第 5 章

树

【知识目标】

- 掌握树的定义、基本术语和逻辑特征。
- 理解二叉树的定义、逻辑结构和性质。
- 熟悉二叉树的顺序存储结构和链式存储结构。
- 掌握二叉树的前序遍历、中序遍历、后序遍历和层次遍历方法。
- 了解线索二叉树的概念、建立和遍历。
- 掌握二叉链表的构造和基本操作。
- 熟悉树的存储结构、森林与二叉树的转换以及树和森林的遍历。
- 理解哈夫曼树的定义，掌握哈夫曼树的构造、哈夫曼算法的实现和哈夫曼编码。

【能力目标】

- 能够根据实际问题构建合适的树和二叉树结构。
- 熟练运用不同的遍历方法对二叉树进行操作。
- 具备建立线索二叉树和进行相关遍历的能力。
- 能够进行二叉链表的构造和基本操作。
- 能够进行森林与二叉树的转换以及树和森林的遍历。
- 能够运用哈夫曼树解决实际问题，如数据压缩等。。

【素质目标】

- 培养逻辑思维能力，通过对树和二叉树等复杂数据结构的理解和操作，提升学生的分析问题和解决问题的能力。
- 增强耐心和细心，在构建和操作树、二叉树等数据结构时，需要仔细处理每一个节点和关系，培养严谨的态度。
- 鼓励创新精神，在应用树和二叉树等数据结构解决实际问题时，引导尝试不

同的方法和思路，培养创新意识。

➢ 提升团队合作能力，在项目实践中，需要共同合作构建和操作复杂的数据结构，培养沟通和协作能力。

➢ 培养自主学习能力，树和二叉树等数据结构的知识较为复杂，需要在课后进行自主学习和探索，提升学习能力和自我提升的意识。

5.1 树的基本概念

5.1.1 树的定义

树是一种非线性数据结构，它是由 n 个结点组成的有限集合。当 n=0 时，称为空树；当 n>0 时，有且仅有一个特定的称为根的结点，它只有直接后继，但没有直接前驱。其余结点可分为 m 个互不相交的有限集，其中每一个集合本身又是一棵树，并且称为根的子树。

在家族树中，最顶层的祖先即为根结点，其子女构成根的子树，每个子女又可有自己的子女，形成下一层的子树，如此递归形成整个家族树结构。这种递归特性是树结构的重要特征。

从图论的角度来看，树是一个无回路的连通图，也就是说，任意两个结点之间有且仅有一条路径相连。

用结构体来表示树的结点，代码如下。

```
// 定义树的结点结构
typedef struct TreeNode {
    int data;                          // 结点数据
    struct TreeNode *children[10]; // 指向子结点的指针数组，这里假设最多有 10 个子结点
    int childCount;                    // 子结点的数量
} TreeNode;

// 创建一个新的树结点
TreeNode* createNode(int value) {
    TreeNode* newNode = (TreeNode*)malloc(sizeof(TreeNode));
    newNode->data = value;
    newNode->childCount = 0;
    for (int i = 0; i < 10; i++) {
        newNode->children[i] = NULL;
    }
    return newNode;
}
```

5.1.2 树的基本术语

1. 结点相关术语

结点：树中的每个元素称为一个结点，它包含数据项以及指向其子树的分支。例如，在图 5–1 所示的树中，A、B、C、D 等都是结点。

度：一个结点所拥有的子树的个数称作该结点的度。比如，结点 A 的度是 3，原因是它有三棵子树，分别以 B、C、D 为根；而结点 E 的度为 1，因其仅有一棵子树。

叶子结点：度为 0 的结点被叫做叶子结点，也称作终端结点。图 5–1 中的 K、L、F、G、M、I、J 都是叶子结点，它们没有子树。

分支结点：度不为 0 的结点称为分支结点，也叫非终端结点。除根结点外的分支结点也可称为内部结点。图 5–1 中的 A、B、C、D、E、H 都是分支结点。

树的度：树中所有结点的度的最大值即为树的度。在图 5–1 的树中，树的度为 3，因为结点 A 的度在所有结点中最大，为 3。

2. 关系相关术语

孩子：结点子树的根被称为该结点的孩子。例如，在图 5–1 中，B、C、D 是 A 的孩子。

双亲：一个结点是它孩子的双亲。所以，A 是 B、C、D 的双亲。

兄弟：具有相同双亲的结点互为兄弟。B、C、D 互为兄弟，因为它们的双亲都是 A。

祖先：从根结点到该结点所经分支上的所有结点都是这个结点的祖先。比如，对于结点 M，它的祖先为 A、D、H，因为从根结点 A 到 M 的路径为 A – D – H – M。

子孙：以某结点为根的子树中的所有结点都称为该结点的子孙。例如，以 D 为根的子树中，H、I、J、M 都是 D 的子孙。

3. 层次与高度术语

结点的层次：从根结点开始定义，根结点为第 1 层，根的子结点为第 2 层，依此类推。例如，在图 5–1 中，A 在第 1 层，B、C、D 在第 2 层，E、F、G、H、I、J 在第 3 层，K、L、M 在第 4 层。

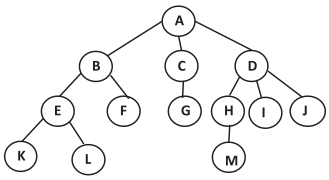


图 5–1　树的图示表示

树的高度（深度）：树中结点的最大层次数就是树的高度或深度。图 5–1 中树的高度为 4。

5.1.3 树的逻辑特征

树作为一种非线性数据结构，具有独特的逻辑特征，这些特征使其在表示层次化数据和解决各种实际问题中发挥着重要作用。

根的唯一性：在任何一棵非空树中，有且仅有一个特定的结点被称为根。这个根结点是整棵树的起始点，它没有直接前驱，所有其他结点都可以通过从根开始的路径访问到。例如，在一个公司的组织架构树中，公司的最高领导者就是根结点，所有其他员工都处于以这个根为起点的不同层级和分支上。这一特性确保了树结构的层次性和方向性，使得数据的组织和管理具有清晰的层次脉络。

子树的独立性：除根结点外，其余每个结点都可以看作是一个新的子树的根，这些子树之间是互不相交的有限集。这意味着每个子树都可以独立地进行操作和处理，而不会影响到其他子树。例如，在一个文件系统树中，每个文件夹都可以看作是一个子树的根，文件夹下的文件和子文件夹构成了该子树的内容。对某个文件夹及其内容的操作（如移动、删除、重命名等）不会影响到其他文件夹及其子树。这种独立性使得树结构非常适合表示具有层次化和模块化的数据，便于数据的管理和维护。

递归性：树的定义本身就是递归的，即树是由根结点和若干棵子树构成，而每棵子树又可以由更小的子树构成。这种递归特性使得树结构在处理复杂问题时具有很强的表达能力。例如，在计算树的高度、遍历树的结点等操作中，都可以通过递归的方式简洁而有效地实现。以计算树的高度为例，树的高度等于其最高子树的高度加 1，而子树的高度又可以通过同样的递归方式计算。这种递归定义使得树结构的实现和操作变得相对简单和直观。

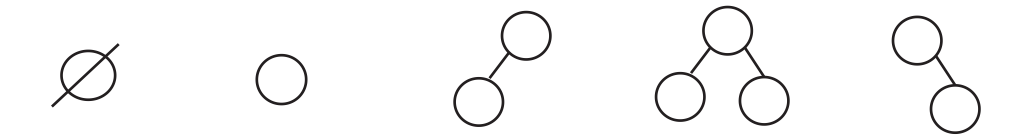
5.2 二叉树

5.2.1 二叉树的定义

二叉树是一种特殊的树型结构，它的每个结点最多有两个子树，分别称为左子树和右子树，且子树的顺序不能颠倒。二叉树可以是空集，也可以由一个根结点和两棵互不相交的左、右子树组成，而这两棵子树本身又都是二叉树，这是一个递归的定义。

从形式化的角度来看，二叉树是 n 个结点的有限集合，当 $n = 0$ 时，为空二叉树；当 $n > 0$ 时，有且仅有一个根结点，其余结点分为两棵互不相交的二叉树 T_1 和 T_2，分别称为根的左子树和右子树。

二叉树具有五种基本形态，分别为：



（a）空二叉树　（b）只有根结点的二叉树　（b）只有根结点的二叉树　（d）有左右子树的二叉树　（e）只有右子树的二叉树

图 5-2　二叉树的 5 种基本形态

空二叉树：没有任何结点，是二叉树的一种特殊情况，表示集合为空。

只有根结点的二叉树：整棵树仅包含一个根结点，没有左子树和右子树。

只有左子树的二叉树：根结点存在且有非空的左子树，但右子树为空。

只有右子树的二叉树：根结点存在且有非空的右子树，但左子树为空。

根结点既有左子树又有右子树的二叉树：这是最常见的形态，根结点同时拥有左子树和右子树，且两棵子树都可以是任意形态的二叉树。

用结构体来表示二叉树的结点，如下所示。

```
// 定义二叉树的结点结构
typedef struct BiTNode {
    int data;                    // 结点数据
    struct BiTNode *lchild;      // 指向左子结点的指针
    struct BiTNode *rchild;      // 指向右子结点的指针
} BiTNode, *BiTree;

// 创建一个新的二叉树结点
BiTNode* createBiT
Node(int value) {
    BiTNode* newNode = (BiTNode*)malloc(sizeof(BiTNode));
    newNode->data = value;
    newNode->lchild = NULL;
    newNode->rchild = NULL;
    return newNode;
}
```

5.2.2 二叉树的性质

1. 层结点数性质

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

证明：采用数学归纳法进行证明。

基础步骤：当 i=1 时，二叉树仅包含一个根结点。此时，$2^{1-1}=1$，因此第 1 层最多有 1 个结点，命题成立。

归纳假设：假设对于所有 $i \leq k$，命题均成立，即第 i 层上结点数至多为 2^{i-1} 个。

归纳步骤：鉴于二叉树的每个结点最多拥有两个子结点，第 k+1 层的结点数最多是第 k 层结点数的 2 倍。依据归纳假设，第 k 层最多有 2^{k-1} 个结点，因此第 k+1 层最多有 2^k 个结点。

因此，二叉树的第 i 层上结点数至多为 2^{i-1} 个。例如，在一棵二叉树中，第 2 层最多有 $2^{2-1}=2$ 个结点，第 3 层最多有 $2^{3-1}=4$ 个结点。

可通过以下代码验证这一性质：

```
// 计算二叉树第 i 层最多的结点数
int maxNodesAtLevel(int i) {
    return 1 << (i – 1);
}
```

2. 深度与结点数性质

性质 2：深度为 k 的二叉树，结点数至多为 2^k-1 个（$k \geqslant 1$）。

证明：根据性质 1，二叉树第 i 层最多有 2^{i-1} 个结点，第 i+1 层最多有 2^i 个结点，依此类推，第 k 层最多有 2^{k-1} 个结点。深度为 k 的二叉树的最大结点数即为各层最大结点数之和，即：

$$2^0+2^1+2^2+\cdots+2^{k-1}=2^k-1$$

这是一个首项为 1、公比为 2、项数为 k 的等比数列求和。依据等比数列求和公式，可得：

$$S_k=\frac{2^k-1}{2-1}=2^k-1$$

例如，深度为 4 的二叉树，最多有 $2^4-1=15$ 个结点。

可通过以下代码验证这一性质：

```
// 计算深度为 k 的二叉树最多的结点数
int maxNodesInTree(int k) {
    return (1 << k) – 1;
}
```

3. 度与叶子结点性质

性质 3：对于任意一棵二叉树，若其叶子结点数为 n_0，度为 2 的结点数为 n_2，则 $n_0=n_2+1$。

证明：设二叉树中度为 1 的结点数为 n_1，总结点数为 n。由于二叉树中所有结点的度均不大于 2，因此有：

$$n=n_0+n_1+n_2$$

再考虑二叉树中的分支数，除根结点外，其余结点均有一个进入分支，设分支总数为 B，则有：

$$B=n-1$$

因为分支总数由度为 1 的结点和度为 2 的结点射出的分支组成，度为 1 的每个结点射出 1 个分支，度为 2 的每个结点射出 2 个分支，则有：

$$B=n_1+2n_2$$

将 B=n−1 代入，可得：

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

结合 n=n0+n1+n2，两式相减消去 n1 和 n，得到：

$$n_0=n_2+1$$

一棵二叉树中有 5 个度为 2 的结点，依据此性质可知叶子结点数为 6 个。

可通过以下代码验证这一性质。

```
// 计算叶子结点数
int countLeafNodes(BiTree root) {
    if (root == NULL) {
        return 0;
    }
    if (root->lchild == NULL && root->rchild == NULL) {
        return 1;
    }
    return countLeafNodes(root->lchild) + countLeafNodes(root->rchild);
}
// 计算度为 2 的结点数
int countDegree2Nodes(BiTree root) {
    if (root == NULL) {
        return 0;
    }
    if (root->lchild != NULL && root->rchild != NULL) {
        return 1+countDegree2Nodes(root->lchild) + countDegree2Nodes(root->rchild);
    }
    return countDegree2Nodes(root->lchild) + countDegree2Nodes(root->rchild);
}
// 验证性质 3
void verifyProperty3(BiTree root) {
    int n0 = countLeafNodes(root);
    int n2 = countDegree2Nodes(root);
```

```
        if (n0 == n2 + 1) {
            printf(“ 性质 3 验证成功：叶子结点数为 %d，度为 2 的结点数为 %d，n0 = n2 +
1 成立 \n”, n0, n2);
        } else {
            printf(“ 性质 3 验证失败：叶子结点数为 %d，度为 2 的结点数为 %d，n0 != n2
+ 1\n”, n0, n2);
        }
}
```

4. 完全二叉树性质

满二叉树：最后一层都是叶子结点，其他各层结点都有左、右子树，即在满二叉树中，每一层的结点都具有最大的结点个数。每个结点的度或者为 2，或者为 0（叶子结点），不存在度为 1 的结点。图 5–3 是一棵满二叉树。

从满二叉树的根结点开始，从上到下，从左到右，依次对每个结点进行连续编号，如图 5–4 所示。

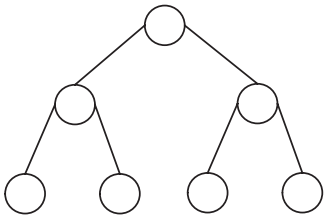


图 5–3　满二叉树

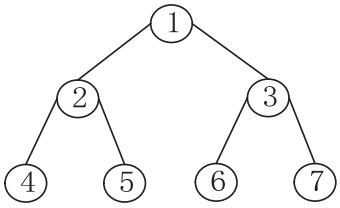


图 5–4　满二叉树及编号

完全二叉树：如果一棵二叉树有 n 个结点，并且二叉树的 n 个结点的结构与满二叉树的前 n 个结点的结构完全相同，则称这样的二叉树为完全二叉树。完全二叉树及对应编号如图 5–5 所示，而图 5–6 所示就不是一棵完全二叉树。

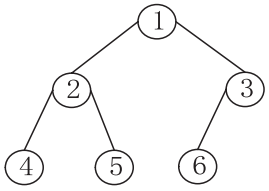


图 5–5　完全二叉树及编号

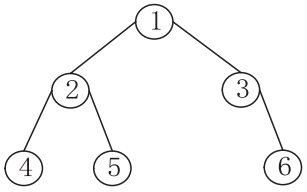


图 5–6　非完全二叉树

如果二叉树的层数为 k，则满二叉树的叶子结点一定是在第 k 层，而完全二叉树的叶子结点一定在第 k 层或者第 k–1 层。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。

性质 4：具有 n 个结点的完全二叉树，其深度为 $\lfloor \log_2 n \rfloor + 1$（其中 $\lfloor \cdot \rfloor$ 表示向下取整）。

证明：设完全二叉树的深度为 k，依据性质 2，深度为 k 的二叉树最多有 2^k-1 个结点，深度为 k–1 的二叉树最多有 $2^{k-1}-1$ 个结点。由于完全二叉树的前 k–1 层是满的，第 k 层可能不满，因此有：

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

对不等式两边同时加 1，得：

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

对不等式两边取以 2 为底的对数，得：

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

由于 k 是整数，故有：

$$k=\lfloor \log_2 n \rfloor+1$$

性质 5：若对一棵有 n 个结点的完全二叉树的结点按层序编号（从第 1 层的根结点开始，从左到右，从上到下），则对于任一结点 i（$1 \leqslant i \leqslant n$），有：

若 i=1，则结点 i 为二叉树的根，无双亲；若 i>1，则其双亲结点编号为 $\lfloor i/2 \rfloor$。

若 $2i \leqslant n$，则结点 i 的左孩子结点编号为 2i；否则结点 i 无左孩子（为叶子结点）。

若 $2i+1 \leqslant n$，则结点 i 的右孩子结点编号为 2i+1；否则结点 i 无右孩子。

例如，对于一棵有 10 个结点的完全二叉树，其深度为 $\lfloor \log_2 10 \rfloor+1=4$。编号为 5 的结点，其双亲编号为 $\lfloor 5/2 \rfloor=2$，左孩子编号为 $2 \times 5=10$（10 小于等于 10 成立，故有左孩子），右孩子编号为 $2 \times 5+1=11$（但 11 超过 10，故无右孩子）。

可通过以下代码验证这一性质。

```
// 完全二叉树的结点结构体
typedef struct {
    int data;
} CompleteBiTreeNode;
// 验证完全二叉树的性质 4 和性质 5
void verifyCompleteTreeProperties(CompleteBiTreeNode tree[], int n) {
    // 验证性质 4
    int depth = (int)(log2(n)) + 1;
    if (depth == (int)(log2(n)) + 1) {
        printf(" 性质 4 验证成功：完全二叉树深度为 %d，符合公式 \n", depth);
    } else {
        printf(" 性质 4 验证失败：完全二叉树深度计算错误 \n");
    }
```

```
        // 验证性质 5
        for (int i = 1; i <= n; i++) {
            // 验证双亲
            if (i == 1) {
                if (i != 1) {
                    printf(“ 性质 5 验证失败：根结点编号错误 \n”);
                }
            } else {
                int parent = (int)(i / 2);
                if (parent != (int)(i / 2)) {
                    printf(“ 性质 5 验证失败：结点 %d 的双亲编号错误 \n”, i);
                }
            }

            // 验证左孩子
            if (2 * i <= n) {
                int leftChild = 2 * i;
                if (leftChild != 2 * i) {
                    printf(“ 性质 5 验证失败：结点 %d 的左孩子编号错误 \n”, i);
                }
            }

            // 验证右孩子
            if (2 * i + 1 <= n) {
                int rightChild = 2 * i + 1;
                if (rightChild != 2 * i + 1) {
                    printf(“ 性质 5 验证失败：结点 %d 的右孩子编号错误 \n”, i);
                }
            }
        }
        printf(“ 性质 5 验证成功：所有结点编号符合性质 \n”);
    }
```

5.2.3 二叉树的存储结构

1. 顺序存储结构

二叉树的顺序存储结构是利用一组地址连续的存储单元，按照自上而下、自左至右的顺序依次存储完全二叉树上的结点元素。这种存储方式特别适合完全二叉树，能够通过数组下标直观地体现结点之间的逻辑关系。

对于一个包含 n 个结点的完全二叉树，按照层序编号后，编号为 i 的结点（1≤i≤n）具有以下关系：

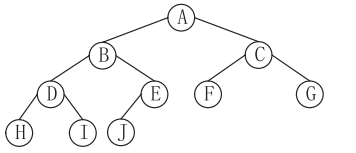
其双亲结点编号为 ⌊i/2⌋；

左孩子结点编号为 2i（若 2i≤n）；

右孩子结点编号为 2i+1（若 2i+1≤n）。

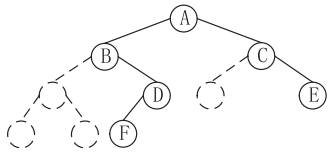
完全二叉树中每个结点的编号可以通过性质 5 计算得到，因此，完全二叉树的存储以按照从上到下、从左到右的顺序依次存储在一维数组中。完全二叉树的顺序存储如图 5 –7 所示。

如果按照完全二叉树的规则对非完全二叉树进行同样的编号，并存放在一维数组中，为了能够正确反映二叉树中结点之间的逻辑关系，需要在一维数组中将二叉树中不存在的结点位置空出，并用‘^’填充。非完全二叉树的顺序存储结构如图 5–8 所示。



位置i	0	1	2	3	4	5	6	7	8	9	10
结点		A	B	C	D	E	F	G	H	I	J

图 5–7　完全二叉树及其顺序存储



位置i	0	1	2	3	4	5	6	7	8	9	10
结点		A	B	C	∧	D	∧	E	∧	∧	F

图 5–8　一般二叉树及其顺序存储

对于一般的二叉树，直接采用顺序存储结构可能会导致大量空间浪费。例如，一棵仅包含右子树的二叉树，在存储时会出现许多空的存储单元。为了避免这种情况，通常需要将一般二叉树补全为完全二叉树，并在没有结点的位置存储空值。

使用一维数组来实现这种存储结构，以下是实现代码。

```
#define MAX_TREE_SIZE 100 // 定义数组的最大容量
typedef int ElemType;

ElemType completeBiTree[MAX_TREE_SIZE];
// 初始化完全二叉树数组
void initCompleteBiTree() {
    for (int i = 0; i < MAX_TREE_SIZE; i++) {
        completeBiTree[i] = -1; // 使用 -1 表示空结点
    }
}
// 将结点插入完全二叉树数组
void insertNode(int value, int index) {
    if (index < 1 || index >= MAX_TREE_SIZE) {
        printf(" 插入位置不合法 \n");
        return;
    }
    completeBiTree[index] = value;
}
```

2. 链式存储结构

二叉树的链式存储结构是指用链表来表示一棵二叉树，链表中每个结点由三

个域组成：数据域（data）、左指针域（lchild）和右指针域（rchild），分别用于存储结点的数据、指向左子结点的指针和指向右子结点的指针，这种链表也称作二叉链表。

其中，数据域存放结点的值，左孩子指针域指向左孩子结点，右孩子指针域指向右孩子结点。这种链式存储结构称为二叉链表存储结构，如图 5–9 所示。

采用二叉链表存储结构表示，其二叉树的二叉链表存储表示如图 5–10 所示。

图 5–9　二叉链表的结点结构

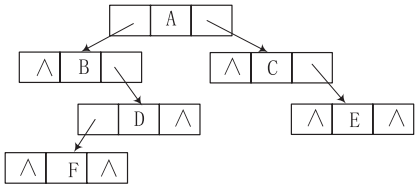


图 5–10　二叉树的二叉链表存储表示

结构体定义如下。

```
// 定义二叉链表的结点结构
typedef struct BiTNode {
int data;                          // 结点数据
    struct BiTNode *lchild;        // 指向左子结点的指针
    struct BiTNode *rchild;        // 指向右子结点的指针
} BiTNode, *BiTree;

// 创建一个新的二叉链表结点
BiTNode* createBiTNode(int value) {
    BiTNode* newNode = (BiTNode*)malloc(sizeof(BiTNode));
    newNode–>data = value;
    newNode–>lchild = NULL;
    newNode–>rchild = NULL;
    return newNode;
}
```

这种链式存储结构能够灵活地表示各种二叉树，适用于频繁的插入和删除操作，不会像顺序存储结构那样在一般二叉树的情况下造成空间浪费。

从二叉链表存储可知，要想找一个结点的孩子结点，只需顺着它的左右孩子结点指针就可以直接找到，但要找该结点的双亲就没那么方便了。为了方便找到结点的双亲结点及孩子结点，只需在二叉链表的基础上增加一个指向双亲结点的指针域 parent，这种存储结构称为三叉链表存储结构，如图 5–11 所示。

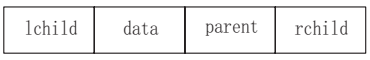


图 5–11　三叉链表的结点结构

三叉链表存储结构的类型定义描述如下。

```
typedef struct TriNode{ / * 三叉链表存储结构类型定义 * /
        DataType data; / * 数据域 * /
        struct TriNode   *lchild; / * 指向左孩子结点 * /
struct TriNode   *parent; / * 指向双亲结点 * /
        struct TriNode   *rchild; / * 指向右孩子结点 * /
}* TriTree,TriNode ;
```

采用三叉链表存储结构，其二叉树的存储表示如图 5–12 所示。

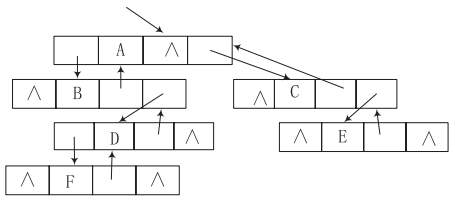


图 5–12　二叉树的三叉链表存储

线索链表将在后面章节说明。

5.3 二叉树的遍历和线索二叉树

在二叉树的应用中，常常需要对二叉树中的每个结点进行访问，即二叉树的遍历，它是二叉树中最基本的运算。

二叉树的遍历，即按照某种规律对二叉树中的每个结点进行访问，且每个结点仅被访问一次的操作。这里的访问，包括对结点的输出、统计结点的个数等。遍历是任何数据结构类型均有的操作，对线性结构而言，因为每个结点（最后一个结点除外）均只有一个后继，因而只有一条搜索路径，不需要单独讨论。而二叉树是非线性结构，结点有两个后继，则存在如何遍历，即按什么样的搜索路径遍历二叉树的结点。二叉树的遍历过程其实就是将二叉树的非线性序列转换成一个线性序列的过程。遍历是二叉树上最重要的运算之一，是二叉树上进行其它运算之基础。

5.3.1 二叉树的遍历

最早提出遍历问题是对存储在计算机中的表达式求值。例如，原表达式（a–b%c）+d*e。该表达式用二叉树表示如图 5–13 所示。当对此二叉树进行先序、中序、后序遍历时，便可获得表达式的前缀、中缀、后缀式。

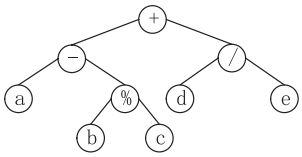


图 5-13　原表达式的二叉树表示

- 前缀：+−a%bc/de
- 中缀 :a−b%c+d*e
- 后缀：abc%−de/+

其中，中缀式与原表达式很相似，只是没有括号。前缀表达式称为波兰式，后缀表达式称为逆波兰式。由于后缀式中运算符出现的次序恰好就是原表达式的运算顺序，所以，一般采用后缀表达式来求值。

下面以二叉树的二叉链表存储结构来具体讨论二叉树的遍历问题。

1. 前序遍历

前序遍历是二叉树遍历的一种重要方式，其遍历规则是：先访问根结点，然后前序遍历左子树，最后前序遍历右子树。对于每一棵子树，同样遵循这个规则，即先访问子树的根结点，再依次访问其左子树和右子树，直到所有结点都被访问完为止。这种遍历方式的特点在于，根结点总是在其左子树和右子树之前被访问，因此得名“前序”。

二叉树的前序遍历，可通过递归方式实现。以下是基于之前定义的二叉树结点结构（BiTNode）的前序遍历代码。

```
// 前序遍历函数
void preOrderTraversal(BiTree root) {
    if (root != NULL) {
        // 访问根结点
        printf("%d ", root->data);
        // 递归前序遍历左子树
        preOrderTraversal(root->lchild);
        // 递归前序遍历右子树
        preOrderTraversal(root->rchild);
    }
}
```

如图 5-14 所示的二叉树，其前序遍历的过程如下。

（1）首先访问根结点 A，输出 A。

（2）接着对 A 的左子树进行前序遍历。左子树的根结点是 B，输出 B。

（3）对 B 的左子树进行前序遍历。B 的左子树的根结点是 D，输出 D。

（4）D 的左子树为空，对 D 的右子树进行前序遍历。D 的右子树为空，此时 B

的左子树遍历结束。

（5）对 B 的右子树进行前序遍历。B 的右子树的根结点是 E，输出 E。

（6）E 的左子树为空，E 的右子树为空，此时 B 的右子树遍历结束，A 的左子树遍历结束。

（7）对 A 的右子树进行前序遍历。A 的右子树的根结点是 C，输出 C。

（8）对 C 的左子树进行前序遍历。C 的左子树的根结点是 F，输出 F。

（9）F 的左子树为空，F 的右子树为空，此时 C 的左子树遍历结束。

（10）对 C 的右子树进行前序遍历。C 的右子树为空，此时 A 的右子树遍历结束，整棵二叉树的前序遍历完成。

该二叉树的前序遍历结果为：A B D E C F。

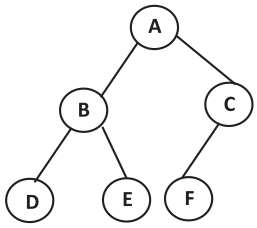


图 5-14　二叉树

2. 中序遍历

中序遍历是二叉树遍历的另一种重要方式，其遍历规则为：先中序遍历左子树，然后访问根结点，最后中序遍历右子树。对于每一棵子树，同样遵循这个规则，即先中序遍历子树的左子树，再访问子树的根结点，最后中序遍历子树的右子树，直到所有结点都被访问完。这种遍历方式使得根结点在其左子树和右子树之间被访问，故而称为“中序”。

通过递归方式实现二叉树的中序遍历代码如下。

```
// 中序遍历函数
void inOrderTraversal(BiTree root) {
    if (root != NULL) {
        // 递归中序遍历左子树
        inOrderTraversal(root->lchild);
        // 访问根结点
        printf("%d ", root->data);
        // 递归中序遍历右子树
        inOrderTraversal(root->rchild);
    }
}
```

对于图 5-14 所示的二叉树，其具体的中序遍历过程如下。

（1）首先对根结点 A 的左子树进行中序遍历。左子树的根结点是 B，继续对 B 的左子树进行中序遍历。

（1）B 的左子树的根结点是 D，由于 D 的左子树为空，此时访问 D，输出 D。

（2）接着对 D 的右子树进行中序遍历，D 的右子树为空，此时 B 的左子树遍历结束。

（3）访问 B，输出 B。

（4）对 B 的右子树进行中序遍历，B 的右子树的根结点是 E，由于 E 的左子树为空，访问 E，输出 E。

（5）E 的右子树为空，此时 B 的右子树遍历结束，A 的左子树遍历结束。

（6）访问根结点 A，输出 A。

（7）对 A 的右子树进行中序遍历。A 的右子树的根结点是 C，继续对 C 的左子树进行中序遍历。

（8）C 的左子树的根结点是 F，由于 F 的左子树为空，访问 F，输出 F。

（9）F 的右子树为空，此时 C 的左子树遍历结束。

（10）对 C 的右子树进行中序遍历，C 的右子树为空，此时 A 的右子树遍历结束，整棵二叉树的中序遍历完成。

该二叉树的中序遍历结果为：D B E A F C 。

3. 后序遍历

后序遍历是二叉树遍历的另一种常见方式，其遍历规则为：先遍历左子树，再遍历右子树，最后访问根结点。对于每一棵子树，同样遵循此规则，即从最底层的左子树开始，逐步向上，先访问子树的左子树，再访问子树的右子树，最后访问子树的根结点，直至整棵二叉树的所有结点都被访问完毕。这种遍历方式使得根结点在其左子树和右子树之后被访问，所以称为“后序”。

后序遍历同样可以通过递归方式实现。基于前面定义的二叉树结点结构（BiTNode），后序遍历的代码如下。

```
// 后序遍历函数
void postOrderTraversal(BiTree root) {
    if (root != NULL) {
        // 递归后序遍历左子树
        postOrderTraversal(root->lchild);
        // 递归后序遍历右子树
        postOrderTraversal(root->rchild);
        // 访问根结点
        printf("%d ", root->data);
    }
}
```

对于图 5–14 所示的二叉树，其后序遍历的详细过程如下。

（1）首先对根结点 A 的左子树进行后序遍历。左子树的根结点是 B，继续对 B 的左子树进行后序遍历。

（2）B 的左子树的根结点是 D，由于 D 的左子树为空，对 D 的右子树进行后序遍历，D 的右子树也为空，此时访问 D，输出 D。

（3）接着访问 B 的右子树，B 的右子树的根结点是 E，由于 E 的左子树为空，对 E 的右子树进行后序遍历，E 的右子树也为空，此时访问 E，输出 E。

（4）然后访问 B，输出 B。此时 A 的左子树遍历结束。

（5）对 A 的右子树进行后序遍历。A 的右子树的根结点是 C，继续对 C 的左子树进行后序遍历。

（6）C 的左子树的根结点是 F，由于 F 的左子树为空，对 F 的右子树进行后序遍历，F 的右子树也为空，此时访问 F，输出 F。

（7）C 的右子树为空，此时访问 C，输出 C。

（8）最后访问根结点 A，输出 A。整棵二叉树的后序遍历完成。

该二叉树的后序遍历结果为：D E B F C A 。

4. 层次遍历

层次遍历是按照二叉树的层次顺序，从根结点开始，逐层从左到右访问每个结点。具体来说，先访问根结点，然后依次访问根结点的左子结点和右子结点，接着访问第二层的所有结点（从左到右），再访问第三层的所有结点，以此类推，直到访问完所有层次的结点。这种遍历方式能够直观地展现二叉树的层次结构，在很多实际应用中非常有用，比如在处理具有层次关系的数据时，层次遍历可以方便地按层次进行分析和处理。

为了实现二叉树的层次遍历，通常会借助队列这一数据结构。其基本思路是：首先将根结点入队，然后从队列中取出一个结点，访问该结点，接着将该结点的左子结点和右子结点（如果存在）依次入队。不断重复这个过程，直到队列变为空，此时所有结点都已被访问，完成了层次遍历。

利用队列实现二叉树层次遍历的代码如下。

```
// 定义二叉树的结点结构
typedef struct BiTNode {
    int data;                   // 结点数据
    struct BiTNode *lchild;     // 指向左子结点的指针
    struct BiTNode *rchild;     // 指向右子结点的指针
} BiTNode, *BiTree;

// 定义队列的结点结构
typedef struct QNode {
    BiTree data;
```

```
    struct QNode *next;
} QNode, *QueuePtr;

// 定义队列结构
typedef struct {
    QueuePtr front;
    QueuePtr rear;
} LinkQueue;

// 初始化队列
void InitQueue(LinkQueue *Q) {
    Q->front = Q->rear = (QueuePtr)malloc(sizeof(QNode));
    if (!Q->front) {
        exit(0);
    }
    Q->front->next = NULL;
}

// 判断队列是否为空
int QueueEmpty(LinkQueue Q) {
    return Q.front == Q.rear;
}

// 入队操作
void EnQueue(LinkQueue *Q, BiTree e) {
    QueuePtr p = (QueuePtr)malloc(sizeof(QNode));
    if (!p) {
        exit(0);
    }
    p->data = e;
    p->next = NULL;
    Q->rear->next = p;
    Q->rear = p;
}

// 出队操作
void DeQueue(LinkQueue *Q, BiTree *e) {
    if (Q->front == Q->rear) {
        return;
    }
    QueuePtr p = Q->front->next;
    *e = p->data;
    Q->front->next = p->next;
    if (Q->rear == p) {
        Q->rear = Q->front;
```

```
        }
        free(p);
    }

    // 层次遍历函数
    void LevelOrderTraversal(BiTree root) {
        LinkQueue Q;
        BiTree p;
        InitQueue(&Q);
        if (root) {
            EnQueue(&Q, root);
        }
        while (!QueueEmpty(Q)) {
            DeQueue(&Q, &p);
            printf("%d ", p->data);
            if (p->lchild) {
                EnQueue(&Q, p->lchild);
            }
            if (p->rchild) {
                EnQueue(&Q, p->rchild);
            }
        }
    }
```

例如，对于图 5-14 所示的二叉树，其层次遍历的过程如下。

（1）首先将根结点 A 入队。

（2）从队列中取出 A，输出 A，并将 A 的左子结点 B 和右子结点 C 入队。此时队列中有 B 和 C。

（3）从队列中取出 B，输出 B，并将 B 的左子结点 D 和右子结点 E 入队。此时队列中有 C、D、E。

（4）从队列中取出 C，输出 C，并将 C 的左子结点 F 入队。此时队列中有 D、E、F。

（5）从队列中取出 D，输出 D，D 没有子结点，不进行入队操作。此时队列中有 E、F。

（6）从队列中取出 E，输出 E，E 没有子结点，不进行入队操作。此时队列中有 F。

（7）从队列中取出 F，输出 F，F 没有子结点，不进行入队操作。此时队列空，层次遍历结束。

该二叉树的层次遍历结果为：A B C D E F 。通过这种方式，能够有效地实现对二叉树的层次遍历，按照从上到下、从左到右的顺序访问二叉树的所有结点。

5.3.2 线索二叉树

在传统的二叉树链式存储结构中，每个结点包含数据域以及指向左、右子结点的指针域。然而，这种结构存在一些局限性，尤其是在遍历二叉树时，若要获取某个结点在遍历序列中的前驱和后继结点，往往需要借助额外的辅助空间（如栈或队列）来实现递归或非递归遍历，操作较为繁琐。同时，二叉链表中存在大量的空指针，对于具有 n 个结点的二叉链表，空指针数量达到 n+1 个，这无疑造成了存储空间的浪费。为了解决这些问题，线索二叉树应运而生。

线索二叉树的核心思想是利用二叉链表中的空指针域，使其不再为空，而是用来存放指向该结点在某种遍历序列中的前驱和后继结点的指针，这些指针被称为线索。而将空指针改为线索的过程就叫做线索化。经过线索化处理后的二叉树，称之为线索二叉树，相应的链表则称为线索链表。

线索二叉树的结点结构在普通二叉树结点结构的基础上，增加了两个标志位，用于区分指针域是指向孩子结点还是线索。其定义如下。

```
// 定义线索二叉树的结点结构
typedef struct BiThrNode {
    int data;                       // 结点数据
    struct BiThrNode *lchild;       // 指向左子结点或前驱的指针
    struct BiThrNode *rchild;       // 指向右子结点或后继的指针
    int LTag;       // 左标志位，0 表示 lchild 指向左子结点，1 表示 lchild 指向前驱
    int RTag;       // 右标志位，0 表示 rchild 指向右子结点，1 表示 rchild 指向后继
} BiThrNode, *BiThrTree;
```

如果结点存在左子树，则指针域 lchild 指向其左孩子结点，否则，指针域 lchild 指向其直接前驱结点。如果结点存在右子树，则指针域 rchild 指向其右孩子结点，否则，指针域 rchild 指向其直接后继结点。为了区分指针域指向的是左 (右) 孩子结点还是直接前驱（后继）结点，这里需要增加两个标志域 ltag 和 rtag。结点的存储结构如图 5–15 所示。

lchild	ltag	data	rtag	rchild

图 5–15　线索二叉树结点结构

其中，当 ltag=0 时，lchild 指向结点的左孩子；当 ltag=1 时，lchild 指向结点的直接前驱结点。当 rtag= 0 时，rchild 指向结点的右孩子；当 rtag=1 时，rchild 指向结点的直接后继结点。

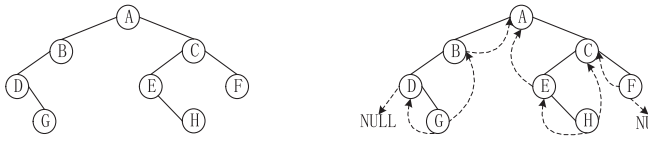
在这种存储结构中，指向前驱和后继结点的指针称为线索。以这种结构组成的二叉链表作为二叉树的存储结构，称为线索链表。对二叉树以某种次序进行遍历并且加上线索的过程称为线索化。线索化了的二叉树称为线索二叉树。

线索化实质上是将二叉链表中的空指针域填上相应结点的遍历前驱或后继结点的地址，而前驱和后继的地址只能在动态的遍历过程中才能得到。因此线索化的过程即为在遍历过程中修改空指针域的过程。对二叉树按照不同的遍历次序进行线索化，可以得到不同的线索二叉树，包括先序线索二叉树、中序线索二叉树和后序线索二叉树。

1. 线索二叉树的建立

以中序线索二叉树为例，建立线索二叉树的过程就是在中序遍历二叉树的过程中，修改空指针域为线索的过程。在遍历过程中，需要设置一个指针 pre，始终指向刚刚访问过的结点，这样在访问当前结点 p 时，pre 就是 p 的前驱结点。

中序线索二叉树的图示表示如图 5–16 所示。



（a）二叉树　　（b）中序线索二叉树

图 5–16　线索二叉树

以下是建立中序线索二叉树的 C 语言实现代码。

```
// 全局变量，始终指向刚刚访问过的结点
BiThrTree pre = NULL;
// 中序线索化二叉树的递归函数
void InThreading(BiThrTree p) {
    if (p) {
        // 递归线索化左子树
        InThreading(p->lchild);

        // 设置前驱线索
        if (!p->lchild) {
            p->LTag = 1;
            p->lchild = pre;
        }

        // 设置后继线索
        if (pre && !pre->rchild) {
            pre->RTag = 1;
            pre->rchild = p;
        }

        // 更新 pre 为当前结点
        pre = p;
```

```
            // 递归线索化右子树
            InThreading(p->rchild);
        }
    }

    // 带头结点的二叉树中序线索化
    BiThrTree InOrderThreading(BiThrTree T) {
        BiThrTree Thrt;

        Thrt = (BiThrTree)malloc(sizeof(BiThrNode));
        if (!Thrt) {
            exit(0);
        }

        // 初始化头结点
        Thrt->LTag = 0;
        Thrt->RTag = 1;
        Thrt->rchild = Thrt;

        if (!T) {
            Thrt->lchild = Thrt;
        } else {
            Thrt->lchild = T;
            pre = Thrt;
            // 对二叉树进行中序线索化
            InThreading(T);
            // 设置最后一个结点的后继线索指向头结点
            pre->rchild = Thrt;
            pre->RTag = 1;
            Thrt->rchild = pre;
        }

        return Thrt;
    }
```

2. 线索二叉树的遍历

对于中序线索二叉树的遍历，由于已经设置了线索，所以遍历过程变得相对简单。可以从二叉树的第一个结点开始，依次通过后继线索找到下一个结点，直到遍历完所有结点。

在中序线索二叉树中，第一个结点是从根结点开始，沿着左子树的指针一直找到左标志位 LTag 为 1 的结点，即最左下端的结点。而对于任意一个结点，若其右标志位 RTag 为 1，则 rchild 指针指向其后继结点；若 RTag 为 0，则其后继结点是其右子树中最左下端的结点。

以下是中序线索二叉树的遍历代码。

```
// 找到中序线索二叉树的第一个结点
BiThrNode* InFirst(BiThrTree Bt) {
    BiThrNode* p = Bt->lchild;
    while (p && p->LTag == 0) {
        p = p->lchild;
    }
    return p;
}

// 找到当前结点的后继结点
BiThrNode* InNext(BiThrNode* p) {
    if (p->RTag == 1) {
        return p->rchild;
    } else {
        BiThrNode* q = p->rchild;
        while (q && q->LTag == 0) {
            q = q->lchild;
        }
        return q;
    }
}

// 中序线索二叉树的遍历算法
void TinOrder(BiThrTree bt) {
    BiThrNode* p;

    p = InFirst(bt);
    while (p) {
        printf("%d ", p->data);
        p = InNext(p);
    }
}
```

对于图 5-17 所示的中序线索二叉树，调用 TinOrder 函数进行遍历的过程如下。

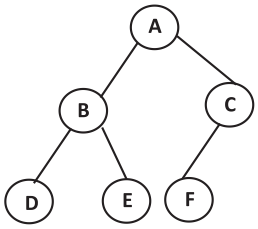


图 5-17　二叉树

（1）首先，调用 InFirst 函数找到第一个结点 D，输出 D 的数据 4。

（2）然后，调用 InNext 函数找 D 的后继结点。由于 D 的 RTag 为 1，所以其后继结点是 D 的 rchild 指针指向的结点，

即 B，输出 B 的数据 2。

（3）接着，找 B 的后继结点。B 的 RTag 为 0，所以其后继结点是 B 右子树中最左下端的结点，即 E，输出 E 的数据 5。

（4）再找 E 的后继结点。E 的 RTag 为 1，所以其后继结点是 E 的 rchild 指针指向的结点，即 A，输出 A 的数据 1。

（5）继续找 A 的后继结点。A 的 RTag 为 0，所以其后继结点是 A 右子树中最左下端的结点，即 F，输出 F 的数据 6。

（6）最后，找 F 的后继结点。F 的 RTag 为 1，所以其后继结点是 F 的 rchild 指针指向的结点，即 C，输出 C 的数据 3。

（7）此时，遍历结束，得到的中序遍历结果为：4 2 5 1 6 3 。

通过这种方式，利用线索二叉树的线索，可以高效地实现二叉树的遍历。

5.4 树和森林

5.4.1 树的存储结构

树的存储结构有多种形式，常见的有双亲表示法、孩子表示法和孩子兄弟表示法。这些存储结构各有特点，适用于不同的应用场景，下面将分别进行介绍。

1. 双亲表示法

双亲表示法是一种以每个结点的双亲结点为线索来存储树的方法。其核心思想是用一组连续的存储单元存储树的结点，同时在每个结点中设置一个指针域，指示其双亲结点在数组中的位置。这种表示法利用了每个结点（除根结点外）都有唯一双亲的性质，使得查找某个结点的双亲变得非常高效。

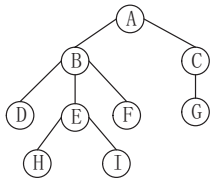
双亲表示法是根据树中除根结点以外其他每个结点都有唯一的双亲这个特点，用一组连续的存储单元存储树的每个结点，每个结点除存放本身的信息外，还要存放其双亲在数组中的位置。这样数组元素就是一个结构体类型的数据，每个结点有两个域，如图 5–18 所示。

data	parent

图 5–18　树的双亲表示结点结构

其中，data 域存放结点本身的信息；parent 域存放该结点的双亲在数组中的位置。

树的双亲表示如图 5–19 所示。



（a）树

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

（b）树的双亲表示

图 5–19　树及其双亲表示

用结构体数组来实现双亲表示法。以下是具体的实现代码。

```
#define MAX_TREE_SIZE 100 // 定义树的最大结点数

typedef struct {
    int data;                    // 结点的数据
    int parent;                  // 双亲结点在数组中的下标
} PTNode;

typedef struct {
    PTNode nodes[MAX_TREE_SIZE]; // 存储树中所有结点的数组
    int root;                    // 根结点的下标
    int n;                       // 树中结点的个数
} PTree;

// 初始化树
void initPTree(PTree *tree) {
    tree->root = -1;
    tree->n = 0;
    for (int i = 0; i < MAX_TREE_SIZE; i++) {
        tree->nodes[i].parent = -1; // 用 -1 表示无双亲，即根结点
    }
}

// 插入结点到树中
void insertNode(PTree *tree, int data, int parent) {
    if (tree->n >= MAX_TREE_SIZE) {
        printf(" 树已满，无法插入新结点 \n");
        return;
    }
    tree->nodes[tree->n].data = data;
    tree->nodes[tree->n].parent = parent;
```

```
        if (parent == -1) {
            tree->root = tree->n;
        }
        tree->n++;
    }
```

2. 孩子表示法

孩子表示法是一种通过存储每个结点的孩子结点来表示树的方法。它有多种实现方式，其中一种常见的方式是为每个结点建立一个孩子链表，链表中存储该结点的所有孩子结点。这种表示法能够直观地反映出每个结点与其孩子结点之间的关系，方便对孩子结点进行操作。

由于树中每个结点可能有多棵子树，则可用多重链表，即每个结点有多个指针域，其中每个指针指向一棵子树的根结点，此时链表中的结点可以有如图 5-20 所示的两种结点格式。

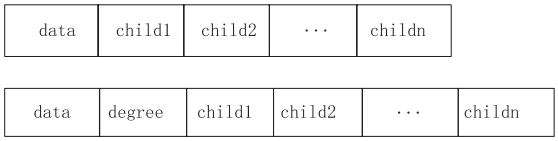


图 5-20　树结点的两种结构

下面图 5-21、5-22 分别表示为树的孩子链表表示及带双亲的孩子链表表示。

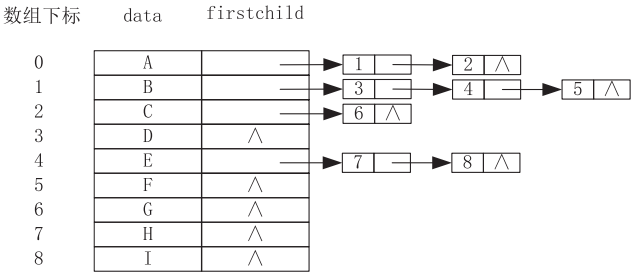


图 5-21　树的孩子链表表示

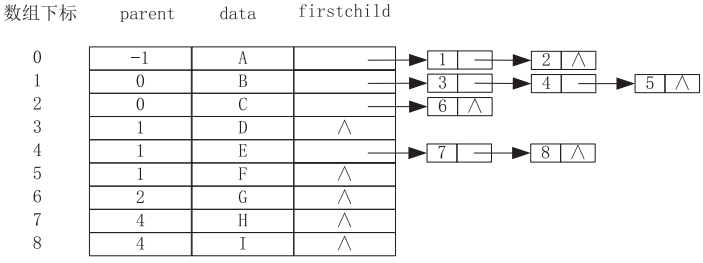


图 5-22　树的带双亲的孩子链表表示

用结构体和链表来实现孩子表示法。以下是具体的实现代码。

```
#define MAX_TREE_SIZE 100 // 定义树的最大结点数
typedef struct ChildNode {
    int child;                    // 孩子结点在数组中的下标
    struct ChildNode *next;       // 指向下一个孩子结点的指针
} ChildNode;

typedef struct {
    int data;                     // 结点的数据
    ChildNode *firstChild;        // 指向第一个孩子结点的指针
} CTNode;

typedef struct {
    CTNode nodes[MAX_TREE_SIZE]; // 存储树中所有结点的数组
    int n;                        // 树中结点的个数
} CTree;

// 初始化树
void initCTree(CTree *tree) {
    tree->n = 0;
    for (int i = 0; i < MAX_TREE_SIZE; i++) {
        tree->nodes[i].firstChild = NULL;
    }
}

// 插入孩子结点
void insertChild(CTree *tree, int parent, int child) {
    if (tree->n >= MAX_TREE_SIZE) {
        printf(" 树已满，无法插入新结点 \n");
        return;
    }
    ChildNode *newChild = (ChildNode *)malloc(sizeof(ChildNode));
    newChild->child = child;
    newChild->next = tree->nodes[parent].firstChild;
    tree->nodes[parent].firstChild = newChild;
    tree->n++;
}
```

3. 孩子兄弟表示法

孩子兄弟表示法是一种用二叉链表来表示树的方法。它的核心思想是将树中的每个结点用一个包含三个域的结点表示，分别是数据域、指向第一个孩子结点的指针域和指向右兄弟结点的指针域。这种表示法将树转化为一种类似于二叉树的结构，使得可以利用二叉树的一些算法来处理树的问题。

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第一个孩子的右兄弟结点。孩子兄弟表示如图 5–23 所示。

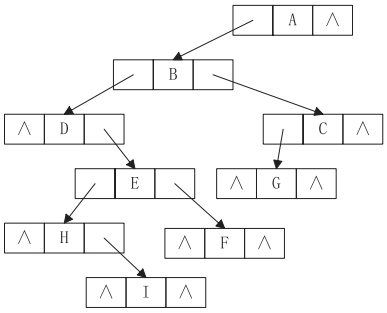


图 5–23　树的孩子兄弟表示

用结构体来实现孩子兄弟表示法。以下是具体的实现代码。

```
typedef struct CSNode {
    int data;                          // 结点的数据
    struct CSNode *firstChild;    // 指向第一个孩子结点的指针
    struct CSNode *nextSibling; // 指向右兄弟结点的指针
} CSNode, *CSTree;

// 创建树的结点
CSNode* createCSNode(int value) {
    CSNode* newNode = (CSNode*)malloc(sizeof(CSNode));
    newNode–>data = value;
    newNode–>firstChild = NULL;
    newNode–>nextSibling = NULL;
    return newNode;
}

// 插入孩子结点
void insertChild(CSTree *parent, CSNode *child) {
    if ((*parent)–>firstChild == NULL) {
        (*parent)–>firstChild = child;
    } else {
        CSNode *p = (*parent)–>firstChild;
        while (p–>nextSibling != NULL) {
            p = p–>nextSibling;
        }
        p–>nextSibling = child;
    }
}
```

5.4.2 树、森林与二叉树的转换

树、森林与二叉树之间存在着密切的联系，它们可以相互转换。这种转换关系在实际应用中非常重要，例如在数据处理、编译原理等领域，可以通过将树或森林转换为二叉树，利用二叉树的高效算法进行处理，然后再将结果转换回树或森林。

1. 树转换为二叉树

树转换为二叉树的步骤如下。

加线：在树中所有兄弟结点之间加一条连线。这一步的目的是将树中原本分散的兄弟关系通过连线明确表示出来，为后续的转换做准备。例如，对于图 5–24 (a) 所示的树，在 B、C、D 之间和 E、F、G 之间添加连线，得到图 5–24(b)。

去线：对树的每个结点，只保留它与第一个孩子的连线，删除它与其他孩子的连线。经过这一步，每个结点只保留了一个主要的分支，即第一个孩子分支，其他孩子分支被删除，使得树的结构更接近二叉树的形式。如图 5–24(c) 所示，删除了 A 与 C、D 和 C 与 F、G 的连线。

层次调整：以树的根结点为轴心，将整棵树顺时针旋转一定的角度，使结构层次分明。旋转后，结点的左孩子是原来结点的第一个孩子，右孩子是结点原来的兄弟。这样就完成了树到二叉树的转换，得到图 5–24(d)。

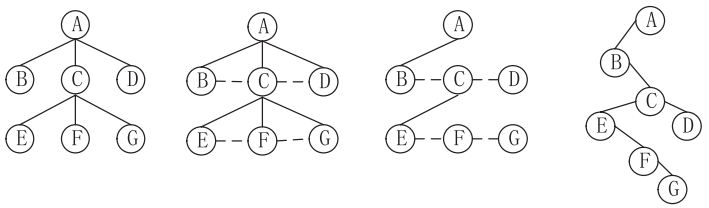


图 5–24　树转换为二叉树的过程

实现树到二叉树的转换可以通过递归的方式进行。假设已经定义了树的孩子兄弟表示法的结构体，以下是转换的代码。

```
// 树转换为二叉树的递归函数
BiTree treeToBiTree(CSTree t) {
    BiTree b;

    if
 (t == NULL) {
        b = NULL;
    } else {
        b = (BiTree)malloc(sizeof(BiTNode));
        b->data = t->data;
```

```
        b->lchild = treeToBiTree(t->firstChild);   // 第一个孩子转换为左孩子
        b->rchild = treeToBiTree(t->nextSibling); // 右兄弟转换为右孩子
    }

    return b;
}
```

2. 森林转换为二叉树

森林是由若干棵树组成的，将森林转换为二叉树的步骤如下。

将每棵树分别转换成二叉树：按照树转换为二叉树的方法，将森林中的每棵树逐一转换为二叉树。例如，对于包含两棵树的森林，分别对这两棵树进行树到二叉树的转换操作。

将每棵树的根结点用线相连：第一棵二叉树不动，从第二棵二叉树开始，依次把后一棵二叉树的根结点作为前一棵二叉树的根结点的右孩子，用线连接起来。通过这种方式，将多棵二叉树连接成一棵更大的二叉树，完成森林到二叉树的转换。

假设有一个森林，包含两棵树 T1 和 T2，T1 的根为 A，T2 的根为 B。将 T1 和 T2 分别转换为二叉树后，设转换后的二叉树分别为 B1 和 B2。此时，将 B2 的根结点 B 作为 B1 的根结点 A 的右孩子，从而得到一个完整的二叉树。

实现森林到二叉树的转换可以基于树到二叉树的转换函数进行扩展。假设用一个数组来存储森林中的树，数组元素为指向树结点的指针，以下是转换的代码。

```
// 森林转换为二叉树的函数
BiTree forestToBiTree(CSTree forest[], int n) {
    BiTree root = NULL;
    BiTree lastBiTree = NULL;

    for (int i = 0; i < n; i++) {
        BiTree biTree = treeToBiTree(forest[i]);
        if (i == 0) {
            root = biTree;
            lastBiTree = biTree;
        } else {
            lastBiTree->rchild = biTree;
            lastBiTree = biTree;
        }
    }

    return root;
}
```

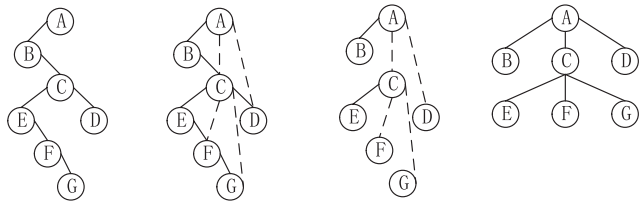
3. 二叉树转换为树

二叉树转换为树是树转换为二叉树的逆过程，步骤如下。

加线：若某结点 X 的左孩子结点存在，则将这个左孩子的右孩子结点、右孩子的右孩子结点、右孩子的右孩子的右孩子结点…，都作为结点 X 的孩子。将结点 X 与这些右孩子结点用线连接起来。这一步是为了恢复树中原本的孩子关系，通过右孩子指针链找到所有的孩子结点。例如，对于图 5–25(a) 所示的二叉树，从根结点 A 的左孩子 B 开始，B 的右孩子是 C，C 的右孩子是 D，所以将 A 与 C、D 用线连接起来；C 的左孩子 E 的右孩子是 F、G，将 C 与 F、G 用线连接起来，得到图 5–25(b)。

去线：删除原二叉树中所有结点与其右孩子结点的连线。这一步是去除在二叉树转换过程中添加的用于表示兄弟关系的右孩子指针，恢复树的结构。如图 5–25(c) 所示，删除 B 与 C、C 与 D、E 与 F、F 与 G 的右孩子连线。

层次调整：对树的结构进行调整，使其层次清晰。将图 5–25(c) 调整后得到图 5–25(d) 所示的树。



(a) 二叉树　(b) 加线　(c) 去线　(d) 旋转

图 5–25　二叉树转换为树的过程

实现二叉树到树的转换可以通过递归的方式进行。假设已经定义了树的孩子兄弟表示法的结构体 CSNode 和二叉树的结构体 BiTNode，以下是转换的代码。

```
// 二叉树转换为树的递归函数
CSTree biTreeToTree(BiTree b) {
    CSTree t;

    if (b == NULL) {
        t = NULL;
    } else {
        t = (CSTree)malloc(sizeof(CSNode));
        t->data = b->data;
        t->firstChild = biTreeToTree(b->lchild);   // 左孩子转换为第一个孩子
        CSTree rightSibling = biTreeToTree(b->rchild); // 右孩子转换为右兄弟
        CSTree current = t->firstChild;

        if (current != NULL) {
            while (current->nextSibling != NULL) {
```

```
                current = current->nextSibling;
            }
            current->nextSibling = rightSibling;
        } else {
            t->firstChild = rightSibling;
        }
    }

    return t;
}
```

4. 二叉树转换为森林

判断一棵二叉树是否能够转换成一棵树还是森林，标准就是看这颗二叉树根结点有没有右孩子，有右孩子就是森林，没有就是一棵树。二叉树转换为森林的步骤如下：

从根节点开始，若右孩子存在，则把与右孩子结点的连线删除：这一步将二叉树分解为多棵独立的二叉树，每棵二叉树对应森林中的一棵树。根结点有右孩子，删除根与右孩子的连线，得到多棵独立的二叉树。

再查看分离后的二叉树，若其根节点的右孩子存在，则连线删除… ：重复上述步骤，直到所有这些根节点与右孩子的连线都删除为止。

将每棵分离后的二叉树转换为树：按照二叉树转换为树的方法，将每棵分离后的二叉树转换为树，从而得到森林。

实现二叉树到森林的转换可以基于二叉树到树的转换函数进行扩展。假设用一个数组来存储森林中的树，数组元素为指向树结点的指针，以下是转换的代码。

```
// 二叉树转换为森林的函数
void biTreeToForest(BiTree b, CSTree forest[], int *n) {
    *n = 0;

    while (b != NULL) {
        BiTree rightChild = b->rchild;

        b->rchild = NULL;

        forest[*n] = biTreeToTree(b);

        (*n)++;

        b = rightChild;
    }
}
```

5.4.3 树和森林的遍历

1. 树的遍历

树的遍历主要有两种方式：先根遍历和后根遍历。这两种遍历方式基于树的递归结构，能够全面地访问树中的每一个结点，为后续的树操作和数据处理提供基础。

先根遍历：先访问树的根结点，然后依次先根遍历根的每棵子树。

后根遍历：先依次后根遍历每棵子树，然后再访问根结点。

基于孩子兄弟表示法的先根遍历和后根遍历实现代码如下。

```
// 树的先根遍历
void preOrderTraversal(CSTree t) {
    if (t != NULL) {
        // 访问根结点
        printf("%d ", t->data);
        // 递归先根遍历第一棵子树
        preOrderTraversal(t->firstChild);
        // 递归先根遍历下一棵兄弟子树
        preOrderTraversal(t->nextSibling);
    }
}

// 树的后根遍历
void postOrderTraversal(CSTree t) {
    if (t != NULL) {
        // 递归后根遍历第一棵子树
        postOrderTraversal(t->firstChild);
        // 递归后根遍历下一棵兄弟子树
        postOrderTraversal(t->nextSibling);

        // 访问根结点
        printf("%d ", t->data);
    }
}
```

2. 森林的遍历

森林的遍历方式主要有先序遍历和中序遍历。这两种遍历方式是基于森林由多棵树组成的特点，通过对每棵树的遍历，实现对整个森林的遍历。

森林的先序遍历：若森林非空，则先访问森林中第一棵树的根结点，然后先序遍历第一棵树的子树森林，最后先序遍历除去第一棵树之后剩余的树构成的森林。

森林的中序遍历：若森林非空，则先中序遍历森林中第一棵树的子树森林，然后访问第一棵树的根结点，最后中序遍历除去第一棵树之后剩余的树构成的森林。

基于孩子兄弟表示法的森林先序遍历和中序遍历实现代码如下。

```
// 森林的先序遍历
void preOrderForest(CSTree forest[], int n) {
    for (int i = 0; i < n; i++) {
        if (forest[i] != NULL) {
            // 访问第一棵树的根结点
            printf("%d ", forest[i]->data);
            // 先序遍历第一棵树的子树森林
            preOrderTraversal(forest[i]->firstChild);
            // 先序遍历除去第一棵树之后剩余的树构成的森林
            preOrderForest(forest + i + 1, n - i - 1);
        }
    }
}
// 森林的中序遍历
void inOrderForest(CSTree forest[], int n) {
    for (int i = 0; i < n; i++) {
        if (forest[i] != NULL) {
            // 中序遍历第一棵树的子树森林
            inOrderTraversal(forest[i]->firstChild);
            // 访问第一棵树的根结点
            printf("%d ", forest[i]->data);
            // 中序遍历除去第一棵树之后剩余的树构成的森林
            inOrderForest(forest + i + 1, n - i - 1);
        }
    }
}
```

5.5 哈夫曼树及其应用

5.5.1 哈夫曼树的定义

在深入探讨哈夫曼树之前，需要先了解几个重要的概念。首先是路径，在一棵树中，从一个结点到另一个结点所经过的分支序列被称为路径。例如，在图 5-26 的二叉树中，从根结点 A 到结点 D 的路径为 A-B-D。路径长度则是路径上分支的数目，所以从 A 到 D 的路径长度为 2。

结点的权：将树中结点赋予一个具有某种含义的数值，这个数值就称为该结点的权。比如在一个表示不同城市人口数量的树结构中，每个城市对应的结点的权可以是该城市的人口数。

结点的带权路径长度是指从根结点到该结点之间的路径长度与该结点的权的乘积。例如，在图 5–27 中，假设结点 A 的权为 1，结点 B 的权为 2，结点 D 的权为 3。那么结点 D 的带权路径长度为从根结点 A 到结点 D 的路径长度 2 乘以结点 D 的权 3，即 $2 \times 3 = 6$。

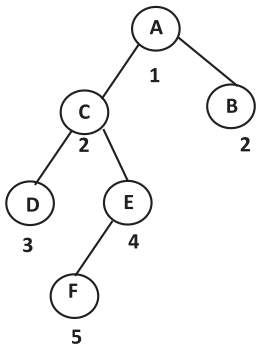


图 5–26　二叉树

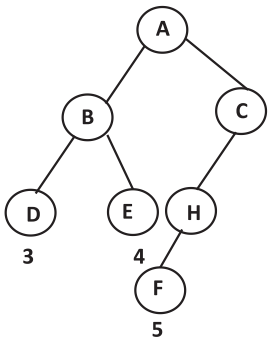
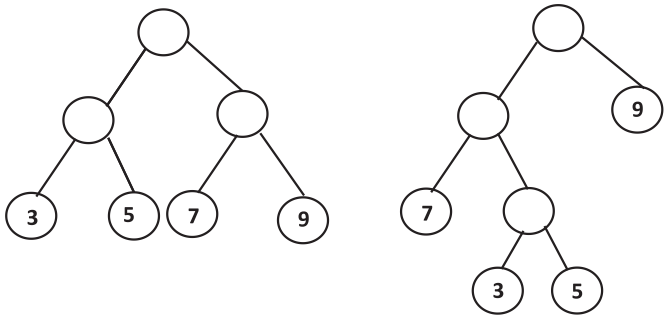


图 5–27　带权二叉树

而树的带权路径长度规定为所有叶子结点的带权路径长度之和，通常记为 WPL（Weighted Path Length）。对于图 5–26 所示的二叉树，其叶子结点为 D、E、F，假设它们的权分别为 3、4、5。从根结点 A 到 D 的路径长度为 2，到 E 的路径长度为 2，到 F 的路径长度为 3。则该树的带权路径长度 $WPL = 3 \times 2 + 4 \times 2 + 5 \times 3 = 6 + 8 + 15 = 29$。

有了这些概念，就可以给哈夫曼树下定义了。给定 n 个权值作为 n 个叶子结点，构造一棵二叉树，若该二叉树的带权路径长度达到最小，那么这样的二叉树就被称为最优二叉树，也叫做哈夫曼树。例如，给定权值集合 {3, 5, 7, 9}，可以构造出不同的二叉树，如图 5–28(a) 和图 5–28(b) 所示。

A 到结点 D 的路径长度 2 乘以结点 D 的权 3，即 $2 \times 3 = 6$。



(a)　　(b)

图 5–28　WPL 二叉树

对于图 5–28(a) 所示的二叉树，其带权路径长度 $WPL = 3 \times 2 + 5 \times 2 + 7 \times 2 + 9 \times 2 = 6 + 10 + 14 + 18 = 48$。

对于图 5–28(b) 所示的二叉树，其带权路径长度 WPL = 3 × 3 + 5 × 3 + 7 × 2 + 9 × 1 = 9 + 15 + 14 + 9 = 47。

通过比较可以发现，图 5–28(b) 所示的二叉树带权路径长度更小，当继续尝试其他可能的二叉树结构时，会发现图 5–28(b) 所示的二叉树就是这组权值对应的哈夫曼树，其带权路径长度在所有可能的二叉树中是最小的。哈夫曼树在数据压缩、通信编码等领域有着广泛的应用，其核心优势就在于能够通过合理的树结构构建，使得带权路径长度最小化，从而实现高效的数据处理。

5.5.2 哈夫曼树的构造

哈夫曼树的构造算法是基于贪心策略的，其核心思想是每次从当前的结点集合中选取权值最小的两个结点，将它们合并为一个新的结点，新结点的权值为这两个被合并结点的权值之和。重复这个过程，直到最终形成一棵完整的二叉树，这棵二叉树就是哈夫曼树。

具体的构造步骤如下。

初始化：假设有 n 个权值，将每个权值都视为一个独立的结点，这些结点构成一个森林，每个结点就是一棵只有根结点的树，其权值为 W_i。

选择与合并：从森林中选取两个根结点权值最小的树，将它们作为新结点的左右子树，新结点的权值为这两个子树根结点权值之和。

更新森林：从森林中删除这两棵被选取的树，并将新生成的树加入森林。

重复步骤：不断重复步骤 2 和步骤 3，直到森林中只剩下一棵树，这棵树就是哈夫曼树。

下面通过一个实例来演示哈夫曼树的构造过程。假设有一组权值 {5, 9, 12, 13, 16, 45}，构造哈夫曼树的过程如下。

初始化森林：将每个权值都视为一棵独立的树，此时森林中有 6 棵树，分别为：

- 树 1：权值为 5
- 树 2：权值为 9
- 树 3：权值为 12
- 树 4：权值为 13
- 树 5：权值为 16
- 树 6：权值为 45

第一次选择与合并：在森林中选择权值最小的两棵树，即权值为 5 和 9 的树。将它们合并为一棵新树，新树的权值为 5 + 9 = 14，新树的左子树为权值 5 的树，右子树为权值 9 的树。此时森林中剩下 5 棵树，分别为：

- 树 1：权值为 14（由 5 和 9 合并而成）
- 树 2：权值为 12
- 树 3：权值为 13

– 树 4：权值为 16

– 树 5：权值为 45

第二次选择与合并：在当前森林中选择权值最小的两棵树，即权值为 12 和 13 的树。将它们合并为一棵新树，新树的权值为 12 + 13 = 25，新树的左子树为权值 12 的树，右子树为权值 13 的树。此时森林中剩下 4 棵树，分别为：

– 树 1：权值为 14（由 5 和 9 合并而成）

– 树 2：权值为 25（由 12 和 13 合并而成）

– 树 3：权值为 16

– 树 4：权值为 45

第三次选择与合并：在当前森林中选择权值最小的两棵树，即权值为 14 和 16 的树。将它们合并为一棵新树，新树的权值为 14 + 16 = 30，新树的左子树为权值 14 的树，右子树为权值 16 的树。此时森林中剩下 3 棵树，分别为：

– 树 1：权值为 30（由 14 和 16 合并而成）

– 树 2：权值为 25（由 12 和 13 合并而成）

– 树 3：权值为 45

第四次选择与合并：在当前森林中选择权值最小的两棵树，即权值为 25 和 30 的树。将它们合并为一棵新树，新树的权值为 25 + 30 = 55，新树的左子树为权值 25 的树，右子树为权值 30 的树。此时森林中剩下 2 棵树，分别为：

– 树 1：权值为 55（由 25 和 30 合并而成）

– 树 2：权值为 45

第五次选择与合并：将最后两棵树（权值为 45 和 55 的树）合并为一棵新树，新树的权值为 45 + 55 = 100，新树的左子树为权值 45 的树，右子树为权值 55 的树。此时森林中只剩下一棵树，这棵树就是最终构造的哈夫曼树。

通过结构体和链表来实现哈夫曼树的构造。以下是具体的实现代码。

```
#include <stdio.h>
#include <stdlib.h>

// 定义哈夫曼树的结点结构
typedef struct HuffmanNode {
    int weight;                 // 结点的权值
    struct HuffmanNode *left;   // 指向左子结点的指针
    struct HuffmanNode *right;  // 指向右子结点的指针
} HuffmanNode, *HuffmanTree;

// 创建一个新的哈夫曼树结点
HuffmanNode* createHuffmanNode(int weight) {
    HuffmanNode* newNode = (HuffmanNode*)malloc(sizeof(HuffmanNode));
    newNode->weight = weight;
```

```
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

// 比较函数，用于在优先队列中比较结点的权值大小
int compare(const void *a, const void *b) {
    return ((HuffmanNode *)a)->weight - ((HuffmanNode *)b)->weight;
}

// 构造哈夫曼树
HuffmanTree buildHuffmanTree(int weights[], int n) {
    HuffmanNode *nodes[n];

    // 初始化每个权值为一个独立的结点
    for (int i = 0; i < n; i++) {
        nodes[i] = createHuffmanNode(weights[i]);
    }

    // 使用 qsort 函数对结点按权值进行排序
    qsort(nodes, n, sizeof(HuffmanNode *), compare);

    // 不断合并权值最小的两个结点，直到只剩下一棵树
    while (n > 1) {
        // 取出权值最小的两个结点
        HuffmanNode *left = nodes[0];
        HuffmanNode *right = nodes[1];

        // 创建一个新的父结点，权值为两个子结点权值之和
        HuffmanNode *parent = createHuffmanNode(left->weight + right->weight);
        parent->left = left;
        parent->right = right;

        // 将新的父结点插入到数组中，并调整数组顺序
        for (int i = 2; i < n; i++) {
            nodes[i - 2] = nodes[i];
        }
        nodes[n - 2] = parent;
        n--;

        // 重新对数组进行排序
        qsort(nodes, n, sizeof(HuffmanNode *), compare);
    }
    return nodes[0];
}
```

5.5.3 哈夫曼算法的实现

在上述哈夫曼树构造代码的基础上，可以进一步实现哈夫曼树的一些基本操作，如计算哈夫曼树的带权路径长度。带权路径长度的计算对于验证哈夫曼树的最优性非常重要，它体现了哈夫曼树在数据处理中对权值和路径长度综合考量的优化效果。

以下是计算哈夫曼树带权路径长度的 C 语言实现代码。

```
// 计算哈夫曼树的带权路径长度
int calculateWPL(HuffmanTree root, int depth) {
    if (root == NULL) {
        return 0;
    }

    // 如果是叶子结点，返回权值乘以深度
    if (root->left == NULL && root->right == NULL) {
        return root->weight * depth;
    }

    // 递归计算左子树和右子树的带权路径长度
    return calculateWPL(root->left, depth + 1) + calculateWPL(root->right, depth + 1);
}
```

对于之前构造的哈夫曼树，可以通过以下方式调用 calculateWPL 函数来计算其带权路径长度。

```
int main() {
    int weights[] = {5, 9, 12, 13, 16, 45};
    int n = sizeof(weights) / sizeof(weights[0]);

    HuffmanTree root = buildHuffmanTree(weights, n);

    int wpl = calculateWPL(root, 0);

    printf(" 哈夫曼树的带权路径长度为 : %d\n", wpl);

    return 0;
}
```

在 main 函数中，首先定义了权值数组 weights 和数组元素个数 n，然后调用 buildHuffmanTree 函数构造哈夫曼树，得到根结点指针 root。接着调用 calculateWPL 函数计算哈夫曼树的带权路径长度，并将结果存储在 wpl 变量中，最后输出带权路径长度的值。

5.5.4 哈夫曼编码

哈夫曼编码是哈夫曼树在数据编码领域的重要应用，它利用哈夫曼树的特性，为每个字符分配最优的二进制编码，从而实现数据的高效压缩和传输。其核心思想是根据字符出现的频率来构建哈夫曼树，然后通过遍历哈夫曼树为每个字符生成唯一的二进制编码，使得出现频率高的字符对应的编码较短，出现频率低的字符对应的编码较长，这样在整体上能够使编码后的文本长度最短。

对于字符串“ABACCDA”，统计每个字符出现的频率如下：A 出现 3 次，B 出现 1 次，C 出现 2 次，D 出现 1 次。以这些频率作为权值来构造哈夫曼树，构造过程如下：

初始化森林：将每个字符及其频率作为独立的结点，形成一个森林。此时森林中有 4 棵树，分别为：

- 树 1：字符 A，权值 3
- 树 2：字符 B，权值 1
- 树 3：字符 C，权值 2
- 树 4：字符 D，权值 1

第一次选择与合并：在森林中选择权值最小的两棵树，即字符 B 和字符 D，将它们合并为一棵新树。新树的权值为 1 + 1 = 2，左子树为字符 B，右子树为字符 D。此时森林中剩下 3 棵树，分别为：

- 树 1：字符 A，权值 3
- 树 2：新树（由 B 和 D 合并而成），权值 2
- 树 3：字符 C，权值 2

第二次选择与合并：在当前森林中选择权值最小的两棵树，即新树（由 B 和 D 合并而成）和字符 C，将它们合并为一棵新树。新树的权值为 2 + 2 = 4，左子树为新树（由 B 和 D 合并而成），右子树为字符 C。此时森林中剩下 2 棵树，分别为：

- 树 1：字符 A，权值 3
- 树 2：新树（由 B、D、C 合并而成），权值 4

第三次选择与合并：将最后两棵树合并为一棵新树，即哈夫曼树。新树的权值为 3 + 4 = 7，左子树为字符 A，右子树为新树（由 B、D、C 合并而成）。

构建好哈夫曼树后，通过遍历哈夫曼树为每个字符生成编码。约定从根结点到叶子结点的路径中，向左走分配代码“0”，向右走分配代码“1”。

对于字符 A，从根结点到 A 的路径为左子树，所以 A 的编码为“0”。

对于字符 B，从根结点开始，先到右子树（新树，由 B、D、C 合并而成），再到左子树（B 所在的子树），所以 B 的编码为“100”。

对于字符 C，从根结点开始，先到右子树（新树，由 B、D、C 合并而成），再

到右子树（C 所在的子树），所以 C 的编码为“11”。

对于字符 D，从根结点开始，先到右子树（新树，由 B、D、C 合并而成），再到左子树（新树，由 B 和 D 合并而成），最后到右子树（D 所在的子树），所以 D 的编码为“101”。

通过这种方式，得到的哈夫曼编码为：

A: 0

B: 100

C: 11

D: 101

原字符串“ABACCDA”经过哈夫曼编码后变为“01000011111010”，相比等长编码（假设采用 3 位等长编码，原字符串编码后为“000001000000010010101000”），编码长度明显缩短，实现了数据的压缩。

实现哈夫曼编码的代码如下。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// 定义哈夫曼树的结点结构
typedef struct HuffmanNode {
    char data;                  // 结点的字符
    int weight;                 // 结点的权值
    struct HuffmanNode *left;   // 指向左子结点的指针
    struct HuffmanNode *right;  // 指向右子结点的指针
} HuffmanNode, *HuffmanTree;

// 创建一个新的哈夫曼树结点
HuffmanNode* createHuffmanNode(char data, int weight) {
    HuffmanNode* newNode = (HuffmanNode*)malloc(sizeof(HuffmanNode));
    newNode->data = data;
    newNode->weight = weight;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

// 比较函数，用于在优先队列中比较结点的权值大小
int compare(const void *a, const void *b) {
    return ((HuffmanNode *)a)->weight - ((HuffmanNode *)b)->weight;
}

// 构造哈夫曼树
```

```
HuffmanTree buildHuffmanTree(char data[], int weight[], int n) {
    HuffmanNode *nodes[n];

    // 初始化每个权值为一个独立的结点
    for (int i = 0; i < n; i++) {
        nodes[i] = createHuffmanNode(data[i], weight[i]);
    }

    // 使用 qsort 函数对结点按权值进行排序
    qsort(nodes, n, sizeof(HuffmanNode *), compare);

    // 不断合并权值最小的两个结点，直到只剩下一棵树
    while (n > 1) {
        // 取出权值最小的两个结点
        HuffmanNode *left = nodes[0];
        HuffmanNode *right = nodes[1];

        // 创建一个新的父结点，权值为两个子结点权值之和
        HuffmanNode *parent = createHuffmanNode('\0', left->weight + right->weight);
        parent->left = left;
        parent->right = right;

        // 将新的父结点插入到数组中，并调整数组顺序
        for (int i = 2; i < n; i++) {
            nodes[i - 2] = nodes[i];
        }
        nodes[n - 2] = parent;
        n--;

        // 重新对数组进行排序
        qsort(nodes, n, sizeof(HuffmanNode *), compare);
    }

    return nodes[0];
}

// 生成哈夫曼编码
void generateHuffmanCodes(HuffmanTree root, char code[], int depth) {
    if (root == NULL) {
        return;
    }

    if (root->left == NULL && root->right == NULL) {
        code[depth] = '\0';
        printf("%c: %s\n", root->data, code);
```

```
            return;
        }

        code[depth] = '0';
        generateHuffmanCodes(root->left, code, depth + 1);

        code[depth] = '1';
        generateHuffmanCodes(root->right, code, depth + 1);
    }

    int main() {
        char data[] = {'A','B','C','D'};
        int weight[] = {3, 1, 2, 1};
        int n = sizeof(data) / sizeof(data[0]);

        HuffmanTree root = buildHuffmanTree(data, weight, n);

        char code[100];
        generateHuffmanCodes(root, code, 0);

        return 0;
    }
```

5.6 项目实践

为了实现文件压缩与解压缩功能，需要设计合适的数据结构来存储哈夫曼树的结点以及字符频率等信息。

首先定义哈夫曼树的结点结构如下。

```
// 定义哈夫曼树的结点结构
typedef struct HuffmanNode {
    char data;                  // 结点的字符
    int weight;                 // 结点的权值
    struct HuffmanNode *left;   // 指向左子结点的指针
    struct HuffmanNode *right;  // 指向右子结点的指针
} HuffmanNode, *HuffmanTree;
```

在这个结构体中，data 用于存储字符，weight 表示该字符在文件中出现的频率，left 和 right 分别指向左、右子结点。

为了方便统计字符频率，使用一个数组来存储每个字符及其对应的频率。假设字符集为 ASCII 码，数组大小为 256，因为 ASCII 码共有 256 个字符。

```
// 定义字符频率数组
int frequency[256] = {0};
```

通过遍历文件内容，每读取一个字符，就在对应的数组位置上增加频率计数。例如，当读取到字符 'A' 时，frequency['A']++。

同时，为了生成哈夫曼编码，需要一个辅助数组来存储每个字符的编码。可以定义一个二维字符数组，第一维表示字符的数量，第二维表示编码的最大长度（根据实际情况确定合适的长度）。

```
// 定义哈夫曼编码数组
char huffmanCodes[256][100];
```

这里假设每个字符的哈夫曼编码最长为 100 位，实际应用中可根据具体情况调整。通过遍历哈夫曼树，为每个字符生成编码并存储在这个数组中。

1. 字符频率统计算法

遍历输入文件，每次读取一个字符，将其对应的频率数组位置的值加 1。

```
// 统计字符频率
void countFrequency(const char *filename) {
        FILE *file = fopen(filename, “rb”);
        if (file == NULL) {
            printf(“ 无法打开文件 \n”);
            return;
        }
        int ch;
        while ((ch = fgetc(file)) != EOF) {
            frequency[ch]++;
        }

        fclose(file);
}
```

2. 哈夫曼树构建算法

基于字符频率数组，选择权值最小的两个结点，合并成一个新结点，不断重复这个过程，直到形成一棵完整的哈夫曼树。

```
// 比较函数，用于在优先队列中比较结点的权值大小
int compare(const void *a, const void *b) {
    return ((HuffmanNode *)a)->weight – ((HuffmanNode *)b)->weight;
}
// 构造哈夫曼树
HuffmanTree buildHuffmanTree() {
```

```
    HuffmanNode *nodes[256];
    int count = 0;
    for (int i = 0; i < 256; i++) {
        if (frequency[i] > 0) {
            nodes[count] = createHuffmanNode(i, frequency[i]);
            count++;
        }
    }

    qsort(nodes, count, sizeof(HuffmanNode *), compare);

    while (count > 1) {
        HuffmanNode *left = nodes[0];
        HuffmanNode *right = nodes[1];

        HuffmanNode *parent = createHuffmanNode('\0', left->weight + right->weight);
        parent->left = left;
        parent->right = right;

        for (int i = 2; i < count; i++) {
            nodes[i - 2] = nodes[i];
        }

        nodes[count - 2] = parent;
        count--;

        qsort(nodes, count, sizeof(HuffmanNode *), compare);
    }
    return nodes[0];
}
```

3. 哈夫曼编码生成算法

从根结点开始，递归地遍历哈夫曼树，向左走分配代码 '0'，向右走分配代码 '1'，为每个字符生成编码。

```
// 生成哈夫曼编码
void generateHuffmanCodes(HuffmanTree root, char code[], int depth) {
    if (root == NULL) {
        return;
    }

    if (root->left == NULL && root->right == NULL) {
        code[depth] = '\0';
        strcpy(huffmanCodes[root->data], code);
```

```
            return;
        }

        code[depth] = '0';
        generateHuffmanCodes(root->left, code, depth + 1);

        code[depth] = '1';
        generateHuffmanCodes(root->right, code, depth + 1);
}
```

4. 文件压缩算法

再次遍历输入文件，根据生成的哈夫曼编码，将每个字符替换为对应的编码，将编码结果写入输出文件。

```
// 文件压缩
void compressFile(const char *inputFile, const char *outputFile) {
        FILE *inFile = fopen(inputFile, "rb");
        FILE *outFile = fopen(outputFile, "wb");

        if (inFile == NULL || outFile == NULL) {
            printf(" 无法打开文件 \n");
            return;
        }

        int ch;
        while ((ch = fgetc(inFile)) != EOF) {
            fputs(huffmanCodes[ch], outFile);
        }

        fclose(inFile);
        fclose(outFile);
}
```

5. 文件解压缩算法

从哈夫曼树的根结点开始，遍历压缩文件，根据编码规则，匹配到叶子结点时输出对应的字符，从而恢复原始文件内容。

```
// 文件解压缩
void decompressFile(const char *compressedFile, const char *outputFile, HuffmanTree root) {
        FILE *inFile = fopen(compressedFile, "rb");
        FILE *outFile = fopen(outputFile, "wb");

        if (inFile == NULL || outFile == NULL) {
```

```
            printf(“ 无法打开文件 \n”);
            return;
        }

        HuffmanTree current = root;

        int ch;
        while ((ch = fgetc(inFile)) != EOF) {
            if (ch == ‘0’ ) {
                current = current–>left;
            } else {
                current = current–>right;
            }

            if (current–>left == NULL && current–>right == NULL) {
                fputc(current–>data, outFile);
                current = root;
            }
        }

        fclose(inFile);
        fclose(outFile);
}
```

6. 测试函数

```
void testHuffmanCompression() {
    const char *inputFile = “test.txt”;
    const char *compressedFile = “compressed.txt”;
    const char *decompressedFile = “decompressed.txt”;

    countFrequency(inputFile);

    HuffmanTree root = buildHuffmanTree();

    char code[100];
    generateHuffmanCodes(root, code, 0);

    compressFile(inputFile, compressedFile);

    decompressFile(compressedFile, decompressedFile, root);

    // 对比原始文件和解压缩后的文件内容
    FILE *originalFile = fopen(inputFile, “rb”);
    FILE *decompressed = fopen(decompressedFile, “rb”);
```

```
    int ch1, ch2;
    int match = 1;

    while ((ch1 = fgetc(originalFile)) != EOF && (ch2 = fgetc(decompressed)) != EOF) {
        if (ch1 != ch2) {
            match = 0;
            break;
        }
    }

    if (match) {
        printf(" 文件压缩和解压缩测试成功 \n");
    } else {
        printf(" 文件压缩和解压缩测试失败 \n");
    }

    fclose(originalFile);
    fclose(decompressed);
}
```

本章小结

本章主要介绍了树相关的数据结构，涵盖树和二叉树的定义、存储结构、遍历方式，以及树、森林与二叉树的转换，哈夫曼树的相关知识等内容。

（1）树的基础：明确树是非线性结构，由根结点和若干子树构成，介绍了结点的度、叶子结点等关键术语，阐述其根的唯一性、子树的独立性和递归性等逻辑特征。

（2）二叉树：作为特殊树，每个结点最多有两个子树。介绍其定义、五种形态、性质，以及顺序存储和链式存储结构的优缺点与适用场景。

（3）二叉树遍历：包括前序、中序、后序和层次遍历，详述遍历顺序并以 C 语言代码实现，用于获取结点信息。

（4）线索二叉树：为解决传统二叉树遍历和空间问题引入，介绍其概念、建立方法及遍历算法。

（5）树、森林与二叉树的转换：阐述树、森林与二叉树间的转换关系，以及树和森林的遍历方式。

（6）哈夫曼树：是最优二叉树，带权路径长度最小。介绍其定义、构造步骤，以 C 语言代码实现构造和带权路径长度计算，在哈夫曼编码中有重要应用。

（7）实践应用：通过文件压缩项目，将哈夫曼树理论应用于实际，实现文件压缩与解压缩。

精选练习

一、选择题

1. 设数据结构 D–S 可以用二元组表示为 D–S=(D，S)，r ∈ S，其中：
D={A，B，C，D}，
r={<A，B>，<A，C>，<B，D>}，则数据结构 D–S 是（　）。

A. 线性结构　　B. 树形结构
C. 图形结构　　D. 集合

2. 树最适合用来表示（　）。

A. 有序数据元素　　B. 无序数据元素
C. 元素之间具有分支层次关系的数据　　D. 元素之间无联系的数据

3. 有关二叉树下列说法正确的是（　）。

A. 二叉树是度为 2 的有序树
B. 二叉树中结点的度可以小于 2
C. 二叉树中至少有一个结点的度为 2
D. 二叉树中任何一个结点的度都为 2

4. 深度为 10 的完全二叉树，第 3 层上的的结点数是（　）。

A . 15　　B. 16
C. 4　　D. 32

5. 设一棵树的度为 4，其中度为 1、2、3、4 的结点个数分别为 6、3、2、1，则这棵树中叶子结点的个数为（　）。

A .8　　B. 9
C. 10　　D. 11

6. 对于二叉树来说，第 i 层上最多包含的结点个数为（　）。

A. 2i – 1　　B. 2i + 1
C. 2i　　D. 2i

7. 一棵度为 4 的树，其结点个数至少为（　）。

A. 4　　B. 5
C. 6　　D. 7

8. 设森林 F 中有三棵树，第一，第二，第三棵树的结点个数分别为 M1，M2 和 M3。与森林 F 对应的二叉树根结点的右子树上的结点个数是（　）。

A. M1　　B. M1+M2
C. M3　　D. M2+M3

二、填空题

1. 二叉树具有 10 个度为 2 的结点，5 个度为 1 的结点，则度为 0 的结点个数是 _____。

2. 包含 n 个结点的二叉树，高度最大为 _____，高度最小为 _____。

3. 某完全二叉树共有 200 个结点，则该二叉树中有 _____ 个度为 1 的结点。

4. 一棵高度为 10 的满二叉树中的结点总数为 _____ 个，其中叶子结点数为 _____。

5. 某完全二叉树结点按层顺序编号（根结点的编号是 1），若 21 号结点有左孩子结点，则它的左孩子结点的编号为 _____。

6. 深度为 k 的完全二叉树至少有 _____ 个结点，至多有 _____ 个结点。

7. n 个结点的线索二叉树上含有 _____ 条线索。

三、简答题

1. 画出包含三个结点的树和二叉树的所有不同形态。

2. 找出满足下面条件的二叉树：

（1）前序遍历与中序遍历结果相同。

（2）前序遍历和后序遍历结果相同。

（3）中序遍历和后序遍历结果相同。

3. 一棵二叉树的中序、后序遍历序列分别为：G L D H B E I A C J F K 和 L G H D I E B J K F C A，请回答：

（1）画出二叉树逻辑结构的图示。

（2）画出此二叉树的二叉链表存储结构的图示并给出 C 语言描述。

（3）画出上题中二叉树的中序线索二叉树。

（4）画出中序线索二叉链表存储结构图示并给出 C 语言描述。

4 设有森林如图 1 所示，请回答：

（1）画出与该森林对应的二叉树的逻辑结构图示。

（2）写出该二叉树的前序、中序、后序遍历序列。

（3）画出该二叉树的中序线索二叉链表的图示并给出 C 语言描述。

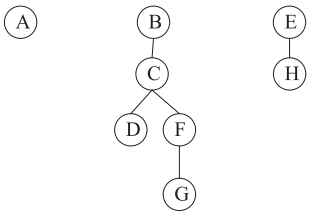


图 1　森林

5. 设有森林 B=(D,S),

D={A,B,C,D,E,F,G,H,I,J}, r ∈ S

r={<A,B>,<A,C>,<A,D>,<B,E>,<C,F>,<G,H>,<G,I>,<I,J>} 请回答：

（1）画出与森林对应的二叉树的逻辑结构图示。

（2）写出此二叉树的前序、中序、后序遍历序列。

（3）画出此二叉树的二叉链表存储结构的图示并给出 C 语言描述。

6. 请画出图 2 中的各二叉树对应的森林。

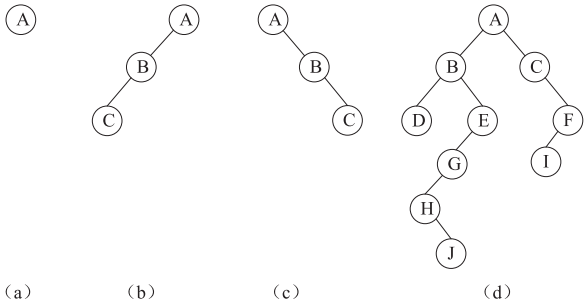


图 2　二叉树

7. 假设用于通信的电文由字符集 {a，b，c，d，e，f，g，h} 中的字母构成，这 8 个字母在电文中出现的概率分别为 {0.07，0.19，0.02，0.06，0.32，0.03，0.21，0.10}，试为这 8 个字母进行哈夫曼编码。请回答：

（1）画出哈夫曼树（按根点权值左小右大的原则）。

（2）写出依此哈夫曼树对各个字母的哈夫曼编码。

（3）求出此哈夫曼树的带权路径长度 WPL。

四、完善程序题

设计一个算法，其功能为：利用中序线索求结点的中序后继。请将代码补充完整。

```
typedef char DataType;
typedef struct Node
{   DataType data;
    struct Node *lchild, _______;
    int ltag,rtag;
}BiThrTree;
BiThrTree * InOrderNext(BiThrTree *p)          /* 求中序后继 */
{   if( _______) p=p->rchild;                  /* 若结点 *p 无右孩子 */
    else{                                      /* 若结点 *p 有右孩子 */
        _______;
      while(p->ltag==0)        _______;
   }
   return _______;
}
```