第5章

树和二叉树

CHAPTER 5

前面几章分别介绍了几种常见的线性结构，本章的树与第 6 章的图属于非线性数据结构。线性结构中的每个元素有唯一的前驱元素和唯一的后继元素，即前驱元素和后继元素是一对一的关系。非线性结构中元素间的前驱和后继关系并不具有唯一性，树形结构结点间的关系是前驱唯一而后继不唯一，即结点间是一对多的关系；图结构结点间的关系是前驱和后继都不唯一，即结点间是多对多的关系；树形结构应用非常广泛，特别是在大量数据处理（如文件系统、编译系统、目录组织等）方面显得更加突出。

本章主要内容：

- 树的相关概念、性质及存储结构
- 二叉树的概念、性质及存储结构
- 二叉树的遍历
- 二叉树的线索化
- 树、森林和二叉树的相互转换
- 哈夫曼树的定义及编码实现

视频讲解

5.1 树

树是一种非线性的数据结构,树中各元素之间的关系是一对多的层次关系。

视频讲解

5.1.1 树的定义

树(tree)是 $n(n\geqslant 0)$个结点的有限集合。当 $n=0$ 时,称为空树;当 $n>0$ 时,称为非空树。非空树满足以下条件:

(1) 有且只有一个称为根(root)的结点。

(2) 当 $n>1$ 时,其余 $n-1$ 个结点可以划分为 m 个有限集合 $T_1,T_2,\cdots,T_m$,且这 m 个有限集合不相交。其中,$T_i(1\leqslant i\leqslant m)$也是一棵树,称为根的子树。

图 5-1 给出了一棵树的逻辑结构,它像一棵倒立的树。

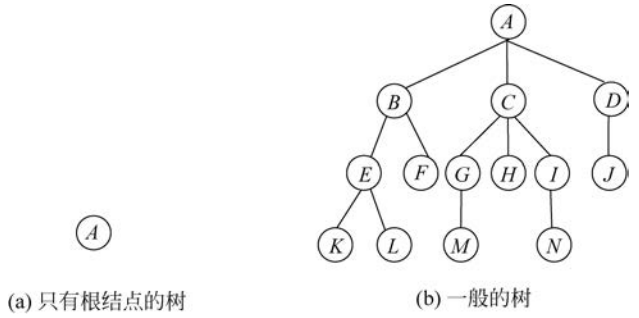


图 5-1 树的逻辑结构

在图 5-1 中,A 为根结点,左边的树只有根结点,右边的树有 14 个结点,除了根结点外,其余的 13 个结点分为 3 个不相交的子集:$T_1=\{B,E,F,K,L\}$、$T_2=\{C,G,H,I,M,N\}$和 $T_3=\{D,J\}$。其中,T_1、T_2 和 T_3 是根结点 A 的子树,并且它们本身也是一棵树。例如,T_2 的根结点是 C,其余的 5 个结点又分为 3 个不相交的子集:$T_{21}=\{G,M\}$、$T_{22}=\{H\}$和 $T_{23}=\{I,N\}$。其中,T_{21}、T_{22} 和 T_{23} 是 T_2 的子树,G 是 T_{21} 的根结点,$\{M\}$是 G 的子树,I 是 T_{23} 的根结点,$\{N\}$是 I 的子树。

表 5-1 是关于树的一些基本概念,表中举例基于图 5-1(b)的树。

表 5-1 树的基本概念

术　语	定　义	示　例
树的结点	包含一个数据元素及若干指向子树分支的信息	A、B、C、F 和 M 等都是结点
结点的度	一个结点拥有子树的个数称为结点的度	结点 C 有 3 个子树,度为 3
叶子结点	没有子树(度为 0)的结点称为叶子结点也称为终端结点	K、L、F、M、H、N 和 J 都是叶子结点
分支结点	度不为 0 的结点称为非终端结点,也称为非终端结点	B、C、D、E 等都是分支结点
孩子结点	一个结点的子树的根结点称为孩子结点	B 是 A 的孩子结点,E 是 B 的孩子结点,H 是 C 的孩子结点

续表

术　语	定　义	示　例
双亲结点	如果一个结点存在孩子结点，则该结点称为孩子结点的双亲结点，也称父结点	A 是 B 的双亲结点，B 是 E 的双亲结点，I 是 N 的双亲结点
子孙结点	一个根结点的子树中的任何一个结点都称为该根结点的子孙结点	$\{G,H,I,M,N\}$ 是 C 的子树，子树中的结点 G、H、I、M 和 N 都是 C 的子孙结点
祖先结点	从根结点开始到达一个结点，所经过的所有分支结点都称为该结点的祖先结点	N 的祖先结点为 A、C 和 I
兄弟结点	一个双亲结点的所有孩子结点之间互相称为兄弟结点	E 和 F 是 B 的孩子结点，因此 E 和 F 互为兄弟结点
树的度	树中所有结点的度的最大值	结点 C 的度为 3，结点 A 的度为 3，这两个结点的度是树中拥有最大的度的结点，因此树的度为 3
结点的层次	从根结点开始，根结点为第 1 层，根结点的孩子结点为第 2 层……如果某一个结点是第 L 层，则其孩子结点位于第 $L+1$ 层	A 的层次为 1，B 的层次为 2，G 的层次为 3，M 的层次为 4
树的深度	树中所有结点的层次最大值称为树的深度，也称为树的高度	树的深度为 4
有序树	如果树中各个子树的次序是有先后次序的，则称该树为有序树	
无序树	如果树中各个子树的次序是没有先后次序的，则称该树为无序树	
森林	m 棵互不相交的树构成一个森林。如果把一棵非空树的根结点删除，则该树就变成了一个森林，森林中的树由原来根结点的各个子树构成。如果把一个森林加上一个根结点，将森林中的树变成根结点的子树，则该森林就转换为一棵树	

5.1.2　树的逻辑表示

视频讲解

树的逻辑表示可分为 4 种：树形表示法、文氏图表示法、广义表表示法和凹入表示法。

(1) 树形表示法。图 5-1 就是树形表示法。树形表示法是最常用的一种表示法，它能直观、形象地表示出树的逻辑结构，能够清晰地反映出树中结点之间的逻辑关系。树中的结点使用圆圈表示，结点间的关系使用直线表示，位于直线上方的结点是双亲结点，直线下方的结点是孩子结点。

(2) 文氏图表示法。文氏图表示是利用数学中的集合来图形化描述树的逻辑关系。图 5-1 的树的文氏图表示如图 5-2 所示。

(3) 广义表表示法。采用广义表的形式表示树的逻辑结构，广义表的子表表示结点的子树。图 5-1 的树的广义表表示如下。

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

(4) 凹入表示法。图 5-1 的树的凹入表示如图 5-3 所示。

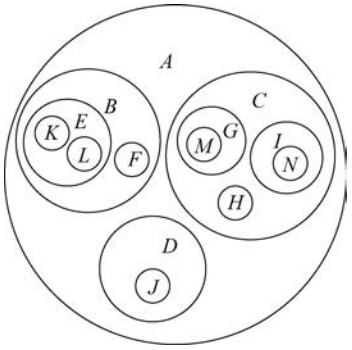


图 5-2 树的文氏图表示法

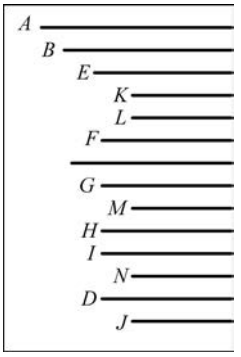


图 5-3 树的凹入表示法

在这 4 种树的表示法中,树形表示法最为常用。

5.1.3 树的抽象数据类型

树的抽象数据类型定义了树中的数据对象、数据关系及基本操作。树的抽象数据类型如表 5-2 所示。

表 5-2 树的抽象数据类型描述

数据对象	D 是具有相同特性的数据元素的集合	
数据关系	若 D 为空集,则称为空树。若 D 仅含一个数据元素,则 D 为空集,否则 D 与 H 的关系如下。 (1) 在 D 中存在唯一的根数据元素 root,它在关系 H 下无前驱。 (2) 若 $D-\{\text{root}\}\neq\varnothing$,则存在 $D-\{\text{root}\}$ 的一个划分 $D_1,D_2,\cdots,D_m(m>0)$,对任意的 $j\neq k(1\leqslant j,k\leqslant m)$,有 $D_j\cap D_k=\varnothing$,且对任意的 $i(1\leqslant i\leqslant m)$,唯一存在数据元素 $x_i\in D_i$,有 $<\text{root},x_i>\in H$。 (3) 对应于 $D-\{\text{root}\}$ 的划分,$H-\{<\text{root},x_1>\},\cdots,<\text{root},x_n>\}$ 有唯一的一个划分 $H_1,H_2,\cdots,H_m(m>0)$,对任意的 $j\neq k(1\leqslant j,k\leqslant m)$,有 $D_j\cap D_k=\varnothing$,且对任意的 $i(1\leqslant i\leqslant m)$,$H_i$ 是 D_i 上的二元关系。$(D_i,\{H_i\})$ 是一棵符合本定义的树,称为 root 的子树	
基本操作	InitTree(&T)	初始条件:树 T 不存在。 操作结果:构造空树 T
	DestroyTree(&T)	初始条件:树 T 存在。 操作结果:销毁树 T
	CreateTree(&T)	初始条件:树 T 存在。 操作结果:根据给定条件构造树 T
	TreeEmpty(T)	初始条件:树 T 存在。 操作结果:若树 T 为空树,则返回 1;否则,返回 0
	Root(T)	初始条件:树 T 存在。 操作结果:若树 T 非空,则返回树的根结点;否则,返回 null
	Parent(T,e)	初始条件:树 T 存在,e 是 T 中的某个结点。 操作结果:若 e 不是根结点,则返回该结点的双亲;否则返回空

续表

基本操作	FirstChild(T,e)	初始条件：树 T 存在，e 是 T 中的某个结点。 操作结果：若 e 是树 T 的非叶子结点，则返回该结点的第一个孩子结点；否则，返回 null
	NextSibling(T,e)	初始条件：树 T 存在，e 是 T 中的某个结点。 操作结果：若 e 不是其双亲结点的最后一个孩子结点，则返回它的下一个兄弟结点；否则，返回 null
	InsertChild(&T, p, Child)	初始条件：树 T 存在，p 指向 T 中的某个结点，非空树 Child 与 T 不相交且无右子树。 操作结果：将非空树 Child 插入 T 中，使 Child 成为 p 指向的结点的子树
	DeleteChild(&T, p, i)	初始条件：树 T 存在，p 指向 T 中的某个结点，$1 \leqslant i \leqslant d$，$d$ 为 p 所指向结点的度。 操作结果：将 p 所指向的结点的第 i 棵子树删除。如果删除成功，则返回 1；否则，返回 0
	TraverseTree(T)	初始条件：树 T 存在。 操作结果：按照某种次序对 T 的每个结点访问且仅访问一次
	TreeDepth(T)	初始条件：树 T 存在。 操作结果：若树 T 非空，返回树的深度；否则，返回 0

5.2　二叉树

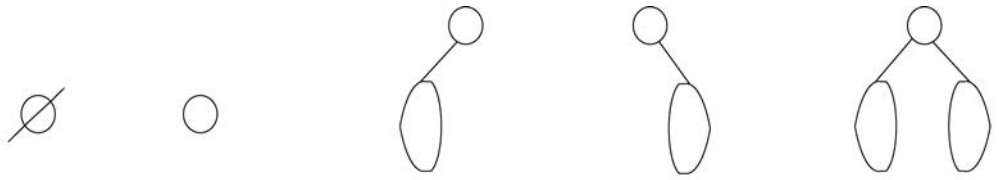
在深入学习树之前，需要先认识一种比较简单的树——二叉树。

5.2.1　二叉树的定义

视频讲解

二叉树(binary tree)是另一种树结构，它的特点是每个结点最多只有两棵子树。在二叉树中，每个结点的度只可能是 0、1 或 2，每个结点的孩子结点有左右之分，位于左边的孩子结点称为左孩子结点或左孩子，位于右边的孩子结点称为右孩子结点或右孩子。如果 $n=0$，则称该二叉树为空二叉树。

下面给出二叉树的 5 种基本形态，如图 5-4 所示。



(a) 空二叉树　(b) 只有根结点的二叉树　(c) 只有左子树的二叉树　(d) 只有右子树的二叉树　(e) 左、右子树非空的二叉树

图 5-4　二叉树的 5 种基本形态

一个由 12 个结点构成的二叉树如图 5-5 所示。F 是 C 的左孩子结点，G 是 C 的右孩子结点，L 是 G 的右孩子结点，G 的左孩子结点不存在。

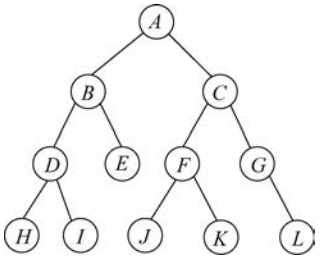


图 5-5 二叉树示意图

对于深度为 k 的二叉树,若结点数为 2^k-1,即除了叶子结点外,其他结点都有两个孩子结点,则称这样的二叉树为满二叉树。在满二叉树中,每层的结点都具有最大的结点个数,每个结点的度为 2 或 0(即叶子结点),不存在度为 1 的结点。从根结点出发,从上到下、从左到右地依次对每个结点进行连续编号。一棵深度为 4 的满二叉树及其编号如图 5-6 所示。

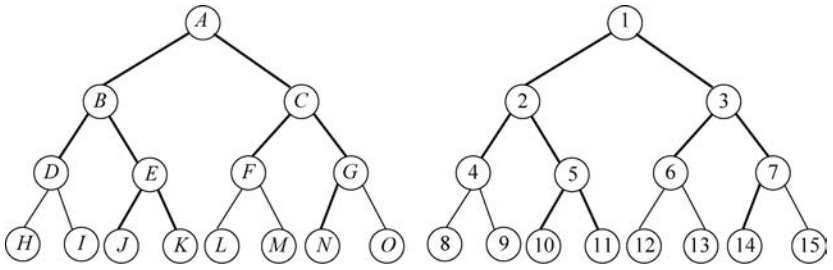


图 5-6 一棵深度为 4 的满二叉树及其编号

如果一棵二叉树有 n 个结点,并且二叉树的 n 个结点的结构与满二叉树的前 n 个结点的结构完全相同,则称这样的二叉树为完全二叉树。完全二叉树及其编号如图 5-7 所示。图 5-8 所示为一棵非完全二叉树。

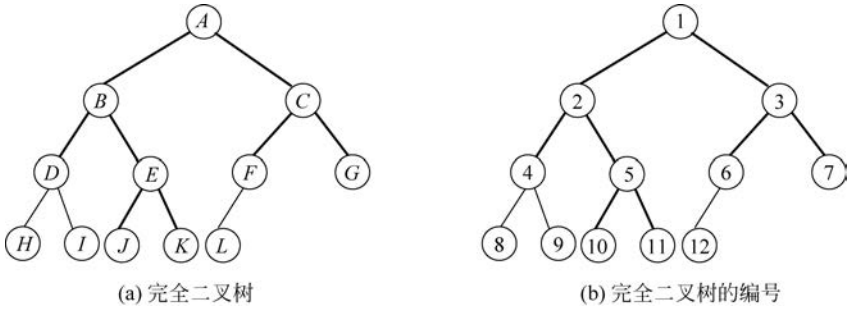


图 5-7 完全二叉树及其编号

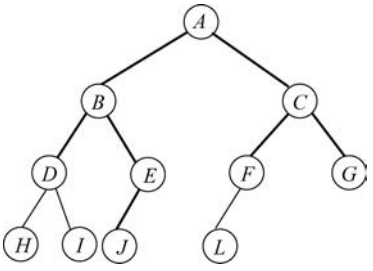


图 5-8 非完全二叉树

由此可以看出，如果二叉树的层数为 k，则满二叉树的叶子结点一定是在第 k 层，而完全二叉树的叶子结点一定在第 k 层或第 k-1 层。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。

5.2.2　二叉树的性质

视频讲解

二叉树具有以下重要的性质。

性质 1　在二叉树中，第 $m(m\geqslant 1)$层上至多有 2^{m-1} 个结点(规定根结点为第 1 层)。

证明：利用数学归纳法证明。

当 $m=1$ 时，即根结点所在的层次，有 $2^{m-1}=2^{1-1}=2^0=1$，命题成立。

假设当 $m=k$ 时，命题成立，即第 k 层至多有 2^{k-1} 个结点。因为在二叉树中，每个结点的度最大为 2，则第 $k+1$ 层结点的个数最多是第 k 层的 2 倍，即 $2\times 2^{k-1}=2^{k-1+1}=2^k$。当 $m=k+1$ 时，命题成立。

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

证明：第 i 层结点的个数最多为 2^{i-1}，将深度为 k 的二叉树中的每层结点的最大值相加，即可得到二叉树中结点的最大值。因此，深度为 k 的二叉树的结点总数至多为

$$\sum_{i=1}^{k}(\text{第 } i \text{ 层结点的最大个数})=\sum_{i=1}^{k}2^{i-1}=2^0+2^1+\cdots+2^{k-1}=\frac{2^0(2^k-1)}{2-1}=2^k-1。$$

由此得证，命题成立。

性质 3　对任何一棵二叉树 T，如果叶子结点总数为 n_0，度为 2 的结点总数为 n_2，则有 $n_0=n_2+1$。

证明：假设在二叉树中，结点总数为 n，度为 1 的结点总数为 n_1。二叉树中结点的总数 n 等于度为 0、度为 1 和度为 2 的结点总数的和，即 $n=n_0+n_1+n_2$。

假设二叉树的分支数为 Y。在二叉树中，除了根结点外，每个结点都存在一个进入的分支，所以有 $n=Y+1$。

又因为二叉树的所有分支都是由度为 1 和度为 2 的结点发出，所以分支数 $Y=n_1+2\times n_2$。故 $n=Y+1=n_1+2\times n_2+1$。

联合 $n=n_0+n_1+n_2$ 和 $n=n_1+2\times n_2+1$ 两式，得到 $n_0+n_1+n_2=n_1+2\times n_2+1$，即 $n_0=n_2+1$。

由此得证，命题成立。

性质 4　如果完全二叉树有 n 个结点，则深度为 $\lfloor\log_2 n\rfloor+1$。符号 $\lfloor x\rfloor$ 表示不大于 x 的最大整数，而 $\lceil x\rceil$ 表示不小于 x 的最小整数。

证明：假设具有 n 个结点的完全二叉树的深度为 k。k 层完全二叉树的结点个数介于 $k-1$ 层满二叉树与 k 层满二叉树结点个数之间。根据性质 2，$k-1$ 层满二叉树的结点总数为 $n_1=2^{k-1}-1$，k 层满二叉树的结点总数为 $n_2=2^k-1$。因此有 $n_1<n\leqslant n_2$，即 $n_1+1\leqslant n<n_2+1$，且 $n_1=2^{k-1}-1$，$n_2=2^k-1$，故得到 $2^{k-1}-1\leqslant n<2^k-1$。同时对该不等式两边取对数，有 $k-1\leqslant\log_2 n<k$。因为 k 是整数，$k-1$ 也是整数，所以 $k-1=\lfloor\log_2 n\rfloor$，即 $k=\lfloor\log_2 n\rfloor+1$。由此得证，命题成立。

性质 5 如果完全二叉树有 n 个结点,按照从上到下、从左到右的顺序对二叉树中的每个结点从 1～n 进行编号,则对于任意结点 i 有以下性质:

5.1 如果 $i=1$,则序号 i 对应的结点就是根结点,该结点没有双亲结点。如果 $i>1$,则序号为 i 的结点的双亲结点序号为 $\lfloor i/2 \rfloor$。

5.2 如果 $2i>n$,则序号为 i 的结点没有左孩子结点。如果 $2\times i\leqslant n$,则序号为 i 的结点的左孩子结点序号为 $2i$。

5.3 如果 $2i+1>n$,则序号为 i 的结点没有右孩子结点。如果 $2i+1\leqslant n$,则序号为 i 的结点的右孩子结点序号为 $2i+1$。

证明:(1)利用性质 5.2 和性质 5.3 证明性质 5.1。当 $i=1$ 时,该结点一定是根结点,根结点没有双亲结点。当 $i>1$ 时,假设序号为 m 的结点是序号为 i 的结点的双亲结点。如果序号为 i 的结点是序号为 m 的结点的左孩子结点,则根据性质 5.2 有 $2m=i$,即 $m=i/2$。如果序号为 i 的结点是序号为 m 的结点的右孩子结点,则根据性质 5.3 有 $2m+1=i$,即 $m=(i-1)/2=i/2-1/2$。综合以上两种情况,当 $i>1$ 时,序号为 i 的结点的双亲结点序号为 $\lfloor i/2 \rfloor$。结论成立。

(2) 利用数学归纳法证明。当 $i=1$ 时,有 $2i=2$,如果 $2>n$,则二叉树中不存在序号为 2 的结点,也就不存在序号为 i 的左孩子结点。如果 $2\leqslant n$,则该二叉树中存在两个结点,序号 2 是序号为 i 的结点的左孩子结点序号。

假设序号 $i=k$,当 $2k\leqslant n$ 时,序号为 k 的结点的左孩子结点存在且序号为 $2k$;当 $2k>n$ 时,序号为 k 的结点的左孩子结点不存在。

当 $i=k+1$ 时,在完全二叉树中,如果序号为 $k+1$ 的结点的左孩子结点存在 $2i\leqslant n$,则其左孩子结点的序号为 k 的结点的右孩子结点序号加 1,即序号为 $k+1$ 的结点的左孩子结点序号为 $(2k+1)+1=2\times(k+1)=2i$。因此,当 $2i>n$ 时,序号为 i 的结点的左孩子不存在。结论成立。

(3) 同理,利用数学归纳法证明。当 $i=1$ 时,如果 $2i+1=3>n$,则该二叉树中不存在序号为 3 的结点,即序号为 i 的结点的右孩子结点不存在。如果 $2i+1=3\leqslant n$,则该二叉树存在序号为 3 的结点,且序号为 3 的结点是序号 i 的结点的右孩子结点。

假设序号 $i=k$,当 $2k+1\leqslant n$ 时,序号为 k 的结点的右孩子结点存在且序号为 $2k+1$,当 $2k+1>n$ 时,序号为 k 的结点的右孩子结点不存在。

当 $i=k+1$ 时,在完全二叉树中,如果序号为 $k+1$ 的结点的右孩子结点存在 $2i+1\leqslant n$,则其右孩子结点的序号为 k 的结点的右孩子结点序号加 2,即序号为 $k+1$ 的结点的右孩子结点序号为 $(2k+1)+2=2\times(k+1)+1=2i+1$。因此,当 $2i+1>n$ 时,序号为 i 的结点的右孩子不存在。结论成立。

视频讲解

5.2.3 二叉树的抽象数据类型

二叉树的抽象数据类型定义了二叉树中的数据对象、数据关系及基本操作。二叉树的抽象数据类型描述如表 5-3 所示。

表 5-3　二叉树的抽象数据类型描述

数据对象	D 是具有相同特性的数据元素的集合	
数据关系	若 $D=\varnothing$，则称为空二叉树。 若 $D\neq\varnothing$，则 D 与 H 的关系如下 (1) 在 D 中存在唯一的根数据元素 root，它在关系 H 下无前驱。 (2) 若 D-{root}$\neq\varnothing$，则存在 D-{root}$=\{D_l,D_r\}$，且 $D_l\cap D_r=\varnothing$。 (3) 若 $D_l\neq\varnothing$，则 D_1 中存在唯一的元素 x_1，$<$root，$x_1>\in H$，且存在 D_1 上的关系 $H_1\subset H$；若 $Dr\neq\varnothing$，则 D_r 中存在唯一的元素 x_r，$<$root，$x_r>\in H$，且存在 Dr 上的关系 $H_r\subset H$；$H=\{<$root，$x_1>$，$<$root，$x_r>$，H_l，$H_r\}$。 (4) (D_1，$\{H_1\}$)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	InitBiTree(&T)	初始条件：二叉树 T 不存在。 操作结果：构造空二叉树 T
	CreateBiTree(&T)	初始条件：给出了二叉树 T 的定义。 操作结果：创建一棵非空的二叉树 T
	DestroyBiTree(&T)	初始条件：二叉树 T 存在。 操作结果：销毁二叉树 T
	InsertLeftChild (p，c)	初始条件：二叉树 c 存在且非空。 操作结果：将 c 插入 p 所指向的左子树，使 p 所指结点的左子树成为 c 的右子树
	InsertRightChild(p，c)	初始条件：二叉树 c 存在且非空。 操作结果：将 c 插入 p 所指向的右子树，使 p 所指结点的右子树成为 c 的右子树
	LeftChild(&T，e)	初始条件：二叉树 T 存在，e 是 T 中的某个结点。 操作结果：若结点 e 存在左孩子结点，则将 e 的左孩子结点返回，否则，返回空
	RigthChild(&T，e)	初始条件：二叉树 T 存在，e 是 T 的某个结点。 操作结果：若结点 e 存在右孩子结点，则将 e 的右孩子结点返回，否则返回空
	DeleteLeftChild (&T，p)	初始条件：二叉树 T 存在，p 指向 T 中的某个结点。 操作结果：将 p 所指向的结点的左子树删除。如果删除成功，则返回 1，否则，返回 0
	DeleteRightChild (&T，p)	初始条件：二叉树 T 存在，p 指向 T 中的某个结点。 操作结果：将 p 所指向的结点的右子树删除。如果删除成功，则返回 1，否则，返回 0
	PreOrderTraverse (T)	初始条件：二叉树 T 存在。 操作结果：先序遍历二叉树 T。即先访问根结点，再访问左子树，最后访问右子树，对二叉树中的每个结点访问且仅访问一次
	InOrderTraverse(T)	初始条件：二叉树 T 存在。 操作结果：中序遍历二叉树 T。即先访问左子树，再访问根结点，最后访问右子树，对二叉树中的每个结点访问且仅访问一次
	PostOrderTraverse (T)	初始条件：二叉树 T 存在。 操作结果：后序遍历二叉树 T。即先访问左子树，再访问右子树，最后访问根结点，对二叉树中的每个结点访问且仅访问一次

续表

基本操作	LevelTraverse(T)	初始条件：二叉树 T 存在。 操作结果：对二叉树进行层次遍历。即按照从上到下、从左到右的顺序依次对二叉树中的每个结点进行访问
	BiTreeDepth(T)	初始条件：二叉树 T 存在。 操作结果：若二叉树非空，则返回二叉树的深度；若二叉树为空，则返回 0

视频讲解

5.2.4 二叉树的存储表示

二叉树的存储结构有两种：顺序存储表示和链式存储表示。

1. 二叉树的顺序存储

完全二叉树中的结点编号可以通过公式计算得到，因此，完全二叉树的存储可以按照从上到下、从左到右的顺序依次存储在一维数组中。完全二叉树的顺序存储如图 5-9 所示。

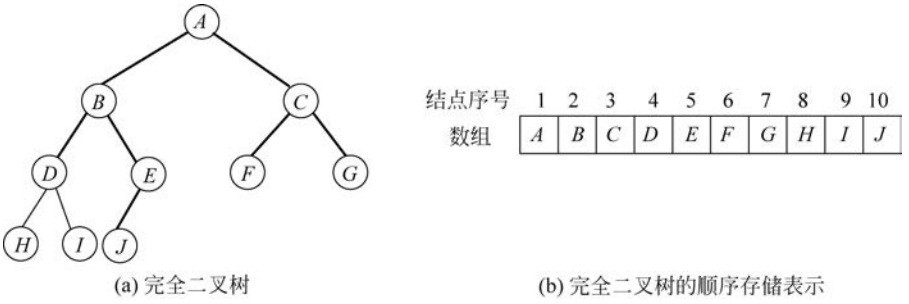


图 5-9 完全二叉树的顺序存储表示

按照从上到下、从左到右的顺序将非完全二叉树也进行同样的编号，将结点依次存放在一维数组中。为了能够正确反映二叉树中结点之间的逻辑关系，需要在一维数组中将二叉树中不存在的结点位置空出，并用'^'填充。非完全二叉树的顺序存储结构如图 5-10 所示。

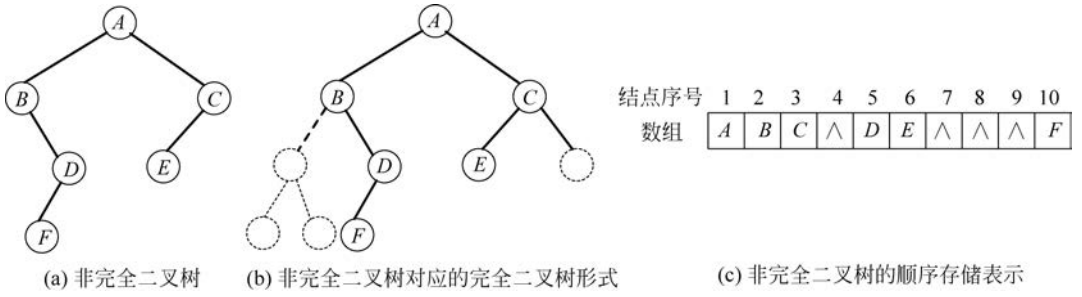


图 5-10 非完全二叉树的顺序存储表示

顺序存储对于完全二叉树来说是比较适合的，因为采用顺序存储能够节省内存单元，并能够利用公式得到每个结点的存储位置。但是，对于非完全二叉树来说，这种存储方式会造成内存空间的浪费。在最坏的情况下，如果每个结点只有右孩子结点而没有左孩子结点则需要占用 2^k-1 个存储单元，而实际上该二叉树只有 k 个结点。

2. 二叉树的链式存储

在二叉树中，每个结点有一个双亲结点和两个孩子结点。从一棵二叉树的根结点开始，通过结点的左右孩子地址就可以找到二叉树的每一个结点。因此，二叉树的链式存储结构包括 3 个域：数据域、左孩子指针域和右孩子指针域。其中，数据域存放结点的值，左孩子指针域指向左孩子结点，右孩子指针域指向右孩子的结点。这种链式存储结构称为二叉链表存储结构，如图 5-11 所示。

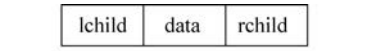


图 5-11　二叉链表的结点结构

如果二叉树采用二叉链表存储结构表示，则其二叉树的存储表示如图 5-12 所示。

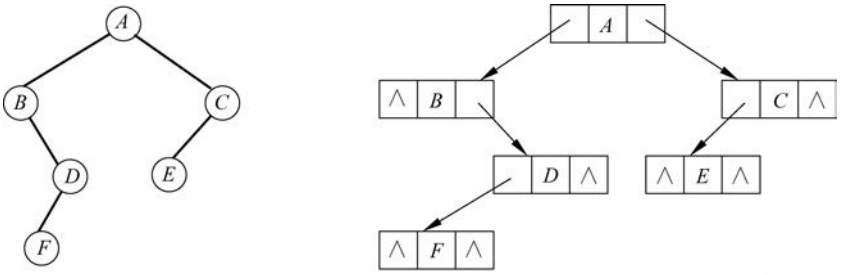


图 5-12　非完全二叉树的二叉链表存储表示

有时为了方便找到结点的双亲结点，在二叉链表的存储结构中增加一个指向双亲结点的指针域 parent。这种存储结构称为三叉链表结点存储结构，如图 5-13 所示。

图 5-13　三叉链表结点结构

通常情况下，二叉树采用二叉链表进行表示。二叉链表存储结构的类型定义描述如下：

```
class BiTreeNode                          //二叉树的结点类型
{
    char data;
    BiTreeNode lchild,rchild;
    BiTreeNode(char data)
    {
        this.data = data;
        lchild = null;
        rchild = null;
    }
}
```

在定义了二叉树的存储结点后，为了实现二叉树的插入、删除、遍历和线索化，必须先创建二叉树。二叉树的操作可通过定义 BiTree 类来实现。二叉树的初始化如下：

```
public class BiTree {
    BiTreeNode root;
    int num;
    static int pi = 0;
    final int MAXSIZE = 20;
    BiTree()
    {
```

```
            root = new BiTreeNode();
            num = 0;
        }
    }
```

创建二叉树的算法实现如下：

```
BiTreeNode CreateBiTree(char str[])
{
    if (str[pi] == '#')                    //本层构建 root、root.lchild、root.rchild 三个结点
{
        ++pi;
        return null;
    }
    BiTreeNode root = new BiTreeNode();
    root.data = str[pi];
    ++pi;
    root.lchild = CreateBiTree(str);       //构造左子树
    root.rchild = CreateBiTree(str);       //构造右子树
    return root;                           //递归结束返回构造好的树的根结点
}
```

5.3 二叉树的遍历

在二叉树的应用中，常常需要对二叉树中的每个结点进行访问，即二叉树的遍历。

视频讲解

5.3.1 二叉树遍历的定义

二叉树的遍历，即按照某种规律对二叉树的每个结点进行访问，使得每个结点仅被访问一次。这里的访问可以是对结点的输出、统计结点的个数等。

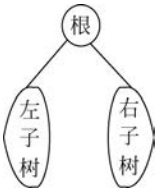


图 5-14 二叉树结点的基本结构

二叉树的遍历过程是将二叉树的非线性序列转换成一个线性序列的过程。二叉树是一种非线性的结构，通过遍历二叉树，按照某种规律对二叉树中的每个结点访问一次，即可得到一个顺序序列。

由二叉树的定义，二叉树是由根结点、左子树和右子树构成的。将这 3 个部分依次遍历，就完成了整个二叉树的遍历。二叉树结点的基本结构如图 5-14 所示。如果用 D、L、R 分别代表遍历根结点、遍历左子树和遍历右子树，则根据组合原理，可以得到 6 种遍历方案：DLR、DRL、LDR、LRD、RDL 和 RLD。

如果限定先左后右的次序，则只剩下 3 种方案：DLR、LDR 和 LRD。其中，DLR 称为先序遍历，LDR 称为中序遍历，LRD 称为后序遍历。

5.3.2 二叉树的先序遍历

二叉树先序遍历的递归定义如下。

如果二叉树为空，则执行空操作；如果二叉树非空，则执行以下操作：

（1）访问根结点；

（2）先序遍历左子树；

（3）先序遍历右子树。

根据二叉树先序遍历的递归定义，可以得到图 5-15 的二叉树的先序序列为 $ABDGEHICFJ$。

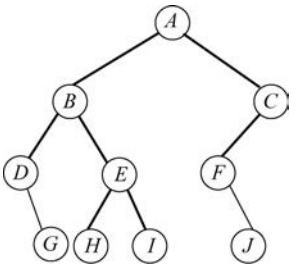


图 5-15　二叉树

在二叉树先序遍历的过程中，对每一棵二叉树重复执行以上的递归遍历操作，就可以得到先序序列。例如，在遍历根结点 A 的左子树$\{B,D,E,G,H,I\}$时，根据先序遍历的递归定义，先访问根结点 B，然后遍历 B 的左子树$\{D,G\}$，最后遍历 B 的右子树$\{E,H,I\}$。在访问 B 之后，开始遍历 B 的左子树$\{D,G\}$，先访问根结点 D，因为 D 没有左子树，所以遍历其右子树，右子树只有一个结点 G，所以访问 G。B 的左子树遍历完毕，按照以上方法遍历 B 的右子树。最后得到结点 A 的左子树先序序列：$BDGEHI$。

依据二叉树的先序递归定义，可以得到二叉树的先序递归算法。

```
public void PreOrderTraverse(BiTreeNode T)
 //先序遍历二叉树的递归实现
{
    if(T!= null) {
        System.out.print(T.data + " ");  //访问根结点
        nodal point(T.lchild);           //先序遍历左子树
        subtree(T.rchild);               //先序遍历右子树
    }
}
```

下面介绍二叉树的非递归算法实现。在第 4 章学习栈时，已经对递归的消除进行了具体讲解，现在利用栈来实现二叉树的非递归算法。

算法实现：从二叉树的根结点开始，访问根结点，然后将根结点的指针入栈，重复执行以下两个步骤。

（1）如果该结点的左孩子结点存在，则访问左孩子结点，并将左孩子结点的指针入栈。重复执行此操作，直到结点的左孩子不存在。

（2）将栈顶的元素（指针）出栈，如果该指针指向的右孩子结点存在，则将当前指针指向右孩子结点。

重复执行以上两个步骤，直到栈空为止。

以上算法思想的执行流程如图 5-16 所示。

二叉树先序遍历的非递归算法实现如下。

```
public void PreOrderTraverse2(BiTreeNode T)
//二叉树先序遍历的非递归实现
{
    BiTreeNode stack[ ] = new BiTreeNode[MAXSIZE];     //定义一个栈,用于存放结点的指针
    int top = 0;                          //定义栈顶指针, 初始化栈
    BiTreeNode p = T;
    while(p != null || top > 0) {
        while (p != null)                 //如果 p 不空,则访问根结点,遍历左子树
        {
```

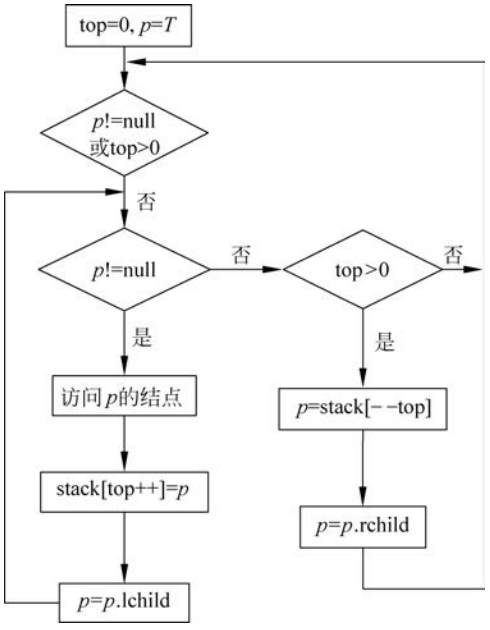


图 5-16　二叉树非递归先序遍历的执行流程图

```
            System.out.print(p.data + " ");          //访问根结点
            stack[top++] = p;
            p = p.lchild;                            //遍历左子树
        }
        if(top > 0)                                  //如果栈不空
        {
            p = stack[ --top];                       //栈顶元素出栈
            p = p.rchild;                            //遍历右子树
        }
    }
}
```

以上算法是直接利用数组来模拟栈的实现,当然也可以定义一个栈类型实现。如果用链式栈实现,则需要将数据类型改为指向二叉树结点的指针类型。

视频讲解

5.3.3　二叉树的中序遍历

二叉树中序遍历的递归定义如下。

如果二叉树为空,则执行空操作;如果二叉树非空,则执行以下操作:

(1) 中序遍历左子树;

(2) 访问根结点;

(3) 中序遍历右子树。

根据二叉树中序遍历的递归定义,图 5-15 的二叉树的中序序列为 $DGBHEIAFJC$。

在二叉树中序遍历的过程中,对每一棵二叉树重复执行以上的递归遍历操作,就可以得到二叉树的中序序列。

例如,如果要中序遍历 A 的左子树$\{B,D,E,G,H,I\}$,根据中序遍历的递归定义,需要先中序遍历 B 的左子树$\{D,G\}$,然后访问根结点 B,最后中序遍历 B 的右子树$\{E,H,I\}$。

在子树{D,G}中,D 是根结点,没有左子树,因此访问根结点 D,接着遍历 D 的右子树,因为右子树只有一个结点 G,所以直接访问 G。

在左子树遍历完毕后,访问根结点 B。遍历 B 的右子树{E,H,I},E 是子树{E,H,I}的根结点,需要先遍历左子树{H},因为左子树只有一个 H,所以直接访问 H,然后访问根结点 E;最后遍历右子树{I},右子树也只有一个结点,所以直接访问 I,B 的右子树访问完毕。因此,A 的右子树的中序序列为 $DGBHE$ 和 I。

从中序遍历的序列可以看出,A 左边的序列是 A 的左子树元素,A 右边的序列是 A 的右子树元素。同样,B 的左、右序列分别为其左、右子树元素。根结点把二叉树的中序序列分为左右两棵子树序列,左边为左子树序列,右边为右子树序列。

依据二叉树的中序递归定义,可以得到二叉树的中序递归算法如下。

```
public void InOrderTraverse(BiTreeNode T)
//中序遍历二叉树的递归实现
{
    if(T!= null) {
        InOrderTraverse(T.lchild);        //中序遍历左子树
        subtree(T.data + " ");            //访问根结点
        nodal point(T.rchild);            //中序遍历右子树
    }
}
```

下面介绍二叉树中序遍历的非递归算法实现。

二叉树中序遍历的非递归算法实现:从二叉树的根结点开始,将根结点的指针入栈,执行以下两个步骤。

(1) 如果该结点的左孩子结点存在,则将左孩子结点的指针入栈。重复执行此操作,直到结点的左孩子不存在。

(2) 将栈顶的元素(指针)出栈,并访问该指针指向的结点,如果该指针指向的右孩子结点存在,则将当前指针指向右孩子结点。

重复执行以上两个步骤,直到栈空为止。

以上算法思想的执行流程如图 5-17 所示。

二叉树中序遍历的非递归算法实现如下。

```
public void InOrderTraverse2(BiTreeNode T)
//中序遍历二叉树的非递归实现
{
    BiTreeNode stack[] = new BiTreeNode[MAXSIZE];     //定义一个栈,用于存放结点的指针
    int top = 0;                                      //定义栈顶指针, 初始化栈
    BiTreeNode p = T;
    while (p != null || top > 0) {
        while (p != null)                             //如果 p 不空,则遍历左子树
        {
            stack[top++] = p;                         //将 p 入栈
            p = p.lchild;                             //遍历左子树
        }
        if (top > 0)                                  //如果栈不空
        {
            p = stack[ -- top];                       //栈顶元素出栈
            System.out.print(p.data + " ");           //访问根结点
```

```
                p = p.rchild;                    //遍历右子树
            }
        }
    }
```

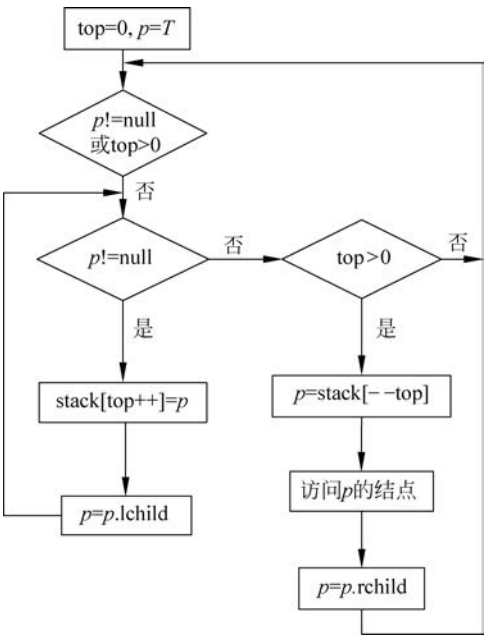


图 5-17 二叉树非递归中序遍历的执行流程图

视频讲解

5.3.4 二叉树的后序遍历

二叉树后序遍历的递归定义如下。

如果二叉树为空,则执行空操作;如果二叉树非空,则执行以下操作:

(1) 后序遍历左子树;

(2) 后序遍历右子树;

(3) 访问根结点。

根据二叉树后序遍历的递归定义,图 5-15 的二叉树的后序序列为 $GDHIEBJFCA$。

在二叉树后序遍历的过程中,对每一棵二叉树重复执行以上的递归遍历操作,就可以得到二叉树的后序序列。

例如,如果要后序遍历 A 的左子树$\{B,D,E,G,H,I\}$,根据后序遍历的递归定义,需要先后序遍历 B 的左子树$\{D,G\}$,然后后序遍历 B 的右子树$\{E,H,I\}$,最后访问根结点 B。在子树$\{D,G\}$中,D 是根结点,没有左子树,因此遍历 D 的右子树,因为右子树只有一个结点 G,所以直接访问 G,接着访问根结点 D。

在左子树遍历完毕后,需要遍历 B 的右子树$\{E,H,I\}$。E 是子树$\{E,H,I\}$的根结点,需要先遍历左子树$\{H\}$,因为左子树只有一个 H,所以直接访问 H;然后遍历右子树$\{I\}$,右子树也只有一个结点,所以直接访问 I,最后访问子树$\{E,H,I\}$的根结点 E。B 的左、右子树均访问完毕,此时访问结点 B。因此,A 的右子树的后序序列为 $GDHIEB$。

依据二叉树的后序递归定义,可以得到二叉树的后序递归算法如下。

```
public void PostOrderTraverse(BiTreeNode T)
 //后序遍历二叉树的递归实现
{
    if(T!= null)                          //如果二叉树不为空
    {
        PostOrderTraverse(T.lchild);      //后序遍历左子树
        subtree(T.rchild);                //后序遍历右子树
        subtree(T.data + " ");            //访问根结点
    }
}
```

下面介绍二叉树后序遍历的非递归算法实现。

二叉树后序遍历的非递归算法实现：从二叉树的根结点开始，将根结点的指针入栈，执行以下两个步骤。

（1）如果该结点的左孩子结点存在，则将左孩子结点的指针入栈。重复执行此操作，直到结点的左孩子不存在。

（2）取栈顶元素（指针）并赋给 p，如果 p.rchild＝＝null 或 p.rchild＝q，即 p 没有右孩子或右孩子结点已经访问过，则访问 p 指向的根结点，并用 q 记录刚刚访问过的结点指针，将栈顶元素退栈；如果 p 有右孩子且右孩子结点没有被访问过，则执行 $p=p$.rchild。重复执行以上两个步骤，直到栈空为止。

以上算法思想的执行流程如图 5-18 所示。

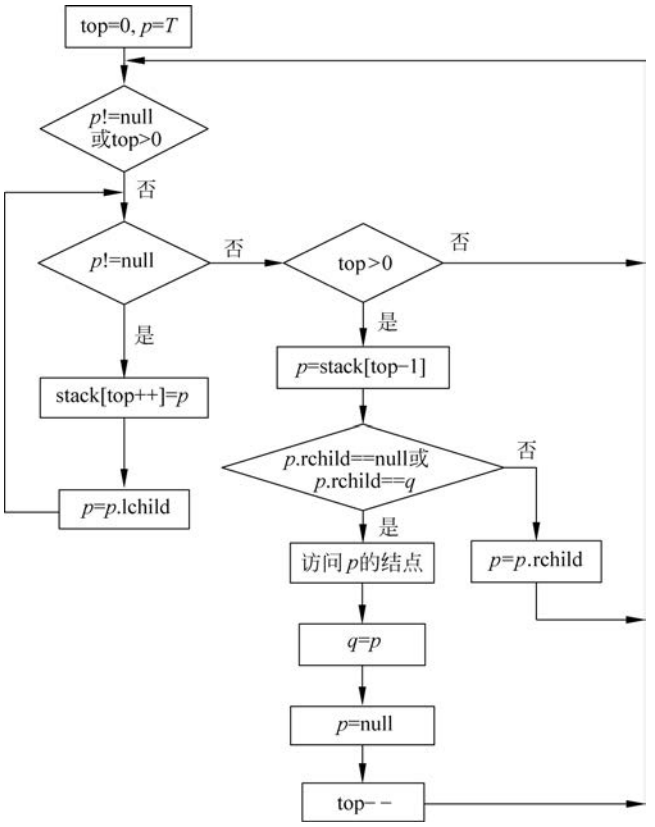


图 5-18　二叉树非递归后序遍历的执行流程图

二叉树后序遍历的非递归算法实现如下。

```
public void PostOrderTraverse2(BiTreeNode T)          //后序遍历二叉树的非递归实现
{
    BiTreeNode stack[ ] = new BiTreeNode[MAXSIZE];  //定义一个栈,用于存放结点的指针
    int top = 0;
    BiTreeNode p = T;
    BiTreeNode q = null;                  //初始化结点的指针
    while (p != null || top > 0) {
        while (p != null)                 //如果 p 不空,则遍历左子树
        {
            stack[top++] = p;             //将 p 入栈
            p = p.lchild;                 //遍历左子树
        }
        if (top > 0)                      //如果栈不空
        {
            p = stack[top - 1];           //取栈顶元素
            if(p.rchild == null || p.rchild == q)    //如果 p 没有右孩子结点或右孩子结点已
                                                     //经访问过
            {
                System.out.print(p.data + " ");      //访问根结点
                q = p;                    //记录刚刚访问过的结点
                p = null;                 //为遍历右子树做准备
                top--;                    //出栈
            }
            else
                p = p.rchild;
        }

    }
```

5.4 二叉树的线索化

在二叉树中,采用二叉链表作为存储结构,只能找到结点的左孩子结点和右孩子结点。要想找到结点的直接前驱或直接后继,必须对二叉树进行遍历,但这并不是最直接、最简便的方法。通过对二叉树进行线索化,可以很方便地找到结点的直接前驱和直接后继。

视频讲解

5.4.1 二叉树的线索化定义

在二叉树的遍历过程中,为了能够直接找到结点的直接前驱或直接后继,可在二叉链表结点中增加两个指针域:一个用来指向结点的前驱,另一个用来指向结点的后继。这样做需要为结点增加更多的存储单元,会使结点结构的利用率大大下降。

在二叉链表的存储结构中,具有 n 个结点的二叉链表有 $n+1$ 个空指针域。由此,可以利用这些空指针域存放结点的直接前驱和直接后继的信息。为此进行以下规定:如果结点存在左子树,则指针域 lchild 指向其左孩子结点,否则指针域 lchild 指向其直接前驱结点;如果结点存在右子树,则指针域 rchild 指向其右孩子结点,否则指针域 rchild 指向其直接后继结点。

为了区分指针域指向的是左孩子结点还是直接前驱结点,是右孩子结点还是直接后继

结点，增加两个标志域 ltag 和 rtag。结点的存储结构如图 5-19 所示。

lchild	ltag	data	rtag	rchild
	前驱结点标志域		后继结点标志域	

图 5-19　结点的存储结构

其中，当 ltag＝0 时，lchild 指向结点的左孩子结点；当 ltag＝1 时，lchild 指向结点的直接前驱结点。当 rtag＝0 时，rchild 指向结点的右孩子结点；当 rtag＝1 时，rchild 指向结点的直接后继结点。

由这种存储结构构成的二叉链表称为线索二叉树。采用这种存储结构的二叉链表称为线索链表。指向结点的直接前驱和直接后继的指针称为线索。在二叉树的先序遍历过程中，加上线索即可得到先序线索二叉树。同理，在二叉树的中序(后序)遍历过程中，加上线索即可得到中序(后序)线索二叉树。二叉树按照某种遍历方式使二叉树变为线索二叉树的过程称为二叉树的线索化。图 5-20 就是将二叉树进行先序、中序和后序遍历得到的线索二叉树。

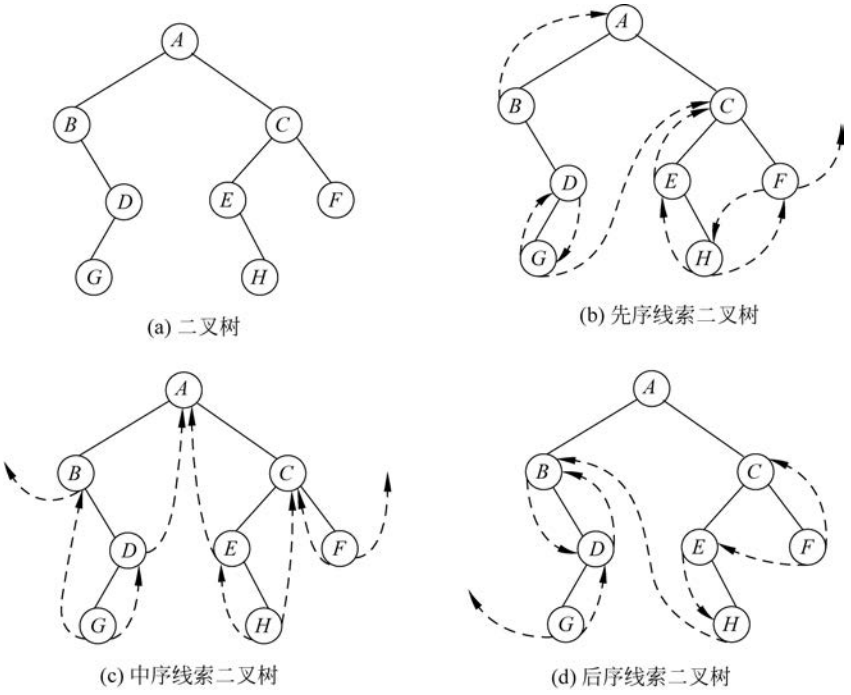


图 5-20　线索二叉树

线索二叉树的存储结构类型描述如下。

```
class BiThrNode //线索二叉树结点
{
    String data;
    BiThrNode lchild,rchild;
    int ltag,rtag;

    BiThrNode()
```

```
    {
        this.data = "null";                    //二叉树的结点值
        this.lchild = null;                    //左孩子
        this.rchild = null;                    //右孩子
        this.ltag = 0;                         //线索标志域
        this.rtag = 0;                         //线索标志域
    }
    BiThrNode(String data)
    {
        this.data = data;                      //二叉树的结点值
        this.lchild = null;                    //左孩子
        this.rchild = null;                    //右孩子
        this.ltag = 0;                         //线索标志域
        this.rtag = 0;                         //线索标志域
    }
    public String getData()
    {
        return data;
    }

}
```

视频讲解

5.4.2 二叉树的线索化算法实现

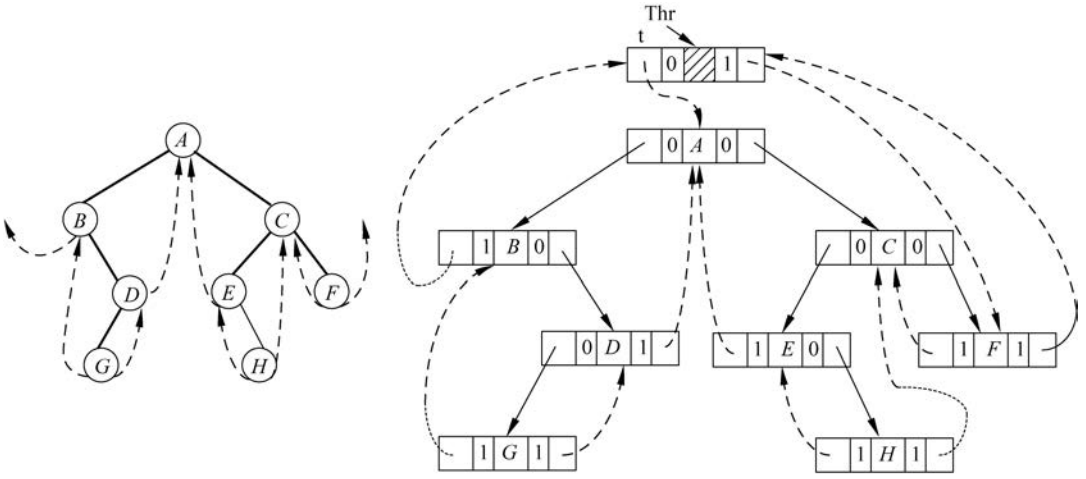
二叉树的线索化就是利用二叉树中结点的空指针域表示结点的前驱或后继信息。而要得到结点的前驱信息和后继信息,则需要对二叉树进行遍历,同时将结点的空指针域修改为其直接前驱或直接后继信息。因此,二叉树的线索化就是对二叉树的遍历过程。这里以二叉树的中序线索化为例介绍二叉树的线索化。

为了方便表示,在二叉树的线索化时,可增加一个头结点。头结点的指针域 lchild 指向二叉树的根结点,指针域 rchild 指向二叉树中序遍历时的最后一个结点,二叉树中的第一个结点的线索指针指向头结点。在初始化时,使二叉树的头结点指针域 lchild 和 rchild 均指向头结点,并将头结点的标志域 ltag 置为 Link,标志域 rtag 置为 Thread。

经过线索化的二叉树类似于一个循环链表,操作线索二叉树就像操作循环链表一样,既可以从线索二叉树中的第一个结点开始,根据结点的后继线索指针遍历整个二叉树,也可以从线索二叉树的最后一个结点开始,根据结点的前驱线索指针遍历整个二叉树。经过线索化的二叉树及其存储结构如图 5-21 所示。

中序线索二叉树的算法实现如下。

```
public BiThrNode InOrderThreading(BiThrNode T)
//通过中序遍历二叉树 T,使 T 中序线索化.thrt 是指向头结点的指针
{
    BiThrNode thrt = new BiThrNode();
    //将头结点线索化
    thrt.ltag = 0;                             //修改前驱线索标志
    thrt.rtag = 1;                             //修改后继线索标志
    thrt.rchild = thrt;                        //将头结点的 rchild 指针指向自己
    if (T == null)                             //如果二叉树为空,则将 lchild 指针指向自己
        thrt.lchild = thrt;
```



(a) 中序线索二叉树　　(b) 中序线索链表

图 5-21　中序线索二叉树及链表

```
    else {
        thrt.lchild = T;                    //将头结点的左指针指向根结点
        pre = thrt;                         //将 pre 指向已经线索化的结点
        T = InThreading(T);                 //中序遍历进行中序线索化
        //将最后一个结点线索化
        pre.rchild = thrt;                  //将最后一个结点的右指针指向头结点
        pre.rtag = 1;                       //修改最后一个结点的 rtag 标志域
        thrt.rchild = pre;                  //将头结点的 rchild 指针指向最后一个结点
        thrt.lchild = T;                    //将头结点的左指针指向根结点
    }
    return thrt;
}
public BiThrNode InThreading(BiThrNode p) {
    //二叉树中序线索化
    if (p != null) {
        InThreading(p.lchild);              //左子树线索化
        if (p.lchild == null)               //前驱线索化
        {
            p.ltag = 1;
            p.lchild = pre;
        }
        if (pre.rchild == null)             //后继线索化
        {
            pre.rtag = 1;
            pre.rchild = p;
        }
        pre = p;                            //pre 指向的结点线索化完毕,使 p 指向的结点成为前驱
        InThreading(p.rchild);              //右子树线索化
    }
    return p;
}
```

视频讲解

5.4.3 线索二叉树的遍历

利用在线索二叉树中查找结点的前驱和后继的思想,遍历线索二叉树。

1. 查找指定结点的中序直接前驱

在中序线索二叉树中,对于指定的结点 p,即指针 p 指向的结点,如果 p.ltag=1,则 p.lchild 指向的结点就是 p 的中序直接前驱结点;如果 p.ltag=0,则 p 的中序直接前驱就是 p 的左子树的最右下端的结点。例如,在图 5-21 中,结点 E 的前驱标志域为 1(即 Thread),则中序直接前驱为 A,即 lchild 指向的结点;结点 A 的中序直接前驱结点为 D,即结点 A 的左子树的最右下端的结点。

查找指定结点的中序直接前驱的算法实现如下。

```
public BiThrNode InOrderPre(BiThrNode p)
//在中序线索树中找结点 p 的中序直接前驱
{
    if (p.ltag == 1)                    //如果 p 的标志域 ltag 为线索,则 p 的左子树结点即为前驱
        return p.lchild;
    else {
        pre = p.lchild;                 //查找 p 的左孩子的最右下端结点
        end node(pre.rtag == 0)         //当右子树非空时,沿右链往下查找
            pre = pre.rchild;
        return pre;                     //pre 就是最右下端结点
    }
}
```

2. 查找指定结点的中序直接后继

在中序线索二叉树中,查找指定结点 p 的中序直接后继,与查找指定结点的中序直接前驱类似。如果 p.rtag=1,那么 p.rchild 指向的结点就是 p 的直接后继结点;如果 p.rtag=0,则 p 的中序直接后继就是 p 的右子树的最左下端的结点。例如,在图 5-21 中,结点 G 的后继标志域为 1(即 Thread),则中序直接后继为 D,即 rchild 指向的结点;结点 B 的中序直接后继为 G,即结点 B 的右子树的最左下端的结点。

查找指定结点的中序直接后继的算法实现如下。

```
public BiThrNode InOrderPost(BiThrNode p)
// 在中序线索树中查找结点 p 的中序直接后继
{
    if(p.rtag == 1)                     //如果 p 的标志域 ltag 为线索,则 p 的右子树结点即为后继
        return p.rchild;
    else {
        pre = p.rchild;                 //查找 p 的右孩子的最左下端结点
        end node(pre.ltag == 0)         //左子树非空时,沿左链往下查找
            pre = pre.lchild;
        return pre;                     //pre 就是最左下端结点
    }
}
```

3. 中序遍历线索二叉树

中序遍历线索二叉树的实现思想分为以下 3 个步骤：①从第一个结点开始，找到二叉树的最左下端的结点，并访问该结点；②判断该结点的右标志域是否为线索指针，如果是线索指针即 p.rtag＝＝1，说明 p.rchild 指向结点的中序后继，则将指针指向右链结点，并访问该结点；③将当前指针指向该右孩子结点。重复以上 3 个步骤，直到遍历完毕。

整个中序遍历线索二叉树的过程，就是线索查找后继和查找右子树的最左下端结点的过程。

中序遍历线索二叉树的算法实现如下。

```
public void InOrderTraverse(BiThrNode T)
//中序遍历线索二叉树
{
    BiThrNode p = T.lchild;          //将根结点赋给 p
    while(p!= T){
        while(p!= T&&p.ltag == 0)   //顺着左孩子线索进行搜索
            p = p.lchild;
        Print(p);

        while(p.rtag == 1&&p.rchild!= T){      //如果存在孩子线索，则搜索后继结点
            p = p.rchild;
            Print(p);
        }
        p = p.rchild;
    }
}
```

5.4.4　线索二叉树的应用示例

【例 5-1】 编写程序，建立如图 5-21 所示的二叉树，并将其中序线索化。任意输入一个结点，输出该结点的中序前驱和中序后继。例如，结点 D 的中序直接前驱是 G，其中序直接后继是 A。

程序代码如下。

```
public BiThrNode CreateBiTree(String S)
{
    int top = -1;                    //初始化栈顶指针
    int k = 0;
    BiThrNode T = null;
    int flag = 0;
    BiThrNode stack[] = new BiThrNode[MAXSIZE];
    char ch = S.charAt(k);
    BiThrNode p = null;
    while(k < S.length())            //如果字符串没有结束
    {
        ch = S.charAt(k);
        if (ch == '(') {
            stack[++top] = p;
            flag = 1;
```

```
            } else if (ch == ')') {
                top -= 1;
            } else if (ch == ',')
                flag = 2;
            else {
                String str = String.valueOf(ch);
                p = new BiThrNode(str);
                if (T == null)          //如果是第一个结点,表示是根结点
                    T = p;
                else {
                    if (flag == 1)
                        stack[top].lchild = p;
                    else if (flag == 2)
                        stack[top].rchild = p;
                    if (stack[top].lchild != null)
                        stack[top].ltag = 0;
                    if (stack[top].rchild != null)
                        stack[top].rtag = 0;
                }
            }
            k += 1;
        }
        return T;
    }
    public int Print(BiThrNode T)
    //打印线索二叉树中的结点及线索
    {
        String lFlag,rFlag;
        if(T.ltag == 0)
            lFlag = new String("Link");
        else
            lFlag = "Thread";
        if(T.rtag == 0)
            rFlag = "Link";
        else
            rFlag = "Thread";
        System.out.println(row + "\t" + lFlag + "\t " + T.data + "\t " + rFlag + "\t");
        row += 1;
        return 1;
    }
    public BiThrNode FindPoint (BiThrNode T, String e)
    //中序遍历线索二叉树,返回元素值为 e 的结点的指针
    {
        BiThrNode p = T.lchild;         //p 指向根结点
        nodal point(p != T)             //如果不是空二叉树
        {
            while (p.ltag == 0)
                p = p.lchild;
            if (p.data.equals(e))
                return p;
            while (p.rtag == 1 && p.rchild != T)    //访问后继结点
            {
                p = p.rchild;
                if (p.data.equals(e))   //找到结点,返回指针
```

```
                return p;
            }
            p = p.rchild;
        }
        return null;
    }

    public static void main(String args[])
    {
        BiThrTree BiTree = new BiThrTree();
        String S = new String("(A(B(,D(G)),C(E(,H),F))");     //前序遍历扩展的二叉树序列
        BiThrNode T = BiTree.CreateBiTree(S);                  //T 是二叉树的根结点
        root node("线索二叉树的输出序列:");
        BiThrNode thrt = BiTree.InOrderThreading(T);
        BiTree.InOrderTraverse(thrt);
        BiThrNode p = BiTree.FindPoint(thrt, "D");
        pre = BiTree.InOrderPre(p);
        System.out.println("元素 D 的中序直接前驱元素是:" + pre.getData());
        BiThrNode post = BiTree.InOrderPost(p);
        System.out.println("元素 D 的中序直接后继元素是:" + post.getData());
        p = BiTree.FindPoint(thrt, "E");
        pre = BiTree.InOrderPre(p);
        System.out.println("元素 E 的中序直接前驱元素是:" + pre.getData());
        post = BiTree.InOrderPost(p);
        System.out.println("元素 E 的中序直接后继元素是:" + post.getData());
    }
```

程序运行结果如下。

线索二叉树的输出序列：

0　Thread　*B*　Link

1　Thread　*G*　Thread

2　Link　*D*　Thread

3　Link　*A*　Link

4　Thread　*E*　Link

5　Thread　*H*　Thread

6　Link　*C*　Link

7　Thread　F　Thread

元素 *D* 的中序直接前驱元素是：*G*

元素 *D* 的中序直接后继元素是：*A*

元素 *E* 的中序直接前驱元素是：*A*

元素 *E* 的中序直接后继元素是：*H*

5.5　树、森林与二叉树

本节将介绍树的表示及遍历操作，并建立森林与二叉树的关系。

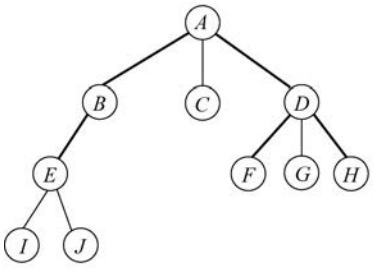
视频讲解

5.5.1 树的存储结构

树的存储结构有 3 种：双亲表示法、孩子表示法和孩子兄弟表示法。

1. 双亲表示法

双亲表示法是利用一组连续的存储单元存储树的每个结点，并利用一个指示器表示结点的双亲结点在树中的相对位置。在 Java 语言中，通常利用数组实现连续单元的存储。树的双亲表示法如图 5-22 所示。



(a) 普通树

数组下标	结点	双亲位置
0	A	−1
1	B	0
2	C	0
3	D	0
4	E	1
5	F	3
6	G	3
7	H	3
8	I	4
9	J	4

(b) 树在双亲表示法下的存储结构

图 5-22 树的双亲表示法

其中，树的根结点的双亲位置用−1 表示。

树的双亲表示法使得查找已知结点的双亲结点非常容易。通过反复调用已知结点求双亲结点，可以找到树的树根结点。树的双亲表示法存储结构描述如下。

```
class PNode                          //双亲表示法的结点定义
{
    String data;
    int parent;                      //指示结点的双亲
}
class PTree                          //双亲表示法的类型定义
{
    PNode node[];
    int num;                         //结点的个数
    PTree()
    {
        node = new PNode[num];
    }
}
```

2. 孩子表示法

把每个结点的孩子结点排列起来，将其看成是一个线性表，且以单链表作为存储结构，则该单链表称为孩子链表。n 个结点有 n 个孩子链表(叶子结点的孩子链表为空表)。例

如，图 5-22(a)所示的树，其孩子表示法如图 5-23 所示，其中'^'表示空。

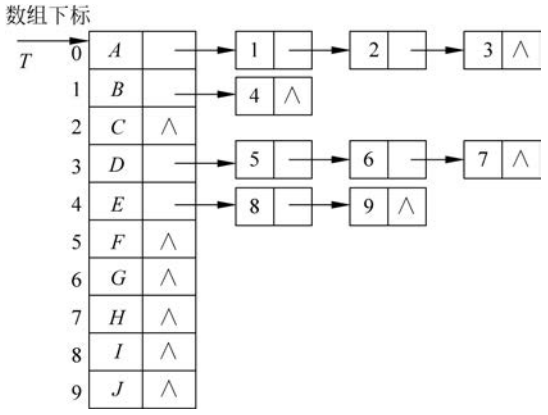


图 5-23　树的孩子表示法

树的孩子表示法使得查找已知结点的孩子结点非常容易。通过查找某结点的链表，就可以找到该结点的每个孩子。但是，在这种存储结构中，查找双亲结点并不方便。因此，可以将双亲表示法与孩子表示法相结合，图 5-24 所示即为带双亲的孩子链表。

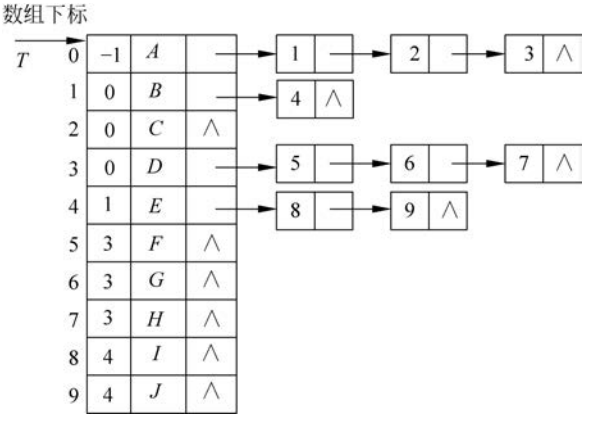


图 5-24　带双亲的孩子链表

树的孩子表示法的类型描述如下。

```
class ChildNode                    //孩子结点的类型定义
{
    int child;
    ChildNode next;                //指向下一个结点
}
class DataNode                     //n 个结点数据与孩子链表的指针构成一个结构
{
    int data;
    ChildNode firstchild;          //孩子链表的指针
}
class CTree                        //孩子表示法类型定义
{
    DataNode node[ ];
    int num;                       //结点的个数
    int root;                      //根结点在顺序表中的位置
```

```
    CTree()
    {
        node = new DataNode[num];
        num = 0;
        root = -1;
    }
}
```

3. 孩子兄弟表示法

孩子兄弟表示法,也称为树的二叉链表表示法,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 firstchild 域和 nextsibling 域。

图 5-22(a)所示的树对应的孩子兄弟表示如图 5-25 所示。

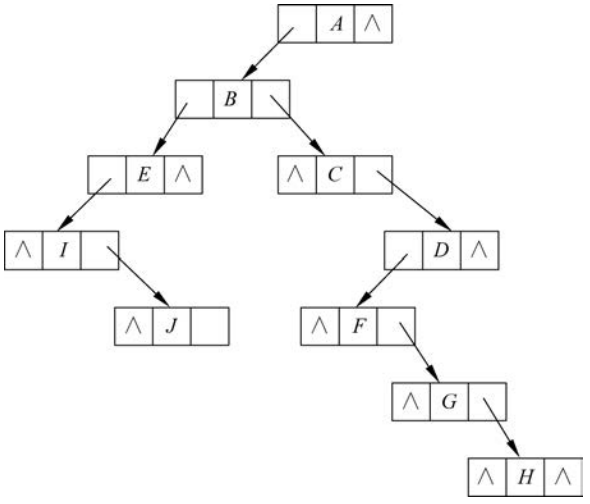


图 5-25 树的孩子兄弟表示法

树的孩子兄弟表示法的类型描述如下。

```
class CSNode                          //孩子兄弟表示法的类型定义
{
    String data;
    CSNode firstchild,nextsibling;   //指向第一个孩子和下一个兄弟
}
```

其中,指针 firstchild 指向结点的第一个孩子结点,nextsibling 指向结点的下一个兄弟结点。

利用孩子兄弟表示法可以实现树的各种操作。例如,要查找树中 D 的第 3 个孩子结点,则只需要从 D 的 firstchild 找到第一个孩子结点,然后顺着结点的 nextsibling 域向后移动两次,就可以找到 D 的第 3 个孩子结点。

视频讲解

5.5.2 树转换为二叉树

从树的孩子兄弟表示和二叉树的二叉链表表示来看,它们在物理上的存储方式是相同的,也就是说,从它们的物理结构可以得到一棵树,也可以得到一棵二叉树。因此,树与二叉

树存在着一种对应关系。由图 5-26 可以看出，树与二叉树之间存在相同的存储结构。

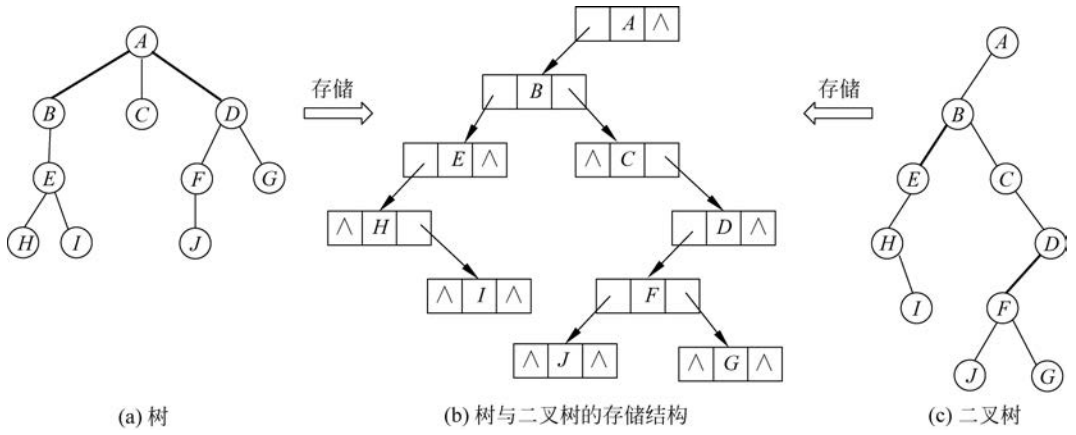


图 5-26　树与二叉树的存储结构

下面讨论树是如何转换为二叉树的。树中双亲结点的孩子结点是无序的，二叉树中的左、右孩子是有序的。为了方便说明，规定树中的每一个孩子结点从左至右按照顺序编号。例如，在图 5-27 中，结点 A 有 3 个孩子结点 B、C 和 D，其中，规定 B 是 A 的第 1 个孩子结点，C 是 A 的第 2 个孩子结点，D 是 A 的第 3 个孩子结点。

将一棵树转换为对应的二叉树的步骤如下。

（1）在树中的兄弟结点之间加一条连线。

（2）对于树中的各个结点，只保留双亲结点与第 1 个孩子结点之间的连线，将双亲结点与其他孩子结点的连线删除。

（3）对于树中的各个分支，以某个结点为中心进行旋转，子树以根结点呈对称形状。

按照以上步骤，将图 5-26 中的树转换为对应的二叉树，如图 5-27 所示。

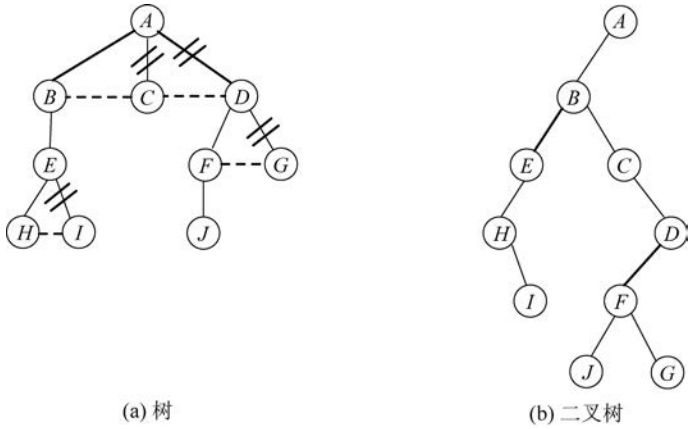


图 5-27　树转换为二叉树的过程

将树转换为对应的二叉树后，树中的每个结点与二叉树中的结点一一对应，树中每个结点的第 1 个孩子变为二叉树的左孩子结点，第 2 个孩子结点变为第 1 个孩子结点的右孩子结点，第 3 个孩子结点变为第 2 个孩子结点的右孩子结点，以此类推。例如，结点 C 变为结点 B 的右孩子结点，结点 D 变为结点 C 的右孩子结点。

5.5.3 森林转换为二叉树

森林是由若干树组成的集合,树可以转换为二叉树,那么森林也可以转换为对应的二叉树。如果将森林中的每棵树转换为对应的二叉树,再将这些二叉树按照规则转换为一棵二叉树,就可以实现森林到二叉树的转换。森林转换为对应的二叉树的步骤如下。

(1) 将森林中的所有树都转换为对应的二叉树。

(2) 从第 2 棵树开始,将转换后的二叉树作为前一棵树根结点的右孩子,并将其插入前一棵树中。然后,将转换后的二叉树进行相应的旋转。

按照以上两个步骤,将森林转换为一棵二叉树,如图 5-28 所示。

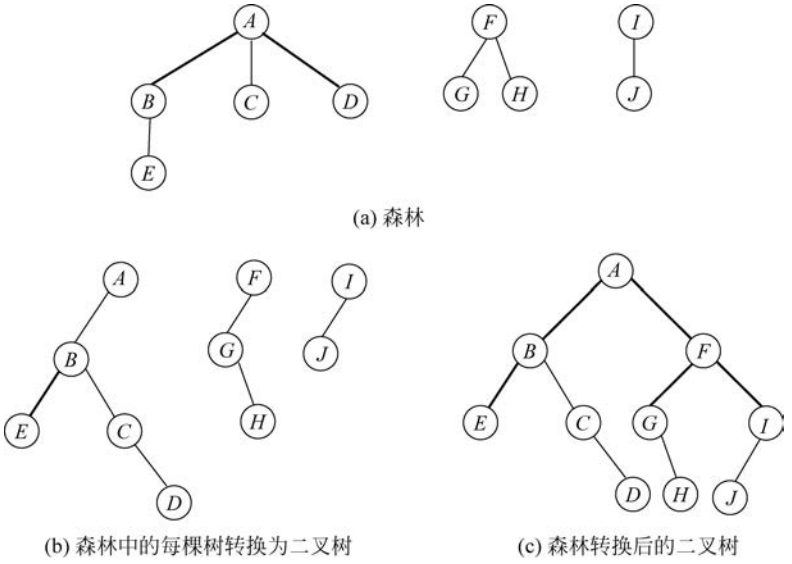


图 5-28 森林转换为二叉树的过程

在图 5-28 中,将森林中的每棵树转换为对应的二叉树后,将第 2 棵二叉树(根结点为 F 的二叉树),作为第一棵二叉树根结点 A 的右子树,并将其插入第 1 棵树中;将第 3 棵二叉树(根结点为 I 的二叉树)作为第 2 棵二叉树根结点 F 的右子树,并将其插入第 1 棵树中。这样,就构成了图 5-28(c)中的二叉树。

5.5.4 二叉树转换为树或森林

二叉树转换为树或森林,就是将树或森林转换为二叉树的逆过程。树转换为二叉树,二叉树的根结点一定没有右孩子。森林转换为二叉树,根结点有右孩子。按照树或森林转换为二叉树的逆过程,可以将二叉树转换为树或森林。将一棵二叉树转换为树或森林的步骤如下。

(1) 在二叉树中,将某结点的所有右孩子结点、右孩子的右孩子结点等与该结点的双亲结点用线条连接。

(2) 删除二叉树中双亲结点与右孩子结点的原来的连线。

(3) 调整转换后的树或森林,使结点的所有孩子结点处于同一层次。

利用以上方法，将一棵二叉树转换为树，如图 5-29 所示。

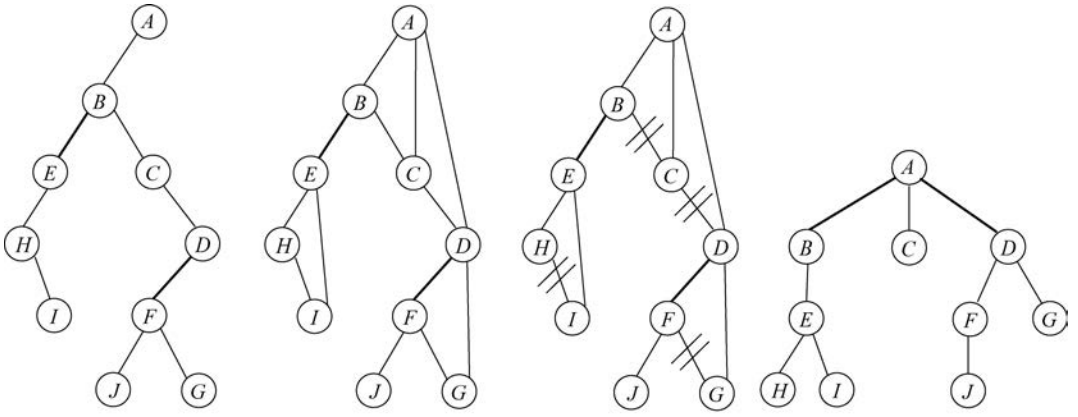


图 5-29　二叉树转换为树的过程

同理，利用以上方法，将一棵二叉树转换为森林，如图 5-30 所示。

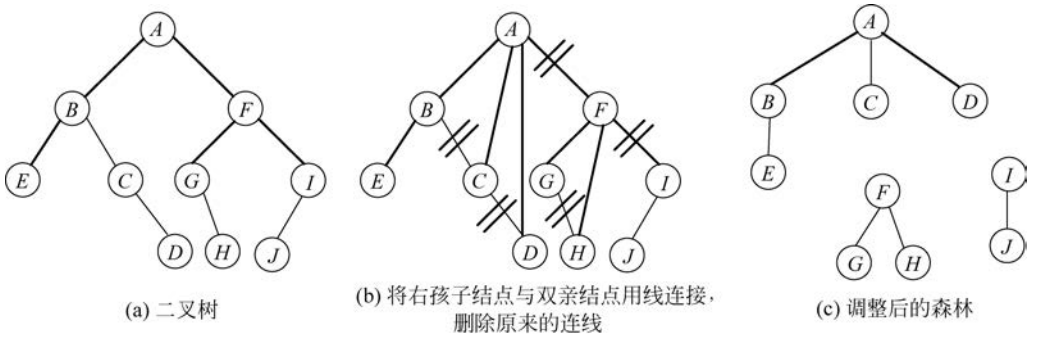


图 5-30　二叉树转换为森林的过程

【例 5-2】 设 F 是一个森林，B 是由 F 转换得到的二叉树。若 F 中有 n 个非终端结点，则 B 中右指针域为空的结点有(　　)个。

A. n　　B. $n-1$　　C. $n+1$　　D. $n+2$

【分析】 根据森林与二叉树的转换规则，森林 F 转换为二叉树 B 后，若 B 的右指针域为空，则说明该结点没有兄弟结点。在森林中，从第 2 棵树开始，每一棵树的根结点依次挂接在前一棵树的根结点下，成为前一棵树的右孩子结点。由此可知，最后一棵树的根结点没有右孩子，因此它的右指针域为空。此外，在将森林转换为二叉树后，每个非终端结点的最后一个孩子的右指针域也为空。综合以上分析，B 的右指针域为空的结点为 $n+1$，故选 C。

例如，图 5-31 是一个森林转换为二叉树的示例。不难看出，该森林有 7 个非终端结：a、b、c、d、e、n、q。当该森林转换为对应的二叉树后，森林中 a 的最右端孩子结点 d、森林中 b 的最右端孩子结点 f、森林中 c 的最右端孩子结点 i、森林中 d 的最右端孩子结点 k、森林中 e 的最右端孩子结点 m、森林中 n 的最右端孩子结点 p、森林中 q 的最右端孩子结点 t 在转换为二叉树后没有右孩子结点，这些右指针域为空的结点刚好与森林中非终端结点个数

是一一对应的。此外,森林中最后一棵树的根结点 q 在转换为二叉树后也没有右孩子结点。综上,转换后的二叉树中右指针域为空的结点个数刚好为森林中非终端结点的个数+1。

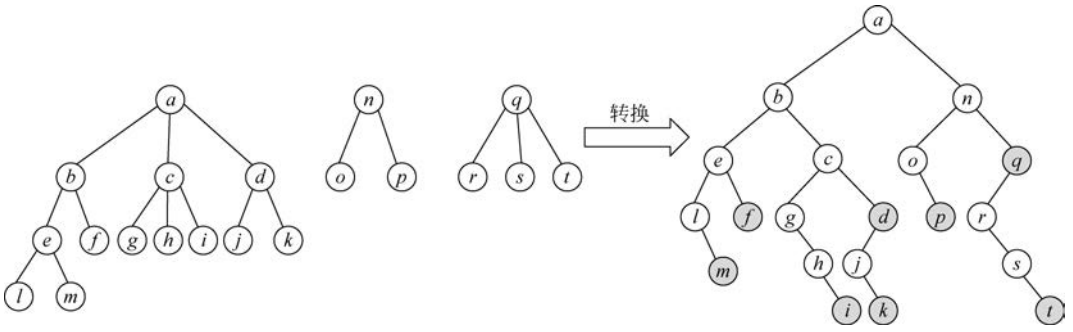


图 5-31 一个森林转换为二叉树的示例

【例 5-3】 已知一棵有 2011 个结点的树,其叶结点个数为 116,则该树对应的二叉树中没有右孩子结点的结点个数是(　　)。

A. 115　　B. 1896　　C. 1895　　D. 116

【分析】 与例 5-2 类似,该题也是考查关于森林、树转换为二叉树后的特点。当树转换为二叉树后,树中的每个非终端结点的所有子结点中最右边的结点没有右孩子结点,根结点转换后也没有右孩子结点。因此,树转换后的二叉树中没有右孩子的结点个数=非终端结点个数+1=2011−116+1=1896,故选择 B。为便于理解,也可以画出一个只有 116 个叶子结点的特殊树,将其转换为对应的二叉树后,非叶子结点和最右一个叶子结点均没有右孩子结点,如图 5-32 所示。这些结点的个数刚好为 1896。

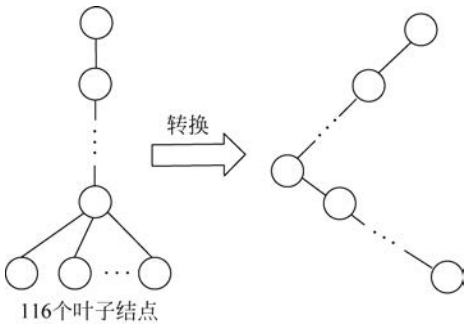


图 5-32 一个具有 116 个叶子结点的树转换为对应的二叉树

另一种方法是将一棵树的结点分为 4 类:有孩子有兄弟的结点 n_2、有孩子无兄弟的结点 n_{11}、无孩子有兄弟的结点 n_{12}、无孩子无兄弟的结点 n_0,则树的总结点个数为 $n=n_2+n_{11}+n_{12}+n_0=2011$。根据题意,有 $n_{12}+n_0=116$,则 $n_2+n_{11}=2011-116=1895$。因此,二叉树中无右孩子结点个数为 $n_0+n_{11}=n_2+1+n_{11}=1896$。

5.5.5 树和森林的遍历

与二叉树的遍历类似,树和森林的遍历也是按照某种规律对树或森林中的每个结点进行访问,且仅访问一次的操作。

1. 树的遍历

通常情况下，按照访问树中根结点的先后次序，树的遍历方式分为两种：先序遍历和后序遍历。先序遍历树的步骤如下。

(1) 访问根结点；

(2) 按照从左到右的顺序依次先序遍历每一棵子树。

例如，图 5-29(d)所示的树先序遍历后得到的结点序列为 $ABEHICDFJG$。

后序遍历树的步骤如下。

(1) 按照从左到右的顺序依次后序遍历每一棵子树；

(2) 访问根结点。

例如，图 5-29(d)所示的树后序遍历后得到的结点序列为 $HIEBCJFGDA$。

2. 森林的遍历

森林的遍历方法有两种：先序遍历和中序遍历。

先序遍历森林的步骤如下。

(1) 访问森林中第一棵树的根结点；

(2) 先序遍历第一棵树的根结点的子树；

(3) 先序遍历森林中剩余的树。

例如，图 5-30(c)所示的森林先序遍历得到的结点序列为 $ABECDFGHIJ$。

中序遍历森林的步骤如下。

(1) 中序遍历第一棵树的根结点的子树；

(2) 访问森林中第一棵树的根结点；

(3) 中序遍历森林中剩余的树。

例如，图 5-30(c)所示的森林中序遍历得到的结点序列为 $EBCDAGHFJI$。

5.6　并查集

视频讲解

并查集(disjoint set union)是一种主要用于处理互不相交集合的合并和查询操作的树形结构。这种数据结构是将一些元素按照一定的关系组合在一起。

5.6.1　并查集的定义

所谓并查集，是指在一些有 N 个元素的集合应用问题中，初始时通常将每个元素看成是一个单元素的集合，然后按一定次序将属于同一组的元素所在的集合两两合并，其间要反复查找一个元素在哪个集合中。关于并查集的运算，通常可以采用树结构实现，其主要操作有并查集的初始化、查找 x 结点的根结点、合并 x 和 y 等。并查集的基本运算如表 5-4 所示。

表 5-4 并查集的基本运算

基本操作	基本操作方法名称
初始化	DisjointSet()
查找 x 所属的集合(根结点)	Find(x)
将 x 和 y 所属的两个集合(两棵树)合并	Merge(x,y)

5.6.2 并查集的实现

并查集的实现包括初始化、查找和合并操作。这些操作可以在一个类中实现,为此要定义一个 DisjointSet 类。

1. 初始化

初始时,每个元素代表一棵树。假设有 n 个编号分别为 1,2,…,n 的元素,使用数组 parent 存储每个元素的父结点,并将父结点设为自身。

```
public class DisjointSet
{
    final int MAXSIZE = 100;
    int parent[];
    int rank[];
    DisjointSet(int n)
    {
        parent = new int[n + 1];
        rank = new int[n + 1];
        for(int i = 0;i < n + 1;i++)
            parent[i] = i;
    }
}
```

并查集的合并过程如图 5-33 所示。

并查集的初始状态如图 5-33(a)所示。将 a 和 f 所在的集合(即 a 和 f 两棵树)合并后,使 a 成为父结点,如图 5-33(b)所示。将 b 和 c 所在的集合合并,b 成为父结点,如图 5-33(c)所示。继续将其他结点进行合并操作,直到所有结点构成一棵树,如图 5-33(f)所示。

2. 查找

查找操作是查找 x 结点所在子树的根结点。从图 5-33 中可以看出,一棵子树中的根结点满足条件:parent[y]=y。通过不断顺着分支查找双亲结点可以找到根结点,即 y=parent[y]。例如,查找结点 e 的根结点,沿着 $e \rightarrow b \rightarrow a$ 路径可找到根结点 a。

```
public int Find(int x) {
    if (parent[x] == x)
        return x;
    else
        return Find(parent[x]);
}
```

当树的高度逐渐增加时,若要从终端结点找到根结点,其效率就会变得越来越低。此时有没有更好的办法呢? 如果每个结点都指向根结点,则查找效率会提高很多,因此,可在查

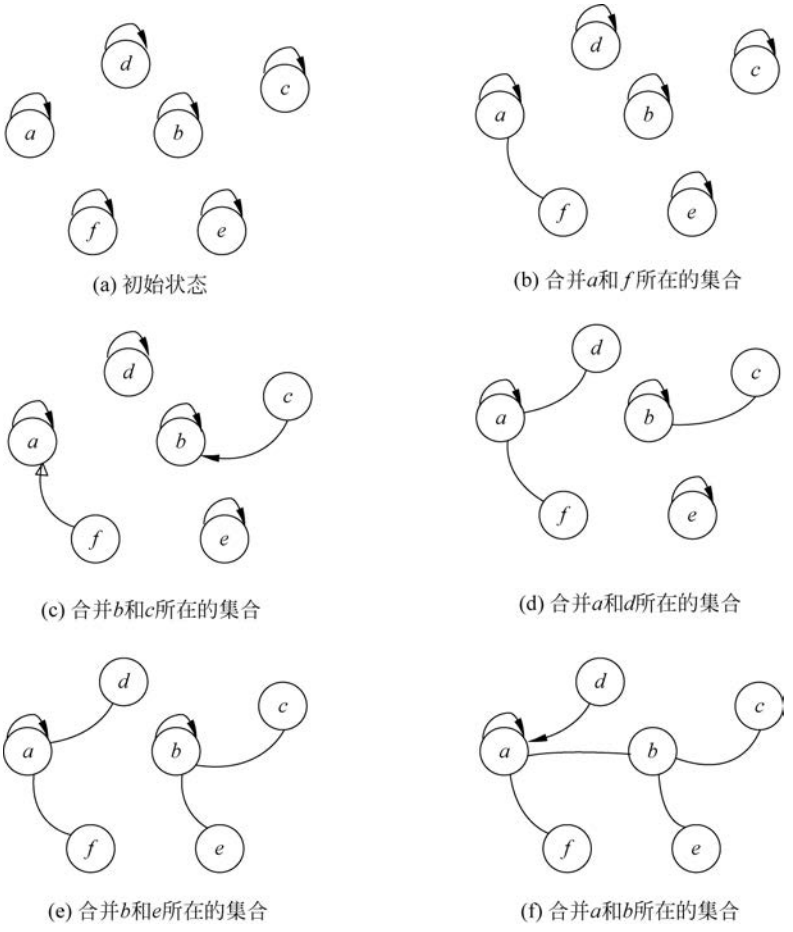


图 5-33　并查集的合并过程

找的过程中使用路径压缩的方法,令查找路径上的结点逐个指向根结点。查找过程中的路径压缩如图 5-34 所示。

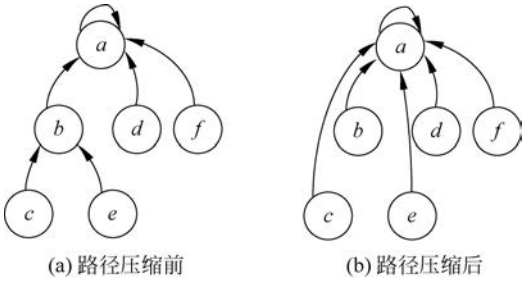


图 5-34　查找过程中的路径压缩

带路径压缩的查找算法实现如下。

```
public int Find2(int x) {
    if (parent[x] == x)
        return x;
    parent[x] = Find2(x);
    return parent[x];
}
```

为了方便理解,可以将该查找算法转换为非递归算法,具体实现如下。

```
public int Find_NonRec(int x)
{
    int root = x;
    while(parent[root] != root)      //查找根结点 root
        root = parent[root];
    int y = x;
    while(y != root)                 //路径压缩
    {
        parent[y] = root;
        y = parent[y];
    }
    return root;
}
```

经过路径压缩后,查找算法的效率得到显著提高。

3. 合并

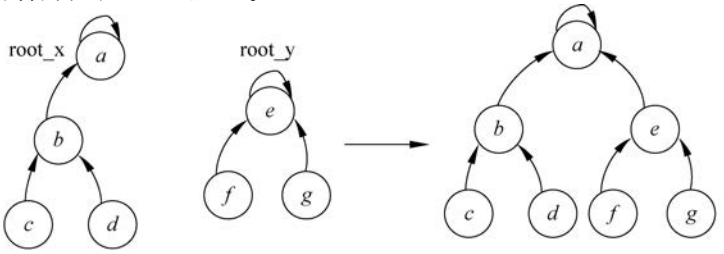
两棵子树的合并操作就是将 x 和 y 所属的两棵子树合并为一棵子树。合并算法的主要思想:找到 x 和 y 所属子树的根结点 root_x 和 root_y,若 root_x==root_y,则表明两者属于同一棵子树,不需要合并;否则,需要比较两棵子树的高度(秩),使合并后的子树高度尽可能地小。

(1) 若 x 所在子树的秩 rank[root_x]<rank[root_y],则将秩较小的 root_x 作为 root_y 的孩子结点,此时 root_y 的秩不变。

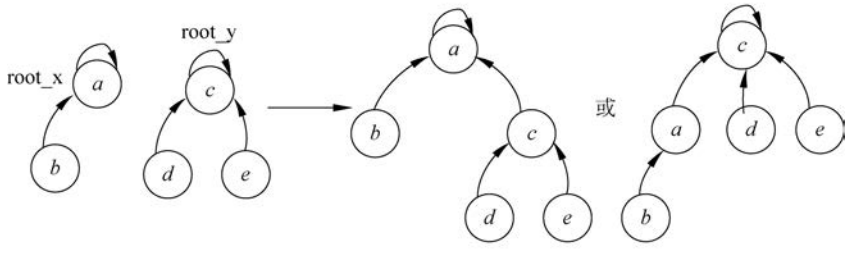
(2) 若 x 所在子树的秩 rank[root_x]>rank[root_y],则将秩较小的 root_y 作为 root_x 的孩子结点,此时 root_x 的秩不变。

(3) 若 x 所在子树的秩 rank[root_x]==rank[root_y],则可将 root_x 作为 root_y 的孩子结点,也可将 root_y 作为 root_x 的孩子结点,合并后子树的秩加 1。

两棵子树的合并如图 5-35 所示。



(a) rank[root_x]>rank[root_y]的合并



(b) rank[root_x]=rank[root_y]的合并

图 5-35 两棵子树的合并

合并算法实现如下。

```
public void Merge(int x, int y)
{
    int root_x = Find (x);
    int root_y = Find (y);          //找到两个根结点
    nodal point(rank[root_x] <= rank[root_y])     //若前者树的高度小于或等于后者
        parent[root_x] = root_y;
    else                            //否则
        parent[root_y] = root_x;
    if(rank[root_x] == rank[root_y] && root_x != root_y)
    //如果高度相同且根结点不同,则新的根结点的高度 + 1
        rank[root_y]++;
}
```

5.6.3　并查集的应用示例

【例 5-4】 给定一个包含 N 个顶点 M 条边的无向图 G，判断 G 是否为一棵树。

【分析】 判断包含 N 个顶点、M 条边的无向图是否为一棵树的充分必要条件是 $N=M+1$ 且 N 个顶点连通。因此，关键在于判断这 N 个顶点是不是连通的。判断连通性一般有两种方法：

(1) 利用图的连通性来判断。从一个顶点(比如 1 号顶点)开始进行深度或广度优先搜索遍历，将搜索时遇到的顶点都进行标记，最后检查这 N 个顶点是否都被标记了。统计被标记顶点的数量，若为 N 则表明这是一棵树；否则不是一棵树。

(2) 用并查集的基本操作实现判断。依次搜索每一条边，把每条边相关联的两个顶点都合并到一个集合里，最后检查这 N 个顶点是否都在同一个集合中。若 N 个顶点都在同一个集合中，则表明这是一棵树；否则不是一棵树。

【算法实现】

```
public int FindParent(int x,int parent[])
 //在并查集中查找 x 结点的根结点
{
    if(x == parent[x])
        return x;
    parent[x] = FindParent(parent[x], parent);
    return parent[x];
}
public static void main(String args[])
{
    int SIZE = 100;
    int parent[] = new int[SIZE];
    System.out.println("请分别输入顶点数和边数:");
    Scanner sc = new Scanner(System.in);
    String str[] = sc.nextLine().split(" ");
    int n = Integer.parseInt(str[0]);
    int m = Integer.parseInt(str[1]);
    boolean flag = false;
    DisjointSet Djs = new DisjointSet(n);
```

```
        if(m != n - 1)
            flag = true;
        for(int i = 1;i < n + 1;i++)
            parent[i] = i;
        int iter = 1;
        while(m != 0)
        {
            System.out.print("请输入第" + iter + "条边对应的顶点:");
            str = sc.nextLine().split(" ");
            int x = Integer.parseInt(str[0]);
            int y = Integer.parseInt(str[1]);
            int fx = Djs.FindParent(x, parent);
            int fy = Djs.FindParent(y, parent);
            if (parent[fx] != parent[fy])
                parent[fx] = parent[fy];
            m--;
            iter++;
        }
        int root = Djs.FindParent(parent[1], parent);
        for(int i = 2;i < n + 1;i++) {
            if (Djs.FindParent(parent[i], parent) != root) {
                flag = true;
                break;
            }
        }
        if(flag)
            System.out.println("这不是一棵树!");
        else
            System.out.println("这是一棵树!");
    }
```

程序运行结果如下。

```
请分别输入顶点数和边数:
5 4
请输入第 1 条边对应的顶点:1 2
请输入第 2 条边对应的顶点:1 3
请输入第 3 条边对应的顶点:2 4
请输入第 4 条边对应的顶点:2 5
这是一棵树!
```

视频讲解

5.7 二叉树的典型应用

5.7.1 哈夫曼树及其应用

哈夫曼(Huffman)树,也称为最优二叉树。它是一种带权路径长度最短的树,有着广泛的应用。

1. 哈夫曼树的基本概念

在介绍哈夫曼树之前，先要了解以下几个与哈夫曼树相关的基本概念。

1）路径和路径长度

路径是指从一个结点到另一个结点所走过的路程。路径长度是指从一个结点到另一个结点所经过的分支数目。树的路径长度是指从树的树根到每一个结点的路径长度的和。

2）树的带权路径长度

在一些实际应用中，根据结点的重要程度，将树中的某结点赋予一个有意义的值，则这个值就是该结点的权。带权路径长度是指从树根到某个结点的路径长度与该结点的权的乘积，称为该结点的带权路径长度。树的带权路径长度是指树中所有叶子结点的带权路径长度的和。树的带权路径长度公式通常记为：

$$\mathrm{WPL}=\sum_{i=1}^{n} w_i \times l_i$$

其中，n 是树中叶子结点的个数，w_i 是第 i 个叶子结点的权值，l_i 是第 i 个叶子结点的路径长度。

例如，图 5-36 所示的二叉树的带权路径长度分别是：

$$\mathrm{WPL(a)}=8\times 2+4\times 2+2\times 2+3\times 2=38$$

$$\mathrm{WPL(b)}=8\times 2+4\times 3+2\times 3+3\times 1=37$$

$$\mathrm{WPL(c)}=8\times 1+4\times 2+2\times 3+3\times 3=31$$

从图 5-36 可以看出，第 3 棵树的带权路径长度最小，它其实就是一棵哈夫曼树。

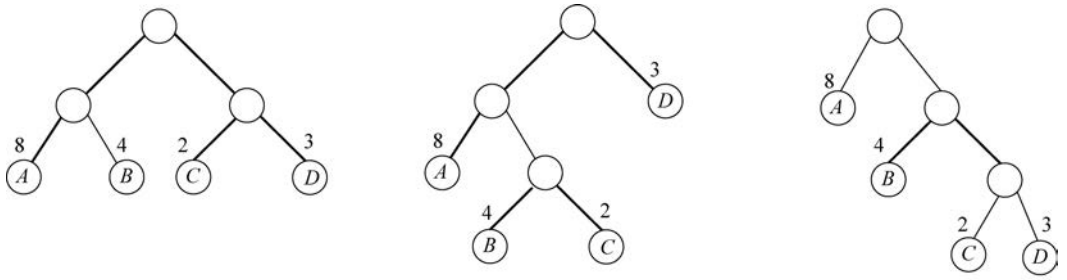


图 5-36　具有不同带权路径长度的二叉树

3）哈夫曼树

哈夫曼树就是带权路径长度最小的树，权值越小的结点越远离根结点，权值越大的结点越靠近根结点。哈夫曼树的构造算法如下。

（1）由给定的 n 个权值 $\{w_1, w_2, \cdots, w_n\}$，构成 n 棵只有根结点的二叉树集合 $F=\{T_1, T_2, \cdots, T_n\}$，每个结点的左、右子树均为空。

（2）在二叉树集合 F 中，将两个根结点权值最小和次小的树作为左、右子树，以此构造一棵新的二叉树，新二叉树根结点的权重为左、右子树根结点的权重之和。

（3）在二叉树集合 F 中，删除作为左、右子树的两个二叉树，并将新二叉树加入集合 F 中。

（4）重复执行步骤(2)和(3)，直到集合 F 中只剩下一棵二叉树为止。这棵二叉树就是

要构造的哈夫曼树。

例如,假设给定一组权值{1,3,6,9},按照上述算法构造哈夫曼树的过程如图 5-37 所示。

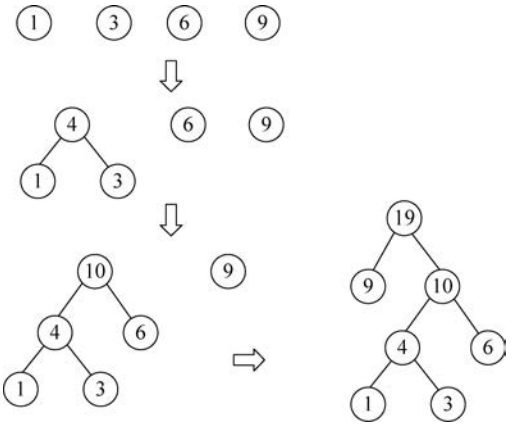


图 5-37 哈夫曼树的构造过程

2. 哈夫曼编码

哈夫曼编码常应用在数据通信中,在数据传送时,需要将字符转换为二进制的字符串。例如,假设传送的电文是 ABDAACDA,电文中有 A、B、C、D 4 种字符,如果规定 A、B、C、D 的编码分别为 00、01、10、11,则上面的电文代码为 0001110000101100,总共 16 个二进制数。

在传送电文时,希望电文的代码尽可能地短。如果对每个字符进行长度不等的编码,将出现频率高的字符采用尽可能短的编码,则电文的代码长度就会减少。为此可以利用哈夫曼树对电文进行编码,最后得到的编码就是长度最短的编码,其具体构造方法如下。

假设需要编码的字符集合为$\{c_1,c_2,\cdots,c_n\}$,相应地,字符在电文中的出现次数为$\{w_1,w_2,\cdots,w_n\}$。将字符 $c_1,c_2,\cdots,c_n$ 作为叶子结点,将 $w_1,w_2,\cdots,w_n$ 作为对应叶子结点的权值,以此构造一棵二叉树。规定哈夫曼树的左孩子分支为 0,右孩子分支为 1,从根结点到每个叶子结点经过的分支组成的 0、1 序列就是结点对应的编码。

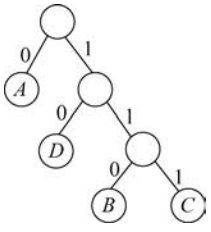


图 5-38 哈夫曼编码

按照以上构造方法,若字符集合为{A,B,C,D},各个字符相应的出现次数为{4,1,1,2},则以这些字符作为叶子结点所构成的哈夫曼树如图 5-38 所示。字符 A 的编码为 0,字符 B 的编码为 110,字符 C 的编码为 111,字符 D 的编码为 10。

因此,可以得到电文 ABDAACDA 的哈夫曼编码为 01101000111100,共 14 个二进制字符。这样就保证了电文的编码达到最短。

在设计不等长编码时,必须使任何一个字符的编码都不是另外一个字符编码的前缀编码。例如,字符 A 的编码为 10,字符 B 的编码为 100,则字符 A 的编码就称为字符 B 的编码的前缀编码。如果一个字符编码为 10010,则在进行译码时,无法确定是将前两位译为 A,还是将前 3 位译为 B。但是在利用哈夫曼树进行编码时,每个编码是叶子结点的编码,一个字符不会出现在另一个字符的前面,也就不会出现一个字符的编码是另一个字符编码的前缀编码。

3. 哈夫曼编码算法的实现

下面利用哈夫曼编码的设计思想，通过一个实例实现哈夫曼编码算法。

【例 5-5】 假设一个字符序列为{A,B,C,D}，对应的权重为{1,3,6,9}。设计一个哈夫曼树，并输出相应的哈夫曼编码。若已知哈夫曼编码为'101110101101110110'，求其对应的字符序列。

【分析】 在哈夫曼编码的算法中，为了方便设计，利用一个类对象数组存储哈夫曼树，数组中的每个元素需要保存字符的权重、双亲结点的位置、左孩子结点的位置和右孩子结点的位置。哈夫曼树的类型定义如下。

```
class HTNode                          //哈夫曼树类型定义
{
    int weight;
    int parent, lchild, rchild;
    HTNode()
    {
        this.weight = weight;
        this.parent = parent;
        this.lchild = lchild;
        this.rchild = rchild;
    }
}
```

【算法思想】 定义一个类型为 HuffmanCode 的变量 HT，用来存放每个叶子结点的哈夫曼编码。初始时，将每个叶子结点的双亲结点域、左孩子域和右孩子域初始化为 0。如果有 n 个叶子结点，则非叶子结点有 $n-1$ 个，所以总共结点数目是 $2n-1$ 个。同时，将剩下的 $n-1$ 个双亲结点域初始化为 0，这主要是为了方便查找权值最小的结点。

依次选择两个权值最小的结点，分别作为左子树结点和右子树结点，修改它们的双亲结点域，使它们指向同一个双亲结点；修改双亲结点的权值，使其等于两个左、右子树结点权值的和；修改左、右孩子结点域，使其分别指向左、右孩子结点。重复执行上述操作 $n-1$ 次，即求出 $n-1$ 个非叶子结点的权值，即可得到一棵哈夫曼树。

通过求得的哈夫曼树，得到每个叶子结点的哈夫曼编码。从叶子结点 c 开始，通过结点 c 的双亲结点域，找到结点的双亲。通过双亲结点的左孩子域和右孩子域判断该结点 c 是其双亲结点的左孩子还是右孩子，如果是左孩子，则编码为'0'；否则编码为'1'。按照这种方法，直到找到根结点为止，即可求出叶子结点的编码。

(1) 构造哈夫曼树。

这部分主要是哈夫曼树的构造，程序代码实现如下。

```
public void HuffmanCoding(HTNode HT[], String HC[], int w[], int n)
//构造哈夫曼树 HT,哈夫曼树的编码存放在 HC 中,w 为 n 个字符的权值
{
    if (n <= 1)
        return;
    int m = 2 * n - 1;
    for (int i = 1; i <= n; i++)    //初始化 n 个叶子结点
    {
```

```
            HT[i] = new HTNode();
            HT[i].weight = w[i-1];
            HT[i].parent = 0;
            HT[i].lchild = HT[i].rchild = 0;
        }
        for (int i = n+1; i <= m; i++)//将 n - 1 个非叶子结点的双亲结点初始化为 0
        {
            HT[i] = new HTNode();
            HT[i].parent = 0;
        }
        for (int i = n + 1; i <= m; i++)          //构造哈夫曼树
        {
            List<Integer> s = Select (HT,i - 1);  //查找树中权值最小的两个结点
            int s1 = s.get(0),s2 = s.get(1);
            HT[s1].parent = i;
            HT[s2].parent = i;
            HT[i].lchild = s1;
            HT[i].rchild = s2;
            HT[i].weight = HT[s1].weight + HT[s2].weight;
        }

        //从叶子结点到根结点,求每个字符的哈夫曼编码
        //求 n 个叶子结点的哈夫曼编码
        for(int i=1;i<=n;i++)
        {
            char cd[] = new char[n+1];
            int start = n-1;
            for(int c=i,f=HT[i].parent;f!=0;c=f,f=HT[f].parent)  //从叶子结点到根结点求编码
            {
                if (HT[f].lchild == c)
                    cd[--start] = '0';
                else
                    cd[--start] = '1';
            }
            HC[i] = new String();
            String str = new String(cd);
            HC[i] = str;                  //将当前求出结点的哈夫曼编码复制到 HC
        }
    }
```

(2) 查找权值最小和次小的两个结点。

这部分主要是在结点的权值中选择权值最小和次小的两个结点,将其作为二叉树的叶子结点,其程序代码实现如下。

```
    public List Select(HTNode t[],int n)
    //在 n 个结点中选择两个权值最小的结点序号,其中 s1 最小,s2 次小
    {
        List result = new ArrayList();
        int s1 = Min(t, n);
        int s2 = Min(t,n);
        if(t[s1].weight > t[s2].weight) //如果序号 s1 的权值大于序号 s2 的权值,将两者交换,使
```

```
                                        //s1 最小,s2 次小
     {
          int x = s1;
          s1 = s2;
          s2 = x;
     }
    result.add(s1);
    result.add(s2);
    return result;
}
public int Min(HTNode t[], int n)
 //返回树的 n 个结点中权值最小的结点序号
{
    int f = Integer.MAX_VALUE;      //f 为一个无限大的值
    int flag = 0;
    for (int i = 1; i < n + 1; i++) {
        if (t[i].weight < f && t[i].parent == 0) {
            f = t[i].weight;
            flag = i;
        }
    }
    t[flag].parent = 1;             //给选中的结点的双亲结点赋值 1,避免再次查找该结点
    return flag;
}
```

(3) 将哈夫曼编码翻译为字符串序列。

这部分主要是将哈夫曼编码翻译为字符串序列,其实现原理及程序代码实现如下。

根据哈夫曼树的构造原理,从根结点开始遍历,如果遇到的编码是 0,则沿着左孩子结点往下遍历;若遇到的编码是 1,则沿着右孩子结点往下遍历。以此类推,对其他结点重复执行以上操作,直到遍历到叶子结点为止,此时扫描到的编码就是该叶子结点对应的字符。

```
public void GetStr(HTNode HT[], int nums, char w[], char str[]) {
    int i = 0, n = 2 * nums - 1;
    int length = str.length;
    for (i = 0; i < length; i++) {
        if (str[i] == '1')
            n = HT[n].rchild;
        else if (str[i] == '0')
            n = HT[n].lchild;
        else
            return;
        for (int j = 1; j <= nums; j++) {
            if (j == n) {
                n = 2 * nums - 1;
                System.out.print(w[j-1] + " ");
                break;
            }
        }
    }
}
```

(4) 运行测试代码。

这部分是运行测试代码,测试代码主要包括头文件、宏定义、函数的声明和主函数,其程序代码实现如下。

```
public static void main(String args[])
{
    System.out.print("请输入叶子结点的个数: ");
    Scanner sc = new Scanner(System.in);
    int n = Integer.parseInt(sc.nextLine());
    int m = 2 * n - 1;
    HTNode HT[ ] = new HTNode[m + 1];
    String HC[ ] = new String[n + 1];
    int w[ ] = new int[n];              //为 n 个结点的权值分配内存空间
    for(int i = 0;i < n;i++)
    {
        System.out.print(String.format("请输入第 %d 个结点的权值:",i + 1));
        w[i] = Integer.parseInt(sc.nextLine());
    }
    HuffmanTree HTree = new HuffmanTree();
    HTree.HuffmanCoding(HT,HC,w, n);
    for(int i = 1;i <= n;i++)
        System.out.println("哈夫曼编码:" + HC[i]);
    char wname[ ] = {'A','B','C','D','E','F'};
    System.out.print("100101011" + "的译码结果:");
    HTree.GetStr(HT,n,wname,"100101011".toCharArray());
}
```

在算法的实现过程中,对象数组 HT 在初始时和哈夫曼树生成后的状态如图 5-39 所示。

下标	weight	parent	lchild	rchild
1	1	0	0	0
2	3	0	0	0
3	6	0	0	0
4	9	0	0	0
5		0		
6		0		
7		0		

(a) 初始时的HT状态

下标	weight	parent	lchild	rchild
1	1	5	0	0
2	3	5	0	0
3	6	6	0	0
4	9	7	0	0
5	4	6	1	2
6	10	7	5	3
7	19	0	4	6

(b) 生成哈夫曼树后的HT状态

图 5-39 HT 在初始时和生成哈夫曼树后的状态变化

生成的哈夫曼树如图 5-40 所示。可以看出,权值为 1、3、6 和 9 的哈夫曼编码分别是 100、101、11 和 0。

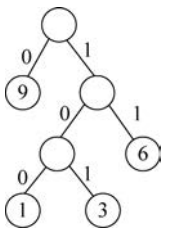


图 5-40 哈夫曼树

以上算法是从叶子结点开始到根结点逆向求哈夫曼编码的算法。当然也可以从根结点开始到叶子结点正向求哈夫曼编码,该问题留给读者思考。

程序运行结果如下。

请输入叶子结点的个数：4
请输入第 1 个结点的权值：1
请输入第 2 个结点的权值：3
请输入第 3 个结点的权值：6
请输入第 4 个结点的权值：9
哈夫曼编码：100
哈夫曼编码：101
哈夫曼编码：11
哈夫曼编码：0
"100101011"的译码结果：A B D C

5.7.2　利用二叉树求解算术表达式的值

【例 5-6】 通过键盘输入一个表达式，如 6+(7−1)×3+9/2，将其转换为二叉树(表达式树)，然后通过二叉树的遍历操作求解表达式的值。

【分析】 与利用栈求解表达式值的思想类似，将中缀表达式转换为表达式树也需要借助栈的后进先出特性实现，区别在于将运算符出栈之后不是将该运算符直接输出，而是将其作为子树的根结点创建一棵二叉树，再将该二叉树作为一个表达式入栈。在算法结束时，由此构造出一棵由这些运算符和操作数组成的二叉树，并利用二叉树的后序遍历求解表达式的值。

【算法思想】 设置两个栈：运算符栈 OptrStack 和表达式栈 ExpTreeStack，分别用于存放运算符和表达式树的根结点。假设 θ_1 为栈顶运算符，θ_2 为当前扫描的运算符。依次读入表达式中的每个字符，根据扫描到的当前字符进行以下处理。

(1) 初始化栈，并将'#'入栈。

(2) 若当前读入的字符 θ_2 是操作数，则将该操作数压入 ExpTreeStack 栈，并读入下一个字符。

(3) 若当前字符 θ_2 是运算符，则将 θ_2 与栈顶的运算符 θ_1 比较。

① 若 θ_1 优先级低于 θ_2，则将 θ_2 压入 OptrStack 栈，继续读入下一个字符。

② 若 θ_1 优先级高于 θ_2，则从 OptrStack 栈中弹出 θ_1，将其作为子树的根结点，并使 ExpTreeStack 栈执行两次出栈操作，弹出的两个表达式 rcd 和 lcd 分别作为 θ_1 的右子树和左子树，并将所创建二叉树的根结点压入 ExpTreeStack 栈中。

③ 若 θ_1 的优先级与 θ_2 相等，且 θ_1 为'('，θ_2 为')'，则将 θ_1 出栈，继续读入下一个字符。

(4) 如果 θ_2 的优先级与 θ_1 相等，且 θ_1 和 θ_2 都为'#'，则从 OptrStack 栈中将 θ_1 弹出。此时 OptrStack 栈为空，中缀表达式转换为表达式树，ExpTreeStack 的栈顶元素就是表达式树的根结点，算法结束。

重复执行步骤(2)～(4)，直到所有字符读取完毕且 OptrStack 为空为止。

利用以上算法可以将 6+(7−1)×3+9/2 转换为一棵二叉树，其过程如图 5-41 所示。

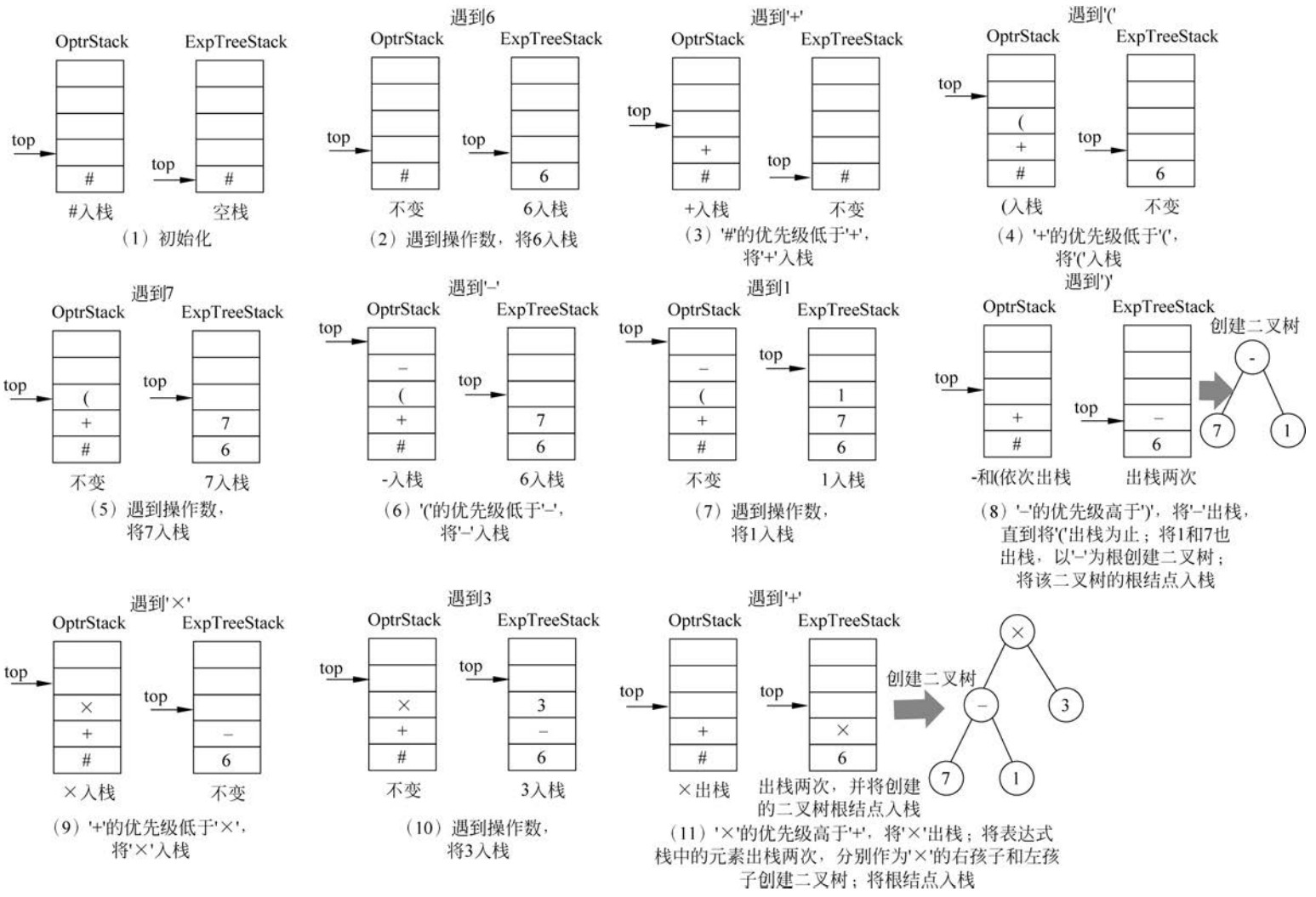
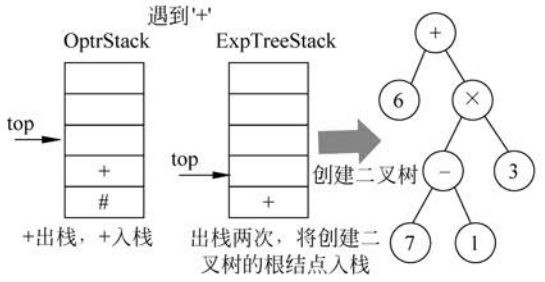
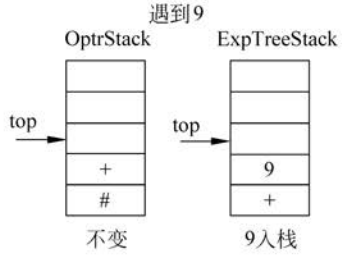


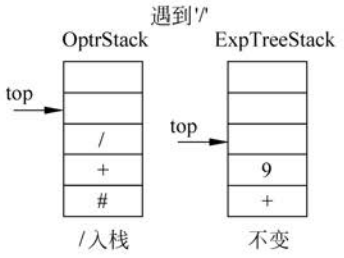
图 5-41　创建表达式树的过程



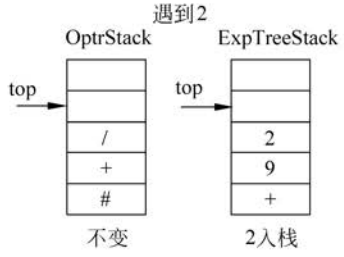
（12）栈顶的'+'的优先级高于当前的'+'，将'+'出栈；将表达式栈中的元素出栈两次，分别作为'+'的右孩子结点和左孩子结点创建二叉树；将根结点入栈，'+'入运算符栈



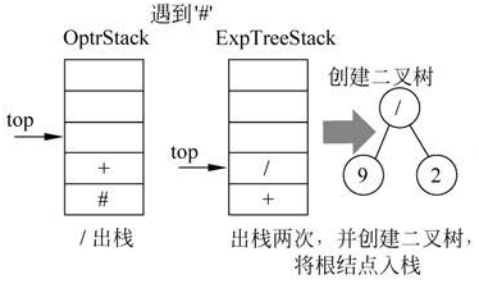
（13）遇到操作数，将9入栈



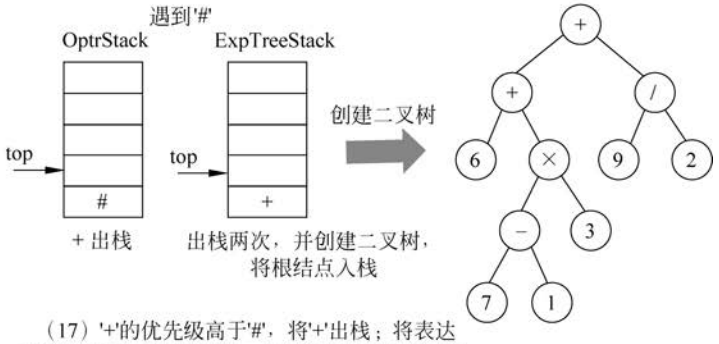
（14）'+'的优先级低于'/'，将'/'入栈



（15）遇到操作数，将2入栈



（16）'/'的优先级高于'#'，将'/'出栈；将表达式栈中的元素出栈两次，分别作为'/'的右孩子和左孩子结点创建二叉树；将根结点入表达式栈



（17）'+'的优先级高于'#'，将'+'出栈；将表达式栈中的元素出栈两次，分别作为'+'的右孩子和左孩子结点创建二叉树；将根结点入表达式栈

图 5-41 （续）

对于表达式树,通过中序遍历可以得到对应的中缀表达式,通过后序遍历可以得到对应的后缀表达式。根据输入的字符串 str,创建表达式树的算法如下。

```
public BitNode CreateExpTree(String str)
//表达式树的创建
  {
        char data[ ] = new char[10];
        ExpTreeStack Expt = new ExpTreeStack();
        OptrStack Optr = new OptrStack();
        ExpBiTree T = new ExpBiTree();
        Optr.Push("#");
        int n = str.length();
        int i = 0;
        while (i < n || Optr.GetTop() != null) {
            if (i < n && !IsOperator(str.charAt(i))) {
                int j = 0;
                data[j] = str.charAt(i);
                j++;
                i++;
                while (i < n && !IsOperator(str.charAt(i)))
                {
                    data[j] = str.charAt(i);
                    i++;
                }
                if (i >= n)
                    j--;
                BitNode p = T.CreateETree(String.valueOf(data), null, null);
                Expt.Push(p);
            }
            else
            {
                if (Precede(Optr.GetTop(), str.substring(i, i + 1)) == '<') {
                    Optr.Push(str.substring(i, i + 1));
                    i += 1;
                } else if (Precede(Optr.GetTop(), str.substring(i, i + 1)) == '>') {
                    String theta = Optr.Pop();
                    BitNode rcd = Expt.Pop();
                    BitNode lcd = Expt.Pop();
                    BitNode p = T.CreateETree(theta, lcd, rcd);
                    Expt.Push(p);
                }
                else if (Precede(Optr.GetTop(), str.substring(i, i + 1)) == '=') {
                    String theta = Optr.Pop();
                    i += 1;
                }
            }
        }
        return Expt.GetTop();
    }
```

根据得到的表达式树,利用二叉树的后序遍历即可求出表达式的值,其算法实现如下。

```
public float CalcExpTree(BitNode T) throws Exception
//后序遍历表达树进行表达式求值
  {
```

```
        float lvalue = 0, rvalue = 0;
        int len = 0;
        if (T.lchild == null && T.rchild == null)
        {
            len = 0;
            for(int i = 0;i < T.data.length();i++)
            {
                if(T.data.charAt(i)> = '0'&&T.data.charAt(i)< = '9')
                    len++;
            }
            return StrtoInt(T.data,len);
        }
        else
        {
            lvalue = CalcExpTree(T.lchild);
            rvalue = CalcExpTree(T.rchild);
            return GetValue(T.data, lvalue, rvalue);
        }
    }
```

求表达式值的主函数如下。

```
public class ExpressTree
{
    //7 种运算符
    operator()#";
    //运算符优先级比较表
    char prior_table[][] = {{'>', '>', '<', '<', '<', '>', '>'},
            {'>', '>', '<', '<', '<', '>', '>'},
            {'>', '>', '>', '>', '<', '>', '>'},
            {'>', '>', '>', '>', '<', '>', '>'},
            {'<', '<', '<', '<', '<', ' = ', ' '},
            {'>', '>', '>', '>', ' ', '>', '>'},
            {'<', '<', '<', '<', '<', ' ', ' = '}};

    public boolean IsOperator(char ch)
    //判断 ch 是否为运算符
    {
        int i = 0;
        int length = operator.length();
        while (i < length && operator.charAt(i) != ch)
            i += 1;
        if (i > = length)
            return false;
        else
            return true;
    }

    public int StrtoInt(String str, int n)
    //将数值型字符串转换为 int 型
    {
        int res = 0;
        int i = 0;
        while (i < n) {
            res = res * 10 + str.charAt(i) - '0';
```

```
            i += 1;
        }
        return res;
    }

    public char Precede(String ch1, String ch2)
    //判断运算符的优先级
    {
        int i = 0, j = 0;
        while (i < operator.length() && !ch1.equals(String.valueOf(operator.charAt(i))))
            i += 1;
        while (j < operator.length() && !ch2.equals(String.valueOf(operator.charAt(j))))
            j += 1;
        return prior_table[i][j];
    }
    public float GetValue(String ch, float a, float b) throws Exception
//求值
    {
        if (ch.equals("+"))
            return a + b;
        else if (ch.equals("-"))
            return a - b;
        else if (ch.equals("*"))
            return a * b;
        else if (ch.equals("/"))
            return a / b;
        else
            throw new Exception("运算符错误!");
    }

    public static void main(String args[]) throws Exception
    //测试方法
    {
        System.out.println("请输入算术表达式串:");
        Scanner sc = new Scanner(System.in);
        String str = sc.nextLine();
        ExpressTree Tree = new ExpressTree();
        BitNode T = Tree.CreateExpTree(str);
        ExpBiTree root = new ExpBiTree();
        System.out.println("先序遍历:");
        root.PreOrderTree(T);
        System.out.println("\n中序遍历:");
        root.InOrderTree2(T);
        System.out.print("\n表达式的值:");
        float value = Tree.CalcExpTree(T);
        System.out.println(value);
    }
}
```

程序运行结果如下。

请输入算术表达式串:

6+(7-1)×3+9/2#

先序遍历:

＋　＋　6　×　－　7　1　3　/　9　2

中序遍历：

6　＋　7　－　1　×　3　＋　9　/　2

表达式的值：28.5

思政元素
哈夫曼树的构造是整体和部分关系的具体体现，由于每次选择的都是权值最小的结点，因此最终构成的二叉树的权值才会最小。在做任何事情时都应该有全局观念，把握好整体和局部的关系，增强大局意识和协同意识，只有这样才能把事情做到最好。“大河有水小河满，小河无水大河干”“不谋全局者，不足谋一域”体现了整体与部分的关系，整体和部分不可分割且相互影响，任何部分的变化都会影响整体，整体的变化也会影响部分。

5.8　实验

5.8.1　基础实验

1. 基础实验 1：实现二叉树的基本运算

实验目的：理解二叉树的存储结构，熟练掌握其基本操作。

实验要求：创建一棵如图 5-42 所示的二叉树，并要求进行以下基本运算：

(1) 创建二叉树；

(2) 按照先序遍历方式输出二叉树的各结点；

(3) 按照中序遍历方式输出二叉树的各结点；

(4) 按照后序遍历方式输出二叉树的各结点；

(5) 按照层次遍历方式输出二叉树的各结点。

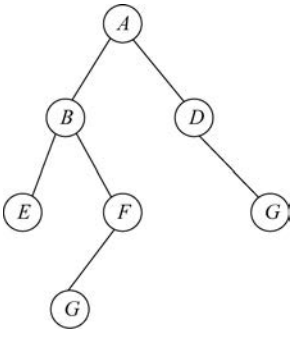


图 5-42　二叉树示例

2. 基础实验 2：利用二叉树的遍历方式构造二叉树

实验目的：熟练掌握二叉树的先序、中序、后序遍历算法思想。

实验要求：已知二叉树的先序遍历序列为 *ABDEGCF*，中序遍历序列为 *DBGEACF*，编写算法创建二叉树。创建一个 BiTree 类，该类应至少包含以下基本运算。

(1) 构造二叉树；

(2) 按后序遍历方式输出二叉树的各结点；

(3) 按层次遍历方式输出二叉树的各结点；

(4) 输出二叉树的高度。

5.8.2 综合实验

1. 综合实验 1：哈夫曼树及其应用

实验目的：深入理解二叉树的存储结构，熟练掌握哈夫曼树的构造及哈夫曼编码。

实验要求：一个单位有 12 个部门，每个部门都有一部电话，但是整个单位只有一根外线。当有电话打过来时，由转接员转到内线电话。已知各部门使用外线电话的频率为 5、20、10、12、8、43、5、6、9、15、19、32(次/天)。

利用哈夫曼树算法思想设计内线电话号码，使得接线员拨号次数尽可能少，具体设计要求如下。

(1) 依据使用外线电话的频率构造二叉树。

(2) 输出设计出的各部门内线电话号码。

实验思路：将各部门外线电话的使用频率作为权值，以此构造哈夫曼树，对哈夫曼树进行先序遍历即可得到内线电话号码。

2. 综合实验 2：求解算术表达式的值

实验目的：深入理解二叉树的存储结构，熟练掌握二叉树的先序、中序、后序遍历思想及其应用。

实验要求：实现一个简单的运算器。通过键盘输入一个包含圆括号、加、减、乘、除等符号组成的算术表达式字符串，输出该算术表达式的值。该运算器的设计要求如下。

(1) 系统应至少能实现加、减、乘、除等运算。

(2) 利用二叉树算法思想求解表达式的值。先构造由表达式构成的二叉树，然后对二叉树进行后序遍历，以此求解算术表达式的值。

实验思路：依次扫描输入的算术表达式中的每个字符，当遇到运算符时，则将扫描到的运算符与栈顶运算符的优先级比较，若栈顶运算符的优先级小于当前运算符优先级，则将当前运算符入栈；若栈顶运算符优先级高于当前运算符优先级，则将栈顶运算符出栈，且将操作数栈中的元素出栈两次，由该运算符作为根结点构造二叉树，其左、右孩子结点为操作数栈出栈的操作数。对构造好的二叉树进行后序遍历即可得到后序遍历序列，最后利用栈对该后缀表达式求值。

小结

树在数据结构中占据着非常重要的地位，树反映的是一种层次结构的关系。在树中，每个结点只允许有一个直接前驱结点，但允许有多个直接后继结点，结点与结点之间是一种一对多的关系。

树的定义是递归的。一棵非空树或者为空，或者是由 m 棵子树 $T_1, T_2, \cdots, T_m$ 构成的，这 m 棵子树又是由其他子树构成的。树中的孩子结点没有次序之分，是一种无序树。

二叉树最多有两棵子树，这两棵子树分别叫作左子树和右子树。二叉树可以看作是树的特例，但是与树不同的是，二叉树的两棵子树有次序之分。二叉树也是递归定义的，二叉

树的两棵子树由左子树和右子树构成。

在二叉树中，存在两种特殊的树：满二叉树和完全二叉树。满二叉树中的每个非叶子结点都存在左子树和右子树，所有的叶子结点都处在同一层次上。完全二叉树的前 n 个结点结构与满二叉树相同。满二叉树是一种特殊的完全二叉树。

采用顺序存储的完全二叉树可以实现随机存取，实现起来也比较方便。但是，如果二叉树不是完全二叉树，则采用顺序存储会浪费大量的存储空间。因此，一般情况下，二叉树采用链式存储——二叉链表。在二叉链表中，结点有一个数据域和两个指针域，其中，一个指针域指向左孩子结点，另一个指针域指向右孩子结点。

二叉树的遍历分为先序遍历、中序遍历和后序遍历。二叉树遍历的过程就是将二叉树这种非线性结构转换成线性结构。通过将二叉树线索化，不仅可以充分利用二叉链表中的空指针域，而且能很方便地找到指定结点的前驱结点。

在哈夫曼树中，只有叶子结点和度为 2 的结点。哈夫曼树是带权路径最小的二叉树，通常用于解决最优化问题。

树、森林和二叉树可以相互进行转换。在实际应用中，树实现起来不是很方便，为此可以将树的问题转化为二叉树的相关问题加以解决。

习题

本书提供在线测试习题，扫描下面的二维码，可以获取本章习题。

在线测试