数据结构课程设计文档

简易计算器

小组人员：刘子欣、史文翰、舒星源

1. **问题描述**

**设二叉树结点值为大写字母，输入二叉树的前序遍历和中序遍历序列，生成此二叉树，输出该二叉树的后序遍历和按层次遍历序列。输入某结点值，在二叉树中查找该结点，若该结点存在，则输出从根到该结点的路径，否则给出不存在信息。**

1. **算法思路**

**首先是建立二叉树：前序遍历的第一个元素即为根节点，可以在中序遍历中找到该节点，该节点的左侧为左子树，右侧为右子树，很明显该过程是递归的。**

**后序遍历：比较简单，过程大体描述为先遍历左子树，再遍历右子树，最后访问根节点。**

**层序遍历：可以用队列的数据结构存储，队首元素出队的同时，按顺序入队该结点的左孩子和右孩子，直到队列为空。**

**查找结点及输出路径：与图算法中深度优先搜寻类似，对某一节点进行分析，如果它有左（右）孩子并且没有被访问过，那么就把它的左（右）孩子压入栈中，如果走到某一条路径的端点（叶节点），或者左右孩子均已经被访问，则出栈。匹配到那个元素的同时，栈中所剩元素即为路径集合。**

1. **核心算法描述**

**bool createBiTree\_DLR\_LDR(BiTree &T,DLR dlr[MAX],LDR ldr[MAX])**

**{**

**if(dlr首元素为空)**

**{**

**T为空**

**}**

**else**

**{**

**建立T结点**

**找到dlr中首元素在ldr中的位置**

**if(T为叶节点)**

**{**

**左右子树赋值为空**

**}**

**else if( T没有左孩子 )**

**{**

**左子树赋值为空**

**递归建立右子树**

**}**

**else if( T没有右孩子 )**

**{**

**右子树赋值为空**

**递归建立左子树**

**}**

**else**

**{**

**递归建立左子树**

**递归建立右子树**

**}**

**}**

**return 1;**

**}**

**bool LRDTraverse(BiTree &T)**

**{**

**if(该结点非空)**

**{**

**遍历左子树**

**遍历右子树**

**访问该元素**

**}**

**return 1;**

**}**

**bool levelTraverse(BiTree &T)**

**{**

**建立队列来存储待访问的结点**

**while(队列非空 || 首元素不为空)**

**{**

**队首元素出队列，并访问它**

**如果该元素有左孩子 把它的左孩子入队**

**如果该元素有右孩子 把它的右孩子入队**

**}**

**return 1;**

**}**

**bool searchNode(BiTree &T,char ch)**

**{**

**建立栈结构来存储路径**

**根结点入栈**

**得到栈顶元素**

**while(栈非空&&栈底元素不为空指针&&栈顶元素！=搜寻变量)**

**{**

**if(如果栈顶元素有左孩子且它的左孩子没有被访问过)**

**{**

**左孩子入栈**

**}**

**else if(栈顶元素有右孩子且右孩子没有被访问过)**

**{**

**右孩子入栈**

**}**

**else**

**{**

**弹出栈顶元素**

**把弹出的元素放在“已访问”的垃圾栈中**

**}**

**获得栈顶元素**

**}**

**if(栈为空)**

**{**

**打印非法**

**}**

**else 打印这个栈（路径） return 1;**

**}**

1. **源程序**

**#include<stdio.h>**

**#include<stdlib.h>**

**#include<string.h>**

**#include<iostream>**

**using namespace std;**

**const int MAX=100;**

**//data struct**

**typedef struct treeNode{**

**treeNode \*rc,\*lc;**

**char data;**

**}treeNode, \* BiTree;**

**typedef char LDR,DLR;**

**typedef struct QUEUE{**

**treeNode \*\*front,\*\*tail;**

**}QUEUE;**

**typedef struct STACK{**

**treeNode \*\*base,\*\*top;**

**}STACK;**

**//stack**

**void initStack(STACK &S)**

**{**

**S.base=NULL;**

**S.top=NULL;**

**}**

**bool createStack(STACK &S,int n)**

**{**

**if(S.base=(treeNode\*\*)malloc(n\*sizeof(treeNode\*)) )**

**{**

**S.top=S.base;**

**return 1;**

**}**

**else**

**{**

**cout<<"Create Stack failed."<<endl;**

**return 0;**

**}**

**}**

**void push(STACK &S,treeNode\* temp)**

**{**

**\*S.top=temp;**

**S.top++;**

**}**

**bool pop(STACK &S,treeNode\* &temp)**

**{**

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

**{**

**cout<<"Error:The Stack is empty."<<endl;**

**return 0;**

**}**

**S.top--;**

**temp=\*S.top;**

**return 1;**

**}**

**bool emptyStack(STACK &S)**

**{**

**if(S.top==S.base) return 1;**

**return 0;**

**}**

**bool getStackTop(STACK &S,treeNode\* &temp)**

**{**

**if(!emptyStack(S))**

**{**

**S.top--;**

**temp=\*S.top;**

**S.top++;**

**return 1;**

**}**

**else**

**{**

**cout<<"Error:The Stack is empty."<<endl;**

**return 0;**

**}**

**}**

**bool isVisited(STACK &RUB,treeNode\* temp)**

**{//Check if the node is visited to find the path**

**treeNode\*\* sec=RUB.base;**

**while(sec!=RUB.top)**

**{**

**if(\*sec==temp) return 1;**

**sec++;**

**}**

**return 0;**

**}**

**bool printStack(STACK &S)**

**{**

**if(emptyStack(S)||\*S.base==NULL)**

**{**

**cout<<"Error:Can not print an empty OR a null\_tree stack."<<endl;**

**return 0;**

**}**

**else**

**{**

**treeNode \*\*temp=S.base;**

**while(temp!=S.top)**

**{**

**cout<<(\*temp)->data;**

**temp++;**

**}**

**cout<<endl;**

**}**

**return 1;**

**}**

**//Queue**

**void initQueue(QUEUE &Q)**

**{**

**Q.front=NULL;**

**Q.tail=NULL;**

**}**

**bool createQueue(QUEUE &Q,int n)**

**{**

**if(Q.front=(treeNode\*\*)malloc(n\*sizeof(treeNode\*)) )**

**{**

**Q.tail=Q.front;**

**return 1;**

**}**

**else**

**{**

**cout<<"Error:Create Queue failed."<<endl;**

**return 0;**

**}**

**}**

**bool emptyQueue(QUEUE &Q)**

**{**

**if(Q.front==Q.tail) return 1;**

**return 0;**

**}**

**void inQueue(QUEUE &Q,treeNode\* temp)**

**{**

**\*Q.tail=temp;**

**Q.tail++;**

**}**

**bool deQueue(QUEUE &Q,treeNode\* &temp)**

**{**

**if(emptyQueue(Q))**

**{**

**cout<<"Error:Empty Queue."<<endl;**

**return 0;**

**}**

**else**

**{**

**temp=\*Q.front;**

**Q.front++;**

**return 1;**

**}**

**}**

**//BiTree**

**void initBiTree(BiTree &T)**

**{**

**T=NULL;**

**}**

**bool createBiTree(BiTree &T)**

**{**

**char ch;**

**scanf("%c",&ch);**

**if(ch=='#')**

**{**

**T=NULL;**

**}**

**else**

**{**

**T=(treeNode\*)malloc(sizeof(treeNode));**

**T->data=ch;**

**createBiTree(T->lc);**

**createBiTree(T->rc);**

**}**

**return 1;**

**}**

**bool blank(int x)**

**{**

**if(x<0) return 0;**

**else for(int i=1;i<=x;i++) printf(" ");**

**return 1;**

**}**

**bool printBiTree(BiTree &T,int depth)**

**{//the order is RDL**

**if(T)**

**{**

**printBiTree(T->rc,depth+1);**

**if( !( depth<0 ) )**

**{**

**blank(depth);//use blanks to represent the depth(x positon)**

**cout<<T->data<<endl;//use endl(enter) to represent the y position**

**}**

**else return 0;**

**printBiTree(T->lc,depth+1);**

**}**

**return 1;**

**}**

**int findRoot(char s[MAX],char root)**

**{**

**char\* ps=s;**

**for(int i=0;i<=strlen(s)-1;i++)**

**{**

**if(s[i]==root) return i;**

**}**

**return -1;**

**}**

**bool createBiTree\_DLR\_LDR(BiTree &T,DLR dlr[MAX],LDR ldr[MAX])**

**{**

**int num;**

**DLR lc\_dlr[MAX],rc\_dlr[MAX];**

**LDR lc\_ldr[MAX],rc\_ldr[MAX];**

**if(!dlr[0])**

**{**

**T=NULL;**

**}**

**else**

**{**

**T=(treeNode\*)malloc(sizeof(treeNode));**

**T->data=dlr[0];**

**num=findRoot(ldr,dlr[0]);**

**if(strlen(dlr)==1 && num==0)**

**{**

**T->lc=NULL;**

**T->rc=NULL;**

**}**

**else if( num==0 )**

**{**

**T->lc=NULL;**

**strcpy(rc\_ldr,&ldr[num+1]);**

**strcpy(rc\_dlr,&dlr[num+1]);**

**createBiTree\_DLR\_LDR(T->rc,rc\_dlr,rc\_ldr);**

**}**

**else if( num==strlen(dlr)-1 )**

**{**

**T->rc=NULL;**

**strncpy(lc\_ldr,ldr,num);**

**strncpy(lc\_dlr,&dlr[1],num);**

**createBiTree\_DLR\_LDR(T->lc,lc\_dlr,lc\_ldr);**

**}**

**else**

**{**

**strncpy(lc\_ldr,ldr,num);**

**strncpy(lc\_dlr,&dlr[1],num);**

**strcpy(rc\_ldr,&ldr[num+1]);**

**strcpy(rc\_dlr,&dlr[num+1]);**

**createBiTree\_DLR\_LDR(T->lc,lc\_dlr,lc\_ldr);**

**createBiTree\_DLR\_LDR(T->rc,rc\_dlr,rc\_ldr);**

**}**

**}**

**return 1;**

**}**

**bool LRDTraverse(BiTree &T)**

**{**

**if(T)**

**{**

**LRDTraverse(T->lc);**

**LRDTraverse(T->rc);**

**cout<<T->data;**

**}**

**return 1;**

**}**

**bool levelTraverse(BiTree &T)**

**{**

**QUEUE Q;**

**initQueue(Q);**

**createQueue(Q,100);**

**treeNode\* temp;**

**inQueue(Q,T);**

**while(!emptyQueue(Q) && \*Q.front!=NULL)**

**{**

**deQueue(Q,temp);//visit the node**

**cout<<temp->data;**

**if(temp->lc) inQueue(Q,temp->lc);**

**if(temp->rc) inQueue(Q,temp->rc);**

**}**

**return 1;**

**}**

**bool searchNode(BiTree &T,char ch)**

**{**

**STACK s,rub;//s to save the path,rub to save the unable nodes**

**treeNode \*temp\_s,\*temp\_rub;**

**initStack(s);**

**initStack(rub);**

**createStack(s,100);**

**createStack(rub,100);**

**push(s,T);**

**getStackTop(s,temp\_s);**

**while(!emptyStack(s) && \*s.base!=NULL && temp\_s->data!=ch )**

**{**

**if(temp\_s->lc && !isVisited(rub,temp\_s->lc))**

**{**

**push(s,temp\_s->lc);//save the left child**

**}**

**else if(temp\_s->rc && !isVisited(rub,temp\_s->rc))**

**{**

**push(s,temp\_s->rc);//save the right child**

**}**

**else**

**{**

**pop(s,temp\_rub);//nothing to save**

**push(rub,temp\_rub);//go back**

**}**

**getStackTop(s,temp\_s);**

**}**

**if(emptyStack(s))**

**{**

**cout<<"Searching failed.The node doesn't exist."<<endl;**

**return 0;**

**}**

**else printStack(s);**

**return 1;**

**}**

**int main(void)**

**{**

**char ch;**

**DLR dlr[MAX]="";**

**LDR ldr[MAX]="";**

**BiTree T;**

**initBiTree(T);**

**back:**

**cout<<"Input the DLR & LDR to create a BiTree:"<<endl;**

**cout<<"Input the DLR:"<<endl;**

**gets(dlr);**

**cout<<"Input the LDR:"<<endl;**

**gets(ldr);**

**if(strlen(dlr)==strlen(ldr))**

**createBiTree\_DLR\_LDR(T,dlr,ldr);**

**else**

**{**

**cout<<"Error: The length of DLR != LDR "<<endl;**

**goto back;**

**}**

**cout<<"Display the BiTree:"<<endl;**

**printBiTree(T,0);**

**cout<<"LRD:"<<endl;**

**LRDTraverse(T);**

**cout<<endl;**

**cout<<"Level Traverse:"<<endl;**

**levelTraverse(T);**

**cout<<endl;**

**cout<<"Input the node u want to search:"<<endl;**

**cin>>ch;**

**cout<<"The result is:"<<endl;**

**searchNode(T,ch);**

**return 0;**

**}**

1. **测试数据及说明**

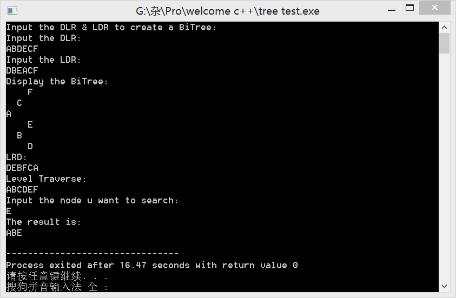
****1.****

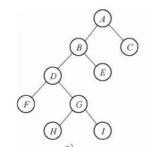
此二叉树的前序遍历为:ABDECF

此二叉树的中序遍历为:DBEACF

搜索结点E

测试结果如下：





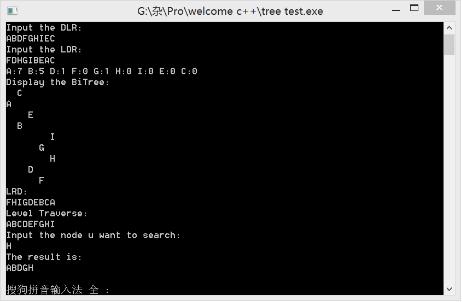
**2.**

此二叉树的前序遍历为：ABDFGHIEC

次二叉树的中序遍历为：FDHGIBEAC

搜索节点H

测试结果如下：



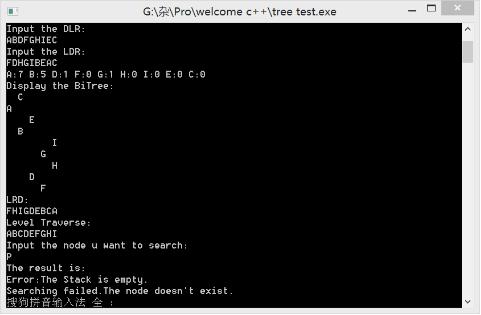
**3.树形和（2）相同，可搜索节点不存在**

此树的前序遍历为:ABDFGHIEC

此树的中序遍历为：FDHGIBEAC

搜索节点为：P（不存在）

测试结果为：



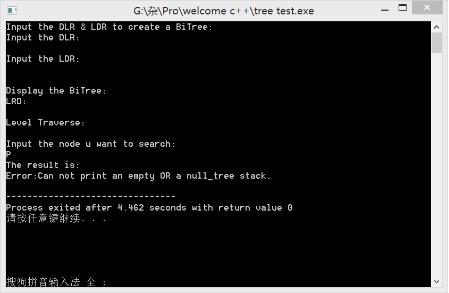
**4.空树**

此树的前序遍历为：（直接回车）

此树的中序遍历为：（直接回车）

搜索节点P（不存在）

测试结果如下：

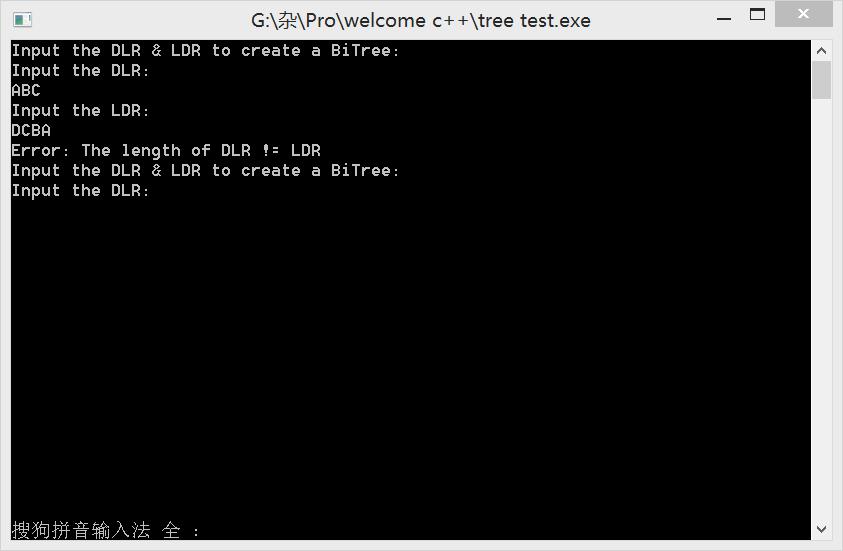


**5.非法输入**

**此树的前序遍历为：ABC**

**此树的后序遍历为：DCBA（不匹配）**

**测试结果如下：**



1. **结果分析和结论**

**本程序可以由前序遍历和中序遍历唯一确定一棵二叉树，因为他的输出是唯一的。对于层序遍历，由于每一个元素访问到之后，与他相关的元素（在二叉树中是结点的左右孩子）都要准备着被访问，因此用队列实现很方便，这与我们接下来要学的图的广度优先搜索有相似之处。对于查找路径，由于涉及到回溯和标记是否被访问过，这就与图算法的深度优先遍历很像，总之，我们可以看出树的操作是图操作的特例，图操作是树（或者更一般地讲，非线性结构）操作的推广。能否把二者有机地结合，是这次编码和以后学习的关键。**

1. **心得体会**

史文翰:这次我来编码，考验对树的问题的基本功的理解和认识。在递归建立二叉树的时候，我遇到不少问题。比如，对于strcpy函数的运用和理解，对strlen函数细节的认识和对字符串处理中‘\0’的认识都关系到你建立的树是否站得住脚。有很多次隐藏的BUG没有被发掘出来，这还要靠测试人员的细心和广泛地测试。再比如，对于空树的处理，同样需要一套完整的体系，无论是栈还是最后的树，都要考虑到树为空的问题。这也就引出了一些赶紧措施，比如引入树的头结点（在代码中未体现）。我在这次编码中，由于层序遍历需要用队列，查找路径需要用栈，我没有简单的申请数组来这样做（虽然这样做会节省时间和空间），我试着强迫自己建立一套完整的结构化体系，为以后的对象编程打好基础。对于栈和队列，我又定义了一套操作来适应本实验，可以让我的代码结构变得更加清晰可读，也算是复习了前面学过的栈和队列吧。

刘子欣：这次我做的测试，对于这个题目的树的测试无非考虑两种：第一种是正常的树，非空且路径存在，第二种是非空且路径不存在，第三种是空树且路径不存在。每一种都进行详细的测试是程序正确运行的保证，这三种测试在上面均有体现。此外，对一些容错处理也做了一些测试，例如，对于一些非法输入（DLR和LDR不匹配）也做了goto的回退调整。

舒星源：这次的文档比较冗长，因为史文翰同学做的编码比较严谨，但有些地方也过于繁杂，可以用一些更简单的方式来替代他设计的函数。文档和编码不同，不可过于简洁，而是要详述，希望这些文档可以给今后的工作带来一些基础性的贡献。

1. **分工与签名**

编码：史文翰

文档：舒星源

测试：刘子欣