沈阳航空航天大学

人工智能学院

**实验报告**

|  |  |
| --- | --- |
| 课程名称 | 算法导论 |
| 专 业 | 物联网工程 |
| 班 级 | 物联网2202 |
| 学 号 | 223428010210 |
| 学生姓名 | 陈梓欣 |
| 指导教师 | 孙恩岩 |
| 实验时间 | 2024年6月2日 1、2节 |
| 实验地点 | 机械馆410-3 |

# 一、实验名称

二叉树的建立和遍历算法

# 二、实验目的

掌握二叉树的链式存储结构，实现二叉树的创建、遍历及队列的创建、入队、出队等。

# 三、实验内容和要求

建立二叉树，并实现二叉树的遍历，先序遍历采用递归算法实现，层次遍历用非递归算法来实现。

# 四、实验环境

Microsoft Visual Studio 2022

# 五、实验设计

本次实验涉及到了二叉树的创建、遍历和打印，涉及的算法和数据结构设计如下：

1. 算法设计：

- 创建二叉树的算法：根据先序遍历字符串递归创建二叉树。该算法遍历输入的先序遍历字符串，根据字符创建节点，并设置节点的左右子树指针。

- 先序遍历算法：通过递归方式实现二叉树的先序遍历，即先访问根节点，然后递归遍历左子树和右子树。

- 层次遍历算法：使用队列实现二叉树的层次遍历，从根节点开始，依次访问每一层的节点，并将它们的子节点按顺序入队，然后逐个出队访问节点。

- 获取树高度算法：通过递归方式实现，每次递归返回当前节点的高度，并根据左右子树的高度取较大值再加上1作为当前节点的高度，最终返回树的高度。

2. 数据结构设计：

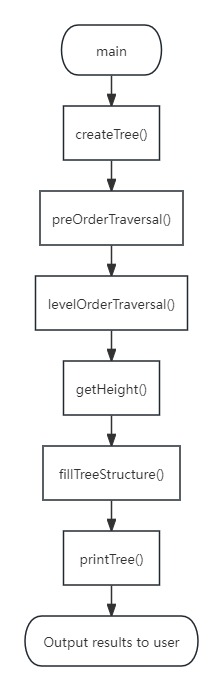
- `TreeNode` 结构体：表示二叉树的节点，包括节点的数据、左子树指针和右子树指针。

- 队列（Queue）：用于层次遍历算法中暂存节点，实现层次遍历的顺序。

3. 网络拓扑结构设计：

- 该实验中未涉及网络拓扑结构设计，主要集中在二叉树的算法和数据结构设计上。

以上算法和数据结构设计为实现二叉树的创建、遍历和打印提供了基础，使得程序能够清晰地展示二叉树的结构和遍历方式。



**图1 主模块流程图**

程序代码要加入必要注释，程序示例如下：

#include <iostream>

#include <queue>

#include <vector>

#include <iomanip>

#include <cmath>

using namespace std;

// 定义二叉树的节点结构

struct TreeNode {

char data; // 节点的数据

TreeNode\* left; // 左子树指针

TreeNode\* right; // 右子树指针

TreeNode(char val) : data(val), left(nullptr), right(nullptr) {}

};

// 根据先序遍历字符串递归创建二叉树

TreeNode\* createTree(const string& str, int& index) {

// 如果到达字符串末尾或遇到'#'字符，返回空节点

if (index >= str.length() || str[index] == '#') {

index++;

return nullptr;

}

// 创建当前节点，并递归创建左子树和右子树

TreeNode\* root = new TreeNode(str[index++]);

root->left = createTree(str, index);

root->right = createTree(str, index);

return root;

}

// 递归实现先序遍历

void preOrderTraversal(TreeNode\* root) {

if (root == nullptr) {

return;

}

cout << root->data << " "; // 访问节点

preOrderTraversal(root->left); // 访问左子树

preOrderTraversal(root->right); // 访问右子树

}

// 非递归实现层次遍历

void levelOrderTraversal(TreeNode\* root) {

if (root == nullptr) {

return;

}

queue<TreeNode\*> q; // 创建队列

q.push(root); // 将根节点入队

while (!q.empty()) {

TreeNode\* node = q.front(); // 获取队首节点

q.pop(); // 移除队首节点

cout << node->data << " "; // 访问节点

// 将左子节点入队

if (node->left != nullptr) {

q.push(node->left);

}

// 将右子节点入队

if (node->right != nullptr) {

q.push(node->right);

}

}

}

// 获取树的高度

int getHeight(TreeNode\* root) {

if (root == nullptr) return 0;

return 1 + max(getHeight(root->left), getHeight(root->right));

}

// 辅助函数，用于递归生成树的文本表示

void fillTreeStructure(TreeNode\* root, int depth, int pos, int offset, vector<vector<string>>& res) {

if (root == nullptr) return;

// 确保结果向量有足够的空间来保存当前层级

if (depth >= res.size()) {

res.push\_back(vector<string>((1 << (res.size() + 1)) - 1, " "));

}

// 在适当的位置存放节点值

res[depth][pos] = string(1, root->data);

// 递归处理左子树和右子树

fillTreeStructure(root->left, depth + 1, pos - offset, offset / 2, res);

fillTreeStructure(root->right, depth + 1, pos + offset, offset / 2, res);

}

// 打印树的文本表示

void printTree(TreeNode\* root) {

int height = getHeight(root);

int width = (1 << height) - 1;

vector<vector<string>> res(height, vector<string>(width, " "));

fillTreeStructure(root, 0, (width - 1) / 2, (width + 1) / 4, res);

for (const auto& level : res) {

for (const auto& node : level) {

cout << node;

}

cout << endl;

}

}

int main() {

string preOrder = "ABDH###E#I##CFJ###G##"; // 输入的先序遍历字符串

int index = 0; // 字符串索引

// 创建二叉树

TreeNode\* root = createTree(preOrder, index);

// 输出先序遍历结果

cout << "Preorder Traversal: ";

preOrderTraversal(root);

cout << endl;

// 输出层次遍历结果

cout << "Level Order Traversal: ";

levelOrderTraversal(root);

cout << endl;

// 输出树的文本表示

cout << "Tree Structure:" << endl;

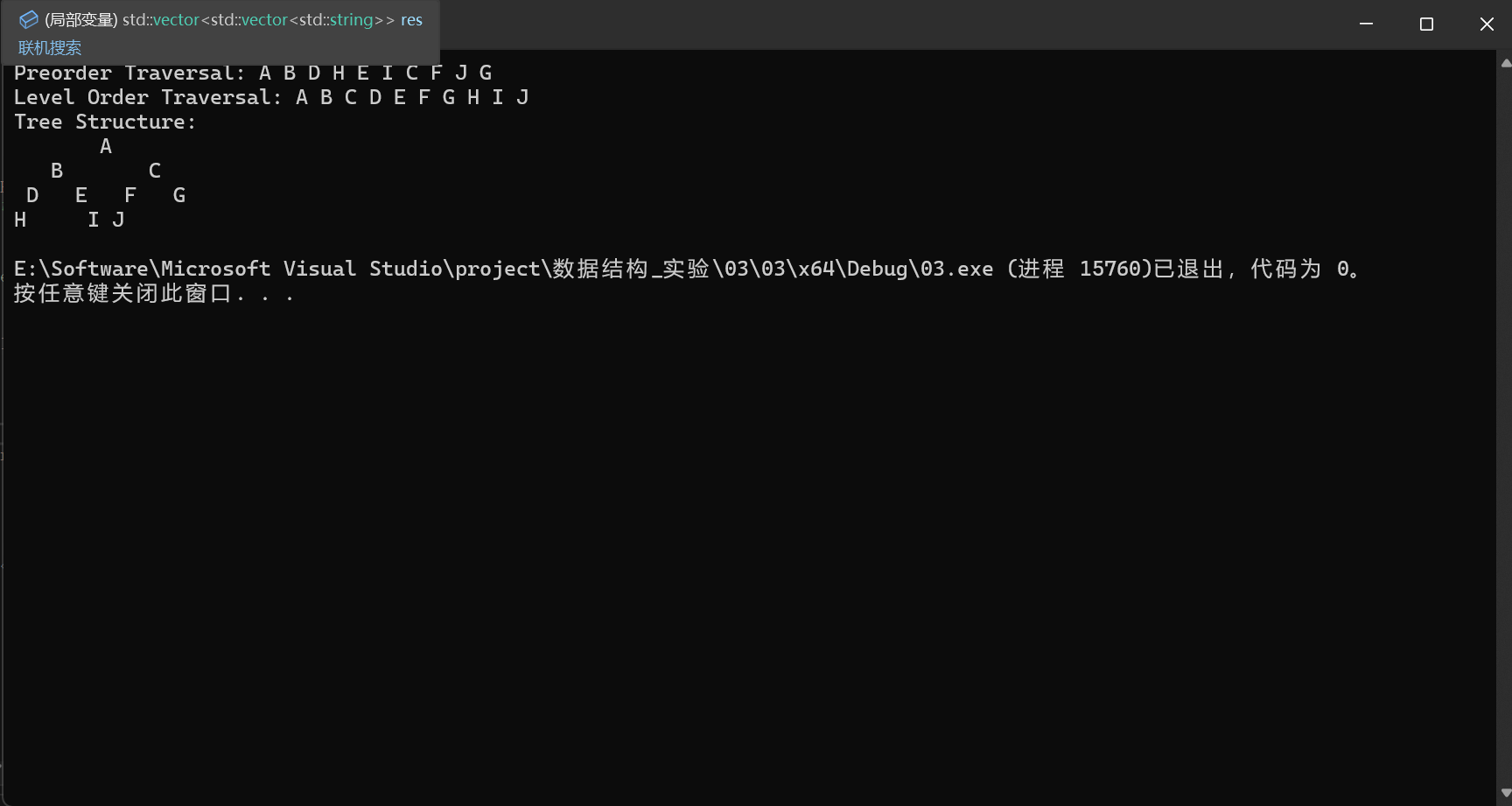
printTree(root);

return 0;

}

# 六、实验步骤及实验结果

本节用文字和屏幕截图详实记录实验的设计结果、完成过程（步骤）和每一步的测试结果



**图2 打印信息**

# 七、结论

本次实验通过二叉树的创建、遍历和打印，深入验证了多个重要算法和数据结构的应用。首先，利用递归方法根据先序遍历字符串创建二叉树，验证了递归在处理树形结构中的高效性和简洁性。

其次，通过实现先序遍历和层次遍历，分别展示了递归和迭代（结合队列）在不同遍历方式中的适用性。递归方法适用于深度优先搜索（DFS），而队列在广度优先搜索（BFS）中的应用则进一步验证了其在层次遍历中的不可或缺性。

此外，通过计算树的高度，深入理解了递归深度与树高度的关系。最后，通过文本表示法打印树结构，不仅直观展示了树形数据结构，还帮助调试和验证算法的正确性。

实验结果表明，选择合适的数据结构和算法对于简化问题解决过程、提升代码可读性和维护性至关重要。总体而言，本次实验有效地加深了对二叉树及其相关算法的理解和应用。