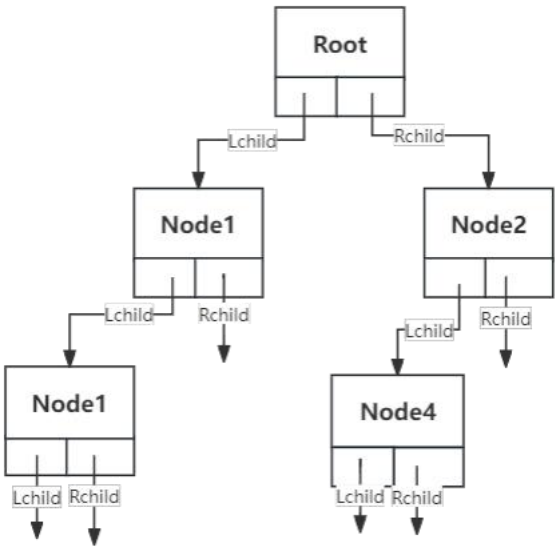
作业 HW3* 实验报告

姓名：朱俊泽 学号：2351114 日期：2024 年 11 月 1 日

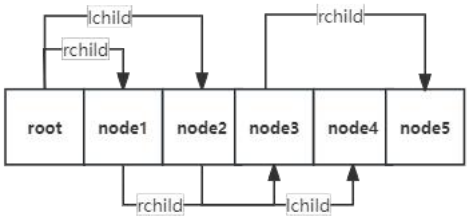
实验报告格式要求按照模板（使用 Markdown 等也请保证报告内包含模板中的要素）
对字体大小、缩进、颜色等不做强制要求（但尽量代码部分和文字内容有一定区分，可参考 vscode 配色）
实验报告要求在文字简洁的同时将内容表示清楚
报告内不要大段贴代码，尽量控制在 20 页以内

1. 涉及数据结构和相关背景

题目或实验涉及数据结构的相关背景
本次实验针对树结构展开，主要内容包括二叉以及多叉树的构建、遍历以及搜索，借助树完成了数据的存储、搜索、排序等任务。
树作为一种非线性数据结构，展现了结点间的从属关系。其具体实现的物理结构主要有两种：
① 链表实现：结点分为数据域和指针域。指针域分别指向 Lchild 左节点和 Rchild 右结点



② 数组实现：借助数组存储数组可以更节约内存空间。也可以利用指针位置来从逻辑上模拟树中子结点和父节点的从属关系。



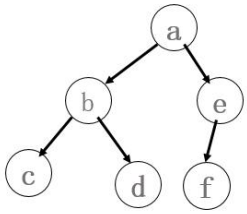
在数据结构的具体实现的时候可以控制 data 的具体内容，然后记得动态开点

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。例如对于由 abc##d##ef###先序建立的二叉树，如下图 1 所示，中序非递归遍历（参照课本 p131 算法 6.3）可以通过如下一系列栈的入栈出栈操作来完成：push(a) push(b) push(c) pop pop push(d)pop pop push(e) push(f) pop pop。



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

构建树结构，利用链表模拟
1) 定义结点类型：
数据域:data；
指针域:lchild / rchild
2) 树结构：定义指向根结点的指针 root，表示指针所指的结构为树：

```
typedef struct TreeNode {//定义树结点的结构
    char data;
    struct TreeNode* lchild;
    struct TreeNode* rchild;
}*Node, Elem;    //结点是TreeNode，指针Node保存结点的地址
typedef struct {//定义树的结构
    Node root;
}Tree;
```

3）生成树 InitTree()

```
int InitTree(Tree* t) {//初始化二叉树
    if (t == NULL)
        return -1;
    Node node = (Node)malloc(sizeof(struct TreeNode));//为结点初始化内存空间
    //定义根节点
    t->root = node;
    //左右指针指向空
    (t->root)->lchild = NULL;
    (t->root)->rchild = NULL;

    return 1;
}
```

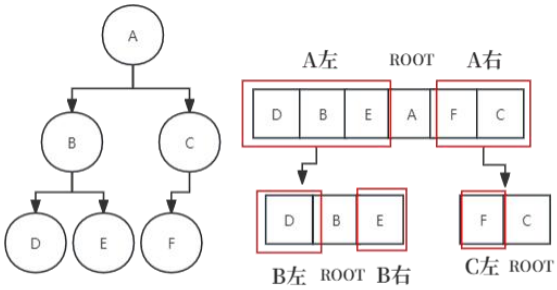
4）生成结点 Makenode()

```
int Makenode(vector<Node> &temp_r,char e,int j) {   //创建新结点，其地址为*node，结点值为e
    if (&temp_r[j] == NULL) {
        return -1;
    }
    if (((temp_r[j]) = (Node)malloc(sizeof(struct TreeNode))) == NULL) {
        return -1;
    }
    (temp_r[j])->lchild = NULL;
    (temp_r[j])->rchild = NULL;
    (temp_r[j])->data = e;
    return 1;
}
```

2.1.4 功能说明（函数、类）

本题需要先通过模拟进出栈，获得中序遍历序列。根据前序和中序遍历序列，可以确定唯一的二叉树，再对二叉树进行非递归方法的后序遍历即可。

② 根据前序和中序序列构建二叉树。
这里采用非递归的方法构建，以下对遍历序列的顺序特点进行分析：
前序遍历序列的顺序是：中->左->右
中序遍历序列的顺序是：左->中->右



对于前序 ABDECF，中序 DBEAFC 的序列，其完整的建树过程如下：
① 前序遍历到 A，作为根结点，直接入栈；
② 前序遍历到 B，栈顶元素 A 与中序指针指向的 D 不同，故 B 连接到栈顶的 A 的左支并入栈。
③ 前序遍历到 D，栈顶元素 B 与中序指针指向的 D 不同，故连接 D 到栈顶的 B 的左支并入栈。
④ 前序遍历到 E，栈顶元素 D 与中序指针指向的 D 相同，D 出栈；中序指针向后移动一位，此时中序指针指向 B，与栈顶元素 B 相同，B 出栈；中序指针向后移动一位，此时中序指针指向 E，与栈顶元素 A 不同，则 E 连接到刚刚出栈的 B 的右支，E 再入栈；
⑤ 前序遍历到 C，栈顶元素 E 与中序指针指向的 E 相同，E 出栈；中序指针向后移动一位，此时中序指针指向 A，与栈顶元素 A 相同，A 出栈；中序指针向后移动一位，此时中序指针指向 F，栈为空，则 C 连接到刚刚出栈的 A 的右支，C 再入栈；
⑥ 前序遍历到 F，栈顶元素 C 与中序指针指向的 F 不同，故 F 连接到栈顶的 C 的左支并入栈；
⑦ 此时前序遍历结束，建树完成。

```
while (i <= preorder.size() - 1) {
    //在中序序列中，比较当前遍历位置与栈顶（即上一个先序遍历元素的关系）
    if (midorder[j] == st.top()->data) {
        t.Makenode(temp_r, preorder[i], i);//新建待连接的右节点，由于会新建很多右节点，可以开一个链表来单独存放他们的信息
        k = j;//保护一下j，因为后面还要用到
        while (1) {  //在树中从下向上，寻找该右结点应当连接的父节点
            if (!st.empty()&&midorder[k] == st.top()->data){
                tmp_linked = st.top();//tmp_linked存放了最后出栈结点
                st.pop();
            }
            else
                break;
            ++k;
        }
        (tmp_linked)->rchild = temp_r[i];//连接到右节点上
        st.push(temp_r[i]);//不要忘了每遍历一个就要入一个的栈，这里是将遍历得到的新连接右节点入栈
        j = k;
    }
    else if (midorder[j] != st.top()->data) {
        t.Makenode(temp_l, preorder[i], i);//新建待连接的左节点，存放元素是先序序列当前指向的元素，同样也会新建很多左节点，故要用链表来分别存放
        tmp_linked = st.top();
        (tmp_linked)->lchild = temp_l[i];//左节点直接连到上一个元素
        st.push(temp_l[i]);
    }
    else;
    ++i;
}
```

由于对一棵树的访问方式是自上而下的，而后序遍历本质是一种自下而上的遍历方式，因此需要借助一个栈，先自上而下入栈访问，再依次出栈，从而实现自下而上的后序形式输出。
这里要注意，由于入栈顺序是与出栈相反的，因此入栈时遵循的是中->左->右，因此入栈时，对一个结点先入栈右支结点、再入栈左支结点。
非递归方式的后序遍历为 PostTraverse().

```
void PostTraverse(Tree tree, stack<Elem> st, vector<char>& postorder) {
    Elem postorder_elem;

    st.push(*((&tree)->root));//将首节点存到栈中
    while (st.empty() != 1) {
        postorder_elem = st.top();
        st.pop();
        postorder.push_back(postorder_elem.data);
        if (postorder_elem.lchild != NULL) {
            st.push( *(postorder_elem.lchild));
        }
        if (postorder_elem.rchild != NULL) {
            st.push( *(postorder_elem.rchild));
        }
    }
}
```

2.1.5 调试分析（遇到的问题和解决方法）

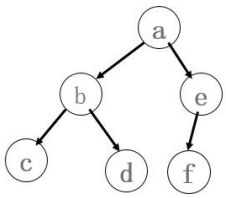
2.1.6 总结和体会

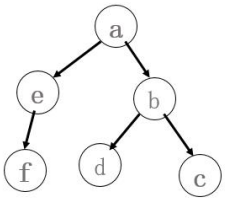
本题主要用到了树的前序、中序、后序三种遍历的知识。在借助前序、中序方法构建树的时候，需要充分考虑其特点与关系，借助栈来保存对已遍历过的结点的返回访问。另外该过程用递归的方法，通过分治的思想也可以实现树的构造，该迭代方法也是对递归过程的模拟。
此外，对非递归方法获取后序遍历序列时，实际上是自上而下访问入栈后再自下而上出栈，从而得到左->右->中的后序遍历结果，用到了栈先进后出的性质

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。下图两个就是同构的





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。

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。

下载 p81_data.cpp 并编译运行以生成随机测试数据

输出:

共三行。

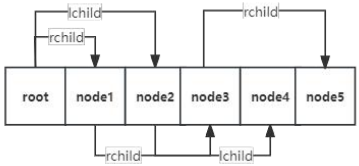
第一行，如果两棵树是同构的，输出“Yes”，否则输出“No”。

后面两行分别是两棵树的深度。

2.2.3 数据结构设计

(1) 定义数组形式的树：存贮好需要使用到的节点值，左儿子右儿子。

```
// 定义树节点结构体
struct TreeNodeStruct {
    char value;
    int leftIndex, rightIndex;
};
```



（2）实例化：根据动态开点大小的需求开一个数组形式的树去使用

```
cin >> nodeCount1;

vector<TreeNodeStruct> tree1(nodeCount1);

cin >> nodeCount2;

vector<TreeNodeStruct> tree2(nodeCount2);
```

2.2.4 函数设计

本题有三个功能需要实现：寻找节点，寻找深度，判断同构

（1）首先需要寻找两棵树的 root 节点，在观察输入数据和树的结构得到的规律：没有节点指向的那个节点就是树的根节点：根据这个特点：首先创建一个 isChildNode 布尔向量，初始化为 false，用于标记每个节点是否是其他节点的子节点。遍历 treeNodes，对于每个节点，如果它有左子节点或右子节点，则在 isChildNode 中对应位置标记为 true。最后，遍历 isChildNode，找到第一个未被标记为子节点的索引，这个索引就是根节点的索引如果所有节点都被标记为子节点，说明输入有误，返回-1。

```
// 寻找树的根节点
int getRootIndex(int nodeCount, const vector<TreeNodeStruct>& treeNodes) {
    vector<bool> isChildNode(nodeCount, false);
    for (const auto& node : treeNodes) {
        if (node.leftIndex != -1) isChildNode[node.leftIndex] = true;
        if (node.rightIndex != -1) isChildNode[node.rightIndex] = true;
    }
    for (int i = 0; i < nodeCount; ++i) {
        if (!isChildNode[i]) return i;
    }
    return -1;
}
```

（2）其次实现递归寻找深度，递归计算左子树和右子树的深度，树的深度为 1 加上左右子树深度的最大值。注意这个+1。

```
int computeTreeDepth(int rootNodeIndex, const vector<TreeNodeStruct>& treeNodes) {
    if (rootNodeIndex == -1) return 0;
    int leftDepth = computeTreeDepth(treeNodes[rootNodeIndex].leftIndex, treeNodes);
    int rightDepth = computeTreeDepth(treeNodes[rootNodeIndex].rightIndex, treeNodes);
    return 1 + max(leftDepth, rightDepth);
}
```

（3）最后判断同构问题，需要注意的是，题目里面提到了一个点“**不保证测试点每个节点数据不同**”所以需要注意是否出现这样一个情况两棵树从某一个节点开始出现两个相同的节点，然后我们没有正确对齐导致的错误，所以需要遍历一次，就是左左右右，左右左右的判断，交换一次判断，就是代码里面的 swapping 内容：然后递归到-1 节点就是叶子节点即可。

```
// 暂时满足，往下递归
bool flag = true;

if (n1l == n2l && n1r == n2r)
{
    if (t1[node1].leftIndex != -1) // 修改 leftson 为 leftIndex
    {
        flag = flag && make_cmp(t1, t1[node1].leftIndex, t2, t2[node2].leftIndex); // 修改 leftson 为 leftIndex
    }
    if (t1[node1].rightIndex != -1) // 修改 rightson 为 rightIndex
    {
        flag = flag && make_cmp(t1, t1[node1].rightIndex, t2, t2[node2].rightIndex); // 修改 rightson 为 rightIndex
    }
}
if (n1l == n2r && n1r == n2l)
{
    if (t1[node1].leftIndex != -1) // 修改 leftson 为 leftIndex
    {
        flag = flag && make_cmp(t1, t1[node1].leftIndex, t2, t2[node2].rightIndex); // 修改 leftson 为 leftIndex
    }
    if (t1[node1].rightIndex != -1) // 修改 rightson 为 rightIndex
    {
        flag = flag && make_cmp(t1, t1[node1].rightIndex, t2, t2[node2].leftIndex); // 修改 rightson 为 rightIndex
    }
}
return flag;
```

2.2.5 题目心得

本题主要考察了递归思维，还要注意数组形式模拟的树，然后最重要的一点就是注意递归的写法，来控制不能爆内存

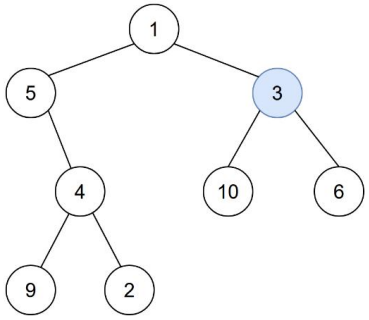
2.3 题目三 感染二叉树需要的总时间

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

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

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

下载编译并运行 p128_data.cpp 生成随机测试数据

输出

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

（1）树形结构：构建一个合适的邻接表来表示树形结构即可

```
typedef pair<int, int> pi;
const int N = 1e5 + 10;
vector<int> g[N];
```

（2）其他信息存储：还需要存贮一个是否走过的点的标记 vis

```
bool visited[N] = {false};
```

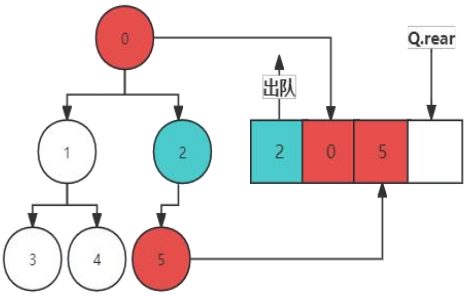
2.3.4 函数设计

一个和第二题相似度很高的广度优先搜索 bfs 问题，不过本题目中根节点自己定义，然后这个树是双向边，注意给的 start 作为根节点就好，和第二题方法差不多。

```
void bfs(int start) {
    visited[start] = true;
    queue<pi> q;
    q.push({start, 0});

    while (!q.empty()) {
        pi current = q.front();
        q.pop();
        int node = current.first;
        int distance = current.second;

        for (int neighbor : g[node]) {
            if (!visited[neighbor]) {
                q.push({neighbor, distance + 1});
                res = max(res, distance + 1);
                visited[neighbor] = true;
            }
        }
    }
}
```



2.3.5 题目心得

考查了一个标准的 bfs 广度优先搜索问题，这里可以使用邻接表而不是只用数组模拟的树形结构

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。

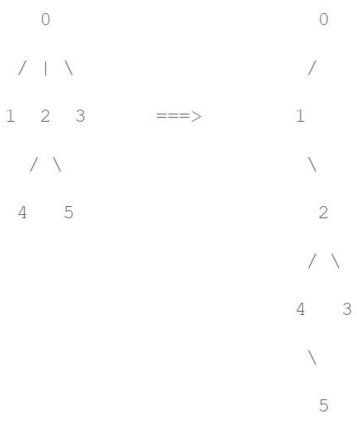
通常将一个有序树表示为二叉树是更加有用的，这样每个节点可以存储在相同内存空间中。有序树到二叉树的转化步骤为：

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

如图所示：



2.4.2 基本要求

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)。
例如上面的树就是 dudduduudu,表示从 0 下行到 1,1 上行到 0,0 下行到 2 等等。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。

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

本题用到了三种数据结构：栈、队列以及树。
其中栈可以通过模拟深度优先搜索过程来构建树；队列用于存储层序遍历树时遍历过的结点信息。

其中构建栈、队列方式与问题一中相同。这里的树结构为多叉树，即指针域是由链表构成。同时该过程中还需要特别设置一个 rchild 指针，用于连接树重构过程中新连接的结点。

多叉树的构建：

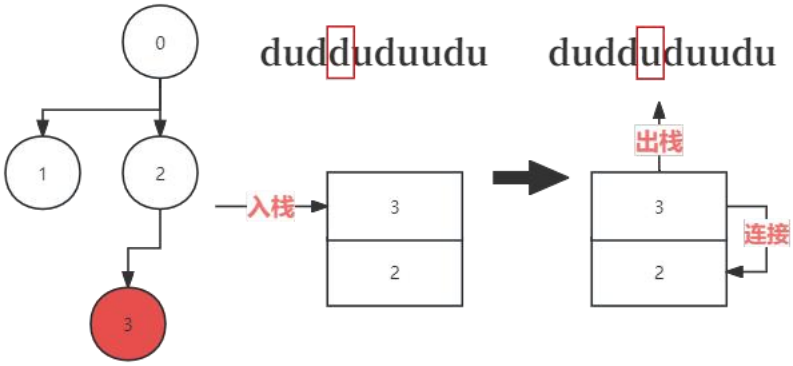
```
struct TreeNode {//定义树结点的结构
    int data;
    TreeNode *children[100];  //children为数组，存放各个子节点的地址
    TreeNode* rchild=nullptr;
    int vec_len = 0;
};    //结点是TreeNode，指针Node保存结点的地址
typedef TreeNode Elem;
```

主要还是遍历树的方法 dfs 函数和重构的函数

2.4.4 函数设计

本题共分为三步：根据深度搜索序列构建树、将多叉树重构为二叉树、层序遍历求解树的深度。

(1) 先 dfs 构建树：先按照 d 模拟走到底



（2）再进行重构：出队一个结点（初始化时入队根节点），对其各子节点进行操作对父节点的第一个子节点，将其近邻的第一个兄弟（若存在）结点连接为自己的右孩子。入队保存，以待下一次遍历。对父节点的其他子节点，将其近邻的第一个兄弟结点（若存在）连接为自己的右孩子，同时断开其与父节点的连接。入队保存，以待下一次遍历。

```
queue.push(tree[0]);//根节点入队
while (queue.empty() != 1) {
    tmp_node = queue.front();
    queue.pop();

    //遍历子节点
    tmp = tmp_node.data;

    if (tmp_node.vec_len >= 1) {
        //先让子节点入队
        for (int i = 0; i < tree[tmp].vec_len; i++) {
            queue.push(*tree[tmp].children[i]);
        }
        for (int i = 0; i < tree[tmp].vec_len; i++) {
            if (i == 0) {   //对第一个子结点，只需要在新的右枝上连接其近邻的兄弟结点即可
                if (i != tree[tmp].vec_len-1) {    //首先要保证子节点有兄弟结点
                    tree[tmp].children[i]->rchild = tree[tmp].children[i + 1];
                }
            }
            else {        //对其他结点，除了需要让该子节点连接近邻兄弟节点外，还需要父节点自己与该子节点断开连接
                if (i != tree[tmp].vec_len - 1) {    //首先要保证子节点有兄弟结点
                    tree[tmp].children[i]->rchild = tree[tmp].children[i + 1];
                    for (int j = i; j <= tree[tmp].vec_len - 2; j++) {
                        tree[tmp].children[j] = tree[tmp].children[j + 1];
                    }
                }
                --tree[tmp].vec_len;//断开连接，子节点数量少1
                --i;
            }
        }
```

（3）最后和之前一样做一个 bfs 找深度即可，就不多展示代码了，节省字数

2.4.5 题目心得

本题主要涉及了 **DFS 算法**和 BFS 算法，体会到了两者实现思路的区别以及各自的应用。主要是实现一次 DFS 遍历

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

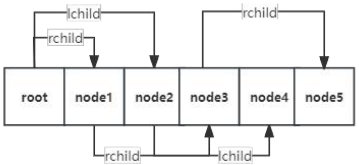
输入数据包含 T 个测试样本，每个样本 i 包含 Ni 个节点和 Ni-1 条边和 Mi 个问题，树中节点从 1 到 Ni 编号
输入第一行是测试样本数 T
每个测试样本 i 第一行为两个整数 Ni 和 Mi
接下来 Ni-1 行，每行 2 个整数 a、b，表示 a 是 b 的父节点
接下来 Mi 行，每行两个整数 x、y，表示询问 x 和 y 的共同祖先
对于 100%的数据，
1<=T<=100;
5<=N<=1000;
5<=M<=1000;

输出

对于每一个询问输出一个整数，表示共同祖先的编号

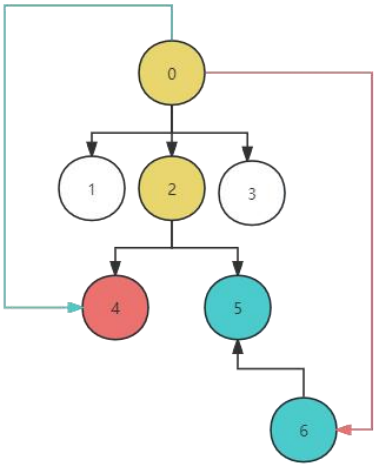
2.5.3 数据结构设计

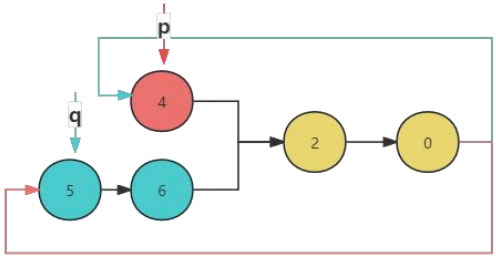
本题借助数组模拟了二叉树结构，然后基本的解决一个 LCA 问题



2.5.4 函数设计

任意两个子结点均有共同祖先，本题重点在于如何寻找到最近的共同祖先。而该问题可以转化为求解相交链表的第一个交点的位置





如左图，对于 4、6 两个结点，分别定义两个遍历指针，同时自下向上遍历。该问题即等价于相交链表寻找第一个交点的问题。其方法是：使双指针 p、q 分别从两个带判断的结点开始自下而上遍历，直到重合；如果两指针一直未重合，则遍历到根节点后，p 指针返回到 q 指针的出发位置，q 指针返回到 p 指针的出发位置，进行循环重复遍历，这时必然会发生重合，返回此时指针的位置，即为第一个相交结点的位置。

```
int FindCommonAncestor(TreeNode Tree[], int root, int a, int b) {
    int p = a;
    int q = b;
    if (p == q) {
        return p;
    } else {
        while (true) {
            if (p != root)
                p = Tree[p].parent;
            else
                p = Tree[p].back_a;
            if (q != root)
                q = Tree[q].parent;
            else
                q = Tree[q].back_b;
            if (p == q)
                break;
        }
    }
    return p;
}
```

2.5.5 题目心得

本题通过观察形式结构，将问题类比为相交链表寻找第一个交点的问题。解题中使用了经典的双指针方法。

2.6 题目六 求树的后序

2.6.1 问题描述

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
输入保证一颗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

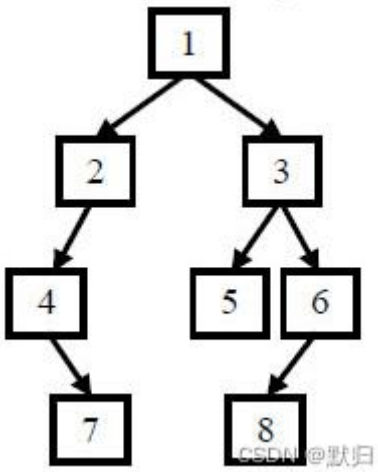
2.6.3 数据结构设计

本题利用链表结构存储树。递归方法求后序。

```
struct Node
{
    char data;
    Node * lchild;
    Node * rchild;
};
```

2.6.4 函数设计

当我们已知一颗二叉树的前序、中序遍历，或者是中序、后序遍历的结果时，可以重建出该二叉树，而已知前序、后序遍历则无法重建。因此，想要根据二叉树的前序、中序遍历，求得它的后序遍历，只需先将二叉树重建出来，再对其进行后序遍历即可。



问题的关键在于如何根据二叉树的前序、中序遍历重建出二叉树。这里继续以上文的二叉树为例，它的前序遍历为：1 2 4 7 3 5 6 8、中序遍历为：4 7 2 1 5 3 8 6。注意前序遍历的顺序为根左右，因此它的第一个元素（1）即为二叉树的根结点。知道根结点后，在中序遍历的结果中找到它的位置，中序遍历的顺序为左根右，因此在它左侧的结点即为根结点的左子树，在它右侧的结点即为根结点的右子树，对于子树再进行相同的分析即可。完整过程示例如下：

1 为根结点，在中序遍历中左侧节点有：4 7 2，为它的左子树、右侧结点有：5 3 8 6，为它的右子树；

对于根结点的左子树 4 7 2，前序遍历顺序为 2 4 7，因此它的根结点为 2，左子树为 4 7，无右子树；

对于左子树 4 7，前序遍历顺序为 4 7，因此它的根结点为 4，无左子树，右子树为 7；

对于右子树 7，它无左右子树，是叶子结点。

对于根结点的右子树 5 3 8 6，前序遍历顺序为 3 5 6 8，因此它的根结点为 3，左子树为 5，右子树为 8 6；

对于左子树 5，它无左右子树，是叶子结点；

对于右子树 8 6，前序遍历顺序为 6 8，因此它的根结点为 6，左子树为 8，无右子树；

对于左子树 8，它无左右子树，是叶子结点。

```
bool vis = true;
Node* CreateTree(string pre, string in)
{
    Node * root = NULL;  //树的初始化
    if(pre.length() > 0)
    {
        root = new Node;  //为根结点申请结构体所需要的内存
        root->data = pre[0]; //先序序列的第一个元素为根结点
        int index = in.find(root->data);  //查找中序序列中的根结点位置
        // 判断是否能够正确构建左右子树
        if (index == string::npos || index >= pre.length()) {
            //cout << "Error" << endl;
            //exit(0);
            vis = false;
            return nullptr;
        }
        root->lchild = CreateTree(pre.substr(1, index), in.substr(0, index));  //递归创建左子树
        root->rchild = CreateTree(pre.substr(index + 1), in.substr(index + 1)); //递归创建右子树
    }
    return root;
}
```

模拟上述过程确定一个前序和中序的树，判断一个问题就是不能建树，就是建树方式两个不一样的时候，把全局的 vis 修改一下，主函数里面判断输出 Error 就行

```
void PostOrder(Node * root)  //递归后序遍历
{
    if(root != NULL)
    {
        PostOrder(root->lchild);
        PostOrder(root->rchild);
        cout << root->data;
    }
}
```

最后递归输出后序遍历

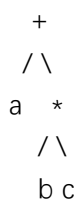
2.6.5 题目心得

本题在通过前序和中序遍历序列构建树的过程中使用了递归的方法，相比迭代去用栈执行该流程，递归算法更加直观且符合逻辑，编写的代码也更加简洁。但是会占用更多的运行空间。两种方法各有优劣，适用于不同的需求场景

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：



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。
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；
下载编译并运行 p129_data.cpp 生成随机测试数据

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

和之前几题的差别不大，存好左右儿子节点和值

```
struct Node {
    char val;
    Node* l;
    Node* r;

    Node(char v) : val(v), l(nullptr), r(nullptr) {}
};
```

2.7.4 函数设计

功能：返回操作符 op 的优先级。+和-优先级为 1，*和/优先级为 2。

```
int getPrec(char op) {
    switch (op) {
        case '+':
        case '-':
            return 1;
        case '*':
        case '/':
            return 2;
        default:
            return 0;
    }
}
```

（1）将中缀表达式转换为后缀表达式（也称为逆波兰表达式）

```
string inToPost(const string& infix) {
    stack<char> opStack;
    string postfix;
    for (char c : infix) {
        if (isalpha(c)) {
            postfix += c;
        } else if (c == '(') {
            opStack.push(c);
        } else if (c == ')') {
            while (!opStack.empty() && opStack.top() != '(') {
                postfix += opStack.top();
                opStack.pop();
            }
            opStack.pop(); // pop '('
        } else if (isOp(c)) {
            while (!opStack.empty() && getPrec(opStack.top()) >= getPrec(c)) {
                postfix += opStack.top();
                opStack.pop();
            }
            opStack.push(c);
        }
    }
    while (!opStack.empty()) {
        postfix += opStack.top();
        opStack.pop();
    }
    return postfix;
}
```

遍历中缀表达式的每个字符 c。

如果 c 是字母（操作数），直接添加到 postfix 字符串中。
如果 c 是左括号 '('，将其推入 opStack 栈中。
如果 c 是右括号 ')'，则：
从 opStack 中弹出元素并添加到 postfix，直到遇到左括号 '('。
弹出左括号（但不添加到 postfix）。
如果 c 是操作符（通过 isOp(c) 判断），则：出 opStack 中所有优先级大于或等于 c 的操作符，并将它们添加到 postfix 中。
将 c 推入 opStack 中。
遍历结束后，将 opStack 中剩余的所有操作符添加到 postfix 中。
返回构建的后缀表达式 postfix。

（2）然后转为二叉表达式树

```
Node* buildTree(const string& postfix) {
    stack<Node*> nodeStack;
    for (char c : postfix) {
        Node* newNode = new Node(c);
        if (isOp(c)) {
            newNode->r = nodeStack.top(); nodeStack.pop();
            newNode->l = nodeStack.top(); nodeStack.pop();
        }
        nodeStack.push(newNode);
    }
    return nodeStack.top();
}
```

遍历后缀表达式的每个字符 c。
为字符 c 创建一个新的节点 newNode。
如果 c 是操作符 (通过 isOp(c) 判断) , 则从 nodeStack 弹出两个节点, 分别作为 newNode 的右子节点和左子节点。
注意：后缀表达式的性质保证了先弹出的节点是右子节点，后弹出的节点是左子节点。
将 newNode 推入 nodeStack 中。
遍历结束后，nodeStack 中应该只剩下一个节点，它就是整棵表达式树的根节点。
返回根节点。

（3）表达式树的可视化

```cpp
void printTree(Node* root, int lvl, int pos, int maxHt, vector<string>& out) {
    if (!root || lvl >= 2 * maxHt - 1) return;
    out[lvl][pos] = root->val;
    if (root->l) {
        out[lvl + 1][pos - 1] = '/';
        printTree(root->l, lvl + 2, pos - (1 << (maxHt - lvl / 2 - 2)), maxHt, out);
    }
    if (root->r) {
        out[lvl + 1][pos + 1] = '\\';
        printTree(root->r, lvl + 2, pos + (1 << (maxHt - lvl / 2 - 2)), maxHt, out);
    }
}
```

2.7.5 题目心得

对于较大的树结构，实现可视化的输出（如树的层次遍历打印、图形化显示等）有助于理解树的结构和调试程序。这种方式能够直观地展示树的组织方式和节点间的关系

对特殊树结构的可视化可以使用到普通的树里面，帮助 debug

3. 实验总结

本次实验主要聚焦于树结构来完成。但是对于树的遍历、搜索等问题，常常需要栈和队列来存储遍历的结点，因此对于一道题目经常需要同时构建多种数据结构来完成。

此外，本次题目主要考察了树的三种遍历方式，对于各类遍历方法都可以有不同的方式来实现（递归与非递归）。同时注意到，根据前序和中序遍历序列可以唯一的二叉树，这需要深刻理解遍历顺序。

题目中构建的多为顺序树，这种情况下用数组来模拟树的存储结构会更加方便，也会占用更少的内存。如问题二 二叉树感染时间、问题四 最近公共结点、问题三 树的重构，其只用到了各个结点间的连接逻辑关系，并没有真正用到树结构的性质，也只需要在此基础上进行简单的 BFS 遍历即可，因此对于这类题目采用数组模拟会有更大的优势。

此次题目中对于算法的考察同样值得关注。BFS 与 DFS 在对树的整颗遍历中具有重要作用，尤其是 BFS，由于可以自上而下根据连接顺序逐层遍历树，则在对树的重构修改上具有重要应用，通过此方法，可以实现对多叉树的二叉重构、镜像翻转等操作。

在本次实验中还涉及到了多叉树的构建。多叉树有多种表示方法，常见的有多指针孩子表示法、孩子兄弟表示法以及双亲表示法。三种表示方法适用于不同任务场景。如在树的重构中，虽然使用父亲兄弟结点法会更方便重构，但是用该表示方法通过 DFS 序列构建起来会十分困难，因此在解题中仍然选择了多指针孩子表示法。