第5章 树与二叉树

5.1 树

5.1.1 树的定义

树是一种数据结构，它是由 $n(n\geqslant 0)$ 个有限结点组成的一个具有层次关系的集合。空集合也是树，称为空树。把它叫作“树”是因为它看起来像一棵倒挂的树，也就是说它是根朝上，而叶朝下。它具有以下的特点：每个结点有零个或多个子结点；没有父结点的结点称为根结点；每一个非根结点有且只有一个父结点；除了根结点外，每个子结点可以分为多个不相交的子树。

树是由根结点和若干棵子树构成的。树是由一个集合以及在该集合上定义的一种关系构成的。集合中的元素称为树的结点，所定义的关系称为父子关系。父子关系在树的结点之间建立了一个层次结构。在这种层次结构中有一个结点具有特殊的地位，这个结点称为该树的根结点，或称为树根。

5.1.2 树的基本术语

下面根据图 5.1 介绍关于树的基本术语和概念。

孩子结点或子结点：一个结点含有的子树的根结点称为该结点的子结点。

结点的度：一个结点含有的子结点的个数称为该结点的度。

叶结点或终端结点：度为 0 的结点称为叶结点。

非终端结点或分支结点：度不为 0 的结点。

双亲结点或父结点：若一个结点含有子结点，则这个结点称为其子结点的父结点。

兄弟结点：具有相同父结点的结点互称为兄弟结点。

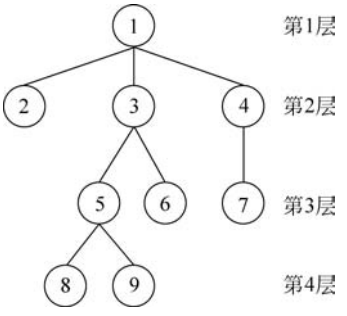


图 5.1 树的树形表示

树的度：一棵树中，最大的结点的度称为树的度。

结点的层次：从根开始定义起，根为第 1 层，根的子结点为第 2 层，依此类推。

树的高度或深度：树中结点的最大层次。

堂兄弟结点：双亲在同一层的结点互为堂兄弟结点。

结点的祖先：从根到该结点所经分支上的所有结点。

子孙：以某结点为根的子树中任一结点都称为该结点的子孙。

有序树：如果树中各棵子树的次序是有先后次序，则称该树为有序树。

无序树：如果树中各棵子树没有先后次序，则称该树为无序树。

森林：各棵互不相交的树的集合称为森林。

路径和路径长度：树中两个结点之间的路径是由这两个结点之间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。

5.1.3 树的种类

无序树：树中任意结点的子结点之间没有顺序关系，这种树称为无序树，也称为自由树。

有序树：树中任意结点的子结点之间有顺序关系，这种树称为有序树。

二叉树：每个结点最多含有两棵子树的树称为二叉树。

满二叉树：叶结点除外的所有结点均含有两棵子树的树称为满二叉树。

完全二叉树：除最后一层外，所有层都是满结点，且最后一层缺右边连续结点的二叉树称为完全二叉树。

哈夫曼树(最优二叉树)：带权路径最短的二叉树称为哈夫曼树或最优二叉树。

5.1.4 树的性质

树具有如下最基本的性质：

(1) 树中的结点数等于所有结点的度数加上 1。

(2) 度为 m 的树中第 i 层上至多有 m^{i-1} 个结点($i \geqslant 1$)。

(3) 高度为 h 的 m 叉树至多有$(m^h - 1)/(m-1)$个结点。

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

5.2 二 叉 树

5.2.1 二叉树的定义及特性

1. 二叉树的定义

二叉树(binary tree)是树形结构的一个重要类型。许多实际问题抽象出来的数据结构往往是二叉树形式，即使是一般的树也能简单地转换为二叉树，而且二叉树的存储结构及其算法都较为简单，因此二叉树显得特别重要。二叉树特点是每个结点最多只能有两棵子树，且有左右之分，如图 5.2 所示。

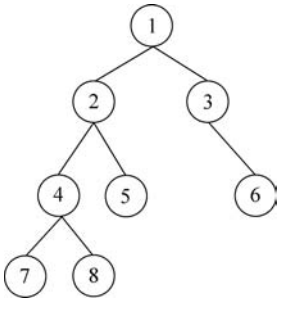


图 5.2 二叉树的形态

二叉树是 n 个有限元素的集合，该集合或者为空、或者由一个称为根(root)的元素及两个不相交的、分别称为左

子树和右子树的二叉树组成,是有序树。当集合为空时,称该二叉树为空二叉树。在二叉树中,一个元素也称作一个结点。

不能把二叉树与度为 2 的有序树等同起来。它们之间有如下几个区别:

(1) 度为 2 的有序树至少有 3 个结点,而二叉树可以为空;

(2) 度为 2 的有序树的左右次序是相对另一个孩子而言的,若某个结点只有一个孩子,则这个孩子就不需要区分左右次序;而二叉树无论其孩子数是否为 2,都需要确认其左右次序,二叉树的结点次序是确定的。

2. 几种特殊二叉树

二叉树有以下几种特殊类型。

(1) 满二叉树:如果一棵二叉树只有度为 0 的结点和度为 2 的结点,并且度为 0 的结点在同一层上,则这棵二叉树为满二叉树。一棵高度为 h 的满二叉树含有 2^h-1 个结点,即树中的每一层都含有最多的结点,如图 5.3(a)所示。

(2) 完全二叉树:高度为 h,有 n 个结点的二叉树,当且仅当其每一个结点都与高度为 h 的满二叉树中编号 1～n 的结点一一对应时,称为完全二叉树,如图 5.3(b)所示。

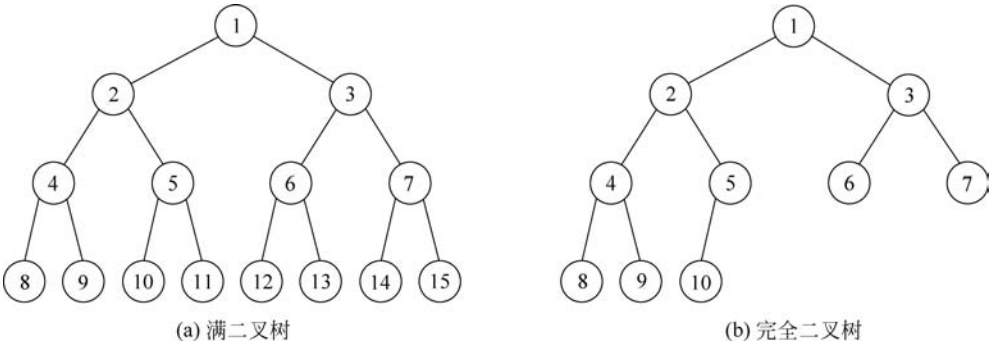


图 5.3　满二叉树和完全二叉树

(3) 二叉排序树:左子树上所有结点的关键字均小于根结点的关键字;右子树上所有结点的关键字均大于根结点的关键字;左子树和右子树又各是一棵二叉排序树,如图 5.4 所示。

(4) 平衡二叉树:根上任一结点的左子树和右子树的深度之差不超过 1,如图 5.5 所示。

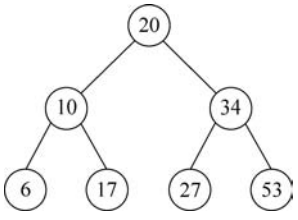


图 5.4　二叉排序树

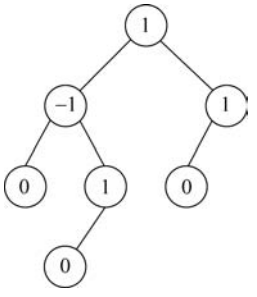


图 5.5　平衡二叉树

3. 二叉树性质

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

性质 2：深度为 h($h \geqslant 1$)的二叉树中至多含有 2^h-1 个结点。

性质 3：若在任意一棵二叉树中，有 n_0 个叶子结点，有 n_2 个度为 2 的结点，则必有 $n_0=n_2+1$。

证明：设度为 0、1 和 2 的结点个数分别为 n_0、n_1 和 n_2，结点总数 $n=n_0+n_1+n_2$。在二叉树的分支数中，除根结点外，其余结点都有一个分支进入，设 A 为分支总数，则 $n=A+1$。由于这些分支都是由度为 1 或 2 的结点射出的，所有 $A=n_1+2n_2$。于是可得 $n_0+n_1+n_2=n_1+2n_2+1$，化简后得 $n_0=n_2+1$。

性质 4：具有 n 个结点的完全二叉树高度为$\lceil \log_2(n+1) \rceil$或$\lfloor \log_2 n \rfloor+1$。

性质 5：若对一棵有 n 个结点的完全二叉树进行顺序编号($0 \leqslant i < n$)，那么，对于编号为 i($i \geqslant 0$)的结点：

当 $i=0$ 时，该结点为根，它无双亲结点；

当 $i>0$ 时，该结点的双亲结点的编号为$\lfloor (i-1)/2 \rfloor$；

若 $2i+1<n$，则有编号为 $2i+1$ 的左结点，否则没有左结点；

若 $2i+2<n$，则有编号为 $2i+2$ 的右结点，否则没有右结点。

5.2.2 二叉树的存储结构

二叉树一般都采用链式存储结构，用链表结点来存储二叉树中的每个结点。在二叉树中，结点结构通常包括若干数据域和若干指针域，二叉链表至少包含 3 个域：数据域 data、左指针域 lchild 和右指针域 rchild，如图 5.6 所示。

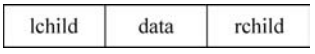


图 5.6 二叉树链式存储的结点结构

二叉树结点类的存储结构描述如下：

```
class BiTreeNode:
    def __init__(self, data = None, lchild = None, rchild = None):
        self.data = data                    # 数据域值
        self.lchild = lchild                # 左孩子指针
        self.rchild = rchild                # 右孩子指针
```

二叉树类的存储结构描述如下：

```
class BiTree:
    def __init__(self, root = None):
        self.root = root                    # 二叉树的根节点
```

5.2.3 二叉树的遍历

遍历是对树的一种最基本的运算，所谓遍历二叉树，就是按一定的规则和顺序走遍二叉树的所有结点，使每一个结点都被访问一次，而且只被访问一次。由于二叉树是非线性结

构,因此,树的遍历实质上是将二叉树的各个结点转换成为一个线性序列来表示。

1. 先序遍历

先序遍历(PreOrder)的操作过程如下。

若二叉树为空,则什么也不做,否则:

(1) 访问根结点;

(2) 先序遍历左子树;

(3) 先序遍历右子树。

先序遍历的递归算法如下:

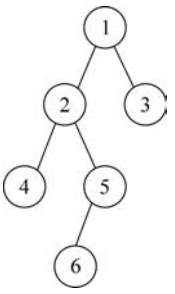


图 5.7　二叉树结构

```
def pre_order(self, root):
    if root is not None:
        print(root.data, end = ' ')          # 输出根结点值
        self.pre_order(root.lchild)          # 递归遍历左子树
        self.pre_order(root.rchild)          # 递归遍历右子树
```

在图 5.7 所表示的二叉树中,先序遍历所得到的结点序列为 1 2 4 5 6 3。

2. 中序遍历

中序遍历(InOrder)的操作过程如下。

若二叉树为空,则什么也不做,否则:

(1) 中序遍历左子树;

(2) 访问根结点;

(3) 中序遍历右子树。

中序遍历的递归算法如下:

```
def in_order(self, root):
    if root is not None:
        self.in_order(root.lchild)           # 递归遍历左子树
        print(root.data, end = ' ')          # 输出根结点值
        self.in_order(root.rchild)           # 递归遍历右子树
```

在图 5.7 中所表示的二叉树中,中序遍历所得到的结点序列为 4 2 6 5 1 3。

3. 后序遍历

后序遍历(PostOrder)的操作过程如下。

若二叉树为空,则什么也不做,否则:

(1) 后序遍历左子树;

(2) 后序遍历右子树;

(3) 访问根结点。

后序遍历的递归算法如下:

```
def post_order(self, root):
    if root is not None:
        self.post_order(root.lchild)         # 递归遍历左子树
        self.post_order(root.rchild)         # 递归遍历右子树
        print(root.data, end = ' ')          # 输出根结点值
```

在图 5.7 所表示的二叉树中,后序遍历所得到的结点序列为 4 6 5 2 3 1。

4. 层次遍历

二叉树的层次遍历，按照层次顺序对二叉树的各个结点进行访问，如图 5.8 所示。

用一个队列保存被访问的当前结点的左右孩子结点以实现层序遍历。在进行层次遍历时，设置一个队列结构，遍历从二叉树的根结点开始，首先将根结点指针入队列，然后从队头取出一个元素，每取一个元素，执行下面两个操作：

(1) 访问该元素所指向的结点；

(2) 若该元素所指结点的左右孩子结点非空，则将该元素所指结点的左孩子指针和右孩子指针顺序入队。此过程不断进行，当队列为空时，二叉树的层次遍历结束。

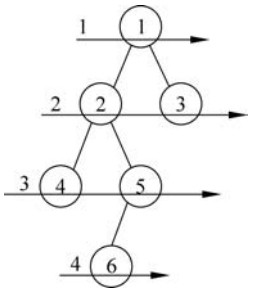


图 5.8　二叉树的层次遍历

二叉树的层次遍历算法如下：

```
def level_order(self, root):
    queue = deque()                          # 建立队列
    queue.append(root)                       # 首先将根结点入队
    while queue:                             # 队列不空则继续遍历
        cur = queue.popleft()                # 队头结点出队
        print(cur.data, end = ' ')           # 输出队头结点数据域值
        if cur.lchild:                       # 左子树不为空则左子树根结点入队
            queue.append(cur.lchild)
        if cur.rchild:                       # 右子树不为空则右子树根结点入队
            queue.append(cur.rchild)
```

5. 由遍历序列构造二叉树

由二叉树的先序序列和中序序列可以唯一地确定一棵二叉树。

由二叉树的后序序列和中序序列可以唯一地确定一棵二叉树。

只知道二叉树的先序序列和后序序列无法唯一确定一棵二叉树。

【实例操作】

以先序遍历的方式构造一个二叉树，输出该二叉树的前序遍历序列、中序遍历序列、后续遍历序列和层次遍历序列。

```
from collections import deque                # Python 数据结构常用模块
class BiTreeNode:
    def __init__(self, data = None, lchild = None, rchild = None):
        self.data = data                     # 数据域值
        self.lchild = lchild                 # 左孩子指针
        self.rchild = rchild                 # 右孩子指针


class BiTree:
    def __init__(self, root = None):
        self.root = root                     # 二叉树的根结点
    # 先序遍历
    def pre_order(self, root):
        if root is not None:
            print(root.data, end = ' ')      # 输出根结点值
```

```
            self.pre_order(root.lchild)      # 递归遍历左子树
            self.pre_order(root.rchild)      # 递归遍历右子树

    # 中序遍历
    def in_order(self, root):
        if root is not None:
            self.in_order(root.lchild)       # 递归遍历左子树
            print(root.data, end = ' ')      # 输出根结点值
            self.in_order(root.rchild)       # 递归遍历右子树

    # 后序遍历
    def post_order(self, root):
        if root is not None:
            self.post_order(root.lchild)     # 递归遍历左子树
            self.post_order(root.rchild)     # 递归遍历右子树
            print(root.data, end = ' ')      # 输出根结点值

    # 层次遍历
    def level_order(self, root):
        queue = deque()                      # 建立队列
        queue.append(root)                   # 首先将根结点入队
        while queue:                         # 队列不空则继续遍历
            cur = queue.popleft()            # 队头结点出队
            print(cur.data, end = ' ')       # 输出队头结点数据域值
            if cur.lchild:                   # 左子树不为空则左子树根结点入队
                queue.append(cur.lchild)
            if cur.rchild:                   # 右子树不为空则右子树根结点入队
                queue.append(cur.rchild)


# 以先序遍历构造二叉树
def create(root = None):
    x = input()
    # 输入'#'表示该结点为 None
    if x == '#':
        root = None
        return
    root = BiTreeNode(x)
    root.lchild = create(root.lchild)
    root.rchild = create(root.rchild)
    return root

tree = BiTree()
tree.root = create()
print("该二叉树先序遍历为:")
tree.pre_order(tree.root)
print()
print("该二叉树中序遍历为:")
```

```
tree.in_order(tree.root)
print()
print("该二叉树后序遍历为:")
tree.post_order(tree.root)
print()
print("该二叉树层次遍历为:")
tree.level_order(tree.root)
```

【输入】(与图 5.7 一样的二叉树结构)

```
1
2
4
#
#
5
6
#
#
#
3
#
#
```

【输出】

```
该二叉树先序遍历为:
1 2 4 5 6 3
该二叉树中序遍历为:
4 2 6 5 1 3
该二叉树后序遍历为:
4 6 5 2 3 1
该二叉树层次遍历为:
1 2 3 4 5 6
```

5.2.4 二叉排序树

1. 二叉排序树的定义

二叉排序树(Binary Sort Tree,BST),又称二叉查找树,亦称二叉搜索树,是数据结构中的一类。在一般情况下,其查询效率比链表结构要高。

二叉排序树或者是一棵空树,或者是具有下列性质的二叉树:

(1) 若左子树不空,则左子树上所有结点的值均小于它的根结点的值;

(2) 若右子树不空,则右子树上所有结点的值均大于它的根结点的值;

(3) 左、右子树也分别为二叉排序树;

(4) 没有键值相等的结点。

根据二叉排序树的定义,左子树结点值<根结点值<右子树结点值,所以对二叉排序树进行中序遍历会得到一个递增的有序序列。如图 5.9 所示,该二叉排序树的中序遍历序列

为 1,3,5,6,7,9。

2. 二叉排序树的查找

二叉排序树的查找从根结点开始,沿某个分支逐层向下比较。若二叉排序树非空,先将给定值与根结点的关键字比较,若相等,则查找成功;若不相等,如果小于根结点的关键字,则在根结点的左子树上查找,否则在根结点的右子树上查找。

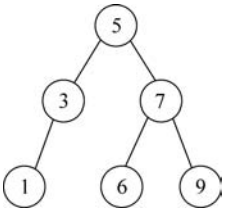


图 5.9 一棵二叉排序树

二叉排序树的查找算法如下:

```
def search(self, x):
    T = self.root
    while T and x != T.data:                    # 若树空或等于根结点值则循环结束
        if x < T.data:                          # 小于则在左子树上查找
            T = T.lchild
        else:                                   # 大于则在右子树上查找
            T = T.rchild
    return T
```

二叉树排序树的查找效率主要取决于树的高度。若二叉排序树左、右子树的高度之差的绝对值不超过 1,则这样的二叉排序树称为平衡二叉树,它的平均查找长度为 $O(\log_2^n)$。若二叉排序树是一个只有右(左)孩子的单支树,则其平均查找长度为 $O(n)$。

3. 二叉排序树的插入

二叉排序树是一种动态树表。其特点是:树的结构通常不是一次生成的,而是在查找过程中,当树中不存在关键字等于给定值的结点时再进行插入。新插入的结点一定是一个新添加的叶子结点,并且是查找不成功时查找路径上访问的最后一个结点的左孩子或右孩子结点。

插入结点的过程如下:首先执行查找算法,找出被插结点的父结点;判断被插结点是其父结点的左、右孩子结点;将被插结点作为叶子结点插入。若二叉树为空,则首先单独生成根结点。

二叉排序树的插入操作算法如下:

```
def insert(self, root, x):
    if root is None:                            # 原树若为空,新插入的记录为根结点
        root = BiTreeNode(x)
        return root
    elif x < root.data:                         # 插入到左子树
        root.lchild = self.insert(root.lchild, x)
    else:                                       # 插入到右子树
        root.rchild = self.insert(root.rchild, x)
    return root
```

4. 二叉排序树的构造

从一棵空树开始,依次输入元素,把它们插入到二叉排序树的合适位置。

构造二叉排序树的算法如下:

```
def create_tree(self, root):
    x = int(input())
```

```
    while x != -1:                          # 输入-1时表示停止输入
        root = self.insert(root, x)         # 将新元素插入到二叉排序树
        x = int(input())
    return root
```

【实例操作】

依次输入数值构造如图5.9所示的二叉排序树,输出该二叉排序树的中序遍历序列。

```
class BiTreeNode:
    def __init__(self, data=None, lchild=None, rchild=None):
        self.data = data                    # 数据域值
        self.lchild = lchild                # 左孩子指针
        self.rchild = rchild                # 右孩子指针


class BiTree:
    def __init__(self, root=None):
        self.root = root                    # 二叉树的根结点

    # 插入操作
    def insert(self, root, x):
        if root is None:                    # 原树若为空,新插入的记录为根结点
            root = BiTreeNode(x)
            return root
        elif x < root.data:                 # 插入到左子树
            root.lchild = self.insert(root.lchild, x)
        else:                               # 插入到右子树
            root.rchild = self.insert(root.rchild, x)
        return root

    # 构造操作
    def create_tree(self, root):
        x = int(input())
        while x != -1:                      # 输入-1时表示停止输入
            root = self.insert(root, x)     # 将新元素插入到二叉排序树
            x = int(input())
        return root

    # 查找操作
    def search(self, x):
        T = self.root
        while T and x != T.data:            # 若树空或等于根结点值则循环结束
            if x < T.data:                  # 小于则在左子树上查找
                T = T.lchild
            else:                           # 大于则在右子树上查找
                T = T.rchild
        return T

    # 中序遍历
    def in_order(self, root):
        if root is not None:
```

```
            self.in_order(root.lchild)        # 递归遍历左子树
            print(root.data, end = ' ')       # 输出根结点值
            self.in_order(root.rchild)        # 递归遍历右子树
```



```
tree = BiTree()
tree.root = tree.create_tree(tree.root)
print("该二叉树中序遍历为:")
tree.in_order(tree.root)
```

【输入】

```
5
3
7
1
6
9
-1
```

【输出】

```
该二叉树中序遍历为:
1 3 5 6 7 9
```

5.2.5 平衡二叉树

平衡二叉树(BalanceBinaryTree,BBT),是由苏联数学家Adelse-Velskil和Landis在1962年提出的高度平衡的二叉树,也称为AVL树。它具有如下几个性质:

(1) 可以是空树。

(2) 假如不是空树,则任何一个结点的左子树与右子树都是平衡二叉树,并且高度之差的绝对值不超过1。

定义结点左子树和右子树的高度差为该结点的平衡因子,则平衡二叉树结点的平衡因子值只可能为-1,0或1,如图5.10所示。

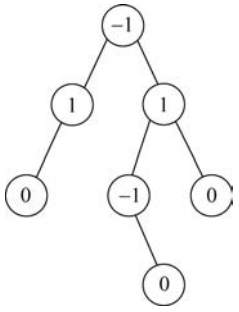


图5.10 一棵平衡二叉树

5.2.6 哈夫曼树

1951年,哈夫曼在麻省理工学院攻读博士学位,他和修读"信息论"课程的同学得选择是完成学期报告还是期末考试。导师Robert Fano出的学期报告题目是:查找最有效的二进制编码。由于无法证明哪个已有编码是最有效的,哈夫曼放弃对已有编码的研究,转向新的探索,最终发现了基于有序频率二叉树编码的想法,并很快证明了这个方法是最有效的。哈夫曼使用自底向上的方法构建二叉树,避免了次优算法香农-法诺编码(Shannon-Fano coding)的最大弊端——自顶向下构建树。

1952 年，哈夫曼于论文《一种构建极小多余编码的方法》(*A Method for the Construction of Minimum-Redundancy Codes*)中发表了这个编码方法。

David Albert Huffman(1925 年 8 月 9 日—1999 年 10 月 7 日)，生于美国俄亥俄州，计算机科学家，为哈夫曼编码的发明者。他也是折纸数学领域的先驱人物。1944 年，在俄亥俄州立大学取得电机工程学士学位。在第二次世界大战期间，进入美国海军服役两年。退伍后，他回到俄亥俄州立大学，取得电机工程硕士学位。其后进入麻省理工学院攻读博士，主修电机工程。1953 年，取得自然科学博士学位。在攻读博士期间，于 1952 年发表了哈夫曼编码。在取得博士学位后，他成为麻省理工学院教师。1967 年，转至圣塔克鲁兹加利福尼亚大学任教，在此，他协助创立了计算机科学系，1970—1973 年，他担任系主任。1994 年，他从学校退休。1999 年，被诊断出癌症，在同年 10 月病逝，享年 74 岁。

1. 哈夫曼树的定义

给定 N 个权值作为 N 个叶结点，构造一棵二叉树，若该树的带权路径长度达到最小，称这样的二叉树为最优二叉树，也称为哈夫曼树(Huffman Tree)。哈夫曼树是带权路径长度最短的树，权值较大的结点离根较近。

哈夫曼树是一种带权路径长度最短的二叉树。所谓树的带权路径长度，就是树中所有的叶结点的权值乘上其到根结点的路径长度(若根结点为 0 层，则叶结点到根结点的路径长度为叶结点的层数)。树的路径长度是从树根到每一结点的路径长度之和，记为 WPL(Weighted Path Length of Tree)$=(W_1 * L_1+W_2 * L_2+W_3 * L_3+\cdots+W_n * L_n)$，$N$ 个权值 $W_i(i=1,2,\cdots,n)$构成一棵有 N 个叶结点的二叉树，相应的叶结点的路径长度为$L_i(i=1,2,\cdots,n)$。例如，在图 5.11 的哈夫曼树中，该树的 WPL$=8*1+5*2+2*3+3*3=33$。同时，该树的 WPL 在所有带这 4 个结点的二叉树中里为最小值。

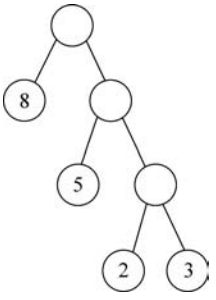


图 5.11　一棵哈夫曼树

2. 哈夫曼树的构造

假设有 n 个权值，则构造出的哈夫曼树有 n 个叶结点。n 个权值分别设为 w_1、w_2、…、w_n，则哈夫曼树的构造规则为：

(1) 将 w_1、w_2、…、w_n 看成有 n 棵树的森林(每棵树仅有一个结点)；

(2) 在森林中选出两个根结点的权值最小的树合并，作为一棵新树的左、右子树，且新树的根结点权值为其左、右子树根结点权值之和；

(3) 从森林中删除选取的两棵树，并将新树加入森林；

(4) 重复步骤(2)、(3)，直到森林中只剩一棵树为止，该树即为所求的哈夫曼树，如图 5.12 所示。

构造哈夫曼树的算法如下：

```
# 树结点结构
class HuffmanNode:
    def __init__(self, data, weight):
        self.data = data
        self.weight = weight
        self.lchild = None
```

```
        self.rchild = None


class HuffmanTree:
    def __init__(self, data):
        nodes = [HuffmanNode(x, w) for x, w in data]        # 建立树结点列表
        self.index = {}                                     # 记录每个结点编码的字典
        while len(nodes) > 1:                               # 循环到列表只剩一个树结点
            nodes = sorted(nodes, key = lambda x: x.weight) # 依据每个树结点的权值进行
                                                              排序
            s = HuffmanNode(None, nodes[0].weight + nodes[1].weight)
            # 取前两个树结点求和获得新的树结点
        # 将两个旧树结点作为新树结点的左右孩子结点
        s.lchild = nodes[0]
        s.rchild = nodes[1]
        nodes = nodes[2:]                                   # 在列表中删除前两个旧树结点
        nodes.append(s)                                     # 添加新树结点加入到列表
    self.root = nodes[0]                                    # 更新 root 为最终得到的哈夫曼树
    self.cal_index(self.root, '')                           # 统计各个结点对应的编码

# 以递归方式计算每个字符的哈夫曼编码
def cal_index(self, root, code):
    if root.data is not None:                               # 当遇到有效树结点,保存对应字符的
                                                              当前编码
        self.index[root.data] = code
    else:
        self.cal_index(root.lchild, code + '0')             # 遍历左子树编码则 + '0'
        self.cal_index(root.rchild, code + '1')             # 遍历右子树编码则 + '1'
```

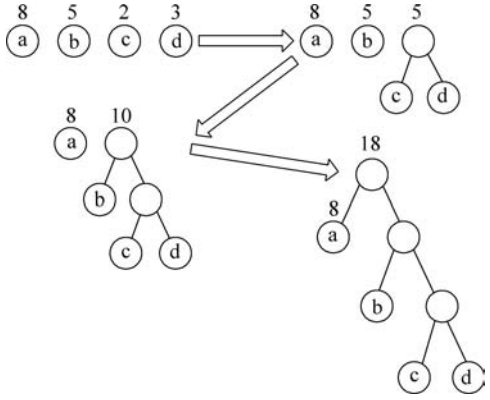


图 5.12　哈夫曼树的构造过程

3. 哈夫曼编码

在数据通信中,需要将传送的文字转换成二进制的字符串,用 0,1 码的不同排列来表示字符。例如,需传送的报文为 AFTERDATAEARAREARTAREA,这里用到的字符集为 A,E,R,T,F,D,各字母出现的次数为{8,4,5,3,1,1}。现要求为这些字母设计编码。要区别 6 个字母,最简单的二进制编码方式是等长编码,固定采用 3 位二进制,可分别用 000、001、010、011、100、101 对“A,E,R,T,F,D”进行编码发送,当对方接收报文时再按照三位一

分进行译码。显然，编码的长度取决于报文中不同字符的个数。若报文中可能出现 26 个不同字符，则固定编码长度为 5。然而，传送报文时总是希望总长度尽可能短。在实际应用中，各个字符的出现频度或使用次数是不相同的，如 A、B、C 的使用频率远远高于 X、Y、Z，自然会想到设计编码时，让使用频率高的用短码，使用频率低的用长码，以优化整个报文编码。

为使不等长编码为前缀编码（即要求一个字符的编码不能是另一个字符编码的前缀），可用字符集中的每个字符作为叶结点生成一棵编码二叉树，为了获得传送报文的最短长度，可将每个字符的出现频率作为字符结点的权值赋予该结点。显然，字使用频率越小权值越小，权值越小叶子就越靠下，于是频率小编码长，频率高编码短，这样就保证了此树的最小带权路径长度效果上就是传送报文的最短长度。因此，求传送报文的最短长度问题转化为求由字符集中的所有字符作为叶结点，由字符出现频率作为其权值所产生的哈夫曼树的问题。利用哈夫曼树设计二进制的前缀编码，既满足前缀编码的条件，又保证报文编码总长最短，如图 5.13 所示。

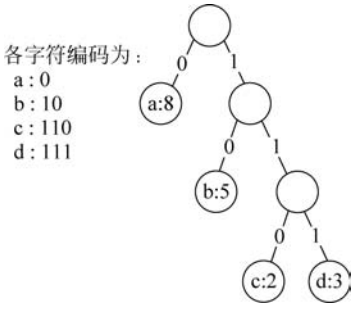


图 5.13　由哈夫曼树构造哈夫曼编码

【实例操作】

```
# 构造如图 5.13 所示的哈夫曼树，并输出每个字符的编码
class HuffmanNode:
    def __init__(self, data, weight):
        self.data = data
        self.weight = weight
        self.lchild = None
        self.rchild = None

class HuffmanTree:
    def __init__(self, data):
        nodes = [HuffmanNode(x, w) for x, w in data]          # 建立树结点列表
        self.index = {}                                       # 记录每个结点编码的字典
        while len(nodes) > 1:                                 # 循环到列表只剩一个树结点
            nodes = sorted(nodes, key = lambda x: x.weight)   # 依据每个树结点的权值进行排序
            s = HuffmanNode(None, nodes[0].weight + nodes[1].weight)  # 取前两个树结点
                                                                      求和获得新的树
                                                                      结点
            # 将两个旧树结点作为新树结点的左右孩子结点
            s.lchild = nodes[0]
            s.rchild = nodes[1]
            nodes = nodes[2:]                                 # 在列表中删除前两个旧树结点
```

```
            nodes.append(s)                                    # 添加新树结点到列表
        self.root = nodes[0]                                   # 更新 root 为最终得到的哈夫曼树
        self.cal_index(self.root, '')                          # 统计各个结点对应的编码

        # 以递归方式计算每个字符的哈夫曼编码
        def cal_index(self, root, code):
            if root.data is not None:          # 当遇到有效树结点,保存对应字符的当前编码
                self.index[root.data] = code
            else:
                self.cal_index(root.lchild, code + '0')    # 遍历左子树编码则 + '0'
                self.cal_index(root.rchild, code + '1')    # 遍历右子树编码则 + '1'

        # 打印各字符哈夫曼编码
        def print_code(self):
            for ch in sorted(self.index):                  # 按字符顺序输出各哈夫曼编码
                print("字符 %s 的哈夫曼编码为: %s" %(ch, self.index[ch]))


data = [['a', 8], ['b', 5], ['c', 2], ['d', 3]]            # 输入数据
huffman = HuffmanTree(data)                                # 构造哈夫曼树
huffman.print_code()                                       # 打印各字符的编码
```

【输出】

```
字符 a 的哈夫曼编码为: 0
字符 b 的哈夫曼编码为: 10
字符 c 的哈夫曼编码为: 110
字符 d 的哈夫曼编码为: 111
```

5.3 树 与 森 林

5.3.1 树的存储结构

树的存储方式有多种,既能采用顺序存储结构,也能采用链式存储结构,但无论采用哪种存储方式,都必须要满足能够唯一地反映树中各结点之间的逻辑关系。下面介绍 3 种常用的存储结构。

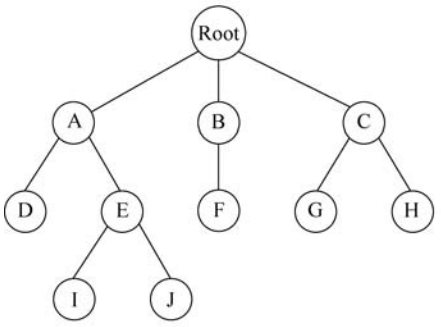
1. 双亲表示法

双亲表示法采用顺序存储结构来实现,采用一组连续空间存储每个结点,同时在每个结点增设一个变量,该变量值为其双亲结点在列表中的索引位置,如图 5.14 所示。

该存储结构利用了每个结点(除根结点外)只有唯一一个双亲的性质,可以很快得到每个结点的双亲结点,但求结点的孩子结点时则需要遍历整个顺序表。

2. 孩子表示法

孩子表示法是将每个结点的孩子结点都用单链表连接起来形成一个线性结构,此时 n 个结点就有 n 个孩子链表(叶结点的孩子链表为空表)。使用孩子表示法来表示图 5.14(a)中的树,如图 5.15 所示。



(a) 一棵树

index	data	parent
0	Root	−1
1	A	0
2	B	0
3	C	0
4	D	1
5	E	1
6	F	2
7	G	3
8	H	3
9	I	5
10	J	5

(b) 双亲表示法

图 5.14　树的双亲表示法

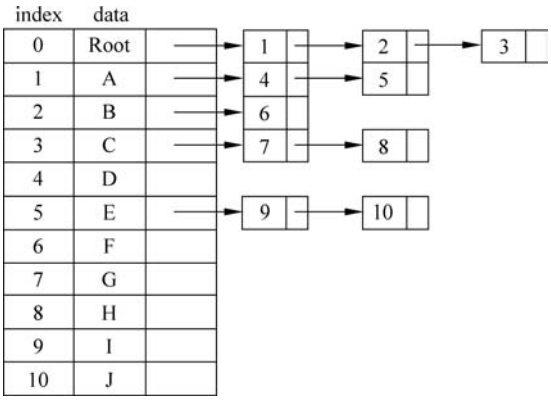


图 5.15　树的孩子表示法

孩子表示法寻找孩子结点的操作非常直接，但寻找双亲结点的操作需要遍历 n 个结点中的孩子链表指针域所指向的 n 个孩子链表。

3. 孩子兄弟表示法

孩子兄弟表示法又称二叉树表示法，即以二叉链表作为树的存储结构。孩子兄弟表示法中，每个结点包括三个部分内容：结点值、指向结点第一个孩子结点的指针和指向结点下一个兄弟结点的指针。用孩子兄弟表示法来表示图 5.14(a)中的树，如图 5.16 所示。

孩子兄弟表示法比较灵活，可以方便地实现树转换为二叉树的操作。通过图 5.16 可以把构建过程总结为“左孩子右兄弟”：把自己的第一个孩子结点作为自己的左孩子，把自己的第一个兄弟结点作为自己的右孩子。如果从某结点一直往右孩子遍历则可以找到该结点的所有兄弟结点。如果要找某结点的所有子结点，则可以先找到该结点的左孩子，再遍历该左孩子结点的右孩子路径，如 A 的左孩子为 D，从 D 往右孩子遍历得到 D 和 E，故 D 和 E 都是 A 的孩子结点。

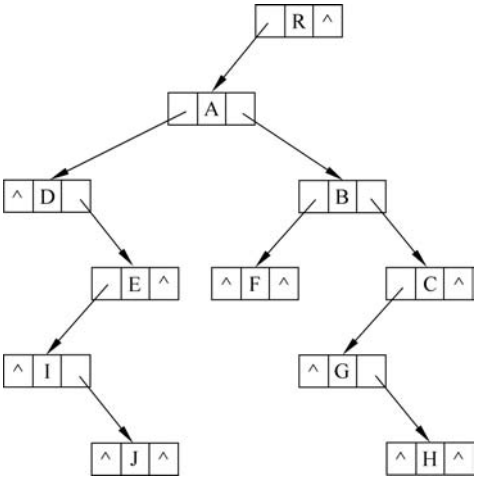


图 5.16　树的孩子兄弟表示法

5.3.2　森林与二叉树的转换

森林(forest)是 $n(n\geqslant 0)$棵互不相交的树的集合。任何一棵树,删除了根结点就变成了森林。

将森林转换为二叉树的规则与树类似,都使用了“左孩子右兄弟”的规律。首先将森林中的每棵树转换为二叉树,由于任何一棵和树对应的二叉树的右子树必定为空,可以把森林中第二棵树根视为第一棵树根的右兄弟,即将第二棵树对应的二叉树当作第一棵二叉树根的右子树,将第三棵树对应的二叉树当作第二棵二叉树根的右子树,依此类推,最终可以将森林转换为二叉树,如图 5.17 所示。

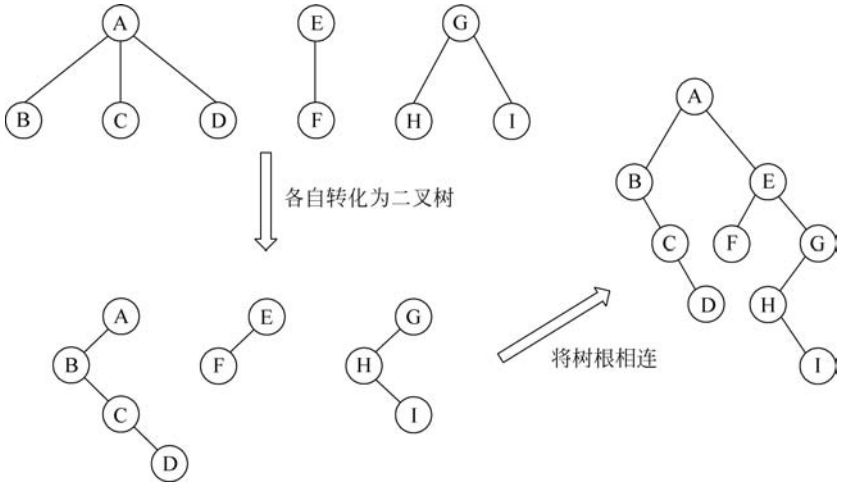


图 5.17　森林转换为二叉树的过程

5.4 二叉树相关算法设计与分析

【例 5.1】 二叉树的最大深度

给定一棵二叉树，求出其最大深度。二叉树的深度为根结点到最远叶结点的最长路径上的结点数。

【分析】

如果已知二叉树左子树和右子树的最大深度分别为 l 和 r，那么就可以得出该二叉树的最大深度为 $\max(l,r)+1$。而左子树和右子树的最大深度又可以相同的方式进行计算。通过自底向上的计算，就能最终得到二叉树的最大深度。根据这个规律，可以选择递归的方法来求得二叉树最大深度。

【算法】

通过递归获得左子树和右子树的最大深度，返回两者的较大值并＋1。递归出口结点为空。

【代码】

```
class Solution:
    def maxDepth(self, root):
        if root is None:                                    # 递归出口
            return 0
        else:
            left_height = self.maxDepth(root.left)          # 获取左子树最大深度
            right_height = self.maxDepth(root.right)        # 获取右子树最大深度
            return max(left_height, right_height) + 1       # 返回最大深度
```

【例 5.2】 二叉树的路径总和

给定二叉树的根结点 root 和一个表示目标和的整数 target，判断二叉树中是否存在从根结点到叶结点的路径，该路径上所有结点值相加和为 target。

【分析】

假定根结点到当前结点路径上的值之和为 val，可以将该问题转化为小问题：求是否存在从当前结点的子结点到叶结点的路径，满足其路径之和为 target－val。故该问题也满足递归的性质，随着向下递归而更新 target 值。若当前结点是叶结点，那么可以直接判断该叶结点值 val 是否等于 target，如果等于则表示找到，否则返回继续遍历二叉树。如果当前结点不是叶结点，则递归地访问其子结点判断其是否满足条件。

【代码】

```
class Solution(object):
    def hasPathSum(self, root, target):
        if not root:
            return False
        # 假如叶结点等于 target 则返回 True
        if not root.left and not root.right and target == root.val:
            return True
        # 只要找到一个结果为 True 的路径，整体就为 True
```

```
        # 往下递归并更新 target 值
        return self.hasPathSum(root.left, target - root.val) or self.hasPathSum(root.right,
target - root.val)
```


【例 5.3】 翻转一棵二叉树

给定一棵二叉树,请翻转该二叉树,如图 5.18 所示。

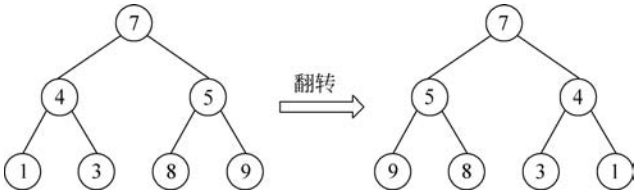


图 5.18 二叉树的翻转过程

【分析】

解决二叉树问题基本都可以用递归的方法来求解。为了翻转二叉树,可以从根结点开始,递归地对二叉树进行遍历,同时交换左右子树的位置,最终则可以得到翻转后的二叉树。

【算法】

(1) 递归出口为空结点;

(2) 递归地得到翻转后的左子树和翻转后的右子树;

(3) 将左右子树位置交换;

(4) 得出翻转后的二叉树。

【代码】

```
class Solution(object):
    def invertTree(self, root):
        if not root:                                    # 递归出口
            return root
        left = self.invertTree(root.left)               # 获取翻转后的左子树
        right = self.invertTree(root.right)             # 获取翻转后的右子树
        root.left, root.right = right, left             # 交换左右子树
        return root
```

【复杂度分析】

时间复杂度:$O(n)$,其中 n 为二叉树的结点数,每个结点都在递归中遍历一次。

空格复杂度:$O(\text{height})$,其中 height 为二叉树的高度。递归函数需要栈空间,而栈空间取决于递归的深度,故空间复杂度等价于二叉树的高度。

小　　结

(1) 树是一种数据结构,它是由 $n(n\geqslant 0)$个有限结点组成的一个具有层次关系的集合,属于非线性结构。树与线性结构不同,树中的数据元素具有一对多的逻辑关系。

(2) 二叉树是一种特殊的有序树,它也是由 $n(n\geqslant 0)$个有限结点组成的集合。当 $n=0$ 时称为空二叉树。二叉树的每个结点最多只有两棵子树,其子树也为二叉树,互不相交且有

左右之分。

(3) 二叉树具有先序遍历、中序遍历、后序遍历和层次遍历 4 种遍历方式。

(4) 哈夫曼树是指给定 n 个带有权值的结点作为叶结点构造出的具有最小带权路径长度的二叉树,也叫哈夫曼树。

(5) 哈夫曼编码是数据压缩技术中的无损压缩技术,是一种不等长的编码方案,使所有数据的编码总长度变短。