文章目录

一、前言

树的遍历操作需要和栈相结合，虽然C++中有许多关于数据结构的头文件可以直接拿来使用，但对于数据结构的初学者，栈的常见操作、基本原理、代码实现都应该了熟于心。
关于数据结构——栈的文章：链接: 数据结构与算法分析（C++）——栈.

二、二叉树的一些常见操作

①二叉树的存储结构

二叉树的顺序存储表示

```
/*-------二叉树的顺序存储表示------*/
#define OK 1
#define ERROR 0
#define MAXSIZE 100
typedef TElemType SqBiTree[MAXSIZE];
SqBiTree bt;
```

二叉树的二叉链表存储表示

```

/*-------二叉树的二叉链表存储表示------*/
#define OK 1
#define ERROR 0
#define MAXSIZE 100
typedef int Status;
```

```
typedef char TElemType;//定义树结点的数据类型
typedef struct BiTNode {
        TElemType data;//结点数据域
        struct BiTNode* lchild, * rchild;//结点指针域
        bool isFirst;//非递归的后序遍历用来判断某结点是否第一次出现在栈顶
}BiTNode,*BiTree;
```

②二叉树的先序遍历

先序遍历的递归算法

```
先序遍历的递归算法思路：
1：访问根结点
2：先序遍历左子树
3：先序遍历右子树
```

```
/*-------先序遍历二叉树T的递归算法---------*/
void InOrderTraverse011(BiTree T) {
        //先序遍历的递归算法
        if (T) {
                cout << T->data;//访问根结点
                InOrderTraverse011(T->lchild);//先序遍历左子树
                InOrderTraverse011(T->rchild);//先序遍历右子树
        }
}
```

先序遍历的非递归算法

```
先序遍历的非递归算法思路：
1：从根结点开始，先访问根结点，输出根结点的数据域的值
2：将根结点压入栈，以便于后续弹栈遍历右子树
3：遍历左子树且不断输出子树根结点，并将结点压入栈
4：从栈顶弹出无左子树的结点，开始遍历右子树
```

```
/*------先序遍历二叉树T的非递归算法------*/
void InOrderTraverse012(BiTree T) {
        LinkStack S = new StackNode;//定义一个链栈
        InitStack(S);//初始化此链栈
        BiTNode* p = new BiTNode;
        p = T;
        BiTNode* q = new BiTNode;//q指针用于接收出栈元素
        while (p || !StackEmpty(S)) {
```

```
                if (p) {
                        cout << p->data;//访问根结点
```

③二叉树的中序遍历

中序遍历的递归算法

```
中序遍历的递归算法思路：
1：中序遍历左子树
2：访问根结点
3：中序遍历右子树
```

```
/*-------中序遍历的递归算法---------*/
void InOrderTraverse001(BiTree T) {
        //中序遍历二叉树T的递归算法
        if (T)
        {
                InOrderTraverse001(T->lchild);//中序遍历左子树
                cout << T->data;//访问根节点
                InOrderTraverse001(T->rchild);//中序遍历右子树
        }
}
```

中序遍历的非递归算法

```
中序遍历的非递归算法思路：
1：从根结点开始，遇到结点则将结点压栈
2：当遇到无左子树的结点时，将此结点弹栈且访问它并遍历它的右子树
3：若该结点为叶子结点，则继续弹栈，开始遍历它的父节点的右子树
```

```
/*-------中序遍历的非递归算法---------*/
void InOrderTraverse002(BiTree T) {
        //中序遍历二叉树T的非递归算法
        LinkStack S = new StackNode;//定义一个链栈
        InitStack(S);//初始化此链栈
        BiTNode* p = new BiTNode;
        p = T;
        BiTNode* q = new BiTNode;//q指针用于接收出栈元素
        while (p || !StackEmpty(S)) {

```

```
11                  if (p) {//p非空
```

④二叉树的后序遍历

后序遍历的递归算法

```
后序遍历的递归算法思路：
1：后序遍历左子树
2：后序遍历右子树
3：访问根结点
```

```
/*-------后序遍历二叉树T的递归算法-------*/
void InOrderTraverse021(BiTree T) {
        if (T) {
                InOrderTraverse021(T->lchild);//后序遍历左子树
                InOrderTraverse021(T->rchild);//后序遍历右子树
                cout << T->data;//访问根结点
        }
}
```

后序遍历的非递归算法

```
后序遍历的非递归算法思路：
1：从根结点开始，沿其左子树一直往下搜索且压栈，直至出现没有左子树的结点
2：将此结点的isFirst域置为true，表面该结点第一次出现在栈顶
3：取栈顶元素并弹栈，若该栈顶元素第一次出现在栈顶则压栈，并将其isFirst域置为false，开始
4：若该栈顶元素第二次出现在栈顶则访问该结点，并将p指针置空以便继续弹栈，操作其父节点。
```

```
/*------后序遍历二叉树T的非递归算法-------*/
void InOrderTraverse022(BiTree T) {
        LinkStack S = new StackNode;//定义一个链栈
        InitStack(S);//初始化此链栈
        BiTNode* p = new BiTNode;
        p = T;
        BiTNode* q = new BiTNode;//出栈时接收栈顶元素
        BiTNode* t = new BiTNode;//接收栈顶元素
        while (p || !StackEmpty(S)) {
                while (p)//沿左子树一直往下搜索，直至出现没有左子树的结点
                {
```

⑤复制二叉树

复制二叉树 。

```
/*-----------复制二叉树-------------*/
void Copy(BiTree T, BiTree& NewT) {
        //复制一颗和T完全相同的二叉树
        if (T == NULL) {//如果是空栈，递归结束
                NewT = NULL;
                return;
        }
        else {
                NewT = new BiTNode;
                NewT->data = T->data;//复制根结点
                Copy(T->lchild, NewT->lchild);//递归复制左子树
                Copy(T->rchild, NewT->rchild);//递归复制右子树
        }
}
```

⑤计算二叉树的深度

计算二叉树的深度 。

```
/*----------计算二叉树的深度-----------*/
int Depth(BiTree T) {
        //计算二叉树T的深度
        if (T == NULL) return 0;//如果是空树，深度为0，递归结束
        else {
                int m, n;
                m = Depth(T->lchild);//递归计算左子树的深度记为m
                n = Depth(T->rchild);//递归计算右子树的深度记为n
                if (m > n) return (m + 1);//二叉树的深度为m与n的较大者加1
                else return (n + 1);
        }
}
```

⑥统计二叉树中结点的个数

统计二叉树中结点的个数 。

```
/*------------统计二叉树中结点的个数-------------*/
int NodeCount(BiTree T) {
```

```
        //统计二叉树T中结点的个数
        if (T == NULL) return 0;//如果是空树，则结点个数为0，递归结束
        else return NodeCount(T->lchild) + NodeCount(T->rchild) + 1;
        //否则结点个数为左子树的结点个数+右子树的结点个数+1
}
```

----------一道华丽的分割线----------

三、完整代码

操作二叉树的完整代码（含main函数）。

```
#include<iostream>
using namespace std;
#define OK 1
#define ERROR 0
#define MAXSIZE 100
typedef int Status;
typedef char TElemType;//定义树结点的数据类型


/*-------二叉树的二叉链表存储表示------*/
typedef struct BiTNode {
        TElemType data;//结点数据域
        struct BiTNode* lchild, * rchild;//结点指针域
        bool isFirst;//非递归的后序遍历用来判断某结点是否第一次出现在栈顶
}BiTNode,*BiTree;
typedef BiTree SElemType;
/*--------------------------------------------------------------*/
/*----------栈的存储结构-----------*/
typedef struct SqStack {
        SElemType data;//结点数据域
        struct SqStack* next;//结点的指针域
}StackNode, * LinkStack;
/*---------栈的初始化-------------*/
Status InitStack(LinkStack& S) {
        //栈的初始化
        S = NULL;
        return OK;
}
/*---------判断是否栈空-----------*/
Status StackEmpty(LinkStack& S) {
        if (S == NULL) return OK;
        return ERROR;
}
/*---------链栈的入栈-----------*/

```

```
Status Push(LinkStack& S, SElemType e) {
        StackNode* p = new StackNode;
        p->data = e;
        p->next = S;
        S = p;
        return OK;
}
/*--------链栈的出栈------------*/
Status Pop(LinkStack& S, SElemType& e) {
        if (S == NULL) return ERROR;
        e = S->data;
        StackNode* p = new StackNode;
        p = S;
        S = S->next;
        delete p;
        return OK;
}
/*---------链栈的取栈顶元素------------*/
SElemType GetTop(LinkStack& S) {
        if (S!=NULL)
        {
                return S->data;
        }
}
/*-----------------------------------------------------------------*/


/*-------中序遍历的递归算法---------*/
void InOrderTraverse001(BiTree T) {
        //中序遍历二叉树T的递归算法
        if (T)
        {
                InOrderTraverse001(T->lchild);//中序遍历左子树
                cout << T->data;//访问根节点
                InOrderTraverse001(T->rchild);//中序遍历右子树
        }
}
/*-------中序遍历二叉树T的非递归算法-------*/
void InOrderTraverse002(BiTree T) {
        //中序遍历二叉树T的非递归算法
        LinkStack S = new StackNode;//定义一个链栈
        InitStack(S);//初始化此链栈
        BiTNode* p = new BiTNode;
        p = T;
        BiTNode* q = new BiTNode;//q指针用于接收出栈元素
        while (p || !StackEmpty(S)) {
                if (p) {//p非空
                        Push(S, p);//根指针进栈
                        p = p->lchild;//遍历左子树
```

```
                }
                else {
                        Pop(S, q);//退栈
                        cout << q->data;//访问根结点
                        p = q->rchild;//遍历右子树
                }
        }
}

/*-------先序遍历二叉树T的递归算法---------*/
void InOrderTraverse011(BiTree T) {
        //先序遍历的递归算法
        if (T) {
                cout << T->data;//访问根结点
                InOrderTraverse011(T->lchild);//先序遍历左子树
                InOrderTraverse011(T->rchild);//先序遍历右子树
        }
}
/*------先序遍历二叉树T的非递归算法------*/
void InOrderTraverse012(BiTree T) {
        LinkStack S = new StackNode;//定义一个链栈
        InitStack(S);//初始化此链栈
        BiTNode* p = new BiTNode;
        p = T;
        BiTNode* q = new BiTNode;//q指针用于接收出栈元素
        while (p || !StackEmpty(S)) {
                if (p) {
                        cout << p->data;//访问根结点
                        Push(S, p);//根指针进栈
                        p = p->lchild;//遍历左子树
                }
                else {
                        Pop(S, q);//退栈
                        p = q->rchild;//遍历右子树
                }
        }
}

/*-------后序遍历二叉树T的递归算法-------*/
void InOrderTraverse021(BiTree T) {
        if (T) {
                InOrderTraverse021(T->lchild);//后序遍历左子树
                InOrderTraverse021(T->rchild);//后序遍历右子树
                cout << T->data;//访问根结点
        }
}
/*------后序遍历二叉树T的非递归算法-------*/
void InOrderTraverse022(BiTree T) {
        LinkStack S = new StackNode;//定义一个链栈
```

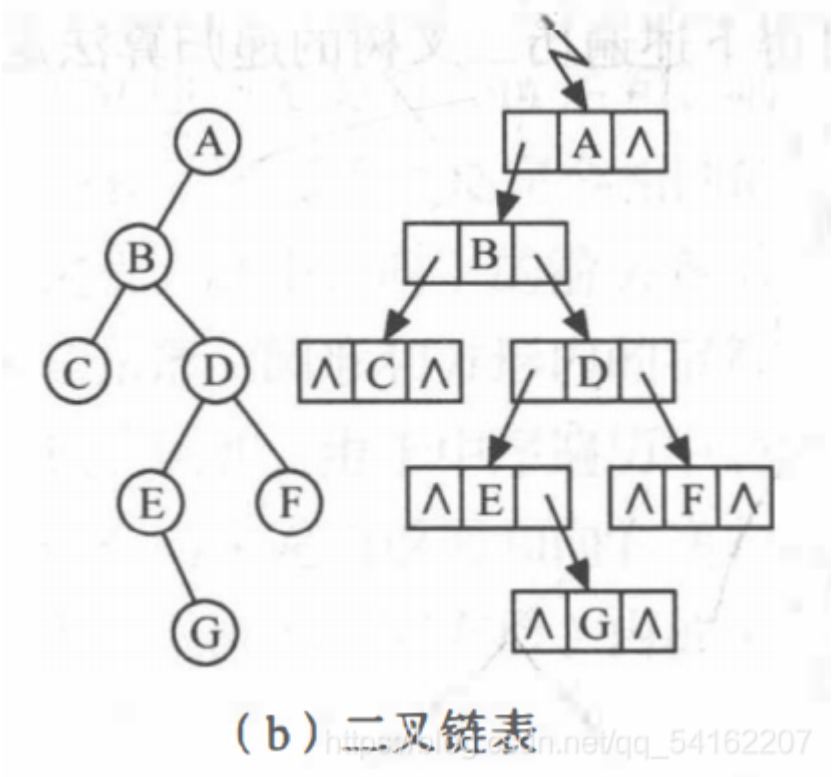
```
        InitStack(S);//初始化此链栈
        BiTNode* p = new BiTNode;
        p = T;
        BiTNode* q = new BiTNode;//出栈时接收栈顶元素
        BiTNode* t = new BiTNode;//接收栈顶元素
        while (p || !StackEmpty(S)) {
                while (p)//沿左子树一直往下搜索，直至出现没有左子树的结点
                {
                        Push(S, p);
                        p->isFirst = true;//此时结点均是第一次成为栈顶元素
                        p = p->lchild;//遍历左子树
                }
                if (!StackEmpty(S))
                {
                        t = GetTop(S);//取栈顶元素
                        Pop(S, q);//出栈
                        if (t->isFirst == true) {//若t为第一次出现在栈顶元素
                                Push(S, q);
                                t->isFirst = false;
                                p = q->rchild;//遍历栈顶元素的右子树
                        }
                        else
                        {//此时t为第二次出现在栈顶元素了
                                cout << t->data;//访问栈顶元素
                                p = NULL;//将指针置空
                        }
                }
        }
}

/*------先序遍历的顺序建立二叉链表-------*/
void CreateBiTree(BiTree& T) {
        //按先序次序输入二叉树中结点的值（一个字符）、创建二叉链表表示的二叉树T
        TElemType ch;
        cin >> ch;
        if (ch == '#') T = NULL;//递归结束，建空树
        else//递归创建二叉树
        {
                T = new BiTNode;//生成根结点
                T->data = ch;//根结点数据域置为ch
                CreateBiTree(T->lchild);//递归创建左子树
                CreateBiTree(T->rchild);//递归创建右子树
        }
}
/*-----------复制二叉树-------------*/
void Copy(BiTree T, BiTree& NewT) {
        //复制一颗和T完全相同的二叉树
        if (T == NULL) {//如果是空栈，递归结束
```

```
                NewT = NULL;
                return;
        }
        else {
                NewT = new BiTNode;
                NewT->data = T->data;//复制根结点
                Copy(T->lchild, NewT->lchild);//递归复制左子树
                Copy(T->rchild, NewT->rchild);//递归复制右子树
        }
}
/*----------计算二叉树的深度----------*/
int Depth(BiTree T) {
        //计算二叉树T的深度
        if (T == NULL) return 0;//如果是空树，深度为0，递归结束
        else {
                int m, n;
                m = Depth(T->lchild);//递归计算左子树的深度记为m
                n = Depth(T->rchild);//递归计算右子树的深度记为n
                if (m > n) return (m + 1);//二叉树的深度为m与n的较大者加1
                else return (n + 1);
        }
}
/*------------统计二叉树中结点的个数------------*/
int NodeCount(BiTree T) {
        //统计二叉树T中结点的个数
        if (T == NULL) return 0;//如果是空树，则结点个数为0，递归结束
        else return NodeCount(T->lchild) + NodeCount(T->rchild) + 1;
        //否则结点个数为左子树的结点个数+右子树的结点个数+1
}
/*-----------运行主函数-----------*/
int main()
{
        BiTree T1 = new BiTNode;
        CreateBiTree(T1);
        InOrderTraverse021(T1);
        cout << endl;
        InOrderTraverse022(T1);
        //BiTree T2 = new BiTNode;
        //Copy(T1, T2);
        //InOrderTraverse022(T2);
        return 0;
}
//a#b#cdef#####
//ABC##DE#G##F###
```

测试结果

```
输入：ABC##DE#G##F###
```

先序遍历构建的二叉树：



（b）二叉链表

先序遍历输出结果

```
输出：ABCDEGF
```

中序遍历输出结果

```
输出：CBEGDFA
```

后序遍历输出结果

```
输出：CGEFDBA
```