第5章 chapter 5

树

本章导读

树形结构是一种应用很广泛的非线性结构，是一种以分支关系定义的层次结构。本章首先介绍树的定义和表示方法，然后介绍树的遍历等操作，最后介绍树形结构的几个应用实例。

教学目标

本章要求掌握以下内容。

- 树的概念和树的基本操作。
- 二叉树的定义及存储结构。
- 二叉树的遍历。
- 二叉树的应用，包括二叉排序树、哈夫曼树等。
- 树的存储，以及树、森林和二叉树的相互转换。

5.1 树的概述

树形结构是一类重要的非线性结构，是以分支关系定义的层次结构。树形结构在现实生活和计算机领域都有广泛的应用。本节着重介绍树的基本定义和常用术语，以便用户对树形结构有一个全面的理解。

5.1.1 树的定义及基本术语

树是 $n(n\geqslant 0)$ 个结点的有限集合 T。当 $n=0$ 时，称为空树，否则称为非空树。在任一非空树中：

- 有且仅有一个特定的称为根的结点。
- 除根结点外的其余结点被分成 $m(m\geqslant 0)$ 个互不相交的集合 $T_1,T_2,\cdots,T_m$，且其中每个集合本身又是一棵树，它们被称为根的子树。

显然，这是一个递归定义，即在树的定义中又用到树的概念。它反映了树的固有特

性，即树中每个结点都是该树中某一棵子树的根。

在如图 5-1 所示的树 T 中，A 是根结点，其余结点分成 3 个互不相交的子集 $T_1=\{B,E,F\}$，$T_2=\{C,G\}$，$T_3=\{D,H,I,J\}$。T_1、T_2、T_3 都是根结点 A 的子树，B、C、D 分别为这 3 棵子树的根。而子树本身也是树，按照定义可继续划分，如 T_1 中 B 为根结点，其余结点又可分为两个互不相交的子集 $T_{11}=\{E\}$ 和 $T_{12}=\{F\}$，显然 T_{11}、T_{12} 是只含一个根结点的树（对 T_2、T_3，可做类似的划分）。由此可见，树中每一个结点都是该树中某一棵子树的根。

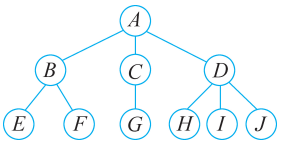


图 5-1　树 T

下面介绍树结构中常用的术语。

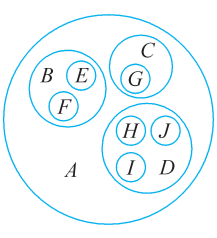
树中结点之间的连线称为分支。结点的子树个数称为结点的度。一棵树中结点度的最大值称为树的度。度为零的结点称为叶子结点或终端结点。度不为零的结点称为分支结点或非终端结点。结点的各子树的根称为该结点的孩子，反之，该结点称为孩子的双亲。同一双亲下的同层结点称为兄弟。将这些关系进一步推广，结点的祖先是从根到该结点所经分支上的所有结点，反之，以该结点为根的子树上的所有结点都是此结点的子孙。例如，如图 5-1 所示的树 T 中，B 的度为 2，是分支结点。E 的度为 0，是叶子结点。树的度为 3。B、C、D 是兄弟。A 是 E 的祖先，B 的子孙是 E、F。

结点的层次是从根结点算起的，设根结点在第一层上，则根结点的孩子为第二层。若某结点在第 L 层，则其子树的根就在第 $L+1$ 层。树中结点的最大层次称为树的高度或深度。图 5-1 中，树 T 的高度为 3。

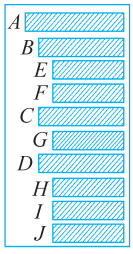
森林是 $n(n\geqslant 0)$ 棵互不相交的树的集合。森林的概念与树非常接近，如果去掉一棵树的根，就得到森林。例如，在图 5-1 中去掉根结点 A，就得到由 3 棵树 T_1、T_2、T_3 组成的森林。反之，给由 n 棵树组成的森林加一个根结点，就生成一棵树。

5.1.2　树的表示

树的表示方法除图 5-1 所示外，还可以用一种表示集合包含关系的文氏图表示，如图 5-2(a)所示；或用凹入法表示，如图 5-2(b)所示；或用广义表的形式表示，根作为由子树森林组成的表的名字写在表的左边，如图 5-2(c)所示。



(a) 文氏图表示法



(b) 凹入表示法

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

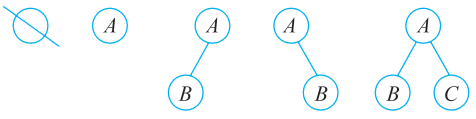
(c) 广义表表示法

图 5-2　树的不同表示法

5.2 二叉树及其遍历

5.2.1 二叉树的定义

二叉树是 $n(n\geqslant 0)$个结点的有限集合，它或为空二叉树($n=0$)，或由一个根结点和两棵分别称为根的左子树和右子树的互不相交的二叉树组成，这是二叉树的递归定义。根据这个定义，可以导出二叉树的 5 种基本形态，如图 5-3 所示。其中，图 5-3(a)为空二叉树，图 5-3(b)为仅有一个根结点的二叉树，图 5-3(c)为右子树为空的二叉树，图 5-3(d)为左子树为空的二叉树，图 5-3(e)为左、右子树均为非空的二叉树。



(a) (b) (c) (d) (e)

图 5-3 二叉树的 5 种形态

5.2.2 二叉树的重要性质

1. 二叉树的性质

二叉树具有下列 3 个重要特性。

性质 1 在二叉树的第 i 层上，至多有 2^{i-1}个结点($i\geqslant 1$)。

可利用归纳法证明性质 1 的正确性。根据结点层次的定义，二叉树的根结点在第一层上，当 $i=1$ 时，只有一个根结点，$2^{i-1}=2^0=1$，则上述结论成立。若第 $j-1$ 层上有 2^{j-2} 个结点($1\leqslant j\leqslant i$)，由于二叉树中每个结点至多有两个孩子结点，若其结点在第 $j-1$ 层，则孩子结点必在第 j 层，故在第 j 层上最多有 $2\times 2^{j-2}=2^{j-1}$ 个结点。由此，结论成立。

性质 2 高度为 k 的二叉树中至多含 2^k-1 个结点($k\geqslant 1$)。

由性质 1 可见，高度为 k 的二叉树的最大结点数为

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

性质 3 在任意一棵二叉树 T 中，若其叶子结点数为 n_0，度为 1 的结点数为 n_1，度为 2 的结点数为 n_2，由于二叉树中所有结点的度均小于或等于 2，所以其结点总数为

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

二叉树中除根结点之外的每个结点都有一个指向其双亲结点的分支，则分支数 B 和结点总数 n 之间存在如下关系：

$$n=B+1 \tag{5-2}$$

从另一个角度看，这些分支可看成度为 1 的结点和度为 2 的结点与它们的孩子结点

之间的连线，则分支数 B 和 n_1 及 n_2 之间存在下列关系：

$$B = n_1 + 2n_2$$

代入式(5-2)得

$$n = n_1 + 2n_2 + 1 \tag{5-3}$$

由式(5-1)和式(5-3)，可得

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

化简得

$$n_0 = n_2 + 1$$

2. 完全二叉树和满二叉树的性质

完全二叉树和满二叉树是两种特殊形态的二叉树。

满二叉树：一棵高度为 k 且含有 $2^k - 1$ 个结点的二叉树称为满二叉树。对一棵满二叉树，若从第 1 层的根结点开始，自上而下、从左到右地对结点进行连续编号，则可给出满二叉树的顺序表示法，如图 5-4 所示。

完全二叉树：若高度为 k、有 n 个结点的二叉树是一棵完全二叉树，且当且仅当每个结点都与高度为 k 的满二叉树中编号从 1 到 n 的结点一一对应，则称该二叉树为完全二叉树，如图 5-5 所示。完全二叉树的特点是：除最下面一层外，每一层的结点个数都达到最大值，且最下面一层的结点都集中在该层最左边的若干位置。显然，一棵满二叉树一定是完全二叉树，但一棵完全二叉树不一定是满二叉树。

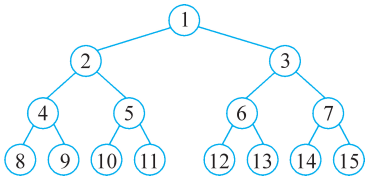


图 5-4　满二叉树

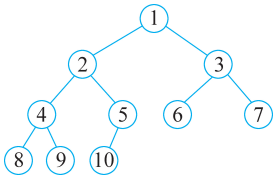


图 5-5　完全二叉树

完全二叉树具有以下两个性质。

性质 4　如果对一棵有 n 个结点的完全二叉树的结点按顺序编号，则任一结点 $I(1 \leqslant i \leqslant n)$ 有以下特性。

- 若 $i \neq 1$，则 i 的双亲结点是结点 $[i/2]$；若 $i = 1$，则 i 是根结点，无双亲。
- 若 $2i \leqslant n$，则 i 的左孩子是结点 $2i$；若 $2i > n$，则 i 无左孩子。
- 若 $2i + 1 \leqslant n$，则 i 的右孩子是结点 $2i + 1$；若 $2i + 1 > n$，则 i 无右孩子。

性质 5　具有 n 个结点的完全二叉树的高度为 $[\log_2 n] + 1$。

证明：假设高度为 k，则根据性质 2 和完全二叉树的定义，有

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

或

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

于是

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

由于 k 为整数，所以

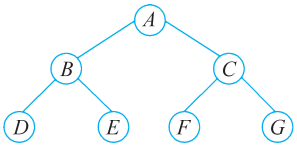
$$k = [\log_2 n] + 1$$

5.2.3 二叉树的存储结构

1. 顺序存储结构

存储二叉树时，是用一组连续的存储单元存储二叉树的数据元素，按满二叉树的结点顺序编号依次存放二叉树中的数据元素。用一维数组 T 存放二叉树时，如图 5-6 所示。

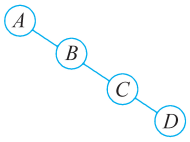
这种存储结构适用于存放完全二叉树和满二叉树。但对一般二叉树，这种存储结构会造成内存的浪费。如图 5-7 所示，在最坏的情况下，一个高度为 k 且只有 k 个结点的单支树（二叉树中没有度为 2 的结点），却需要 2^k-1 个存储单元。可见，此时二叉树使用顺序存储结构，会浪费较多的存储空间。另外，顺序分配时的插入和删除操作是很不方便的，会造成大量结点的移动。因此，二叉树通常采用链式存储结构。



T[7]	0	1	2	3	4	5	6
	A	B	C	D	E	F	G

用一维数组存放满二叉树

图 5-6 满二叉树的顺序存储结构



T[15]	0	1	2	3	4	5	6	7	…	14
	A		B				C		…	D

用一维数组存放一般二叉树

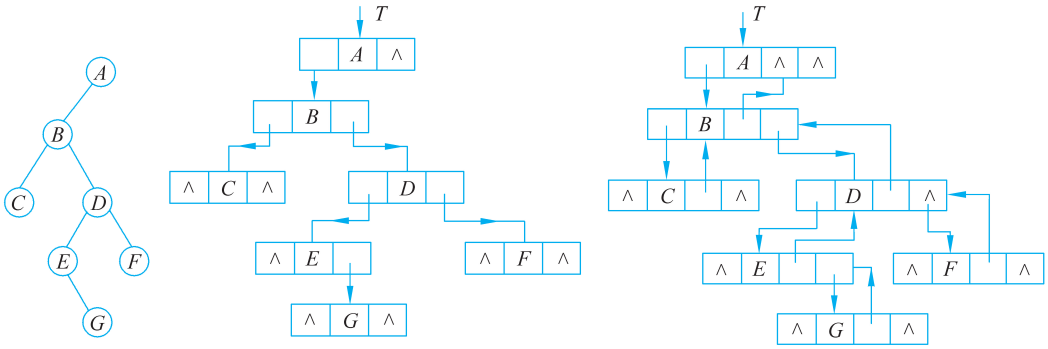
图 5-7 一般二叉树的顺序存储结构

2. 链式存储结构

由于二叉树的每个结点最多可有左、右两棵子树，故链表的结点结构除数据域外，还可设两个链域：左孩子域（lchild）、右孩子域（rchild），分别指向其左、右孩子。结点由两个链域组成的链表称为二叉链表。但有时为了便于找到双亲结点，还要另设一个指向双亲的链域，即结点由 3 个链域组成，此链表称为三叉链表。二叉树 T 的二叉链表表示及三叉链表表示如图 5-8 所示。

3. 建立二叉树

建立二叉树的方法有很多，这里介绍一个基于前序遍历的构造方法。算法输入的是二叉树的扩充前序序列，即在前序序列中加入空指针。若用“#”表示空指针，要建立如图 5-9 所示的二叉树，其输入的扩充前序序列为（$-*a\#\#b\#\#c\#\#$），即可建立相应的二叉链表，用 Python 语言描述的算法如下。



(a) 二叉树　　(b) 二叉链表表示　　(c) 三叉链表表示

图 5-8　二叉树的链表表示示例

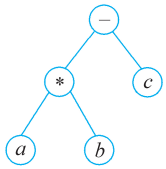


图 5-9　二叉树

例 5-1

```
class TreeNode():                                    #二叉树结点
  def __init__(self,val,lchild=None,rchild=None):
    self.val=val                                     #二叉树的结点值
    self.lchild=lchild                               #左孩子
    self.rchild=rchild                               #右孩子
def Creat_Tree(Root,val):
  if len(val)==0:#终止条件:val 列表中的数据用完了
    return Root
  if val[0]!='#':#构建 Root、Root.lchild、Root.rchild 3 个结点
    Root = TreeNode(val[0])
    vals.pop(0)
    Root.lchild = Creat_Tree(Root.lchild,val)
    Root.rchild = Creat_Tree(Root.rchild,val)
    return Root#本次递归要返回给上一次本层构造好的树的根结点
  else:
    Root=None
    vals.pop(0)
    return Root#本次递归要返回给上一次本层构造好的树的根结点

if __name__ == '__main__':
  Root = None
  strs="- * a##b##c##"#前序遍历扩展的二叉树序列
```

```
  vals = list(strs)
  #递归创建二叉树
  print("程序构建由前序序列:\n%s\n 构建的二叉树。\n" %vals)
  Root=Creat_Tree(Root,vals)#Root 就是要构建的二叉树的根结点
```

上述程序的运行结果如下所示。

```
程序构建由前序序列:
['-', '*', 'a', '#', '#', 'b', '#', '#', 'c', '#', '#']
构建的二叉树。
```

从程序运行结果可以看出，没有任何数据显示给我们，这是因为这一切操作都是在计算机内存中完成的，如果想看到我们建立的二叉树呈现出来，就需要对二叉树按照某种方式访问所建立的结点数据，形成有序序列，这就是二叉树的遍历。

5.2.4 二叉树的遍历

遍历二叉树是指按一定的规律访问二叉树的每个结点，且每个结点仅被访问一次（访问结点可理解为打印该结点的数据域值或其他操作）。遍历是二叉树最重要和最基本的运算，并且有很多实际的应用。遍历对线性结点来说，是一个容易解决的问题。由于二叉树是一个非线性结构，每个结点都可能有两棵子树，所以要找到一个完整的、有规律的走法，以使二叉树上的结点按被访问的先后顺序排列起来，得到一个线性序列。

分析二叉树的结构特性可知，一棵非空二叉树由根结点、左子树、右子树 3 个基本部分组成。若分别令 D、L 和 R 表示访问根结点、遍历左子树和遍历右子树，则可有 DLR、LDR、LRD、DRL、RDL、RLD 6 种遍历次序。若在左、右子树的遍历次序上限定先左后右，则仅有前 3 种情况，分别称为前序遍历、中序遍历、后序遍历。

1. 前序遍历

前序遍历的递归定义可描述为：若二叉树不空，则进行下列操作。

- 访问根结点。
- 前序遍历左子树。
- 前序遍历右子树。

若定义二叉树的存储结构为二叉链表，则根据前序遍历的递归定义，可以写出相应的 Python 语言算法。

例 5-2

```
class TreeNode():#二叉树结点
  def __init__(self,val,lchild=None,rchild=None):
    self.val=val                                    #二叉树的结点值
    self.lchild=lchild                              #左孩子
    self.rchild=rchild                              #右孩子
def Creat_Tree(Root,val):
```

```
    if len(val)==0:#终止条件:val 列表中的数据用完了
        return Root
    if val[0]!='#':#构建 Root、Root.lchild、Root.rchild 3 个结点
        Root = TreeNode(val[0])
        vals.pop(0)
        Root.lchild = Creat_Tree(Root.lchild,val)
        Root.rchild = Creat_Tree(Root.rchild,val)
        return Root#本次递归要返回给上一次本层构造好的树的根结点
    else:
        Root=None
        vals.pop(0)
        return Root#本次递归要返回给上一次本层构造好的树的根结点

def preOrderTraversal(root):
    if root is None:
        return
    #输出当前访问的根结点值
    print(root.val, end=' ')
    #按前序遍历左子树
    preOrderTraversal(root.lchild)
    #按前序遍历右子树
    preOrderTraversal(root.rchild)

if __name__ == '__main__':
    Root = None
    strs="-*a##b##c##"#前序遍历扩展的二叉树序列
    vals = list(strs)
    #递归创建二叉树
    print("程序构建由前序序列:\n%s\n 构建的二叉树。\n" %vals)
    Root=Creat_Tree(Root,vals)#Root 就是要构建的二叉树的根结点
    print("二叉树的前序遍历结果为:\n")
    preOrderTraversal(Root)
```

例如图 5-9 所示的二叉树，若按前序遍历的方法，则输出结点的序列是－ * *abc* 。上述程序的运行结果如下所示。

```
程序构建由前序序列:
['-', '*', 'a', '#', '#', 'b', '#', '#', 'c', '#', '#']
构建的二叉树。
二叉树的前序遍历结果为:
- * a b c
```

2. 中序遍历

中序遍历的递归定义是：若二叉树不空，则进行下列操作。

- 按中序遍历左子树。
- 访问根结点。
- 按中序遍历右子树。

用 Python 语言描述的算法如下。

例 5-3

```
class TreeNode():                                    #二叉树结点
  def __init__(self,val,lchild=None,rchild=None):
    self.val=val                                     #二叉树的结点值
    self.lchild=lchild                               #左孩子
    self.rchild=rchild                               #右孩子
def Creat_Tree(Root,val):
  if len(val)==0:#终止条件:val 列表中的数据用完了
    return Root
  if val[0]!='#':#构建 Root、Root.lchild、Root.rchild 3 个结点
    Root = TreeNode(val[0])
    vals.pop(0)
    Root.lchild = Creat_Tree(Root.lchild,val)
    Root.rchild = Creat_Tree(Root.rchild,val)
    return Root#本次递归要返回给上一次本层构造好的树的根结点
  else:
    Root=None
    vals.pop(0)
    return Root#本次递归要返回给上一次本层构造好的树的根结点

def inOrderTraversal(root):
    if root is None:
        return

    #按中序遍历左子树
    inOrderTraversal(root.lchild)
    #输出当前访问的根结点值
    print(root.val, end=' ')
    #按中序遍历右子树
    inOrderTraversal(root.rchild)

if __name__ == '__main__':
  Root = None
  strs="-*a##b##c##"#前序遍历扩展的二叉树序列
  vals = list(strs)
  #递归创建二叉树
```

```
  print("程序构建由前序序列:\n%s\n 构建的二叉树。\n" %vals)
  Root=Creat_Tree(Root,vals)#Root 就是要构建的二叉树的根结点
  print("二叉树的中序遍历结果为:\n")
  inOrderTraversal(Root)
```

若对图 5-9 所示的二叉树执行中序遍历，则输出的结点序列为 a * b－c。上述程序的运行结果如下所示。

```
程序构建由前序序列:
['-', '*', 'a', '#', '#', 'b', '#', '#', 'c', '#', '#']
构建的二叉树。
二叉树的中序遍历结果为:
a * b - c
```

3. 后序遍历

后序遍历的递归定义为：若二叉树不空，则执行如下操作。

- 按后序遍历左子树。
- 按后序遍历右子树。
- 访问根结点。

用 Python 语言描述的算法如下。

例 5-4

```
class TreeNode():                                    #二叉树结点
  def __init__(self,val,lchild=None,rchild=None):
    self.val=val                                     #二叉树的结点值
    self.lchild=lchild                               #左孩子
    self.rchild=rchild                               #右孩子
def Creat_Tree(Root,val):
  if len(val)==0:#终止条件:val 列表中的数据用完了
    return Root
  if val[0]!='#':#构建 Root、Root.lchild、Root.rchild 3 个结点
    Root = TreeNode(val[0])
    vals.pop(0)
    Root.lchild = Creat_Tree(Root.lchild,val)
    Root.rchild = Creat_Tree(Root.rchild,val)
    return Root#本次递归要返回给上一次本层构造好的树的根结点
  else:
    Root=None
    vals.pop(0)
    return Root#本次递归要返回给上一次本层构造好的树的根结点

#后序遍历,左右根
def postOrderTraversal(root):
```