**数据结构实验报告**

**实验三：树和二叉树**

专 业： 网络工程

班 级： 1501

姓 名： 邓泽先

学 号： 150200325

指导教师： 王梅

日 期： 2016-11-16

**一、实验目的**

（1）掌握二叉树的存储实现。

（2）掌握二叉树的遍历思想。

（3）掌握二叉树的常见算法的程序实现。

**二、实验内容**

**1.基本训练**

（1）输入字符序列，建立二叉链表。

（2）先序、中序、后序遍历二叉树:递归算法。

（3）中序遍历二叉树:非递归算法（最好也能实现先序，后序非递归算法）。

（4）求二叉树的高度 。

（5）求二叉树的叶子个数。

（6）对于树中每一个元素值为x的结点，删去以它为根的子树，并释放相应的空间。

（8）借助队列实现二叉树的层次遍历。

（9）在主函数中设计一个简单的菜单，分别调试上述算法。

**2. 综合训练：哈夫曼编/译码器**

【问题描述】

利用哈夫曼编码进行通信可以大大提高信道利用率，缩短信息传输时间，降低传输成本。但是，这要求在发送端通过一个编码系统对待传数据预先编码，在接收端将传来的数据进行译码(复原)。对于双工信道(即可以双向传输信息的信道)，每端都需要一个完整的编/译码系统。试为这样的信息收发站写一个哈夫曼码的编/译码系统。

【基本要求】

一个完整的系统应具有以下功能:

(1) I：初始化(Initialization)。从终端读入字符集大小n，以及n个字符和n个权值，建立哈夫曼树，并将它存于文件hfmTree中。

(2) E：编码(Encoding)。利用已建好的哈夫曼树〔如不在内存，则从文件hfmTree中读入)，对文件中的正文进行编码，然后将结果存入文件CodeFile中。

(3) D：译码(Decoding)。利用已建好的哈夫曼树将文件CodeFile中的代码进行译码，结果存入文件TextFile中。

(4) P：印代码文件(Print)。将文件CodeFile以紧凑格式显示在终端上，每行50个代码。同时将此字符形式的编码文件写入文件CodePrint中。

(5) T：印哈夫曼树(Tree printing)。将已在内存中的哈夫曼树以直观的方式显示在终端上，同时将此字符形式的哈夫曼树写入文件TreePrint中。

**三、需求分析**

1.输入的形式

实验内容1中的输入节点时每个节点的元素以字符的形势读取，读取过程中会过滤掉’\t’、’\n’、’ ’等空白字符。

实验内容2中需要输入字符集的大小和字符集以及它的权重，程序将其处理为先输入字符集的大小后输入字符和该字符的权重，每组数据输入结束后回车键再输入下面的数据。

2.输出形式

实验内容1中的所需输出的内容都会在终端输出；实验内容2会将实验得到的数据输出到相对应的文件中，并将一部分内容输出到终端。

3.程序所能达到的功能

建立二叉树和霍夫曼编/译码器，能够实现实验内容中所包含的功能。

4.测试数据：见第八部分。

**四、概要设计**

(1) 本程序包含的函数

基本训练：

int main(void) 主函数

Status InitBiTree(BiTree t) 树初始化函数

Boolean BiTreeEmpty(BiTree t) 检验树是否为空的函数

BiTree CreateBiTree(BiTree t) 创建树（树中的节点）函数

void PreOrderResu(BiTree t) 递归先序遍历树函数

void MidOrderResu(BiTree t) 递归中序遍历树函数

void PasOrderResu(BiTree t) 递归后序遍历树函数

void PreOrderNotResu(BiTree t) 非递归先序遍历树函数

void MidOrderNotResu(BiTree t) 非递归中序遍历树函数

void PasOrderNotResu(BiTree t) 非递归后序遍历树函数

void BiTreeDepth(BiTree t, int level, int \* depth) 求树深度函数

int Leaf(BiTree t) 求树叶子个数函数

void DeleteNode(BiTree deleteElem) 删除节点函数

void DeleteChildTree(BiTree t, char x) 删除子树函数

void TraverseByQueue(BiTree t) 层次遍历函数

int Menu() 菜单显示和选择函数

综合训练：

int main(void) 主函数

void Select(HuffmanTree ht, int i, int \* section1, int \* section2) 挑选函数

void createHFMTreeFile(HuffmanTree ht, HuffmanCode hc, int n, char \* ch) 创建hfmFile文件函数

void Initialization(HuffmanTree \* ht, HuffmanCode \* hc, int \* N, char \*\* ch) 初始化函数

void Encoding(HuffmanTree \* ht, HuffmanCode \* hc, int \* n, char \* str, char \*\* ch) 编码函数

void Decoding(HuffmanTree ht, int n, char \* ch) 译码函数

void Print() 编码打印函数

void TreePrint(HuffmanTree ht, HuffmanTree start, int count, FILE \* fp) 打印树中某节点函数

void TreePrinting(HuffmanTree ht, int n) 霍夫曼树打印函数

int Menu() 菜单显示和选择函数

(2) 本程序包含的自定义数据类型

基本训练：

typedef int Status;

typedef int Boolean;

typedef char ElemType;

typedef struct BiTNode{

ElemType data;

struct BiTNode \* lchild, \* rchild;

}BiTNode, \* BiTree;

综合训练：

typedef struct HTNode{

unsigned int weight;

unsigned int parent, lchild, rchild;

}HTNode, \* HuffmanTree;

typedef char \*\* HuffmanCode;

**五、详细设计**

(1) 本程序包含的函数

基本训练：

int main(void)

直接或间接调用基本训练中的其他函数。在程序运行最后返回0

Status InitBiTree(BiTree t)

传入一个二叉树指针t，将该二叉树指针t初始化，如果初始化成功返回Ok

Boolean BiTreeEmpty(BiTree t)

传入一个二叉树指针t，检验以t为根的树是否为空，若为空返回True；否则返回False

BiTree CreateBiTree(BiTree t)

传入一个二叉树指针t，以该指针为根从终端输入元素来创建树，并返回这个树的根节点

void PreOrderResu(BiTree t)

传入一个二叉树指针t，以递归的方法先序遍历该树并输出各节点中元素的内容

void MidOrderResu(BiTree t)

传入一个二叉树指针t，以递归的方法中序遍历该树并输出各节点中元素的内容

void PasOrderResu(BiTree t)

传入一个二叉树指针t，以递归的方法后序遍历该树并输出各节点中元素的内容

void PreOrderNotResu(BiTree t)

传入一个二叉树指针t，以非递归的方法先序遍历该树并输出各节点中元素的内容

void MidOrderNotResu(BiTree t)

传入一个二叉树指针t，以非递归的方法中序遍历该树并输出各节点中元素的内容

void PasOrderNotResu(BiTree t)

传入一个二叉树指针t，以非递归的方法后序遍历该树并输出各节点中元素的内容

void BiTreeDepth(BiTree t, int level, int \* depth)

函数需传入一个树节点元素、当前节点的深度和当前所到达节点中深度最大的树的指针。通过递归的方法得到树的深度并将depth指针指向树的深度的值

int Leaf(BiTree t)

传入一个二叉树指针t，以递归的方法求得树的叶子个数并将其返回

void DeleteNode(BiTree deleteElem)

传入一个二叉树节点指针deleteElem，删除释放以该指针为根的树（被DeleteChildTree(BiTree t, char x)函数调用使用）

void DeleteChildTree(BiTree t, char x)

传入一个二叉树指针t和一个元素x，删除以t为根的树中以值和x内容相同的节点为根的子树

void TraverseByQueue(BiTree t)

传入一个二叉树指针t，利用队列将指针t中的各节点层次遍历

int Menu()

输出菜单，并让用户进行选择，将用户的选择返回

综合训练：

int main(void)

直接或间接调用综合训练中的其他函数。在程序运行最后返回0

void Select(HuffmanTree ht, int i, int \* section1, int \* section2)

函数接收霍夫曼树节点指针ht、int类型数据i以及两个int\*的指针，在ht的第1到第i个元素中找到parent为0的最小两个节点，并把下标较小的元素的下标由section1指向，让section2指向较大的元素的下标

void createHFMTreeFile(HuffmanTree ht, HuffmanCode hc, int n, char \* ch)

函数接收霍夫曼树节点指针ht、指向树中叶子节点对应字符的霍夫曼编码的HuffmanCode类变量hc、字符的个数n以及n个字符的地址，根据这些创建一个名为hfmTree.txt的文件。文件的首行保存字符的个数n，随后保存霍夫曼树，再后面保存字符和该字符对应的霍夫曼编码（每个字符和其对应的霍夫曼编码组成一组数据占据一行）

void Initialization(HuffmanTree \* ht, HuffmanCode \* hc, int \* N, char \*\* ch)

函数接收霍夫曼树节点指针ht的地址、指向树中叶子节点对应字符的霍夫曼编码的HuffmanCode类变量hc的地址、字符的个数n的地址以及n个字符的地址的地址，从终端读入字符集大小n，以及n个字符和n个权值，建立霍夫曼树，并通过调用createHFMTreeFile(HuffmanTree ht, HuffmanCode hc, int n, char \* ch) 将它存于文件hfmTree.txt中。将ht、hc，N，ch指向对应的霍夫曼树、各字符的编码、字符的个数、字符集。

void Encoding(HuffmanTree \* ht, HuffmanCode \* hc, int \* n, char \* str, char \*\* ch)

函数接收霍夫曼树节点指针ht的地址、指向树中叶子节点对应字符的霍夫曼编码的HuffmanCode类变量hc的地址、字符的个数n的地址、指向正文的指针str以及n个字符的地址的地址，利用已建好的哈夫曼树〔如不在内存，则从文件hfmTree.txt中读入)，对文件中的正文进行编码，然后将结果存入文件CodeFile中

void Decoding(HuffmanTree ht, int n, char \* ch)

函数接收霍夫曼树节点指针ht、字符的个数n以及n个字符的地址。利用已建好的哈夫曼树ht将文件CodeFile.txt中的代码进行译码，结果存入文件TextFile.txt中

void Print()

函数将文件CodeFile.txt中的编码以紧凑格式显示在终端上，每行50个代码。同时将此字符形式的编码文件写入文件CodePrint.txt中

void TreePrint(HuffmanTree ht, HuffmanTree start, int count, FILE \* fp)

函数接收霍夫曼树节点指针ht、霍夫曼树节点指针start、整型数字count，文件指针fp。通过递归的方法将已在内存中ht中的以start为开端的哈夫曼树以直观的方式显示在终端上，并将其写入fp指向的文件中

void TreePrinting(HuffmanTree ht, int n)

函数接收霍夫曼树节点指针ht、字符的个数n，将已在内存中的哈夫曼树ht以直观的方式显示在终端上，同时利用TreePrint(HuffmanTree ht, HuffmanTree start, int count, FILE \* fp)函数将此字符形式的哈夫曼树写入文件TreePrint.txt中

int Menu()

输出菜单，并让用户进行选择，将用户的选择返回

(2) 本程序包含的自定义数据类型

基本训练：

#define True 1

#define False 0

#define OK 1

#define Error 0

typedef int Status;

函数运行状态，正常运行返回Ok；否则返回Error

typedef int Boolean;

返回结果，True或False

typedef char ElemType;

元素数据类型ElemType

typedef struct BiTNode{

ElemType data;

struct BiTNode \* lchild, \* rchild;

}BiTNode, \* BiTree;

包含一个元素数据类型和左右孩子节点的二叉树节点BiTNode和二叉树节点指针BiTree（简称为二叉树指针）

综合训练：

typedef struct HTNode{

unsigned int weight;

unsigned int parent, lchild, rchild;

}HTNode, \* HuffmanTree;

包含节点权重、双亲下标、左右孩子下标的霍夫曼树节点HTNode和霍夫曼树节点指针HuffmanTree

typedef char \*\* HuffmanCode;

用来指向字符对应霍夫曼编码的HuffmanCode

**六、调试分析**

调试过程比较麻烦，尤其是综合训练中的读文件部分。

**七、使用说明**

基本训练程序后会有以下的菜单:

1.初始化并创建树：

2.先序遍历链表（递归）：

3.中序遍历链表（递归）：

4.后序遍历链表（递归）：

5.先序遍历链表（非递归）：

6.中序遍历链表（非递归）：

7.后序遍历链表（非递归）：

8.二叉树的高度：

9.二叉树的叶子个数：

10.请输入一个值，然后会删除树中每一个元素值为x的结点：

11.实现二叉树的层次遍历：

0.退出程序：

请输入您的选择：

综合训练程序运行后会出现以下的菜单：

有以下选择：

1.初始化

2.编码

3.译码

4.印代码文件

5.印哈夫曼树

0.退出程序

请输入您的选择：

选择要进行的操作，如果出现了非法的操作，会有合理提示。

例如：二叉树为空时就进行其他操作，会有提示：树为空，未初始化或输入值，请重新输入!！内存中没有霍夫曼树就进行了需要操作内存中的霍夫曼树，可能会有提示：霍夫曼树为空，请先确保内存中有霍夫曼树。

**八、测试结果**

基本训练：

1. 初始化并创建树：

选择1，然后输入树的内容124##5##3##，初始化并创建树

1. 先序遍历链表（递归）

选择2，然后会输出：先序遍历的结果为：12453

1. 中序遍历链表（递归）

选择3，然后会输出：中序遍历的结果为：42513

1. 后序遍历链表（递归）

选择4，然后会输出：后序遍历的结果为：45231

1. 先序遍历链表（非递归）

选择5，然后会输出：先序遍历的结果为：12453

1. 中序遍历链表（非递归）

选择6，然后会输出：中序遍历的结果为：42513

1. 后序遍历链表（非递归）

选择7，然后会输出：后序遍历的结果为：45231

1. 二叉树的高度

选择8，然后会输出：树的深度为：3

1. 二叉树的叶子个数

选择9，然后会输出：叶子的个数为：3

1. 请输入一个值，然后会删除树中每一个元素值为x的结点：

选择10，然后再输入4，删除以值4为元素值的节点

1. 实现二叉树的层次遍历：

选择11，然后会输出：层次遍历的结果为：1235

1. 退出程序

选择0，退出程序

综合训练：

有三组测试数据：

一、

1. 初始化

选择1，然后输入字符集的大小8，然后输入字符和对应的权重：

1 5

2 29

3 7

4 8

5 14

6 23

7 3

8 11

1. 编码

选择2，然后输入正文：18723465。

1. 译码

选择3，然后会输出译码结果为：18723465

1. 印代码文件

选择4，然后会输出代码文件中的代码为：01100100111101110111100110

1. 印哈夫曼树

选择5，然后会输出打印霍夫曼树：

8

15

7

29

14

58

29

100

3

8

5

19

11

42

23

1. 退出程序

选择0，退出程序

二、

1. 初始化

选择1，然后输入27，然后输入：

186

A 64

B 13

C 22

D 32

E 103

F 21

G 15

H 47

I 57

J 1

K 5

L 32

M 20

N 57

O 63

P 15

Q 1

R 48

S 51

T 80

U 23

V 8

W 18

X 1

Y 16

Z 1

1. 编码

选择2，然后输入THIS PROGRAM IS MY FAVORITE

1. 译码

选择3，然后会输出译码结果为：THIS PROGRAM IS MY FAVORITE

1. 印代码文件

选择4，然后会输出：

代码文件中的代码为：

11100000011000111101001100010100010010100101010111

11111001100011110111111100111110111110101011110101

000001001101110010

1. 印哈夫曼树（为正常显示结果，霍夫曼树用了较小的字体）

选择5，然后会输出：打印霍夫曼树：

20

41

21

76

1

2

1

4

1

2

1

9

5

17

8

35

18

156

80

342

186

592

32

64

32

128

64

250

16

31

15

59

15

28

13

122

63

1000

57

114

57

217

103

408

51

99

48

191

23

45

22

92

47

1. 退出程序

选择0，退出程序

三、（第三组测试数据是为测试操作2中的从文件中读取树是否能够正常实现。PS：测试前请先确保hfmTree.txt文件中的数据符合要求）

1. （不使用操作1）
2. 编码

选择2，然后输入THIS PROGRAM IS MY FAVORITE

1. 译码

选择3，然后会输出译码结果为：THIS PROGRAM IS MY FAVORITE

1. 印代码文件

选择4，然后输出：

代码文件中的代码为：

11100000011000111101001100010100010010100101010111

11111001100011110111111100111110111110101011110101

000001001101110010

1. 印哈夫曼树（为正常显示结果，霍夫曼树用了较小的字体）

选择5，然后会输出：打印霍夫曼树：

20

41

21

76

1

2

1

4

1

2

1

9

5

17

8

35

18

156

80

342

186

592

32

64

32

128

64

250

16

31

15

59

15

28

13

122

63

1000

57

114

57

217

103

408

51

99

48

191

23

45

22

92

47

1. 退出程序

选择0，退出程序

**九、附录：部分代码**

**基本训练：**

BiTreeMain.c:

#include<stdio.h>

#include "BiTree.h"

int main(void)

{

BiTree t = NULL;

char ch;

int choice, depth = 0;

while((choice = Menu()) != 0)

{

switch(choice)

{

case 1:

InitBiTree(t);

printf("请输入树的内容（输入'#'时当前节点为空）\n");

t = CreateBiTree(t);

break;

case 2:

if(BiTreeEmpty(t))

continue;

printf("先序遍历的结果为：\n");

PreOrderResu(t);

break;

case 3:

if(BiTreeEmpty(t))

continue;

printf("中序遍历的结果为：\n");

MidOrderResu(t);

break;

case 4:

if(BiTreeEmpty(t))

continue;

printf("后序遍历的结果为：\n");

PasOrderResu(t);

break;

case 5:

if(BiTreeEmpty(t))

continue;

printf("先序遍历的结果为：\n");

PreOrderNotResu(t);

break;

case 6:

if(BiTreeEmpty(t))

continue;

printf("中序遍历的结果为：\n");

MidOrderNotResu(t);

break;

case 7:

if(BiTreeEmpty(t))

continue;

printf("后序遍历的结果为：\n");

PasOrderNotResu(t);

break;

case 8:

if(BiTreeEmpty(t))

continue;

BiTreeDepth(t, 1, &depth);

printf("树的深度为：%d\n", depth);

break;

case 9:

if(BiTreeEmpty(t))

continue;

printf("叶子的个数为：%d\n", Leaf(t));

break;

case 10:

if(BiTreeEmpty(t))

continue;

printf("请输入内容：\n");

scanf(" %c", &ch);

DeleteChildTree(t, ch);

break;

case 11:

if(BiTreeEmpty(t))

continue;

printf("层次遍历的结果为：\n");

TraverseByQueue(t);

break;

default:

break;

}

printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

}

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*END\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

return 0;

}

BiTree.h:

#include<stdio.h>

#include<malloc.h>

#include<stdlib.h>

#define True 1

#define False 0

#define OK 1

#define Error 0

int MAX = 0;

typedef int Status;

typedef int Boolean;

typedef char ElemType;

typedef struct BiTNode{

ElemType data;

struct BiTNode \* lchild, \* rchild;

}BiTNode, \* BiTree;

Status InitBiTree(BiTree t)

{

t = NULL;

return OK;

}

Boolean BiTreeEmpty(BiTree t)

{

if(!t)

{

printf("树为空，未初始化或输入值，请重新输入!!\n");

printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

return True;

}

return False;

}

BiTree CreateBiTree(BiTree t)//创建树，输入#时该方向结束

{

ElemType e;

scanf(" %c", &e);

if(e != '#')

{

MAX++;

t = (BiTree) malloc (sizeof(BiTNode));

if(BiTreeEmpty(t))

{

printf("内存分配失败！！\n");

exit(-1);

}

t->data = e;

t->lchild = CreateBiTree(t->lchild);

t->rchild = CreateBiTree(t->rchild);

}

else

{

t = NULL;

}

return t;

}

void PreOrderResu(BiTree t)//递归先序遍历树

{

if(t != NULL)

{

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

PreOrderResu(t->lchild);

PreOrderResu(t->rchild);

}

}

void MidOrderResu(BiTree t)//递归中序遍历树

{

if(t != NULL)

{

MidOrderResu(t->lchild);

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

MidOrderResu(t->rchild);

}

}

void PasOrderResu(BiTree t)//递归后序遍历树

{

if(t != NULL)

{

PasOrderResu(t->lchild);

PasOrderResu(t->rchild);

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

}

}

void PreOrderNotResu(BiTree t)//非递归先序遍历树

{

BiTree p = t;

BiTree node[MAX];

int top = 0;

do

{

while(p != NULL)

{

node[top++] = p;

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

p = p->lchild;

}

if(top > 0)

{

p = node[--top];

p = p->rchild;

}

}while(top > 0 || p != NULL);

}

void MidOrderNotResu(BiTree t)//非递归中序遍历树

{

BiTree p = t;

BiTree node[MAX];

int top = 0;

do

{

while(p != NULL)

{

node[top++] = p;

p = p->lchild;

}

if(top > 0)

{

p = node[--top];

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

p = p->rchild;

}

}while(top > 0 || p != NULL);

}

void PasOrderNotResu(BiTree t)//非递归后序遍历树

{

BiTree p = t;

BiTree node[MAX];

int top = 0;

Boolean flag[MAX];//记录当前节点的右儿子是否访问过

do

{

while(p != NULL)

{

flag[top] = False;

node[top++] = p;

p = p->lchild;

}

if(top > 0)

{

p = node[--top];

if(p->rchild != NULL && !flag[top])

{

flag[top] = True;

top++;

p = p->rchild;

}

else

{

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

p = NULL;

}

}

}while(top > 0 || p != NULL);

}

void BiTreeDepth(BiTree t, int level, int \* depth)

{

if(t != NULL)

{

if(level > \*depth)

\*depth = level;

BiTreeDepth(t->lchild, level+1, depth);

BiTreeDepth(t->rchild, level+1, depth);

}

}

int Leaf(BiTree t)

{

if(t == NULL)

return 0;

else if(t->lchild == NULL && t->rchild == NULL)

return 1;

else

return Leaf(t->lchild) + Leaf(t->rchild);

}

void DeleteNode(BiTree deleteElem)//删除传递过来的节点以及该节点的子节点

{

if(deleteElem != NULL)

{

DeleteNode(deleteElem->lchild);

DeleteNode(deleteElem->rchild);

free(deleteElem);

}

deleteElem = NULL;

}

void DeleteChildTree(BiTree t, char x)

{

BiTree p = t;

int top = 0, count = 0;

BiTree a[MAX], deleteElems[MAX];

do

{

while(p != NULL)

{

if(p->data == x)

{

if(p == t)

{

DeleteNode(p);

return;

}

else if(a[top]->rchild == p)

{

a[top]->rchild = NULL;

deleteElems[count++] = p;

}

else if(a[top-1]->lchild == p)

{

a[top-1]->lchild = NULL;

deleteElems[count++] = p;

}

}

a[top++] = p;

p = p->lchild;

}

if(top > 0)

{

p = a[--top];

p = p->rchild;

}

}while(top > 0 || p != NULL);

while(count > 0)

{

DeleteNode(deleteElems[--count]);

}

}

void TraverseByQueue(BiTree t)

{

BiTree p = t;

BiTree que[MAX];

int front = 0, rear = 0;

if(p)

{

que[rear++] = p;

}

while(rear != front)

{

p = que[front++];

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

if(p->lchild)

{

que[rear++] = p->lchild;

}

if(p->rchild)

{

que[rear++] = p->rchild;

}

}

}

int Menu()

{

int choice;

printf("1.初始化并创建树：\n");

printf("2.先序遍历链表（递归）：\n");

printf("3.中序遍历链表（递归）：\n");

printf("4.后序遍历链表（递归）：\n");

printf("5.先序遍历链表（非递归）：\n");

printf("6.中序遍历链表（非递归）：\n");

printf("7.后序遍历链表（非递归）：\n");

printf("8.二叉树的高度：\n");

printf("9.二叉树的叶子个数：\n");

printf("10.请输入一个值，然后会删除树中每一个元素值为x的结点：\n");

printf("11.实现二叉树的层次遍历：\n");

printf("0.退出程序：\n");

printf("请输入您的选择：\n");

scanf("%d", &choice);

return choice;

}

**综合训练：**

Huffman.c:

#include<stdio.h>

#include"HuffMan.h"

int main (void)

{

HuffmanTree ht = NULL;

HuffmanCode hc = NULL;

int n = 0, choice;

char \* ch = NULL;

char str[100];

while((choice = Menu()) != 0)

{

switch(choice)

{

case 1:

Initialization(&ht, &hc, &n, &ch);

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*操作完毕\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

break;

case 2:

printf("请输入正文：\n");

getchar();

gets(str);

Encoding(&ht, &hc, &n, str, &ch);

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*操作完毕\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

break;

case 3:

if(ht == NULL)

{

printf("霍夫曼树为空，请先确保内存中有霍夫曼树\n");

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*操作完毕\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

continue;

}

printf("译码结果为：\n");

Decoding(ht, n, ch);

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*操作完毕\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

break;

case 4:

printf("代码文件中的代码为：\n");

Print();

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*操作完毕\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

break;

case 5:

if(ht == NULL)

{

printf("霍夫曼树为空，请先确保内存中有霍夫曼树\n");

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*操作完毕\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

continue;

}

printf("打印霍夫曼树：\n");

TreePrinting(ht, n);

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*操作完毕\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

break;

default:

printf("输入内容无效，请重新输入！！\n");

break;

}

printf("\n");

}

FreeAll(ht, hc, ch);

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*退出程序\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

return 0;

}

HuffMan.h:

#include<stdio.h>

#include<malloc.h>

#include<stdlib.h>

#include<string.h>

typedef struct HTNode{

unsigned int weight;

unsigned int parent, lchild, rchild;

}HTNode, \* HuffmanTree;

typedef char \*\* HuffmanCode;

void Select(HuffmanTree ht, int i, int \* section1, int \* section2)

{

int j, s1 = 0, s2 = 0;

for(j = 1; j <= i; j++)

{

if(ht[j].parent == 0)

{

if(!s1 || ht[s1].parent)

{

s1 = j;

}

else

{

if(ht[s1].weight > ht[j].weight)

s1 = j;

}

}

}

for(j = 1; j <= i; j++)

{

if(ht[j].parent == 0 && j != s1)

{

if(!s2 || ht[s2].parent)

{

s2 = j;

}

else

{

if(ht[s2].weight > ht[j].weight)

s2 = j;

}

}

}

if(s1 < s2)

{

\*section1 = s1;

\*section2 = s2;

}

else

{

\*section2 = s1;

\*section1 = s2;

}

}

void createHFMTreeFile(HuffmanTree ht, HuffmanCode hc, int n, char \* ch)

{

FILE \* fp = NULL;

int i;

if((fp = fopen("hfmTree.txt", "wt")) == NULL)

{

printf("Cannot open file strike any key exit!");

exit(1);

}

fprintf(fp, "%d\n", n);

for(i = 1; i <= 2\*n - 1; i++)

{

fprintf(fp, "%d %d %d %d\n", ht[i].weight, ht[i].parent, ht[i].lchild, ht[i].rchild);

fflush(fp);

}

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

{

fputc(ch[i], fp);

fputs(hc[i], fp);

fputc('\n', fp);

fflush(fp);

}

fclose(fp);

fp = NULL;

}

void Initialization(HuffmanTree \* ht, HuffmanCode \* hc, int \* N, char \*\* ch)

{

HuffmanTree HT;

HuffmanCode HC;

char \* CH;

int m, n, i, section1, section2;

printf("请输入字符集的大小:\n");

scanf("%d", &n);

int weig[n+1];

m = 2 \* n - 1;

HT = (HuffmanTree) malloc ((m+1) \* sizeof(HTNode));

CH = (char\*) malloc (n \* sizeof(char));

if(!HT || !CH)

{

printf("内存分配失败！！\n");

exit(-1);

}

printf("请输入字符和该字符的权重：\n");

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

{

getchar();

CH[i] = getchar();

scanf("%d", &weig[i]);

}

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

{

HT[i].weight = weig[i];

HT[i].parent = HT[i].lchild = HT[i].rchild = 0;

}

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

{

HT[i].weight = HT[i].parent = HT[i].lchild = HT[i].rchild = 0;

}

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

{

Select(HT, i-1, &section1, &section2);

HT[section1].parent = HT[section2].parent = i;

HT[i].lchild = section1;

HT[i].rchild = section2;

HT[i].weight = HT[section1].weight + HT[section2].weight;

}

char \* cd;

HC = (HuffmanCode) malloc ((n+1) \* sizeof(char \*));

cd = (char \*) malloc (n \* sizeof(char));

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

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

{

int start = n-1, child = i, parent;

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

{

if(child == HT[parent].lchild)

cd[--start] = '0';

else

cd[--start] = '1';

}

HC[i] = (char \*) malloc (n \* sizeof(char));

strcpy(HC[i], &cd[start]);

}

free(cd);

createHFMTreeFile(HT, HC, n, CH);

\*ht = HT;

\*ch = CH;

\*N = n;

\*hc = HC;

}

void Encoding(HuffmanTree \* ht, HuffmanCode \* hc, int \* n, char \* str, char \*\* ch)

{

FILE \* fp = NULL;

HuffmanTree HT = \*ht;

HuffmanCode HC = \*hc;

int N = \*n;

char \* CH = \*ch;

if(\*ht == NULL)

{

char code;

int i = 1;

if((fp = fopen("hfmTree.txt", "r")) == NULL)

{

printf("Cannot open file strike any key exit!");

exit(1);

}

fscanf(fp, "%d", &N);

HT = (HuffmanTree) malloc ((2\*N) \* sizeof(HTNode));

HC = (HuffmanCode) malloc ((N+1) \* sizeof(char \*));

CH = (char \*) malloc ((N+1) \* sizeof(char));

for(i = 1; i <= 2\*N - 1; i++)

{

fscanf(fp, "%d %d %d %d", &HT[i].weight, &HT[i].parent, &HT[i].lchild, &HT[i].rchild);

}

for(i = 1; i <= N; i++)

{

int j = 0;

while((code = fgetc(fp)) == '\n');

CH[i] = code;

HC[i] = (char \*) malloc (N \* sizeof(char));

while((code = fgetc(fp)) != '\n')

{

HC[i][j++] = code;

}

HC[i][j] = '\0';

}

fclose(fp);

fp = NULL;

\*ht = HT;

\*hc = HC;

\*n = N;

\*ch = CH;

}

int i = 0;

if((fp = fopen("CodeFile.txt", "wt")) == NULL)

{

printf("Cannot open file strike any key exit!");

exit(1);

}

while(str[i] != '\0')

{

int j;

for(j = 1; j <= N; j++)

{

if(str[i] == CH[j])

break;

}

fputs(HC[j], fp);

fputc('\n', fp);

fflush(fp);

i++;

}

fclose(fp);

fp = NULL;

}

void Decoding(HuffmanTree ht, int n, char \* ch)

{

int m = 2 \* n - 1;

char code;

FILE \* fp1 = NULL, \* fp2 = NULL;

if((fp1 = fopen("CodeFile.txt", "rt")) == NULL)

{

printf("Cannot open file strike any key exit!");

exit(1);

}

if((fp2 = fopen("TextFile.txt", "wt")) == NULL)

{

printf("Cannot open file strike any key exit!");

exit(1);

}

while(1)

{

if(ht[m].lchild || ht[m].rchild)

{

code = fgetc(fp1);

if(code == EOF)

break;

if(code == '0')

m = ht[m].lchild;

else if(code == '1')

m = ht[m].rchild;

}

else

{

printf("%c", ch[m]);

fprintf(fp2, "%c", ch[m]);

fflush(fp2);

m = 2 \* n -1;

}

}

printf("\n");

fclose(fp1);

fp1 = NULL;

fclose(fp2);

fp2 = NULL;

}

void Print()

{

int count = 0;

char code;

FILE \* fp1, \* fp2;

if((fp1 = fopen("CodeFile.txt", "rt")) == NULL)

{

printf("Cannot open file strike any key exit!");

exit(1);

}

if((fp2 = fopen("CodePrint.txt", "wt")) == NULL)

{

printf("Cannot open file strike any key exit!");

exit(1);

}

while((code = fgetc(fp1)) != EOF)

{

if(code == '\n')

continue;

putchar(code);

fprintf(fp2, "%c", code);

count++;

if(!(count % 50))

{

putchar('\n');

fprintf(fp2, "%c", '\n');

}

}

printf("\n");

fclose(fp1);

fp1 = NULL;

fclose(fp2);

fp2 = NULL;

}

void TreePrint(HuffmanTree ht, HuffmanTree start, int count, FILE \* fp)

{

int i;

if(start != ht)

{

TreePrint(ht, ht+start->rchild, count+1, fp);

for(i = 0; i < count; i++)

{

printf("\t");

fprintf(fp, "%c", '\t');

}

printf("%d\n", start->weight);

fprintf(fp, "%d\n", start->weight);

TreePrint(ht, ht+start->lchild, count+1, fp);

}

}

void TreePrinting(HuffmanTree ht, int n)

{

int m = 2 \* n - 1, count = 0;

FILE \* fp = NULL;

if((fp = fopen("TreePrint.txt", "wt")) == NULL)

{

printf("Cannot open file strike any key exit!");

exit(1);

}

TreePrint(ht, ht+m, count, fp);

fclose(fp);

fp = NULL;

}

int Menu()

{

int choice;

printf("有以下选择：\n");

printf("1.初始化\n");

printf("2.编码\n");

printf("3.译码\n");

printf("4.印代码文件\n");

printf("5.印哈夫曼树\n");

printf("0.退出程序\n");

printf("请输入您的选择：\n");

scanf("%d", &choice);

return choice;

}

void FreeAll(HuffmanTree ht, HuffmanCode hc, char \* ch)

{

free(ht);

free(hc);

free(ch);

}