



**数据结构实验报告**

**题 目 抽像数据类型——树**

**学 院 计算机学院**

**专业班级 网络工程1班**

**学 号 3213006449**

**姓 名 张丽蓉**

**指导教师 杨劲涛**

**2015年 7月 5日**

目录

一、题目 2

二、实验目的 2

三、编程环境 2

四、基本操作描述 2

数据对象D 2

数据关系R 2

基本操作P 2

五、存储结构的定义和声明 4

公用头文件以及宏定义 4

树的双亲表存储结构 4

树的辅助存储结构——单链队列的链式存储结构 4

六、算法设计 6

树的基本操作 6

七、测试与分析 14

测试与调试模块 14

算法设计的时间复杂度及优缺点分析 19

调试过程 20

测试结果分析 22

八、思考与小节 23

九、总结与体会 23

#### 一、题目

采用双亲表存储结构，实现树的抽象数据类型。

#### 二、实验目的

对某个具体的抽象数据类型，运用课程所学的知识和方法，设计合理的数据结构，并在此基础上实现该抽象数据类型的全部基本操作。通过本设计性实验，检验所学知识和能力，发现学习中存在的问题。 进而达到熟练地运用本课程中的基础知识及技术的目的。

#### 三、编程环境

硬件：PC机

软件：DEV C++

#### 四、基本操作描述

树的结构定义和树的一组基本操作：

ADT Tree{

数据对象D**：**D是具有相同特性的数据元素的集合。

数据关系R**：**

若D为空集，则称为空树；

若D仅含有一个数据元素，则R为空集，否则R={H}，H是如下二元关系：

(1) 在D中存在唯一的称为根的数据元素root，它在关系H下无前驱；

(2) 若D-{root}≠NULL，则存在D-{root}的一个划分D1,D2,D3, …,Dm(m>0)，对于任意j≠k(1≤j,k≤m)有Dj∩Dk=NULL,且对任意的i(1≤i≤m)，唯一存在数据元素xi∈Di有<root,xi>∈H;

(3) 对应于D-{root}的划分，H-{<root,xi>,…,<root,xm>}有唯一的一个划分H1，H2,…,Hm(m>0)，对任意j≠k(1≤j,k≤m)有Hj∩Hk=NULL，且对任意i(1≤i≤m),Hi是Di上的二元关系，(Di,{Hi})是一棵符合本定义的树，称为根root的子树。

基本操作P**：**

**InitTree(&T);**

操作结果：构造空树T。

**DestroyTree(&T);**

初始条件：树T存在。

操作结果：销毁树T。

**CreateTree(&T,definition);**

初始条件：definition给出树T的定义。

操作结果：按definition构造树T。

**ClearTree(&T);**

初始条件：树T存在。

操作结果：将树T清为空树。

**TreeEmpty(T);**

初始条件：树T存在。

操作结果：若T为空树，则返回TRUE，否则返回FALSE。

**TreeDepth(T);**

初始条件：树T存在。

操作结果：返回Ｔ的深度。

**Root(T);**

初始条件：树T存在。

操作结果：返回T的根。

**Value(T,cur\_e);**

初始条件：树T存在，cur\_e是T中某个结点。

操作结果：返回cur\_e的值。

**Assign(T,cur\_e,value);**

初始条件：树T存在，cur\_e是T中某个结点。

操作结果：结点cur\_e赋值为value。

**Parent(T,cur\_e);**

初始条件：树T存在，cur\_e是T中某个结点。

操作结果：若cur\_e是T的非根结点，则返回它的双亲，否则函数值为“空”。

**LeftChild(T,cur\_e);**

初始条件：树T存在，cur\_e是T中某个结点。

操作结果：若cur\_e是T的非叶子结点，则返回它的最左孩子，否则返回“空”。

**RightSibling(T,cur\_e);**

初始条件：树T存在，cur\_e是T中某个结点。

操作结果：若cur\_e有右兄弟，则返回它的右兄弟，否则返回“空”。

**InsertChild(&T,&p,I,c);**

初始条件：树Ｔ存在，ｐ指向Ｔ中某个结点，1≤i≤p指结点的度＋１，非空树ｃ与Ｔ不相交。

操作结果：插入c为Ｔ中ｐ指结点的第ｉ棵子树。

**DeleteChild(&T,&p,i);**

初始条件：树T存在，p指向T中某个结点，1≤i≤p指结点的度。

操作结果：删除Ｔ中ｐ所指结点的第ｉ棵子树。

**TraverseTree(T,visit());**

初始条件：树T存在，visit是对结点操作的应用函数。

操作结果：按某种次序对T的每个结点调用函数visit()一次且至多一次。一旦visit()失败，则操作失败。

}ADT Tree

在双亲表存储结构中添加以下基本抽象数据类型：

**Status Print(PTree T)；**

附加函数：用于显示树的所有内容。

初始条件：树T存在；

操作结果：将树T的所有结点显示出来。

在双亲表存储结构中，TraverseTree(T,visit())函数是按层次次序对T的每个结点进行访问的。

#### 五、存储结构的定义和声明

###### 公用头文件以及宏定义

#include<stdio.h>

#include<string.h>

#include<stdlib.h>

#include<stdio.h>

#include<string.h>

#include<stdlib.h>

#include<conio.h>

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define OVERFLOW 0

#define NIL '#'

#define MAX\_TREE\_SIZE 100

###### 树的双亲表存储结构

typedef char TElemType;//树结点存储的信息为字符型

typedef int Status;

typedef struct PTNode{ //结点结构

TElemType data;//元素值

int parent; //双亲位置,根结点的parent为-1

}PTNode;

typedef struct{ //树结构

PTNode nodes[MAX\_TREE\_SIZE];//由初始换操作分配的结点数组

int root,nodeNum; //根的位置和结点数

}PTree;

###### 树的辅助存储结构——单链队列的链式存储结构

typedef struct

{

int num;//结点的位置

TElemType data;//结点信息

}QElemType; //定义队列元素类型

//单链队列－－队列的链式存储结构

typedef struct LQNode

{

QElemType data;

struct LQNode \*next;

}LQNode,\*QueuePtr;//结点和结点指针类型

typedef struct

{

QueuePtr front,rear; //队头、队尾指针

}LinkQueue;//链队列类型

* **需要用到的辅助存储结构的基本操作——单链队列的链式存储结构的基本操作**

// 构造一个空队列Q

Status InitQueue(LinkQueue &Q)

{

Q.front=Q.rear=(LQNode\*)malloc(sizeof(LQNode));

if(!Q.front)

exit(OVERFLOW);

Q.front->next=NULL;

return OK;

}

//销毁队列Q

Status DestroyQueue(LinkQueue &Q)

{

while(Q.front)

{

Q.rear=Q.front->next;

free(Q.front);

Q.front=Q.rear;

}

return OK;

}

//将Q清为空队列

Status ClearQueue(LinkQueue &Q)

{

QueuePtr p,q;

Q.rear=Q.front;

p=Q.front->next;

Q.front->next=NULL;

while(p)

{

q=p;

p=p->next;

free(q);

}

return OK;

}

//若Q为空队列,则返回TRUE,否则返回FALSE

Status QueueEmpty(LinkQueue Q)

{

if(Q.front==Q.rear)

return TRUE;

else

return FALSE;

}

//插入元素e为Q的新的队尾元素

Status EnQueue(LinkQueue &Q,QElemType e)

{

LQNode \*p;

p=(LQNode\*)malloc(sizeof(LQNode));

if(!p) //存储分配失败

return OVERFLOW;

p->data=e;

p->next=NULL;

Q.rear->next=p;

Q.rear=p;

return OK;

}

//若队列不空,删除Q的队头元素,用e返回其值,并返回OK,否则返回ERROR

Status DeQueue(LinkQueue &Q,QElemType &e)

{

QueuePtr p;

if(Q.front==Q.rear)

return ERROR;

p=Q.front->next;

e=p->data;

Q.front->next=p->next;

if(Q.rear==p)

Q.rear=Q.front;

free(p);

return OK;

}

#### 六、算法设计

###### 树的基本操作

构造空树T

Status InitTree(PTree &T)

{

T.nodeNum=0;//树的结点数置为0

return OK;

}

销毁树T

void DestroyTree(PTree &T)

{

T.nodeNum=0;//树的双亲表存储结构是用定长的数据类型，无法销毁

}

构造树T

Status CreateTree(PTree &T)

{

LinkQueue Q;

QElemType q,t;

char temp[MAX\_TREE\_SIZE];

int i=1,j,len;

InitQueue(Q);

printf("\t请输入树的根结点(如果输入是'#'字符的话，则根为空):");

T.nodes[0].data=getche();

printf("\n");

if(T.nodes[0].data!=NIL)//根结点不为'#'符，即不为空

{

T.nodes[0].parent=-1;

q.data=T.nodes[0].data;

q.num=0;

EnQueue(Q,q);//根入队

while(i<MAX\_TREE\_SIZE&&!QueueEmpty(Q))

{

DeQueue(Q,q);

//输入所有q.data的孩子结点，临时保存到temp[MAX\_TREE\_SIZE]中

//这里还要考虑怎么结束输入创建树

printf("\t请输入%c的所有孩子结点(如果没有孩子，按回车键输入下一个结点的孩子):",q.data);

gets(temp);

len=strlen(temp);

for(j=0;j<len;j++,i++)

{

T.nodes[i].parent=q.num;

T.nodes[i].data=temp[j];

t.data=temp[j];

t.num=i;

EnQueue(Q,t);

}

}

if(i>=MAX\_TREE\_SIZE)//超出范围

exit(OVERFLOW);

T.nodeNum=i;

}

else

T.nodeNum=0;

return OK;

}

将树清为空树

Status ClearTree(PTree &T)

{

T.nodeNum=0;

return OK;

}

//初始条件：树T存在；操作结果：若T为空树，则返回TRUE，否则返回FALSE

Status TreeEmpty(PTree T)

{

if(T.nodeNum==0)

return TRUE;

else

return FALSE;

}

求树T的深度

Status TreeDepth(PTree T)

{

int depthmax=0,depth,i,parent;

//depthmax存放树的最大深度，depth存在遍历到该结点的深度，parent表示双亲的位置

for(i=0;i<T.nodeNum;i++)

{

depth=1;

parent=T.nodes[i].parent;

while(parent!=-1)//从T.nodes[i]结点退回根结点,退一步，深度加1

{

parent=T.nodes[parent].parent;

depth++;

}

if(depthmax<depth) depthmax=depth;

}

return depthmax;

}

求树T的根结点

TElemType Root(PTree T)

{

if(T.nodeNum)

return T.nodes[0].data;

else {

printf("\n\t树为空，没有根结点!");

return NIL;

}

}

求树T中的某个结点的值

TElemType Value(PTree T,int position)

{

if(position<=T.nodeNum)

return T.nodes[position-1].data;

else

{

printf("\n\t给出的第%d结点超出树原有的结点数，错误！\n",position);

return NIL;

}

}

将树T中某个结点重新赋值

Status Assign(PTree &T,int position,TElemType value)

{

if(position<=T.nodeNum)

{

T.nodes[position-1].data=value;

return OK;

}

else

return ERROR;

}

求树T某个结点的双亲

TElemType Parent(PTree T,int position)

{

if(position==1)

{

printf("\n\t给出的第%d结点为树的根结点，根结点没有双亲！\n",position);

return NIL;

}

else if(position<=T.nodeNum)

return T.nodes[T.nodes[position-1].parent].data;

else

{

printf("\n\t给出的第%d结点超出树原有的结点数，错误！\n",position);

return NIL;

}

}

求树T某个结点的最左孩子

TElemType LeftChild(PTree T,TElemType value)

{

int i,j;

for(i=0;i<T.nodeNum;i++)

if(T.nodes[i].data==value)

break;//找到值为value的结点T.nodes[i]

if(i>=T.nodeNum) {

printf("\n\t树中不存在值为%c的结点！错误！",value);

return NIL;

}

for(j=i+1;j<T.nodeNum;j++)

if(T.nodes[j].parent==i)

return T.nodes[j].data;

if(j>=T.nodeNum) {

printf("\n\t没有找到%c的最左孩子结点，说明值为%c的结点没有孩子！",value,value);

return NIL;

}

}

插入树T中某结点的第i棵子树

Status InsertChild(PTree &T,TElemType p,int i,PTree Tree1)

{

PTNode temp;//临时变量，用于交换两个结点的附加变量

int j,k,inposition,childnum;

int flag;

if(i<=0) {printf("\n\t插入非空树c的位置，第%d结点位置有错误！",i);return ERROR;}//i的数据有错

if(Tree1.nodeNum==0){

printf("\n\t插入的树Tree1为空，树T未变！");

return ERROR;

}

if(T.nodeNum!=0)

{

for(j=0;j<T.nodeNum;j++)

if(T.nodes[j].data==p)break;//找出值为p的结点T.nodes[j]

if(j>=T.nodeNum) {

printf("\n\t树中没有值为%c的结点，错误！",p);

return ERROR;

}

//找出插入点inposition

if(i==1) //i为1时，插入点为p的第一棵子树

{

for(k=j+1;k<T.nodeNum;k++)//找出p的第一棵子树T.nodes[k]

if(T.nodes[k].parent==j)break;

if(k>=T.nodeNum)

inposition=j+1;//如果p没有子树，则插入点为j+1

else

inposition=k;//找到p的第一棵子树，则插入点为k

}

else //插入的不是p的第一棵子树，求插入点

{

for(k=j+1,childnum=0;k<T.nodeNum;k++)//往后查找到p的第i棵子树的位置，查找前孩子数为0

if(T.nodes[k].parent==j){

childnum++;//孩子数加1

if(childnum==i-1)//孩子数为i-1，则得到inposition为k+1

break;

}

inposition=k+1;

}

if(inposition>=T.nodeNum){

printf("\n\t插入非空树Tree1的位置，第%d结点位置有错误！",i);

return ERROR;

}

//将非空树Tree1插入T中，首先要后移出Tree1.nodeNum个结点数，当插入点在T的最后结点后，则不用后移

if(inposition<T.nodeNum)

for(k=T.nodeNum-1;k>=inposition;k--)

{

T.nodes[k+Tree1.nodeNum]=T.nodes[k];//结点的信息后移

if(T.nodes[k].parent>=inposition)//结点的双亲大于插入点，双亲域要加Tree1.nodeNum

T.nodes[k+Tree1.nodeNum].parent+=Tree1.nodeNum;

}

//将非空树Tree1插入到插入点处

for(k=0;k<Tree1.nodeNum;k++){

T.nodes[k+inposition].data=Tree1.nodes[k].data;

T.nodes[k+inposition].parent=Tree1.nodes[k].parent+inposition;

}

T.nodes[inposition].parent=j;/\*插入到插入点处的结点的双亲应该为j\*/

T.nodeNum+=Tree1.nodeNum;

//将插入子树后的结点按层序排序，采用冒泡排序的方式

flag=1;

while(flag){

flag=0;

for(k=inposition;k<T.nodeNum-1;k++)

if(T.nodes[k].parent>T.nodes[k+1].parent){

temp=T.nodes[k];

T.nodes[k]=T.nodes[k+1];

T.nodes[k+1]=temp;

flag=1;

}

}

return OK;

}

else {

printf("\n\t树为空，错误！");

return ERROR;

}

}

删除T中某结点所指向的第i棵子树

Status DeleteChild(PTree &T,TElemType p,int i)

{

int j,k,childnum=0,deposition;//childnum为记录孩子数，deposition为第一删除结点

int deleteflag[MAX\_TREE\_SIZE];//删除结点的标志，为1即是要删除的，为0即是不用删除的

LinkQueue Q;

QElemType q1,q2;

if(i<=0) {//i的数据有错

printf("\n\t删除非空树Tree1的位置，第%d结点位置有错误(i>=1)！",i);

return ERROR;

}

if(i>=T.nodeNum){

printf("\n\t删除非空树Tree1的位置，要删除的%d结点超出树T的范围！",i);

return ERROR;

}

for(j=0;j<MAX\_TREE\_SIZE;j++)//删除标志置为0

deleteflag[j]=0;

if(T.nodeNum!=0){

for(j=0;j<T.nodeNum;j++)//找出值为p的结点T.nodes[j]

if(T.nodes[j].data==p)break;

if(j>=T.nodeNum){

printf("\n\t树中没有值为%c的结点，错误！",p);

return ERROR;

}

//找出删除点deposition

for(k=j+1;k<T.nodeNum;k++){

if(T.nodes[k].parent==j)

childnum++;//孩子数加1

if(childnum==i)

break;

}

deposition=k;

if(deposition>=T.nodeNum){//删除点超出范围

printf("\n\t树中值为%c的结点不存在第%d棵子树，没法删除，错误！",p,i);

return ERROR;

}

//删除点不超出范围

InitQueue(Q);//初始化队列

q1.data=T.nodes[deposition].data;

q1.num=deposition;

EnQueue(Q,q1);//将第一个要删除的结点入队列

deleteflag[deposition]=1;//将删除标志置为1

k=deposition;

while(!QueueEmpty(Q)){

DeQueue(Q,q2);//出队列

for(k=q2.num+1;k<T.nodeNum;k++)//如果结点的双亲域和出队列的q2.num相等

if(q2.num==T.nodes[k].parent){

q1.num=k;

q1.data=T.nodes[k].data;

EnQueue(Q,q1);//结点入队列

deleteflag[k]=1;//删除标志置为1

}

}

for(k=deposition;k<T.nodeNum;k++)//根据结点的删除标志来删除结点，删除即是前移结点位置

if(deleteflag[k]==1){

for(j=k;j<T.nodeNum;j++){//前移结点

deleteflag[j]=deleteflag[j+1];//删除标志前移

T.nodes[j]=T.nodes[j+1];//结点前移

if(T.nodes[j-1].parent>k)//结点的双亲大于要删除的结点的位置，减1

T.nodes[j-1].parent--;

}

k--;//k减1，倒回去继续检查删除标志

T.nodeNum--;//结点数减1

}

return OK;

}

else

{

printf("\t树为空，不能删除子树！错误");

return ERROR;

}

}

按层次次序对树T进行遍历

Status TraverseTree(PTree T,void(\*visit)(TElemType))

{

int i;

printf("\n\t树按照层次遍历的顺序为：");

for(i=0;i<T.nodeNum;i++)

visit(T.nodes[i].data);

return OK;

}

//访问对结点操作的应用函数

void visit(TElemType value)

{

printf("%c",value);

}

#### 七、测试与分析

###### 测试与调试模块

//主菜单函数

char menu()

{

system("cls");//清屏

printf("\t+================================================+\n");

printf("\t|\*\*\*\*\*\*\*\*\*\* 网络工程1班张丽蓉 \*\*\*\*\*\*\*\*\*\*|\n");

printf("\t+================================================+\n");

printf("\t| 菜单栏 |\n");

printf("\t+~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~+\n");

printf("\t| A-----创建一棵空树 |\n");

printf("\t| B-----销毁树 |\n");

printf("\t| C-----将清为空树 |\n");

printf("\t| D-----判断树是否为空 |\n");

printf("\t| E-----返回树的深度 |\n");

printf("\t| F-----返回树的根节点 |\n");

printf("\t| G-----返回树中某个结点的值 |\n");

printf("\t| H-----将树中的某个结点赋值 |\n");

printf("\t| I-----求树中的某个结点的双亲 |\n");

printf("\t| J-----求树中的某个结点的最左孩子 |\n");

printf("\t| K-----求树中的某个结点的右兄弟 |\n");

printf("\t| L-----在某个结点下插入一棵子树 |\n");

printf("\t| M-----删除某个结点的子树 |\n");

printf("\t| N-----显示树 |\n");

printf("\t| O-----层次遍历树 |\n");

printf("\t| P-----退出 |\n");

printf("\t+================================================+\n");

printf("\t请选择菜单号(A---P):");

return(getche());

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//主函数

int main()

{

int i;

PTree T,Tree1;

TElemType e,e1;

InitTree(T);//初始化树

CreateTree(T);//创建一棵树

Print(T);//显示 创建的树的所有内容

printf("\n\t树创建成功！");

getchar();

while(1)

{

system("cls");//清屏

switch(menu())

{

case 'a':

case 'A'://创建一棵树

ClearTree(T);

printf("\n\t重新创建一棵新树：\n");

CreateTree(T);

getch();

system("cls");

Print(T);

printf("\n\t树创建成功！");

getche();break;

case 'b':

case 'B'://销毁树

DestroyTree(T);

printf("\n\t树已被销毁!\n");

getche();

break;

case 'c':

case 'C'://将树清空

ClearTree(T);

printf("\n\t树已被清为空树!\n");

getche();

break;

case 'd':

case 'D'://判断树是否为空

if(TreeEmpty(T))

printf("\n\t树为空！");

else

printf("\n\t树不为空！");

getche();

break;

case 'e':

case 'E'://求树的深度

printf("\n\t树的深度为：%d",TreeDepth(T));

getche();

break;

case 'f':

case 'F'://求树的根结点

e=Root(T);

if(e!=NIL)

printf("\n\t树的根结点为：%c",e);

getche();

break;

case 'g':

case 'G'://求树中某个结点的值

printf("\n\t输入你要得到第几个结点的值：");

scanf("%d",&i);

e=Value(T,i);

if(e!=NIL) printf("\n\t第%d个结点的值为：%c",i,e);

getche();

break;

case 'h':

case 'H'://将树中某个结点赋值

printf("\n\t输入你要赋值的结点的是第几个结点：");

scanf("%d",&i);

printf("\n\t输入你要赋值的值：");

e=getche();

if(Assign(T,i,e))

printf("\n\t赋值成功！");

else

printf("\n\t赋值失败！");

getche();

break;

case 'i':

case 'I'://求树中的某个结点的双亲

printf("\n\t输入你要得到双亲的结点是第几个结点：");

scanf("%d",&i);

e=Parent(T,i);

if(e!=NIL)

printf("\n\t第%d个结点的双亲为：%c",i,e);

getche();

break;

case 'j':

case 'J'://求树中的某个结点的最左孩子

printf("\n\t输入你要求其最左孩子的结点的值：");

e=getche();

e1=LeftChild(T,e);

if(e1!=NIL)printf("\n\t值为%c的结点的最左孩子为：%c",e,e1);

getche();

break;

case 'k':

case 'K'://求树中的某个结点的右兄弟

printf("\n\t输入你要求其右兄弟的结点的值：");

e=getche();

e1=RightSibling(T,e);

if(e1!=NIL)printf("\n\t值为%c的右兄弟为：%c",e,e1);

getche();break;

case 'l':

case 'L'://在某个结点下边插入一棵子树

printf("\n\t插入一棵子树，创建子树：\n");

ClearTree(Tree1);

InitTree(Tree1);

CreateTree(Tree1);

Print(Tree1);

printf("\n\t子树创建成功！");

printf("\n输入要插入到哪个结点下：(输入结点的值)");

e=getche();

printf("\n\t输入要插入到结点%c的那棵子树下：",e);

scanf("%d",&i);

if(InsertChild(T,e,i,Tree1))

{

printf("\n\t插入子树成功！其插入后的树如下：");

Print(T);

}

getche();

break;

case 'm':

case 'M'://删除某个结点的子树

printf("\n\t删除一棵子树，原来的树如下：");

Print(T);

printf("\n\t输入要删除哪个结点的子树：(输入结点的值)");

e=getche();

printf("\n\t输入要删除结点%c的第几棵子树：",e);

scanf("%d",&i);

if(DeleteChild(T,e,i))

{

printf("\n\t删除成功！删除后的树如下：");

Print(T);

}

getche();

break;

case 'n':

case 'N'://显示树

printf("\n\t该树的显示情况如下:\n");

Print(T);

getche();

break;

case 'o':

case 'O'://层次遍历树

TraverseTree(T,visit);

getche();

break;

case 'p':

case 'P'://退出

printf("\n\t请按任意键退出!\n");

getche();

exit(1);

default :

printf("\n\t输入选择错误！请重新输入！");

getche();

break;

}

}

}

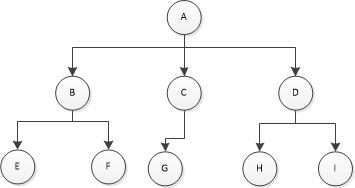
/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

###### 算法设计的时间复杂度及优缺点分析

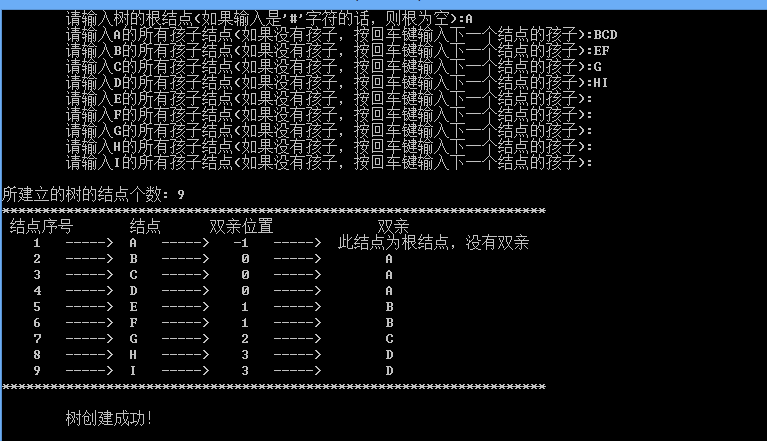
|  |  |  |
| --- | --- | --- |
| **树的双亲表示法基本操作时间复杂度分析** | **InitTree(&T);** | O(1) |
| **DestroyTree(&T);** | O(1) |
| **CreateTree(&T,definition);** | O(n) |
| **ClearTree(&T);** | O(1) |
| **TreeEmpty(T);** | O(1) |
| **TreeDepth(T);** | O(n^2) |
| **Root(T);** | O(1) |
| **Value(T,cur\_e);** | O(1) |
| **Assign(T,cur\_e,value);** | O(1) |
| **Parent(T,cur\_e);** | O(1) |
| **LeftChild(T,cur\_e);** | O(n) |
| **RightSibling(T,cur\_e);** | O(n) |
| **InsertChild(&T,&p,I,c);** | O(n^2) |
| **DeleteChild(&T,&p,i);** | O(n) |
| **Print(T)；** | O(n) |
| **TraverseTree(T,visit());** | O(n) |
| **LevelOrderTraverseTree(T,visit())** | - |
| **PreOrderTraverseTree(T,visit())** | - |
| **PostOrderTraverseTree(T,visit())** | - |
| **优缺点分析** | **优点** | 可以随机存取，不用从头找起  存储的信息位置直观明了 |
| **缺点** | 插入删除时需要移动元素 |

###### 调试过程

以下图的树为例进行测试:



运行程序后，首先是先创建树T：



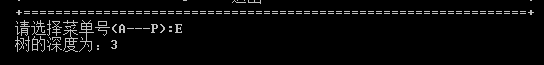
接着按任意键跳到主菜单进行调试：



选择D,判断树是否为空树



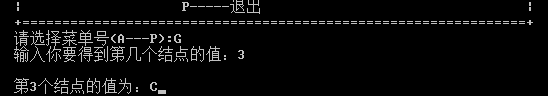
选择E,返回树的深度



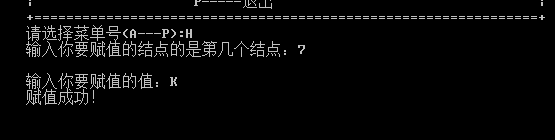
选择F,返回树的根结点



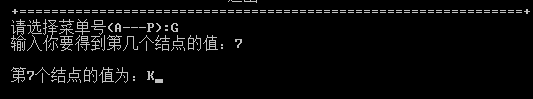
选择G,求第三个结点的值



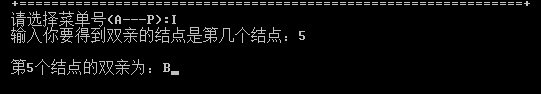
选择F,将第7个结点改为K



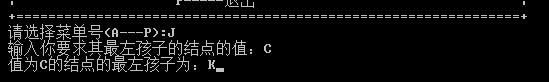
检查是否赋值成功,选择G,求第7个结点的值



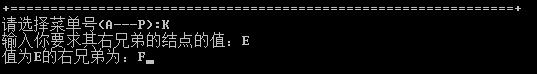
选择I,求第五个结点的双亲



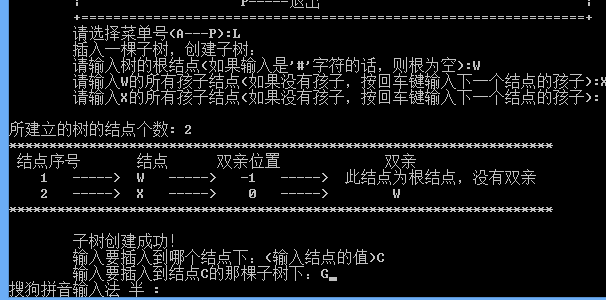
选择J,求C的最左孩子的结点值



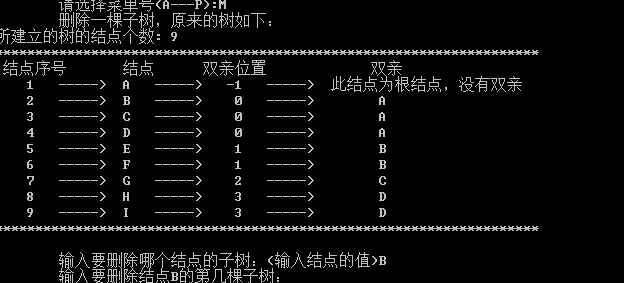
选择K,求结点E的右兄弟



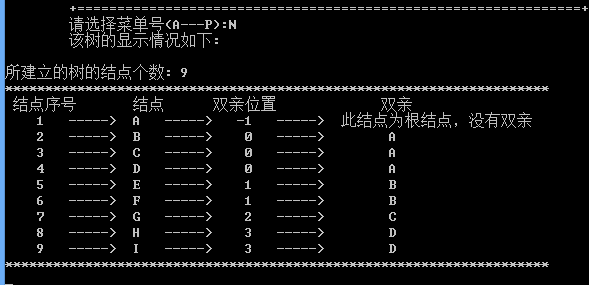
选择L,插入一棵子树



选择M,删除子树



选择N,显示树



选择O,层次遍历树



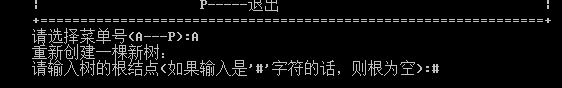
选择C，将树清空



选择B，将树销毁



选择A,构建一棵空树





再选择D,判断树是否为空



###### 选择P,退出



###### 测试结果分析

双亲表存储结构的调试过程，发现些许错误,如插入子树时重新构建子树那里出现了错误,通过查找改正后,最终达到了完成各个操作的基本功能，实验结果与预期基本相符,但其中还是有不少的细微的BUG。

#### 八、思考与小节

1. 在定义变量时要时刻注意该变量的类型，以免影响数值的精度和出现一些不必要的错误。
2. 变量的命名尽量用英文全称，方便使用，同时容易别人阅读。比如子树的名称，要是单用一个字母c,程序一但多了,很容易不知道c是什么意思还有很多ijk等变量,要是将子树命名为Tree1,别人一看代码就知道跟树有关的，大大提高了程序的阅读性。
3. 在算法设计时参数值的合法性一定要注意判断。

#### 九、总结与体会

此次实验是我第一次站在数据结构的思维层次上完成一个程序量相对较大编程任务。在未接触数据结构之前，我的编程经历就只是停留在一点一点的慢慢写出需要实现的功能的层面。总的来说是思路比较混乱，没有太多章法可言的。本次抽象数据类型树的实现以及《数据结构》这门课对我的编程思想进行了一番革命性的洗礼，在此我对如何建立一个合理高效的算法有了一个清楚的认识。在进行实际编程工作是，如果遇到一个没见过的算法问题，那么我会考虑以下问题：

1. 问题是否是建立在某种已知的熟悉的数据结构(例如,二叉树)上?如果不是,则要自己设计数据结构。

2. 问题所要求编写的算法属于以下哪种类型?(建立数据结构,修改数据结构,遍历,查找,排序...) 。

3. 分析问题所要求编写的算法的数学性质.是否具备递归特征?(对于递归程序设计,只要设计出合理的参数表以及递归结束的条件,则基本上大功告成.)。

4. 继续分析问题的数学本质.根据你以前的编程经验,设想一种可能是可行的解决办法,并证明这种解决办法的正确性.如果题目对算法有时空方面的要求,证明你的设想满足其要求.一般的,时间效率和空间效率难以兼得.有时必须通过建立辅助存储的方法来节省时间。

5. 通过一段时间的分析,你对解决这个问题已经有了自己的一些思路.或者说,你已经可以用自然语言把你的算法简单描述出来.继续验证其正确性,努力发现其中的错误并找出解决办法.在必要的时候(发现了无法解决的矛盾),推翻自己的思路,从头开始构思.。

6. 确认你的思路可行以后,开始编写程序.在编写代码的过程中,尽可能把各种问题考虑得详细,周密.程序应该具有良好的结构,并且在关键的地方配有注释.。

7. 举一个例子,然后在纸上用笔执行你的程序,进一步验证其正确性.当遇到与你的设想不符的情况时,分析问题产生的原因是编程方面的问题还是算法思想本身有问题。

8. 如果程序通过了上述正确性验证,那么在将其进一步优化或简化。

9. 撰写思路分析,注释。

这样问题的解决就有了一个清晰的思路，而这种思路正是源于该类实验的训练，以及对《数据结构》这门课的学习。在此非常感谢老师在这一学年对我的指导。谢谢！