# 实验报告三

## 一用先序递归过程建立二叉树

先序递归建立二叉树

班级：2015211306 姓名：李航 学号：2015211284 分工：代码+文档班级：2015211306 姓名：魏晓 学号：2015211301 分工：代码+文档

1. 需求分析

以先序遍历的序列输入一个字符串集，其中没有左子树或者右子树的节点后标记‘\*’，要求以二叉链表结构存储，建立一棵二叉树。如果输入正确，那么程序会输出该二叉树的先序遍历，中序遍历和后序遍历，进而可以检验出存储的正确性。但如果输入不合法，这里的不合法主要是指‘\*’的数量和位置不正确导致无法构成一个确定的二叉树，那么程序将会终止。

测试数据：（1）abc\*\*d\*\*e\*\*

(2) ab\*cd\*\*\*ef\*\*g\*\*

(3) abcd\*\*e\*\*\*fg\*h\*\*\*

2. 概要设计

这里最重要的一步是如何将输入的一系列字符串存储到二叉链表中

首先定义二叉链表的存储结构：

typedef struct BiTNode{

char data; //数据存储区，存储节点字符

struct BiTNode \*lchild,\*rchild; //分别指向该节点的左右孩子

}\*BiTree;

其次，要把输入的字符串存储到该链表中，伪代码如下：

CreatTree(BiTree root)

{

if(当前读到的字符不为空且不为换行符)

root=(BiTree)malloc(sizeof(struct BiTNode)); //创建结点

CreatTree(root->lchild); //创建该节点左子树

CreatTree(root->rchild); //创建该节点右子树

｝

在者就是二叉树的前序遍历，中序遍历和后序遍历的算法，用以输出该二叉树。

1. 详细设计

首先是二叉树的生成代码：

void CreatTree(BiTree & root)

{

char c;

if((c=getchar())!='\*'&&c!='\n')

｛

root=(BiTree)malloc(sizeof(struct BiTNode));

root->data=c;

CreatTree(root->lchild);

CreatTree(root->rchild);

}

else

root=NULL;

}

待创建完二叉树之后就可以用三种遍历方法对该二叉树进行输出了

先序遍历：

void PrePrintTree(BiTree root)

{

if(root!=NULL)

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

if(root->lchild!=NULL)

PrePrintTree(root->lchild);

if(root->rchild!=NULL)

PrePrintTree(root->rchild);

}

中序遍历：

void InPrintTree(BiTree root)

{

if(root==NULL)

exit(0);

if(root->lchild!=NULL)

InPrintTree(root->lchild);

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

if(root->rchild!=NULL)

InPrintTree(root->rchild);

}

后序遍历：

void LatePrintTree(BiTree root)

{

if(root==NULL)

exit(0);

if(root->lchild!=NULL)

LatePrintTree(root->lchild);

if(root->rchild!=NULL)

LatePrintTree(root->rchild);

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

}

这样一颗完整的二叉树就生成了，只要输出结果与预期相符。

1. 调试分析报告

二叉树的建立算法与所有的遍历算法都用的是递归算法，基本操作都是访问结点，建立二叉树时还包括建立操作，遍历二叉树时还包括输出数据操作，但不论是二叉树建立还是二叉树的遍历算法，其时间复杂度都为O（n），所需辅助空间为遍历过程中栈的最大容量，最坏情况下为n，则空间复杂度也为O（n）。算法的设计如果仔细看书的话书上都有指导，当前节点建立好时先递归建立该节点的左子树，然后再递归建立该节点的右子树。

1. 用户使用说明

点击程序运行，会提示你输入一棵二叉树的先序遍历，并且不存在的结点要用‘\*’来代替。输入结束后按下回车，会显示出你所输入的二叉树的先序遍历，中序遍历和后序遍历的结果。输入时候‘\*’的数量和位置一定要保证输入正确，不然可能会建立出和你预料不同的二叉树，更糟糕的情况，程序可能会因为错误而终止。建议先画出你先要存储的二叉树，然后先序遍历一遍再进行输入。

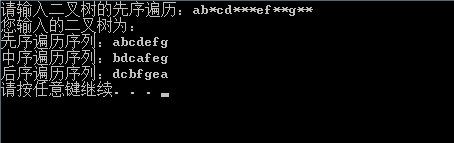
1. 测试结果

之前提到了，有三组测试数据，三组数据的测试结果如下：

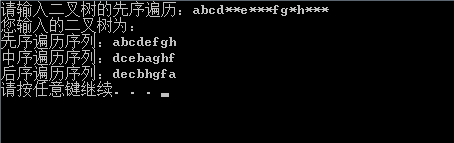
（1）abc\*\*d\*\*e\*\* 该二叉树是题目中所给的二叉树，运行结果如下：



(2) ab\*cd\*\*\*ef\*\*g\*\*

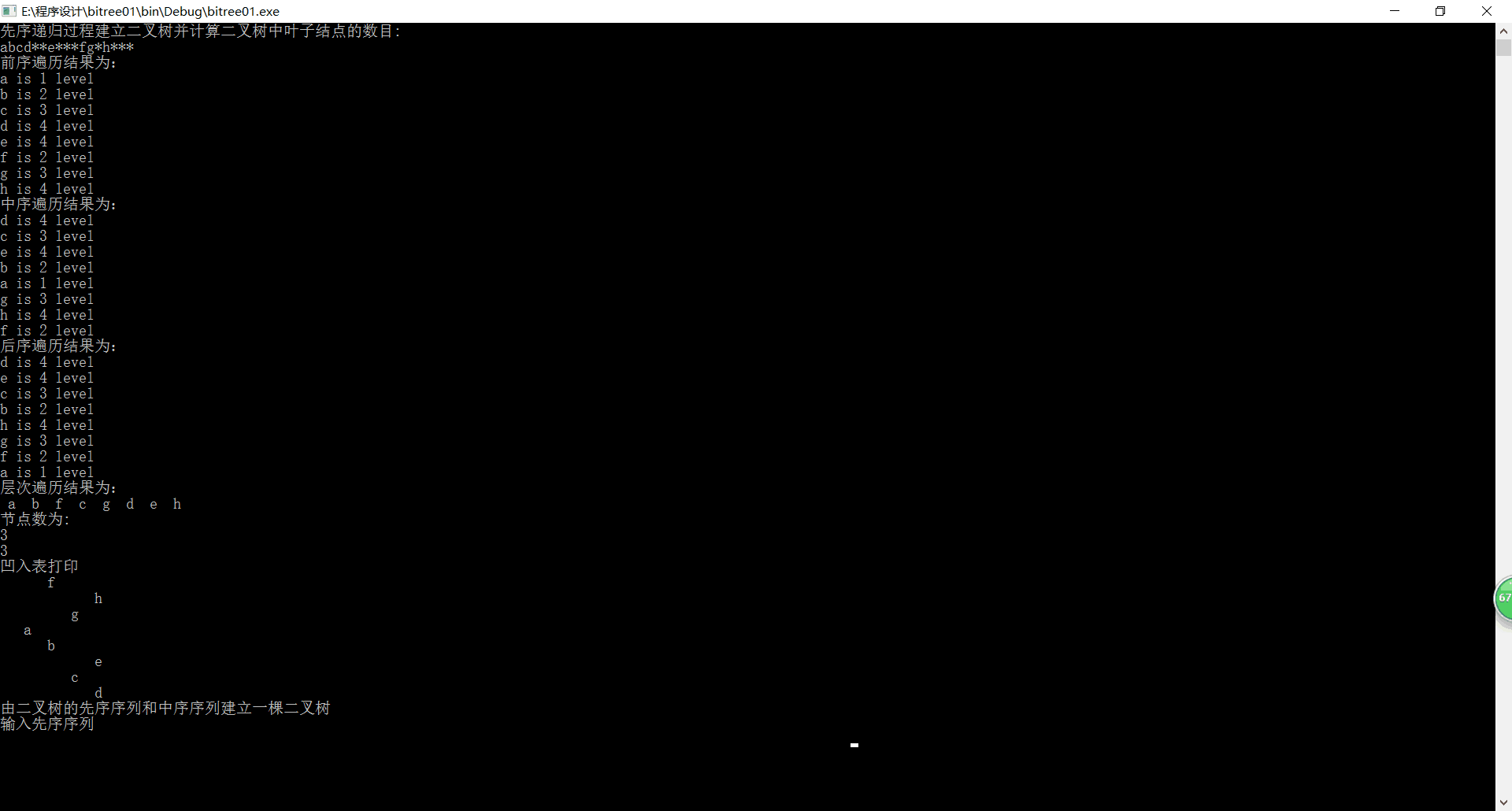


(3) abcd\*\*e\*\*\*fg\*h\*\*\*



经检验，输出的二叉树确是预料之中的二叉树，二叉树建立正确！

另一版本：



二哈夫曼编码

1. 需求分析

将一个文件进行二进制编码，生成一个二进制文件，反过来可以将二进制文件还原为原来的文件。

2概要设计

哈夫曼树存储结构

typedef struct{

unsigned int weight;

unsigned int parent,lchild,rchild;

}HTNode,\*HuffmanTree;

typedef char \*\*HuffmanCode; //字符二维数组，存储编码表

void Info(int \*s); //统计文本文档每个字符的出现次序，每出现一次，对应的下表为其ASSIC码值的数组值加一

void HuffmanCoding(HuffmanTree & HT,HuffmanCode & HC,int \*w,int n); //哈夫曼编码函数

void Select(HuffmanTree HT,int k,int & s1,int & s2); //在HT【1...i-1】选择parent为0且权值最小的两个结点，其序号分别为s1，s2

void BinFileGen(HuffmanCode HC); //生成二进制文件函数

void codee(HuffmanCode HC);

void Decode(HuffmanCode HC,HuffmanTree HT,int n); //解码函数

main

{

HuffmanTree HT;

HuffmanCode HC; //存储编码表

int i,k=0;

int s[256]={0};

Info(s); //用来统计字符的出现次数

HuffmanCoding(HT,HC,s,256); //哈夫曼编码，将编码表存储在HC中

codee(HC);

Decode(HC,HT,n) //解码函数

}

1. 详细设计

首先是建立哈夫曼树和哈夫曼编码表的程序：

void HuffmanCoding(HuffmanTree & HT,HuffmanCode & HC,int \*w,int n) //哈夫曼编码

{

if(n<=1) return;

int m=2\*n-1;

HT=(HuffmanTree)malloc((m+1)\*sizeof(HTNode)); //0号单元未启用

HuffmanTree p;

int i;

for(p=HT+1,i=1;i<=n;++i,++p,++w)

{

p->weight=\*(w+1);

p->parent=p->lchild=p->rchild=0;

}

for(;i<=m;++i,++p)

p->weight=p->parent=p->lchild=p->rchild=0;

int s1,s2;

for(i=n+1;i<=m;++i)

{ //建哈夫曼树

Select(HT,i-1,s1,s2); //在HT【1...i-1】选择parent为0且权值最小的两个结点，其序号分别为s1，s2

HT[s1].parent=i;

HT[s2].parent=i;

HT[i].lchild=s1;

HT[i].rchild=s2;

HT[i].weight=HT[s1].weight+HT[s2].weight;

}

//从叶子结点到根逆向求每个字符的哈夫曼编码

HC=(HuffmanCode)malloc((n+1)\*sizeof(char\*)); //分配n个字符编码的头指针向量

char \*cd=(char\*)malloc(n\*sizeof(char)); //分配编码字符串空间

cd[n-1]='\0';

int c,f; //编码结束符

for(i=1;i<=n;++i) //逐个字符求哈夫曼树

{

int start=n-1; //编码结束符位置

for(c=i,f=HT[i].parent;f!=0;c=f,f=HT[f].parent)

{

if(HT[f].lchild==c)

cd[--start]='0';

else

cd[--start]='1';

}

HC[i]=(char\*)malloc((n-start)\*sizeof(char)); //为第i个字符编码分配空间

strcpy(HC[i],&cd[start]); //从cd复制码串到HC

}

free(cd);

}

然后是将文件通过已有的哈夫曼编码表，解成二进制的形式存储在另一个文件中：

void codee(HuffmanCode HC)

{

ofstream oop("FIFE.txt",ios::app);

ifstream fin;

string exm;

char ch;

queue<int> goal;

fin.open("file.txt");

fin.get(ch);

while(fin.good())

{

exm=HC[(int)ch];

cout<<exm<<endl;

int numm=exm.size();

for(int i=0;i<numm;i++)

{

int sun=exm[i]-'0';

goal.push(sun);

}

if(goal.size()>8)

{

//影响出战顺序

int er=0;

for(int we=0;we<8;we++)

{

//cout<<goal.front()<<endl;

er=er+goal.front()\*((int)pow(2,7-we));

goal.pop();

if(goal.empty())

we=9;

}

cout<<er<<endl;

char c=er;

printf("%c\n",er);

oop<<(char)er;

}

fin.get(ch);

}

int er=0;

for(int we=0;!goal.empty();we++)

{

er=er+goal.front()\*((int)pow(2,7-we));

goal.pop();

}

cout<<(char)er<<endl;

oop<<(char)er;

fin.close();

}

紧接着将二进制文件解码还原成原文件：

void Decode(HuffmanCode HC,HuffmanTree HT,int n)

{

ifstream fin("FIFE.txt");

ofstream fout;

char ch,a;

string str;

ch=fin.get();

cout<<ch<<endl;

while(fin.good())

{

int k=1,t=0;

for (int i=0; i<8; i++,ch>>1)

{

if (ch & k)

str[t\*8+8-i]='1';

else

str[t\*8+8-i]='0';

++t;

}

ch=fin.get();

//cout<<ch<<endl;

}

cout<<&str[1];

fin.close();

int f=n;

fout.open("file1.txt");

if(!fout.is\_open())

cout<<"error";

for(int i=1;str[i];++i)

{

if(str[i]=='1')

{

if(HT[f].rchild)

{

f=HT[f].rchild;

}

else

{

ch=(char)f;

//fout<<f;

fout<<ch;

f=n;

--i;

}

}

else

{

if(HT[f].lchild)

{

f=HT[f].lchild;

}

else

{

//ch=(char)f;

fout<<ch;

f=n;

--i;

}

}

}

fout.close();

}

1. 调试分析报告

调试过程中遇到的问题主要是在文件读取方面，关于文件的代码平时练习太少，有时候会读到莫名其妙的字符，然后字符转二进制串，二进制串转字符也有些费力。

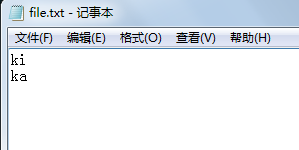
建立哈夫曼树，文本文件转二进制，二进制还原的时间复杂度都为O（n^2），所以程序总的时间复杂度就是O（n^2）。过程中使用了一个字符串数组来存储整个哈夫曼编码表，哈夫曼树使用了一个结构数组来存储，所以程序的空间复杂度为O（n^2）。

5. 用户使用说明

点击运行之后，会将目录下的file.txt文件编码成二进制存储在另一个文件中FILE.txt，同时还会生成一个file1.txt文件，该文件是将FILE.txt解码之后的生成文件。

1. 测试结果

原始文件



还原后的文件

