**实验6.1 二叉树的前序、中序非递归算法实现**

背景:

对于树的遍历，我们最常用的三种遍历方法分别是前序遍历、中序遍历和后序遍历，使用的方法是深度优先遍历树的每一个节点，这些遍历方法都是使用**递归函数**来进行实现的，在**数据量大的情况下是比较低效**的，原因在于系统帮助我们调用了一个栈并且做了诸如保护现场和恢复现场等复杂的操作，才使得遍历可以使用非常简单的代码来实现。

因此，对于树的遍历，我们可以模仿系统调用栈的过程，自义定一个栈，**用非递归方式**模仿递归中的进栈、出栈过程实现二叉树遍历。使用自定义的栈来代替系统栈，将递归改为非递归，可以得到效率上的提升。

现有如下二叉树，请用非递归的前序遍历、中序遍历 方式，将树型结构输出为线性结构。



**作业：**

1、自定义的 栈 操作代码

#include<iostream>

using namespace std;

template <typename Type>

class MyStack

{

public:

 MyStack(int size);//初始化是要指定stack的大小

 ~MyStack();

 bool isEmpty();

 bool isFull();

 int size();

 int pushStack(Type model);

 Type topStack();

 int popStack(Type\* popResult);

private:

 Type \*stk;//栈是由动态数组实现的，stk表示数组的首地址

 int top;//始终指向栈顶元素，是栈顶元素在动态数组中的下标值

 int MAXN;//栈的最大容量

};

template <typename Type>

//注意要加上尖括号模板类型

MyStack<Type>::MyStack(int size)

{

 top = -1;

 MAXN = size;

 stk = new Type[MAXN];

}

template<typename Type>

MyStack<Type>::~MyStack()

{

 delete stk;

}

template<typename Type>

bool MyStack<Type> :: isEmpty()

{

 //小学生代码

 //if (top == -1) return true;

 //return false;

 //优化

 return top == -1;

}

template<typename Type>

bool MyStack<Type> ::isFull()

{

 return top == MAXN - 1;

}

template<typename Type>

int MyStack<Type> ::size()

{

 return top+1;

}

template<typename Type>

int MyStack<Type> :: pushStack(Type model)

{

 //入栈时要判断栈已经满了，无法插入的情况，所以要带返回值

 if (isFull()) return -1;

 stk[++top] = model;

 return 0;

}

template<typename Type>

Type MyStack<Type> ::topStack()

{

 if (isEmpty()) return -1;

 return stk[top];

}

//pop失败返回整形数据int,成功返回弹出的数据，弹出的数据类型可能不是整形，

//所以这里要返回值要带两个属性1.pop的数据2.pop操作是否成功,成功可以没有固定返回，失败必须返回失败值

template<typename Type>

int MyStack<Type> ::popStack(Type\* popResult)

{

 if (isEmpty()) return -1;

 \*popResult = stk[top--];

 return 0;//可以注释

}

int main()

{

 MyStack<int>testStack(20);

 std::cout << "是否是空栈:" << testStack.isEmpty() << std::endl;

 testStack.pushStack(1);

 testStack.pushStack(2);

 testStack.pushStack(3);

 testStack.pushStack(4);

 testStack.pushStack(5);

 testStack.pushStack(6);

 std::cout << "栈内有几个元素:" << testStack.size() << std::endl;

 std::cout << "栈顶元素是:" << testStack.topStack() << std::endl;

 int temp = 0;

 testStack.popStack(&temp);

 std::cout << "弹出元素是:" << temp << std::endl;

 std::cout << "栈内有几个元素:" << testStack.size() << std::endl;

}

2、非递归的前序遍历 代码 及运行结果截图

#include<stdio.h>

#include<stdlib.h>

#include<stdbool.h>

#define QueueSize 100 // 借助队列实现层序遍历

#define StackSize 100 // 借助栈实现非递归遍历

typedef char DataType;

typedef struct BiTNode

{

 DataType data;

 struct BiTNode \* lchild;

 struct BiTNode \* rchild;

}BiTNode,\*BiTree;

// 栈元素类型

typedef struct SNode

{

 BiTNode \* bt; // 结点域

 int flag; // 标志域

}ElemType;

// 注意产生空指针

void Creat\_BiTree(BiTree\* T)

{

 char ch;

 fflush(stdin);

 scanf("%c",&ch);

 if(ch == '#')

 {

 \*T = NULL;

 return;

 }

 else

 {

 \*T = (BiTree)malloc(sizeof(BiTNode));

 (\*T)->data = ch;

 printf("请输入%c的左子树结点：", ch);

 Creat\_BiTree(&(\*T)->lchild);

 printf("请输入%c的右子树结点：", ch);

 Creat\_BiTree(&(\*T)->rchild);

 }

}

int Depth\_BiTree(BiTree T)

{

 if(!T)

 return;

 // 递归求树深度

 return Depth\_BiTree(T->lchild) > Depth\_BiTree(T->rchild) ? Depth\_BiTree(T->lchild)+1 : Depth\_BiTree(T->rchild)+1;// 深度=最大层数+1

}

int Count\_BiTree(BiTree T)

{

 if(!T)

 return;

 // 递归求树结点数

 return Count\_BiTree(T->lchild) + Count\_BiTree(T->rchild) + 1; // 树结点：左子树结点+右子树结点+根结点

}

int LeafCount\_BiTree(BiTree T)

{

 if(!T)

 return;

 if(T->lchild==NULL && T->rchild==NULL) // 左、右孩子指针域均为空时为叶子结点

 return 1;

 else

 return LeafCount\_BiTree(T->lchild) + LeafCount\_BiTree(T->rchild);

}

bool Destroy\_BiTree(BiTree T)

{

 if(!T)

 return false;

 Destroy\_BiTree(T->lchild);

 Destroy\_BiTree(T->rchild);

 free(T);

 T = NULL; // 防止产生野指针

 return true;

}

void PreOrder\_Traverse(BiTree T)

{

 if(!T)

 return;

 printf("%3c",T->data);

 PreOrder\_Traverse(T->lchild);

 PreOrder\_Traverse(T->rchild);

}

void InOrder\_Traverse(BiTree T)

{

 if(!T)

 return;

 InOrder\_Traverse(T->lchild);

 printf("%3c",T->data);

 InOrder\_Traverse(T->rchild);

}

void PostOrder\_Traverse(BiTree T)

{

 if(!T)

 return;

 PostOrder\_Traverse(T->lchild);

 PostOrder\_Traverse(T->rchild);

 printf("%3c",T->data);

}

void Level\_Traverse(BiTree T)

{

 // 初始化队列

 BiTree Q[QueueSize];

 int front = 0;

 int rear = 0;

 if(!T)

 return;

 Q[rear++] = T;

 while(front != rear)

 {

 BiTree p = Q[front++];

 printf("%3c",p->data);

 if(p->lchild!=NULL)

 Q[rear++] = p->lchild;

 if(p->rchild!=NULL)

 Q[rear++] = p->rchild;

 }

 return;

}

void Pre\_Traverse(BiTree T)

{

 BiTree p = T; // 初始化工作指针

// 将栈初始化

 BiTree Stack[StackSize];

 int top = -1;

 while(p!=NULL || top!=-1) // 两个条件都不成立时，才能退出循环

 {

 if(p!=NULL)

 {

 printf("%3c",p->data); // 访问输出结点后，将其入栈

 Stack[++top] = p;

 p = p->lchild;

 }

 else

 {

 p = Stack[top--];

 p = p->rchild;

 }

 }

}

void In\_Traverse(BiTree T)

{

 BiTree p = T; // 初始化工作指针

// 栈初始化

 BiTree Stack[StackSize];

 int top = -1;

 while(p!=NULL || top!=-1)

 {

 if(p!=NULL)

 {

 Stack[++top] = p;

 p = p->lchild;

 }

 else

 {

 p = Stack[top--];

 printf("%3c",p->data); // 出栈后，访问输出结点

 p = p->rchild;

 }

 }

}

void Post\_Traverse(BiTree T)

{

 BiTree p = T;

 ElemType Stack[StackSize];

 int top = -1;

 while(p!=NULL || top!=-1 )

 {

 if(p!=NULL)

 {

 ++top;

 Stack[top].bt = p;

 Stack[top].flag = 1;

 p = p->lchild;

 }

 else

 {

 if(Stack[top].flag==1)// 左子树遍历完，访问栈顶元素但不出栈，随后将标志域设为2，并遍历右子树

 {

 p = Stack[top].bt;

 Stack[top].flag = 2;

 p = p->rchild;

 }

 else

 {

 p = Stack[top--].bt;

 printf("%3c",p->data);

 p = NULL; // p置空，以便继续退栈

 }

 }

 }

}

int main()

{

 BiTree T;

 printf("请输入根结点：");

 Creat\_BiTree(&T);

 printf("\n");

 printf("树深度：%d\n",Depth\_BiTree(T)) ;

 printf("树结点个数：%d\n",Count\_BiTree(T));

 printf("树叶子结点个数：%d\n",LeafCount\_BiTree(T));

 printf("\n————————层序遍历算法————————\n");

 printf("层序遍历：");

 Level\_Traverse(T);

 printf("\n\n");

 printf("\n————————递归遍历算法————————\n");

 printf("递归前序遍历：");

 PreOrder\_Traverse(T);

 printf("\n\n");

 printf("递归中序遍历：");

 InOrder\_Traverse(T);

 printf("\n\n");

 printf("递归后序遍历：");

 PostOrder\_Traverse(T);

 printf("\n\n");

 printf("\n————————非递归遍历算法————————\n");

 printf("非递归前序遍历：");

 Pre\_Traverse(T);

 printf("\n\n");

 printf("非递归中序遍历：");

 In\_Traverse(T);

 printf("\n\n");

 printf("非递归后序遍历：");

 Post\_Traverse(T);

 printf("\n\n");

 return 0;

}

3、非递归的中序遍历 代码 及运行结果截图

#include<iostream>

using namespace std;

typedef struct BiNode

{

 char data;

 struct BiNode \*lchild,\*rchild;

}BiTNode,\*BiTree;

typedef struct StackNode

{

 BiTNode data;

 struct StackNode \*next;

}StackNode,\*LinkStack;

void CreateBiTree(BiTree &T)

{

 char ch;

 cin >> ch;

 if(ch=='#') T=NULL;

 else{

 T=new BiTNode;

 T->data=ch;

 CreateBiTree(T->lchild);

 CreateBiTree(T->rchild);

 }

}

void InitStack(LinkStack &S)

{

 S=NULL;

}

bool StackEmpty(LinkStack S)

{

 if(!S)

 return true;

 return false;

}

void Push(LinkStack &S,BiTree e)

{

 StackNode \*p=new StackNode;

 p->data=\*e;

 p->next=S;

 S=p;

}

void Pop(LinkStack &S,BiTree e)

{

 if(S!=NULL)

 {

 \*e=S->data;

 StackNode \*p=S;

 S=S->next;

 delete p;

 }

}

void InOrderTraverse1(BiTree T)

{

 LinkStack S; BiTree p;

 BiTree q=new BiTNode;

 InitStack(S); p=T;

 while(p||!StackEmpty(S))

 {

 if(p)

 {

 Push(S,p);

 p=p->lchild;

 }

 else

 {

 Pop(S,q);

 cout<<q->data;

 p=q->rchild;

 }

 }

}

void main()

{

 BiTree tree;

 cout<<"请输入建立二叉链表的序列：\n";

 CreateBiTree(tree);

 cout<<"中序遍历的结果为：\n";

 InOrderTraverse1(tree);

 cout<<endl;

}