作业 HW3 实验报告

姓名：贺胡鸣 学号：2353626 日期：2025 年 11 月 5 日

1. 涉及数据结构和相关背景

在计算机科学领域，数据结构是组织和存储数据的核心方式，而树（Tree）无疑是其中最重要且应用最广泛的结构之一。树是一种非线性的、分层的数据结构，它模拟了自然界中树的形态，由节点（Node）和边（Edge）组成。与线性结构的数组和链表不同，树呈现出一对多的关系，一个父节点可以拥有多个子节点，这种特性使其天生适合表示具有层次关系的数据。

树的诞生背景与价值在于解决传统线性结构的局限性。在计算机科学发展的早期，人们发现许多实际问题中的数据并非简单的序列关系，而是存在复杂的层次和分支。例如，文件系统的目录结构、公司的组织架构、家族的血缘关系等，这些场景用线性结构表示既低效又不直观。树的出现完美地解决了这一问题，它使得数据的插入、删除和查找操作在保持层次关系的同时，能够达到比线性结构更高的效率。

在众多树结构中，二叉树（Binary Tree）占据了特殊而重要的地位。二叉树规定每个节点最多只能有两个子节点，通常称为左子节点和右子节点。这种简洁的约束带来了算法实现上的便利和效率上的优势。二叉树不仅是理解更复杂树结构的基础，其本身也有着极其丰富的变体和应用。从作为高效搜索工具的二叉搜索树（BST），到保证平衡性能的 AVL 树和红黑树，再到用于数据压缩的哈夫曼树，以及用于堆排序的二叉堆，二叉树几乎渗透到了计算机科学的各个角落。

特别值得强调的是，二叉树的理论研究催生了许多经典的算法思想。递归在二叉树遍历（前序、中序、后序）中得到了最自然和优雅的体现，而树的深度、平衡性、路径等概念也成为算法设计和分析的重要模型。本文讨论的“同构”问题，正是二叉树理论研究的一个典型代表，它探讨的是在忽略左右顺序差异的情况下，两棵树在结构上的等价性，这种抽象为模式识别、编译器设计等领域提供了理论基础。

2. 实验内容

2.1 题目一 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。提示：本题有多种解法，仔细分析二叉树非递归遍历过程中栈的操作规律与遍历序列的关系，可将二叉树构造出来。

2.1.2 基本要求

输入格式
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)
对于 20%的数据，0<n<=10
对于 40%的数据，0<n<=20
对于 100%的数据，0<n<=83

输出格式
一行，后序遍历序列。

2.1.3 数据结构设计

本程序采用二叉树结构来存储树形数据，并使用栈来模拟中序遍历过程，具体数据结构定义如下：

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

该结构体表示二叉树的节点，包含三个字段：`data` 存储节点的字符值，`lchild` 指向左子节点，`rchild` 指向右子节点。通过这种递归定义的结构，可以完整地表示任意形态的二叉树。
程序中使用 C++标准库的栈来模拟中序遍历过程：

```
stack<BiTree> stk;
BiTree root = NULL;
BiTree lastpop = NULL;
bool start = true;
```

2.1.4 功能说明（函数、类）

（1）后序遍历访问函数
简单输出节点字符并返回 true

```
bool print(char e)
{
    cout << e;
    return true;
}
```

（2）递归后序遍历函数
该函数采用递归方式实现后序遍历（左-右-根），接受一个函数指针参数用于访问节点数据。

```
bool postorder(BiTree T,visitfuc visit)
{
    if (T) {
        if (postorder(T->lchild, visit))
            if (postorder(T->rchild, visit))
                if (visit(T->data))
                    return true;
        return false;
    }
    else
        return true;
}
```

（3）二叉树构建函数

```
void CreateBiTree()
{
    stack<BiTree> stk;
    BiTree root = NULL;
    BiTree lastpop = NULL;
    bool start = true;

    for (int i = 0; i < 2 * n; i++) {
        string str;
        cin >> str;

        if (str == "push") {
            char val;
            cin >> val;
            BiTree newnode = new BiTNode;
            newnode->data = val;
            newnode->lchild = NULL;
            newnode->rchild = NULL;
```

```
            if (start) { //根节点
                root = newnode;
                start = false;
            }
            else {
                if (lastpop) {  //如果上一个命令是pop，则这个节点是pop出来的节点的右孩子
                    lastpop->rchild = newnode;
                    lastpop = NULL;
                }
                else  //如果上个命令是push，则这个节点是push节点的左孩子
                    stk.top()->lchild = newnode;
            }
            stk.push(newnode);
        } //end of if push
        else if (str == "pop") {
            lastpop = stk.top();
            stk.pop();
        } //end of if pop
}
```

```
    }
    //cout << root->data << " " << lastpop->data << endl;
    //后序遍历
    postorder(root, print);
}
```

这是程序的核心函数，通过分析中序遍历的栈操作序列来构建二叉树。关键思路是：

push 操作创建新节点并入

如果是第一个节点，设为根节点

如果前一个操作是 pop，当前 push 的节点作为弹出节点的右孩子

如果前一个操作是 push，当前 push 的节点作为栈顶节点的左孩子

pop 操作弹出栈顶节点并记录为 lastpop

2.1.5 调试分析（遇到的问题和解决方法）

1. 节点父子关系确定问题

最初难以确定如何根据栈操作序列建立正确的父子关系。通过分析中序遍历的非递归算法特点发现：当连续 push 时，后 push 的节点是前 push 节点的左孩子；当 pop 后立即 push 时，新 push 的节点是弹出节点的右孩子。这一发现是解决问题的关键。

2. 根节点特殊处理问题

第一个 push 的节点是根节点，需要特殊处理。通过 start 标志位来区分根节点和其他节点，确保根节点正确建立。

3. 右子树连接时机问题

在 pop 操作后，需要记录弹出的节点，以便在下一个 push 操作时将其作为右孩子。使用 lastpop 指针来记录最近弹出的节点，并在下一次 push 时使用后及时置空。

2.1.6 总结和体会

通过本次实现，深入理解了二叉树中序遍历非递归算法的栈操作规律。中序遍历的栈操作序列实际上隐含了二叉树的完整结构信息，通过分析 push 和 pop 的序列可以重建整棵二叉树。

在解决问题的过程中，体会到了从具体算法执行过程中抽象出构造规律的重要性。中序遍历的非递归算法中，push 操作对应着向左子树深入，pop 操作对应着回溯并转向右子树。这种规律性的认识是解决问题的核心。

2.2 题目二 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。例如图 1 给出的两棵树就是同构的，因为我们把其中一棵树的结点 a、b、e 的左右孩子互换后，就得到另外一棵树。而图 2 就不是同构的。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

输入格式
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。

数据范围
对于 20%的数据，有 0<N1=N2<=10
对于 40%的数据，有 0<=N1=N2<=100
对于 100%的数据，有 0<=N1,N2<=10100
注意：题目不保证每个结点中存储的字母是不同的。

输出格式
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

本程序采用结构体数组来表示二叉树节点，具体数据结构定义如下：

```
struct BiTNode {
    char data;
    int lchild = -1, rchild = -1;
};

int n1, n2;
```

该结构体包含三个字段：`data` 存储节点的字符数据，`lchild` 存储左孩子节点的索引（使用-1 表示空），`rchild` 存储右孩子节点的索引（使用-1 表示空）。
程序使用向量来存储整棵二叉树：

```
vector<BiTNode> tree1(n1);
int root1 = -1;
if(n1>0)
    root1=CreateBiTree(tree1, n1);
cin >> n2;
vector<BiTNode> tree2(n2);
int root2 = -1;
if(n2>0)
    root2 = CreateBiTree(tree2, n2);
```

此外，在构建二叉树时使用了一个辅助向量来标记节点是否有父节点，以便找到根节点：

```
vector<bool> hasparent(n, false);
```

2.2.4 功能说明（函数、类）

（1）二叉树构建函数

```
//创建二叉树并且返回根节点索引
int CreateBiTree(vector<BiTNode>& tree, int n)
{
    vector<bool> hasparent(n, false);
    for (int i = 0; i < n; i++) {
        char data;
        string leftidx, rightidx;
        cin >> data >> leftidx >> rightidx;
        tree[i].data = data;
        if (leftidx != "-") {
            tree[i].lchild = stoi(leftidx);
            hasparent[tree[i].lchild] = true;
        }
        if (rightidx != "-") {
            tree[i].rchild = stoi(rightidx);
            hasparent[tree[i].rchild] = true;
        }
    }

    //找根节点
    for (int i = 0; i < n; i++)
        if (!hasparent[i])
            return i;
    return -1;
}
```

该函数读取输入数据并构建二叉树，同时通过标记父节点关系来找到根节点（没有父节点的节点即为根节点）。

（2）同构判断函数

```
//判断两棵树是否同构
bool judge(vector<BiTNode>& tree1, int root1, vector<BiTNode>& tree2, int root2)
{
    //两棵空树
    if (root1 == -1 && root2 == -1) return true;
    //一棵空一棵不空
    if ((root1 == -1 && root2 != -1) || (root1 != -1 && root2 == -1)) return false;
    //根节点数据不同
    if (tree1[root1].data != tree2[root2].data) return false;

    //情况1：左对左，右对右
    bool case1 = judge(tree1, tree1[root1].lchild, tree2, tree2[root2].lchild)
        && judge(tree1, tree1[root1].rchild, tree2, tree2[root2].rchild);
    //情况2：左对右，右对左
    bool case2 = judge(tree1, tree1[root1].lchild, tree2, tree2[root2].rchild)
        && judge(tree1, tree1[root1].rchild, tree2, tree2[root2].lchild);

    return case1 || case2;
}
```

该函数通过递归判断两棵树是否同构。首先确定终止条件，当两棵树对应子树的根节点为空

（-1）时结束递归。如果出现两棵子树一棵为空一棵不为空则判负，如果出现对应层的子树的根节点数据不同则判负。递归部分过程如下：

取一棵树的左右孩子节点作为下一层左右子树的根节点，对于每一层的节点来说，只有对应位置节点数据相同 or 左子树根节点数据=右子树根节点数据 && 右子树根节点数据=左子树根节点数据 两种情况。

（3）树深度计算函数

```
//求树的深度
int getdepth(vector<BiTNode>& tree, int root)
{
    if (root == -1) return 0;
    int leftdepth = getdepth(tree, tree[root].lchild);
    int rightdepth = getdepth(tree, tree[root].rchild);
    return max(leftdepth, rightdepth) + 1;
}
```

求深度时也采用递归方法，每一棵树的深度=左子树深度 和 右子树深度 中的较大值，递归终止条件是子树根节点为空-1.

2.2.5 调试分析（遇到的问题和解决方法）

1. 根节点识别问题

二叉树输入数据中没有直接指明根节点，需要通过查找没有父节点的节点来确定根节点。使用 hasparent 数组来标记所有有父节点的节点，剩下的就是根节点：

```
//找根节点
for (int i = 0; i < n; i++)
    if (!hasparent[i])
        return i;
return -1;
```

2. **空树处理问题**

当节点数为 0 时，需要特殊处理。在主函数中添加了对空树的判断：

```
bool result = false;
if (n1 == 0 && n2 == 0)
    result = true;
else if (n1 == n2 && n1 > 0)
    result = judge(tree1, root1, tree2, root2);
```

3. 同构判断的递归终止条件

在 judge 函数中需要正确处理各种边界情况：

两棵空树：同构

一棵空一棵不空：不同构

根节点数据不同：不同构

其他情况：递归判断子树

2.2.6 总结和体会

通过本次实现，深入理解了二叉树同构的概念和判断方法。同构判断的核心在于递归比较两棵树的结构，同时考虑左右子树可能互换的情况。

递归算法在这种树结构问题中表现出强大的表达能力。judge 函数通过两种情况的或运算（case1 || case2）简洁地表达了同构的定义：要么直接对应，要么交换后对应。

将二叉树表示为结构体数组并使用索引而非指针，这种设计在处理大规模数据时具有更好的性能和内存管理特性。同时，通过辅助数组寻找根节点的方法也很巧妙，避免了复杂的指针操作。

2.3 题目三 感染二叉树需要的总时间

2.3.1 问题描述

给你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，如果节点满足以下全部条件，就会被感染：节点此前还没有感染；节点与一个已感染节点相邻。返回感染整棵树需要的分钟数。

2.3.2 基本要求

输入格式
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。0 号节点为根节点，0<=start<n。

数据范围
对于 20%的数据，1<=n<=10
对于 40%的数据，1<=n<=1000
对于 100%的数据，1<=n<=100000

输出格式
一个整数，表示感染整棵二叉树所需要的时间。

2.3.3 数据结构设计

本程序采用邻接表来表示二叉树结构，将二叉树视为无向图进行处理，具体数据结构如下：

```
vector<vector<int>> graph(n);
for (int i = 0; i < n; i++) {
    int lchild, rchild;
    cin >> lchild >> rchild;
    if (lchild != -1){
        graph[i].push_back(lchild);
        graph[lchild].push_back(i);
    }
    if (rchild != -1) {
        graph[i].push_back(rchild);
        graph[rchild].push_back(i);
    }
}
```

这是一个二维向量，用于存储图的邻接表表示。其中`graph[i]`存储与节点 i 相邻的所有节点编号。对于二叉树来说，每个节点最多有三个邻居：父节点、左子节点、右子节点。

此外，程序还使用以下辅助数据结构：

```
vector<bool> visited(n, false);  //标记节点是否被访问过
vector<int> distance(n, 0);      //记录每个节点到起始节点的距离
queue<int> q;                    //BFS遍历使用的队列
```

2.3.4 功能说明（函数、类）

（1）广度优先搜索函数

```
void bfs(vector<vector<int>>& graph,int start)
{
    vector<bool> visited(n, false);
    vector<int> distance(n, 0);
    queue<int> q;
    visited[start] = true;
    distance[start] = 0;
    q.push(start);

    int max_time = 0;

    while (!q.empty()) {
        int current = q.front();
        q.pop();
        for (int neighbor : graph[current]) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                distance[neighbor] = distance[current] + 1;
                max_time = max(max_time, distance[neighbor]);
                q.push(neighbor);
            }
        }
    }

    cout << max_time << endl;
}
```

该函数实现标准的 BFS 遍历，从起始节点开始，逐层遍历所有可达节点。使用`distance`数组记录每个节点到起始节点的距离（即感染时间），并在遍历过程中维护最大距离

`max_time`，最终输出这个最大值作为感染整棵树所需的时间。

（2）主函数

```
int main()
{
    cin >> n >> start;
    vector<vector<int>> graph(n);
    for (int i = 0; i < n; i++) {
        int lchild, rchild;
        cin >> lchild >> rchild;
        if (lchild != -1){
            graph[i].push_back(lchild);
            graph[lchild].push_back(i);
        }
        if (rchild != -1) {
            graph[i].push_back(rchild);
            graph[rchild].push_back(i);
        }
    }

    bfs(graph, start);
    return 0;
}
```

主函数负责读取输入数据并构建图的邻接表。对于每个节点的左右孩子，不仅将孩子节点加入当前节点的邻居列表，还将当前节点加入孩子节点的邻居列表，从而构建无向图的连接关系。最后调用 BFS 函数计算并输出结果。

2.3.5 调试分析（遇到的问题和解决方法）

1. 二叉树到无向图的转换问题

最初可能只考虑了父子关系的单向连接，但感染过程是双向的，既可以从父节点感染子节点，也可以从子节点感染父节点。通过构建无向图的邻接表解决了这个问题：

```
if (lchild != -1){
    graph[i].push_back(lchild);
    graph[lchild].push_back(i);  //双向连接
}
```

2.最大感染时间的确定问题

感染整棵树的时间不是最后一个节点被感染的时间，而是所有节点中被感染时间最晚的那个时间。通过维护 `max_time` 变量来跟踪最大距离：

```
max_time = max(max_time, distance[neighbor]);
```

2.3.6 总结和体会

通过本次实现，深入理解了二叉树感染问题的本质。虽然问题描述基于二叉树结构，但实际的感染过程更符合图的传播特性，因为感染可以沿着任意相邻边传播，不仅限于父子关系。

将二叉树重新建模为无向图是解决本题的关键思路。这种思维方式展示了如何将特定结构

的问题转化为更一般的图论问题，从而利用成熟的算法（如 BFS）来解决问题。

BFS 算法在这种传播类问题中表现出色，因为它天然地按照距离（时间）顺序遍历节点，非常适合模拟感染扩散过程。算法中的队列机制确保了节点按照感染先后顺序被处理，distance 数组则准确记录了每个节点的感染时间。

2.4 题目四 树的重构

2.4.1 问题描述

树木在计算机科学中有许多应用。也许最常用的树是二叉树，但也有其他的同样有用的树类型。其中一个例子是有序树（任意节点的子树是有序的）。每个节点的子节点数是可变的，并且数量没有限制。一般而言，有序树由有限节点集合 T 组成，并且满足：1.其中一个节点置为根节点，定义为 root(T); 2.其他节点被划分为若干子集 T1,T2,...Tm,每个子集都是一个树。同样定义 root(T1),root(T2),...root(Tm)为 root(T)的孩子，其中 root(Ti)是第 i 个孩子。节点 root(T1),...root(Tm)是兄弟节点。

通常将一个有序树表示为二叉树是更加有用的，这样每个节点可以存储在相同内存空间中。有序树到二叉树的转化步骤为：1.去除每个节点与其子节点的边；2.对于每一个节点，在它与第一个孩子节点（如果存在）之间添加一条边，作为该节点的左孩子；3.对于每一个节点，在它与下一个兄弟节点（如果存在）之间添加一条边，作为该节点的右孩子。

在大多数情况下，树的深度（从根节点到叶子节点的边数的最大值）都会在转化后增加。这是不希望发生的事情因为很多算法的复杂度都取决于树的深度。现在，需要你实现一个程序来计算转化前后的树的深度。

2.4.2 基本要求

输入格式

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)。输入的截止为以#开始的行。可以假设每棵树至少含有 2 个节点，最多 10000 个节点。

输出格式

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。

2.4.3 数据结构设计

本程序采用两种不同的数据结构来表示转换前后的树结构:

（1） 原始有序树的数据结构

```
//用于记录初始的有序树
typedef struct SqTNode {
    int data;
    int parent = -1;
    vector<int> child; //记录每个节点的所有孩子
}SqTNode;
```

该结构体用于存储原始有序树的节点信息，包含节点的数据、父节点索引以及所有子节点的索引向量。这种设计能够很好地表示多叉树结构，每个节点可以有任意数量的子节点。

（2） 转换后二叉树的数据结构

```
//用于记录重构后的二叉树
typedef struct BiTNode {
    int data;
    int lchild = -1, rchild = -1;
}BiTNode;
```

该结构体用于存储转换后的二叉树节点信息，采用标准的二叉树表示法，包含节点的数据、左孩子索引和右孩子索引。

程序还使用了栈来辅助构建原始树结构：

```
stack<int> stk;
```

以及向量来存储转换后的二叉树：

```
vector<BiTNode> newnode(current_index + 1);
```

2.4.4 功能说明（函数、类）

（1）二叉树深度计算函数

```
int getDepth(vector<BiTNode>& newnode)
{
    if (newnode.empty()) return 0;

    vector<int> depth(newnode.size(), 0);
    int result = 0;

    for (int i = 0; i < newnode.size(); i++) {
        // 计算左孩子深度（深度+1）
        if (newnode[i].lchild != -1) {
            depth[newnode[i].lchild] = depth[i] + 1;
            result = max(result, depth[newnode[i].lchild]);
        }

        // 计算右孩子深度（深度+1）
        if (newnode[i].rchild != -1) {
            depth[newnode[i].rchild] = depth[i] + 1;
            result = max(result, depth[newnode[i].rchild]);
        }
    }

    return result;
}
```

该函数采用迭代方式计算二叉树的深度，通过维护一个深度数组来记录每个节点的深度，并在遍历过程中更新最大深度。

（2）主函数

```
//构造初始有序树，并求深度（前缀和）
stack<int> stk;
int current_node = 0; //记录路径上走过的每一个节点
int current_index = 0; //记录当前入栈的节点编号
SqTNode node[max_points];
stk.push(current_index);
for (int i = 0; i < str.size(); i++) {
    if (str[i] == 'd') {
        sum1++;
        stk.push(++current_index);
        node[current_index].parent = current_node;
        node[current_node].child.push_back(current_index);
        current_node = current_index;
    }
    else if (str[i] == 'u') {
        sum1--;
        stk.pop();
        current_node = stk.top();
    }
    depth1 = max(depth1, sum1);
}
```

首先构造初始的有序树，并且求原始深度，深度优先创建有序树用栈来模拟，sum1 和 depth1 记录最大深度，node 数组记录 dfs 路径上每个节点的父节点和孩子节点。

```
//重构二叉树
vector<BiTNode> newnode(current_index + 1);
for (int i = 0; i <= current_index; i++) {
    newnode[i].data = i;
    if (!node[i].child.empty()) {
        newnode[i].lchild = node[i].child[0];
        for (int j = 0; j < node[i].child.size() - 1; j++)
            newnode[node[i].child[j]].rchild = node[i].child[j + 1];
    }

}
depth2 = getDepth(newnode);
cout << "Tree " << idx << ": " << depth1 << " => " << depth2 << endl;
```

接下来重构成二叉树，对于每个节点，如果有孩子节点，那么第一个孩子节点就是新树的左孩子，如果有兄弟节点，那么前一个节点的右孩子是后一个节点。

最后计算二叉树的深度 depth2.

2.4.5 调试分析（遇到的问题和解决方法）

1. 原始树构建问题

输入序列使用'd'和'u'表示深度优先遍历的路径，需要正确构建出原始的多叉树结构。通过使用栈来跟踪当前路径，当遇到'd'时创建新节点并建立父子关系，当遇到'u'时回溯到父节点。

2. 原始树深度计算问题

原始树的深度可以通过跟踪当前路径长度来计算。使用变量 `sum1` 记录当前深度，遇到'd'

时增加，遇到'u'时减少，并维护最大值。

3. 树转换算法实现问题

将有序树转换为二叉树需要按照特定规则：第一个孩子作为左孩子，后续兄弟作为右孩子。通过遍历每个节点的子节点列表来实现。

2.4.6 总结和体会

通过本次实现，深入理解了有序树与二叉树之间的转换关系以及它们深度特性的变化。有序树到二叉树的转换是一种重要的数据结构转换技术，在实际的计算机系统中有广泛应用。

转换算法的核心思想是将多叉树中节点的第一个孩子作为二叉树的左孩子，将节点的兄弟作为二叉树的右孩子。这种转换虽然改变了树的结构，但保留了原始树的所有信息。

在实现过程中，体会到了栈在树构建过程中的重要作用。通过栈来跟踪深度优先遍历的路径，可以高效地构建出树结构。同时，也认识到对于大规模数据，需要选择适当的算法来避免性能问题，如使用迭代而非递归来计算深度。

2.5 题目五 最近公共祖先

2.5.1 问题描述

给出一颗多叉树，请你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

输入格式

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。

输出格式

对于每一个询问输出一个整数，表示共同祖先的编号。

数据范围

对于 100% 的数据:

1 <= T <= 100

5 <= N <= 1000

5 <= M <= 1000

2.5.3 数据结构设计

由于除了根节点之外，每个节点有且只有一个父节点，所以本程序采用的主要数据结构是一个一维数组 `parent`，用于存储每个节点的父节点信息。具体实现如下:

```
vector<int> parent(Ni + 1, 0); //记录每个点的父节点，下标为当前点，数值为父节点
for (int i = 0; i < Ni - 1; i++) {
    int a, b;
    cin >> a >> b;
    parent[b] = a;
}
```

此外，程序还使用两个动态数组 `qa_path` 和 `qb_path` 分别存储从查询节点 `qa` 和 `qb` 到根节点的路径上经过的所有节点。

```
int qa, qb;
cin >> qa >> qb;  //两个查询节点
vector<int> qa_path, qb_path; //记录查询节点到根节点路径上经过的节点
```

这种设计利用了树结构的层次性，通过回溯父节点的方式构建路径，便于后续比较查找最近公共祖先。

2.5.4 功能说明（函数、类）

首先构建父节点数组 parent，通过 Ni-1 条边的输入，记录每个节点二点父节点。

然后处理每个查询：

从查询节点 qa 开始，依次向上回溯父节点，将路径上所有节点存入 qa_path，直到根节点。同样方式处理 qb，存入 qb_path。

```
qa_path.push_back(qa);
while (1) {
    int father = parent[qa];
    if (father == 0)
        break;
    else {
        qa = father;
        qa_path.push_back(father);
    }
}
qb_path.push_back(qb);
while (1) {
    int father = parent[qb];
    if (father == 0)
        break;
    else {
        qb = father;
        qb_path.push_back(father);
    }
}
```

最后查找最近共同祖先，遍历 `qa_path` 中的每个节点，检查是否出现在 `qb_path` 中，第一个找到的共同节点即为最近共同祖先。

```
//比较找到两条路径中第一个相同的节点
int flag = 0;
int ans = 0;
for (int i = 0; i < qa_path.size(); i++) {
    int cur = qa_path[i];
    for (int j = 0; j < qb_path.size(); j++) {
        if (qb_path[j] == cur) {
            ans = cur;
            flag = 1;
            break;
        }
    }
    if (flag)
        break;
}
cout << ans << endl;
```

2.5.5 调试分析（遇到的问题和解决方法）

--路径构建错误
最初在回溯父节点时，未正确更新当前节点 qa 和 qb，导致路径构建不完整或陷入死循环。通过增加 qa = father 和 qb = father 的赋值操作，确保每次循环向上移动一层。
--根节点处理不当
在构建路径时，若父节点为 0 表示已到达根节点，应终止循环。最初未处理该情况，导致路径中包含非法节点 0，影响结果正确性。

2.5.6 总结和体会

通过本次实现，加深了对树结构及其父节点表示法的理解。利用父节点数组构建路径是一种直观且易于实现的方法，适用于节点数不多的情况。

同时，也认识到当前代码在处理多个查询时存在局限性，未来可考虑预处理整棵树的深度和父节点信息。

2.6 题目六 求树的后序

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。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。输入保证一颗二叉树内不存在相同的节点。

2.6.2 基本要求

输入格式
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。

输出格式
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

本程序采用二叉树结构来存储树形数据，具体数据结构定义如下：

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

该结构体表示二叉树的节点，包含三个字段：

data：存储节点的值，为字符类型

lchild：指向左子节点的指针

rchild：指向右子节点的指针

2.6.4 功能说明（函数、类）

（1）二叉树创建函数

```
BiTree createBiTree(string preorder, string inorder, bool& success)
{
    if (preorder.empty() || inorder.empty()) {
        if (preorder.empty() && inorder.empty()) {
            return NULL;
        }
        success = false;
        return NULL;
    }

    BiTree root = new BiTNode;
    root->data = preorder[0];
    root->lchild = NULL;
    root->rchild = NULL;
```

```
//在中序中找到根节点的位置
auto it = inorder.find(root->data);
if (it == string::npos) {
    success = false;
    delete root;
    return nullptr;
}

//左子树的中序和前序
string left_inorder = inorder.substr(0, it);
string left_preorder = preorder.substr(1, left_inorder.size());

//右子树的中序和前序
string right_inorder = inorder.substr(it + 1);
string right_preorder = preorder.substr(left_inorder.size() + 1);
```

```
//递归构建左子树、右子树
root->lchild = createBiTree(left_preorder, left_inorder, success);
if (!success) {
    delete root;
    return NULL;
}

root->rchild = createBiTree(right_preorder, right_inorder, success);
if (!success) {
    if (root->lchild)
        delete root->lchild;
    delete root;
    return nullptr;
}

return root;
```

先序+中序构造二叉树的算法:

1、从先序遍历序列中取出第一个结点，该结点必为根 结点。然后在中序遍历中找出根结点，其前的序列为左 子树，其后的序列为右子树

2、对左右子树的先序遍历和中序遍历序列再重复第一 步，直到得出所有叶结点为止。

该函数递归地构建二叉树。参数包括前序遍历字符串、中序遍历字符串和一个引用传递的成功标志。函数首先创建根节点（前序遍历的第一个字符），然后在中序遍历中找到根节点的位置，以此分割出左子树和右子树的前序、中序遍历序列，最后递归构建左右子树。

（2）后序遍历输出函数

```
bool postorder(BiTree T)
{
    if (T) {
        if (postorder(T->lchild))
            if (postorder(T->rchild)) {
                cout << T->data;
                return true;
            }
        return false;
    }
    else
        return true;
}
```

该函数采用递归方式实现后序遍历（左-右-根）。当树非空时，先递归遍历左子树，再递归遍历右子树，最后访问根节点并输出节点数据。

（3）内存释放函数

```
// 释放二叉树内存
void deleteTree(BiTree root) {
    if (root) {
        deleteTree(root->lchild);
        deleteTree(root->rchild);
        delete root;
    }
}
```

2.6.5 调试分析（遇到的问题和解决方法）

--子树序列分割问题

在确定左右子树的序列时，需要确保前序和中序序列的长度匹配。通过中序序列中左子树的长度来确定前序序列中左子树的范围，这是算法正确性的关键。

2.6.6 总结和体会

通过本次实现，深入理解了二叉树三种遍历序列之间的关系。前序遍历的第一个节点总是根节点，通过在中序遍历中找到该根节点，可以准确地将树分割为左右子树，这是递归构建二叉树的核心思想。

在实现过程中，体会到了递归思想在树结构处理中的强大作用。通过递归调用，可以优雅地解决复杂的树构建问题。同时，也认识到在涉及动态内存分配的程序中，必须仔细处理各种异常情况，确保内存的正确释放，避免内存泄漏。

2.7 题目七 表达式树

2.7.1 问题描述

任何一个表达式，都可以用一棵表达式树来表示。例如，表达式 a+b*c，可以表示为如下的表达式树:

```
   +
  / \
 a   *
    / \
    b c
```

现在，给你一个中缀表达式，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

输入格式
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
对于 20%的数据，1<=n<=3，1<=x<=5；
对于 40%的数据，1<=n<=5，1<=x<=10；
对于 100%的数据，1<=n<=10，1<=x<=100；

输出格式
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，如样例输出所示。如果该二叉树是一棵满二叉树，则最底部的叶子结点，分别占据横坐标的第 1、3、5、7……个位置（最左边的坐标是 1），然后它们的父结点的横坐标，在两个子结点的中间。如果不是满二叉树，则没有结点的地方，用空格填充（但请略去所有的行末空格）。每一行父结点与子结点中隔开一行，用斜杠（/）与反斜杠（\）来表示树的关系。/出现的横坐标位置为父结点的横坐标偏左一格，\出现的横坐标位置为父结点的横坐标偏右一格。也就是说，如果树高为 m，则输出就有 2m-1 行。第三部分为一个整数，表示将值代入变量之后，该中缀表达式的值。需要注意的一点是，除法代表整除运算，即舍弃小数点后的部分。同时，测试数据保证不会出现除以 0 的现象。

2.7.3 数据结构设计

本程序采用二叉树结构来表示表达式树，具体数据结构定义如下：

```
struct TreeNode {
    char val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(char x) : val(x), left(nullptr), right(nullptr) {}
};
```

该结构体表示表达式树的节点，包含三个字段：val 存储节点的值（运算符或变量），left 指向左子节点的指针，right 指向右子节点的指针。

程序还使用其他辅助数据结构:

stack<char>：用于中缀转后缀时的运算符栈

stack<TreeNode*>：用于从后缀表达式构建表达式树时的节点栈

map<char, int>：用于存储变量名到变量值的映射关系

vector<vector<char>>：用于存储表达式树的图形化输出画布

2.7.4 功能说明（函数、类）

（1） 运算符优先级函数

```
// 运算符优先级
int precedence(char op) {
    if (op == '+' || op == '-') return 1;
    if (op == '*' || op == '/') return 2;
    return 0;
}
```

（2）中缀转后缀函数

```
string infixToPostfix(const string& infix) {
    stack<char> st;
    string postfix;

    for (char c : infix) {
        if (isalpha(c)) {
            // 操作数直接输出
            postfix += c;
        }
        else if (c == '(') {
            // 左括号入栈
            st.push(c);
        }
        else if (c == ')') {
            // 右括号：弹出直到遇到左括号
            while (!st.empty() && st.top() != '(') {
                postfix += st.top();
                st.pop();
            }
            st.pop(); // 弹出左括号
        }
```

该函数实现中缀表达式到后缀表达式（逆波兰式）的转换，使用栈来处理运算符的优先级和括号。

（2） 表达式树构建函数

```
TreeNode* buildTree(const string& postfix) {
    stack<TreeNode*> st;

    for (char c : postfix) {
        if (isalpha(c)) {
            // 操作数：创建叶子节点
            st.push(new TreeNode(c));
        }
        else {
            // 运算符：创建根节点，弹出两个操作数作为左右子树
            TreeNode* node = new TreeNode(c);
            node->right = st.top(); st.pop();
            node->left = st.top(); st.pop();
            st.push(node);
        }
    }
    return st.top();
}
```

（4） 表达式树图形化输出函数

该函数实现表达式树的图形化输出，使用二维画布来布局树的节点和连接线。

2.7.5 调试分析（遇到的问题和解决方法）

1. 中缀转后缀的优先级处理问题

最初在运算符优先级处理上存在问题，特别是括号的处理。通过明确运算符优先级规则和栈的使用顺序，确保转换的正确性。

2. 表达式树构建时的左右子树顺序问题

在构建表达式树时，从栈中弹出节点的顺序很重要。由于后缀表达式是左操作数先出栈，但构建树时需要先弹出右操作数。

3. 实验总结

树和二叉树是典型的非线性、递归结构。

树的每个节点可以有多个孩子，而二叉树每个节点最多有两个孩子（左、右子树）。这种“自我相似”的结构——即一个树由根节点和若干子树构成——天然适合用递归来处理。

递归是操作树结构的核心思想。绝大多数算法，如遍历（先序、中序、后序）、搜索、计算深度等，其本质都是：

基准情形：处理空树或叶子节点。

递归步骤：相信递归函数能正确处理好当前节点的子树，我们只需处理好当前节点，并将子树的处理结果组合起来。

可以说，理解了“将问题分解为更小的相同子问题（子树）”，就掌握了操作树结构的精髓。