南京理工大学
NANJING UNIVERSITY OF SCIENCE & TECHNOLOGY

数据结构实验二报告

班　　级 9191062301

学生姓名 孙傲歆

学　　号 919106840333

题　　目 二叉树的遍历

指导教师 练智超

一、实验要求

输入是一个基于链表存储的二叉树
目标：基于非递归算法，实现一个二叉树的先序、中序以及后序遍历。

二、算法思路

在实际的代码运行中，递归算法虽然简洁易读，但效率往往很低。所以我们在实际应用中一般将递归算法转化为非递归算法。而二叉树非递归遍历算法一般都需要用到栈。

1）先序遍历

先序遍历定义为:如果二叉树为空，则遍历结束；否则:（1）访问根节点（2）先序遍历左子树（3）先序遍历右子树

非递归先序遍历算法对于任一结点 P:（1）访问结点 P，并将结点 P 入栈;（2）判断结点 P 的左孩子是否为空，若为空，则取栈顶结点并进行出栈操作，并将栈顶结点的右孩子置为当前的结点 P，循环至（1）；若不为空，则将 P 的左孩子置为当前的结点 P；（3）直到 P 为 NULL 并且栈为空，则遍历结束。

2）中序遍历

中序遍历定义为:如果二叉树为空，则遍历结束；否则:（1）中序遍历左子树（2）访问根节点（3）中序遍历右子树

非递归中序遍历算法对于任一结点 P，（1）若其左孩子不为空，则将 P 入栈并将 P 的左孩子置为当前的 P，然后对当前结点 P 再进行相同的处理；（2）若其左孩子为空，则取栈顶元素并进行出栈操作，访问该栈顶结点，然后将当前的 P 置为栈顶结点的右孩子；（3）直到 P 为 NULL 并且栈为空则遍历结束

3）后序遍历

后序遍历定义为:如果二叉树为空，则遍历结束；否则:（1）后序遍历左子树（2）后序遍历右子树（3）访问根节点

非递归后序遍历算法相较先序和中序较为复杂。在后序遍历中，要先遍历左子树，再遍历右子树，最后才可以访问根节点。所以，在工作栈记录中，结点要入栈两次，出栈两次。这两种情况的含义以及处理方法如下:

（1）第一次出栈:只遍历完左子树，右子树尚未遍历，则该节点不能访问，利用栈顶点结点找到它的右子树，准备遍历其右子树

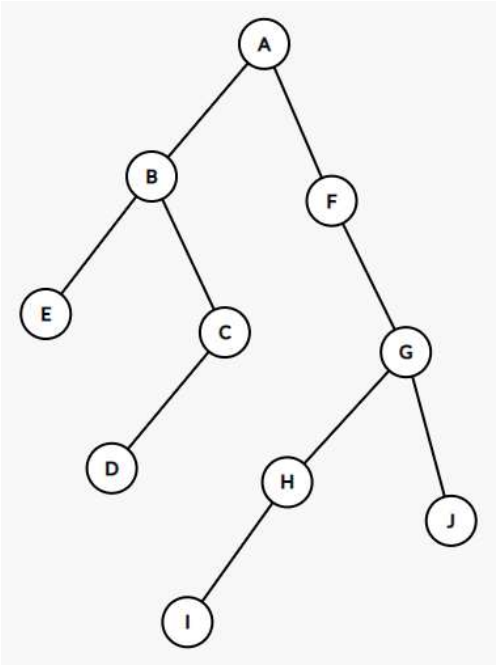
（2）第二次出栈:遍历完右子树，将该结点弹出栈，并对它进行访问。因此为了区别一个结点两次出栈，设置标志 tag，令:tag=1 第一次出栈，只遍历完左子树，该结点不能访问；tag=2 第二次出栈，已遍历完右子树，该结点可以访问。

三、代码运行结果

输入:以先序遍历顺序输入二叉树的结点，0 表示空节点
输出:依次输出二叉树的先序、中序、后序遍历结果
代码输入输出结果:

```
A
B
E
0
0
C
D
0
0
0
0
F
0
G
H
0
I
0
0
J
0
0
A B E C D F G H I J
E B D C A F H I G J
E D C B I H J G F A
```

输入二叉树对应图:



四、代码

```
using namespace std;
#include<bits/stdc++.h>
struct TreeNode
{
    char data;
```

```
    TreeNode *right;
    TreeNode *left;
};
struct TreeNode1
{
    TreeNode *tn;
    int tag;
};
TreeNode *create()
{
    char a;
    TreeNode *T;
    cin>>a;
    if(a=='0')
        T=NULL;
    else
    {
        T=new TreeNode;
        T->data=a;
        T->left=create();
        T->right=create();
    }
    return T;
}
void Preorder(TreeNode *p)
{
    stack<TreeNode *>s;
    while(p||!s.empty())
    {
        if(p)
        {
            cout<<p->data<<' ';
            s.push(p);
            p=p->left;
        }
        else
        {
            p=s.top();
            s.pop();
            p=p->right;
        }
    }
}
void Inorder(TreeNode *p)
```

```
{
    stack<TreeNode *>s;
    while(p||!s.empty())
    {
        if(p)
        {
            s.push(p);
            p=p->left;
        }
        else
        {
            p=s.top();
            s.pop();
            cout<<p->data<<' ';
            p=p->right;
        }
    }
}
void Postorder(TreeNode *p)
{
    stack<TreeNode1 *>s;
    TreeNode1 *temp;
    while(p||!s.empty())
    {
        if(p)
        {
            TreeNode1 *t=new TreeNode1;
            t->tn=p;
            t->tag=1;
            s.push(t);
            p=p->left;
        }
        else
        {
            temp=s.top();
            s.pop();
            if(temp->tag==1)
            {
                s.push(temp);
                temp->tag=2;
                p=temp->tn->right;
            }
            else
            {
```

```
                    cout<<temp->tn->data<<' ';
                    p=NULL;
                }
            }
        }
}
int main()
{
    TreeNode *T;
    T=create();

    Preorder(T);
    cout<<endl;

    Inorder(T);
    cout<<endl;

    Postorder(T);
    cout<<endl;
}
```