哈尔滨工业大学计算机科学与技术学院  
实验报告

课程名称：数据结构与算法  
课程类型：必修  
实验项目名称： 实验二  
实验题目：建树、哈夫曼树

实验时间： 2021.10.25

班级： 2003009班

姓名： 孙莹

1. 实验目的  
   1.熟悉树的先序、中序、后序非递归算法  
   2.理解哈夫曼树的定义，熟悉其建立方法、使用示例  
   二、实验要求及实验环境

1、已知二叉树的前序序列和中序序列，建立相应的二叉树，完成遍历算法（前、

中、后序的非递归遍历）；要求输出二叉树，二叉树结点个数不少于8 个；

1. 给定任意一篇英文文章，建立哈夫曼树，对每个字符进行编码、译码。

三、设计思想（用到的主要函数、数据类型的定义，主要功能的流程图（1-2个）及各程序模块之间的调用关系，自己扩展内容的等）

1.1主要函数

TREE\* tree\_creat(char A[], char B[], int l1, int h1, int l2, int h2);

**//根据中序和前序序列建树**

void PreOrder(TREE\* T);  **//前序非递归**

void InOrder(TREE\* T);  **//中序非递归**

void PostOrder(TREE\* T);  **//后序非递归**

1.2数据类型

typedef struct TREE {

char data;

TREE\* lchild;

TREE\* rchild;

}TREE;

典型的树结点定义，存储内容、左孩子、右孩子

1.3各程序模块之间的调用关系

**前序非递归遍历思想：**

根-左-右，一边走到最左一边访问，同时需要入栈便于下一步访问右子树，然后出栈，访问出栈结点后，访问出栈结点的右子树，重复以上过程。

**具体实现：**

①栈s 初始化；

② 循环直到root 为空且栈s 为空

当root 不空时循环

输出root->data;

将指针root 的值保存到栈中；

继续遍历root 的左子树

如果栈s 不空，则

将栈顶元素弹出至root； ；

准备遍历root 的右子树；

**中序非递归遍历思想：**

左-根-右，首先走到最左但是不访问，然后出栈访问，然后访问出栈结点的右子树，重复上述过程。

**具体实现：**

①栈 s 初始化；

②循环直到 root 为空且栈s 为空

当 root 不空时循环

将指针 root 的值保存到栈中；

继续遍历 root 的左子树

如果栈s 不空，则

将栈顶元素弹出至 root ；

输出 root- - >data;

准备遍历 root 的右子树

**后序非递归遍历思想：**

左-根-右，首先走到最左但是不访问，然后查看栈顶结点（不出栈！），如果栈顶结点右子树未被访问过，则先访问其右子树；如果其右子树已被访问过，则出栈并访问

**具体实现：**

①栈s 初始化；

②循环直到 root 为空且栈s为空

当 root 非空时循环

将 root 入栈；

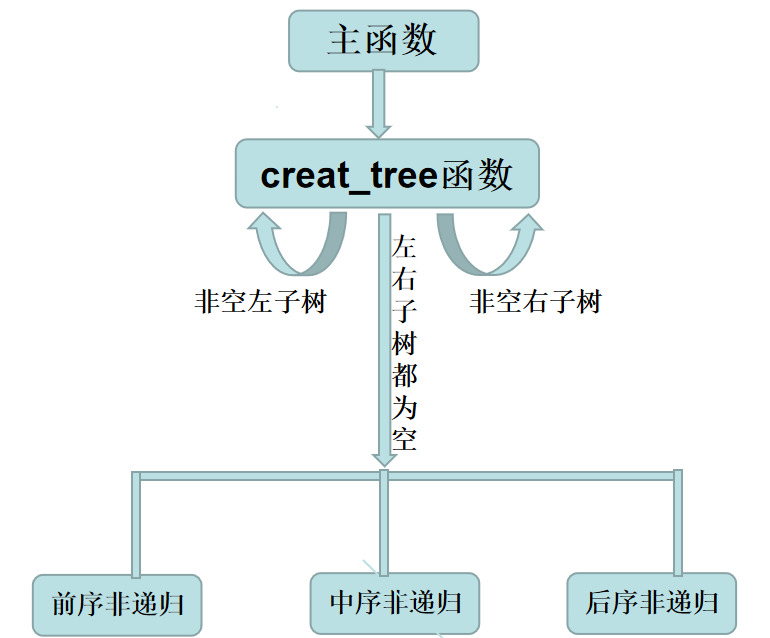
继续遍历 root 的左子树；

当栈 s 非空且刚访问过其右节点时：

出栈并输出栈顶结点;

若栈非空且还未访问其右节点 ：

遍历栈顶结点的右子树；



2.1主要函数

int find\_min(int& min1, int& min2);**//用于建树时寻找值最小的可用结点**

void writein(TREE\* root);**//将编码写入文件**

void readout(TREE\* root);**//读文件将哈夫曼编码翻译为文章**

void Encoding();**//上溯算法找到每个叶结点对应的哈夫曼编码**

2.2数据类型

typedef struct TREE {

int frecy; **//此节点权重值**

int asc; **//用于标志a~z字符位置**

TREE\* lchild;**//左孩子**

TREE\* rchild;**//右孩子**

int parent;  **//父节点在结点表中的位置**

int been; **//标志结点是否可用**

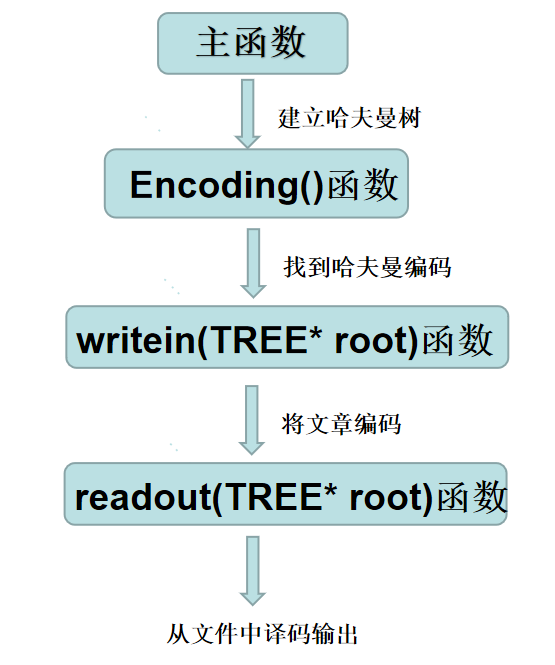
}TREE;

TREE\* eng[100]; **//前26为叶结点，剩下的用于建树时存储新建结点**

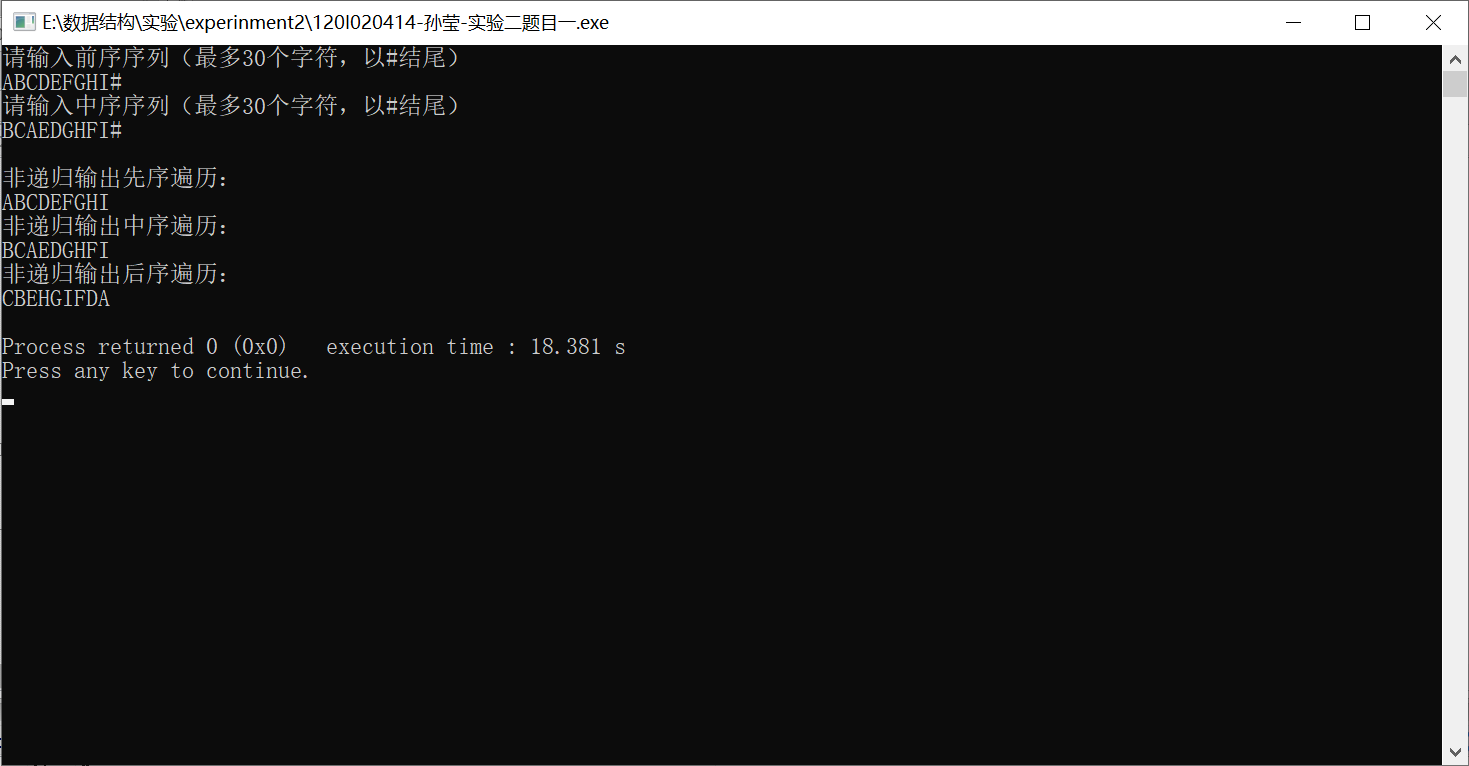
char code[27][50];**//存储哈夫曼编码**

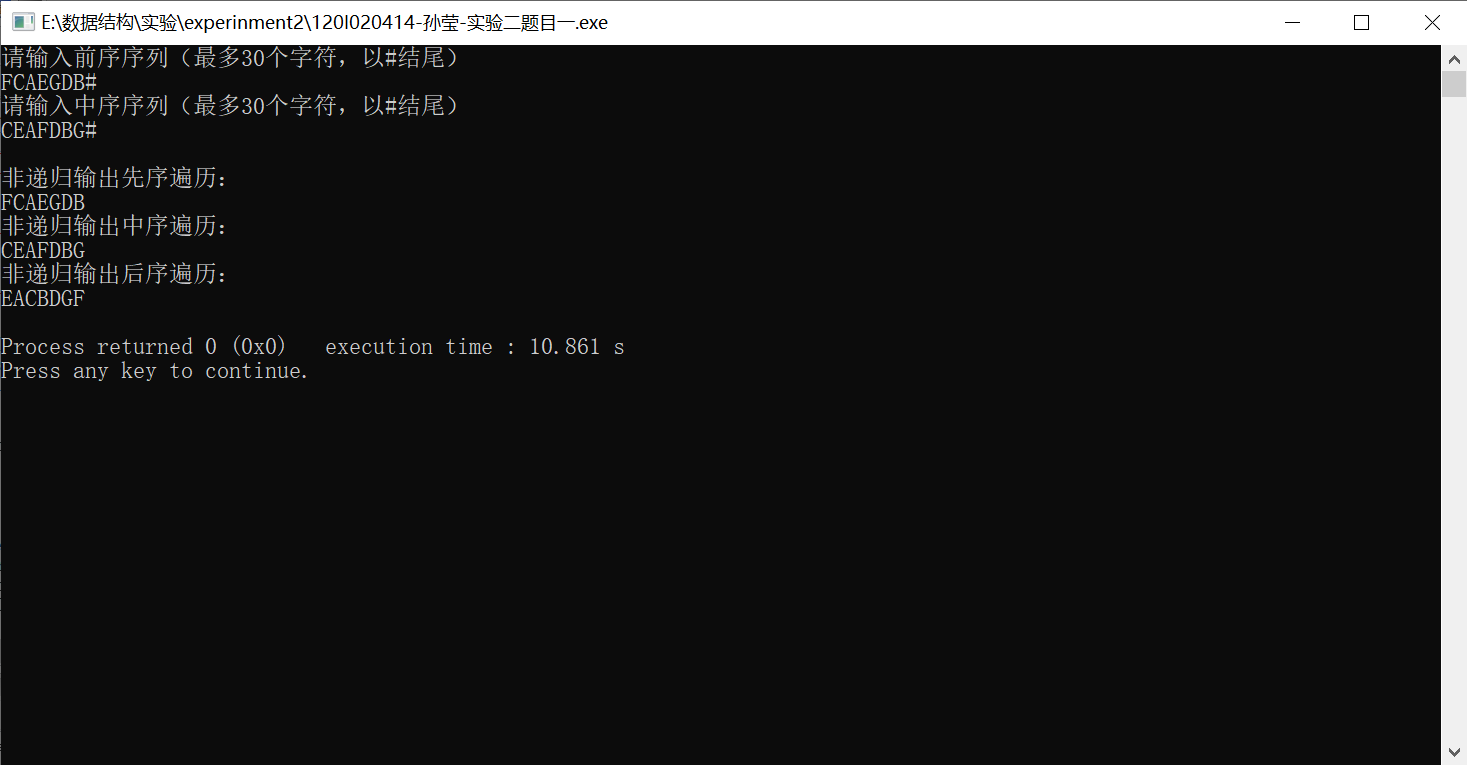
int flag = 26;**//标志有意义结点位置**

2.3各模块之间调用关系

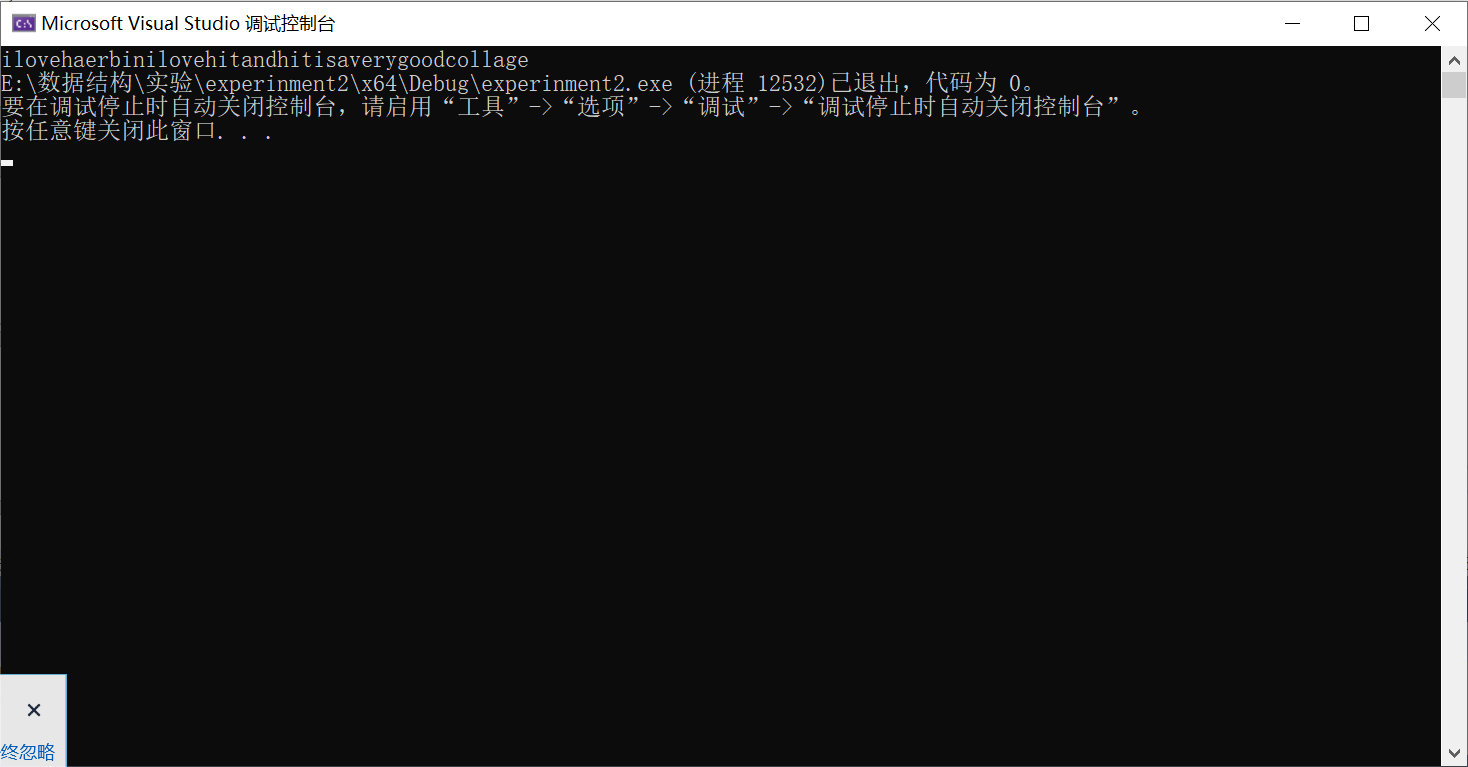


四、测试结果  
1.建树：





2.哈夫曼树



此实例储存在“data.txt”文件内的编码：

110010000011010001010110101001011111001111000000110010000011010001010111101110010100000001101011110111011010001010110100010111110000000010001000100110100100001010001000101000010010

五、系统不足与经验体会

1.建树

不足：只有前序+中序功能，没有后序+中序功能

体会：掌握好前序和中序序列的特点，才能根据这两者建树；前序、中序、后序的非递归算法相对于递归算法代码量更大，但是更容易理解，同时遍历时借用栈结构的方法我觉得很巧妙。

2.哈夫曼树

不足：①只能编码26个小写英文字母，未考虑到大写字母、标点符号等其他字符。

②做到了基础的哈夫曼编码功能，但是没有压缩空间、进一步提高空间利用率。

体会：①精巧的数据结构和储存方式可以大大提高程序运行的空间、时间效率，建立哈夫曼树这种考虑到字符出现频率的编码方式很独特。

②要根据当前存储结构、数据特点设计算法。在建立完哈夫曼树，需要找到每个字符对应的哈夫曼编码时，我开始选择了后序遍历的非递归算法，程序频频出错，后面改为老师PPT中的上溯算法，需要在定义树结构时加上父节点，然后从叶结点上溯到根节点，程序简洁、意思明了，没有再出错。

1. 带注释的源代码

1.//已知二叉树的前序序列和中序序列，建立相应的二叉树，完成遍历算法（前、中、后序的非递归遍历）；要求输出二叉树，二叉树结点个数不少于8 个；

//测试用例①： 前序：ABCDEFGHI#

// 中序：BCAEDGHFI#

// 输出后序：CBEHGIFDA

//测试用例②： 前序：ABDEC#

// 中序：DBEAC#

// 输出后序：DEBCA

//测试用例③： 前序：FCAEGDB#

// 中序：CEAFDBG#

// 输出后序：EACBDGF

#include <iostream>

#include<stack>

using namespace std;

typedef struct TREE {

char data;

TREE\* lchild;

TREE\* rchild;

}TREE;

TREE\* tree\_creat(char A[], char B[], int l1, int h1, int l2, int h2); //根据中序和前序序列建树

void PreOrder(TREE\* T); //前序非递归

void InOrder(TREE\* T); //中序非递归

void PostOrder(TREE\* T); //后序非递归

int main()

{

char A[40], B[40];

int l1 = 1, l2 = 1;

int h1 = 0, h2 = 0;

cout << "请输入前序序列（最多30个字符，以#结尾）" << endl;

do {

h1++;

cin >> A[h1];

} while (A[h1] != '#');

h1--;

cout << "请输入中序序列（最多30个字符，以#结尾）" << endl;

do {

h2++;

cin >> B[h2];

} while (B[h2] != '#');

h2--;

TREE\* root;

root = (TREE\*)malloc(sizeof(TREE));

root = tree\_creat(A, B, l1, h1, l2, h2);

cout << endl;

cout << "非递归输出先序遍历：" << endl;

PreOrder(root);

cout << endl;

cout << "非递归输出中序遍历：" << endl;

InOrder(root);

cout << endl;

cout << "非递归输出后序遍历：" << endl;

PostOrder(root);

cout << endl;

}

TREE\* tree\_creat(char A[], char B[], int l1, int h1, int l2, int h2)

{

TREE\* root;

root = (TREE\*)malloc(sizeof(TREE));

root->data = A[l1];

int i, llen, rlen;

for (i = l2; B[i] != root->data; i++);

llen = i - l2; //左子树长度

rlen = h2 - i; //右子树长度

if (llen != 0)

{

root->lchild = tree\_creat(A, B, l1 + 1, l1 + llen, l2, l2 + llen - 1);

//递归建立左子树

}

else

{

root->lchild = NULL;

}

if (rlen != 0)

{

root->rchild = tree\_creat(A, B, h1 - rlen + 1, h1, h2 - rlen + 1, h2);

//递归建立右子树

}

else

{

root->rchild = NULL;

}

return root;

}

void PreOrder(TREE\* T)

{

TREE\* S[40]; // 递归工作栈

int flag = 0;

struct TREE\* p = T;

while (p != NULL && flag >= 0) {

cout << p->data;

if (p->rchild != NULL)

{

S[flag] = p->rchild;

flag++;

}

if (p->lchild != NULL)

p = p->lchild; // 进左子树

else

{

p = S[flag - 1];

flag--;

}

} // 左子树空, 访问右子树

}

void InOrder(TREE\* T)

{

TREE\* S[40];

int flag = -1;

while (T != NULL || flag != -1)

{

while (T != NULL)

{

S[++flag] = T;

T = T->lchild;

//走到最左

}

if (flag >= 0)

{

T = S[flag];

flag--;

//弹出栈顶

cout << T->data;

T = T->rchild;

//进入右子树

}

}

}

void PostOrder(TREE\* T)

{

TREE\* S[40];

TREE\* r = NULL; //用来标志上一个访问的结点

int top = -1;

while (T != NULL || top != -1)

{

if (T != NULL)

{

top++;

S[top] = T;

T = T->lchild;//走到最左

}

else

{

T = S[top];//读取栈顶【不是弹栈！！】

if (T->rchild != NULL && T->rchild != r)

T = T->rchild;

//如果右子树还未访问过，要先访问右子树

else {

//如果右子树访问过了，那则访问他自己

cout << T->data;

top--;

r = T;

T = NULL;

}

}

}

}

2，//本程序中建树规则为左子树为1，右子树为0

//要求文章文件名为“paper.txt”，文章中只含a~z 26个英文小写字母；储存编码的文件名为“data.txt”。

#include<iostream>

#include<string.h>

using namespace std;

typedef struct TREE {

int frecy; //此节点权重值

int asc; //用于标志a~z字符位置

TREE\* lchild;//左孩子

TREE\* rchild;//右孩子

int parent; //父节点在结点表中的位置

int been; //标志结点是否可用

}TREE;

TREE\* eng[100]; //前26为叶结点，剩下的用于建树时存储新建结点

char code[27][50];//存储哈夫曼编码

int flag = 26;//标志有意义结点位置

int find\_min(int& min1, int& min2);//用于建树时寻找值最小的可用结点

void writein(TREE\* root);//将编码写入文件

void readout(TREE\* root);//读文件将哈夫曼编码翻译为文章

void Encoding();//上溯算法找到每个叶结点对应的哈夫曼编码

int main() {

FILE\* fp;

fopen\_s(&fp, "paper.txt", "r");

if (fp == NULL) {

cout << "打不开文件";

exit(0);

}

//结点表初始化

for (int i = 0; i < 100; i++)

{

eng[i] = (TREE\*)malloc(sizeof(TREE));

eng[i]->lchild = NULL;

eng[i]->rchild = NULL;

eng[i]->frecy = 1;

eng[i]->been = 0;

eng[i]->asc = i;

eng[i]->parent = -1;

}

//计算每个字母的频率

char temp;

while (fscanf\_s(fp, "%c", &temp, 1) != EOF)

{

eng[temp - 'a']->frecy++;

}

fclose(fp);

//开始建树

TREE\* root;

root = (TREE\*)malloc(sizeof(TREE));

int min1, min2;

while (find\_min(min1, min2) != 0)

//如果找到两个最小可用结点，需要新结点作为这两个结点的父节点

{

eng[min1]->parent = flag;

eng[min2]->parent = flag;

eng[flag]->lchild = eng[min1];

eng[flag]->rchild = eng[min2];

eng[flag]->frecy = eng[min1]->frecy + eng[min2]->frecy;

flag++;

}

root = eng[min1];

//最后剩下一个没有父节点的可用结点，也就是根节点

Encoding(); //上溯找每个字母编码

writein(root);//编码

readout(root);//译码

}

int find\_min(int& min1, int& min2)

{

int temp = 9999999;

int min1\_temp = -1, min2\_temp = -1;

//找权值最小结点

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

{

if (eng[i]->been == 0 && eng[i]->frecy < temp)

{

min1\_temp = i;

temp = eng[i]->frecy;

}

}

if (min1\_temp != -1) eng[min1\_temp]->been = 1;

//找权值第二小的结点

temp = 999999;

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

{

if (eng[i]->been == 0 && eng[i]->frecy < temp)

{

min2\_temp = i;

temp = eng[i]->frecy;

}

}

if (min2\_temp != -1) eng[min2\_temp]->been = 1;

min1 = min1\_temp;

min2 = min2\_temp;

if (min1\_temp != -1 && min2\_temp != -1)

{

return 1;

}

else

{

return 0;

}

}

void writein(TREE\* root)

{

FILE\* fp2;

fopen\_s(&fp2, "data.txt", "w");

if (fp2 == NULL)

{

cout << "写入文件失败\n";

exit(0);

}

FILE\* fp;

fopen\_s(&fp, "paper.txt", "r");

if (fp == NULL) {

cout << "打不开文件";

exit(0);

}

char temp;

while (fscanf\_s(fp, "%c", &temp, 1) != EOF)

{

//从“paper.txt”中每次读取一个字符，将其哈夫曼编码写入“data.txt”

for (int j = 0; code[temp - 'a'][j] != '\0'; j++)

{

fprintf(fp2, "%c", code[temp - 'a'][j]);

}

}

fclose(fp);

fclose(fp2);

}

void readout(TREE\* root)

{

TREE\* p = root;

FILE\* fp;

fopen\_s(&fp, "data.txt", "r");

if (fp == NULL) {

cout << "打不开文件";

exit(0);

}

char temp;

while (fscanf\_s(fp, "%c", &temp) != EOF)

{

//从“data.txt”中读取编码

if (temp == '1')//1向左子树走

{

p = p->lchild;

}

else if (temp == '0')//0向右子树走

{

p = p->rchild;

}

if (p->lchild == NULL && p->rchild == NULL)//走到叶结点就输出

{

printf("%c", 'a' + p->asc);

p = root;

}

}

fclose(fp);

}

void Encoding()

{ // 根据Huffman 树T 求Huffman 编码表 H

int c, p, i; // c 和p 分别指示T 中孩子和双亲的位置

char cd[100 + 1]; // 临时存放编码

int start; // 指示编码在cd 中的位置

cd[100] = '\0'; // 编码结束符

for (i = 0; i < 26; i++) { // 依次求叶子T[i] 的编码

start = 100; // 编码起始位置的初值

c = i; // 从叶子T[i] 开始上溯

while ((p = eng[c]->parent) >= 0) { // 直到上溯到T[c] 是树根位置

if (eng[p]->lchild->asc == c)

{

cd[--start] = '1';

}

else

{

cd[--start] = '0';

}

// 若T[c] 是T[p] 的左孩子，则生成代码1 ，否则生成代码0

c = p; // 继续上溯

}

for (int j = 0; j <= 100 - start; j++)

{

code[i][j] = cd[start + j];

} // 复制编码为串于编码表H

}

}