**实验二 树和二叉树的基本操作与实现**

目录

[**实验二 树和二叉树的基本操作与实现** 1](#_Toc183203566)

[实验内容1：先序序列创建二叉树，输出先序序列、中序序列、后序序列并输出叶子结点数 2](#_Toc183203567)

[1、 数据结构设计和核心算法（用图和文字描述清楚）： 3](#_Toc183203569)

[数据结构设计 3](#_Toc183203570)

[核心算法： 3](#_Toc183203571)

[2、 程序及注释 5](#_Toc183203572)

[3、 运行结果： 7](#_Toc183203574)

[4、 心得体会 7](#_Toc183203575)

[实验内容2：求根结点到x结点的路径 8](#_Toc183203576)

[1、 数据结构设计和核心算法（用图和文字描述清楚）： 8](#_Toc183203577)

[数据结构设计 8](#_Toc183203578)

[核心算法描述 8](#_Toc183203579)

[2、 程序及注释 10](#_Toc183203580)

[3、 运行结果： 12](#_Toc183203581)

[4、 心得体会 12](#_Toc183203582)

1. **实验目的**
2. 复习C语言中指针、结构体、子程序调用等基础知识。
3. 掌握树和二叉树的基本结构和操作方法，主要包括：树的建立与遍历实现方法，培养学生灵活使用数据结构解决实际问题的能力。
4. **实验设备**

微机

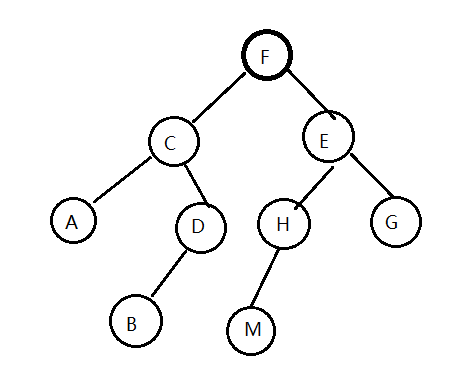
1. **预习要求**
2. 复习C语言中指针的用法，特别是结构体的指针的用法；
3. 理解线二叉树的基本概念以及相关操作的实现方法；主要包括：二叉树的建立、先序、中序、后序遍历等；
4. 掌握应用树解决实际问题的一般方法和步骤。
5. **实验内容（具体见pintia）**
6. 先序序列创建二叉树，输出先序序列、中序序列、后序序列并输出叶子结点数。
7. 求根结点到x结点的路径
8. **注意事项**
9. 提前准备实验的内容
10. 代码需要通过pintia测试
11. 将以上实验报告及程序代码**分别**在超星平台上提交。
    1. 实验报告：学号-姓名-实验2.docx
    2. 代码：学号-姓名-实验2.zip (包含exp1.c和exp2.c两个文件)

实验内容1：先序序列创建二叉树，输出先序序列、中序序列、后序序列并输出叶子结点数

文件名：实验二二叉树遍历.cpp

1. **问题描述：**

对于给定的二叉树，输出其先序序列、中序序列、后序序列并输出叶子结点数。



**输入格式:**

一棵二叉树的先序序列是一个字符串，若字符是‘#’,表示该二叉树是空树，否则该字符是相应结点的数据元素。

**输出格式:**

若是非空二叉树，则输出四行内容 第一行是二叉树的先序遍历序列； 第二行是二叉树的中序遍历序列； 第三行是二叉树的后序遍历序列； 第四行是叶子结点数；若是空二叉树 只需输出叶子数0

**输入样例1:**

FCA##DB###EHM###G##

**输出样例1:**

FCADBEHMG

ACBDFMHEG

ABDCMHGEF

4

**输入样例2:**

#

**输出样例2:**

0

1. 数据结构设计和核心算法（用图和文字描述清楚）：

数据结构设计

**树节点结构：**TreeNode

每个节点包含以下成员：

char data: 存储节点的数据值。

TreeNode\* left: 指向左子节点的指针。

TreeNode\* right: 指向右子节点的指针。

typedef struct TreeNode

{

char data;

struct TreeNode \*left, \*right;

} TreeNode;

核心算法：

**1.构建二叉树** TreeNode\* createTree(const char\* preorder)

从先序序列中递归读取节点值，逐层创建树。如果当前字符是 # 或结束符 \0，表示空节点，返回 NULL。否则，创建一个新节点，递归地构建其左子树和右子树。

int idx = 0;

TreeNode\* createTree(const char\* preorder)

{

if (preorder[idx] == '#' || preorder[idx] == '\0') {

idx++;

return NULL;

}

TreeNode\* node = (TreeNode\*)malloc(sizeof(TreeNode));

node->data = preorder[idx++];

node->left = createTree(preorder);

node->right = createTree(preorder);

return node;

}

2.**先序遍历 void** preorderTraversal**(**TreeNode**\*** root**)**

先递归访问根节点，然后递归访问左子树和右子树。最后输出顺序：根 → 左 → 右。

void preorderTraversal(TreeNode\* root)

{

if (root == NULL) return;

printf("%c", root->data);

preorderTraversal(root->left);

preorderTraversal(root->right);

}

3.中序遍历**void** inorderTraversal**(**TreeNode**\*** root**)**

先递归访问左子树，然后访问根节点，最后访问右子树。最后输出顺序：左 → 根 → 右。

void inorderTraversal(TreeNode\* root)

{

if (root == NULL) return;

inorderTraversal(root->left);

printf("%c", root->data);

inorderTraversal(root->right);

}

4.后序遍历 **void** postorderTraversal**(**TreeNode**\*** root**)**

先递归访问左子树和右子树，最后访问根节点。最后输出顺序：左 → 右 → 根。

void postorderTraversal(TreeNode\* root)

{

if (root == NULL) return;

postorderTraversal(root->left);

postorderTraversal(root->right);

printf("%c", root->data);

}

5.叶子节点计数 **int** countLeaves**(**TreeNode**\*** root**)**

如果当前节点为 NULL，返回 0；如果当前节点没有左右子树（left == NULL 且 right == NULL），返回 1；否则递归求左右子树的叶子节点数之和。

int countLeaves(TreeNode\* root)

{

if (root == NULL) return 0;

if (root->left == NULL && root->right == NULL) return 1;

return countLeaves(root->left) + countLeaves(root->right);

}

1. 程序及注释

程序名： 实验二二叉树遍历.cpp

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <stdlib.h>

typedef struct TreeNode

{

char data;

struct TreeNode \*left, \*right;

} TreeNode;

*// 构建二叉树*

int idx = 0; *// 输入的先序序列的当前位置*

TreeNode\* createTree(const char\* preorder)

{

if (preorder[idx] == '#' || preorder[idx] == '\0') {

idx++;

return NULL;

}

TreeNode\* node = (TreeNode\*)malloc(sizeof(TreeNode));

node->data = preorder[idx++];

node->left = createTree(preorder);

node->right = createTree(preorder);

return node;

}

*// 先序*

void preorderTraversal(TreeNode\* root)

{

if (root == NULL) return;

printf("%c", root->data);

preorderTraversal(root->left);

preorderTraversal(root->right);

}

*// 中序*

void inorderTraversal(TreeNode\* root)

{

if (root == NULL) return;

inorderTraversal(root->left);

printf("%c", root->data);

inorderTraversal(root->right);

}

*// 后序*

void postorderTraversal(TreeNode\* root)

{

if (root == NULL) return;

postorderTraversal(root->left);

postorderTraversal(root->right);

printf("%c", root->data);

}

*// 叶子节点数*

int countLeaves(TreeNode\* root)

{

if (root == NULL) return 0;

if (root->left == NULL && root->right == NULL) return 1;

return countLeaves(root->left) + countLeaves(root->right);

}

*// 释放二叉树*

void freeTree(TreeNode\* root)

{

if (root == NULL) return;

freeTree(root->left);

freeTree(root->right);

free(root);

}

int main()

{

char preorder[100];

scanf("%s", preorder);

if (preorder[0] == '#') {

printf("0\n");

return 0;

}

TreeNode\* root = createTree(preorder);

preorderTraversal(root);

printf("\n");

inorderTraversal(root);

printf("\n");

postorderTraversal(root);

printf("\n");

*// 叶子节点数*

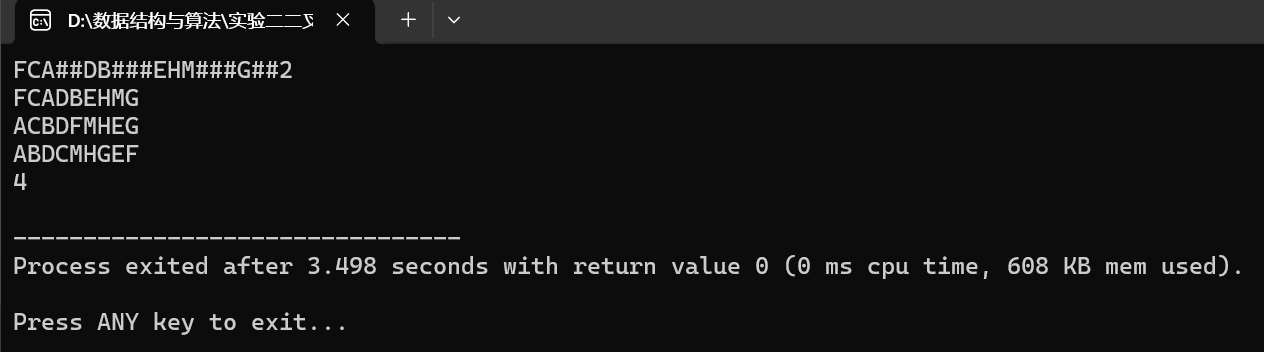
printf("%d\n", countLeaves(root));

freeTree(root);

return 0;

}

1. 运行结果：



1. 心得体会
2. 递归方法简洁高效，能够准确地按照先序序列生成树结构。
3. 对于多层嵌套递归，需要特别关注边界条件。
4. 遍历的顺序反映了不同的递归操作顺序，三种遍历方法逻辑清晰，可通过递归轻松实现。
5. 递归统计叶子节点数是一种直观的做法，逻辑上与树的遍历相似。
6. 在程序结束时，需要递归释放每个节点的内存以避免内存泄漏。

实验内容2：求根结点到x结点的路径

文件名：实验二求根节点到目标节点距离.cpp

1. **问题描述：**

从键盘输入任意多个正整数，输入以-1结束。逆序输出这些整数(不包括-1)。 提示： 1、逆序创建单链表。结点数据域是整型数。每输入一个整数，向链表中插入一个结点。当输入-1时结束链表的创建。 2、遍历链表，输出结点数据域的值。 3、遍历完成后，要求销毁该链表。

**输入格式:**

任意多的正整数，输入序列以-1结束。

输出格式:

逆序输出这些整数（不包括-1）。

**输入样例:**

5 2 8 2 9 12 99 56 235 77 52 33 -1

**输出样例:**

33 52 77 235 56 99 12 9 2 8 2 5

1. 数据结构设计和核心算法（用图和文字描述清楚）：

数据结构设计

**树节点结构：**Node

使用一个结构体 Node 表示二叉树中的一个节点：

data: 用于存储节点的字符数据。

LChild 和 RChild: 分别指向左子树和右子树。

**typedef** **struct** Node   
**{**    **char** data**;**    **struct** Node**\*** LChild**;**    **struct** Node**\*** RChild**;  
}** Node**,** Tree**;**

核心算法描述

1. **二叉树的构建void** create\_tree**(**Tree**\*\*** root**)**

使用递归方法，从先序序列中逐个读取字符。

如果读取到 #，表示当前节点为空，返回 NULL。

如果是普通字符，则创建一个新节点，并递归构建左子树和右子树。

void create\_tree(Tree\*\* root)

{

char userKey = 0;

scanf(" %c", &userKey);

if (userKey == '#') {

\*root = NULL;

} else {

\*root = (Tree\*)malloc(sizeof(Tree));

assert(\*root);

(\*root)->data = userKey;

create\_tree(&(\*root)->LChild);

create\_tree(&(\*root)->RChild);

}

}

2. **查找路径的算法bool** find\_path**(**Tree**\*** root**,** **char** x**,** **char\*** path**,** **int\*** pathLen**)**

如果当前节点为空，返回 false。

将当前节点加入路径数组，并递增 pathLen。

如果当前节点是目标节点，返回 true。

递归查找左子树或右子树。

如果找到目标节点，返回 true。如果目标节点不在左子树和右子树中，则从路径中移除当前节点（回溯），并减少 pathLen。

bool find\_path(Tree\* root, char x, char\* path, int\* pathLen)

{

if (root == NULL) {

return false;

}

*// 当前节点加入路径*

path[\*pathLen] = root->data;

(\*pathLen)++;

*// 检查当前节点是否是目标节点*

if (root->data == x) {

return true;

}

*// 在左子树或右子树中查找目标节点*

if (find\_path(root->LChild, x, path, pathLen) ||

find\_path(root->RChild, x, path, pathLen)) {

return true;

}

*// 回溯：移除当前节点*

(\*pathLen)--;

return false;

}

1. 程序及注释

程序名： 实验二求根节点到目标节点距离.cpp

#include <stdio.h>

#include <stdbool.h>

#include <stdlib.h>

#include <assert.h>

typedef struct Node

{

char data;

struct Node\* LChild;

struct Node\* RChild;

} Node, Tree;

*// 创建二叉树*

void create\_tree(Tree\*\* root)

{

char userKey = 0;

scanf(" %c", &userKey);

if (userKey == '#') {

\*root = NULL;

} else {

\*root = (Tree\*)malloc(sizeof(Tree));

assert(\*root);

(\*root)->data = userKey;

create\_tree(&(\*root)->LChild);

create\_tree(&(\*root)->RChild);

}

}

*// 查找从根节点到目标节点的路径*

bool find\_path(Tree\* root, char x, char\* path, int\* pathLen)

{

if (root == NULL) {

return false;

}

*// 当前节点加入路径*

path[\*pathLen] = root->data;

(\*pathLen)++;

*// 检查当前节点是否是目标节点*

if (root->data == x) {

return true;

}

*// 在左子树或右子树中查找目标节点*

if (find\_path(root->LChild, x, path, pathLen) ||

find\_path(root->RChild, x, path, pathLen)) {

return true;

}

*// 回溯：移除当前节点*

(\*pathLen)--;

return false;

}

int main()

{

Tree\* A = NULL;

create\_tree(&A);

char x;

scanf(" %c", &x);

char path[100];

int pathLen = 0;

if (find\_path(A, x, path, &pathLen)) {

for (int i = 0; i < pathLen; i++) {

printf("%c ", path[i]);

}

printf("\n");

} else {

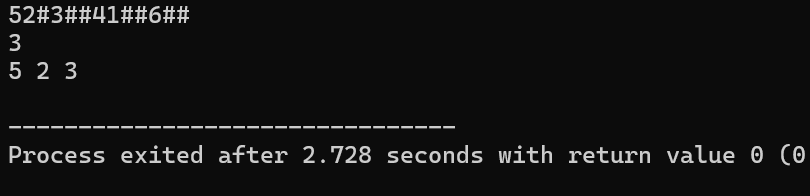
printf("Node not found.\n");

}

return 0;

}

1. 运行结果：



使用递归函数 find\_path 从根节点开始查找，路径存储在 path 数组中，pathLen 记录路径的长度。根节点到目标节点 3 的路径依次经过 5 -> 2 -> 3。

1. 心得体会
2. 递归方法简洁高效，能够准确地按照先序序列生成树结构。
3. 回溯（移除路径节点）保证了 path 数组始终仅保存有效路径，节省了空间和时间。
4. 对空节点和无效输入的边界处理严谨，提升了代码严谨性。
5. 使用递归算法实现避免了繁琐的手动管理栈操作，逻辑清晰，符合直观的树遍历思维。