哈尔滨工业大学计算学部

实验报告

课程名称：数据结构与算法

课程类型：专业基础（必修）

实验项目：﻿树形结构及其应用

实验题目：﻿哈夫曼编码与译码方法

实验日期：2022.10.22

班级：2103103

学号：2021110608

姓名：孟悦琦

|  |  |  |
| --- | --- | --- |
| 设计成绩 | 报告成绩 | 指导老师 |
|  |  | 翟德明 |

**一、实验目的**

﻿﻿ 哈夫曼编码是一种以哈夫曼树（最优二叉树，带权路径长度最小的二叉树）

为基础变长编码方法。其基本思想是：将使用次数多的代码转换成长度较短的编

码，而使用次数少的采用较长的编码，并且保持编码的唯一可解性。在计算机信

息处理中，经常应用于数据压缩。是一种一致性编码法（又称"熵编码法"），用

于数据的无损压缩。要求实现一个完整的哈夫曼编码与译码系统。

**二、实验要求及实验环境**

实验要求：﻿1. 从文件中读入任意一篇英文文本文件，分别统计英文文本文件中各字符（包括标点符号和空格）的使用频率；

2. 根据已统计的字符使用频率构造哈夫曼编码树，并给出每个字符的哈夫曼编

码（字符集的哈夫曼编码表）；

3. 将文本文件利用哈夫曼树进行编码，存储成压缩文件（哈夫曼编码文件）；

4. 计算哈夫曼编码文件的压缩率；

5. 将哈夫曼编码文件译码为文本文件， 并与原文件进行比较。

**三、设计思想**（本程序中的用到的所有数据类型的定义，主程序的流程图及各程序模块之间的调用关系、核心算法的主要步骤）

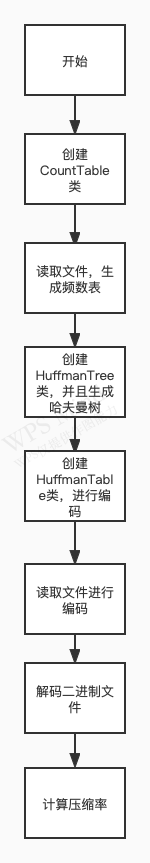
1．逻辑结构

2．存储结构

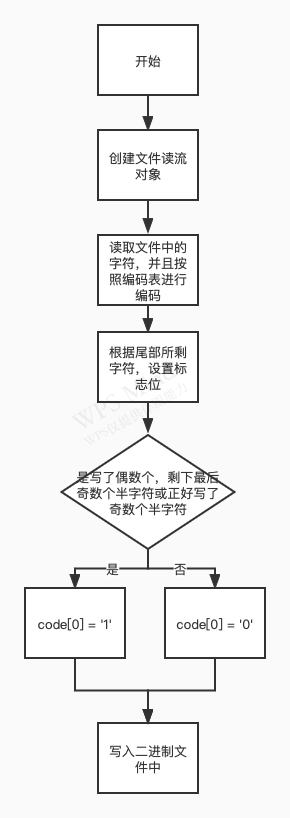
本程序中用到的所有数据类型：

1. HuffmanTable类，用来记录哈夫曼编码表，并且根据编码表对文章进行编码、对编码好的文件进行译码。内含：存储哈夫曼编码的数组HuffTable，二进制和十六进制相互译码的全译表和缺省时的译码表。更新哈夫曼编码的update\_HFTable方法，打印哈夫曼编码的printTable方法，以及编码方法encode和解码方法decode。
2. CountTable类，用来记录一个文章中的字符出现频率。内含：核心列表cnt\_table，储存字符出现的频数。列表的长度len。接口方法：getlen，getcnt。构造方法：CountTable。读文件的方法readtext。打印表方法printTable。
3. HuffmanTree类，Haffman树，用来储存根据CountTable构造的哈夫曼树。内含HuffmanTree数组，存储哈夫曼树。int型变量count，记录HuffmanTree内含元素数。两个辅助元素node1，node2。选择最小元素的方法select\_min。寻某元素父元素的方法getparent。寻某元素左儿子的方法getlchild。寻某元素左儿子的方法getrchild。
4. FILEUtil工具类，用来计算文件的大小。内含读取文件大小的方法size。

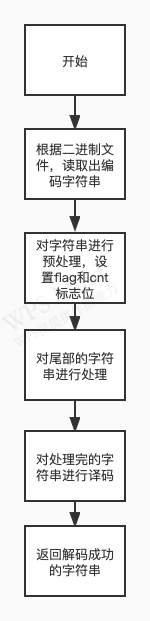
主函数流程图：



编码函数流程图：



译码函数流程图：



**四、测试结果**

测试源文件：

If the latest news is any indication, the big move—or any move at all—the Lakers need isn't coming anytime soon. Here's what Adrian Wojnarowski said on ESPN's NBA Countdown (h/t Hoops Rumors' Luke Adams):

“I’m told to expect Rob Pelinka and the Lakers to wait until post-Thanksgiving, 20 games into the season. And see what teams may start pivoting who don’t start off well, who decide that they may start to unload players and perhaps get involved in the Victor Wembanyama sweepstakes.”

Um, yeah, no. The Lakers cannot afford to wait 20 games. Seven of their next 18 come against probable tankers, but they could very easily lose the other 11. And starting off 6-13 in this year's hellfire of a Western Conference is akin to consigning yourself to the play-in race. That's inexcusable, always, when you have LeBron. But especially when you have LeBron in his age-38 season.

To be sure, this isn't about Westbrook alone. The Lakers' problems run so much deeper than Westbrook. He's merely a symptom of the disease that is Los Angeles' ass backwards roster construction.

General manager Rob Pelinka and the rest of humanity watched as LeBron made recurring trips to the Finals and won multiple championships in Cleveland and Miami surrounded by shooting and, oftentimes, defensive depth. Heck, Pelinka and the rest of the Lakers front reaped the firsthand benefits of that model in 2020, when they won the NBA title. Their response to watching proven models rack up titles was to drown LeBron (and Davis) with a supporting cast of non-wings and non-shooters each of the past two seasons.

Los Angeles has opened this season shooting 19-of-85 from deep22.4 percent. That is putrid. And totally expected.

编码文件：



(01形式的太长了，就不展示了)

**五、经验体会与不足**

我认为这次实验让我很好的了解了Huffman编码的实现原理。同时我也对树的应用和算法有了更多的认识。

这次的不足我认为主要是，首先，有一些非ASCII编码的字符难以处理。其次，代码在处理二进制文件读写的地方略显冗余。

**六、附录：源代码（带注释）**

main.c

#include <iostream>

#include "HuffmanTree.h"

#include "CountTable.h"

#include "HuffmanTable.h"

#include "FILEUtil.h"

int main() {

HuffmanTree h;

CountTable CT;

CT.readtext("/Users/mengfanxing/Desktop/DS\_lab2/source.txt");

std::cout << "The countTabel is :" << std::endl;

CT.printTable();

h.creat\_HFTree(CT);

HuffmanTable HT;

HT.update\_HFTable(h);

std::cout << "The HuffmanTable is:" << std::endl;

HT.printTable();

std::cout << "The code after encode is:" << std::endl;

std::cout << HT.encode("/Users/mengfanxing/Desktop/DS\_lab2/source.txt") << std::endl;

std::ofstream ofs;

ofs.open("/Users/mengfanxing/Desktop/DS\_lab2/16bite.txt",std::ios::out);

ofs << HT.encode("/Users/mengfanxing/Desktop/DS\_lab2/source.txt");

std::cout << "The code after decode is:" << std::endl;

std::cout << HT.decode("/Users/mengfanxing/Desktop/DS\_lab2/code.dat") << std::endl;

FILEUtil fl;

float cr = fl.size("/Users/mengfanxing/Desktop/DS\_lab2/code.dat")/fl.size("/Users/mengfanxing/Desktop/DS\_lab2/source.txt");

std::cout << "The compression ratio is " << cr << std::endl;

}

HuffmanTree.h

//

// Created by 孟繁兴 on 2022/10/10.

//

#ifndef DS\_LAB2\_HUFFMANTREE\_H

#define DS\_LAB2\_HUFFMANTREE\_H

#include <iostream>

#include "CountTable.h"

#include <string>

#include <stack>

typedef struct {

int weight;

int parent;

int lchild;

int rchild;

}HTNODE;

class HuffmanTree {

private:HTNODE \*HuffmanTree;

int count; // HuffmanTree内含元素数

int node1,node2;

public:void read\_text();

void creat\_HFTree(CountTable CT);

void select\_min(int len);

void print\_Tree(); // 调试函数

int getparent(int i) const;

int getlchild(int i) const;

int getrchild(int i) const;

};

#endif //DS\_LAB2\_HUFFMANTREE\_H

HuffmanTree.cpp

//

// Created by 孟繁兴 on 2022/10/10.

//

#include "HuffmanTree.h"

// 读取文章的算法

void HuffmanTree::read\_text() {

char ch1;

while (ch1 != '\0') {

}

}

// 选择哈夫曼树的最小权两节点。

void HuffmanTree::select\_min(int len) {

int i, temp;

node1 = node2 = -1;

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

if (HuffmanTree[i].parent == -1) {

// 如果node1,node2还未被标记，那么标记他

if (node1 == -1) {

node1 = i;

continue;

}

if (node2 == -1) {

node2 = i;

continue;

}

// 保持node1中存储的元素最小

if (HuffmanTree[node1].weight > HuffmanTree[node2].weight) {

temp = node1;

node1 = node2;

node2 = temp;

}

if (HuffmanTree[node2].weight > HuffmanTree[i].weight) {

node2 = i;

continue;

}

}

}

}

// 通过读取的频数表，生成哈夫曼树的算法

void HuffmanTree::creat\_HFTree(CountTable CT) {

HuffmanTree = new HTNODE[2\*CT.getlen()-1];

count = 2\*CT.getlen()-1;

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

// 初始化树

HuffmanTree[i].parent = -1;

HuffmanTree[i].lchild = -1;

HuffmanTree[i].rchild = -1;

if (i < CT.getlen()) {

HuffmanTree[i].weight = CT.getcnt(i);

}

}

// 生成哈夫曼树

for (int j = CT.getlen(); j < count; ++j) {

select\_min(j);

HuffmanTree[j].weight = HuffmanTree[node1].weight + HuffmanTree[node2].weight;

HuffmanTree[node1].parent = HuffmanTree[node2].parent = j;

HuffmanTree[j].lchild = node1;

HuffmanTree[j].rchild = node2;

}

}

void HuffmanTree::print\_Tree() {

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

std::cout << HuffmanTree[i].weight << "\t" << HuffmanTree[i].parent << "\t"

<< HuffmanTree[i].rchild << "\t" << HuffmanTree[i].lchild << std::endl;

}

}

// 调用第i个位置parent值的算法

int HuffmanTree::getparent(int i) const {

return HuffmanTree[i].parent;

}

// 调用第i个位置lchild值的算法

int HuffmanTree::getlchild(int i) const {

return HuffmanTree[i].lchild;

}

// 调用第i个位置rchild值的算法

int HuffmanTree::getrchild(int i) const {

return HuffmanTree[i].rchild;

}

HuffmanTable.h

//

// Created by 孟繁兴 on 2022/10/11.

//

#ifndef DS\_LAB2\_HUFFMANTABLE\_H

#define DS\_LAB2\_HUFFMANTABLE\_H

#include "HuffmanTree.h"

#include "CountTable.h"

#include <fstream>

std::string CHARTO16STR(unsigned char c);

typedef struct {

char c;

std::string code;

} Huffcode;

typedef struct {

std::string \_01code;

std::string \_16code;

} encodeTable;

class HuffmanTable {

private:Huffcode HuffTable[0x80-0x0A];

void setcode(int n);

// 储存编码的辅助栈s

std::stack<char> s;

// 全译表

encodeTable full[16] = {"0000","0",

"0001","1",

"0010","2",

"0011","3",

"0100","4",

"0101","5",

"0110","6",

"0111","7",

"1000","8",

"1001","9",

"1010","a",

"1011","b",

"1100","c",

"1101","d",

"1110","e",

"1111","f"};

// 缺省时的译码表

encodeTable lack[14] = {"0","0",

"1","1",

"10","2",

"11","3",

"00","0",

"01","1",

"000","0",

"001","1",

"010","2",

"011","3",

"100","4",

"101","5",

"110","6",

"111","7"};

public:void update\_HFTable(HuffmanTree HFTree);

void printTable();

std::string encode(std::string name);

std::string decode(std::string name);

};

#endif //DS\_LAB2\_HUFFMANTABLE\_H

HuffmanTable.cpp

//

// Created by 孟繁兴 on 2022/10/11.

//

#include "HuffmanTable.h"

// 设置某个位置的Huffman编码

void HuffmanTable::setcode(int n) {

std::string code;

char c;

while (!s.empty()){

c = s.top();

s.pop();

code.push\_back(c);

}

HuffTable[n].code = code;

HuffTable[n].c = (char)0x0A+n;

}

// 更新Huffman编码表

void HuffmanTable::update\_HFTable(HuffmanTree HFTree) {

for (int i = 0; i < 0x80-0x0A; ++i) {

int j=i; // 用j对储存的数组进行遍历

while (HFTree.getparent(j) != -1) {

// 如果处在左支，那么取0，如果处在右支，那么取1

if(j == HFTree.getlchild(HFTree.getparent(j))) {

s.push('0');

} else if(j == HFTree.getrchild(HFTree.getparent(j))){

s.push('1');

}

j = HFTree.getparent(j);

}

setcode(i);

}

}

// 打印编码表

void HuffmanTable::printTable() {

for (int i = 0; i < 0x80-0x0A; ++i) {

std::cout << HuffTable[i].c << " " << HuffTable[i].code << std::endl;

}

}

// 对某个文件，根据本哈夫曼表进行编码

std::string HuffmanTable::encode(std::string name) {

// 创建流对象

std::ifstream ifs;

ifs.open(name,std::ios::out);

char c;

int cnt = 0; // 记录结尾所剩的字符数

int byte\_num = 0; // 记录输入的十六进制字符的个数

std::string pre\_code; // 返回的编码预处理

std::string code; // 返回的编码

std::string half\_byte; // 半个字节的数据

int i = 0;

bool flag = false;

while ((c=ifs.get()) != EOF) {

i = 0;

while (true) {

if (c == i + 0x0A) {

std::string s = HuffTable[i].code;

pre\_code.append(HuffTable[i].code); // 字符串连接

break;

}

if(i+0x0A > 0x80) {

flag = true;

break;

}

i++;

}

}

// 如果存在非法字符，则输出提醒用户的文字

if (flag) {

std::cout << "Do not have some letters' pre\_code" << std::endl;

}

// 关闭文件

ifs.close();

// 调试用输出

std::cout << pre\_code << std::endl;

// 将01文件压缩成16进制文件

while (pre\_code.size() != 0) {

if (pre\_code.size() >= 4) {

half\_byte = pre\_code.substr(0, 4);

pre\_code.erase(0, 4);

// 将半个byte的01字符串转为16进制字符

for (int i = 0; i < 16; ++i) {

if (half\_byte.compare(full[i].\_01code) == 0) {

code.append(full[i].\_16code);

byte\_num++;

}

}

} else {

half\_byte = pre\_code;

pre\_code.clear();

cnt = half\_byte.size();

// 对于不满四个字符的01字符串，在前面补0，并转为16进制

for (int i = 0; i < 14; ++i) {

if (half\_byte.compare(lack[i].\_01code) == 0) {

code.append(lack[i].\_16code);

}

}

}

}

// 判断cnt的大小，并将尾部所剩字符存储在第一个字节

switch (cnt) {

case 0:

code.insert(0,"00");

break;

case 1:

code.insert(0,"01");

break;

case 2:

code.insert(0,"02");

break;

case 3:

code.insert(0,"03");

break;

}

// 两种情况，一种是写了偶数个，剩下最后奇数个半字符，另外一种是正好写了奇数个半字符

if (byte\_num % 2 == 0 || (byte\_num%2==1 && cnt==0)) {

code[0] = '1'; // 如果有偶数个，则说明生成的16进制字符串未满，将首位置为1

} else {

code[0] = '0';

}

// 将编码写入二进制文件中

pre\_code = code;

unsigned char p;

std::string char\_byte; // int所占的4个字节

// 创造向二进制文件写的输出流

std::ofstream outfile ( "/Users/mengfanxing/Desktop/DS\_lab2/code.dat", std::ios::out | std::ios::binary );

// 判断是否打开

if (!outfile) {

std::cerr << "open error!" << std::endl;

return code;

}

while (pre\_code.size() != 0) {

char\_byte = pre\_code.substr(0, 2);

pre\_code.erase(0,2);

p = std::stoi(char\_byte, 0, 16);

outfile.write((char \*)&p,sizeof (p));

}

return code;

}

// 根据文件的操作暂时搁置

// 根据次编码表译码

std::string HuffmanTable::decode(std::string name) {

// 获取文章大小

long size = 0;

FILE \*fp = fopen(name.c\_str(), "r");

fseek(fp, 0, SEEK\_END);

size = ftell(fp);

fclose(fp);

std::string pre\_code;

std::string code;

unsigned char c;

FILE \*fin = fopen(name.c\_str(), "r");

// 从二进制文件中读出16进制字符串

for (int i = 0; i < size/ sizeof(unsigned char); ++i) {

c = fgetc(fin);

pre\_code.append(CHARTO16STR(c));

}

fclose(fin);

// 字符串译码预处理

int cnt; // 01字符串编码时最后一位剩余的字符数

int flag; // 做尾处理的flag

std::string s; // 读取一个16进制字符编码

flag = atoi(pre\_code.substr(0,1).c\_str());

pre\_code.erase(0,1);

cnt = atoi(pre\_code.substr(0,1).c\_str());

pre\_code.erase(0,1);

// 字符串译码

while (pre\_code.size() > 2) {

s = pre\_code.substr(0,1);

pre\_code.erase(0,1);

// 匹配16进制字符串，并将其转译为01字符串

for (int i = 0; i < 16; ++i) {

if (s == full[i].\_16code) {

s = full[i].\_01code;

break;

}

}

code.append(s);

}

// 对尾部进行最后的译码

std::string tail;

// 将尾部字符添加到tail字符串里进行后序处理

while (pre\_code.size() > 0) {

s = pre\_code.substr(0, 1);

pre\_code.erase(0, 1);

for (int i = 0; i < 16; ++i) {

if (s == full[i].\_16code) {

s = full[i].\_01code;

break;

}

}

tail.append(s);

}

// flag标记了结尾是否只有一个16进制字符

if (flag == 1) {

tail.erase(0,4);

}

// 对0的处理，全部输出，其余仅保留cnt位

if (cnt != 0) {

if (flag == 1) {

tail.erase(0, 4 - cnt);

} else {

tail.erase(4, 4 - cnt);

}

}

code.append(tail);

pre\_code = code;

code.clear();

// 开始译为字符编码

int i;

while (!pre\_code.empty()) {

for (i = 0; i < 0x80-0x0A; ++i) {

if (pre\_code.substr(0,HuffTable[i].code.size()).compare(HuffTable[i].code) == 0) {

code.append(1,HuffTable[i].c);

pre\_code.erase(0,HuffTable[i].code.size());

break;

}

}

if (i == 0x80-0x0A) {

std::cerr << "Can't find the Huffman code!" << std::endl;

}

}

return code;

}

std::string CHARTO16STR(unsigned char c) {

// 十六进制结果

std::string result;

// 十六进制数字表

char hex\_digits[16] = {

'0',

'1',

'2',

'3',

'4',

'5',

'6',

'7',

'8',

'9',

'a',

'b',

'c',

'd',

'e',

'f',

};

// 用do while可以支持对0的转换

do {

// 因为转换是反向取余，所以每次得出的十六进制数字要插入到最开头

result.insert(0, std::string() + hex\_digits[c % 16]);

c /= 16;

} while (c > 0);

// 统一使一个字节为两位0-f码

while (result.size() < 2) {

result.insert(0,"0");

}

return result;

}

FILEUtil.h

//

// Created by 孟繁兴 on 2022/10/19.

//

#ifndef DS\_LAB2\_FILEUTIL\_H

#define DS\_LAB2\_FILEUTIL\_H

class FILEUtil {

public:

float size(char \*name);

};

#endif //DS\_LAB2\_FILEUTIL\_H

FILEUtil.cpp

//

// Created by 孟繁兴 on 2022/10/19.

//

#include <cstdio>

#include "FILEUtil.h"

float FILEUtil::size(char \*name) {

FILE \*fp = fopen(name, "r");

fseek(fp, 0, SEEK\_END);

return ftell(fp);

}

CountTable.h

//

// Created by 孟繁兴 on 2022/10/11.

//

#ifndef DS\_LAB2\_COUNTTABLE\_H

#define DS\_LAB2\_COUNTTABLE\_H

#include <iostream>

#include <fstream>

#include <string>

typedef struct {

char c;

int cnt;

}cnt\_of\_word;

class CountTable {

private:

// 核心列表，储存字符出现的频数

cnt\_of\_word cnt\_table[0x80-0x0A];

// 列表长度

int len = 0x80-0x0A;

public:CountTable();

int getlen() const;

int getcnt(int i) const;

void readtext(std::string name);

void printTable();

};

#endif //DS\_LAB2\_COUNTTABLE\_H

CountTable.cpp

//

// Created by 孟繁兴 on 2022/10/11.

//

#include "CountTable.h"

// 构造一个CountTable

CountTable::CountTable() {

// 使用可以打印的字符为例，但由于\n的存在，把数据集扩大0x16的容量，扩展到0x80-0x0A

for (int i = 0; i < 0x80-0x0A; ++i) {

cnt\_table[i].c = (char)0x0A+i;

cnt\_table[i].cnt = 0;

}

// // 测试数据

// cnt\_table[0].cnt = 19;

// cnt\_table[1].cnt = 10;

// cnt\_table[2].cnt = 3;

// cnt\_table[3].cnt = 29;

}

// 返回频数列表的长度

int CountTable::getlen() const {

return len;

}

// 返回第i个数的频次

int CountTable::getcnt(int i) const {

return cnt\_table[i].cnt;

}

// 读取数据并设置频数表的函数

void CountTable::readtext(std::string name) {

// 创建流对象

std::ifstream ifs;

ifs.open(name,std::ios::out);

char c;

while ((c=ifs.get()) != EOF) {

if (c<=0x80 && c>=0x0A) {

cnt\_table[c-0x0A].cnt += 1; // 如果数据合法，对频数加一

}

}

// 关闭文件

ifs.close();

}

// 打印CountTable

void CountTable::printTable() {

for (int i = 0; i < 0x80 - 0x0A; ++i) {

std::cout << cnt\_table[i].c << "\t" << cnt\_table[i].cnt << std::endl;

}

}