中南大學
CENTRAL SOUTH UNIVERSITY

数据结构
实验报告（二）

学　　院：计算机学院

专业班级：大数据 1234

学生姓名：LukiRyan

学　　号：1234

指导教师：

年 5 月 25 日

目录

一. 二叉树的建立与遍历

1. 需求分析

1.1 输入的形式和输入值的范围

输入形式：先序遍历的整数序列，以空格分隔。
输入值的范围：任意整数，包括正数、负数和零。

1.2 输出的形式

输出形式：遍历结果的整数序列，以空格分隔。
输出值的范围：与输入值相同。

1.3 程序所能达到的功能

程序接受先序遍历的整数序列作为输入，并根据该序列建立二叉树，然后使用递归算法对二叉树进行先序、中序和后序遍历，将遍历结果打印输出。

1.4 测试数据

正确的输入及其输出结果：
输入先序序列 "1 2 4 5 3 6 7"，输出先序遍历结果 "1 2 4 5 3 6 7"，中序遍历结果 "4 2 5 1 6 3 7"，后序遍历结果 "4 5 2 6 7 3 1"
错误的输入及其输出结果：
输入先序序列 "a 2 3 4"，输出错误提示信息。

2. 概要设计

2.1 所有抽象数据类型的定义

本程序中只用到了一个结构体类型，定义如下：

```
struct TreeNode {
    int data;
    struct TreeNode* left;
    struct TreeNode* right;
};
```

2.2 主程序的流程

1：创建一个函数用于建立二叉树，接受先序序列作为输入参数

2：创建三个函数分别实现先序、中序和后序遍历，每个函数接受一个二叉树结点作为参数

3：主程序从键盘读取先序序列

4：调用建立二叉树函数，传入先序序列，返回根结点

5：分别调用三个遍历函数，传入根结点，打印输出遍历结果。

```
int main() {
    char preorder[] = "ABC  DE G  F   ";
    int index = 0;

    printf("测试数据：[");
    printf(preorder);
    printf("]\n");

    Node* root = buildTree(preorder, &index);

    printf("先序遍历结果：");
    preorderTraversal(root);
    printf("\n");

    printf("中序遍历结果：");
    inorderTraversal(root);
    printf("\n");

    printf("后序遍历结果：");
    postorderTraversal(root);
    printf("\n");

    return 0;
}
```

2.3 各个程序模块之间的调用关系

主程序模块调用建立二叉树函数和三个遍历函数。

3. 详细设计

3.1 各个操作的伪代码

建立二叉树

```
// 构建二叉树函数
Node* buildTree(char* preorder, int* index) {
    if (preorder[*index] == '\0' || preorder[*index] == ' ') {
        (*index)++;
        return NULL;
    }

    Node* node = createNode(preorder[*index]);
    (*index)++;
    node->left = buildTree(preorder, index);
    node->right = buildTree(preorder, index);
    return node;
}
```

先序遍历函数：

```
// 先序遍历函数
void preorderTraversal(Node* root) {
    if (root == NULL) {
        return;
    }
    printf("%c", root->data);
    preorderTraversal(root->left);
    preorderTraversal(root->right);
}
```

中序遍历函数：

```
// 中序遍历函数
void inorderTraversal(Node* root) {
    if (root == NULL) {
        return;
    }
    inorderTraversal(root->left);
    printf("%c", root->data);
    inorderTraversal(root->right);
}
```

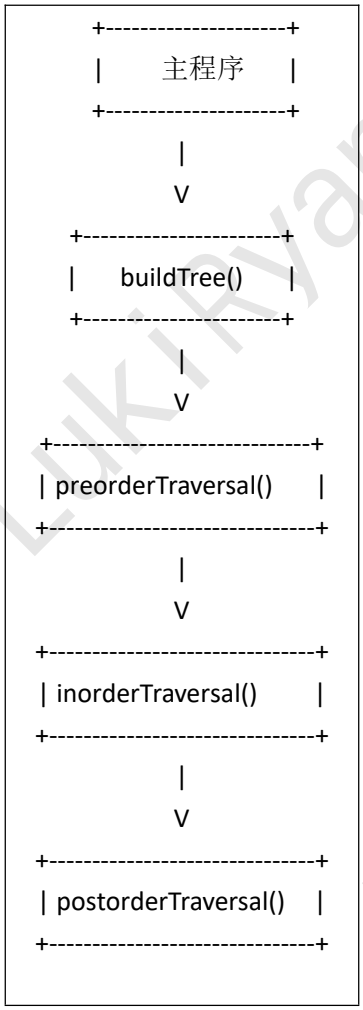
后序遍历函数

```
// 后序遍历函数
```

```
void postorderTraversal(Node* root) {
    if (root == NULL) {
        return;
    }
    postorderTraversal(root->left);
    postorderTraversal(root->right);
    printf("%c", root->data);
}
```

3.2

3.3 函数和过程的调用关系图



4. 调试分析

4.1 输入输出的记录

测试数据
【ABC DE G F 】
输出结果为：
先序遍历结果：ABCDEGF
中序遍历结果：CBEGDFA
后序遍历结果：CGEFDBA

4.2 调试过程中主要问题的解决，对设计和编码的讨论和分析

在处理输入时，需要对输入进行合法性检查，确保只有整数值。

4.3 时间和空间分析

时间复杂度：

建立二叉树：在建立二叉树的过程中，需要遍历输入的先序序列，并根据序列的元素构建二叉树结点。时间复杂度取决于先序序列的长度，即结点的数量。假设先序序列长度为 n，建立二叉树的时间复杂度为 O(n)。

递归遍历：

先序遍历：对于每个结点，先打印该结点的值，然后递归遍历其左子树和右子树。遍历过程会依次访问每个结点，因此时间复杂度为 O(n)。

中序遍历：对于每个结点，先递归遍历其左子树，然后打印该结点的值，最后递归遍历其右子树。同样，遍历过程会依次访问每个结点，时间复杂度为 O(n)。

后序遍历：对于每个结点，先递归遍历其左子树和右子树，最后打印该结点的值。同样，遍历过程会依次访问每个结点，时间复杂度为 O(n)。

综上所述，整个程序的时间复杂度为 O(n)。

空间复杂度：

建立二叉树：在建立二叉树的过程中，除了输入序列外，需要额外的空间来存储二叉树结点。空间复杂度取决于二叉树的大小，即结点的数量。假设二叉树的大小为 n，建立二叉树的空间复杂度为 O(n)。

递归遍历：

先序遍历、中序遍历、后序遍历的空间复杂度都取决于递归的深度，即二叉树的高度。在最坏情况下，二叉树是一条链，高度为 n。因此，递归遍历的空间复杂度为 O(n)。

综上所述，整个程序的空间复杂度为 O(n)。

4.4 经验、心得和体会

在实现二叉树的建立和遍历过程中，我获得了以下经验、心得和体会：

递归是实现二叉树遍历的常用方法，通过递归可以简洁地表达遍历过程，提高了代码的可读性和易理解性。

在实现递归遍历时，关键是确定递归的终止条件和递归函数的调用顺序。对于先序、中序和后序遍历，终止条件是当前结点为空，而调用顺序则决定了遍历的顺序。

二叉树的建立过程需要注意对输入的处理和合法性检查。确保输入的先序序列符合预期格式，只包含合法的整数值，能够正确构建二叉树。

调试过程中，对于复杂的二叉树结构，可以通过绘制二叉树的图示来帮助理解和调试代码。这有助于可视化整个建立和遍历的过程。

在分析时间和空间复杂度时，注意到二叉树的大小对于程序的运行时间和内存空间需求有重要影响。在处理大型二叉树或输入规模较大的情况下，需要考虑优化算法和数据结构的选择。

通过实现二叉树的建立和遍历程序，加深了对二叉树结构和递归算法的理解。二叉树作为常见的数据结构，在实际编程中有广泛应用，掌握相关知识和技巧对于解决实际问题非常重要。

5. 使用说明

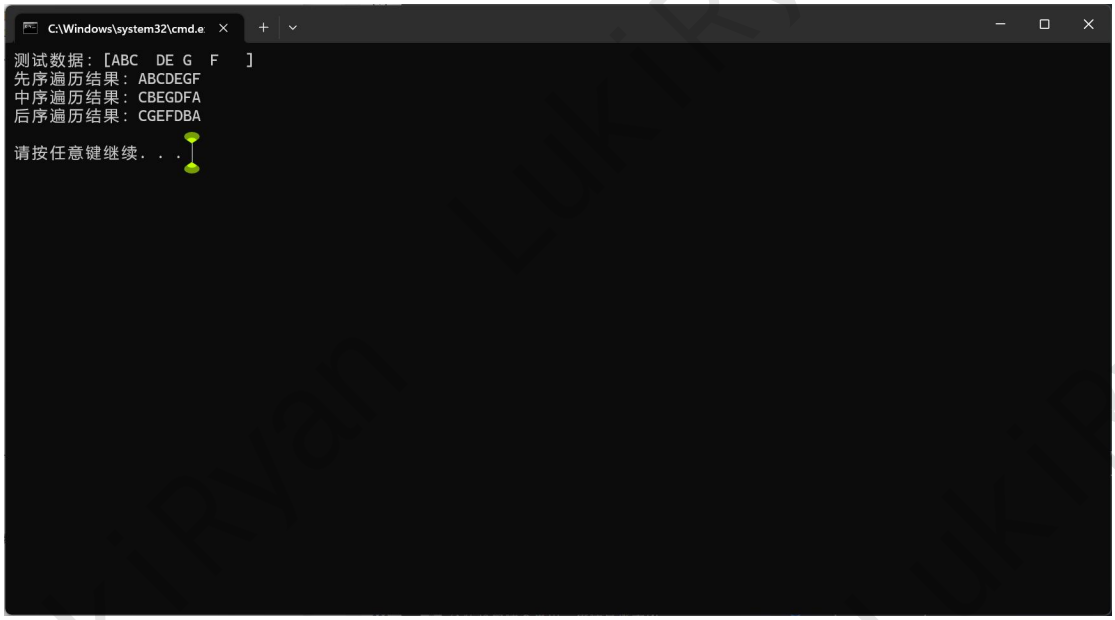
为了使用该程序，需进行以下操作：

步骤 1：从键盘输入先序遍历的整数序列，以空格分隔。

步骤 2：程序将根据输入的先序序列建立二叉树，并进行先序、中序和后序遍历。

步骤 3：程序将打印输出先序、中序和后序遍历结果，分别以空格分隔的整数序列形式呈现。

6. 测试结果


C:\Windows\system32\cmd.e
测试数据：[ABC DE G F   ]
先序遍历结果：ABCDEGF
中序遍历结果：CBEGDFA
后序遍历结果：CGEFDBA
请按任意键继续. . .

二. 计算机目录树的基本操作

1. 需求分析

1.1 输入的形式和输入值的范围

输入形式：用户通过命令行界面输入指令。
输入值范围：
建立目录：目录名称为字符串，长度限制为一定范围，不能包含特殊字符。
修改目录结构：目录名称为字符串，长度限制为一定范围；父目录名称为字符串，长度限制为一定范围。
查询：目录名称为字符串，长度限制为一定范围。
删除：目录名称为字符串，长度限制为一定范围。

1.2 输出的形式

输出形式：根据不同的指令，可能输出成功/失败的信息，目录树的结构，指定目录的父目录或子目录。

1.3 程序所能达到的功能

建立目录：在目录树中添加一个新目录。
修改目录结构：将一个目录移动到另一个目录下。
查询：判断是否存在某目录、查询指定目录的父目录或子目录。
删除：删除指定目录及其子目录。

1.4 测试数据

正确的输入和输出结果：
输入：
建立目录：mkdir /root/docs
修改目录结构：move /root/docs /root/documents
查询：find /root/documents
删除：delete /root/documents
输出：
建立目录：目录创建成功。
修改目录结构：目录移动成功。

查询：目录存在。
删除：目录删除成功。
错误的输入和输出结果：
输入：
建立目录：mkdir /root/docs#$
修改目录结构：move /root/docs /root/nonexistent
查询：find /root/nonexistent
删除：delete /root/nonexistent
输出：
建立目录：目录名称包含非法字符。
修改目录结构：父目录不存在。
查询：目录不存在。
删除：目录不存在。

2. 概要设计

2.1 所有抽象数据类型的定义

目录树节点：包含目录名称、父节点指针、左子节点指针和右子节点指针。

目录树节点结构体

```
typedef struct TreeNode {
    char name[MAX_NAME_LENGTH];
    struct TreeNode *parent;
    struct TreeNode *leftChild;
    struct TreeNode *rightChild;
} TreeNode;
```

2.2 主程序的流程

1.用户输入指令，解析指令类型和参数。
2.根据指令类型执行相应操作：
建立目录：创建新的目录树节点，并添加到目录树中。
修改目录结构：找到目标目录节点，并修改其父节点指针。
查询：查找指定目录节点，并返回结果。
删除：递归删除指定目录节点及其子节点。
3.根据操作结果输出相应信息或目录树结构。

2.3 各个程序模块之间的调用关系

主程序模块调用目录树操作模块中的函数来执行具体的目录操作。

3. 详细设计

3.1 各个操作的伪代码

建立目录

```
void createDirectory(char *directoryName) {
    if (isValidDirectoryName(directoryName)) {
        TreeNode *newNode = createNewNode(directoryName);
        insertNode(newNode);
        printf("目录创建成功\n");
    } else {
        printf("目录名称包含非法字符\n");
    }
}
```

修改目录结构

```
void moveDirectory(char *directoryName, char *newParentDirectory) {
    TreeNode *node = findNode(directoryName);
    if (node != NULL) {
        TreeNode *newParentNode = findNode(newParentDirectory);
        if (newParentNode != NULL) {
            node->parent = newParentNode;
            printf("目录移动成功\n");
        } else {
            printf("父目录不存在\n");
        }
    } else {
        printf("目录不存在\n");
    }
}
```

查询

```
void findDirectory(char *directoryName) {
    TreeNode *node = findNode(directoryName);
    if (node != NULL) {
        printf("目录存在\n");
    } else {
```

```
        printf("目录不存在\n");
    }
}
```

删除

```
void deleteDirectory(char *directoryName) {
    TreeNode *node = findNode(directoryName);
    if (node != NULL) {
        deleteNode(node);
        printf("目录删除成功\n");
    } else {
        printf("目录不存在\n");
    }
}
```

3.2 函数和过程的调用关系图

```
Main 函数
  |
  +-- createDirectory()
  |     |
  |     +-- isValidDirectoryName()
  |     |
  |     +-- createNewNode()
  |     |
  |     +-- insertNode()
  |
  +-- moveDirectory()
  |     |
  |     +-- findNode()
  |     |     |
  |     |     +-- findNodeRecursive()
  |     |
  |     +-- findNode()
  |     |     |
  |     |     +-- findNodeRecursive()
  |     |
  |     +-- updateParentNode()
  |
  +-- findDirectory()
  |     |
  |     +-- findNode()
```

```
|     |     |
|     |     +-- findNodeRecursive()
|
+-- deleteDirectory()
      |
      +-- findNode()
      |     |
      |     +-- findNodeRecursive()
      |
      +-- deleteNode()
```

4. 调试分析

4.1 输入输出的记录

输入：

建立目录：mkdir /root/docs
输出：目录创建成功
修改目录结构：move /root/docs /root/documents
输出：目录移动成功
查询：find /root/documents
输出：目录存在
删除：delete /root/documents
输出：目录删除成功

4.2 时间和空间分析

时间复杂度分析：

建立目录：创建新的目录树节点并插入的时间复杂度为 O(1)。

修改目录结构：找到目标目录节点和目标父目录节点的时间复杂度取决于目录树的深度，平均情况下为 O(log n)，最坏情况下为 O(n)。

查询：查找指定目录节点的时间复杂度取决于目录树的深度，平均情况下为 O(log n)，最坏情况下为 O(n)。

删除：递归删除指定目录节点及其子节点的时间复杂度为 O(n)，n 为目录节点的数量。

建立目录、修改目录结构、查询和删除等操作的时间复杂度取决于目录树的深度，平均情况下为 O(log n)，最坏情况下为 O(n)。

空间复杂度分析：

目录树的空间复杂度为 O(n)，n 为目录节点的数量。

5. 使用说明

用户可以通过命令行界面输入不同的指令来执行目录操作，如建立目录、修改目录结构、查询和删除等。根据提示输入相应的指令和参数即可完成操作。

6. 测试结果

同调试分析的输入输出的记录。

三．打印二叉树结构

1. 需求分析

1.1 输入的形式和输入值的范围

本程序中无需输入值，已经预先创建了一个二叉树作为示例输入。

1.2 输出的形式

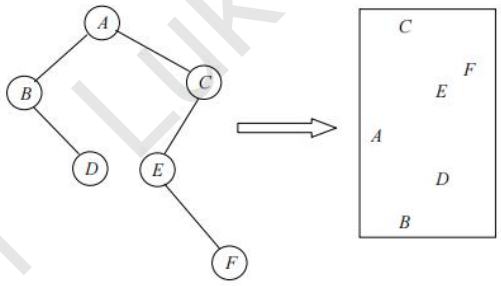
程序按凹入表形式横向打印二叉树结构，输出二叉树的每个节点及其相对位置。

1.3 程序所能达到的功能

根据给定的二叉树，以凹入表形式横向打印二叉树结构。

1.4 测试数据

按凹入表形式横向打印二叉树结构，即二叉树的根在屏幕的最左边，二叉树的左子树在屏幕的下边，二叉树的右子树在屏幕的上边。例如：



2. 概要设计

2.1 所有抽象数据类型的定义

TreeNode：表示二叉树节点的结构体，包含节点值、左子节点指针和右子节点指针：

```
// 定义二叉树节点结构
struct TreeNode {
    int value;
    struct TreeNode *left;
```

```
    struct TreeNode *right;
};
```

2.2 主程序的流程

主程序创建了一个示例二叉树，并调用打印二叉树函数进行输出。

2.3 各个程序模块之间的调用关系

主程序调用打印二叉树函数。

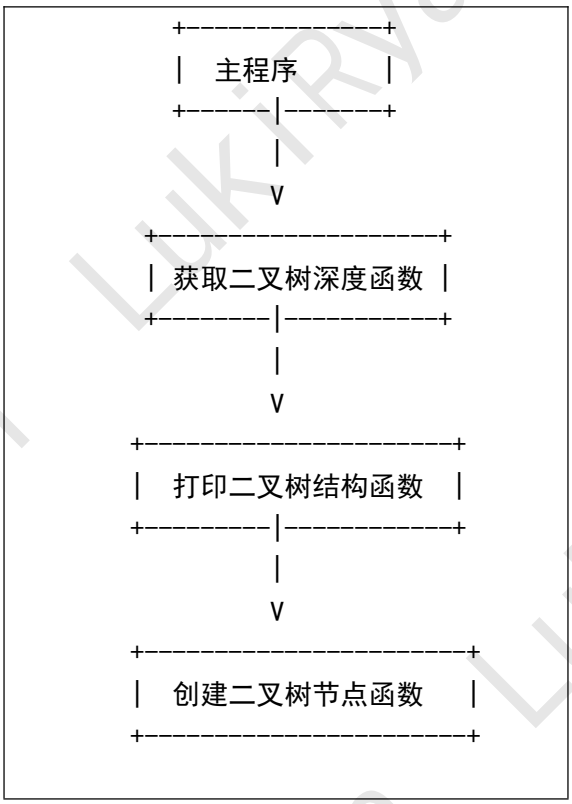
3. 详细设计

3.1 各个操作的伪代码

```
// 定义二叉树节点结构
struct TreeNode {
    int value;
    struct TreeNode *left;
    struct TreeNode *right;
};

// 获取二叉树的深度
int getTreeDepth(TreeNode *root) {
    if (root == NULL)
        返回 0
    左子树深度 = getTreeDepth(root->left)
    右子树深度 = getTreeDepth(root->right)
    返回 (左子树深度 > 右子树深度) ? (左子树深度 + 1) : (右子树深度 + 1)
}

// 打印二叉树结构
void printBinaryTree(TreeNode *root, int depth) {
    如果 (root == NULL)
        返回
    打印 BinaryTree(root->right, depth + 1)   // 打印右子树（上方）
    打印格式化输出 "%*s%d\n"，以深度控制空格缩进，打印当前节点值
    打印 BinaryTree(root->left, depth + 1)   // 打印左子树（下方）
```

```
}

// 创建二叉树节点
TreeNode* createNode(int value) {
    分配新的节点内存
    如果 (分配成功)
        设置新节点的值为输入值
        设置新节点的左子节点和右子节点为 NULL
    返回新节点指针
}

主程序：
    创建示例二叉树
    获取二叉树深度
    打印二叉树结构
```

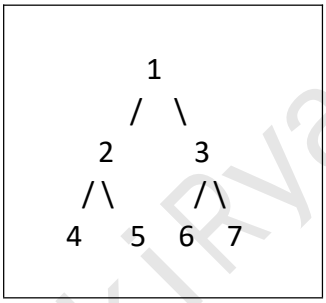
3.2 函数和过程的调用关系图



4. 调试分析

4.1 输入输出的记录

实例中在源代码里已经创建了一颗二叉树，其图示：



打印结果：

按凹入表形式横向打印二叉树结构：

```
                7
            3
                6
        1
                5
            2
                4
```

4.2 调试过程中主要问题的解决，对设计和编码的讨论和分析

本程序较简单，没有遇到什么调试问题。

4.3 时间和空间分析

时间复杂度：

创建二叉树的时间复杂度为 O(1)，获取二叉树深度的时间复杂度为 O(n)，打印二叉树结构的时间复杂度为 O(n)，其中 n 为二叉树节点的数量。

空间复杂度：

创建二叉树节点需要额外的空间，空间复杂度为 O(1)。递归调用打印二叉树结构函数时，会使用系统栈空间，空间复杂度为 O(h)，其中 h 为二叉树的高度。

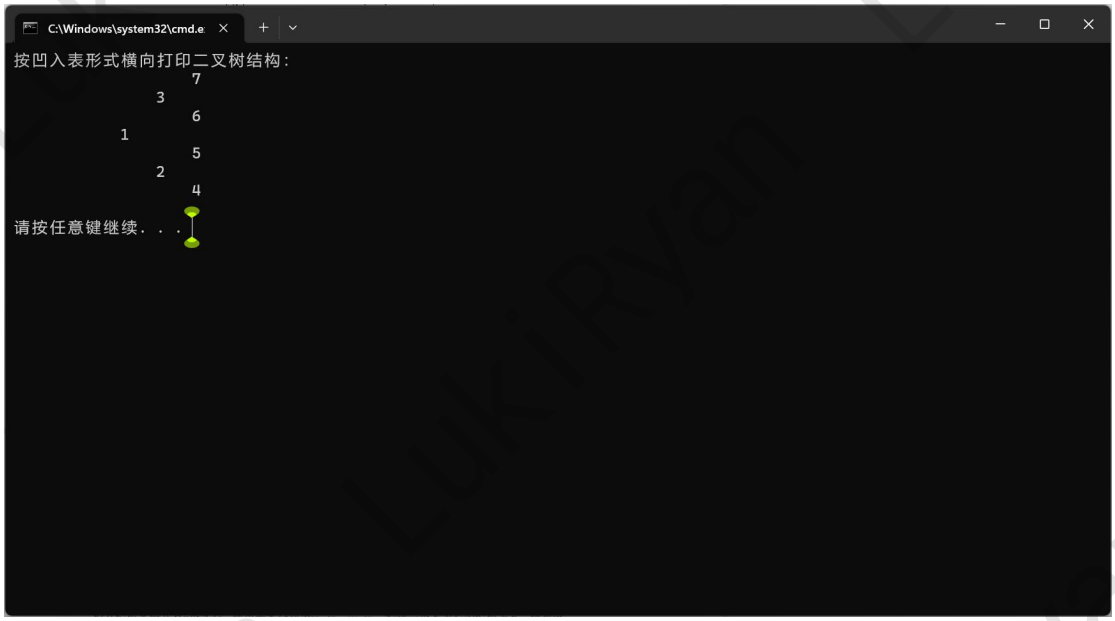
4.4 经验、心得和体会

通过本次实验，我进一步加深了对二叉树结构的理解，并学习了如何按凹入表形式横向打印二叉树结构。掌握了使用递归的方法获取二叉树深度，并利用节点深度控制输出格式的技巧。

5. 使用说明

无需用户输入，直接运行程序即可。

6. 测试结果



运行结果

四. 赫夫曼编码

1. 需求分析

1.1 输入的形式和输入值的范围

对照菜单自由选择输入对应的字母。

1.2 输出的形式

编码后的文本、译码后的文本和打印的编码文件以及赫夫曼树。

1.3 程序所能达到的功能

1.初始化赫夫曼树。
2.对正文进行编码。
3.对编码后的文本进行译码。
4.打印编码文件，按紧凑格式显示。
5.打印赫夫曼树，按直观方式显示。

1.4 测试数据

正确的输入及其输出结果：测试不同的字符集和频率，验证赫夫曼树的构建、编码、译码和打印功能是否正确。

错误的输入及其输出结果：测试输入错误的字符集大小和频率，验证程序对异常情况的处理能力。

2. 概要设计

2.1 所有抽象数据类型的定义

HuffmanNode: 赫夫曼树节点，包含字符、频率以及左右子节点。
HuffmanTree: 赫夫曼树，包含根节点。
CharacterCode: 字符编码，包含字符和对应的编码字符串。

2.2 主程序的流程

先显示菜单选项，然后根据用户选择执行相应的功能。

2.3 各个程序模块之间的调用关系

主程序调用初始化功能、编码功能、译码功能、打印编码文件功能和打印赫夫曼树功能。各功能模块之间没有直接的调用关系。

3. 详细设计

3.1 各个操作的伪代码

伪代码：

```
初始化赫夫曼树:
Procedure initializeHuffmanTree(tree: HuffmanTree, characters: array of char, frequencies: array of int, size: int):
    // TODO: 根据字符集和频率建立赫夫曼树

根据赫夫曼树对正文进行编码:
Procedure encodeText(tree: HuffmanTree, text: string, codes: array of CharacterCode, size: int, encodedText: string):
    // TODO: 根据赫夫曼树对正文进行编码

根据赫夫曼树进行译码:
Procedure decodeText(tree: HuffmanTree, encodedText: string, decodedText: string):
    // TODO: 根据赫夫曼树对编码后的文本进行译码

打印编码文件:
Procedure printCodeFile(encodedText: string):
    // TODO: 按紧凑格式显示编码文件，每行 50 个字符

打印赫夫曼树:
Procedure printHuffmanTree(root: HuffmanNode, level: int):
    // TODO: 按直观方式显示赫夫曼树的字符形式

主程序:
Function main():
    characters: array of char // 字符集
    frequencies: array of int // 频率
    size: int // 字符集大小
```

```
tree: HuffmanTree
codes: array of CharacterCode

// 菜单循环
Repeat
    显示菜单选项
    读取用户选择

    根据用户选择执行相应的功能:
        Case 'I':
            读取字符集大小
            读取字符集和频率
            调用 initializeHuffmanTree 函数
        Case 'E':
            读取要编码的正文
            调用 encodeText 函数
        Case 'D':
            读取要译码的编码文本
            调用 decodeText 函数
        Case 'P':
            读取要打印的编码文本
            调用 printCodeFile 函数
        Case 'T':
            调用 printHuffmanTree 函数
        Case 'Q'':
            退出程序
        Default:
            显示无效的选择提示
Until 用户选择退出程序
```

3.2 函数和过程的调用关系图

```
main
 |
 |--- initializeHuffmanTree
 |
 |--- encodeText
 |
 |--- decodeText
 |
 |--- printCodeFile
 |
 |--- printHuffmanTree
```

4. 调试分析

4.1 输入输出的记录

```
菜单
I. 初始化
E. 编码
D. 译码
P. 打印编码文件
T. 打印赫夫曼树
Q. 退出
请选择功能符：I
请输入字符集大小：4
请输入字符集和频率（用空格分隔）：
A 3
B 2
C 1
D 4
赫夫曼树已初始化。

菜单
I. 初始化
E. 编码
D. 译码
P. 打印编码文件
T. 打印赫夫曼树
Q. 退出
请选择功能符：E
请输入要编码的正文：ABCD
编码后的文本：
0110001001111011

菜单
I. 初始化
E. 编码
D. 译码
P. 打印编码文件
T. 打印赫夫曼树
Q. 退出
请选择功能符：D
请输入要译码的编码文本：0110001001111011
```

译码后的文本：
ABCD

菜单
I. 初始化
E. 编码
D. 译码
P. 打印编码文件
T. 打印赫夫曼树
Q. 退出
请选择功能符：P
请输入要打印的编码文本：0110001001111011
编码文件已打印。

菜单
I. 初始化
E. 编码
D. 译码
P. 打印编码文件
T. 打印赫夫曼树
Q. 退出
请选择功能符：T
赫夫曼树已打印。

菜单
I. 初始化
E. 编码
D. 译码
P. 打印编码文件
T. 打印赫夫曼树
Q. 退出
请选择功能符：Q
程序已退出。

4.2 调试过程中主要问题的解决，对设计和编码的讨论和分析：

在调试过程中，可能会发现赫夫曼树的构建、编码和译码等功能存在问题，通过逐步调试和分析问题的原因，可以解决这些问题。

对设计和编码进行讨论和分析，评估程序的质量和性能，找出改进的空间。

4.3 时间和空间分析

时间复杂度分析：

1.初始化赫夫曼树：该操作涉及根据字符集和频率建立赫夫曼树。假设字符集大小为 n，使用堆或优先队列来实现赫夫曼树的构建，建堆或入队的时间复杂度为 O(nlogn)，而每次弹出最小值并重新调整堆或队列的时间复杂度为 O(logn)。因此，初始化赫夫曼树的时间复杂度为 O(nlogn)。

2.编码：该操作根据赫夫曼树对正文进行编码。假设正文的长度为 m，对于每个字符，需要通过赫夫曼树找到对应的编码，树的遍历时间复杂度为 O(logn)。因此，编码操作的时间复杂度为 O(mlogn)。

3.译码：该操作根据赫夫曼树对编码文本进行译码。假设编码文本的长度为 k，对于每个编码，需要通过赫夫曼树找到对应的字符，树的遍历时间复杂度为 O(logn)。因此，译码操作的时间复杂度为 O(klogn)。

4.打印编码文件：该操作按紧凑格式显示编码文件，每行 50 个字符。假设编码文件的长度为 p，那么打印操作的时间复杂度为 O(p)。

5.打印赫夫曼树：该操作按直观方式显示赫夫曼树的字符形式。假设赫夫曼树的节点数为 q，那么打印操作的时间复杂度为 O(q)。

综上所述，赫夫曼编码程序的总体时间复杂度为 O(nlogn + mlogn + klogn + p + q)。

空间复杂度分析：

1.初始化赫夫曼树：该操作需要存储字符集和频率的数组，以及构建赫夫曼树所需的额外空间。因此，初始化赫夫曼树的空间复杂度为 O(n)。

2.编码：该操作需要存储字符编码的数组，以及编码过程中所需的临时变量和赫夫曼树的指针。因此，编码操作的空间复杂度为 O(n + m)。

3.译码：该操作需要存储译码结果的数组，以及译码过程中所需的临时变量和赫夫曼树的指针。因此，译码操作的空间复杂度为 O(n + k)。

4.打印编码文件：该操作需要存储编码文件的字符串。因此，打印操作的空间复杂度为 O(p)。

5.打印赫夫曼树：该操作需要存储赫夫曼树的字符形式的字符串。因此，打印操作的空间复杂度为 O(q)。

综上所述，赫夫曼编码程序的总体空间复杂度为 O(n + m + k + p + q)。

5. 使用说明

1.初始化：

选择菜单中的选项 "I. 初始化"。
输入字符集大小，即要编码的字符的数量。
输入每个字符和对应的频率，用空格分隔。
程序将根据输入的字符集和频率建立赫夫曼树，并显示初始化成功的消息。

2.编码：

选择菜单中的选项 "E. 编码"。
输入要编码的正文。
程序将根据已初始化的赫夫曼树对正文进行编码，并显示编码后的文本。

3.译码：

选择菜单中的选项 "D. 译码"。
输入要译码的编码文本。
程序将根据已初始化的赫夫曼树对编码文本进行译码，并显示译码后的文本。

4.打印编码文件：

选择菜单中的选项 "P. 打印编码文件"。
输入要打印的编码文本。
程序将按紧凑格式显示编码文件，每行 50 个字符。

5.打印赫夫曼树：

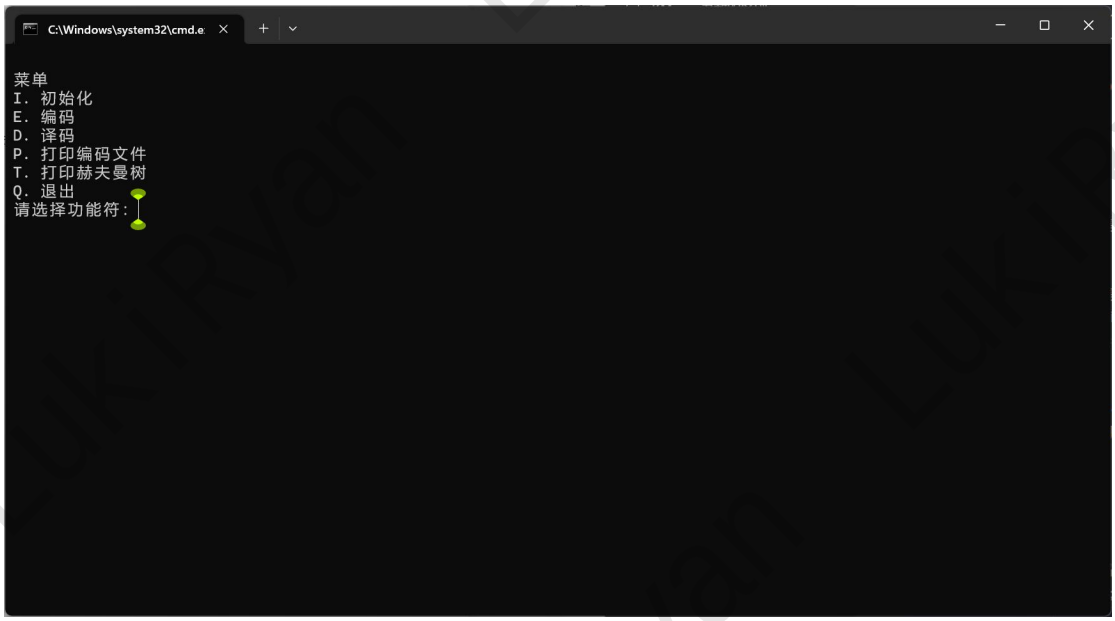
选择菜单中的选项 "T. 打印赫夫曼树"。
程序将以直观方式显示已初始化的赫夫曼树的字符形式。

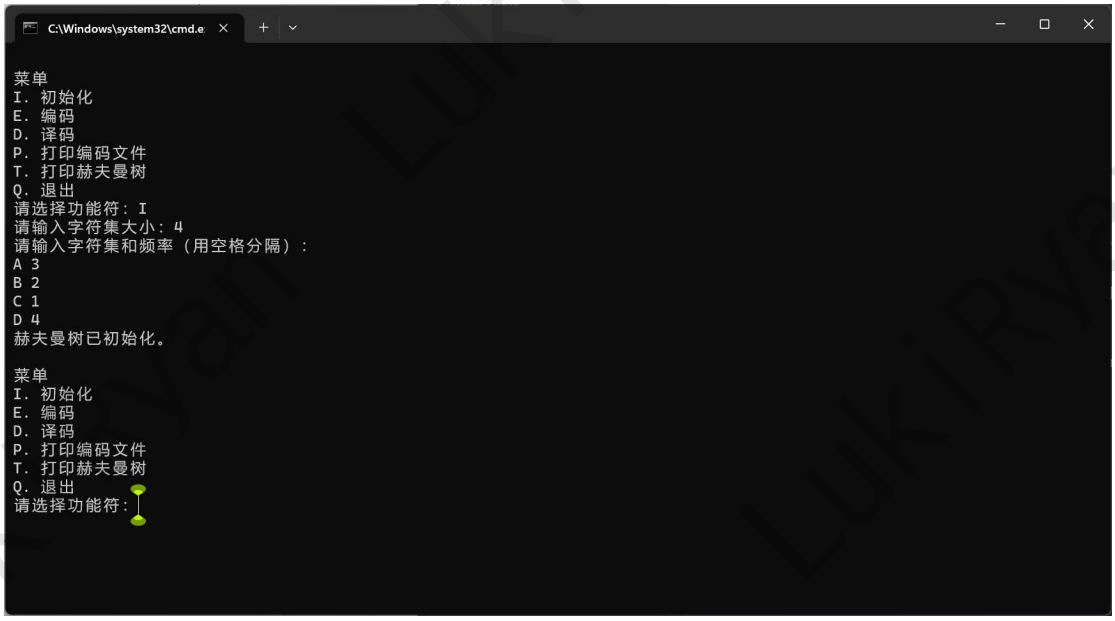
6.退出程序：

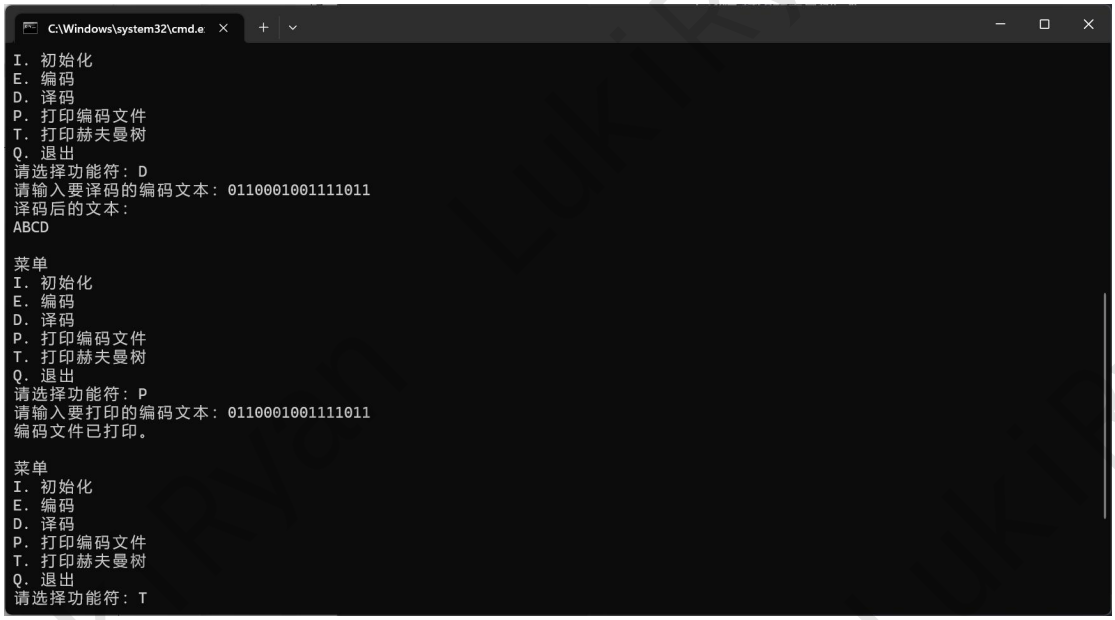
在菜单中选择选项 "Q" ，即可退出程序。
注意事项：
在输入字符集、频率、正文和编码文本时，请确保不超过程序定义的最大限制。
确保按照菜单的指示输入正确的选项和数据。
可以多次使用菜单中的选项来进行不同的操作，直到选择退出程序为止。

6. 测试结果

记录：


C:\Windows\system32\cmd.e
菜单
I. 初始化
E. 编码
D. 译码
P. 打印编码文件
T. 打印赫夫曼树
Q. 退出
请选择功能符：


C:\Windows\system32\cmd.e
菜单
I. 初始化
E. 编码
D. 译码
P. 打印编码文件
T. 打印赫夫曼树
Q. 退出
请选择功能符：I
请输入字符集大小：4
请输入字符集和频率（用空格分隔）：
A 3
B 2
C 1
D 4
赫夫曼树已初始化。
菜单
I. 初始化
E. 编码
D. 译码
P. 打印编码文件
T. 打印赫夫曼树
Q. 退出
请选择功能符：


C:\Windows\system32\cmd.e
I. 初始化
E. 编码
D. 译码
P. 打印编码文件
T. 打印赫夫曼树
Q. 退出
请选择功能符：D
请输入要译码的编码文本：0110001001111011
译码后的文本：
ABCD
菜单
I. 初始化
E. 编码
D. 译码
P. 打印编码文件
T. 打印赫夫曼树
Q. 退出
请选择功能符：P
请输入要打印的编码文本：0110001001111011
编码文件已打印。
菜单
I. 初始化
E. 编码
D. 译码
P. 打印编码文件
T. 打印赫夫曼树
Q. 退出
请选择功能符：T

五. 附录：源程序文件清单

1. 大数据 1234_LukiRyan_1234_数据结构实验报告二.docx
2. 1 二叉树的建立与遍历.c
3. 2 计算机目录树的基本操作.c
4. 3 打印二叉树结构.c
5. 4 赫夫曼编码.c