# 作业 HW3 实验报告

姓名：　　学号　　日期：2024 年 11 月 5 日

## 1. 涉及数据结构和相关背景

本次实验题目主要针对**树和二叉树**进行。

**一、树**

树是一种非线性数据结构，由节点和边组成。每个节点可以有零个或多个子节点，但有且仅有一个父节点（除了根节点，它没有父节点）。

**二、二叉树**

二叉树是一种特殊的树，特殊在其每个节点最多有两个子节点，分别称为左孩子节点和右孩子节点。

## 2. 实验内容

### 2.1 二叉树的非递归遍历

#### 2.1.1 问题描述

二叉树的非递归遍历可通过**栈**来实现。如果已知**中序遍历**的**栈**的操作序列，就可唯一地确定一棵二叉树。请编程输出该二叉树的后序遍历序列。

#### 2.1.2 基本要求

输入：第一行一个整数 n，表示二叉树的结点个数。接下来 2n 行，每行描述一个栈操作，格式为：push X 表示将结点 X 压入栈中，pop 表示从栈中弹出一个结点。(X 用一个字符表示)

输出：一行，后序遍历序列。

#### 2.1.3 数据结构设计

#1.自定义一个 SqStack 栈类，便于后续设想的一些操作

```
struct SqStack {
private:
    SElemType *base;
    SElemType *top;
    int stacksize;
public:
    ~SqStack();                 //销毁已有的栈
    SqStack();                  //构造空栈
    Status ClearStack();        //把现有栈置空栈
    Status Top(SElemType& e);   //取栈顶元素
    Status Pop(SElemType& e);   //弹出栈顶元素
    Status Push(SElemType e);   //新元素入栈
    bool StackEmpty();          //是否为空栈
};
```

#2.定义二叉树节点结构

```
typedef struct BiTNode {
    TElemType data;
    BiTNode* lchild, * rchild;
}*BiTree;
```

#3.定义一个具有特定状态的二叉树节点，便于入栈出栈行为的记录

```
struct Node {
    BiTree bitree;
    int state;
    Node(BiTree b = NULL, int s = 0)
    {
        bitree = b;
        state = s;
    }
}p;
```

### 2.1.4 功能说明（函数、类）

```
/*
BiTree T    指向二叉树根节点的指针

Status(*visit)(TElemType e)   函数指针

TElemType 是二叉树节点中存储的数据类型

visit 函数，打印节点值
*/

//递归方法-后序遍历
Status PostOrderTraverse(BiTree T, Status(*visit)(TElemType e))
{
    if (树不空) {
        if (PostOrderTraverse(T->lchild, visit) == OK) //后序遍历该节点的左子树
            if (PostOrderTraverse(T->rchild, visit) == OK)  //后序遍历该节点的右子树
                if ((*visit)(T->data) == OK) //访问该节点的数据
                    return OK;
        return ERROR;
    }
    else {//空树返回OK
        return OK;
    }
}
```

```
//打印节点值的函数
Status VisitPrintNormal(TElemType e)
{
    cout << e;
    return OK;
}
```

```
/*
BiTree& T    指向待构建的二叉树的根节点
n    节点个数
*/

//根据中序遍历栈的操作构建二叉树

Status CreateBiTree(BiTree& T, const int n)
{
    struct Node { //定义结构体 Node
        BiTree bitree; //指向子树的指针
        int state; //表示节点状态，0为左子节点未设置，1为左子节点已设置，2表示左右子节点均设置
        Node(BiTree b = NULL, int s = 0)
        {
            bitree = b;
            state = s;
        }
    }p;

    SqStack<Node> stack; //定义栈 stack，用于存储 Node 类型的元素

    int i = 0, j = 0;

    do
    {
        string instr; //读取用户输入的指令 instr
        char name;
        cin >> instr;
       如果 instr 为 "push"
            i++
            读取用户输入的节点值 name
            创建新节点 newnode

            如果 i == 1（即创建的是根节点）
                T = newnode（设置根节点）
                将 Node(newnode, 0) 压入栈中
            否则
                从栈中弹出一个节点 p（代表当前节点的父节点）

                如果 p.state == 0（左右子节点均未设置）
                    p.state 设为 1（左子节点已设置）
                    将 p 重新压入栈中
                    设置 p.bitree 的左子节点为 newnode
                    将 Node(newnode, 0) 压入栈中
                否则（p.state == 1，即左子节点已设置）
                    设置 p.bitree 的右子节点为 newnode
                    p.state 设为 2（左右子节点均已设置）

    } while (i < n || j < n);

    return OK;
}
```

### 2.1.5 调试分析（遇到的问题和解决方法）

调试过程描述：

1. 由于本题自定义了 Sqstack 类，没有使用 c++自带的 stack 类，所以需要验证自定义类函数操作是否符合预期。　　解决方法，另开一个新程序单独测试自定义的类的函数。
2. 测试自定义的 Sqstack 类没有问题后，就整体写完程序运行，根据编译器报错修改程序中的语法错误。
3. 利用题目所给的测试数据生成文件，利用重定向输入/输出的方法，对照程序运行结果和提供的测试数据结果，验证程序逻辑的正确性。

### 2.1.6 总结和体会

收获：上课时老师说中序+先序/后序的遍历结果才能构造出一棵二叉树，但本题只给出中序遍历中栈的操作就也可以正确构造出一棵二叉树。加深了我对遍历过程中栈的变化过程的理解和树与栈的知识联系。

题目难点：本题的难点就在于分析中序遍历中栈的操作与节点的关系。通过设置节点 state 的状态来判断左右子节点是否已经设置，就能很好地构建出二叉树。

## 2.2 二叉树的同构

### 2.2.1 问题描述

给定两棵树 T1 和 T2。如果 T1 可以通过若干次左右孩子互换变成 T2，则我们称两棵树是“同构”的。现给定两棵树，请你判断它们是否是同构的。并计算每棵树的深度。

### 2.2.2 基本要求

输入：第一行是一个非负整数 N1，表示第 1 棵树的结点数；随后 N 行，依次对应二叉树的 N 个结点（假设结点从 0 到 N−1 编号），每行有三项，分别是 1 个英文大写字母、其左孩子结点的编号、右孩子结点的编号。如果孩子结点为空，则在相应位置上给出“-”。给出的数据间用一个空格分隔。接着一行是一个非负整数 N2，表示第 2 棵树的结点数；随后 N 行同上描述一样，依次对应二叉树的 N 个结点。

输出：共三行。第一行，如果两棵树是同构的，输出“Yes”，否则输出“No”。后面两行分别是两棵树的深度。

### 2.2.3 数据结构设计

```
//定义二叉树节点结构
typedef struct BiTNode {
    TElemType data;    //数据域 TElemType定义为char型
    BiTNode* lchild, * rchild;  //左孩子右孩子指针
}*BiTree;
```

### 2.2.4 功能说明（函数、类）

```
//功能：根据输入构建二叉树的函数
Status CreateBiTree(BiTree& T)  //引用参数 T
{
    int n;
    cin >> n; //输入节点个数
    BiTNode* base = new(nothrow) BiTNode[n];  //生成对应数量的节点
    bool* vis = new(nothrow) bool[n]; //用于标记每个节点是否已被访问过

    //关键代码
    for (int i = 0; i < n; i++) {
        char data;
        int lchild, rchild;
        string lchilds, rchilds;
        //输入每个节点的数据域和左右孩子节点
        cin >> data >> lchilds >> rchilds;
        /*检查左孩子索引和右孩子输入是否为"-"。
        如果是，则设置为-1，表示没有孩子；
        如果不是，则使用stoi函数将字符串转换为整数。*/
        if (lchilds == "-")
```

```
            lchild = -1;
        else
            lchild = stoi(lchilds);
        if (rchilds == "-")
            rchild = -1;
        else
            rchild = stoi(rchilds);

        /*将当前节点的数据存储到base[i]的data字段中；
        如果左/右孩子不是-1，则将左孩子指针设置为base[lchild]的地址，
        并将vis[lchild]标记为true，表示该节点已被访问过；
        如果左/右孩子索引是-1，则将左孩子指针设置为NULL。*/
        base[i].data = data;

        if (lchild != -1) {
            base[i].lchild = &base[lchild];
            vis[lchild] = true;
        }
        else
            base[i].lchild = NULL;

        if (rchild != -1) {
            base[i].rchild = &base[rchild];
            vis[rchild] = true;
        }
        else
            base[i].rchild = NULL;
    }

    return OK;
}
```

```
//功能--判断两棵二叉树是否同构
bool IsSimilarTree(BiTree T1, BiTree T2)
{
    //两边都走到头了 跳出
    if (!T1 && !T2)
        return true;
    //当前节点不同就跳出
    if ((T1 && !T2) || (!T1 && T2) || T1->data != T2->data)
        return false;

    return ((IsSimilarTree(T1->lchild, T2->lchild) && IsSimilarTree(T1->rchild, T2->rchild))
        || (IsSimilarTree(T1->lchild, T2->rchild) && IsSimilarTree(T1->rchild, T2->lchild)));
}
```

```
//功能--求一棵二叉树的深度
int Depth(BiTree T ) {
    if (T == NULL) {//递归到叶子节点的“ 子树” ，不存在，其深度为0，递归结束。
        return 0;
    }
    return max(Depth(T->lchild), Depth(T->rchild)) + 1;//如果有子树，其深度为左右子树的最大值+1。
}
```

### 2.2.5 调试分析（遇到的问题和解决方法）

调试过程描述：

1. 整体写完程序后运行，根据编译器报的错误修改程序中的语法错误。
2. 生成可执行文件，根据题目中给出的测试数据生成文件，利用输入输出重定向的方法，检测程序的输出结果是否正确。以此来验证函数逻辑是否正确。

### 2.2.6 总结和体会

收获：虽然本题比较简单，思路很明显，但是这道题涉及了对二叉树的深度求解，建立二叉树等基本操作，通过对这些基本操作的代码实现，让我更加熟悉了二叉树的相关知识。

题目难点分析：本题的难点可能在于判断二叉树是否同构吧。但只要想到了方法，巧妙利用递归求解，代码还是比较简单的，可能存在思维上的难点吧（用循环可能代码实现比较复杂，容易出错）。

## 2.3 感染二叉树所需要的总时间

### 2.3.1 问题描述

给一棵二叉树的根节点 root ， 二叉树中节点的值 互不相同 。另给你一个整数 start 。在第 0 分钟，感染将会从值为 start 的节点开始爆发。每分钟，如果节点满足 1）节点与一个已感染节点相邻；2）节点此前还没有感染，就会被感染：返回感染整棵树需要的分钟数。

### 2.3.2 基本要求

输入：第一行包含两个整数 n 和 start。接下来包含 n 行，描述 n 个节点的左、右孩子编号。

输出：一个整数，表示感染整棵二叉树所需要的时间。

**（ps：本题目好像有点问题，因为输入的时候没有输入节点值。可能是题目没有说清楚，最后根据多次尝试发现就是按照默认的，第一个输入的记为 1，以此类推，编号即节点值。）**

### 2.3.3 数据结构设计

```
//定义二叉树的节点
struct BiTNode {
    int value = 0;  //value存储节点值
    BiTNode* lchild = NULL;
    BiTNode* rchild = NULL; //设置左右孩子节点
   //构造函数，用于创建并初始化一个 BiTNode 节点
    BiTNode(int x) : value(x), lchild(nullptr), rchild(nullptr) {}
};
```

### 2.3.4 功能说明（函数、类）

```
//参考力扣题解->解决感染问题
//定义一个Solution的 类
class Solution {
public:
    int amountOfTime(BiTNode* root, int start) {
        unordered_map<int, vector<int>> graph; // 邻接表表示图
        // 深度优先搜索（DFS）构建图
        function<void(BiTNode*)> dfs = [&](BiTNode* node) {
            if (node == nullptr) return; // 如果节点为空，返回
            if (node->lchild != nullptr) {
                graph[node->value].push_back(node->lchild->value);
                graph[node->lchild->value].push_back(node->value); // 建立双向连接
                dfs(node->lchild); //递归访问左子节点
```

```
        }
        if (node->rchild != nullptr) {
            graph[node->value].push_back(node->rchild->value);
            graph[node->rchild->value].push_back(node->value); // 建立双向连接
            dfs(node->rchild); //递归访问右子节点
        }
        };

    dfs(root); // 从根节点开始构建图
    queue<int> q; // BFS队列，存储节点值
    q.push(start); // 将起始节点加入队列
    unordered_set<int> visited; //记录已访问的节点
    visited.insert(start); // 标记起始节点为已访问
    int time = 0; //记录传播所需的时间
    // BFS遍历
    while (!q.empty()) {
        int size = q.size(); // 当前层的节点数量
        for (int i = 0; i < size; i++) {
            int nodeVal = q.front(); // 获取当前节点值
            q.pop(); // 出队
            // 遍历当前节点的所有邻接节点
            for (int childVal : graph[nodeVal]) {
                if (!visited.count(childVal)) { // 如果邻接节点未被访问
                    visited.insert(childVal); // 标记为已访问
                    q.push(childVal); // 将邻接节点加入队列
                }
            }
        }
        if (!q.empty()) time++; // 当前层结束后增加时间
    }

    return time;
}
};



// 根据输入创建二叉树
//输入：给定的节点数量n和节点间的父子关系vector<pair<int, int>>& children
//返回值：构建好的树的根节点
BiTNode* CreateBiTree(int n, vector<pair<int, int>>& children) {
    vector<BiTNode*> nodes(n, nullptr);
    //分配新的 BiTNode 节点
    for (int i = 0; i < n; ++i) {
        nodes[i] = new BiTNode(i);
    }
    //循环寻找根节点
    for (int i = 0; i < n; ++i) {
        int lchild = children[i].first;
        int rchild = children[i].second;
        if (lchild != -1) {
            nodes[i]->lchild = nodes[lchild];
        }
        if (rchild != -1) {
            nodes[i]->rchild = nodes[rchild];
        }
    }
    return nodes[0];
}
```

### 2.3.5 调试分析（遇到的问题和解决方法）

调试过程：

1. 由于 Solution 类参考了力扣题解，得对 Solution 类进行适当修改以符合整个程序的逻辑。
2. 整体写完程序后运行，根据编译器报的错误修改程序中的语法错误。
3. 生成可执行文件，根据题目中给出的测试数据生成文件，利用输入输出重定向的方法，检测程序的输出结果是否正确。以此来验证函数逻辑是否正确。

遇到的问题：起初我并不想用力扣题解中的解答，因为对图的了解不是很多，我就想用栈之类的结构去解决，但发现特别容易溢出，最终还是选择了改造力扣题解中的解答，并对图进行了简单的了解。

### 2.3.6 总结和体会

题目难点分析：其实分析题目可以意识到这个问题实际是对树的深度求解问题的进阶--即求出值为 start 的节点到其他节点最远的距离，只是这个 start 节点并非一定是根节点，而这个也就是本题的难点。因此需要先将树的结构用深度优先搜索解析成无向图，再用广度优先搜索来求最长距离。而目前没有接触到图，就需要自己去了解，然后适当改造。

收获：在难点分析处写到需要自主学习图的相关知识，虽然耗时很久，但我提前对图这一数据结构有了大致的了解，我觉得这就是很大的收获。并且在改造力扣题解的过程中，也提高了个人阅读代码和编辑代码的能力。

## 2.4 树的重构

### 2.4.1 问题描述

将一个有序树表示为二叉树，并且求出转换前后的树的深度。

### 2.4.2 基本要求

输入：输入由多行组成，每一行都是一棵树的深度优先遍历时的方向。其中 d 表示下行(down)，u 表示上行(up)。可以假设每棵树至少含有 2 个节点，最多 10000 个节点。

输出：对每棵树，打印转化前后的树的深度，采用以下格式 Tree t: h1 => h2。其中 t 表示样例编号(从 1 开始)，h1 是转化前的树的深度，h2 是转化后的树的深度。

数据范围：行数<5000，总字符数<2,000,000

### 2.4.3 数据结构设计

```
/*定义一个二叉树的节点结构
  （和前面几道题一样的，不再过多解释）*/
typedef struct BiTNode {
    TElemType data;
    BiTNode* lchild, * rchild;
}*BiTree;
```

```
//创建二叉树时记录节点和访问状态的结构
 struct VisitNode {
        BiTree node;
        bool visit;  //标记当前节点是否已访问
        VisitNode(BiTree nn = NULL, bool vv = false)
        {
            node = nn;
            visit = vv;
        } //定义一个函数，记录节点和访问状态
    }q;
```

```
//构建一个结构体存储二叉树中的节点及其对应的深度
struct NodeWithDepth {
        BiTree node;
        int depth;
        NodeWithDepth(BiTree bb = NULL, int dd = 1)
        {
            node = bb;
            depth = dd;
        }//初始化node和depth的函数
    };
```

### 2.4.4 功能说明（函数、类）

```
//自定义一个SqStack类，以满足后序操作
struct SqStack {
private:
    SElemType* base;
    SElemType* top;
    int stacksize;
public:
    SqStack();                    //构造空栈
    ~SqStack();                   //销毁已有的栈
    Status ClearStack();          //把现有栈置空栈
    Status Top(SElemType& e);     //取栈顶元素
    Status Pop(SElemType& e);     //弹出栈顶元素
    Status Push(SElemType e);     //新元素入栈
    bool StackEmpty();            //是否为空栈
};
```

```
/*功能--根据输入的某棵树的dfs遍历顺序构建一棵二叉树*/
int CreateBiTree(BiTree& T, string input)
{
    /*创建一个栈VisitNode结构体的stack来存储节点及其访问状态*/
    //关键代码部分
    int depth = 0, nowdepth = 0;

    for (unsigned int i = 0; i < input.size(); i++) {
        switch (input[i]) {
            case 'd':
                nowdepth++;//增加当前深度nowdepth
                //更新最大深度depth为当前深度nowdepth和depth中的较大值
                depth = max(depth, nowdepth);
                //从栈中弹出顶部节点即当前节点的父节点
                stack.Pop(q);
                . . .
```

```
                p->lchild = p->rchild = NULL;
                if (!q.visit) { //还没有访问子节点
                    //将q标记为已访问，并将新节点p作为q的左孩子
                    q.visit = true;
                    q.node->lchild = p;
                    stack.Push(q);
                }
                else {          //是回溯回来的其他子节点 接到第一个左节点的后面
                    stack.Push(q);
                    //将新节点p作为last节点的右孩子
                    last->rchild = p;
                }
                stack.Push(VisitNode(p, false));
                break;
            case 'u':
                nowdepth--;
                stack.Pop(q);
                last = q.node;
                break;
        }
    }

    return depth;
}
```

**（并且顺便求出了这棵有序树的深度）**

```
//功能--求一棵二叉树的深度
int Depth(BiTree T ) {
    if (T == NULL) {//递归到叶子节点的"子树"，不存在，其深度为0，递归结束。
        return 0;
    }
    return max(Depth(T->lchild), Depth(T->rchild)) + 1;//如果有子树，其深度为左右子树的最大值+1。
}
```

### 2.4.5 调试分析（遇到的问题和解决方法）

调试过程：

1. 整体写完程序后运行，根据编译器报的错误修改程序中的语法错误。
2. 生成可执行文件，根据题目中给出的测试数据生成文件，利用输入输出重定向的方法，检测程序的输出结果是否正确。以此来验证函数逻辑是否正确。

遇到的问题：在编写 CreatBiTree 函数时，由于在构造时候顺便得到了有序树的深度，逻辑比较复杂，故单独开了一个测试程序去验证函数逻辑，然后在验证函数逻辑的过程中发现深度总是不对，后来检查程序发现缺少了 depth = max(depth, nowdepth);这一句，改正后函数逻辑即正确了。

### 2.4.6 总结和体会

题目难点分析：整个题目的解决分为三部分，一是有序树深度的求解，二是有序树向二叉树的转换，三是二叉树深度的求解。二叉树深度的求解在之前的题目中有所涉及，不难。难点就在于有序树向二叉树的转换。

收获：课堂上老师展示了有序树向二叉树的转换，但这道题改用代码实现，加深了我对树与二叉树之间转换的理解。

## 2.5 最近公共祖先

### 2.5.1 问题描述

给出一棵多叉树，求出两个节点的最近公共祖先。

一个节点的祖先节点可以是该节点本身，树中任意两个节点都至少有一个共同祖先，即根节点。

### 2.5.2 基本要求

输入：输入数据包含 T 个测试样本，每个样本 i 包含 Ni 个节点和 Ni-1 条边和 Mi 个问题，树中节点从 1 到 Ni 编号。输入第一行是测试样本数 T；每个测试样本 i 第一行为两个整数 Ni 和 Mi；接下来 Ni-1 行，每行 2 个整数 a、b，表示 a 是 b 的父节点；接下来 Mi 行，每行两个整数 x、y，表示询问 x 和 y 的共同祖先

数据范围：1<=T<=100;5<=N<=1000;5<=M<=1000;

输出：对于每一个询问输出一个整数，表示共同祖先的编号

### 2.5.3 数据结构设计

```
//定义多叉树节点结构
struct treenode {
    int val = 0;
    //多叉树孩子节点不定，但只有一个父节点
    treenode* parent = nullptr;
};
```

### 2.5.4 功能说明（函数、类）

```
// 功能--构建有序树
/*输入：firstnode--指向 treenode的指针
n--节点数
a，b--父，子节点*/
void buildtree(treenode*& firstnode, int n) {
    firstnode = new(nothrow) treenode[n + 1];
    int a, b; //a 父 ,b子
    . . .
   //关键代码--建立父子关系
    for (int i = 1; i < n; i++) { // n - 1 条边
        cin >> a >> b;
        firstnode[b].parent = &firstnode[a];
    }
}
```

```
//功能--寻找最近公共祖先
int search(treenode* firstnode, int a, int b) {
    . . .
    //关键代码
    // 将 a 的路径标记为已访问
    while (a != -1) {
        visited[a] = true;
        a = firstnode[a].parent ? firstnode[a].parent->val : -1; // 向上查找父节点
    }

    // 查找 b 的路径，找到第一个访问过的节点
    while (b != -1) {
        if (visited[b]) {
            return b; // 找到最近公共祖先
        }
        b = firstnode[b].parent ? firstnode[b].parent->val : -1; // 向上查找父节点
    }
```

### 2.5.5 调试分析（遇到的问题和解决方法）

调试过程：

1. 整体写完程序后运行，根据编译器报的错误修改程序中的语法错误。
2. 生成可执行文件，根据题目中给出的测试数据生成文件，利用输入输出重定向的方法，检测程序的输出结果是否正确。以此来验证函数逻辑是否正确。

### 2.5.6 总结和体会

收获：本道题不再聚焦于二叉树的相关问题解决，而是考查了多叉树的简单应用。通过这道题，我学到了多叉树节点结构的构造方法，以及寻找公共祖先节点的方法，并思考了这一方法在二叉树中的应用，对我解题思维的扩展有很大的帮助。

题目难点：题目难点可能就是向上回溯父节点吧，但增加了父指针后也还比较容易回溯。

## 2.6 求二叉树的后序

### 2.6.1 问题描述

给出二叉树的前序遍历和中序遍历，求树的后序遍历

### 2.6.2 基本要求

输入：输入包含若干行，每一行有两个字符串，中间用空格隔开；同行的两个字符串从左到右分别表示树的前序遍历和中序遍历，由单个字符组成，每个字符表示一个节点；字符仅包括大小写英文字母和数字，最多 62 个；输入保证一棵二叉树内不存在相同的节点

输出：每一行输入对应一行输出；若给出的前序遍历和中序遍历对应存在一棵二叉树，则输出其后序遍历；否则输出 Error

### 2.6.3 数据结构设计

```
//定义二叉树的节点
struct TreeNode {
    char val;//数据域类型
    TreeNode* left;//左孩子节点
    TreeNode* right;//右孩子节点
    //初始化节点的函数
    TreeNode(char x) : val(x), left(nullptr), right(nullptr) {}
};
```

### 2.6.4 功能说明（函数、类）

```
//功能--根据给定的前序遍历和中序遍历重建一棵二叉树
/*
输入:输入前序遍历和中序遍历的字符串以及开始和结尾的索引值
返回值：构建好的二叉树的根节点
*/
TreeNode* buildTreeHelper(const string& preorder, int preStart, int preEnd,
    . . .
    //关键代码
    // 前序遍历的第一个节点是根节点
    char rootVal = preorder[preStart];
    TreeNode* root = new TreeNode(rootVal);

    // 在中序遍历中找到根节点的位置
```

```
    int rootIndex = inorderMap[rootVal];

    // 递归构建左子树和右子树
    int leftSubtreeSize = rootIndex - inStart;
    root->left = buildTreeHelper(preorder, preStart + 1, preStart + leftSubtreeSize,
        inorder, inStart, rootIndex - 1, inorderMap);
    root->right = buildTreeHelper(preorder, preStart + leftSubtreeSize + 1, preEnd,
        inorder, rootIndex + 1, inEnd, inorderMap);

    return root;
}
```

```
// 功能--判断前序和中序遍历能否构成唯一的二叉树
/*输入：前序遍历和中序遍历的字符串
返回值：bool值
*/
bool canFormUniqueTree(const string& preOrder, const string& inOrder) {
    // 长度一致性检查
    if (preOrder.length() != inOrder.length()) {
        return false;
    }

    // 构建中序遍历字符位置映射
    inOrderMap.clear();
    for (int i = 0; i < inOrder.length(); i++) {
        inOrderMap[inOrder[i]] = i;
    }
```

```
// 功能--递归后序遍历二叉树
void postorderTraversal(TreeNode* root, vector<char>& result) {
    if (root == nullptr) {
        return;
    }
    postorderTraversal(root->left, result);
    postorderTraversal(root->right, result);
}
```

### 2.6.5 调试分析（遇到的问题和解决方法）

调试过程：

1. 整体写完程序后运行，根据编译器报的错误修改程序中的语法错误。
2. 生成可执行文件，根据题目中给出的测试数据生成文件，利用输入输出重定向的方法，检测程序的输出结果是否正确。以此来验证函数逻辑是否正确。

遇到的问题：在设计判断能否构成二叉树的函数时，一开始只是设计了检查长度是否相等，发现测试数据有误，于是反思自己的逻辑，增加了映射关系的检查。

### 2.6.6 总结和体会

题目难点分析：该题目难点对我来说其实不在根据中序、先序构建二叉树，而是在能否构成二叉树的检查上。因为映射关系的化设计了图的相关知识，属于我的薄弱区。

收获：做完本题，加深了我对中序-先序构建二叉树的过程理解，同时设计检查能否构成二叉树的函数的过程也使我思考问题变得更加全面了。

## 2.7 表达式树

### 2.7.1 问题描述

给一个中缀表达式，这个中缀表达式用变量来表示（不含数字），请你将这个中缀表达式用表达式二叉树的形式输出出来。

### 2.7.2 基本要求

输入：分为三个部分；第一部分为一行，即中缀表达式(长度不大于 50)；中缀表达式可能含有小写字母代表变量（a-z），也可能含有运算符（+、-、*、/、小括号）；不含有数字，也不含有空格；第二部分为一个整数 n(n <= 10)，表示中缀表达式的变量数；第三部分有 n 行，每行格式为 C　x，C 为变量的字符，x 为该变量的值。

数据范围：1<=n<=10，1<=x<=100；

输出：输出分为三个部分，第一个部分为该表达式的逆波兰式，即该表达式树的后根遍历结果。占一行；第二部分为表达式树的显示，

### 2.7.3 数据结构设计

```
//定义二叉树节点的结构
typedef struct BiTNode {
    TElemType data;
    BiTNode* lchild, * rchild;
}*BiTree;
```

### 2.7.4 功能说明（函数、类）

```
//自定义一个SqStack类，以满足后序操作
struct SqStack {
private:
    SElemType* base;
    SElemType* top;
    int stacksize;
public:
    SqStack();                  //构造空栈
    ~SqStack();                 //销毁已有的栈
    Status ClearStack();        //把现有栈置空栈
    Status Top(SElemType& e);   //取栈顶元素
    Status Pop(SElemType& e);   //弹出栈顶元素
    Status Push(SElemType e);   //新元素入栈
    bool StackEmpty();          //是否为空栈
};
```

```
//自定义一个SqQueue（循环队列）类，以满足后序操作
class SqQueue {
private:
    QElemType* base;
    int MAXQSIZE;
    int front;
    int rear;
public:
    SqQueue(int maxqsize);          //建立空队列（构造函数）
    ~SqQueue();                     //销毁队列（析构函数）
    int QueueLength();          //获取队列长度
    Status EnQueue(QElemType e);          //新元素入队
    Status DeQueue(QElemType& e);          //队首元素出队
    Status GetHead(QElemType& e);          //获取队首元素
    Status ClearQueue();     //清空队列
    bool QueueEmpty();          //判断队列是否为空
};
```

**//功能--构建表达式树**

```
Status CreateBiTree(BiTree& T)
{
    . . .
    /*初始化两个栈Operator和Value
    用于后续的运算符和操作数的存储
    具体代码略*/

    //遍历字符串中的每个字符，根据类型分类处理
    for (unsigned int i = 0; i < input.size(); i++) {
        . . .
        //如果是运算符，根据运算优先级入Operator栈
        if (input[i] == '(' || input[i] == ')' || input[i] == '+' || input[i] == '-' || input[i] == '*' || input[i] == '/') {
            switch (input[i]) {
                case '('://(直接压入Operator栈
                    Operator.Push(p);
                    break;
                case ')'://)则持续从Operator栈中弹出运算符并构建子树，直至遇到(为止
                    while (!Operator.StackEmpty() && (Operator.Top(q), q->data != '('))
                    {
                        MakeExpressionTree(Value, Operator);
                    }
                    Operator.Pop(q); //去除左括号
                    break;
                case '*':
                case '/'://*或/，需检查Operator栈顶元素
                    while (!Operator.StackEmpty() && (Operator.Top(q), q->data == '*' || q->data == '/'))
                    {
                        MakeExpressionTree(Value, Operator);
                    }
                    Operator.Push(p);
                    break;
                case '+':
                case '-'://+或-，需检查Operator栈顶元素
                    while (!Operator.StackEmpty() && (Operator.Top(q), q->data != '('))
                    {
                        MakeExpressionTree(Value, Operator);
                    }
                    Operator.Push(p);
                    break;
            }
        }
        else {//如果是变量值，直接入栈
            Value.Push(p);
        }
    }

    . . .
}
```

```
//递归方法-后序遍历--输出逆波兰表达式（很熟悉的操作了，前面多次解释过，此处不再解释）
Status PostOrderTraverse(BiTree T, SqStack<int>& stack, Status(*visit)(SqStack<int>& stack,
                                TElemType e, set* list, int n), set* list, int n)
{
    if (T) {
        if (PostOrderTraverse(T->lchild, stack, visit, list, n) == OK)
            if (PostOrderTraverse(T->rchild, stack, visit, list, n) == OK)
                if ((*visit)(stack, T->data, list, n) == OK)
                    return OK;
        return ERROR;
    }
    else   //空树返回OK
        return OK;
}
```

```
//功能--打印表达式树
Status PrintTree(BiTree T)
{
    int depth = 0;
    {
        . . .
        /*遍历整棵树，使用队列来存储节点和它们的深度。*/

        //NodeWithDepth 结构体用于存储节点和对应的深度
        SqQueue<NodeWithDepth> queue(1000);  //queue的最大长度

        NodeWithDepth p;
        queue.EnQueue(NodeWithDepth(T, 1));
        while (!queue.QueueEmpty()) {
            if (queue.DeQueue(p) != 1)
                return -1;
            depth = max(depth, p.depth);
            if (p.node->lchild)
            //遍历过程中，不断更新 depth 变量以记录树的最大深度
                queue.EnQueue(NodeWithDepth(p.node->lchild, p.depth + 1));
            if (p.node->rchild)
                queue.EnQueue(NodeWithDepth(p.node->rchild, p.depth + 1));
        }
    }
```

```
{
    . . .
    //结构体 NodeWithNo 来存储节点、深度和节点编号
    . . .
    //详细打印逻辑
    while (!queue.QueueEmpty()) {//queue 用于层次遍历
        . . .
        if (p.node->lchild)
            queue.EnQueue(NodeWithNo(p.node->lchild, p.depth + 1, 2 * p.no));
        if (p.node->rchild)
            queue.EnQueue(NodeWithNo(p.node->rchild, p.depth + 1, 2 * p.no + 1));
        //输出当前节点
        cout << p.node->data;
        //queue2 用于跟踪当前层的节点，以便正确地添加斜杠
        queue2.EnQueue(p);
        if (queue.QueueEmpty())
            cout << endl;
        else {
            queue.GetHead(q);
            if (q.depth == p.depth) {   //在相同层次
                PrintSpace((q.no - p.no) * pow2((depth - p.depth) + 1) - 1);
            }
```

```
            else {                          //在不同层次
                cout << endl;
                //先处理行首空格
                queue2.GetHead(r);
                PrintSpace(pow2(depth - p.depth) - 2 + (r.no - pow2(p.depth - 1)) * pow2((depth - p.depth) + 1));
                while (!queue2.QueueEmpty()) {
                    queue2.DeQueue(r);
                    cout << (r.node->lchild ? '/' : ' ') << ' ' << (r.node->rchild ? '\\' : ' ');
                    if (!queue2.QueueEmpty()) {  //不是行末
                        queue2.GetHead(s);
                        PrintSpace((s.no - r.no) * pow2((depth - s.depth) + 1) - 3);
                    }
                }
                cout << endl;
                PrintSpace(pow2(depth - q.depth) - 1 + (q.no - pow2(p.depth)) * pow2((depth - q.depth) + 1));
            }
        }
    }
}
```

**其他类似打印空格等函数由于篇幅问题不再单列（其实现也很简单，上述列出了重要的函数实现逻辑）**

### 2.7.5 调试分析（遇到的问题和解决方法）

调试过程：

1. 由于该程序体量非常大，涉及的函数很多，因此需要单列一些程序，设计简单的测试数据仔细检查每个函数的实现逻辑。
2. 整体写完程序后运行，根据编译器报的错误修改程序中的语法错误。
3. 生成可执行文件，根据题目中给出的测试数据生成文件，利用输入输出重定向的方法，检测程序的输出结果是否正确。以此来验证函数逻辑是否正确。

遇到的问题：在设计表达式树的打印时，牵扯了很多函数。第一代表达式树打印的函数，打印出来的表达式树错位--经逐行检查，发现是追踪错误；第二次的表达式打印函数，打印出的空格不对，又进行检查--后和舍友一起研究发现是打印空格的函数有误……最终查阅了相关代码资料，完善最终的函数。

### 2.7.6 总结和体会

题目难点分析：题目的输出分为三部分，因此函数的设计也可以从三部分入手。第一部分后序遍历，只要能构建出表达式数，后序遍历即可，不是很复杂；第三部分带入值求表达式值，在输入的表达式基础上赋值即可，也不是很复杂；最复杂也就是题目的难点就是第二部分的处理--如何打印出符合要求的表达式树。涉及了大量其他辅助函数，如空格打印等等，而且一行行打印也牵扯了大量队列的使用，非常非常复杂。

收获：本题的难度可以说是非常非常大了，无论是从代码量还是从设计逻辑上来讲难度都非常非常大。做完本题，最大的收获是自己分析问题的能力和查阅资料的能力得到了提高。

## 3. 实验总结

本次实验过后，我总结了以下几点：

1. 加深了对基本概念的理解。题目综合性高，综合考查了树、二叉树的深度求解、遍历、构建等基本操作，这些过程都无形中强化了我对基本概念-树的深度、度等的理解和记忆。
2. 加强了解决实际问题的能力。本次实验都是基于一定的背景给出树的应用，每道题目都需要分析解题的步骤和数据结构之间的关联以及在结构之上的算法操作等，对我的解题思维有很大的提高。

3. 意识到了自己的薄弱处。在解决题目的过程中，很明显地感觉题目难度加大后，自己需要查阅很多的资料以及和周围人讨论才能做出，说明自己本身在代码撰写方面存在明显的不足，需要后期继续加强，同时为我后期的学习树立了明确的目标。