第5章 树和二叉树

线性表、栈、队列、串、数组都是线性的逻辑结构，数据元素之间具有前驱后继的线性关系。树形结构是比线性结构更为复杂的逻辑结构，其中以树和二叉树最为常用，比较适合描述具有层次关系的数据，如家族族谱、社会组织机构等。在计算机领域，树形结构也具有广泛的应用，如在编译软件中用语法树表示源程序的语法结构，在人工智能应用中用决策树进行分类等。

树的结构较为复杂，为处理方便，可转换为二叉树进行存储和处理。本章在讲述树的基础上，重点讨论二叉树的存储和实现，并研究了树和二叉树的转换关系，最后给出了二叉树的典型应用。

【学习重点】

◆ 树的遍历操作；

◆ 二叉树的基本性质；

◆ 二叉树的遍历操作；

◆ 二叉树的存储结构、操作实现；

◆ 二叉树和树的相互转换及关系；

◆ 哈夫曼树和哈夫曼编码。

【学习难点】

◆ 二叉树的构造过程及实现；

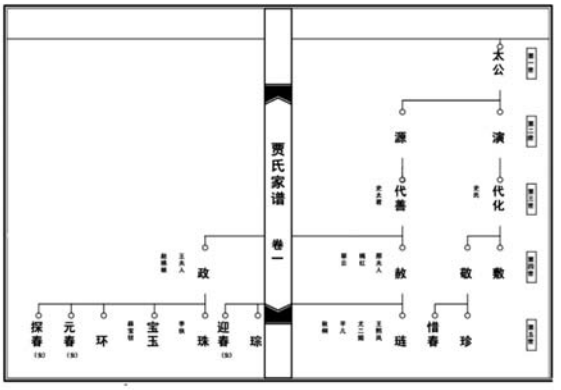
◆ 二叉树遍历过程的非递归实现；

◆ 线索二叉树的建立。

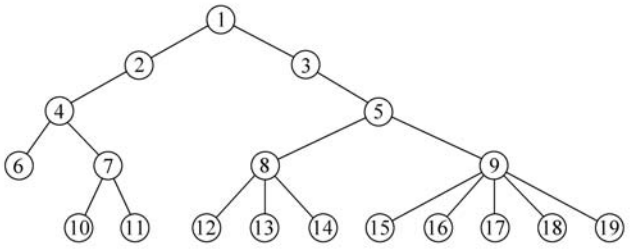
5.1 引　　言

树形结构是日常生活和计算机领域常用的数据结构之一，用来描述数据元素之间的一对多关系或者层次关系，许多应用和问题的数学模型都可以抽象成树形结构。下面介绍两个树形结构的应用实例。

图 5.1(a)显示的是《红楼梦》中贾氏宗族家谱图的一部分。家谱中的每个成员具有姓名、性别、配偶等信息，可以抽象成一个数据元素，用圆圈表示；成员之间具有孩子、双亲、子孙、祖先、兄弟等联系，其中最为直接的联系是双亲和孩子的关系，可以用两个圆圈之间的连线表示。因此，图 5.1(a)的家谱图可以抽象成图 5.1(b)所示的树形图。古代家谱一般是从右到左书写的，为了阅读方便，图 5.1(b)调整为从左向右。



(a) 红楼梦贾氏家谱图



1：太公 2：演 3：源 4：代化 5：代善 6：敷 7：敬 8：赦 9：政 10：珍
11：惜春 12：琏 13：琮 14：迎春 15：珠 16：宝玉 17：环 18：元春 19：探春

(b) 家谱图对应的数据结构

图 5.1 《红楼梦》贾氏宗族家谱图及对应的数据结构

下面再看另一个例子。2016 年 3 月，阿尔法围棋(AlphaGo)与围棋世界冠军、职业九段棋手李世石进行围棋人机大战，以 4∶1 的总比分获胜，该事件成为人工智能发展史上的一个里程碑。下围棋时，棋手根据当前盘面的状态，从众多可能的落子点中选择最佳的位置进行应对，然后对手再根据新的当前状态进行应对，依此类推，一直到分出胜负为止。对弈过程如图 5.2(a)所示，很显然这是一个树形演变的过程。阿尔法围棋作为一款人工智能软件，其工作过程与人类棋手类似，但由于围棋网格点数量庞大，有 361 个，如果采用暴力破解方法，等概率地尝试每个可能的着棋位置，据推算大概有 10^{359} 种可能性，远远超出了当前计算机的运算能力。因此阿尔法围棋采用两个深度学习网络，即策略网络和评价网络，通过学习人类围棋下法，预测出下一步棋的最佳位置，成功将绝大多数搜索分支进行了裁剪。围棋对弈树可以抽象成图 5.2(b)所示的树形图。在这个图中，每个数据元素代表一个盘面状态，记录当前黑白棋子的分布情况，同时还应该记录当前状态出现的概率。

从上述两个例子可以看出，在树形结构中，数据元素之间的关系比线性关系复杂，图中只有一个根结点，上一层结点与下一层相邻结点之间，是一对多的关系，同一层结点之间是并列关系，总体来说结点之间是一个层次关系或者树形关系。对于这种树形结构，如何存储，如何实现遍历、查询、创建和销毁等基本操作，是本章重点讲解的内容。

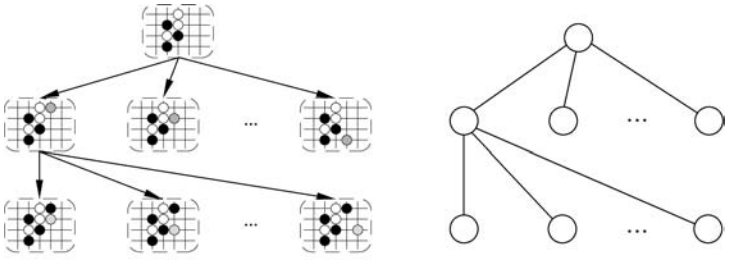


图 5.2　围棋对弈树及其数据结构

5.2　树与树的存储结构

树是一种最为常见的树形结构，数据元素之间是一对多的关系，是一种双亲和孩子的关系，也可以看成是一种层次关系。下面将详细介绍树的基本概念、逻辑结构和存储结构。

5.2.1　树的基本概念

1. 树的定义

树(tree)是由 $n(n\geqslant 0)$个结点组成的有限集合。当 $n=0$ 时，称为空树，这是一种特殊情况。当 $n>0$ 时，在任意一棵非空树中：①有且仅有一个特定的称为根(root)的结点；②当 $n>1$ 时，其余结点可分为 $m(m>0)$个互不相交的有限集合 $T_1,T_2,\cdots,T_m$，其中每一个集合本身又是一棵树，称为根结点的子树(subtree)。

图 5.3 是一棵具有 10 个结点的树，可以用一个二元组 $T=(M,R)$表示，其中 $M=\{A,B,C,D,E,F,G,H,I,J\}$，表示结点集合；$R=\{<A,B>,<A,C>,<A,D>,<B,E>,<B,F>,<D,G>,<D,H>,<D,I>,<G,J>\}$，表示联系集合。在这棵树中，根结点是 A，去掉根结点及相关联的边后，其余结点分为三部分，它们之间没有交集，并且每部分仍然满足树的定义，是原树的 3 棵子树。依此类推，每部分仍然可作相同的分解，直到每棵子树只剩一个根结点为止。

显然，树的定义是递归的，在树的定义中又用到了树的定义，是一种递归的数据结构。不难看出，在树中从根结点到每个结点的路径是唯一的，中间经过其他结点的数目也是唯一的。与根结点距离相同的结点可以看作一层，因此树结构也可以看成是一种分层结构。

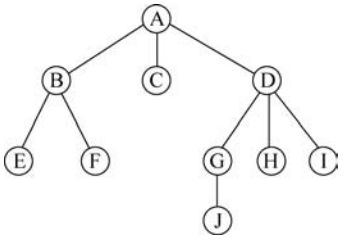


图 5.3　树结构

为了更方便地描述树结构，下面给出几个与其相关的基本术语。

2. 树的基本术语

1) 结点的度和树的度

某结点所拥有的子树的个数，称为该结点的度(degree)。图 5.3 中，根结点 A 有 3 棵子树，其度为 3；结点 B 有 2 棵子树，其度为 2；结点 C、E、F 等结点，度为 0。

树中所有结点的度的最大值，称为树的度。图 5.3 中，结点 A、D 结点的度都为 3，是所

有结点度的最大值,所以该树的度为3。

2) 叶子结点和分支结点

如果一个结点的度为0,则该结点称为叶子结点(leaf node)。图5.3中,结点C、E、F、J、H、I的度为0,是叶子结点。

如果一个结点的度不为0,则该结点称为分支结点(branch node)。一个结点,要么是叶子结点,要么是分支结点。图5.3中除叶子结点之外的其他结点,都是分支结点。

3) 孩子结点、双亲结点和兄弟结点

假设某结点M的度为$n(n>0)$,意味着它有n棵子树,则这些子树的根结点被称为结点M的孩子结点(children node),与此相对,结点M被称为这些孩子结点的双亲结点(parent node)。双亲结点是一个结点,不要按照字面意思错以为是两个结点。具有同一个双亲结点M的孩子结点之间,互称为兄弟结点(brother node)。在图5.3中,结点D有3棵子树,其根结点G、H、I是结点D的孩子结点,结点D被称为这些孩子结点的双亲结点,根结点G、H、I互相称为兄弟结点。

不难理解,在树中,一个结点可以有多个孩子结点(分支结点),也可以没有孩子结点(叶子结点);根结点没有双亲结点;除根结点之外的其他结点,只能有一个双亲结点。

4) 路径和路径长度

树中两个结点之间的路径(path)是由它们之间沿着边序列所经过的结点序列构成的,而路径长度(path length)是路径上所经过边的个数。由于树中的边是有方向的,可以看作由双亲指向孩子,树中的路径也有方向,是由上向下的,因此同一双亲的两个孩子之间,甚至同一结点不同子树上的结点之间都不存在路径。假设$p_1,p_2,\cdots,p_n$是一条路径,则必然有结点p_i是$p_{i+1}(i\geqslant 1$且$i+1\leqslant n)$的双亲。在图5.3中,A,D,G,J是一条路径,任何两个相邻的结点,前一个是后一个的双亲;在C,A,D,G中,C和D、G在A的不同子树上,并且C不是A的双亲,因此不是一条路径。

5) 祖先和子孙

在树中,如果从结点A到结点B有一条路径,则称结点A为结点B的祖先(ancestor);与此相对,结点B称为结点A的子孙(descendant)。显然,可以把双亲看作一种特殊的祖先,把孩子看作一种特殊的子孙。在以某结点为根的树中,该结点是树中其他任意结点的祖先。

6) 有序树和无序树

如果一棵树中,各结点子树位置从左到右的次序是有实际意义的,交换后对应的意义发生变化,形成不同的树,则该树称为有序树(ordered tree);反之,则称为无序树(unordered tree)。例如在家谱树中,同一结点的不同子树之间,左右次序代表不同孩子的排行,最左边的表示长子,如果交换次序,意味着他们的排行发生了变化,也就意味着家谱树发生了变化,因此家谱树是有序树。在本书中,如果不做特殊说明,出现的树都是有序树。

3. 树的性质

由树的基本概念,可以得出树具有如下基本性质:

(1) 树中的结点数目为所有结点度数加1(加根结点);

(2) 度为m的树中第i层最多有m^{i-1}个结点;

(3) 高度为h、度为m的树至多$(m^h-1)/(m-1)$个结点;

(4) 具有 n 个结点的度为 m 的树的最小高度为 $\lceil \log_m(n(m-1)+1) \rceil$。

由于篇幅关系，本书只列出这些结论，不给出它们的证明，感兴趣的读者可以自行推导。

4. 森林

m 棵($m \geqslant 0$)互不相交的树组成的集合，称为森林(forest)。任何具有根结点的树，删除根结点后剩余的部分，就构成了森林，有时称为该根结点的子树森林。

5.2.2 树的逻辑结构

线性表中数据元素之间的关系主要体现在前驱-后继关系上，是一对一的关系。树作为一种逻辑结构，它是一种分支结构，同时也是一种分层结构。树中数据元素之间的逻辑关系主要体现在双亲-孩子关系上，每个结点只有一个双亲，但可以具有多个孩子，是一种一对多的关系。与线性表相比，树具有以下两个特点：

(1) 树的根结点没有前驱结点，除根结点之外的所有结点有且只有一个前驱结点；

(2) 树中所有结点可以有零个或多个后继结点，其中叶子结点没有后继结点，分支结点可以有一个或者多个后继结点。

1. 树的抽象数据类型

作为一种基本的数据结构，树的应用很广泛，在不同的实际应用中，树的基本操作不尽相同。简单起见，基本操作只包含树的遍历，一棵树的抽象数据类型定义如下：

```
ADT Tree
{
    数据对象:D = {a_i | a_i ∈ ElemSet, i = 1,2,…,n, n >= 0, a_i 可以为任意数据}
    数据关系:R = {<a_i-1, a_i> | a_i-1, a_i ∈ D, i = 2,…,n 且 a_i-1 和 a_i 之间逻辑关系满足双亲与孩
                 子的关系}
    基本操作:
        initTree:初始化操作,建立一棵结点数目为 0 的树。
        destroyTree:销毁操作,销毁一棵已经存在的树,释放该树占用的存储空间。
        preOrder:前序遍历树,不改变树结构的情况下,输出树的前序遍历序列。
        postOrder:后序遍历树,不改变树结构的情况下,输出树的后序遍历序列。
        levelOrder:层序遍历树,不改变树结构的情况下,输出树的层序遍历序列。
}
```

2. 树的遍历操作

所谓树的遍历，就是从树的根结点出发，按照某种次序，访问树中的所有结点，每个结点访问一次并且只能访问一次。遍历的过程是按照时间顺序依次访问结点的过程，也是时间上的线性关系。对于线性表来说，线性的逻辑结构转换为时间上的线性结构，是非常自然和简单的过程；对于树结构来说，逻辑结构是非线性的，转换为时间上的线性结构，在访问规则上要复杂得多。由树的定义可知，树由根结点和子树构成，根据根结点和子树的访问次序，可以有前序(根)遍历和后序(根)遍历；同时，树是一种层次结构，可以按照与根结点路径长度从小到大的顺序依次遍历各个结点，这就是树的层序遍历。

树的遍历方式主要有以下 3 种。

1) 前序遍历

树的前序遍历操作的定义如下：

(1) 若树为空,则空操作返回;

(2) 访问根结点;

(3) 按照从左到右的顺序依次前序遍历根结点的每一棵子树。

对如图 5.4 所示的树进行前序遍历操作,结果为 ABEFCDGJHI。

2) 后序遍历

树的后序遍历操作的定义如下:

(1) 若树为空,则空操作返回;

(2) 按照从左到右的顺序依次后序遍历根结点的每一棵子树;

(3) 访问根结点。

对如图 5.4 所示的树进行后序遍历操作,结果为 EFBCJGHIDA。

3) 层序遍历

在树中,规定根结点的层数为 1,其余的任一结点,若结点在第 k 层,则其孩子在第 $k+1$ 层。树中所有结点的最大层数称为树的深度(depth),也称为树的高度。如图 5.4 所示的树中,树的高度为 4,结点所在层数也在图中用虚线标出。

树的层序遍历也称为树的广度遍历,其操作定义为:从树的根结点(第 1 层)开始,按自上而下的顺序,在同一层中,按从左到右的顺序依次访问每个结点。对如图 5.4 所示的树进行层序遍历操作,结果为 ABCDEFGHIJ。

在树中,如果从 1 开始,按照层序遍历的顺序对每个结点进行编号,则称这种编号方式为层序编号。

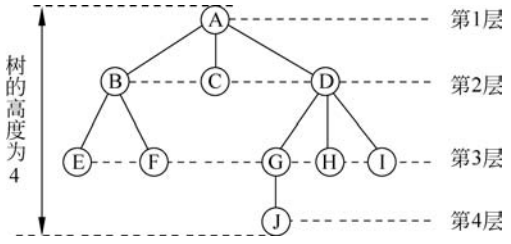


图 5.4　树的层次

5.2.3　树的存储结构

任何一种逻辑结构的存储,都包括数据元素的存储和数据元素之间逻辑关系的存储,树结构也是如此。树结构中逻辑关系的存储主要体现为双亲和孩子关系的存储,是一对多关系的存储,其主要的存储方式包括双亲表示法、孩子链表表示法、双亲孩子表示法和孩子兄弟表示法等。

1. 双亲表示法

由树的定义可以知道,树中每个结点除根结点之外只有一个双亲结点。根据这个特点,树中的数据元素用一个一维数组来存储。每个数组元素都是一个结点结构,如图 5.5 所示,包含 data 和 parent 两部分。其中,data 是数据域,存储树中数据元素即树结点的值;parent 是指针域,也称为游标域,是该结点的双亲结点在一维数组中的下标。

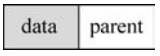


图 5.5　双亲表示法中数组元素的结构

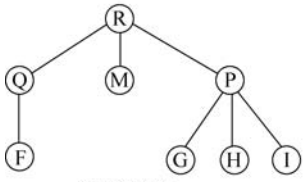
该结点结构可用C++语言中的结构体表述。由于数据元素的类型没有指定，因此可采用模板机制来实现。结点结构的定义见代码5.1。

代码5.1 双亲表示法中结点结构的定义

```
template < typename Element >
struct ParNode {
    Element data;          //数据域,存储结点的数据元素
    int parent;            //指针域,也称为游标,是该结点的双亲结点在一维数组中的下标
};
```

图5.6(a)所示树结构的双亲表示法的存储结构如图5.6(b)所示。其中，指针域parent为－1，表示该结点没有双亲结点，是根结点；数据元素在数组中的位置没有统一规定，可以是任意的，本示例中是按层序遍历的顺序进行存储的。

可以看出，在双亲表示法中，给定一个树结点的存储位置，可以很方便地得到其双亲结点，也可以比较方便地查找根结点。由于没有直接存储孩子结点的信息，因此查找某个结点的孩子信息并不方便，但可以在遍历整棵树的过程中间接得到。



(a) 树结构

下标	data	parent
0	R	−1
1	Q	0
2	M	0
3	P	0
4	F	1
5	G	3
6	H	3
7	I	3

(b) 双亲表示法示意图

图5.6 树及其双亲表示法示意图

需要特别强调的是，双亲表示法虽然使用一维数组来存储树的逻辑结构，但数据元素的位置(数组下标)并没有直接反映数据元素之间的关系，因此并不是一种顺序存储结构。从本质上讲，双亲表示法是一种静态链表结构。

2. 孩子链表表示法

在树的孩子表示法中，也是用一维数组存储所有数据元素，每个数组元素都是一个结点结构，也称为表头结点，包含数据域和指针域两部分，其中数据域存储数据元素，指针域存储数据元素之间的关系，即存储该结点的所有孩子结点信息。由于一个结点可能有多个孩子，因此用一个单链表结构进行组织，也就是说，每个结点都对应一个孩子结点链表。表头结点的指针域指向孩子结点链表中的第一个孩子。

综上所述，在孩子链表表示法中，包含两个基本结构：一是表头结点数组，存储数据元素信息；二是孩子链表，为单链表结构。表头结点结构和孩子链表结点结构分别如图5.7(a)和图5.7(b)所示。

(a) 表头结点结构　　(b) 孩子链表结点结构

图5.7 孩子链表表示法

这两种结点结构分别用 C++语言中的结构体实现,具体描述见代码 5.2。

代码 5.2 孩子表示法中结点结构的定义

```
struct ChildNode {                    //孩子链表结点
    int child;                        //数据域,孩子信息,表头数组下标
    ChildNode * next;                 //指针域,下一个孩子结点的指针
};
template < class Element >
struct HeadNode {                     //表头结点
    Element data;                     //数据域,存储数据元素
    ChildNode * firstChild;           //指针域,指向孩子结点链表的头指针
};
```

图 5.6(a)所示树结构的孩子链表表示法的存储结构如图 5.8 所示。数据元素在数组中的位置没有统一规定,本示例中是按层序遍历的顺序进行存储的。

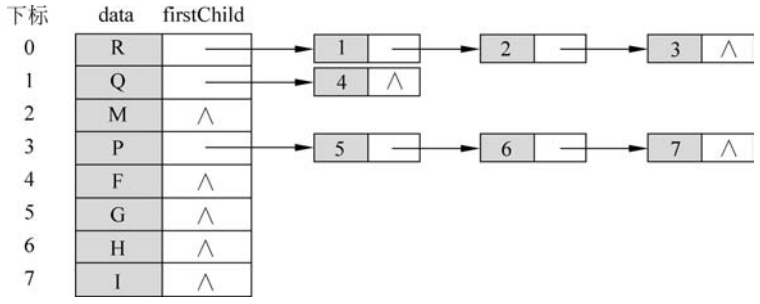


图 5.8 树的孩子链表表示法示意图

与双亲表示法相反,在孩子链表表示法中,查找孩子信息比较方便,查找双亲信息比较麻烦,必须要对整个存储结构进行遍历才能完成。

3. 双亲孩子表示法

双亲孩子表示法是融合了双亲表示法和孩子链表表示法所有结构要素的一种存储方法。该方法在孩子链表表示法的基础上,将表头结点进行改进,增加了新的指针域 parent,用以存储双亲结点的信息。改造后的表头结点结构如图 5.9 所示。

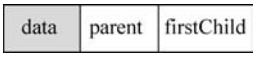


图 5.9 双亲孩子表示法的表头结点结构

图 5.6(a)所示树结构的双亲孩子表示法的存储结构如图 5.10 所示。

可以看出,在双亲孩子表示法中,既可以方便地查找一个结点的双亲结点,又可以方便地查找孩子结点的信息,兼具双亲表示法和孩子链表表示法的优点。

4. 孩子兄弟表示法

树的孩子兄弟表示法(children brother express)又称为二叉链表表示法,基本思想是从树的根结点开始,依次用链表存储各个结点的长子结点和右兄弟结点。链表中的结点除数据域外,还设置了两个指针域分别指向该结点的第一个孩子和右边第一个兄弟,如图 5.11 所示。其中,data 为数据域,存储结点的数据信息; firstChild 为第一个指针域,指向结点的第一个孩子结点; rightBrother 为第二个指针域,指向该结点右边的第一个兄弟。

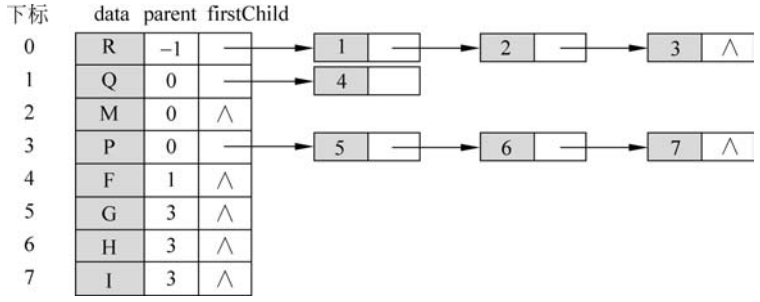


图 5.10　树的双亲孩子表示法示意图

图 5.11　孩子兄弟表示法结点结构

可以用 C++语言中的结构体实现该结点结构，具体描述见代码 5.3。

代码 5.3　孩子兄弟表示法中结点结构的定义

```
template < class Element >
struct TreeNode {
    Element data;                    //数据域,存储数据元素
    TreeNode * firstChild;           //第一个指针域,指向长子
    TreeNode * rightBrother;         //第二个指针域,指向右兄弟
};
```

图 5.6(a)所示树结构的孩子兄弟表示法的存储结构如图 5.12 所示。

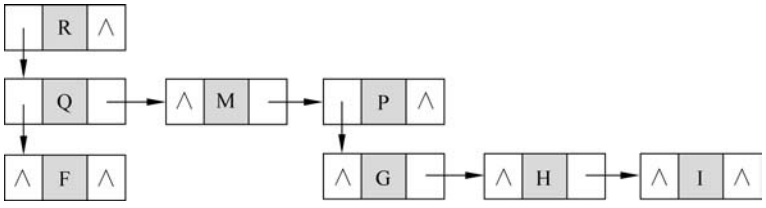


图 5.12　树的孩子兄弟表示法示意图

孩子兄弟表示法可以方便地查找某个结点的孩子信息，即 firstChild 指向的是其第一个孩子，而第一个孩子的 rightBrother 指向其第二个孩子，第二个孩子的 rightBrother 指向其第三个孩子，依此类推直到 rightBrother 是 nullptr 为止。但这种表示方法同样不方便查找结点的双亲信息。

5.3　二叉树的逻辑结构

二叉树结构简单，存储效率高，基本运算也较为简单。在实际生活中所用到的树的问题，如树存储和操作的实现，一般较为复杂，但都可以转化为二叉树的问题加以解决，二叉树本身在排序和查找领域也具有广泛的应用。因此，二叉树在数据结构学习中具有极其重要

的地位。

5.3.1 二叉树的基本概念

1. 二叉树

二叉树(binary tree)是有限个结点的集合,该集合或者为空集(称为空二叉树),或者由一个根结点和两棵互不相交的子树(分别称为左子树和右子树)组成。

和树的定义一样,二叉树的定义也是递归的。与5.2节讲述的树结构相比,二叉树具有如下特点:

(1) 每个结点最多有两棵子树;

(2) 两棵子树是有序的,不能任意颠倒;

(3) 即使结点只有一棵子树,也有左右之分。

由此可见,二叉树是另一种树形结构,不是树的子集,与度为2的树的区别主要体现在以下两点:

(1) 度为2的树至少有一个结点的度为2,而二叉树没有这个要求;

(2) 度为2的树中的结点如果只有一棵子树,是不区分左右的,而二叉树需要严格区分左右。

二叉树具有5种基本形态:①空二叉树;②只有根结点的二叉树;③根结点只有左子树;④根结点只有右子树;⑤根结点既有左子树又有右子树。任何复杂的二叉树都可以看成是这些形态的组合。具有3个结点的树和二叉树,其形态上是不同的,树有2种形态,二叉树有5种形态,具体如图5.13所示,这也说明了树和二叉树是两种完全不同的树形结构。

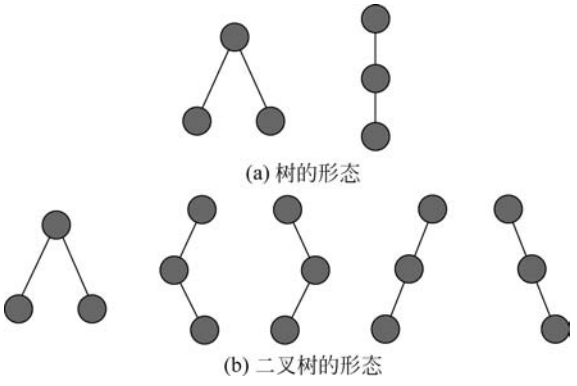


图5.13 3个结点的树和二叉树的形态

2. 特殊形态的二叉树

在一棵二叉树中,如果所有分支结点都有左孩子和右孩子,并且叶子结点都在二叉树的最下一层,这样的二叉树称为满二叉树(full binary tree)。也可以从结点总数和树高之间的关系的角度来定义,即一棵高度为h且有2^h-1个结点的二叉树称为满二叉树。图5.14(a)所示为高度为4的满二叉树。满二叉树的特点是:①叶子只能出现在最下一层;②只有度为0和度为2的结点;③每一层上的结点数都是最大结点数。

可以对满二叉树进行层序编号,即从根结点为1开始,按照层数从上到下,同一层从左到右的次序依次对结点进行编号,如图5.14(a)所示。深度为k的,有n个结点的二叉树,

当且仅当其每一个结点都与深度为 k 的满二叉树中编号从 1～n 的结点一一对应，称之为完全二叉树(complete binary tree)。显然，满二叉树是完全二叉树的一种特殊形式。完全二叉树的特点是：①叶子结点只能出现在最下两层且最下层的叶子结点都集中在二叉树的左边；②完全二叉树中如果有度为 1 的结点，只可能有一个，且该结点只有左孩子；③深度为 k 的完全二叉树在前 $k-1$ 层上一定是满二叉树；④在同样结点个数的二叉树中，完全二叉树的深度最小。图 5.14(b)是完全二叉树示例。

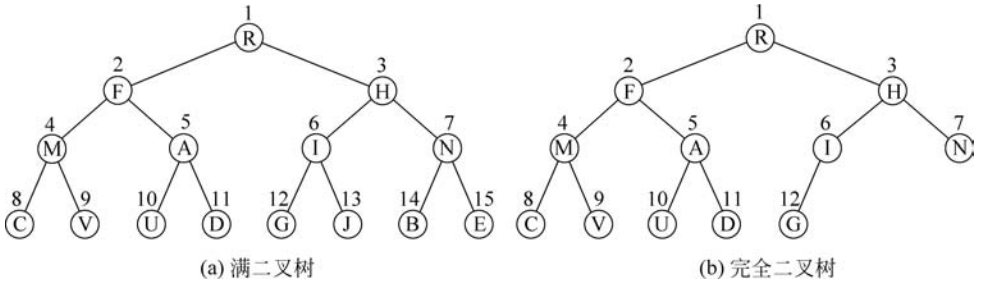


图 5.14　满二叉树和完全二叉树

5.3.2　二叉树的基本性质

性质 1　二叉树的第 i 层上最多有 2^{i-1} 个结点($i \geqslant 1$)。

证明：采用归纳法证明。

当 $i=1$ 时，第 1 层只有一个根结点，而 $2^{i-1}=2^0=1$，结论显然成立。

假定 $i=k(k \geqslant 1)$时结论成立，即第 k 层上最多有 2^{k-1} 个结点，则当 $i=k+1$ 时，因为第 $k+1$ 层上的结点是第 k 层上结点的孩子，而二叉树中每个结点最多有 2 个孩子，故在第 $k+1$ 层上最大结点个数为第 k 层上的最大结点个数的 2 倍，即 $2\times 2^{k-1}=2^k$。结论成立。

性质 2　深度为 k 的二叉树最多有 2^k-1 个结点。

证明：由性质 1，深度为 k 的二叉树中结点个数最多为

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

性质 3　在一棵二叉树中，如果叶子结点数为 n_0，度为 2 的结点数为 n_2，则有 $n_0=n_2+1$。

证明：设 n 为二叉树的结点总数，n_0、n_1、n_2 分别为二叉树中度为 0、度为 1 和度为 2 的结点数，则有

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

在二叉树中，除了根结点外，其余每个结点都有唯一的双亲，对应一条边；同时，一个度为 1 的结点只有一个孩子，对应一条边，一个度为 2 的结点具有两个孩子，对应两条边，所以有

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

联立式(5.1)和式(5.2)可得 $n_0=n_2+1$。

满二叉树中没有度为 1 的结点，只有度为 0 的叶子结点和度为 2 的分支结点，所以有 $n=n_0+n_2$，$n_0=n_2+1$，叶子结点的个数为 $n_0=(n+1)/2$。

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

证明：假设具有 n 个结点的完全二叉树的深度为 k，根据完全二叉树的定义和性质 2 可知，有

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

对不等式取对数，有

$$k-1 \leqslant \log_2 n < k$$

即

$$\log_2 n < k \leqslant \log_2 n + 1$$

由于 k 是整数，故必有 $k = \lfloor \log_2 n \rfloor + 1$。

性质 5 对一棵具有 n 个结点的完全二叉树中从 1 开始按层序编号，则对于任意序号为 $i(1 \leqslant i \leqslant n)$的结点(简称为结点 i)，有以下结论：

(1) 如果 $i>1$，则结点 i 的双亲结点的序号为 $\lfloor i/2 \rfloor$；如果 $i=1$，则结点 i 是根结点，无双亲结点。

(2) 如果 $2i \leqslant n$，则结点 i 的左孩子的序号为 $2i$；如果 $2i>n$，则结点 i 无左孩子。

(3) 如果 $2i+1 \leqslant n$，则结点 i 的右孩子的序号为 $2i+1$；如果 $2i+1>n$，则结点 i 无右孩子。

上述结论可采用归纳法证明，请读者自己完成。

性质 5 表明，在完全二叉树中，结点的层序编号反映了结点之间的逻辑关系。

5.3.3 二叉树的抽象数据类型

和树类似，在不同的实际应用中，二叉树的基本操作不尽相同。简单起见，基本操作主要包含树的遍历和构造，一个二叉树的抽象数据类型定义如下：

```
ADT BiTree
{
    数据对象:D = {a_i | a_i ∈ ElemSet, i = 1,2,…,n, n >= 0, a_i 可以为任意数据}
    数据关系:R = {< a_(i-1), a_i > | a_(i-1), a_i ∈ D, i = 2,…,n, a_(i-1) 和 a_i 之间的关系满足二叉树的定
                义}
    基本操作:
        initBiTree:初始化操作,建立一棵结点数目为 0 的二叉树。
        destroyBiTree:销毁操作,销毁一棵二叉树,释放该二叉树占用的存储空间。
        preBiOrder:前序遍历操作,在不改变二叉树结构的情况下,前序遍历二叉树,输出遍历
                   序列。
        inBiOrder:中序遍历操作,在不改变二叉树结构的情况下,中序遍历二叉树,输出遍历
                  序列。
        postBiOrder:后序遍历操作,在不改变二叉树结构的情况下,后序遍历二叉树,输出遍历
                    序列。
        levelBiOrder:层序遍历操作,在不改变二叉树结构的情况下,层序遍历二叉树,输出遍历
                     序列。
}
```

5.3.4 二叉树的遍历

二叉树的遍历是指从根结点出发，按照某种次序访问二叉树中的所有结点，使得每个结

点被访问一次且仅被访问一次。这里的访问是抽象操作，可以是对结点进行的各种处理，可以简化为输出结点的数据。在二叉树遍历过程中，访问次序有前序、中序、后序和层序。与树的遍历类似，二叉树遍历的结果，就是把逻辑上的非线性结构转化为访问时间上的线性序列。

从组成上看，二叉树包含根结点 D、左子树 L 和右子树 R 三部分。二叉树的遍历方式有 DLR、LDR、LRD、DRL、RDL 和 RLD 共 6 种，前 3 种遍历方式和后 3 种遍历方式是对称操作，可以限定在子树的访问次序上，先左子树 L 后右子树 R，从而得到 DLR、LDR 和 LRD 3 种遍历方式，分别称为前序(根)遍历、中序(根)遍历和后序(根)遍历。

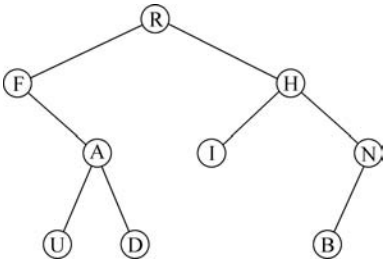


图 5.15　一棵二叉树

图 5.15 是一棵二叉树，下面给出二叉树的前序遍历、中序遍历、后序遍历和层序遍历过程。

1. 前序(根)遍历

若二叉树为空，则空操作返回；否则：

(1) 访问根结点；

(2) 前序遍历根结点的左子树；

(3) 前序遍历根结点的右子树。

如图 5.15 所示的二叉树，其前序遍历的结点序列为 RFAUDHINB。

2. 中序(根)遍历

若二叉树为空，则空操作返回；否则：

(1) 中序遍历根结点的左子树；

(2) 访问根结点；

(3) 中序遍历根结点的右子树。

如图 5.15 所示的二叉树，其中序遍历的结点序列为 FUADRIHBN。

3. 后序(根)遍历

若二叉树为空，则空操作返回；否则：

(1) 后序遍历根结点的左子树；

(2) 后序遍历根结点的右子树；

(3) 访问根结点。

如图 5.15 所示的二叉树，其后序遍历的结点序列为 UDAFIBNHR。

通过上述 3 种遍历的定义，容易看出它们具有以下几个共同点：

(1) 位于二叉树中同一子树上的结点，在遍历后的序列中是相邻的。

(2) 左子树的结点先于右子树上的结点被访问。

(3) 所有叶子结点在遍历序列中的相对次序相同，都是按照在二叉树中从左到右的顺序依次被访问。如在图 5.15 中，二叉树中叶子结点从左至右的顺序是 U、D、I、B，与它们在 3 种遍历序列中的相对次序相同。

(4) 已知一棵二叉树，其前序、中序和后序遍历序列都是唯一的；但不同的二叉树有可能产生相同的遍历序列，因此由前序、中序和后序遍历序列都不能唯一确定一棵二叉树。

4. 层序遍历

二叉树的层序遍历是指从二叉树的第一层(即根结点)开始,从上至下逐层遍历,在同一层中,则按从左到右的顺序对结点逐个访问,其访问次序与层序编号相同。

如图 5.15 所示的二叉树,其层序遍历的结点序列为 RFHAINUDB。

5.3.5 二叉树的构造

二叉树可以进行前序、中序、后序或者层序遍历,得到唯一的遍历序列,那么反过来会成立吗? 也就是说,给定一个遍历序列,能唯一确定一棵二叉树吗?

如给定一个前序遍历序列 ABC,其对应的二叉树有 5 棵,如图 5.16 所示。给定一个中序遍历序列或者后序遍历序列,也存在类似情况,即单独给出一个遍历序列,无法唯一确定一棵二叉树。为了能唯一构造一棵二叉树,可以采取两类方法:第一类是采用扩展二叉树,第二类是采用两个不同的遍历序列。

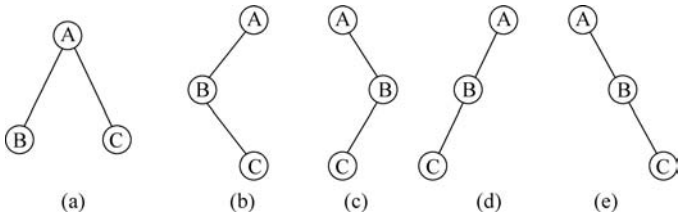


图 5.16 前序序列 ABC 对应的树

1. 由扩展二叉树构造二叉树

对于二叉树中的每个结点,如果没有左孩子,则扩充一个虚拟结点作为左孩子;如果没有右孩子,则扩充一个虚拟结点作为右孩子;虚拟结点统一用特定值(如"#")填充,标识为空。这样处理后的二叉树称为扩展二叉树(extended binary tree)。图 5.15 二叉树所对应的扩展二叉树如图 5.17 所示。

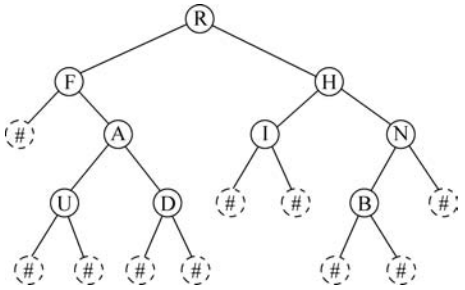


图 5.17 扩展二叉树示例

根据图 5.17 的扩展二叉树,可以得到其前序遍历序列为 RF#AU##D##HI##NB###,后序遍历序列为###U##DAF##I##B#NHR。

利用扩展二叉树的前序遍历序列或者后序遍历序列,能够唯一确定这棵扩展二叉树,进而构造其所对应的二叉树。

前序遍历序列确定二叉树的过程如下：

(1) 将序列中的每个字母和＃都看作是一个独立结点，从第 1 个字符开始从左至右依次扫描前序遍历序列中的各个字符，空右子树位置从 1 开始编号，依次增大。

(2) 如果字符不是＃，构造二叉树结点，值为当前字符，设置右子树位置为空并编号，然后转移到该结点的左子树位置，继续扫描序列中的下一个字符；如果字符是＃，跳转到步骤(3)，否则跳转到步骤(2)。

(3) 构造二叉树虚拟结点，并转移到编号最大的空右子树位置，继续扫描序列中的下一个字符，如果没有右子树为空的位置，构造结束，跳转到步骤(4)。

(4) 扩展二叉树构造完毕。

(5) 将扩展二叉树中的虚拟结点删除，即为二叉树。

已知扩展二叉树的前序遍历序列为 RF＃AU＃＃＃H＃I＃＃，则构造对应二叉树的过程如图 5.18 所示，图 5.18(l)为最终生成的二叉树。

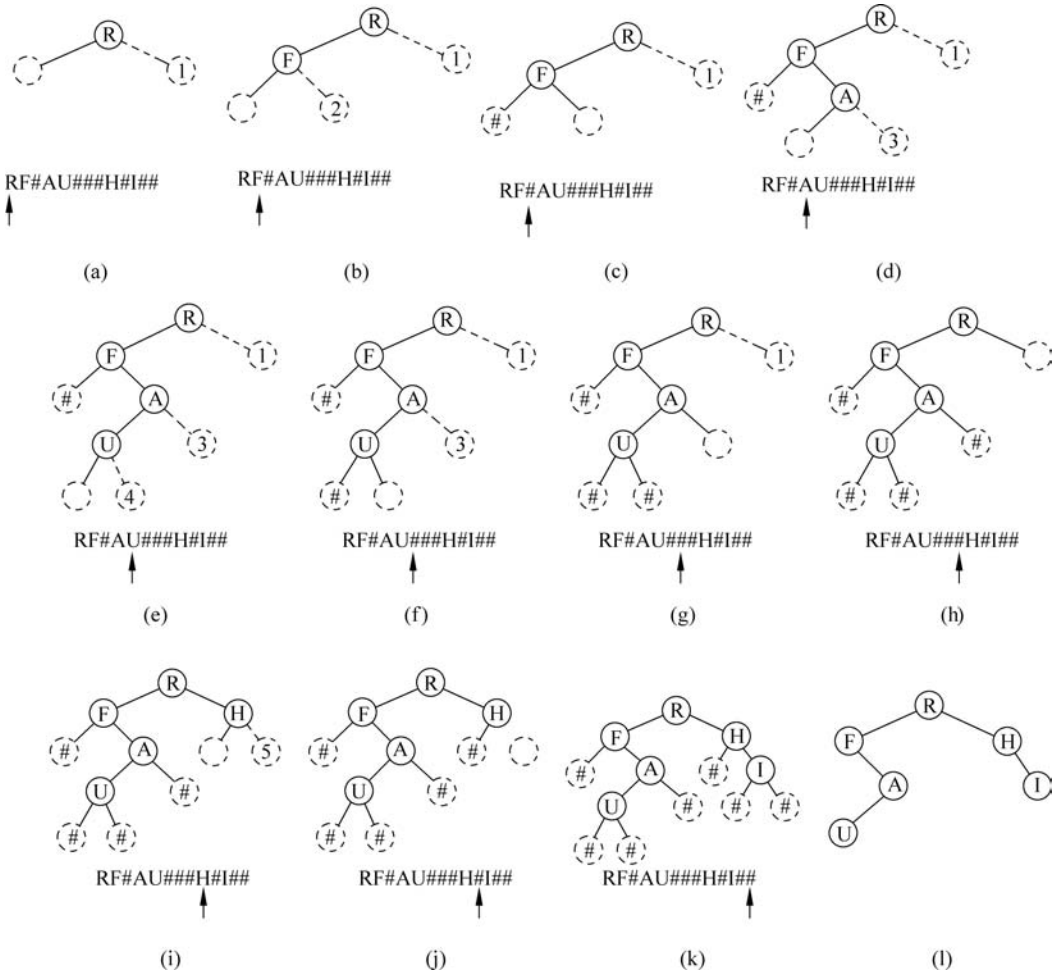


图 5.18　由扩展二叉树前序遍历序列构造二叉树

利用扩展二叉树的后序遍历序列也可以唯一确定一棵二叉树，其过程与前序序列确定二叉树的过程类似，不再给出具体过程，读者可以自行完成。

2. 由两种遍历序列构造二叉树

关于由两种遍历序列构造二叉树,有以下两个结论:

(1) 任何 $n(n\geqslant 0)$个不同结点的二叉树,都可由它的中序遍历序列和前序遍历序列唯一确定。

(2) 任何 $n(n\geqslant 0)$个不同结点的二叉树,都可由它的中序遍历序列和后序遍历序列唯一确定。

由于篇幅关系,本书不给出这两个结论的证明,感兴趣的读者可以自行完成。

由中序遍历序列和前序遍历序列确定二叉树的具体过程如下:

(1) 找到前序遍历序列中的第一个结点,即确定了当前二叉树的根结点;

(2) 在对应的中序遍历序列中,找到根结点,则根结点将中序序列划分为左右两部分,即得到该二叉树左右子树的中序遍历序列;

(3) 在二叉树的前序遍历序列中,去掉根结点,确定左右子树的前序遍历序列;

(4) 如果左子树非空则跳转到步骤(1),如果右子树非空则跳转到步骤(1);

(5) 完成当前二叉树的构造。

由此可见,前序遍历序列的作用是确定根结点,中序遍历序列的作用是确定左右子树,不断递归即可构造完成二叉树。例如,已知一棵二叉树的前序遍历序列是 ABCDEF,中序遍历序列是 CBAEDF,则构造这棵二叉树的过程如下:

(1) 由前序遍历序列 ABCDEF 可知,二叉树根结点为 A,结合中序遍历序列,可以确定左子树包含结点 C、B,右子树包含结点 E、D、F,如图 5.19(a)所示。

(2) 对比前序遍历序列和中序遍历序列,可以确定左子树的前序遍历序列为 BC,中序遍历序列为 CB; 右子树的前序遍历序列为 DEF,中序遍历序列为 EDF。

(3) 对于左子树,可知其根结点为 B,左子树为 C,右子树为空,如图 5.19(b)所示。

(4) 对于右子树,可知其根结点为 D,左子树为 E,右子树为 F,如图 5.19(c)所示。

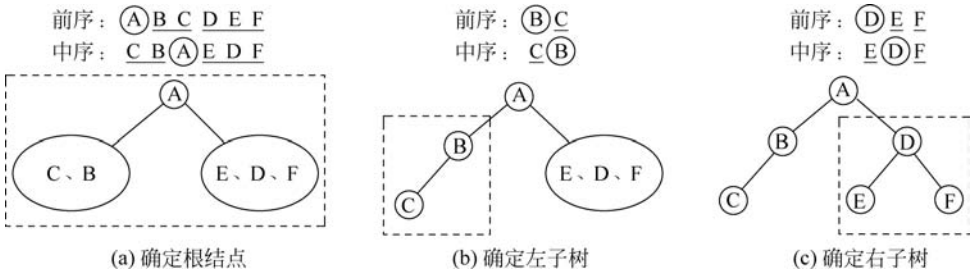


图 5.19 由前序遍历序列和中序遍历序列构造二叉树

由后序遍历序列和中序遍历序列也可以构造二叉树,后序遍历序列的作用是确定根结点,中序遍历序列的作用是确定左右子树,具体的构造过程与由前序遍历序列和中序遍历序列构造二叉树的过程类似,不再具体分析。

5.4 二叉树的存储结构

二叉树的存储结构不仅要存储数据元素本身,还要存储数据元素之间的关系。在二叉树中,数据元素之间的关系不仅体现为双亲关系和孩子关系,孩子之间还要体现出是左孩子

还是右孩子。在二叉树中，这种关系可以用顺序存储结构或链式存储结构进行存储。

5.4.1 顺序存储结构

由二叉树的性质5可以看出，完全二叉树的层序编号能够反映结点（数据元素）之间的双亲和孩子关系，由此可以设计出完全二叉树的顺序存储结构。将完全二叉树中的数据元素按照层序编号顺序，即层次按照从上到下，同一层按照从左到右的顺序依次存储在一维数组中。在C++语言中，数组下标是从0开始的，为了处理方便，不使用0号元素，从下标1开始存储数据元素，从而使得数据元素的层序编号和其所存储的位置即数组下标完全对应。如图5.14(b)所示的完全二叉树，其对应的存储结构如图5.20所示。其中，0代表数组元素未使用，没有该结点。可以看出，给定任意结点，都可以根据二叉树的性质5方便地查找其双亲和孩子结点，例如结点A，存储在下标为5的数组元素中，其双亲结点存储在下标为$\lfloor 5/2 \rfloor=2$的数组元素中，即F结点，其左孩子存储在10号位置的U结点，其右孩子是存储在11号位置的D结点。

下标	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…	MaxSize
元素值	0	R	F	H	M	A	I	N	C	V	U	D	G	0	0	…	0

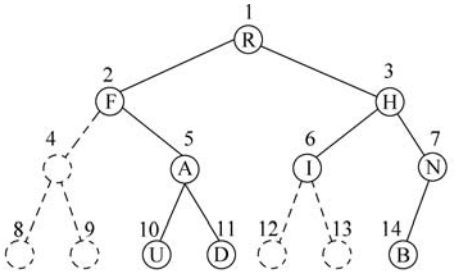
图5.20 完全二叉树的顺序存储结构

对于普通的二叉树来说，如果直接采用层序编号顺序把结点依次存储到一维数组中，如图5.15所示的二叉树，其对应的层序存储结果如图5.21所示，可以看出其存储位置并不能反映结点之间的关系。

下标	0	1	3	3	4	5	6	7	8	9	10	11	…	MaxSize
元素值	0	R	F	H	A	I	N	U	D	B	0	0	…	0

图5.21 普通二叉树结点存储到一维数组中

对于普通的二叉树，为了让数组的下标能够反映结点之间的逻辑关系，必须扩充成完全二叉树的形式，重新按层次编号后再进行存储。对于图5.15所示的二叉树，其扩充后的二叉树如图5.22(a)所示，对应的顺序存储结构如图5.22(b)所示。其中，新扩充的虚拟结点也占用编号所在的数组单元，但结点值为0。



(a) 普通二叉树扩充为完全二叉树

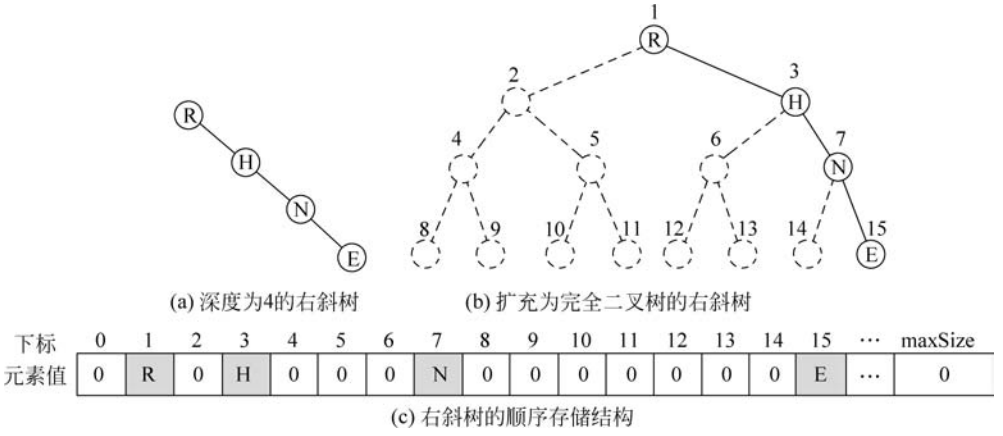
下标	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	…	MaxSize
元素值	0	R	F	H	0	A	I	N	0	0	U	D	0	0	B	0	…	0

(b) 普通二叉树的顺序存储结构

图5.22 普通二叉树扩充及其顺序存储结构

很显然,对于普通二叉树来说,存储扩充的虚拟结点存在存储空间的浪费,对右斜树(每个结点只存在右子树的二叉树)来说更是如此。如图 5.23 所示是深度为 4 的右斜树的顺序存储结构。

因此,一般情况下只有完全二叉树才考虑使用顺序存储结构。对于普通的二叉树,由于存在大量空间浪费,并且在进行结点插入和删除时,可能会涉及大量数据移动操作,因此,一般情况下二叉树的计算机实现更多是使用链式存储结构。



下标	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	…	maxSize
元素值	0	R	0	H	0	0	0	N	0	0	0	0	0	0	0	E	…	0

图 5.23　右斜树处理和对应的顺序存储结构

5.4.2　链式存储结构

二叉树最为常用的链式存储结构是二叉链表(binary linked list)。顾名思义,二叉链表是一种链式存储,其基本思想是:令二叉树的每个结点对应一个链表结点,链表结点除了存放与二叉树结点有关的数据信息外,还要设置指示左右孩子的指针。二叉链表结点包含三部分,具体如图 5.24 所示。

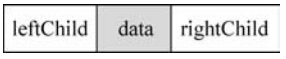


图 5.24　二叉链表的结点结构

其中,data 是数据域,可为任意类型;leftChild 是第一个指针域,指向结点的左孩子;rightChild 是第二个指针域,指向结点的右孩子。

二叉链表结点可用 C++语言描述,具体定义见代码 5.4。

代码 5.4　二叉链表中结点结构的定义

```
template < class Element >
struct BiNode {
        Element data;          //数据域,存储数据元素
        BiNode * leftChild;  //第一个指针域,指向结点的左孩子
        BiNode * rightChild; //第二个指针域,指向结点的右孩子
};
```

图 5.25 给出了图 5.15 所示二叉树的二叉链表结构示意图。

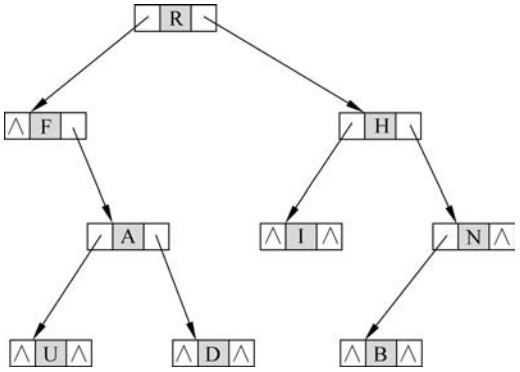


图 5.25　二叉树的二叉链表示意图

在二叉链表中，可以方便地查找结点的孩子信息。虽然在遍历二叉树的过程中能够查到其双亲信息，但时间复杂度比较高。为了解决这个问题，可以在二叉链表结点结构的基础上增加一个指针域，指向双亲结点，这就是二叉树的三叉链表(trident linked list)存储结构。此时，三叉链表的结点结构包含四部分，如图 5.26 所示。

其中，parent 是双亲域，为指向双亲结点的指针；其他域的含义与二叉链表结点结构相同。

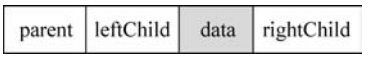


图 5.26　三叉链表结点结构

图 5.27 给出了图 5.15 所示二叉树的三叉链表结构示意图。

二叉树的逻辑结构及基本操作可以用 ADT 进行描述，二叉链表是二叉树逻辑结构的存储方式，其对应的类图如图 5.28 所示，转换成 C++语言中的类进行编程实现，具体见代码 5.5。

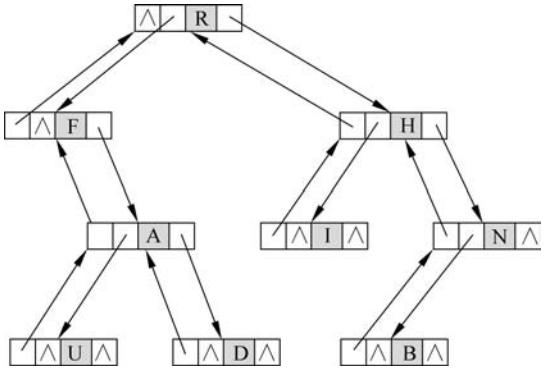


图 5.27　二叉树的三叉链表示意图

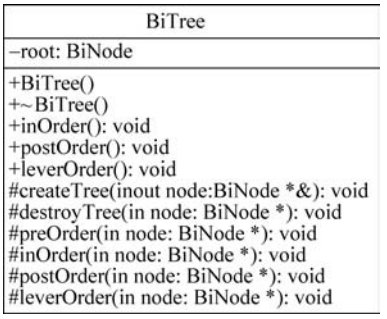


图 5.28　二叉链表的类图

代码 5.5　二叉链表的类定义

```
template < class Element >
class BiTree
{
public:
```

```
    BiTree() { createTree (root); }                    //构造函数,对应 ADT 中 initBiTree
    ~BiTree(){ destroyTree (root); }                   //析构函数,对应 destroyBiTree
    void preOrder();                                   //前序遍历,对应 preBiOrder
    void inOrder();                                    //中序遍历,对应 inBiOrder
    void postOrder();                                  //后序遍历,对应 postBiOrder
    void levelOrder();                                 //层序遍历,对应 levelBiOrder
private:
    BiNode<Element> * root;                            //指向二叉链表根结点的头指针
protected:
    void createTree(BiNode<Element> * &node);          //创建二叉树
    void destroyTree(BiNode<Element> * node);          //销毁二叉树
    void preOrder(BiNode<Element> * node);             //前序遍历 node 的子树
    void inOrder(BiNode<Element> * node);              //中序遍历 node 的子树
    void postOrder(BiNode<Element> * node);            //后序遍历 node 的子树
};
```

1. 二叉树的创建

在 5.3.5 节中已经提到,通过扩展二叉树的前序序列可以唯一地构造二叉树。可以通过从键盘输入的方式将该序列依次输入,每输入一个值,就为它建立结点。该结点作为根结点,其地址通过函数的引用型参数 node 直接链接到作为实际参数的指针中。然后,分别对根的左、右子树递归地建立子树,直到输入“#”建立空子树递归结束。其活动图如图 5.29 所示,C++实现见代码 5.6。

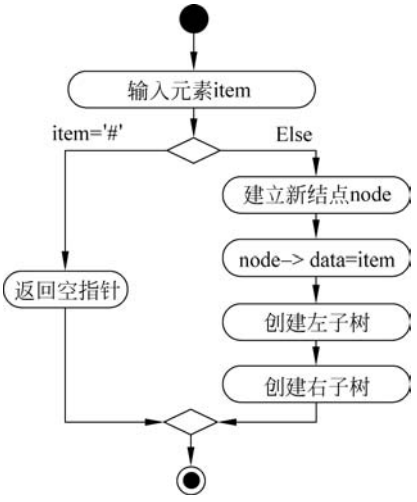


图 5.29　创建二叉树的活动图

代码 5.6　二叉树的创建

```
template<class Element>
void BiTree<Element>::createTree(BiNode<Element>* &node)
{                                                      //以引用方式传递参数,构建二叉树
    char item;
```

```
        cin >> item;
        if(item == '#'){
            node = nullptr;
        }
        else{
            node = new BiNode < Element >;              //构造结点
            node -> data = item;
            createTree(node -> leftChild);              //递归创建左子树
            createTree(node -> rightChild);             //递归创建右子树
        }
    }
```

2. 二叉树的销毁

二叉树的链式存储结构需要动态分配内存,在析构函数中必须把动态分配的内存释放掉,即删除二叉链表中的所有结点。为保证删除结点以后不断链,经常采用后序遍历操作。具体实现见代码 5.7。

代码 5.7 二叉树的销毁

```
template < class Element >
void BiTree < Element >::destroyTree(BiNode < Element > * node)
{
    if (node!= nullptr){                            //以后序遍历方式析构
        destroyTree(node -> leftChild);
        destroyTree(node -> rightChild);
        delete node;
    }
}
```

3. 二叉树前序、中序和后序遍历的递归实现

二叉树的前序、中序和后序遍历操作的定义是递归的,其递归形式的代码实现非常简单,具体实现见代码 5.8。

代码 5.8 二叉树前序、中序和后序遍历的递归实现

```
template < class Element >
void BiTree < Element >::preOrder(BiNode < Element > * node)
{                                                       //前序遍历
    if(node != nullptr){
        cout << node -> data << " ";                    //先访问根结点
        preOrder(node -> leftChild);                    //再访问左子树
        preOrder(node -> rightChild);                   //最后访问右子树
    }
}
template < class Element >
void BiTree < Element >:: inOrder (BiNode < Element > * node)
{                                                       //中序遍历
```

```
    if(node != nullptr){
      inOrder (node->leftChild);                    //先访问左子树
      cout << node->data << " ";                    //再访问根结点
      inOrder (node->rightChild);                   //最后访问右子树
    }
}
template<class Element>
void BiTree<Element>::postOrder(BiNode<Element> *node)
{                                                   //后序遍历
    if(node != nullptr){
      postOrder(node->leftChild);                   //先访问左子树
      postOrder(node->rightChild);                  //再访问右子树
      cout << node->data << " ";                    //最后访问根结点
    }
}
```

但是,由于递归遍历必须传递结点参数,而二叉树的根结点为私有,主函数中无法获取,因此需要在类的内部用重载方式实现,以前序遍历为例:定义在类内部函数这几种遍历操作的定义是递归的。二叉树递归遍历的调用过程见代码 5.9。

代码 5.9 二叉树递归遍历的调用

```
template<class Element>
void BiTree<Element>::preOrder()            //主函数中调用无参的遍历函数
{
    preOrder (root);                        //调用有参的重载函数
}
```

4. 二叉树的层序遍历

在进行层序遍历时,完成对某一层的结点访问后,再按照它们的访问次序依次访问各结点的左孩子和右孩子,先被访问的结点,其孩子的访问也要先被访问,这符合队列的操作特性。层序访问二叉树的过程需要利用一个队列,在访问二叉树的某一层结点时,把下一层结点指针预先保存在队列中,利用队列安排逐层访问的次序。因此,每当访问一个结点时,将它的子女依次加到队列的队尾,然后再访问已在队列队头的结点,从而可以实现二叉树结点的层序访问。层序遍历的活动图如图 5.30 所示,C++实现见代码 5.10。

代码 5.10 二叉树的层序遍历

```
template<class Element>
void BiTree<Element>:: levelOrder()                         //层序遍历
{
    queue<BiNode<Element>*> q;                              //定义辅助队列
    q.push(root);                                           //根结点入队
    while (!q.empty()){
        bt = q.front();
        q.pop();
        cout << bt->data << " ";                            //访问队头元素的数据
        if (bt->leftChild != nullptr){
            q. push(bt->leftChild);
        }
```

```
            if (bt->rightChild != nullptr){
                q.push(bt->rightChild);
            }
        }
    }
```

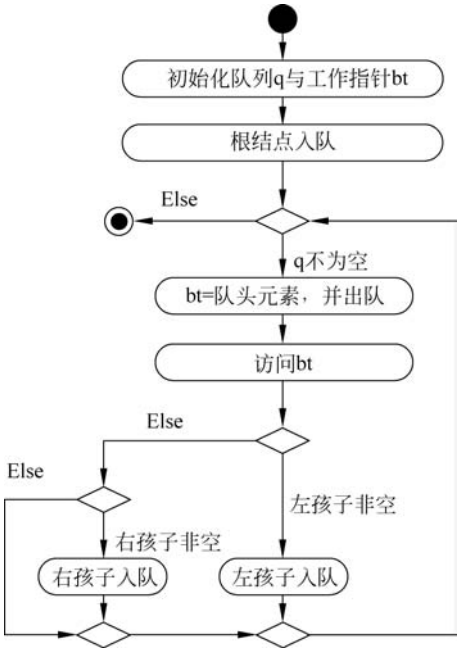


图 5.30　二叉树层序遍历的活动图

5. 二叉树前序、中序和后序遍历的非递归实现

递归在计算机中是用栈来实现的，因此递归实现可以通过一个栈转化为非递归实现。非递归实现不需要外部访问根结点，也不需要使用类似代码 5.9 的重载调用，程序结构更加简洁清晰。前序遍历的非递归实现如图 5.31 所示，C++实现见代码 5.11。

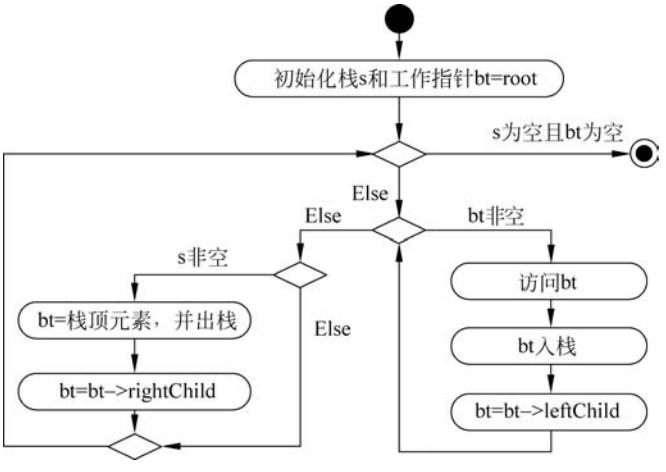


图 5.31　二叉树前序遍历的非递归算法

代码 5.11 二叉树前序遍历的非递归实现

```
template < typename Element >
void BiTree < Element >:: preOrder(){
    stack < BiNode < Element > * > s;          //定义辅助栈
    BiNode < Element > * bt = root;            //初始化工作指针
    while(bt!= nullptr || !s.empty()){         //工作指针和栈均为空时结束循环
        while(bt!= nullptr){
            cout << bt -> data;                //访问结点
            s.push(bt);
            bt = bt -> leftChild;              //遍历左子树
        }
        if(!s.empty()){                        //左子树遍历结束
            bt = s.top();
            bt = bt -> rightChild;             //取右子树继续遍历
            s.pop();
        }
    }
}
```

二叉树中序遍历的非递归实现与前序遍历相类似,区别仅在于访问结点的时机,应在左子树遍历结束后访问根结点,C++实现见代码 5.12。

代码 5.12 二叉树中序遍历的非递归实现

```
template < typename Element >
void BiTree < Element >:: inOrder(){
    stack < BiNode < Element > * > s;
    BiNode < Element > * bt = root;
    while(bt!= nullptr || !s.empty()){
        while(bt!= nullptr){
            s.push(bt);
            bt = bt -> leftChild;              //遍历左子树
        }
        if(!s.empty()){                        //左子树遍历结束
            bt = s.top();
            cout << bt -> data;                //访问结点
            bt = bt -> rightChild;             //取右子树继续遍历
            s.pop();
        }
    }
}
```

二叉树的后序遍历思路仍然相同,但访问根结点的时机是在左右子树全部遍历结束时,因此要在结点定义中添加当前结点左右子树均遍历完成的标志。左右子树均遍历结束时再执行出栈和访问的操作。C++实现见代码 5.13。

代码 5.13 二叉树后序遍历的非递归实现

```
template < typename Element >
void BiTree < Element >:: postOrder(){
    stack < BiNode < Element > * > s;
    BiNode < Element > * bt = root;
    while(bt!= nullptr || !s.empty()){
        while(bt!= nullptr){
            s.push(bt);
            bt = bt -> leftChild;                    //遍历左子树
        }
        while(!s.empty()&&s.top() -> flag == true)  //左右子树均遍历完成
        {
            bt = s.top();
            cout << bt -> data;                      //访问结点
            s.pop();                                 //出栈
        }
        if(bt == root) break;                        //如根结点出栈,则遍历结束
        if(!s.empty()){                              //仅左子树遍历结束
            bt = s.top();
            bt -> flag = true;                       //修改标志位
            bt = bt -> rightChild;                   //遍历右子树
        }
    }
}
```

5.4.3 线索链表

二叉树的遍历就是把结构上是非线性的数据结构,转换为时间上的先后关系,把数据元素之间的一对多关系转换为一对一的线性关系,即前驱后继关系。在具体应用中,如果频繁进行二叉树的遍历操作,会有较大的时间开销。在 5.4.2 节的学习中可以得知,对于 n 个结点的二叉链表来说,共有 $2n$ 个指针域,其中用来存放孩子信息的指针域只有 $n-1$ 个,剩余 $n+1$ 个指针域的值为 nullptr。能否利用这些空闲的指针域存放遍历时的前驱后继关系来加快遍历的过程,减少时间的开销呢?

答案是肯定的。具体做法就是在二叉树的遍历过程中,将空闲的左孩子域指向结点的前驱,空闲的右孩子域指向后继。但原来的空指针也是有含义的,即该位置没有左孩子或者右孩子,指向前驱或者后继后,就没有办法区别该指针域到底是指向左右孩子还是指向前驱后继。因此,必须对原有的二叉链表结点结构进行改造,增加两个标志域,分别指示对应的指针域存储的是孩子还是前驱或后继。为了区分是左右孩子指针,还是指向前驱或后继结点的指针,专门给后者取了一个新的名字——线索(thread)。使二叉链表中结点的空链域存放其前驱或后继信息的过程称为线索化,加上线索的二叉树称为线索二叉树,加上线索的二叉链表称为线索链表。线索链表的结点结构如图 5.32 所示。

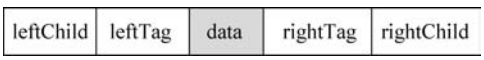


图 5.32 线索链表的结点结构

其中,左标志 leftTag 和右标志 rightTag 表示如下:

$$\text{左标志 leftTag}=\begin{cases}0 & \text{表示指向左孩子结点}\\ 1 & \text{表示指向前驱结点}\end{cases}$$

$$\text{右标志 rightTag}=\begin{cases}0 & \text{表示指向右孩子结点}\\ 1 & \text{表示指向后继结点}\end{cases}$$

线索链表的结点结构可用 C++语言描述,具体实现见代码 5.14。

代码 5.14 线索链表的结点结构

```
template < typename Element >
struct ThrNode {
    Element data;              //数据域,存储数据元素
    ThrNode * leftChild;       //左指针域
    ThrNode * rightChild;      //右指针域
    int leftTag;               //左标志:0 代表左孩子,1 代表前驱结点
    int rightTag;              //右标志:0 代表右孩子,1 代表后继结点
};
```

由于二叉树常用的遍历方式有前序、中序、后序、层序遍历,可以建立对应的线索链表。当然,由于存储空间的限制,同一时刻只能建立一种线索链表。图 5.15 所示二叉树的中序线索二叉树存储结构如图 5.33 所示。

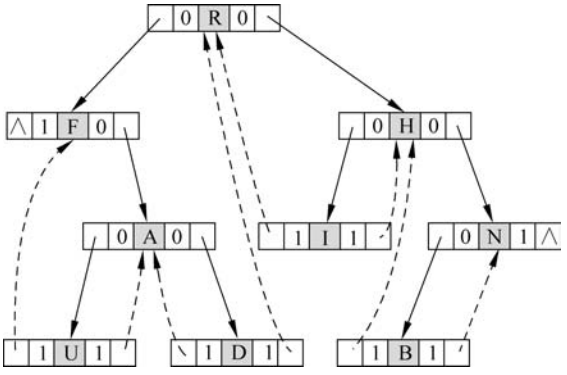


图 5.33 中序线索二叉树存储结构

下面以中序线索二叉树为例,讨论线索二叉树的定义和操作。中序线索链表的 C++类声明见代码 5.15。

代码 5.15 线索链表类定义

```
template < typename Element >
class InThrBiTree
{
public:
    inThrBiTree ();
    ~ inThrBiTree ();
```

```
    inOrder();                                                    //中序遍历
private:
    ThrNode<Element> *root;                                       //根结点指针
    void createTree(ThrNode<Element>* &node);                     //创建二叉树(未线索化)
    ThrNode<Element> * first(ThrNode<Element>* node)              //寻找起始结点
    ThrNode<Element> * next(ThrNode<Element>* node);              //查找后继
    void createInThread(ThrNode<Element>*&node, ThrNode<Element>* &pre);
                                                                  //中序线索化
};
```

1. 中序线索化过程

中序线索化的过程是在构造函数中完成的。建立线索链表分为两步,首先完成二叉链表存储结构的构造,然后在此基础上将二叉链表中的空指针域指向前驱或者后继。很显然,线索化过程必须在对二叉树中序遍历的过程中完成。

1) 使用扩展的前序遍历创建二叉树(未线索化)

与普通二叉树的构造一致,通过扩展二叉树的前序序列可以唯一地构造二叉树,并将结点的 leftTag 和 rightTag 置 0,具体实现请读者自行完成。

2) 中序遍历对二叉树进行线索化

对一个已存在的二叉树按中序遍历进行线索化的算法中用到了一个指针 pre,它在遍历过程中总是指向遍历指针 node 的前驱结点,即在中序遍历过程中刚刚访问过的结点。在中序遍历线索化过程中,只要遇到空指针域,就应填入前驱或后继线索,但后继线索只能等到访问到下一个结点时才能最终确定。假设当前结点指针为 node,前驱结点指针为 pre,当 node 为根结点指针时,pre 为 nullptr,则具体处理过程如下:

(1) 如果 node 为 nullptr,空操作;

(2) 访问 node 的左孩子,并对其进行线索化操作,返回前驱结点 pre;

(3) 如果左孩子 leftChild 为 nullptr,将 leftTag 置为 1,将 leftChild 指向前驱结点 pre,因为此时还不知道中序遍历的下一个结点,所以 rightChild 和 rightTag 暂不修改;

(4) 如果 pre 非 nullptr,并且其右孩子 rightChild 为 nullptr,则前驱结点 pre 的右孩子指针 rightChild 指向 node,rightTag=1;

(5) pre=node,并对 node 右孩子进行线索化操作,返回前驱结点 pre。

该过程的 C++实现见代码 5.16。

代码 5.16 中序线索化实现

```
template<typename Element>
void InThrBiTree<Element>::
        createInThread(ThrNode<T> *&node, ThrNode<Element>* &pre)
{
    if (node == nullptr) return;
    createInThread(node->leftChild, pre);              //递归的线索化左子树
    if (node->leftChild == nullptr)                    //建立前驱线索
    {
```

```
            node->leftChild = pre;
            node->leftTag = 1;
        }
        if (pre != nullptr&&pre->rightChild == nullptr)     //建立 pre 的后继线索
        {
            pre->rightChild = node;
            pre->rightTag = 1;
        }
        pre = node;                                         //更新 pre
        createInThread(node->rightChild, pre);              //递归的线索化左子树
    }
```

3) 构造函数

在构造函数中,完成整个线索链表的构建,具体过程的 C++实现见代码 5.17。

代码 5.17 线索链表的构造函数

```
template<typename Element>
InThrBiTree<Element>::InThrBiTree()
{
    createTree(root);                       //创建二叉树
    ThrNode<Element> *pre = nullptr;        //pre 初始化
    if (root != nullptr) {
        createInThread(root, pre);          //创建线索
        pre->rightTag = 1;                  //收尾处理
    }
}
```

2. 中序线索链表的基本函数

1) 查找中序遍历序列中的起始结点

二叉树中序遍历序列的第一个结点为整棵树最左下角的结点,即从 node 结点开始,沿着左链一直到叶子结点,具体实现过程见代码 5.18。

代码 5.18 查找第一个结点

```
template<typename Element>
ThrNode<Element> * InThrBiTree<Element>::first(ThrNode<Element>* node)
{
    p = node;
    while (p->leftTag == 0)             //循环找到最左下角结点
        p = p->leftChild;
    return p;
}
```

2) 查找中序遍历序列中的后继结点

二叉树中序遍历序列中某个结点的后继结点,位于其右子树的最左边,即右子树中的第一个结点,或者位于其后继线索中,具体实现过程见代码 5.19。

代码 5.19 查找后继结点

```
template < typename Element >
ThrNode < Element > * InThrBiTree < Element > ::next(ThrNode < Element > * node)
{
    p = node -> rightChild;
    if(node -> rightTag == 1){              //标志位为 1,直接返回右线索
        return p;
    }
    else return first(p);                   //标志位为 0,查找子树的起始结点
}
```

3. 中序线索链表的遍历操作

在上述基本函数的基础上,实现中序线索链表的遍历过程变得非常简单,先找到并遍历第一个结点,然后依次遍历后继结点直到线索链表结束为止。具体实现过程见代码 5.20。

代码 5.20 中序线索链表的遍历操作

```
template < typename Element >
void InThrBiTree < Element > :: inOrder()
{
    p = first(root);
    while(p!= nullptr){
        cout << p -> data <<" ";
        p = next(p);
    }
    cout << endl;
}
```

利用线索链表,还可以很容易地查找结点的前驱和进行逆中序遍历,感兴趣的读者可以参考 first、next 和 inOrder 这三个函数的实现自行完成。

5.5 树、森林和二叉树的转换

在前面的章节中,我们已经学习了树、森林和二叉树的相关内容,其中树和森林结构较为复杂,其操作相对烦琐。其实树和森林结构的存储和操作都可以转化为二叉树进行处理;同时,一棵二叉树也可以唯一对应一棵树或者森林。下面将详细讲述它们之间的关系和相互转换方法,以及它们之间遍历操作的对应关系。

5.5.1 树和二叉树的对应关系

通过前面章节所学内容可以知道,树结构可以采用孩子兄弟链表法进行存储。对于图 5.34(a)中的树,其对应的存储结构如图 5.34(b)所示,它们之间为一一对应关系,即一棵树对应唯一的孩子兄弟链表,同时一个孩子兄弟链表也只能唯一对应一棵树。同样地,图 5.34(c)中的二叉树和图 5.34(d)中的二叉链表也具有一一对应关系。不难看出,图 5.34(b)和

图 5.34(d)两种存储结构其实是完全等价的。给定一个图 5.34(b)的存储结构,没有办法确定它到底是孩子兄弟链表还是二叉链表,它既有可能是图 5.34(a)这棵树的存储结构,也有可能是图 5.34(c)这棵二叉树的存储结构。因此,树和二叉树也存在一一对应关系,树中的左右兄弟关系,对应二叉树中的双亲和右孩子关系;树中的双亲和第一个孩子的关系,对应二叉树中双亲和左孩子的关系。一般情况下,把树结构的存储和操作转换为二叉树来进行处理。

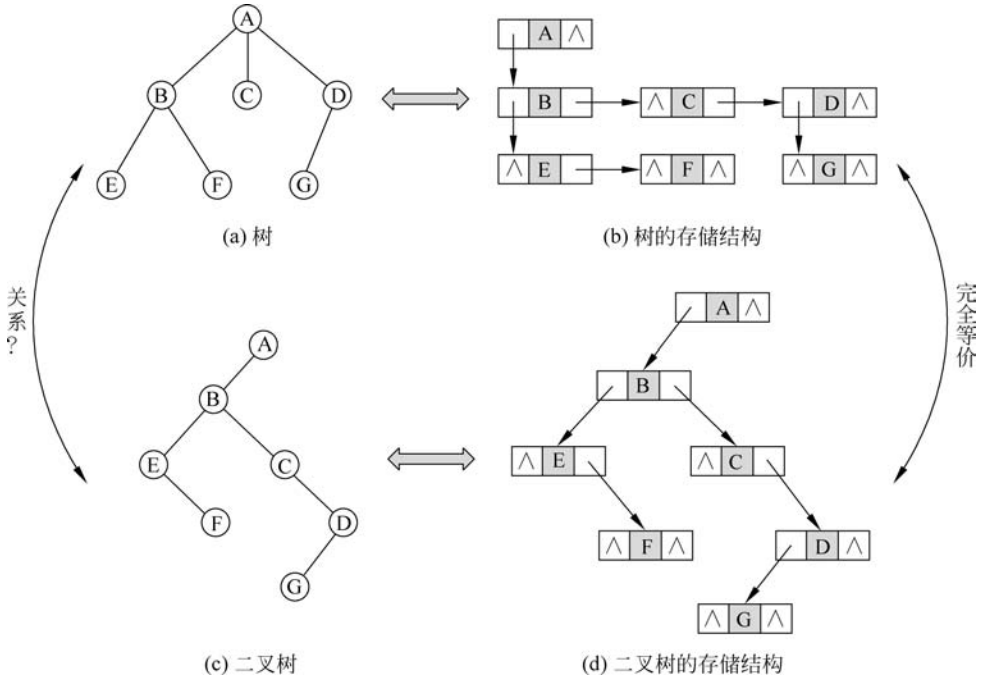


图 5.34　树和二叉树关系示意图

5.5.2　树、森林和二叉树的相互转换

1. 树转换为二叉树

将树转换为二叉树的步骤如下:

(1) 加线,在所有兄弟结点之间加一条连线;

(2) 抹线,对树中的每个结点,只保留它与第一个孩子结点之间的连线,删除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,将整棵树顺时针旋转一定角度,使之结构层次分明。

图 5.35 给出了树转换为二叉树的过程。从图中可以看出,树中结点 B、C、D 之间和 E、F 之间的兄弟关系,转换成了二叉树中相应结点之间的双亲和右孩子关系;树中结点 A、B、E 之间双亲和第一个孩子的关系,对应着二叉树中相应结点之间双亲和左孩子的关系。

2. 森林转换为二叉树

森林是由若干不相交的树组成,可以将森林中每棵树的根结点看作兄弟,由于每棵树都可以转换为二叉树,所以森林也可以转换为一棵二叉树。将森林转换为二叉树的步骤如下:

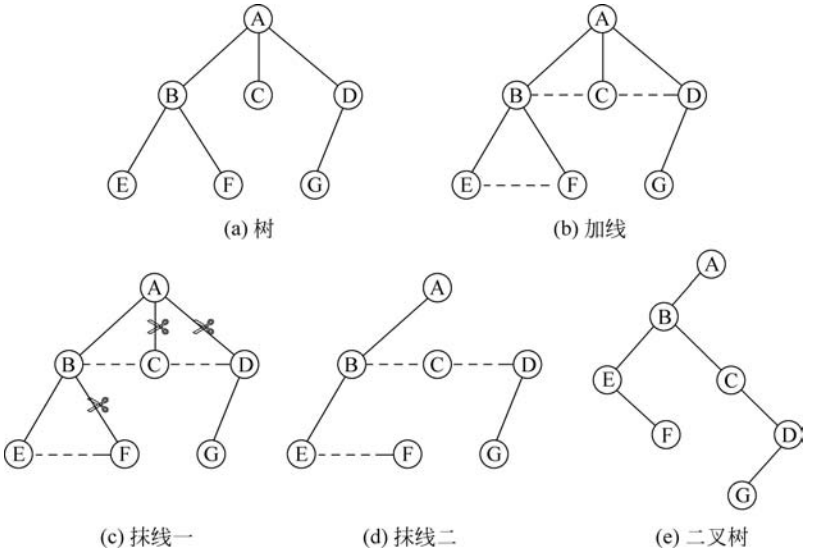


图 5.35　树转换为二叉树

(1) 先把每棵树转换为二叉树；

(2) 第一棵二叉树不动，从第二棵二叉树开始，依次把后一棵二叉树的根结点作为前一棵二叉树的根结点的右孩子结点，用线连接起来。当所有的二叉树连接起来后得到的二叉树就是由森林转换得到的二叉树。

图 5.36 给出了森林转换为二叉树的过程。可以看出，森林是由若干树构成的，但可以转换成一棵二叉树。树转换过来的二叉树根结点只有左子树没有右子树，而森林转换过来的二叉树根结点是有右子树的，并且二叉树根结点对应的是森林中第一棵树的根结点。

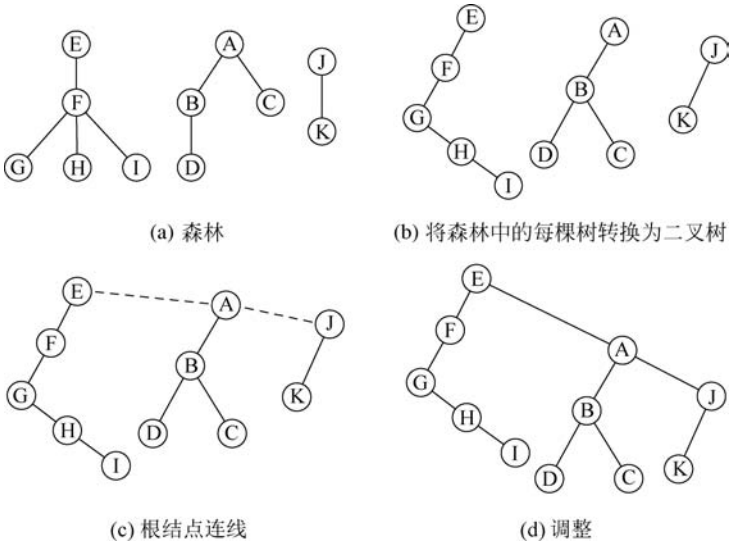


图 5.36　森林转换为二叉树

3. 二叉树转换为树或森林

二叉树转换为树或森林是树或森林转换为二叉树的逆过程,其步骤如下:

(1) 加线,若某结点的左孩子结点存在,将左孩子结点的右孩子结点、右孩子结点的右孩子结点等都作为该结点的孩子结点,将该结点与这些右孩子结点用线连接起来;

(2) 抹线,删除原二叉树中所有结点与其右孩子结点的连线;

(3) 调整,整理步骤(1)和步骤(2)两步得到的树或森林,使之结构层次分明。

图 5.37 给出了二叉树转换为森林的过程。二叉树转换为树和转换为森林的过程是完全一致的,如果二叉树根结点有右子树,则转换为森林,否则转换为树。

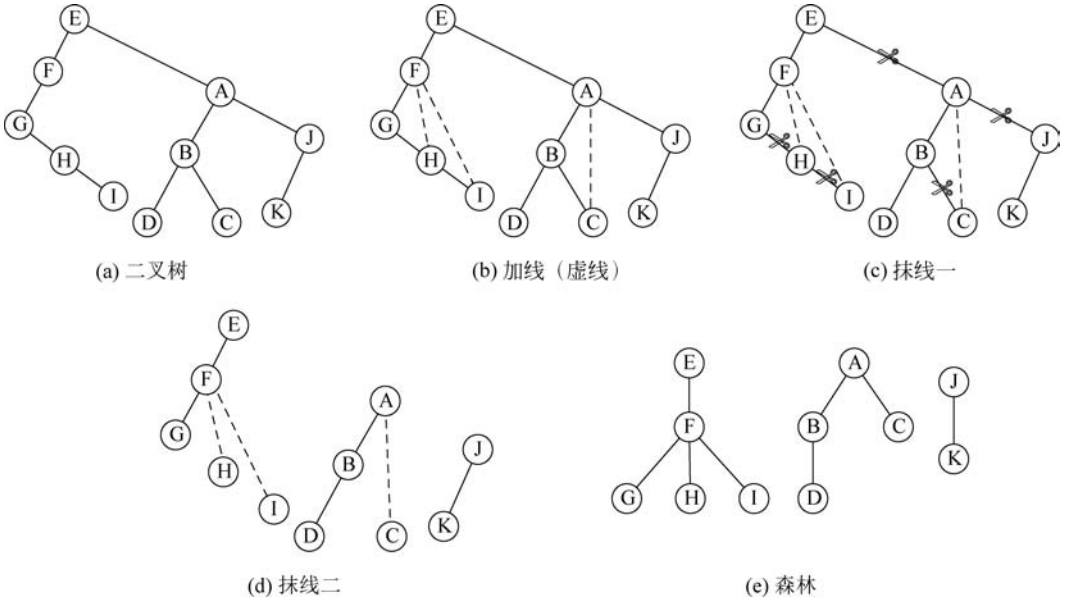


图 5.37 二叉树转换为森林

5.5.3 树、森林和二叉树遍历操作的关系

树、森林和二叉树之间不仅在结构上存在一一对应关系,它们在遍历操作上也存在对应关系。任何一种数据结构,都必须定义相应的遍历方法。在 5.2.2 节和 5.3.4 节中已经讲述了树和二叉树遍历操作的相关内容,下面简单介绍森林的遍历。森林是由若干互不相交的树组成的集合,树是组成森林的基本单位,森林的遍历操作是建立在树的遍历操作的基础上的。在本节的讨论中,可以把树看成是一种特殊的森林,即只有一棵树的森林,针对森林讨论所得到的结论完全适用于树。

1. 森林的前序遍历

森林的前序遍历过程定义为从左至右依次前序遍历各棵子树。如果把森林看成由第一棵树根结点、第一棵树子树森林和除第一棵树之外的兄弟森林三部分组成,那么森林的前序遍历也可以定义为先遍历第一棵树根结点,后遍历第一棵树子树森林,最后遍历除第一棵树之外的兄弟森林。

2. 森林的后序遍历

森林的后序遍历过程定义为从左至右依次后序遍历各棵子树,也可以定义为先遍历第

一棵树的子树森林，后遍历第一棵树根结点，最后遍历除第一棵树之外的兄弟森林。

按照树、森林和二叉树之间的转换规则，在树和森林转换为二叉树的过程中，树中任一结点的子树森林转换为对应二叉树的左子树，该结点的兄弟森林转换为对应二叉树的右子树。森林的后序遍历过程为定义为先遍历第一棵树的子树森林，后遍历第一棵树根结点，最后遍历除第一棵树之外的兄弟森林，对应到二叉树就是先遍历左子树，后遍历根结点，最后遍历右子树，这正是二叉树中序遍历的过程。也就是说，森林和树的后序遍历序列，和其对应的二叉树的中序遍历序列是完全一致的。同理也不难得到，树和森林的前序遍历序列也完全对应于二叉树的前序遍历序列。通过如图 5.38 所示的例子，可以证实上述结论。

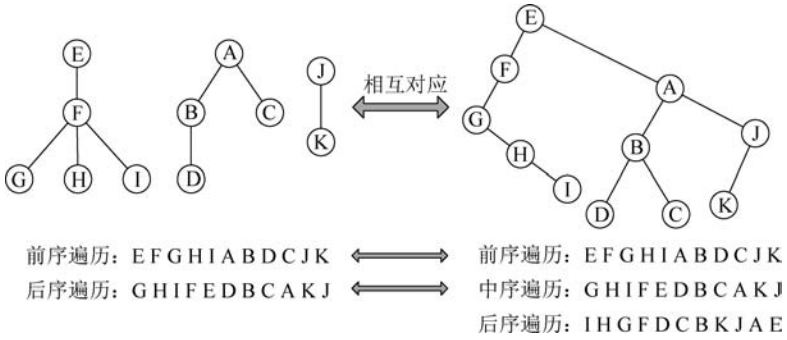


图 5.38　树、森林和二叉树遍历操作的关系

5.6　哈夫曼树和哈夫曼编码

哈夫曼树是二叉树的一个典型应用，利用哈夫曼树，可以进行哈夫曼编码和解码，进而实现对数据的压缩与解压处理。本节将在引入哈夫曼树和前缀编码的基础上，详细介绍哈夫曼编码定义、编码过程和解码过程。

5.6.1　哈夫曼树的定义

在树的许多应用中，需要将树结点赋予一个有实际含义的数值，称此数值为该结点的权。前面已经介绍过路径和路径长度概念，路径是由树中两个结点之间沿着边序列所经过的结点序列构成的，而路径长度是路径上所经过边的个数。但在有些应用中，需要用到根结点到某一结点路径长度与权值的乘积，该数值称为带权路径长度（Weighted Path Length，WPL），而树中所有叶子结点的带权路径长度之和称为该树的带权路径长度，可以表示为

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

其中，w_i 为第 i 个叶子结点的权值，l_i 为第 i 个叶子结点到根结点的路径长度。

给定 n 个权值作为 n 个叶子结点，构造一棵二叉树，若该树的带权路径长度达到最小，这样的二叉树称为哈夫曼树（Huffman tree），也称为最优二叉树。

例如，给定 4 个叶子结点，其权值分别为{1,3,4,7}，可以构造出各种形状不同或者形状相同但叶子结点分布不同的二叉树，如图 5.39 所示。图中包含 4 棵二叉树，其对应的带权路径长度分别如下。

(1) 图 5.39(a)：WPL＝1×2＋3×2＋4×2＋7×2＝30

(2) 图 5.39(b)：WPL＝1×3＋3×3＋4×2＋7×1＝27

(3) 图 5.39(c)：WPL＝1×3＋3×3＋4×2＋7×1＝27

(4) 图 5.39(d)：WPL＝1×1＋3×3＋4×2＋7×3＝39

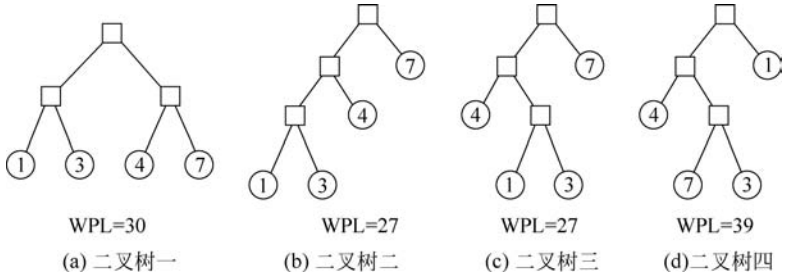


图 5.39　不同带权路径长度的二叉树

可以看出，图 5.39(b)和图 5.39(c)虽然形状不同，但带权路径长度都最小，为 27，这两棵二叉树都是哈夫曼树；图 5.39(c)和图 5.39(d)虽然形状相同，但叶子结点权值分布不同，图 5.39(d)的带权路径长度较大，为 39。容易得出，哈夫曼树具有如下几个特点：

(1) 权值越大的叶子结点越靠近根结点，而权值越小的叶子结点越远离根结点；

(2) 只有度为 0(叶子结点)和度为 2(分支结点)的结点，不存在度为 1 的结点；

(3) 同一组权值，对应的哈夫曼树不唯一。

5.6.2　哈夫曼树的构造

同一组权值，对应多个二叉树，其带权路径长度也不同。如何找到带权路径长度最小的二叉树，即哈夫曼树呢？根据哈夫曼树权值越大越靠近根结点的特点，哈夫曼最早提出了一个带有一般规律的算法，叫哈夫曼算法。哈夫曼算法具体描述如下：

(1) 初始化，由给定的 n 个权值$\{w_1,w_2,\cdots,w_n\}$构造 n 棵只有一个根结点的二叉树，从而得到一个二叉树集合 $F=\{T_1,T_2,\cdots,T_n\}$；

(2) 选取与合并，在 F 中选取根结点权值最小的两棵二叉树分别作为左、右子树构造一棵新的二叉树，这棵新二叉树的根结点的权值为其左、右子树根结点的权值之和；

(3) 删除与加入，在 F 中删除作为左、右子树的两棵二叉树，并将新建立的二叉树加入到 F 中；

(4) 重复步骤(2)、步骤(3)，当集合 F 中只剩下一棵二叉树时，这棵二叉树便是哈夫曼树。

例如，给定取值集合 $W=\{4,1,7,3\}$，其哈夫曼树的构造过程如图 5.40 所示。

在哈夫曼算法中，根结点权值最小的二叉树优先合并，最后两棵二叉树合并生成的根结点即为哈夫曼树的根结点。因此不难理解，早合并的结点权值较小，离根结点远，后合并的权值较大的结点离根结点近。同时也容易理解，每次合并都是将两棵二叉树合二为一，二叉树的数目减少 1 个，增加 1 个分支结点，从一开始有 n 个二叉树，到最后只剩 1 棵二叉树，哈夫曼算法过程一共需要合并 $n-1$ 次，生成 $n-1$ 个分支结点，最终的哈夫曼树中，总的结点个数是 $2n-1$ 个。

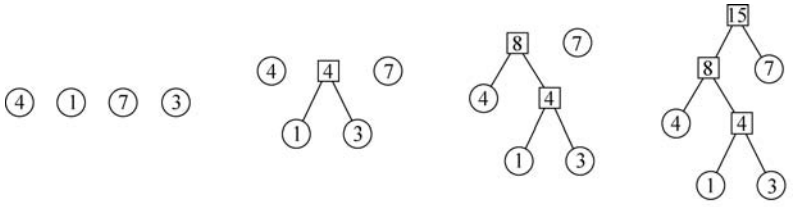


图 5.40　哈夫曼树的构造

为了实现哈夫曼算法，必须为哈夫曼树选择合适的存储结构。给定 n 个结点权值，对应的哈夫曼树中共有 $2n-1$ 个结点，可以用一个大小为 $2n-1$ 的一维数组来存放。由于哈夫曼树的构建是从叶子结点开始，不断构建新的父结点，直至树根，所以结点中应包含指向父结点的指针；使用哈夫曼树时是从树根开始，根据需要遍历树中的各个结点，因此每个结点需要有指向其左孩子和右孩子的指针。所以整个存储结构构成一个静态三叉链表。链表的结点结构如图 5.41 所示。

parent	weight	leftChild	rightChild

图 5.41　哈夫曼树结点的结构

其中，weight 表示权值，parent 指向双亲结点的位置，leftChild 指向左孩子结点的位置，rightChild 指向右孩子结点的位置。结点结构对应的 C++结构体定义见代码 5.21。

代码 5.21　哈夫曼树结点结构

```
struct{
    double weight;
    int parent;                    //双亲域,双亲结点在一维数组中的下标
    int leftChild;                 //左孩子域,左孩子结点的下标
    int rightChild;                //右孩子域,右孩子结点的下标
} HTNode, HuffmanTree[M + 1];
```

给定取值集合 $W=\{4,1,7,3\}$，哈夫曼算法执行过程如图 5.40 所示，其对应的存储状态变化如图 5.42 所示，其中加粗的字表示该步骤更新的数值。

基于以上分析，哈夫曼算法的实现实质上是选择两棵根结点权值最小的子树，并将两棵子树合并，不断重复直至最终只剩一棵二叉树。

1）select()函数，选择两棵根结点权值最小的树

请读者自行完成。

2）将两棵子树合并成一棵，迭代创建哈夫曼树

将两棵子树合并成一棵，只需要保存新的根结点，并更新原子树的双亲结点即可。创建哈夫曼树的 C++实现见代码 5.22。

下标	parent	weight	leftChild	rightChild
1	0	4	0	0
2	0	1	0	0
3	0	7	0	0
4	0	3	0	0
5	0	0	0	0
6	0	0	0	0
7	0	0	0	0

(a) 初始化

下标	parent	weight	leftChild	rightChild
1	0	4	0	0
2	**5**	1	0	0
3	0	7	0	0
4	**5**	3	0	0
5	0	**4**	**2**	**4**
6	0	0	0	0
7	0	0	0	0

(b) 第一次合并

下标	parent	weight	leftChild	rightChild
1	**6**	4	0	0
2	5	1	0	0
3	0	7	0	0
4	5	3	0	0
5	**6**	4	2	4
6	0	**8**	**1**	**5**
7	0	0	0	0

(c) 第二次合并

下标	parent	weight	leftChild	rightChild
1	6	4	0	0
2	5	1	0	0
3	**7**	7	0	0
4	5	3	0	0
5	6	4	2	4
6	**7**	8	1	5
7	0	**15**	**3**	**6**

(d) 第三次合并

图 5.42　哈夫曼算法执行过程中存储状态的变化

代码 5.22　createHuffmanTree()函数

```
template < typename Element >
void createHuffmanTree(HuffmanTree ht[], int w[], int n)
{
    int i,s1,s2,m;
    for(i = 1;i < = 2 * n - 1;++i){                          //初始化
        ht[i].parent = 0;
        ht[i].leftChild = 0;
        ht[i].rightChild = 0;
        if(i < = n) ht[i].weight = w[i];
        else ht[i].weight = 0;
    }
    for(i = n + 1;i < = m;++i){
        select(ht,i - 1,&s1,&s2);                            //找两棵根结点权值最小的树
        ht[i].weight = ht[s1].weight + ht[s2].weight;        //根结点权值
        ht[s1].parent = i; ht[s2].parent = i;                //修改子树双亲域
        ht[i].leftChild = s1; ht[i].rightChild = s2;         //修改根结点左右孩子域
    }
}
```

5.6.3　前缀编码和哈夫曼编码

在数据存储及通信过程中,经常需要将字符转换为二进制字符 0 和 1 组成的二进制码

串，这个过程称为编码，例如 ASCII 码、指令系统编码等。编码方式有等长编码和不等长编码两类。能够唯一进行译码的编码方式，才是有效和可用的。

1. 等长编码

等长编码表示一组对象的二进制位串的长度相等，如 ASCII 码。

例如：假设传送的电文为英文序列 ABACCDA，采用的编码方案是 A(00)、B(01)、C(10)、D(11)，每个英文字母用 2 比特表示，为等长编码。利用该编码方案，英文序列的编码为 00010010101100，总长 14 比特。译码方式比较简单，两位一分即可。

2. 不等长编码

不等长编码表示一组对象的二进制位串的长度不相等。一般情况下，在英文字母 A～Z 中，E 的使用频率比 X、Z 要大得多。使用频率高的用短码，使用频率低的用长码，编码不等长，大部分情况下电文长度会减少。

例如：在传送的英文电文序列 ABACCDA 中，A、B、C、D 的使用频率分别为 3、1、2、1，根据频率高用短码、频率低用长码的原则，采用的不等长编码方案可以是 A(0)、B(00)、C(1)、D(10)。利用该编码方案，英文序列的编码为 000011100，总长 9 比特，比等长编码的长度小。但存在的问题是，利用该编码方案进行编码的序列，不能唯一进行译码，如前 4 比特 0000，可以译码为 AAAA、ABA 或 BB。出现这种情况的原因是，编码方案中存在一个字符的编码是另一个字符编码的前缀。

3. 前缀编码

设计不等长编码时，必须保证某字符的编码不是另一字符编码的前缀(最左子串)，这种编码称为前缀编码。例如对于上述例子，编码方案 A(0)、B(110)、C(10)、D(111)就是一种前缀编码。

4. 哈夫曼编码

哈夫曼编码是一种前缀编码，该方法以字符出现的概率为权值来构造哈夫曼树，并得到平均长度最短的码字。

假设电文中共有 n 种字符，第 i 种字符出现的次数为 w_i，对应的编码长度为 l_i，则电文总长为

$$l=\sum_{i=1}^{n} w_i l_i \tag{5.3}$$

假设 w_i 是叶子结点的权值，其对应的树的带权路径长度的定义与式(5.3)恰好相同，可以认为前缀编码的实现过程，与哈夫曼树的构造有关系。设计前缀编码时，肯定希望编码后的电文总长 l 最小，对应的过程就是给定叶子结点权值，构造哈夫曼树的过程。因此，设计电文总长最短的问题转变为设计哈夫曼树的问题。

该编码的具体构造过程如下：①以字符出现频率 w_i 为叶子结点的权值，构造哈夫曼树；②树中的左分支用字符 0 表示，右分支用字符 1 表示，从根到叶子的路径上分支字符组成的字符串作为该叶子结点字符的编码。这种编码方式就是哈夫曼编码，因为根结点到任一叶子结点的路径，不会经过其他叶子结点，避免了一个字符的编码是另一个字符编码的前缀的情况，是编码长度最短的前缀编码。

例如：一组字符{A,B,C,D,E,F,G}，假设出现的频率分别是{9,11,5,7,8,2,3}，其对应的哈夫曼树如图 5.43 所示，给出的哈夫曼编码方案为 A(00)，B(10)，C(010)，D(110)，

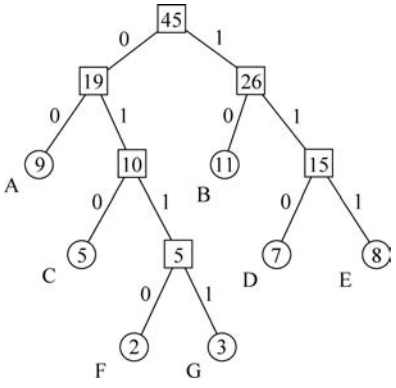


图 5.43　构造哈夫曼树

E(111),F(0110),G(0111)。

哈夫曼编码的译码过程如下：从哈夫曼树根开始,从待译码电文中逐位取码。若编码是0,则向左走；若编码是1,则向右走；一旦到达叶子结点,则译出一个字符；重新从根结点出发重复上述过程,直到电文结束。如对于电文编码0101100110110l0,只能给出唯一的译文CDFDB。从前面所学已经知道,对于同一组叶子结点权值集合,构造出的哈夫曼树不是唯一的,其对应的哈夫曼编码方案也不是唯一的,但编码长度都是最小的。显然,哈夫曼编码和译码所对应的哈夫曼树必须是同一棵,否则译码不会成功。

5.7　本章小结

树和二叉树是非常重要的非线性数据结构,在计算机科学中具有广泛的应用。本章主要讲述了树和二叉树的逻辑结构和存储结构,树、森林和二叉树的转换,以及树的典型应用——哈夫曼树和哈夫曼编码。树和二叉树是两种不同的树形结构,它们的主要操作是遍历操作。树的存储结构有双亲表示法、孩子表示法、双亲孩子表示法和孩子兄弟表示法,都可看成是链式存储结构；二叉树主要有顺序存储结构和二叉链表结构,但前者只适合存储完全二叉树。为了方便遍历操作,还可以在二叉链表的基础上构造线索二叉树。树和森林的存储和操作实现都比较烦琐,可以转换为二叉树进行处理。树、森林和二叉树具有一一对应的关系,树和森林的遍历操作也可以相互对应。哈夫曼树是典型的二叉树的应用,可以用来进行前缀编码。

本章习题

一、选择题

1. 若将一棵树 T 转化为对应的二叉树 bt,则下列对 bt 的遍历中,其遍历序列与 T 的后序遍历序列相同的是(　　)。

A. 先序遍历　　B. 中序遍历　　C. 后序遍历　　D. 按层遍历

2. 对 n 个互不相同的符号进行哈夫曼编码。若生成的哈夫曼树共有 115 个结点,则 n 的值是(　　)。

A. 56　　B. 57　　C. 58　　D. 60

3. 设一棵非空完全二叉树 T 的所有叶结点均位于同一层,且每个非叶结点都有 2 个子结点。若 T 有 k 个叶结点,则 T 的结点总数是(　　)。

A. $2k-1$　　B. $2k$　　C. k^2　　D. 2^k-1

4. 要使一棵非空二叉树的前序序列与中序序列相同,则所有非叶结点需要满足的条件是(　　)。

A. 只有左子树　　B. 只有右子树

C. 结点的度均为 1　　D. 结点的度均为 2

5. 在二叉树结点的前序、中序和后序序列中，所有叶子结点的先后顺序(　　)。

A. 都不相同　　B. 完全相同

C. 前序与中序相同，后序不同　　D. 中序和后序相同，前序不同

6. 已知字符集{a,b,c,d,e,f}，若各字符出现的次数分别为 6,3,8,2,10,4，则对应字符集中各字符的哈夫曼编码可能是(　　)。

A. 00,1011,01,1010,11,100　　B. 00,100,110,000,0010,01

C. 10,1011,11,0011,00,010　　D. 0011,10,11,0010,01,000

7. 已知一棵完全二叉树的第 6 层(设根是第 1 层)有 8 个叶结点，则该完全二叉树的结点个数最多是(　　)。

A. 39　　B. 52　　C. 111　　D. 119

8. 一棵二叉树高度为 h，所有结点的度或为 0，或为 2，则这棵二叉树最少有(　　)结点。

A. $2h$　　B. $2h-1$　　C. $2h+1$　　D. $h+1$

9. 在下列存储形式中，(　　)不是树的存储形式。

A. 双亲表示法　　B. 孩子链表表示法

C. 孩子兄弟表示法　　D. 顺序存储表示法

10. 采用双亲表示法表示树，则具有 n 个结点的树至少需要(　　)个指向双亲的指针。

A. n　　B. $n+1$　　C. $n-1$　　D. $2n$

11. 现有一“遗传”关系，设 x 是 y 的父亲，则 x 可以把它的属性遗传给 y，表示该遗传关系最适合的数据结构为(　　)。

A. 向量　　B. 树　　C. 图　　D. 二叉树

12. 已知一棵二叉树树形如图 5.44 所示，其后序序列为 eacbdgf，则树中与 a 同层的结点是(　　)。

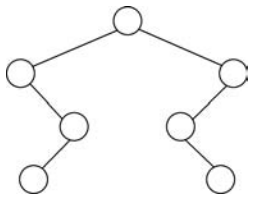


图 5.44　二叉树树形

A. c　　B. d

C. f　　D. g

13. 将森林转化为对应的二叉树，若二叉树中，结点 u 是结点 v 的父结点，则在原来的森林中，u 和 v 可能的关系是(　　)。

Ⅰ. 父子关系　　Ⅱ. 兄弟关系　　Ⅲ. u 的父结点与 v 的父结点是兄弟关系

A. 只有Ⅱ　　B. Ⅰ和Ⅱ　　C. Ⅰ和Ⅲ　　D. Ⅰ、Ⅱ和Ⅲ

14. 设树的度为 4，已知度为 4 的结点有 20 个，度为 3 的结点有 15 个，度为 2 的结点有 10 个，度为 1 的结点有 5 个，则叶子结点数为(　　)。

A. 82　　B. 151　　C. 50　　D. 101

15. 若二叉树的前序遍历序列为 abcdef，中序遍历序列为 cbaedf，则后序遍历序列为(　　)。

A. cbefda　　B. fedcba　　C. cbedfa　　D. 不确定

16. 若 X 是二叉中序线索树中一个有左孩子的结点，且 X 不是根结点，则 X 的前驱是(　　)。

A. X 的双亲　　B. X 的右子树中最左侧结点

C. X 的左子树中最右侧结点　　D. X 的左子树中最右的叶子结点

二、填空题

1. 树在计算机内的表示方式有_____、_____、_____。

2. 中缀式 $a+b*3+4*(c-d)$对应的前缀式为_____,若 $a=1,b=2,c=3,d=4$,则后缀式 $db/cc*a-b*+$的运算结果为_____。

3. 深度为 H 的完全二叉树至少有_____个结点,最多有_____个结点,H 和结点总数 N 之间的关系是_____。

4. 已知二叉树中叶子数为 40,仅有一个孩子的结点数为 20,则总结点数为_____。

5. 若按层序将一棵 n 个结点的完全二叉树从 1 开始编号,那么结点 i 没有右兄弟的条件为_____。

6. 一棵有 n 个结点的满二叉树有_____个度为 1 的结点,有_____个分支(非终端)结点和_____个叶子,该满二叉树的深度为_____。

7. 在树的孩子兄弟表示法中,二叉链表的左指针指向_____,右指针指向_____。

8. 先序遍历森林时,首先访问森林中第一棵树的_____。

9. 已知一棵二义树的前序序列为 abdecfhg,中序序列为 dbeahfcg,则该二叉树的根为_____,左子树中有_____,右子树中有_____。

10. 若以{4,5,6,7,8}作为叶子结点的权值构造哈夫曼树,则其带权路径长度是_____。

11. 将一棵树转换成二叉树后,根结点没有_____。

12. 以下程序是求二叉树深度的递归算法,请填空完善之。

```
int depth(BiTree *bt)          /*bt 为根结点的指针*/
{
    int h1,hr;
    if(bt==NULL) return _____;
    h1=height(bt->lchild);
    hr=height(bt->rchild);
    if(_____)_____;
    return(hr+1);
}
```

三、应用题

1. 一棵完全二叉树有 892 个结点,试求:

(1) 树的高度;

(2) 叶结点个数;

(3) 单支结点数;

(4) 最后一个非终端结点的序号。

2. 有 n 个结点并且高度为 n 的二叉树的数目是多少?

3. 将算术表达式$((a+b)+c\times(d+e)+f)\times(g+h)$转化为二叉树。

4. 在某二叉树上进行前序、中序遍历后发现该二叉树的前序序列的最后一个结点和中

序序列的最后一个结点是同一个结点。请问该结点具有何种性质？为什么？

5. 有 6 个叶子结点分别为{a,b,c,d,e,f}，它们的权值相应为{0.32,0.12,0.16,0.28,0.08,0.04}，画出它们的哈夫曼树，按左 0 右 1 规则给出每个字符的哈夫曼编码，并求出这棵树的带权路径长度。

6. 分别给出满足下列条件的二叉树：

(1) 前序与中序遍历序列相同；

(2) 前序与中序遍历序列相反；

(3) 中序与后序遍历序列相同；

(4) 前序与后序遍历序列相同。

7. 一棵二叉树的前序、中序和后序遍历序列如下，其中有部分未给出，试构造该二叉树。

前序序列：__CDE_GHI_K。

中序序列：CB__FA_JKIG。

后序序列：_EFDB_JIH_A。

8. 将图 5.45 所示的二叉树转化为相应的树或者森林。

9. 已知一棵二叉树的每个结点，要么其左右子树均为空，要么其左右子树均不为空，即没有度为 1 的结点。又知其前序和后序序列如下。

前序序列：JFDBACEHXIK。

后序序列：ACBEDXIHFKJ。

试画出二叉树，给出中序序列，简要说明分析过程。

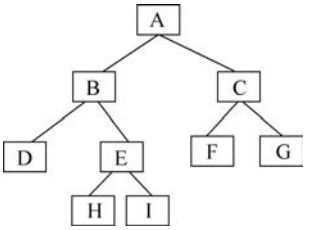


图 5.45　二叉树示例

四、算法设计题

1. 已知二叉树采用二叉链表方式存储，设计一个算法，不使用栈或者递归，返回二叉树后序遍历序列的第一个结点。

2. 要求二叉树按照二叉链表存储，写一个判别二叉树是否为完全二叉树的算法。

3. 设计一个算法，将二叉链表中所有结点的左右子树交换。

4. 设计一个算法，利用叶子结点的空指针域，将所有叶子结点链接为一个带有头结点的双向链表，返回头结点的地址。

5. 设计一个算法，输入二叉树的前序和中序遍历序列，构造二叉链表。

6. 二叉树的带权路径长度(WPL)是二叉树中所有叶结点的带权路径长度之和。给定一棵二叉树 T，采用二叉链表存储，结点结构如下：

left	weight	right

其中，叶结点的 weight 域保存该结点的非负权值。设 root 为指向 T 的根结点的指针，请设计求 T 的 WPL 的算法，要求：

(1) 给出算法的基本设计思想；

(2) 使用 C 或 C++语言，给出二叉树结点的数据类型定义；

(3) 根据设计思想，采用 C 或 C++语言描述算法，关键之处给出注释。

7. 设计一个算法,将用二叉树存储的表达式转换为等价的中缀表达式(通过括号反映操作符的计算次序)。

(1) 给出算法的基本设计思想;

(2) 使用 C 或 C++语言,关键之处给出注释。

扩展阅读:树形结构的设计模式——组合模式

本章我们学习了树形结构的逻辑结构和存储结构,重点介绍了二叉链表的存储和应用。在解决很多实际问题时,经常把树转化为二叉树来进行处理。在面向对象程序设计中,针对常见的反复出现的问题有一些模式化的解决方案,称为设计模式。其中,有一种设计模式专门针对树形结构,称为组合(composite)模式。下面介绍这种设计模式的基本思想。

组合模式是一种结构型设计模式,专门解决树形结构的存储、表述和处理等问题。与二叉链表不同,组合模式不使用指针链接双亲和孩子,而是将分支结点和其孩子结点递归地组合在一起,形成整体-部分的关系。在组合模式中,把分支结点称为容器对象,叶子结点称为叶子对象。

组合模式由以下四种基本对象组成。

(1) Component:抽象构件对象。

(2) Leaf:叶子构件对象。

(3) Composite:容器构件对象(分支构件)。

(4) Client:客户类对。

其类图如图 5.46 所示。

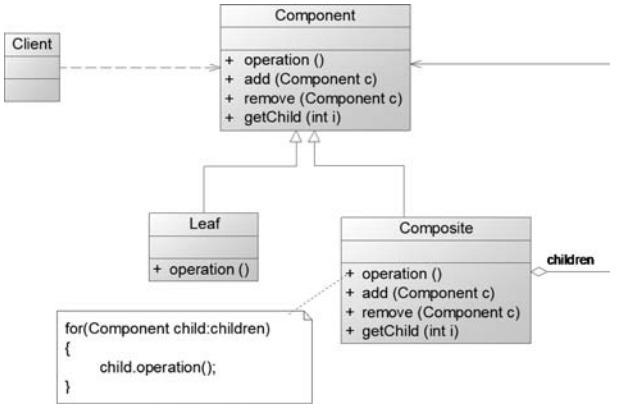


图 5.46 组合模式类图

抽象构件 Component 定义为抽象类,包含了 add、remove、getChild 和 operation 四个抽象操作。其中,add、remove 和 getChild 分别对应于增加孩子结点、删除孩子结点和获取孩子结点信息等三种基本操作,operation 为结点本身的行为,根据实际问题的特点来确定。叶子构件 Leaf 和容器构件 Composite 都是抽象构件的子类,需要对各种基本操作给出具体实现。其中,由于叶子结点没有孩子,因此只需要实现 operation;而 Composite 需要实现全部函数,Composite 的一般实现见代码 5.23。

代码 5.23 Composite 的一般实现

```
public class Composite extends Component
{
    private ArrayList list = new ArrayList();          //用于存放孩子结点
    public void add(Component c)                       //增加孩子结点
    {
            list.add(c);
    }
    public void remove(Component c)                    //删除孩子结点
    {
            list.remove(c);
    }
    public Component getChild(int i)                   //获取孩子结点
    {
            (Component)list.get(i);
    }
    public void operation()
    {
            for(Object obj:list)                       //本例遍历执行所有孩子的 operation
            {
                ((Component)obj).operation();
            }
    }
}
```

组合模式在处理树形结构时，不必关注叶子结点和分支结点的区别，从而模糊了简单元素（叶子结点）和复杂元素（分支结点）的概念，客户程序可以像处理简单元素一样来处理复杂元素，从而使得客户程序与复杂元素的内部结构解耦。

组合模式展示了一种全新的设计思想，用纯正面向对象的方式来存储树结构。读者可以尝试自行编程实现树和二叉树的存储。