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：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：2020.11.3

班级：1903002

学号：1190200526

姓名：沈城有

|  |  |  |
| --- | --- | --- |
| 设计成绩 | 报告成绩 | 指导老师 |
|  |  | 张岩 |

**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）为基础变长编码方式。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"），用于数据的无损耗压缩。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；

6. 能否利用堆结构，优化的哈夫曼编码算法；

7. 上述1-5的编码和译码是基于字符的压缩，考虑基于单词的压缩，完成上述

工作，讨论并比较压缩效果；

8. 上述1-5的编码是二进制的编码，可以采用K叉的哈夫曼树完成上述工作，实现“K进制”的编码和译码，并与二进制的编码和译码进行比较。

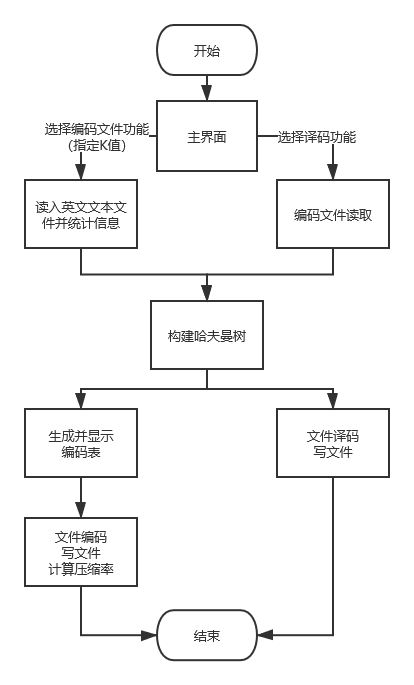
实验环境：Windows 10 Visual Studio 2019

**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）

1．逻辑设计

（1）主程序流程图

两部分的程序流程大体一致，具体流程如下图所示。



（2）各程序模块间调用关系：

1）排序使用C++提供sort函数，并自定义了cmp函数进行调用实现按权值降序排序；

2）构建哈夫曼树函数调用选取最小权值函数和初始化哈夫曼树函数，在堆优化的思考部分中更需深层调用最小堆删除顶部最小值函数、最小堆插入元素函数及最小堆初始化函数；

3）显示统计信息、显示编码表、计算压缩率等屏幕信息显示函数依赖于其他处理函数进行数据处理和传入结果数据；

4）通过程序主菜单选择执行编码或译码，并由主程序直接调用这两部分功能的主要实现函数及压缩率计算和统计信息显示函数。

注：具体函数定义及实现详见源代码。

2．物理设计（数据类型简要说明）

A.主体部分

（1）静态三叉链表哈夫曼树实现：



如图，利用此结构结点构成结构体数组形成哈夫曼树。

（2）结构体数组存储编码表：

typedef struct codingtable

{

int num;

char ch;

string code;

}CodeTab;

（3）使用C++STL中map统计字符及其频率信息：

map<char, double> result;

（4）文件存储文本信息、编码信息。

B.思考部分

此部分是在主体部分的基础上进行的补充和调整，除主体部分所使用的数据类型外，还使用了：

（1）最小堆（具体实现见源代码），利用这一结构优化了选取最小权值的过程，提高了整体的效率（单次选取时间复杂度从O(n)下降到O(log n)）；

（2）动态链表，用于构建K叉哈夫曼树的孩子域，提高了灵活性，减少了内存的占用（特别是当K值较小时），链表中每个结点包括两个域：一个保存子结点下标，另一个指向链表中下一结点。

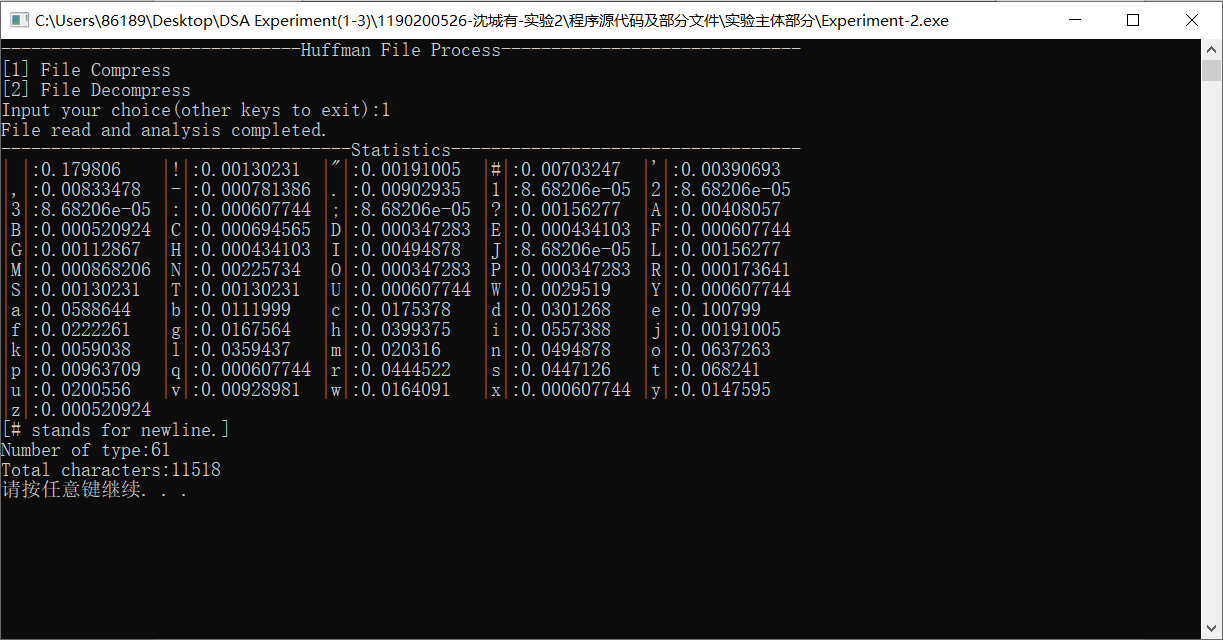
**四、测试结果**

**Ⅰ.实验主体部分**

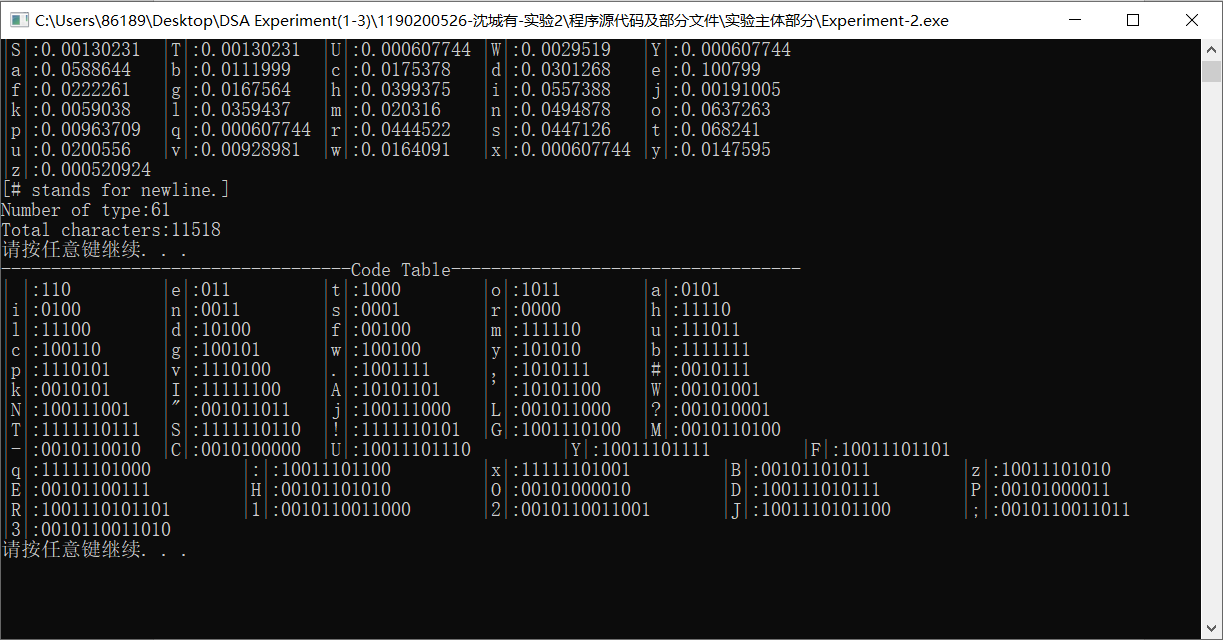
注：此部分主要是要求1-5的实现，并附加实现了01字符串与二进制01间转换以达到真正压缩文件的目的，增强了实际应用性。

1、读入文件并统计字符及其使用频率信息

下图为统计的字符及其频率信息，其中“#”代表换行符，信息使用制表符尽量对齐。

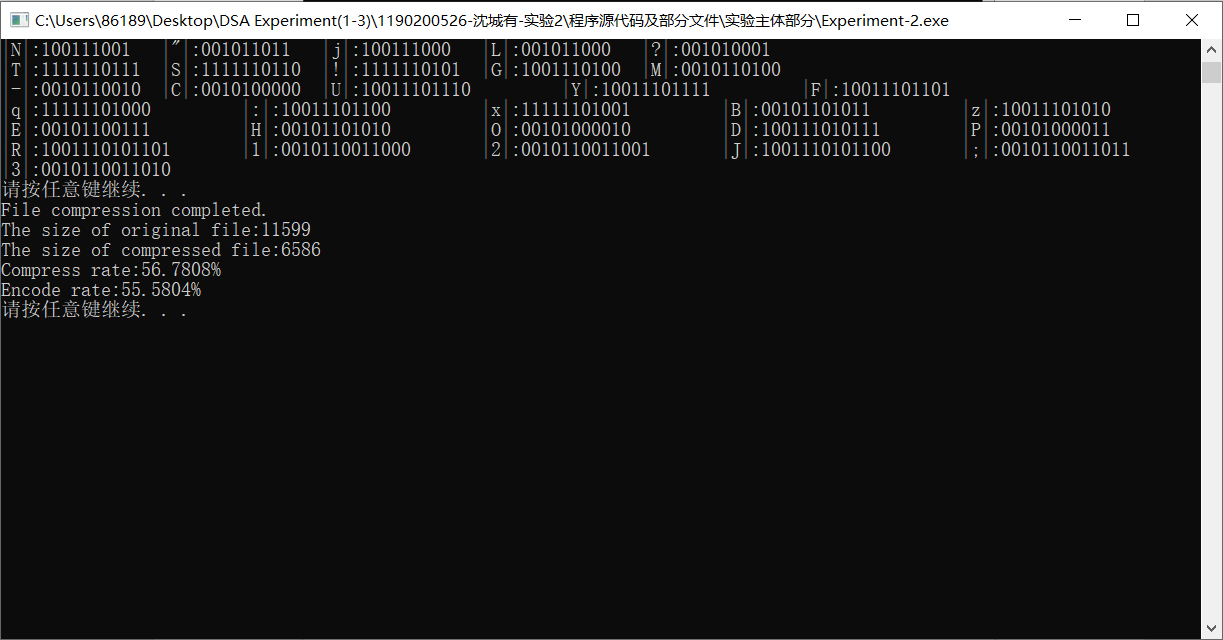


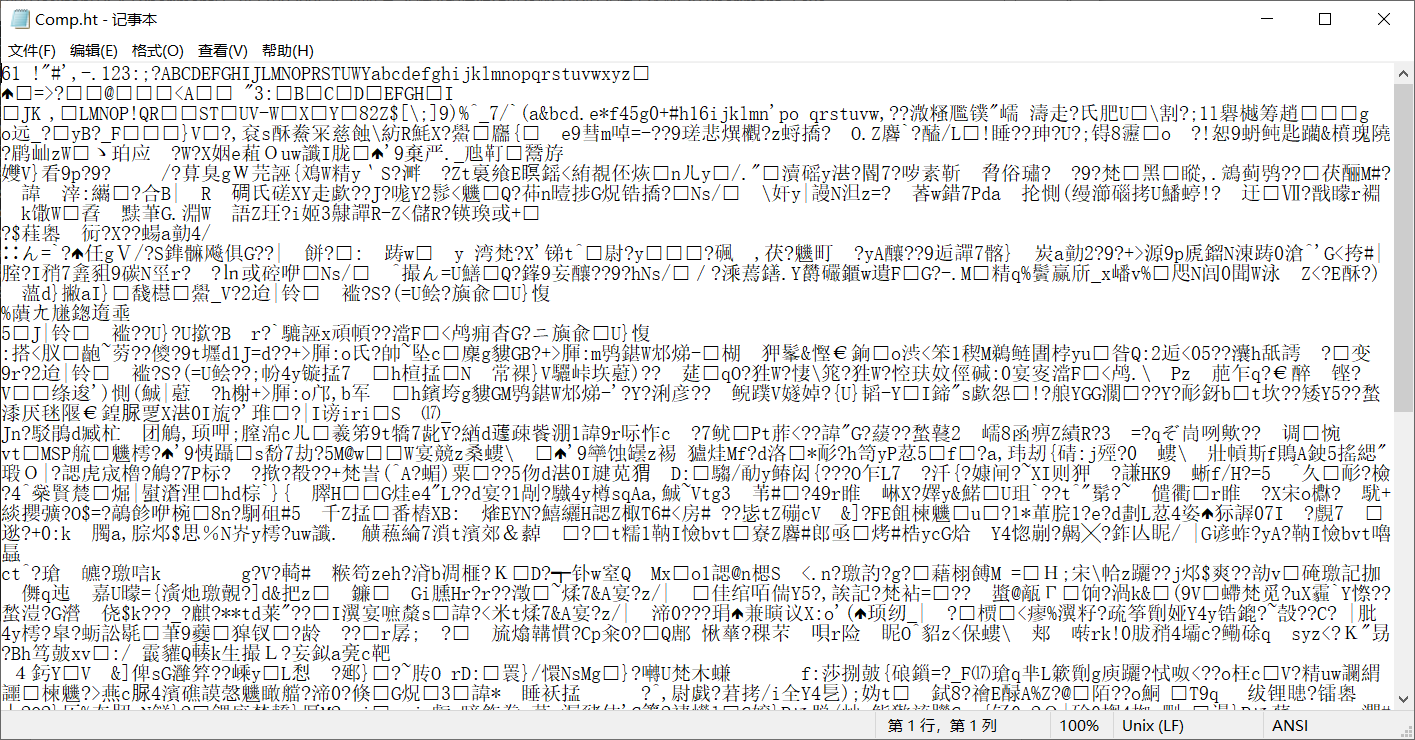
2、构造哈夫曼树并显示编码表



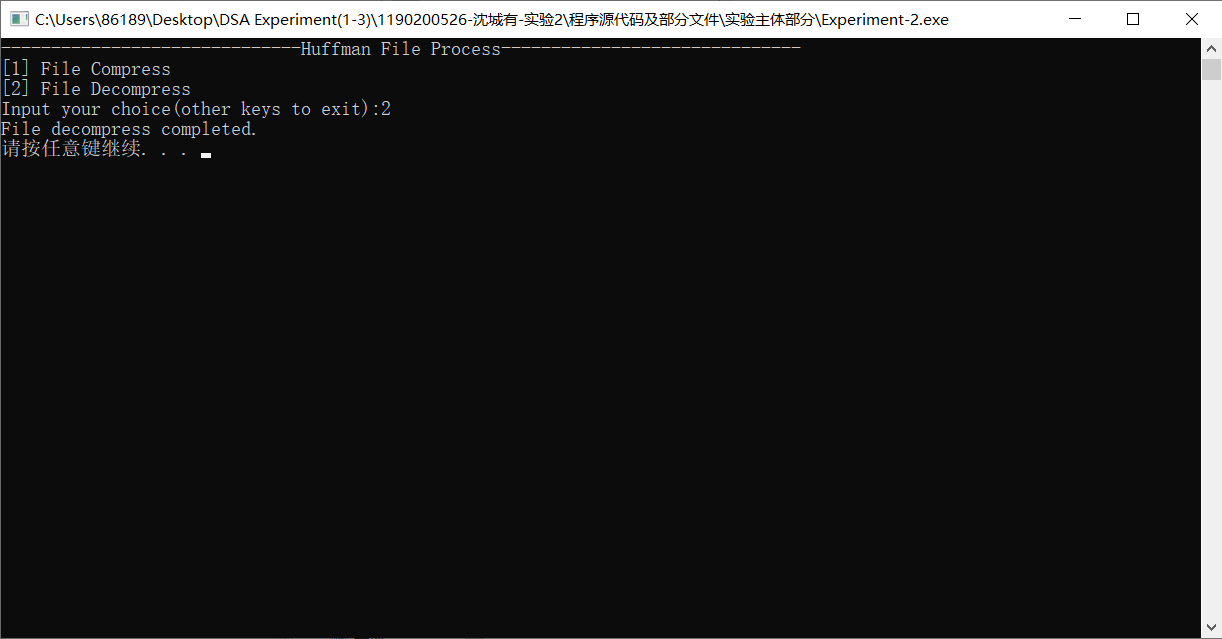
3、文件压缩并计算压缩率

下图为程序显示，数据中第一个为原文件大小（字节），第二个为压缩文件大小（字节），第三个、第四个分别为文件压缩率及编码率。

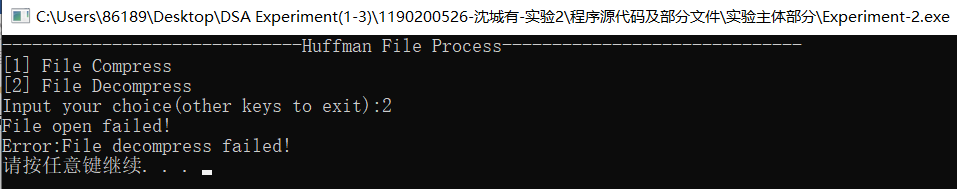


下图为编码后的文件，为真正做到压缩文件并脱离原文件及程序中暂存哈夫曼树进行解码，在文件开头写入了哈夫曼树并采用01串转二进制实现减少空间占用，故此处展示为乱码（实验思考部分展示了正常写入的编码文件）。

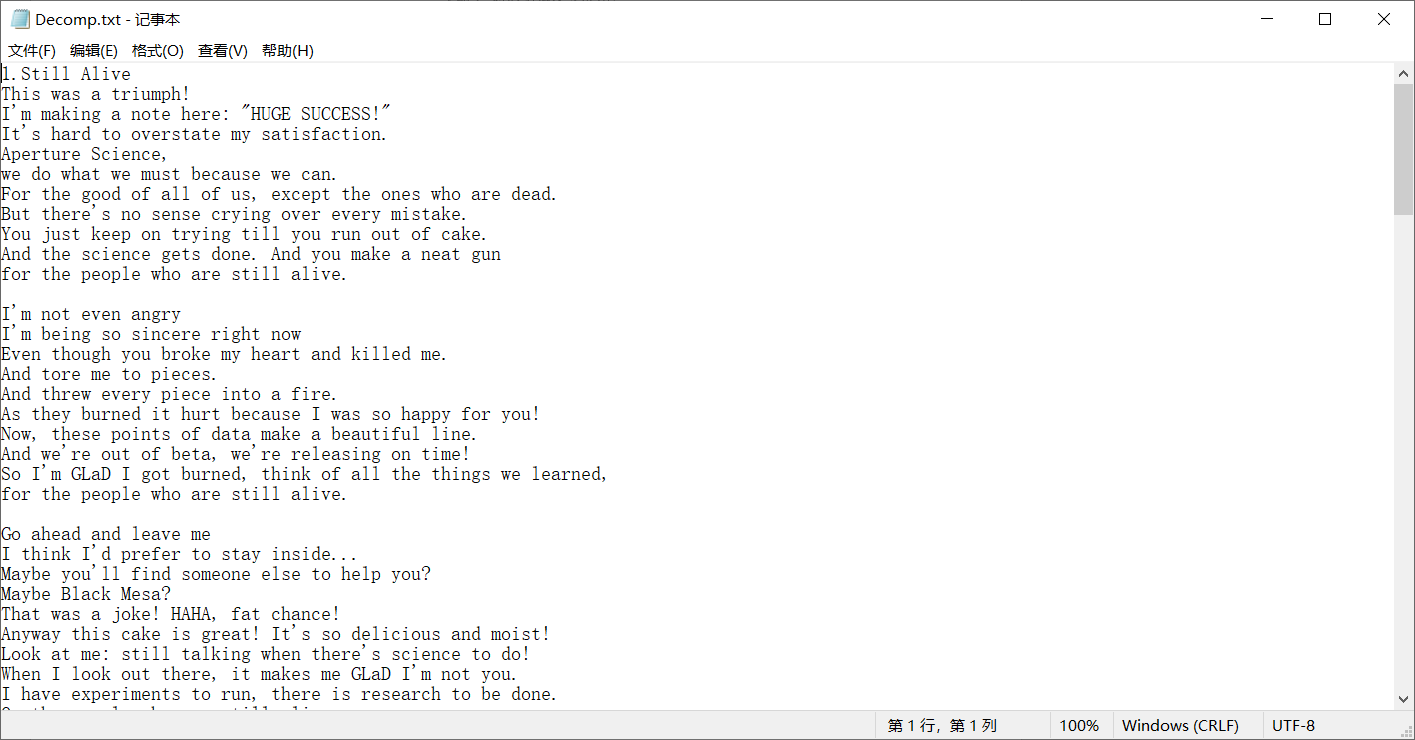
4、文件解码

图为文件解码成功时程序显示。

图为未找到或无法打开编码文件时程序显示。



5、解码后文件展示

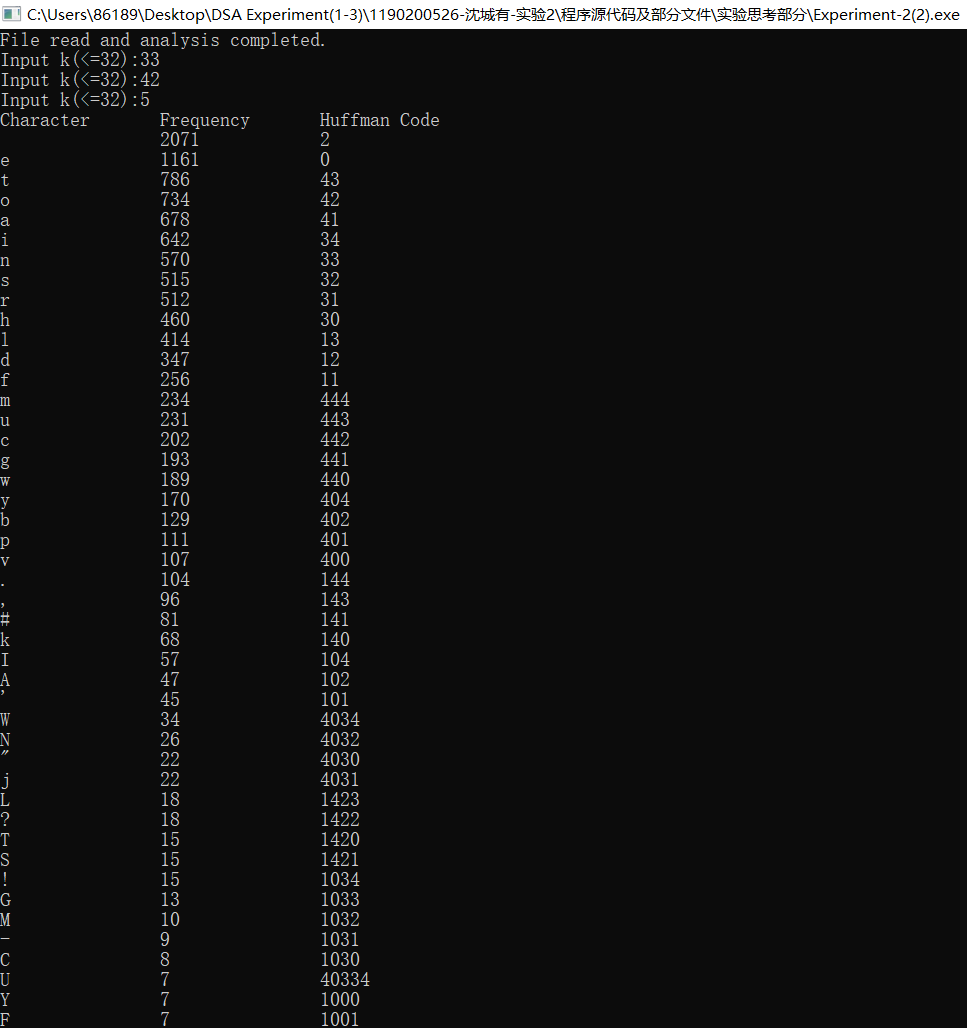


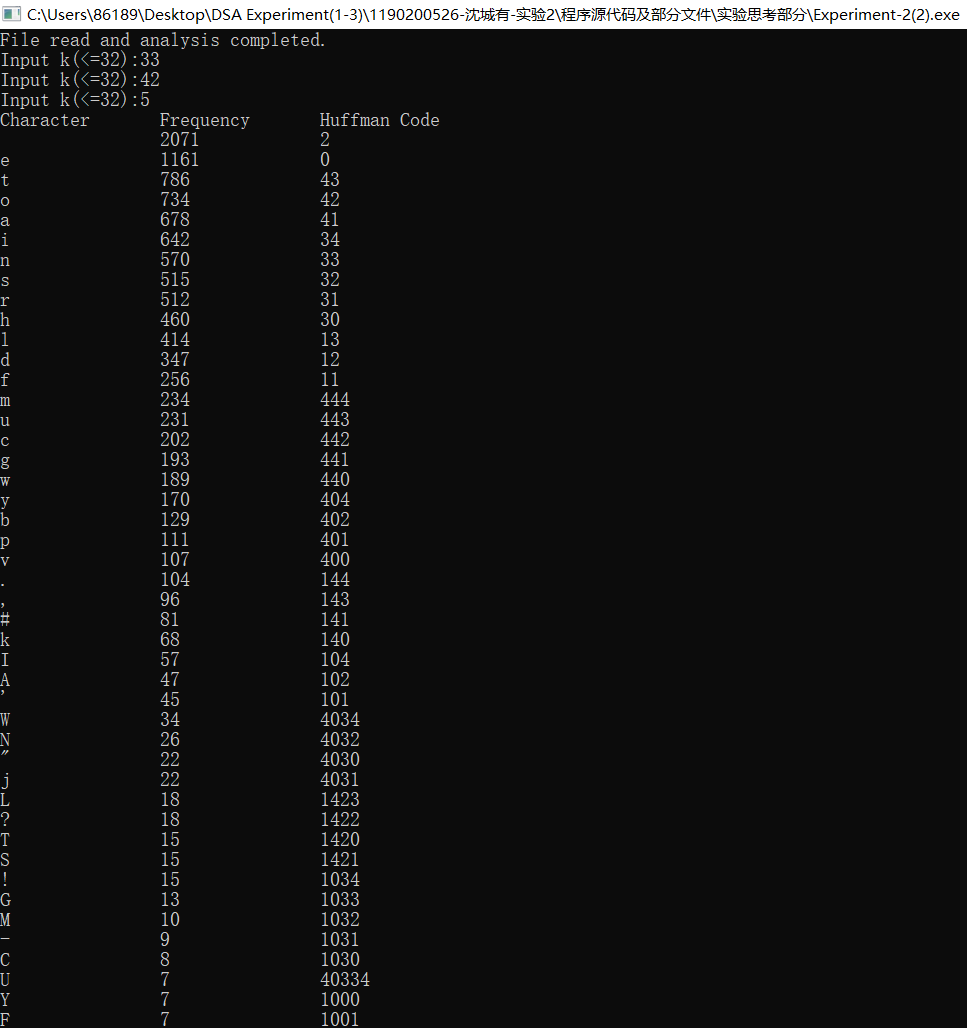
**Ⅱ.实验思考部分程序测试过程及结果展示**

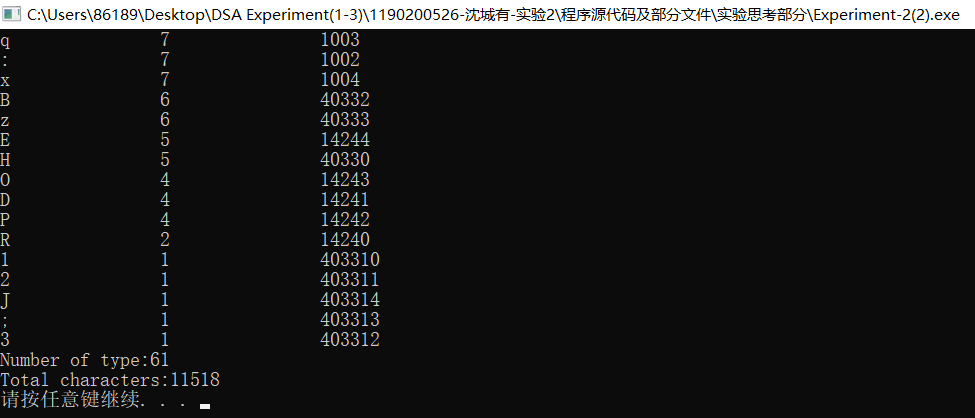
注：此部分主要是K叉哈夫曼树的实现及利用最小堆优化的最小权值选取。

1、用户输入K值、文件读入及统计信息和编码表程序显示

读入文件内容与主体部分一致，此处不再展示，仅展示程序部分（k=5）。



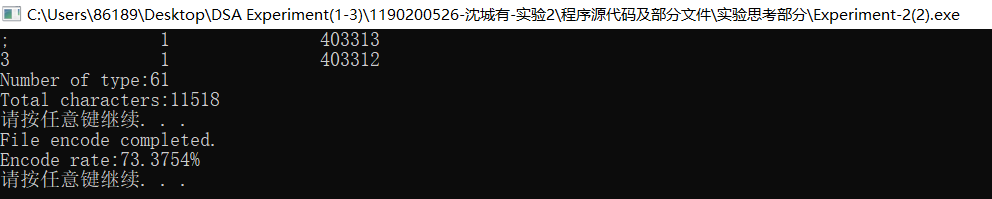




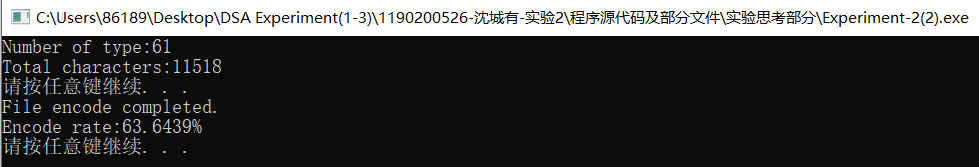
2、文件编码及计算压缩率

这里测试了三组K值，并分别计算了压缩率。

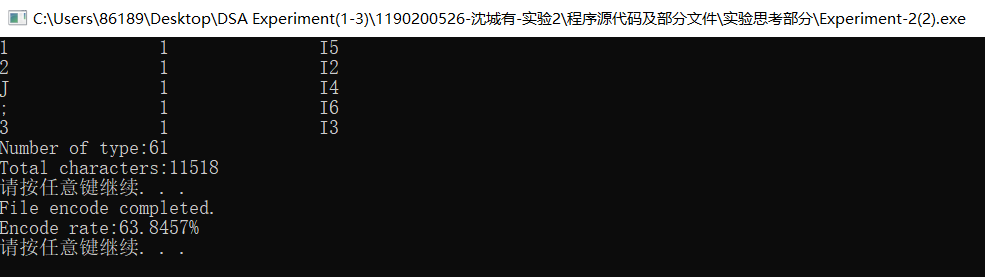
(1)K = 5:



(2)K = 13:



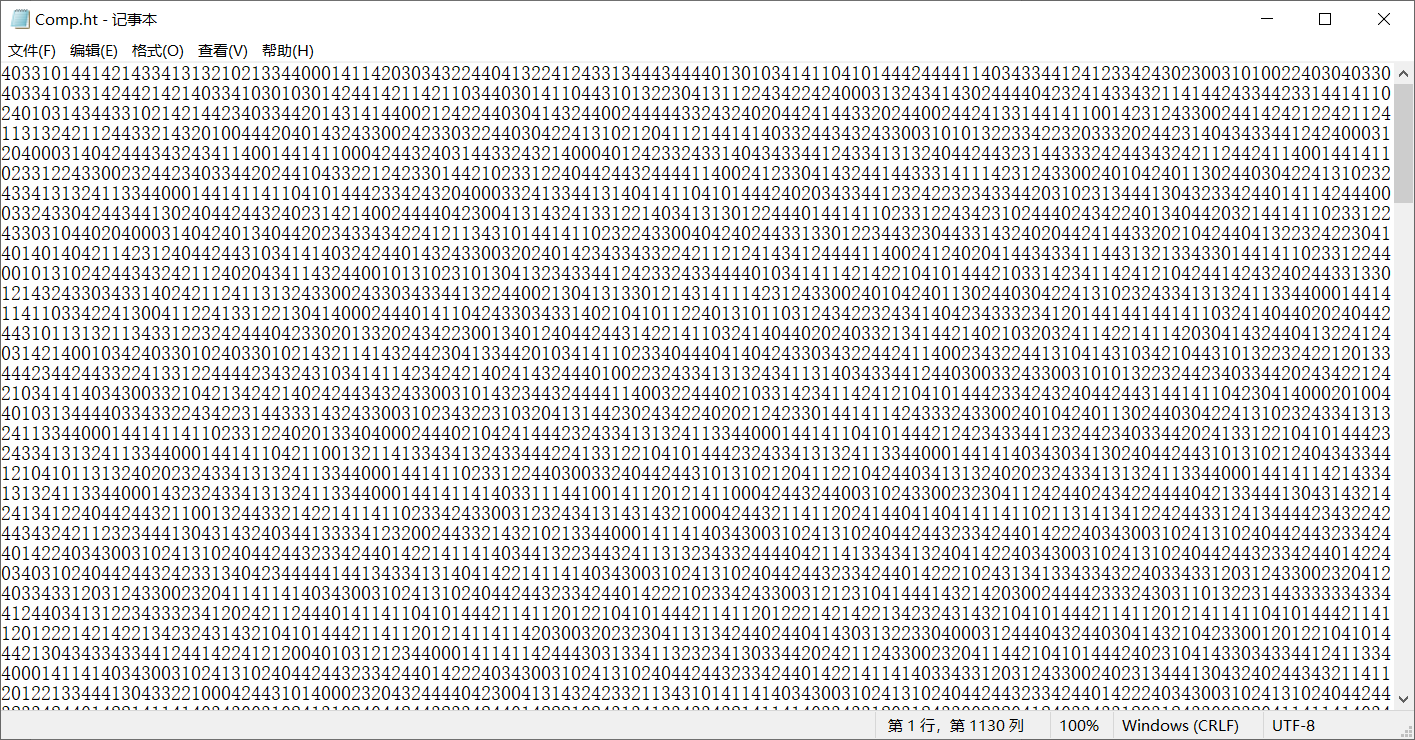
(3)K = 32:



3、编码文件展示

此处展示2中三种K值对应编码文件

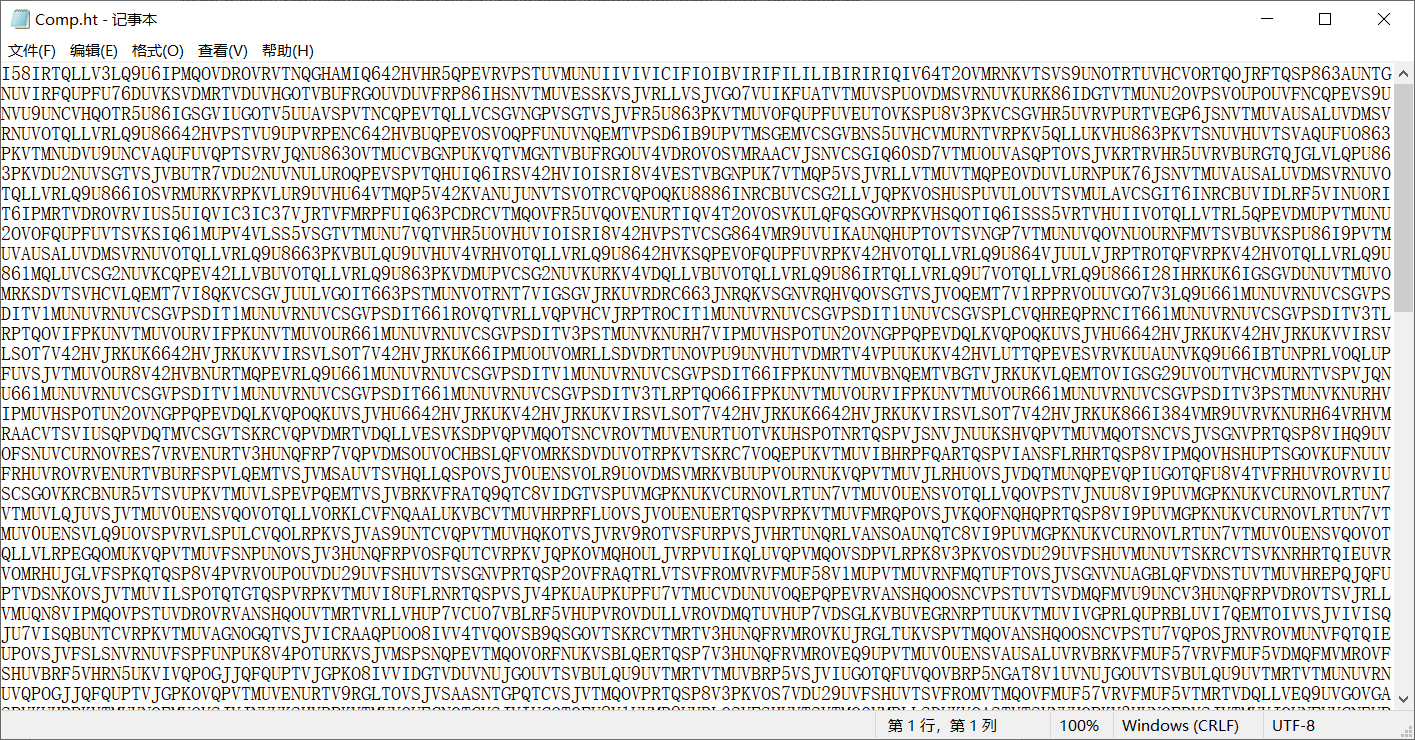
(1)K = 5:



(2)K = 13:



(3)K = 32:



此后译码与实验主体部分效果一致，不再展示。

**五、经验体会与不足**

经验体会：

1、通过测试程序，我认为K叉哈夫曼树的压缩效果和处理效率并不是十分理想（大多数时候不如二叉哈夫曼树）；

2、利用最小堆优化哈夫曼树构造过程是可行的，也是具有一定实际意义的；

3、若将统计字符改为单词，进行相同操作，在某些情况下（文本中单词种类相对较少，单个单词使用频率较高）压缩率较好，另外一些情况下效果不理想。

收获：

1、掌握了哈夫曼树及相关算法的基本思想；

2、通过尝试使用堆优化哈夫曼树的构建，初步理解了堆结构的基本操作和一些特性、优势；

3、通过编写K叉哈夫曼树，深化了对哈夫曼编码、效率及优缺点的认识；

4、利用C++STL及部分库函数优化了代码的部分结构和性能。

不足：

1、进行程序实现前初步构想不完善，导致了编写代码时一些不必要的工作量；

2、程序许多地方还需优化和调整。

**六、附录：源代码（带注释）**

|  |
| --- |
| **程序的全部源代码：** |
| 【实验主体部分：Experiment-2.cpp】   1. /\* 2. \* 实验2 树型结构及其应用（主体部分） 3. \* 实验项目：树形结构及其应用 4. \* 实验题目：哈夫曼编码与译码方法 5. \* 学号：1190200526 6. \* 姓名：沈城有 7. \*/ 8. #include <iostream> 9. #include <fstream> 10. #include <algorithm> 11. #include <map> 12. #include <string> 13. using namespace std; 14. #define InputDir "Input.txt" //英文文本路径 15. #define CompDir "Comp.ht" //压缩文本保存路径（写入+读取） 16. #define DeCompDir "Decomp.txt" //解压文本保存路径（写入） 17. #define N 256 //最大叶子数（字符种类数） 18. #define M 2 \* N - 1 //最大结点总数 19. typedef struct htnode 20. { 21. double weight; 22. int lchild; 23. int rchild; 24. int parent; 25. }HTNode; 26. typedef struct codingtable 27. { 28. int num; 29. char ch; 30. string code; 31. }CodeTab; 32. HTNode HTree[M]; //哈夫曼树静态三叉链表 33. map<char, double> result; //使用STL中的map统计信息 34. CodeTab CT[N]; //存储编码表 35. //功能实现相关函数声明 36. int ReadFromFile(void); //从文件中读取并统计信息 37. void ShowStat(void); //显示统计信息 38. void HTNodeInit(void); //用统计数据初始化生成结点 39. void SelectMin(int n, int& p1, int& p2); //选取两个最小权值 40. void CreateHT(void); //构造哈夫曼树 41. bool cmp(CodeTab a, CodeTab b); //sort所需比较函数 42. void PrintCodingTable(void); //屏幕显示编码表 43. int CompressFile(void); //根据编码生成压缩文件 44. void CalculateRate(int counts, int countt); //计算压缩率 45. bool DecompressFile(void); //压缩文件解码 46. //从文件中读取并统计信息 47. int ReadFromFile(void) 48. { 49. ifstream OpenFile(InputDir); 50. map<char, int> data; //统计字符及频率 51. map<char, int>::iterator it; //查找字符所用迭代器 52. char temp; 53. int countt = 0; 54. double rate; 55. if (!OpenFile) 56. { 57. cout << "File open failed!" << endl; 58. return 0; 59. } 60. while (OpenFile.get(temp)) 61. { 62. if (temp == '\n') 63. temp = '#'; //转换换行符便于显示 64. it = data.find(temp); 65. if (it != data.end()) 66. ++it->second; 67. else 68. data.insert(pair<char, int>(temp, 1)); 69. } 70. OpenFile.close(); 71. for (it = data.begin(); it != data.end(); ++it) 72. countt += it->second; 73. for (it = data.begin(); it != data.end(); ++it) 74. { 75. rate = (double)it->second / (double)countt; 76. result.insert(pair<char, double>(it->first, rate)); 77. } 78. cout << "File read and analysis completed." << endl; 79. return countt; 80. } 81. //显示统计信息 82. void ShowStat(void) 83. { 84. int countn = 0; //总字符种类数 85. map<char, double>::iterator it; 86. cout << "-----------------------------------Statistics-----------------------------------" << endl; 87. for (it = result.begin(); it != result.end(); ++it) 88. { 89. cout << "|" << it->first << "|" << ":" << it->second; 90. cout << "\t"; 91. ++countn; 92. if (countn % 5 == 0) 93. cout << endl; 94. } 95. cout << "\n[# stands for newline.]" << endl; 96. cout << "Number of type:" << countn << endl; 97. } 98. //用统计数据初始化生成结点 99. void HTNodeInit(void) 100. { 101. map<char, double>::iterator it; 102. int i = 0; 103. for (it = result.begin(); it != result.end(); ++it) 104. { 105. HTree[i].weight = it->second; 106. HTree[i].parent = -1; 107. HTree[i].lchild = -1; 108. HTree[i].rchild = -1; 109. ++i; 110. } 111. for (; i < M; ++i) 112. { 113. HTree[i].parent = -1; 114. HTree[i].lchild = -1; 115. HTree[i].rchild = -1; 116. } 117. } 118. //选取两个最小权值 119. void SelectMin(int n, int& p1, int& p2) 120. {//简单遍历实现版 121. int i, j; 122. for (i = 0; i < n; ++i) 123. { 124. if (HTree[i].parent == -1) 125. { 126. p1 = i; 127. break; 128. } 129. } 130. for (j = i + 1; j < n; ++j) 131. { 132. if (HTree[j].parent == -1) 133. { 134. p2 = j; 135. break; 136. } 137. } 138. for (i; i < n; ++i) 139. { 140. if ((HTree[p1].weight > HTree[i].weight) && (HTree[i].parent == -1) && (p2 != i)) 141. p1 = i; 142. } 143. for (j; j < n; ++j) 144. { 145. if ((HTree[p2].weight > HTree[j].weight) && (HTree[j].parent == -1) && (p1 != j)) 146. p2 = j; 147. } 148. } 149. //构造哈夫曼树 150. void CreateHT(void) 151. { 152. int i, p1, p2; 153. int maxn = result.size(); 154. HTNodeInit(); 155. for (i = maxn; i < 2 \* maxn - 1; ++i) 156. { 157. SelectMin(i, p1, p2); 158. HTree[p1].parent = HTree[p2].parent = i; 159. HTree[i].lchild = p1; 160. HTree[i].rchild = p2; 161. HTree[i].weight = HTree[p1].weight + HTree[p2].weight; 162. } 163. } 164. //sort所需比较函数 165. bool cmp(CodeTab a, CodeTab b) 166. { 167. return HTree[a.num].weight > HTree[b.num].weight; 168. } 169. //屏幕显示编码表 170. void PrintCodingTable(void) 171. { 172. int c, p, i; 173. int max = result.size(); 174. char code[100]; 175. map<char, double>::iterator it; 176. int start; 177. string temp; 178. for (i = 0, it = result.begin(); i < max, it != result.end(); ++i, ++it) 179. { 180. CT[i].ch = it->first; 181. CT[i].num = i; 182. } 183. sort(CT, CT + max, cmp); //按频率降序排序 184. for (i = 0; i < max; ++i) 185. { 186. c = CT[i].num; 187. start = 99; 188. temp = ""; 189. while ((p = HTree[c].parent) >= 0) 190. { 191. code[--start] = (HTree[p].lchild == c) ? '0' : '1'; 192. c = p; //继续上溯; 193. } 194. for (int j = start; j < 99; ++j) 195. { 196. temp += code[j]; 197. } 198. CT[i].code = temp; 199. } 200. for (int k = 1; k <= max; ++k) 201. { 202. cout << "|" << CT[k - 1].ch << "|:" << CT[k - 1].code; 203. cout << " \t"; 204. if (k % 5 == 0 && k) 205. cout << endl; 206. } 207. } 208. //根据编码生成压缩文件 209. int CompressFile(void) 210. { 211. ifstream ReadFile; 212. ofstream WriteFile; 213. char temp; 214. char bin = 0; //用来储存当前的huffman编码 215. int i = 0; //记录处理位数（到8就写一次） 216. int q; 217. int count = 0; 218. ReadFile.open(InputDir); 219. WriteFile.open(CompDir, ios::binary); 220. string::iterator it; 221. map<char, double>::iterator mt; 222. if (!ReadFile || !WriteFile) 223. { 224. cout << "File open failed!" << endl; 225. return 0; 226. } 227. //首先写入编码表等解压缩所需信息 228. WriteFile << result.size(); //字符种类数 229. for (mt = result.begin(); mt != result.end(); ++mt) 230. WriteFile << mt->first; //写入所有种类字符 231. for (unsigned int k = result.size(); k < 2 \* result.size() - 1; ++k) //写入哈夫曼树中lchild、rchild（解码无需回溯故略去parent) 232. WriteFile << (char)HTree[k].lchild << (char)HTree[k].rchild; 233. while(ReadFile.get(temp)) 234. { 235. for (q = 0; q < (int)result.size(); ++q) 236. { 237. if (temp == '\n') 238. temp = '#'; //转换换行符 239. if (CT[q].ch == temp) 240. break; 241. } 242. count += CT[q].code.size(); 243. for(it = CT[q].code.begin(); it != CT[q].code.end(); ++it) 244. {//01串转二进制处理 245. if (\*it == '0') 246. bin = bin << 1; 247. else 248. bin = (bin << 1) + 1; 249. ++i; 250. if(i == 8) 251. { 252. WriteFile << bin; 253. i = 0; 254. bin = 0; 255. } 256. } 257. } 258. q = (8 - i) % 8; //计算末尾补零数 259. bin = bin << q; 260. WriteFile << bin << q; 261. ReadFile.close(); 262. WriteFile.close(); 263. result.clear(); 264. cout << "File compression completed." << endl; 265. return count; 266. } 267. //计算压缩率 268. void CalculateRate(int counts, int countt) 269. { 270. double TextSize; 271. double ZipSize; 272. double crate; 273. double srate; 274. struct \_stat textinfo, zipinfo; 275. \_stat(InputDir, &textinfo); 276. \_stat(CompDir, &zipinfo); 277. TextSize = textinfo.st\_size; 278. ZipSize = zipinfo.st\_size; 279. crate = ZipSize / TextSize \* 100; 280. srate = (double)counts / 8 / countt \* 100; 281. cout << "The size of original file:" << TextSize << endl; 282. cout << "The size of compressed file:" << ZipSize << endl; 283. cout << "Compress rate:" << crate << "%" << endl; //真实文件压缩率 284. cout << "Encode rate:" << srate << "%" << endl; //编码率 285. } 286. //压缩文件解码 287. bool DecompressFile(void) 288. { 289. ifstream ReadFile; 290. ofstream WriteFile; 291. int count; //读入字符类型数 292. int flag = -1; 293. int n; //保存在末尾补的0 294. char b1, b2, b3; //储存从文件中读取的数,检测是否到最后一个需要处理补0 295. char test[5]; //检测文件末尾（似乎可以不用） 296. int i = 0; //位数计数 297. int end; //保存哈夫曼树结点总数,并作为中间变量参与处理 298. ReadFile.open(CompDir, ios::binary); 299. WriteFile.open(DeCompDir); 300. if (!ReadFile || !WriteFile) 301. { 302. cout << "File open failed!" << endl; 303. return false; 304. } 305. //首先读入编码表等解压缩所需信息 306. ReadFile >> count; 307. ReadFile >> noskipws; //避免忽略空格 308. for (int k = 0; k < count; ++k) 309. ReadFile >> CT[k].ch; 310. end = 2 \* count - 2; 311. for (int k = count; k <= end; ++k) 312. { 313. ReadFile >> b1 >> b2; 314. HTree[k].lchild = (int)b1; 315. HTree[k].rchild = (int)b2; 316. } 317. ReadFile >> b1 >> b2 >> b3; 318. do{ 319. if (!((b1 >> (7 - i)) & 1)) 320. end = HTree[end].lchild; 321. else 322. end = HTree[end].rchild; 323. ++i; 324. if (end < 0) 325. return false; 326. if (end < count) 327. { 328. if (CT[end].ch != '#') 329. WriteFile << CT[end].ch; 330. else 331. WriteFile << "\n"; 332. end = 2 \* count - 2; 333. } 334. if (i == 8) 335. { 336. i = 0; 337. b1 = b2; 338. b2 = b3; 339. if (flag >= 0) 340. { 341. b3 = test[flag]; 342. flag = -1; 343. } 344. else if (!ReadFile.get(b3)) 345. { 346. for (int k = 0; k < 5; ++k) 347. { 348. if (ReadFile.get(test[k])) 349. flag = k; 350. } 351. if (flag == -1) 352. break; 353. } 354. } 355. }while(1); 356. n = b2 - '0'; 357. if (end < 0) 358. return false; 359. while (i <= 7 - n) 360. { 361. if (!((b1 >> (7 - i)) & 1)) 362. end = HTree[end].lchild; 363. if ((b1 >> (7 - i)) & 1) 364. end = HTree[end].rchild; 365. ++i; 366. if (end < count) 367. { 368. if (CT[end].ch != '#') 369. WriteFile << CT[end].ch; 370. else 371. WriteFile << endl; 372. end = 2 \* count - 2; 373. } 374. } 375. ReadFile.close(); 376. WriteFile.close(); 377. cout << "File decompress completed." << endl; 378. return true; 379. } 380. int main(void) 381. { 382. int countt; //记录字符总数 383. int counts; //记录压缩后长度 384. char ch; 385. cout << "------------------------------Huffman File Process------------------------------" << endl; 386. cout << "[1] File Compress\n" << "[2] File Decompress" << endl; 387. cout << "Input your choice(other keys to exit):"; 388. ch = getchar(); 389. if (ch == '1') 390. { 391. countt = ReadFromFile(); 392. if (!countt) 393. { 394. cout << "Illegal operation or empty file!" << endl; 395. return 0; 396. } 397. ShowStat(); 398. cout << "Total characters:" << countt << endl; 399. system("pause"); 400. CreateHT(); 401. cout << "-----------------------------------Code Table-----------------------------------" << endl; 402. PrintCodingTable(); 403. cout << endl; 404. system("pause"); 405. counts = CompressFile(); 406. if (!counts) 407. { 408. cout << "Error:File compress failed!" << endl; 409. return 0; 410. } 411. CalculateRate(counts, countt); 412. system("pause"); 413. } 414. else if (ch == '2') 415. { 416. if (!DecompressFile()) 417. cout << "Error:File decompress failed!" << endl; 418. system("pause"); 419. } 420. return 0; 421. }   【实验思考部分：Experiment-2(2).cpp】   1. /\* 2. \* 实验2 树型结构及其应用（思考部分） 3. \* 实验项目：树形结构及其应用 4. \* 实验题目：哈夫曼编码与译码方法 5. \* 学号：1190200526 6. \* 姓名：沈城有 7. \*/ 8. #include <iostream> 9. #include <fstream> 10. #include <algorithm> 11. #include <map> 12. #include <string> 13. #include <cmath> 14. using namespace std; 15. #define InputDir "Input.txt" //英文文本路径 16. #define CompDir "Comp.ht" //编码文本保存路径 17. #define N 256 //最大叶子数（字符种类数） 18. #define Max\_K 32 //最大k值 19. #define M Max\_K \* N - 1 //最大结点总数 20. //最小堆数据结构实现 21. typedef struct heap 22. { 23. int data[M + 1][2]; //编号+频率 24. int n; 25. }Heap; 26. //K叉树孩子链表 27. typedef struct childnode 28. { 29. int child; 30. childnode\* next; 31. }HTC; 32. //K叉哈夫曼树结构定义 33. typedef struct K\_htnode 34. { 35. int parent; 36. int freq; 37. HTC head; //孩子链表表头 38. }HTNode; 39. //字符编码表 40. typedef struct codingtablec 41. { 42. int num; 43. char ch; 44. string code; 45. }CodeTab\_C; 46. //单词编码表 47. typedef struct codingtables 48. { 49. int num; 50. string word; 51. string ch; 52. }CodeTab\_S; 53. //数据结构相关基本操作声明 54. void InitHeap(Heap& heap); 55. bool HeapEmpty(Heap heap); 56. bool HeapFull(Heap heap); 57. void HeapInsert(Heap& heap, int num, int freq); 58. void HeapDeleteMin(Heap& heap, int& num); 59. //功能实现相关函数声明 60. int ReadFromFile(char ch); //从文件中读取并统计信息 61. void CreateHeap(char ch, Heap& heap); //创建初始堆 62. HTC\* HTCNodeInit(int k); //初始化K叉树孩子链表 63. int HTNodeInit(char ch); //用统计数据初始化生成结点 64. void SelectMin(Heap& heap, int k, int num[]); //选取k个最小权值（利用堆） 65. void CreateHT(Heap heap, char ch, int& k); //构造哈夫曼树 66. void ProcessCodeTable(char ch); //根据哈夫曼树生成并保存编码表 67. void ShowInfo(char ch); //显示统计信息及编码表 68. void EncodeFile(void); //根据编码生成压缩文件 69. void CalculateRate(int countt, int k); //计算压缩率 70. void InitHeap(Heap& heap) 71. { 72. heap.n = 0; 73. } 74. bool HeapEmpty(Heap heap) 75. { 76. return (!heap.n); 77. } 78. bool HeapFull(Heap heap) 79. { 80. return (heap.n == N - 1); 81. } 82. void HeapInsert(Heap& heap, int num, int freq) 83. { 84. int i; 85. if (!HeapFull(heap)) 86. { 87. i = ++heap.n; 88. while ((i != 1) && (freq < heap.data[i / 2][1])) 89. { 90. heap.data[i][0] = heap.data[i / 2][0]; 91. heap.data[i][1] = heap.data[i / 2][1]; 92. i /= 2; 93. } 94. } 95. heap.data[i][0] = num; 96. heap.data[i][1] = freq; 97. } 98. void HeapDeleteMin(Heap& heap, int& num) 99. { 100. int parent = 1, child = 2; 101. int temp[2]; 102. if (!HeapEmpty(heap)) 103. { 104. num = heap.data[1][0]; 105. temp[0] = heap.data[heap.n][0]; 106. temp[1] = heap.data[heap.n--][1]; 107. while (child <= heap.n) 108. { 109. if ((child < heap.n) && (heap.data[child][1] > heap.data[child + 1][1])) 110. ++child; 111. if (temp[1] <= heap.data[child][1]) 112. break; 113. heap.data[parent][0] = heap.data[child][0]; 114. heap.data[parent][1] = heap.data[child][1]; 115. parent = child; 116. child \*= 2; 117. } 118. } 119. else 120. { 121. cout << "Heap is empty!" << endl; 122. return; 123. } 124. heap.data[parent][0] = temp[0]; 125. heap.data[parent][1] = temp[1]; 126. } 127. HTNode HTree[M]; //K叉哈夫曼树 128. map<char, int> resultc; //统计字符用 129. map<string, int> results; //统计单词用 130. CodeTab\_C CT\_C[N]; //存储字符编码表 131. CodeTab\_S CT\_S[N]; //存储单词编码表 132. //从文件中读取并统计信息 133. int ReadFromFile(char ch) 134. { 135. ifstream OpenFile(InputDir); 136. char temp; 137. int countt = 0; 138. if (ch == '1') 139. { 140. map<char, int>::iterator it; 141. if (!OpenFile) 142. { 143. cout << "File open failed!" << endl; 144. return 0; 145. } 146. while (OpenFile.get(temp)) 147. { 148. if (temp == '\n') 149. temp = '#'; //转换换行符便于显示 150. it = resultc.find(temp); 151. if (it != resultc.end()) 152. ++it->second; 153. else 154. resultc.insert(pair<char, int>(temp, 1)); 155. } 156. OpenFile.close(); 157. for (it = resultc.begin(); it != resultc.end(); ++it) 158. countt += it->second; 159. cout << "File read and analysis completed." << endl; 160. return countt; 161. } 162. return 0; 163. /\*else 164. { 165. map<string, int>::iterator it; 166. string s; 167. int flag = 0; 168. if (!OpenFile) 169. { 170. cout << "File open failed!" << endl; 171. return 0; 172. } 173. while (OpenFile.get(temp)) 174. { 175. if (flag && s.size() != 0) 176. { 177. it = results.find(s); 178. if (it != results.end()) 179. ++it->second; 180. else 181. results.insert(pair<string, int>(s, 1)); 182. } 183. if (temp == '\n') 184. flag = 1; 185. else if(temp == ' ') 186. } 187. OpenFile.close(); 188. cout << "File read and analysis completed." << endl; 189. return countt; 190. }\*/ //单词部分 191. } 192. //创建初始堆 193. void CreateHeap(char ch, Heap& heap) 194. { 195. if (ch == '1') 196. { 197. map<char, int>::iterator it; 198. int max = resultc.size(); 199. int i; 200. for (i = 0, it = resultc.begin(); i < max, it != resultc.end(); ++i, ++it) 201. { 202. CT\_C[i].ch = it->first; 203. CT\_C[i].num = i; 204. } 205. for (i = 0; i < max; ++i) 206. { 207. it = resultc.find(CT\_C[i].ch); 208. HeapInsert(heap, CT\_C[i].num, it->second); 209. } 210. } 211. else 212. { 213. /// 214. } 215. } 216. //初始化K叉树孩子链表 217. HTC\* HTCNodeInit(int k) 218. { 219. HTC\* pcur; 220. HTC\* q; 221. q = (HTC\*)malloc(sizeof(HTC)); //记录第一个结点 222. if (q == NULL) 223. { 224. cout << "Memory allocation failed!" << endl; 225. return NULL; 226. } 227. q->child = -1; 228. pcur = q; 229. for (int i = 2; i <= k; ++i) 230. { 231. pcur->next = (HTC\*)malloc(sizeof(HTC)); 232. if (pcur->next == NULL) 233. { 234. cout << "Memory allocation failed!" << endl; 235. return NULL; 236. } 237. pcur->next->child = -1; 238. pcur = pcur->next; 239. } 240. pcur->next = NULL; 241. return q; 242. } 243. //用统计数据初始化生成结点 244. int HTNodeInit(char ch) 245. { 246. int k; 247. do { 248. cout << "Input k" << "(<=" << Max\_K << "):"; 249. cin >> k; 250. } while (k > Max\_K); 251. if (ch == '1') 252. { 253. map<char, int>::iterator it; 254. int i = 0; 255. for (it = resultc.begin(); it != resultc.end(); ++it) 256. { 257. HTree[i].freq = it->second; 258. HTree[i].parent = -1; 259. HTree[i].head.next = HTCNodeInit(k); 260. ++i; 261. } 262. for (; i < M; ++i) 263. { 264. HTree[i].parent = -1; 265. HTree[i].head.next = HTCNodeInit(k); 266. } 267. } 268. else 269. { 270. /// 271. } 272. return k; 273. } 274. //选取k个最小权值（利用堆） 275. void SelectMin(Heap& heap, int k, int num[]) 276. { 277. for (int i = 0; i < k; ++i) 278. HeapDeleteMin(heap, num[i]); 279. } 280. //构造哈夫曼树 281. void CreateHT(Heap heap, char ch, int& k) 282. { 283. int i, a; 284. int p[Max\_K]; 285. HTC\* pcur; 286. if (ch == '1') 287. { 288. int maxn = resultc.size(); 289. k = HTNodeInit(ch); 290. CreateHeap(ch, heap); 291. i = maxn; 292. if ((maxn - 1) % (k - 1)) //补充几个无意义的点以保证每次都能取k个 293. { 294. for (int m = 1; m <= (k - 1) - (maxn - 1) % (k - 1); ++m) 295. HeapInsert(heap, -1, 0); 296. } 297. while(heap.n > 1) 298. { 299. SelectMin(heap, k, p); 300. for (a = 0; a < k; ++a) 301. { 302. HTree[p[a]].parent = i; 303. HTree[i].freq += HTree[p[a]].freq; 304. } 305. HeapInsert(heap, i, HTree[i].freq); 306. for (pcur = HTree[i].head.next, a = 0; pcur != NULL; pcur = pcur->next) 307. pcur->child = p[a++]; 308. ++i; 309. } 310. } 311. else 312. { 313. /// 314. } 315. } 316. //sort所需比较函数1 317. bool cmp\_1(CodeTab\_C a, CodeTab\_C b) 318. { 319. return HTree[a.num].freq > HTree[b.num].freq; 320. } 321. //sort所需比较函数2 322. bool cmp\_2(CodeTab\_S a, CodeTab\_S b) 323. { 324. return HTree[a.num].freq > HTree[b.num].freq; 325. } 326. //根据哈夫曼树生成并保存编码表 327. void ProcessCodeTable(char ch) 328. { 329. int c, p, i; 330. int max, count = -1; 331. int start; 332. char code[100]; 333. HTC\* pcur; 334. string temp; 335. if (ch == '1') 336. { 337. max = resultc.size(); 338. map<char, int>::iterator it; 339. sort(CT\_C, CT\_C + max, cmp\_1); 340. for (i = 0; i < max; ++i) 341. { 342. c = CT\_C[i].num; 343. start = 99; 344. temp = ""; 345. while ((p = HTree[c].parent) >= 0) 346. { 347. pcur = HTree[p].head.next; 348. while (pcur != NULL) 349. { 350. ++count; 351. if (pcur->child == c) 352. break; 353. pcur = pcur->next; 354. } 355. if (count <= 9) 356. code[--start] = '0' + count; 357. else 358. code[--start] = 'A' + count - 10; 359. count = -1; 360. c = p; //继续上溯; 361. } 362. for (int j = start; j < 99; ++j) 363. { 364. temp += code[j]; 365. } 366. CT\_C[i].code = temp; 367. } 368. } 369. else 370. { 371. /// 372. } 373. } 374. //显示统计信息 375. void ShowInfo(char ch) 376. { 377. if (ch == '1') 378. { 379. map<char, int>::iterator it; 380. cout << "Character\t" << "Frequency\t" << "Huffman Code" << endl; 381. for (unsigned int i = 0; i < resultc.size(); ++i) 382. { 383. cout << CT\_C[i].ch << "\t\t"; 384. it = resultc.find(CT\_C[i].ch); 385. cout << it->second << "\t\t"; 386. cout << CT\_C[i].code << endl; 387. } 388. cout << "Number of type:" << resultc.size() << endl; 389. } 390. else 391. { 392. /// 393. } 394. } 395. //根据编码生成压缩文件 396. void EncodeFile(void) 397. { 398. ifstream ReadFile; 399. ofstream WriteFile; 400. char temp; 401. int q; 402. int count = 0; 403. ReadFile.open(InputDir); 404. WriteFile.open(CompDir); 405. string::iterator it; 406. map<char, double>::iterator mt; 407. if (!ReadFile || !WriteFile) 408. { 409. cout << "File open failed!" << endl; 410. return; 411. } 412. while (ReadFile.get(temp)) 413. { 414. if (temp == '\n') 415. temp = '#'; //转换换行符 416. for (q = 0; q < (int)resultc.size(); ++q) 417. { 418. if (CT\_C[q].ch == temp) 419. break; 420. } 421. WriteFile << CT\_C[q].code; 422. } 423. ReadFile.close(); 424. WriteFile.close(); 425. cout << "File encode completed." << endl; 426. } 427. //计算压缩率 428. void CalculateRate(int countt, int k) 429. { 430. double rate; 431. int count = 0; 432. int encode = ceil(log2(1.0 \* k)); 433. map<char, int>::iterator it; 434. string::iterator its; 435. for (unsigned int i = 0; i < resultc.size(); ++i) 436. { 437. it = resultc.find(CT\_C[i].ch); 438. count += encode \* it->second \* CT\_C[i].code.size(); 439. } 440. rate = (double)count / 8 / (double)countt \* 100; 441. cout << "Encode rate:" << rate << "%" << endl; 442. } 443. int main(void) 444. { 445. int countt, k; 446. char ch = '1'; 447. Heap heap; 448. InitHeap(heap); 449. /\*cout << "[1] Character\t[2] Word" << endl; 450. cout << "Input choice:"; 451. cin >> ch;\*/ 452. countt = ReadFromFile(ch); 453. if (!countt) 454. { 455. cout << "Illegal operation or empty file!" << endl; 456. return 0; 457. } 458. CreateHT(heap, ch, k); 459. ProcessCodeTable(ch); 460. ShowInfo(ch); 461. cout << "Total characters:" << countt << endl; 462. system("pause"); 463. EncodeFile(); 464. CalculateRate(countt, k); 465. system("pause"); 466. return 0; 467. } |