南京邮电大学

实 验 报 告

（2024 / 2025 学年 第 一 学期）

课程名称	数据结构
实验名称	树运算
实验时间	2024 年 10 月 30 日
指导单位	计算机学院
指导教师	孙海安

学生姓名	于明宏	班级学号	B23041011
学院(系)	计算机学院	专　　业	信息安全

实 验 报 告

实验名称	树运算			指导教师	孙海安
实验类型	设计	实验学时	2	实验时间	2024.10.30

一、 实验目的和要求

1. 掌握二叉树的二叉链表存储表示及遍历操作实现方法。

2. 实现二叉树遍历运算的应用：求二叉树中叶结点个数、结点总数、二叉树的高度，交换二叉树的左右子树。

3. 掌握二叉树的应用——哈夫曼编码的实现。

二、实验环境(实验设备)

硬件：微型计算机

软件：Windows 11 Professional Edition 23H2、Microsoft Visual C++ 2022

三、实验原理及内容

题目 1：

1、算法实现

```
#include <stdio.h>        // Include standard input/output library
#include <stdlib.h>       // Include standard library for memory allocation
#define ElemType char // Define ElemType as char

typedef struct btnode {    // Define binary tree node structure
    ElemType element;        // Element of the node
    struct btnode* lChild; // Pointer to the left child
    struct btnode* rChild; // Pointer to the right child
} BTNode;
```

```
void PreOrderTransverse(BTNode* t);    // Function prototype for pre-order traversal
void InOrderTransverse(BTNode* t);     // Function prototype for in-order traversal
void PostOrderTransverse(BTNode* t); // Function prototype for post-order traversal

void PreOrderTransverse(BTNode* t) {
    if (t == NULL) {    // If the node is null
        return;             // Return from the function
    }
    printf("%c ", t->element); // Print the element of the node
    PreOrderTransverse(t->lChild); // Recursively traverse the left child
    PreOrderTransverse(t->rChild); // Recursively traverse the right child
}

void InOrderTransverse(BTNode* t) {
    if (t == NULL) {    // If the node is null
        return;             // Return from the function
    }
    InOrderTransverse(t->lChild); // Recursively traverse the left child
    printf("%c ", t->element); // Print the element of the node
    InOrderTransverse(t->rChild); // Recursively traverse the right child
}

void PostOrderTransverse(BTNode* t) {
    if (t == NULL) {    // If the node is null
        return;             // Return from the function
    }
    PostOrderTransverse(t->lChild); // Recursively traverse the left child
    PostOrderTransverse(t->rChild); // Recursively traverse the right child
    printf("%c ", t->element); // Print the element of the node
}

BTNode* PreCreateBt(BTNode* t) {
    char c; // Declare a character variable
    c = getchar(); // Read a character from input
    if (c == '#') { // If the character is '#'
```

```
            t = NULL; // Set the node to NULL
        } else {
            t = (BTNode*)malloc(sizeof(BTNode)); // Allocate memory for the node
            t->element = c; // Set the element of the node
            t->lChild = PreCreateBt(t->lChild); // Recursively create the left child
            t->rChild = PreCreateBt(t->rChild); // Recursively create the right child
        }
        return t; // Return the created node
    }

    int main() {
        BTNode* t = NULL; // Declare a binary tree node pointer and initialize it to NULL
        printf("Enter the pre-order traversal of the binary tree (use # for null nodes):\n"); // Prompt for input
        t = PreCreateBt(t); // Create the binary tree from input

        printf("\nPre-order traversal result:\n"); // Print pre-order traversal header
        PreOrderTransverse(t); // Call pre-order traversal function

        printf("\n\nIn-order traversal result:\n"); // Print in-order traversal header
        InOrderTransverse(t); // Call in-order traversal function

        printf("\n\nPost-order traversal result:\n"); // Print post-order traversal header
        PostOrderTransverse(t); // Call post-order traversal function

        printf("\n"); // Print a newline
        return 0; // Return from the main function
    }
```

2、复杂度分析

（1）PreCreateBt 函数

时间复杂度：O(n)

该函数通过先序遍历构建二叉树，输入的字符数量为 n，每个字符都需要进行处理，因此时间复杂度为 O(n)。

（2）PreOrderTransverse 函数

时间复杂度：O(n)

该函数遍历整个二叉树，并访问每个节点一次，因此时间复杂度为 O(n)，其中 n 为树中节点的数量。

（3）InOrderTransverse 函数

时间复杂度：O(n)

该函数也遍历整个二叉树，访问每个节点一次，因此时间复杂度为 O(n)。

（4）PostOrderTransverse 函数

时间复杂度：O(n)

该函数同样遍历整个二叉树，访问每个节点一次，因此时间复杂度为 O(n)。

（5）main 函数

时间复杂度：O(n)

在 main 函数中，调用了 PreCreateBt 函数和三个遍历函数，总的时间复杂度为 O(n)，因为所有操作都与节点数量成线性关系。

（6）内存释放操作

时间复杂度：O(n)

如果实现了二叉树的销毁操作，需要遍历所有节点并释放内存，因此时间复杂度为 O(n)。

3、实验结果与结论

Enter the pre-order traversal of the binary tree (use # for null nodes):

123##4##5#6##

Pre-order traversal result:

1 2 3 4 5 6

In-order traversal result:

3 2 4 1 5 6

Post-order traversal result:

3 4 2 6 5 1

题目 2：

1、算法实现

```
#include <stdio.h>       // Include standard input/output library
#include <stdlib.h>      // Include standard library for memory allocation
#define ElemType char // Define ElemType as char

typedef struct btnode {    // Define binary tree node structure
    ElemType element;       // Element of the node
```

```
    struct btnode* lChild; // Pointer to the left child
    struct btnode* rChild; // Pointer to the right child
} BTNode;

void PreOrderTransverse(BTNode* t); // Function prototype for pre-order traversal
void InOrderTransverse(BTNode* t);  // Function prototype for in-order traversal
void PostOrderTransverse(BTNode* t); // Function prototype for post-order traversal
int CountNodes(BTNode* t);          // Function prototype to count nodes
int CountLeafNodes(BTNode* t);      // Function prototype to count leaf nodes
int CalculateHeight(BTNode* t);     // Function prototype to calculate height

void PreOrderTransverse(BTNode* t) { // Pre-order traversal function
    if (t == NULL) {                 // Check if the current node is NULL
        return;                      // Return if it is NULL
    }
    printf("%c ", t->element);       // Print the current node's element
    PreOrderTransverse(t->lChild);   // Traverse the left child
    PreOrderTransverse(t->rChild);   // Traverse the right child
}

void InOrderTransverse(BTNode* t) { // In-order traversal function
    if (t == NULL) {                 // Check if the current node is NULL
        return;                      // Return if it is NULL
    }
    InOrderTransverse(t->lChild);    // Traverse the left child
    printf("%c ", t->element);       // Print the current node's element
    InOrderTransverse(t->rChild);    // Traverse the right child
}

void PostOrderTransverse(BTNode* t) { // Post-order traversal function
    if (t == NULL) {                 // Check if the current node is NULL
        return;                      // Return if it is NULL
    }
    PostOrderTransverse(t->lChild);  // Traverse the left child
    PostOrderTransverse(t->rChild);  // Traverse the right child
```

```
        printf("%c ", t->element);              // Print the current node's element
    }

    BTNode* PreCreateBt(BTNode* t) {            // Function to create binary tree from pre-order input
        char c;                                 // Character to hold input
        c = getchar();                          // Read a character from input
        if (c == '#') {                         // Check for the null node indicator
            t = NULL;                           // Set the node to NULL
        }
        else {
            t = (BTNode*)malloc(sizeof(BTNode)); // Allocate memory for a new node
            t->element = c;                     // Assign the character to the node's element
            t->lChild = PreCreateBt(t->lChild); // Recursively create the left subtree
            t->rChild = PreCreateBt(t->rChild); // Recursively create the right subtree
        }
        return t;                               // Return the created tree node
    }

    int CountNodes(BTNode* t) {                 // Function to count total nodes in the tree
        if (t == NULL) {                        // Check if the current node is NULL
            return 0;                           // Return 0 if it is NULL
        }
        return 1 + CountNodes(t->lChild) + CountNodes(t->rChild); // Count the current node and children
    }

    int CountLeafNodes(BTNode* t) {             // Function to count leaf nodes in the tree
        if (t == NULL) {                        // Check if the current node is NULL
            return 0;                           // Return 0 if it is NULL
        }
        if (t->lChild == NULL && t->rChild == NULL) { // Check if the node is a leaf
            return 1;                           // Return 1 for a leaf node
        }
        return CountLeafNodes(t->lChild) + CountLeafNodes(t->rChild); // Count leaf nodes in children
```

```
    }

    int CalculateHeight(BTNode* t) {          // Function to calculate the height of the tree
        if (t == NULL) {                      // Check if the current node is NULL
            return 0;                         // Return 0 for NULL node
        }
        int leftHeight = CalculateHeight(t->lChild); // Calculate left subtree height
        int rightHeight = CalculateHeight(t->rChild); // Calculate right subtree height
        return (leftHeight > rightHeight ? leftHeight : rightHeight) + 1; // Return max height + 1
    }

    int main() {                              // Main function
        BTNode* t = NULL;                     // Declare the root of the tree
        printf("Enter the pre-order traversal of the binary tree (use # for null nodes):\n"); // Prompt for
input
        t = PreCreateBt(t);                   // Create the binary tree from input

        int nodeCount = CountNodes(t);        // Count total nodes
        int leafCount = CountLeafNodes(t); // Count leaf nodes
        int height = CalculateHeight(t);      // Calculate tree height

        printf("\nNumber of nodes in the binary tree: %d", nodeCount); // Print total nodes
        printf("\nNumber of leaf nodes in the binary tree: %d", leafCount); // Print leaf nodes
        printf("\nHeight of the binary tree: %d\n", height); // Print height

        printf("\nPre-order traversal result:\n"); // Print pre-order result
        PreOrderTransverse(t);                // Perform pre-order traversal

        printf("\n\nIn-order traversal result:\n"); // Print in-order result
        InOrderTransverse(t);                 // Perform in-order traversal

        printf("\n\nPost-order traversal result:\n"); // Print post-order result
        PostOrderTransverse(t);               // Perform post-order traversal

        return 0;                             // Return success
```

}

2、复杂度分析

（1）PreCreateBt 函数

时间复杂度：O(n)

该函数通过先序遍历构建二叉树，每次调用都会处理一个节点。在最坏情况下，需要处理 n 个节点，因此时间复杂度为 O(n)。

（2）PreOrderTransverse 函数

时间复杂度：O(n)

该函数通过先序遍历访问每个节点并输出其值。每个节点都被访问一次，因此时间复杂度为 O(n)。

（3）InOrderTransverse 函数

时间复杂度：O(n)

该函数通过中序遍历访问每个节点并输出其值。每个节点都被访问一次，因此时间复杂度为 O(n)。

（4）PostOrderTransverse 函数

时间复杂度：O(n)

该函数通过后序遍历访问每个节点并输出其值。每个节点都被访问一次，因此时间复杂度为 O(n)。

（5）CountNodes 函数

时间复杂度：O(n)

该函数递归计算二叉树中节点的个数。每个节点都被访问一次，因此时间复杂度为 O(n)。

（6）CountLeafNodes 函数

时间复杂度：O(n)

该函数递归计算二叉树中叶子节点的个数。每个节点都被访问一次，因此时间复杂度为 O(n)。

（7）CalculateHeight 函数

时间复杂度：O(n)

该函数递归计算二叉树的高度。每个节点都被访问一次，因此时间复杂度为 O(n)。

（8）main 函数

时间复杂度：O(n)

在主函数中，所有操作最终都涉及遍历和处理 n 个节点，包括构建树、计算节点数、计算叶子节点数、计算高度及遍历输出结果，因此整体时间复杂度为 O(n)。

3、实验结果与结论

Enter the pre-order traversal of the binary tree (use # for null nodes):

123##4##5#6##

Number of nodes in the binary tree: 6

Number of leaf nodes in the binary tree: 3

Height of the binary tree: 3

Pre-order traversal result:

1 2 3 4 5 6

In-order traversal result:

3 2 4 1 5 6

Post-order traversal result:

3 4 2 6 5 1

题目 3：

1、算法实现

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

// Huffman tree node structure definition
typedef struct hfmNode {
    int weight;                 // Weight of the node
    char character;             // Character associated with the node
    struct hfmNode* left;       // Pointer to the left child
    struct hfmNode* right;      // Pointer to the right child
} HfmNode;

// Create a Huffman tree node
HfmNode* CreateHfmNode(int weight, char character) {
    HfmNode* node = (HfmNode*)malloc(sizeof(HfmNode)); // Allocate memory for the node
    node->weight = weight;              // Set the weight
    node->character = character;        // Set the character
    node->left = NULL;                  // Initialize left child to NULL
    node->right = NULL;                 // Initialize right child to NULL
    return node;                        // Return the newly created node
}
```

```
// Define comparison function for Huffman tree nodes (for priority queue)
int Compare(const void* a, const void* b) {
    return ((HfmNode*)a)->weight - ((HfmNode*)b)->weight; // Compare weights of two nodes
}

// Create Huffman tree
HfmNode* CreateHuffmanTree(HfmNode** nodes, int n) {
    while (n > 1) { // Continue until there is only one node left
        // Sort the nodes based on weight
        qsort(nodes, n, sizeof(HfmNode*), Compare);

        // Take the two nodes with the smallest weights
        HfmNode* left = nodes[0];      // First node
        HfmNode* right = nodes[1];    // Second node

        // Create a new parent node with weight as the sum of the two nodes
        HfmNode* parent = CreateHfmNode(left->weight + right->weight, '\0');
        parent->left = left;              // Set left child to the first node
        parent->right = right;            // Set right child to the second node

        // Replace the original nodes with the new parent node
        nodes[0] = parent;                // Place parent node at the front
        for (int i = 1; i < n - 1; i++) {
            nodes[i] = nodes[i + 1]; // Shift remaining nodes left
        }
        n--; // Decrease the number of nodes
    }

    return nodes[0]; // Return the root of the Huffman tree
}

// Print Huffman codes
void PrintHuffmanCodes(HfmNode* root, char* code, int length) {
    if (!root) return; // Base case: if the node is NULL, return
```

```
    // If it's a leaf node, print the code
    if (!root->left && !root->right) {
        code[length] = '\0'; // End the string
        printf("Character %c: %s\n", root->character, code); // Print character and its code
    }

    // Traverse left subtree
    code[length] = '0'; // Append '0' for left traversal
    PrintHuffmanCodes(root->left, code, length + 1);

    // Traverse right subtree
    code[length] = '1'; // Append '1' for right traversal
    PrintHuffmanCodes(root->right, code, length + 1);
}

// Decode Huffman codes
void DecodeHuffmanCodes(HfmNode* root, const char* encoded) {
    HfmNode* current = root; // Start at the root of the Huffman tree
    for (int i = 0; encoded[i] != '\0'; i++) { // Loop through the encoded string
        if (encoded[i] == '0') {
            current = current->left; // Move to left child for '0'
        }
        else {
            current = current->right; // Move to right child for '1'
        }

        // If it's a leaf node, output the character
        if (!current->left && !current->right) {
            printf("%c", current->character); // Print the character
            current = root; // Go back to the root for the next character
        }
    }
    printf("\n"); // Print newline after decoding
}
```

```
// Main function
int main() {
    int n;
    printf("Enter the number of characters: "); // Prompt user for number of characters
    scanf_s("%d", &n); // Read the number of characters

    char* characters = (char*)malloc(n * sizeof(char)); // Allocate memory for characters
    int* weights = (int*)malloc(n * sizeof(int)); // Allocate memory for weights

    // Input characters
    printf("Enter the characters: "); // Prompt user for characters
    scanf_s("%s", characters, n + 1); // Read characters (+1 to prevent overflow)

    // Input weights
    printf("Enter the weights for each character: "); // Prompt user for weights
    for (int i = 0; i < n; i++) {
        scanf_s("%d", &weights[i]); // Read weights
    }

    // Create Huffman tree
    HfmNode** nodes = (HfmNode**)malloc(n * sizeof(HfmNode*)); // Allocate memory for
nodes
    for (int i = 0; i < n; i++) {
        nodes[i] = CreateHfmNode(weights[i], characters[i]); // Create each node
    }
    HfmNode* root = CreateHuffmanTree(nodes, n); // Create the Huffman tree

    // Print Huffman codes
    char code[100]; // Array to store codes
    printf("\nHuffman Codes:\n"); // Print header
    PrintHuffmanCodes(root, code, 0); // Print the Huffman codes

    // Input encoded string and decode
    char encoded[100]; // Array for encoded input
```

```
    printf("\nEnter the binary string to decode: "); // Prompt for encoded string
    scanf_s("%s", encoded, sizeof(encoded)); // Read the encoded string
    printf("Decoded result: "); // Print header for decoded result
    DecodeHuffmanCodes(root, encoded); // Decode the encoded string


    return 0; // Exit program
}
```

2、复杂度分析

（1）CreateHfmNode 函数

时间复杂度：O(1)

该函数分配内存并初始化一个 Huffman 树节点，所有操作都是常数时间内完成的，因此时间复杂度为 O(1)。

（2）Compare 函数

时间复杂度：O(1)

该函数比较两个 Huffman 树节点的权重，所有操作都是常数时间内完成的，因此时间复杂度为 O(1)。

（3）CreateHuffmanTree 函数

时间复杂度：O(n log n)

该函数在创建 Huffman 树的过程中，每次都需要对节点数组进行排序。排序操作使用 qsort 函数，其平均时间复杂度为 O(n log n)。由于该函数在每次迭代中都减少一个节点，因此最多需要执行 n-1 次循环。最终的时间复杂度为 O(n log n)。

（4）PrintHuffmanCodes 函数

时间复杂度：O(n)

该函数通过递归访问每个节点并打印相应的 Huffman 码。每个叶子节点都会被访问一次，因此时间复杂度为 O(n)，其中 n 是树中的节点数。

（5）DecodeHuffmanCodes 函数

时间复杂度：O(m)

该函数解码给定的二进制字符串。假设编码字符串的长度为 m，函数会遍历字符串的每一位，查找 Huffman 树的相应节点。时间复杂度为 O(m)。

（6）main 函数

时间复杂度：O(n)

在主函数中，创建 Huffman 树的操作和打印 Huffman 码的操作都是 O(n)。用户输入的字符和权重的处理也是 O(n)。因此，主函数的整体时间复杂度为 O(n)。在 main 函数中，用户输入字符和权值的过程涉及 n 次输入操作，因此时间复杂度为 O(n)。同时，调用的编码和解码操作也分别为 O(n) 和 O(m)，整体时间复杂度由最慢的部分决定，即 O(n + m)。

3、实验结果与结论

Enter the number of characters: 6

Enter the characters: ABCDEF

Enter the weights for each character: 9 11 13 3 5 12

Huffman Codes:

Character C: 000

Character D: 001

Character E: 01

Character F: 10

Character A: 110

Character B: 111

Enter the binary string to decode: 1101010

Decoded result: AFF

四、实验小结（包括问题和解决方法、心得体会、意见与建议等）

（一）实验中遇到的主要问题及解决方法

1.问题：在构建二叉树时，输入格式不正确导致程序崩溃。

解决方法：在输入之前添加输入格式的提示，并在读取输入时增加对字符的有效性检查，确保格式正确后再进行树的构建。

2.问题：遍历二叉树时，输出结果与预期不符。

解决方法：通过逐步调试每个遍历函数，发现部分情况下访问顺序错误。对遍历函数进行了逐行检查，确保按照正确的顺序访问每个节点。

3.问题：在计算树的高度时，程序未能正确处理空树的情况。

解决方法：在计算树高的函数中增加对空树的处理逻辑，确保在树为空时返回高度为 0。

4.问题：哈夫曼编码的生成和查找速度较慢。

解决方法：通过优化哈夫曼树的生成逻辑，减少不必要的排序操作，提升编码的生成和查找效率。

（二）实验心得

通过此次实验，我对树运算的基本操作有了更深入的理解。在解决具体问题的过程中，我意识到细节的重要性，尤其是在指针操作时的边界处理。此外，优化代码逻辑以提高性能也是我在本次实验中重要的收获。这些经验将对我今后的编程实践大有裨益。

（三）意见与建议（没有可省略）

可以提供更多的时间上机操作，以确保更多程序设计思路得以实现，提升面向对象语言的掌握程度和编程能力。

五、支撑毕业要求指标点

《数据结构》课程支撑毕业要求的指标点为：

1.2-M 掌握计算机软硬件相关工程基础知识，能将其用于分析计算机及应用领域的相关工程问题。

3.2-H 能够根据用户需求，选取适当的研究方法和技术手段，确定复杂工程问题的解决方案。

4.2-H 能够根据实验方案，配置实验环境、开展实验，使用定性或定量分析方法进行数据分析与处理，综合实验结果以获得合理有效的结论。

实验内容	支撑点 1.2	支撑点 3.2	支撑点 4.2
线性表及多项式的运算	√		
二叉树的基本操作及哈夫曼编码译码系统的实现		√	√
图的基本运算及智能交通中的最佳路径选择问题		√	√
各种内排序算法的实现及性能比较	√		√

六、指导教师评语 (含学生能力达成度的评价)

如评分细则所示

成　绩	XX	批阅人	XX	日　期	XXX

	评分项	优秀	良好	中等	合格	不合格
评	遵守实验室规章制度					
	学习态度					
分	算法思想准备情况					
	程序设计能力					
	解决问题能力					
	课题功能实现情况					
	算法设计合理性					
	算法效能评价					
细	回答问题准确度					
	报告书写认真程度					
	内容详实程度					
	文字表达熟练程度					
则	其它评价意见					
	本次实验能力达成评价（总成绩）					