|  |
| --- |
| 中南大学  《数据结构》课程实验  实验报告  实验题目 二叉树的基本操作  专业班级 软件工程2005班  学 号 8209200504  姓 名 李均浩  **实验成绩：**  **批阅教师：**  2021年4月15日 |

一、需求分析

1.程序任务

1、根据输入的数据建立一个二叉树；

2、分别采用前序、中序、后序的遍历方式显示输出二叉树的遍历结果

3、采用非递归的编程方法，分别统计二叉树的节点个数、度为1、度为2和叶子节点的个数，以及数据值的最大值和最小值。

4、按层次顺序遍历二叉树。

2.输入以及输出的形式



图1 程序输入输出形式

3.程序功能

实现对二叉树的创建、遍历、以及节点总数，叶子节点数，度为1、2的节点数的统计。

4.测试数据

(a) +\*\*/A##B##C##D##E##

预期输出：

递归方法的先序遍历的结果：

+\*\*/ABCDE

递归方法的中序遍历的结果：

A/B\*C\*D+E

递归方法的后序遍历的结果：

AB/C\*D\*E+

非递归方法的先序遍历的结果：

+\*\*/ABCDE

非递归方法的中序遍历的结果：

A/B\*C\*D+E

非递归方法的后序遍历的结果：

AB/C\*D\*E+

使用队列按层次遍历的结果：

+\*E\*D/CAB

利用递归方法查找到的二叉树的节点总数为：9

非递归方法查找到的二叉树的节点总数为：9

非递归方法查找到的二叉树度为2的节点总数为：4

非递归方法查找到的二叉树度为1的节点总数为：0

非递归方法查找到的二叉树叶子节点总数为：5

二叉树中数值最小的元素数值为：42 转换为原始类型即：\*

二叉树中数值最大的元素数值为：69 转换为原始类型即：E

(b) ab#cdefg######

递归方法的先序遍历的结果：

abcdefg

递归方法的中序遍历的结果：

bgfedca

递归方法的后序遍历的结果：

gfedcba

非递归方法的先序遍历的结果：

abcdefg

非递归方法的中序遍历的结果：

bgfedc

非递归方法的后序遍历的结果：

gfedcba

使用队列按层次遍历的结果：

abcdefg

利用递归方法查找到的二叉树的节点总数为：7

非递归方法查找到的二叉树的节点总数为：7

非递归方法查找到的二叉树度为2的节点总数为：0

非递归方法查找到的二叉树度为1的节点总数为：6

非递归方法查找到的二叉树叶子节点总数为：1

二叉树中数值最小的元素数值为：97 转换为原始类型即：a

二叉树中数值最大的元素数值为：103 转换为原始类型即：g

(c)a##

预期输出：

递归方法的先序遍历的结果：

a

递归方法的中序遍历的结果：

a

递归方法的后序遍历的结果：

a

非递归方法的先序遍历的结果：

a

非递归方法的中序遍历的结果：

a

非递归方法的后序遍历的结果：

a

使用队列按层次遍历的结果：

a

利用递归方法查找到的二叉树的节点总数为：1

非递归方法查找到的二叉树的节点总数为：1

非递归方法查找到的二叉树度为2的节点总数为：0

非递归方法查找到的二叉树度为1的节点总数为：0

非递归方法查找到的二叉树叶子节点总数为：1

二叉树中数值最小的元素数值为：97 转换为原始类型即：a

二叉树中数值最大的元素数值为：97 转换为原始类型即：a

二、概要设计

1.抽象数据类型定义：

ADT BinaryTree {

数据对象集：一个有穷的结点集合。

若不为空，则由根结点和其左、右二叉子树组成。

操作集：{BT ∈ BinTree, Item ∈ElementType}

bool IsEmpty( BinTree BT )： 判别BT是否为空；

void Traversal( BinTree BT )：遍历，按某顺序访问每个结点；

BinTree CreatBinTree( )：创建一个二叉树。

void PreOrderTraversal( BinTree BT )：先序----根、左子树、右子树；

void InOrderTraversal( BinTree BT )： 中序—左子树、根、右子树；

void PostOrderTraversal( BinTree BT )：后序—左子树、右子树、根

void LevelOrderTraversal( BinTree BT )：层次遍历，从上到下、从左到右

}

2.主程序的流程



图2 主程序的流程

三、详细设计

1.模块伪码

(1) //创建二叉树节点

Node create Begin

输出 "create() is called!" ;

Node n;

DataType temp\_var;

temp\_var = getchar();

如果(temp\_var == '#')

返回 空指针;

n = (转换为 Node)申请内存块 (大小为(TreeNode));

如果(n == 空指针)

exit(OVERFLOW);

n->data = temp\_var;

n->pre\_order\_counted\_times = 0;

n->in\_order\_isCounted = false;

n->post\_order\_isCounted = false;

n->left = create();

n->right = create();

return n;

End

(2) //元素出队

Status DelQueue(SqQueue\* Q, QElemType\* e) Begin

如果 (Q->front == Q->rear)//队列空

返回 ERROR;

\*e = Q->data[Q->front];//返回队头元素

Q->front = (Q->front + 1) % QUEUE\_MAXSIZE;//队头指针后移，如到最后转到头部

返回 SUCCESS;

End

(3) //在队尾插入元素

Status EnterQueue(SqQueue\* Q, QElemType e) Begin

如果 ((Q->rear + 1) % QUEUE\_MAXSIZE == Q->front)//队列已满

返回 ERROR;

Q->data[Q->rear] = e;//插入队尾

Q->rear = (Q->rear + 1) % QUEUE\_MAXSIZE;//尾部指针后移，如果到最后则转到头部

返回 SUCCESS;

End

(4) //返回队头元素

Status GetHead(SqQueue Q, QElemType\* e) Begin

如果 (Q.front == Q.rear)//是否为空队列

返回 ERROR;

\*e = Q.data[Q.front];

返回 SUCCESS;

End

(5) //非递归中序遍历

void in\_order\_stack\_traverse(Tree t) Begin

Stack s;

InitStack(s);

Node n = t.root;

如果 (n->left == 空指针 并且 n->right == 空指针)

{

访问(n);

返回;

}

一直循环

{

如果(n->left != 空指针&& !n->left->in\_order\_isCounted)

{

入栈(s, n);

n = n->left;

迭代;

}

否则

{

如果(!n->in\_order\_isCounted)

{

访问(n);

n->in\_order\_isCounted = true;

}

如果(n->right != 空指针&& !n->right->in\_order\_isCounted)

{

入栈(s, n);

n = n->right;

迭代;

}

出栈(s, n);

}

如果(栈为空(s))

{

如果(n->left != 空指针&& n->right != 空指针 && n->left->in\_order\_isCounted &&

n->right->in\_order\_isCounted)

返回;

if (n->left != 空指针&& n->right == 空指针 && n->left->in\_order\_isCounted)

return;

if (n->left == 空指针&& n->right != 空指针 && n->right->in\_order\_isCounted)

返回;

}

}

End

(6) //中序递归遍历

Status in\_order\_traverse(Node n) Begin

如果 (n == 空指针)

返回 NULL;

in\_order\_traverse(n->left);//递归

访问(n);

in\_order\_traverse(n->right);

返回 SUCCESS;

End

(7) //初始化空队列

Status InitQueue(SqQueue\* sQ) Begin

sQ->front = 0;

sQ->rear = 0;

返回 SUCCESS;

End

(8) //初始化一个栈

Status InitStack(Stack& s) Begin

s.base = (转换为 Elemtype\*)申请内存块 大小为(预先设定的栈初始化大小 \* sizeof(Elemtype));

如果 (s.base == 空指针)

{

弹出错误("Unable to allocate to memory space");

退出(代码为 OVERFLOW);

}

否则 {

s.top = s.base;

s.stack\_size = 预先设定的栈初始化大小;

返回 SUCCESS;

}

End

(9) //出栈

Status Pop(Stack& s, Elemtype& e) Begin

如果 (s.top == s.base)

{

返回 ERROR;

}

否则

{

s.top--;

e = \*s.top;

返回 SUCCESS;

}

End

(10) //非递归方式后序遍历

void post\_order\_stack\_traverse(Tree t) Begin

Node n;

n = t.root;

Stack s;

InitStack(s);

如果 (n->left == 空指针 && n->right == 空指针 )

{

访问(n);

返回;

}

持续循环

{

如果 (n->left != 空指针 && !n->left->post\_order\_isCounted)

{

入栈 (s, n);

n = n->left;

迭代;

}

否则

{

如果(n->right != 空指针 && !n->right->post\_order\_isCounted)

{

入栈(s, n);

n = n->right;

迭代;

}

否则

{

如果 (!n->post\_order\_isCounted)

{

访问(n);

n->post\_order\_isCounted = true;

}

出栈(s, n);

}

}

如果 (栈为空(s))

{

如果(n->left != 空指针 && n->right != 空指针 && n->left->post\_order\_isCounted &&

n->right->post\_order\_isCounted && n->post\_order\_isCounted)

返回 ;

如果 (n->left != 空指针 && n->right == 空指针 && n->left->post\_order\_isCounted && n->post\_order\_isCounted)

返回 ;

如果 (n->left == 空指针 && n->right != 空指针 && n->right->post\_order\_isCounted && n->post\_order\_isCounted)

返回 ;

}

}

End

(11) //后序递归遍历

Status post\_order\_traverse(Node n) Begin

如果 (n == 空指针 )

返回 NULL;

post\_order\_traverse(n->left);

post\_order\_traverse(n->right);

访问(n);

return SUCCESS;

End

(12) //非递归先序遍历并统计节点数据

void pre\_get\_node\_info(Tree t) Begin

Node n = t.root;

Stack node\_stack;

InitStack(node\_stack);

持续循环

{

如果(n == t.root)

{

如果(t.root->left != 空指针 && t.root->right != 空指针)

{

if (t.root->pre\_order\_counted\_times == 0)

{

max\_elem = 转换为<int>(t.root->data);//在根节点初始化最大值，最小值基准

min\_elem = 转换为<int>(t.root->data);

degree\_2\_node\_amount++;

访问(n);

node\_amount++;

t.root->pre\_order\_counted\_times++;

Push(node\_stack, t.root);

n = t.root->left;

迭代;

}

如果(t.root->pre\_order\_counted\_times == 1)

{

t.root->pre\_order\_counted\_times++;

Push(node\_stack, t.root);

n = t.root->right;

迭代;

}

如果 (t.root->pre\_order\_counted\_times == 2)

{

跳出循环;

}

}

如果(t.root->left != 空指针 && t.root->right == 空指针 )

{

如果 (t.root->pre\_order\_counted\_times == 0)

{

max\_elem = 转换为<int>(t.root->data);//在根节点初始化最大值，最小值基准

min\_elem = 转换为<int>(t.root->data);

degree\_1\_node\_amount++;

访问(n);

node\_amount++;

t.root->pre\_order\_counted\_times++;

入栈(node\_stack, t.root);

n = t.root->left;

迭代;

}

if (t.root->pre\_order\_counted\_times == 1)

{

跳出循环;

}

}

如果(t.root->left == 空指针 && t.root->right != 空指针)

{

如果(t.root->pre\_order\_counted\_times == 0)

{

max\_elem = 转换为<int>(t.root->data);//在根节点初始化最大值，最小值基准

min\_elem = 转换为<int>(t.root->data);

degree\_1\_node\_amount++;

访问(n);

node\_amount++;

t.root->pre\_order\_counted\_times++;

入栈(node\_stack, t.root);

n = t.root->right;

迭代;

}

如果(t.root->pre\_order\_counted\_times == 1)

{

跳出循环;

}

}

如果(t.root->left == 空指针 && t.root->right == 空指针)

{

max\_elem = 转换为<int>(t.root->data);//最大/最小值皆为根节点本身

min\_elem = 转换为<int>(t.root->data);

leaf\_node\_amount++;

访问(n);

node\_amount = 1;

跳出循环;

}

}

如果(n->left != nullptr && n->right != nullptr && n != t.root)

{

如果 (n->pre\_order\_counted\_times == 0)

{

如果 (static\_cast<int>(n->data) > max\_elem)

max\_elem = n->data;

如果 (static\_cast<int>(n->data) < min\_elem)

min\_elem = n->data;

degree\_2\_node\_amount++;

访问(n);

node\_amount++;

n->pre\_order\_counted\_times++;

入栈(node\_stack, n);

n = n->left;

迭代;

}

如果 (n->pre\_order\_counted\_times == 1)

{

n->pre\_order\_counted\_times++;

Push(node\_stack, n);

n = n->right;

迭代;

}

如果 (n->pre\_order\_counted\_times == 2)

{

出栈(node\_stack, n);

}

}

如果(n->left != nullptr && n->right == nullptr)

{

如果(n->pre\_order\_counted\_times == 0)

{

如果 (static\_cast<int>(n->data) > max\_elem)

max\_elem = n->data;

如果 (static\_cast<int>(n->data) < min\_elem)

min\_elem = n->data;

degree\_1\_node\_amount++;

访问(n);

node\_amount++;

n->pre\_order\_counted\_times++;

入栈(node\_stack, n);

n = n->left;

迭代;

}

如果 (n->pre\_order\_counted\_times == 1)

{

出栈(node\_stack, n);

迭代;

}

}

如果(n->left == 空指针 && n->right != 空指针)

{

如果(n->pre\_order\_counted\_times == 0)

{

如果(转换为<int>(n->data) > max\_elem)

max\_elem = n->data;

如果(转换为<int>(n->data) < min\_elem)

min\_elem = n->data;

degree\_1\_node\_amount++;

访问(n);

node\_amount++;

n->pre\_order\_counted\_times++;

入栈(node\_stack, n);

n = n->right;

迭代;

}

如果(n->pre\_order\_counted\_times == 1)

{

出栈(node\_stack, n);

迭代;

}

}

如果(n->left == 空指针 && n->right == 空指针)

{

如果(转换为<int>(n->data) > max\_elem)

max\_elem = n->data;

如果(static\_cast<int>(n->data) < min\_elem)

min\_elem = n->data;

leaf\_node\_amount++;

访问(n);

node\_amount++;

出栈(node\_stack, n);

}

}

End

(13) //先序递归遍历

Status pre\_order\_traverse(Node n) Begin

如果 (n == 空指针)

返回 NULL;

访问 (n);

pre\_order\_traverse(n->left);

pre\_order\_traverse(n->right);

返回 SUCCESS;

End

(14) //将新的元素推入栈中

Status Push(Stack& s, Elemtype e) Begin

if ((s.top - s.base) >= s.stack\_size) {//检查是否栈存满

//重新追加空间，大小为STACK\_INCREMENT

s.base = (Elemtype\*)realloc(s.base, s.stack\_size + STACK\_INCREMENT);

//检查时是否成功分配到了内存空间

if (s.base == nullptr)

{

perror("Unable to allocate to memory space");

exit(OVERFLOW);

}

//更新栈顶位置和栈大小(stack\_size)记录

s.top = s.base + s.stack\_size;

s.stack\_size = s.stack\_size + STACK\_INCREMENT;

}

//\*s.top++ = e;

\*s.top = e;

s.top++;

return SUCCESS;

End

(15) //遍历队列

Status queue\_traverse(Node n) Begin

如果 (n == root\_node && n->left == 空指针 && n->right == 空指针)

{

访问(n);

返回 SUCCESS;

}

访问(n);

如果 (n->left != 空指针)

EnterQueue(&q, n->left);

如果 (n->right != 空指针)

EnterQueue(&q, n->right);

如果 (QueueEmpty(q) && n != root\_node)

return SUCCESS;

出队(&q, &n);

queue\_traverse(n);

End

(16) //访问队列

Status queue\_visit(QElemType item) Begin

printf("%p", item);

return SUCCESS;

End

(16) //判断队列是否为空

Status QueueEmpty(SqQueue Q) Begin

如果 (Q.front == Q.rear)

返回 TRUE;

否则

返回 FALSE;

End

(17) //判断栈是否为空

Status StackEmpty(Stack s) Begin

如果 (s.base == s.top)

返回 TRUE;

否则

返回 FALSE;

End

(18) //访问二叉树节点并输出储存的数据

void visit(Node n) Begin

输出 << n->data;

End

2.函数调用关系图



图3 函数调用关系图

四、调试分析

1.问题复现

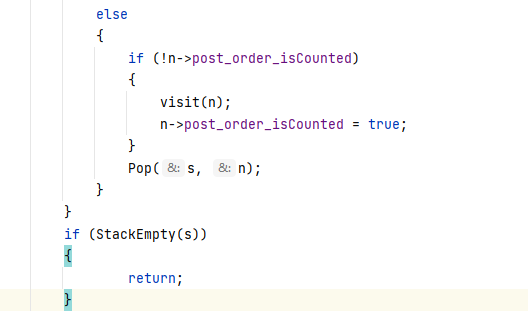
(1) 非递归后序遍历不完全

(a)错误信息





(b)错误源码



(c)错误解释

在左子树遍历完毕后，进行了出栈操作，此时栈为空，在if (StackEmpty(s))判断中return，然而此时根节点以及右子树尚未遍历。

(d)解决方案

增加if (StackEmpty(s))中的判断条件，如下图所示：



2. 算法的时空分析

(1)改进设想

部分判断条件分类可以合并，以减少操作的繁琐。

程序编写中有部分变量可以通过一定方式省去，能节省运行占用的空间。

3. 经验与体会

树与二叉树是经常会使用到的数据结构，这次的实验加深了我对二叉树的认识，让我对树的遍历有了更加深刻的理解，感受到了递归这种简洁的算法设计方式的奇妙之处。在遍历时要对情况分好类，仔细设定好判断条件，避免出现本次实验过程中的遍历不完全的问题。

五、用户使用说明

1.按照提示输入二叉树的节点数据，每个节点存储一个字符，使用’#’代表二叉树分支的终结。

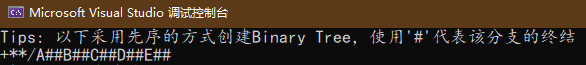


图4.1 程序输入演示

2.从控制台获得二叉树的遍历结果、度为1/2的节点数，叶子节点数，总结点数，数值最大/小元素的信息。

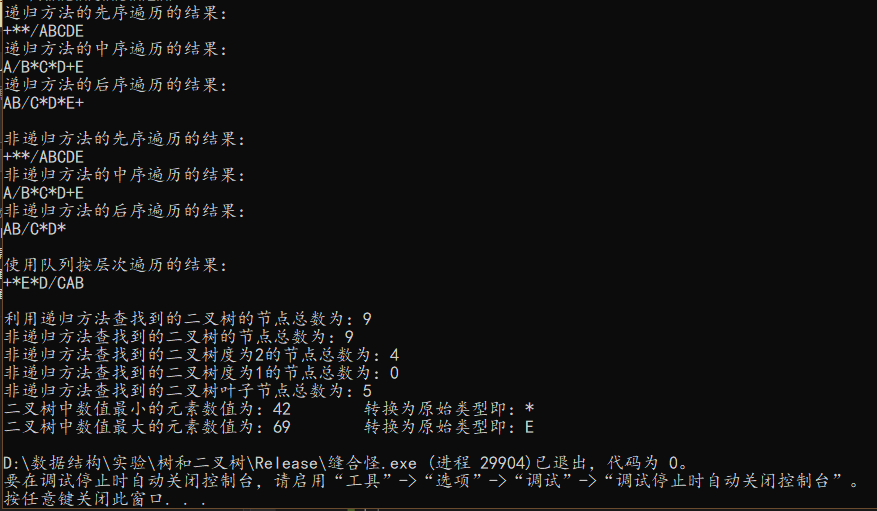
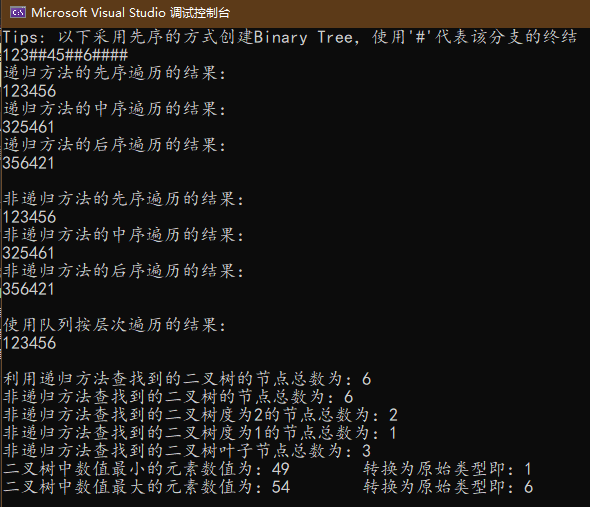


图4.2 程序运行结果

六、测试结果

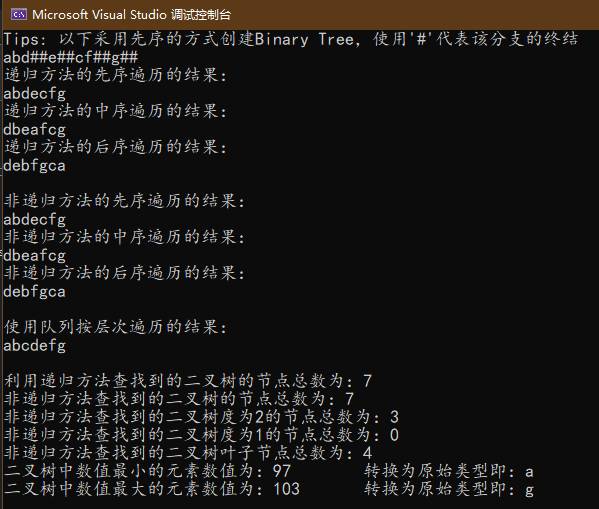
(1)输入：123##45##6####

输出：



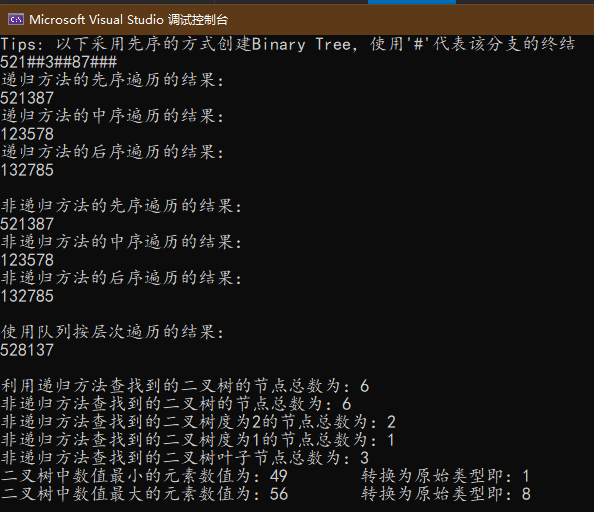
(2)输入：abd##e##cf##g##

输出：



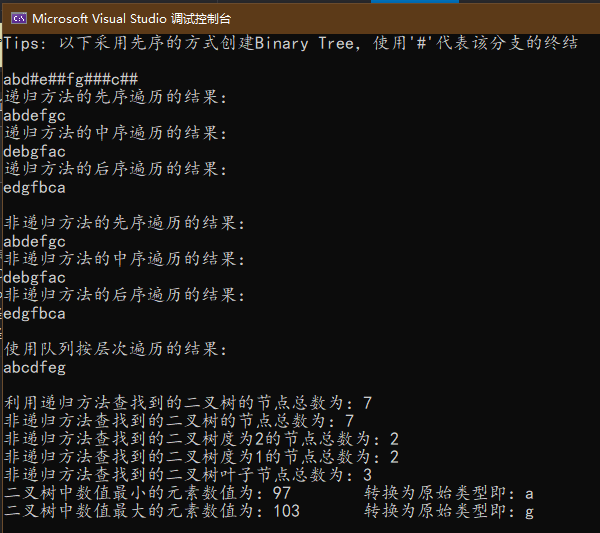
(3)输入：521##3##87###

输出：



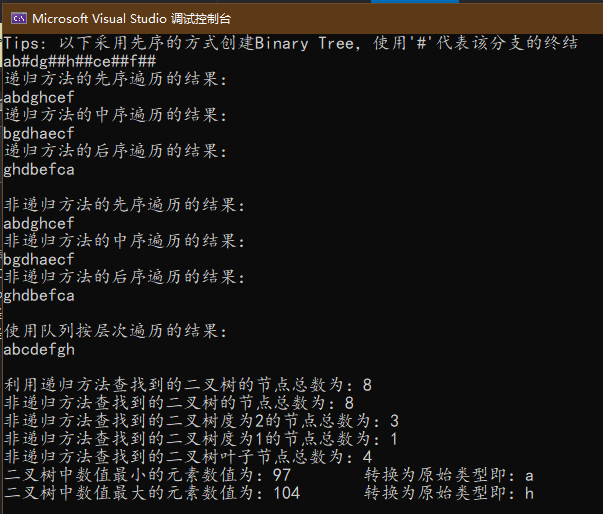
(4)输入：abd#e##fg###c##

输出：



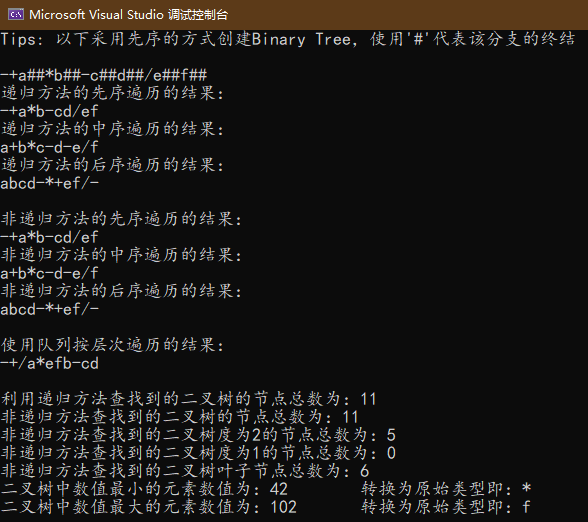
(5)输入：ab#dg##h##ce##f##

输出：



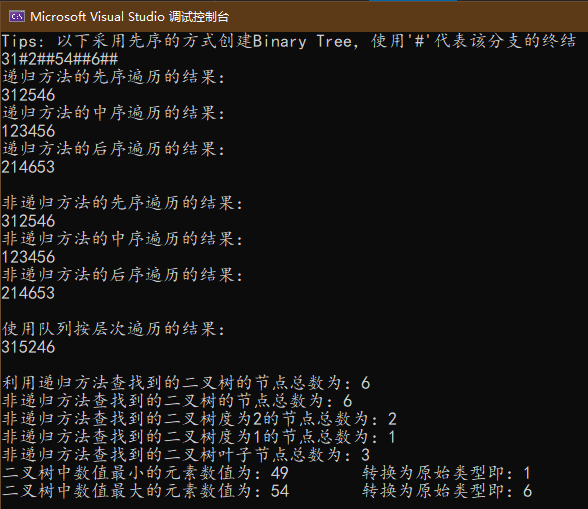
(6)输入：-+a##\*b##-c##d##/e##f##

输出：



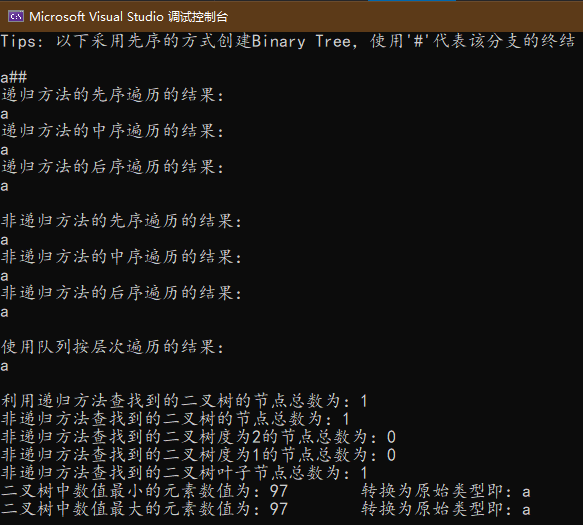
(7)输入：31#2##54##6##

输出：



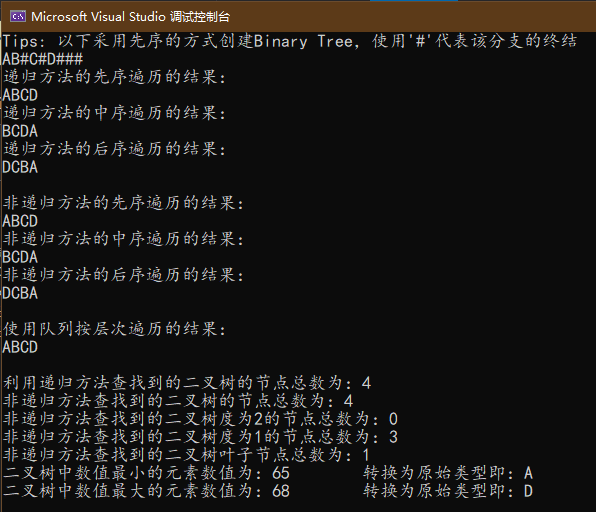
(8)输入：a##

输出：



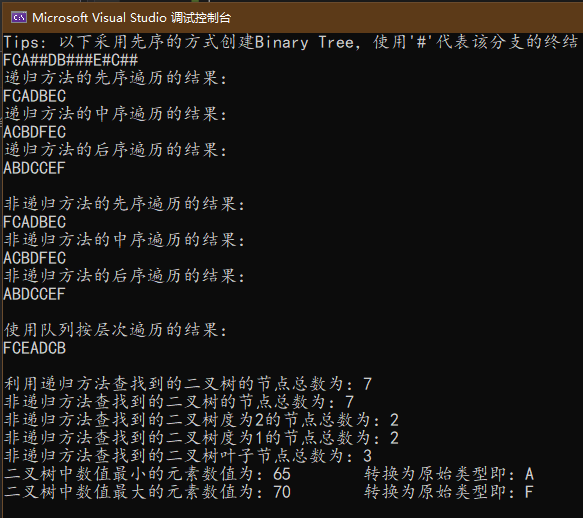
(9)输入：AB#C#D###

输出：



(10)输入：FCA##DB###E#C##

输出：



七、附录

#include <iostream>

#include <malloc.h>

#include <cstdio>

#include<cassert>

#define ERROR 0

#define SUCCESS 1

#define TRUE 1

#define FALSE 0

#define STACK\_INIT\_SIZE 300

#define STACK\_INCREMENT 10

#define QUEUE\_MAXSIZE 100

//ERROR\_EXIT\_CODE

#define UNKNOWN\_ERROR 0x474544D8

//开启DEBUG输出

#define DEBUG\_MODE\_ON

typedef int Status;

typedef char DataType;

//全局变量

int node\_amount = 0;

int degree\_1\_node\_amount = 0;

int degree\_2\_node\_amount = 0;

int leaf\_node\_amount = 0;

long long signed int max\_elem;

long long signed int min\_elem;

//二叉树节点

typedef struct TreeNode

{

TreeNode\* left;

TreeNode\* right;

DataType data;

int pre\_order\_counted\_times;

bool in\_order\_isCounted;

bool post\_order\_isCounted;

}\*Node;

//二叉树

typedef struct BinaryTree

{

Node root;

int node\_amount;

}Tree;

//循环队列的顺序存储结构

typedef Node QElemType;

typedef struct {

QElemType data[QUEUE\_MAXSIZE];

int front; //头指针

int rear;//尾指针，队列非空时，指向队尾元素的下一个位置

}SqQueue;

//访问队列

Status queue\_visit(QElemType item) {

printf("%p", item);

return SUCCESS;

}

//初始化空队列

Status InitQueue(SqQueue\* sQ) {

sQ->front = 0;

sQ->rear = 0;

return SUCCESS;

}

//将队列清空

Status ClearQueue(SqQueue\* Q) {

Q->front = Q->rear = 0;

return SUCCESS;

}

//判断队列是否为空

Status QueueEmpty(SqQueue Q) {

if (Q.front == Q.rear)

return TRUE;

else

return FALSE;

}

//返回队列中的元素个数

int QueueLength(SqQueue Q) {

return (Q.rear - Q.front + QUEUE\_MAXSIZE) % QUEUE\_MAXSIZE;

}

//返回队头元素

Status GetHead(SqQueue Q, QElemType\* e) {

if (Q.front == Q.rear)//是否为空队列

return ERROR;

\*e = Q.data[Q.front];

return SUCCESS;

}

//在队尾插入元素

Status EnterQueue(SqQueue\* Q, QElemType e) {

if ((Q->rear + 1) % QUEUE\_MAXSIZE == Q->front)//队列已满

return ERROR;

Q->data[Q->rear] = e;//插入队尾

Q->rear = (Q->rear + 1) % QUEUE\_MAXSIZE;//尾部指针后移，如果到最后则转到头部

return SUCCESS;

}

//元素出队

Status DelQueue(SqQueue\* Q, QElemType\* e) {

if (Q->front == Q->rear)//队列空

return ERROR;

\*e = Q->data[Q->front];//返回队头元素

Q->front = (Q->front + 1) % QUEUE\_MAXSIZE;//队头指针后移，如到最后转到头部

return SUCCESS;

}

//遍历队列元素

Status QueueTraverse(SqQueue Q) {

int i = Q.front;

while ((i + Q.front) != Q.rear) {

queue\_visit(Q.data[i]);

i = (i + 1) % QUEUE\_MAXSIZE;

}

printf("\n");

return SUCCESS;

}

//以下为栈模块

typedef Node Elemtype;

typedef struct SqStack

{

Elemtype\* base;

Elemtype\* top;

int stack\_size;

}Stack;

//初始化一个栈

Status InitStack(Stack& s)

{

s.base = (Elemtype\*)malloc(STACK\_INIT\_SIZE \* sizeof(Elemtype));

if (s.base == nullptr)

{

perror("Unable to allocate to memory space");

exit(OVERFLOW);

}

else {

s.top = s.base;

s.stack\_size = STACK\_INIT\_SIZE;

return SUCCESS;

}

}

//将新的元素推入栈中

Status Push(Stack& s, Elemtype e)

{

if ((s.top - s.base) >= s.stack\_size) {//检查是否栈存满

//重新追加空间，大小为STACK\_INCREMENT

s.base = (Elemtype\*)realloc(s.base, s.stack\_size + STACK\_INCREMENT);

//检查时是否成功分配到了内存空间

if (s.base == nullptr)

{

perror("Unable to allocate to memory space");

exit(OVERFLOW);

}

//更新栈顶位置和栈大小(stack\_size)记录

s.top = s.base + s.stack\_size;

s.stack\_size = s.stack\_size + STACK\_INCREMENT;

}

//\*s.top++ = e;

\*s.top = e;

s.top++;

return SUCCESS;

}

//出栈

Status Pop(Stack& s, Elemtype& e)

{

if (s.top == s.base)

{

return ERROR;

}

else

{

s.top--;

e = \*s.top;

return SUCCESS;

}

}

//判断栈是否为空

Status StackEmpty(Stack s)

{

if (s.base == s.top)

return TRUE;

else

return FALSE;

}

//创建二叉树节点

Node create()

{

//std::cout << "create() is called!" << std::endl;

Node n;

DataType temp\_var;

temp\_var = getchar();

if (temp\_var == '#')

return nullptr;

n = (Node)malloc(sizeof(TreeNode));

if (n == nullptr)

exit(OVERFLOW);

n->data = temp\_var;

n->pre\_order\_counted\_times = 0;

n->in\_order\_isCounted = false;

n->post\_order\_isCounted = false;

n->left = create();

n->right = create();

return n;

}

//访问二叉树节点并输出储存的数据

void visit(Node n)

{

std::cout << n->data;

}

//先序递归遍历

Status pre\_order\_traverse(Node n)

{

if (n == nullptr)

return NULL;

visit(n);

pre\_order\_traverse(n->left);

pre\_order\_traverse(n->right);

return SUCCESS;

}

//中序递归遍历

Status in\_order\_traverse(Node n)

{

if (n == nullptr)

return NULL;

in\_order\_traverse(n->left);

visit(n);

in\_order\_traverse(n->right);

return SUCCESS;

}

//后序递归遍历

Status post\_order\_traverse(Node n)

{

if (n == nullptr)

return NULL;

post\_order\_traverse(n->left);

post\_order\_traverse(n->right);

visit(n);

return SUCCESS;

}

SqQueue q;

Node root\_node;

//遍历队列

Status queue\_traverse(Node n)

{

if (n == root\_node && n->left == nullptr && n->right == nullptr)

{

visit(n);

return SUCCESS;

}

visit(n);

if (n->left != nullptr)

EnterQueue(&q, n->left);

if (n->right != nullptr)

EnterQueue(&q, n->right);

if (QueueEmpty(q) && n != root\_node)

return SUCCESS;

DelQueue(&q, &n);

queue\_traverse(n);

}

void count\_record()

{

node\_amount++;

}

int recursion\_get\_node\_amount(Node n)//递归做法

{

if (n == nullptr)

return NULL;

count\_record();

recursion\_get\_node\_amount(n->left);

recursion\_get\_node\_amount(n->right);

return SUCCESS;

}

//非递归先序遍历并统计节点数据

void pre\_get\_node\_info(Tree t)

{

Node n = t.root;

Stack node\_stack;

InitStack(node\_stack);

while (true)

{

if (n == t.root)

{

if (t.root->left != nullptr && t.root->right != nullptr)

{

if (t.root->pre\_order\_counted\_times == 0)

{

max\_elem = static\_cast<int>(t.root->data);//在根节点初始化最大值，最小值基准

min\_elem = static\_cast<int>(t.root->data);

degree\_2\_node\_amount++;

visit(n);

node\_amount++;

t.root->pre\_order\_counted\_times++;

Push(node\_stack, t.root);

n = t.root->left;

continue;

}

if (t.root->pre\_order\_counted\_times == 1)

{

t.root->pre\_order\_counted\_times++;

Push(node\_stack, t.root);

n = t.root->right;

continue;

}

if (t.root->pre\_order\_counted\_times == 2)

{

break;

}

}

if (t.root->left != nullptr && t.root->right == nullptr)

{

if (t.root->pre\_order\_counted\_times == 0)

{

max\_elem = static\_cast<int>(t.root->data);//在根节点初始化最大值，最小值基准

min\_elem = static\_cast<int>(t.root->data);

degree\_1\_node\_amount++;

visit(n);

node\_amount++;

t.root->pre\_order\_counted\_times++;

Push(node\_stack, t.root);

n = t.root->left;

continue;

}

if (t.root->pre\_order\_counted\_times == 1)

{

break;

}

}

if (t.root->left == nullptr && t.root->right != nullptr)

{

if (t.root->pre\_order\_counted\_times == 0)

{

max\_elem = static\_cast<int>(t.root->data);//在根节点初始化最大值，最小值基准

min\_elem = static\_cast<int>(t.root->data);

degree\_1\_node\_amount++;

visit(n);

node\_amount++;

t.root->pre\_order\_counted\_times++;

Push(node\_stack, t.root);

n = t.root->right;

continue;

}

if (t.root->pre\_order\_counted\_times == 1)

{

break;

}

}

if (t.root->left == nullptr && t.root->right == nullptr)

{

max\_elem = static\_cast<int>(t.root->data);//最大/最小值皆为根节点本身

min\_elem = static\_cast<int>(t.root->data);

leaf\_node\_amount++;

visit(n);

node\_amount = 1;

break;

}

}

if (n->left != nullptr && n->right != nullptr && n != t.root)

{

if (n->pre\_order\_counted\_times == 0)

{

if (static\_cast<int>(n->data) > max\_elem)

max\_elem = n->data;

if (static\_cast<int>(n->data) < min\_elem)

min\_elem = n->data;

degree\_2\_node\_amount++;

visit(n);

node\_amount++;

n->pre\_order\_counted\_times++;

Push(node\_stack, n);

n = n->left;

continue;

}

if (n->pre\_order\_counted\_times == 1)

{

n->pre\_order\_counted\_times++;

Push(node\_stack, n);

n = n->right;

continue;

}

if (n->pre\_order\_counted\_times == 2)

{

Pop(node\_stack, n);

}

}

if (n->left != nullptr && n->right == nullptr)

{

if (n->pre\_order\_counted\_times == 0)

{

if (static\_cast<int>(n->data) > max\_elem)

max\_elem = n->data;

if (static\_cast<int>(n->data) < min\_elem)

min\_elem = n->data;

degree\_1\_node\_amount++;

visit(n);

node\_amount++;

n->pre\_order\_counted\_times++;

Push(node\_stack, n);

n = n->left;

continue;

}

if (n->pre\_order\_counted\_times == 1)

{

Pop(node\_stack, n);

continue;

}

}

if (n->left == nullptr && n->right != nullptr)

{

if (n->pre\_order\_counted\_times == 0)

{

if (static\_cast<int>(n->data) > max\_elem)

max\_elem = n->data;

if (static\_cast<int>(n->data) < min\_elem)

min\_elem = n->data;

degree\_1\_node\_amount++;

visit(n);

node\_amount++;

n->pre\_order\_counted\_times++;

Push(node\_stack, n);

n = n->right;

continue;

}

if (n->pre\_order\_counted\_times == 1)

{

Pop(node\_stack, n);

continue;

}

}

if (n->left == nullptr && n->right == nullptr)

{

if (static\_cast<int>(n->data) > max\_elem)

max\_elem = n->data;

if (static\_cast<int>(n->data) < min\_elem)

min\_elem = n->data;

leaf\_node\_amount++;

visit(n);

node\_amount++;

Pop(node\_stack, n);

}

}

}

//非递归中序遍历

void in\_order\_stack\_traverse(Tree t)

{

Stack s;

InitStack(s);

Node n = t.root;

if (n->left == nullptr && n->right == nullptr)

{

visit(n);

return;

}

while (true)

{

if (n->left != nullptr && !n->left->in\_order\_isCounted)

{

Push(s, n);

n = n->left;

continue;

}

else

{

if (!n->in\_order\_isCounted)

{

visit(n);

n->in\_order\_isCounted = true;

}

if (n->right != nullptr && !n->right->in\_order\_isCounted)

{

Push(s, n);

n = n->right;

continue;

}

Pop(s, n);

}

if (StackEmpty(s))

{

if (n->left != nullptr && n->right != nullptr && n->left->in\_order\_isCounted &&

n->right->in\_order\_isCounted)

return;

if (n->left != nullptr && n->right == nullptr && n->left->in\_order\_isCounted && n->in\_order\_isCounted == true)

return;

if (n->left == nullptr && n->right != nullptr && n->right->in\_order\_isCounted && n->in\_order\_isCounted == true)

return;

}

}

}

//非递归方式后序遍历

void post\_order\_stack\_traverse(Tree t)

{

Node n;

n = t.root;

Stack s;

InitStack(s);

if (n->left == nullptr && n->right == nullptr)

{

visit(n);

return;

}

while (true)

{

if (n->left != nullptr && !n->left->post\_order\_isCounted)

{

Push(s, n);

n = n->left;

continue;

}

else

{

if (n->right != nullptr && !n->right->post\_order\_isCounted)

{

Push(s, n);

n = n->right;

continue;

}

else

{

if (!n->post\_order\_isCounted)

{

visit(n);

n->post\_order\_isCounted = true;

}

Pop(s, n);

}

}

if (StackEmpty(s))

{

if (n->left != nullptr && n->right != nullptr && n->left->post\_order\_isCounted &&

n->right->post\_order\_isCounted && n->post\_order\_isCounted)

return;

if (n->left != nullptr && n->right == nullptr && n->left->post\_order\_isCounted && n->post\_order\_isCounted)

return;

if (n->left == nullptr && n->right != nullptr && n->right->post\_order\_isCounted && n->post\_order\_isCounted)

return;

}

}

}

//测试用例:

//Input:

//+\*\*/A##B##C##D##E##

//InOrder

//A/B\*C\*D+E

//PostOrder

//AB/C\*D\*E+

int main()

{

//初始化一颗树（先序创建）

Tree test\_tree;

std::cout << "Tips: 以下采用先序的方式创建Binary Tree，使用\'#\'代表该分支的终结" << std::endl;

test\_tree.root = create();

//递归方法遍历

std::cout << "递归方法的先序遍历的结果：" << std::endl;

pre\_order\_traverse(test\_tree.root);

std::cout << std::endl;

std::cout << "递归方法的中序遍历的结果：" << std::endl;

in\_order\_traverse(test\_tree.root);

std::cout << std::endl;

std::cout << "递归方法的后序遍历的结果：" << std::endl;

post\_order\_traverse(test\_tree.root);

std::cout << std::endl << std::endl;

//非递归方法遍历

std::cout << "非递归方法的先序遍历的结果：" << std::endl;

pre\_get\_node\_info(test\_tree);

std::cout << std::endl;

std::cout << "非递归方法的中序遍历的结果：" << std::endl;

in\_order\_stack\_traverse(test\_tree);

std::cout << std::endl;

std::cout << "非递归方法的后序遍历的结果：" << std::endl;

post\_order\_stack\_traverse(test\_tree);

std::cout << std::endl << std::endl;

//队列按层次遍历

InitQueue(&q);

root\_node = test\_tree.root;

std::cout << "使用队列按层次遍历的结果：" << std::endl;

queue\_traverse(test\_tree.root);

std::cout << std::endl << std::endl;

//输出节点统计信息

std::cout << "利用递归方法查找到的二叉树的节点总数为：" << node\_amount << std::endl;

std::cout << "非递归方法查找到的二叉树的节点总数为：" << node\_amount << std::endl;

std::cout << "非递归方法查找到的二叉树度为2的节点总数为：" << degree\_2\_node\_amount << std::endl;

std::cout << "非递归方法查找到的二叉树度为1的节点总数为：" << degree\_1\_node\_amount << std::endl;

std::cout << "非递归方法查找到的二叉树叶子节点总数为：" << leaf\_node\_amount << std::endl;

std::cout << "二叉树中数值最小的元素数值为：" << static\_cast<int> (min\_elem) << '\t' << \

"转换为原始类型即：" << static\_cast<DataType>(min\_elem) << std::endl;

std::cout << "二叉树中数值最大的元素数值为：" << static\_cast<int> (max\_elem) << '\t' << \

"转换为原始类型即：" << static\_cast<DataType>(max\_elem) << std::endl;

return 0;

}