第5章 树和二叉树

CHAPTER

教学重点	二叉树的性质；二叉树和树的存储表示；二叉树的遍历及算法实现；树与二叉树的转换关系；哈夫曼树				
教学难点	二叉树的层序遍历算法；二叉树的建立算法；哈夫曼算法				
教学内容和教学目标	知识点	教学要求			
		了解	理解	掌握	熟练掌握
	树的定义和基本术语			√	
	树和二叉树的抽象数据类型定义		√		
	树和二叉树的遍历				√
	树的存储结构			√	
	二叉树的定义				√
	二叉树的基本性质				√
	二叉树的顺序存储结构		√		
	二叉链表				√
	三叉链表	√			
	二叉树遍历的递归算法				√
	二叉树的层序遍历算法		√		
	二叉树的建立算法			√	
	树、森林和二叉树之间的转换			√	
	哈夫曼树及哈夫曼编码			√	
教学提示	本章的内容由树和二叉树两部分组成，并且由二叉链表存储结构，使得树和二叉树之间具有一一对应关系。对于树的逻辑结构，要从树的定义出发，在与线性表定义比较的基础上，把握要点理解其逻辑特征，通过具体实例识记树的基本术语。对于二叉树的逻辑结构，要从二叉树的定义出发，在与树的定义比较的基础上，理解树和二叉树是两种树结构。对于树和二叉树的存储结构，要以如何表示结点之间的逻辑关系为出发点，掌握不同存储方法以及他们之间的关系。 二叉树的遍历是本章的重点，首先从逻辑上掌握二叉树的遍历操作，然后给出二叉树遍历的递归算法，在理解二叉树遍历执行过程的基础上给出对应的非递归算法。				

5.1 引　　言

前面讨论的数据结构都属于线性结构，主要描述具有单一的前驱和后继关系的数据。树结构是一种比线性结构更复杂的数据结构，比较适合描述具有层次关系的数据，如祖先——后代、上级——下属、整体——部分以及其他类似的关系。许多问题抽象出的数据模型具有层次关系，请看下面两个例子。

【例 5-1】 文件系统。Windows 操作系统的文件目录结构如图 5-1 所示，其中，/表示文件夹，括号内的数字表示文件的大小，单位是 KB，假设文件夹本身的大小是 1KB，如果要统计每个文件和文件夹的大小，即输出如图 5-2 所示的结果，应该如何存储这个文件目录结构并进行统计呢？

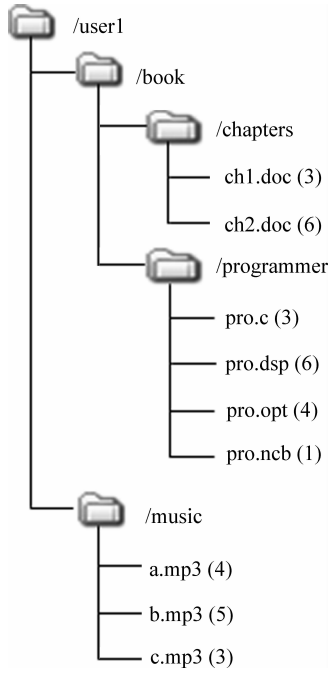


图 5-1　文件系统目录示例

a.mp3 (4)
b.mp3 (5)
c.mp3 (3)
/music (13)
pro.c (3)
pro.dsp (6)
pro.opt (4)
pro.ncb (1)
/programmer (15)
ch1.doc (3)
ch2.doc (6)
/chapters (10)
/book (26)
/user1 (40)

图 5-2　输出结果

【例 5-2】 二叉表示树。一个算术表达式可以用二叉树来表示，这样的二叉树称为二叉表示树。二叉表示树具有以下两个特点：

(1) 叶子结点一定是操作数；

(2) 分支结点一定是运算符。

将一个算术表达式转化为二叉表示树基于如下规则：

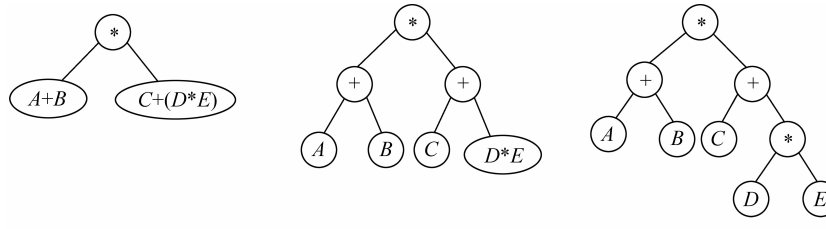
(1) 考虑运算符的优先顺序，将表达式结合成(左操作数 运算符 右操作数)的形式；

(2) 由外层括号开始，运算符作为二叉表示树的根结点，左操作数作为根结点的左子树，右操作数作为根结点的右子树；

(3) 如果某子树对应的操作数为一个表达式，则重复第(2)步的转换，直到该子树对

应的操作数不能再分解。

例如，将表达式(A＋B)＊(C＋D＊E)结合成((A＋B)＊(C＋(D＊E)))，则二叉表示树的根结点是运算符＊，左操作数即左子树是(A＋B)，右操作数即右子树是(C＋(D＊E))。对于左子树(A＋B)，其根结点是运算符＋，左子树是 A，右子树是 B。对于右子树(C＋(D＊E))，其根结点是运算符＋，左子树是 C，右子树是(D＊E)，依此类推。二叉表示树的构造过程如图 5-3 所示。



(a) 过程1　　(b) 过程2　　(c) 最终结果

图 5-3　二叉表示树的构造过程

将算术表达式转换为二叉表示树后，就可以通过遍历二叉树来判断算术表达式是否存在语法错误，并将算术表达式转换为后缀形式(也称逆波兰式)。例如，图 5-3(c)所示二叉表示树的后序遍历序列是 AB＋CDE＊＋＊，该序列对应算术表达式的后缀形式。

5.2　树的逻辑结构

5.2.1　树的定义和基本术语

1. 树的定义

在树中通常将数据元素称为**结点**(node)。

树(tree)是 $n(n\geqslant 0)$个结点的有限集合[①]。当 $n=0$ 时，称为空树；任意一棵非空树满足以下条件：

(1) 有且仅有一个特定的称为**根**(root)的结点；

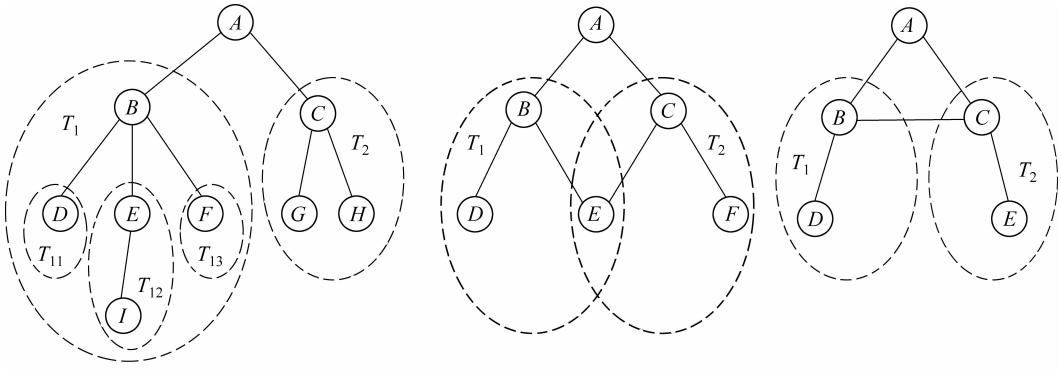
(2) 当 $n>1$ 时，除根结点之外的其余结点被分成 $m(m>0)$个**互不相交**的有限集合 T_1、T_2、…、T_m，其中每个集合又是一棵树[②]，并称为这个根结点的**子树**(subtree)。

图 5-4(a)是一棵具有 9 个结点的树，$T=\{A,B,C,D,E,F,G,H,I\}$，结点 A 为树 T 的根结点。除根结点 A 之外的其余结点分为两个不相交的集合：$T_1=\{B,D,E,F,I\}$和 $T_2=\{C,G,H\}$，T_1 和 T_2 构成了根结点 A 的两棵子树。子树 T_1 的根结点为 B，其余结点又分为三个不相交的集合：$T_{11}=\{D\}$，$T_{12}=\{E,I\}$和 $T_{13}=\{F\}$，T_{11}、T_{12}和 T_{13}构成了根结点 B 的三棵子树。依此类推，直到每棵子树只有一个根结点为止。

---

① 这里定义的树，在数学上属于有根树(rooted tree)。从数学角度讲，树结构是图的特例，无回路的连通图就是树，称为自由树(free tree)。本章讨论的是有根有序树，所谓有序树是指根结点的子树从左到右是有顺序的。

② 显然，树的定义是递归的。由于树结构本身具有递归特性，因此，对树的操作通常采用递归方法。



(a) 树结构示例　　(b) 非树结构示例　　(c) 非树结构示例

图 5-4　树结构和非树结构的示意图

需要强调的是，树中根结点的子树之间是互不相交的。例如，图 5-4(b)由于根结点 A 的两个子树之间存在交集，结点 E 既属于集合 T_1 又属于集合 T_2，所以不是树；图 5-4(c)中根结点 A 的两个子树之间也存在交集，边(B,C)依附的两个结点属于根结点 A 的两个子集 T_1 和 T_2，所以也不是树。

2. 树的基本术语

(1) 结点的度、树的度

某结点所拥有的子树的个数称为该**结点的度**(degree)；树中各结点度的最大值称为该**树的度**。如图 5-4(a)所示的树中，结点 A 的度为 2，结点 B 的度为 3，该树的度为 3。

(2) 叶子结点、分支结点

度为 0 的结点称为**叶子结点**(leaf node)，也称为终端结点；度不为 0 的结点称为**分支结点**(branch node)，也称为非终端结点。如图 5-4(a)所示的树中，结点 D、I、F、G 和 H 是叶子结点，其余结点都是分支结点。

(3) 孩子结点、双亲结点、兄弟结点

某结点的子树的根结点称为该结点的**孩子结点**(children node)；反之，该结点称为其孩子结点的**双亲结点**(parent node)；具有同一个双亲的孩子结点互称为**兄弟结点**(brother node)。如图 5-4(a)所示的树中，结点 B 是结点 A 的孩子，结点 A 是结点 B 的双亲，结点 B 和 C 互为兄弟，结点 I 没有兄弟。

(4) 路径、路径长度

如果树的结点序列 $n_1n_2\cdots n_k$ 满足如下关系：结点 n_i 是结点 n_{i+1}的双亲$(1\leqslant i<k)$，则 $n_1n_2\cdots n_k$ 称为一条由 n_1 至 n_k 的**路径**(path)；路径上经过的边数称为**路径长度**(path length)。显然，在树中路径是唯一的。如图 5-4(a)所示的树中，从结点 A 到结点 I 的路径是 $ABEI$，路径长度为 3。

(5) 祖先、子孙

如果从结点 x 到结点 y 有一条路径，那么 x 就称为 y 的**祖先**(ancestor)，y 称为 x 的**子孙**(descendant)。显然，以某结点为根的子树中的任一结点都是该结点的子孙。如图 5-4(a)所示的树中，结点 A、B、E 均为结点 I 的祖先，结点 B 的子孙有 D、E、F、I。

(6) 结点的层数、树的深度(高度)、树的宽度

规定根结点的**层数**(level)为 1,对其余任何结点,若某结点在第 k 层,则其孩子结点在第 $k+1$ 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。树中每一层结点个数的最大值称为**树的宽度**(breadth)。如图 5-4(a)所示的树中,结点 D 的层数为 3,树的深度为 4,树的宽度为 5。

5.2.2　树的抽象数据类型定义

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。下面给出一个树的抽象数据类型定义的例子,简单起见,基本操作只包含树的遍历,针对具体应用,需要重新定义其基本操作。

```
ADT Tree
DataModel
  树由一个根结点和若干棵子树构成,树中结点具有层次关系
Operation
  InitTree
    输入:无
    功能:初始化一棵树
    输出:一个空树
  DestroyTree
    输入:无
    功能:销毁一棵树
    输出:释放该树占用的存储空间
  PreOrder
    输入:无
    功能:前序遍历树
    输出:树的前序遍历序列
  PostOrder
    输入:无
    功能:后序遍历树
    输出:树的后序遍历序列
  LeverOrder
    输入:无
    功能:层序遍历树
    输出:树的层序遍历序列
endADT
```

5.2.3　树的遍历操作

树结构的最基本操作是**遍历**(traverse)。树的遍历是指从根结点出发,按照某种次序访问树中所有结点,使得每个结点被访问一次且仅被访问一次。“访问”的含义很广,是一种抽象操作,在实际应用中,可以是对结点进行的各种处理,比如输出结点的信息、修改结点的某些数据等,对应到算法上,访问可以是一条简单语句,可以是一个复合语句,也可以是一个模块。不失一般性,在此将访问定义为输出结点的数据信息。树的遍历次序通常

有前序(根)遍历和后序(根)遍历两种方式,此外,如果按树的层序依次访问各结点,则可得到另一种遍历次序:层序遍历。

树的前序遍历操作定义为:若树为空,则空操作返回;否则执行以下操作:

① 访问根结点;

② 按照从左到右的顺序前序遍历根结点的每一棵子树。

树的后序遍历操作定义为:若树为空,则空操作返回;否则执行以下操作:

① 按照从左到右的顺序后序遍历根结点的每一棵子树;

② 访问根结点。

树的层序遍历也称树的广度遍历,其操作定义为从树的根结点开始,自上而下逐层遍历,在同一层中,按从左到右的顺序对结点逐个访问。

例如,图 5-4(a)所示树的前序遍历序列为:$A\ B\ D\ E\ I\ F\ C\ G\ H$,后序遍历序列为:$D\ I\ E\ F\ B\ G\ H\ C\ A$,层序遍历序列为:$A\ B\ C\ D\ E\ F\ G\ H\ I$。

5.3 树的存储结构

在大量的实际应用中,人们使用多种存储方法来表示树。无论采用何种存储方法,都要求存储结构不仅能存储树中各结点的数据信息,还要表示结点之间的逻辑关系——父子关系。下面介绍几种基本的存储方法。

5.3.1 双亲表示法

由树的定义可知,除根结点外,树中每个结点都有且仅有一个双亲结点,根据这一特性,可以用一维数组来存储树的各个结点(一般按层序存储),数组中的一个元素对应树中的一个结点,数组元素包括树中结点的数据信息以及该结点的双亲在数组中的下标。树的这种存储方法称为**双亲表示法**(parent express),数组元素的结构如图 5-5 所示。其中,data 存储树中结点的数据信息;parent 存储该结点的双亲在数组中的下标。

例如,图 5-4(a)所示树采用双亲表示法的存储示意图如图 5-6 所示,图中 parent 域的值为 −1 表示该结点无双亲,即该结点是根结点。

data	parent

图 5-5 双亲表示法的数组元素

下标	data	parent
0	A	−1
1	B	0
2	C	0
3	D	1
4	E	1
5	F	1
6	G	2
7	H	2
8	I	4

图 5-6 双亲表示法的存储示意图

下面给出双亲表示法的存储结构定义：

```
#define MaxSize 100          /* 假设树中最多有 100 个结点 */
typedef char DataType;       /* 定义树结点的数据类型,假设为 char 型 */
typedef struct
{
    DataType data;           /* 树结点的数据信息 */
    int parent;              /* 该结点的双亲在数组中的下标 */
} PNode;
typedef struct               /* 定义双亲表示法存储结构 */
{
    PNode tree[MaxSize];
    int treeNum;             /* 树的结点个数 */
} PTree;
```

5.3.2　孩子表示法

树的**孩子表示法**(child express)是一种基于链表的存储方法，即把每个结点的孩子排列起来，看成是一个线性表，且以单链表存储，称为该结点的孩子链表，则 n 个结点共有 n 个孩子链表(叶子结点的孩子链表为空表)。n 个孩子链表共有 n 个头指针，这 n 个头指针又构成了一个线性表，为了便于进行查找操作，可采用顺序存储。最后，将存放 n 个头指针的数组和存放 n 个结点数据信息的数组结合起来，构成孩子链表的表头数组。所以在孩子表示法中，存在两类结点：孩子结点和表头结点，其结点结构如图 5-7 所示。

(a) 孩子结点　　(b) 表头结点

图 5-7　孩子表示法的结点结构

例如，图 5-8 是图 5-4(a)所示树采用孩子表示法的存储示意图。孩子表示法不仅表示了孩子结点的信息，而且链在同一个孩子链表中的结点具有兄弟关系。

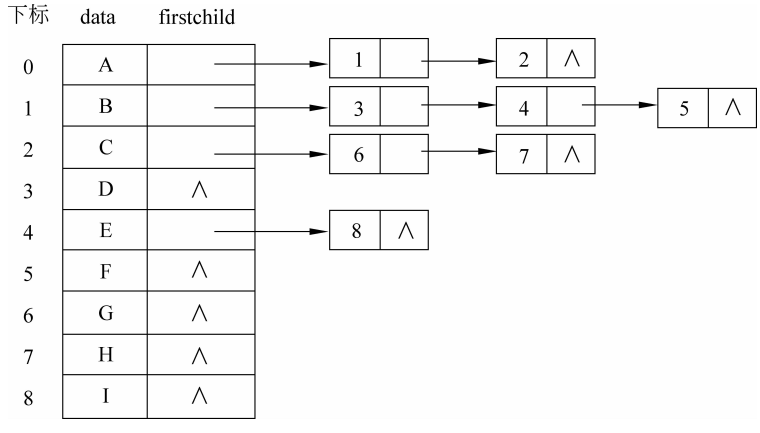


图 5-8　孩子表示法的存储示意图

下面给出孩子表示法的存储结构定义：

```
#define MaxSize 100                 /* 假设树中最多有 100 个结点 */
typedef char DataType;              /* 定义树结点的数据类型,假设为 char 型 */
typedef struct ChildNode            /* 定义孩子结点 */
{
    int child;                      /* 该结点在表头数组中的下标 */
    struct ChildNode *next;
} ChildNode;
typedef struct                      /* 定义表头结点 */
{
    DataType data;
    ChildNode *first;               /* 指向孩子链表的头指针 */
} TreeNode;
typedef struct                      /* 定义孩子表示法存储结构 */
{
    TreeNode tree[Maxsize];
    int treeNum;                    /* 树的结点个数 */
} CTree;
```

5.3.3 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，其方法是链表中的每个结点除数据域外，还设置了两个指针分别指向该结点的第一个孩子和右兄弟，链表的结点结构如图 5-9 所示。其中，data 存储该结点的数据信息；firstchild 存储该结点的第一个孩子结点的存储地址；rightsib 存储该结点的右兄弟结点的存储地址。

例如，图 5-10 是图 5-4(a)所示树采用孩子兄弟表示法的存储示意图，这种存储方法便于实现树的各种操作。例如，若要访问某结点 x 的第 i 个孩子，只需从该结点的第一个孩子指针找到第 1 个孩子后，沿着孩子结点的右兄弟域连续走 $i-1$ 步，便可找到结点 x 的第 i 个孩子。

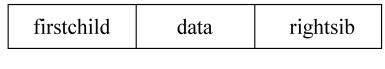


图 5-9 孩子兄弟表示法的结点结构

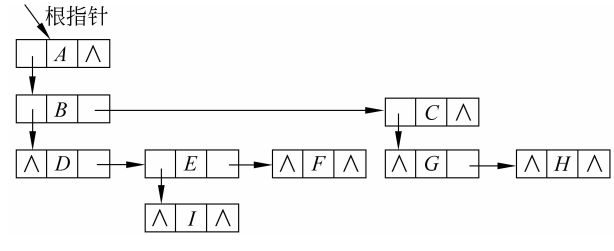


图 5-10 树的孩子兄弟表示法的存储示意图

下面给出孩子兄弟表示法的存储结构定义：

```
typedef char DataType;              /* 定义树结点的数据类型,假设为 char 型 */
typedef struct CSNode               /* 定义孩子兄弟表示法的结点结构 */
{
```

```
    DataType data;
    struct CSNode * firstchild, * rightsib;
} CSNode;
CSNode * root;                    /* 定义根指针 */
```

5.4　二叉树的逻辑结构

二叉树是一种最简单的树结构，而且任何树都可以简单地转换为对应的二叉树。所以，二叉树是本章的重点。

5.4.1　二叉树的定义

二叉树(binary tree)是 $n(n\geqslant 0)$个结点的有限集合，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、分别称为根结点的**左子树**(left subtree)和**右子树**(right subtree)的二叉树组成。观察图 5-11 所示的一棵二叉树，可以发现二叉树具有如下特点：

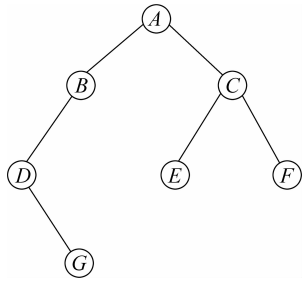
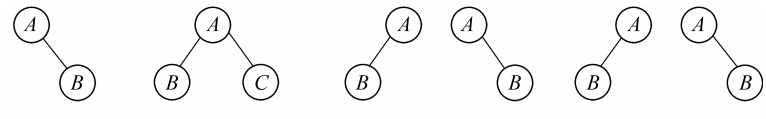


图 5-11　一棵二叉树

(1) 每个结点最多有两棵子树，所以二叉树中不存在度大于 2 的结点；

(2) 二叉树的左右子树不能任意颠倒，如果某结点只有一棵子树，一定要指明它是左子树还是右子树。

需要强调的是，二叉树和树是两种不同的树结构。首先，二叉树不是度为 2 的树。例如图 5-12(a)所示是一棵二叉树，但这棵二叉树的度是 1；假设图 5-12(b)是一棵度为 2 的树，则结点 B 是结点 A 的第一个孩子，结点 C 是结点 A 的第二个孩子，并且可以为结点 A 再增加孩子。其次，树的孩子只有序的关系，即第 1 个孩子，第 2 个孩子，…，第 i 个孩子，但二叉树的孩子却有左右之分，即使二叉树中某结点只有一个孩子，也要区分它是左孩子还是右孩子。例如，假设图 5-12(c)所示是二叉树，则它们是两棵不同的二叉树，假设图 5-12(d)所示是树，则它们是同一棵树。



(a) 二叉树　(b) 度为2的树　(c) 两棵不同的二叉树　(d) 同一棵树

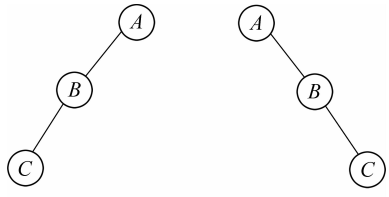
图 5-12　二叉树和树是两种树结构

实际应用中，经常用到如下几种特殊的二叉树。

(1) 斜树

所有结点都只有左子树的二叉树称为**左斜树**(left oblique tree)；所有结点都只有右子树的二叉树称为**右斜树**(right oblique tree)；左斜树和右斜树统称为**斜树**(oblique tree)。斜树的特点是：①每一层只有一个结点；②斜树的结点个数与其深度相同。斜树

示例如图 5-13 所示。

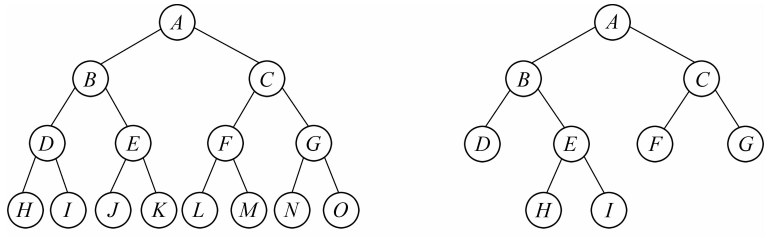


(a) 左斜树　　(b) 右斜树

图 5-13　斜树示例

(2) 满二叉树

在一棵二叉树中，如果所有分支结点都存在左子树和右子树，并且所有叶子都在同一层上，这样的二叉树称为**满二叉树**(full binary tree)。图 5-14(a)是一棵满二叉树，(b)不是满二叉树，因为，虽然所有分支结点都存在左右子树，但叶子不在同一层上。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。

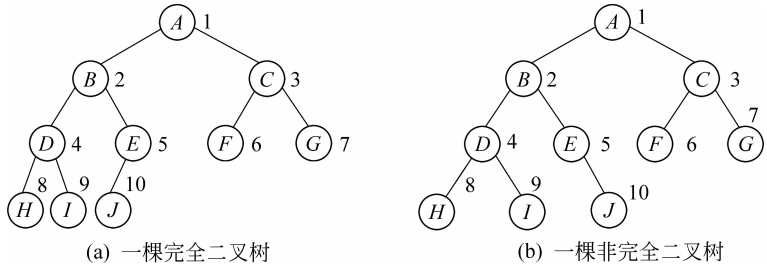


(a) 一棵满二叉树　　(b) 一棵非满二叉树

图 5-14　满二叉树和非满二叉树示例

(3) 完全二叉树

对一棵具有 n 个结点的二叉树按层序编号，如果编号为 $i(1\leqslant i\leqslant n)$的结点与同样深度的满二叉树中编号为 i 的结点在二叉树中的位置完全相同，则这棵二叉树称为**完全二叉树**(complete binary tree)，如图 5-15(a)所示。显然，一棵满二叉树必定是一棵完全二叉树。完全二叉树的特点是：①深度为 k 的完全二叉树在 $k-1$ 层是满二叉树；②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左侧连续的位置；③如果有度为 1 的结点，只可能有一个，且该结点只有左孩子。显然，图 5-15(b)中的树是非完全二叉树。



(a) 一棵完全二叉树　　(b) 一棵非完全二叉树

图 5-15　完全二叉树和非完全二叉树示例