第5章 树

本章学习要点

(1) 熟悉树和二叉树的递归定义、有关的术语及基本概念。

(2) 熟练掌握二叉树的性质，了解相应的证明方法。

(3) 熟练掌握二叉树的两种存储方法、特点及适用范围。

(4) 遍历二叉树是二叉树的各种运算的基础，因此，不仅要熟练掌握各种次序的遍历算法，而且还要能灵活运用遍历算法，实现二叉树的其他各种运算。

(5) 了解二叉树的线索化及其实质，是建立结点及其在相应次序(先根、中根或后根)下的前驱和后继之间的直接联系，目的是加速遍历过程，迅速查找给定结点在指定次序下的前驱和后继。

(6) 熟练掌握树、森林与二叉树之间的转换方法。

(7) 了解最优二叉树的特性，掌握建立最优二叉树和哈夫曼编码的方法。

线性结构用于描述数据元素间的线性关系，然而实际应用中数据元素之间的关系错综复杂，很难完全用线性关系来描述。从本章开始将讨论非线性的数据结构。树是一种典型的非线性的数据结构，它描述了客观世界中事物之间的层次关系，这种结构有着广泛的应用，一切具有层次关系的问题都可以用树来描述。例如：家族的家谱、各种社会机构的组织都呈现出树形的层次结构；在操作系统的文件系统中，用树来表示目录结构；在编译程序中，用树来表示源程序的语法结构等。

5.1 树的概念与操作

5.1.1 树的概念

1. 树(tree)的定义

首先，可以注意到，自然界中的树有树根、树枝(不妨称为子树)和树叶，由此可以给出以下关于树的定义。

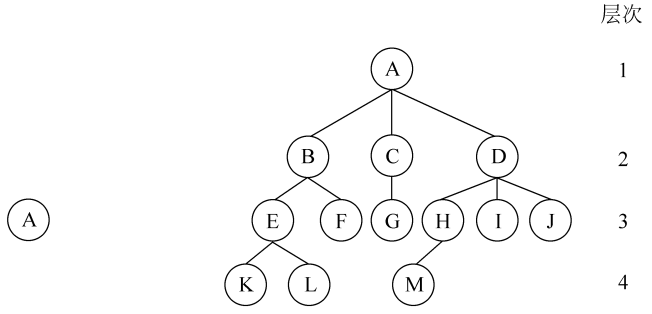
定义 5.1 树是由 $n(n\geqslant 0)$ 个结点组成的有限集合，当 $n=0$ 时称为空树；否则，在任一非空树中：

(1) 必有一个特定的称为根的结点；

(2) 剩下的结点被分成 $m\geqslant 0$ 个互不相交的集合 $T_1,T_2,\cdots,T_m$，而且这些集合中的每

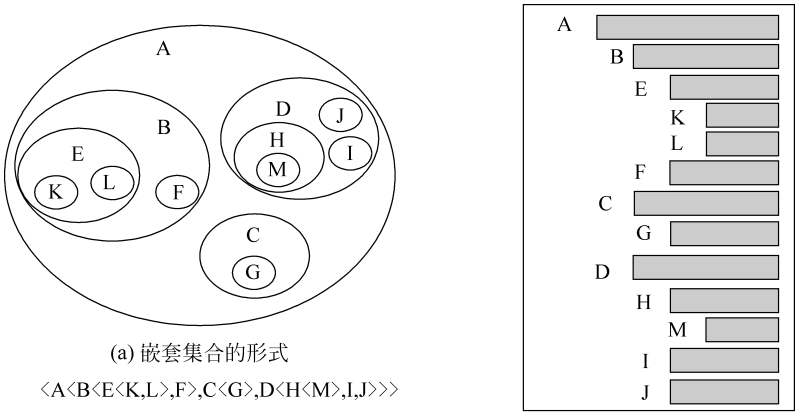
一个又都是树。树 $T_1, T_2, \cdots, T_m$ 被称作根的子树。

显然,这是一个递归的定义,因为它用树自身来定义树。树的定义显示了树的固有特性:树中的每一个结点都是该树中的某一棵子树的根。在定义中,特别强调子树的互不相交特性,即每个结点只属于一棵树(或子树),只有一个双亲。图 5.1(a)表示只有一个结点的树,图 5.1(b)是一般的树,有 13 个结点。树还可有其他的表示形式,图 5.2 所示为图 5.1(b)中树的各类表示,其中图 5.2(a)是以嵌套集合的形式表示的(即是一些集合的集合;对于其中任意两个集合,或者不相交,或者一个包含另一个);图 5.2(b)是以广义表的形式表示的;图 5.2(c)用的是凹入表示法(类似书的编目)。一般来说,分等级的分类方案都可用层次结构来表示,也就是说,都可表示为一个树结构。



(a) 只有一个结点的树　　(b) 一般的树

图 5.1　树的示例



(a) 嵌套集合的形式

<A<B<E<K,L>,F>,C<G>,D<H<M>,I,J>>>

(b) 广义表的形式　　(c) 凹入表示法

图 5.2　树的其他 3 种表示法

2. 树的基本术语

下面给出树结构中的一些基本术语。

树包含若干个结点以及若干指向其子树的分支。

结点拥有的子树数称为结点的度(degree)。例如在图 5.1(b)中 A 的度为 3,C 的度为 1,F 的度为 0。

度为0的结点称为叶子(leaf)或终端结点。图5.1(b)中的结点K、L、F、G、M、I、J都是树的叶子。度不为0的结点称为非终端结点或分支结点。除根结点之外,分支结点也称为内部结点。

树的度是树中各结点的度的最大值。如图5.1(b)中树的度为3。结点的子树的根称为该结点的孩子(child),相应地,该结点称为孩子的双亲(parent)。例如,在图5.1(b)所示的树中,D为A的子树的根,则D是A的孩子,而A则是D的双亲。

同一个双亲的孩子之间互称兄弟(sibling)。例如,在图5.1(b)所示的树中,H、I和J互为兄弟。

将这些关系进一步推广,可认为D是M的祖父。结点的祖先是从根到该结点所经分支上的所有结点,例如M的祖先为A、D、H。反之,以某结点为根的子树中的任一结点都称为该结点的子孙,如B的子孙为E、K、L和F。

结点的层次(level)从根开始定义起,根为第1层,根的孩子为第2层。若某结点在第C层,则其子树的根就在第$C+1$层。

其双亲在同一层的结点互称为堂兄弟。例如,结点G与E、F、H、I、J互为堂兄弟。

树中结点的最大层次称为树的深度(depth)或高度。图5.1(b)所示的树的深度为4。

如果将树中结点的各子树看成从左至右是有次序的(即不能互换),则称该树为有序树,否则称为无序树。在有序树中最左边的子树的根称为第一个孩子,最右边的子树的根称为最后一个孩子。

森林(forest)是$m(m\geqslant 0)$棵互不相交的树的集合。对树中每个结点而言,其子树的集合即为森林。

5.1.2 树的基本操作

树的基本操作有下列几种。

(1) 初始化操作:INITATE(T),置T为空树。

(2) 求根函数:ROOT(T)或ROOT(x),求树T的根或求结点x所在的树的根结点。若T是空或x不在任何一棵树上,则函数值为“空”。

(3) 求双亲函数:PARENT(T,x),求树T中结点x的双亲结点,若结点x是树T的根结点或结点x不在树T中,则函数值为“空”。

(4) 求孩子结点函数:CHILD(T,x,i),求树T中结点x的第i个孩子结点,若结点x是树T的叶子或无第i个孩子或结点x不在树T中,则函数值为“空”。

(5) 求右兄弟函数:RIGHT_SIBLING(T,x),求树T中结点x右边的兄弟,若结点x是其双亲的最右边的孩子结点或结点x不在树T中,则函数值为“空”。

(6) 建树函数:CRT_TREE(x,F),生成一棵以x结点为根,以森林F为子树森林的树。

(7) 插入子树操作:INS_CHILD(y,i,x),置以结点x为根的树是结点y的第i棵子树,若原树中无结点y或结点y的子树个数小于i-1,则为空操作。

(8) 删除子树操作:DEL_CHILD(x,y),删除结点x的第i棵子树,若无结点x或结点x的子树个数小于i,则为空操作。

(9) 遍历操作:TRAVERSE(T),按某个次序依次访问树中各个结点,并使每个结点只

被访问一次。

(10) 清除结构操作 CLEAR(T),将树 T 置为空树。

5.2 二叉树

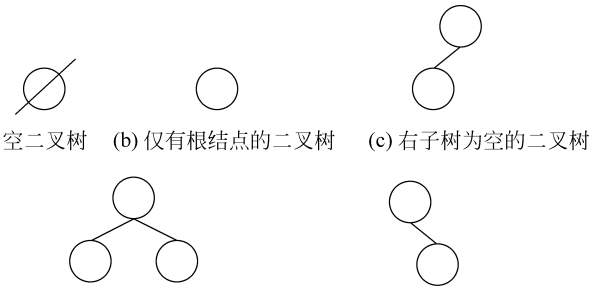
树形结构和自然界的树一样具有各种各样的形态,这增加了研究树形结构的问题的复杂性。为此,首先定义并研究规范化的二叉树,讨论二叉树的性质、存储结构和运算,然后给出二叉树与一般树之间的转换规则,这样就解决了树的存储结构及其运算复杂性的问题。

5.2.1 二叉树的概念

1. 二叉树(binary tree)的定义

定义 5.2 二叉树是结点的有限集合,这个集合或者是空的,或者由一个根结点或两棵互不相交的称为左子树的和右子树的二叉树组成。

这个递归定义表明二叉树或者为空,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。由于这两棵子树也是二叉树,则由二叉树的定义,它们也可以是空树。由此,二叉树可以有五种基本形态,如图 5.3 所示。



(a) 空二叉树 (b) 仅有根结点的二叉树 (c) 右子树为空的二叉树

(d) 左、右子树均非空的二叉树 (e) 左子树为空的二叉树

图 5.3 二叉树的五种基本形态

二叉树的特点是:树中的每个结点最多只能有两棵子树,即树中任何结点的度数不大于 2;二叉树的子树有左、右之分,而且,子树的左、右次序是重要的,即使在只有一棵子树的情况下,也应分清是左子树还是右子树。

前面引入的有关树的术语也都适用于二叉树。

为了说明二叉树的性质,下面先给出满二叉树和完全二叉树的定义。

定义 5.3 一棵深度为 k 的满二叉树,是有 2^k-1 个结点的深度为 k 的二叉树。

2^k-1 个结点是二叉树所具有的最大结点个数。例如,图 5.4 所示为一棵深度为 4 的满二叉树。为便于访问满二叉树的结点,对满二叉树从第 1 层的结点(即根)开始,自上而下,从左到右,按顺序给结点编号,便得到满二叉树的一个顺序表。

定义 5.4 一棵具有 n 个结点,深度为 k 的二叉树,当且仅当其所有结点对应于深度为 k 的满二叉树中编号由 $1\sim n$ 的那些结点时,该二叉树便是完全二叉树。

若用一个一维数组 tree 来表示完全二叉树，则其编号为 i 的结点对应于数组元素 tree[i]。图 5.5 所示为一棵完全二叉树。

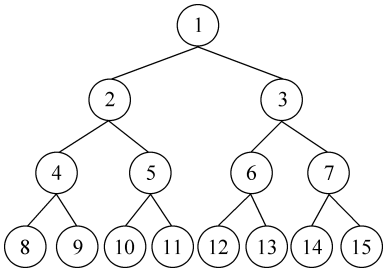


图 5.4 满二叉树

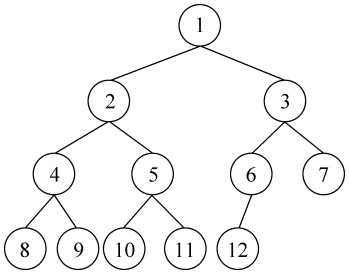


图 5.5 完全二叉树

2. 二叉树的基本操作

与树的基本操作相类似，二叉树有如下一些基本操作。

(1) 初始化操作：INITATE(BT)，置 BT 为空树。

(2) 求根函数：ROOT(BT)或 ROOT(x)，求二叉树 BT 的根结点或求结点 x 所在二叉树的根结点，若 BT 是空树或 x 不在任何二叉树上，则函数值为“空”。

(3) 求双亲函数：PARENT(BT,x)，求二叉树 BT 中结点 x 的双亲结点，若结点 x 是二叉树 BT 的根结点或二叉树 BT 中无 x 结点，则函数值为“空”。

(4) 求孩子结点函数：LCHILD(BT,x)和 RCHILD(BT,x)，分别求二叉树 BT 中结点 x 的左孩子和右孩子结点，若结点 x 为叶子结点或不在二叉树 BT 中，则函数值为“空”。

(5) 求兄弟函数：LSIBLING(BT,x)和 RSIBLING(BT,x)，分别求二叉树 BT 中结点 x 的左兄弟和右兄弟结点；若结点 x 是根结点，或不在 BT 中，或是其双亲的左/右子树根，则函数值为“空”。

(6) 建树操作：CRT_BT(x,LBT,RBT)，生成一棵以结点 x 为根，以二叉树 LBT 和 RBT 为左、右子树的二叉树。

(7) 插入子树操作：INS_LCHILD(BT,y,x)和 INS_RCHILD(BT,y,x)，将以结点 x 为根且右子树为空的二叉树分别置为二叉树 BT 中结点 y 的左子树和右子树，若结点 y 有左子树/右子树，则插入后 y 的左子树/右子树成为结点 x 的左子树/右子树。

(8) 删除子树操作：DEL_LCHILD(BT,x)和 DEL_RCHILD(BT,x)，分别删除二叉树 BT 中以结点 x 为根的左子树或右子树，若 x 无左子树或右子树，则为空操作。

(9) 遍历操作：TRAVERSE(BT)，按某个次序依次访问二叉树中各个结点，并使每个结点只被访问一次。

(10) 清除结构操作：CLEAR(BT)，将二叉树 BT 置为空树。

在已知二叉树的逻辑结构和运算后就可以定义二叉树的抽象数据类型。ADT5.1 是二叉树的抽象数据类型描述，其中只包含最常见的二叉树运算。

ADT5.1 二叉树 ADT

ADT BTree{

数据对象：

$D=\{a_i \mid a_i \in$ 元素集合,$i=1,2,\cdots,n,n\geqslant 0\}$

数据关系 R:

若 $D=\varnothing$,则 $R=\varnothing$,称 BTree 为空二叉树。

若 $D\neq\varnothing$,则 $R=\{H\}$,H 是如下二元关系。

(1) 在 D 中存在唯一的称为根的数据元素 root,它在关系 H 下无前驱。

(2) 若 $D-\{\text{root}\}\neq\phi$,则存在 $D-\{\text{root}\}=\{D_l,D_r\}$,且 $D_l\cap D_r=\phi$。

(3) 若 $D_l\neq\phi$,则 D_l 中存在唯一的元素 x_l,$\langle\text{root},x_l\rangle\in H$,且存在 D_l 上的关系 $H_l\subset H$;若 $D_r\neq\phi$,则 D_r 中存在唯一的元素 x_r,$\langle\text{root},x_r\rangle\in H$,且存在 D_r 上的关系 $H_r\subset H$;$H=\{\langle\text{root},x_l\rangle,\langle\text{root},x_r\rangle,H_l,H_r\}$。

(4) $(D_l,\{H_l\})$是一棵符合本定义的二叉树,称为根的左子树。$(D_r,\{H_r\})$是一棵符合本定义的二叉树,称为根的右子树。

基本操作:

creat(): 创建一个空二叉树。

destroy(): 撤销一个二叉树。

isempty(): 若二叉树空,则返回 1; 否则返回 0。

clear(): 移去所有结点,成为空二叉树。

root(x): 若二叉树非空,则 x 为根的值,并返回 1,否则返回 0。

maketree(x,left,right): 构造一棵二叉树,根的值为 x,以 left 和 right 为左、右子树。

breaktree(x,left,right): 拆分二叉树为三部分,x 为根的值,以 left 和 right 分别为原树的左右子树。

preorder(visit): 使用函数 visit()访问结点,先根遍历二叉树。

inorder(visit): 使用函数 visit()访问结点,中根遍历二叉树。

postorder(visit): 使用函数 visit()访问结点,后根遍历二叉树。

}ADT BTree

5.2.2 二叉树的性质

二叉树具有下列重要性质。

性质 5.1 在二叉树的第 i 层上至多有 2^{i-1} 个结点($i\geqslant 1$)。

利用归纳法容易证得此性质。

当 $i=1$ 时,只有一个根结点。显然,$2^{i-1}=2^0=1$ 是对的。

现假设对所有的 j,$1\leqslant j<i$,命题成立,即第 j 层上至多有 2^{j-1} 个结点。那么可以证明 $j=i$ 时命题成立。

由此归纳假设:第 $i-1$ 层上至多有 2^{i-2} 个结点。由于二叉树的每个结点的度至多为 2,故在第 i 层上的最大结点数为第 $i-1$ 层上的最大结点数的 2 倍,即 $2*2^{i-2}=2^{i-1}$。

性质 5.2 深度为 $k(k\geqslant 1)$的二叉树至多有 2^k-1 个结点。

由性质 5.1 可见,深度为 k 的二叉树的最大结点数为

$$\sum_{i=1}^{k}(\text{第 } i \text{ 层上的最大结点数})=\sum_{i=1}^{k}2^{i-1}=2^k-1$$

性质 5.3 对任何一棵二叉树 T，如果其终端结点数为 n_0，度为 2 的结点数为 n_2，则 $n_0=n_2+1$。

设 n_1 为二叉树 T 中度为 1 的结点数。因为二叉树中所有结点的度均小于或等于 2，所以其结点总数为

$$n=n_0+n_1+n_2 \tag{5.1}$$

再看二叉树中的分支数。除根结点外，其余结点都有一个分支进入，设 B 为分支数，则 $n=B+1$。由于这些分支是由度为 1 或 2 的结点引出的，所以又有 $B=n_1+2n_2$。于是得

$$n=n_1+2n_2+1 \tag{5.2}$$

由式(5.1)和式(5.2)可得

$$n_0=n_2+1$$

性质 5.4 具有 n 个结点的完全二叉树的深度为 $\lfloor \log_2 n \rfloor+1$。

证明：假设深度为 k，则根据性质 5.2 和完全二叉树的定义有

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

或

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

于是

$$k-1\leqslant \mathrm{lb}n<k$$

因为 k 是整数，所以

$$k=\lfloor \mathrm{lb}n \rfloor+1$$

性质 5.5 如果对一棵有 n 个结点的完全二叉树(其深度为 $\lfloor \mathrm{lb}n \rfloor+1$)的结点按层序号编号(从第 1 层到 $\lfloor \mathrm{lb}n \rfloor+1$ 层，每层从左到右)，则对任一结点 $i(1\leqslant i\leqslant n)$，有

(1) 如果 $i=1$，则结点 i 是二叉树的根，无双亲；如果 $i>1$，则双亲是结点 $i/2$。

(2) 如果 $2i>n$，则结点 i 无左孩子(结点 i 为叶子结点)；否则其左孩子是结点 $2i$。

(3) 如果 $2i+1>n$，则结点 i 无右孩子；否则其右孩子是结点 $2i+1$。

只要先证明(2)和(3)，便可从(2)和(3)导出(1)。

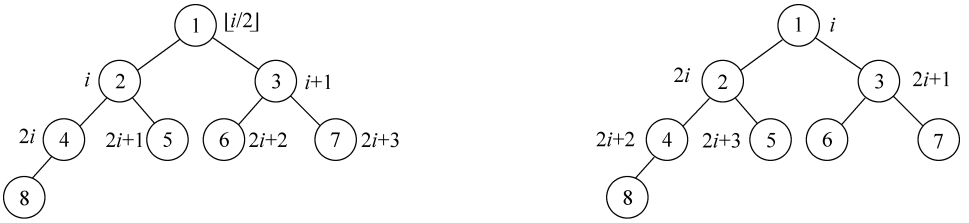
对于 $i=1$，由完全二叉树的定义，其左孩子是结点 2，若 $2>n$，即不存在结点，此时，结点 i 无左孩子。结点 1 的右孩子也只能是结点 3，若结点 3 不存在，即 $3>n$，此时，结点 i 无右孩子。

对于 $i>1$，可分两种情况讨论。

(1) 设第 $j(1\leqslant j\leqslant \lfloor \mathrm{lb}n \rfloor)$层的第一个结点的编号为 i(由二叉树的定义和性质 5.2 可知 $i=2^{j-1}$)，则左孩子必为第 $j+1$ 层的第一个结点，其编号为 $2^j=2(2^{j-1})=2i$，若 $2i>n$，则无左孩子；其右孩子必为第 $j+1$ 层的第二个结点，其编号为 $2i+1$，若 $2i+1>n$，则无右孩子。

(2) 假设第 $j(1\leqslant j\leqslant \lfloor \mathrm{lb}n \rfloor)$层上某个结点的编号为 $i(2^{j-1}\leqslant i<2^j-1)$，且 $2i+1<n$，则左孩子为 $2i$，右孩子为 $2i+1$。又编号为 $i+1$ 的结点是编号为 i 的结点的右兄弟或者堂兄弟，若它有左孩子，则编号必为 $2i+2=2(i+1)$，若它有右孩子，则编号必为 $2i+3=2(i+1)+1$。

图 5.6 所示为完全二叉树中结点及其左、右孩子结点间的关系。



(a) 结点i和$i+1$在同一层上　　(b) 结点i和$i+1$不在同一层上

图 5.6　完全二叉树中结点及其左、右孩子结点间的关系

5.2.3　二叉树的存储结构及其实现

1. 顺序存储结构

用一组连续的存储单元存储二叉树的数据元素，将二叉树中编号为 i 的结点的数据元素存放在分量 tree[$i-1$]中，如图 5.7 所示。对于图 5.5 中的完全二叉树，可以用向量(一维数组)tree[0..11]作为它的相应存储结构；对于如图 5.8 所示的一般二叉树，其顺序存储结构如图 5.9 所示。

1	2	3	4	5	6	7	8	9	10	11	12

图 5.7　完全二叉树的顺序存储结构

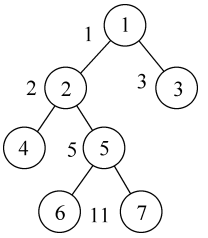


图 5.8　一般二叉树

1	2	3	4	5	0	0	0	0	6	7

图 5.9　一般二叉树的顺序存储结构

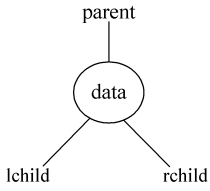
根据完全二叉树的特性，结点在向量中的相对位置蕴含着结点间的关系，如 tree[i]的双亲为 tree[$(i+1)/2-1$]，而其左、右孩子则分别为 tree[$2i$]和 tree[$2i+1$]中。显然，这种顺序存储结构仅适于完全二叉树，因为在顺序存储结构中，仅以结点在向量中的相对位置表示结点之间的关系，因此，一般的二叉树也必须应按完全二叉树的形式来存储，这就有可能造成存储空间的浪费。如图 5.8 所示的一般二叉树，其存储结构如图 5.9 所示，图中“0”表示不存在此结点。在最坏的情况下，一个深度为 k 且只有 k 个结点的单支树(树中无度为 2 的结点)却需 2^k-1 个存储分量。

2. 链式存储结构

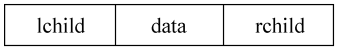
由二叉树的定义可知，二叉树的结点由一个数据元素和分别指向其左、右子树的两个分支构成，如图 5.10(a)所示。也就是说，二叉树的链表中的结点至少包含三个域：数据域和左、右指针域，如图 5.10(b)所示。但是，设计不同的结点结构可构成不同形式的链式存储

结构。有时，为了便于找到结点的双亲，还可以在结构中增加一个指向其双亲结点的指针域，如图 5.10(c)所示。利用这两种结点结构所得二叉树的存储结构分别称为二叉链表和三叉链表，如图 5.11 所示。链表的头指针指向二叉树的根结点。

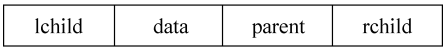
在不同的存储结构中实现二叉树的操作方法也不同，如查找结点 x 的双亲 parent(tree,x)，在三叉链表中很容易实现，而在二叉链表中则需从根结点出发巡查。由此，在具体应用中采用什么存储结构，除考虑二叉树的形态之外还应考虑需要进行何种操作。



(a) 二叉树的结点

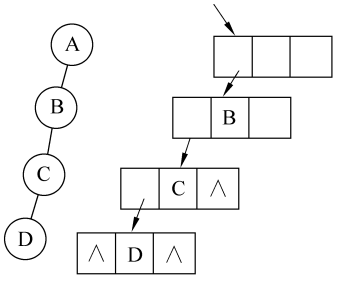


(b) 含有两个指针域的结点结构

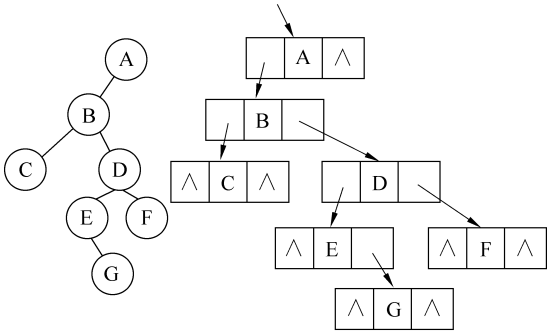


(c) 含有三个指针域的结点结构

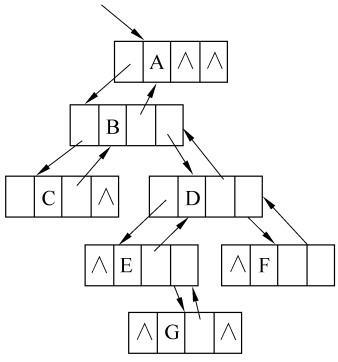
图 5.10 二叉树的结点及其存储结构



(a) 二叉树及其二叉链表示例1



(b) 二叉树及其二叉链表示例2



(c) 示例2中二叉树的三叉链表

图 5.11 二叉树的链式存储结构

5.3 二叉树的遍历

5.3.1 递归的遍历算法

遍历(traversal)是树的一种最基本的运算。所谓遍历二叉树,就是按一定的规则和次序走遍二叉树的所有结点,使得每个结点都被访问一次,而且只被访问一次。遍历二叉树的目的在于得到二叉树中各结点的一种线性序列,使非线性的二叉树线性化,从而简化有关的运算和处理。对于线性结构,遍历的问题十分简单,因其结构本身就是线性的。但二叉树是非线性的,要得到树中各结点的一种线性序列就不那么容易。因为从二叉树的任意结点出发,既可向左走,也可向右走,存在两种可能。所以,必须为遍历确定一个完整而有规则的走法,以便按同样的方法处理每个结点及其子树。

二叉树的基本结构形态如图 5.3 所示,如果用 L、D、R 分别表示遍历左子树、访问根结点、遍历右子树,并遵循先左后右的规则,那么,遍历二叉树可以有三种不同的走法:DLR、LDR、LRD。分别称为先根遍历、中根遍历、后根遍历。三种走法的定义如下。

1. 先根遍历(DLR)

若二叉树为空,则返回,否则依次执行以下操作:
访问根结点;
按先根遍历左子树;
按先根遍历右子树;
返回。

2. 中根遍历(LDR)

若二叉树为空,则返回,否则依次执行以下操作:
按中根遍历左子树;
访问根结点;
按中根遍历右子树;
返回。

3. 后根遍历(LRD)

若二叉树为空,则返回,否则依次执行以下操作:
按后根遍历左子树;
按后根遍历右子树;
访问根结点;
返回。
根据上述描述,对于图 5.5 所示的二叉树:
按先根遍历,得到的结点序列是 1,2,4,8,9,5,10,11,3,6,12,7;
按中根遍历,得到的结点序列是 8,4,9,2,10,5,11,1,12,6,3,7;

按后根遍历,得到的结点序列是 8,9,4,10,11,5,2,12,6,7,3,1。

对于图 5.4 所示的二叉树:

按先根遍历得到的结点序列是 1,2,4,8,9,5,10,11,3,6,12,13,7,14,15;

按中根遍历得到的结点序列是 8,4,9,2,10,5,11,1,12,6,13,3,14,7,15;

按后根遍历得到的结点序列是 8,9,4,10,11,5,2,12,13,6,14,15,7,3,1。

显然,先根遍历、中根遍历和后根遍历这些术语本身,就反映着根结点相对于其子树的位置关系。

遍历算法的语言描述形式随存储结构的不同而不同。若定义二叉树的存储结构为如下说明的二叉链表:

```
typedef int datatype;
struct bnodept
{
    datatype data;
    struct bnodept * lchild, * rchild;
};
typedef struct bnodept * bitreptr;
```

则三种遍历的递归算法如下。

算法 5.1 二叉树的先根遍历递归算法。

```
//按先根遍历二叉树 t,t 的每个根结点有三个域:lchild,data,rchild
void preorder(bitreptr t)
{
    if(t)                          //为非空二叉树
    {
        visit(t->data);            //访问根结点
        preorder(t->lchild);       //先根遍历左子树
        preorder(t->rchild);       //先根遍历右子树
    }
}
```

算法 5.2 二叉树的中根遍历递归算法。

```
//按中根遍历二叉树 t,t 的每个结点有三个域:lchild,data,rchild
void inorder(bitreptr t)
{
    if(t)
    {
        inorder(t->lchild);
        visit(t->data);
        inorder(t->rchild);
    }
}
```

算法 5.3 二叉树的后根遍历递归算法。

```
//按后根遍历二叉树 t,t 的每个结点有三个域:lchild,data,rchild
void postorder(bitreptr t)
{
```

```
    if(t)
    {
        postorder(t->lchild);
        postorder(t->rchild);
        visit(t->data);
    }
}
```

例如,图 5.12 所示的二叉树表示下述表达式

$$a+b*(c-d)-e/f$$

若先根遍历图 5.12 所示的二叉树,按访问结点的先后次序将结点排列起来,可得二叉树的先根序列为:

$$-+a*b-cd/ef \qquad (5.3)$$

类似地,中根遍历图 5.12 所示的二叉树,可得此二叉树的中根序列为:

$$a+b*c-d-e/f \qquad (5.4)$$

后根遍历图 5.12 所示的二叉树的序列为:

$$abcd-*+ef/- \qquad (5.5)$$

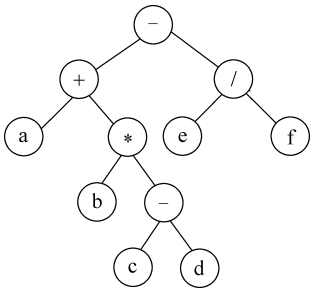
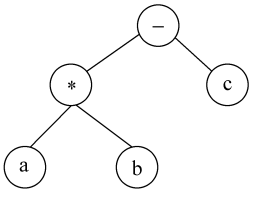


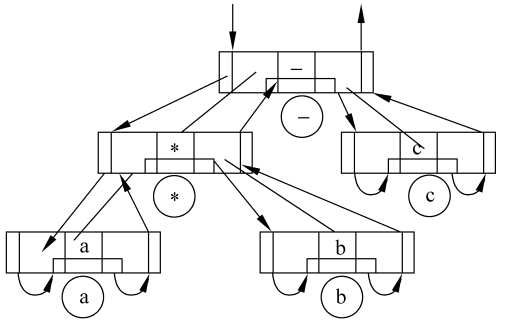
图 5.12 表达式(a+b*(c-d)-e/f)的二叉树

从表达式来看,以上三个序列恰好为表达式的前缀表示(波兰式)、中缀表示和后缀表示(逆波兰式)。

从上述二叉树遍历的定义可知,三种遍历算法的不同之处仅在于访问根结点和遍历左、右子树的先后次序不同。如果在算法中暂且抹去与递归无关的 visit 语句,则三个遍历算法完全相同。因此,从递归执行的过程角度来看,先根、中根和后根遍历也是完全相同的。图 5.13(b)中用带箭头的虚线表示了这三种遍历算法的递归执行过程。其中向下的箭头表示更深一层的递归调用,向上的箭头表示从递归调用退出返回,虚线旁的字符表示了中根遍历二叉树过程中访问结点时输出的信息。由于中根遍历中访问根结点是在遍历左子树之后、遍历右子树之前进行的,则带圆形的字符标在向左递归返回和向右调用之间。由此,只要沿虚线从 1 出发到 2 结束,将沿途所见的圆形内的字符记下,便得到二叉树的中根序列,例如,从图 5.13(b)可得图 5.13(a)所示表达式的中根序列为:a*b-c。



(a) 表达式(a*b-c)的二叉树表示



(b) 遍历的递归执行过程

图 5.13 三种遍历过程示意图

执行一个递归程序,需要借助栈的作用,因此可以直接使用栈,把上面的递归算法改写成一个等价的非递归算法。

在遍历二叉树的过程中，通过根结点可以立刻找到它的左孩子（即左子树的根结点）和右孩子（即右子树的根结点），但不能直接从左孩子或右孩子到达它的双亲，除非重新从二叉树的根开始扫描。对于前根遍历二叉树而言，在访问根结点之后，可以直接到达左子树进行遍历；在左子树遍历完毕之后，还必须设法从左子树返回到根结点，再到达它的右子树进行遍历。因此，在从根结点走向左子树之前，必须将根结点的指针送入一个栈中暂存起来。这样，在左子树遍历完毕之后，再从栈中取回根结点的指针，便得到了根结点的地址，再走向右子树进行遍历。

算法 5.4 前根遍历二叉树的非递归算法。

```
void preorder(bitreptr t)
{
    bitreptr stack[MAX + 1];                //顺序栈
    int top = 0;                            //栈顶指针
    do
    {
        while(t)
        {
            visit(t -> data);               //访问根结点
            if(top == MAX)                  //栈已满
            {
                printf("stack full");
                return;                     //不能再遍历下去
            }
            stack[++top] = t;               //根指针进栈
            t = t -> lchild;                //移向左子树
        }
        if(top!= 0)                         //栈中还有根指针
        {
            t = stack[top -- ];             //取出根指针
            t = t -> rchild;                //移向右子树
        }
    }while(top!= 0||t!= NULL);              //栈非空或为非空子树
}
```

对二叉树进行遍历的搜索路径除了按先根、中根或后根外，还可以从上到下、从左到右按层次进行。

显然，遍历二叉树的算法中的基本操作是访问根结点，无论按哪一种次序进行遍历，对含 n 个结点的二叉树，其时间复杂度均为 $O(n)$。所需辅助空间为遍历过程中栈的最大容量，即树的深度，最坏情况下为 n，则空间复杂度也为 $O(n)$。遍历时也可采用二叉树的其他存储结构，如带标志域的三叉链表，此时因存储结构中已存储遍历所需的足够信息，则遍历过程中不需另设栈。另外，采用带标志域的二叉链表做存储结构，并在遍历过程中利用指针域暂存遍历路径，也可省略栈的空间，但这样做将使时间上有很大损失。

5.3.2 二叉树遍历操作应用举例

遍历是二叉树各种操作的基础，可以在遍历过程中对结点进行各种操作，如对于一棵已知二叉树可求结点的双亲、求结点的孩子结点、判定结点所在层次等，反之，也可以在遍历过

程中生成结点,建立二叉树的存储结构。

(1) 求二叉树中以值为 x 的结点为根的子树的深度。

算法 5.5 求二叉树中值为 x 的结点为根的子树的深度算法。

```
//求子树深度的递归算法
int Get_Depth(bitreptr T)
{
    int m,n;
    if(!T) return 0;                          //递归函数有返回值时注意对每个分支赋值
    else
    {
        m = Get_Depth(T->lchild);
        n = Get_Depth(T->rchild);
        return (m>n?m:n) + 1;
    }
}
//求二叉树中以值为 x 的结点为根的子树深度
void Get_Sub_Depth(bitreptr T, datatype x)
{
    if(T->data == x)
    {
        printf("%d\n",Get_Depth(T));      //找到了值为 x 的结点,求其深度
        exit(1);
    }
    else
    {
        if(T->lchild)
            Get_Sub_Depth(T->lchild,x);
        if(T->rchild)
            Get_Sub_Depth(T->rchild,x); //在左、右子树中继续寻找
    }
}
```

(2) 在二叉树中求指定结点的层数。

算法 5.6 在二叉树中求指定结点的层数算法。

```
//在二叉树 root 中求值为 ch 的结点所在的层数
int preorder(bitreptr root,datatype ch)
{
    int lev,m,n;
    if(root == NULL)
        lev = 0;                              //空树
    else if (root->data == ch)
        lev = 1;                              //ch 所在结点为根结点
        else
        {
            m = preorder(root->lchild,ch); //在左子树中查找 ch 所在结点
            n = preorder(root->rchild,ch); //在右子树中查找 ch 所在结点
            if (m == 0&&n == 0) lev = 0;       //在左、右子树中查找失败
            else lev = ((m>n)?m:n) + 1;       //在左子树或右子树中查找成功时,层数加 1
        }
    return(lev);
}
```

(3) 按先根序列建立二叉树的二叉链表。

对图 5.11(b)所示的二叉树,按下列次序顺序读入字符,其中#作为结束标志。

A B C # # D E # G # # F # # #

算法 5.7 按先根序列建立二叉树的二叉链表算法。

```
//按先根序列建立二叉树的二叉链表.函数的返回值指向根结点
bitreptr crt_bt_pre()
{
    char ch;
    bitreptr bt;
    ch = getchar();                                    //从键盘上输入一个字符
    if (ch == '#') return(NULL);                       //#作为结束标志
    else
    {
        bt = (bitreptr)malloc(sizeof(struct bnodept));  //产生新结点
        bt -> data = ch;
        bt -> lchild = crt_bt_pre();
        bt -> rchild = crt_bt_pre();
        return (bt);
    }
}
```

(4) 求二叉树的叶子数。

可以将此问题视为一种特殊的遍历问题,这种遍历中“访问一个结点”的具体内容为判断该结点是不是叶子,若是则将叶子数加 1。显然可以采用任何遍历方法,这里用先根遍历。

算法 5.8 求二叉树的叶子数算法。

```
//先根遍历根指针为 root 的二叉树以计算其叶子数
int countleaf(bitreptr root)
{
    int i;
    if(root == NULL)
        i = 0;
    else if((root -> lchild == NULL)&&(root -> rchild == NULL))
      i = 1;
    else
      i = countleaf(root -> lchild) + countleaf(root -> rchild);
    return(i);
}
```

5.4 线索二叉树

5.4.1 线索二叉树的定义

当用二叉链表作为二叉树的存储结构时,由于每个结点中只有指向其左、右孩子结点的

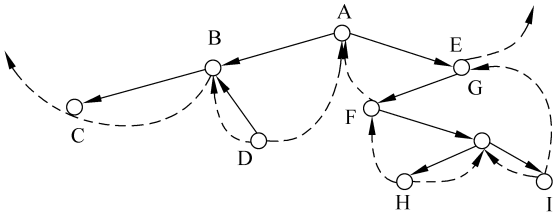
指针域,所以从任一结点出发只能直接找到该结点的左、右孩子,一般情况下无法直接找到该结点在某种遍历序列中的前驱和后继结点。为此,若在每个结点中增加两个指针域来存放遍历时得到的前驱和后继信息,则将大大降低存储空间的利用率。由于在 n 个结点的二叉链表中含有 $n+1$ 个空指针域,因此可以利用这些空指针域,存放指向结点在某种遍历次序下的前驱和后继结点的指针,这种附加的指针称为线索,加上了线索的二叉链表称为线索链表,相应的二叉树称为线索二叉树(threaded binary tree)。

为了区分一个结点的指针域是指向其孩子的指针,还是指向其前驱或后继的线索,可在每个结点中增加两个标志域,这样,线索链表中的结点结构为:

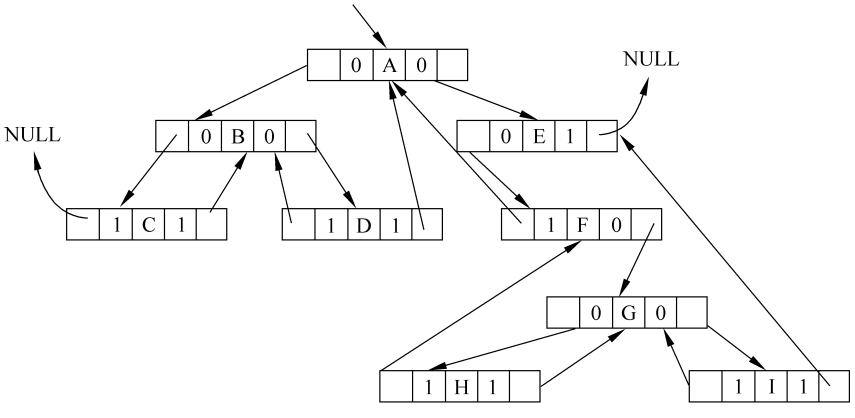
lchild	ltag	data	rtag	rchild

其中:左标志 ltag=0 表示 lchild 是指向结点的左孩子的指针;否则,为指向结点的前驱的左线索。右标志 rtag=0 表示 rchild 是指向结点的右孩子的指针;否则,为指向结点的后继的右线索。

如图 5.14(a)所示的中根线索二叉树,它的线索链表见图 5.14(b)。图中的实线表示指针,虚线表示线索。结点 C 的左线索为空,表示 C 是中根序列的开始结点,它没有前驱;结点 E 的右线索为空,表示 E 是中根序列的终端结点,它没有后继。显然在线索二叉树中,一个结点是叶子结点的充要条件是:它的左、右标志均是 1。



(a) 中根线索二叉树



(b) 中根线索链表

图 5.14 中根线索二叉树及其存储结构

将二叉树转换为线索二叉树的过程称为线索化。按某种次序将二叉树线索化,只要按该次序遍历二叉树,在遍历过程中用线索取代空指针即可。为此,附加一个指针 pre 始终指

向刚访问过的结点，而指针 p 指向当前正在访问的结点。显然结点 * pre 是结点 * p 的前驱，而 * p 是 * pre 的后继。下面给出将二叉树按中根线索化的算法。该算法与中根遍历算法类似，只需要将遍历算法中访问结点 * p 的操作具体化为在 * p 及其中根前驱 * pre(若 pre!＝NULL)之间建立线索的操作即可。显然 pre 的初值应为 NULL。

算法 5.9 二叉树中根线索化算法。

```
typedef int datatype;
typedef enum {link,thread} pointertag;                    //枚举值 link 和 thread 分别为 0,1
typedef struct node
{
  datatype data;
  pointertag ltag,rtag;                                   //左、右标志
  struct node * lchild, * rchild;
}binthrnode;
typedef binthrnode * binthrtree;
binthrnode * pre = NULL;                                  //全局变量
void in_thread(binthrtree p)
{
     if(p)                                                //p 非空时,当前访问结点是 * p
    {
        in_thread(p->lchild);                             //左子树线索化
       //以下直至右子树线索化之前相当于遍历算法中访问结点的操作
        p->ltag = (p->lchild)?link:thread;
        //左指针非空时左标志为 link(即为 0),否则为 thread(即 1)
      p->rtag = (p->rchild)?link:thread;
      if (pre)                                            //若 * p 的前驱 * pre 存在
        {  if(pre->rtag == thread)                        // * p 的前驱右标志为线索
             pre->rchild = p;                             //令 * pre 的右线索指向中根后继
           if(p->ltag == thread)                          // * p 的左标志为线索
             p->lchild = pre;                             //令 * p 的左线索指向中根前驱
        }
       pre = p;                                           //令 pre 时下一访问结点的中根前驱
       in_thread(p->rchild);
    }
}
```

显然，和中根遍历算法一样，递归过程中对每个结点仅做一次访问，因此对于 n 个结点的二叉树，算法的时间复杂度也为 $O(n)$。

类似地可得前根线索化和后根线索化算法。

5.4.2 线索二叉树的常用运算

下面介绍线索二叉树上两种常用的运算。

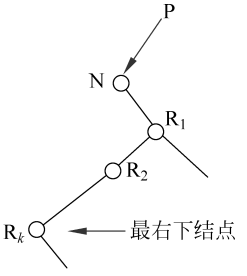
1. 查找某结点 * p 在指定次序下的前驱和后继结点

在中根线索二叉树中，查找结点 * p 的中根线索后继结点分以下两种情形。

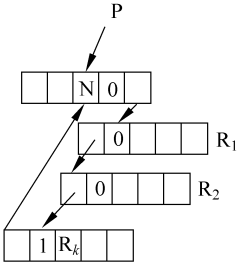
(1) 若 * p 的右子树为空(即 p-> rtag 为 thread)，则 p-> rchild 为右线索，直接指向 * p

的中根后继。例如,图 5.14 中 D 的中根后继是 A。

(2) 若 * p 的右子树非空(即 p-> rtag 为 link),则 * p 的中根后继必是其右子树中第一个中根遍历到的结点,也就是从 * p 的右孩子开始,沿该孩子的左链往下查找,直到找到一个没有左孩子的结点为止。该结点是 * p 的右子树中最左下的结点,它就是 * p 的中根线索后继结点。如图 5.15 所示,* p 的中根线索后继结点是 R_k($k \geqslant 1$)。R_k 可能有右孩子也可能无右孩子,若 R_k 无右孩子,则它必定是叶结点。若 $k=1$,则表示 * p 的右孩子 R_1 是 * p 的中根线索后继,如图 5.14 中,A 的中根线索后继是 F,它有右孩子; F 的中根线索后继是 H,它无右孩子; B 的中根线索后继是 D,它是 B 的右孩子(即 D 相当于是 R_1)。



(a) 结点*p的右子树非空时其中根线索后继结点R_k示例



(b) 结点*p的右子树非空时中根线索后继结点存储结构

图 5.15 结点 * p 的右子树非空时其中根线索后继结点是 R_k

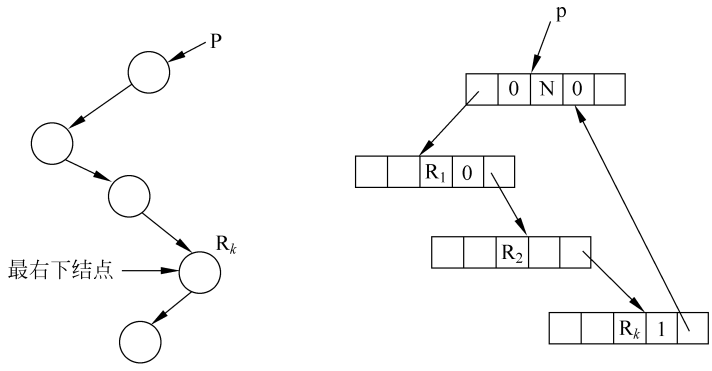
基于上述分析,不难给出中根线索二叉树中求中根线索后继结点的算法。

算法 5.10 中根线索二叉树中求中根线索后继结点算法。

```
//在中根线索树中找结点 * p 的中根线索后继,设 p 非空
binthrnode * in_succ(binthrnode * p)
{
  binthrnode * q;
  if (p -> rtag == thread)                //* p 的右子树为空
     return p -> rchild;                  //返回右线索所指的中根线索后继
  else
     {
       q = p -> rchild;
       while (q -> ltag == link)
            q = q -> lchild;              //左子树非空时,沿左链往下查找
       return q;                          //当 q 的左子树为空时,它就是最左下结点
     }
}
```

显然，该算法的时间复杂度不超过树的高度 h，即 $O(h)$。

由于中根遍历是一种对称遍历操作，故在中根线索二叉树中查找结点 * p 的中根线索前驱结点与找中根线索后继结点的方法完全对称。若 * p 的左子树为空，则 p-> lchild 为左线索，直接指向 * p 的中根线索前驱结点；若 * p 的左子树非空，则从 * p 的左孩子出发，沿右指针链往下查找，直到找到一个没有右孩子的结点为止。该结点是 * p 的左子树中最右下的结点，它是 * p 的左子树中最后一个中根线索遍历到的结点，即 * p 的中根线索前驱结点，如图 5.16 所示。



(a) 结点*p中根线索前驱结点R_k示例　　(b) 结点*p中根线索前驱结点R_k存储结构

图 5.16　结点 * p 的左子树非空时，其中根线索前驱结点是 R_k

由上述讨论可知：若结点 * p 的左子树(或右子树)非空，则 * p 的中根线索前驱(或中根线索后继)是从 * p 的左孩子(或右孩子)开始往下查找，由于二叉链表中结点的链域是向下链接的，所以在非线索二叉树中也同样容易找到 * p 的中根线索前驱(或中根线索后继)；若结点 * p 的左子树(或右子树)为空，则在中根线索二叉树中是通过 * p 的左线索(或右线索)直接到找到 * p 的中根线索前驱(或中根线索后继)，但中根线索一般都是向上指向其祖先结点，而二叉链表中没有向上的链接，因此在这种情况下，对于非线索二叉树，仅从 * p 出发无法找到其中根线索前驱(或中根线索后继)，而必须从根结点开始中根线索遍历，才能找到 * p 的中根线索前驱(或中根线索后继)。由此可见，线索使得查找中根线索前驱和中根线索后继变得简单有效，而对于查找指定结点的前根线索前驱和后根线索后继却没有什么帮助。

在后根线索二叉树中，查找指定结点 * p 的后根线索前驱结点的规律是：

(1) 若 * p 的左子树为空，则 p-> lchild 是前驱线索，指示其后根线索前驱结点。例如，在图 5.17 中，H 的后根线索前驱是 B，F 的后根线索前驱是 G。

(2) 若 * p 的左子树为非空，则 p-> lchild 不是前驱线索。但因为在后根遍历时，根是在遍历其左右子树之后被访问的，故 * p 的后根线索前驱必是两子树中最后一个遍历到的结点。因此，当 * p 的右子树非空时，* p 的右孩子必是其后根线索前驱，例如，图 5.17 中 A 的后根线索前驱是 E；当 * p 无右子树时，* p 的后根线索前驱必是其左孩子，如图 5.17 中 E 的后根线索前驱是 F。

在后根线索二叉树中，查找指定结点 * p 的后根线索后继结点的规律是：

(1) 若 * p 是根，则 * p 是该二叉树后根遍历过程中最后一个访问到的结点，因此，* p 的后根线索后继为空。

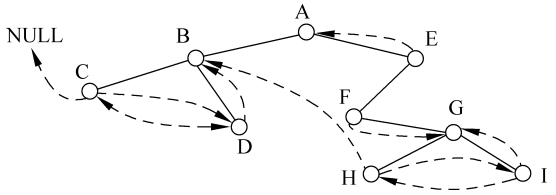


图 5.17 后根线索二叉树

(2) 若 * p 是其双亲的右孩子,则 * p 的后根线索后继结点就是其双亲结点,如图 5.17 中,E 的后根线索后继是 A。

(3) 若 * p 是其双亲的左孩子,但 * p 无右兄弟时, * p 的后根线索后继结点是其双亲结点,如图 5.17 中,F 的后根线索后继是 E。

(4) 若 * p 是其双亲的左孩子,但 * p 有右兄弟时,则 * p 的后根线索后继结点是其双亲的右子树中第一个后根线索遍历到的结点,它是该子树中"最左下的叶结点"。例如图 5.17 中,B 的后根线索后继是双亲 A 的右子树中最左下的叶结点 H。注意,F 是孩子树中最左下结点,但它不是叶子。

由上述讨论可知,在后根线索树中,仅从 * p 出发就能找到其后根线索前驱结点;而找 * p 的后根线索后继结点,仅当 * p 的右子树为空时,才能直接由 * p 的右线索 p-> rchild 得到,否则就必须知道 * p 的双亲结点才能找到其后根线索后继。因此,如果线索二叉树中的结点没有指向其双亲结点的指针,就可能要从根开始进行后根线索遍历才能找到结点 * p 的后根线索后继。由此可见,线索对查找指定结点的后根线索后继并无多大帮助。

类似地,在先根线索二叉树中,找某一点 * p 的先根线索后继也很简单,仅从 * p 出发就可以找到;但找其先根线索前驱也必须知道 * p 的双亲结点,当树中结点未设双亲指针时,同样要进行从根开始的先根线索遍历才能找到结点 * p 的先根线索前驱。详细过程建议读者自行分析。

2. 遍历线索二叉树

遍历某种次序的线索二叉树,只要从该次序下的开始结点出发,反复找到结点在该次序下的后继,直至终端结点。这对于中根和先根线索二叉树是十分简单的。下面给出中根遍历算法。

算法 5.11 遍历中根线索二叉树算法。

```
void traverseinorderthrtree(binthrtree p)      //遍历中根线索二叉树
{
    if(p)                                      //树非空
    {
        while (p->ltag == link)
            p = p->lchild;                     //从根往下找最左下结点,即中根序列的开始结点
        do
        {
            printf("%c",p->data);              //访问结点
            p = in_succ(p);                    //找 * p 的中根线索后继
        }while(p);
```

```
    }
}
```

由于中根序列的终端结点的右线索为空，所以 do 语句的终止条件是 p==NULL。显然，该算法的时间复杂度为 $O(n)$，但因为它是非递归算法，所以在常数因子上小于递归的遍历算法。因此，若对一棵二叉树要经常遍历，或查找结点在指定次序下的前驱和后继，则应采用线索链表作为存储结构为宜。

本节介绍的线索二叉树是一种全线索树，即左、右线索均要建立，但在许多应用中只要建立左、右线索中的一种即可。此外，若在线索链表中增加一个头结点，令头结点的左指针指向根，右指针指向其遍历序列的开始或终端结点会更方便。

5.5 一般树的表示和遍历

5.5.1 一般树的表示

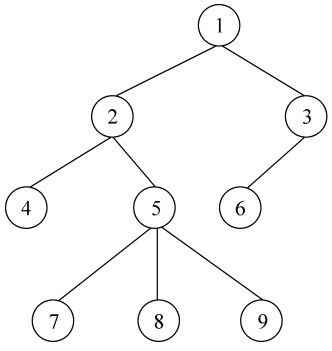
在实际应用中，树（这里特指非二叉树）有多种存储结构。下面介绍三种常用的链表结构。

1. 双亲表示法

假设以一组连续的空间存储树的结点，同时在每个结点中附设一个指示器指示其双亲结点在链表中的位置，其形式说明如下：

```
#define MAXNODE                          //最大结点数
typedef struct
    {datatype data;                      //数据域
     int parent:;                        //双亲域(静态指针域)
    }tnode
typedef tnode tree[MAXNODE + 1];         //静态双亲链表
```

图 5.18 展示了一棵树及其双亲表示的存储结构。



(a) 示例树

结点序号	data	parent
1	1	0
2	2	1
3	3	1
4	4	2
5	5	2
6	6	3
7	7	5
8	8	5
9	9	5

(b) 双亲链表

图 5.18 树的双亲表示法

这种存储结构利用了每个结点(除根以外)只有唯一双亲的性质。反复进行求双亲的操作,直到遇到无双亲的结点时,便找到了树的根。但是,在这种表示法中,求结点的孩子时,需要遍历整个向量。

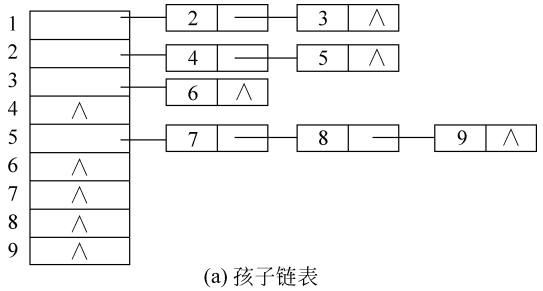
2. 孩子表示法

这种存储方式可以有两种结点结构。一种结构根据树中每个结点可以有多棵子树,则可用多重链表,即每个结点有多个指针域,其中每个指针指向一棵子树的根结点,此时链表中的结点可以有两种格式:一种格式称为同构格式,即若树的度是 d,则每个结点就有 d 个指针域。但是,如果树中很多结点的度小于 d,链表中就会有许多空链域,造成较大的空间浪费。另一种格式称为异构格式,每个结点的指针域的个数与该结点的度数相同,这种方式虽能节约空间,但操作不便。

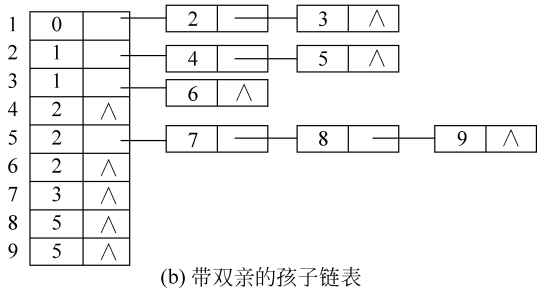
还有一种办法是把每个结点的孩子结点排列成一个线性表,以单链表作为存储结构,则 n 个结点就有 n 个孩子链表(叶子的孩子链表为空表)。而 n 个头指针又组成一个线性表,为了便于查找,由这 n 个头指针组成的线性表可用向量表示。这种存储结构形式说明如下:

```
typedef struct node
{int child;
  struct node * next;
} * link;
typedef link tree[MAXNODE + 1];
```

图 5.19(a)是图 5.18 中的树的孩子表示法。与双亲表示法相反,孩子表示法便于那些涉及孩子的操作,却不适用于求双亲的操作。也可以把双亲表示法和孩子表示法结合起来,即将双亲向量和孩子表头指针向量合在一起。图 5.19(b)就是这样一种存储结构,它和图 5.19(a)表示的是同一棵树。



(a) 孩子链表



(b) 带双亲的孩子链表

图 5.19 图 5.18 所示树的另外两种表示法

3. 孩子兄弟表示法

孩子兄弟表示法又称二叉树表示法或二叉链表表示法。即以二叉链表作为树的存储结构。链表中结点的两个链域分别指向该结点的第一个孩子结点和下一个兄弟结点，分别命名为 fch 域和 nsib 域。

```
typedef struct tnodetp
{
    datatype data;
    tnodetp * fch, * nsib;
} * tlinktp;
```

图 5.20 是图 5.18 中的树的孩子兄弟链表表示法。利用这种存储结构便于实现各类树的操作。首先易于实现找结点孩子等的操作。例如：若要访问结点 x 的第 i 个孩子，则只要先从 fch 域找到第一个孩子结点，然后沿着孩子结点的 nsib 域连续走 $i-1$ 步，便可找到 x 的第 i 个孩子。当然，如果为每个结点增设一个 parent 域，也同样能方便地实现求双亲的操作。

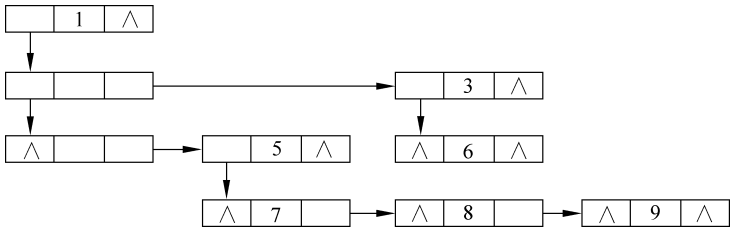


图 5.20 图 5.18 中树的二叉链表表示法

5.5.2 二叉树与树、森林之间的转换

1. 二叉树与树之间的转换

由于二叉树和树都可用二叉链表作为存储结构，因此以二叉链表作为媒介可导出树与二叉树之间的一个对应关系。也就是说，给定一棵树，可以找到唯一的一棵二叉树与之对应，从物理结构来看，它们的二叉链表是相同的，只是解释不同而已。

图 5.21 直观地展示了树与二叉树之间的对应关系。

2. 二叉树与森林之间的转换

从树的二叉链表表示的定义可知，任何一棵与树对应的二叉树，其根的右子树必为空。若把森林中第二棵树的根结点看成是第一棵树的根结点的兄弟，则同样可导出森林和二叉树的对应关系。

图 5.22 展示了森林与二叉树之间的对应关系。

这种一一对应的关系使得森林或树与二叉树可以相互转换，其形式定义如下。

1) 森林转换成二叉树

如果 $F=\{T_1,T_2,\cdots,T_m\}$ 是森林，则可按如下规则转换成一棵二叉树 $B=$(root,LB,RB)。

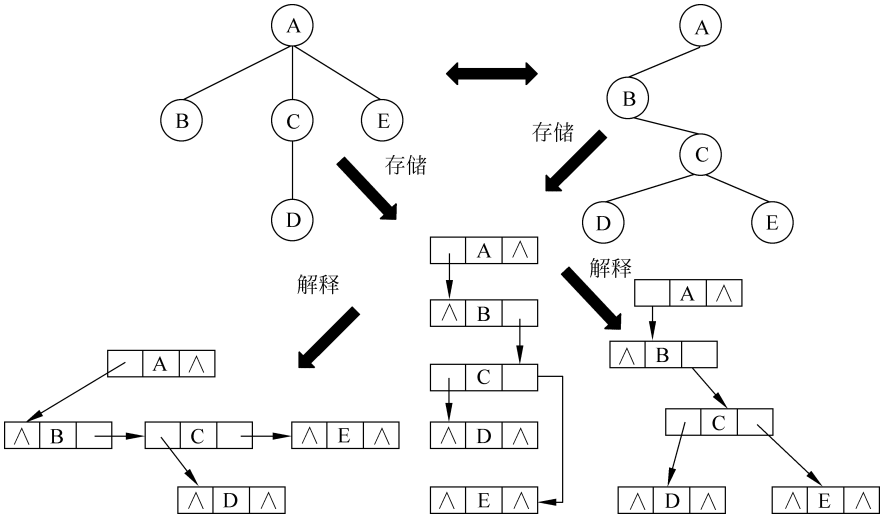


图 5.21 树与二叉树之间的对应关系示例

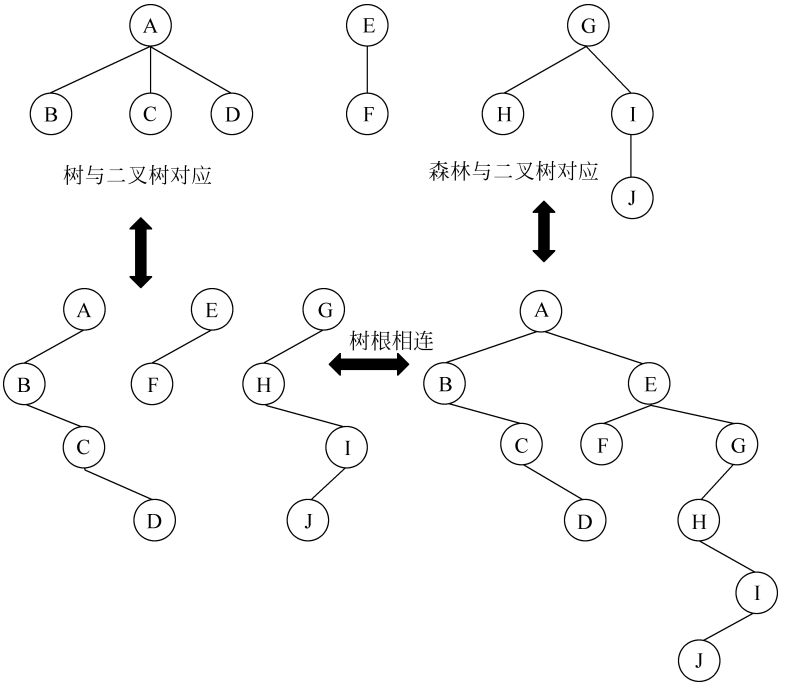


图 5.22 森林与二叉树的对应关系示例

① 若 F 为空,即 $m=0$,则 B 为空树。

② 若 F 为非空,即 $m\neq 0$,则 B 的根 root 即为森林中第一棵子树的根 root(T_1);B 的左子树 LB 是从 T_1 中根结点的子树森林 $F_1=\{T_{11},T_{12},\cdots,T_{1m}\}$转换而成的二叉树;其右子树 RB 是从森林 $F'=\{T_2,T_3,\cdots,T_m\}$转换而成的二叉树。

2) 二叉树转换成森林

如果 $B=$(root,LB,RB)是一棵二叉树,则可按如下规则转换成森林 $F=\{T_1,T_2,\cdots,T_m\}$。

① 若 B 为空，则 F 为空。

② 若 B 为非空，则 F 中第一棵树 T_1 的根 root(T_1)即为二叉树 B 的根 root；T_1 中根结点的子树森林 F_1 是由 B 的左子树 LB 转换而成的森林；F 中除 T_1 之外其余的树组成的森林 $F'=\{T_2,T_3,\cdots,T_m\}$是由 B 的右子树 RB 转换而成的森林。

从上述递归定义容易写出相互转换的递归算法。这样，森林和树的操作就可以转换成二叉树的操作来实现，从而简化了操作方法。

5.5.3 一般树的遍历

与二叉树类似，遍历是树的一种重要运算。树的主要遍历方法有以下三种。

1. 先根遍历(与对应的二叉树的先根遍历序列一致)

若树非空，则：

(1) 访问根结点。

(2) 依次先根遍历根的各个子树。

2. 后根遍历(与对应的二叉树的中根遍历序列一致)

若树非空，则：

(1) 依次后根遍历根的各个子树。

(2) 访问根结点。

3. 层次遍历

(1) 若树非空，访问根结点。

(2) 若第 1～i($i\geqslant 1$)层结点已被访问，且第 $i+1$ 层结点尚未访问，则从左到右依次访问第 $i+1$ 层。

显然，按层次遍历所得的结点访问序列中，各结点的序号与按层编号所得的编号一致。

例如，对图 5.23 所示树来说：

先根遍历结点序列为 A,B,D,E,H,I,J,C,F,G；

后根遍历结点序列为 D,H,I,J,E,B,F,G,C,A；

层次遍历结点序列为 A,B,C,D,E,F,G,H,I,J。

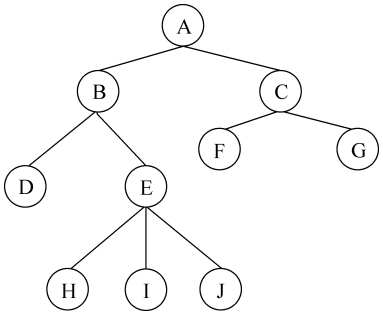


图 5.23 一般树示例

5.6 哈夫曼树及其应用

哈夫曼(Huffman)树又称最优二叉树,是一种带权路径长度最短的树,有着广泛的应用。本节先讨论哈夫曼树的概念,然后讨论它的应用:最佳判断过程和哈夫曼编码。

5.6.1 哈夫曼树

1. 树的路径长度和带权路径长度

结点间的路径长度: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间的路径,路径上的分支数目称为这两个结点之间的路径长度。

树的路径长度:从树根到每个结点的路径长度之和。这种路径长度最短的树是前面定义的完全二叉树。

在许多应用中,常常将树中结点赋予一个有某种意义的实数,称为该结点的权。

结点的带权路径长度为:从该结点到树根之间的路径长度与结点上权的乘积。

树的带权路径长度为:树中所有叶子结点的带权路径长度之和,通常记作

$$\mathrm{WPL}=W_1L_1+W_2L_2+W_3L_3+\cdots+W_iL_i+\cdots+W_nL_n$$

$$=\sum_{i=1}^{n}W_iL_i$$

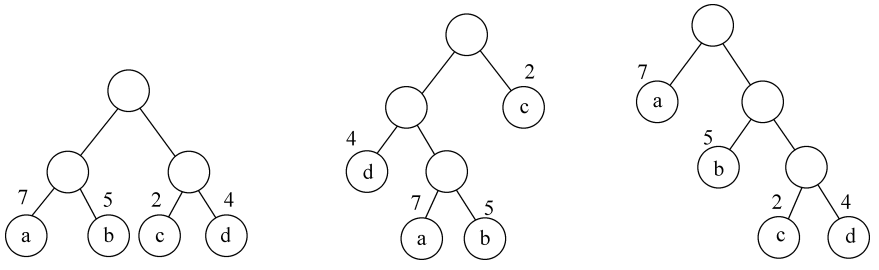
其中,n 为二叉树的叶子结点的个数,W_i 为第 i 个叶子结点的权值,L_i 为从根结点到第 i 个叶子结点的路径长度。

例如,图 5.24 中的三棵二叉树,都有 4 个叶子结点 a,b,c,d,权值分别为 7,5,2,4,它们的带权路径长度分别为

$$\mathrm{WPL}=7\times2+5\times2+2\times2+4\times2=36$$

$$\mathrm{WPL}=7\times3+5\times3+2\times1+4\times2=46$$

$$\mathrm{WPL}=7\times1+5\times2+2\times3+4\times3=35$$



(a) WPL为36的二叉树　(b) WPL为46的二叉树　(c) WPL为35的二叉树

图 5.24　具有不同带权路径的二叉树

2. 哈夫曼树和哈夫曼算法

假设有 n 个权值 $W_1,W_2,\cdots,W_n$,试构成一棵有 n 个叶子结点的二叉树,每个叶子结点

权值为 W_i，则其中带权路径长度 WPL 最小的二叉树称作哈夫曼树(或最优二叉树)。

在图 5.24(c)中的树的 WPL 最小。可以验证，它恰为哈夫曼树，即其带权路径长度在所有权值为 7,5,2,4 的 4 个叶子结点的二叉树中最小。

怎样根据 n 个权值 $W_1,W_2,\cdots,W_n$ 构造哈夫曼树呢？哈夫曼在 1952 年提出了一种算法，很好地解决了这个问题。该算法被称为哈夫曼算法，简述如下。

(1) 根据给定的 n 个权值 $W_1,W_2,\cdots,W_n$，构成 n 棵二叉树的集合 $F=\{T_1,T_2,\cdots,T_n\}$，其中每棵二叉树 T_i 中只有一个权为 W_i 的根结点，其左、右子树均为空。

(2) 在 F 中任选两棵根结点的权值最小的树作为左、右子树，构成一棵新的二叉树，且置新的二叉树的根结点的权值为其左、右子树上根结点的权值之和。

(3) 从 F 中删除这两棵树，同时将新得到的二叉树加入到 F 中。

(4) 重复(2)和(3)步，直到 F 中只含一棵树为止。这棵树便是哈夫曼树。

例如有 4 个叶子结点 a,b,c,d，权值分别为 6,5,3,4，其哈夫曼树的构造过程如图 5.25 所示。

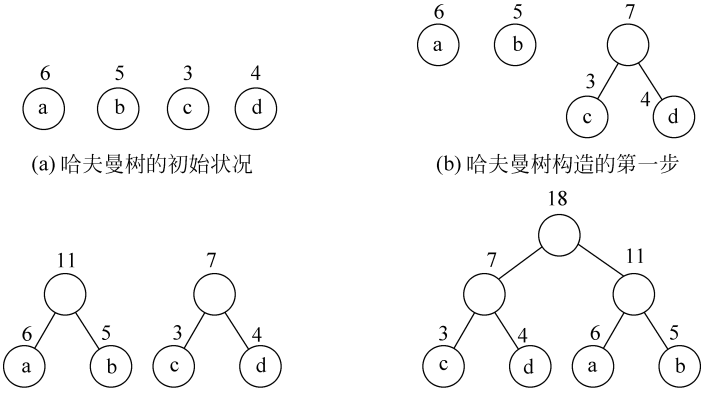


图 5.25 哈夫曼树的构造过程

下面讨论哈夫曼树的存储结构及哈夫曼算法的实现。

由哈夫曼算法可知，初始森林中共有 n 棵二叉树，每棵树中都仅有一个孤立的结点，它们既是根，又是叶子。算法的第(2)步是：将当前森林中的两棵根结点权值最小的二叉树，合并成一棵新二叉树。每合并一次，森林中就减少一棵树。显然，要进行 $n-1$ 次合并，才能使森林中的二叉树的数目由 n 棵减少到剩下一棵最终的哈夫曼树。并且，每次合并都要产生一个新结点，合并 $n-1$ 次共产生 $n-1$ 个新结点，显然它们都是具有两个孩子的分支结点。由此可知，最终求得的哈夫曼树中共有 $2n-1$ 个结点，其中 n 个叶子结点是初始森林中的 n 个孤立结点。显然，哈夫曼树中没有度为 1 的分支结点，这类树常称为严格的二叉树。实际上，所有具有 n 个叶子结点的严格二叉树都恰有 $2n-1$ 个结点。可以用一个大小为 $2n-1$ 的向量来存储哈夫曼树中的结点，其存储结构为：

```
#define n 叶子数
#define m 2*n-1                          //树中结点总数
typedef struct
{ //结点类型
```

```
    int weight;                          //权值
    int plink,llink,rlink;               //双亲及左右孩子指针(静态指针)
}node;
node tree[m+1];                          //下标取值从 1 到 m,0 作为空指针标志
```

在上述存储结构上实现的哈夫曼树算法可大致描述为：

(1) 初始化。将 tree[1..m]中每个结点里的三个指针均置为空(即置为 0)。

(2) 输入。读入 n 个叶子的权值,分别保存于 tree 的前 n 个分量中,它们是初始森林中 n 个孤立的根结点上的权值。

(3) 合并。

对森林中的树共进行 $n-1$ 次合并,所产生的新结点依次放入 tree 的第 i 个分量中($n<i\leqslant m$)。每次合并分两步：

① 在当前森林 tree[1..$i-1$]的所有结点中,选取权值最小和次小的两个根结点 tree[x_1]和 tree[x_2]作为合并对象,这里 $1\leqslant x_1, x_2\leqslant i-1$。

② 将根为 tree[x_1]和 tree[x_2]的两棵树作为左右子树合并成为一棵新的树,新树的根是新结点 tree[i]。因此,应将 tree[x_1]和 tree[x_2]的双亲 plink 置为 i,将 tree[i]的 llink 和 rlink 分别置为 x_1 和 x_2,而新结点 tree[i]的权值应置为 tree[x_1]和 tree[x_2]的权值之和。注意,合并后 tree[x_1]和 tree[x_2]在当前森林中已不再是根,因为它们的双亲指针均已指向了 tree[i],所以下一次合并时不会被选为合并对象。

哈夫曼算法实现如下。

算法 5.12 哈夫曼树的构造。

```
void sethuftree(node tree[])
{
    int i,x1,x2;
    inithafumantree(tree);               //将 tree 初始化
    inputweight(tree);                   //输入叶子权值
    for(i=n+1;i<=m;i++)                  //共进行 n-1 次合并,新结点依次存于 tree[i]中
    {
        select(i-1,&x1,&x2);
        //在 tree[1..i-1]中选择两个权值最小的根结点,其序号分别为 x1 和 x2
        tree[x1].plink=i;
        tree[x2].plink=i;
        tree[i].llink=x1;                //权值最小的根结点是新结点的左孩子
        tree[i].rlink=x2;                //权值次小的根结点是新结点的右孩子
        tree[i].weight=tree[x1].weight+tree[x2].weight;
    }
}
```

5.6.2 哈夫曼树的应用

1. 最佳判定算法

在解决某些判定问题时,利用哈夫曼树可以得到最佳判定算法。例如,要编制一个将百分制转换成五级分制的程序,只需利用条件语句便可完成。如：

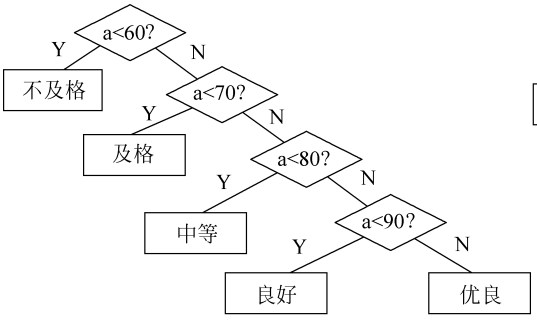
```
if(a < 60)
        b = "bad";
else if(a < 70)
             b = "pass";
     else if(a < 80)
                  b = "general";
          else if(a < 90)
                       b = "good";
               else b = "excellent";
```

这个判定过程可用图 5.26(a)的判定树来表示。如果上述程序需反复使用,而且每次的输入量很大,则应考虑上述程序的执行效率问题,即其操作所需时间。因为在实际问题处理中,学生的成绩在五个等级上的分布是不均匀的。假设其分布如表 5.1 所示,显然,80%以上的数据需进行 3 次或 3 次上的比较才能得出结果。

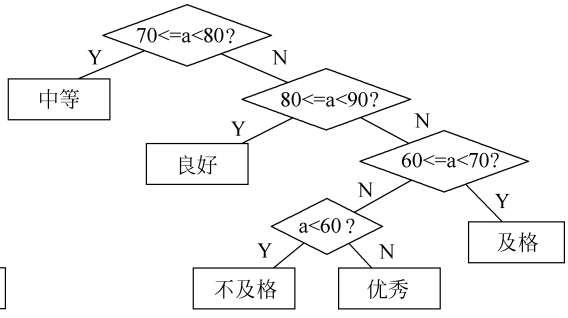
表 5.1 学生成绩分布表

分数	0～59	60～69	70～79	80～89	90～100
比例数	0.05	0.15	0.40	0.30	0.10

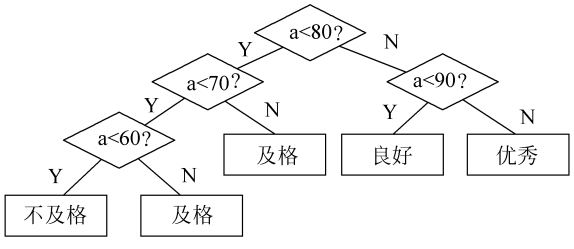
假定以 5、15、40、30、10 为权值构成一棵有五个叶子结点的哈夫曼树,则可得到如图 5.26(b)所示的判定过程,它可使大部分数据经过较少的比较次数即得到结果。但由于每个判定框都有两次比较,将这两次比较分开,即得到如图 5.26(c)所示的判定树,按此判定树写出相应的程序。假设现有 10 000 个输入数据,按图 5.26(a)所示的判定过程操作总共需进行 31 500 次比较;而按图 5.26(c)所示的判定过程进行操作,则总共需进行 22 000 次比较。显然,优化的判定过程极大地提高了效率。



(a) 不考虑成绩分布比例值的判定树



(b) 考虑成绩分布比例值的判定树(哈夫曼树)



(c) 优化的判定树

图 5.26 转换五级分制的判定过程

2. 哈夫曼编码

电报是进行快速远距离的通信手段之一。发送电报时需将传送的文字转换成二进制的字符组成的字符串。例如,假设需传送的电文为'ABACCDA',它只有四种字符,只需两位二进制字符串便可分辨。假设 A、B、C、D 的编码分别为 00,01,10 和 11,则上述七个字符的电文编码为'00010010101100',总长 14 位,对方接收时可按二位一分进行译码。

当然,在传送电文时,希望总长尽可能的短。如果对每个字符设计长度不等的编码,且让电文中出现次数较多的字符采用尽可能短的编码,则传送电文的总长度便可减短。如果设计 A、B、C、D 的编码分别为 0、00、1 和 01,则上述电文被编码成长度为 9 的字符串'000011010'。但是,这样的电文无法翻译。例如,传送过去的字符串中前四个字符的子串'0000'就可有多种译法,或是'AAAA'或是'ABAA'等。产生该问题的原因是 A 的编码与 B 的编码的开始部分(前缀)相同。因此,若对某字符集进行不等长编码,就要求字符集中任一字符的编码都不是其他字符编码的前缀,这种编码称为前缀编码。显然,等长编码也是前缀编码。

问题是应该怎样设计前缀编码?什么样的前缀编码才能使得电文的总长最短?可以利用哈夫曼树设计二进制的前缀编码来解决此问题。

假设有一棵如图 5.27 所示的二叉树,其四个叶子结点分别表示 A、B、C、D 四个字符,且约定左分支表示字符'0',右分支表示字符'1',则可将从根结点到叶子结点的路径上分支字符组成的字符串作为该叶子结点字符的编码。不难理解,如此得到的必为二进制前缀编码,如图 5.27 所示,A、B、C、D 的二进制前缀编码分别为 0、10、110 和 111。

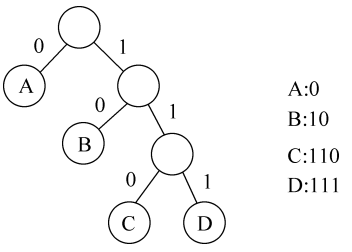


图 5.27 前缀编码示例

那么如何得到使电文总长最短的二进制前缀编码呢?假设每种字符在电文中出现的次数为 W_i,其编码长度为 L_i,电文中只有 n 种字符,则电文总长为 $W_1L_1+W_2L_2+\cdots+W_iL_i+\cdots+W_nL_n$。对应到二叉树上,若置 W_i 为叶子结点的权,L_i 恰为从根到叶子的路径长度。则 $W_1L_1+W_2L_2+\cdots+W_iL_i+W_nL_n$ 恰为二叉树的带权路径长度。由此可见,设计电文总长最短的二进制前缀编码,也就是以 n 种字符出现的频率作叶子结点的权,设计一个哈夫曼树的问题,由此得到的二进制前缀编码便称为哈夫曼编码。

在求出了给定字符集的哈夫曼树后,求该字符集的哈夫曼编码的具体过程是:依次以叶子 tree[i](1≤i≤n)为出发点,向上回溯至根为止。上溯时走左分支则生成编码 0,走右分支则生成编码 1。显然,这样生成的编码与要求的编码反序。因此,将生成的代码从后往前依次存放在一个临时向量中,并设一个指针 start 指示编码在该向量中的起始位置。当某字符编码完成时,从临时向量的 start 处将编码复制到该字符相应的位串 bits 中即可。因为字符集大小为 n,故变长编码的长度不会超过 n,加上一个结束符'\0',bits 的大小应为 $n+1$。

字符集编码的存储结构及其算法描述如算法 5.13。

算法 5.13 哈夫曼编码。

```
typedef struct
{                                        //结点类型
```

```
    int weight;                                  //权值
    int plink,llink,rlink;                       //双亲及左右孩子指针(静态指针)
}node;
typedef struct
{
    int start;                                   //存放起始位置
    char bits[n+1];                              //存放编码位串
}codetype;
typedef struct
{
    char symbol;                                 //存储字符
    codetype code;                               //存储编码
}element;
element table[n+1];
void huffcode(node tree[],element table[])       //根据哈夫曼树 tree 求哈夫曼编码表 table
{
    int i,s,f;                                   //s 和 f 分别指示 tree 中孩子和双亲的位置
    codetype c;                                  //临时存放编码
    for(i=1;i<=n;i++)                            //依次求叶子 tree[i]的编码
    {
        c.start=n+1;
        s=i;                                     //从叶子 tree[i]开始上溯
        while(f=tree[s].plink)                   //直至上溯到树根为止
        {
            c.bits[--c.start]=(s==tree[f].llink)?'0':'1';
            s=f;
            f=tree[s].plink;
        };
        table[i].code=c;                         //临时编码复制到最终位置
    }
}
```

例 5.1 已知某系统在通信网络中只可能出现八种字符(A、B、C、D、E、F、G、H),其频率分别为 0.05、0.29、0.07、0.08、0.14、0.23、0.03、0.11,试设计哈夫曼编码。

设权 $W=(5,29,7,8,14,23,3,11)$,字符数目 $n=8$,按照哈夫曼算法可构造一棵哈夫曼树,如图 5.28 所示,根据哈夫曼树得到的哈夫曼编码如图 5.29 所示。

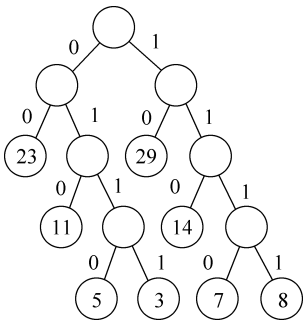


图 5.28 例 5.1 的哈夫曼树

有了字符集的哈夫曼编码表之后,对数据文件的编码过程是:依次读入文件中的字符 C,在哈夫曼编码表 table 中找到此字符,若 table[i].symbol==C,则将字符 C 转换为 table[i].code 中存放的编码串。

对压缩后的数据文件进行解码则必须借助于哈夫曼树 tree。其过程是:依次读入文件的二进制码,从哈夫曼树的根结点出发,若当前读入 0,则走向左孩子,否则走向右孩子,一旦达到某一叶子 tree[i]时便译出相应的字符 table[i].symbol,然后重新从根出发继续译码,直至文件结束。

	Weight	Parent	Lch	Rch
1	5	0	0	0
2	29	0	0	0
3	7	0	0	0
4	8	0	0	0
5	14	0	0	0
6	23	0	0	0
7	3	0	0	0
8	11	0	0	0
9	-	0	0	0
10	-	0	0	0
11	-	0	0	0
12	-	0	0	0
13	-	0	0	0
14	-	0	0	0
15	-	0	0	0

(a) 哈夫曼树初始化

	Weight	Parent	Lch	Rch
1	5	9	0	0
2	29	14	0	0
3	7	10	0	0
4	8	10	0	0
5	14	12	0	0
6	23	13	0	0
7	3	9	0	0
8	11	11	0	0
9	8	11	1	7
10	15	12	3	4
11	19	13	8	9
12	29	14	5	10
13	42	15	6	11
14	58	15	2	12
15	100	0	13	14

(b) 构造的哈夫曼树

	0	1	2	3	4	5	6	7	Start
1					0	1	1	0	4
2							1	0	6
3					1	1	1	0	4
4					1	1	1	1	4
5						1	1	0	5
6							0	0	6
7					0	1	1	1	4
8						0	1	0	5

(c) 哈夫曼树对应的哈夫曼编码

图 5.29 哈夫曼编码

5.7 C++中的树

5.7.1 C++中的二叉树结点类

例5.2是二叉链表结点的C++类BTNode,每个结点包含三个数据成员和三个构造函数。

例5.2 二叉树结点类。

```
template<class T>
struct BTNode
{
BTNode(){ lChild=rChild=NULL;}
//lChild和rChild分别是指向左孩子和右孩子的指针
BTNode(const T& x)
{
element=x; lChild=rChild=NULL;
}
BTNode(const T& x, BTNode<T>* l,BTNode<T>* r)
{
element=x; lChild=l; rChild=r;
}
T element;
BTNode<T> *lChild, *rChild;
};
```

5.7.2 C++中的二叉树类

例5.3定义了由二叉链表表示的二叉树类BinaryTree。类BinaryTree包含唯一的数据成员,它指向一个二叉链表根结点的指针root。请务必注意区分二叉树对象和由二叉树对象的根指针root所指示的二叉树(即二叉链表)。一个二叉树对象的根指针root所指示的二叉树,是该二叉树对象所包含的一棵二叉树。在不引起混淆的情况下,将二叉树对象和它所包含的二叉树统称为二叉树。例5.4是二叉树类BinaryTree的部分运算。例5.5是二叉树类BinaryTree的递归方式先根遍历二叉树运算。

例5.3 二叉树类。

```
template<class T>
class BinaryTree
{
public:
   BinaryTree(){root=NULL;}
   ~BinaryTree(){Clear();}
   bool IsEmpty()const;
   void Clear();
   bool Root(T &x)const;
   void MakeTree(const T &e ,BinaryTree<T> &left, BinaryTree<T>& right);
```

```
    //构造二叉树
    void BreakTree(T &e ,BinaryTree<T> &left, BinaryTree<T> & right);
    void PreOrder(void ( * Visit)(T& x));       //递归方式先根遍历二叉树
    void InOrder(void ( * Visit)(T& x));
    void PostOrder(void ( * Visit)(T& x));
protected:
    BTNode<T> * root;
private:
    void Clear(BTNode<T> * t);
    void PreOrder(void ( * Visit)(T& x),BTNode<T> * t);
    void InOrder(void ( * Visit)(T& x),BTNode<T> * t);
    void PostOrder(void ( * Visit)(T& x),BTNode<T> * t);
};
```

例 5.4 部分二叉树运算。

```
template<class T>
bool BinaryTree<T>::Root(T &x)const
{
if(root){
         x=root->element; return true;
      }
else return false;
}
template<class T>
void BinaryTree<T>::MakeTree(const T &x ,BinaryTree<T> &left,BinaryTree<T> & right)
{
if(root||&left==&right) return;
root=new BTNode<T>(x,left.root, right.root);
     left.root=right.root=NULL;
}
template<class T>
void BinaryTree<T>::BreakTree(T &x, BinaryTree<T> &left, BinaryTree<T> & right)
{
if (!root||&left==&right||left.root||right.root)return;
  x=root->element;
left.root=root->lChild;right.root=root->rChild;
  delete root;root=NULL;
}
```

例 5.5 递归方式先根遍历二叉树。

```
template<class T>
void BinaryTree<T>::PreOrder(void ( * Visit)(T& x))
{
PreOrder(Visit,root);
}
template<class T>
void BinaryTree<T>::PreOrder(void ( * Visit)(T& x),BTNode<T> * t)
{
if (t){
         Visit(t->element);                  //递归遍历根结点
```

```
        PreOrder(Visit,t->lChild);          //递归遍历左子树
         PreOrder(Visit,t->rChild);         //递归遍历右子树
            }
}
```

5.7.3 C++中二叉树的非递归遍历

二叉树的遍历可分为递归方式和非递归方式。用C++来描述二叉树的非递归遍历如例5.6所示的遍历器类BIterator,由它可派生三个具体实施先根、中根和后根遍历的遍历器类,例5.7是非递归方式的中根遍历器类。

例5.6 遍历器类。

```
template<class T>
class BIterator
{
public:
virtual T* GoFirst(const BinaryTree<T>& bt)=0;
virtual T* Next (void)=0;
virtual void Traverse(void (*Visit)(T& x),const BinaryTree<T>& bt);
protected:
    BTNode<T>* r, *current;
};
template<class T>
void BIterator<T>::Traverse(void (*Visit)(T& x),const BinaryTree<T>& bt)
{
            T* p=GoFirst(bt);
            while (p){
              Visit(*p);p=Next();
            }
}
```

例5.7 中根遍历器类。

```
template<class T>
class IInOrder:public BIterator<T>
{
public:
    IInOrder(BinaryTree<T>& bt,int mSize)
    {
        r=bt.root; current=NULL;
        s=new SeqStack<BTNode<T>* >(mSize);
    }
    T* GoFirst(const BinaryTree<T>& bt);
    T* Next (void);
private:
    SeqStack<BTNode<T>*>* s;
};
template<class T>
T* IInOrder<T>::GoFirst(const BinaryTree<T>&bt)
{
```

```
current = bt.root;
    if (!current) return NULL;
    while (current->lChild!= NULL){
        s->Push(current); current = current->lChild;
        }
        return &current->element;
}
template <class T>
T* IInOrder<T>::Next(void)
{
BTNode<T>* p;
  if (current->rChild!= NULL){
     p = current->rChild;
      while (p->lChild!= NULL){
            s->Push(p);p = p->lChild;
}
     current = p;
}
else if (!s->IsEmpty()){
             s->Top(current); s->Pop();
        }
        else {
            current = NULL; return NULL;
        }
return &current->element;
}
```

习题 5

1. 已知一棵树边的集合为(I,M)、(I,N)、(E,I)、(B,E)、(B,D)、(A,B)、(G,J)、(G,K)、(C,G)、(C,F)、(T,L)、(C,T)、(A,C),画出这棵树,并回答下列问题:

(1) 哪个是根结点?

(2) 哪些是叶子结点?

(3) 哪个是结点 G 的双亲?

(4) 哪些是结点 G 的祖先?

(5) 哪些是结点 G 的孩子?

(6) 哪些是结点 E 的子孙?

(7) 哪些是结点 E 的兄弟?哪些是结点 F 的兄弟?

(8) 结点 B 和 N 的层次号分别是什么?

(9) 树的深度是多少?

(10) 以结点 C 为根的子树的深度是多少?

2. 一棵度为 2 的树与一棵二叉树有何区别?

3. 试分别画出具有 3 个结点的树和 3 个结点的二叉树的所有不同形态。

4. 一棵深度为 N 的满 K 叉树有如下性质:第 N 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。如果按层次顺序从 1 开始对全部结点编号，问：

(1) 各层的结点数目是多少？

(2) 编号为 n 的结点的父结点(若存在)的编号是多少？

(3) 编号为 n 的结点的第 i 个儿子(若存在)的编号是多少？

(4) 编号为 n 的结点有右兄弟的条件是什么？其右兄弟的编号是多少？

5. 已知一棵度为 m 的树中有 n_1 个度为 1 的结点，n_2 个度为 2 的结点，…，n_m 个度为 m 的结点，问该树中有多少个叶子结点？

6. 试列出图 5.30 所示的二叉树的终端结点、非终端结点以及每个结点的层次。

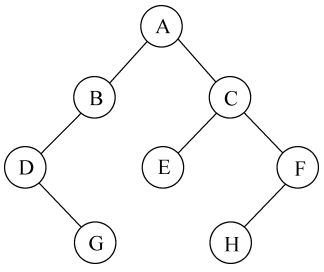


图 5.30 二叉树示例

7. 对于图 5.30 所示的二叉树，分别列出先根遍历、中根遍历、后根遍历的结点序列。

8. 在二叉树的顺序存储结构中，实际上隐含着双亲的信息，因此可和三叉链表对应。假设每个指针域占 4 字节的存储空间，每个信息占 k 字节的存储空间。试问对于一棵有 n 个结点的二叉树，且在顺序存储结构中最后一个结点的下标为 m，在什么条件下顺序存储结构比二叉链表更节省空间？

9. 假定用两个一维数组 L(1：n)和 R(1：n)作为有 n 个结点的二叉树的存储结构，L(i)和 R(i)分别指示结点 i 的左孩子和右孩子，0 表示空。

(1) 试写一个算法判别结点 u 是否为结点 v 的子孙；

(2) 先由 L(1：n)和 R(1：n)建立一维数组 T(1：n)，使 T 中第 i(i=1,2,…,n)个分量指示结点 i 的双亲，然后编写判别结点 u 是否为结点 v 的子孙的算法。

10. 假设 n 和 m 为二叉树中的两结点，用 1、0、Φ(分别表示肯定、恰恰相反和不一定)填写表 5.2。

表 5.2 第 10 题表

	先根遍历时	中根遍历时	后根遍历时
已知	n 在 m 前？	n 在 m 前？	n 在 m 前？
n 在 m 的左方			
n 在 m 的右方			
n 是 m 的祖先			
n 是 m 的子孙			

注：① 如果离 a 和 b 最近的共同祖先 p 存在，且②a 在 p 的左子树中，b 在 p 的右子树中，则称 a 在 b 的左方(即 b 在 a 的右方)。

11. 假设以二叉链表作为存储结构，试分别写出先根遍历和后根遍历的非递归算法，可直接利用栈的基本运算。

12. 假设在二叉链表中增设两个域：双亲域(parent)以指示其双亲结点；标志域(mark)为 0..2，以区分在遍历过程中到达该结点时应该继续向左或向右或访问该结点。试以此存储结构编写不用栈的后根遍历的算法。

13. 试编写算法在一棵以二叉链表存储的二叉树中求这样的结点：它在先根序列中第

K 个位置。

14. 试以二叉链表作为存储结构,编写计算二叉树中叶子结点数目的递归算法。

15. 以二叉链表作为存储结构,编定算法将二叉树中所有结点的左、右子树相互交换。

16. 已知一棵二叉树以二叉链表作为存储结构,编写完成下列操作的算法:对于树中每个元素值为 x 的结点,删去以它为根的子树,并释放相应的空间。

17. 已知一棵以二叉链表作存储结构的二叉树,试编写复制这棵二叉树的非递归算法。

18. 已知一棵以二叉链表为存储结构的二叉树,试编写层次顺序(同一层自左向右)遍历二叉树的算法。

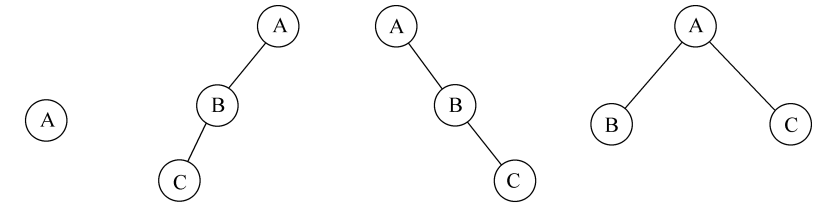
19. 试以二叉链表作为存储结构,编写算法判别给定二叉树是否为完全二叉树。

20. 已知一棵完全二叉树存在于顺序存储结构 A(1 : max)中,A[1 : n]含结点值。试编写算法由此顺序结点建立该二叉树的二叉链表。

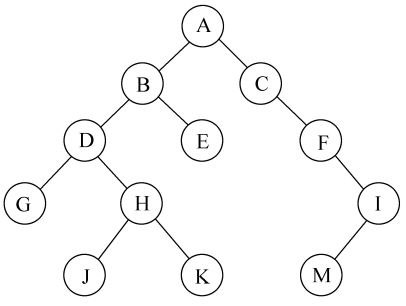
21. 编写一个算法,输出以二叉树表示的算术表达式,若该表达式中含有括号,则在输出时应该添上,已知二叉树的存储结构为二叉链表。

22. 一棵二叉树的直径定义为,从二叉树的根结点到所有叶子结点的路径长度的最大值。假设以二叉链表作为存储结构,试编写算法求给定二叉树的直径和其长度等于直径的一条路径(即从根到该叶子结点的序列)。

23. 试分别画出图 5.31 中各二叉树的先根、中根、后根的线索二叉树。



(a) 二叉树1　(b) 二叉树2　(c) 二叉树3　(d) 二叉树4



(e) 二叉树5

图 5.31　二叉树示例

24. 试编写一个算法,在中根线索二叉树中,结点 p 之下插入一棵以结点 x 为根,只有左子树的中根全线索化二叉树,使 x 为根的二叉树成为 p 的左子树,若 p 原来有左子树,则令它为 x 的右子树。完成插入之后二叉树应保持线索化特性。

25. 已知一棵以线索链表作为存储结构的中根线索二叉树,试编写在此二叉树上找后根线索后继的算法。

26. 将图 5.32 所示森林转换为相应的二叉树,并按中根遍历进行线索化:

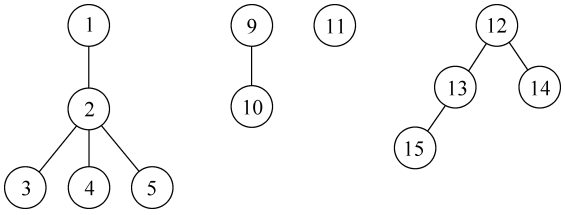


图 5.32 森林示例

27. 画出图 5.31 所示各二叉树相应的森林。

28. 对以下存储结构分别写出计算树的深度的算法。

(1) 双亲表示法；

(2) 孩子链表表示法；

(3) 孩子兄弟表示法。

29. 假设一棵二叉树的层序序列为 A B C D E F G H I J，中根序列为 D B G E H J A C I F。请画出该树。

30. 证明：树中结点 u 是结点 v 的祖先，当且仅当在先根序列中 u 在 v 之前，且在后根序列中 u 在 v 之后。

31. 证明：在结点数大于 1 的哈夫曼树中不存在度为 1 的结点。

32. 设有一组权 WG=1、4、9、16、25、36、49、64、81、100，试画出其哈夫曼树，并计算带权的路径长度。

33. 假设用于通信的电文仅由 8 个字母组成，字母在电文中出现的频率分别为 7、19、2、6、32、3、21、10，试为这 8 个字母设计哈夫曼编码，使用 0～7 的二进制表示形式是另一编码方案。对于上述实例，比较两个方案的优缺点。

34. 证明：由一棵二叉树的先根序列和中根序列可唯一确定这棵二叉树。

35. 已知一棵二叉树的先根序列和中根序列分别存在于两个一维数组中，试编写算法建立该二叉树的二叉链表。

上机练习 5

1. 建立一棵二叉排序树并中根遍历(根据题目完善程序)。

```
#include "stdio.h"
#include "malloc.h"
struct node{
char data;
struct node *lchild, *rchild;
} bnode;

typedef struct node * blink;

blink add(blink bt,char ch)                    //二叉排序树的插入算法
{
  if(bt == NULL)
```

```
    {
      bt = nalloc(sizeof(bnode));
      bt -> data = ch;
      bt -> lchild = bt -> rchild = NULL;
     }
    else
         if ( ch < bt -> data)
               bt -> lchild = add(bt -> lchid,ch);
    else
               bt -> rchild = add(bt -> rchild,ch);
  return bt;
}

void inorder(blink bt)
{
if(bt)
   { inorder(bt -> _____);
      printf(" % c", _____);
      inorder(bt -> _____);
   }
}
void main()
{
blink root = NULL;
int i,n;
char x;
scanf(" % c",&n);
for(i = 1;i < = n;i++)
{
   x = getchar();
   root = add(root,x);
}
inorder(root);
printf("\n");
}
```

2. 由前缀表达式建立二叉树的二叉链表结构,求该表达式对应的后缀、中缀表达式。

3. 编写程序,实现按层次遍历二叉树。

4. 建立由合法的表达式字符串确定的只含二元操作符的非空表达式树,其存储结构为二叉链表,用二叉树的遍历算法求该中缀表达式对应的后缀、前缀表达式。