|  |
| --- |
| 中南大学  《数据结构》课程实验  实验报告  实验题目 Huffman 编码  专业班级 软件工程2005班  学 号 8209200504  姓 名 李均浩  **实验成绩：**  **批阅教师：**  2021年5月5日 |

一、需求分析

1.程序任务

1、根据所提供的字母数据建立一个Huffman树；

2、根据生成的Huffman树的结构，显示输出所有字母的Huffman编码。

3、根据产生的Huffman编码，实现Huffman编/译码器。

2.输入以及输出的形式



图1 程序输入输出形式

3.程序功能

对各个规定的字符进行Huffman编码，对英文原文进行编码或者对Huffman编码进行解码。实现Huffman编码的编码器以及解码器。

4.测试数据

(a) data structure is fantastic

预期输出：010010101001001010001100001011011111011111001011110110110100010011100000001100010110000010010111000010100111111

(b) I LOVE Programing

预期输出：00010010000100001100011111101000011101110101100111101101010100110110011000011110

(c) CSU is Number ONE

预期输出：1111111001111000010011100000100011110001101011111101110100001101000101

(d) 10010000100001100011111101000011100011011110

预期输出：i love you

(e) 1001000100010100111111011101000011000111010000000101000011110100111010100000000110011011001101100001001000011111111100010000100100011111010110000001000101001110000010110000000110011111001111011110111111001000001111101110000011111011101011101

预期输出：i never own a girl friend but i can new an object by cpp

(f) (错误输入) 45456

预期输出：未知编码阻止了解码程序的运行！（随后退出程序）

(g) (错误输入) 01011011101010101011

预期输出：未知编码阻止了解码程序的运行！（随后退出程序）

二、概要设计

1.抽象数据类型定义：

ADT Huffmantree

{

数据对象：D={ai| ai ∈Charset,i=1,2,3,„„n,n≥0}

数据关系：R1＝{< ai-1, ai >| ai-1, ai ∈D, i=2,3,„„n}

基本操作：

Initialization(&HT，&HC,w，n,ch)

操作结果：根据n个字符及其它们的权值w[i],建立Huffman树HT,用字符数组ch[i]作为中间存储变量，最后字符编码存到HC中；

Encodeing(n)

操作结果：根据建好的Huffman树，对文件进行编码，编码结果存入到文件CodeFile中；

Decodeing(HT,n)

操作结果：根据已经编译好的包含n个字符的Huffman树HT，将文件的代码进行翻译，结果存入文件TextFile中。

} ADT Huffmantree

2.主程序的流程



图2 主程序的流程

三、详细设计

1.模块伪码

(1) void GetAlphabetFreq(TreeNode\* node\_array) 开始

输出<< "\*\*\*\*\*\*\*\*\*\*获取字母频度表\*\*\*\*\*\*\*\*\*\*" << endl;

auto sp = (TreeNode)malloc(sizeof(HuffmanTreeNode));

如果(sp == 空指针)

退出程序(OVERFLOW);

sp->letter = ' ';

输出<< "请输入\' 空格 \'的权值：";

输入>> sp->weight;

sp->isDeleted = false;

sp->left = nullptr;

sp->right = nullptr;

sp->pre\_order\_counted\_times = 0;

node\_array[0] = sp;

for (int i = 0; i < 26; ++i) {

auto tn = (TreeNode)malloc(sizeof(HuffmanTreeNode));

if (tn == nullptr)

exit(OVERFLOW);

tn->letter = (char)('a' + i);

输出 << "请输入\' " << tn->letter << " \'的权值：";

输入 >> tn->weight;

tn->isDeleted = false;

tn->left = nullptr;

tn->right = nullptr;

tn->pre\_order\_counted\_times = 0;

node\_array[i + 1] = tn;

}

输出 << endl << "\*\*\*\*\*\*\*\*\*\*获取结束\*\*\*\*\*\*\*\*\*\*" << endl;

array\_length = 27;

结束

(2) TreeNode GetMinWeight(TreeNode\* node\_array) 开始

int min\_index = 10000;

int min\_weight = 10000;

for (int i = 0; i < array\_length; ++i) {

如果 (node\_array[i]->weight < min\_weight && !node\_array[i]->isDeleted) {

min\_weight = node\_array[i]->weight;

min\_index = i;

}

}

node\_array[min\_index]->isDeleted = true;

返回 node\_array[min\_index];

结束

(3) bool isEmpty(TreeNode\* node\_array) 开始

for (int i = 0; i < array\_length; ++i)

如果 (!node\_array[i]->isDeleted)

返回 false;

返回 true;

结束

(4) void SaveNodeToArray(TreeNode new\_node, TreeNode\* node\_array) 开始

node\_array[array\_length] = new\_node;

array\_length++;

结束

(5) void CreateHuffmanTree(HuffmanTree& t, TreeNode\* node\_array) 开始

TreeNode parent\_node;

(无限循环) {

TreeNode node\_1, node\_2;

node\_1 = GetMinWeight(node\_array);

node\_2 = GetMinWeight(node\_array);

parent\_node = 分配类型为(TreeNode)大小为(sizeof(HuffmanTreeNode))的内存空间;

如果 (parent\_node == 空指针)

退出程序(OVERFLOW);

parent\_node->isDeleted = false;

parent\_node->left = node\_1;

parent\_node->right = node\_2;

parent\_node->letter = '#';

parent\_node->weight = node\_1->weight + node\_2->weight;

parent\_node->pre\_order\_counted\_times = 0;

如果 (isEmpty(node\_array))

退出循环;

SaveNodeToArray(parent\_node, node\_array);

}

t.root = parent\_node;

结束

(6) void HuffmanCodeGenerator(HuffmanTree t, string\* huffman\_code\_list) 开始

string s;

s = "";

TreeNode n = t.root;

Stack node\_stack;

初始化栈(node\_stack);

无限循环 {

如果 (n == t.root) {

如果 (n->pre\_order\_counted\_times == 0) {

n->pre\_order\_counted\_times++;

Push(node\_stack, n);

n = n->left;

s += '1';

}

如果 (n->pre\_order\_counted\_times == 1) {

n->pre\_order\_counted\_times++;

入栈(node\_stack, n);

n = n->right;

s += '0';

}

如果 (n->pre\_order\_counted\_times == 2)

break;

}

否则 {

如果 (n->letter == '#') {

如果 (n->pre\_order\_counted\_times == 0) {

n->pre\_order\_counted\_times++;

入栈 (node\_stack, n);

n = n->left;

s += '1';

}

如果 (n->pre\_order\_counted\_times == 1) {

n->pre\_order\_counted\_times++;

入栈 (node\_stack, n);

n = n->right;

s += '0';

}

如果 (n->pre\_order\_counted\_times == 2) {

出栈 (node\_stack, n);

s = s.substr(0, s.length() - 1);

}

}

否则 {

std::cout << '\'' << n->letter << '\'' << "的权值为：" << n->weight << ",Huffman编码为：" << s << endl;

//n->huffman\_code = (string\*)malloc(100);

//\*(n->huffman\_code) = s;

如果 (n->letter != ' ')

huffman\_code\_list[n->letter - 'a' + 1] = s;

如果 (n->letter == ' ')

huffman\_code\_list[0] = s;

否则 (node\_stack, n);

s = s.substr(0, s.length() - 1);

}

}

}

结束

(7) string SearchHuffmanCode(char c, string\* huffman\_code\_list) 开始

返回 huffman\_code\_list[c - 'a' + 1];

结束

(8) string HuffmanEncoder(string plaintext, string\* huffman\_code\_list) 开始

将plaintext全部转为小写;

string ciphertext;

for (char i : plaintext) {

如果 (i == ' ')

ciphertext += huffman\_code\_list[0];

否则

ciphertext += SearchHuffmanCode(i, huffman\_code\_list);

}

返回 ciphertext;

结束

(9) string HuffmanDecoder(const string& ciphertext, string\* huffman\_code\_list) {

string plaintext;

int length = 1; //10010000100001100011111101000011100011011110

int start\_index = 0;

bool isMatch = false;

无限循环 {

如果 (start\_index + length > ciphertext.length())

返回 plaintext;

string temp = ciphertext.substr(start\_index, length);

for (int i = 0; i < 27; ++i) {

如果 (temp == huffman\_code\_list[i]) {

isMatch = true;

如果 (i == 0)

plaintext += ' ';

否则

plaintext += 转为(char)(i + 'a' - 1);

}

}

如果 (isMatch) {

start\_index += length;

length = 1;

isMatch = false;

}

否则{

如果 (start\_index + length >= ciphertext.length()) {

输出 << "未知编码阻止了解码程序的运行！" << endl;

退出程序(错误代码：INVALID\_INPUT);

}

length++;

}

}

结束

(10)string GetBinCode(const string& s) 开始

string bin\_code;

输出 << "\*\*\*\*\*\*\*\*\*\*原文的二进制编码\*\*\*\*\*\*\*\*\*\*" << endl;

for (char i : s)

{

char temp[100];

将第s的第i位转为二进制编码存入temp(i, temp, 2);

输出 << i << '\t' << temp << endl;

bin\_code += temp;

}

返回 bin\_code;

结束

2.函数调用关系图



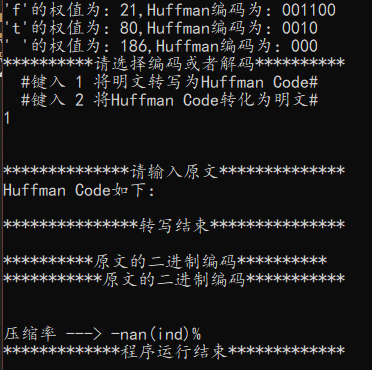
图3 函数调用关系图

四、调试分析

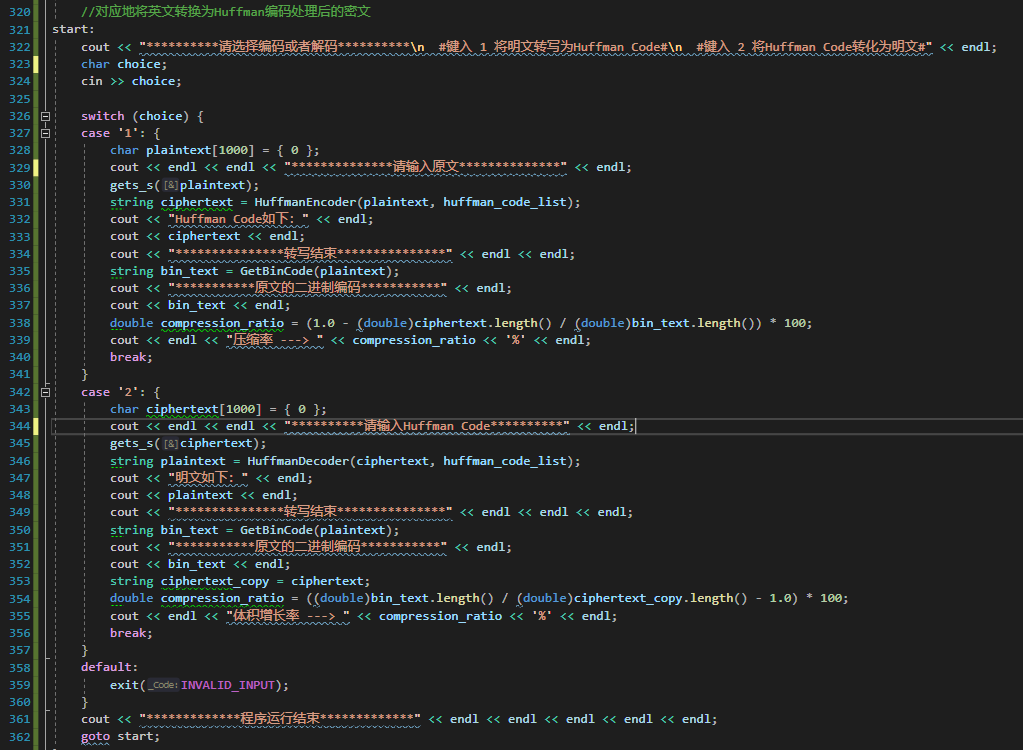
1.问题复现

(1) 输入字符选择程序功能之后无法正常输入字符串

(a)错误信息



(b)错误源码

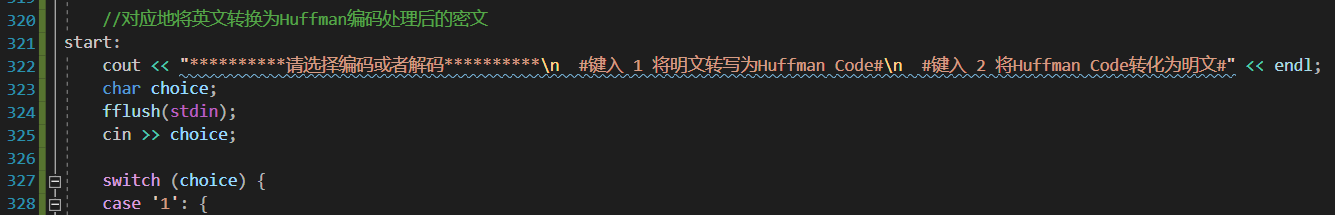


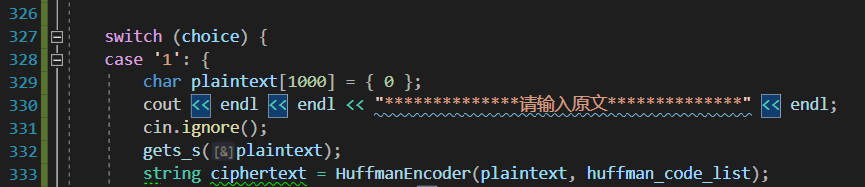
(c)错误解释

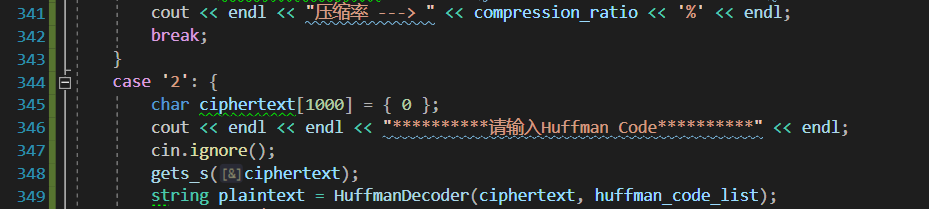
在执行cin >> choice之后，缓冲区中存留了一个’\n’字符，导致程序运行至case '1'或case '2'时首先读入了’\n’导致了输入的结束，即无法正常输入字符串至编码/解码模块中。

(d)解决方案

增加三处对缓存区的处理语句，如下图所示：







2. 算法的时空分析

(1)改进设想

部分判断条件分类可以合并，以减少操作的繁琐。

程序编写中有部分变量可以通过一定方式省去，能节省运行占用的空间。

3. 经验与体会

使用Huffman树生成的Huffman编码在实际应用非常广泛而且意义重大，Huffman编码使用变长编码表对字符进行编码，它通过评估来源符号出现机率的方法得到，出现机率高的字母使用较短的编码，反之出现机率低的则使用较长的编码，这便使编码之后的字符串的平均长度、期望值降低，从而达到无损压缩数据的目的。在我编写程序的过程中能明显感觉到这一点，压缩率一般能到30%左右，这样的话就能更大程度地使用存储资源。

五、用户使用说明

1. 根据提示输入从空格到’a’到’z’的频率。

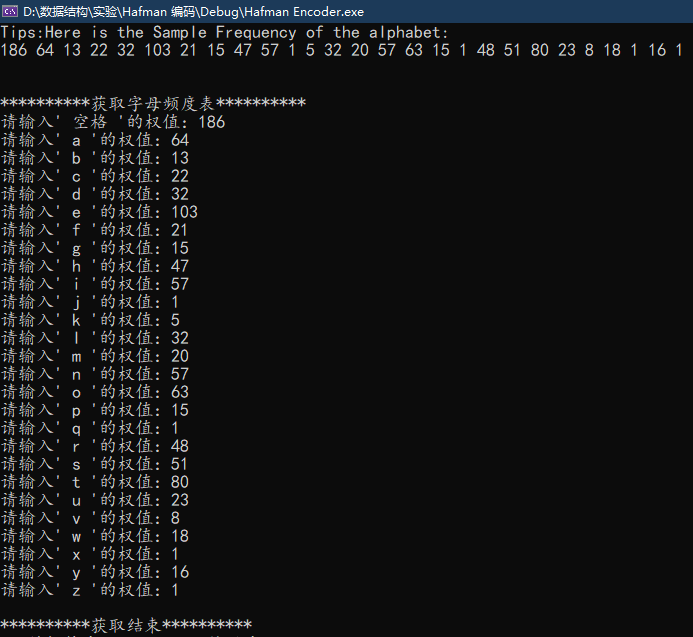


图4.1 操作演示1

2. 输出各个字符的对应Huffman编码，显示给用户。

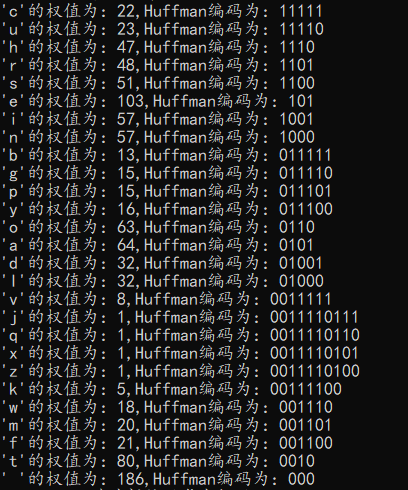


图4.2 操作演示2

3. 根据提示，选择进行编码或者解码。

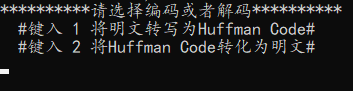


图4.3 操作演示3

4. 若选择了编码，则输入一段文字（允许大小写英文字母以及空格），随后输出Huffman编码串以及压缩率。

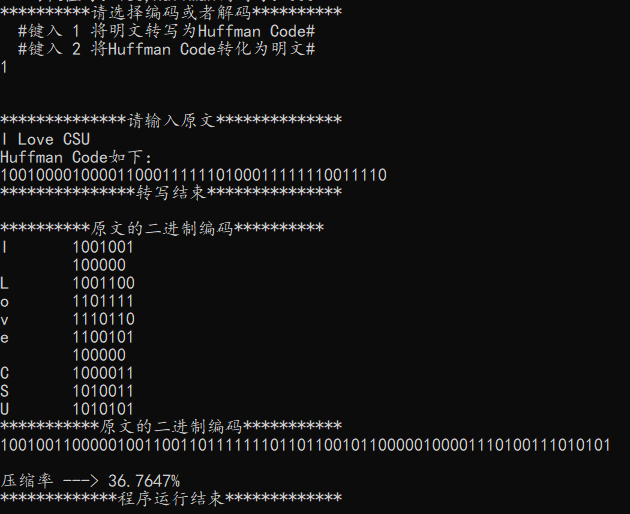


图4.4 操作演示4

5. 若选择了解码，则输入一段Huffman编码串，随后输出Huffman编码串对应的原文以及以及解压后的体积增加率。

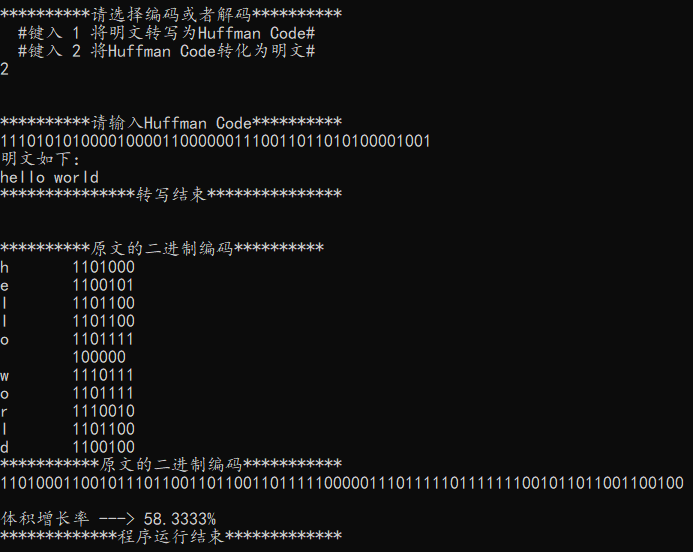
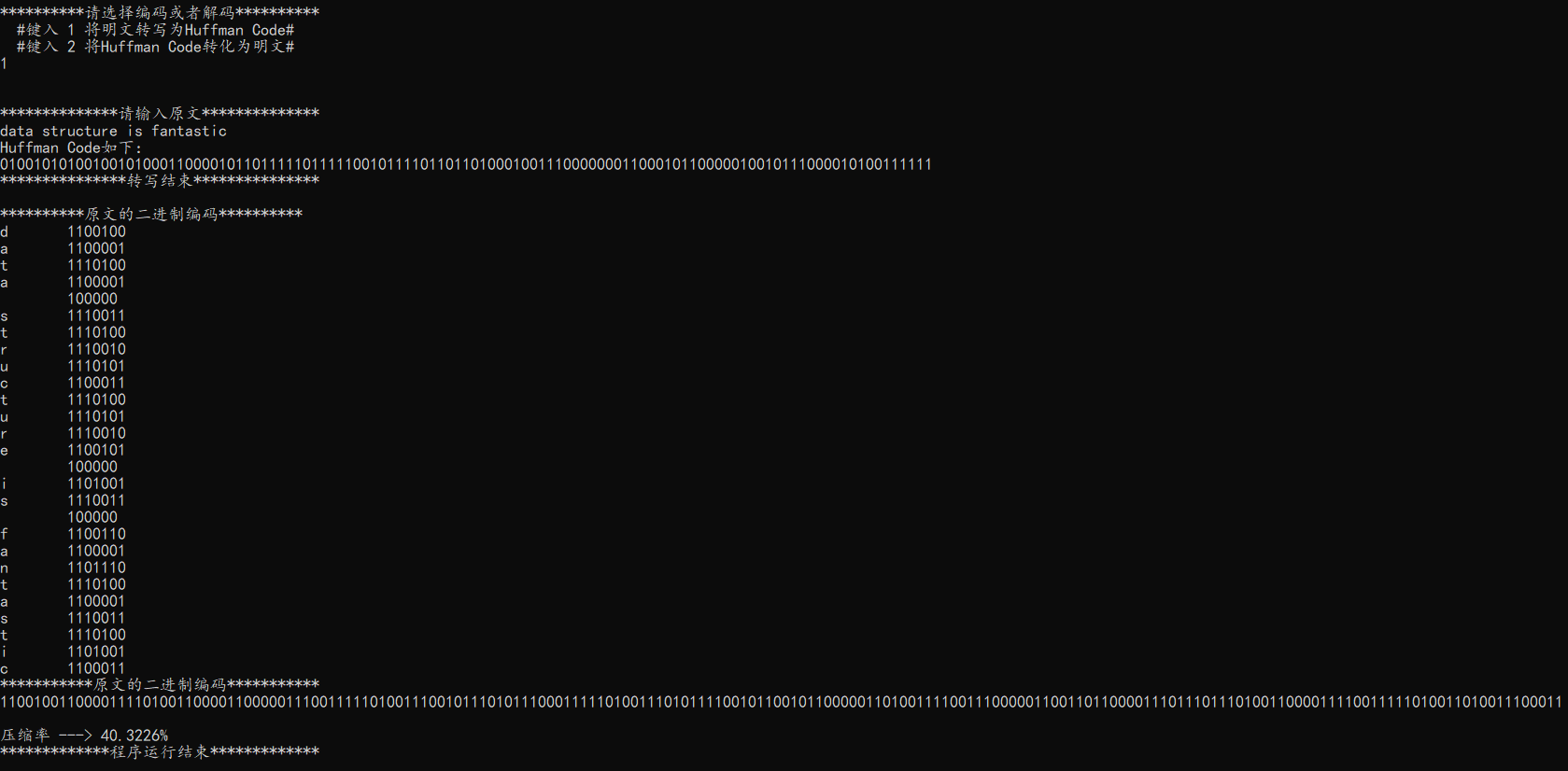


图4.5 操作演示5

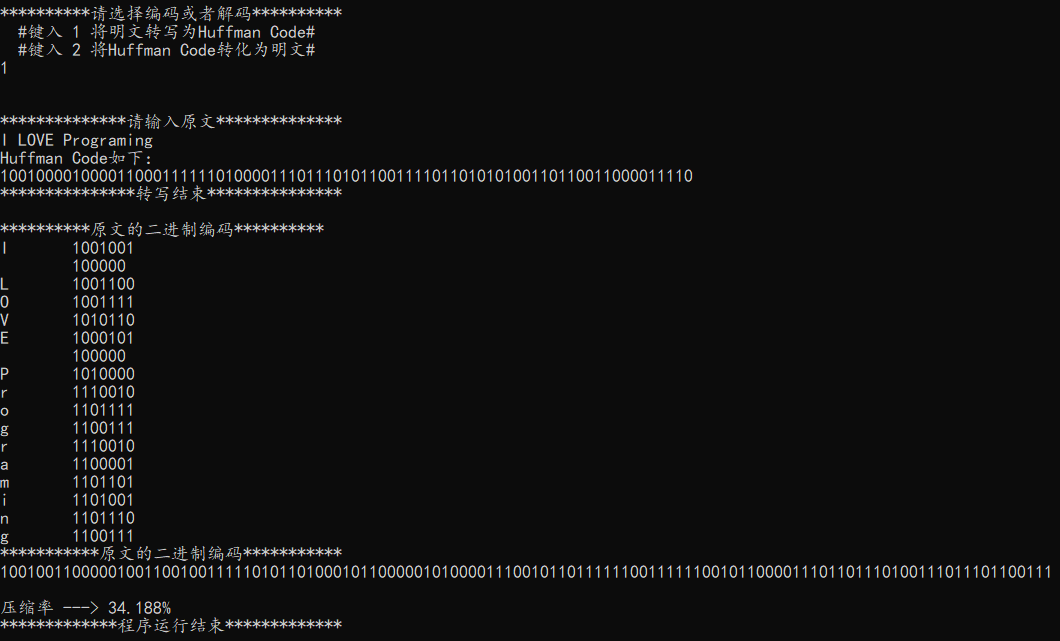
6. 随后程序会跳转至第3步，供用户使用同一套字符频率产生的Huffman编码，进行重复编码/转码。

六、测试结果

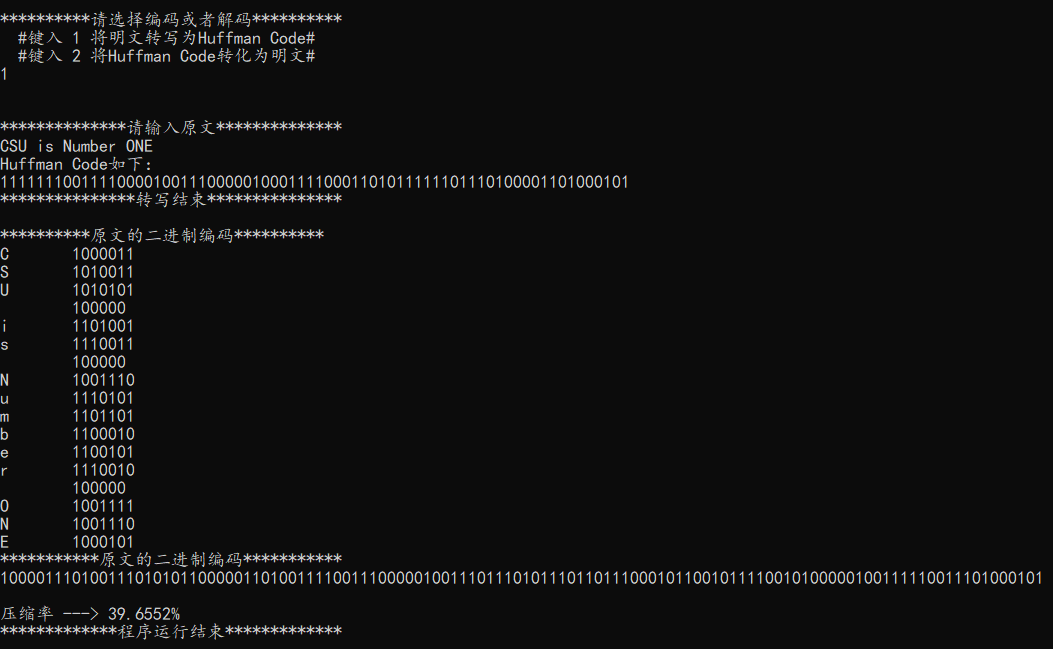
(1)输入：（选择编码功能）data structure is fantastic

输出：

(2)输入：（选择编码功能）I LOVE Programing

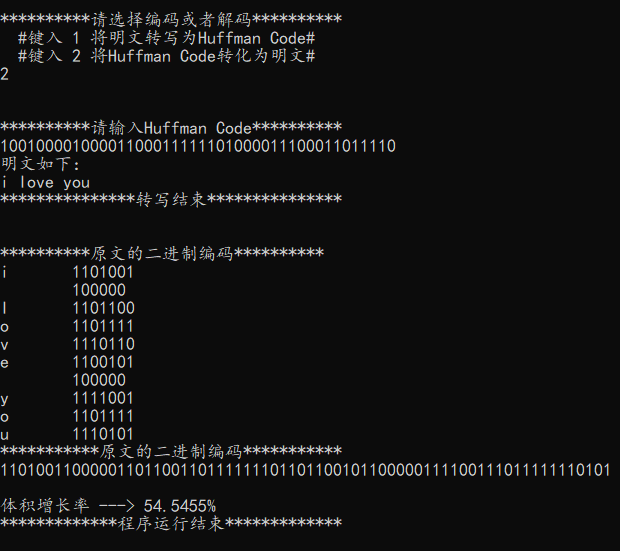
输出：

(3)输入：（选择编码功能）CSU is Number ONE

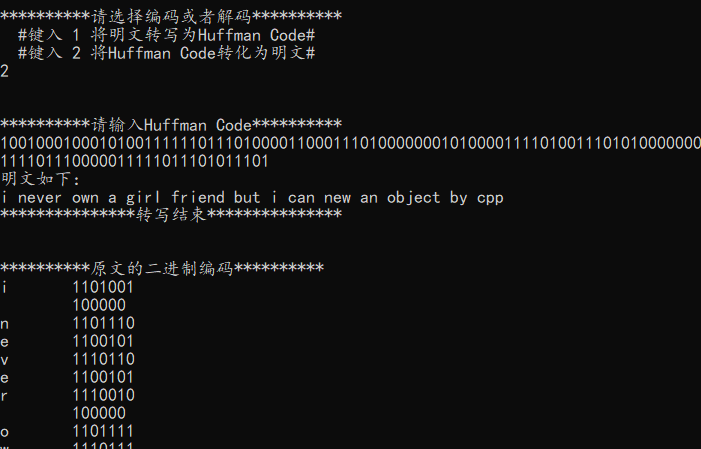
输出：

(4)输入：（选择解码功能）10010000100001100011111101000011100011011110

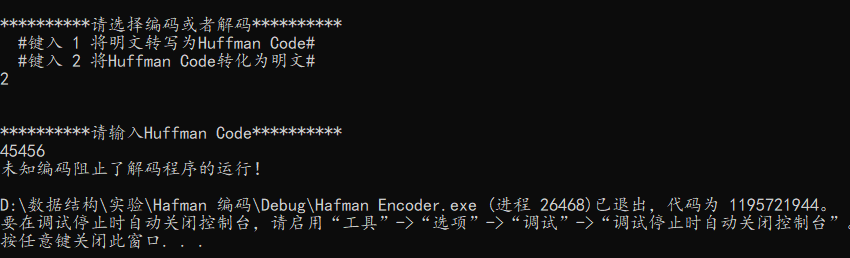
输出：



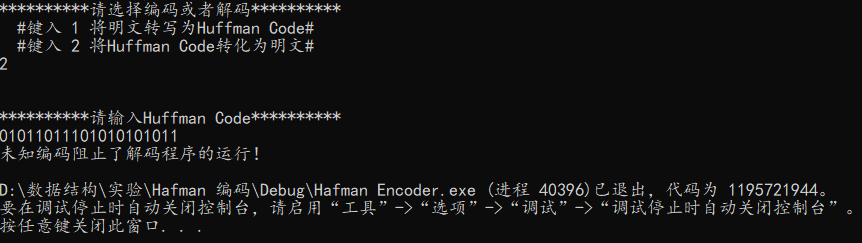
(5)输入：（选择解码功能）1001000100010100111111011101000011000111010000000101000011110100111010100000000110011011001101100001001000011111111100010000100100011111010110000001000101001110000010110000000110011111001111011110111111001000001111101110000011111011101011101

输出：

(6)输入：（选择解码功能）(错误输入，不合法的Huffman编码串) 45456

输出：

(7)输入：（选择解码功能）(错误输入，不存在的Huffman编码串) 01011011101010101011

输出：

七、附录

#pragma warning (disable:4996)

#include <iostream>

#include <cstdio>

#include<cassert>

#include <string>

#include <algorithm>

#define ERROR 0

#define SUCCESS 1

#define TRUE 1

#define FALSE 0

#define STACK\_INIT\_SIZE 300

#define STACK\_INCREMENT 10

//ERROR\_EXIT\_CODE

#define INVALID\_INPUT 0x474544D8

//开启DEBUG输出

//#define DEBUG\_MODE\_ON

using namespace std;

//全局变量

int array\_length;

typedef int Status;

//哈夫曼树节点

typedef struct HuffmanTreeNode {

char letter;

int weight;

HuffmanTreeNode\* left;

HuffmanTreeNode\* right;

bool isDeleted;

int pre\_order\_counted\_times;

} \*TreeNode;

//哈夫曼树

struct HuffmanTree {

HuffmanTreeNode\* root;

};

typedef TreeNode ElemType;

typedef struct SqStack {

ElemType\* base;

ElemType\* top;

int stack\_size;

} Stack;

//初始化一个栈

Status InitStack(Stack& s) {

s.base = (ElemType\*)malloc(STACK\_INIT\_SIZE \* sizeof(ElemType));

if (s.base == nullptr) {

perror("Unable to allocate to memory space");

exit(OVERFLOW);

}

else {

s.top = s.base;

s.stack\_size = STACK\_INIT\_SIZE;

return SUCCESS;

}

}

//将新的元素推入栈中

Status Push(Stack& s, TreeNode e) {

if ((s.top - s.base) >= s.stack\_size) {//检查是否栈存满

//重新追加空间，大小为STACK\_INCREMENT

s.base = (ElemType\*)realloc(s.base, s.stack\_size + STACK\_INCREMENT);

//检查时是否成功分配到了内存空间

if (s.base == nullptr) {

perror("Unable to allocate to memory space");

exit(OVERFLOW);

}

//更新栈顶位置和栈大小(stack\_size)记录

s.top = s.base + s.stack\_size;

s.stack\_size = s.stack\_size + STACK\_INCREMENT;

}

\*s.top = e;

s.top++;

return SUCCESS;

}

//出栈

Status Pop(Stack& s, ElemType& e) {

if (s.top == s.base) {

return ERROR;

}

else {

s.top--;

e = \*s.top;

return SUCCESS;

}

}

//判断栈是否为空

Status StackEmpty(Stack s) {

if (s.base == s.top)

return TRUE;

else

return FALSE;

}

//将字母频度表存入二叉树节点，并将二叉树节点的地址存入数组中

void GetAlphabetFreq(TreeNode\* node\_array) {

cout << "\*\*\*\*\*\*\*\*\*\*获取字母频度表\*\*\*\*\*\*\*\*\*\*" << endl;

auto sp = (TreeNode)malloc(sizeof(HuffmanTreeNode));

if (sp == nullptr)

exit(OVERFLOW);

sp->letter = ' ';

cout << "请输入\' 空格 \'的权值：";

cin >> sp->weight;

sp->isDeleted = false;

sp->left = nullptr;

sp->right = nullptr;

sp->pre\_order\_counted\_times = 0;

node\_array[0] = sp;

for (int i = 0; i < 26; ++i) {

auto tn = (TreeNode)malloc(sizeof(HuffmanTreeNode));

if (tn == nullptr)

exit(OVERFLOW);

tn->letter = (char)('a' + i);

cout << "请输入\' " << tn->letter << " \'的权值：";

cin >> tn->weight;

tn->isDeleted = false;

tn->left = nullptr;

tn->right = nullptr;

tn->pre\_order\_counted\_times = 0;

node\_array[i + 1] = tn;

}

cout << endl << "\*\*\*\*\*\*\*\*\*\*获取结束\*\*\*\*\*\*\*\*\*\*" << endl;

array\_length = 27;

}

TreeNode GetMinWeight(TreeNode\* node\_array) {

int min\_index = 10000;

int min\_weight = 10000;

for (int i = 0; i < array\_length; ++i) {

if (node\_array[i]->weight < min\_weight && !node\_array[i]->isDeleted) {

min\_weight = node\_array[i]->weight;

min\_index = i;

}

}

node\_array[min\_index]->isDeleted = true;

return node\_array[min\_index];

}

bool isEmpty(TreeNode\* node\_array) {

for (int i = 0; i < array\_length; ++i)

if (!node\_array[i]->isDeleted)

return false;

return true;

}

void SaveNodeToArray(TreeNode new\_node, TreeNode\* node\_array) {

node\_array[array\_length] = new\_node;

array\_length++;

}

void CreateHuffmanTree(HuffmanTree& t, TreeNode\* node\_array) {

TreeNode parent\_node;

while (true) {

TreeNode node\_1, node\_2;

node\_1 = GetMinWeight(node\_array);

node\_2 = GetMinWeight(node\_array);

parent\_node = (TreeNode)malloc(sizeof(HuffmanTreeNode));

if (parent\_node == nullptr)

exit(OVERFLOW);

parent\_node->isDeleted = false;

parent\_node->left = node\_1;

parent\_node->right = node\_2;

parent\_node->letter = '#';

parent\_node->weight = node\_1->weight + node\_2->weight;

parent\_node->pre\_order\_counted\_times = 0;

if (isEmpty(node\_array))

break;

SaveNodeToArray(parent\_node, node\_array);

}

t.root = parent\_node;

}

//产生Huffman编码

void HuffmanCodeGenerator(HuffmanTree t, string\* huffman\_code\_list) {

string s;

s = "";

TreeNode n = t.root;

Stack node\_stack;

InitStack(node\_stack);

while (true) {

if (n == t.root) {

if (n->pre\_order\_counted\_times == 0) {

n->pre\_order\_counted\_times++;

Push(node\_stack, n);

n = n->left;

s += '1';

}

if (n->pre\_order\_counted\_times == 1) {

n->pre\_order\_counted\_times++;

Push(node\_stack, n);

n = n->right;

s += '0';

}

if (n->pre\_order\_counted\_times == 2)

break;

}

else {

if (n->letter == '#') {

if (n->pre\_order\_counted\_times == 0) {

n->pre\_order\_counted\_times++;

Push(node\_stack, n);

n = n->left;

s += '1';

}

if (n->pre\_order\_counted\_times == 1) {

n->pre\_order\_counted\_times++;

Push(node\_stack, n);

n = n->right;

s += '0';

}

if (n->pre\_order\_counted\_times == 2) {

Pop(node\_stack, n);

s = s.substr(0, s.length() - 1);

}

}

else {

std::cout << '\'' << n->letter << '\'' << "的权值为：" << n->weight << ",Huffman编码为：" << s << endl;

if (n->letter != ' ')

huffman\_code\_list[n->letter - 'a' + 1] = s;

if (n->letter == ' ')

huffman\_code\_list[0] = s;

Pop(node\_stack, n);

s = s.substr(0, s.length() - 1);

}

}

}

}

string SearchHuffmanCode(char c, string\* huffman\_code\_list) {

return huffman\_code\_list[c - 'a' + 1];

}

string HuffmanEncoder(string plaintext, string\* huffman\_code\_list) {

transform(plaintext.begin(), plaintext.end(), plaintext.begin(), ::tolower);

string ciphertext;

for (char i : plaintext) {

if (i == ' ')

ciphertext += huffman\_code\_list[0];

else

ciphertext += SearchHuffmanCode(i, huffman\_code\_list);

}

return ciphertext;

}

string HuffmanDecoder(const string& ciphertext, string\* huffman\_code\_list) {

string plaintext;

int length = 1; //10010000100001100011111101000011100011011110

int start\_index = 0;

bool isMatch = false;

while (true) {

if (start\_index + length > ciphertext.length())

return plaintext;

string temp = ciphertext.substr(start\_index, length);

for (int i = 0; i < 27; ++i) {

if (temp == huffman\_code\_list[i]) {

isMatch = true;

if (i == 0)

plaintext += ' ';

else

plaintext += (char)(i + 'a' - 1);

}

}

if (isMatch) {

start\_index += length;

length = 1;

isMatch = false;

}

else {

if (start\_index + length >= ciphertext.length()) {

cout << "未知编码阻止了解码程序的运行！" << endl;

exit(INVALID\_INPUT);

}

length++;

}

}

}

string GetBinCode(const string& s)

{

string bin\_code;

cout << "\*\*\*\*\*\*\*\*\*\*原文的二进制编码\*\*\*\*\*\*\*\*\*\*" << endl;

for (char i : s)

{

char temp[100];

itoa(i, temp, 2);

cout << i << '\t' << temp << endl;

bin\_code += temp;

}

return bin\_code;

}

int main() {

cout << "Tips:Here is the Sample Frequency of the alphabet:" << endl

<< "186 64 13 22 32 103 21 15 47 57 1 5 32 20 57 63 15 1 48 51 80 23 8 18 1 16 1" << endl << endl << endl;

//创建Huffman树

auto\* node\_array = new TreeNode[1000];

GetAlphabetFreq(node\_array);

HuffmanTree test\_tree{};

CreateHuffmanTree(test\_tree, node\_array);

auto\* huffman\_code\_list = new string[30];

//获取27种字符的Huffman编码

HuffmanCodeGenerator(test\_tree, huffman\_code\_list);

//对应地将英文转换为Huffman编码处理后的密文

start:

cout << "\*\*\*\*\*\*\*\*\*\*请选择编码或者解码\*\*\*\*\*\*\*\*\*\*\n #键入 1 将明文转写为Huffman Code#\n #键入 2 将Huffman Code转化为明文#" << endl;

char choice;

fflush(stdin);

cin >> choice;

switch (choice) {

case '1': {

char plaintext[1000] = { 0 };

cout << endl << endl << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*请输入原文\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;

cin.ignore();

gets\_s(plaintext);

string ciphertext = HuffmanEncoder(plaintext, huffman\_code\_list);

cout << "Huffman Code如下：" << endl;

cout << ciphertext << endl;

cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*转写结束\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl << endl;

string bin\_text = GetBinCode(plaintext);

cout << "\*\*\*\*\*\*\*\*\*\*\*原文的二进制编码\*\*\*\*\*\*\*\*\*\*\*" << endl;

cout << bin\_text << endl;

double compression\_ratio = (1.0 - (double)ciphertext.length() / (double)bin\_text.length()) \* 100;

cout << endl << "压缩率 ---> " << compression\_ratio << '%' << endl;

break;

}

case '2': {

char ciphertext[1000] = { 0 };

cout << endl << endl << "\*\*\*\*\*\*\*\*\*\*请输入Huffman Code\*\*\*\*\*\*\*\*\*\*" << endl;

cin.ignore();

gets\_s(ciphertext);

string plaintext = HuffmanDecoder(ciphertext, huffman\_code\_list);

cout << "明文如下：" << endl;

cout << plaintext << endl;

cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*转写结束\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl << endl << endl;

string bin\_text = GetBinCode(plaintext);

cout << "\*\*\*\*\*\*\*\*\*\*\*原文的二进制编码\*\*\*\*\*\*\*\*\*\*\*" << endl;

cout << bin\_text << endl;

string ciphertext\_copy = ciphertext;

double compression\_ratio = ((double)bin\_text.length() / (double)ciphertext\_copy.length() - 1.0) \* 100;

cout << endl << "体积增长率 ---> " << compression\_ratio << '%' << endl;

break;

}

default:

exit(INVALID\_INPUT);

}

cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*程序运行结束\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl << endl << endl << endl << endl;

goto start;

}

*源代码1 Huffman Encoder and Decoder.cpp*