数据结构与算法

实验报告（二）

实验题目：利用二叉树结构实现哈弗曼编/解码器

姓 名： 魏靖

学 号： 2021213513

日 期： 2022.05.13

自我评分： 【 A+ 】95

自我评分说明：A+，A，B+，B，B-，C，D，分别对应分数95、90、85、80、75、70、60

诚信声明

本人郑重承诺：本实验程序和实验报告均是本人独立学习和工作所获得的成果。尽我所知，实验报告中除特别标注的地方外，不包含其他同学已经发表或撰写过的成果；实验程序中对代码工作的任何帮助者所作的贡献均做了明确的说明，并表达了谢意。

如有抄袭，本人原因承担因此而造成的任何后果。

特此声明。

签名：魏靖 日期：2022.05.13

程序引用说明

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 引用项 | 来源 | 相同代码行数 |
| 1 | 查找函数 | 《书名》 |  |
| 2 | 构造有序链表函数 | 互联网网址 |  |
| 3 |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
| 小计 | | |  |

总代码行数\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_; 引用占比\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1、实验简介

通过具体代码实现简单的哈弗曼编码和解码，以此提高代码应用能力，同时加深对哈弗曼编码思维的理解。

1. 程序框架

（核心：哈弗曼树的实现，尤其是创建部分内存的读取引用等等）

●建树：

1. 找出序列中的两个最小的权重；（Point）
2. 使较小的一个是左节点，另一个是右节点，它们的父节点的权重是它的两个子节点的和。
3. 把这组树放在序列的后面。
4. 循环回（1），直到Huffman树被建立起来。

#include<iostream>

#include<cstring>

#include<stdio.h>

using namespace std;

struct HNode //哈夫曼结点

{

int weight;

int parent;

int LChild;

int RChild;

};

struct HCode //编码表

{

char data;

char code[100];

};

class Huffman //哈夫曼类

{

private:

HNode\* HTree;

HCode\* HCodetable;

char str[1024] = { 0 }; //原始字符串

char leaf[256] = { 0 }; //叶子结点对应字符

int amount[256] = { 0 }; //字符出现次数

public:

int n=0; //叶子结点数

void putin(); //写入原字符串并记录出现过的字符

void getmin(HNode, int, int &, int &); //找权值

void createHTree(); // 创建哈夫曼树（重难点）

void Init();

void CreateTable();

void Encoding();

void Decoding();

void printHTree(int i, int j);

~Huffman();

};

1. **关键代码实现**

**3.1权值记录**

void Huffman::putin()

{

int a[256] = { 0 };

int x = cin.get(); //键盘输入

int i = 0;

while ((x != '\n') && (x != '\r'))

//实现对空格输入（输入换行或回车则结束str的录入）

{

a[x]++;

str[i++] = x; //写入str

x = cin.get();

}

str[i] = '\0';

n = 0;

for (i = 0; i < 256; i++)

{

if (a[i] > 0) //判断字符是否出现，出现则录入（=0说明未出现）

{

leaf[n] = (char)i;

amount[n] = a[i];

n++;

}

}

}

**3.2最小权值查找**

具体思路：

建立哈弗曼树要求我们每次都选权值最小的点构成节点，那么：

首先找一个比较值的起始值（设从0开始），当这个节点没有父节点的时候，把这个节点作为最小权值的参考点（lf），j自增，同理碰上没有父节点的节点，把这个节点作为次小权值的参考点（ls），比较HTree【lf】和HTree【ls】的weight，谁小谁就是lf（即是说如果ls<lf，那么两者互换）。j再自增，当j无父节点（未被接入Huffman树中），比较j、lf，若j<lf,则记为lf，lf记为ls;若j>lf&&<ls，则j记为ls……直到遍历所有节点得到两个最小值。

所以应该注意：到底应该遍历多少个节点,每个都便利的过程中会不会出现问题，所以我用到了循环加break，保证每个都能遍历到：

void Huffman::getmin(HNode\*Htree, int n, int &lf, int&ls)

{

int j;

//找一个比较值的起始值

for (j = 0; j < 2\*n-1; j++) //找lf

{

if (Htree[j].parent == -1)

{

lf = j; break;

}

}

j++;

for (; j < 2\*n-1; j++) //找ls

{

if (Htree[j].parent == -1)

{

ls = j; break;

}

}

if (Htree[lf].weight > Htree[ls].weight) //lf指向最小的

{

int x = ls; ls = lf; lf = x;

}

j++;

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

{

if (Htree[j].parent == -1)

{

if (Htree[j].weight < Htree[lf].weight)

{

ls =lf; lf = j;

}

else if (Htree[j].weight < Htree[ls].weight)

{

ls = j;

}

}

}

}

但在create处有个疑问是，如果我查到的lf、ls都不是当前i值，那么当前的i对应节点权值就会被修改和遗忘，但是编码好像并没有出错，引人深思。

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

{

HTree[i].weight = amount[i];

HTree[i].LChild = -1;

HTree[i].RChild = -1;

HTree[i].parent = -1;

}

int lf, ls; //最小两权重的下标

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

{

getmin(HTree,n,lf, ls);

HTree[lf].parent = HTree[ls].parent = i;

HTree[i].weight= HTree[lf].weight +HTree[ls].weight ;

//我这里修改了HTree【i】的权重，会不会导致我原本权重是x，但是在后面他就被忽略了？

HTree[i].LChild = lf;

HTree[i].RChild = ls;

HTree[i].parent = -1;

}

改进如方法尚未找到，所以不敢妄言。

**3.3哈夫曼树打印**

因为要体现树的结构，我想用一种直观的绘图方式绘出Huffman树，但是发现具体代码未曾接触过，所以对此展开了一个小研究。

因为关注于核心代码实现，在这方面并没有形成实际应用，但有以下收获：

1. 与我思路相似，按打印\*图案一样，按矩阵的形式打出value和/和\（eg. 12

/ \

3 4

/ \ \

1 2 5）

1. 打印的基础是遍历整个线性表：前序/中序。因为我们的HNode里面记录得清楚他的parent、LCh和RCh，所以执行起来会相对来说简单一些（有的时候parent和child到底取哪一个比较难以分得哪一个代码最简）

**疑问：有关函数地址调用/值调用的内容**

值得注意的是，这部分是上一学期C++编程语言和思维学习过程中的一个重点，那在下面这段代码中也出了这样的错误。

HNode\* Huffman::createHTree()

{

HNode\* HTree = new HNode[2 \* n - 1];

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

{

HTree[i].weight = amount[i];

HTree[i].LChild = -1;

HTree[i].RChild = -1;

HTree[i].parent = -1;

}

int lf, ls; //最小两权重的下标

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

{

getmin(HTree,n,lf, ls);

HTree[lf].parent = HTree[ls].parent = i;

HTree[i].weight= HTree[lf].weight +HTree[ls].weight ;

HTree[i].LChild = lf;

HTree[i].RChild = ls;

HTree[i].parent = -1;

}

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

{

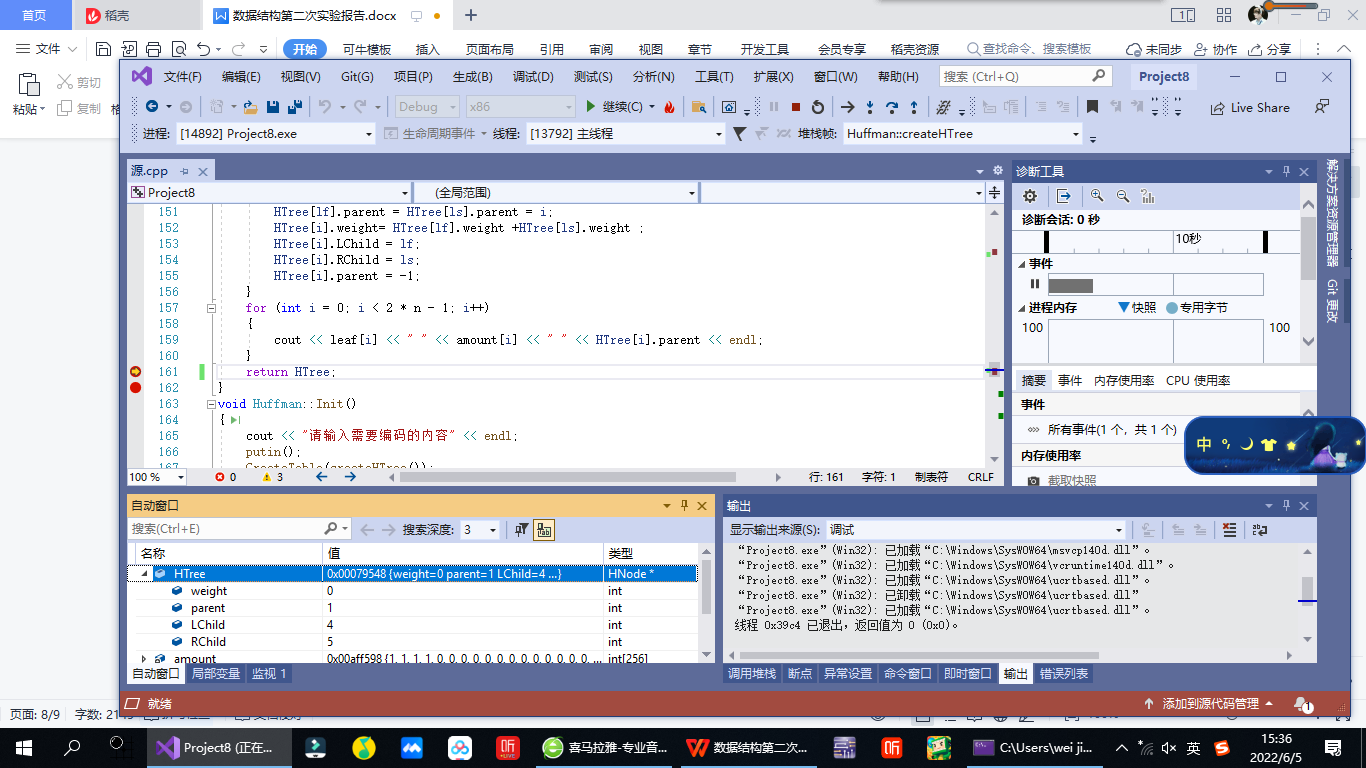
cout << leaf[i] << " " << amount[i] << " " << HTree[i].parent << endl;

}

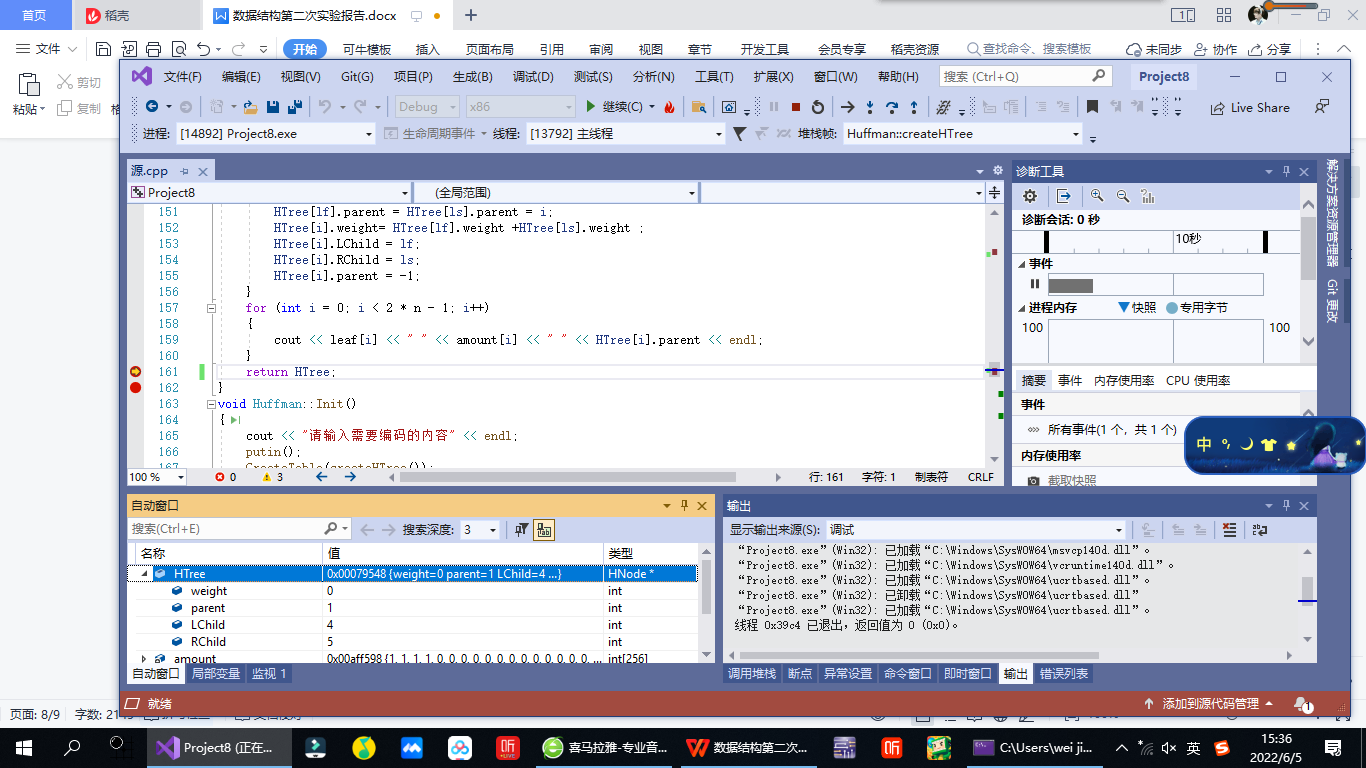
return HTree;

}

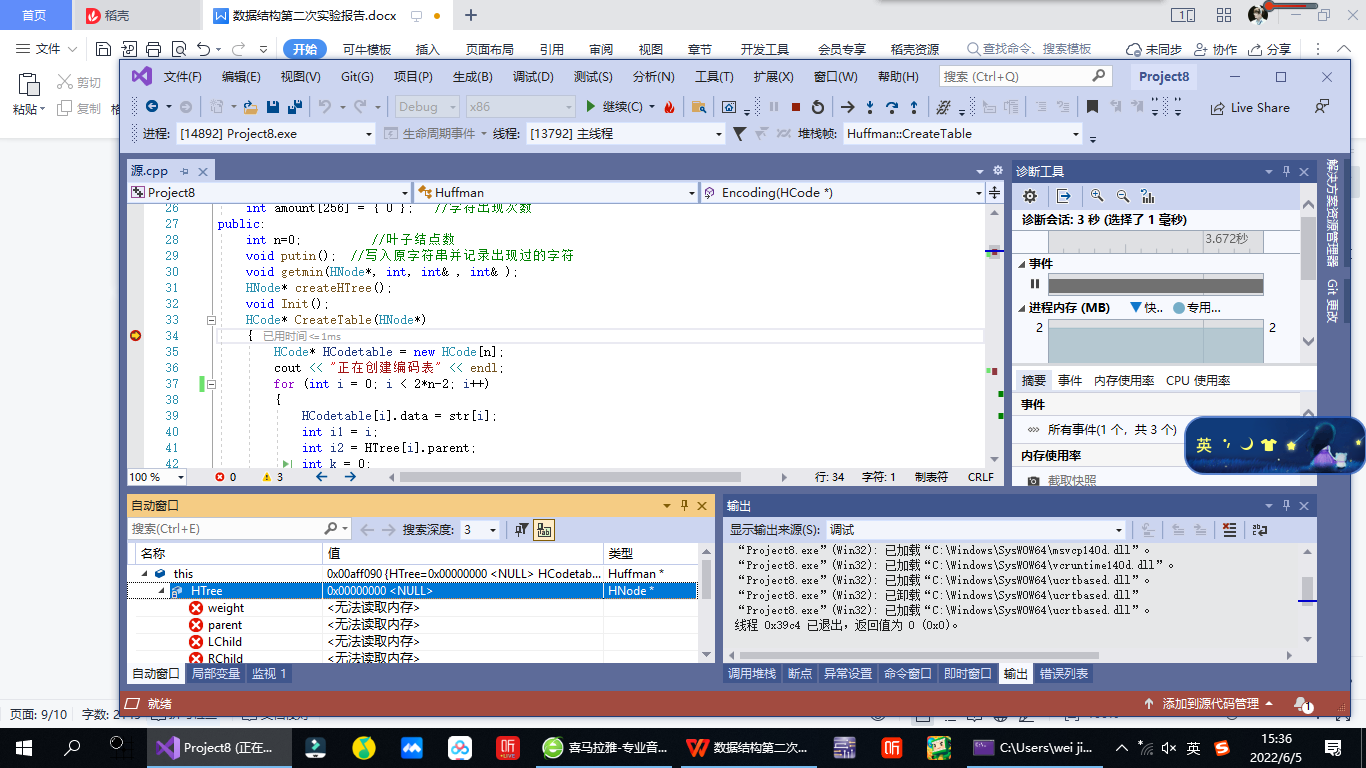
（即使这里return了HTree，但我们看到，假设输入“love”，在函数调用完毕之后，HTree还是恢复到了“NULL”）



（这是执行return之前）



（这是执行return之后，HTree还有取值，在函数内部没有问题）



（这是跳出了函数createHTree之后，HTree的内存直接无法访问，即使定义在类的公有成员函数也无法读取到HTree的值）

改进：用指向HNode/HCode的指针记录地址，改\*为\*&

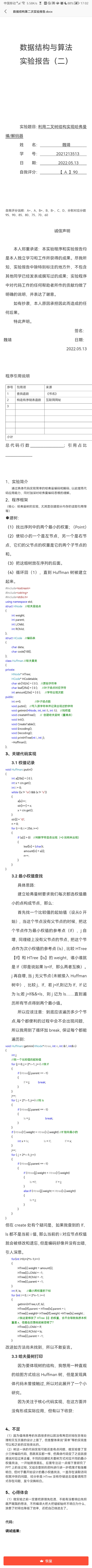
1. **不足**

（1）因为值得思考的东西很多所以就没有再花时间在异常处理和交互页面的设计上面了，但是整体来说“菜单”等的实现是可以和之前的实验类比的。

（2）树这一块的代码实现可能还是有点问题，做实验查了至少三种编码代码，思路其实都一样，但具体代码变了之后就很难说对应过来去看，不同的创建哈夫曼树方式对应不同的最小权值找法，一开始就很混乱，后面专注这一点细下里研究了PPT上的全过程，又结合查到的资料进行进一步梳理才勉强看明白，但对于最开始设计的最小权值找法，一直存在读取访问权限冲突的问题，经分析是HTree没有存储进去或者调用方式存在问题，至今没搞明白。

1. **心得体会**
2. 哈弗曼编码的压缩率十分震撼。首先他不会因为不等长而出现二义性（当然这取决于本身我们不允许他有歧义，但是这个原理还是很棒），其次就是在编码过程中，weight越大越靠近根节点，所需比特越少，在不知不觉中就节省了空间（不需要去算占空间量，就比如说“a”一篇文章中使用1000次（最频繁），编码如果原本4位一个字母的话，那么哈夫曼编码实现了对’a’的1位编码，整个最大一部分内存占比就从原来的4000bit变成了1000比特。当然这只是形象的一个比喻，实际运用中对空间的压缩情况还是要靠具体式子计算）。
3. 做实验之前一定要把原理先吃透，不能有没看明白先照葫芦画瓢的想法，不然编译大把大把错却始终不明白为什么，浪费了时间也降低了效率，还把自己绕进去了。
4. 先写伪代码和思路，便于理清楚哪一步做什么，对分析内存占用和读取也会更加清晰。
5. **实验轶事**

在写完读书报告保存的时候自信地把源代码删了，第二天准备改进一下上交，却发现文件损坏无法修复，最多能找到的只有这么一张图片，还被迫斥巨资买会员想着会员应该就能修复，钱给了发现依旧不能。



所以这篇报告就参考了历史版本进行改进，重新写代码，重新调试……一切重头再来。就很感谢自己的。

其次就是我去了解了一下有关文档修复的问题。它既然会乱码，也必然有一方面问题出在数据上，会不会跟我们的数据结构有关，为此我展开了思考和调查。

在没有更改格式、正常保存、没有出现全盘文件乱码的情况下，加之电脑D盘较满的实际情况，我想可能有以下原因：

1. 磁盘内存碎片化，碎片化部分被清理。

我们在课程第一章时曾经思考过一个动态内存管理问题，讲到的三种优先方式各有特点，但归于一我觉得可以这么理解就是在点了保存之后，成功存入寄存器，但在寄存器存到硬盘的时候发生溢出或者硬盘分区结构变化，那么能够保存的则是上一次编辑的已存储部分，这一问题一般可以通过远程联机或者一些解码软件进行处理。（不过对于严重损坏的文件修复成功的几率另言之，所以我们时刻存档到云端就是很有必要的）

1. “头文件”缺失。（了解到的严重损坏的原因之一）像我们写C++代码，如果在调用某些函数前没有include相应的头文件，则不可能运行成功（除非在源代码里重复一遍相关内容）。

**辅助分析：**

借助winhex软件查看文件头是否出现问题或缺失（如果没有就直接否定了第二种可能性了）。注意：扇区分配表不可能为零。

**教训：**

数据恢复是很“伤心”和“伤钱”，所以平时要做好磁盘保护内存整理，重要资料及时转存避免因硬件或软件问题导致的重要数据丢失（更严重是被盗用），其次便是学以致用，用所学知识结合分析，冷静应对，找到突破口。

**代码：**

#include <iostream>

#include <cstring>

#include<stdio.h>

#include <algorithm>

using namespace std;

using std::string;

struct HNode

{

int weight; // 权值

int parent, LChild, RChild; // 双亲及左右孩子的下标

};

struct HCode

{

char data;

char code[100];

};

class Huffman {

private:

HNode\* HTree;

HCode\* HCodetable;

char str[1024] = { 0 };

char leaf[256] = { 0 };

int amount[256] = { 0 };

public:

int n = 0;

void putin();

void getmin(HNode\*, int, int&, int&);

void createHTree(HNode\*&);

void Init();

void CreateTable(HNode\*,HCode\*&);

void Encoding();

void Decoding();

~Huffman();

};

void Huffman::putin()

{

int a[256] = { 0 };

int x = cin.get();

int i = 0;

while ((x != '\n') && (x != '\r'))

{

a[x]++;

str[i++] = x;

x = cin.get();

}

str[i] = '\0';

n = 0;

for (i = 0; i < 256; i++)

{

if (a[i] > 0)

{

leaf[n] = (char)i;

amount[n] = a[i];

n++;

}

}

}

void Huffman::getmin(HNode \*HTree, int n, int& lf, int& ls)

{

int j;

for (j= 0; j < 2\*n-1; j++)

{

if (HTree[j].parent==-1)

{

lf = j;

break;

}

}

j++;

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

{

if (HTree[j].parent == -1)

{

ls = j;

break;

}

}

if (HTree[lf].weight > HTree[ls].weight) {

int x = ls;

ls = lf;

lf = x;

}

j++;

for (; j < 2\*n-1; j++) {

if (HTree[j].parent==-1)

{

if (HTree[j].weight < HTree[lf].weight)

{

ls = lf;

lf = j;

}

else if (HTree[j].weight < HTree[ls].weight) {

ls = j;

}

}

}

}

void Huffman::createHTree(HNode\*& HTree)

{

HTree= new HNode[2\*n-1];

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

{

HTree[i].weight = amount[i];

HTree[i].parent =HTree[i].LChild = HTree[i].RChild = -1;

}

int lf, ls;

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

{

getmin(HTree, n, lf, ls);

HTree[lf].parent = HTree[ls].parent = i;

HTree[i].LChild = lf;

HTree[i].RChild = ls;

HTree[i].parent = -1;

// 作为新节点的孩子

HTree[i].weight = HTree[lf].weight + HTree[ls].weight; // 新节点为左右孩子节点权值之和

}

}

void Huffman::CreateTable( HNode\*HTree,HCode\*&HCodetable)

{

cout << "正在创建编码表" << endl;

HCodetable = new HCode[2 \* n - 1];

char\* s = new char[1000];

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

{

HCodetable[i].data = str[i];

int i1 = i;

int i2 = HTree[i].parent;

int k = 0;

while (i2 != -1)

{

if (i1 == HTree[i2].LChild)

HCodetable[i].code[k] = '0';

else

HCodetable[i].code[k] = '1';

k++;

i1 = i2;

i2 = HTree[i1].parent;

}

HCodetable[i].code[k] = '\0';

char tmp = '\0';

for (int j = strlen(HCodetable[i].code) - 1; j >= 0; j--)

{

tmp = HCodetable[i].code[j];

}

int m = 0;

s[m] = tmp;

m++;

}

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

{

for (int j = 0; j < strlen(HCodetable[i].code); j++)

{

HCodetable[i].code[j]= s[j];

}

delete[]s;

}

}

void Huffman::Init()

{

cout << "请输入需要编码的内容" << endl;

putin();

createHTree(HTree);

CreateTable(HTree,HCodetable);

}

void Huffman::Encoding()

{

cout << "正在编码" << endl;

char\* s = str;

char\* am = new char[1000];

while (\*s != '\0')

{

for (int i = 0; i < strlen(str); i++)

if (\*s == HCodetable[i].data)

{

for (int j = 0; j < strlen(str); j++)

am[j] = HCodetable[i].code[j];

break;

}

s++;

}

for (int i = 0; i < strlen(str); i++)

str[i] = am[i];

cout << "编码结果：" << endl << str << endl;

delete[]am;

}

void Huffman::Decoding()

{

cout << "正在解码" << endl;

char\* s = str;

char\* d = new char[1000];

while (\*s != '\0')

{

int k = 2 \* n - 2;

while (HTree[k].LChild != -1)

{

if (\*s == '0')

k = HTree[k].LChild;

else

k = HTree[k].RChild;

s++;

}

\*d = HCodetable[k].data;

d++;

}

cout << \*d;

delete[]d;

}

Huffman::~Huffman()

{

delete[]HTree;

delete[]HCodetable;

}

int main()

{

Huffman huffman;

huffman.Init();

huffman.Encoding();

cout << "是否查看解码结果? 1 yes,0 no" << endl;

int t = 0;

if (t)

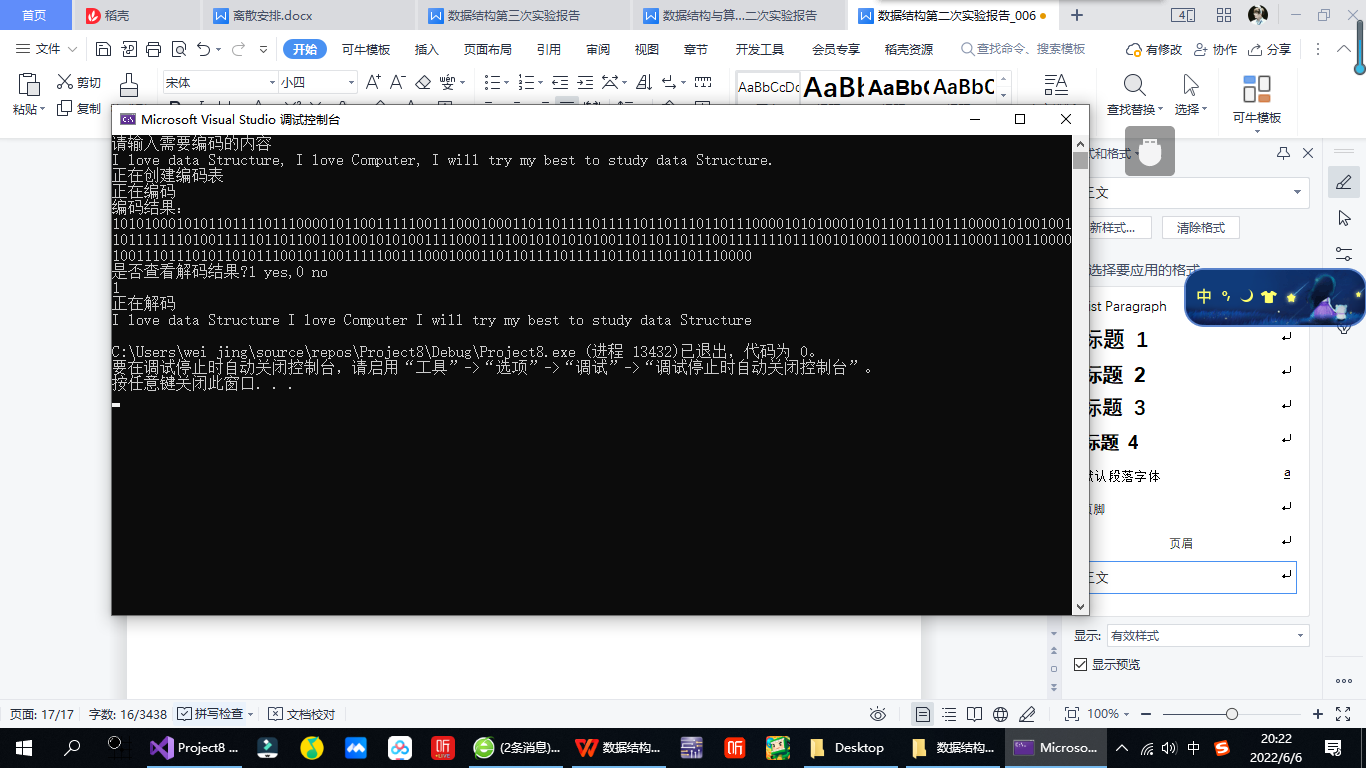
huffman.Decoding();

else;

return 0;

}

**调试结果：**



I 10101

（空格） 00

l 0101

o 0110

v 111101

e 1100

d 10110

a 0111

t 110

S 01000

r 1101

u 1110

c 111110

C 1010010

m 111111

p 1010011

w 1111000

i 1111001

y 10111

b 101000

s 01001

符合预期。