第5章　树

本章学习目标

- 理解树的基本概念：包括树的定义、术语(如结点、边、根、叶、子树等)。
- 掌握树的性质：例如，每个结点最多只有一个父结点，除了根结点外。
- 学习树的遍历：前序遍历、中序遍历、后序遍历和层序遍历。
- 了解不同类型的树：如二叉树、平衡二叉树、二叉搜索树、AVL 树、红黑树、B 树等。
- 学习树的应用：如何使用树解决实际问题，例如，在搜索、排序、索引构建等方面的应用。

树，作为计算机科学中一种基础而关键的数据结构，以其独特的层次结构在组织数据和管理信息中发挥着重要作用。从文件系统到网络通信，从决策支持到抽象概念的表示，树结构的应用无处不在。树由结点组成，每个结点有零个或多个子结点，并且有一个父结点，除了根结点以外。这种结构不仅体现了数据的层次关系，而且支持高效的数据检索和操作。

本章将深入探讨树这一数据结构的丰富内涵和广泛应用，不仅从技术层面介绍其定义、术语和性质，更将融入课程元素，以全新的视角展现树的深远意义。

首先，从树的基本概念出发，理解其结构和特性，这不仅是对自然现象的模拟，也是对人类社会结构的一种抽象。树的层次分明和有序性，可以引导我们思考如何在社会中建立合理的组织架构，促进和谐与效率。接着，深入探讨二叉树及其变种，如完全二叉树、平衡二叉树和二叉搜索树等。这些结构不仅在计算机科学中有着广泛的应用，也象征着社会中的平衡与秩序。通过学习这些树的变种，能够体会到在复杂系统中寻求平衡与效率的重要性。在树的存储结构部分，学习如何使用数组和链表来存储树，这不仅是一种技术实现，也是对资源合理分配和优化利用的思考。这种思想可以启发在社会资源配置中寻求最优化的解决方案。深入分析树的遍历算法，如前序、中序、后序遍历，帮助理解不同问题解决策略的多样性和适用性。这可以类比社会问题的处理，不同的方法可能适用于不同的情境，需要灵活运用。

此外，本章还将介绍树的转换、动态查询等。这些内容不仅锻炼了我们的逻辑思维和解决问题的能力，也体现了在面对复杂问题时，如何通过创新思维找到解决方案。

通过本章的学习，不仅能够让读者对树数据结构有一个全面的认识，更能够在教育的引导下，学会将技术知识与社会责任相结合，培养出既有专业技能又有社会责任感的新时代人才。

5.1　树和二叉树

在计算机科学中，树和二叉树是两种非常重要的数据结构，它们在组织和存储数据方面扮演着关键角色。以下是树和二叉树的定义，以及一些相关术语的简要介绍。

5.1.1 树的定义与基本术语

1. 树的定义

树(Tree)是一种抽象数据类型,它由结点(或称为顶点)组成,每个结点有零个或多个子结点,并且有一个特定的结点被称为根结点。树中的结点通过边相连,表示结点之间的层次关系。树的特点是不存在环,且任意两个结点之间只有一条唯一的路径。

2. 树的基本术语

在树的数据结构中,除了前面提到的基本术语,还有一些其他重要的概念,这些概念帮助我们更好地理解和描述树中结点之间的关系,如图 5.1 所示。

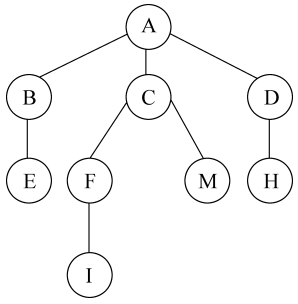

图 5.1 树

堂兄弟结点(Cousins):如果两个结点在树中有相同的深度,但不是直接的兄弟(即它们没有共同的父结点),则这两个结点被称为堂兄弟结点。

祖先结点(Ancestor):从根到某个结点的路径上的所有结点,包括该结点本身,都被称为该结点的祖先。

子孙结点(Descendant):如果一个结点在另一个结点的路径上,那么这个结点被称为另一个结点的子孙结点。

树的层(Level):树中的结点可以根据它们距离根结点的边数被分为不同的层。根结点位于第 1 层,它的直接子结点位于第 2 层,以此类推。

满二叉树(Full Binary Tree):如果除了最后一层外,每一层的结点数都达到最大,并且最后一层的结点尽可能地靠左排列,则该树被称为满二叉树。

完全二叉树(Complete Binary Tree):如果所有叶子结点都在最后两层,并且最后一层的结点尽可能地靠左排列,则该树被称为完全二叉树。

外部结点(External Node):没有子结点的结点,也称为叶子结点。

内部结点(Internal Node):至少有一个子结点的结点。

结点的度(Degree of a Node):一个结点的度是指它拥有的子结点数量。

树的度(Degree of a Tree):树的度是指树中度最大的结点的度。

有序树(Ordered Tree):如果树中每个结点的子结点都有一个顺序,即从左到右,那么这棵树被称为有序树。

无序树(Unordered Tree):如果树中每个结点的子结点没有顺序,即子结点的顺序不重要,那么这棵树被称为无序树。

树的森林(Forest of Trees):一组树的集合,其中每棵树都是独立的,没有结点属于多棵树。

这些术语帮助我们更精确地描述和讨论树的结构和性质,对于实现和分析树算法非常重要。

5.1.2 二叉树的定义与特点

1. 二叉树

二叉树(Binary Tree)是一种特殊的树,其中每个结点最多有两个子结点,通常称为左

子结点和右子结点。二叉树的子结点数量可以是 0、1 或 2。

2. 二叉树的性质

（1）左子树上的所有结点的值都小于或等于其父结点的值（左子树是有序的）。

（2）右子树上的所有结点的值都大于或等于其父结点的值（右子树也是有序的）。

3. 二叉树的类型

（1）满二叉树（Full Binary Tree）：除了最后一层外，每一层都被完全填满的二叉树。

（2）完全二叉树（Complete Binary Tree）：最后一层的左侧被尽可能多的结点填满的二叉树。

（3）平衡二叉树（Balanced Binary Tree）：任何两个叶子结点的深度差异不超过 1 的二叉树。

5.1.3 树与二叉树的示例描述

对树和二叉树的简单表示方法，使用 Python 代码段描述如下。

```
#树的简单表示
class TreeNode:
def __init__(self, value):
self.value = value
self.children = []
def add_child(self, child):
self.children.append(child)
#二叉树的简单表示
class BinaryTreeNode:
def __init__(self, value):
self.value = value
self.left = None
self.right = None
def set_left(self, left_child):
self.left = left_child
def set_right(self, right_child):
self.right = right_child
```

在上述代码中，TreeNode 类表示一个树结点，它可以有任意数量的子结点。而 BinaryTreeNode 类表示一个二叉树结点，它最多有两个子结点。这些类提供了创建和连接树或二叉树结点的基本功能。

5.2 二叉树案例引入

（1）背景介绍。

二叉树是一种特殊的二叉树，它具有以下性质。

① 若任意结点的左子树不空，则左子树上所有结点的值均小于它的结点值。

② 若任意结点的右子树不空，则右子树上所有结点的值均大于或等于它的结点值。

③ 任意结点的左、右子树也分别为二叉搜索树。

（2）案例描述。

假设你正在开发一个用于存储和检索大量数据的应用程序，如一个在线字典或一个音乐库。你需要一种数据结构，可以快速地插入新数据、查找特定数据以及删除数据。二叉搜索树是解决这些问题的理想选择。

1. 案例分析

1）插入操作

当你向二叉搜索树中插入一个新的数据项时，从根结点开始，如果数据项小于当前结点的值，则移动到左子结点；如果数据项大于或等于当前结点的值，则移动到右子结点。重复这个过程，直到找到一个空位置插入新结点。

2）查找操作

查找过程与插入过程类似。从根结点开始，比较目标值与当前结点的值。如果目标值较小，继续在左子树上查找；如果目标值较大或相等，继续在右子树上查找。如果到达叶子结点仍未找到，则目标值不在树中。

3）删除操作

删除操作稍微复杂一些，需要考虑三种情况：删除叶子结点、删除只有一个子结点的结点和删除有两个子结点的结点。对于每种情况，都有相应的策略来保持二叉搜索树的性质。

2. 案例实现

以下是使用 Python 代码段实现二叉搜索树的一个简单示例。

```
class BSTNode:
def __init__(self, key):
self.left = None
self.right = None
self.key = key
def insert(root, key):
if root is None:
return BSTNode(key)
else:
if key < root.key:
root.left = insert(root.left, key)
else:
root.right = insert(root.right, key)
return root
def search(root, key):
if root is None or root.key == key:
return root
if key < root.key:
return search(root.left, key)
return search(root.right, key)
#创建一个 BST 并插入一些结点
root = None
keys = [20, 8, 22, 4, 12, 10, 14]
for key in keys:
root = insert(root, key)
#搜索一个键值
search_key = 10
found = search(root, search_key)
```

```
if found:
print(f"Key {search_key} found in the BST.")
else:
print(f"Key {search_key} not found in the BST.")
```

通过这个示例，读者不仅能够理解二叉搜索树的基本概念和性质，而且能够看到它是如何在实际应用中发挥作用的。这个示例可以进一步扩展，包括删除操作的实现和对二叉搜索树性能的讨论。

5.3 二叉树的性质和存储结构

二叉树作为数据结构中的一个重要分支，以其独特的结构和性质在计算机科学领域扮演着关键角色。它是一种特殊的树，每个结点最多有两个子结点，通常称为左子结点和右子结点。二叉树不仅在数据存储和组织方面表现出色，而且在实现算法如搜索、排序和遍历等方面具有显著优势。本节将深入探讨二叉树的基本性质，包括它的递归定义、树的遍历方法以及二叉树在不同场景下的应用。同时，还将讨论二叉树的存储结构，包括数组和指针的使用方法，以及它们如何影响二叉树的效率和操作。通过对二叉树性质和存储结构的理解，读者将能够更加高效地使用这种数据结构来解决实际问题。

5.3.1 二叉树的性质

性质 1 在二叉树的第 i 层至多有 $2(i-1)$ 个结点（$i \geqslant 1$）。用数学归纳法证明方法如下。

证明：当 $i=1$ 时，只有根结点 $2^{(i-1)}=2^0=1$。

（1）假设：对所有 j，$i>j \geqslant 1$，命题成立，即第 j 层上至多有 $2^{(j-1)}$ 个结点。

（2）由归纳假设第 $i-1$ 层上至多有 $2^{(i-2)}$ 个结点。

（3）由于二叉树的每个结点的度至多为 2，故在第 i 层上的最大结点数为第 $i-1$ 层上的最大结点数的 2 倍，即 $2\times 2^{(i-2)}=2^{(i-1)}$。

证毕。

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：由性质 1 可见，深度为 k 的二叉树的最大结点数为

$$\begin{aligned}&\sum_{i=1\cdots k} \text{第 } i \text{ 层上的最大结点数}\\ &=\sum_{i=1} 2^{(i-1)}\\ &=2^0+2^1+\cdots 2^{(k-1)}\\ &=2^k-1\end{aligned}$$

性质 3 对任何一棵二叉树 T，如果其叶子结点数为 n_0，度为 2 的结点数为 n_2，则 $n_0=n_2+1$。

证明：若度为 1 的结点有 n_1 个，总结点个数为 n，总边数为 e，则根据二叉树的定义，

$$n=n_0+n_1+n_2$$

$$e=2n_2+n_1=n-1\text{（除了根结点，每个结点对应一条边）}$$

因此，有

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

$$n_2 = n_0 - 1 \geqslant n_0 = n_2 + 1$$

性质 4 具有 $n(n \geqslant 0)$ 个结点的完全二叉树的深度为 $\lfloor \log_2 n \rfloor + 1$。

证明：设完全二叉树的深度为 h，则根据性质 2 和完全二叉树的定义有

$$2^{(h-1)} - 1 < n \leqslant 2^h - 1 \quad 或 \quad 2^{(h-1)} \leqslant n < 2^h$$

取对数 $h-1 < \log_2 n \leqslant h$，又因 h 是整数，因此有 $h = \lfloor \log_2 n \rfloor + 1$。

完全二叉树和满二叉树是二叉树的两种特殊类型，它们的定义如下。

完全二叉树：如果在一棵二叉树中，除了最后一层外，每一层都被完全填满，并且在最后一层中，所有的结点尽可能地靠左排列，那么这棵树就是完全二叉树，如图 5.2 所示。

满二叉树：如果在一棵二叉树中，除了最后一层外，每一层都被完全填满，并且最后一层的所有结点也都被填满，那么这棵树就是满二叉树，如图 5.3 所示。

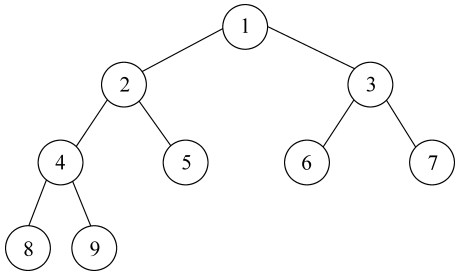

图 5.2 完全二叉树

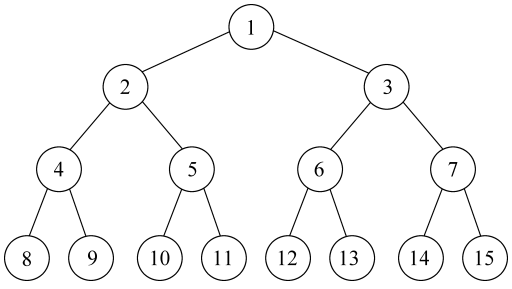

图 5.3 满二叉树

性质 5 如将一棵有 n 个结点的完全二叉树自顶向下，同层自左向右连续为结点编号为 $0,1,\cdots,n-1$，则有：

(1) 若 $i=0$，则 i 无双亲，若 $i>0$，则 i 的双亲为 $\lfloor (i-1)/2 \rfloor$。

(2) 若 $2i+1<n$，则 i 的左子女为 $2i+1$，若 $2i+2<n$，则 i 的右子女为 $2i+2$。

(3) 若结点编号 i 为偶数，且 $i!=0$，则左兄弟结点为 $i-1$。

(4) 若结点编号 i 为奇数，且 $i!=n-1$，则右兄弟结点为 $i+1$。

(5) 结点 i 所在层次为 $\lfloor \log_2{}^{(i+1)} \rfloor$。

5.3.2 二叉树的存储结构

二叉树是一种常见的数据结构，用于表示具有层次结构的数据。在计算机中，二叉树可以通过两种主要的存储结构来实现：顺序存储和链式存储。下面将详细介绍这两种存储结构。

1. 顺序存储(数组表示)

顺序存储结构，也称为数组表示，是使用数组来存储二叉树的一种方式。在这种结构中，二叉树的结点被存储在一个一维数组中，通常按照层次顺序排列。

特点如下。

(1) 顺序存储结构简单，易于实现。

(2) 它允许通过索引直接访问任何结点。

（3）但是，如果二叉树不是完全二叉树，这种结构会浪费空间，因为数组的大小需要足够大以容纳所有可能的结点。

实现如下。

（1）假设二叉树的根结点存储在数组的第一个位置（索引 0）。

（2）对于任意结点 i，其左子结点的索引为 $2i+1$，右子结点的索引为 $2i+2$。

（3）如果结点 i 是叶子结点，那么其子结点的索引将超出数组的界限。

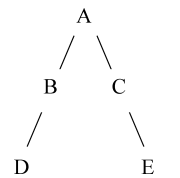

图 5.4　二叉树

假设有一棵二叉树如图 5.4 所示。

使用顺序存储结构，可以表示为数组：[A,B,C,D,E]。

2. 链式存储（指针表示）

链式存储结构，也称为指针表示，是通过指针（或引用）来链接二叉树结点的一种方式。在这种结构中，每个结点包含数据以及指向其子结点和（可选的）父结点的指针。

特点如下。

（1）链式存储结构允许灵活地表示任意形状的二叉树，包括完全二叉树、满二叉树和不平衡二叉树。

（2）它不浪费空间，因为只需要为实际存在的结点分配内存。

（3）但是，它需要额外的内存来存储指针，并且访问特定结点可能需要遍历树。

实现如下。

（1）每个结点通常由一个结构体或类表示，包含三个主要部分：数据域、左子结点指针和右子结点指针。

（2）可以使用链表、动态数组或其他数据结构来维护结点之间的链接。

假设有二叉树如图 5.4 所示。

在 Python 中，通常使用类来定义数据结构，如二叉树结点。Python 中的类定义和C++有一些不同，但概念是相似的。以下是使用 Python 代码段描述如何表示相同的链式存储结构的二叉树。

```
class TreeNode:
def __init__(self, data):
self.data = data
self.left = None
self.right = None
#使用 Python 创建二叉树
root = TreeNode('A')
root.left = TreeNode('B')
root.right = TreeNode('C')
root.left.left = TreeNode('D')
root.left.right = TreeNode('E')
```

在这个 Python 代码示例中，定义了一个名为 TreeNode 的类，它具有一个初始化方法 __init__，该方法接收一个 data 参数，并设置 left 和 right 属性为 None，表示初始时结点没有子结点。

然后，我们创建了树的根结点 root，并为它以及它的子结点赋值，就像在 C++ 中使用 new 操作符创建结点一样。在 Python 中，不需要显式地使用类似 new 的操作符来创建对

象；只须调用类名并传递相应的参数即可。

这样，我们就用 Python 代码段表示了一个与 C/C++ 中相同的链式存储结构的二叉树。

二者的比较如下。

(1) 空间效率：链式存储通常比顺序存储更节省空间，因为它只为实际存在的结点分配内存。

(2) 时间效率：顺序存储允许快速随机访问，而链式存储可能需要线性时间来访问特定结点。

(3) 灵活性：链式存储提供了更高的灵活性，可以轻松地添加或删除结点。

在选择二叉树的存储结构时，需要根据实际应用的需求来权衡这些因素。例如，如果需要频繁访问特定结点，顺序存储可能是更好的选择；如果树的形状变化很大，链式存储可能更合适。

5.4 遍历二叉树和线索二叉树

在数据结构领域，二叉树的遍历是理解和操作二叉树的基础。本节将深入探讨这一主题，包括二叉树的各种遍历方法，如前序遍历、中序遍历、后序遍历和层序遍历。这些遍历方法对于算法设计和程序开发至关重要，因为它们允许系统地访问树中的所有结点，从而实现不同的操作，如搜索、排序和树结构的重建。

此外，本节还将介绍线索二叉树，这是一种特殊的二叉树，其中每个结点的空指针被替换为指向其他结点的线索。这种结构简化了某些类型的遍历，并提高了访问效率，尤其是在需要频繁进行特定类型遍历的场合。线索二叉树的引入，不仅丰富了二叉树的应用场景，也为算法优化提供了新的视角。

通过本节的学习，读者将能够掌握二叉树遍历的基本概念和技巧，理解线索二叉树的设计原理及其优势，从而在实际问题解决中更加得心应手。

5.4.1 遍历二叉树

1. 遍历二叉树算法描述

二叉树的遍历是递归地访问树中的每个结点的过程。遍历可以按照不同的顺序进行，每种顺序都有其特定的应用场景。以下是三种主要的遍历方法：先序、中序和后序，以及它们的语法定义和案例。

(1) 先序遍历(Pre-order Traversal)。

语法定义：先序遍历首先访问根结点，然后递归地进行先序遍历左子树，最后递归地进行先序遍历右子树。

遍历顺序：根-左-右。

假设有如下二叉树如图 5.5 所示。

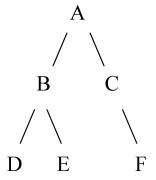

图 5.5 二叉树

先序遍历的结果将是：A,B,D,E,C,F。

(2) 中序遍历(In-order Traversal)。

语法定义：中序遍历首先递归地进行中序遍历左子树，然后访问根结点，最后递归地进行中序遍历右子树。

遍历顺序：左-根-右。

使用相同的二叉树如图 5.6 所示。

中序遍历的结果将是：D,B,E,A,C,F。

(3) 后序遍历(Post-order Traversal)。

语法定义：后序遍历首先递归地进行后序遍历左子树，然后递归地进行后序遍历右子树，最后访问根结点。

遍历顺序：左-右-根。

使用相同的二叉树如图 5.7 所示。

后序遍历的结果将是：D,E,B,F,C,A。

使用 Python 代码段描述一个二叉树的先序、中序与后序遍历算法如下。

(1) 先序遍历。

先序遍历的递归实现首先访问根结点，然后递归地遍历左子树，最后递归地遍历右子树。

```
class TreeNode:
def __init__(self, x):
self.val = x
self.left = None
self.right = None
def preOrderRecursive(root):
if root is None:
return
print(root.val, end=' ')                #访问根结点
preOrderRecursive(root.left)            #遍历左子树
preOrderRecursive(root.right)           #遍历右子树
#示例
#构建二叉树如图 5.8 所示
root = TreeNode('A')
root.left = TreeNode('B')
root.right = TreeNode('C')
root.left.left = TreeNode('D')
root.left.right = TreeNode('E')
#执行先序遍历
preOrderRecursive(root)
```

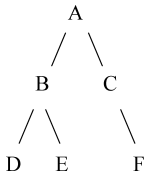

图 5.6　二叉树

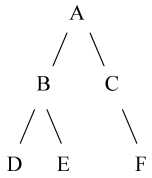

图 5.7　二叉树

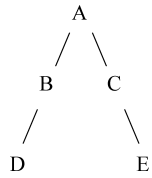

图 5.8　二叉树

(2) 中序遍历(In-order Traversal)。

中序遍历的递归实现首先递归地遍历左子树，然后访问根结点，最后递归地遍历右子树。

```
def inOrderRecursive(root):
```

```
if root is None:
return
inOrderRecursive(root.left)          #遍历左子树
print(root.val, end=' ')             #访问根结点
inOrderRecursive(root.right)         #遍历右子树
#执行中序遍历
inOrderRecursive(root)
```

(3) 后序遍历(Post-order Traversal)。

后序遍历的递归实现首先递归地遍历左子树,然后递归地遍历右子树,最后访问根结点。

```
def postOrderRecursive(root):
if root is None:
return
postOrderRecursive(root.left)        #遍历左子树
postOrderRecursive(root.right)       #遍历右子树
print(root.val, end=' ')             #访问根结点
#执行后序遍历
postOrderRecursive(root)
```

非递归实现(迭代)方法如下。

在这些示例中,我们首先定义了一个 TreeNode 类,然后创建了一个示例二叉树。每个遍历函数首先检查根结点是否为空,如果不为空,就将根结点入栈,并开始遍历过程。在先序遍历中,我们首先打印结点,然后在栈中添加右孩子和左孩子。在中序遍历中,首先遍历到最左边的叶子结点,然后打印当前结点,并转向右子树。在后序遍历中,使用两个栈来分别存储遍历过程中的结点和它们的反向顺序,最后按顺序打印结点。

在 Python 中,非递归(迭代)实现二叉树的遍历可以通过栈来完成。以下是先序、中序、后序遍历的迭代实现。

(1) 先序遍历的迭代实现使用一个栈来辅助遍历,使用 Python 代码段描述如下。

```
def preOrderIterative(root):
if not root:
return
stack = [root]
while stack:
node = stack.pop()
if node:
print(node.data, end=' ')
#右孩子先入栈,保证左孩子后出栈
stack.append(node.right)
stack.append(node.left)
#执行先序遍历
preOrderIterative(root)
```

(2) 中序遍历(迭代实现)。

中序遍历的迭代实现稍微复杂一些,需要使用一个栈来记录访问的顺序。

```
def inOrderIterative(root):
if not root:
```