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	树结构
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	树结构			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 int        // Define ElemType as int for the tree node data type

// Define the structure for a binary tree node
```

```
typedef struct btnode {
    ElemType element;                // Value of the node
    struct btnode* lChild;       // Pointer to the left child
    struct btnode* rChild;       // Pointer to the right child
} BTNode;

// Function declarations
int GetNodeNum(BTNode* t);       // Get the number of nodes in the binary tree
int GetLeafNum(BTNode* t);       // Get the number of leaf nodes in the binary tree
int GetTreeHeight(BTNode* t); // Get the height of the binary tree
void SwapSubTree(BTNode* t);    // Swap all left and right subtrees in the binary tree

// Custom max function to find the maximum of two integers
int max(int a, int b) {
    return (a > b) ? a : b; // Return the greater of a and b
}

// Pre-order traversal to create a binary tree from input
BTNode* PreCreateBt(BTNode* t) {
    char ch;                         // Variable to store the input character
    ch = getchar();              // Read a character from standard input
    if (ch == '#') {             // If the input is '#', create an empty node
        t = NULL;                    // Set the current node to NULL
    } else {
        t = (BTNode*)malloc(sizeof(BTNode)); // Allocate memory for a new node
        t->element = ch;      // Set the node's value to the input character
        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 node
}

// Pre-order traversal of the binary tree
void PreorderTransverse(BTNode* t) {
    if (t == NULL) { // If the node is NULL, return
```

```
            return;
        }
        printf("%c", t->element); // Print the node's value
        PreorderTransverse(t->lChild); // Traverse the left subtree
        PreorderTransverse(t->rChild); // Traverse the right subtree
    }

    // In-order traversal of the binary tree
    void MediumorderTransverse(BTNode* t) {
        if (t == NULL) { // If the node is NULL, return
            return;
        }
        MediumorderTransverse(t->lChild); // Traverse the left subtree
        printf("%c", t->element); // Print the node's value
        MediumorderTransverse(t->rChild); // Traverse the right subtree
    }

    // Post-order traversal of the binary tree
    void PostorderTransverse(BTNode* t) {
        if (t == NULL) { // If the node is NULL, return
            return;
        }
        PostorderTransverse(t->lChild); // Traverse the left subtree
        PostorderTransverse(t->rChild); // Traverse the right subtree
        printf("%c", t->element); // Print the node's value
    }

    // Get the number of nodes in the binary tree
    int GetNodeNum(BTNode* t) {
        if (t == NULL) return 0; // If the node is NULL, return 0
        return GetNodeNum(t->lChild) + GetNodeNum(t->rChild) + 1; // Count the nodes
    }

    // Get the number of leaf nodes in the binary tree
    int GetLeafNum(BTNode* t) {
```

```
    if (t == NULL) return 0; // If the node is NULL, return 0
    if ((t->lChild == NULL) && (t->rChild == NULL)) return 1; // If it's a leaf node, return 1
    return GetLeafNum(t->lChild) + GetLeafNum(t->rChild); // Count leaf nodes recursively
}

// Get the height of the binary tree
int GetTreeHeight(BTNode* t) {
    if (t == NULL) return 0; // If the node is NULL, height is 0
    return 1 + max(GetTreeHeight(t->lChild), GetTreeHeight(t->rChild)); // Calculate height
}

// Swap all left and right subtrees in the binary tree
void SwapSubTree(BTNode* t) {
    if (t) { // If the node is not NULL
        BTNode* temp = t->lChild; // Store the left child
        t->lChild = t->rChild; // Swap left child with right child
        t->rChild = temp; // Assign the stored left child to the right
        SwapSubTree(t->lChild); // Recursively swap left subtree
        SwapSubTree(t->rChild); // Recursively swap right subtree
    }
}

int main() {
    BTNode* t = NULL; // Initialize the root of the tree to NULL

    // Prompt user to enter the pre-order traversal of the binary tree
    printf("Please enter the pre-order traversal of the binary tree:\n");
    t = PreCreateBt(t); // Create the binary tree from input

    // Output the number of nodes, leaf nodes, and height of the tree
    printf("Number of nodes in the binary tree: %d\n", GetNodeNum(t));
    printf("Number of leaf nodes in the binary tree: %d\n", GetLeafNum(t));
    printf("Height of the binary tree: %d\n", GetTreeHeight(t));

    SwapSubTree(t); // Swap all left and right subtrees
```

```
        printf("\nAfter swapping all left and right subtrees:\n\n");
        printf("Pre-order traversal:\n");
        PreorderTransverse(t); // Print pre-order traversal
        printf("\n\nIn-order traversal:\n");
        MediumorderTransverse(t); // Print in-order traversal
        printf("\n\nPost-order traversal:\n");
        PostorderTransverse(t); // Print post-order traversal
        printf("\n");
        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）MediumorderTransverse 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）GetNodeNum 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）GetLeafNum 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）GetTreeHeight 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）SwapSubTree 函数

时间复杂度：O(n)

该函数递归交换二叉树中所有子树。每个节点都被访问一次，因此时间复杂度为 O(n)。

（9）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，因此整体时间复杂度为 O(n)。

3、实验结果与结论

Please enter the pre-order traversal of the binary tree:

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

After swapping all left and right subtrees:

Pre-order traversal:

156243

In-order traversal:

651423

Post-order traversal:

654321

题目 3：

1、算法实现

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

#define MAX_TREE_HEIGHT 100
#define MAX_CHARACTERS 256

// Huffman tree node structure
typedef struct HfmNode {
```

```
    char character;                 // Character
    int weight;                     // Weight
    struct HfmNode* left;           // Left subtree
    struct HfmNode* right;          // Right subtree
} HfmNode;

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

// Compare the weights of two nodes
int compare(const void* a, const void* b) {
    return ((HfmNode*)a)->weight - ((HfmNode*)b)->weight; // Return difference in weights
}

// Create the Huffman tree
HfmNode* createHuffmanTree(char characters[], int weights[], int size) {
    HfmNode** nodes = (HfmNode**)malloc(size * sizeof(HfmNode*)); // Allocate memory for nodes
    int i;

    // Create initial nodes
    for (i = 0; i < size; i++) {
        nodes[i] = createNode(characters[i], weights[i]); // Create nodes for each character
    }

    // Build the Huffman tree
    while (size > 1) {
        // Sort nodes
```

```
            qsort(nodes, size, sizeof(HfmNode*), compare); // Sort nodes by weight

            // Merge the two smallest nodes
            HfmNode* left = nodes[0];                          // Get the smallest node
            HfmNode* right = nodes[1];                         // Get the second smallest node
            HfmNode* parent = createNode('\0', left->weight + right->weight); // Create a new parent node
            parent->left = left;                               // Set left child of parent
            parent->right = right;                             // Set right child of parent

            nodes[1] = parent;                                 // Replace the two smallest nodes with the new parent
            size--;                                            // Decrease the number of nodes
        }

        HfmNode* root = nodes[0];                              // The last node is the root
        free(nodes);                                           // Free the array of nodes
        return root;                                           // Return the root of the tree
    }

    // Generate codes
    void createCode(HfmNode* node, char* code, int depth, char codes[][MAX_TREE_HEIGHT]) {
        if (node == NULL) return;                              // Base case for recursion

        // Leaf node
        if (node->left == NULL && node->right == NULL) {
            code[depth] = '\0';                                // End the string
            strcpy(codes[node->character], code);              // Save the code for the character
            return;                                            // Return from function
        }

        // Recursively generate codes
        code[depth] = '0';                                     // Add '0' for left edge
        createCode(node->left, code, depth + 1, codes); // Traverse left subtree
        code[depth] = '1';                                     // Add '1' for right edge
```

```
        createCode(node->right, code, depth + 1, codes); // Traverse right subtree
    }

    // Encoding
    void encode(HfmNode* root) {
        char codes[MAX_CHARACTERS][MAX_TREE_HEIGHT] = {0}; // Store codes
        char code[MAX_TREE_HEIGHT];                        // Temporary code storage
        createCode(root, code, 0, codes);                  // Generate codes

        printf("Enter a character to get its encoding: ");  // Prompt for character
        char inputChar;                                     // Variable to store input character
        scanf(" %c", &inputChar);                           // Read the character

        // Check if the character has a corresponding encoding
        if (inputChar >= 0 && inputChar < MAX_CHARACTERS && codes[inputChar][0] != '\0') {
            printf("Encoding for '%c': %s\n", inputChar, codes[inputChar]); // Print the encoding
        } else {
            printf("Character not found in encoding.\n");   // Handle case where character is not found
        }
    }

    // Decoding
    void decode(HfmNode* root) {
        char encoded[256];                                  // Array to store the encoded string
        printf("Enter the encoded string: ");               // Prompt for encoded string
        scanf("%s", encoded);                               // Read the encoded string

        HfmNode* current = root;                            // Start from the root
        for (int i = 0; encoded[i] != '\0'; i++) {
            current = (encoded[i] == '0') ? current->left : current->right; // Traverse the tree
            if (current->left == NULL && current->right == NULL) { // If it's a leaf node
                printf("%c", current->character);           // Print the character
                current = root;                             // Return to root node
```

```
            }
        }
        printf("\n");                                   // New line after decoding
    }

    // Free the Huffman tree memory
    void freeHuffmanTree(HfmNode* root) {
        if (root) {
            freeHuffmanTree(root->left);                // Free left subtree
            freeHuffmanTree(root->right);               // Free right subtree
            free(root);                                 // Free current node
        }
    }

    // Main function
    int main() {
        printf("Enter the number of characters to encode: "); // Prompt for character count
        int size;                                               // Variable to store character
count
        scanf("%d", &size);                                     // Read character count

        printf("Enter the characters to encode: ");         // Prompt for characters
        char charArr[MAX_CHARACTERS];                           // Array to store characters
        for (int i = 0; i < size; i++) {
            scanf(" %c", &charArr[i]);                      // Read characters (skip whitespace)
        }

        printf("Enter the corresponding weights for the characters: "); // Prompt for weights
        int weightArr[MAX_CHARACTERS];                          // Array to store weights
        for (int i = 0; i < size; i++) {
            scanf("%d", &weightArr[i]);                     // Read weights
        }

        HfmNode* hfmTree = createHuffmanTree(charArr, weightArr, size); // Create the Huffman tree
```

```
        while (1) {
            printf("\nChoose an option:\n");                    // Display options
            printf("1. Encode\n");                              // Option to encode
            printf("2. Decode\n");                              // Option to decode
            printf("3. Exit\n");                                // Option to exit
            int choice;                                         // Variable to store user's choice
            scanf("%d", &choice);                               // Read user's choice

            switch (choice) {
                case 1:
                    encode(hfmTree);                            // Call encode function
                    break;
                case 2:
                    decode(hfmTree);                            // Call decode function
                    break;
                case 3:
                    freeHuffmanTree(hfmTree);                   // Free memory before exiting
                    return 0;                                   // Exit the program
                default:
                    printf("Invalid choice, please try again.\n"); // Invalid option message
                    break;
            }
        }
    }
```

2、复杂度分析

（1）createNode 函数

时间复杂度：O(1)

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

（2）compare 函数

时间复杂度：O(1)

该函数仅比较两个节点的权值，操作简单且时间固定，因此时间复杂度为 O(1)。

（3）createHuffmanTree 函数

时间复杂度：O(n log n)

在构建 Huffman 树的过程中，需要对节点进行排序，使用快速排序的时间复杂度为 O(n log n)。由于需要重复进行 n-1 次合并操作，因此整体时间复杂度为 O(n log n)。

（4）createCode 函数

时间复杂度：O(n)

该函数通过递归遍历 Huffman 树来生成编码，最坏情况下访问每个节点一次，因此时间复杂度为 O(n)。

（5）encode 函数

时间复杂度：O(n)

该函数生成编码后，查找对应字符的编码，最坏情况下遍历所有可能的字符，因此时间复杂度为 O(n)。

（6）decode 函数

时间复杂度：O(m)

该函数遍历输入的编码字符串并根据 Huffman 树解码，假设编码字符串长度为 m，每个字符需访问树中的节点，因此时间复杂度为 O(m)。

（7）freeHuffmanTree 函数

时间复杂度：O(n)

该函数递归地释放 Huffman 树的所有节点，最坏情况下需要访问每个节点一次，因此时间复杂度为 O(n)。

（8）main 函数

时间复杂度：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 to encode: 6

Enter the characters to encode: ABCDEF

Enter the corresponding weights for the characters: 9 11 13 3 5 12

Choose an option:

1. Encode

2. Decode

3. Exit

1

Enter a character to get its encoding: D

Encoding for 'D': 001

Choose an option:

1. Encode

2. Decode

3. Exit

2

Enter the encoded string: 0010101

DEE

Choose an option:

1. Encode

2. Decode

3. Exit

3

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

评分细则	评分项	优秀	良好	中等	合格	不合格
评分细则	遵守实验室规章制度					
	学习态度					
	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					
	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					
	其它评价意见					
	本次实验能力达成评价（总成绩）					