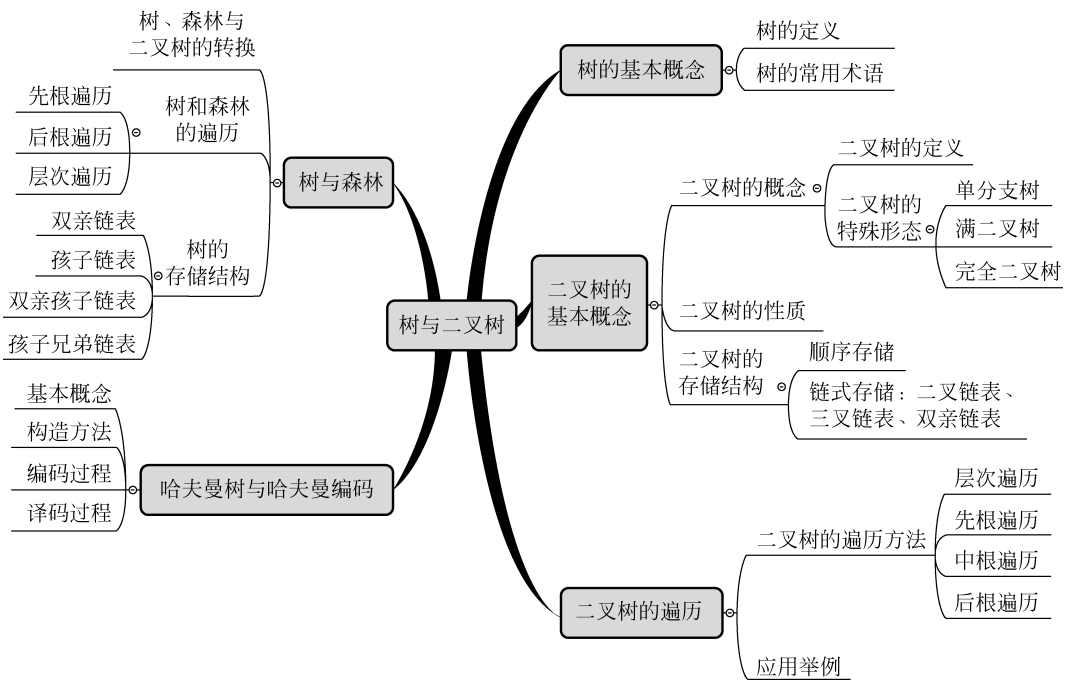
第5章 树与二叉树

前面章节中介绍的数据结构都属于线性结构，从这一章开始我们将阐述非线性结构。树形结构是一种非常重要的非线性结构。在线性结构中数据元素之间的逻辑关系为一对一的线性关系；而在树形结构中，数据元素之间具有一对多的逻辑关系，它反映了数据元素之间的层次关系，一个数据元素可以有多个后继但最多只有一个前驱的特点。树形结构在现实世界中广泛存在，它是很多事物与关系的抽象模型，例如，人类社会的亲属关系按层次表示成家谱树；公司、学校等社会机构按层次化方式形成的单位组织机构图；文件系统中用树形结构表示的目录树；等等。在计算机领域中，树形结构也得到了广泛的应用，例如，操作系统中的文件管理；编译程序中的语法分析；系统设计时对系统功能的划分；等等。此外，在对数据进行排序或查找操作时，若采用某种树形结构来组织待查找或待排序的数据，能有效地提高操作效率。

【本章主要知识导图】



5.1 树的基本概念

本章讲解的重点是一种特殊的树——二叉树，但在讲解二叉树之前，我们先从广义上介绍树的概念及相关的基本术语。

1. 树的定义

树是由 $n(n\geqslant 0)$个结点所构成的有限集合。当 $n=0$ 时，它被称为空树；当 $n>0$ 时，树中的 n 个结点应满足下述条件。

(1) 有且仅有一个根结点(Root)。

(2) 其余结点可分为 $m(m\geqslant 0)$个互不相交的有限集合 $T_1, T_2, \cdots, T_m$，其中每个子集本身又是一棵符合本定义的树，称为根结点(Root)的子树。

上述定义采用的是递归方式，即在树的定义中又用到了树这一概念。事实上，树的层次结构体现了数据元素之间具有的层次关系，即对于一棵非空树，有且仅有一个没有前驱的结点，这个结点就是根结点，其余结点有且仅有一个前驱，但可以有零个或多个后继。图 5-1 给出了用树形表示法给出的 3 棵树的逻辑结构示例图，图 5-1(a)是一棵只有一个根结点的树；图 5-1(b)是一棵只含有一棵子树的树；图 5-1(c)是一棵含有 3 棵子树的树。

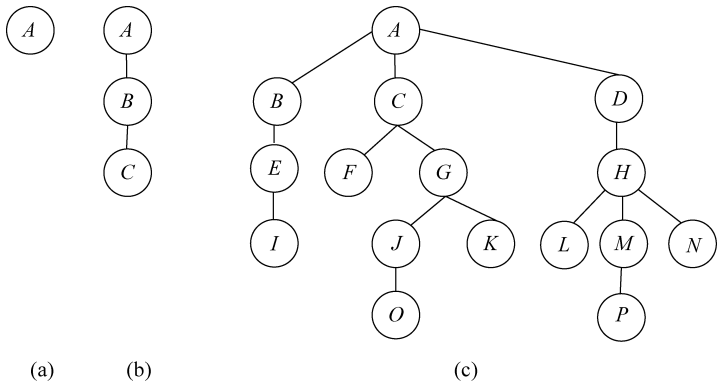


图 5-1 树的示例图(树形表示法)

从图 5-1 可见，在树形表示法中，树中结点用圆圈表示，圆圈内的符号代表该结点的数据元素，连接结点的连线代表边。在非空树的顶层总是只有一个结点，它通过边连接到第二层的多个结点，然后第二层结点再连向第三层的多个结点，以此类推，形成了一棵直观形象上倒置的树(树的根结点在上，树的叶结点在下)。根据树的定义，树的抽象数据类型描述如下：

ADT Tree {

数据对象：D = { a_i | $a_i \in$ TElemType, $1\leqslant i\leqslant n$, $n\geqslant 0$, TElemType 是约定的树中结点的数据元素类型，使用时用户需根据具体情况进行自定义}

数据关系：R = {<a_i, a_j> | a_i, $a_j \in$ D, $1\leqslant i\leqslant n$, $1\leqslant j\leqslant n$, 其中每个结点只有一个前驱(除根结点)，可以有零个或多个后继，有且仅有一个结点(根结点)没有前驱}

基本操作：

Root(T)，**求根结点操作**：求树 T 中的根结点并返回其值。

Value(T,cur_e),**求元素值操作**:求树 T 中当前结点 cur_e 的元素值。

Parent(T,cur_e),**求双亲结点操作**:求树 T 中当前结点 cur_e 的双亲结点。若 cur_e 是树 T 的非根结点,则返回它的双亲结点;否则,函数值为空。

LeftChild(T,cur_e),**求最左孩子操作**:求树 T 中当前结点 cur_e 的最左孩子。若 cur_e 是树 T 的非叶结点,则返回它的最左孩子;否则,函数值为空。

RightSibling(T,cur_e),**求右兄弟操作**:求树 T 中当前结点 cur_e 的右兄弟。若 cur_e 有右兄弟,则返回它的右兄弟;否则,函数值为空。

TreeEmpty(T),**判空操作**:判断树 T 是否为空。若为空,则函数返回 TRUE;否则,函数返回 FALSE。

TreeDepth(T),**求深度操作**:求树 T 的深度并返回其值。

TraverseTree(T,Visit()),**遍历操作**:按照某种次序对树 T 中的每个结点调用函数 Visit()一次且至多一次。其中,Visit()是对结点操作的应用函数,一旦 Visit()失败,则操作失败。

InitTree(&T),**初始化操作**:创建空树 T。

CreateTree(&T,definition),**构造操作**:按照 definition 构造树 T。其中,definition 给出树 T 的定义。

Assign(T,&cur_e,value),**赋值操作**:给树 T 中当前结点 cur_e 赋为 value。

InsertChild(&T,&p,i,c),**插入操作**:插入 c 为树 T 中 p 所指结点的第 i 棵子树。其中,p 指向树 T 中的某个结点,1≤i≤(p 所指结点的度 + 1),非空树 c 与树 T 不相交。

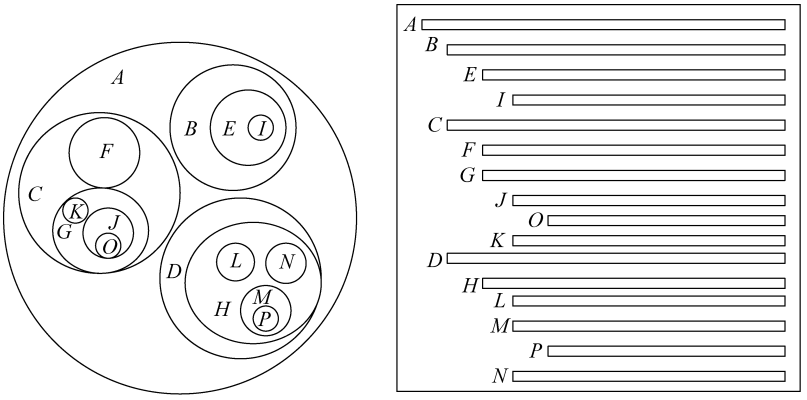
ClearTree(&T),**清空操作**:将已经存在的树 T 置成空树。

DestroyTree(&T),**销毁操作**:释放已经存在的树 T 的存储空间。

DeleteChild(&T,&p,i),**删除操作**:删除树 T 中 p 所指结点的第 i 棵子树。其中,p 指向树 T 中的某个结点,1≤i≤(p 所指结点的度)。

}ADT Tree

树的表示方法可以有多种。常见的有:树形表示法、文氏图表示法、凹入图表示法和广义表(括号)表示法等。图 5-1 所示的就是树形表示法。图 5-2(a)、图 5-2(b)和图 5-2(c)分别是用文氏图表示法、凹入图表示法和广义表(括号)表示法表示的图 5-1(c)中树的形式。



(a) 文氏图表示法　　(b) 凹入图表示法

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

(c) 广义表(括号)表示法

图 5-2　树的不同表示方法

2. 树的常用术语

为了使读者更易理解后续内容,需要首先知道一些有关树的常用术语。树的常用术语中大部分都是以现实世界中树的相关名词或以家谱中的成员关系命名,所以记忆起来并

不难。

(1) 树的结点。树的结点是由一个数据元素及关联其子树的边所组成的。例如，图 5-1(a)中只有 1 个结点，图 5-1(b)中有 3 个结点，图 5-1(c)中有 16 个结点。单个结点是一棵树，这棵树的根结点就是该结点本身。空树中没有结点。在实际应用中，结点中的数据元素通常代表着一些实体，例如，人、单位部门、汽车零件等，结点中的边代表着实体与实体之间的逻辑关系。虽然边没有方向(未带有箭头)，但它仍是有向的，其隐含的方向是从上到下，即上层结点是下层结点的前驱结点，下层结点是上层结点的后继结点。

(2) 结点的路径。结点的路径是指从根结点到某个结点所经历的所有结点和分支的顺序排列。例如，图 5-1(c)中结点 J 的路径是 $A \rightarrow C \rightarrow G \rightarrow J$。

(3) 路径的长度。路径的长度是指结点的路径中所包括的分支数。例如，图 5-1(c)中结点 J 的路径长度为 3。

(4) 结点的度。一个结点的度是指该结点所拥有的子树的个数。例如，图 5-1(c)中结点 A 的度为 3，结点 B 的度为 1，结点 C 的度为 2，结点 I、O、P 的度都为 0。

(5) 树的度。一棵树的度是指该树中结点的最大度数。例如，图 5-1(a)所示的树的度为 0，图 5-1(b)所示的树的度为 1，图 5-1(c)所示的树的度为 3。

(6) 叶结点(终端结点)。叶结点是指一棵树中度为 0 的结点，叶结点也称为终端结点。例如，图 5-1(c)中结点 I、F、O、K、L、P、N 都是叶结点。

(7) 分支结点(非终端结点)。分支结点是指一棵树中度不为 0 的结点，分支结点也称为非终端结点。树中除叶结点之外的所有结点都是分支结点。

(8) 孩子结点(子结点)。一个结点的孩子结点是指这个结点的子树的根结点。例如，图 5-1(c)中结点 B、C、D 是结点 A 的孩子结点，或者说结点 A 的孩子结点是 B、C、D。

(9) 双亲结点(父结点)。一个结点的双亲结点是指：若树中某个结点有孩子结点，则这个结点就称为是该孩子结点的双亲结点。例如，图 5-1(c)中结点 A 是结点 B、C、D 的双亲结点。双亲结点和孩子结点是具有前驱和后继关系的两个结点。其中，双亲结点是孩子结点的前驱；孩子结点是双亲结点的后继。

(10) 子孙结点。一个结点的子孙结点是指该结点的所有子树中的任意结点。例如，图 5-1(c)中结点 H 的子孙结点有 L、M、N、P。

(11) 祖先结点。一个结点的祖先结点是指该结点的路径中除该结点之外的所有结点。例如，图 5-1(c)中结点 P 的祖先结点有结点 A、D、H、M。在树中，子孙结点和祖先结点是对父子关系的拓展，它们定义了树中各个结点之间的纵向次序。

(12) 兄弟结点，堂兄弟结点。兄弟结点是指具有同一个双亲的结点。例如，图 5-1(c)中结点 B、C、D 是兄弟结点，它们的双亲都是结点 A；结点 L、M、N 也是兄弟结点，它们的双亲都是结点 H。双亲在同一层次上的结点互为堂兄弟。例如，图 5-1(c)中结点 E 和 F 互为堂兄弟，因为结点 E 的双亲结点 B 和结点 F 的双亲结点 C 是兄弟结点。

(13) 结点的层次。结点的层次从根结点开始算起，假设根结点的层次数为 1，则其余结点的层次数是其双亲结点的层次数加 1。例如，图 5-1(c)中结点 P 的层次数为 5，也可称结点 P 在树中处于第 5 层。

(14) 树的深度(树的高度)。树的深度是指树中所有结点的层次数的最大值加 1。例

如,图5-1(a)中树的深度为1,图5-1(b)中树的深度为3,图5-1(c)中树的深度为5。

(15) 有序树。有序树是指树中各个结点的所有子树之间,从左到右有严格的次序关系,不能互换。通常在有序树中,某个结点的所有孩子结点从左到右有长幼之分。也就是说,如果子树的次序不同,则对应着不同的有序树。图5-3所示的是两棵不同的有序树,它们的不同点在于结点A的两棵子树的左右次序不相同。

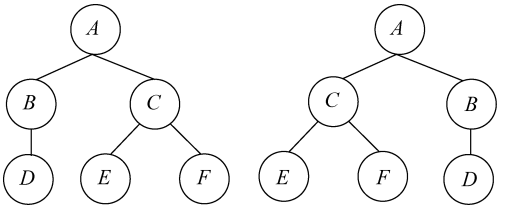


图5-3 两棵不同的有序树

(16) 无序树。与有序树相反,无序树是指树中各个结点的所有子树之间没有严格的次序关系。按前面树的定义可知,树是无序树,即树中的结点从左到右没有次序之分,其次序可以任意颠倒。

(17) 森林。森林是指由$m(m\geqslant 0)$棵互不相交的树所构成的集合。删除一棵树的根结点,这棵树就变成了一个森林;反之,在一个森林中加上一个结点作为根结点,这个森林就变成了一棵树。

(18) 同构。有两棵树,若通过对树中的各个结点适当地命名,使得这两棵树完全相等(结点的位置一一对应),则这两棵树被认为是同构的。

5.2 二叉树的基本概念

5.2.1 二叉树的概念

二叉树是一种特殊的树,它的每个结点最多只有两棵子树,并且这两棵子树也是二叉树。由于二叉树中的两棵子树有左、右之分,所以二叉树是有序树。下面给出二叉树的具体定义。

1. 二叉树的定义

二叉树是由$n(n\geqslant 0)$个结点所构成的有限集合。当$n=0$时,这个集合为空,此时的二叉树为空二叉树;当$n>0$时,这个集合由一个根结点(Root)和两棵互不相交的分别称为左子树和右子树的二叉树构成。可见,二叉树也是递归定义的。

二叉树和树是两种不同的树形结构,二叉树不是树的特殊情形。它们之间的主要区别在于以下几个方面。

(1) 二叉树和树的定义不同。尽管二叉树和树都是递归定义,但树是包含$n(n\geqslant 0)$个结点的有限集合,该集合或为空集(空树),或由一个根结点和$m(m\geqslant 0)$棵互不相交的子树组成;而二叉树是包含由$n(n\geqslant 0)$个结点的有限集合,该集合或为空集(空二叉树),或由一个根结点和两棵互不相交的、分别被称为左子树和右子树的二叉树组成。

(2) 二叉树和度为2的有序树不同。树中的子树通常是不分次序的。假设一棵树是有

序树，虽然树中各个结点的孩子结点之间是有长幼之分，但若某个结点只有一个孩子结点，就不需要区分长幼关系；而在二叉树中，即使只有一个孩子结点，也必须有严格的左、右之分。图 5-4 给出了二叉树的 5 种基本形态。

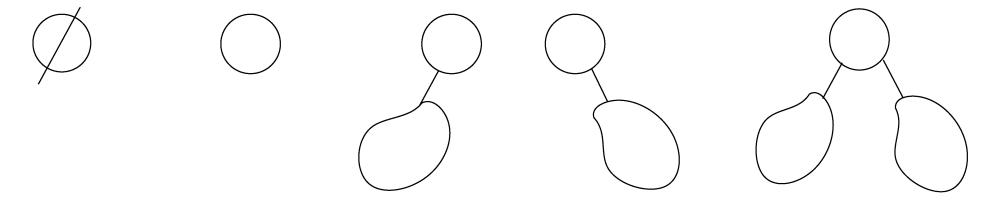


图 5-4　二叉树的 5 种基本形态

（3）二叉树与度小于等于 2 的树不同。在二叉树中，允许某些结点只有右子树而没有左子树；而在树中，一个结点若没有第一棵子树，则它就不可能有第二棵子树存在。图 5-5 和图 5-6 给出了只含有 3 个结点的二叉树和树的不同形态。

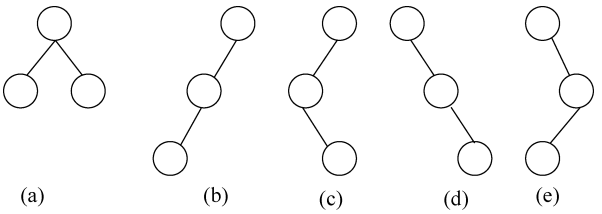


图 5-5　具有 3 个结点的二叉树的 5 种形态

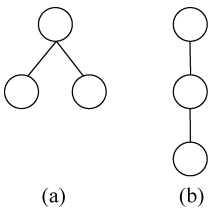


图 5-6　具有 3 个结点的树的两种形态

根据二叉树的定义，二叉树的抽象数据类型描述如下：

ADT BinaryTree {

数据对象：D = { a_i | a_i ∈ TElemType, 1≤i≤n, n≥0, TElemType 是约定的二叉树中结点的数据元素类型，使用时用户需根据具体情况进行自定义}

数据关系：R = {< a_i, a_j > | a_i, a_j ∈ D, 1≤i≤n, 1≤j≤n, 其中每个结点只有一个前驱(除根结点)，可以有零个、一个或两个后继，有且仅有一个结点(根结点)没有前驱}

基本操作：

Root(T)，**求根结点操作**：求二叉树 T 中的根结点并返回其值。

Value(T,e)，**求元素值操作**：求二叉树 T 中当前结点 e 的元素值。

Parent(T,e)，**求双亲结点操作**：求二叉树 T 中当前结点 e 的双亲结点。若 e 是二叉树 T 的非根结点，则返回它的双亲结点；否则，函数值为空。

LeftChild(T,e)，**求左孩子操作**：求二叉树 T 中当前结点 e 的左孩子。若 e 无左孩子，则返回为空。

RightChild(T,e)，**求右孩子操作**：求二叉树 T 中当前结点 e 的右孩子。若 e 无右孩子，则返回为空。

BiTreeEmpty(T)，**判空操作**：判断二叉树 T 是否为空。若为空，则函数返回 TRUE；否则，函数返回 FALSE。

BiTreeDepth(T)，**求深度操作**：求二叉树 T 的深度并返回其值。

PreOrderTraverse(T,Visit())，**先根遍历操作**：按照先根遍历次序对二叉树 T 中的每个结点调用函数 Visit()一次且至多一次。其中，Visit()是对结点操作的应用函数，一旦 Visit()失败，则操作失败。

InOrderTraverse(T,Visit())，**中根遍历操作**：按照中根遍历次序对二叉树 T 中的每个结点调用函数 Visit()一次且至多一次。其中，Visit()是对结点操作的应用函数，一旦 Visit()失败，则操作失败。

PostOrderTraverse(T,Visit()),**后根遍历操作**:按照后根遍历次序对二叉树T中的每个结点调用函数Visit()一次且至多一次。其中,Visit()是对结点操作的应用函数,一旦Visit()失败,则操作失败。

LevelOrderTraverse(T,Visit()),**层次遍历操作**:按照层次遍历次序对二叉树T中的每个结点调用函数Visit()一次且至多一次。其中,Visit()是对结点操作的应用函数,一旦Visit()失败,则操作失败。

InitBiTree(&T),**初始化操作**。创建空二叉树T。

CreateBiTree(&T,definition),**构造操作**:按照definition构造二叉树T。其中,definition给出二叉树T的定义。

Assign(T,&e,value),**赋值操作**:给二叉树T中当前结点e赋值为value。

InsertChild(&T,&p,LR,c),**插入操作**:根据LR为0或1,插入c为二叉树T中p所指结点的左子树或右子树,p所指结点的原有左子树或右子树则称为c的右子树。其中,p指向二叉树T中的某个结点,LR为0或1,非空二叉树c与T不相交且右子树为空。

ClearBiTree(&T),**清空操作**:将已经存在的二叉树T置成空二叉树。

DestroyBiTree(&T),**销毁操作**:释放已经存在的二叉树T的存储空间。

DeleteChild(&T,&p,LR),**删除操作**:根据LR为0或1,删除二叉树T中p所指结点的左子树或右子树。其中,p指向二叉树T中的某个结点,LR为0或1。

}ADT BinaryTree

二叉树的表示方法与树的表示方法相同,也可以有多种。常见的有:树形表示法、文氏图表示法、凹入图表示法和广义表(括号)表示法等。

2. 二叉树的特殊形式

1) 满二叉树

满二叉树是二叉树的一种特殊形态。如果在一棵二叉树中,它的所有结点或者是叶结点,或者是左、右子树都非空,并且所有叶结点都在同一层上,则称这棵二叉树为满二叉树,如图5-7(a)所示。可以对满二叉树中的所有结点按层次自上到下、由左向右进行编号,约定根结点的编号为1,在图5-7(a)中每个结点边上的数字就是该结点的编号。

满二叉树的特点如下。

(1) 叶结点只能出现在最下层上。

(2) 只有度为0或度为2的结点,不存在度为1的结点。

(3) 在具有相同深度的多棵二叉树中,满二叉树的结点个数最多,其叶结点的个数也最多。

2) 完全二叉树

完全二叉树也是二叉树的一种特殊形态。如果在一棵具有 n 个结点的二叉树中,它的逻辑结构与满二叉树的前 n 个结点的逻辑结构相同,则称这样的二叉树为完全二叉树,如图5-7(b)所示。

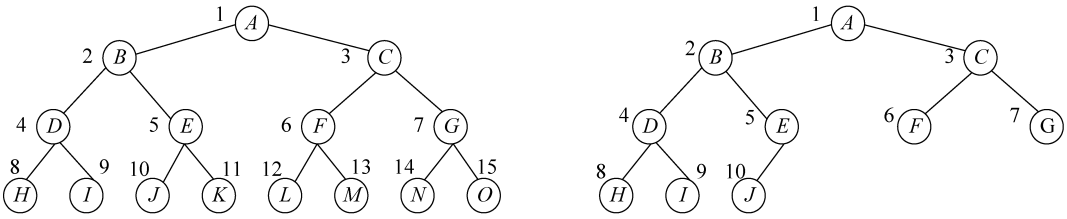
完全二叉树的特点如下:

(1) 叶结点或出现在最下层上,并且它们都集中在左边连续的位置上;或出现在倒数第二层上,并且它们都集中在右边连续的位置上。

(2) 若存在度为1的结点,它只可能存在一个,并且该结点只有左孩子,而无右孩子。

(3) 深度为 h 的完全二叉树,其倒数第二层上应是满二叉树。

(4) 当结点总数为奇数时,度为1的结点个数为0;当结点总数为偶数时,度为1的结点个数为1。



(a) 满二叉树　　(b) 完全二叉树

图 5-7　满二叉树和完全二叉树

(5) 完全二叉树是具有 n 个结点的多棵二叉树中深度最小的一棵树。

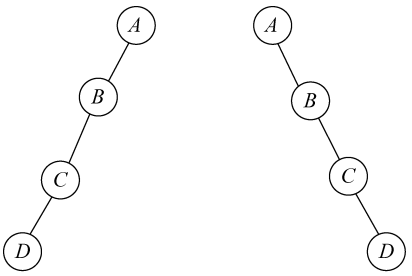
满二叉树是完全二叉树的一种特例，并且完全二叉树与相同深度的满二叉树对应位置的结点具有同样的编号。深度为 h 的完全二叉树，其最少结点的情况是前 $h-1$ 层是满二叉树，而第 h 层上只有一个结点；其最多结点的情况是构成一棵 h 层的满二叉树。

注意：满二叉树一定是完全二叉树，但完全二叉树不一定是满二叉树。

3) 单分支树(斜树)

若二叉树的所有结点都没有右孩子或都没有左孩子，则称此二叉树为单分支树(斜树)。单分支树又可分为左支树(左斜树)和右支树(右斜树)。其中，所有结点都没有右孩子的二叉树称为左支树；所有结点都没有左孩子的树称为右支树。图 5-8 所示分别是含有 4 个结点的左支树和右支树。

注意：由于单分支树的每一层上都只有一个结点，故其结点个数和深度相同，即具有 n 个结点的单分支树的深度为 n。



(a) 左支树　　(b) 右支树

图 5-8　单分支树

5.2.2　二叉树的性质

性质 1　二叉树中第 $i(i\geqslant 1)$ 层上的结点数最多为 2^{i-1}。

证明：用数学归纳法证明如下。

当 $i=1$ 时，$2^{i-1}=2^0=1$，因为二叉树中的第一层只有一个根结点，所以命题正确。

假设对所有的 $j(1\leqslant j<i)$ 命题成立，即第 j 层上最多有 2^{j-1} 个结点。下面需要证明：当 $j=i$ 时，命题成立。根据归纳假设，第 $i-1$ 层上的结点数最多为 2^{i-2}。由于二叉树中的每个结点最多有两个孩子结点，所以第 i 层上的结点数最多是 2^{i-2} 的 2 倍，即第 i 层上的结点个数最多为 $2\times 2^{i-2}=2^{i-1}$ 个，故命题成立。

性质 2　深度为 $h(h\geqslant 1)$ 的二叉树中最多有 2^h-1 个结点。

证明：由性质 1 可知，第 i 层的最大结点数为 2^{i-1}，再将二叉树中各层的结点数相加，即可得到：深度为 h 的二叉树中结点的总数最多为 $2^h-1=2^0+2^1+2^2+\cdots+2^{h-1}$ 个，故性质 2 得证。

性质 3　对于任何一棵二叉树，若其叶结点的个数为 n_0，度为 2 的结点个数为 n_2，则有 $n_0=n_2+1$。

证明：设二叉树中度为1的结点个数为n_1，二叉树的结点总数为n，则有：

$$n = n_0 + n_1 + n_2 \tag{5-1}$$

再设二叉树中具有e个分支，而度为1的结点是引出一个分支，度为2的结点是引出两个分支，则有：

$$e = n_1 + 2 \times n_2 \tag{5-2}$$

又因为在二叉树中除根结点外，每个结点均对应一个进入它的分支，所以有：

$$n = e + 1 \tag{5-3}$$

最后根据式(5-1)、(5-2)和(5-3)，整理后得到：

$$n_0 = n_2 + 1 \tag{5-4}$$

故性质3得证。

性质4 具有n个结点的完全二叉树，其深度为$\lfloor \log_2 n \rfloor + 1$或者$\lceil \log_2 (n+1) \rceil$(其中，符号$\lfloor x \rfloor$表示不大于$x$的最大整数，也称为对$x$向下取整，符号$\lceil x \rceil$表示不小于$x$的最小整数也称为对$x$向上取整)。

证明：假设具有n个结点的完全二叉树的深度为h。由性质2可知，深度为$h-1$的满二叉树的结点总数为$2^{h-1}-1$，深度为h的满二叉树的结点总数2^h-1。再根据完全二叉树的定义可得：

$$2^{h-1}1 < n \leqslant 2^h - 1 \tag{5-5}$$

在式(5-5)的两边加1，可得：

$$2^{h-1} \leqslant n < 2^h \tag{5-6}$$

对式(5-6)的各项取对数，可得：

$$h - 1 \leqslant \log_2 n < h \tag{5-7}$$

因为h为整数，所以$h-1=\lfloor \log_2 n \rfloor$，即$h=\lfloor \log_2 n \rfloor + 1$，故性质4得证(读者可以自行推导公式得到$\lceil \log_2 (n+1) \rceil$)。

性质5 对于具有n个结点的完全二叉树，若从根结点开始自上到下并且按照层次由左向右对结点从1开始进行编号，则对于任意一个编号为$i(1 \leqslant i \leqslant n)$的结点有：

(1) 若$i=1$，则编号为i的结点是二叉树的根结点，它没有双亲；若$i>1$，则编号为i的结点，其双亲的编号为$\lfloor i/2 \rfloor$。

(2) 若$2i>n$，则编号为i的结点无左孩子；否则，编号为$2i$的结点就是其左孩子。

(3) 若$2i+1>n$，则编号为i的结点无右孩子；否则，编号为$2i+1$的结点就是其右孩子。

证明：下面用数学归纳法来证明(2)和(3)，(1)可由结论(2)和(3)推导得到。

当$i=1$时，由完全二叉树的定义可知，若其左孩子存在，则左孩子结点的编号为$2i=2$；若其右孩子存在，则右孩子结点的编号为$2i+1=3$。若$2i>n$，则说明不存在编号为$2i$的结点，即编号为i的结点没有左孩子。同理，若$2i+1>n$，则说明不存在编号为$2i+1$的结点，即编号为i的结点没有右孩子。所以命题成立。

假设当$i=j(j \geqslant 1)$时，命题成立，即若$2j>n$，则编号为j的结点无左孩子；否则，编号为$2j$的结点就是其左孩子结点；若$2j+1>n$，则编号为i的结点无右孩子，否则编号为$2j+1$的结点就是其右孩子结点。下面需证明当$i=j+1$时命题成立。

当$i=j+1$时，由完全二叉树的定义，若其左孩子存在，则左孩子结点的编号一定是编

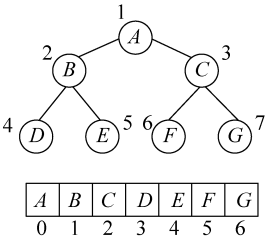
号为 j 的右孩子结点的编号加 1，即为 $(2j+1)+1=2(j+1)=2i$，且有 $2i \leqslant n$，若 $2i>n$，则说明其左孩子不存在；若其右孩子存在，则右孩子结点的编号一定是其左孩子结点的编号加 1，即 $2i+1$，且有 $2i+1 \leqslant n$；若 $2i+1>n$，则说明其右孩子不存在。故(2)和(3)得证。

5.2.3 二叉树的存储结构

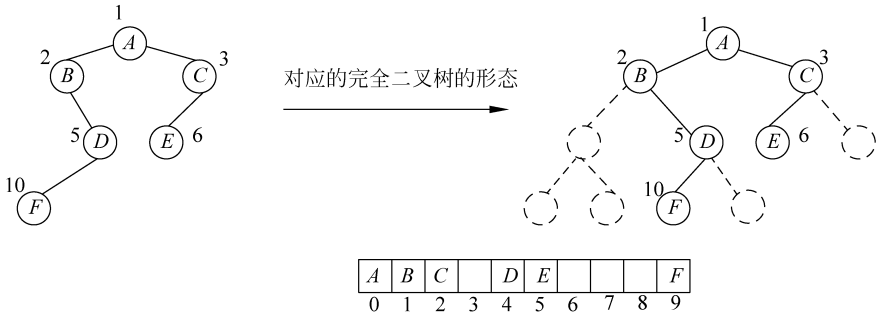
二叉树的存储实现方法有多种，但归纳起来主要分为顺序存储和链式存储两大类。下面详细介绍二叉树的顺序存储结构和链式存储结构。

1. 二叉树的顺序存储结构

二叉树的顺序存储是指按照某种顺序依次将二叉树中的各个结点的数据元素存放在一组地址连续的存储单元中。由于二叉树是非线性结构，所以需要将二叉树中的各个结点先按照一定的顺序排列成一个线性序列，再通过这些结点在线性序列中的相对位置，确定二叉树中各个结点之间的逻辑关系。由二叉树的性质 5 可知，对于一棵完全二叉树来说，可以从根结点开始自上而下并按照层次由左向右对结点依次进行编号，然后按照编号顺序依次将其存放在一维数组中，其结点之间的逻辑关系可由性质 5 中双亲结点与孩子结点编号之间的关系式计算得到，如图 5-9(a)所示。对于一棵非完全二叉树来说，可以先在此树中增加一些并不存在的虚结点并使其成为一棵完全二叉树，然后用与完全二叉树相同的方法对结点进行编号，再将编号为 i 的结点的值存放到数组下标为 $i-1$ 的数组单元中，虚结点不存放任何值，如图 5-9(b)所示。



(a) 完全二叉树的顺序存储



(b) 非完全二叉树的顺序存储

图 5-9　二叉树的顺序存储结构

二叉树的顺序存储结构的优点是结构简单、查找方便；缺点是浪费存储空间，特别是当二叉树需要经常进行插入、删除结点操作时，会引起大量元素的移动。对于满二叉树和完全二叉树而言，顺序存储结构是一种最简单、最节省空间的存储方式，而且能使操作变得简单，

所以这种方式非常适用于满二叉树和完全二叉树。但对于非完全二叉树,由于有“虚结点”的存在,从而造成了存储空间的浪费,特别是对右支树来说,若其深度为 h,则此右支树只有 h 个结点,却要为此分配 2^h-1 个存储单元,这种情况下的存储空间浪费最大。

二叉树的顺序存储结构描述:

```
#define MAX_TREE_SIZE 100                          //二叉树的最大结点数
typedef TElemType SqBiTree[MAX_TREE_SIZE];         //0 号单元存储根结点
```

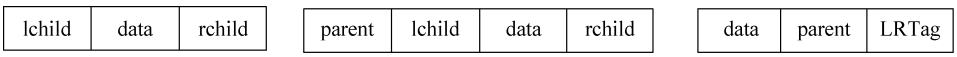
2. 二叉树的链式存储结构

二叉树的链式存储是指将二叉树的各个结点随机地存放在位置任意的内存空间中,各个结点之间的逻辑关系通过指针来反映。由于二叉树中的任意一个结点至多只有一个双亲结点和两个孩子结点,所以在用链式存储方式来实现二叉树的存储时,可以有 3 种形式。第 1 种是二叉链表存储结构;第 2 种是三叉链表存储结构;第 3 种是双亲链表存储结构。

在二叉链表存储结构中,二叉树中的每个结点设置有 3 个域:数据域、左孩子域和右孩子域。其中,数据域用来存放结点的数据元素;左、右孩子域分别用来存放该结点的左、右孩子结点的存储地址。其结点的结构如图 5-10(a)所示。

三叉链表存储结构是指在二叉链表的结点结构中增加一个双亲结点域,该域用来存放该结点的双亲结点的存储地址。其结点的结构如图 5-10(b)所示。

双亲链表存储结构中的每个结点存放的信息除包含结点本身的数据域信息以外,还包含指示双亲结点在存储结构中的位置信息,以及该结点是双亲结点的左孩子还是右孩子的信息,即二叉树中的每个结点设置有 3 个域:数据域、双亲结点位置域、左右孩子标志域。其结点的结构如图 5-10(c)所示。



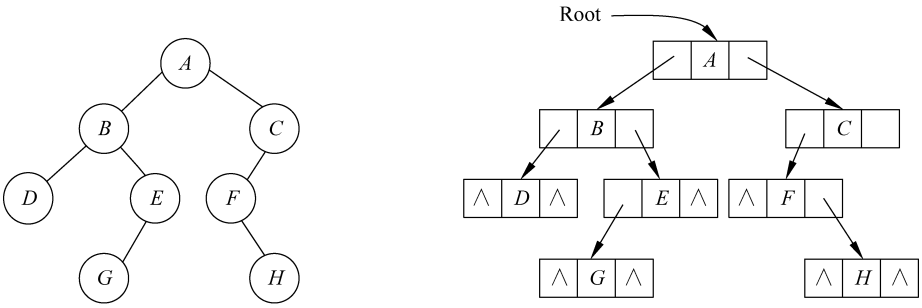
(a) 二叉链表的结点结构　　(b) 三叉链表的结点结构　　(c) 双亲链表的结点结构

图 5-10　二叉树链式存储的结点结构

图 5-11、图 5-12 所示的是一棵二叉树,以及其二叉链表、三叉链表、双亲链表的存储结构。二叉树的 3 种链式存储结构中,二叉链表存储结构的优点是方便、直观,进行插入、删除结点操作时不需要移动结点;缺点是结点的遍历操作较困难。相对于二叉链表存储结构,三叉链表存储结构既便于查找孩子结点,又便于查找双亲结点,但它增加了存储空间的开销。采用双亲链表存储结构实现查找一个指定结点的双亲结点是容易的,但要实现查找一个指定结点的孩子结点却不容易,需要扫描一遍整个链表。因此,在实际应用中,二叉链表存储结构是二叉树最常用的存储结构。后面有关二叉树的算法都是在以二叉链表为存储结构的前提下设计的。

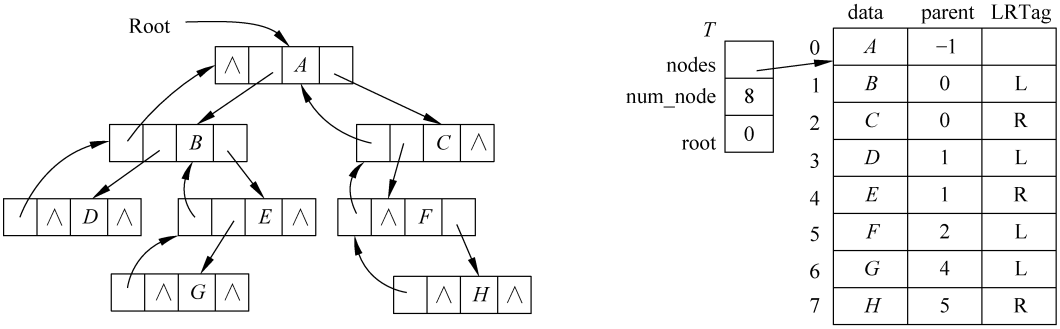
二叉树的二叉链表存储结构与单链表一样,也可分不带头结点和带头结点两种情况(通常采用的是不带头结点的形式)。图 5-13 所示是图 5-11(a)中二叉树的不带头结点和带头结点的二叉链表存储结构。

注意:具有 n 个结点的二叉树若采用不带头结点的二叉链表存储结构存放结点时,共有 $2n$ 个指针域,其中只有 $n-1$ 个指针域用来指向结点的左孩子和右孩子,其余 $n+1$ 个指针域为空。



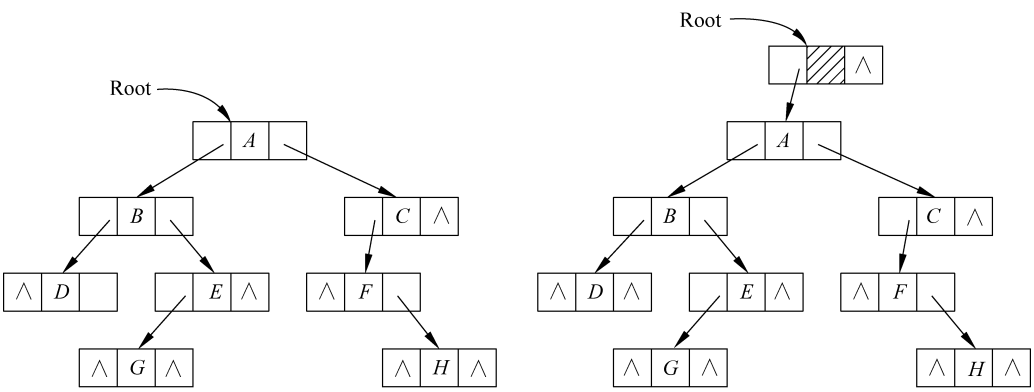
(a) 一棵二叉树　　(b) 二叉树的不带头结点的二叉链表存储结构

图 5-11　二叉树及其二叉链表存储结构



(a) 二叉树的三叉链表存储结构　　(b) 二叉树的双亲链表存储结构

图 5-12　图 5-11(a)中二叉树及其三叉链表存储结构、双亲链表存储结构



(a) 二叉树的不带头结点的二叉链表存储结构　　(b) 二叉树的带头结点的二叉链表存储结构

图 5-13　图 5-11(a)中二叉树的不带头结点和带头结点的二叉链表存储结构

二叉树的二叉链表存储结构描述如下：

```
typedef struct BiTNode {                    //结点结构
  TElemType data;                           //数据域
  struct BiTNode * lchild, * rchild;        //左孩子域和右孩子域
}BiTNode, * BiTree;
```

二叉树的三叉链表存储结构描述如下：

```
typedef struct TriTNode {                          //结点结构
  TElemType data;                                  //数据域
  struct TriTNode * lchild, * rchild;              //左孩子域和右孩子域
  struct TriTNode * parent;                        //双亲结点域
}TriTNode, * TriTree;
```

二叉树的双亲链表存储结构描述如下：

```
typedef struct BPTNode {                           //结点结构
  TElemType data;                                  //数据域
  int parent;                                      //双亲结点位置域
  char LRTag;                                      //左右孩子标志域
}BPTNode;
typedef struct BPTree {                            //二叉树结构
  BPTNode * nodes;                                 //指向存放结点存储空间的首地址
  int num_node;                                    //结点数目
  int root;                                        //根结点的位置
}BPTree;
```

5.3 二叉树的遍历

二叉树的遍历是指沿着某条搜索路径对二叉树中的每个结点进行访问，使得每个结点均被访问一次，而且仅被访问一次。其中，“访问”的含义较为广泛，它可以是输出二叉树中的结点信息，也可以是对结点进行任何其他处理。

二叉树的遍历操作是二叉树中最基本的操作，也是二叉树其他操作的基础。下面分别介绍二叉树的遍历方法及其实现，还有几个二叉树遍历算法的典型应用。

5.3.1 二叉树的遍历方法及其实现

1. 二叉树的遍历方法

根据二叉树的结构特点，可以将一棵二叉树划分成 3 个部分，即根结点、左子树和右子树。其次，二叉树中的所有结点都有层次之分。因此，对于一棵二叉树来说，它有 3 条搜索路径，分别是先上后下、先左(子树)后右(子树)和先右(子树)后左(子树)。如果规定用 D、L 和 R 分别表示访问根结点、处理左子树和处理右子树，则可得到二叉树的 7 种遍历方法：层次遍历、DLR、LDR、LRD、DRL、RDL 和 RLD。其中，第 1 种方法是按先上后下且同一层次按先左后右的路径顺序得到的，第 2 至第 4 种方式是按先左后右的路径顺序得到的，第 5 至第 7 种方法则是按先右后左的路径顺序得到的。由于先左后右和先右后左的遍历操作在算法设计上没有本质区别，并且通常对子树的处理也总是按照先左后右的顺序进行，所以下面只给出前面 4 种遍历方法的定义。

1) 层次遍历操作

若二叉树为空，则为空操作；否则，先访问第 1 层的根结点，然后从左到右依次访问

第 2 层的每个结点。以此类推，当第 i 层的所有结点访问完后，再从左到右依次访问第 $i+1$ 层的每个结点，直到最后一层的所有结点都访问完为止。

根据根结点访问的先后次序不同，这里将 DLR 称为先根遍历或先序遍历，LDR 称为中根遍历或中序遍历，LRD 称为后根遍历或后序遍历。由于二叉树是递归定义的，所以 DLR、LDR、LRD 3 种遍历方法给出的定义也都是递归定义的。

2) 先根遍历(DLR)操作

若二叉树为空，则为空操作；否则：

(1) 访问根结点 D。

(2) 先根遍历左子树 L。

(3) 先根遍历右子树 R。

3) 中根遍历(LDR)操作

若二叉树为空，则为空操作；否则：

(1) 中根遍历左子树 L。

(2) 访问根结点 D。

(3) 中根遍历右子树 R。

4) 后根遍历(LRD)操作

若二叉树为空，则为空操作；否则：

(1) 后根遍历左子树 L。

(2) 后根遍历右子树 R。

(3) 访问根结点 D。

采用不同方法对二叉树进行遍历可以得到二叉树结点的不同线性序列。例如，对于图 5-11(a)所示的二叉树，其层次遍历序列为 $ABCDEFGH$，先根遍历序列为 $ABDEGCFH$，中根遍历序列为 $DBGEAFHC$，后根遍历序列为 $DGEBHFCA$。

可见，经过对二叉树遍历后，可将非线性结构的二叉树转变成线性序列，从而可以确定二叉树中某个指定结点在某种遍历序列中的前驱和后继。例如，对于图 5.11(a)所示的二叉树中的结点 B，它在层次遍历序列中的前驱结点是 A，后继结点是 C；在先根遍历序列中的前驱结点是 A，后继结点是 D；在中根遍历序列中的前驱结点是 D，后继结点是 G；在后根遍历序列中的前驱结点是 E，后继结点是 H。

值得一提的是，二叉树的遍历方法也可以分为深度优先遍历和广度优先遍历。其中，先根遍历、中根遍历和后根遍历属于深度优先遍历方法，层次遍历属于广度优先遍历方法。

2. 二叉树遍历的一个应用

针对二叉树进行不同的遍历后得到的不同的线性序列，往往具有特定的实际意义。例如，图 5-14 是表达式 $(A+B)*C-D/E$ 所对应的语法树，对此语法树分别进行先根遍历、中根遍历和后根遍历后，得到的遍历序列为 $-*+ABC/DE$、$A+B*C-D/E$ 和 $AB+C*DE/-$，它们就是这个表达式所对应的前缀表达式、去掉括号的中缀表达式和后缀表达式。关于由原表达式如何转换成后缀表达式，再由后缀表达式如何计算表达式的值的问题，

在前面3.1.5节中已有讨论。

3. 二叉树遍历操作实现的递归算法

根据二叉树遍历的递归定义,很容易给出实现二叉树遍历操作的递归算法。下面以二叉链表作为二叉树的存储结构,并规定访问结点的操作就是输出结点的值,分别给出二叉树的先根遍历、中根遍历和后根遍历的递归算法描述。

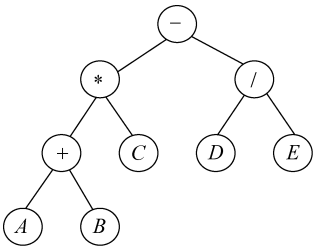


图5-14 表达式$(A+B)*C-D/E$的语法树

【算法5-1】 先根遍历操作实现的递归算法。

```
void PreRootTraverse(BiTree T)
//先根遍历二叉树 T 的递归算法
{  if(T!= NULL)
    {    printf(" % c",T -> data);                    //访问根结点
         PreRootTraverse(T -> lchild);                //先根遍历左子树
         PreRootTraverse(T -> rchild);                //先根遍历右子树
    }
}//算法 5 - 1 结束
```

【算法5-2】 中根遍历操作实现的递归算法。

```
void InRootTraverse(BiTree T)
//中根遍历二叉树 T 的递归算法
{  if(T!= NULL)
    {    InRootTraverse(T -> lchild);                 //中根遍历左子树
         printf(" % c",T -> data);                    //访问根结点
         InRootTraverse(T -> rchild);                 //中根遍历右子树
    }
}//算法 5 - 2 结束
```

【算法5-3】 后根遍历操作实现的递归算法。

```
void PostRootTraverse(BiTree T)
//后根遍历二叉树 T 的递归算法
    {  if(T!= NULL)
          {  PostRootTraverse(T -> lchild);           //后根遍历左子树
             PostRootTraverse(T -> rchild);           //后根遍历右子树
             printf(" % c",T -> data);                //访问根结点
    }
}//算法 5 - 3 结束
```

从上面给出的3种遍历二叉树的递归算法可知,它们只是访问根结点以及遍历左子树和右子树的先后次序不同。先根遍历是指每次进入一层递归调用时先访问根结点,然后再依次对它的左、右子树执行递归调用;中根遍历是指在执行完左子树的递归调用后再访问根结点,然后对右子树执行递归调用;后根遍历则是指执行完左、右子树的递归调用后,最后访问根结点。为了更为形象地对二叉树遍历的递归算法加以理解,这里给出二叉树先根遍历操作的递归算法的执行过程(如图5-15所示),读者可以自行给出其余两种遍历操作的递归算法的执行过程。

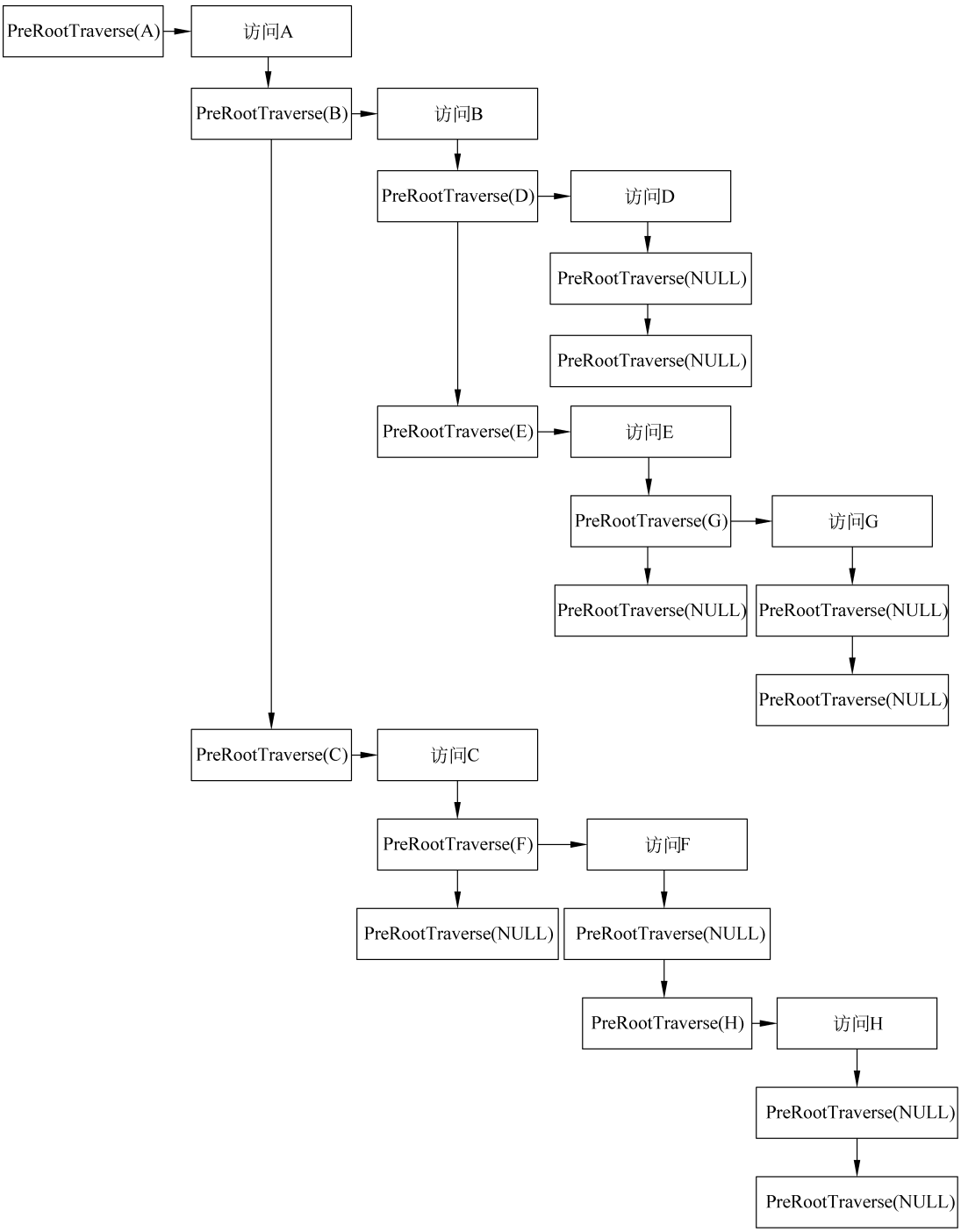


图 5-15　图 5-11(a)中二叉树的先根遍历操作的递归算法的执行过程

4. 二叉树遍历操作实现的非递归算法

前面给出的二叉树遍历算法都采用递归算法。递归算法虽然结构简洁,但在时间和空间开销上相对较大,从而导致运行效率较低,并且有些程序设计环境不支持递归,这就要求我们要将递归算法转换成非递归算法。将递归算法转换为非递归算法有两种方式:一种是直接转换法,不需要回溯;另一种是间接转换法,需要回溯。前者使用一些变量保存中间结果,递归过程用循环结构来替代;后者需要利用栈保存中间结果,故引入一个栈结构,并依照递归算法执行过程中编译栈的工作原理,得到相应的非递归算法。二叉树遍历操作的非递归实现采用的就是间接转换法。

1) 先根遍历操作的实现

先根遍历的递归过程是指先访问根结点,再沿着该结点的左子树向下依次访问其左子树的根结点,直到最后访问的结点都无左子树为止,再继续依次向上先根遍历根结点的右子树。

先根遍历的非递归过程则要借助一个栈来记载当前被访问结点的右孩子结点,以便遍历完一个结点的左子树后,能顺利地进入这个结点的右子树,继续进行遍历。

实现先根遍历操作的非递归算法的主要思想是:从二叉树的根结点出发,沿着该结点的左子树向下搜索,在搜索过程中每遇到一个结点就先访问该结点,并令该结点的非空右孩子结点入栈。当左子树结点访问完成后,从栈顶弹出结点的右孩子结点,然后用上述同样的方法去遍历该结点的右子树,以此类推,直到二叉树中所有的结点都被访问为止。其主要操作过程可描述如下。

(1) 创建一个栈对象,根结点入栈。

(2) 当栈非空时,将栈顶结点出栈并访问该结点。

(3) 对当前访问结点的非空左孩子结点相继依次访问,并将当前访问结点的非空右孩子结点压入栈内。

(4) 重复执行步骤(2)和(3),直到栈为空为止。

【算法 5-4】 先根遍历操作实现的非递归算法。

```
Status NPreRootTraverse(BiTree T)
//先根遍历二叉树 T 的非递归算法
{  SqStack S;                                          //采用顺序栈结构
   InitStack(S);                                       //初始化顺序栈
   while(!StackEmpty(S)||(T!= NULL))
    {  if (T!= NULL)                                   //若二叉树非空
          {  printf("%c",T->data);                     //访问根结点
             Push(S,T);                                //入栈
             T = T->lchild;                            //指向左子树
          }
       else                                            //若栈非空
          {  Pop(S,T);                                 //出栈
             T = T->rchild;                            //指向右子树
          }
   }
   return OK;
}//算法 5-4 结束
```

2）中根遍历操作的实现

中根遍历的非递归过程也要借助一个栈来记载遍历过程中所经历的而未被访问的所有结点，以便遍历完一个结点的左子树后能顺利地返回它的父结点。

实现中根遍历操作的非递归算法的主要思想是：从二叉树的根结点出发，沿着该结点的左子树向下搜索，在搜索过程中将所遇到的每个结点依次入栈，直到二叉树中最左下的结点入栈为止，然后从栈中弹出栈顶结点并对其进行访问，访问完后再进入该结点的右子树并用上述同样的方法去遍历该结点的右子树。以此类推，直到二叉树中所有的结点都被访问为止。其操作的实现过程描述如下。

（1）创建一个栈对象，根结点入栈。

（2）若栈非空，则将栈顶结点的非空左孩子相继入栈。

（3）栈顶结点出栈并访问非空栈顶结点，并使该栈顶结点的非空右孩子结点入栈。

（4）重复执行步骤(2)和(3)直到栈为空为止。

【算法 5-5】 中根遍历操作实现的非递归算法。

```
Status NInRootTraverse(BiTree T)
 //中根遍历二叉树 T 的非递归算法
 { SqStack S;                                          //采用顺序栈结构
   InitStack(S);                                       //初始化顺序栈
   while(!StackEmpty(S)||T!= NULL)
   {    if (T!= NULL)                                  //若二叉树非空
          { Push(S,T);                                 //入栈
            T = T -> lchild;                           //指向左子树
           }
        else
          {  Pop(S,T);                                 //出栈
             printf("%c",T -> data);                   //访问根结点
             T = T -> rchild;                          //指向右子树
           }
         }
   return OK;
}//算法 5 - 5 结束
```

3）后根遍历操作的实现

由于后根遍历是先处理左子树，后处理右子树，最后才访问根结点，所以在遍历搜索过程中也是从二叉树的根结点出发，沿着该结点的左子树向下搜索。在搜索过程中每遇到一个结点便判断该结点是否第一次经过，若是，则不立即访问，而是将该结点入栈保存，遍历该结点的左子树；当左子树遍历完毕后再返回该结点，这时还不能访问该结点，而是应继续进入该结点的右子树进行遍历；当左、右子树均遍历完毕后，才能从栈顶弹出该结点并访问它。由于在决定栈顶结点是否能访问时，需要知道该结点的右子树是否被遍历完毕，因此为解决这个问题，在算法中引入一个布尔型的访问标记变量 flag 和一个结点指针 p。其中，flag 用来标记当前栈顶结点是否被访问，当值为 TRUE 时，表示栈顶结点已被访问；当值为 FALSE 时，表示当前栈顶结点未被访问。指针 p 指向当前遍历过程中最后一个被访问的结点。若当前栈顶结点的右孩子结点是空，或者就是 p 指向的结点，则表明当前结点的右子树已遍历完毕，此时就可以访问当前栈顶结点。其操作的实现过程描述如下。

(1) 创建一个栈对象,根结点入栈,p 赋初始值 NULL。

(2) 若栈非空,则栈顶结点的非空左孩子相继入栈。

(3) 若栈非空,查看栈顶结点,若栈顶结点的右孩子为空,或者与 p 相等,则将栈顶结点出栈并访问它,同时使 p 指向该结点,并置 flag 值为 TRUE;否则,将栈顶结点的右孩子入栈,并置 flag 值为 FALSE。

(4) 若 flag 值为 TRUE,则重复执行步骤(3);否则,重复执行步骤(2)和(3),直到栈为空为止。

【算法 5-6】 后根遍历操作实现的非递归算法。

```
Status NPostRootTraverse(BiTree T)
//后根遍历二叉树 T 的非递归算法
{   SqStack S;                                  //采用顺序栈结构
    BiTNode * p;                                //定义指针 p
    int flag;                                   //定义标志 flag
    InitStack(S);                               //初始化顺序栈
    do
    {
       while(T!= NULL)                          //若二叉树非空
       {  Push(S,T);                            //入栈
          T = T -> lchild;                      //指向左子树
       }
       p = NULL;                                //p 指向栈顶元素的前一个已访问的结点
       flag = TRUE;                             //设置已访问过的标记
      while(!StackEmpty(S)&&(flag))
      {       T = * (S.top - 1);                //取栈顶元素
             if(T -> rchild == p)
              {        printf(" % c",T -> data);
                       S.top -- ;
                       p = T;                   //p 指向刚访问过的结点

              }
            else
              {        T = T -> rchild;         //指向右子树
                       flag = FALSE;            //设置未被访问的标记
              }
      }
    }while(!StackEmpty(S));
    return OK;
} //算法 5 - 6 结束
```

对于有 n 个结点的二叉树,上面 3 种遍历算法的时间复杂度都为 $O(n)$。如果对每个结点的处理时间是一个常数,那么遍历二叉树就可以在线性时间内完成。不管采用哪种算法,遍历二叉树操作的实现过程中所需的辅助空间为遍历过程中栈的最大容量,即树的高度。最坏情况下,具有 n 个结点的树的高度为 n,因此,3 种遍历算法的空间复杂度为 $O(n)$。

4) 层次遍历操作的实现

层次遍历操作的实现过程中需要使用一个队列作为辅助的存储结构。在遍历开始时,首先将根结点入队,然后每次从队列中取出队首元素进行处理。每处理一个结点,都是先访

问该结点，再按从左到右的顺序把它的孩子结点依次入队。这样，上一层的结点总排在下一层结点的前面，从而实现了二叉树的层次遍历。其操作的实现过程描述如下。

（1）创建一个队列对象，根结点入队。

（2）若队列非空，则将队首结点出队并访问该结点，再将该结点的非空左、右孩子结点依次入队。

（3）重复执行步骤(2)，直到队列为空为止。

【算法 5-7】 层次遍历操作实现的非递归算法。

```
Status DLevelOrderTraverse(BiTree T)
//层次遍历操作实现的非递归算法
{    SqQueue Q;
     InitQueue(Q);
     EnQueue(Q,T);                                  //根结点入队
     while(!QueueEmpty(Q))                          //当队列非空时
       {    DeQueue(Q,T);                           //队首元素出队
            printf("%c",T->data);
            if(T->lchild!=NULL)                     //有左孩子时将其入队
                EnQueue(Q,T->lchild);
            if(T->rchild!=NULL)                     //有右孩子时将其入队
                EnQueue(Q,T->rchild);
       }
     return OK;
 }//算法 5-7 结束
```

对于有 n 个结点的二叉树，层次遍历算法的时间复杂度为 $O(n)$。所需的辅助空间为遍历过程中队列所需的最大容量，而队列的最大容量由二叉树中相邻两层的最大结点总数所决定，相邻两层的最大结点总数与 n 只是一个线性关系，所以层次遍历算法的空间复杂度也为 $O(n)$。

5.3.2 二叉树遍历算法的应用举例

二叉树的遍历操作是实现对二叉树其他操作的一个重要基础，利用二叉树的遍历方法可以解决二叉树的许多应用问题。二叉树遍历操作中的“访问根结点”是某种抽象的操作，即不确定的操作，在解决实际问题时，可以采取不同的解决策略。下面给出几个典型的二叉树遍历操作应用问题及其实现方法。下面的不同应用举例，可以更好地说明二叉树遍历操作中的访问根结点是可以根据实际情况，对根结点进行各种不同的操作的。

【例 5-1】 二叉树上的查找：编写算法完成在二叉树中查找值为 x 的结点的操作。

【问题分析】 在二叉树中查找结点的操作的要求是：在以 T 为根结点的二叉树中查找值为 x 的结点。若找到，则返回该结点；否则，返回空值。

要实现该查找操作，可在二叉树的先根遍历过程中进行，并且在遍历时将访问根结点的操作视为是将根结点的值与 x 进行比较的操作。其主要操作步骤描述如下。

（1）若二叉树为空，则不存在这个结点，返回空值；否则，将根结点的值与 x 进行比较，若相等，则返回该结点。

(2) 若根结点的值与 x 不相等,则在左子树中进行查找,若找到,则返回找到的结点。

(3) 若在左子树中没找到值为 x 的结点,则继续在右子树中进行查找,并返回查找结果。

说明:此操作的实现过程在描述时要注意其程序结构与先根遍历递归算法的不同之处。因为在二叉树上按先根遍历搜索时,只要找到了值为 x 的结点就不必继续进行搜索。也就是说,只有当根结点不是值为 x 的结点时,才需进入左子树进行查找,而且也只有当在左子树中仍未查找到值为 x 的结点时,才需要进入右子树继续查找。

【算法 5-8】 二叉树上的查找算法。

```
Status SearchNode(BiTree &T,char x)
//在以 T 为根结点的二叉树中查找值为 x 的结点。若找到,则返回该结点;否则,返回空值
  {   if(T!= NULL)
      {   if(T->data == x)                          //对根结点进行判断
             return OK;
          else
            return (SearchNode(T->lchild,x)!= ERROR?SearchNode(T->lchild,x):SearchNode(T
->rchild,x));
                  //若在左子树中查找到值为 x 的结点,则返回该结点;否则,在右子树中查找该结点
      }
    return ERROR;
}//算法 5-8 结束
```

【例 5-2】 计算二叉树中结点的个数:编写算法实现统计二叉树中结点的个数的操作。

【问题分析】 由于二叉树中结点的个数等于一个根结点再分别加上它的左、右子树中结点的个数,所以可以运用不同的遍历递归算法的思想来统计出二叉树中结点的个数。这里以先根遍历为例。在二叉树的先根遍历递归算法中,引入一个计数变量 num,将访问根结点的操作视为对结点计数变量加 1 的操作,将遍历左、右子树的操作视为统计左、右子树的结点个数并将其值分别加到结点的计数变量中的操作。其主要操作步骤描述如下。

(1) 若二叉树非空,则:

① 计数变量 num 值加 1;

② 统计根结点的左子树的结点个数,并加到计数变量 num 中;

③ 统计根结点的右子树的结点个数,并加到计数变量 num 中。

(2) 返回 count 值。

【算法 5-9】 统计二叉树中结点个数的算法。

```
void CountNode(BiTree T,int &num)
  //采用先根遍历操作对二叉树遍历,在遍历的过程中统计二叉树中的结点个数
    { if(T!= NULL)
      {      num++;                          //计数器加 1
             CountNode(T->lchild,num);       //对其左孩子域自递归,加上左子树中的结点个数
             CountNode(T->rchild,num);       //对其右孩子域自递归,加上右子树中的结点个数
      }
    } //算法 5-9 结束
```

说明:针对这个问题,最容易想到的解决办法是引入一个计数变量,其初值为 0。在二叉树的遍历过程中,访问一个结点就对该结点进行计数,即计数变量加 1;当整个二叉树都

遍历完毕后，计数变量的值就是二叉树的结点的个数值。按照这种思想，统计二叉树的结点个数也可以使用不同遍历的非递归算法来实现。下面给出在层次遍历过程中对二叉树的结点进行计数的程序代码，在其他遍历过程中对二叉树的结点进行计数的算法可依照同样的方法设计。

【算法 5-10】 统计二叉树中结点个数的算法。

```
void CountNode1(BiTree T,int &num)
//采用层次遍历操作对二叉树遍历,在遍历的过程中统计二叉树中的结点个数
  {    SqQueue Q;
       InitQueue(Q);
       EnQueue(Q,T);                          //根结点入队
       while(!QueueEmpty(Q))                  //当队列非空时
        {     DeQueue(Q,T);                   //队首元素出队
              num++;                          //计数器加 1
              if(T->lchild!= NULL)            //有左孩子时将其入队
                  EnQueue(Q,T->lchild);
              if(T->rchild!= NULL)            //有右孩子时将其入队
                  EnQueue(Q,T->rchild);
        }
    } //算法 5-10 结束
```

思考：如果需要统计二叉树中叶结点的个数，应如何设计算法？

【例 5-3】 求二叉树的深度：编写算法完成求二叉树的深度的操作。

【问题分析】 要求二叉树的深度，一种方法是先求出左子树的深度，再求出右子树的深度，二叉树的深度就是左子树的深度和右子树的深度中的最大值加 1。按照这种思路，自然就会想到使用后根遍历的递归算法来解决求二叉树的深度问题。其主要操作步骤描述如下。

若二叉树为空，则返回 0 值，否则：

(1) 求左子树的深度。

(2) 求右子树的深度。

(3) 将左、右子树深度的最大值加 1 并返回其值。

【算法 5-11】 求二叉树深度的算法。

```
int Depth(BiTree T)
//采用后根遍历操作对二叉树遍历,在遍历的过程中求二叉树的深度
  {   int depthLeft,depthRight,depthval;
      if(T!= NULL)
       {    depthLeft = Depth(T->lchild);              //对左孩子域自递归,求左子树的深度
            depthRight = Depth(T->rchild);             //对右孩子域自递归,求右子树的深度
            depthval = 1 + (depthLeft > depthRight?depthLeft:depthRight);
                                                       //返回左子树的深度和右子树的深度
                                                       //中的最大值加 1
       }
       else depthval = 0;
       return depthval;
} //算法 5-11 结束
```

【例5-4】 判断两棵树是否相等：编写算法判断两棵二叉树是否相等。若相等，则返回TRUE；否则，返回FALSE。

【问题分析】 由于一棵二叉树可以看成是由根结点、左子树和右子树3个基本单元所组成的树形结构，所以若两棵树相等，只有两种可能的情况：一种情况是这两棵二叉树都为空；另一种情况是当两棵二叉树都为非空时，两棵树的根结点、左子树和右子树都分别对应相等。所谓根结点相等就是指两棵树的根结点的数据值相等，而左、右子树的相等判断可以用对二叉树相等判断的同样方法来实现，即可用递归调用。下面用模拟先根遍历的思路来描述算法的操作步骤。

(1) 若两棵二叉树都为空，则两棵二叉树相等，返回TRUE。

(2) 若两棵二叉树都非空，则：

① 若根结点的值相等，则继续判断它们的左子树是否相等；

② 若左子树相等，则再继续判断它们的右子树是否相等；

③ 若右子树也相等，则两棵二叉树相等，返回TRUE。

(3) 其他任何情况都返回FALSE。

【算法5-12】 判断两棵二叉树是否相等的算法。

```
Status IsEqual(BiTree T1,BiTree T2)
//采用先根遍历思想,判断两棵二叉树是否相等
 {    if(T1 == NULL&&T2 == NULL)                              //同时为空
          return TRUE;
      if(T1!= NULL&&T2!= NULL)                                //同时非空,进行比较
          if(T1 -> data == T2 -> data)                        //根结点的值是否相等
              if(IsEqual(T1 -> lchild,T2 -> lchild))          //左子树是否相等
                  if(IsEqual(T1 -> rchild,T2 -> rchild))      //右子树是否相等
                      return TRUE;
      return FALSE;
      } //算法5-12结束
```

说明：此问题可以运用先根、中根和后根中的任何一种遍历思想来实现，描述过程中不相同之处仅在于对根结点、左子树和右子树判断的次序不同而已。读者可以自行完成用中根遍历和后根遍历的算法设计。

注意：递归是算法设计中一种有效的解决策略，它能够将问题转化为规模缩小了的同类问题的子问题，因而采用递归编写的算法具有描述清晰、易于理解的优点。一般地，构造一个递归模型来反映一个递归问题的递归结构，然后根据递归模型写出相应的递归算法。递归模型包括两个要素：一是递归终止条件(又称递归出口)，是所描述问题的最简单情况，本身不再使用递归的定义；二是递归体，是问题向终止条件转化的规则，能够使“大问题”分解成“小问题”，直到找到递归终止条件。例如：阶乘问题的递归模型构造如下：

$$f(n)=\begin{cases}0 & (n=0)\\ 1 & (n=1)\\ f(n-1)*n & (n>1)\end{cases}$$

如前所述，二叉树的许多应用问题都是递归问题，故可以利用二叉树遍历算法的特点进一步拓展得到递归模型。下面分别构造例5-2～例5-4的递归模型，并给出相应的算法。

例5-2的递归模型如下：

$$f(T)=\begin{cases}0 & \text{当 } T \text{ 为空时}\\ f(T.\text{lchild})+f(T.\text{rchild})+1 & \text{其他情况}\end{cases}$$

【算法 5-13】 统计二叉树中结点个数的算法(采用递归模型)。

```
int CountNode2(BiTree T)
//采用递归模型统计二叉树中的结点个数
{   if(T == NULL)
        return ERROR;
    else
        return CountNode2(T -> lchild) + CountNode2(T -> rchild) + 1;
} //算法 5 - 13 结束
```

例 5-3 的递归模型如下:

$$f(T)=\begin{cases}0 & \text{当 } T \text{ 为空时}\\ 1 & \text{当 } T \text{ 为叶结点时}\\ \max(f(T.\text{lchild}),f(T.\text{rchild}))+1 & \text{其他情况}\end{cases}$$

【算法 5-14】 求二叉树深度的算法(采用递归模型)。

```
int Depth1(BiTree T)
//采用递归模型求二叉树的深度
{if(T == NULL)
     return 0;
else if(T -> lchild == NULL&&T -> rchild == NULL)
     return 1;
else
     return 1 + (Depth1(T -> lchild)> Depth1(T -> rchild)?Depth1(T -> lchild):Depth1(T -> rchild));
} //算法 5 - 14 结束
```

例 5-4 的递归模型如下:

$$f(T1,T2)=\begin{cases}\text{TRUE} & \text{当 } T1 \text{ 和 } T2 \text{ 都为空时}\\ T1.\text{data}==T2.\text{data}\&\&f(T1.\text{lchild},T2.\text{lchild})\ \&\&f(T1.\text{rchild},T2.\text{rchild}) & \text{当 } T1 \text{ 和 } T2 \text{ 都非空时}\\ \text{FALSE} & \text{其他情况}\end{cases}$$

【算法 5-15】 判断两棵二叉树是否相等的算法(采用递归模型)。

```
Status IsEqual1(BiTree T1,BiTree T2)
//采用递归模型判断两棵二叉树是否相等
  {if(T1 == NULL&&T2 == NULL)
       return TRUE;
   else if(T1!= NULL&&T2!= NULL)
       return (T1 -> data == T2 -> data)&&IsEqual1(T1 -> lchild,T2 -> lchild)&&IsEqual1(T1 -> rchild,T2 -> rchild);
   else
       return FALSE;
}//算法 5 - 15 结束
```

5.3.3 建立二叉树

前面已经提到,二叉树的遍历操作的结果可将已知的一棵二叉树的非线性结构转换成由二叉树所有结点所组成的一个线性遍历序列。反过来,是否可以根据一棵二叉树结点的线性遍历序列来建立一棵二叉树呢?

先从先根遍历序列、中根遍历序列和后根遍历序列的结点排列规律进行分析。它们的排列规律可通过图5-16来描述。

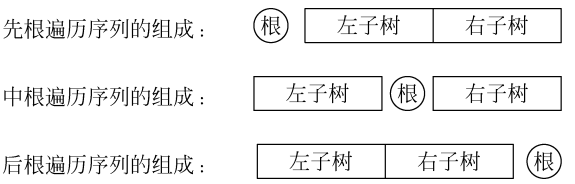


图5-16 二叉树遍历序列的结点排列规律图

从图5-16可知,先根遍历序列或后根遍历序列能反映双亲与孩子结点之间的层次关系,而中根遍历序列能反映兄弟结点之间的左右次序关系。所以,已知一种二叉树的遍历序列是不能唯一确定一棵二叉树的。只有已知先根遍历序列和中根遍历序列,或已知后根遍历序列和中根遍历序列,才能唯一确定一棵二叉树。而已知先根遍历序列和后根遍历序列也无法唯一确定一棵二叉树(读者可以通过反例说明此结论的正确性)。

1. 由先根遍历序列和中根遍历序列建立一棵二叉树

由于二叉树是由具有层次关系的结点所构成的非线性结构,而且二叉树中的每个结点的两棵子树具有左右次序之分,所以要建立一棵二叉树就必须明确:双亲结点和孩子结点之间的层次关系;兄弟结点之间的左右次序关系。由前面分析可知,根据先根遍历序列和中根遍历序列就能确定结点之间的这两种关系。下面给出根据已知二叉树的先根遍历序列和中根遍历序列唯一确定一棵二叉树的主要步骤。

(1) 取先根遍历序列中的第一个结点作为根结点。

(2) 在中根遍历序列中寻找根结点,确定根结点在中根遍历序列中的位置,假设为$i(0\leqslant i\leqslant \text{count}-1)$,其中count为二叉树遍历序列中结点的个数。

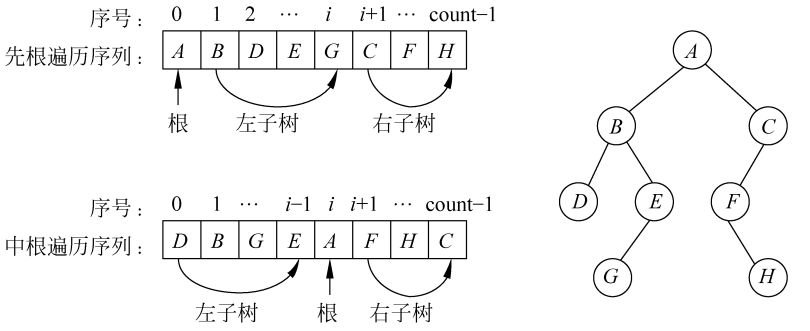
(3) 在中根遍历序列中确定:根结点之前的i个结点序列构成左子树的中根遍历序列,根结点之后的$\text{count}-i-1$个结点序列构成右子树的中根遍历序列。

(4) 在先根遍历序列中确定:根结点之后的i个结点序列构成左子树的先根遍历序列,剩下的$\text{count}-i-1$个结点序列构成右子树的先根遍历序列。

(5) 由步骤(3)和(4)又确定了左、右子树的先根遍历序列和中根遍历序列,接下来可用上述同样的方法来确定左、右子树的根结点,以此递归就可建立唯一的一棵二叉树。

例如,已知先根遍历序列为*ABDEGCFH*,中根遍历序列为*DBGEAFHC*,则按照上述步骤建立一棵二叉树的过程示意图如图5-17所示,具体执行过程如图5-18所示。

要实现上述建立二叉树的算法,需要引入5个参数:PreOrder、InOrder、PreIndex、InIndex和count。其中,参数PreOrder是整棵树的先根遍历序列;InOrder是整棵树的中根遍历序列;PreIndex是先根遍历序列在PreOrder中的开始位置;InIndex是中根遍历序列在InOrder中的开始位置;count表示树中结点的个数。



(a) 先根和中根遍历序列　　(b) 建立的二叉树

图 5-17　由已知先根遍历序列和中根遍历序列建立一棵二叉树的过程示意图

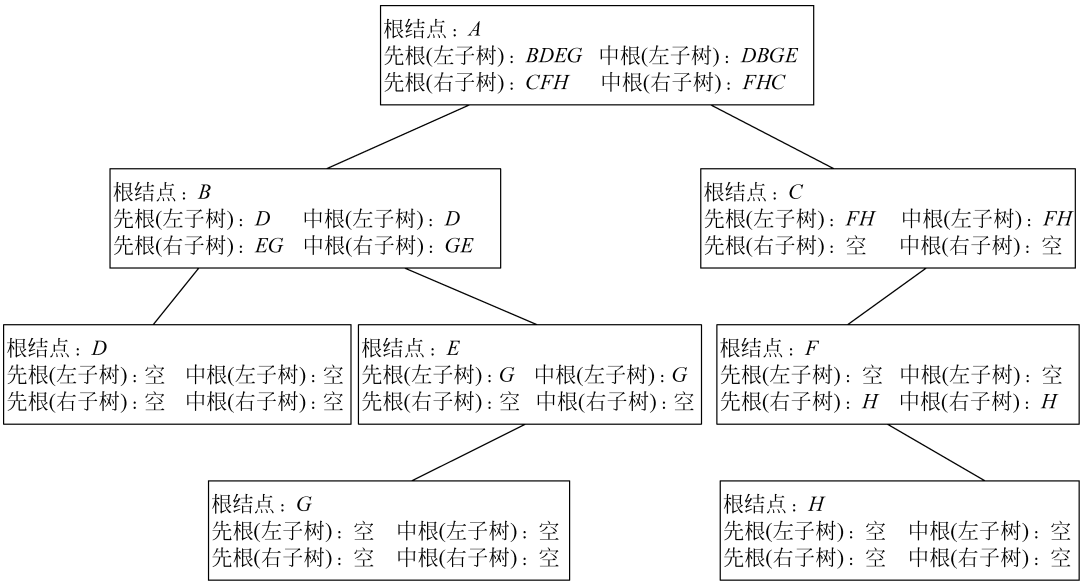


图 5-18　图 5-17 示例的具体执行过程

【算法 5-16】 由先根遍历序列和中根遍历序列建立一棵二叉树的算法。

```
BiTree PICreateBiTree(char PreOrder[],char InOrder[],int PreIndex,int InIndex,int count)
//由先根遍历序列和中根遍历序列建立一棵二叉树 ,并返回其根结点

{   BiTree T;
    if(count>0)                                  //先根和中根非空,count 表示二叉树中结点数
       {   char r=PreOrder[PreIndex];            //取先根遍历序列中的第一个元素
           int i=0;
           for(;i<count;i++)                     //寻找根结点在中根遍历序列中的位置
           {   if(r==InOrder[i+InIndex])
                   break;
           }
           T=(BiTree)malloc(sizeof(BiTNode));    //建立根结点
```

```
            T->data = r;
            T->lchild = PICreateBiTree(PreOrder,InOrder,PreIndex + 1,InIndex,i);
                                                    //建立左子树
            T->rchild = PICreateBiTree(PreOrder,InOrder,PreIndex + i + 1,InIndex + i + 1,count
- i - 1);                                           //建立右子树
        }
    else
        T = NULL;
    return T;
    } //算法 5-16 结束
```

2. 由后根遍历序列和中根遍历序列建立一棵二叉树

由前面分析可知,根据后根遍历序列和中根遍历序列同样能够确定结点之间的关系,即双亲结点和孩子结点之间的层次关系;兄弟结点之间的左右次序关系。下面给出根据已知二叉树的后根遍历序列和中根遍历序列唯一确定一棵二叉树的主要步骤。

(1) 取后根遍历序列中的最后一个结点作为根结点。

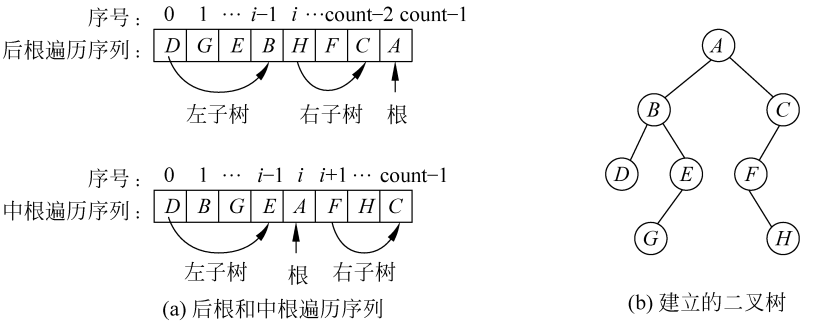
(2) 在中根遍历序列中寻找根结点,确定根结点在中根遍历序列中的位置,假设为 $i(0\leqslant i\leqslant \text{count}-1)$,其中 count 为二叉树遍历序列中结点的个数。

(3) 在中根遍历序列中确定:根结点之前的 i 个结点序列构成左子树的中根遍历序列,根结点之后的 count$-i-1$ 个结点序列构成右子树的中根遍历序列。

(4) 在后根遍历序列中确定:前面的 i 个结点序列构成左子树的后根遍历序列,剩下的 count-i-1 个结点序列构成右子树的后根遍历序列。

(5) 由步骤(3)和(4)又确定了左、右子树的后根遍历序列和中根遍历序列,接下来可用上述同样的方法来确定左、右子树的根结点,以此递归就可建立唯一的一棵二叉树。

例如,已知后根遍历序列为 *DGEBHFCA*,中根遍历序列为 *DBGEAFHC*,则按照上述步骤建立一棵二叉树的过程示意图如图 5-19 所示,具体执行过程如图 5-20 所示。



(a) 后根和中根遍历序列　　(b) 建立的二叉树

图 5-19　由已知后根遍历序列和中根遍历序列建立一棵二叉树的过程

3. 由层次遍历序列和中根遍历序列建立一棵二叉树

由于层次遍历序列反映的就是二叉树中双亲与孩子结点之间的层次关系,因而由已知层次遍历序列和中根遍历序列可以唯一确定一棵二叉树。下面给出根据已知一棵二叉树的层次遍历序列和中根遍历序列唯一确定一棵二叉树的主要步骤。

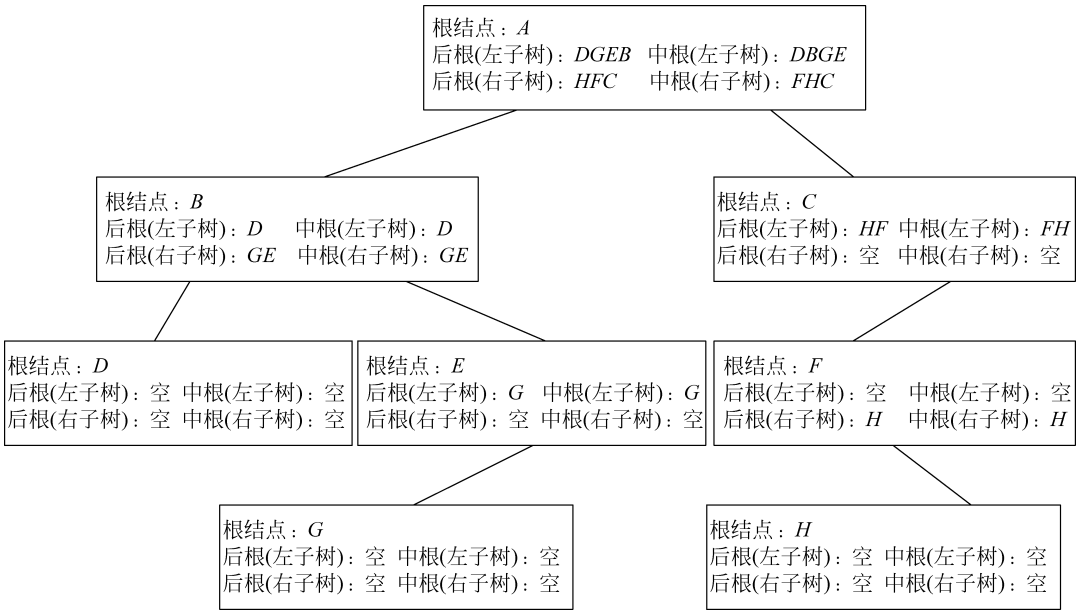


图 5-20　图 5-19 示例的具体执行过程

(1) 取层次遍历序列中的第一个结点作为根结点。

(2) 在中根遍历序列中寻找根结点，确定根结点在中根遍历序列中的位置，假设为 $i(0 \leqslant i \leqslant \text{count}-1)$，其中 count 为二叉树遍历序列中结点的个数。

(3) 在中根遍历序列中确定：根结点之前的 i 个结点序列构成左子树的中根遍历序列，根结点之后的 count$-i-1$ 个结点序列构成右子树的中根遍历序列。

(4) 在层次遍历序列中确定：若根结点之后的第 1 个结点出现在中根遍历序列的前 i 个结点之中，则它是根结点的左子树的根；若根结点之后的第 2 个结点出现在中根遍历序列的剩下的 count$-i-1$ 个结点中，则它是根结点的右子树的根。

(5) 由步骤(3)和(4)又确定了左、右子树的层次遍历序列和中根遍历序列，接下来可用上述同样的方法来确定左、右子树的根结点，以此递归就可建立唯一的一棵二叉树。

例如，已知层次遍历序列为 $ABCDEFGH$，中根遍历序列为 $DBGEAFHC$，则按照上述步骤建立一棵二叉树的过程示意图如图 5-21 所示，具体执行过程如图 5-22 所示。

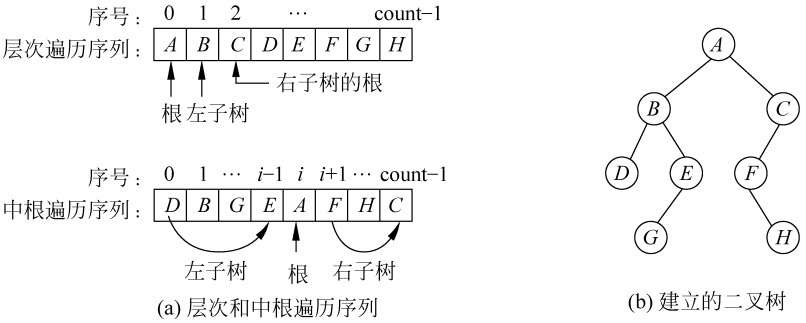


图 5-21　由已知层次遍历序列和中根遍历序列建立一棵二叉树的过程

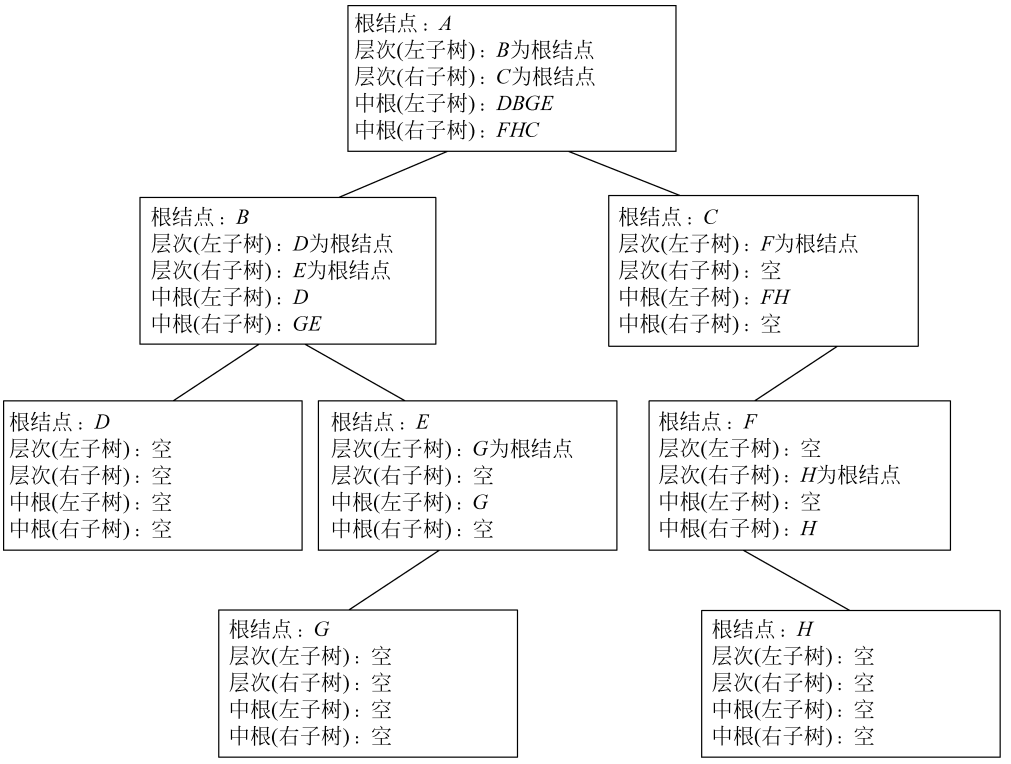


图 5-22　图 5-21 示例的具体执行过程

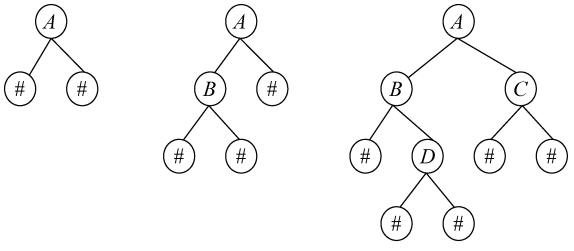
4. 由标明空子树的先根遍历序列建立一棵二叉树

由前面的分析可知，已知二叉树的先根遍历序列是不能唯一确定一棵二叉树的。例如，先根遍历序列为“AB”所对应的就有两棵不同的二叉树，如图 5-23 所示。

如果我们能够在先根遍历序列中加入每个结点的空子树信息，则可明确二叉树中结点与双亲、孩子与兄弟结点之间的关系，因此就可以唯一确定一棵二叉树。例如，图 5-24 所示的是 3 棵标明空子树“#”的二叉树及其对应的先根遍历序列。

(a)　(b)

图 5-23　先根序列为 AB 的两棵不同的二叉树



(a) A##　(b) AB###　(c) AB#D##C##

图 5-24　标明空子树的二叉树及其先根遍历序列

按照标明空子树“#”的先根遍历序列建立一棵二叉树的主要步骤描述如下。

从标明空子树信息的先根遍历序列中依次读入字符，若读入的字符是#，则建立空树，否则：

(1) 建立根结点。

(2) 继续建立树的左子树。

(3) 继续建立树的右子树。

【算法 5-17】 由标明空子树的先根遍历序列建立一棵二叉树的算法。

```
Status CreateBiTree(BiTree &T)
 // 由标明空子树的先根遍历序列建立一棵二叉树,并返回其根结点
{char ch;
scanf("%c",&ch);
if(ch==  '#') T=NULL;                          //#字符表示空二叉树
else{
    T=(BiTree)malloc(sizeof(BiTNode));         //生成根结点
    T->data=ch;
     CreateBiTree(T->lchild);                  //构造左子树
    CreateBiTree(T->rchild);                   //构造右子树
    }
return OK;
} //算法 5-17 结束
```

【例 5-5】 编写一个程序：首先由标明空子树的先根遍历序列建立一棵二叉树，然后输出该二叉树的先根、中根和后根遍历序列。

【问题分析】 这个问题的要求是先输入一个标明空子树信息的二叉树的先根遍历序列，如 $AB\#\#CD\#\#\#$，其代表的二叉树如图 5-25 所示。然后根据标明空子树的先根遍历序列建立二叉树的算法创建一棵二叉树，并输出这棵二叉树的先根遍历序列、中根遍历序列和后根遍历序列。

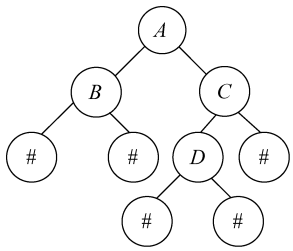


图 5-25 标明空子树的先根遍历序列为 $AB\#\#CD\#\#\#$ 的二叉树

【程序代码】

```
int main()
{   BiTree T;
    printf("输入一棵二叉树 T,空结点用'#'表示:\n");
    CreateBiTree(T);                       //由标明空子树的先根遍历序列建立一棵二叉树
    printf("先根遍历序列为:\n");
    PreRootTraverse(T);                    //调用二叉树的先根遍历函数
    printf("\n");
    printf("中根遍历序列为:\n");
    InRootTraverse(T);                     //调用二叉树的中根遍历函数
    printf("\n");
    printf("后根遍历序列为:\n");
    PostRootTraverse(T);                   //调用二叉树的后根遍历函数
    printf("\n");
    return 0;
}
```

【运行结果】 运行结果如图5-26所示。

图5-26 例5-5程序的运行结果

5. 由完全二叉树的顺序存储结构建立其二叉链式存储结构

对于一棵顺序存储结构的完全二叉树(如图5-9(a)所示),由二叉树的性质5可知,编号为1的结点是根结点;对于编号为i的结点($i>1$),它的双亲的编号是$\lfloor i/2 \rfloor$,若它有左孩子,则左孩子的编号是$2i$,若它有右孩子,则右孩子的编号是$2i+1$。所以利用性质5,根据完全二叉树的顺序存储结构,可以建立完全二叉树的二叉链式存储结构。为了简化算法,这里用串表示顺序存储结构的完全二叉树。下面通过一个例子说明此操作的实现方法。

【例5-6】 编写一个程序:首先根据完全二叉树的顺序存储结构建立一棵二叉树的链式存储结构,然后输出该二叉树的中根遍历序列和该二叉树的深度。

【问题分析】 此问题中所涉及的操作主要有3种,包括根据完全二叉树的顺序存储结构建立一棵二叉树的链式存储结构,对二叉树进行遍历和计算二叉树的深度。这3种操作中的后两种操作的实现方法前面都已经给出,所有只要引用前面所定义的相应方法即可,第1种操作在下面的程序代码中给出了具体的实现方法。

注意:完全二叉树的顺序存储空间中第i个位置存放的是编号为$i+1$的结点(编号为1的根结点存放在第0个位置),故其左、右孩子分别存放在第$2i+1$个位置和第$2i+2$个位置。

【程序代码】

```
BiTree ComCreateBiTree(char ch[],int index)
//由顺序存储的完全二叉树建立一棵二叉树,其中 index 标识根结点在顺序存储结构中的位置
 {
    BiTree T = NULL;
    if(ch[index]!= '\0')
     {      T = (BiTree)malloc(sizeof(BiTNode));
            T -> data = ch[index];                              //生成根结点
            T -> lchild = ComCreateBiTree(ch,2 * index + 1);    //构造左子树
            T -> rchild = ComCreateBiTree(ch,2 * index + 2);    //构造右子树
     }
     return T;
}//ComCreateBiTree
int main()
{     char ch[20] = "ABCDEFG";          //如图5-9(a)所示的顺序存储的完全二叉树
      BiTree T;
      int d;
      T = ComCreateBiTree(ch,0);        //由完全二叉树的顺序存储结构建立一棵二叉树
      printf("先根遍历序列为:\n");
      PreRootTraverse(T);               //调用二叉树的先根遍历函数
      printf("\n");
```

```
    printf("中根遍历序列为:\n");
    InRootTraverse(T);              //调用二叉树的中根遍历函数
    printf("\n");
    printf("后根遍历序列为:\n");
    PostRootTraverse(T);            //调用二叉树的后根遍历函数
    printf("\n");
    printf("此二叉树深度为:");
    d = Depth1(T);                  //调用采用递归模型求二叉树的深度函数,输出二叉树的
深度
    printf(" %d",d);
    printf("\n");
        return 0;
}
```

【运行结果】 运行结果如图 5-27 所示。

图 5-27　例 5-6 程序的运行结果

5.4　哈夫曼树及哈夫曼编码

树形结构除了应用于查找和排序操作时能提高效率外,它在信息通信领域也有着广泛的应用。哈夫曼(Huffman)树就是一种在编码技术方面得到广泛应用的二叉树,也是一种最优二叉树。

5.4.1　哈夫曼树的基本概念

为了给出哈夫曼树的定义,需要从以下几个基本概念出发进行描述。

1. 结点间的路径和结点的路径长度

所谓结点间的路径是指从一个结点到另一个结点所经历的结点和分支序列。结点的路径长度是指从根结点到该结点之间的路径上的分支数目。

2. 结点的权和结点的带权路径长度

在实际应用中,人们往往会给树中的每个结点赋予一个具有某种实际意义的数值,这个数值被称为该结点的权值。结点的带权路径长度就是该结点的路径长度与该结点的权值的乘积。

3. 树的带权路径长度

树的带权路径长度就是树中所有叶结点的带权路径长度之和,通常记为:

$$\mathrm{WPL}=\sum_{i=1}^{n}w_i l_i \tag{5-8}$$

其中，n 为叶结点的个数，w_i 为第 i 个叶结点的权值，l_i 为第 i 个叶结点的路径长度。

注意：在结点个数相同的多棵二叉树中，若不考虑每个结点的权值，则路径长度最短的应是一棵完全二叉树。

4. 哈夫曼树

给定 n 个权值并作为 n 个叶结点按一定规则构造一棵二叉树，使其带权路径长度达到最小值，则这棵二叉树被称为最优二叉树。由于哈夫曼给出了构造这种树的规则，因此，最优二叉树也称为哈夫曼树。要让一棵二叉树的带权路径长度最短，必须使权值越小的叶结点越远离根结点，而权值越大的叶结点越靠近根结点。

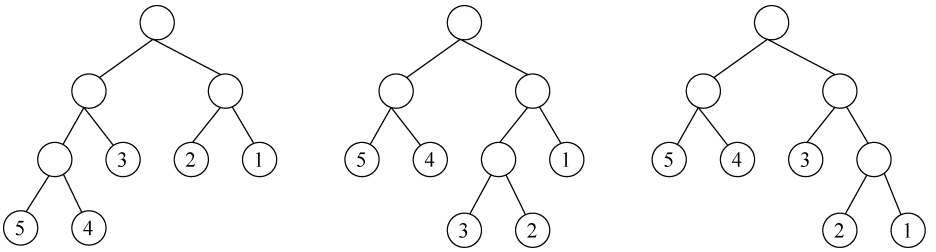
例如，图 5-28 所示的 3 棵二叉树，它们都具有 5 个叶结点，并带有相同权值 5、4、3、2 和 1，但它们的带权路径长度不同，分别为：

(a) WPL=5×3+4×3+3×2+2×2+1×2=39

(b) WPL=5×2+4×2+3×3+2×3+1×2=35

(c) WPL=5×2+4×2+3×2+2×3+1×3=33

其中，图 5-28(c)所示的二叉树带权路径长度值最小，它就是一棵最优二叉树或哈夫曼树。



(a) 带权路径长度为39　(b) 带权路径长度为35　(c) 带权路径长度为33

图 5-28　具有不同带权路径长度的二叉树

5.4.2　哈夫曼树和哈夫曼编码的构造方法

1. 哈夫曼树的构造

从图 5-28 可见，若叶结点个数相同且对应权值也相同，而对应权值的叶结点所处的层次不相同，则二叉树的带权路径长度可能不相同。但是在所有含 n 个叶结点，并且带有相同权值的二叉树中，必定存在一棵其带权路径长度值为最小的二叉树，也就是最优二叉树或哈夫曼树。如何才能构造出哈夫曼树呢？其算法的主要步骤描述如下。

假设 n 个叶结点的权值分别为$\{w_1,w_2,\cdots,w_n\}$，则：

(1) 初始化操作：由已知给定的 n 个权值$\{w_1,w_2,\cdots,w_n\}$，构造一个由 n 棵二叉树所构成的森林 $F=\{T_1,T_2,\cdots,T_n\}$，其中每棵二叉树只有一个根结点，并且根结点的权值分别为 $w_1,w_2,\cdots,w_n$。

(2) 选取与组合操作：在二叉树森林 F 中选取根结点的权值最小和次小的两棵二叉树，分别把它们作为左子树和右子树去构造一棵新二叉树，新二叉树的根结点权值为其左、

右子树根结点的权值之和。

(3) 删除与归并操作：将作为新二叉树的左、右子树的两棵二叉树从森林 F 中删除，并将新产生的二叉树加入森林 F 中。

(4) 重复步骤(2)和(3)，直到森林 F 中只剩下一棵二叉树为止，则这棵二叉树就是所构造成的哈夫曼树。

例如，对于一组给定的权值{5,4,3,2,1}，图 5-29 给出了图 5-28(c)哈夫曼树的构造过程。

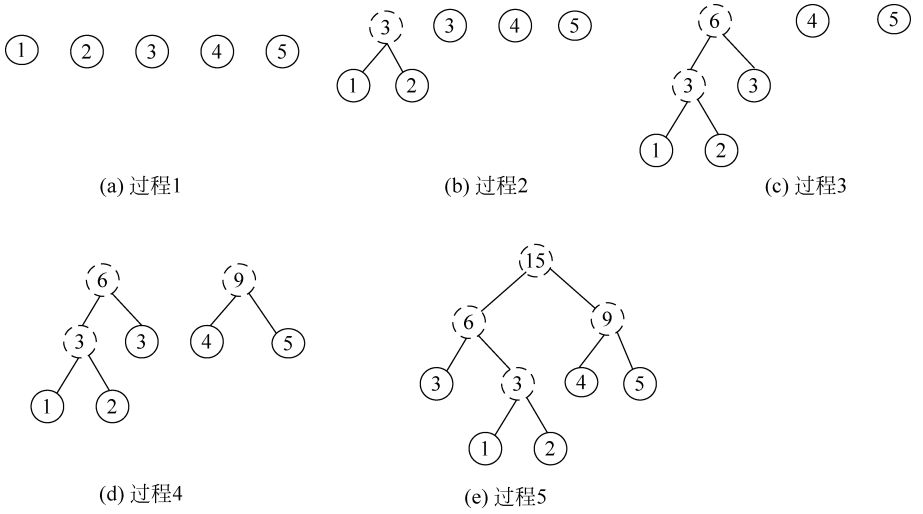


图 5-29　图 5-28(c)哈夫曼树的构造过程

由于在哈夫曼树的构造过程中，选取根结点的权值最小和次小的两棵二叉树来组合一棵新的二叉树时，没有规定哪棵二叉树是左子树，哪棵二叉树是右子树，故左子树和右子树的顺序是任意的。因此，利用哈夫曼算法构造的哈夫曼树并不是唯一的，但它们的带权路径长度是一样的。为了统一起见，可在哈夫曼构造的选取步骤中规定左子树的根结点权值小于等于右子树的根结点权值。

2. 用哈夫曼树进行编码

在信息通信领域里，信息传送的速度至关重要，而传送速度又与传送的信息量有关。在信息传送时需要将信息符号转换成二进制的符号串，这就需要进行二进制的编码。假设要传送的是由 A、B、C 和 D 这 4 个字符组成的电报文“ABCAABCAD”，在计算机中每个字符在没有压缩的文本文件中由一个字节(例如常见的 ASCII 码)或两个字节(例如 Unicode)表示。如果是用这种方案，每个字符都需要 8 或 16 位数。但是为了提高存储和传输效率，在信息存储和传送时总是希望传输信息的总长度尽可能短，这就需要设计一套对字符集进行二进制编码的方法，使得采用这种编码方式对信息进行编码时，信息的传输量最小。如果能对每个字符用不同长度的二进制编码，并且尽可能减少出现次数最多的字符的编码位数，则信息传送的总长度就可以达到最小。

假设将 A、B、C 和 D 分别编码为 0、1、01 和 10，则电报文“ABCAABCAD”的编码长度只有 12，达到最短。然而，在编码序列中，若用起始位组合(或前缀)相同的代码来表示不同的字符，则在不同字符的编码之间必须用分隔符隔开，否则就会产生二义性，电文也就无法

译码。为了在字符间省去不必要的分隔符号,就要求给出的每个字符的编码必须是前缀编码。所谓前缀编码,是指在所有字符的编码中,任何一个字符都不是另一个字符的前缀。

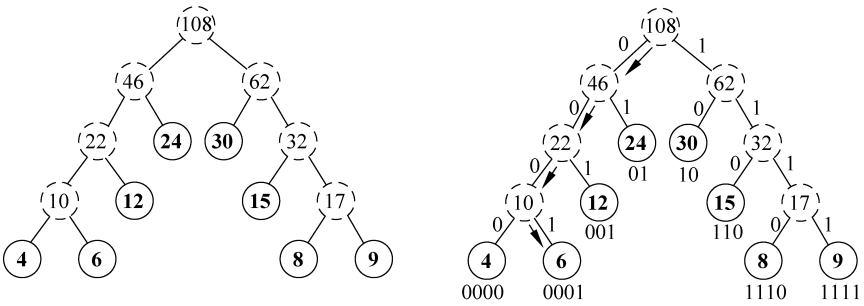
利用哈夫曼树可以构造一种不等长的二进制编码,并且构造所得的哈夫曼编码是一种最优前缀编码。

哈夫曼编码的构造过程是:用电文中各个字符使用的频度作为叶结点的权值,构造一棵具有最小带权路径长度的哈夫曼树,若对树中的每个左分支赋予标记0,右分支赋予标记1,则从根结点到每个叶结点的路径上的标记连接起来就构成一个二进制串,该二进制串被称为哈夫曼编码。

【例5-7】 已知在一个信息通信联络中使用了8个字符:a、b、c、d、e、f、g和h,每个字符的使用频度分别为:6、30、8、9、15、24、4和12。试设计各个字符的哈夫曼编码。

【问题分析】 首先以字符的频度作为叶结点的权值,依照哈夫曼树的构造规则,得到如图5-30(a)所示的一棵哈夫曼树,然后根据哈夫曼编码规则,将哈夫曼树中每个左分支标记为0,每个右分支标记为1,则可得到各个叶结点的哈夫曼编码,如图5-30(b)所示。最后,构造得到的各个字符的哈夫曼编码如下:

```
a:0001    b:10    c:1110    d:1111
e:110     f:01    g:0000    h:001
```



(a) 哈夫曼树　　(b) 哈夫曼树及其哈夫曼编码

图5-30　例5-7的哈夫曼树和哈夫曼编码

3. 用哈夫曼树进行译码

当在通信过程中接收到一条用二进制编码串(哈夫曼编码)表示的信息时,怎么才能把它转回字符呢?这就涉及了译码过程是如何实现的。所谓译码就是分解代表信息的二进制编码串,并从左到右逐位判别编码串,直至确定每个字符。译码与解码过程正好相反。

由于哈夫曼编码是一种由哈夫曼树求得的最优前缀编码,因此译码过程不会产生二义性。利用哈夫曼树进行译码,其过程可用哈夫曼编码过程的逆过程来实现:从哈夫曼树的根结点开始,从左到右对二进制编码的每位进行判别,若遇到0,则选择左分支走向下一个结点;若遇到1,则选择右分支走向下一个结点,直至到达一个叶结点。因为信息中出现的字符在哈夫曼树中是叶结点,所以确定了一条从根结点到叶结点的路径,就意味着译出了一个字符,然后继续用这棵哈夫曼树并用同样的方法去译出其他的二进制编码。如图5-30(b)所示,对于编码为0001的译码过程就是从根结点开始,先左,再左,再左,再右,最后到达使用频度为6的字符a。这个过程如图5-30(b)中的箭头所示。

5.4.3 构造哈夫曼树和哈夫曼编码的算法

在构造哈夫曼树之后，进行译码时要求能方便地实现从双亲结点到左、右孩子结点的操作，而在进行哈夫曼编码时又要求能方便实现从孩子结点到双亲结点的操作，因此，需要为哈夫曼树中的每个结点设置三个域：双亲结点域、左孩子域和右孩子域。此外，每个结点还要设置权值域和数据域。这样，哈夫曼树中的每个结点包含 5 个域，其存储结构如图 5-31 所示。

weight	data	parent	lchild	rchild

图 5-31　哈夫曼树中结点的存储结构

哈夫曼树中结点的存储结构描述如下：

```
typedef struct HTNode {
 int weight;                                    //权值域
 char data;                                     //数据域
 int parent, lchild, rchild;                    //双亲结点域、左孩子域和右孩子域
}HTNode;
```

构造哈夫曼树和哈夫曼编码的算法如下：

```
void CreateHuffmanTree(HuffmanTree HT)
//根据输入权值,构造具有 n 个权值结点的哈夫曼树
{    int i,p1,p2;
     InitHuffmanTree(HT);                       //初始化哈夫曼树
     InputWeight(HT);                           //输入权值
     for(i = n;i < 2 * n - 1;i++)               //构造哈夫曼树
     {       SelectMin(HT,i - 1,p1,p2);         //调用选择两个权值最小的结点的函数
            //在 HT[1...i - 1]中选择 parent 为 0,且权值最小的两个结点,其序号分别为 p1 和 p2
             HT[p1].parent = i;
             HT[p2].parent = i;
             HT[i].lchild = p1;
             HT[i].rchild = p2;
             HT[i].weight = HT[p1].weight + HT[p2].weight;
     }
}//CreateHuffmanTree
void SelectMin(HuffmanTree HT,int i,int &p1,int &p2)
//选择两个权值最小的结点的函数
{     long min1 = 999999;
      long min2 = 999999;
      int j;
      for(j = 0;j <= i;j++)                     //选择最小权值结点
            if(HT[j].parent == - 1)
                if(min1 > HT[j].weight)
                {    min1 = HT[j].weight;
                     p1 = j;
                }
     for(j = 0;j <= i;j++)                      //选择次小权值结点
            if(HT[j].parent == - 1)
                if(min2 > HT[j].weight&&j!= (p1))
                {    min2 = HT[j].weight;
                     p2 = j;
```

```
            }
  }//SelectMin
void HuffmanCoding(HuffmanTree HT)
//根据构造的哈夫曼树求哈夫曼编码
{     int i,j;
      for(i = 0;i < n;i++)
      {   j = 0;
          printf("\n");
          printf("\t\t%c\t",HT[i].data,HT[i].weight);
          HuffmanTreePath(HT,i,j);
      }
}//HuffmanCoding
void HuffmanTreePath(HuffmanTree HT,int i,int j)
//编码的哈夫曼树路径递归算法
{     int a,b;
      a = i;
      b = j = HT[i].parent;
      if(HT[j].parent!= -1)
      {   i = j;
          HuffmanTreePath(HT,i,j);
      }
      if(HT[b].lchild == a) printf("0");
      else printf("1");
} //HuffmanTreePath
```

当哈夫曼树和哈夫曼编码构造完毕后,就可以针对用户输入的电文(字符串)进行编码和译码操作。相应的算法如下:

```
void EnCoding(HuffmanTree HT)
//对用户输入的电文进行编码
{     char r[1000];                          //用来存储输入的字符串
      int i,j;
      gets(r);
      for(j = 0;r[j]!= '\0';j++)
           for(i = 0;i < n;i++)
                if(r[j] == HT[i].data)
                    HuffmanTreePath(HT,i,j);
}//EnCoding
void DeCoding(HuffmanTree HT)                //对用户输入的密文进行译码
{
      char r[100];
      int i,j,len;
      j = 2 * n - 2;
      gets(r);
      len = strlen(r);
      for(i = 0;i < len;i++)
      {    if(r[i] == '0')
           {       j = HT[j].lchild;
                if(HT[j].lchild == -1)
                {    printf("%c",HT[j].data);
                     j = 2 * n - 2;
                }
```

```
            }
            else if(r[i] == '1')
            {   j = HT[j].rchild;
                if(HT[j].rchild == -1)
                {   printf("%c",HT[j].data);
                    j = 2 * n - 2;
                }
            }
        }
    } //DeCoding
```

以例 5-7 为例，根据上述算法可以构造如图 5-30(a)所示的哈夫曼树，并可以得到 a、b、c、d、e、f、g 和 h 这 8 个字符的哈夫曼编码，分别为：a(0001)、b(10)、c(1110)、d(1111)、e(110)、f(01)、g(0000)和 h(001)。当用户输入电文“abach”时，编码为“00011000011110001”；相反，当用户输入密文“1011100111111110”时，译码为“bcfdc”。

5.5 树与森林

由于二叉树中的每个结点至多有左、右两个子结点，使得二叉树的存储和操作的实现方法较简单，而树与森林因为每个结点的子树不受限制，导致它们在存储结构上需要考虑较多的因素。因此，如果树、森林和二叉树之间能够进行相互转换，就可以将问题简化。本节将讨论树、森林与二叉树之间的对应关系，树的存储结构及其遍历操作。

5.5.1 树、森林与二叉树之间的转换

树与二叉树之间、森林与二叉树之间可以相互转换，而且这种转换是一一对应的。树与森林转化成二叉树后，森林或树的相关操作都可以转换成二叉树的操作。

1. 树转换成二叉树

二叉树中的结点有左孩子和右孩子之分，而在无序树中结点的各个孩子结点之间是无次序之分的。为了操作方便，假设树是一棵有序树，树中每个结点的孩子按从左到右的顺序进行编号，依次定义为第 1 个孩子，第 2 个孩子，……，第 i 个孩子。在如图 5-32 所示的一棵树中，A 结点有 3 个孩子，其中 B 是它的第 1 个孩子，C 是它的第 2 个孩子，D 是它的第 3 个孩子。

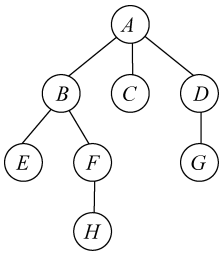


图 5-32 一棵树

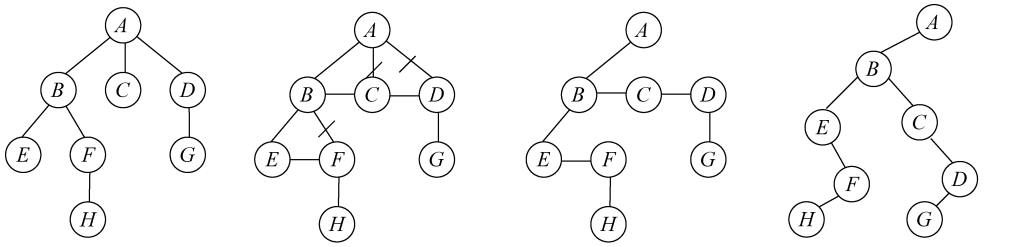
将树转换二叉树的方法可归纳为“加线”“删线”和“旋转”三个步骤，具体描述如下：

(1) 加线：在树中所有相邻的兄弟结点之间加一条连线。

(2) 删线：对树中的每个结点，只保留它与第1个孩子(长子)结点之间的连线，删去它与其他孩子结点之间的连线。

(3) 旋转：以树的根结点为轴心，将树平面顺时针旋转一定的角度并做适当的调整，使得转化后所得的二叉树看起来比较规整。

图5-33给出了将图5-32中树转换成二叉树的过程。



(a) 一棵树　(b) 加线后　(c) 删线后　(d) 旋转并调整后得到的二叉树

图5-33　树转换成二叉树的过程

由转换过程可知，树与由它转换成的二叉树是一一对应的，树中的任意一个结点都对应着二叉树中的一个结点，树中每个结点的第1个孩子结点在二叉树中对应结点的左孩子，而树中每个结点的右邻兄弟在二叉树中对应结点的右孩子(简言之，左孩子右兄弟)。也就是说，在二叉树中，左分支上的各个结点在原来的树中是父子关系，而右分支上的各个结点在原来的树中是兄弟关系。由于树中的根结点没有兄弟，所以由树转换成的二叉树永远都是一棵根结点的右子树为空的二叉树。

2. 二叉树转换成树

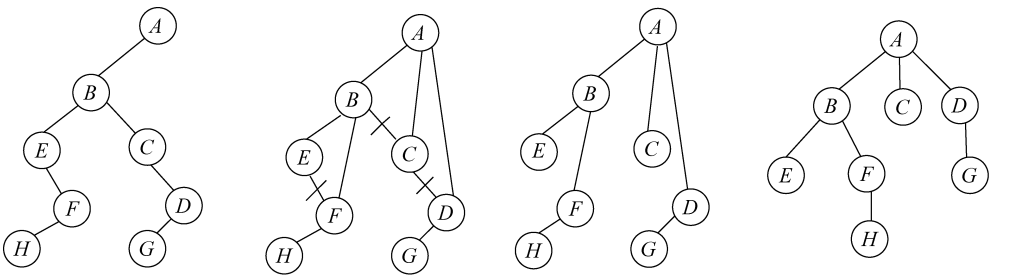
二叉树转换成树是由树转换成二叉树的一个逆过程，也就是一个由二叉树还原成它原来所对应的树的过程。具体操作步骤如下。

(1) 加线：若某结点是其双亲结点的左孩子，则将该结点沿着右分支向下的所有结点与该结点的双亲结点用线连接。

(2) 删线：将树中所有双亲结点与右孩子结点的连线删除。

(3) 旋转：经过(1)、(2)两步操作得到的树以根结点为轴心，沿逆时针方向旋转一定的角度并做适当调整，使得转化后所得的树看起来比较规整。

图5-34给出了将图5-33(d)中的二叉树还原成图5-33(a)中的树的过程。



(a) 二叉树　(b) 加线后　(c) 删线后　(d) 旋转并调整后所得到的树

图5-34　二叉树转换成树的过程

3. 森林与二叉树的转换

森林是若干棵树的集合，而任何一棵和树对应的二叉树其右子树一定为空，因而可得到将森林转换成二叉树的方法如下。

（1）将森林中的每棵树转换成相应的二叉树。

（2）按照森林中树的先后顺序，将后一棵二叉树视为前一棵二叉树的右子树，依次连接起来，从而构成一棵二叉树。

图 5-35 给出了一个森林转换成二叉树的过程。从这个转换过程中可以得出森林与二叉树之间的一一对应关系：森林中第一棵树的根结点对应着二叉树中的根结点；森林中第一棵树的根结点的子树所构成的森林对应着二叉树的左子树；森林中除第一棵树之外的其他树所构成的森林对应着二叉树的右子树。根据这种对应关系及森林转换成二叉树的逆操作，就可将一棵二叉树转换成其对应的森林。图 5-36 给出了一棵二叉树转换成森林的过程。

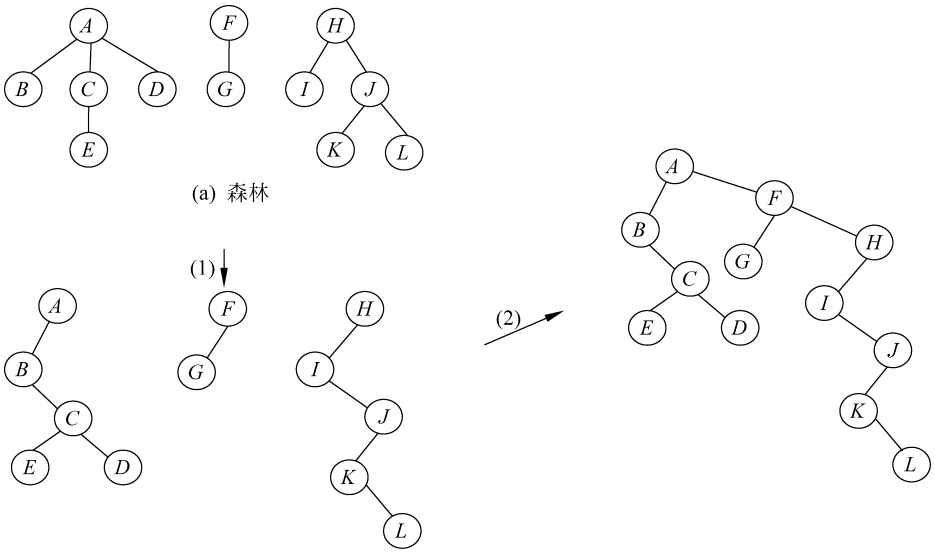


图 5-35　森林转换成二叉树的过程

森林与二叉树之间的转换规则也可以用递归方法加以描述，下面分别给出森林与二叉树之间相互转换的递归形式定义。

1）森林转换成二叉树

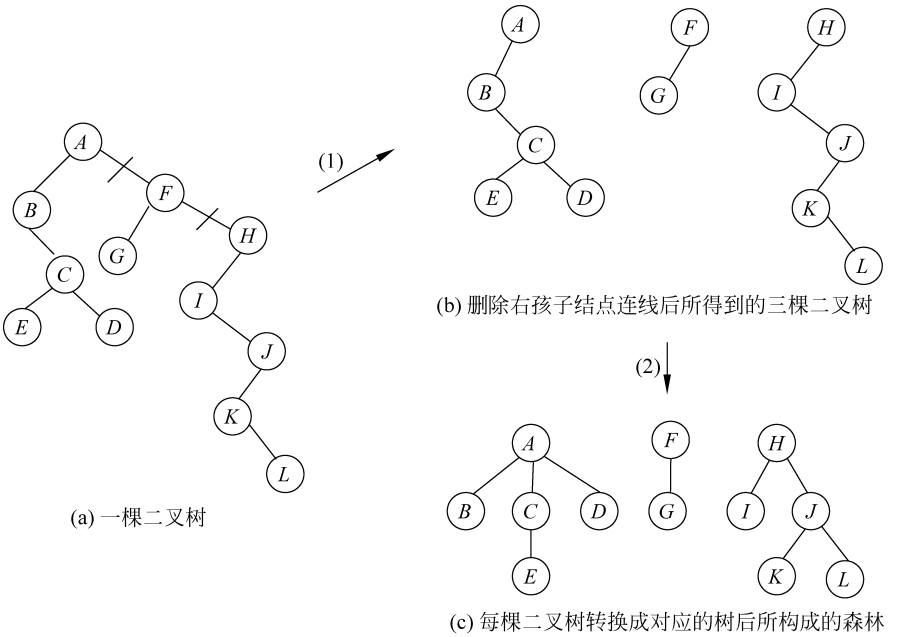
假设有序集合 $F=\{T_1,T_2,\cdots,T_n\}$ 表示由 n 棵树 $T_1,T_2,\cdots,T_n$ 所组成的森林，则森林 F 可按如下规则转换成二叉树 $B(F)$。

（1）若 F 为空，即 $n=0$ 时，则 $B(F)$ 为空。

（2）若 F 为非空，即 $n>0$ 时，则 $B(F)$ 的根是森林中第一棵树 T_1 的根，$B(F)$ 中的左子树是森林中树 T_1 的根结点的子树所构成的森林 $F_1=\{T_{11},T_{12},\cdots,T_{1m}\}$ 转换而成的二叉树 $B(T_{11},T_{12},\cdots,T_{1m})$，而右子树是由森林 $F'=\{T_2,\cdots,T_n\}$ 转换而成的二叉树 $B(T_2,\cdots,T_n)$。

2）二叉树转换成森林

设 B 是一棵二叉树，Root 是 B 的根结点，L 是 B 的左子树，R 是 B 的右子树，并且 B 对



(a) 一棵二叉树

(b) 删除右孩子结点连线后所得到的三棵二叉树

(c) 每棵二叉树转换成对应的树后所构成的森林

图 5-36　二叉树转换成森林的过程

应的森林 $F(B)$中含有 n 棵树：$T_1,T_2,\cdots,T_n$，则二叉树 B 可按如下规则转换成森林 $B(F)$。

(1) 若 B 为空，则 $F(B)$为空森林。

(2) 若 B 为非空，则 $F(B)$中第一棵树 T_1 的根结点为二叉树 B 中的根结点，T_1 中根结点的子树森林由 B 的左子树 L 转换而成，即 $F(L)=\{T_{11},T_{12},\cdots,T_{1m}\}$，$B$ 的右子树 R 转换成 $F(B)$中其余树组成的森林，即 $F(R)=\{T_2,\cdots,T_n\}$。

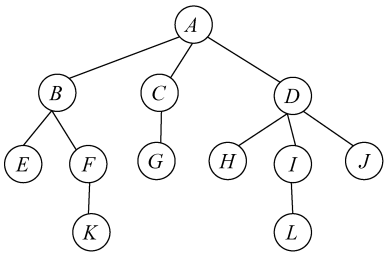
5.5.2　树的存储结构

在实际应用中，可以根据具体操作的特点将树设计成不同的存储结构。但无论采用何种存储方式，都要求树的存储结构不但能够存储各个结点本身的数据域信息，还要求能够准确反映树中各个结点之间的逻辑关系。这里介绍 4 种树的存储结构。

1. 双亲链表存储结构

双亲链表存储结构中的每个结点存放的信息既包含结点本身的数据域信息，又包含指示双亲结点在存储结构中的位置信息，所以这种存储结构可以设计成：用一组地址连续的存储单元存放树中的各个结点，每个结点有两个域，一个是数据域，用来存储树中该结点本身的值；另一个是指针域，用来存储该结点的双亲结点在存储结构中的位置信息。图 5-37 所示的是一棵树及其双亲链表存储结构。其中，图 5-37(b)中的 data 域是数据域，parent 域是指针域(注意：这里的指针域不是真正的指针类型，而是位置数值)。parent 域值为 −1，表示该结点无双亲，即根结点，如 A 结点；parent 域值为 2，表示该结点的双亲结点是数组下标序号为 2 的结点，如 G 结点的双亲结点是 C 结点。

由于双亲链表存储结构中的指针域是向上链接的，因而这种存储结构便于实现与指定结点的双亲结点或祖先结点有关的操作，但不便于实现与其孩子结点或子孙结点有关的操作。



(a) 一棵树

	data	parent
0	*A*	−1
1	*B*	0
2	*C*	0
3	*D*	0
4	*E*	1
5	*F*	1
6	*G*	2
7	*H*	3
8	*I*	3
9	*J*	3
10	*K*	5
11	*L*	8

(b) 双亲链表存储结构

图 5-37　树及其双亲链表存储结构

2. 孩子链表存储结构

在孩子链表存储结构中，除了存放结点本身的数据域信息外，还存放了其所有孩子结点在存储结构中的位置信息。由于每个结点的子结点数不同，则可将一个结点的所有孩子的位置信息按从左到右的顺序链接成一个单链表，称此单链表为该结点的孩子链表。因此，这种存储结构可以设计为：用一组地址连续的存储单元存放树中的各个结点，每个结点有两个域，一个是数据域，用来存储树中该结点的值；另一个是指针域，用来存放该结点的孩子链表的头指针。图 5-38 所示的是图 5-37(a)中树的孩子链表存储结构。其中，data 域是数据域，firstChild 域是指针域。孩子链表中的每个结点也有两个域，一个是 child 域，用来存放孩子结点在数组中的位置；另一个是 next 域，用来存放指向孩子链表中下一个结点的指针值。

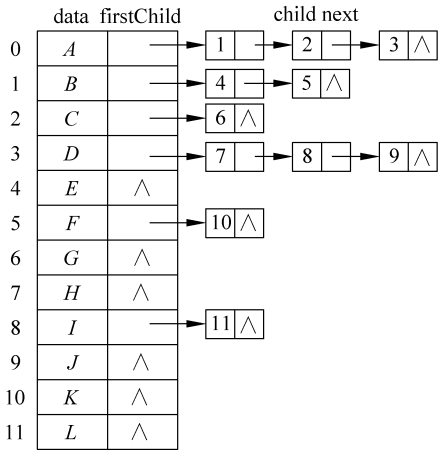


图 5-38　图 5-37(a)中树的孩子链表存储结构

这种存储结构与双亲链表存储结构正好相反，它便于实现与指定结点的孩子结点或子孙结点有关的操作，但不便于实现与其双亲结点或祖先结点有关的操作。

3. 双亲孩子链表存储结构

双亲孩子链表存储结构的设计方法与孩子链表存储结构类似,其主体仍然是一个存储树中各个结点信息的数组。只不过数组中的元素包括三个域,比孩子链表存储结构中多了一个存放该结点的双亲结点在数组中位置的指针域。图 5-39 所示的是图 5-37(a)中树的双亲孩子链表存储结构。

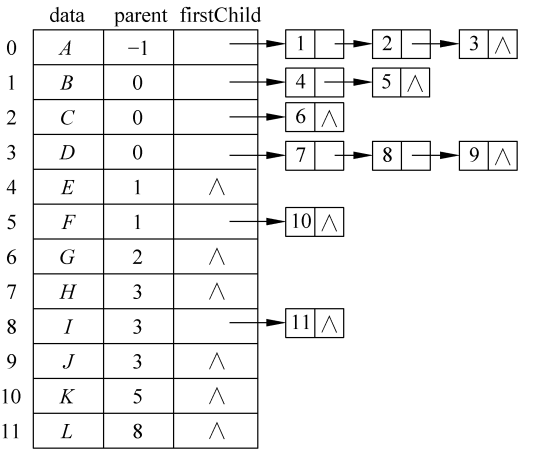


图 5-39　图 5-37(a)中树的双亲孩子链表存储结构

这种存储结构既便于实现与指定结点的孩子结点或子孙结点有关的操作,也便于实现与其双亲结点或祖先结点有关的操作。

4. 孩子兄弟链表存储结构

孩子兄弟链表存储结构又称为“左孩子/右兄弟”二叉链表存储结构,它类似于二叉树的二叉链表存储结构,其差别在于:其链表中每个结点的左指针指向该结点的第一个孩子,而右指针指向该结点的右邻兄弟。结点的存储结构如图 5-40 所示。图 5-41 给出了图 5-37(a)中树的孩子兄弟链表存储结构。

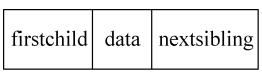


图 5-40　孩子兄弟链表中的结点结构

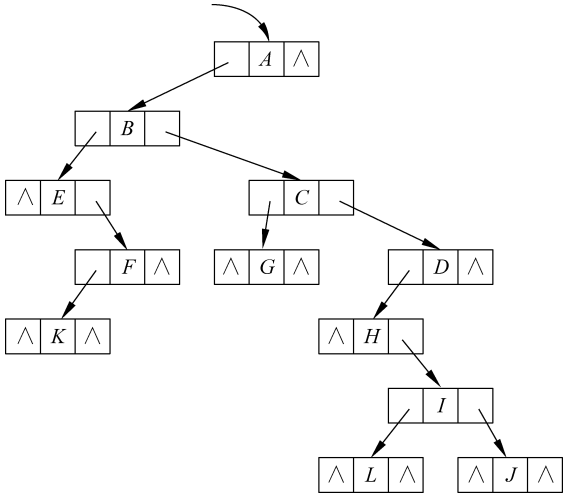


图 5-41　图 5-37(a)中树的孩子兄弟链表存储结构

树的孩子兄弟链表存储结构描述如下：

```
typedef struct CSNode
{
    TElemType data;                          //数据域
    struct CSNode * FirstChild;              //左孩子域
    struct CSNode * NextSibling;             //右兄弟域
}CSNode, * CSTree;
```

这种存储结构与树所对应的二叉树的二叉链表存储结构相同。一切对树的操作都可以通过这种方式转换成对二叉树的操作，所以这种存储方式应用最为广泛。

5.5.3 树和森林的遍历

1. 树的遍历

树可被看成是由树的根结点和根结点的所有子树所构成的森林两部分构成，因此树的遍历操作有先根遍历、后根遍历和层次遍历 3 种方式，下面分别给出它们的定义和它们在孩子兄弟链表存储结构下的实现算法。

1）先根遍历

若树为非空，则：

（1）访问根结点。

（2）从左到右依次先根遍历根结点的每棵子树。

对图 5-32 中的树进行先根遍历后得到的遍历序列为：$ABEFHCDG$。

说明：树的先根遍历序列与树所对应的二叉树的先根遍历序列相同。

【算法 5-18】 树的先根遍历递归算法。

```
void CSPreRootTraverse(CSTree T)
//树的先根遍历递归算法
{    if(T!= NULL)
     {  printf(" % c",T - > data);              //访问树的根结点
        CSPreRootTraverse(T - > FirstChild);  //先根遍历树中根结点的第一棵子树
        CSPreRootTraverse(T - > NextSibling); //先根遍历树中根结点的其他子树
     }
} //算法 5 - 18 结束
```

2）后根遍历

若树为非空，则：

（1）从左到右依次后根遍历根结点的每棵子树。

（2）访问根结点。

对图 5-32 中的树进行后根遍历后得到的遍历序列为：$EHFBCGDA$。

说明：树的后根遍历序列与树所对应的二叉树的中根遍历序列相同。

【算法 5-19】 树的后根遍历递归算法。

```
void CSPostRootTraverse(CSTree T)
 //树的后根遍历递归算法
{    if(T!= NULL)
     {   CSPostRootTraverse(T - > FirstChild);          //后根遍历树中根结点的第一棵子树
```

```
        printf("%c",T->data);                    //访问树的根结点
        CSPostRootTraverse(T->NextSibling);      //后根遍历树中根结点的其他子树
    }
} //算法 5-19 结束
```


3) 层次遍历

若树为非空,则从根结点开始,从上到下依次访问每层的各个结点,在同一层中的结点,则按从左到右的顺序依次进行访问。

对图 5-32 中的树进行层次遍历后得到的遍历序列为:$ABCDEFGH$。

【算法 5-20】 树的层次遍历算法。

```
void CSLevelTraverse(CSTree T)
//树的层次遍历算法
{  CSTree que[20]; int rear=0,front=0;                  //采用队列结构,初始化队首与队尾
   if(T!=NULL)
        que[rear++]=T;                                  //根结点入队
   while(front<rear)
   {   while(que[front]!=NULL)                          //当队列不空时
       {  printf("%c",que[front]->data);
          if(que[front]->FirstChild)
                que[rear++]=que[front]->FirstChild;     //树中根结点的第一棵子树入队
          que[front]=que[front]->NextSibling;           //树中根结点的其他子树入队
        }
       front++;
   }
}  //算法 5-20 结束
```

2. 森林的遍历

按照森林与树的定义,森林也可被看成是由第一棵树的根结点,第一棵树的根结点的子树所构成的森林,以及除第一棵树之外的其余树所构成的森林 3 部分构成。为此,可以得到森林的 3 种遍历方法:先根遍历、后根遍历和层次遍历。

1) 先根遍历

若森林为非空,则:

(1) 访问森林中第一棵树的根结点。

(2) 先根遍历第一棵树中根结点的子树所构成的森林。

(3) 先根遍历森林中除第一棵树之外的其他树所构成的森林。

对图 5-35(a)中的森林进行先根遍历后得到的遍历序列为:$ABCEDFGHIJKL$。

说明:对森林进行先根遍历操作等同于从左到右对森林中的每棵树进行先根遍历操作,并且森林的先根遍历序列与森林所对应的二叉树的先根遍历序列相同。

2) 后根遍历

若森林为非空,则:

(1) 后根遍历森林中第一棵树的根结点的子树所构成的森林。

(2) 访问森林中第一棵树的根结点。

(3) 后根遍历森林中除第一棵树之外的其他树所构成的森林。

对图 5-35(a)中的森林进行后根遍历后得到的遍历序列为：$BECDAGFIKLJH$。

说明：对森林进行后根遍历操作等同于从左到右对森林中的每棵树进行后根遍历操作，并且森林的后根遍历序列与森林所对应的二叉树的中根遍历序列相同。

3）层次遍历

若森林为非空，则按照从左到右的顺序对森林中的每棵树进行层次遍历。

对图 5-35(a)中的森林进行层次遍历，得到的遍历序列为：$ABCDEFGHIJKL$。

说明：对森林进行层次遍历操作等同于从左到右对森林中的每棵树进行层次遍历操作。

小　　结

本章首先介绍了树与二叉树的概念，读者应领会这两个概念的递归定义，并注意树与二叉树的结构差别，明确二叉树是一种应用非常广泛的重要的树结构。在二叉树中，每个结点至多只有两棵子树，这两棵子树仍是二叉树且这两棵子树有明确的左、右之分。要注意树结构与线性结构之间的差别，树结构是一种具有层次关系的非线性结构。在树结构中除根结点之外的其他结点只有一个前驱结点(或父结点)，而每个结点的后继结点(或孩子结点)可能有零个或多个。

完全二叉树和满二叉树是二叉树的两种特殊形态，要注意这两者的概念和它们之间的关系。二叉树具有 5 个重要性质，其中性质 4 是针对完全二叉树或满二叉树的，它说明了一个具有 n 个结点的二叉树其深度至少为$\lfloor \log_2 n \rfloor+1$。

树与二叉树的遍历是树与二叉树各种操作的基础，要求在掌握各种遍历递归算法的基础上，学会灵活运用遍历算法实现树与二叉树的其他操作，特别是二叉树的遍历算法，它是本章需要掌握的重点。对于二叉树，主要介绍了二叉树的先根遍历、中根遍历、后根遍历和层次遍历 4 种遍历方式；对于树，主要介绍了树的先根遍历，后根遍历以及层次遍历 3 种方式。实现树与二叉树遍历的具体算法与所采用的存储结构有关，二叉树的存储结构分为顺序存储和链式存储两种，顺序存储比较适合于满二叉树和完全二叉树。对于一般的二叉树，常用的存储结构是二叉链表存储结构；对于一般的树，在双亲链表、孩子链表、双亲孩子链表和孩子兄弟链表存储结构中，常用的是孩子兄弟链表存储结构。树与二叉树之间的转换最简单的方法就是通过树的孩子兄弟链表与二叉树的二叉链表存储方法实现转换。转换规则是：二叉树中结点的左孩子就是对应树中该结点的第一个孩子，而二叉树中结点的右孩子就是对应树中该结点相邻的右兄弟。也就是说，树的孩子兄弟链表存储结构与该树对应的二叉树的二叉链表存储结构是完全相同的。正因为它们之间的转换实现起来非常容易，所以对树的所有操作都可以转换成对它所对应的二叉树的相关操作来实现。例如，对树的先根遍历操作可以转化成对其相应的二叉树进行先根遍历操作；对树的后根遍历操作可以转化成对其相应的二叉树进行中根遍历操作。

二叉树的一种重要应用是哈夫曼树，利用哈夫曼树构造哈夫曼编码在通信领域具有广泛的应用，它为数据的压缩问题提供了解决方法。哈夫曼树的概念及构造方法也是本章重点介绍的内容。

习 题 5

一、客观测试题

扫码答题

二、算法设计题

1. 编写算法,统计一棵二叉树的叶结点数目。

2. 编写算法,在二叉树中查找值为 x 的结点。

3. 编写算法,将二叉树的每个结点的左、右子树进行交换。

4. 编写算法,求一棵二叉树的根结点 root 到一个指定结点 p 之间的路径并输出。

5. 编写算法,统计树(基于孩子兄弟链表存储结构)的叶结点数目。

6. 编写算法,计算树(基于孩子兄弟链表存储结构)的深度。

三、综合应用题

1. 假设一棵深度为 h 的满 k 叉树具有如下性质:第 h 层上的结点都是叶结点,其余各层上的每个结点都是 k 棵非空子树。若按层次顺序(同一层上自左向右)从1开始对所有结点编号,则:

(1) 各层上的结点个数是多少?

(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?

2. 若一棵二叉树中任意结点的值均不相同,则由二叉树的先根遍历序列和中根遍历序列,或由后根遍历序列和中根遍历序列均能唯一地确定这棵二叉树,但由先根遍历序列和后根遍历序列却不一定能够唯一地确定这棵二叉树,试证明上述结论。

3. 假设有一段正文为“THIS-IS-THE-THESTT-THAT-IT-TITHET-TETTEE”,它所使用的字符集为 $C=\{T,H,I,S,E,A,-\}$。请设计一套 C 中字符的二进制编码,使上述正文的二进制编码最短,并计算出存储上述正文的二进制编码所需要的字节数(每个字节由8个二进制位组成)(提示:以每个字符在正文中出现的频度为权值)。

习题5 主观题参考答案