第5章 树和二叉树

CHAPTER

前面几章主要讨论了线性表、栈、队列、串等线性数据结构。线性结构主要反映了数据元素之间的线性关系。在计算机领域的实际应用场景中，有很多问题若用非线性结构表示和运算则比线性结构更加明确，数据处理更加方便。本章讨论的树、二叉树等树结构就是一种重要的非线性结构。具体来说，树结构是一种层次结构，这种层次结构中任一结点的前驱如果存在则一定唯一，其后继如果存在则可能有多个。树结构在客观世界和计算机技术中的应用十分广泛，如人类社会的家谱和社会组织机构、互联网域名系统等都可以用树来表示；在编译程序中，可用树来表示源程序的语法结构；在操作系统中，可用树来组织文件。

本章先介绍树的相关概念、逻辑结构以及存储结构，再着重介绍二叉树的相关概念、逻辑结构、性质和存储结构，二叉树和树、森林的相互转换，最后介绍哈夫曼树及哈夫曼编码。

5.1 树的逻辑结构


电子课件

5.1.1 树的定义和术语

树(Tree)是由 $n(n\geqslant 0)$[①]个结点构成的有限集合 T。如果结点数为零，称为空树。否则，任何一个非空树满足以下两个条件。

(1) 有且只有一个特定的称为**根**(Root)的结点。

(2) 除根结点以外的其他结点被分成 $m(m\geqslant 0)$ 个互不相交的有限集合 $T_1, T_2, \cdots, T_m$，其中每个集合又是一棵树，并称为根结点的**子树**(Subtree)。如图 5.1 所示为一棵树。

① 对树的定义有两种观点：一种观点认为 $n>0$，空树不能算树；另一种观点认为若不允许树为空，则空二叉树和树的转换无法进行。作者持后一种观点。

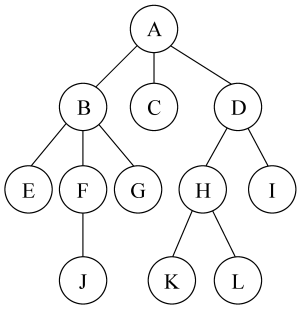


图 5.1 树的示意图

如图 5.1 所示树中 A 为根结点，其余的结点可以分为三个互不相交的子集：$T_1=\{B,E,F,G,J\}$，$T_2=\{C\}$，$T_3=\{D,H,I,K,L\}$；T_1、T_2、T_3 是根结点 A 的三棵子树，它们分别表示一棵树，例如 T_1，其根结点为 B，其余结点可分成三个互不相交的子集 $T_{11}=\{E\}$，$T_{12}=\{F,J\}$，$T_{13}=\{G\}$，它们都是结点 B 的子树。

下面介绍有关树的一些术语。

(1) **结点**(Node)：包含数据项及指向其他结点分支的结构称为结点。如图 5.1 中的树有 12 个结点。

(2) **结点的度**(Degree)：结点所拥有的子树的个数称为结点的度。如图 5.1 中结点 A 的度为 3，结点 B 的度为 3，结点 C 的度为 0，结点 D 的度为 2。

(3) **叶子结点**(Leaf)：树中度为 0 的结点，也称为终端结点。如图 5.1 中的 E、J、G、C、K、L、I 为叶子结点。

(4) **分支结点**(Nonterminal Node)：度不为 0 的结点，又称为非终端结点。如图 5.1 中的 A、B、D、F、H 为分支结点。

(5) **孩子**(Child)：一个结点的直接后继结点称为该结点的孩子。一个结点可以有多个孩子，如图 5.1 中 B 是结点 A 的第一棵子树 T_1 的根，故 B 是 A 的孩子，A 的孩子还有 C 和 D。

(6) **双亲**(Parent)：一个结点的直接前驱结点称为该结点的双亲，每个结点只有一个双亲。如图 5.1 中 A 是结点 B、C、D 的双亲。

(7) **兄弟**(Sibling)：同一双亲结点的孩子结点互称为兄弟。如图 5.1 中 B、C、D 互为兄弟。

(8) **祖先**(Ancestor)：从根结点到某结点所经过的分支上的所有结点，称为该结点的祖先。如图 5.1 中结点 J 的祖先为 A、B、F。

(9) **子孙**(Descendant)：以某一结点为根的子树中的任一结点都称为该结点的子孙。如图 5.1 中结点 B 的子孙为 E、F、G、J。

(10) **层次**(Level)：将根结点的层次设定为 1，则其孩子结点层次为 2，以此类推。如图 5.1 中结点 A 的层次为 1，结点 B、C、D 的层次为 2。

(11) **树的深度**(Depth)：树中结点的最大层次，又称为树的高度。如图 5.1 中树的深度为 4。

(12) **有序树**(Ordered Tree)和**无序树**(Unordered Tree)：如果树中结点的各子树从左到右是有次序的(即不能互换)，则称该树为有序树，否则称为无序树。

(13) **森林**(Forest)：$m(m\geqslant 0)$棵互不相交的树的集合。删除一棵树的根就会得到一个森林，反之，若给森林增加一个统一的根结点，森林就变成了一棵树。

就逻辑结构而言，任何一棵树都是一个二元组 Tree=(Root，F)，其中，Root 是数据元素，称作树的根结点；F 是 $m(m\geqslant 0)$棵树的森林，$F=(T_1,T_2,\cdots,T_m)$，其中，$T_i=(R_i,F_i)$称作根 Root 的第 i 棵子树；当 $m\neq 0$ 时，在树根和其子树森林之间存在下列

关系。

$$RF=\{<Root,R_i>\mid i=1,2,\cdots,m,m>0\}$$

这个定义将为森林和树与二叉树之间的转换提供帮助。

5.1.2　树的逻辑表示方法

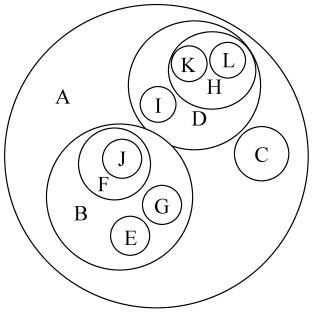
树的逻辑表示方式有很多，但无论哪种表示方式都能体现出树中各数据元素之间的层次关系，以及根结点和子树之间的一对多的关系。下面介绍树的几种常见的逻辑表示方法。

(1) 树状表示法(Tree Representation)：用圆圈表示一个结点，圆圈内的符号代表该结点的数据信息，结点之间的关系通过分支线表示。虽然每条分支线没有显示箭头，但默认存在自上而下的方向，即分支线的上方是下方结点的前驱结点，下方结点是上方结点的后继结点。树状表示法表示的树是一棵倒置的树，即根结点在上，叶子在下，如图 5.1 所示。

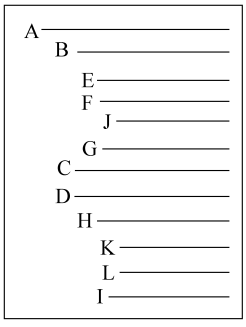
(2) 文氏图表示法(Venn Diagram Representation)：每棵树对应一个圆(椭圆)，圆内包含根结点和子树对应的圆，同一个根结点下的各子树对应的圆是不能相交的。用这种方法表示的树中，结点之间的关系是通过圆的包含关系来表示的。如图 5.2(a)所示为图 5.1 中树对应的文氏图表示法。

(3) 凹入表示法(Concave Representation)：每棵树的根结点对应一个线条，其子树的根对应一个较短的线条，且树的根在上，子树的根在下，同一个根下的各子树的根对应的线条长度相同，所有线条右对齐。如图 5.2(b)所示为图 5.1 中树对应的凹入表示法。

(4) 嵌套括号表示法(Nested Bracket Representation)：每棵树对应一个形如“根(子树 1，子树 2，…，子树 m)”的字符串，每棵子树的表示方式与整棵树类似，各个子树之间用逗号分开。在用这种方法表示的树中，结点之间的关系通过括号的嵌套表示。树的嵌套括号表示法也叫广义表表示法。如图 5.2(c)所示为图 5.1 中树对应的嵌套括号表示法。



(a) 文氏图表示法



(b) 凹入表示法

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

(c) 嵌套括号表示法

图 5.2　树的各种表示法

树表示法的多样性反映了树应用的广泛性。

上述树结构的定义结合树的一组基本操作就构成了树的抽象数据类型定义。

```
ADT Tree
{   数据对象 D:D={ai | ai∈DataSet, i=1,2,…,n,n≥0}。
    数据关系 R:R={<ai-1,ai> | ai-1,ai∈D,i=2,3,…,n,D 中存在唯一的称为根的数据元素
        Root,它无前驱;其余每个结点只有一个前驱,但可以有零个或多个后继结点}
    基本操作:
        TreeInit(T):构造一棵空树 T。                                    //构造空树
        TreeDestroy(T):销毁树 T,即释放树 T 所占的空间。                  //销毁树
        TreeChild(T,x,i):假如树 T 存在,x 为 T 的某结点数据,则返回 x 所在的第 i 棵子
            树;假如树 T 不存在第 i 棵子树,则返回空。                     //求树的第 i 棵子树
        TreeClear(T):清除树 T 的内容,使其为空树。                       //清空树
        PreTraverse(T):按前序次序对 T 中的每个结点进行访问且仅访问一次。
                                                                        //前序遍历树
        PostTraverse(T):按后序次序对 T 中的每个结点进行访问且仅访问一次。
                                                                        //后序遍历树
        TreeRoot(T):返回树 T 的根结点 Root。                            //求树的根结点
        TreeParent(T,x):假如树 T 存在,x 是 T 中的某个结点,如果 x 不是 T 的根结点,返回
            结点 x 的双亲;如果 x 是 T 的根结点,则返回空。                //求结点的双亲
        TreeRightBrother(T,x):假如树 T 存在,x 是 T 的某个结点,如果 x 有右兄弟,返回结
            点 x 的右兄弟;如果 x 没有右兄弟,则返回空。                   //求结点的右兄弟
        TreeInsert(T,y,i,x):假如树 T 存在,y 是 T 中的某个结点,i 为待插入的子树序号,
            则插入以 x 为根的子树为 T 中 y 所指结点的第 i 棵子树。       //插入子树
        TreeDepth(T):返回树的深度。                                     //求树的深度
        TreeEmpty(T):判断树是否为空树,如果树为空返回真,否则返回假。
                                                                        //判断树是否为空
}ADT Tree
```

树的应用十分广泛,在不同的软件系统中树的操作也不尽相同,因此树的抽象数据类型中的基本操作类型可随需要重新设定。


电子课件

5.2 树的存储结构

树的存储结构有很多种,下面重点介绍最为常用的三种。

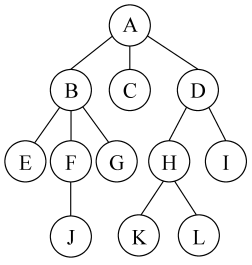
1. 双亲表示法

在一棵树中,根结点无双亲,其他任何结点的双亲只有一个,这是由树的定义决定的。双亲表示法正是利用了树的这种性质,将树中每个结点的信息存放于一个顺序表中,结点的信息包含元素数据域 data 和结点双亲在表中的位置域 parent,其形式说明如下。

```
#define MAXNODE  100                          //用户定义最大结点数
typedef struct                                //树的双亲表示法存储表示
{   DataType  data;                           //数据域
    int   parent;                             //双亲位置域
} Ptnode;
typedef struct
{   Ptnode  nodes[MAXNODE];
    int n;                                    //树中的结点个数
} Ptree;
```

例 5.1　有如图 5.3(a)所示的一棵树，给出其双亲表示法的存储结构。

解答：在双亲表示法中，因根结点无双亲，其双亲位置域用－1 表示。若结点 nodes[i]的双亲结点为结点 j，则 nodes[i].parent＝j，所以这种表示方法对求指定结点的双亲和祖先是十分方便的，可以反复调用求双亲的操作，直到找到树的根结点。但是，如果要求某结点的孩子，则需要遍历整个数组，操作比较费时。因此，求双亲的时间复杂度为 $O(1)$，求孩子的时间复杂度为 $O(n)$。



(a) 树

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

(b) 树的双亲表示法

图 5.3　树及树的双亲表示法

2. 孩子表示法

树也可以采用链式存储结构，因为树的每个结点都可能有自己的孩子，如果在每一个结点中设置若干指针指向该结点的孩子，那么每个结点的指针的个数是难以确定的。可以采用多重链表的方式解决该问题。这种解决方案是根据树的度 d 为每个结点设置 d 个指针域，如图 5.4 所示。

显然，这种链表中的结点是同构的，称为**多重链表**。但是由于树中有很多结点的度小于 d，很多指针域是空的，因此其缺点是造成存储空间的浪费。在这种结构中，具有 n 个结点的树总共有 $n\times d$ 个指针，因为树只有 $n-1$ 个分支，因此只有 $n-1$ 个指针指向树中某结点，其余 $n\times d-(n-1)=n\times(d-1)+1$ 个指针域均为空。其树的度越大，浪费空间越多。这种存储结构的优点是结点同构，易于管理。如果按每个结点实际的孩子个数设置指针，并在结点中设置 degree 域，表示该结点所包含的孩子数，则可以得到如图 5.5 所示的结点结构。

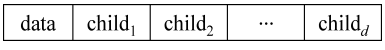


图 5.4　多重链表结点结构

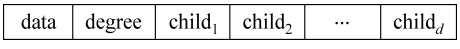
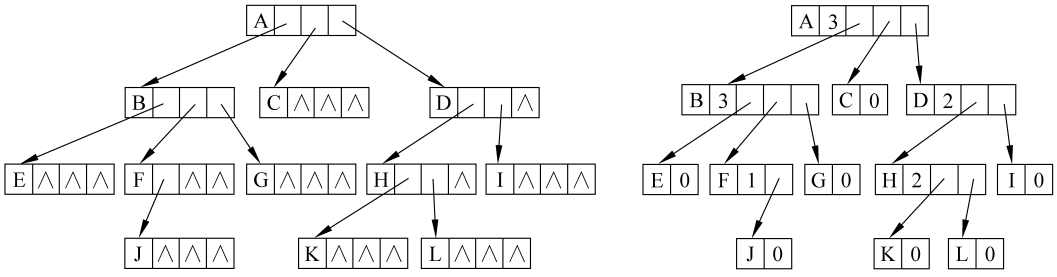


图 5.5　异构的链表结点结构

在这种存储结构中，结点是非同构的，即各结点不等长，这种存储结构的优点是节约了存储空间，但缺点是给运算带来了不便。

图 5.6 表示的是如图 5.3(a)所示的树的同构和非同构两种链式存储结构的表示方法。

树的另一种链式存储方法是把每个结点的孩子排列起来，形成一个链表，这样就为每个结点建立一个孩子链表，其中，叶子结点的孩子链表为空。每个链表增加一个头结点，为便于管理，将各个头结点放在一个顺序表中，这种表示方法也称为孩子链表表示法，其形式说明如下。



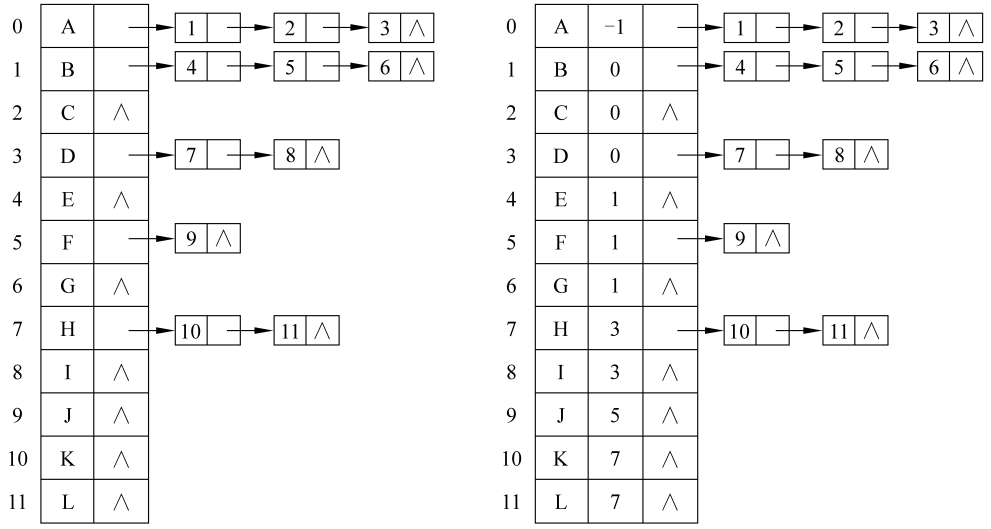
(a) 树结点同构方式　　(b) 树结点非同构方式

图 5.6　图 5.3(a)中树的结点的两种链式存储方式

```
#define MAXNODE  100
typedef struct Ctnode                          //孩子结点
{   int  child;
    struct Ctnode * next;
}  Ctnode, * ChildLink;
typedef struct                                 //头结点
{   DataType  data;
    Ctnode   * firstchild;
} CTBox;
CTBox  nodes[MAXNODE];
```

例 5.2　分别写出如图 5.3(a)所示的树的孩子链表和双亲孩子链表。

解答：如图 5.7(a)所示为图 5.3(a)中树的孩子链表，所有叶子结点的孩子链表为空，而非叶子结点的孩子链表由该结点的所有孩子结点组成。与双亲表示法相反，孩子链表便于实现涉及孩子及子孙的运算，但不利于实现与双亲有关的运算。可以将双亲表示法和孩子链表表示法结合起来，形成双亲孩子链表表示法。图 5.7(b)就是用双亲孩子链表表示法表示图 5.3(a)中树的结果。



(a) 孩子链表　　(b) 双亲孩子链表

图 5.7　图 5.3(a)中树的孩子链表和双亲孩子链表

3. 孩子兄弟表示法

孩子兄弟表示法也称为孩子兄弟链表，是一种链式存储结构。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，两个链域分别命名为 firstchild 和 nextsibling。其说明如下。

```
typedef struct  CSNode                          //树的孩子兄弟表示法
{   DataType      data;
    struct CSNode  *firstchild,  *nextsibling;
} CSNode, *CSTree;
```

例 5.3　写出如图 5.3(a)所示的树的孩子兄弟表示法存储结构。

解答：如图 5.8 所示的是图 5.3(a)的孩子兄弟链表。用这种存储结构很容易实现树的某些操作。例如，要访问结点 x 的第 i 个孩子结点(假定存在)，只要先从结点 x 的 firstchild 域找到第一个孩子结点，然后沿孩子结点的 nextsibling 域连续走 $i-1$ 步，便可以找到结点 x 的第 i 个孩子。更重要的是，这种存储结构和后面要介绍的二叉树的二叉链表存储结构本质上是相同的，因此树和二叉树可以相互转换，树可以采用二叉树的相关算法来实现一些应用。

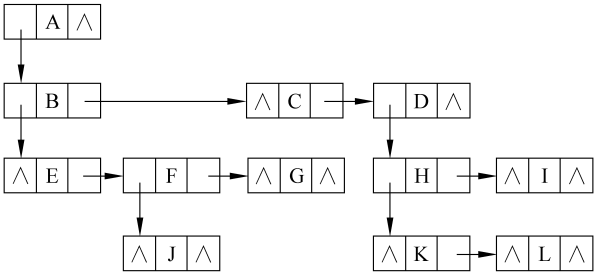


图 5.8　图 5.3(a)中树的孩子兄弟链表

5.3　二叉树的逻辑结构


电子课件

5.3.1　二叉树的定义

二叉树(Binary Tree)是另一种重要的树结构，其递归形式的定义为

二叉树是 $n(n\geqslant 0)$个结点组成的有限集合，该集合或者为空($n=0$)，称为空二叉树，或者是由一个特定的称为根的结点和两棵互不相交的分别称为左子树和右子树的二叉树组成。

二叉树的特点是每个结点最多有两个孩子，分别称为该结点的左孩子和右孩子。即二叉树中不存在度大于 2 的结点，并且二叉树的子树有左、右之分，其子树的次序不能颠倒，即使只有一棵子树，也必须说明其是左子树还是右子树。

根据定义，如图 5.9 所示，为二叉树的 5 种基本形态。

下面给出二叉树的抽象数据类型定义。

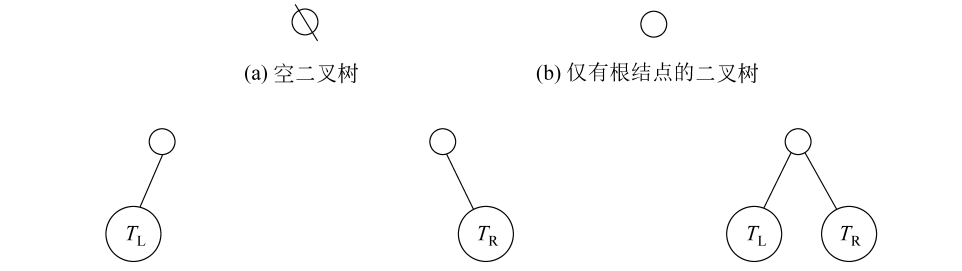


图 5.9　二叉树的 5 种基本形态

ADT BinTree{
数据对象 D: $D=\{a_i \mid a_i \in \text{DataSet},\ i=1,2,\cdots,n,n\geqslant 0\}$
数据关系 R: $R=\{<a_{i-1},a_i> \mid a_{i-1},a_i \in D, i=2,3,\cdots,n, D$ 中存在唯一的称为根的数据元素 Root,它无前驱;其余每个结点只有一个前驱,但可以有最多两个后继}
基本操作:
BinTreeInit (BT):构造空二叉树 BT。　　//构造空二叉树
BinTreeRoot(BT):假如二叉树 BT 存在,则返回 BT 的根结点。　　//返回二叉树的根
BinTreeParent(BT,x):假如二叉树 BT 存在,且 x 是 BT 中的某个结点,则返回 x 的双亲;假如二叉树 BT 不存在,则返回空。　　//求结点的双亲
BinTreeLeftChild (BT,x):假如二叉树 BT 存在,且 x 是 BT 的某个结点,则返回 x 的左孩子;假如二叉树 BT 不存在,则返回空。　　//求结点的左孩子
BinTreeRightChild (BT,x):假如二叉树 BT 存在,且 x 是 BT 的某个结点,则返回 x 的右孩子;假如二叉树 BT 不存在,则返回空。　　//求结点的右孩子
BinTreeBulid(BT,LBT,RBT):假如存在两棵二叉树 LBT 和 RBT,则生成二叉树 BT,且 LBT 和 RBT 分别是 BT 的左子树和右子树。　　//生成二叉树
BinTreeInsertLeft(BT,y,x):假如二叉树 BT 存在,且 y 是 BT 的某个结点,则将子树 x 作为 y 的左子树插入。　　//插入左子树
BinTreeInsertRight(BT,y,x):假如二叉树 BT 存在,且 y 是 BT 的某个结点,则将子树 x 作为 y 的右子树插入。　　//插入右子树
BinTreeDeleteLeft(BT,x):假如二叉树 BT 存在,且 x 是 BT 的某个结点,则将 x 的左子树删除。　　//删除左子树
BinTreeDeleteRight(BT,x):假如二叉树 BT 存在,且 x 是 BT 的某个结点,则将 x 的右子树删除。　　//删除右子树
BinTreeClear(BT):假如二叉树 BT 存在,则将二叉树 BT 清空。　　//清空二叉树
BinTreeTraverse(BT):假如二叉树 BT 存在,则按某种次序访问二叉树中的每个结点一次且仅一次。　　//遍历二叉树
}**ADT** BinTree

二叉树的概念十分重要。首先,从很多实际问题中抽象出来的数据结构往往是二叉树结构,且很多算法问题用二叉树结构来解决非常便利;其次,任何一棵树都可以通过简单转换得到与之对应的二叉树,这样,就可以采用二叉树的存储结构,并利用二叉树的相关算法来解决树的相关应用。

5.3.2　二叉树的性质

二叉树具有 5 个典型的基本性质,具体如下。

性质 1　一个非空二叉树的第 i 层上至多有 2^{i-1} 个结点($i\geqslant 1$)。

证明：利用数学归纳法容易证得此性质。

当 $i=1$ 时，二叉树中只有一个结点即为根结点，$2^{i-1}=2^0=1$，结论显然成立。

假设 $i=k$ 时结论成立，即第 k 层上至多有 2^{k-1} 个结点。因为二叉树的每个结点的度至多为 2，所以在第 $k+1$ 层上的最大结点数是第 k 层上的最大结点数的 2 倍，即 $2\times 2^{k-1}=2^k=2^{(k+1)-1}$，故第 $k+1$ 层上至多有 $2^{(k+1)-1}$ 个结点的结论成立。

综上，性质 1 结论成立。

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

证明：因为深度为 k 的二叉树中，每层的结点数达到最多时，该二叉树结点总数最多。根据性质 1 有，该二叉树结点总数为 $2^0+2^1+\cdots+2^{k-1}=2^k-1$，故结论成立。

性质 3　任何一棵非空二叉树中，若叶子结点个数为 n_0，度为 2 的结点个数为 n_2，则有 $n_0=n_2+1$。

证明：设二叉树中结点总数为 n，n_1 为二叉树中度为 1 的结点个数，因为二叉树中只有度为 0、度为 1 和度为 2 三种类型的结点，所以二叉树的结点总数为

$$n=n_0+n_1+n_2$$

设二叉树中的边(分支)数为 B。除了根结点之外，其余结点都有一个双亲，涉及该结点到其双亲结点的一条边，因此边的条数比结点总数少 1，即有

$$B=n-1$$

又由于每个度为 2 的结点有两个孩子，涉及该结点到其两个孩子结点的两条边，每个度为 1 的结点只有一个孩子，涉及该结点到其孩子结点的一条边，度为 0 的结点无孩子，不涉及边，因此有

$$B=n_1+2n_2$$

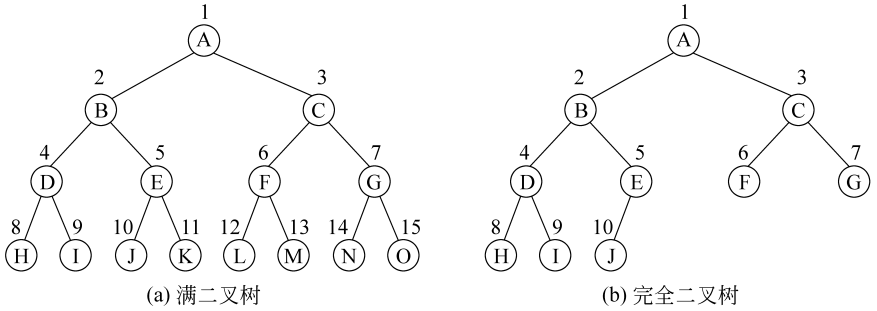
将上述三个表达式进行合并，得到：

$$n_0+n_1+n_2-1=n_1+2n_2$$

经过化简可以得出 $n_0=n_2+1$，故结论成立。

下面给出两种特殊二叉树——满二叉树和完全二叉树的概念。

满二叉树(Full Binary Tree)：每层都有最大结点数目的二叉树，深度为 k 的满二叉树中有 2^k-1 个结点。如图 5.10(a)所示的二叉树为一棵满二叉树。



(a) 满二叉树　　(b) 完全二叉树

图 5.10　满二叉树和完全二叉树

完全二叉树(Complete Binary Tree)：深度为 k，结点数为 n 的二叉树，如果其结点 1～n

的位置序号分别与满二叉树的结点 1～n 的位置序号一一对应，则此二叉树为完全二叉树。如图 5.10(b)所示的二叉树为一棵完全二叉树。

可见，满二叉树一定是完全二叉树，而完全二叉树不一定是满二叉树。完全二叉树的特征是叶子结点至多出现在最下面两层，若某结点无左子树，则该结点一定也无右子树。也可以将完全二叉树看成将一棵满二叉树的最下面一层从右到左连续删除若干叶子结点而得到的二叉树。

性质 4 具有 n 个结点的完全二叉树的深度为$\lceil \log_2(n+1) \rceil$[①]。

证明：假设 n 个结点的完全二叉树的深度为 k，则 n 的值应大于深度为 $k-1$ 的满二叉树的结点数 $2^{k-1}-1$，小于或等于深度为 k 的满二叉树的结点数 2^k-1，即

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

进一步可推导出

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

两边取对数后，有

$$k-1<\log_2(n+1)\leqslant k$$

因为 k 是整数，所以有 $k=\lceil \log_2(n+1) \rceil$。故结论成立。

性质 5 如果对一棵有 n 个结点的完全二叉树中的结点按层次自上而下(每层自左而右)从 1 到 n 进行编号，则任意一个结点 $i(1\leqslant i\leqslant n)$有：

(1) 若 $i=1$，则结点 i 为根，无双亲；若 $i>1$，则结点 i 的双亲结点的编号是$\lfloor i/2 \rfloor$。

(2) 若 $2i\leqslant n$，则 i 的左孩子的编号是 $2i$，否则 i 无左孩子。

(3) 若 $2i+1\leqslant n$，则 i 的右孩子的编号是 $2i+1$，否则 i 无右孩子。

由图 5.10 可以看出性质 5 所描述的结点与编号的对应关系。


电子课件

5.4 二叉树的存储结构

二叉树的存储结构可分为顺序存储结构和链式存储结构，下面依次展开介绍。

5.4.1 二叉树的顺序存储结构

二叉树的顺序存储结构是用一组地址连续的存储单元(一维数组)依次自上而下，自左而右地存储二叉树中的各个结点。根据二叉树性质 5，对于完全二叉树来说，编号为 i 的结点将存储在一维数组中下标为 i 的分量中，如图 5.11(a)所示；对于一般的二叉树，为了让其所有结点的双亲和孩子仍具有对应关系，则需要将二叉树扩展为完全二叉树，将新增加的结点全部记为“ϕ”，表示该结点不存在，但为其在一维数组中留出空位，即将扩展后的二叉树中每个结点存储在一维数组的相应分量中，如图 5.11(b)所示。

由图 5.11 可以看出，二叉树的顺序存储结构空间利用率较低，尤其是二叉树的深度较大，结点个数较少的情况。在最坏的情况下，一个深度为 k 且只有 k 个结点的二叉树(单支树)需要 2^k-1 个存储单元，这显然造成存储空间的极大浪费。因此，这种顺序存储

① 有的教材定义 $k=\lfloor \log_2 n \rfloor+1$，当 $n>0$ 时与性质 4 的描述等效，当 $n=0$ 时此定义没有意义。