

《数据结构》

实验报告

姓名： 庄李晨

学号： 10522238

成绩：

电子与计算机工程学院

School of Electronic & Computer Engineering

2023年11月

【实验名称】二叉树的实现

【实验目的】

一、掌握二叉树的基本操作，理解递归算法。

【实验内容】

1. 实验要求：

1． 仔细阅读二叉树实验代码，补充下面8个函数的实现过程。补充完成main（）函数的定义，以下图所示二叉树调试程序，给出测试结果。

（1）以二叉树的先序序列创建二叉树的二叉链表存储结构，先序序列包含空结点

void CreateBiTree(BiTree &T){ }

（2）用递归算法实现前序遍历二叉树，输出遍历序列

void PreOrderTraverse (BiTree T){ }

（3）用递归算法实现中序遍历二叉树，输出遍历序列

void InOrderTraverse (BiTree T){ }

（4）用递归算法实现后序遍历二叉树，输出遍历序列

void PostOrderTraverse (BiTree T){ }

（5）用非递归算法实现中序遍历二叉树，输出遍历序列

void InOrderTraverseNon (BiTree T){ }

（6）求二叉树深度，返回深度值

int Depth (BiTree T ){ }

（7）统计二叉树结点个数，返回二叉树结点个数

int Count (BiTree T){ }

（8）查找二叉树结点，在根指针为T的二叉树中，查找值为x的结点，查找成功，p赋值为该结点指针，返回true；查找失败，p==NULL ，返回false

bool Search (BiTree T, TElemType x, BiTree &p) {}

2.理解赫夫曼编码代码，并给出注释，然后运行如下案例。

（1）用下表给出的字符集和频度的实际统计数据建立赫夫曼树，并实现以下报文的编码：“THIS PROGRAM IS MY FAVORITE”。

字符 空格 A B C D E F G H I J K L M

频度 186 64 13 22 32 103 21 15 47 57 1 5 32 20

字符 N O P Q R S T U V W X Y Z

频度 57 63 15 1 48 51 80 23 8 18 1 16 1

实现提示：

a) 对于字符集比较大时，也可以将字符集及权值存储在文件中，或者用数组存储。

3.选作题目

**题目一 由二叉树遍历序列构建二叉链表存储结构**

设计一个算法，实现已知二叉树的后序和中序遍历序列，或已知二叉树的先序序列、中序序列，建立该二叉树（ 用递归或非递归的方法都可以）

任务：提示用户输入已知的遍历序列，构造结束后输出二叉树的遍历序列。

**题目二 算数表达式与二叉树**

根据已知中缀算数表达式的创建二叉链表存储结构，并进行遍历输出前缀表达式、中缀表达式和后缀表达式。

**题目三 线索二叉树**

完成中序线索二叉树的定义及实现，并进行测试，主要是完成中序线索二叉树的中序遍历。

**题目四 树的孩子兄弟存储**

按照左孩子右兄弟的规则，建立树的存储结构，并实现树的常规操作：遍历、查找等，并进行测试。

**二、程序清单**

#include<iostream>

#include <stdlib.h>

#define MAXSIZE 100

using namespace std;

typedef char TElemType;

// 二叉链表结点结构

typedef struct BiTNode

{

TElemType data;

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

} \*BiTree; //结点类型名为BiTNode，结点指针类型名为BiTree

typedef struct

{

BiTree data[MAXSIZE];

int top;

}SqStack;

//以二叉树的先序序列创建二叉树的二叉链表存储结构，先序序列包含空结点 ，算法6.4

void CreateBiTree(BiTree& T)

{

char ch;

cin >> ch;

if (ch == '#') T = NULL;//递归结束，建空树

else //递归创建二叉树

{

T = new BiTNode;//生成根结点

T->data = ch;

CreateBiTree(T->lchild);//递归创建左子树

CreateBiTree(T->rchild);//递归创建右子树

}

}

// 用递归算法实现前序遍历二叉树，输出遍历序列 ，算法6.1

void PreOrderTraverse(BiTree T)

{

if (T)

{

cout << T->data;//访问根结点

PreOrderTraverse(T->lchild);//先序遍历左子树

PreOrderTraverse(T->rchild);//先序遍历右子树

}

}

// 用递归算法实现中序遍历二叉树，输出遍历序列

void InOrderTraverse(BiTree T)

{

if (T)

{

InOrderTraverse(T->lchild);//中序遍历左子树

cout << T->data;//访问根结点

InOrderTraverse(T->rchild);//中序遍历右子树

}

}

// 用递归算法实现后序遍历二叉树，输出遍历序列

void PostOrderTraverse(BiTree T)

{

if (T)

{

InOrderTraverse(T->lchild);//中序遍历左子树

InOrderTraverse(T->rchild);//中序遍历右子树

cout << T->data;//访问根结点

}

}

void InitStack(SqStack\* S)

{

S->top = -1;

}

int Push(SqStack\* S, BiTree e)

{

if (S->top == MAXSIZE - 1)

return 0;

S->top++;

S->data[S->top] = e;

return 1;

}

void Pop(SqStack\* S, BiTree\* e)

{

if (S->top == -1)

return;

\*e = S->data[S->top];

S->top--;

}

int StackEmpty(SqStack S)

{

if (S.top == -1)

return 1;

else

return 0;

}

// 用非递归算法实现中序遍历二叉树，输出遍历序列，算法6.2或者6.3

void InOrderTraverseNon(BiTree T)

{

SqStack stack;

InitStack(&stack);

Push(&stack, T);

while (!StackEmpty(stack))

{

BiTree p;

Pop(&stack, &p);

cout << p->data;

if (p->rchild)

Push(&stack, p->rchild);

if (p->lchild)

Push(&stack, p->lchild);

}

}

// 求二叉树深度，返回深度值 ，见ppt

int Depth(BiTree T)

{

int leftHeight, rightHeight, maxHeight;//左子树，右子树，最大深度

if (T != NULL) //如果为空树

{

leftHeight = Depth(T->lchild);//左子树深度

rightHeight = Depth(T->rchild);//右子树深度

if (leftHeight > rightHeight) {

maxHeight = leftHeight;

}

else {

maxHeight = rightHeight;

}//最大深度

return maxHeight + 1;//二叉树深度=最大深度+1

}

else

{

return 0;

}

}

// 统计二叉树结点个数，返回二叉树结点个数 ，见ppt

int Count(BiTree T)

{

if (T == NULL)

return 0;

else

return Count(T->lchild) + Count(T->rchild) + 1;

}

// 查找二叉树结点，在根指针为T的二叉树中，查找值为x的结点

//查找成功，p赋值为该结点指针，返回true；查找失败，p==NULL ，返回false

bool Search(BiTree T, TElemType x, BiTree& p)

{

if (!T)

{

p = NULL;

return false;

}

if (T->data == x) {

p = T;

return true;

}

if (Search(T->lchild, x, p)) {

return true;

}

else {

return Search(T->rchild, x, p);

}

}

int main(void)

{

BiTree T=NULL;

int c = 0;

int d = 0;

int num = 0;

TElemType elem;

BiTree p = NULL;

while (c != 9)

{

cout << endl << "1. 建立二叉树的二叉链表";

cout << endl << "2. 前序遍历二叉树";

cout << endl << "3. 中序遍历二叉树";

cout << endl << "4. 后序遍历二叉树";

cout << endl << "5. 非递归中序遍历二叉树";

cout << endl << "6. 求二叉树深度";

cout << endl << "7. 求二叉树结点个数";

cout << endl << "8. 查找结点";

cout << endl << "9. 退出";

cout << endl << "选择功能(1~9):";

cin >> c;

switch (c)

{

case 1:

CreateBiTree(T);

break;

case 2:

PreOrderTraverse(T);

break;

case 3:

InOrderTraverse(T);

break;

case 4:

PostOrderTraverse(T);

break;

case 5:

InOrderTraverseNon(T);

break;

case 6:

d = Depth(T);

cout << "二叉树深度为：" << d;

break;

case 7:

num = Count(T);

cout << "二叉树结点个数为：" << num;

break;

case 8:

cout << "请输入要查找的结点值：";

cin >> elem;

if (Search(T, elem, p))

cout << "结点查找成功！结点左孩子为：" << p->lchild << "结点右孩子为：" << p->rchild;

else

cout << "查找失败！";

break;

case 9:

cout << "结束操作" << endl;

break;

}

}

}

**哈夫曼树**

#include <iostream>

using namespace std;

//最大字符编码数组长度

#define MAXCODELEN 100

//最大哈夫曼节点结构体数组个数

#define MAXHAFF 100

//最大哈夫曼编码结构体数组的个数

#define MAXCODE 100

#define MAXWEIGHT 10000;

typedef struct Haffman

{

//权重

int weight;

//字符

char ch;

//父节点

int parent;

//左儿子节点

int leftChild;

//右儿子节点

int rightChild;

}HaffmaNode;

typedef struct Code

{

//字符的哈夫曼编码的存储

int code[MAXCODELEN];

//从哪个位置开始

int start;

}HaffmaCode;

HaffmaNode haffman[MAXHAFF];

HaffmaCode code[MAXCODE];

void buildHaffman(int all)

{

//哈夫曼节点的初始化之前的工作, weight为0,parent,leftChile,rightChile都为-1

for (int i = 0; i < all \* 2 - 1; ++i)

{

haffman[i].weight = 0;

haffman[i].parent = -1;

haffman[i].leftChild = -1;

haffman[i].rightChild = -1;

}

std::cout << "请输入需要哈夫曼编码的字符和权重大小" << std::endl;

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

{

std::cout << "请分别输入第个" << i << "哈夫曼字符和权重" << std::endl;

std::cin >> haffman[i].ch;

std::cin >> haffman[i].weight;

}

//每次找出最小的权重的节点,生成新的节点,需要all - 1 次合并

int x1, x2, w1, w2;

for (int i = 0; i < all - 1; ++i)

{

x1 = x2 = -1;

w1 = w2 = MAXWEIGHT;

//注意这里每次是all + i次里面便利

for (int j = 0; j < all + i; ++j)

{

//得到最小权重的节点

if (haffman[j].parent == -1 && haffman[j].weight < w1)

{

//如果每次最小的更新了，那么需要把上次最小的给第二个最小的

w2 = w1;

x2 = x1;

x1 = j;

w1 = haffman[j].weight;

}

//这里用else if而不是if,是因为它们每次只选1个就可以了。

else if(haffman[j].parent == -1 && haffman[j].weight < w2)

{

x2 = j;

w2 = haffman[j].weight;

}

}

//么次找到最小的两个节点后要记得合并成一个新的节点

haffman[all + i].leftChild = x1;

haffman[all + i].rightChild = x2;

haffman[all + i].weight = w1 + w2;

haffman[x1].parent = all + i;

haffman[x2].parent = all + i;

std::cout << "x1 is" << x1 <<" x1 parent is"<<haffman[x1].parent<< " x2 is" << x2 <<" x2 parent is "<< haffman[x2].parent<< " new Node is " << all + i << "new weight is" << haffman[all + i].weight << std::endl;

}

}

//打印每个字符的哈夫曼编码

void printCode(int all)

{

//保存当前叶子节点的字符编码

HaffmaCode hCode;

//当前父节点

int curParent;

//下标和叶子节点的编号

int c;

//遍历的总次数

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

{

hCode.start = all - 1;

c = i;

curParent = haffman[i].parent;

//遍历的条件是父节点不等于-1

while (curParent != -1)

{

//我们先拿到父节点，然后判断左节点是否为当前值，如果是取节点0

//否则取节点1,这里的c会变动，所以不要用i表示，我们用c保存当前变量i

if (haffman[curParent].leftChild == c)

{

hCode.code[hCode.start] = 0;

std::cout << "hCode.code[" << hCode.start << "] = 0" << std::endl;

}

else

{

hCode.code[hCode.start] = 1;

std::cout << "hCode.code[" << hCode.start << "] = 1" << std::endl;

}

hCode.start--;

c = curParent;

curParent = haffman[c].parent;

}

//把当前的叶子节点信息保存到编码结构体里面

for (int j = hCode.start + 1; j < all; ++j)

{

code[i].code[j] = hCode.code[j];

}

code[i].start = hCode.start;

}

}

int main()

{

std::cout << "请输入有多少个哈夫曼字符" << std::endl;

int all = 0;

std::cin >> all;

if (all <= 0)

{

std::cout << "您输入的个数有误" << std::endl;

return -1;

}

buildHaffman(all);

printCode(all);

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

{

std::cout << haffman[i].ch << ": Haffman Code is:";

for (int j = code[i].start + 1; j < all; ++j)

{

std::cout << code[i].code[j];

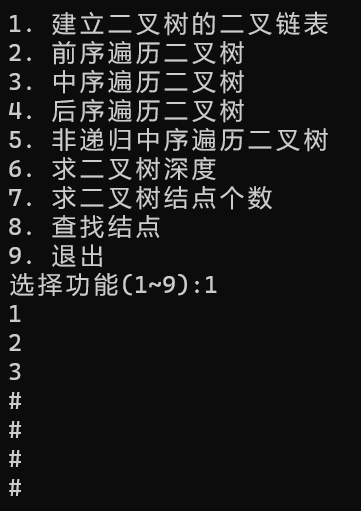
}

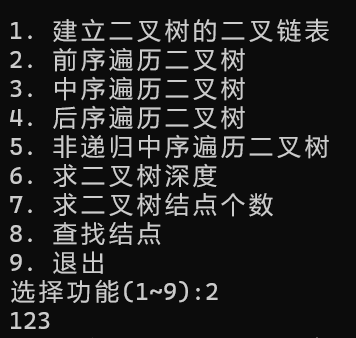
std::cout << std::endl;

}

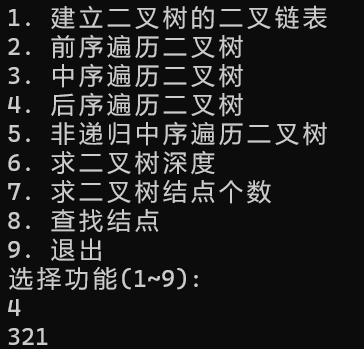
return 0;

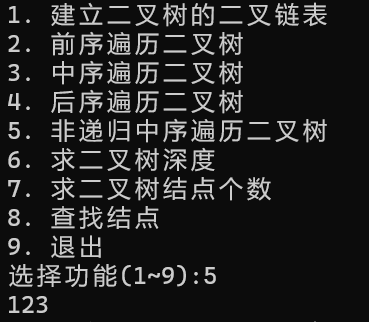
**三、结果截图**



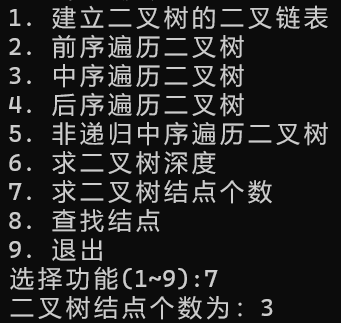






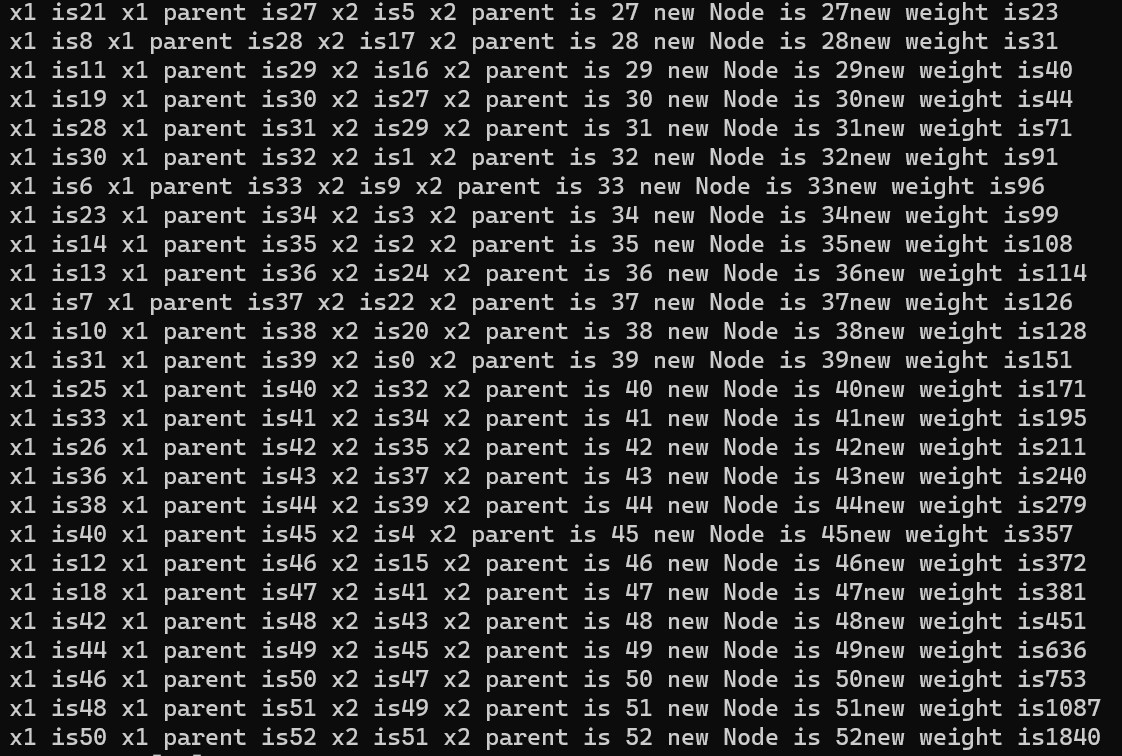


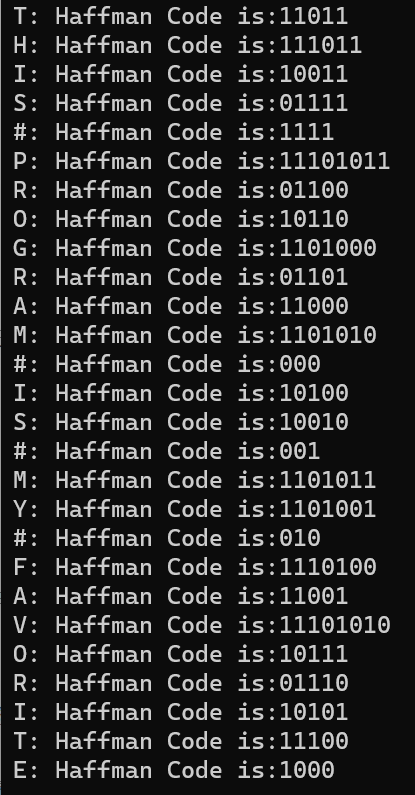






**哈夫曼树**

****



【实验体会】

我熟悉了建立二叉树的方法、先序递归遍历过程和方法、中序递归遍历过程和方法、后序递归遍历过程和方法、求二叉树的高度的函数，以及加深了对哈夫曼树的理解，同时，增强了我对于二叉树孩子与双亲的关系的理解，有助于更好的了解二叉树的结构。