作业 HW3 实验报告

姓名：刘彦 学号：2352018 日期：2024 年 11 月 9 日

1. 涉及数据结构和相关背景

树和二叉树是计算机科学中常见的数据结构之一，广泛用于数据存储和检索。这些结构的特点使得它们在许多应用场景中具有优势，尤其是在需要快速查找、插入和删除数据的场合。

1.1 树的基本概念

树是一种非线性数据结构，由节点和连接节点的边组成，具有以下特点：

- 层次结构：树的节点层次分明，最上层的节点称为根节点。
- 节点关系：每个节点可以有多个子节点，但只有一个父节点（除根节点外）。
- 无环性：树中没有环，即节点之间不会形成循环。

常见的树结构包括：

- 二叉树（Binary Tree）：每个节点最多有两个子节点，分别是左子节点和右子节点。
- 多叉树（M-ary Tree）：每个节点可以有多个子节点，具体取决于 M 的大小。

1.2 二叉树的分类

- 普通二叉树：每个节点有 0 到 2 个子节点，没有其他约束条件。
- 完全二叉树：从根节点开始，逐层填满节点，每层节点从左向右排列。倒数第二层必须全满，最后一层节点尽量靠左。
- 满二叉树：每层节点都是满的，所有叶子节点都在同一层。

1.3 树和二叉树的操作

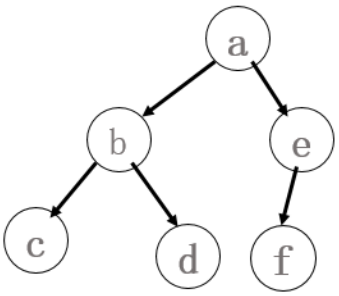
- 遍历：包括前序遍历、中序遍历、后序遍历和层序遍历。在二叉树中，不同的遍历方式可以用于不同的算法和应用场景，例如树的结构输出和树的表达式计算。
- 插入与删除：根据树的定义和结构的不同，插入和删除操作的复杂度和实现方式也不一样。例如，在二叉搜索树中插入一个值需要找到合适的位置，使得树仍然满足排序性。

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。



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：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：一行，后序遍历序列。

2.1.3 数据结构设计

```
struct Node {
    char data;          // 节点值
    Node* left_p;       // 左子树指针
    Node* right_p;      // 右子树指针
    Node(char d) {
        data = d;
        left_p = nullptr;   right_p = nullptr;
    }
};
// 链表节点结构体
struct StackNode {
    Node* data;         // 存储的树节点
    StackNode* next;    // 指向下一个节点
    StackNode(Node* d) {
        data = d;
        next = nullptr;
    }
};
// 栈结构体
struct Stack {
    StackNode* top; // 栈顶指针
    Stack() {
```

```
        top = nullptr; // 初始化栈为空
    }
    // 入栈操作
    void push(Node* node) {
        StackNode* newNode = new StackNode(node); // 创建新栈节点
        newNode->next = top; // 将新节点指向当前栈顶
        top = newNode; // 更新栈顶
    }
    // 出栈操作
    Node* pop() {
        if (top == nullptr) return nullptr; // 如果栈为空，返回 nullptr
        StackNode* temp = top; // 临时保存栈顶节点
        Node* result = temp->data; // 获取栈顶节点数据
        top = top->next; // 更新栈顶
        delete temp; // 释放旧栈顶节点内存
        return result; // 返回栈顶节点数据
    }
    bool isEmpty();// 判断栈是否为空
};
```

2.1.4 功能说明（函数、类）

构建树的逻辑的函数

```
/**
 * @brief           根据输入构建树的逻辑
 * @param n         节点的数量
 * @param root      树的根节点
 */
void my_tree(int n, Node* root) {
    Stack stack; // 创建栈实例
    Node* pointer = root;
    bool left = true;
    while (n) {
        char opt[5]; // 字符数组来存储操作
        cin >> opt; // 读取操作
        if (是pop) {
            if (left) left = false; // 仅在左子树未遍历时 pop
            else {
                if (stack.isEmpty()) continue;
                pointer = stack.pop(); // 回到上一个未遍历的节点
            }
        } else { // 处理 push 操作
            char dt;   cin >> dt; // 读取节点值
            stack.push(pointer); // 保存当前节点到栈
            if (left) {
```

```
                pointer->left_p = new Node(dt); // 插入左子节点
                pointer = pointer->left_p; // 移动到左子节点
            } else {
                插入右子节点，移动到右子节点（类似上面左子节点）
                stack.pop(); // 当前节点已满，回退
            }
            left = true; // 允许插入下一个节点
            n--;
        }
    }
}
```

后序遍历函数

```
/**
 * @brief           后序遍历函数
 * @param root      树的根节点
 */
void back_traverse(Node* root) {
    if (root) {
        back_traverse(root->left_p); // 递归访问左子树
        back_traverse(root->right_p); // 递归访问右子树
        cout << root->data; // 输出当前节点的值
    }
}
```

2.1.5 调试分析（遇到的问题和解决方法）

2.1.5.1 树的构建逻辑错误

一开始在处理栈操作时，pop 的逻辑可能导致栈中节点的丢失，尤其是在连续的 push 和 pop 操作中。树节点的链接关系可能不正确，导致树结构异常。最终我在遍历的过程中，仔细检查了每个操作的条件判断。确保在每次 pop 操作后，只有在实际需要回退时才进行 pop，避免无效的操作导致栈中的节点丢失。

2.1.5.2 插入顺序错误

在构建树时，如果左右节点的插入顺序颠倒，树的结构会错误。后序遍历特性要求在插入过程中优先插入左子节点，然后再插入右子节点。一开始我的程序的插入顺序错误导致树的形状不符合预期。对此，我确保在树的构建过程中，始终遵循先左后右的顺序进行插入。你可以通过调整插入逻辑，确保在处理节点时先检查左子树，再处理右子树。

2.1.6 总结和体会

通过本题，我知道了对于小规模数据，使用线性表来模拟树结构是一种有效且简单的实现方式。可以定义一个结构体来表示树的节点，其中包括数据和左右子节点的索引。这样，可以通过数组的索引来表示节点之间的关系。

本题通过构建二叉树，强化了我对深度优先搜索的理解。DFS 的特性使我能够在树的遍历中有效地管理当前节点的状态，并确保对每个子树的完整遍历。理解 DFS 的过程对于解决类似问题是非常有帮助的。另外，我知道了在栈操作中需要仔细考虑栈是否为空，以及在进行节点插入时，需确认当前节点的左右子树状态。

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。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。

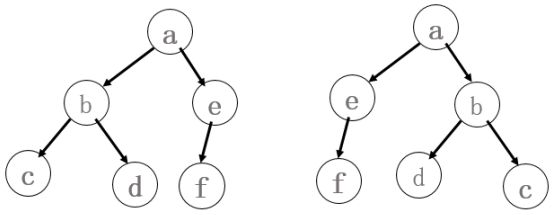


图 1

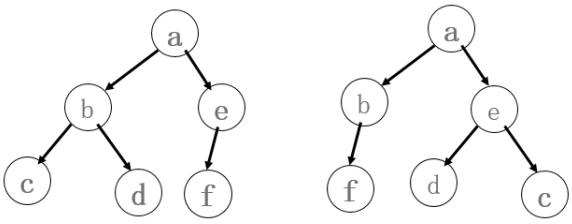


图 2

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。

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: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

```
class Tree {
public:
    struct Node { // 节点结构体
        char name; // 节点的名称（一个字符）
        int l, r, p; // 左子节点索引、右子节点索引、父节点索引
        Node() { // 将左子节点、右子节点和父节点初始化为 -1，表示不存在
            p = l = r = -1;
```

```
        }
    };
    Tree();// 默认构造函数：初始化树为空
    Tree(int n); // 带参数的构造函数：初始化一个有 n 个节点的树
    ~Tree();// 析构函数：释放树的资源
    void Clear();// 清空树，释放所有节点内存
    int InputTree();// 输入树的结构（节点名称和左右子节点的索引）
    int Root() const; // 获取树的根节点索引
    int Size() const; // 获取树的节点数量
    int Depth() const; // 获取树的深度
    // 比较两棵树是否同构
    static bool tree_compare(const Tree &t1, const Tree &t2);
    void Print() const; // 打印树的结构：输出每个节点的信息
private:
    int size, root; // 树的节点数量和根节点的索引
    Node *list; // 指向树的节点数组
    int FindRoot();// 查找树的根节点
    int DepthDFS(int i) const; // 使用递归计算树的深度
    // 静态方法：递归比较两棵树的结构是否相同
    static bool tree_compareDFS(const Tree &t1, const Tree &t2, int i1, int
i2);
};
```

2.2.4 功能说明（函数、类）

带参数的构造函数

```
/**
 * @brief           树的构造函数，初始化树结构
 * @param n         节点的数量
 */
Tree::Tree(int n) {
    size = n;
    list = (Node *)malloc(n * sizeof(Node)); // 分配内存给节点数组
    for (i 从 0 到 n) 在 list[i] 地址上构造一个新的 Node 对象
}
```

输入树的结构的函数

```
/**
 * @brief           输入树的节点信息，并构建树结构
 * @return          返回根节点的索引
 */
int Tree::InputTree() {
    for (i 从 0 到 size) {
        char name;  char s1[10], s2[10]; // 假设最大长度为 10
        cin >> name >> s1 >> s2; // 从输入读取节点信息
        设置节点 list[i].name 名称
```

```
        转换左右子节点 list[i].l 和 list[i].r 索引
        // 更新子节点的父节点索引
        if (list[i].r >= 0) list[list[i].r].p = i;
        if (list[i].l >= 0) list[list[i].l].p = i;
    }
    return root = FindRoot(); // 查找根节点并设置根节点索引
}
```

查找根节点的函数

```
/**
 * @brief           查找树的根节点索引
 * @return          返回根节点的索引，如果找不到则返回 -1
 */
int Tree::FindRoot() {
    分配内存给布尔数组*bo，用于标记访问过的节点，并初始化布尔数组
    int i = 0; // 从第一个节点开始
    while (list[i].p != -1 && !bo[list[i].p]) { // 迭代寻找父节点
        i = list[i].p; // 移动到父节点  bo[i] = true; // 标记父节点已访问
    }
    free(bo); // 释放布尔数组的内存
    return list[i].p == -1 ? i : -1; //找到了根节点，返回根节点索引；否则返回-1
}
```

深度优先搜索计算深度的函数

```
/**
 * @brief           深度优先搜索（DFS）计算树的深度
 * @param i         当前节点的索引
 * @return          返回树的深度
 */
int Tree::DepthDFS(int i) const {
    if (i == -1)  return 0; // 如果索引为 -1，返回深度 0
    int dl = DepthDFS(list[i].l); // 递归计算左子树深度
    int dr = DepthDFS(list[i].r); // 递归计算右子树深度
    return max(dl, dr) + 1; // 返回最大深度加 1
}
```

深度优先搜索比较两棵树的函数

```
/**
 * @brief           使用深度优先搜索（DFS）比较两棵树是否同构
 * @param t1        第一棵树的引用
 * @param t2        第二棵树的引用
 * @param i1        第一棵树当前节点的索引
 * @param i2        第二棵树当前节点的索引
 * @return          如果两棵树同构返回 true，否则返回 false
 */
bool Tree::tree_compareDFS(const Tree &t1, const Tree &t2, int i1, int i2){
    如果其中一棵树遍历结束，检查索引是否相等  return i1 == i2;
```

```
    如果节点名称不同  return false;
    // 检查同构情况
    return ((tree_compareDFS(t1, t2, t1.list[i1].l, t2.list[i2].l) &&
tree_compareDFS(t1, t2, t1.list[i1].r, t2.list[i2].r)) ||
            (tree_compareDFS(t1, t2, t1.list[i1].l, t2.list[i2].r) &&
tree_compareDFS(t1, t2, t1.list[i1].r, t2.list[i2].l)));
}
```

2.2.5 调试分析（遇到的问题和解决方法）

2.2.5.1 根节点查找问题

FindRoot 函数的逻辑有可能导致根节点查找失败，尤其是在某些结构较为特殊的树中。最终我在通过在遍历过程中输出一些调试信息来验证每个节点的 p 是否正确标识父节点，以及是否准确找到了根节点。必要时，可以修改 FindRoot 的逻辑，例如遍历整个 list 数组查找父节点 p 为-1 的节点作为根节点。

2.2.5.2 tree_compareDFS 函数的分析与优化

该函数主要通过递归调用实现了同构树的比较，即判断两棵树的结构是否相同或是对称同构。我采用提前返回的方法，在两个递归调用之间，直接判断左子树与右子树同构和左、右子树对称同构是否满足同构条件。如果在任意步骤返回 false，则可以提前终止递归。

另外，对称同构优化：当树结构较大时，可以先对(tree_compareDFS(t1, t2, t1.list[i1].l, t2.list[i2].l) 和 tree_compareDFS(t1, t2, t1.list[i1].r, t2.list[i2].r)进行并行判断。但对于一般树的大小，这种改进可能没有必要。

```
bool direct_match = tree_compareDFS(t1, t2, t1.list[i1].l, t2.list[i2].l)
&& tree_compareDFS(t1, t2, t1.list[i1].r, t2.list[i2].r);
bool mirror_match = tree_compareDFS(t1, t2, t1.list[i1].l, t2.list[i2].r)
&& tree_compareDFS(t1, t2, t1.list[i1].r, t2.list[i2].l);
return direct_match || mirror_match;
```

2.2.6 总结和体会

2.2.6.1 递归的巧妙性与复杂度控制

树的遍历、深度计算和同构性判断都非常适合递归实现，因为递归天然适用于分层结构的处理。在 DepthDFS 和 tree_compareDFS 的实现中，通过合理的递归终止条件和递归的嵌套调用，可以清晰地表达每个节点的操作。与此同时，控制递归的复杂度也是难点之一。特别是在同构性判断中，不必要的条件判断和分支嵌套容易导致逻辑复杂，因此通过简化判断条件，提升代码的效率和清晰度非常重要。

2.2.6.2 理解数据边界的影响

边界条件（例如树为空、只有一个节点或所有节点都没有左右子树等）在递归处理树时尤为关键。代码在这些特殊情况下容易出现错误，因此考虑所有可能的数据边界显得尤为重要。例如，在判断两棵树是否同构时，任何一棵树为空时的处理，以及单节点树的处理都需仔细检查，以保证代码的健壮性。

2.3 感染二叉树需要的总时间

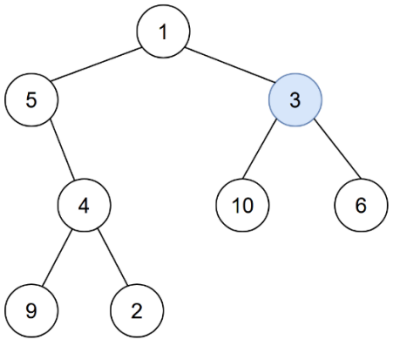
2.3.1 问题描述

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。



例如上图中 start=3，输出：4

解释：节点按以下过程被感染：-第 0 分钟：节点 3-第 1 分钟：节点 1、10、6-第 2 分钟：节点 5-第 3 分钟：节点 4-第 4 分钟：节点 9 和 2 感染整棵树需要 4 分钟，所以返回 4。

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

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

```
struct TreeNode {
    int val; // 节点值
    TreeNode *left; // 左子节点  TreeNode *right; // 右子节点
    TreeNode(int x) {
        val = x;  left = NULL;  right = NULL;
    }
};
class Solution {
private:
    int start; // 开始感染的节点值
    int res = 0; // 记录最大时间
public:
    int cal_time(TreeNode* root, int start);  // 从指定节点感染整棵树的时间
    int dfs(TreeNode* cur); // 深度优先搜索
};
```

2.3.4 功能说明（函数、类）

深度优先搜索的函数

```
/**
 * @brief           执行深度优先搜索
 * @param cur       当前节点指针
 * @return          返回当前节点的深度或特殊值
 */
int Solution::dfs(TreeNode* cur) {
    if (cur == nullptr) return 0; // 递归终止条件
    int rl = dfs(cur->left);  int rr = dfs(cur->right); // 左右子树返回值
    if (cur->val 找到起点) { // 找到感染起点
        res = max(res, max(rl, rr)); // 更新结果
        return -1; // 返回-1 表示找到感染起点
    }
    if (rl < 0) { // 处理从左子树传播的情况
        res = max(res, -rl + rr); // 更新结果
        return rl - 1; // 返回传播路径
    }
    else if (rr < 0) { // 处理从右子树传播的情况
        res = max(res, -rr + rl); // 更新结果
        return rr - 1; // 返回传播路径
    }
    return max(rl, rr) + 1; // 返回左右子树的最大深度
}
```

main 函数中关键部分如下

```
// 输入每个节点的左右子节点
    for (i 从 0 到 n) {
        int left, right;  cin >> left >> right; // 左右子节点编号
        if (有左子节点)  nodes[i]->left = nodes[left]; // 连接左子节点
        if (有右子节点)  nodes[i]->right = nodes[right]; // 连接右子节点
    }
```

2.3.5 调试分析（遇到的问题和解决方法）

2.3.5.1 节点连接问题

一开始在输入和连接左右子节点时，如果输入的左右子节点编号不正确，或连接错误，会导致意外的访问错误或者程序崩溃。最终确保输入的左右子节点编号正确，且输入负值（-1）表示没有左或右子节点。可以通过打印调试信息（如节点连接信息）来确认树的结构是否正确。

```
for (int i = 0; i < n; ++i)
    cout << "Node" << i << ":Left->" << left << "Right->" << right << endl;
```

2.3.5.2 递归返回值的处理问题

在 dfs 函数中，特别是 rl < 0 和 rr < 0 的判断逻辑，对于左右子树传播情况的处理有些复杂，容易出现错误。特别是返回值的负号含义需要准确理解，用于传递感染路径的深度。后来我仔细分析 dfs 返回值的定义，通过调试打印每一层的 rl 和 rr 来确保逻辑的正确性，并在代码中添加注释帮助理解负值传递的含义。

```cpp
int Solution::dfs(TreeNode* cur) {
    // 定义及前序逻辑...
    cout << "Node " << cur->val << ",rl= " << rl << ",rr= " << rr << endl;
    // 继续逻辑处理...
}
```

2.3.6 总结和体会

2.3.6.1 感染路径的动态更新

在设计递归函数时，体会到从子树向上层传递信息的技巧，尤其是通过返回值和状态的标识（例如用负数来传递路径信息）来跟踪感染的传播路径。这种设计在复杂的树递归问题中非常有用。

2.3.6.2 递归返回值的设计

递归函数中对负值的返回用来表示感染传播路径，这种方法尽管有效，但一开始不易理解，稍有不慎就可能导致错误传播。需要确保负值的含义清晰，并且在递归返回中始终保持一致。

2.4 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。

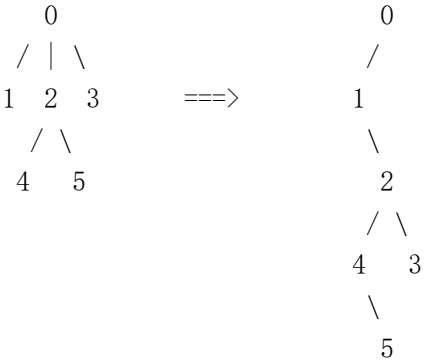
有序树到二叉树的转化步骤为：

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：



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：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。

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。

2.4.3 数据结构设计

```
const int MAX_LENGTH = 2000000; // 假设输入字符串总长度小于 2,000,000
char s[MAX_LENGTH]; // 用于存储输入字符串
int bdp = -1; // 记录变换后树的最大深度
struct Tree {
    int depth;          // 当前树的最大深度
    int currentDepth;   // 当前节点的深度
    Tree() { // 初始化构造函数
        depth = 0;  currentDepth = 0;
    }
};
```

2.4.4 功能说明（函数、类）

深度优先搜索函数

```
/**
 * @brief           执行深度优先搜索（DFS）遍历字符串表示的树结构
 * @param i         当前索引，用于遍历字符串
 * @param fatherDepth 父亲节点的深度
 */
void dfs(int& i, int fatherDepth) {
    bdp = max(bdp, fatherDepth); // 更新最大深度
    int cnt = 0; // 记录当前节点的子节点数量
    while (s[i]) { // 遍历字符串直到结束
        if (s[i] == 'd') { // 如果遇到节点
            dfs(++i, fatherDepth + cnt + 1); // 递归调用，增加深度
            cnt++; // 子节点计数增加
        }
        else 移动到下一个字符，结束当前递归 // 遇到其他字符（叶子节点）
    }
}
```

main 函数中关键部分如下

```
        // 计算原始树的深度
        for (int i = 0; s[i]; i++) {
            if (s[i] == 'd') { // 遇到节点
                treeInfo.currentDepth++; // 深度增加，更新最大深度
                treeInfo.depth = max(treeInfo.depth, treeInfo.currentDepth);
            } else { // 遇到其他字符
                treeInfo.currentDepth--; // 深度减少
            }
        }
        // 计算转换后的树的深度
        int ptr = 0; // 指向字符串的指针
        dfs(ptr, 0); // 调用 DFS 计算深度
```

2.4.5 调试分析（遇到的问题和解决方法）

2.4.5.1 bdp 的值出现错误

bdp 没有在 dfs 函数调用时初始化. bdp 是全局变量,用于记录转换后的树的最大深度。多个测试用例导致 bdp 值在不同树之间残留，导致计算错误。解决方法是在每棵树的深度计算前，重置 bdp 为-1。

2.4.5.2 递归结束条件不明确

dfs 函数引发无限递归，经检查原因是递归调用未明确检查是否已经到达字符串末尾，这在输入不完整或格式不正确的情况下可能引发无限递归。解决方法是在 dfs 函数的 while 循环中添加条件 s[i] != '\0'，确保递归时不会越界。

2.4.6 总结和体会

题目需要我们用递归去遍历和解析字符串中表示的树结构。理解递归的传递逻辑，特别是如何在 dfs 函数中记录和传递深度，是题目的一个关键难点。递归的结束条件和返回值设计不当，可能会导致不正确的深度计算，甚至导致无限递归。

通过此题的调试和优化，我对递归的结束条件设计、递归结果的更新方式有了更清晰的理解。同时，构造合适的递归传参方式（例如利用父节点深度来推导子节点深度）也是一种重要收获。

2.5 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

```
// 定义树节点结构体
struct TreeNode {
    int parent; // 节点的父节点
    TreeNode() {
        parent = 0; // 构造函数初始化父节点为 0
    }
};
```

2.5.4 功能说明（函数、类）

查找最近公共祖先的函数

```
/**
 * @brief           寻找两个节点的最近公共祖先
 * @param Tree      树的节点数组
 * @param x         第一个节点的索引
 * @param y         第二个节点的索引
 * @return          返回最近公共祖先的索引，如果不存在则返回 -1
 */
int get_grapa(TreeNode Tree[], int x, int y) {
    bool ancestors[MAX_NODES] = {false}; // 用于记录 x 的所有祖先
    if (x == y) return x; // 如果两个节点相同，直接返回该节点
    while (x != 0) { // 存储 x 的所有祖先
        ancestors[x] = true; // 标记当前节点为祖先
        x = Tree[x].parent; // 移动到父节点
    }
    while (y != 0) { // 查找 y 是否在 x 的祖先中
        if (ancestors[y]) // 如果 y 是 x 的祖先
            return y; // 返回 y 作为最近公共祖先
        y = Tree[y].parent; // 移动到父节点
    }
    return -1; // 如果没有找到公共祖先，返回 -1
}
```

main 函数中关键部分如下

```
        for (i 从 1 到 N) { // 输入树的结构
            int temp_parent, temp_children; // 临时变量存储父节点和子节点
            cin >> temp_parent >> temp_children; // 输入父子关系
            Tree[temp_children].parent = temp_parent; // 设置子节点的父节点
        }
        for (i 从 1 到 M) { // 处理查询
            int x_node, y_node; // 存储查询的两个节点
            cin >> x_node >> y_node; // 输入要查询的节点
            cout << get_grapa(Tree, x_node, y_node) << endl; // 最近公共祖先
        }
```

2.5.5 调试分析（遇到的问题和解决方法）

一开始采用 dfs 遍历树、lca 函数进行查找，并通过在 dfs 函数中增加调试输出，验证每个节点的父节点和深度是否正确。验证发现 dep[node]的计算逻辑是正确的，同时确认 fa[node][0]（直接父节点）在 dfs 遍历后已正确设置。在 lca 查询中添加调试输出，通过打印每次调整 x 和 y 的位置和深度，确认每一步的变更是否符合预期。

```
int lca(int x, int y) {
    cout << "LCA query for: " << x << " and " << y << endl;
    if (dep[x] < dep[y]) swap(x, y);
    int diff = dep[x] - dep[y];
    cout << "Initial depth difference: " << diff << endl;
    for (int i = 0; diff; i++, diff >>= 1) {
        if (diff & 1) x = fa[x][i];
        cout << "Move up x to: " << x << " current diff: " << diff << endl;
    }
    if (x == y) return x;
    for (int i = LOG - 1; i >= 0; --i) {
        if (fa[x][i] != fa[y][i]) {
            cout << "Moving x to: " << fa[x][i] << " and y to: " << fa[y][i]
<< endl;
            x = fa[x][i];
            y = fa[y][i];
        }
    }
    cout << "LCA: " << fa[x][0] << endl;
    return fa[x][0];
}
void dfs(int node, int parent) {
    fa[node][0] = parent; // 第 0 级祖先
    dep[node] = (parent == 0) ? 0 : dep[parent] + 1; // 深度
    // ……
}
```

发现在测试一个后结果变得不准确，最终决定进行调整，使用 get_grapa，get_grapa 没有预处理节点的多级祖先，而是直接使用 DFS 搜索来记录一个节点的祖先路径。它依赖树的父指针 Tree[].parent 来向上查找公共祖先，通过对每次查询逐步上溯来查找最近公共祖先，空间开销较小，空间复杂度低。而且简单，逐步向上溯源查找，不需要预处理多级祖先关系。

2.5.6 总结和体会

这道题让我更深入理解了树结构的特性，以及如何高效地找到两个节点的最近公共祖先(LCA)。LCA 问题是图算法中的经典问题，掌握其基本方法为处理复杂树形结构问题提供了很好的基础。

在树形结构中，父节点的管理非常重要。在实现过程中，若父子关系设定错误或遍历不完全，可能导致无法正确查询到公共祖先，造成程序运行错误。

2.6 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。

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

```
class BinaryTree {
public:
    // 构造函数，接受前序和中序遍历字符串
    BinaryTree(const char* preorder, const char* inorder) {
        strcpy(this->preorder, preorder); // 复制前序遍历字符串到成员变量
        strcpy(this->inorder, inorder);   // 复制中序遍历字符串到成员变量
        hasError = false;                 // 初始化错误标记
    }
    // 获取后序遍历结果
    void getPostorder(char* result);
private:
    char preorder[100];  // 存储前序遍历字符串，假设最大长度为 100
    char inorder[100];   // 存储中序遍历字符串，假设最大长度为 100
    bool hasError;       // 错误标记
    // 递归计算后序遍历
    void postorder(const char* preorder, const char* inorder, char* result,
int& pos);
};
```

2.6.4 功能说明（函数、类）

实现后序遍历的递归函数

```
/**
 * @brief           根据给定的前序和中序遍历字符串构造二叉树的后序遍历
 * @param preorder  前序遍历字符串
 * @param inorder   中序遍历字符串
 * @param result    后序遍历结果字符串
 * @param pos       后序遍历结果字符串的当前位置
 */
void BinaryTree::postorder(const char* preorder, const char* inorder, char*
result, int& pos) {
```

```
    if (前序或中序字符串为空) return;
    char root = preorder[0]; // 前序的第一个字符是根节点
    // 在中序遍历中找到根节点的位置
    const char* root_pos = strchr(inorder, root);
    if (root_pos == nullptr) { // 根节点不在中序字符串中，设置错误标记并返回
        hasError = true;
        return;
    }
    // 递归分割中序和前序字符串，构造左右子树
    int left_size = root_pos - inorder; // 左子树的大小
    char left_inorder[100], right_inorder[100];
    char left_preorder[100], right_preorder[100];
    strncpy(left_inorder, inorder, left_size); // 左子树的中序
    left_inorder[left_size] = '\0';
    strcpy(right_inorder, root_pos + 1);       // 右子树的中序
    strncpy(left_preorder, preorder + 1, left_size); // 左子树的前序
    left_preorder[left_size] = '\0';
    strcpy(right_preorder, preorder + 1 + left_size); // 右子树的前序
    // 递归调用以获取左右子树的后序遍历
    postorder(left_preorder, left_inorder, result, pos);
    if (hasError) return;  // 如果递归调用出现错误，立即返回
    postorder(right_preorder, right_inorder, result, pos);
    if (hasError) return;  // 如果递归调用出现错误，立即返回
    result[pos++] = root; // 将根节点添加到后序遍历的结果
}
```

2.6.5 调试分析（遇到的问题和解决方法）

2.6.5.1 内存泄漏问题

一开始运行代码后发现内存使用持续增加，导致程序在长时间运行后可能会崩溃。由于 postorder 函数中使用 new 动态分配了内存，但在递归调用结束后并没有释放，导致内存泄漏。后面增加了释放过程，解决了问题。

也可以将 postorder 函数的返回类型改为 std::string，这样可以利用 C++标准库自动管理内存，避免动态内存分配带来的问题。代码示例如下：

```
string BinaryTree::postorder(const string& preorder, const string&
inorder) {
    // ……
    // 递归分割中序和前序字符串，构造左右子树
    string left_inorder = inorder.substr(0, root_index); // 左子树的中序
    string right_inorder = inorder.substr(root_index + 1); // 右子树的中序
    // 左子树的前序
    string left_preorder = preorder.substr(1, left_inorder.length());
    // 右子树的前序
    string right_preorder = preorder.substr(1 + left_inorder.length());
    // 递归调用以获取左右子树的后序遍历
```

```
    string left_postorder = postorder(left_preorder, left_inorder);
    string right_postorder = postorder(right_preorder, right_inorder);
    // ……
    // 返回左右子树的后序遍历加上根节点，形成完整的后序遍历
    return left_postorder + right_postorder + root;
}
```

2.6.5.2 递归返回值的错误判断问题

一开始当输入的前序和中序遍历字符串不匹配或无法构造有效的树时，程序没有返回预期的错误信息，反而输出了错误的后序遍历结果。经检查原因是在递归过程中，如果左右子树的后序遍历返回了"Error"，没有在上一层递归中正确处理这一情况，而是继续进行拼接。最终在 postorder 函数中加入检查，如果左右子树的后序遍历结果包含"Error"，则立即返"Error"，并跳过后续拼接步骤。

2.6.6 总结和体会

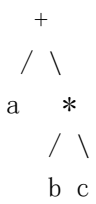
在递归构造二叉树的过程中，需在每一步切割前序和中序遍历的字符串来找到左右子树，这对递归的分解和合并有了更深的理解。对于前序、中序和后序遍历之间的关系，理解如何利用前序找到根节点，并使用根节点分割中序列表是构造递归的核心。

此外，内存管理和错误处理在递归程序中至关重要，使用标准库类（如 std::string）能大大降低内存管理的复杂性。

2.7 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：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

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，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。时间限制：1000ms 内存限制：65535kb

输入样例：

```
a+b*c
3
a 2
b 7
c 5
```

输出样例：

```
abc*+
    +
  / \
 a   *
    / \
    b c
37
```

2.7.3 数据结构设计

```
string s; // 输入表达式
int num; // 变量数量  int map[26] = {0}; // 存储变量的值
int priority[128] = {0}; // 运算符优先级
const int N = 100; // 最大节点数  int root; // 树的根节点
int tree_x, tree_y; // 输出树的位置
char matrix[500][500]; // 存储绘制树的矩阵  int printLen[500]; // 每行打印长度
// 定义树节点类
class Node {
public:
    Node(int sub = -1, char blank = ' ', int left = -1, int right = -1, int
value = 0) {
        this->children = sub; // 节点的索引
        this->c = blank; // 节点存储的字符
        this->l = left; // 左子节点
        this->r = right; // 右子节点
        this->value = value; // 节点的值
    }
    int children; // 子节点索引  char c; // 存储的字符
    int l; // 左子节点索引  int r; // 右子节点索引  int value; // 节点的值
};
Node tree[N]; // 表达式树
```

```
Node stack_of_num[N]; // 数字栈  Node stack_of_operator[N]; // 运算符栈
int index_n = 0; // 数字栈索引  int index_priority = 0; // 运算符栈索引
```

2.7.4 功能说明（函数、类）

计算表达式的值并构建表达式树的函数

```
/**
 * @brief           根据给定的表达式字符串计算表达式的值并构建表达式树
 * @param s         表达式字符串
 * @return          返回表达式的值
 */
int cal_and_build_tree(string s) {
    for (size_t i = 0; i < s.size(); i++) {
        Node node = Node(i, s[i]);
        tree[i] = node;
        if (s[i] >= 'a' && s[i] <= 'z'){ // 判断是否为变量
            node.value = map[s[i] - 'a']; // 获取变量的值
            stack_of_num[index_n++] = node; // 存入数字栈
        }
        else { // 判断是否为运算符
                如果 s[i] 是操作数（数字），将操作数节点加入 stack_of_num
                如果 s[i] 是左括号 '('：将 '(' 入栈 stack_of_operator
                如果 s[i] 是右括号 ')'：
                当 stack_of_operator 顶部元素不是 '(' 时，弹出运算符栈并进行计算
                将计算结果存入数字栈 stack_of_num，弹出 '('
                如果 s[i] 是运算符（如 +, -, *, /）：
                当 stack_of_operator 非空且栈顶运算符优先级>=当前运算符优先级时：
                弹出运算符栈，并执行运算，将运算结果存入数字栈 stack_of_num
                将当前运算符入栈 stack_of_operator
    }
    // 处理剩余的运算符
    while (index_priority > 0) {
        Node nd = stack_of_operator[--index_priority]; // 弹出运算符栈
        nd.value = domatrixh(stack_of_num, index_n, nd.c, nd); // 计算结果
        tree[nd.children] = nd; // 更新树
        stack_of_num[index_n++] = nd; // 将结果存入数字栈
    }
    root = stack_of_num[index_n - 1].children; // 根节点
    return stack_of_num[index_n - 1].value; // 返回表达式值
}
```

广度优先遍历，打印树结构的函数

```
/**
 * @brief           广度优先遍历树并打印树结构
 * @param h         树的高度
 */
```

```
void bfs(int h) {
    tree_x = 0; // 初始 x 坐标  tree_y = (fps(2, h) - 1) / 2; // 初始 y 坐标
    Node stack[N]; // 存储当前节点的栈
    int stackIndex = 0;
    Node currentNode = tree[root]; // 从根节点开始
    stack[stackIndex++] = currentNode;
    int posX[N] = {0}; // 记录 x 坐标  int posY[N] = {0}; // 记录 y 坐标
    posX[0] = tree_x; // 初始化 x 坐标  posY[0] = tree_y; // 初始化 y 坐标
    int index = 0;
    while (index < stackIndex) {
        currentNode = stack[index++]; // 当前节点
        int t_x = posX[index - 1];  int t_y = posY[index - 1]; // 坐标
        从栈中取出当前节点 currentNode，获取当前节点的坐标 (t_x, t_y)
        根据连接关系调整 y 坐标
        将 currentNode 的字符放入矩阵 matrix[t_x][t_y]
        如果当前节点有左子节点:
        绘制 '/' 并将左子节点加入栈，更新左子节点的坐标 (t_x + 2, t_y - 1)
        如果当前节点有右子节点:
        绘制 '\' 并将右子节点加入栈，更新右子节点的坐标 (t_x + 2, t_y + 1)
        如果没有子节点，将连接线位置设为空格
    }
}
```

其他函数

```
int fps(int a, int p) // 快速幂
int cal_depth(int root1) // 计算树的深度
void back_tranverse(int pos, int x, int y) // 反向遍历树并输出结果
// 进行数学计算，根据符号
int domatrixh(Node* num, int& index_n, char sym, Node& fa)
```

2.7.5 调试分析（遇到的问题和解决方法）

2.7.5.1 节点位置重叠

一开始在树的广度优先遍历中，部分节点位置重叠。由于节点的 x 和 y 坐标计算不当，导致树形结构中的节点相互重叠，造成树的显示不清晰。经检查原因是初始情况下，设置了固定的 tree_x 和 tree_y 坐标为根节点的起始位置。接着，每个节点根据其在树中的层级调整位置，但并没有充分考虑节点间的适当间隔。

发现问题时，我开始检查 posX[]和 posY[]数组，确保节点位置之间有足够的间距，避免重叠。我调整了节点的 y 坐标计算逻辑，确保每一层的节点之间有足够的间距。引入了一个动态调整节点间距的机制，每一层根据树的高度调整行间距，避免了节点之间的重叠。

2.7.5.2 矩阵大小不足

树的高度和宽度可能会超过预设的矩阵大小，导致无法正确绘制树。我检查了矩阵的大小是否足够容纳整个树，并发现对于较高的树，矩阵的宽度和高度不够大，导致绘制过程中出错。最后我根据树的深度动态调整矩阵的大小。在初始化矩阵时，根据树的层级动态计算所需的行和列数，确保矩阵能够容纳所有节点和连接线。

2.7.6 总结和体会

树的广度优先遍历要求节点的位置根据其层级进行合理调整。如果树的层数较多，节点的间距和位置容易发生重叠或错位，特别是节点的坐标需要根据树的深度和层级关系动态调整。在实际实现过程中，如何合理分配树中每个节点的位置是一个挑战。

树形结构的可视化需要使用矩阵来存储每个节点的位置。矩阵的大小需要根据树的深度和宽度来动态调整。处理大型树时，矩阵的大小不足以容纳树的所有节点和连接线，这就要求我根据树的最大深度和节点数来动态地调整矩阵的尺寸，避免在绘制过程中发生溢出。

3. 实验总结

3.1 树的建立

建树方法：

- 使用前序、中序或后序序列可以建立树。通过递归的方式可以根据这些序列进行树结构的还原。
- 进栈出栈顺序：可以模拟树的建立过程，例如通过给定的入栈和出栈顺序来还原树结构。
- 前序加空节点：通过在前序序列中加入空节点信息，可以模拟进栈出栈顺序，从而建立树。

3.2 树的遍历

常见遍历方法：前序遍历、中序遍历、后序遍历。

层序遍历：通过广度优先搜索（BFS）来实现层序遍历。可以通过队列（先进先出）来逐层访问树的节点。

递归遍历：递归遍历是最常见的树遍历方式。通过控制递归的顺序来实现不同的遍历方式（前序、中序、后序）。递归的终止条件通常是节点为空（空树）。

非递归遍历：非递归遍历模拟递归过程，利用栈（后进先出）来控制遍历顺序。

- 前序非递归遍历：根节点先打印，左子树遍历完后出栈，再遍历右子树。
- 中序非递归遍历：先遍历左子树，再打印根节点，最后遍历右子树。
- 后序非递归遍历：左子树遍历完后出栈，再遍历右子树，最后打印根节点。可以通过增加计数器来判断节点是否已经遍历过，从而确定何时打印根节点。

3.3 树的线索化

在树的线索化中，如果节点的左子树为空，则将其左指针指向前一个节点，右子树为空时，将其右指针指向下一个节点。

线索化的过程本质上是给树结构添加指针，形成一个有序链表，但这种链接是逻辑上的，并不改变树的实际物理结构。

线索化的标记通常用 ltag 和 rtag 来指示节点的左、右指针是指向孩子节点还是前后节点。

3.4 树的转化

树的转换方法：

- PrintByMode：通过传递不同的模式来控制树的遍历顺序，模式值决定了当前节点的打印时机（前序、中序、后序）。
- 线索化转换：通过 inTheading 函数，树的节点按照线索化规则进行转换。如果节点的左子树为空，则通过 ltag 和 rtag 来记录与前后节点的关系，从而形成双向指针链表。