTNode *Lchild , *Rchild ;
   }BTNode ;
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() ; 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) ;
        }
}
```

当希望创建图6-10(a)所示的二叉树时，输入的字符序列应当是：

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

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。

算法实现：

```
#define  MAX_NODE  50
int  search_leaves( BTNode  *T)
   {  BTNode  *Stack[MAX_NODE] ,*p=T;
      int  top=0, num=0;
      if  (T!=NULL)
```

```
    {  stack[++top]=p ;
       while (top>0)
          {  p=stack[top--] ;
             if (p->Lchild==NULL&&p-
          >Rchild==NULL)  num++ ;
                if  (p->Rchild!=NULL )
                    stack[++top]=p->Rchild;
                if  (p->Lchild!=NULL )
                       stack[++top]=p->Lchild;
             }
       }
   return(num) ;
}
```

3 求二叉树的深度

利用层次遍历算法可以直接求得二叉树的深度。

算法实现：

```
#define  MAX_NODE  50
int  search_depth( BTNode  *T)
   {  BTNode  *Queue[MAX_NODE] ,*p=T;
      int  front=0 , rear=0, depth=0, level ;
         /*  level总是指向访问层的最后一个结点在队列的位置  */
      if  (T!=NULL)
         {  Queue[++rear]=p;    /*   根结点入队  */
            level=rear ;    /*  根是第1层的最后一个节点  */
```

```
        while (front<rear)
            {  p=Queue[++front];
               if (p->Lchild!=NULL)
                    Queue[++rear]=p;   /*  左结点入队  */
               if (p->Rchild!=NULL)
                    Queue[++rear]=p;   /*  左结点入队  */
              if (front==level)
                 /*  正访问的是当前层的最后一个结点  */
                { depth++ ;  level=rear ;  }
            }
    }
}
```

6.4 线索树

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，如图6-10所示。

Lchild	Ltag	data	Rchild	Rtag

图6-10 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

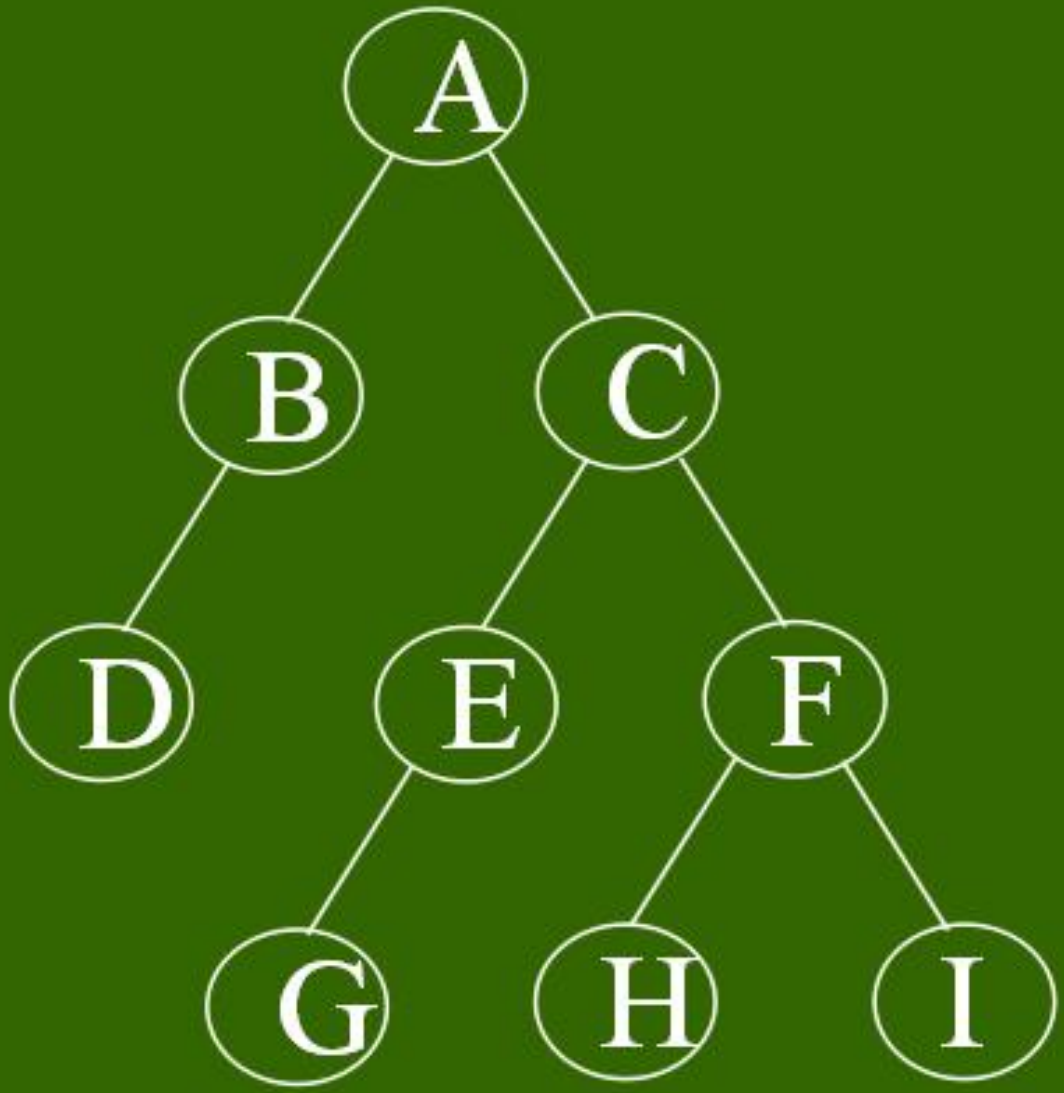
Rtag= { 0：Rchild域指示结点的右孩子
1：Rchild域指示结点的后继

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。

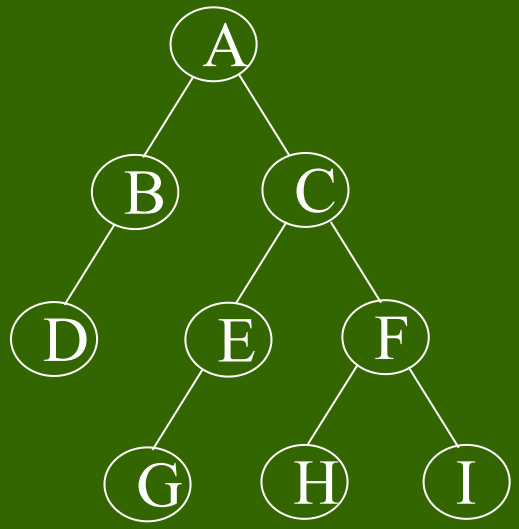
线索二叉树的结点结构与示例

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

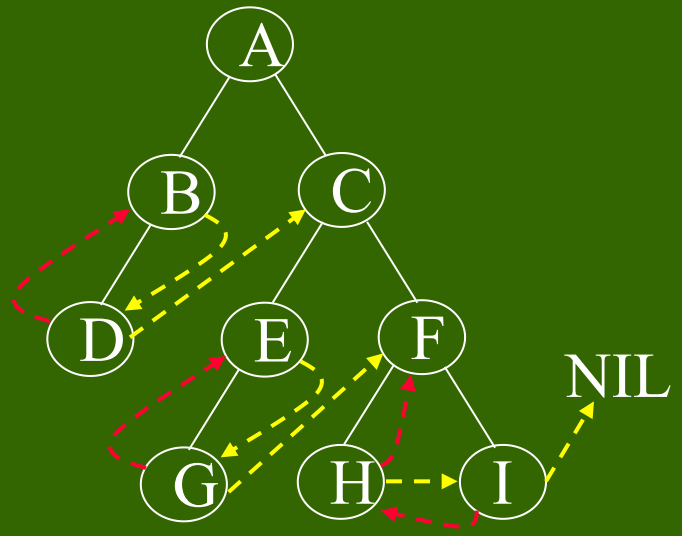
如图6-11是二叉树及相应的各种线索树示例。



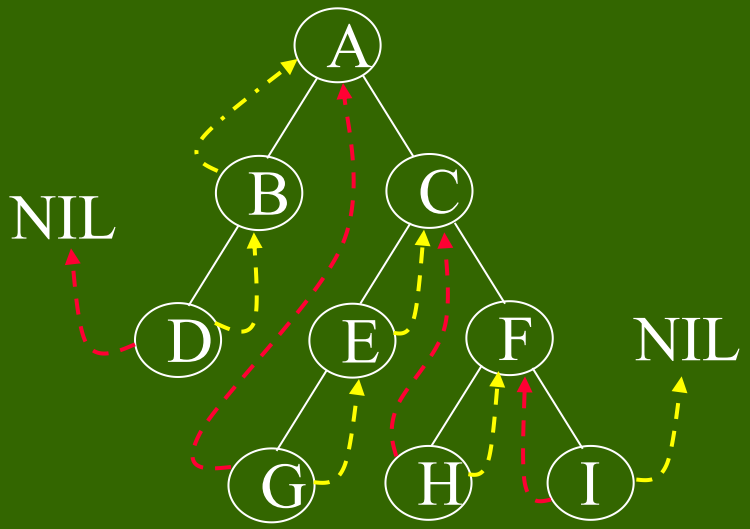
(a) 二叉树



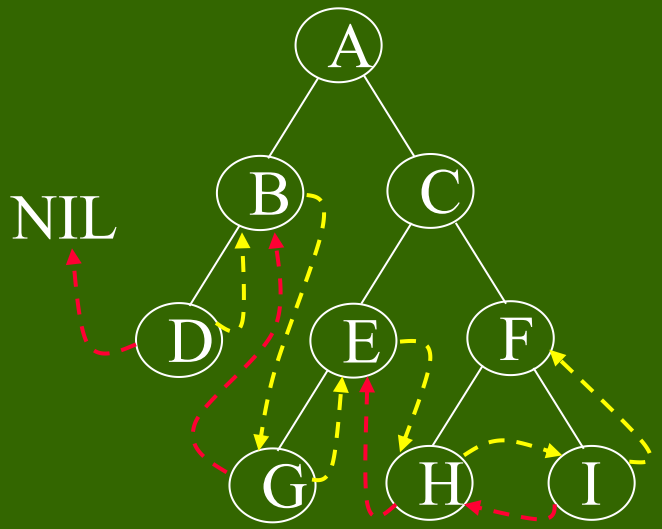
(a) 二叉树



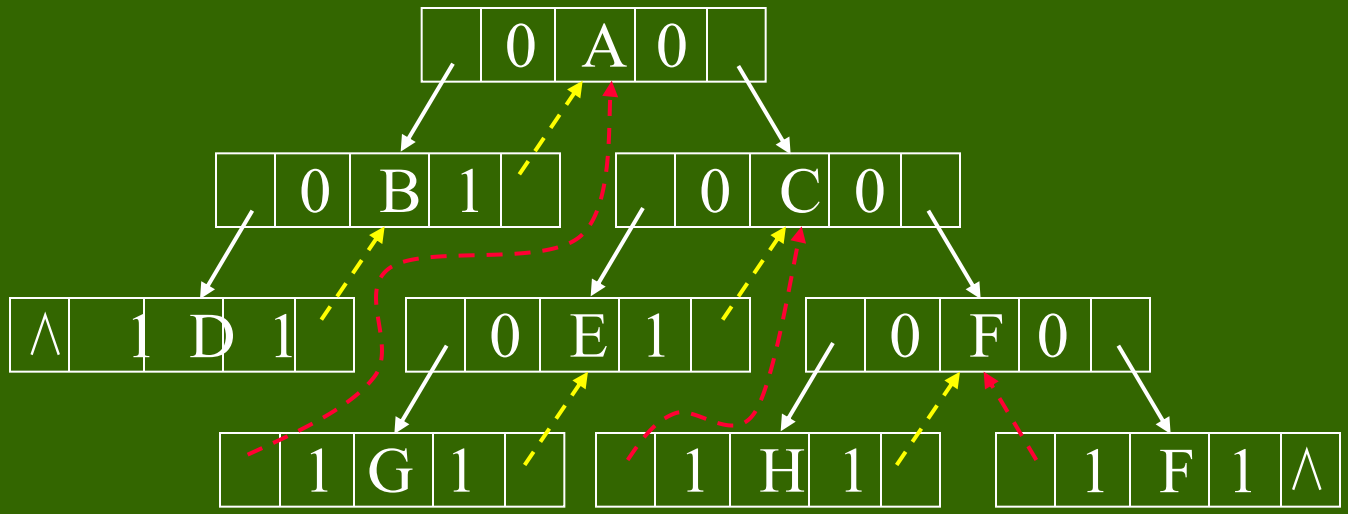
(b) 先序线索树的逻辑形式
结点序列：**ABDCEGFHI**



(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**



(e) 中序线索树的链表结构

图6-11 线索二叉树及其存储结构

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。

在线索树上进行遍历，只要先找到序列中的第一个结点，然后就可以依次找结点的直接后继结点直到后继为空为止。

如何在线索树中找结点的直接后继?以图6-11(c)，(e)所示的中序线索树为例：

- 树中所有叶子结点的左、右链都是线索。右链直接指示了结点的直接后继，如结点G的直接后继是结点E。
- 树中所有非叶子结点的右链都是指针。根据中序遍历的规律，非叶子结点的直接后继是遍历其右子树时访问的第一个结点，即右子树中最左下的(叶子)结点。如结点C的直接后继：沿右指针找到右子树的根结点F，然后沿左链往下直到Ltag=1的结点即为C的直接后继结点H。

如何在线索树中找结点的直接前驱?若结点的Ltag=1，则左链是线索，指示其直接前驱；否则，遍历左子树时访问的最后一个结点(即沿左子树中最右往下的结点）为其直接前驱结点。

对于后序遍历的线索树中找结点的直接后继比较复杂，可分以下三种情况：

- 若结点是二叉树的根结点：其直接后继为空；
- 若结点是其父结点的右孩子或左孩子且其父结点没有右子树：直接后继为其父结点；
- 若结点是其父结点的左孩子且其父结点有右子树：直接后继是对其父结点的右子树按后序遍历的第一个结点。

6.4.1 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。

仿照线性表的存储结构，在二叉树的线索链表上也添加一个头结点head，头结点的指针域的安排是（中序）：

- Lchild域：指向二叉树的根结点；
- Rchild域：指向中序遍历时的最后一个结点；
- 二叉树中序序列中的第一个结点Lchild指针域和最后一个结点Rchild指针域均指向头结点head。

如图为二叉树建立了一个双向线索链表，对一棵线索二叉树既可从头结点也可从最后一个结点开始按寻找直接后继进行遍历。显然，这种遍历不需要堆栈，如图6-12所示。结点类型定义

```
#define  MAX_NODE   50
typedef enmu{Link , Thread} PointerTag ;
   /*  Link=0表示指针，  Thread=1表示线索   */
typedef struct BiThrNode
   {   ElemType  data;
       struct BiTreeNode *Lchild , *Rchild ;
       PointerTag  Ltag , Rtag ;
   }BiThrNode;
```

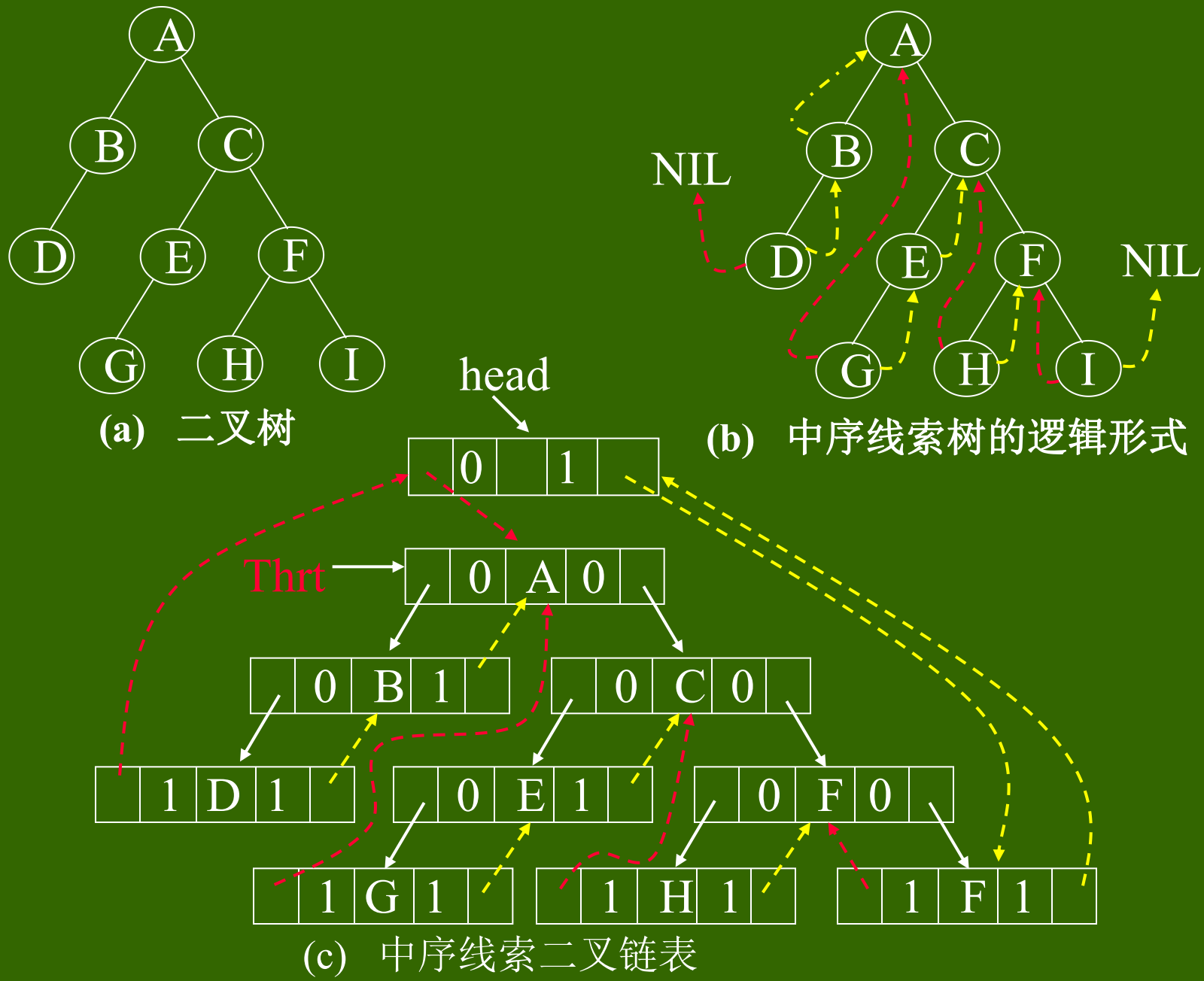


图6-12 中序线索二叉树及其存储结构

1 中序线索化二叉树

```
void inorder_Threading(BiThrNode *T)
    {  BiThrNode  *stack[MAX_NODE];
       BiThrNode  *last=NULL, *p=T ;
       int top=0 ;
       while  (p!=NULL||top>0)
          if  (p!=NULL)  {  stack[++top]=p;  p=p->Lchild;  }
          else
             {  p=stack[top--] ;
                if (p->Lchild!=NULL)  p->Ltag=0 ;
                else  {  p->Ltag=1 ;  p->Lchild==last ;  }
                if  (last!=NULL)
                   if (last->Rchild!=NULL) last->Rtag=0 ;
```

```
                else {  last->Rtag=1 ; last->Rchild!=p ;  }
            last=p ;
            p=p->Rchild;
        }//else
    }//while
     last->Rtag=1;  /*   最后一个结点是叶子结点   */
}
```

2 先序线索化二叉树

```
void preorder_Threading(BiThrNode *T)
    {  BiThrNode  *stack[MAX_NODE];
       BiThrNode  *last=NULL, *p ;
       int top=0 ;
       if  (T!=NULL)
          {  stack[++top]=T;
             while (top>0)
                {  p=stack[top--] ;
                   if (p->Lchild!=NULL)  p->Ltag=0 ;
                   else  {  p->Ltag=1 ;  p->Lchild=last ;  }
                   if  (last!=NULL)
                      if (last->Rchild!=NULL) last->Rtag=0 ;
```

```
            else
                {  last->Rtag=1 ; last->Rchild!=p ;  }
            last=p ;
            if (p->Rchild!=NULL)
                stack[++top]=p->Rchild ;
            if (p->Lchild!=NULL)
                stack[++top]=p->Lchild ;
        }
    Last->Rtag=1;  /*   最后一个结点是叶子结点 */
}
}
```

6.4.2 线索二叉树的遍历

在线索二叉树中，由于有线索存在，在某些情况下可以方便地找到指定结点在某种遍历序列中的直接前驱或直接后继。此外，在线索二叉树上进行某种遍历比在一般的二叉树上进行这种遍历要容易得多，不需要设置堆栈，且算法十分简洁。

1 先序线索二叉树的先序遍历

```
void preorder_Thread_bt(BiThrNode *T)
    {  BiThrNode  *p=T ;
       while (p!=NULL)
          {  visit(p->data) ;
             if (p->Ltag==0)  p=p->Lchild;
             else  p=p->Rchild;
          }
    }
```

2 中序线索二叉树的中序遍历

```
void inorder_Thread_bt(BiThrNode *T)
    {  BiThrNode  *p ;
       if  (T!=NULL)
          {   p=T;
              while  (p->Ltag==0 )
                  p=p->Lchild;   /*  寻找最左的结点  */
              while  (p!=NULL)
                 {   visit(p->data) ;
                     if  (p->Rtag==1)
                         p=p->Rchild ;    /*  通过右线索找到后继  */
                     else    /*  否则，右子树的最左结点为后继  */
                        {  p=p->Rchild ;
```

```
                    while (p->Ltag==0 )  p=p->Lchild;
                }
            }
        }
}
```

6.5 树与森林

本节将讨论树的存储结构、树及森林与二叉树之间的相互转换、树的遍历等。

6.5.1 树的存储结构

树的存储结构根据应用的不同而不同。

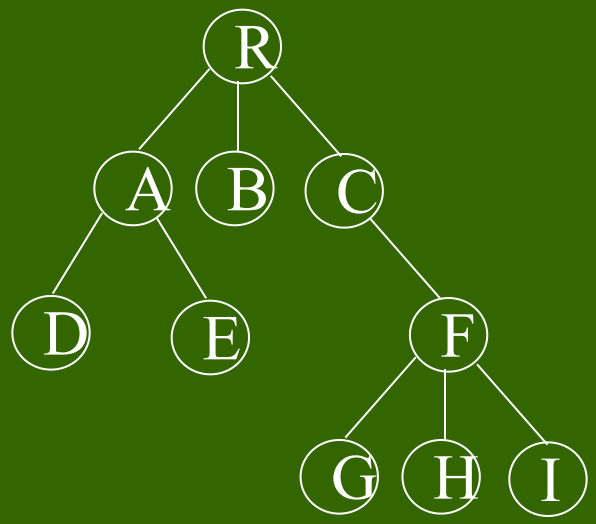
1 双亲表示法(顺序存储结构)

用一组连续的存储空间来存储树的结点，同时在每个结点中附加一个指示器(整数域)，用以指示双亲结点的位置(下标值)。数组元素及数组的类型定义如下：

```
#define MAX_SIZE  100
typedef  struct PTNode
   {  ElemType  data ;
      int  parent ;
   }PTNode ;
```

```
typedef  struct
   {  PTNode  Nodes[MAX_SIZE] ;
      int  root;    /*  根结点位置  */
      int  num ;   /*  结点数   */
   }Ptree ;
```

图6-13所示是一棵树及其双亲表示的存储结构。这种存储结构利用了任一结点的父结点唯一的性质。可以方便地直接找到任一结点的父结点，但求结点的子结点时需要扫描整个数组。



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	I	6

图6-13 树的双亲存储结构

2 孩子链表表示法

树中每个结点有多个指针域，每个指针指向其一棵子树的根结点。有两种结点结构。

(1) 定长结点结构

指针域的数目就是树的度。

其特点是：链表结构简单，但指针域的浪费明显。结点结构如图6-14(a) 所示。在一棵有n个结点，度为k的树中必有n(k-1)+1空指针域。

(2) 不定长结点结构

树中每个结点的指针域数量不同，是该结点的度，如图6-14(b) 所示。没有多余的指针域，但操作不便。

(a) 定长结点结构

(b) 不定长结点结构

图6-14 孩子表示法的结点结构

(3) 复合链表结构

对于树中的每个结点，其孩子结点用带头结点的单链表表示，表结点和头结点的结构如图6-15所示。

n个结点的树有n个(孩子)单链表(叶子结点的孩子链表为空)，而n个头结点又组成一个线性表且以顺序存储结构表示。

(a) 头结点

(b) 表结点

图6-15 孩子链表结点结构

数据结构类型定义如下：

```
#define  MAX_NODE  100
typedef  struct  listnode
   {   int   childno ;    /*  孩子结点编号  */
       struct listno  *next ;
   }CTNode;    /*  表结点结构  */
typedef  struct
   {  ElemType   data ;
       CTNode  *firstchild ;
   }HNode;    /*  头结点结构  */
```

```
typedef  struct
   {  HNode   nodes[MAX_NODE] ;
      int  root;   /*  根结点位置  */
      int  num ;  /*  结点数   */
   }CLinkList;   /*  头结点结构  */
```

图6-13所示的树T的孩子链表表示的存储结构如图6-16所示。

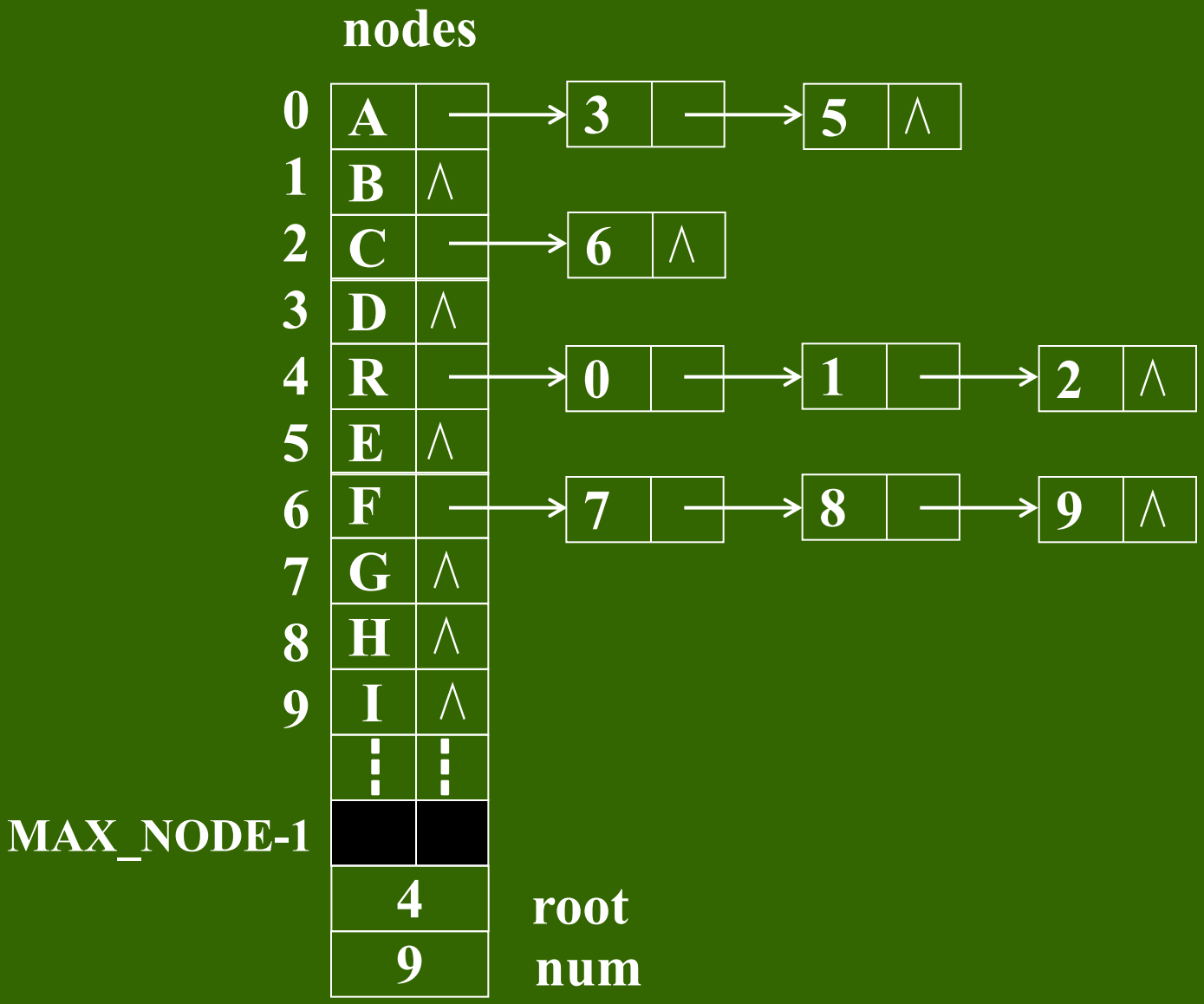


图6-16　图6-13的树T的孩子链表存储结构

3 孩子兄弟表示法(二叉树表示法)

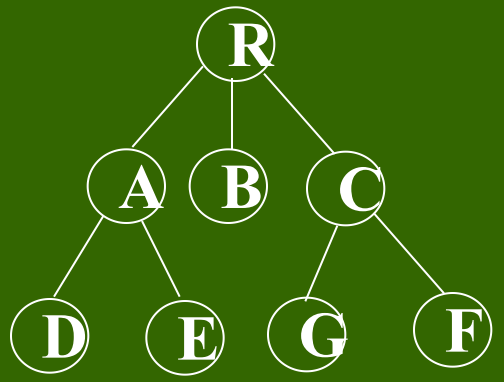
以二叉链表作为树的存储结构，其结点形式如图6-17(a)所示。

两个指针域：分别指向结点的第一个子结点和下一个兄弟结点。结点类型定义如下：

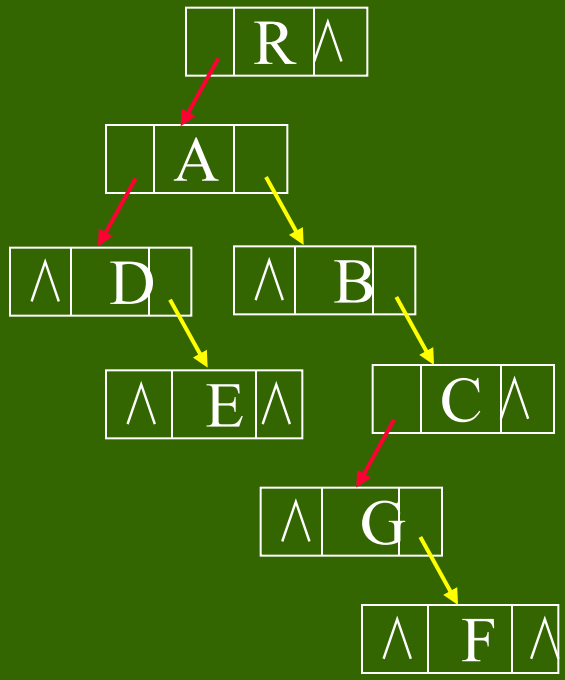
```
typedef  struct   CSnode
  {  ElemType   data ;
     struct   CSnode *firstchild, *nextsibling ;
  }CSNode;
```

图6-17(b)所示树的孩子兄弟表示的存储结构如图6-17(c)。

(a) 结点结构



(b) 树



(c) 孩子兄弟存储结构

图6-17 树及孩子兄弟存储结构

6.5.2 森林与二叉树的转换

由于二叉树和树都可用二叉链表作为存储结构，对比各自的结点结构可以看出，以二叉链表作为媒介可以导出树和二叉树之间的一个对应关系。

- 从物理结构来看，树和二叉树的二叉链表是相同的，只是对指针的逻辑解释不同而已。
- 从树的二叉链表表示的定义可知，任何一棵和树对应的二叉树，其右子树一定为空。

图6-18直观地展示了树和二叉树之间的对应关系。

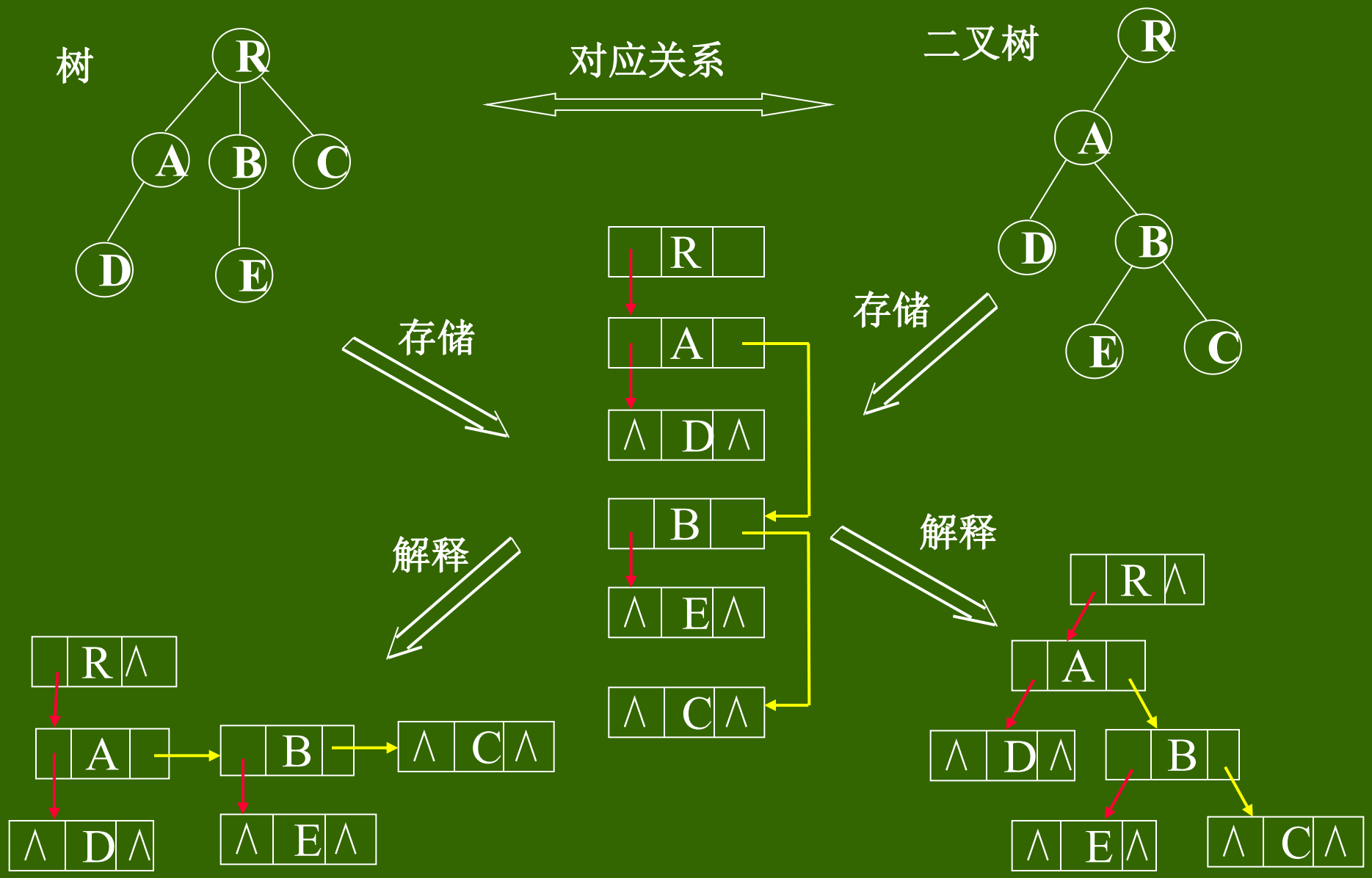


图6-18 树与二叉树的对应关系

1 树转换成二叉树

对于一般的树，可以方便地转换成一棵唯一的二叉树与之对应。将树转换成二叉树在“孩子兄弟表示法”中已给出，其详细步骤是：

⑴ 加虚线。在树的每层按从“左至右”的顺序在兄弟结点之间加虚线相连。

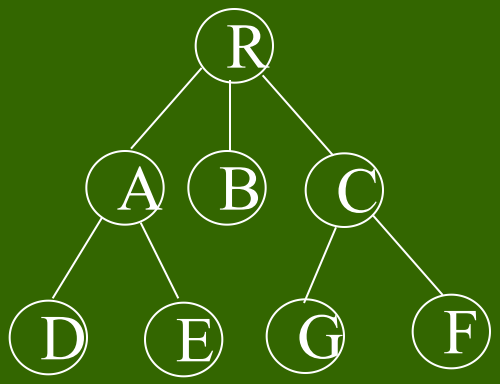
⑵ 去连线。除最左的第一个子结点外，父结点与所有其它子结点的连线都去掉。

⑶ 旋转。将树顺时针旋转45^0，原有的实线左斜。

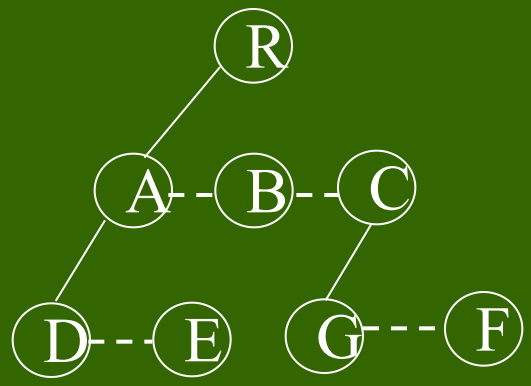
⑷ 整型。将旋转后树中的所有虚线改为实线，并向右斜。该转换过程如图6-19所示。

这样转换后的二叉树的特点是：

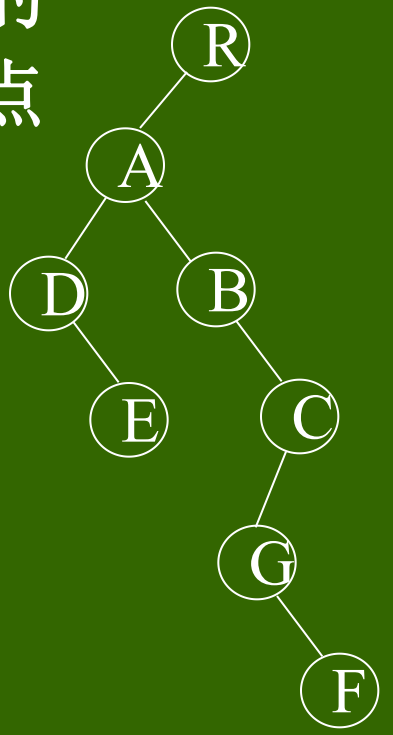
- 二叉树的根结点没有右子树，只有左子树；
- 左子结点仍然是原来树中相应结点的左子结点，而所有沿右链往下的右子结点均是原来树中该结点的兄弟结点。



(a) 一般的树



(b) 加虚线，去连线后



(C) 转换后的二叉树

图6-19 树向二叉树的转换过程

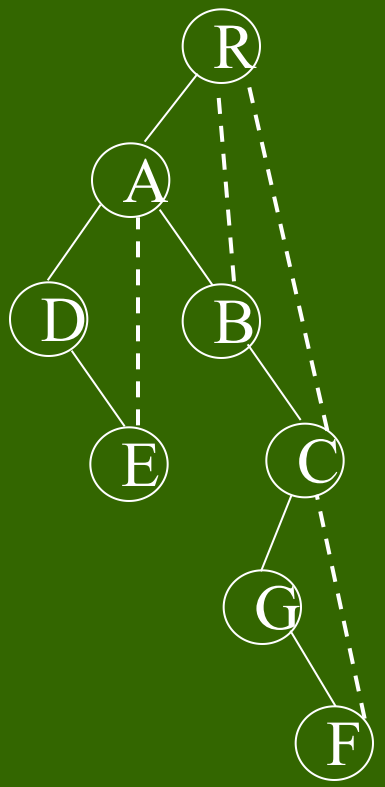
2 二叉树转换成树

对于一棵转换后的二叉树，如何还原成原来的树？其步骤是：

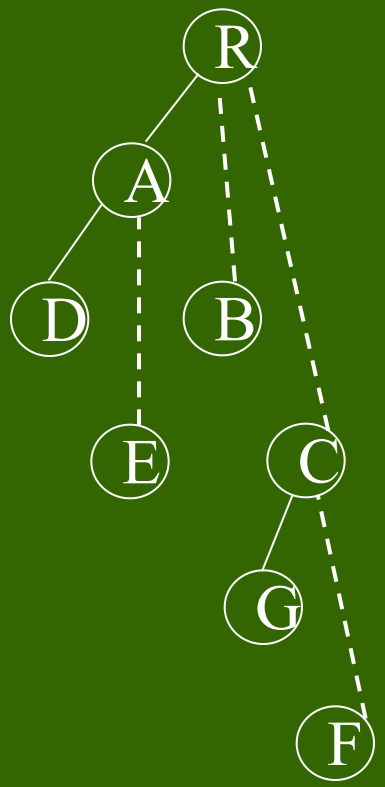
⑴ 加虚线。若某结点i是其父结点的左子树的根结点，则将该结点i的右子结点以及沿右子链不断地搜索所有的右子结点，将所有这些右子结点与i结点的父结点之间加虚线相连，如图6-20(a)所示。

⑵ 去连线。去掉二叉树中所有父结点与其右子结点之间的连线，如图6-20(b)所示。

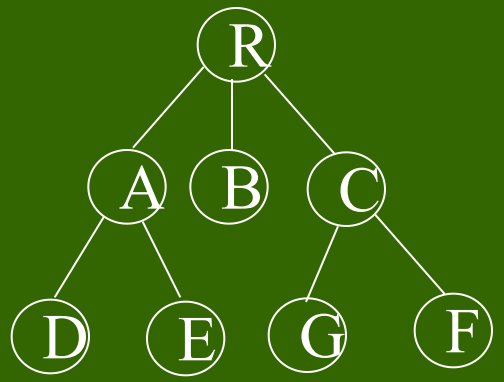
⑶ 规整化。将图中各结点按层次排列且将所有的虚线变成实线，如图6-20(c)所示。



(a) 加虚线后



(b) 去连线后



(C) 还原后的树

图6-20 二叉树向树的转换过程

3 森林转换成二叉树

当一般的树转换成二叉树后，二叉树的右子树必为空。若把森林中的第二棵树(转换成二叉树后)的根结点作为第一棵树(二叉树)的根结点的兄弟结点，则可导出森林转换成二叉树的转换算法如下：

设F={T_1, T_2,… ,T_n}是森林，则按以下规则可转换成一棵二叉树B=(root，LB，RB)

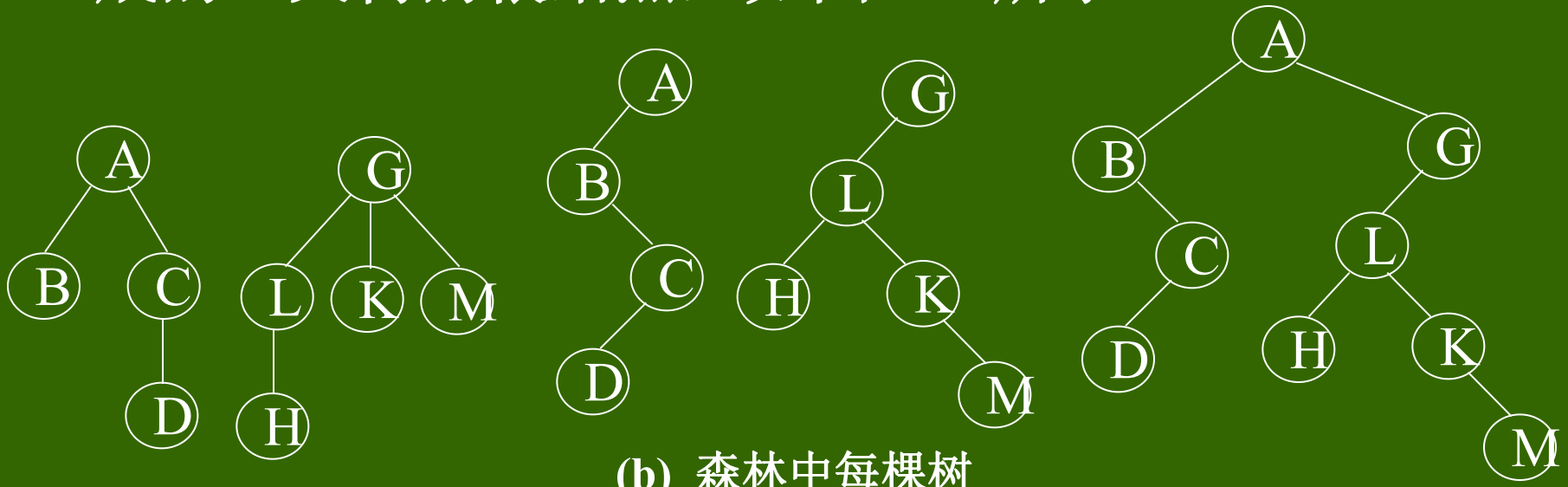
① 若n=0，则B是空树。

② 若n>0，则二叉树B的根是森林T_1的根root(T_1)，B的左子树LB是B(T_{11},T_{12},… ,T_{1m})，其中T_{11},T_{12},… ,T_{1m}是T_1的子树(转换后)，而其右子树RB是从森林F'={T_2, T_3,… ,T_n}转换而成的二叉树。

转换步骤：

① 将$F=\{T_1, T_2, \cdots, T_n\}$中的每棵树转换成二叉树。

② 按给出的森林中树的次序，从最后一棵二叉树开始，每棵二叉树作为前一棵二叉树的根结点的右子树，依次类推，则第一棵树的根结点就是转换后生成的二叉树的根结点，如图6-21所示。



(a) 森林　　(b) 森林中每棵树对应的二叉树　　(c) 森林对应的二叉树

图6-21 森林转换成二叉树的过程

4 二叉树转换成森林

若B=(root，LB，RB)是一棵二叉树，则可以将其转换成由若干棵树构成的森林：$F=\{T_1, T_2, \cdots, T_n\}$。

转换算法：

① 若B是空树，则F为空。

② 若B非空，则F中第一棵树T_1的根root(T_1)就是二叉树的根root，T_1中根结点的子森林F_1是由树B的左子树LB转换而成的森林；F中除T_1外其余树组成的的森林$F'=\{T_2, T_3, \cdots, T_n\}$ 是由B右子树RB转换得到的森林。

上述转换规则是递归的，可以写出其递归算法。以下给出具体的还原步骤。

① 去连线。将二叉树B的根结点与其右子结点以及沿右子结点链方向的所有右子结点的连线全部去掉，得到若干棵孤立的二叉树，每一棵就是原来森林F中的树依次对应的二叉树，如图6-22(b)所示。

② 二叉树的还原。将各棵孤立的二叉树按二叉树还原为树的方法还原成一般的树，如图6- 22(c)所示。

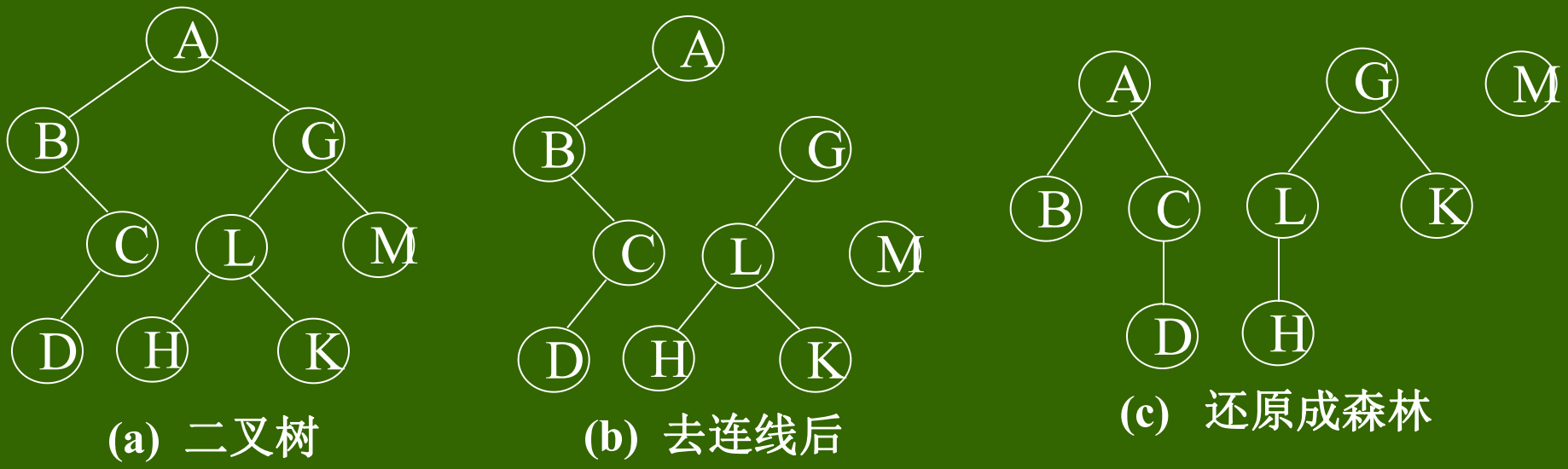


图6-22 二叉树还原成森林的过程

6.5.3 树和森林的遍历

1 树的遍历

由树结构的定义可知，树的遍历有二种方法。

(1) 先序遍历：先访问根结点，然后依次先序遍历完每棵子树。如图6-23的树，先序遍历的次序是：

ABCDEFGIJHK

(2) 后序遍历：先依次后序遍历完每棵子树，然后访问根结点。如图6-23的树，后序遍历的次序是：

CDBFIJGHEKA

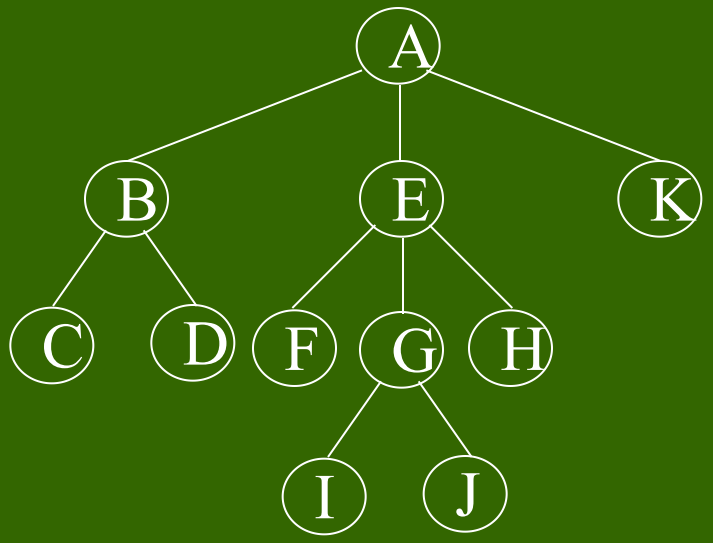


图6-23 树

说明：

- 树的先序遍历实质上与将树转换成二叉树后对二叉树的先序遍历相同。
- 树的后序遍历实质上与将树转换成二叉树后对二叉树的中序遍历相同。

2 森林的遍历

设$F=\{T_1, T_2, \cdots, T_n\}$是森林，对F的遍历有二种方法。

(1) 先序遍历：按先序遍历树的方式依次遍历F中的每棵树。

(2) 中序遍历：按后序遍历树的方式依次遍历F中的每棵树。

6.6 赫夫曼树及其应用

赫夫曼(Huffman)树又称最优树，是一类带权路径长度最短的树，有着广泛的应用。

6.6.1 最优二叉树

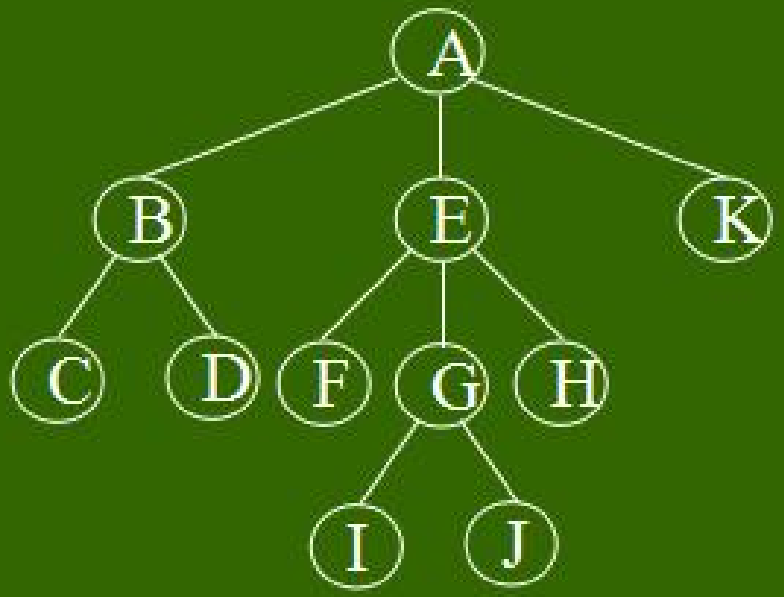


图6-23 树

1 基本概念

① 结点路径：从树中一个结点到另一个结点之间的分支构成这两个结点之

② 路径长度：结点路径上

度。

③ 树的路径长度：从树根到每一个结点的路径长度之和。

例图6-23的树。A到F：结点路径 AEF；路径长度(即边的数目) 2；树的路径长度：

3×1+5×2+2×3=19

④ 结点的带权路径长度：从该结点的到树的根结点之间的路径长度与结点的权(值)的乘积。

权(值)：各种开销、代价、频度等的抽象称呼。

⑤ 树的带权路径长度：树中所有叶子结点的带权路径长度之和，记做：

$$WPL=w_1\times l_1+w_2\times l_2+\cdots+w_n\times l_n=\sum w_i\times l_i \quad (i=1,2,\cdots,n)$$

其中：n为叶子结点的个数；w_i为第i个结点的权值；l_i为第i个结点的路径长度。

⑥ Huffman树：具有n个叶子结点(每个结点的权值为w_i) 的二叉树不止一棵，但在所有的这些二叉树中，必定存在一棵WPL值最小的树，称这棵树为Huffman树(或称最优树) 。

在许多判定问题时，利用Huffman树可以得到最佳判断算法。

如图6-24是权值分别为2、3、6、7，具有4个叶子结点的二叉树，它们的带权路径长度分别为：

(a) WPL=2×2+3×2+6×2+7×2=36；

(b) WPL=2×1+3×2+6×3+7×3=47；

(c) WPL=7×1+6×2+2×3+3×3=34。

其中(c)的 WPL值最小，可以证明是Huffman树。

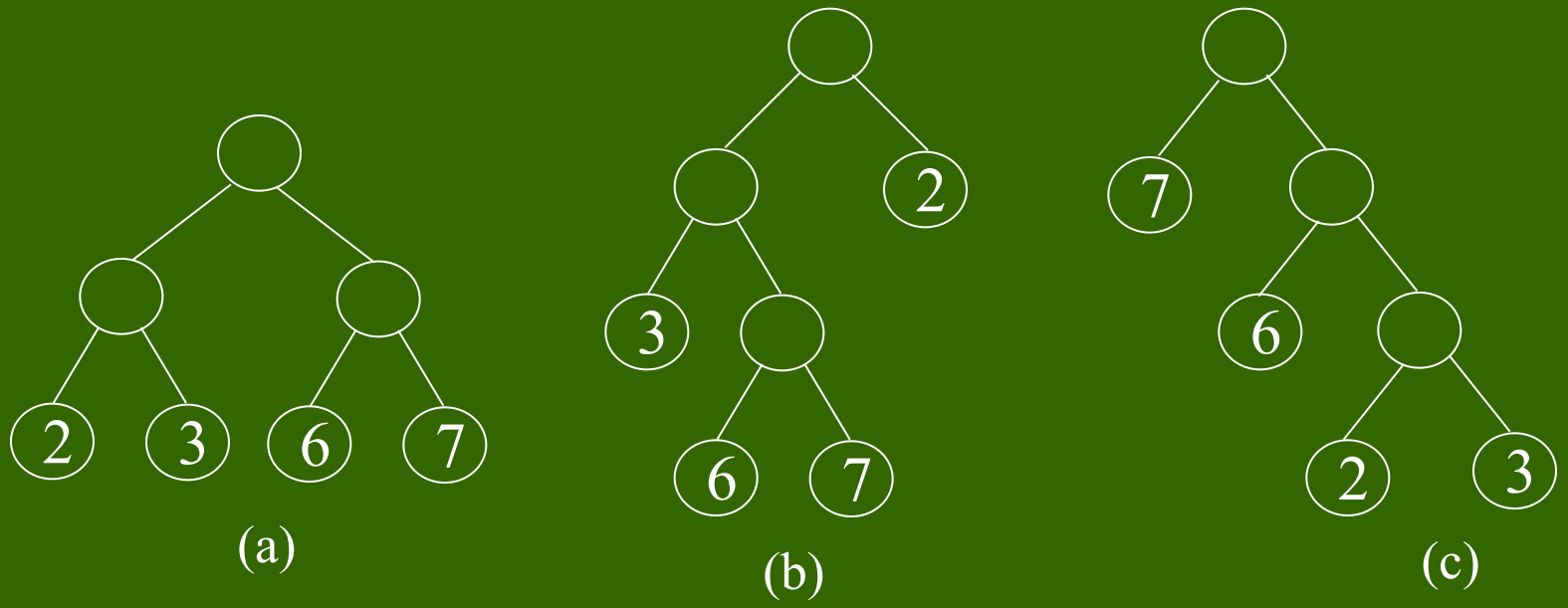


图6-24　具有相同叶子结点，不同带权路径长度的二叉树

2 Huffman树的构造

① 根据n个权值$\{w_1, w_2, \cdots, w_n\}$，构造成n棵二叉树的集合$F=\{T_1, T_2, \cdots, T_n\}$，其中每棵二叉树只有一个权值为$w_i$的根结点，没有左、右子树；

② 在F中选取两棵根结点权值最小的树作为左、右子树构造一棵新的二叉树，且新的二叉树根结点权值为其左、右子树根结点的权值之和；

③ 在F中删除这两棵树，同时将新得到的树加入F中；

④ 重复②、③，直到F只含一颗树为止。

构造Huffman树时，为了规范，规定$F=\{T_1,T_2,\cdots,T_n\}$中权值小的二叉树作为新构造的二叉树的左子树，权值大的二叉树作为新构造的二叉树的右子树；在取值相等时，深度小的二叉树作为新构造的二叉树的左子树，深度大的二叉树作为新构造的二叉树的右子树。

图6-25是权值集合W={8, 3, 4, 6, 5, 5}构造Huffman树的过程。所构造的Huffman树的WPL是：

WPL=6×2+3×3+4×3+8×2+5×3+5×3 =79

图6-25　Huffman树的构造过程

图6-25是权值集合W={8, 3, 4, 6, 5, 5}构造Huffman树的过程。所构造的Huffman树的WPL是：

WPL=6×2+3×3+4×3+8×2+5×3+5×3 =79

图6-25　Huffman树的构造过程

6.6.2 赫夫曼编码及其算法

1 Huffman编码

在电报收发等数据通讯中，常需要将传送的文字转换成由二进制字符0、1组成的字符串来传输。为了使收发的速度提高，就要求电文编码要尽可能地短。此外，要设计长短不等的编码，还必须保证任意字符的编码都不是另一个字符编码的前缀，这种编码称为前缀编码。

Huffman树可以用来构造编码长度不等且译码不产生二义性的编码。

设电文中的字符集$C=\{c_1,c_2,\cdots,c_i,\cdots,c_n\}$，各个字符出现的次数或频度集$W=\{w_1,w_2,\cdots,w_i,\cdots,w_n\}$。

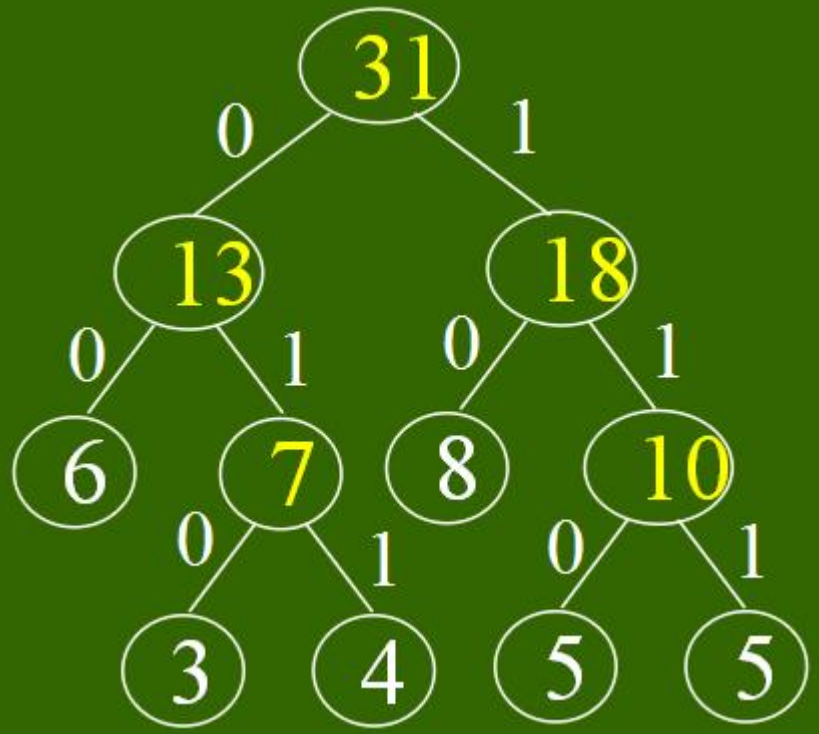
Huffman编码方法

以字符集C作为叶子结点，次数或频度集W作为结点的权值来构造 Huffman树。规定Huffman树中左分支代表“0”，右分支代表“1”。

从根结点到每个叶子结点所经历的路径分支上的“0”或“1”所组成的字符串，为该结点所对应的编码，称之为Huffman编码。

由于每个字符都是叶子结点，不可能出现在根结点到其它字符结点的路径上，所以一个字符的Huffman编码不可能是另一个字符的Huffman编码的前缀。

若字符集C={a, b, c, d, e, f}所对应的权值集合为W={8, 3, 4, 6, 5, 5}，如图6-25所示，则字符a,b, c,d, e,f所对应的Huffman编码分别是：00，010，011，10，110，111。



2 Huffman编码算法实

(1) 数据结构设计

Huffman树中没有度为1的结点的Huffman树共有2n-1个结点，可存储在大小为2n-1的一维数组中。实现编码的原因：

- 求编码需从叶子结点出发走一条从叶子到根的路径；

Weight	Parent	Lchild	Rchild

Weight：权值域；　Parent：双亲结点下标

Lchild, Rchild：分别标识左、右子树的下标

图6-26　Huffman编码的结点结构

◆ 译码需从根结点出发走一条到叶子结点的路径。

结点类型定义：

```
#define  MAX_NODE  200     /*   Max_Node>2n-1    */
typedef struct
    {     unsigned int Weight ;    /*  权值域    */
          unsinged int Parent , Lchild , Rchild ;
    } HTNode ;
```

(2) Huffman树的生成

算法实现

```
void Create_Huffman(unsigned n, HTNode HT[ ], unsigned m)
    /* 创建一棵叶子结点数为n的Huffman树 */
  { unsigned int w ;  int k , j ;
    for (k=1 ; k<m ; k++)
      {  if (k<=n)
           { printf("\n Please Input Weight : w=?");
             scanf("%d", &w) ;HT[k].weight=w ;
           }      /* 输入时，所有叶子结点都有权值 */
         else HT[k].weight=0; /*   非叶子结点没有权值 */
```

```
        HT[k].Parent=HT[k].Lchild=HT[k].Rchild=0 ;
    }     /* 初始化向量HT */
for (k=n+1; k<m ; k++)
    {   unsigned w1=32767 , w2=w1 ;
            /*  w1 , w2分别保存权值最小的两个权值  */
        int  p1=0 , p2=0 ;
            /*  p1 , p2保存两个最小权值的下标  */
        for (j=1 ; j<=k-1 ; j++)
            {   if (HT[k].Parent==0)    /* 尚未合并 */
                    {   if (HT[j].Weight<w1)
                                {   w2=w1 ; p2=p1 ;
                                    w1=HT[j].Weight ; p1=j ;
                                }
```

```
                else if (HT[j].Weight<w2)
                        {  w2=HT[j].Weight ;  p2=j ;  }
            }   /*   找到权值最小的两个值及其下标  */
        }
    HT[k].Lchild=p1 ; HT[k].Rchild=p2 ;
    HT[k].weight=w1+w2 ;
    HT[p1].Parent=k ; HT[p2].Parent=k ;
  }
}
```

说明：生成Huffman树后，树的根结点的下标是2n-1，即m-1 。

(3) Huffman编码算法

根据出现频度(权值)Weight，对叶子结点的Huffman编码有两种方式：

① 从叶子结点到根逆向处理，求得每个叶子结点对应字符的Huffman编码。

② 从根结点开始遍历整棵二叉树，求得每个叶子结点对应字符的Huffman编码。

由Huffman树的生成知，n个叶子结点的树共有2n-1个结点，叶子结点存储在数组HT中的下标值为1到n。

① 编码是叶子结点的编码，只需对数组HT[1…n]的n个权值进行编码；

② 每个字符的编码不同，但编码的最大长度是n。

求编码时先设一个通用的指向字符的指针变量，求得编码后再复制。

算法实现

```
void Huff_coding(unsigned n , Hnode HT[] , unsigned m)
     /*  m应为n+1,编码的最大长度n加1  */
  {  int  k , sp ,fp ;
     char *cd , *HC[m] ;
     cd=(char *)malloc(m*sizeof(char)) ;
        /*  动态分配求编码的工作空间  */
     cd[n]='\0'          /*  编码的结束标志  */
     for (k=1 ; k<n+1 ; k++)      /*  逐个求字符的编码    */
        {  sp=n ; p=k ; fp=HT[k].parent ;
```

```
        for (  ; fp!=0 ;p=fp , fp=HT[p].parent)
              /*  从叶子结点到根逆向求编码  */
            if  (HT[fp].Lchild==p)  cd[--sp]='0' ;
            else  cd[--sp]='1' ;
        HC[k]=(char *)malloc((n-sp)*sizeof(char)) ;
              /*  为第k个字符分配保存编码的空间 */
        trcpy(HC[k] , &cd[sp]) ;
    }
  free(cd) ;
}
```

习 题 六

(1) 假设在树中， 结点x是结点y的双亲时，用(x,y)来表示树边。已知一棵树的树边集合为 { (e,i), (b,e), (b,d), (a,b), (g,j), (c,g), (c,f), (h,l), (c,h), (a,c) }，用树型表示法表示该树，并回答下列问题：

① 哪个是根结点? 哪些是叶子结点? 哪个是g的双亲? 哪些是g的祖先? 哪些是g的孩子? 那些是e的子孙? 哪些是e的兄弟? 哪些是f的兄弟?

② b和n的层次各是多少? 树的深度是多少? 以结点c为根的子树的深度是多少?

(2) 一棵深度为h的满k叉树有如下性质： 第h层上的结点都是叶子结点，其余各层上每个结点都有k棵非空子树。 如果按层次顺序(同层自左至右)从1开始对全部结点编号，问：

① 各层的结点数是多少?

② 编号为i的结点的双亲结点(若存在)的编号是多少?

③ 编号为i的结点的第j个孩子结点(若存在)的编号是多少?

④ 编号为i的结点的有右兄弟的条件是什么? 其右兄弟的编号是多少?

(3) 设有如图6-27所示的二叉树。

① 分别用顺序存储方法和链接存储方法画出该二叉树的存储结构。

② 写出该二叉树的先序、中序、后序遍历序列。

(4) 已知一棵二叉树的先序遍历序列和中序遍历序列分别为ABDGHCEFI和GDHBAECIF，请画出这棵二叉树，然后给出该树的后序遍历序列。

(5) 设一棵二叉树的中序遍历序列和后序遍历序列分别为BDCEAFHG和DECBHGFA，请画出这棵二叉树，然后给出该树的先序序列。

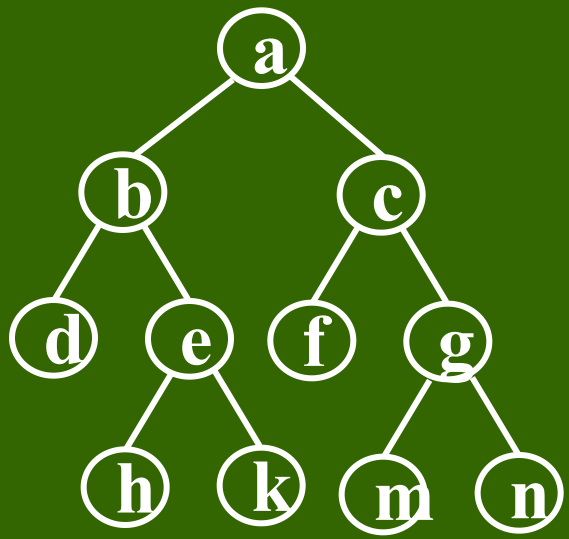


图6-27 二叉树

(6) 已知一棵二叉树的中序遍历序列和后序遍历序列分别为dgbaekchif和gdbkeihfca，请画出这棵二叉树对应的中序线索树和后序线索树。

(7) 以二叉链表为存储结构，请分别写出求二叉树的结点总数及叶子结点总数的算法。

(8) 设图6-27所示的二叉树是森林F所对应的二叉树，请画出森林F。

(9) 设有一棵树，如图6-28所示。

① 请分别用双亲表示法、孩子表示法、孩子兄弟表示法给出该树的存储结构。

② 请给出该树的先序遍历序列和后序遍历序列。

③ 请将这棵树转换成二叉树。

⑽ 设给定权值集合w={3,5,7,8,11,12}，请构造关于w的一棵huffman树，并求其加权路径长度WPL。

⑾ 假设用于通信的电文是由字符集{a, b, c, d, e, f, g, h}中的字符构成，这8个字符在电文中出现的概率分别为{0.07, 0.19, 0.02, 0.06, 0.32, 0.03, 0.21, 0.10}。

① 请画出对应的huffman树(按左子树根结点的权小于等于右子树根结点的权的次序构造)。

② 求出每个字符的huffman编码。

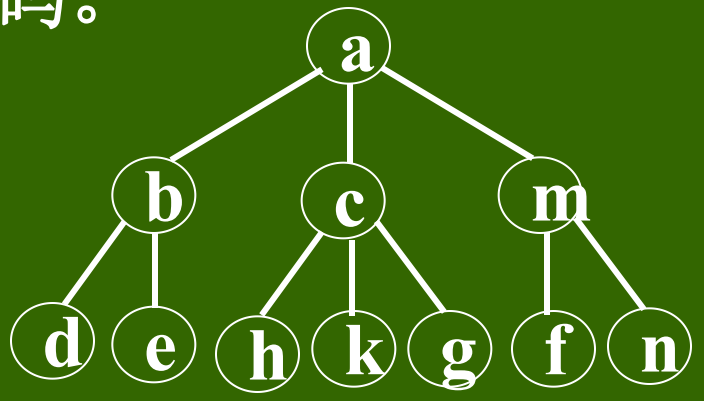


图6-28 一般的树