

数据结构作业 HW3 实验报告

姓名：段威呈 学号：2252109 日期：2023 年 11 月 25 日

1. 涉及数据结构和相关背景

本次实验针对树结构展开，主要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 右节点。

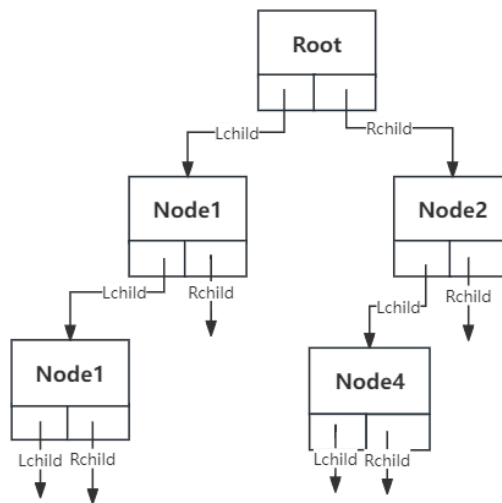


图1.1 链表实现树结构

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。

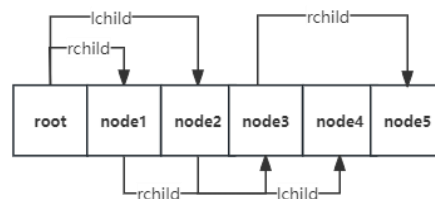


图1.2 数组模拟树结构

2. 实验内容

2.1 问题一 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。例如对于由 `abc##d##ef###` 先序建立的二叉树，可以通过如下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

①构建栈结构，用于模拟进出栈过程，以获得中序遍历的序列。

构建 class 栈 `stack`

1) 定义栈的结构

```
typedef struct {
    Elem* base;
    Elem* top;
    int stacksize;
} SqStack;
```

2) 初始化栈

```
int InitStack(SqStack* S) {
    (*S).base = (Elem*)malloc(STACK_INIT_SIZE * sizeof(Elem)); //为栈分配一段内存空间
    if (!(*S).base) exit(OVERFLOW); //分配内存失败
    (*S).top = (*S).base;
    (*S).stacksize = STACK_INIT_SIZE;
    return 1;
}
```

3) 进栈

```
int push(SqStack& S, Elem e) {
    //如果栈满了，需要另外给栈分配内存空间
    if (S.top - S.base >= S.stacksize) {
        S.base = (Elem*)realloc(S.base, (S.stacksize + STACKINCREMENT) * sizeof(Elem));
        if (!S.base) exit(OVERFLOW);
        S.top = S.base + S.stacksize; //更新指针位置
        S.stacksize += STACKINCREMENT;
    }
    *(S.top++) = e;
    return 1;
}
```

4) 出栈

```
Elem pop(SqStack& S) {
    return *--S.top;
}
```

5) 返回栈顶元素

```
Elem top(SqStack& S) {
    return *(S.top - 1);
}
```

6) 判断栈是否为空

```
int IsEmpty(SqStack S) {
    if (S.base == S.top)
        return 1; // 为空时返回1
    else
        return 0; // 不为空时返回0
}
```

③ 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.

```
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
```

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 { // 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。

① 模拟进出栈, 将序列存储到线性序列 midorder 中:

```
cin >> command;
if (command == "push") { // 如果是压栈操作
    cin >> temp.data;
    s.push(st, temp); // 把结点放到栈里面
    (preorder).push_back(temp.data);
}
if (command == "pop") { // 如果是出栈操作
    temp = s.pop(st); // 结点出栈
    (midorder).push_back(temp.data); // 出的元素放到线性表里
}
```

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

因此在前序序列中，某元素后的所有元素都是其子节点元素，且所有左节点放一起，所有右节点也放一起，如下图所示：

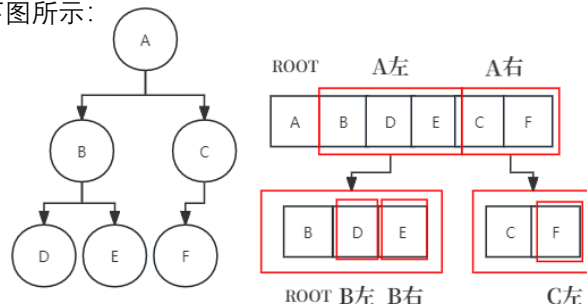


图2.1.1 前序遍历

而在每一级左、右支中，第一个元素又是该支的根结点。

因此，为了确定连接方式，首先需要确定根结点元素之后，哪部分是左支结点、哪部分是右支结点。因此这里需要借助中序遍历序列来判断；

中序遍历中，最左支结点会排在更前，如下图所示：

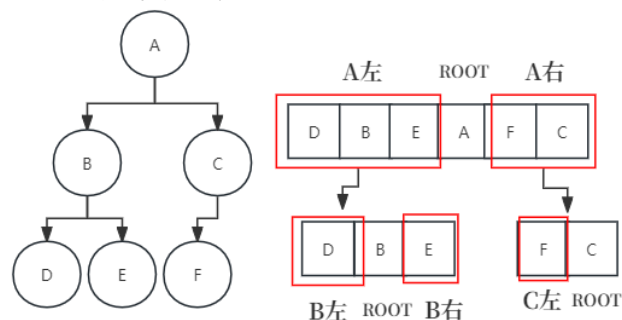


图2.1.2 中序遍历

因此只需要找出每一部分的第一个元素，即为左节点部分的最后一个元素，此后都为右结点。为了确定连接的目标的对象，需要时刻标定根结点，为了实现这一过程，可以借助进出栈模拟，通过适当的进出栈操作，保证待连接的根节点始终保持在栈顶，并根据前、后序来判断连接的是左结点还是右结点。进出栈的流程如下：

设计两个指针 i, j ，分别用于遍历前序遍历序列 $preorder$ 以及中序遍历序列 $midorder$ 。

对指针 i ，每一轮循环都需要向后遍历一个位置。首先指向第一个元素，并入栈，此即为树的根节点。之后再向后遍历一个位置，如果此时栈顶元素与中序指针 j 指向的元素不同，说明上一个遍历到的元素并不是最后一个左子结点，因此可以将现在前序指针 i 指向的元素连接到栈顶元素的左支，并入栈；

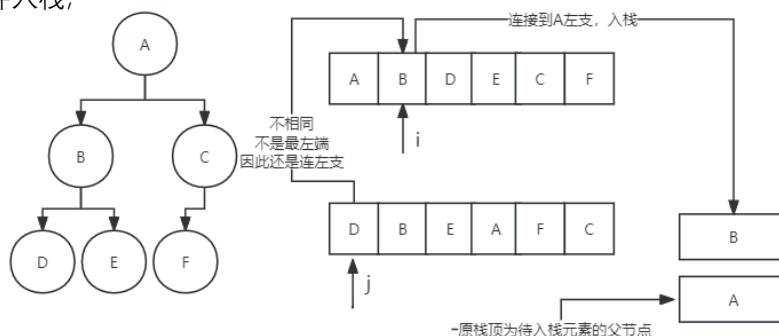


图2.1.3 入栈

同理，这时再入栈 D 元素，此时栈顶是 B，D 要连接为 B 结点的左支；

这是前序指针 i 指向 E 元素，即 E 元素将入栈。由于此时栈顶的 D 结点与中序指针所指的 D 元素相同，说明刚刚入栈的 D 结点已经是该整支 BDE 的最后一个左结点，即待入栈的 E 结点是该支的最底端的最后一个右节点；

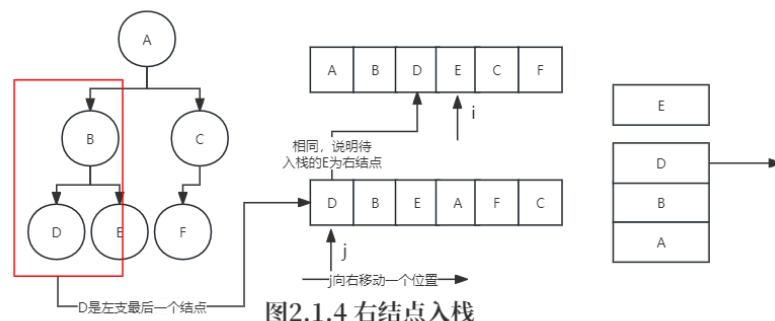


图2.1.4 右结点入栈

这时需要将栈顶元素 D 出栈，同时 j 向右移动一个位置。此时栈顶的元素为 B，同样与中序指针 j 所指的元素 B 相同，说明 B 仍是左支部分，再使 B 出栈，j 向后移动一个位置到达 E，与此时栈顶元素 A 不同，即中序遍历结束了中序“左”与“中”的部分，进入了中序“右”的遍历部分，同时也意味着刚刚出栈的 B 对于 E 而言是父节点，则使得 E 连接到 B 的右支后入栈；

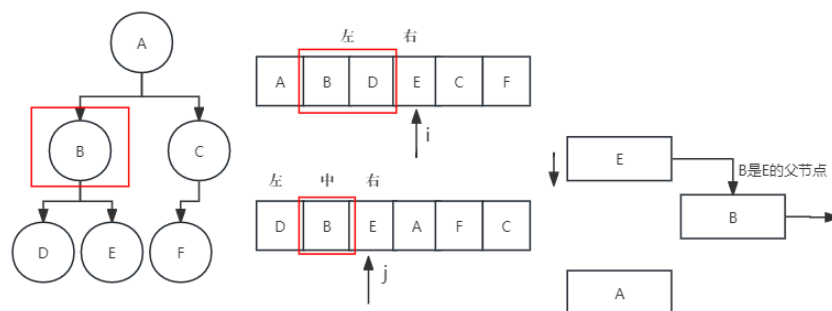


图2.1.5 右结点连接

以上过程展示了左节点以及右节点的连接条件以及出入栈过程；重复以上过程，直至前序序列遍历完成为止。

下面通过一个实例来展示完整的具体操作过程：

对于前序 ABDECF，中序 DBEAF C 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。

以上部分可以用程序编写为 CreatTree()

```
//功能函数：根据前序和中序序列用迭代法创建树
void CreatTree(Tree& tree, vector<char>preorder, vector<char>midorder) {
    stack<Node>st;
    int i = 0;//遍历先序序列的计数结点
    int j = 0;//遍历中序序列的计数结点
    int k;
    char root_elem = preorder[i];
    ((&tree)->root)->data = root_elem;//填入根节点的元素
    st.push((&tree)->root);//把结点的地址入栈保存：这样做的目的有两个，第一保存构建信息，第二保存结点地址方便直接访问结点并进行连接
    ++i;
    Node tmp_linked = ((&tree)->root);//被连接的父节点
    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，寻找该右节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;
    }
}
```

③ 用非递归方式对树进行后序遍历。

后序遍历：左->右->中

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。

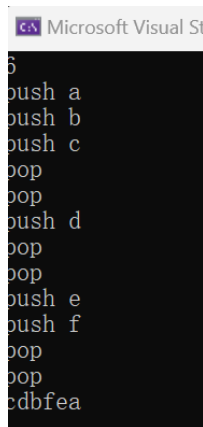
非递归方式的后序遍历为 PostTraverse()。

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

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

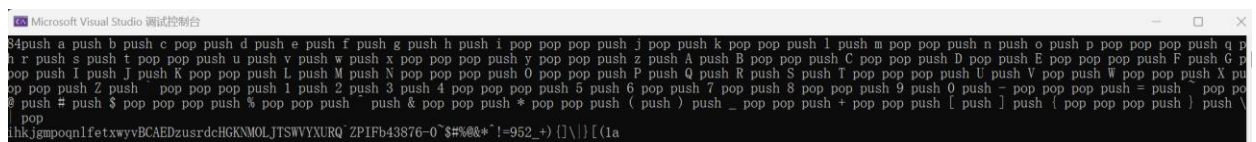
2.1.5 调试分析（遇到的问题解决方法）

运行样例数据，结果如下：



```
push a
push b
push c
pop
pop
push d
pop
pop
push e
push f
pop
pop
cd bfea
```

对于给出的测试运行数据，其中包含各类符号，如下图所示：

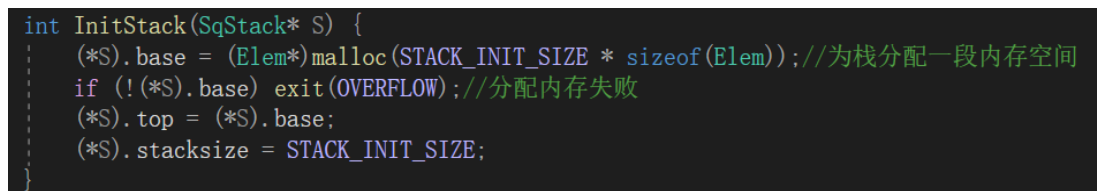


```
push a push b push c pop push d push e push f push g push h push i pop pop push j pop push k pop pop push l push m pop pop push n push o push p pop pop push q pop
r push s push t pop pop push u push v push w push x pop pop pop push y pop pop push z push A push B pop pop push C pop pop push D pop push E pop pop pop push F push G pop
push I push J push K pop pop push L push M push N pop pop pop push O pop pop push P push Q push R push S push T pop pop pop push U push V pop push W pop pop push X pu
pop pop push Z push _ pop pop pop push 1 push 2 push 3 push 4 pop pop pop push 5 push 6 pop push 7 pop push 8 pop pop push 9 push 0 push - pop pop pop push = push ^ pop po
push # push $ pop pop pop push % pop pop push & pop pop push * pop pop push ( push ) push _ pop pop push + pop pop push [ push ] push { pop pop pop push } push \
pop
ihkjgmpoqnlfetxwyvBCAEDzusrdcHGKNMOLJTSWVYXURQ`ZPIFb43876-0~$#%&*&^!=952_+){[]}\[(1a
```

与测试结果相同：

ihkjgmpoqnlfetxwyvBCAEDzusrdcHGKNMOLJTSWVYXURQ`ZPIFb43876-0~\$#%&*&^!=952_+){[]}\[(1a

多组数据测试结果均一样，但是在提交到 OJ 平台测试时所有结果均显示 RunTime Error. 经过排查，发现是因为在初始化栈函数中少写了返回值：



```
int InitStack(SqStack* S) {
    (*S).base = (Elem*)malloc(STACK_INIT_SIZE * sizeof(Elem)); //为栈分配一段内存空间
    if (!(*S).base) exit(OVERFLOW); //分配内存失败
    (*S).top = (*S).base;
    (*S).stacksize = STACK_INIT_SIZE;
}
```

加上返回值或者改为 void 运行成功。该失误由于是写在类中的，因此直接运行时 VS 不会报错，同时 VS 允许忽视返回值运行，但是在 OJ 上会出现错误。提醒在写完函数后都要特别关注返回值。

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。

本题在完成因为一个栈类中的函数忘记添加返回值，导致在 OJ 平台上无法通过，期间做了大量的排查工作、花费了许多时间，因此之后写代码要时刻记住检查函数的返回值是否添加。

2.2 感染二叉树需要的总时间

2.2.1 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。

2.2.2 基本要求

输入：第一行包含两个整数 `n` 和 `start`

接下来包含 `n` 行，描述 `n` 个节点的左、右孩子编号。

输出：一个整数，表示感染整棵二叉树所需要的时间

2.2.3 数据结构设计

1) 用数组模拟树结构。

本题中只用到了数据与数据之间的连接关系，没有用到其他树相关的逻辑思想，因此直接用数组存储树结构，通过指针指明树各结点间的连接关系即可。

① 结点元素组成

数据域:数据内容 `data`;

指针域:自己的位置编号 `i`; 父节点位置编号 `i_pre`; 左孩子结点位置编号 `i_lchild`; 右孩子结点位置编号 `i_rchild`。

```
typedef struct elem {
    int data; //结点的序号
    int i; //自己的序号
    int i_pre; //父节点的序号
    int i_lchild; //左孩子节点序号
    int i_rchild; //右孩子节点序号
};
```

② 初始化元素 `InitElem()`

```
void InitElem(elem& Elem) {
    Elem.data = -1000;
    Elem.i = -2;
    Elem.i_pre = -2;
    Elem.i_lchild = -2;
    Elem.i_rchild = -2;
}
```

③ `Tree_Search()`检索树中某元素的位置

```
//从树列表中寻找元素, 返回元素位置
int Tree_Search(elem tree[], int e, int length) {
    for (int i = 0; i <= length - 1; i++) {
        if (tree[i].data == e) {
            return tree[i].i;
        }
    }
    return -1; //不存在
}
```


3) 队列的构造

本题需要用广度优先搜索的方式遍历整棵树，从而获得感染的轮数。在 bfs 中，需要借助队列来存储每轮遍历过的结点，并在下一轮出队以遍历下一支的近邻结点。

a) 定义队列结构

```
struct queue { //定义队列的结构
    elem* base;
    int front;
    int rear;
};
```



b) 初始化队列 InitQueue()

```
bool InitQueue(queue& Q) {
    Q.base = (elem*)malloc(MAXSIZE * sizeof(elem));
    if (!Q.base) exit(-1);
    Q.front = Q.rear = 0;
    return 1;
}
```

c) 元素入队 EnQueue()

```
bool EnQueue(queue& Q, elem e) { //在队列的尾端添加元素e
    Q.base[Q.rear] = e; //Q.base[Q.rear]还可写为*(Q.base+Q.rear)
    ++Q.rear;
    return 1;
}
```

d) 元素出队 DelQueue()

```
elem DelQueue(queue& Q) { //在队列首端删除元素并返回删除的值
    elem e;
    e = Q.base[Q.front];
    ++Q.front;
    return e;
}
```

d) 判断队列是否为空 isEmpty()

```
bool isEmpty(queue& Q) {
    if (Q.front == Q.rear)
        return 1; //返回1意味着为空
    else
        return 0; //返回0不为空
}
```

2.2.4 功能说明（函数、类）

本题需要先读入、创建树，再用广度优先搜索方法遍历树。

① 创建树 Creat_Tree

由于该树是顺序二叉树，因此为了便于快速查询树中的元素位置，可以采用元素内容与下标键值对应的方式存储树中的元素，即元素内容为 i 的结点存储到数组的位置 i 处。

每次构建的时候读入第 j 行两个元素，只要元素不是 -1，则将元素 i 存储到数组中下标 i 的位置，并分别连接到 j 位置的左右孩子位置。

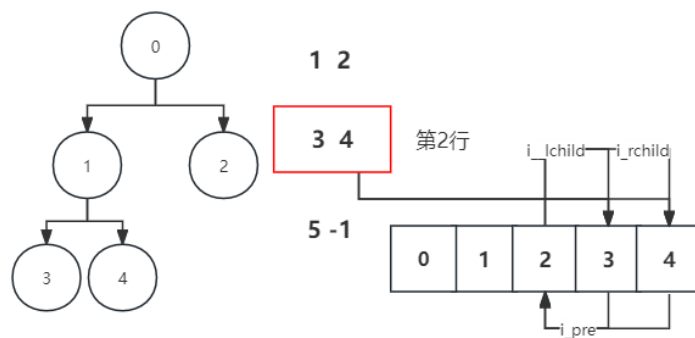


图2.2.1 创建树

Creat_Tree()创建树

```
InitElem(Elem);
switch (v) {
case 1://左节点
    cin >> input;
    if (input != -1) {
        //对于每个元素，需要确定其元素，位置，父节点位置，左节点位置，右节点位置
        Elem.data = input;
        FatherList[Elem.data] = i;
        Elem.i_pre = FatherList[count];
        Elem.i = i;
        tree[i] = Elem;
        tree[Elem.i_pre].i_lchild = i;
        //父节点的子结点指针是由下一轮决定的
        ++i;
    }
    --n;
    break;
case 2://右节点
    cin >> input;
    if (input != -1) {
        Elem.data = input;
        FatherList[Elem.data] = i;
        Elem.i_pre = FatherList[count];
        Elem.i = i;
        tree[i] = Elem;

        tree[Elem.i_pre].i_rchild = i;
        //父节点的子结点指针是由下一轮决定的
        ++i;
    }
    --n;
    break;
}
```

② 广度优先搜索 BFS

从起点开始遍历，入队，并作标记以防重复遍历；每轮出队之后再将其连接的结点入队。
每轮进行后计数器加 1；

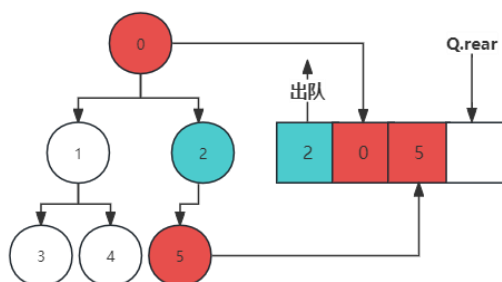


图2.2.2 广度优先搜索

每轮出队的时候都要将上一轮入队的所有结点都出队，并作已遍历标记。再将其各自连接的结点入队。计数器加 1。

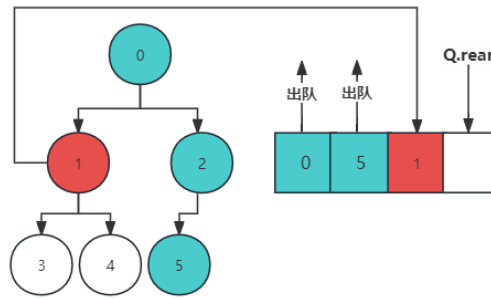


图2.2.2 广度优先搜索

以上过程为 Tree_Traverse()。

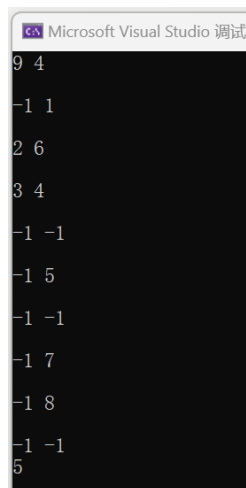
```
while (q.isEmpty(Q) != 1) {
    //一轮遍历
    for (int m = 1; m <= c2; m++) {
        Elem = q.DeQueue(Q);
        i = Elem.i;
        //cout << "感染" << Elem.data << " ";
        for (int v = 1; v <= 3; v++) {
            t.InitElem(Elem);
            switch (v) {
                case 1://左孩子
                    //不是非树区域，不是空节点，也不是遍历过的区域
                    if (tree[i].i_lchild >= 0 && tree[tree[i].i_lchild].data != -2 && tree[tree[i].i_lchild].data != -1) {
                        Elem.data = tree[tree[i].i_lchild].data;
                        Elem.i = tree[tree[i].i_lchild].i;
                        Elem.i_pre = tree[tree[i].i_lchild].i_pre;
                        Elem.i_lchild = tree[tree[i].i_lchild].i_lchild;
                        Elem.i_rchild = tree[tree[i].i_lchild].i_rchild;
                        tree[tree[i].i_lchild].data = -2;//记录已经遍历过
                        q.Enqueue(Q, Elem);
                        --n;
                        ++c1;
                    }
                    break;
                case 2://右孩子
                    if (tree[i].i_rchild >= 0 && tree[tree[i].i_rchild].data != -2 && tree[tree[i].i_rchild].data != -1) {
                        Elem.data = tree[tree[i].i_rchild].data;
                        Elem.i = tree[tree[i].i_rchild].i;
                        Elem.i_pre = tree[tree[i].i_rchild].i_pre;
                        Elem.i_lchild = tree[tree[i].i_rchild].i_lchild;
                        Elem.i_rchild = tree[tree[i].i_rchild].i_rchild;
                        tree[tree[i].i_rchild].data = -2;//记录已经遍历过
                        q.Enqueue(Q, Elem);
                        --n;
                        ++c1;
                    }
                    break;
                case 3://父结点
                    if (tree[i].i_pre >= 0 && tree[tree[i].i_pre].data != -2 && tree[tree[i].i_pre].data != -1) {
                        Elem.data = tree[tree[i].i_pre].data;
                        Elem.i = tree[tree[i].i_pre].i;
                        Elem.i_pre = tree[tree[i].i_pre].i_pre;
                        Elem.i_lchild = tree[tree[i].i_pre].i_lchild;
                        Elem.i_rchild = tree[tree[i].i_pre].i_rchild;
                        tree[tree[i].i_pre].data = -2;//记录已经遍历过
                        q.Enqueue(Q, Elem);
                        --n;
                        ++c1;
                    }
                    break;
            }
        }
    }
}
```

2.2.5 调试分析（遇到的问题和解决方法）

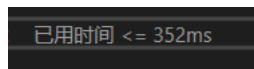
本题在一开始对结点数据直接顺序添加到数组中，在每次搜索元素的过程中需要对整个数组进行遍历，因此效率很低，在 OJ 平台上会有因为超时无法通过的测试点。

因此后来改用键值对应的存储方法，即将元素为 i 的结点直接存储到数组下标为 i 处，这样每次直接搜索元素 i 的时候直接访问 i 位置即可，时间复杂度为 $O(1)$ ，此种方法可以通过全部测试点。

测试样例数据：



利用 freopen 方法测试样例文件，总节点数为 $1e5$ 数量级也通过，耗时 $< 352ms$ 。



2.2.6 总结和体会

本题使用了 bfs 算法对整棵树进行遍历，测试样例的数量达到 $1e5$ 数量级，其中需要用到对某个结点的搜索。用数组的方法存储树的数据不但占用内存空间小，同时也可以通过键值对快速搜索元素。这里体现了数组实现树结构的优势。但是数组的方法表示树结构不够直观，需要在树构建过程中建立好结点间的连接、从属关系。

此外本题还涉及从子节点向其父节点的寻找，因此除了对每个结点设计孩子指针外，还需要单独设置一个父节点指针，用于从该结点出发寻找父结点。

2.3 题目三 树的重构

2.3.1 问题描述

有序树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 T_1, T_2, \dots, T_m , 每个子集都是一个树.

同样定义 $root(T_1), root(T_2), \dots, root(T_m)$ 为 $root(T)$ 的孩子，其中 $root(T_i)$ 是第 i 个孩子。节点 $root(T_1), \dots, root(T_m)$ 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
- 现在，需要你实现一个程序来计算转化前后的树的深度。

2.3.2 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)。

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。

2.3.3 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、二中相同。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：

a) 树结点元素的定义：

定义线性表 children，存放结点地址。通过线性表结构，可以使得该结点构建与多个子节点的关系，从而构成多叉树结构。

```
typedef struct TreeNode { //定义树结点的结构
    int data;
    vector<TreeNode*>* children; //children为链表，存放各个子节点的地址
    TreeNode* rchild;
};
```

b) 初始化多叉树 InitTree()

```
int InitTree(Tree* t) { //初始化多叉树
    if (t == NULL)
        return -1;
    Node node = (Node)malloc(sizeof(struct TreeNode)); //初始化一个结点，用Node存放其地址(Node本身就是一个指针了)
    //定义根节点
    if (node != NULL) {
        // (node->children).push_back(NULL); //初始化的时候children[0]存放NULL，也就是一个空结点
    }
    t->root = node; //根结点即为新结点
    return 0;
}
```

c) 生成结点 Makenode()

```
Node MakeNode(int i) { //创建新结点，其地址为*node，结点值为e，将该结点的!地址!返回
    Node node = new TreeNode(i);
    node->rchild = nullptr;
    return node;
}
```

d) 连接结点 be_children()

```
void be_children(Node parent, Node child) { //将child连接为parent的子节点
    parent->children->push_back(child);
}
```

2.3.4 功能说明（函数、类）

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根据深度搜索序列构建树。

深度搜索的特征是，对前后连续构建的两个结点，前者是后者的父节点。因此对于序列而言，出现 u 说明由一个结点返回上一个结点。通过此方法，可以构建结点间的从属关系。

为记录构建的结点，需要借助栈来模拟多叉树的构建过程。具体规则如下：

- 读取到 d 时，新建结点并使之入栈；
- 读取到 u 时，使得栈顶元素出栈，并作为子节点连接到当前栈顶的元素。

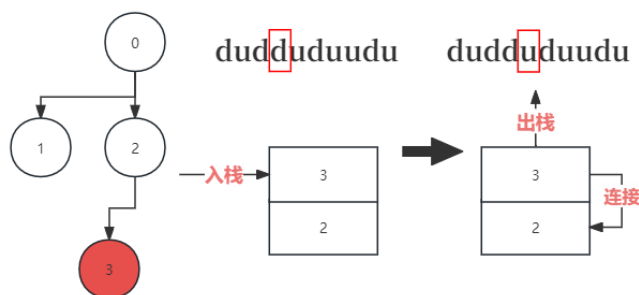


图2.3.1 深度优先搜索构建树

② 将多叉树重构为二叉树。

重构的操作，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，具体如下：

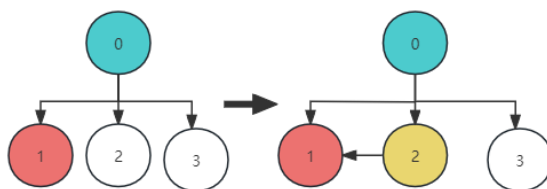


图2.3.2 重构多叉树：对第一个子节点

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。

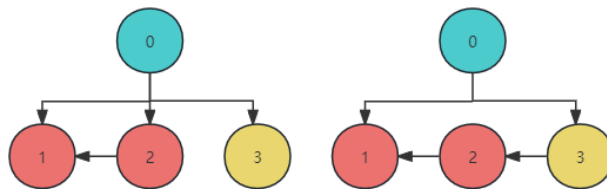


图2.3.3 重构多叉树：对其他子节点

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。

```
void ReCreat_Tree_BFS(Tree* tree) { //用广度优先搜索的方式遍历+重构
    Node tmp_node;
    queue<Node> queue;
    //q.InitQueue(queue);
    queue.push((tree->root)); //根节点入队
    while (queue.empty() != 1) {
        tmp_node = queue.front();
        queue.pop();
        //遍历子节点
        if (tmp_node->children->size() >= 1) {
            //先让子节点入队
            for (int i = 0; i < tmp_node->children->size(); i++) {
                queue.push(tmp_node->children->at(i));
            }
            for (int i = 0; i < tmp_node->children->size(); i++) {
                if (i == 0) { //对第一个子节点，只需要在新的右枝上连接其近邻的兄弟结点即可
                    if (i != tmp_node->children->size() - 1) { //首先要保证子节点有兄弟结点
                        tmp_node->children->at(i)->rchild = tmp_node->children->at(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 != tmp_node->children->size() - 1) { //首先要保证子节点有兄弟结点
                        tmp_node->children->at(i)->rchild = tmp_node->children->at(i + 1);
                    }
                    tmp_node->children->erase(tmp_node->children->begin() + i);
                }
            }
        }
    }
}
```

③ 计算树的深度

借助层序遍历的方法，对每层进行遍历。遍历到的结点入队。每轮遍历要对一层所有的结点全部出队。

```
int Calculate_Tree_Deep(Tree* tree) { //计算树的深度
    queue<Node> queue;
    // q.InitQueue(queue);
    Node tmp_node;
    //利用广度优先搜索的方式层序遍历树
    tmp_node = tree->root;
    queue.push(tmp_node);
    int deep = -1;
    int c = 1;
    int c_tmp = 0;
    while (queue.empty() != 1) {
        for (int i = 1; i <= c; i++) {
            tmp_node = queue.front();
            queue.pop();
            for (int i = 0; i < tmp_node->children->size(); i++) {
                queue.push(tmp_node->children->at(i));
                ++c_tmp;
            }
            if (tmp_node->rchild != nullptr) {
                queue.push(tmp_node->rchild);
                ++c_tmp;
            }
        }
        c = c_tmp;
        c_tmp = 0;
        ++deep;
    }
    return deep;
}
```

2.3.5 调试分析（遇到的问题 and 解决方法）

本题使用链表结构存储树的结点数据。链表由于其位置在内存空间中是非连续分布的，因此占用空间大，在提交到 OJ 平台上会有测试点出现超出内存大小的情况：

```
Test 1
ACCEPT
Test 2
ACCEPT
Test 3
Memory Limit Exceeded
Test 4
Memory Limit Exceeded
```

因此这里仍需要考虑用数组的方式模拟树结构，这样可以占用更小的内存空间。更改后所有测试点均通过。

2.3.6 总结和体会

本题主要涉及了多叉树的表示方式、深度优先搜索的方式以及对树结点重构的方法。通过此题主要掌握了利用链表、数组模拟两种方式实现多叉树，并对掌握了顺序多叉树重构为二叉树的方式。其中设计了 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。

2.4 题目四 最近公共祖先

2.4.1 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。

2.4.2 基本要求

输入：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 N_i 和 M_i

接下来 N_i-1 行，每行 2 个整数 a, b ，表示 a 是 b 的父节点

接下来 M_i 行，每行两个整数 x, y ，表示询问 x 和 y 的共同祖先

输出：对于每一个询问输出一个整数，表示共同祖先的编号。

2.4.3 数据结构设计

本题借助数组模拟了二叉树结构，并借助键值对的方式搜索树结构。具体形式同问题二感染二叉树的时间。

2.4.4 功能说明（函数、类）

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。

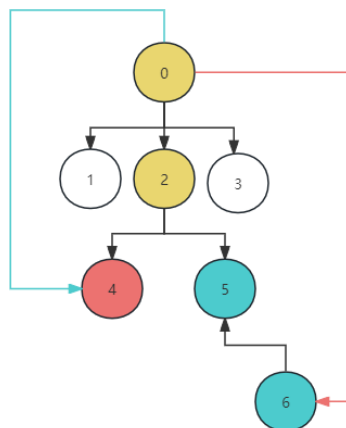


图2.4.1 求解最近公共祖先

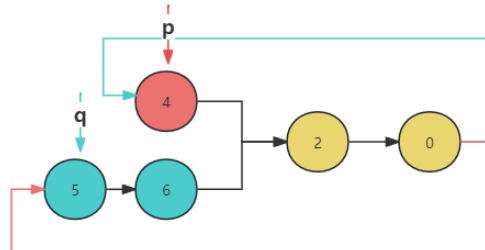


图2.4.2 转换为求第一个交点的问题

如左图，对于4、6两个结点，分别定义两个遍历指针，同时自下而上遍历。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。

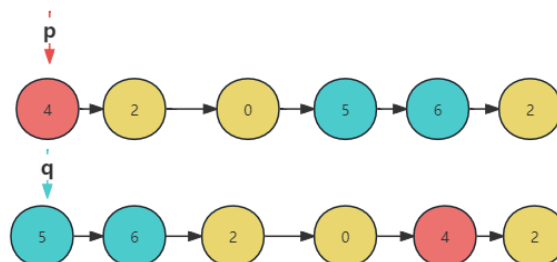


图2.4.3 结点循环遍历等价结构

算法实现：

```
int Find_Same(Elem Tree[1000],int root,int a,int b) {
    int p = a;
    int q = b;
    //双指针法判断最近祖先
    if (p == q) {
        return p; //给的是相同的点，直接返回
    }
    else {
        while (1) {
            //双指针同时走一步
            if(p!=root)
                p = Tree[p].father;
            else
                p = Tree[p].back_a;
            if (q != root)
                q = Tree[q].father;
            else
                q = Tree[q].back_b;
            if (p == q)
                break;
        }
    }
    return p;
}
```

2.4.5 调试分析（遇到的问题 and 解决方法）

本题一开始采用 cin 方式读入数据，会出现诸多不便。改用 C 语言中的 scanf 的方式可以指定读入的格式，更加方便。

2.4.6 总结和体会

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。在这种背景下，用数组方式模拟树结构会访问更加快速方便。

2.5 题目四 求树的后序

2.5.1 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。

2.5.2 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。

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。

2.5.3 数据结构设计

本题利用链表结构存储树。其方法类似问题一 非递归方法求后序。

2.5.4 功能说明（函数、类）

在问题一中展示了一种非递归方法通过前序和中序遍历序列构建二叉树的方式。这里选择使用一种递归的方法实现。以下对遍历序列的特点进行分析：

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

因此对于前序遍历序列，每支的第一个元素即为根节点：

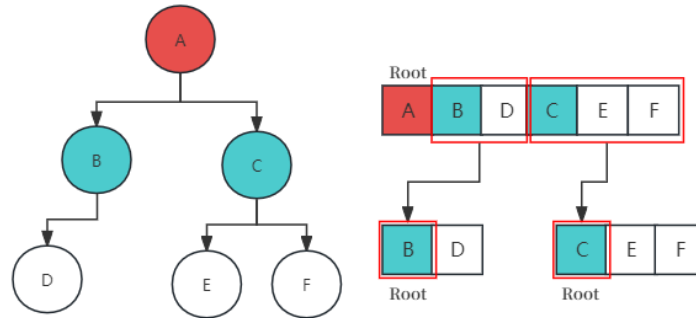


图2.5.1 前序遍历

如果在序遍历寻找对应的根结点，则其左的部分为左支元素，其右的部分为右支元素。

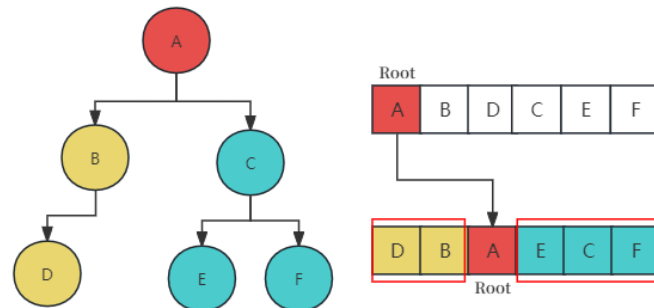


图2.5.2 前序与中序遍历的对应

因此对 A 元素，可以通过向中序查询，找出在前序序列的左支部分元素和右支部分元素。该部分元素在前序中的对应的第一个元素，即为 A 结点直接连接的孩子结点。

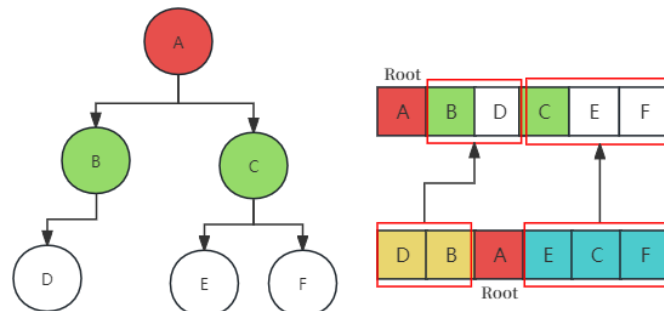


图2.5.3 父节点A连接左右孩子B、C

这时对 A 结点的连接已经完成。而对于左支 D、B 和右支 E、C、F，又可以看做一颗单独的树，重复进行以上结点连接操作。

以上过程用到了递归的思想，只需要在连接左/右支头结点后，将左/右支元素传入函数进行递归操作。

代码实现方法如下：

```
//根据中序遍历序列，创建新的先序遍历序列
//创建左支

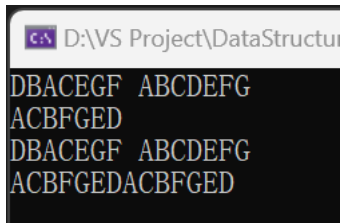
if (i > 0) { //要存在左支
    //左支先序遍历结果
    copy(temp_midorder_l, temp_preorder_l);
    Tidy(temp_preorder_l, preorder);
    //创建左结点并连接
    t.Makenode(&e_l);
    ((&tree)->root)->lchild = e_l;
    //结点保存到树
    temp_tree_l.root = e_l;
    CreatTree(temp_tree_l, temp_preorder_l, temp_midorder_l);
}

//创建右支
if (i < midorder.size() - 1) { //要存在右支
    //右支先序遍历结果
    copy(temp_midorder_r, temp_preorder_r);
    Tidy(temp_preorder_r, preorder);

    //创建右结点并连接
    t.Makenode(&e_r);
    ((&tree)->root)->rchild = e_r;
    //把结点保存到树
    temp_tree_r.root = e_r;
    CreatTree(temp_tree_r, temp_preorder_r, temp_midorder_r);
}
```

2.5.5 调试分析（遇到的问题解决方法）

本题由于要用到多次执行判断，因此在输出后序遍历序列时，要切记每次要清空此次执行的结果，否则可能会出现和上次混为一体导致的输出错误：



```
D:\VS Project\DataStructur
DBACEGF ABCDEFG
ACBFGED
DBACEGF ABCDEFG
ACBFGEDACBFGED
```

2.5.6 总结和体会

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。

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。

此外特殊的二叉树如哈夫曼树、二叉排序树、平衡二叉树虽然在此次实验报告中未涉及，但是作为重要的树结构也具有重要意义，也应当额外关注。