江苏海洋大学计算机工程学院

实验报告书

课程名称： 《数据结构》

实验名称： 实验3：树形数据结构

第6章 实习题1

班 级： 计算机科学与技术1211

学 号： 2019120573

姓 名： 王胡鑫

评语：

成绩： 指导教师：

批阅时间： 年 月 日

1. 目的与要求

**目的**：通过本实验，学生应能理解二叉树的二叉链表存储结构，掌握二叉树遍历操作的实现方法。

**要求**：

1. 应用已学的顺序表或者链表的基本算法，用C/C++语言编程实现实习题，源程序中必需对算法、程序功能等加适当的注释；
2. 用适当的测试数据对程序进行功能测试，并生成正常运行的可执行文件；
3. 编写符合内容完整、格式规范、独立完成的实验报告；
4. 将实现的源程序、可执行文件、实验报告电子稿压缩成包，以自己姓名命名，在规定时间内提交到超星SPOC教学平台上。
5. 实验内容与题目

**内容**：掌握建立二叉树的方法，实现先序、中序、后序三种遍历算法。

**题目**：第6章 实习题1

1. 实验步骤与源程序

#include <stdio.h>

#include <stdlib.h>

#define OK 1

#define ERROR 0

#define TRUE 1

#define FALSE 0

#define OVERFLOW -1

typedef int Status;

//定义最大元素个数

#define MAXSIZE 100

typedef char TElemType;

typedef struct BiNode

{

TElemType data;

struct BiNode \*lchild, \*rchild;

} BiNode, \*BiTree;

typedef BiTree SElemType;

typedef struct SeqStack

{

SElemType \*top;

SElemType \*base;

int stacksize;

}SeqStack;

//1.顺序栈的初始化

Status InitStack(SeqStack &S)

{

//顺序栈的数组的首地址用\*base存储（即\*base指向整个栈）

S.base = (SElemType \*)malloc(sizeof(SElemType) \* MAXSIZE);

if (!(S.base)) exit(OVERFLOW);

S.top = S.base;

S.stacksize = MAXSIZE;

return OK;

}

//2.判断顺序栈是否为空

Status StackEmpty(SeqStack S)

{

if (S.top == S.base) return TRUE;

else return FALSE;

}

//3.求顺序栈的长度

int StackLength(SeqStack S)

{

return (S.top - S.base);

}

//4.清空顺序栈

Status ClearStack(SeqStack &S)

{

if (S.base) S.top = S.base;

return OK;

}

//5.销毁顺序栈

Status DestroyStack(SeqStack &S)

{

if (S.base)

{

delete S.base;

//free(S.base);

S.stacksize = 0;

S.top = S.base = NULL;

}

return OK;

}

//6.顺序栈的压栈

Status Push(SeqStack &S, SElemType e)

{

if (S.top - S.base == S.stacksize) return ERROR;//上溢

\*S.top++ = e;

//等效于 \*S.top = e;

// S.top++;

return OK;

}

//7.顺序栈的弹栈

Status Pop(SeqStack &S, SElemType &e)

{

if (StackEmpty(S)) return ERROR;//下溢

//StackEmpty(S)等价于S.top == S.base

e = \*--S.top;

//等效于 S.top--;

// e = \*S.top;

return OK;

}

//8.获取栈顶元素

Status GetTop(SeqStack S, SElemType &e)

{

if (StackEmpty(S)) return ERROR;

e = \*--S.top;

return OK;

}

// 0.访问结点

void Visit(BiTree T)

{

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

}

// 1.先序构建二叉树

Status CreateBiTree(BiTree &T)

{

TElemType ch;

ch = getchar();

if (ch == '#')//空结点

T = NULL;

else //非空结点

{

if (!(T = (BiNode \*)malloc(sizeof(BiNode)))) //非空结点分配空间

exit(OVERFLOW); //内存不够

T->data = ch; //非空结点赋值

//递归建立左右子树

CreateBiTree(T->lchild);

CreateBiTree(T->rchild);

}

return OK;

}

// 2.1递归先序遍历二叉树

Status PreOrderTraverse(BiTree T)

{

if (T == NULL)

return OK; //空二叉树

else

{

Visit(T); //访问根节点

PreOrderTraverse(T->lchild); //递归遍历左子树

PreOrderTraverse(T->rchild); //递归遍历右子树

}

return OK;

}

// 2.2递归中序遍历二叉树

Status InOrderTraverse(BiTree T)

{

if (T == NULL)

return OK;

else

{

InOrderTraverse(T->lchild);

Visit(T);

InOrderTraverse(T->rchild);

}

return OK;

}

// 2.3递归后序遍历二叉树

Status PostOrderTraverse(BiTree T)

{

if (T == NULL)

return OK;

else

{

PostOrderTraverse(T->lchild);

PostOrderTraverse(T->rchild);

Visit(T);

}

return OK;

}

// 3.1非递归先序遍历二叉树

Status PreOrderTraverse2(BiTree T)

{

BiNode \*p = T;

SeqStack S;

InitStack(S);

Push(S, p);

while (p || !StackEmpty(S))

{

if (p)

{

Visit(p);

Push(S, p);

p = p->lchild;

}

else

{

BiNode \*q;

Pop(S, q);

p = q->rchild;

}

}

return OK;

}

// 3.2非递归中序遍历二叉树

Status InOrderTraverse2(BiTree T)

{

BiNode \*p = T; //使用p指针遍历树T，初始位置为根节点

SeqStack S;

InitStack(S);

while (p || !StackEmpty(S)) //当前节点不为空且栈S不为空，继续循环

{

//p不为空，p进栈，p指向p的左子树，直到p为空，到当前树的最左端

if (p)

{

Push(S, p);

p = p->lchild;

}

//p为空，栈顶元素（当前p的双亲）出栈并访问，

//然后p指向其的右子树，对右子树进行外层相同操作

else

{

BiNode \*q;

Pop(S, q);

Visit(q);

p = q->rchild;

}

}

return OK;

}

// 3.3.1非递归后序遍历二叉树

Status PostOrderTraverse2(BiTree T)

{

BiNode \*p = T, \*q = NULL;

BiNode \*\*S;

int top = 0;

S = (BiNode \*\*)malloc(sizeof(BiNode\*) \* MAXSIZE);

while (p || top) //当前节点不为空且栈S不为空，继续循环

{

//p不为空，p进栈，p指向p的左子树，直到p为空，到当前树的最左端

while (p) {top++; S[top] = p; p = p->lchild;}

if (top > 0)

{

//将p置为当前（栈顶元素）需要判断处理的元素

p = S[top];

//无右孩子，或右孩子已经访问，（当前子树仅剩下根节点未访问），可以访问根节点

if ((p->rchild == NULL) || (p->rchild == q))

{

Visit(p); //访问根结点

q = p;

//记录该次访问的结点（下一次访问的前驱结点）

top--; //当前节点出栈

p = NULL;

//将p置为NULL

//以直接跳转至p = S[top]语句，判断处理的下一个栈中元素

//防止p再次指向当前树的最左端，对左子树重复访问

}

//然后p指向其的右子树，对右子树进行外层相同操作

else

{

p = p->rchild;

}

}

}

free(S);

return OK;

}

int main(void)

{

BiTree T;

printf("请输入二叉树的结点数据（空结点用'#'代替）：");

CreateBiTree(T);

printf("\n递归先序遍历结果：\t");

PreOrderTraverse(T);

printf("\n递归中序遍历结果：\t");

InOrderTraverse(T);

printf("\n递归后序遍历结果：\t");

PostOrderTraverse(T);

printf("\n非递归先序遍历结果：\t");

PreOrderTraverse2(T);

printf("\n非递归中序遍历结果：\t");

InOrderTraverse2(T);

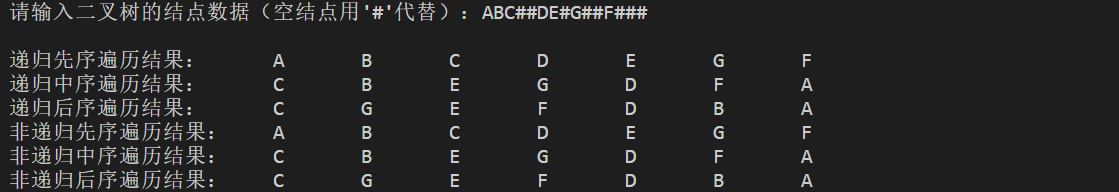
printf("\n非递归后序遍历结果：\t");

PostOrderTraverse2(T);

return 0;

}

1. 测试数据与实验结果



1. 结果分析与实验体会

对于非递归后续遍历二叉树的执行流程虽然基本了解，但是整体实现的细节不清晰，对于该算法设计任然存在问题。