**作业 HW3\* 实验报告**

日期：2024年11月3日

1. **涉及数据结构和相关背景**

* **树的定义：树是由节点组成的非线性结构，包含一个根节点及其子节点。**
* **二叉树：每个节点最多有两个子节点，称为左子节点和右子节点。**
* **树的建立：通过不同的方式建立树结构，如前序、中序和后序建树。**
* **建树方法：**
* **前序加中序建树：利用前序遍历的第一个节点为根，在中序遍历中找到根的位置，递归构建左右子树。**
* **中序加后序建树：利用中序遍历找到根节点的位置，再用后序遍历确定左右子树。**
* **遍历顺序结合栈的出入顺序来建树。**
* **前序加空节点建树：将空节点作为占位符，以便于构建完整的树结构。**
* **树的遍历：**
  + **前序遍历：根节点→左子树→右子树。**
  + **中序遍历：左子树→根节点→右子树。**
  + **后序遍历：左子树→右子树→根节点。**
  + **层序遍历：按层次从上到下遍历树。**
* **遍历方式：**
* **递归：利用函数调用栈实现遍历。**
* **非递归：使用栈或队列来模拟遍历过程。**
* **树的线索化：将树中的空指针利用起来，使遍历更高效。**
* **树的转化：将树结构转化为二叉树，方便存储和遍历。**
* **求树的深度：通过深度优先搜索（DFS）算法计算树的高度。**
* **树的应用：在表达式树中表示运算。**

**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。已知先序遍历序列通过栈操作构建二叉树后，要求输出二叉树的后序遍历序列。

**2.1.2 基本要求**

输入一个整数n表示节点数，接下来2n行描述栈操作（"push X"表示压栈，"pop"表示弹栈），输出构建后的二叉树的后序遍历序列。

**2.1.3 数据结构设计**

使用节点类定义二叉树节点的基本结构，包括数据域和左右子树指针。

// 定义二叉树节点结构

class Node {

public:

char data; // 节点存储的数据

Node\* lc; // 左孩子指针

Node\* rc; // 右孩子指针

// 构造函数初始化

Node(char d, Node\* left = NULL, Node\* right = NULL) {

data = d;

lc = left;

rc = right;

}

};

使用栈记录尚未遍历完的节点，确保按照栈操作序列构建二叉树。

stack<Node\*> st; // 用于存储未完成遍历的节点

**2.1.4功能说明（函数、类）**

**1. 建立二叉树 buildTree**

根据输入的栈操作顺序，通过栈存储和栈顶节点的回溯方式构建二叉树。

/\*\*

\* @brief 根据输入的栈操作建立二叉树

\* @param n 节点个数

\* @param root 树的根节点

\*/

void buildTree(int n, Node\* root) {

Node\* pointer = root;

bool leftAva = true; // 标记是否可以插入左孩子

while (n > 0) {

string operation;

char data;

cin >> operation;

if (operation == "pop") {

// 当左孩子可插入时标记为不可插入，其他情况指针回溯到未完成的父节点

if (leftAva) {

leftAva = false;

}

else {

pointer = st.top(); // 回退到未完成的左子树节点

st.pop();

}

}

else { // 若操作为"push X"，表示插入节点X

cin >> data;

st.push(pointer);

if (leftAva) { // 插入左孩子

pointer->lc = new Node(data);

pointer = pointer->lc;

}

else { // 插入右孩子并出栈

pointer->rc = new Node(data);

pointer = pointer->rc;

st.pop();

}

leftAva = true;

n--;

}

}

}

##### 2. 后序遍历 backTraverse

递归实现后序遍历，先左子树，再右子树，最后输出节点自身数据。

/\*\*

\* @brief 后序遍历二叉树

\* @param root 当前节点

\*/

void backTraverse(Node\* root) {

if (root->lc) backTraverse(root->lc);

if (root->rc) backTraverse(root->rc);

cout << root->data; // 输出当前节点的数据

}

##### 主函数 main

在主函数中调用buildTree构建树，再调用backTraverse输出后序遍历序列。

int main() {

int n;

cin >> n; // 输入节点数量

string operation;

char data;

cin >> operation >> data; // 初始根节点

Node\* root = new Node(data); // 构建根节点

buildTree(n - 1, root); // 构建剩余树

backTraverse(root); // 后序遍历并输出

return 0;

}

**2.1.5 调试分析（遇到的问题和解决方法）**

**建树顺序错误**：曾误将插入左、右孩子的顺序颠倒，导致树结构不正确。通过分析后发现应按先左后右的顺序插入。

**回溯时未正确处理栈顶节点**：在实现中发现左节点未遍历完直接进行pop会出错。调整后确保仅当左子树遍历完毕时才回溯至栈顶。

**2.1.6 总结和体会**

该题通过栈操作构建二叉树，再进行后序遍历输出，考察了非递归遍历中的栈操作使用。在建树过程中，节点的左右子树顺序对最终结果的准确性有重要影响。通过此次实验，加深了对二叉树遍历和栈的操作流程的理解。在处理更大数据量时，还可考虑优化空间复杂度。

**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：首先输入第一棵树的节点数N1，然后是N1行描述每个节点的信息（节点的值、左孩子和右孩子的索引）。接着输入第二棵树的节点数N2和相应的节点信息。
* 输出：若两棵树同构，输出“Yes”，否则输出“No”。并在后面两行分别输出两棵树的深度。

**2.2.3 数据结构设计**

定义一个节点结构体，用于存储节点的值及其左右孩子的索引。

struct Node {

char value; // 节点值

int leftChild; // 左孩子索引

int rightChild; // 右孩子索引

};

**2.2.4功能说明（函数、类）**

建树函数

/\*\*

\* @brief 根据输入的数据建立二叉树

\* @param n 节点个数

\* @return 返回根节点的索引

\*/

function buildTree(n: integer)->integer:

declare nodes[n] as Node

for i from 0 to n - 1 :

read nodes[i].value, leftIndex, rightIndex

if leftIndex != -1 :

nodes[i].leftChild = leftIndex

else :

nodes[i].leftChild = NULL

if rightIndex != -1 :

nodes[i].rightChild = rightIndex

else :

nodes[i].rightChild = NULL

return index of root node

计算树的深度

/\*\*

\* @brief 计算树的深度

\* @param root 根节点索引

\* @param nodes 树的节点数组

\* @return 返回树的深度

\*/

function findDepth(root: integer, nodes : array of Node)->integer:

if root is NULL :

return 0

depthLeft = findDepth(nodes[root].leftChild, nodes)

depthRight = findDepth(nodes[root].rightChild, nodes)

return max(depthLeft, depthRight) + 1

判断同构函数

/\*\*

\* @brief 判断两棵树是否同构

\* @param root1 第一棵树的根节点索引

\* @param root2 第二棵树的根节点索引

\* @param nodes1 第一棵树的节点数组

\* @param nodes2 第二棵树的节点数组

\* @return 如果同构返回true，否则返回false

\*/

function isIsomorphic(root1: integer, root2 : integer, nodes1 : array of Node, nodes2 : array of Node)->boolean:

if root1 is NULL and root2 is NULL :

return true

if root1 is NULL or root2 is NULL :

return false

if nodes1[root1].value != nodes2[root2].value :

return false

// 判断左右子树是否同构

return (isIsomorphic(nodes1[root1].leftChild, nodes2[root2].leftChild, nodes1, nodes2) and

isIsomorphic(nodes1[root1].rightChild, nodes2[root2].rightChild, nodes1, nodes2)) or

(isIsomorphic(nodes1[root1].leftChild, nodes2[root2].rightChild, nodes1, nodes2) and

isIsomorphic(nodes1[root1].rightChild, nodes2[root2].leftChild, nodes1, nodes2))

**2.2.5 调试分析（遇到的问题和解决方法）**

在调试过程中，主要遇到的问题是处理节点为空的情况。在判断同构时，需确保在进入递归之前，先检查当前节点是否为空，避免空指针异常。

**2.2.6 总结和体会**

通过本次实验，掌握了二叉树的基本操作及判断树结构同构的思路。难点在于如何有效地判断左右子树的同构性，特别是在处理交换情况时。设计良好的递归结构可以有效减少重复判断，提高算法效率。

**2.3 感染二叉树需要的总时间**

**2.3.1 问题描述**

给定一棵二叉树的根节点以及一个整数start，表示从值为start的节点开始感染。每分钟，满足以下条件的节点将被感染：

* 节点此前未被感染。
* 节点与至少一个已感染节点相邻。 返回感染整棵树所需的分钟数。

**2.3.2 基本要求**

* 输入：第一行包含两个整数n和start，接下来的n行描述每个节点的左、右孩子编号。
* 输出：一个整数，表示感染整棵二叉树所需的时间。

**2.3.3 数据结构设计**

定义一个节点结构体，用于存储节点的状态（是否被感染）、父节点索引、左孩子和右孩子的索引。

// 节点结构体定义

struct tree {

bool visit = false; // 表示节点是否被感染

int father = -1; // 父节点索引

int lchild = -1; // 左孩子索引

int rchild = -1; // 右孩子索引

};

**2.3.4功能说明（函数、类）**

构建树的函数

/\*\*

\* @brief 根据输入的节点信息构建二叉树

\* @param the\_tree 指向树的节点数组

\* @param num 节点总数

\*/

void cin\_tree(tree\* the\_tree, int num) {

int tem\_left, tem\_right;

for (int i = 0; i < num; i++) {

the\_tree[i].visit = false; // 初始化节点状态

cin >> tem\_left >> tem\_right;

// 设置左孩子

if (tem\_left != -1) {

the\_tree[i].lchild = tem\_left;

the\_tree[tem\_left].father = i;

}

// 设置右孩子

if (tem\_right != -1) {

the\_tree[i].rchild = tem\_right;

the\_tree[tem\_right].father = i;

}

}

}

感染过程的函数

/\*\*

\* @brief 模拟感染过程并计算所需时间

\* @param infect\_tree 指向树的节点数组

\* @param start\_infect 开始感染的节点索引

\* @param num 节点总数

\* @return 感染整棵树所需的时间

\*/

int infect(tree\* infect\_tree, int start\_infect, int num) {

int\* queue = new int[num]; // 创建队列

int end = 0, start = 0; // 队列的头尾指针

infect\_tree[start\_infect].visit = true; // 标记起始节点为已感染

int time = -1;

queue[end++] = start\_infect; // 将起始节点加入队列

while (end != start) {

time++; // 每次循环表示一个时间单位

int tem\_num\_queue = end - start; // 当前队列中的节点数量

for (int i = 0; i < tem\_num\_queue; i++) {

int current = queue[start++]; // 取出当前节点

// 检查父节点

if (infect\_tree[current].father != -1 && !infect\_tree[infect\_tree[current].father].visit) {

queue[end++] = infect\_tree[current].father; // 将父节点加入队列

infect\_tree[infect\_tree[current].father].visit = true; // 标记为已感染

}

// 检查左孩子

if (infect\_tree[current].lchild != -1 && !infect\_tree[infect\_tree[current].lchild].visit) {

queue[end++] = infect\_tree[current].lchild; // 将左孩子加入队列

infect\_tree[infect\_tree[current].lchild].visit = true; // 标记为已感染

}

// 检查右孩子

if (infect\_tree[current].rchild != -1 && !infect\_tree[infect\_tree[current].rchild].visit) {

queue[end++] = infect\_tree[current].rchild; // 将右孩子加入队列

infect\_tree[infect\_tree[current].rchild].visit = true; // 标记为已感染

}

}

}

return time; // 返回所需时间

}

**2.3.5 调试分析（遇到的问题和解决方法）**

在调试过程中，主要遇到的问题是处理节点的边界情况，例如节点的左、右孩子为-1时的处理。在队列操作时也需要确保不越界。此外，确保正确初始化每个节点的visit状态，以避免错误的感染计算。

**2.3.6 总结和体会**

通过这道题，我掌握了二叉树的构建和遍历过程，并了解了如何使用队列实现广度优先搜索（BFS）。难点在于处理节点的连接关系及状态标记，特别是在大规模数据时，需注意算法的效率和内存管理。此次实验增强了我对数据结构和算法设计的理解，为今后类似问题的解决打下了基础。

**2.4树的重构**

**2.4.1 问题描述**

树在计算机科学中有广泛应用，除了常用的二叉树，还有其他有序树类型。有序树可以转化为二叉树，使得每个节点的子节点有固定的存储方式。将有序树转化为二叉树的方法如下：

1. 去除每个节点与其子节点的边。
2. 对于每个节点，将第一个子节点设为其左孩子。
3. 将节点的下一个兄弟节点设为其右孩子。

此转化可能导致树的深度增加。目标是实现程序计算转化前后树的深度。

**2.4.2 基本要求**

输入：多行，每行表示一棵树的深度优先遍历方向（d表示下行，u表示上行）。行以#结束。  
输出：打印树的转化前后深度，格式为Tree t: h1 => h2。

**2.4.3 数据结构设计**

使用深度优先遍历（DFS）记录深度，以实现有序树向二叉树的转化。关键是利用公式：  
当前节点深度 = 父节点深度 + 当前节点左兄弟数量。

以下为树节点的数据结构设计：

struct TreeNode {

TreeNode\* first\_child; // 第一个子节点

TreeNode\* first\_brother; // 下一个兄弟节点

TreeNode() : first\_child(nullptr), first\_brother(nullptr) {}

};

**2.4.4功能说明（函数、类）**

#### 函数 1：cin\_tree

功能：根据输入字符串构建原始的有序树结构，并计算树的深度。 输入：start\_tree（根节点指针）、input（包含方向信息的字符串）。 输出：返回原始树的最大深度。

/\*\*

\* @brief 根据输入字符串构建有序树结构，并计算树的深度

\* @param start\_tree 指向根节点的指针

\* @param input 包含方向信息的字符串

\* @return 构建的树的最大深度

\*/

int cin\_tree(tree\* start\_tree, string input) {

tree\*\* stack = new(nothrow) tree \* [input.size()]; // 动态栈，用于记录回溯路径

int top\_stack = NULL;

tree\* now\_tree = start\_tree;

int now\_high = 0, max\_high = 0;

for (int str = 0; input[str]; str++) {

if (input[str] == 'd') { // 向下：新节点是当前节点的孩子或兄弟

now\_high++;

if (now\_tree->first\_child == NULL) {

now\_tree->first\_child = &start\_tree[str + 1];

}

else {

tree\* brother\_tree = now\_tree->first\_child;

while (brother\_tree->first\_brother) // 找到兄弟链的末端

brother\_tree = brother\_tree->first\_brother;

brother\_tree->first\_brother = &start\_tree[str + 1];

}

stack[top\_stack++] = now\_tree; // 当前节点入栈

now\_tree = &start\_tree[str + 1]; // 更新当前节点

}

else if (input[str] == 'u') { // 向上：回到父节点

now\_high--;

now\_tree = stack[--top\_stack];

}

max\_high = max(max\_high, now\_high); // 记录最大深度

}

delete[] stack;

return max\_high; // 返回原始树的最大深度

}

#### 函数 2：er\_cha\_shu

功能：在构建后的有序树基础上，将其转化为二叉树结构并计算二叉树的深度。 输入：start\_tree（根节点指针）、input（包含方向信息的字符串）。 输出：返回转换后树的最大深度。

/\*\*

\* @brief 将构建的有序树转化为二叉树，并计算二叉树的深度

\* @param start\_tree 指向根节点的指针

\* @param input 包含方向信息的字符串

\* @return 转换后树的最大深度

\*/

int er\_cha\_shu(tree\* start\_tree, string input) {

tree\*\* queue = new(nothrow) tree \* [input.size()]; // 使用队列进行层序遍历

int start = 0, end = 0;

queue[end++] = start\_tree;

int high = -1;

while (start != end) {

high++;

int tem\_queue\_num = end - start;

for (int i\_cycle = 0; i\_cycle < tem\_queue\_num; i\_cycle++) {

tree\* now\_tree = queue[start++]; // 取出队首节点

if (now\_tree->first\_brother) // 加入兄弟节点

queue[end++] = now\_tree->first\_brother;

if (now\_tree->first\_child) // 加入第一个孩子节点

queue[end++] = now\_tree->first\_child;

}

}

delete[] queue;

return high; // 返回二叉树的深度

}

**逻辑描述**：

* 采用队列进行层次遍历。
* 初始化队列，将根节点 start\_tree 加入队列。
* 逐层遍历树的节点，每遍历一层 high 增加 1。
* 每个节点的第一个兄弟和第一个孩子加入队列，直到队列为空，最终返回深度 high。

#### 主函数：main

功能：主控逻辑，接收输入、调用上面两个核心函数，输出结果。 输入：通过标准输入接收树的方向遍历字符串。 输出：标准输出每棵树转换前后的深度，格式为 Tree t: h1 => h2。

/\*\*

\* @brief 主控逻辑，接收输入，调用上面两个函数，输出结果

\* @param 无

\* @return 无

\*/

int main() {

int num = 1;

string input;

while (1) {

cin >> input;

if (input == "#") break; // 检查结束符号

tree\* the\_tree = new tree[input.size()];

int high\_old, high\_new;

high\_old = cin\_tree(the\_tree, input); // 获取原树深度

high\_new = er\_cha\_shu(the\_tree, input); // 获取二叉树深度

delete[] the\_tree;

cout << "Tree " << num++ << ": " << high\_old << " => " << high\_new << endl;

}

}

**逻辑描述**：

* 不断读取输入的树的方向字符串。
* 调用 cin\_tree 和 er\_cha\_shu 获取转换前后深度。
* 输出格式化结果 Tree t: h1 => h2 并递增样例编号 num。

**2.4.5 调试分析（遇到的问题和解决方法）**

在调试过程中，确保递归计算的深度正确。主要挑战在于递归时确定当前节点的深度，关键在于利用公式：  
当前节点深度 = 父节点深度 + 当前节点左兄弟数量。

**2.4.6 总结和体会**

通过此题，加深了对树的深度遍历与广度遍历的理解，尤其是在有序树向二叉树的转化过程中深度变化的计算

**2.5最近公共祖先**

**2.5.1 问题描述**

给出一颗多叉树，求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**

对于每组输入数据，输出每个询问的两个节点的最近公共祖先。

**2.5.3 数据结构设计**

// 定义多叉树节点的数据结构

struct tree {

int high = 0; // 表示此节点所处的深度

tree\* first\_child = NULL; // 指向第一个孩子节点

tree\* first\_brother = NULL; // 指向右边第一个兄弟节点

tree\* father = NULL; // 指向父节点

// 如果某个节点没有相应的孩子或兄弟，指针为空

};

**2.5.4功能说明（函数、类）**

函数 1：cin\_tree

/\*\*

\* @brief 根据输入构建树并为每个节点计算深度

\* @param start\_tree 指向节点数组的指针

\* @param num 节点的总数

\* @return 返回根节点的指针

\*/

tree\* cin\_tree(tree\* start\_tree, int num) {

for (int i\_cycle = 1; i\_cycle < num; i\_cycle++) {

int fa, child;

cin >> fa >> child;

start\_tree[child].father = &start\_tree[fa]; // 设置父节点指针

if (start\_tree[fa].first\_child) {

tree\* now\_tree = start\_tree[fa].first\_child;

while (now\_tree->first\_brother)

now\_tree = now\_tree->first\_brother;

now\_tree->first\_brother = &start\_tree[child]; // 设置兄弟节点指针

}

else {

start\_tree[fa].first\_child = &start\_tree[child]; // 设置第一个孩子节点指针

}

}

// 找到根节点

tree\* root = &start\_tree[1];

while (root->father)

root = root->father;

// 为所有节点赋深度值

tree\*\* queue = new(nothrow) tree \* [num];

if (queue == NULL)

return NULL;

int start = 0, end = 0;

queue[end++] = root;

while (end != start) {

int tem\_num\_queue = end - start;

for (int tem\_cycle\_1 = 0; tem\_cycle\_1 < tem\_num\_queue; tem\_cycle\_1++) {

tree\* now\_tree = queue[start++];

if (now\_tree->first\_brother) {

queue[end++] = now\_tree->first\_brother;

now\_tree->first\_brother->high = now\_tree->high;

}

if (now\_tree->first\_child) {

queue[end++] = now\_tree->first\_child;

now\_tree->first\_child->high = now\_tree->high + 1;

}

}

}

return root;

}

函数 2：search

/\*\*

\* @brief 查找两个节点的最近公共祖先

\* @param start\_tree 指向节点数组的指针

\* @param the\_first 第一个节点的索引

\* @param the\_second 第二个节点的索引

\* @return 返回最近公共祖先节点的索引

\*/

int search(tree\* start\_tree, int the\_first, int the\_second) {

tree\* first = &start\_tree[the\_first];

tree\* second = &start\_tree[the\_second];

const int gap = first->high - second->high;

// 调整两个节点到相同的深度

for (int i = 0; i < gap; i++)

first = first->father;

for (int i = 0; i > gap; i--)

second = second->father;

// 向上移动直到找到公共祖先

while (first != second) {

first = first->father;

second = second->father;

}

return first - start\_tree;

}

主函数：main

/\*\*

\* @brief 主控逻辑，接收输入，构建树并处理查询

\* @return 返回程序运行状态码

\*/

int main() {

int T;

cin >> T;

for (int tem\_1 = 0; tem\_1 < T; tem\_1++) {

int N, M;

cin >> N >> M;

tree\* start = new(nothrow) tree[N + 1];

if (start == NULL)

return -1;

cin\_tree(start, N);

for (int tem\_2 = 0; tem\_2 < M; tem\_2++) {

int first, second;

cin >> first >> second;

cout << search(start, first, second) << endl;

}

}

return 0;

}

**2.5.5 调试分析（遇到的问题和解决方法）**

* 在构建树时，可能会遇到输入数据不完整或内存分配失败的情况。
* 增加错误检查，例如在分配内存后检查是否成功。
* 当调整两个节点的深度时，可能会产生无限循环。
* 确保在search函数中，两个节点在调整到同一深度后，开始同步向上移动。

**2.5.6 总结和体会**

* 本题的难点在于树结构的构建和深度的计算。多叉树在表示上较为复杂，且需要确保节点的深度信息正确。
* 在处理最近公共祖先问题时，调整两个节点到同一深度的技巧值得学习。这种方法有效避免了复杂的递归或其他复杂算法。

**2.6 求树的后续遍历**

**2.6.1 问题描述**

给出一棵二叉树的前序遍历和中序遍历，求出该树的后序遍历。输入的每行包含两个字符串，分别表示前序遍历和中序遍历。若给出的遍历序列对应存在一棵二叉树，则输出其后序遍历，否则输出“Error”。

**2.6.2 基本要求**

通过前序和中序遍历序列构建二叉树，并输出其后序遍历结果。若无法根据给定的遍历序列构建二叉树，则输出“Error”。

**2.6.3 数据结构设计**

// 定义二叉树节点的数据结构

struct tree {

char the\_num[63]; // 节点数据（字符），每个节点最多有63个字符

tree\* lchild; // 指向左子节点

tree\* rchild; // 指向右子节点

};

**2.6.4功能说明（函数、类）**

函数 1：string\_search

/\*\*

\* @brief 查找字符串中的指定字符

\* @param str 字符串

\* @param ch 待查找的字符

\* @return 字符在字符串中的索引，若未找到则返回 -1

\*/

int string\_search(const char\* str, const char ch) {

// 遍历字符串 str

for i 从 0 到 strlen(str) - 1:

if (str[i] == ch) then

return i // 找到字符，返回索引

return -1 // 未找到，返回 -1

}

函数 2：divide

/\*\*

\* @brief 递归地构建二叉树

\* @param now\_divide\_tree 当前子树的根节点

\* @param all\_tree 树的所有节点数组

\* @return 构建成功返回 0，否则返回 -1

\*/

int divide(tree\* now\_divide\_tree, tree\* all\_tree) {

if (strlen(now\_divide\_tree->the\_num) == 1) {

if (now\_divide\_tree->the\_num[0] == input\_fro[now\_top\_input\_fro++]) then

return 0 // 匹配成功，继续构建

else

return -1 // 匹配失败，返回错误

}

else {

// 查找前序序列中的当前字符在中序中的位置

int the\_place = string\_search(now\_divide\_tree->the\_num, input\_fro[now\_top\_input\_fro++])

if (the\_place == -1) then

return -1 // 若未找到，返回错误

// 递归构建左子树

if (the\_place != 0) then

now\_divide\_tree->lchild = &all\_tree[now\_tree\_num++]

strncat(now\_divide\_tree->lchild->the\_num, now\_divide\_tree->the\_num, the\_place)

if (divide(now\_divide\_tree->lchild, all\_tree) == -1) then

return -1

// 递归构建右子树

if (the\_place != (strlen(now\_divide\_tree->the\_num) - 1)) then

now\_divide\_tree->rchild = &all\_tree[now\_tree\_num++]

strncpy(now\_divide\_tree->rchild->the\_num, &now\_divide\_tree->the\_num[the\_place + 1], strlen(now\_divide\_tree->the\_num) - 1 - the\_place)

if (divide(now\_divide\_tree->rchild, all\_tree) == -1) then

return -1

// 将当前节点的 the\_num 设置为根字符

now\_divide\_tree->the\_num[0] = now\_divide\_tree->the\_num[the\_place]

now\_divide\_tree->the\_num[1] = '\0'

}

return 0

}

函数 3：output\_lat

/\*\*

\* @brief 输出二叉树的后序遍历

\* @param all\_tree 树的根节点

\*/

void output\_lat(tree\* all\_tree) {

tree\* stack[100]

int top\_stack = 0

tree \* now\_tree = all\_tree

tree \* tem\_tree = NULL

do {// 遍历左子树

while (now\_tree && now\_tree != tem\_tree) {

stack[top\_stack++] = now\_tree

now\_tree = now\_tree->lchild

}

if (top\_stack > 0) {

now\_tree = stack[--top\_stack]

if (now\_tree->rchild && now\_tree->rchild != tem\_tree) {

stack[top\_stack++] = now\_tree

now\_tree = now\_tree->rchild

}

else {

// 输出当前节点的值

print now\_tree->the\_num

tem\_tree = now\_tree

}

}

} while (top\_stack > 0)

}

主函数：main

/\*\*

\* @brief 主程序入口

\*/

int main() {

while (读取输入行) {

tree all\_tree[100]

cin >> all\_tree[0].the\_num // 读入中序遍历序列

now\_top\_input\_fro = 0

now\_tree\_num = 1

// 构建二叉树

if (divide(all\_tree, all\_tree) == -1) then

print "Error"

else

output\_lat(all\_tree) // 输出后序遍历结果

print endl // 换行

}

return 0

}

**2.6.5 调试分析（遇到的问题和解决方法）**

* 在 divide 函数中，字符串分割的位置不正确，导致树结构错误。
* 确保 the\_place 的分割位置正确，避免分割到不存在的字符。
* output\_lat 中无限循环，无法正确遍历右子树。
* 在访问节点后标记，避免重复访问已输出节点。

**2.6.6 总结和体会**

* 本题的难在二叉树的递归构建及后序遍历的非递归实现。掌握递归和栈是关键。
* 实现过程中需要注意边界条件，如空子树或单节点树的处理。

**2.7表达式树**

**2.7.1 问题描述**

给定一个中缀表达式，用变量（a-z）和运算符（+、-、\*、/、括号）表示，不含数字或空格。将此中缀表达式转换为表达式树，输出逆波兰式（后缀表达式）和树的结构图，最后计算该表达式的值（整数除法）。

**2.7.2 基本要求**

输入：中缀表达式、变量数、变量及其值

输出：逆波兰式表达式、表达式树图、计算结果

表达式中的除法为整除运算

**2.7.3 数据结构设计**

使用结构体 Node 表示表达式树的节点，包含符号、子节点指针和节点值的字段

struct Node {

int subid; // 节点在表达式中的下标

char c; // 表达式元素（变量或运算符）

int l; // 左孩子索引

int r; // 右孩子索引

int val; // 节点值（用于计算表达式结果）

};

Node tree[100]; // 模拟表达式树

优先级映射用于解析运算符优先级：

map<char, int> symblab = {

{'+', 1}, {'-', 1}, {'\*', 2}, {'/', 2}, {'(', 0}, {')', 0}

};

**2.7.4功能说明（函数、类）**

**doMath 函数**  
根据运算符执行计算，更新父节点并连接左右子树：

int doMath(stack<Node>& num, char sym, Node& fa) {

if (sym == '+') {

int num1 = num.top().val; fa.r = num.top().subid; num.pop();

int num2 = num.top().val; fa.l = num.top().subid; num.pop();

return num2 + num1;

}

else if (sym == '-') {

int num1 = num.top().val; fa.r = num.top().subid; num.pop();

int num2 = num.top().val; fa.l = num.top().subid; num.pop();

return num2 - num1;

}

else if (sym == '\*') {

int num1 = num.top().val; fa.r = num.top().subid; num.pop();

int num2 = num.top().val; fa.l = num.top().subid; num.pop();

return num1 \* num2;

}

else if (sym == '/') {

int num1 = num.top().val; fa.r = num.top().subid; num.pop();

int num2 = num.top().val; fa.l = num.top().subid; num.pop();

return num2 / num1;

}

return -1;

}

**buildTree 函数**  
根据表达式构建树，同时计算表达式的值：

int buildTree(string s) {

stack<Node> numst, symst;

for (int i = 0; i < s.size(); i++) {

Node node(i, s[i]);

if (s[i] >= 'a' && s[i] <= 'z') {

node.val = mp[s[i]];

numst.push(node);

}

else {

if (symst.empty() || s[i] == '(') symst.push(node);

else if (s[i] == ')') {

while (symst.top().c != '(') {

Node nd = symst.top(); symst.pop();

nd.val = doMath(numst, nd.c, nd);

tree[nd.subid] = nd; numst.push(nd);

}

symst.pop();

}

else if (symblab[symst.top().c] < symblab[s[i]]) {

symst.push(node);

}

else {

while (!symst.empty() && symblab[symst.top().c] >= symblab[s[i]]) {

Node nd = symst.top(); symst.pop();

nd.val = doMath(numst, nd.c, nd);

tree[nd.subid] = nd; numst.push(nd);

}

symst.push(node);

}

}

}

while (!symst.empty()) {

Node nd = symst.top(); symst.pop();

nd.val = doMath(numst, nd.c, nd);

tree[nd.subid] = nd; numst.push(nd);

}

return numst.top().val;

}

**backTranverse 函数**  
递归后序遍历表达式树，输出逆波表达式：

void backTranverse(int pos) {

if (tree[pos].l != -1) backTranverse(tree[pos].l);

if (tree[pos].r != -1) backTranverse(tree[pos].r);

cout << tree[pos].c;

}

**bfs 函数**  
使用广度优先填充输出矩阵：

void bfs(int h) {

queue<Node> q; q.push(tree[root]);

queue<pair<int, int>> cod; cod.push({ 0, (1 << (h - 1)) - 1 });

while (!q.empty()) {

Node nd = q.front(); q.pop();

pair<int, int> pr = cod.front(); cod.pop();

int t\_x = pr.first, t\_y = pr.second;

mat[t\_x][t\_y] = nd.c;

if (nd.l != -1) {

mat[t\_x + 1][t\_y - 1] = '/';

q.push(tree[nd.l]); cod.push({ t\_x + 2, t\_y - (1 << (h - t\_x / 2 - 2)) });

}

if (nd.r != -1) {

mat[t\_x + 1][t\_y + 1] = '\\';

q.push(tree[nd.r]); cod.push({ t\_x + 2, t\_y + (1 << (h - t\_x / 2 - 2)) });

}

}

}

**2.7.5 调试分析（遇到的问题和解决方法）**

**建树:**运算符优先级实现时，需确保操作数正确链接到节点，并设置左右子树。调试时注意优先级规则，防止相等优先级的误操作。

**树结构输出：**根节点及子节点位置的精确性，使用 bfs 确定每个节点的水平偏移，同时记录每行最远有效字符的横坐标。

**2.7.6 总结和体会**

学习了表达式树的构建，并了解二叉树的遍历方式如何应用到表达式解析。

数据边界和算法效率为主要难点，通过中序和后序构造表达式树拓展了栈的应用技巧。

**3. 实验总结**

**本次实验深入探讨了树的构建、遍历及应用，通过多个题目的实现，加深了对二叉树和多叉树的理解。以下是对实验中所学内容和收获的总结：**

**1.表达式树的构建与解析**

**实验中，表达式树的实现涉及到对中缀表达式的解析，将其转化为逆波兰表达式后建立表达式树。通过栈来管理操作数和操作符，使得树的构建过程符合运算优先级。此过程要求充分理解栈操作的优先级处理和中缀表达式的转化过程，进一步掌握了如何将栈与树结构结合使用。**

**2. 二叉树与多叉树的遍历和同构判断**

**通过非递归遍历二叉树以及判断两棵树的同构性，强化了对二叉树的基本操作。实验要求在实现二叉树的后序遍历时使用栈进行非递归操作，这使我更加熟悉递归与非递归之间的转换，也深入了解了栈在树遍历中的具体应用。对于同构判断，采用了递归处理左右子树交换的方案，使我对同构树的递归判断逻辑有了更深的体会。**

**3. 有序树的转化和深度计算**

**在将有序树转化为二叉树时，要求计算转化前后的深度差。通过深度优先遍历（DFS）和广度优先遍历（BFS），我理解了如何确定不同树结构的深度，特别是在有序树向二叉树转化过程中节点位置的变化。树的深度计算方法和广度优先遍历策略为以后的复杂树结构转化提供了思路。**

**4. 数据边界和特殊情况处理**

**实验中不同题目都涉及数据边界的处理，包括空树、单节点树以及不完整的输入数据。在树的构建和递归过程中，必须仔细考虑节点为空的情况，以确保不会出现空指针异常。这次实验通过各种情况的测试，强化了我对数据边界处理的意识。**

**5. 表达式树的展示和后序遍历的非递归实现**

**表达式树的展示要求通过广度优先遍历将树的结构以图形化的方式输出。这让我在树结构的存储和操作过程中，体会到如何在实现过程中通过队列管理节点的层次关系。非递归实现后序遍历也进一步巩固了我对树的栈操作的理解。**

**本次实验不仅加深了对树和表达式树的理解，也为今后解决更复杂的树结构问题奠定了基础。在解决问题的过程中，通过分析不同树的结构特性，逐步掌握了如何利用栈和队列构建和遍历树。整个实验过程锻炼了我对数据结构的思维能力和实现技巧，同时提升了代码的调试和测试能力。这次实验让我意识到在面对大规模数据结构时，设计高效的算法和实现合理的数据结构对程序性能至关重要。**