**《算法与数据结构》**

**实 验 指 导 手 册**

# 北京邮电大学软件学院

# 2020-2021学年第1学期实验报告

**课程名称： 算法与数据结构**

**实验名称： 实验四、树及其应用**

**实验完成人：**

**姓名：**\_\_\_王宇涵\_\_\_\_\_**学号：**\_\_\_\_2020211730\_\_\_\_**成绩：**\_\_\_\_\_\_\_\_

**指导教师：**\_\_\_\_\_\_\_**\_\_贾红娓** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**日 期： 2021 年 11 月 8 日**

1. **实验目的**

树是一种应用极为广泛的数据结构，也是这门课程的重点。它们的特点在于非线性。广义表本质上是树结构。本章实验继续突出了数据结构加操作的程序设计观点，但根据这两种结构的非线性特点，将操作进一步集中在遍历操作上，因为遍历操作是其他众多操作的基础。遍历逻辑的（或符号形式的）结构，访问动作可是任何操作。本次实验希望帮助学生熟悉各种存储结构的特征，以及如何应用树结构解决具体问题（即原理与应用的结合）。

1. **实验内容**

**必做内容**

* + 1. **二叉树的建立与遍历**

**[问题描述]**

　　建立一棵二叉树，并对其进行遍历（先序、中序、后序），打印输出遍历结果。

**[基本要求]**

　　从键盘接受输入（先序），以二叉链表作为存储结构，建立二叉树（以先序来建立），并采用递归算法对其进行遍历（先序、中序、后序），将遍历结果打印输出。

**[测试数据]**

ABCффDEфGффFффф（其中ф表示空格字符）则输出结果为

先序：ABCDEGF

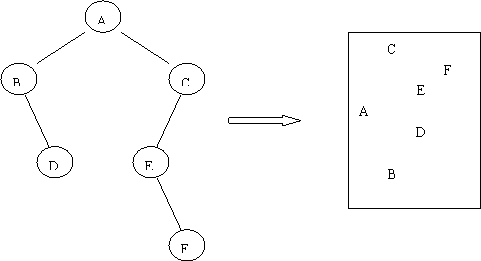
　　中序：CBEGDFA

　　后序：CGBFDBA

* + 1. **打印二叉树结构**

**[问题描述]**

　　按凹入表形式横向打印二叉树结构，即二叉树的根在屏幕的最左边，二叉树的左子树在屏幕的下边，二叉树的右子树在屏幕的上边。例如：



**[测试数据]**

　　由学生依据软件工程的测试技术自己确定。注意测试边界数据，如空二叉树。

**[实现提示]**

　　（1）利用RDL遍历方法；

（2）利用结点的深度控制横向位置。

**选做内容**

**采用非递归算法实现二叉树遍历。**

1. **实验环境**

VC6.0

1. **实验过程和实验结果**

**1.二叉树的建立与遍历**

**1.1问题分析**

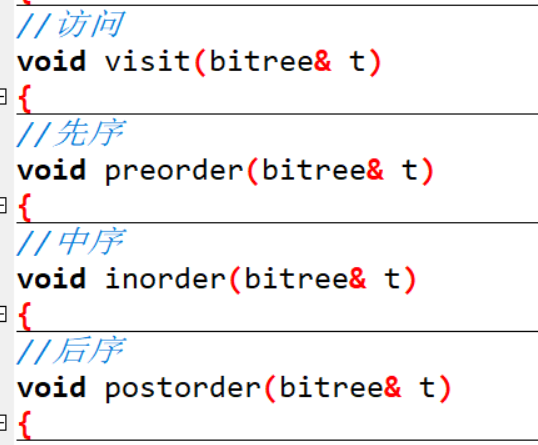
对于二叉树来说，每个节点最多有两个孩子，采用链式方式存储，每个结点的值域有数据域（data），两个指针域左海子（lchild）右孩子（rchild）。对二叉树先序，中序，后序遍历，可以采用递归思想。递归思想：就是把这个大树拆分成N棵小树，每棵小树都进行一次先序遍历。再把这些遍历连合起来就是这棵树的先序遍历了。可以改变访问根节点的顺序来实施遍历。

**1.2设计方案**

**建立二叉树：每个节点最多有两个孩子，采用链式方式存储，每个结点的值域有数据域（data），两个指针域左海子（lchild）右孩子（rchild）。而本题中采用先序遍历，先接受根节点在接受左节点和右节点，同样是通过递归完成建立。使用创建树的函数**

**遍历输出：采用三个不同函数**

1. **先序遍历的函数**
2. **中序遍历的函数**
3. **后序遍历的函数**
4. **访问函数**

****

**1.3算法**

//创建树

void cbtree(bitree &t)

{

scanf("%c",&ch); // abdg e c f

if (ch==' ') t=NULL;

else

{

t=(bitnode \*) malloc(sizeof (bitnode));

/\*if(!t)

{

return false;

} \*/

t->data = ch ;

//cout<<t->data<<endl;

cbtree(t->lchild);

cbtree(t->rchild);

}

//return true;

}

//访问

void visit(bitree& t)

{

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

}

//先序

void preorder(bitree& t)

{

if(t!=NULL)

{

visit(t);//访问根节点

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

preorder(t->lchild);//访问左子树

preorder(t->rchild); //访问右子树

}

}

//中序

void inorder(bitree& t)

{

if(t!=NULL)

{

inorder(t->lchild);//访问左子树

visit(t);//访问根节点

inorder(t->rchild);//访问右子树

}

}

//后序

void postorder(bitree& t)

{

if(t!=NULL)

{

postorder(t->lchild);//访问左子树

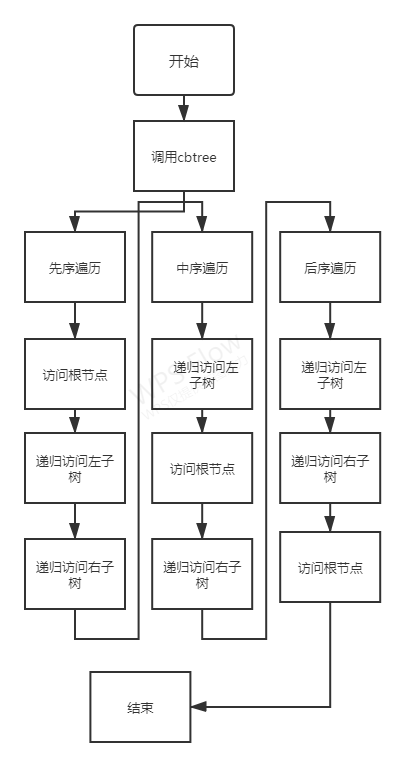
postorder(t->rchild);//访问右子树

visit(t);//访问根节点

}

}

**1.4设计图**



**1.5程序**

**#include<cstdio>**

**#include<cstdlib>**

**#include<cstring>**

**#include<iostream>**

**#include<cmath>**

**#define MAXSIZE 100086**

**using namespace std;**

**//124 8 5 369 7 \***

**typedef struct bitnode**

**{**

**char data;//结点数据元素**

**struct bitnode \*lchild,\*rchild;//左右孩子指针**

**}bitnode,\*bitree;**

**char ch;**

**bitree t;**

**int k;//测试组数**

**int ans;**

**int a[100086];**

**//创建树**

**void cbtree(bitree &t)**

**{**

**scanf("%c",&ch); // abdg e c f**

**if (ch==' ') t=NULL;**

**else**

**{**

**t=(bitnode \*) malloc(sizeof (bitnode));**

**/\*if(!t)**

**{**

**return false;**

**} \*/**

**t->data = ch ;**

**//cout<<t->data<<endl;**

**cbtree(t->lchild);**

**cbtree(t->rchild);**

**}**

**//return true;**

**}**

**//访问**

**void visit(bitree& t)**

**{**

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

**}**

**//先序**

**void preorder(bitree& t)**

**{**

**if(t!=NULL)**

**{**

**visit(t);//访问根节点**

**//printf("%c",t->data);**

**preorder(t->lchild);//访问左子树**

**preorder(t->rchild); //访问右子树**

**}**

**}**

**//中序**

**void inorder(bitree& t)**

**{**

**if(t!=NULL)**

**{**

**inorder(t->lchild);//访问左子树**

**visit(t);//访问根节点**

**inorder(t->rchild);//访问右子树**

**}**

**}**

**//后序**

**void postorder(bitree& t)**

**{**

**if(t!=NULL)**

**{**

**postorder(t->lchild);//访问左子树**

**postorder(t->rchild);//访问右子树**

**visit(t);//访问根节点**

**}**

**}**

**int main()**

**{**

**cbtree(t);**

**cout<<"先序遍历序列为："<<endl;**

**preorder(t);**

**cout<<endl;**

**cout<<"中序遍历序列为："<<endl;**

**inorder(t);**

**cout<<endl;**

**cout<<"后序遍历序列为："<<endl;**

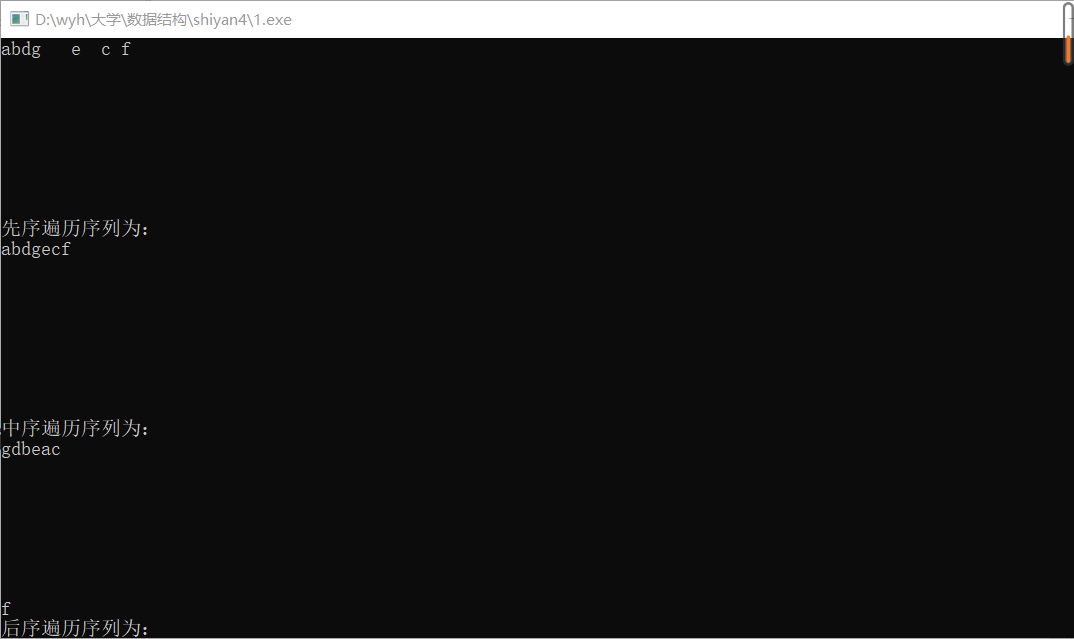
**postorder(t);**

**cout<<endl;**

**return 0;**

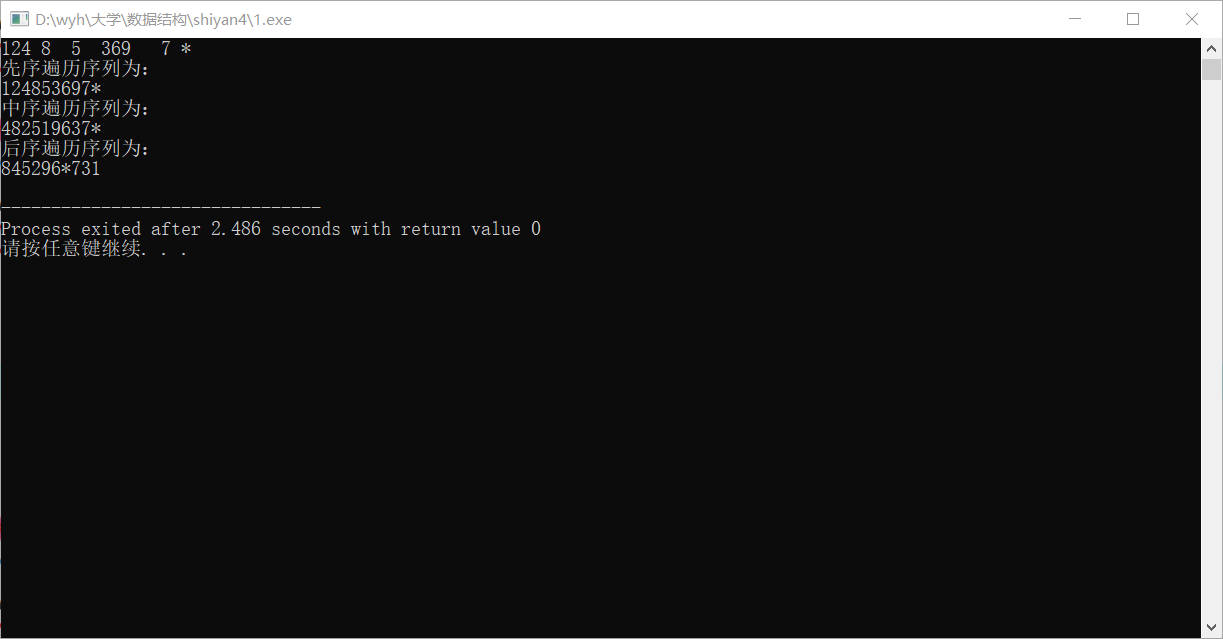
**}**

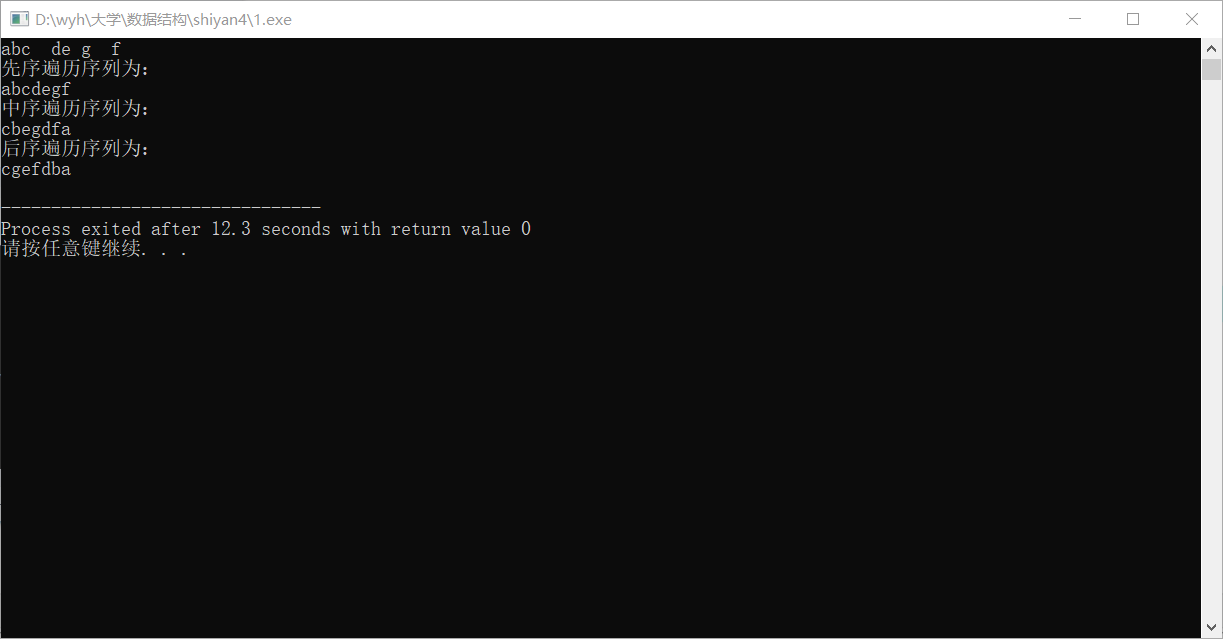
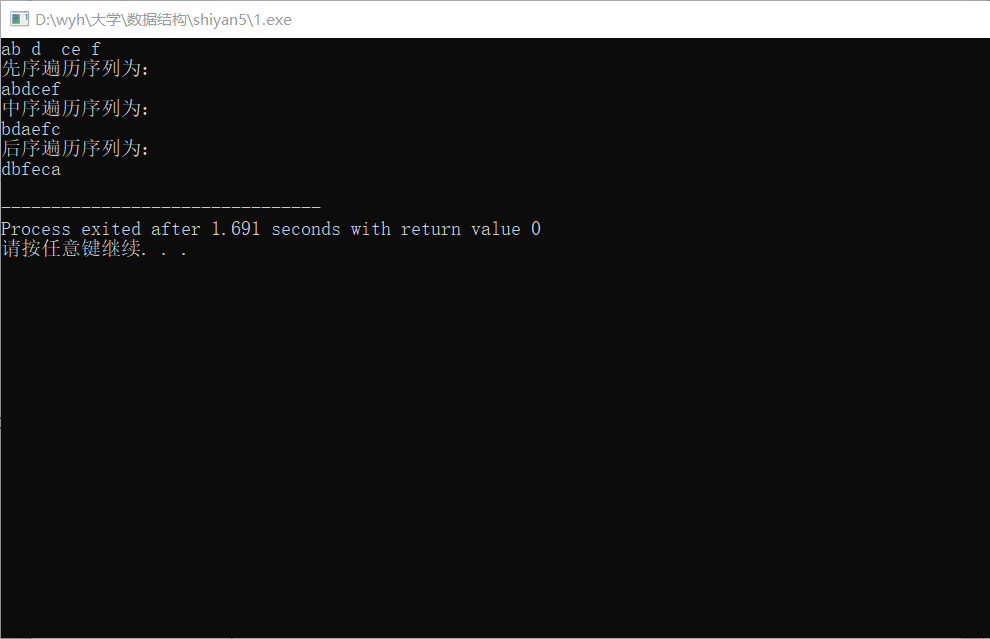
**1.6调试过程截图**

****

**输入测试数据有误，导致空格增多**

**1.7结果截图**

****

****

**2.打印二叉树结构**

**2.1问题分析**

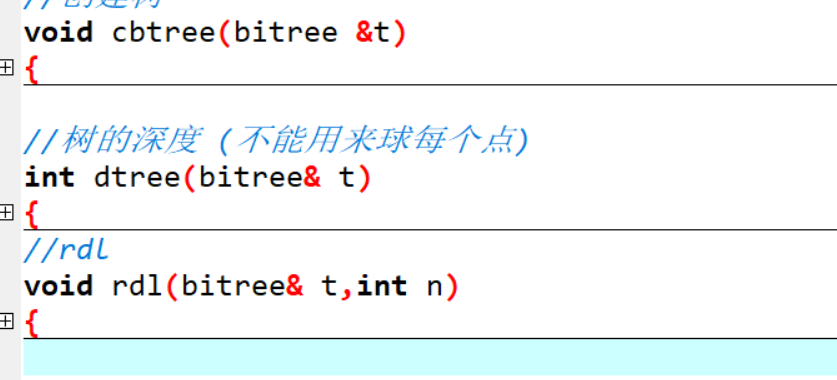
本设计要求按凹入表形式横向打印二叉树结构，即二叉树的根在屏幕的最左边，二叉树的左子数在屏幕的下边，二叉树的右子数在屏幕的上边。 二叉树结构示意图根据题意和图示可以看出，在根结点A右边的所有结点即A 的右子树（CEF）横向打印后都在它的上方打印出来，在根结点A 左边的结点即左子树(BD)都在它的下方打印出来。同理类推，结点C 左边的结点 即左子树(EF)都在C 的下方，而C 结点没有右孩子，所以C 结点打印出来的位置为第一行。把二叉树逆时 针旋转 90，按照二叉树的凸入表示法打印二叉树。显然，可把此算法设计成递归算法。由于把二叉树逆时针旋转90后，在屏幕上方的首先是右子树，然后是根节点，最后是左子树，打印二叉树算法可以用rdl。 幵且还可以看出处于第一局的A 结点的纵向位置是第一列，而第二局的B、C 结点的纵向位置是第二列，依此类推，所以可以利用结点的深度来控制各结点的横向位置，在屏幕上打印出树形结构。

**2.2设计方案**

**1）建立二叉树：每个节点最多有两个孩子，采用链式方式存储，每个结点的值域有数据域（data），两个指针域左海子（lchild）右孩子（rchild）。而本题中采用先序遍历，先接受根节点在接受左节点和右节点，同样是通过递归完成建立。使用创建树的函数**

**2）求树的深度函数**

**3）rdl遍历输出函数**

****

**2.3算法**

**//创建树**

**void cbtree(bitree &t)**

**{**

**scanf("%c",&ch); // abdg e c f**

**if (ch==' ') t=NULL;**

**else**

**{**

**t=(bitnode \*) malloc(sizeof (bitnode));**

**/\*if(!t)**

**{**

**return false;**

**} \*/**

**t->data = ch ;**

**//cout<<t->data<<endl;**

**cbtree(t->lchild);**

**cbtree(t->rchild);**

**}**

**//return true;**

**}**

**//树的深度 (不能用来球每个点)**

**int dtree(bitree& t)**

**{**

**if(t==NULL)**

**{**

**//cout<<"1"<<endl;**

**return 0;**

**}**

**else**

**{**

**int l=dtree(t->lchild);**

**//cout<<l<<" "<<endl;**

**int r=dtree(t->rchild);**

**int ans=((l>r)?l+1:r+1);**

**return ans;//树深度=max(左子树深度，右子树深度)+1**

**}**

**}**

**//rdl**

**void rdl(bitree& t,int n)**

**{**

**if(t!=NULL)**

**{**

**rdl(t->rchild,n+1); //访问右子树**

**//visit(t);**

**for(int i=0;i<n;i++)**

**{**

**//cout<<" ";**

**printf("\t");**

**}**

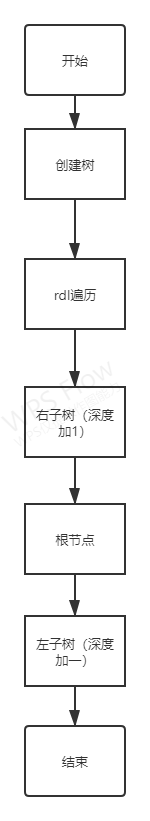
**printf("%2c\n",t->data);//访问根节点**

**rdl(t->lchild,n+1);//访问左子树**

**}**

**}**

**2.4设计图**

****

**2.5程序**

**#include<cstdio>**

**#include<cstdlib>**

**#include<cstring>**

**#include<iostream>**

**#include<cmath>**

**#define MAXSIZE 100086**

**using namespace std;**

**typedef struct bitnode**

**{**

**char data;//结点数据元素**

**int deept;**

**struct bitnode \*lchild,\*rchild;//左右孩子指针**

**}bitnode,\*bitree;**

**char ch;**

**int dn;//整棵树高度**

**bitree t;**

**int k;//测试组数**

**int ans;**

**int a[100086];**

**//abc de g f**

**//ab d ce f**

**//创建树**

**void cbtree(bitree &t)**

**{**

**scanf("%c",&ch); // abdg e c f**

**if (ch==' ') t=NULL;**

**else**

**{**

**t=(bitnode \*) malloc(sizeof (bitnode));**

**/\*if(!t)**

**{**

**return false;**

**} \*/**

**t->data = ch ;**

**//cout<<t->data<<endl;**

**cbtree(t->lchild);**

**cbtree(t->rchild);**

**}**

**//return true;**

**}**

**//树的深度**

**int dtree(bitree& t)**

**{**

**if(t==NULL)**

**{**

**//cout<<"1"<<endl;**

**return 0;**

**}**

**else**

**{**

**int l=dtree(t->lchild);**

**//cout<<l<<" "<<endl;**

**int r=dtree(t->rchild);**

**int ans=((l>r)?l+1:r+1);**

**return ans;//树深度=max(左子树深度，右子树深度)+1**

**}**

**}**

**//rdl**

**void rdl(bitree& t,int n)**

**{**

**if(t!=NULL)**

**{**

**rdl(t->rchild,n+1); //访问右子树**

**//visit(t);**

**for(int i=0;i<n;i++)**

**{**

**//cout<<" ";**

**printf("\t");**

**}**

**printf("%2c\n",t->data);//访问根节点**

**rdl(t->lchild,n+1);//访问左子树**

**}**

**}**

**int main()**

**{**

**cbtree(t);**

**//cout<<"深度："<<endl;**

**dn=dtree(t);**

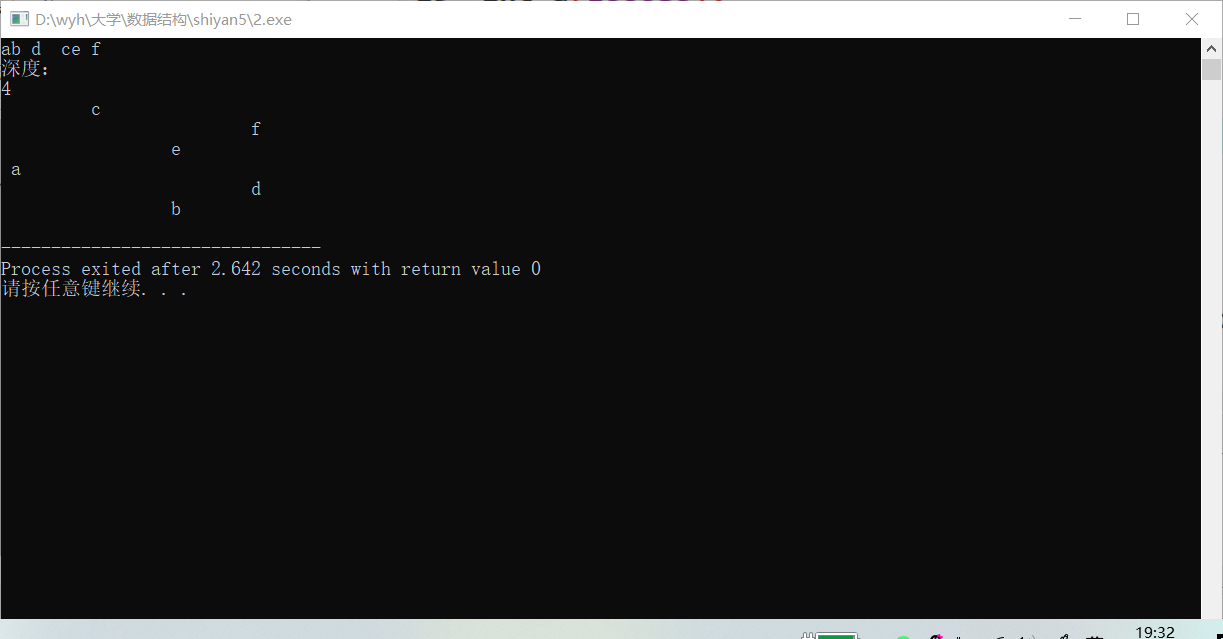
**//cout<<dn<<endl;**

**rdl(t,0);**

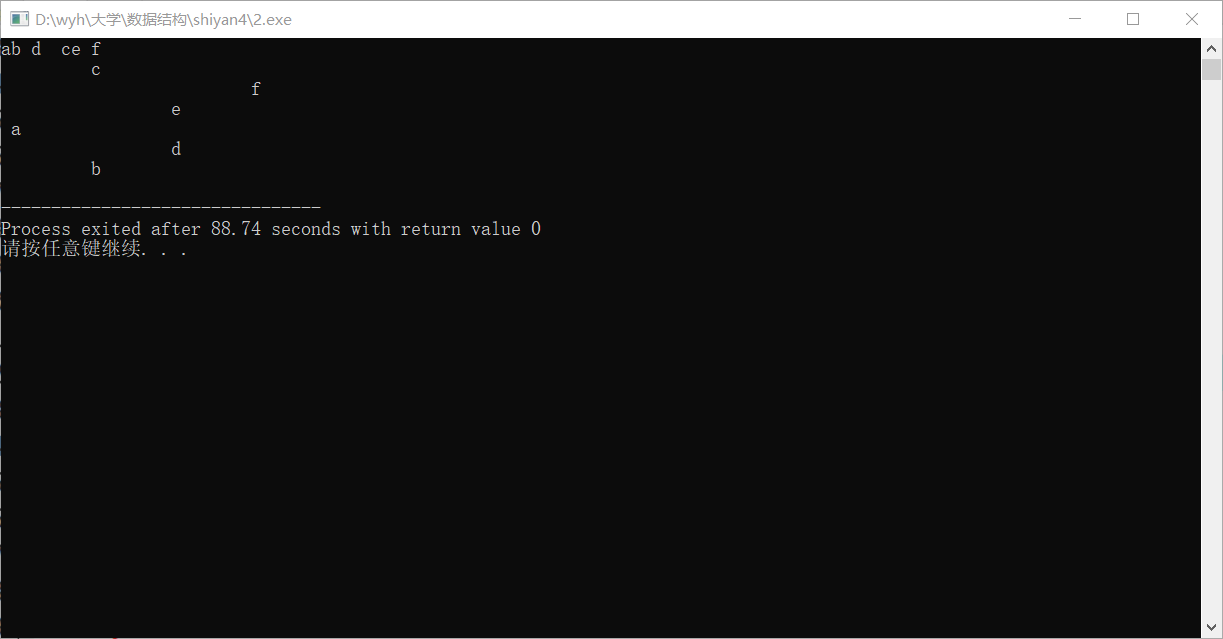
**return 0;**

**}**

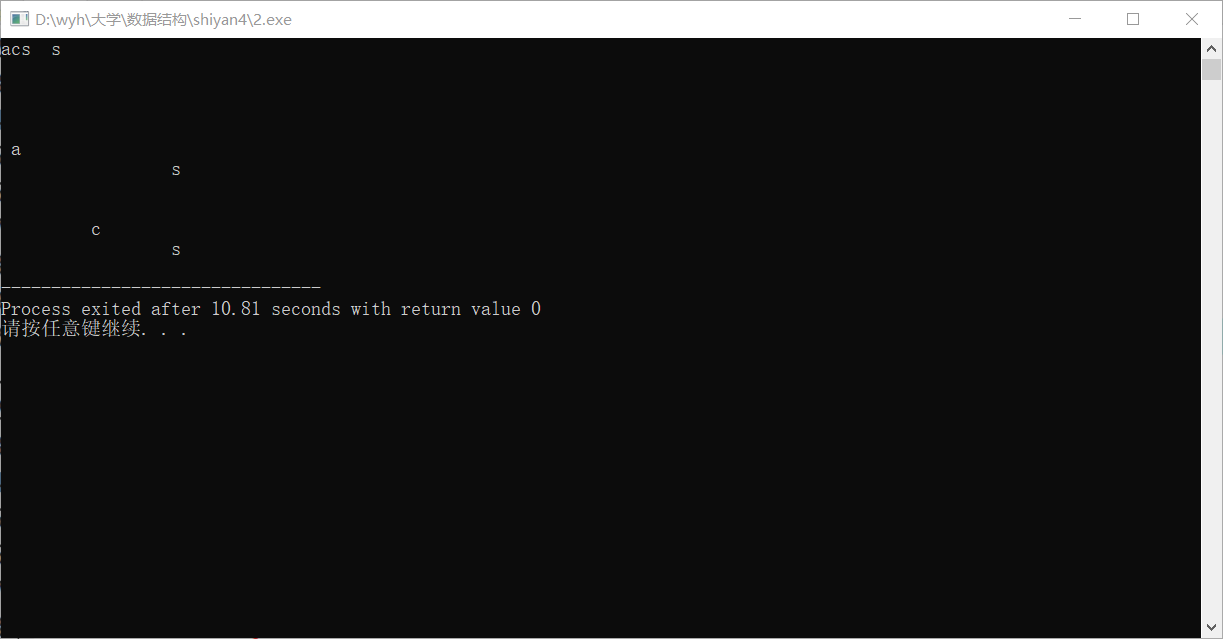
**2.6调试过程截图**

****

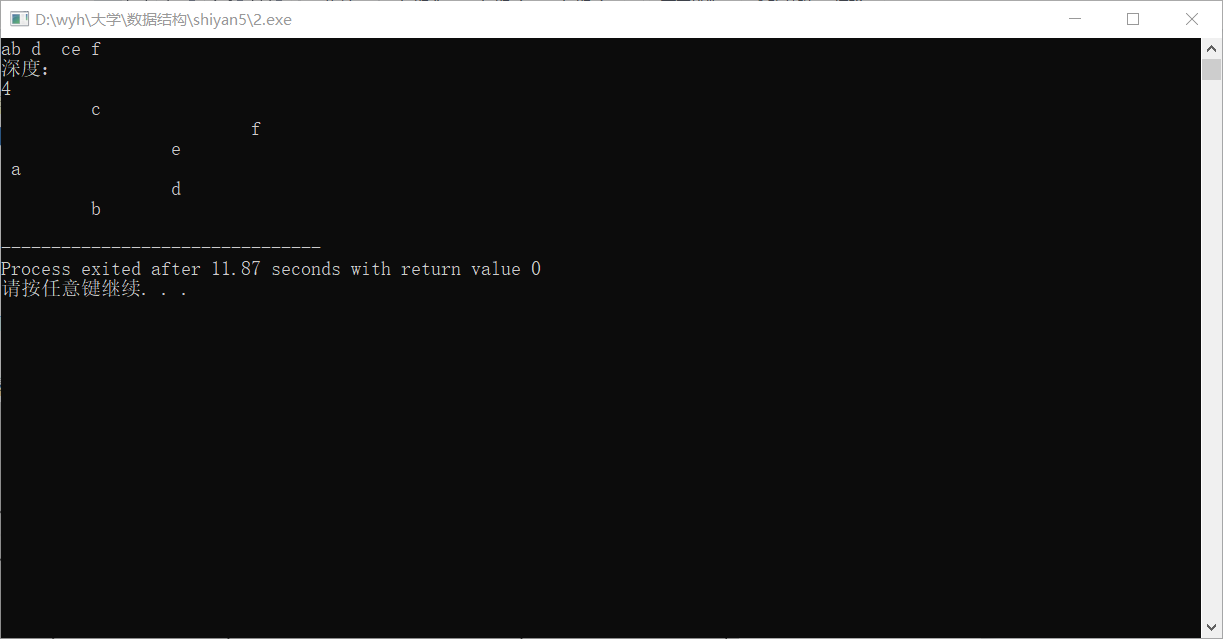
**//调试过程中发现子树的深度出现了错误，因为开始把子树当成一个新树求其深度，但在这种情况下深度的计算是不正确的，因为一棵树的左右子树可能不是一样高**

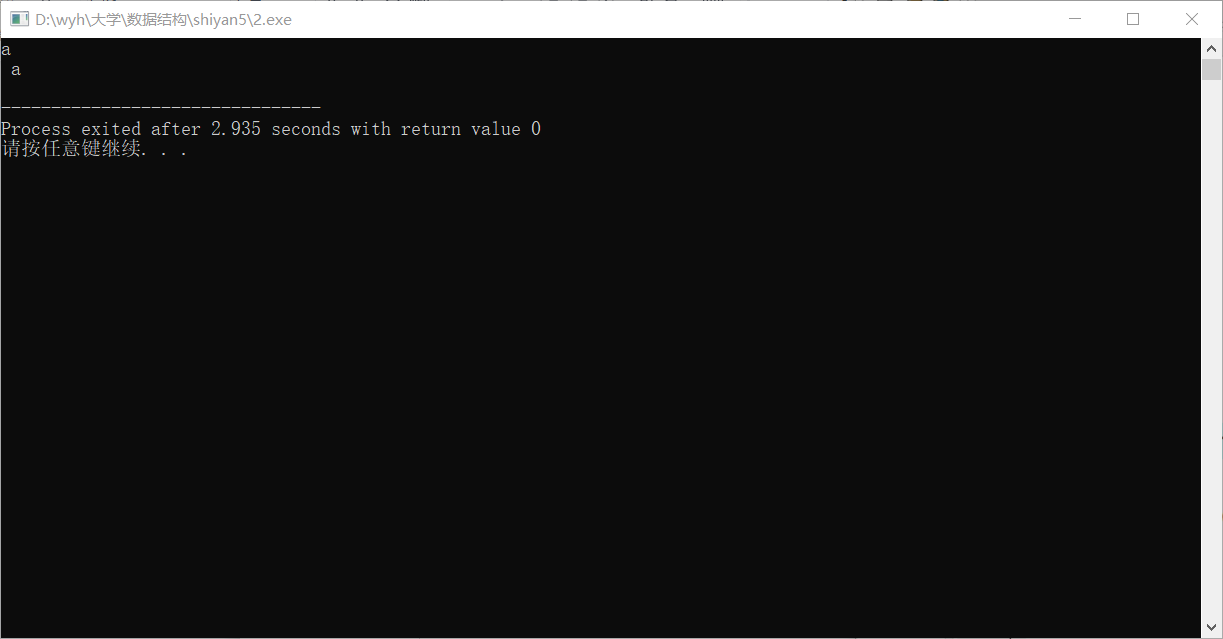
****

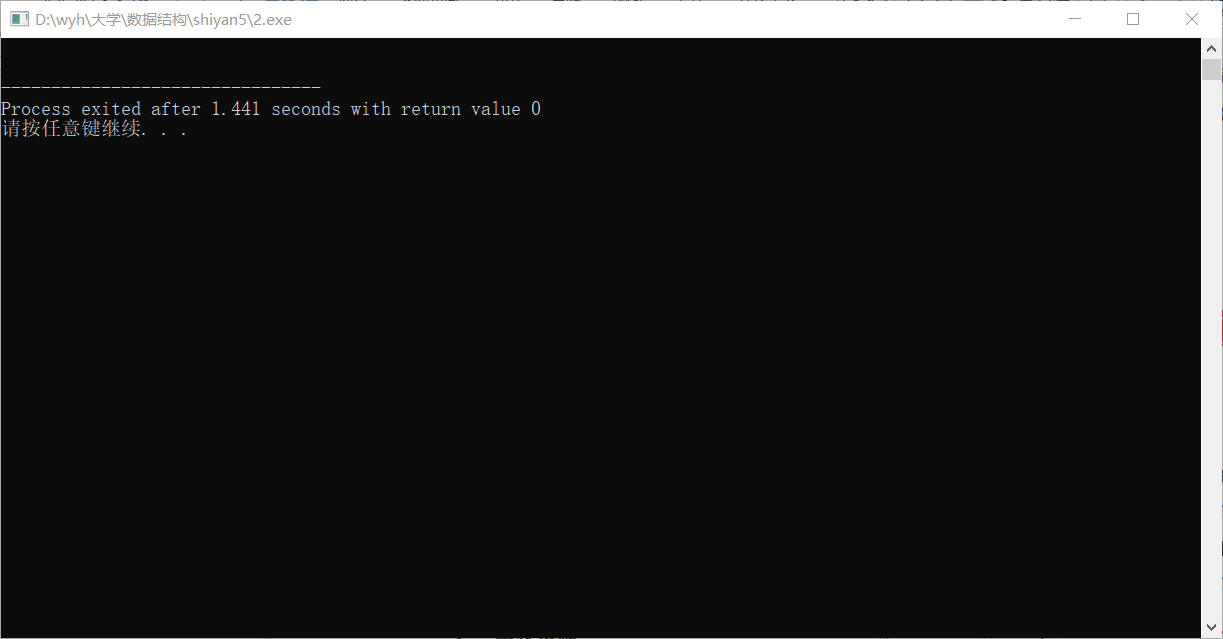
**//调试后**

****

**//随意输入节点需要使输入符合二叉树才能输出**

**2.7运行结果截图**

****

****

**空树**

**3选做**

思路与递归大致一致，只是用手写的算法模拟了系统调用栈，在这里使用了容器。

先序遍历的非递归算法则是：

指针p指向根节点；

while （p不为NULL或栈不空）

｛

反复访问当前节点\*p，

指针p进栈、再将指针左移，直至最左下节点；

当栈不空时，出栈，

取出栈顶指针且右移（返回上面操作，去遍历右子树）；

｝

中序遍历则与先序只是位置不同

后序则需要额外考虑如何从右孩子回到根节点：

保存最后一个访问的节点last，如果满足 (p->right==NULL && last ==p->left) || last=p->right，那么显然p的孩子都访问过了，接下来可以访问p。

**#include<cstdio>**

**#include<cstdlib>**

**#include<cstring>**

**#include<iostream>**

**#include<cmath>**

**#include<stack>**

**#define MAXSIZE 100086**

**using namespace std;**

**typedef struct bitnode**

**{**

**char data;//结点数据元素**

**struct bitnode \*lchild,\*rchild;//左右孩子指针**

**}bitnode,\*bitree;**

**char ch;**

**bitree t;**

**int k;//测试组数**

**int ans;**

**int a[100086];**

**//创建树**

**void cbtree(bitree &t)**

**{**

**scanf("%c",&ch); // abdg###e##c#f##**

**if (ch==' ') t=NULL;**

**else**

**{**

**t=(bitnode \*) malloc(sizeof (bitnode));**

**/\*if(!t)**

**{**

**return false;**

**} \*/**

**t->data = ch ;**

**//cout<<t->data<<endl;**

**cbtree(t->lchild);**

**cbtree(t->rchild);**

**}**

**//return true;**

**}**

**//访问**

**void visit(bitree& t)**

**{**

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

**}**

**//先序**

**void preorder(bitree& t)**

**{**

**if(t!=NULL)**

**{**

**bitnode\* p=t;**

**stack<bitnode\*> s;**

**while(!s.empty()||p)**

**{**

**while(p)**

**{**

**printf("%c",p->data);//访问根节点**

**s.push(p);**

**p=p->lchild;//访问左子树**

**}**

**p=s.top();**

**s.pop();**

**p=p->rchild;//访问右子树**

**}**

**}**

**}**

**//中序**

**void inorder(bitree& t)**

**{**

**if(t!=NULL)**

**{**

**bitnode\* p=t;**

**stack<bitnode\*> s;**

**while(!s.empty()||p)**

**{**

**while(p)**

**{**

**s.push(p);**

**p=p->lchild;//访问左子树**

**}**

**p=s.top();**

**s.pop();**

**printf("%c",p->data);//访问根节点**

**p=p->rchild;//访问右子树**

**}**

**}**

**}**

**//后序**

**void postorder(bitree& t)**

**{**

**if(t!=NULL)**

**{**

**bitnode\* p=t;**

**stack<bitnode\*> s;**

**bitnode\* last=t;//上次访问的**

**//s.push(p);**

**while(!s.empty()||p)**

**{**

**while(p)**

**{**

**s.push(p);**

**p=p->lchild;//访问左子树**

**}**

**if(!s.empty())**

**{**

**p=s.top();**

**//if(p->rchild==NULL||p->rchild==last)**

**if( (p->lchild == NULL && p->rchild == NULL) || (p->rchild == NULL && last == p->lchild) || (last == p->rchild) )**

**{ //叶节点或左右孩子都访问了**

**printf("%c",p->data);//访问根节点**

**s.pop();//栈顶元素出栈**

**last=p;//访问过p了**

**p=NULL;//将当前结点置空**

**}**

**else**

**{**

**p=p->rchild;//访问右子树**

**}**

**}**

**}**

**}**

**}**

**int main()**

**{**

**//ans=0;**

**//memset(a,0,sizeof(a));**

**cbtree(t);**

**cout<<"先序遍历序列为："<<endl;**

**preorder(t);**

**cout<<endl;**

**cout<<"中序遍历序列为："<<endl;**

**inorder(t);**

**cout<<endl;**

**cout<<"后序遍历序列为："<<endl;**

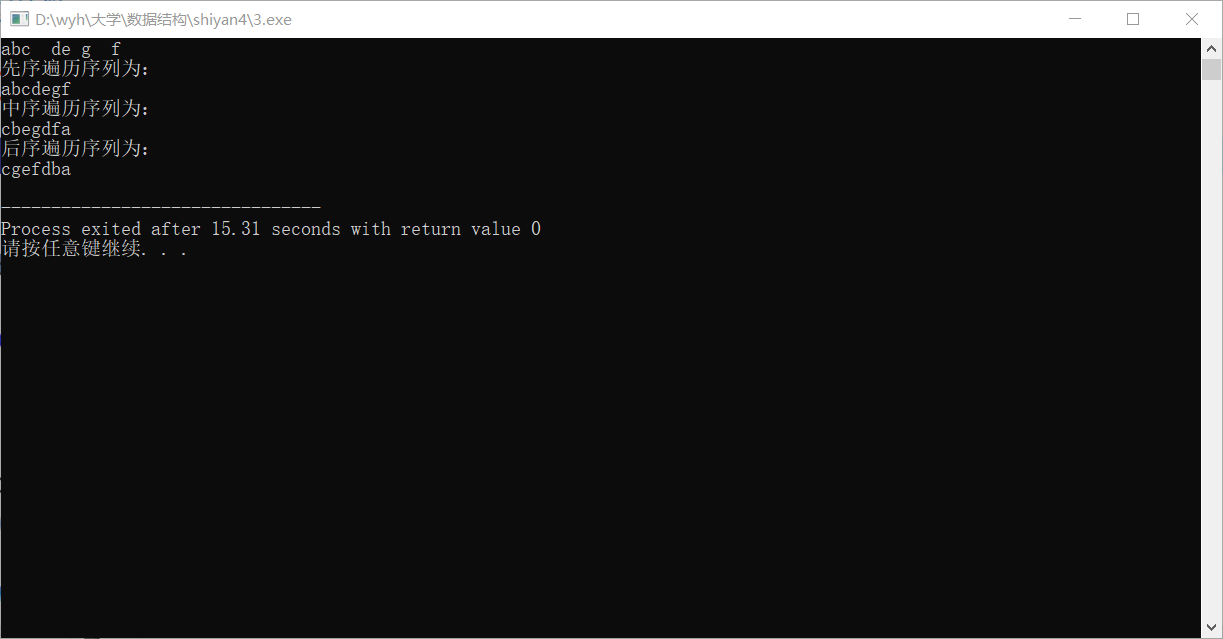
**postorder(t);**

**cout<<endl;**

**return 0;**

**}**

**结果截图：**

****

1. **实验心得**

通过这次实验，我知道了二叉树是树形结构的一个重要的类型，二叉树是n（n>=0）个结点的有限集，它或者是空集（n=0），或者由个根结点及两棵互不相交的、分别称作这个根的左子树和右子树的二叉树组成。二叉树的存储结构和算法比较简单，特别适合计算机处理。即使一般形式的树也可简单的转换为二叉树。二叉树的顺序存储结构是把二叉树的所有结点，按照一定的次序顺序，存储到一片连续的存储单元中。遍历二叉树就是条搜索路径周游二叉树，对树中每个结点访问一次且仅访问一次。在遍历方案中主要有先序遍历、中序遍历、后序遍历。建立二叉树的时候首先申请一个空间，最重要的是要弄清楚在输入二叉树时空字符的位置。在做第二题时犯了一个逻辑错误，不能够吧每颗子树当作新树求其深度。所以在做题时还是要考虑清楚每一种情况.

要灵活运用二叉树中的各种数据关系，根据自己的需要来设计数据结构。必须要熟练掌握树的各种操作才能有一个清晰的认识。做课程设计同时也是对课本知识的巩固和加强，平时看课本时，有些问题就不是很能理解，做完课程设计，那些问题就迎刃而解了，而且还可以记住很多东西。通过编程才能更好的理解掌握数据结构的精髓。从而避免基础语法错误，让代码变得更简洁高效。如此才能准确高效的解决问题。在今后的编程过程中要更注重代码整体设计的提纲记录,让自己的思路更清晰。