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：专业核心基础课（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： 2023/10/10

班级：2022112266

学号：2022112266

姓名：魏圣卓

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

**一、实验目的**

通过哈夫曼编码的实现，加深对树形结构的理解和运用，同时掌握一种压缩数据的方式，对编码和译码的过程有初步的了解

**二、实验要求及实验环境**

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。

**实验环境：VS Code mingw64**

**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．逻辑设计 树形结构（二叉树）

2．物理设计（即存储结构设计） 静态链表、结构体、vector容器

class TreeNode {

public:

    char ch;

    int count;

    bool isChar;

    TreeNode\* left;

    TreeNode\* right;

    // 构造函数

    TreeNode(char ch, int count, bool isChar)

        : ch(ch)

        , count(count)

        , isChar(isChar)

        , left(nullptr)

        , right(nullptr) {};

    TreeNode() { }

};

主要算法：

a.建立哈夫曼树：

首先将每节点建树，形成森林；然后找出频率最小的两棵树形成一棵新树，其根节点的权值为它们的频率之和，左右孩子指向这两棵树/节点，以此类推，直到只剩下一棵树，即为哈夫曼树。

找出最小的两棵树，这里用了两次遍历，分别找出第一小和第二小。

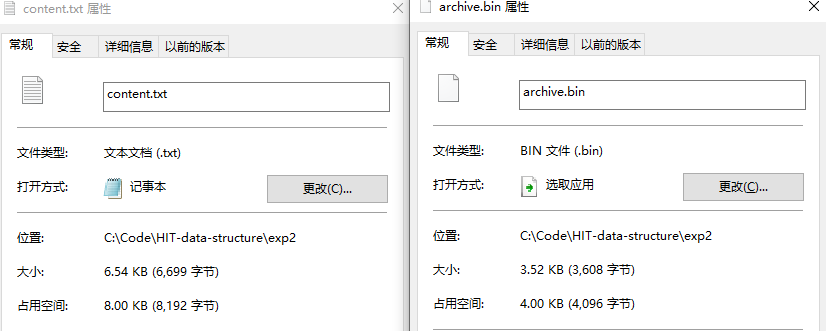
b.编码：

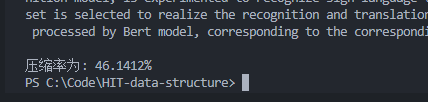
这里采用由根节点递归的方式，每往左遍历，编码+“0”；每往右遍历，编码+“1”,直到遍历到叶节点。

**四、测试结果**（包括**测试数据、结果数据及结果的简单分析和结论，可以用截图得形式贴入此报告**）

原文6700字符，大小6699字节

压缩后3608字节





**五、经验体会与不足**

在构建编码表后，字符、频率及编码表仍是存在内存变量中，没有写入到文件里，后续还要做到将哈夫曼树写进二进制文件里

写入二进制的按位打包很好用，以后也可以多复用此算法

**六、附录：源代码（带注释）**

1. #include <fstream>

2. #include <iostream>

3. #include <map>

4. #include <math.h>

5. #include <queue>

6. #include <stack>

7. #include <string>

8. #include <vector>

9. using namespace std;

10.

11. class TreeNode {

12. public:

13.     char ch;

14.     int count;

15.     bool isChar;

16.     TreeNode\* left;

17.     TreeNode\* right;

18.

19.     // 构造函数

20.     TreeNode(char ch, int count, bool isChar)

21.         : ch(ch)

22.         , count(count)

23.         , isChar(isChar)

24.         , left(nullptr)

25.         , right(nullptr) {};

26.     TreeNode() { }

27. };

28.

29. class BTree {

30. private:

31.     TreeNode\* root;

32.

33. public:

34.     BTree()

35.     {

36.         // root = new TreeNode('#');

37.         root = nullptr;

38.     }

39.     void destroy(TreeNode\* node)

40.     {

41.         if (node != nullptr) {

42.             destroy(node->left);

43.             destroy(node->right);

44.             delete node;

45.         }

46.     }

47.     ~BTree()

48.     {

49.         destroy(root);

50.     }

51.

52.     TreeNode\* getroot()

53.     {

54.         return root;

55.     }

56. };

57.

58. void Huffman\_str(TreeNode\* curr, string str, map<char, string>& m)

59. {

60.     if (curr == nullptr) {

61.         return;

62.     }

63.     if (curr->isChar) {

64.         m[curr->ch] = str;

65.     }

66.     Huffman\_str(curr->left, str + '0', m);

67.     Huffman\_str(curr->right, str + '1', m);

68. }

69.

70. void PrintTree(TreeNode\* n, bool left, string const& indent)

71. {

72.     if (n->right) {

73.         PrintTree(n->right, false, indent + (left ? "|     " : "      "));

74.     }

75.     cout << indent;

76.     // 判断当前节点为左节点还是右节点。然后使用/或者\表示左右节点

77.     cout << (left ? '\\' : '/');

78.     // 打印一个当前节点的数据 一定是序号换行的

79.     cout << "-----" << n->ch << endl;

80.     if (n->left) {

81.         PrintTree(n->left, true, indent + (left ? "      " : "|     "));

82.     }

83. }

84.

85. void PrintTree(TreeNode\* root)

86. {

87.     cout << "图形化打印二叉树" << endl;

88.     cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n"

89.          << endl;

90.     if (root->right) {

91.         PrintTree(root->right, false, "");

92.     }

93.     cout << root->ch << endl;

94.     if (root->left) {

95.         PrintTree(root->left, true, "");

96.     }

97.     cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n"

98.          << endl;

99. }

100. void PrintTree(TreeNode\* root, int indent = 0)

101. {

102.     if (root == nullptr) {

103.         return;

104.     }

105.

106.     // 增加缩进，以表示树的层级结构

107.     indent += 4;

108.

109.     // 递归打印右子树

110.     PrintTree(root->right, indent);

111.

112.     // 打印当前节点，带有缩进

113.     for (int i = 4; i < indent; i++) {

114.         std::cout << " ";

115.     }

116.     std::cout << root->count << std::endl;

117.

118.     // 递归打印左子树

119.     PrintTree(root->left, indent);

120. }

121.

122. void readfile(string filename, string& content)

123. {

124.     ifstream file(filename); // 打开文件

125.

126.     if (!file.is\_open()) {

127.         cerr << "无法打开文件" << endl;

128.     }

129.

130.     string line;

131.

132.     // 读取第一行内容

133.     if (getline(file, line)) {

134.         content = line;

135.         // cout << "第一行内容: " << line << endl;

136.     } else {

137.         // cerr << "文件为空" << endl;

138.     }

139.

140.     file.close(); // 关闭文件

141. }

142. void readBfile(string filename, string& content)

143. {

144.     std::ifstream file(filename, std::ios::binary);

145.     if (!file.is\_open()) {

146.         cerr << "无法打开文件" << endl;

147.     }

148.

149.     string line = "";

150.

151.     // 读取第一行内容

152.     char bit;

153.     while (file.get(bit)) {

154.         for (int i = 7; i >= 0; --i) {

155.             // 使用位运算检查并提取每个位的值

156.             char t = (bit & (1 << i)) ? '1' : '0';

157.             line += t;

158.         }

159.     }

160.     file.close(); // 关闭文件

161.     int NULLCount=0;

162.     string last8Characters = line.substr(line.length() - 8);

163.     int binaryBase = 1; // 二进制的基数每次乘2

164.

165.     // 从右到左遍历二进制字符串

166.     for (int i = last8Characters.length() - 1; i >= 0; --i) {

167.         if (last8Characters[i] == '1') {

168.             NULLCount += binaryBase;

169.         }

170.         binaryBase \*= 2; // 更新二进制的基数

171.     }

172.

173.     NULLCount+=8;

174.     content=line.substr(0, line.length() - NULLCount);

175. }

176. int getarrcnt(TreeNode\* chars[], int size)

177. {

178.     int cnt = 0;

179.     for (int i = 0; i < size; i++) {

180.         if (chars[i] != nullptr) {

181.             cnt++;

182.         }

183.     }

184.     return cnt;

185. }

186.

187. void getmin(TreeNode\* chars[], int size, TreeNode\*& min1, TreeNode\*& min2)

188. {

189.     for (int i = 0; i < size; i++) {

190.

191.         if (chars[i] == nullptr) {

192.             continue;

193.         }

194.         if (min1 == nullptr) {

195.             min1 = chars[i];

196.         } else if (chars[i]->count < min1->count) {

197.             min1 = chars[i];

198.         }

199.     }

200.     for (int i = 0; i < size; i++) {

201.         if (chars[i] == nullptr) {

202.             continue;

203.         }

204.         if (min2 == nullptr && chars[i] != min1) {

205.             min2 = chars[i];

206.         } else if (chars[i]->count < min2->count && chars[i] != min1) {

207.             min2 = chars[i];

208.         }

209.     }

210. }

211.

212. string decode(string archive, TreeNode\* root)

213. {

214.     string content = "";

215.     TreeNode\* curr = root;

216.     for (auto i : archive) {

217.         if (i == '0') {

218.             curr = curr->left;

219.         } else {

220.             curr = curr->right;

221.         }

222.         if (curr->isChar) {

223.             content += curr->ch;

224.             curr = root;

225.             continue;

226.         }

227.     }

228.

229.     return content;

230. }

231.

232. void write\_into\_Bfile(const string& filename, const string& archive)

233. {

234.     ofstream binaryFile(filename, ios::binary | ios::out | ios::trunc); // 打开二进制文件并清空

235.     if (binaryFile.is\_open()) {

236.         vector<char> packedData; // 用于存储打包后的数据

237.

238.         char currentByte = 0; // 当前字节

239.         int bitCount = 0; // 记录当前字节中已经写入的位数

240.

241.         for (size\_t i = 0; i < archive.length(); i++) {

242.             char bit = archive[i];

243.             if (bit == '0') {

244.                 currentByte <<= 1; // 将当前字节左移一位

245.             } else if (bit == '1') {

246.                 currentByte = currentByte << 1;

247.                 currentByte = currentByte | 1; // 在当前字节最低位置1

248.             } else {

249.                 cerr << "无效的字符：" << bit << endl;

250.                 return; // 退出函数以避免写入无效数据

251.             }

252.

253.             bitCount++;

254.

255.             // 如果当前字节已经装满8位，则将其添加到容器并重置

256.             if (bitCount == 8) {

257.                 packedData.push\_back(currentByte);

258.                 currentByte = 0;

259.                 bitCount = 0;

260.             }

261.         }

262.

263.         // 如果还有未装满的字节，将其添加到容器

264.         if (bitCount > 0) {

265.             currentByte <<= (8 - bitCount);

266.             packedData.push\_back(currentByte);

267.         }

268.         packedData.push\_back(8-bitCount);

269.

270.         // 将打包后的数据写入二进制文件

271.         binaryFile.write(packedData.data(), packedData.size());

272.         binaryFile.close();

273.         cout << "二进制文件写入成功" << endl;

274.     } else {

275.         cerr << "无法打开文件" << endl;

276.     }

277. }

278.

279. double calculateCompressionRate(const string& filename, const string& archive)

280. {

281.     // 打开原始文件

282.     ifstream originalFile(filename, ios::binary | ios::ate);

283.     if (!originalFile.is\_open()) {

284.         cerr << "无法打开原始文件" << endl;

285.         return -1.0;

286.     }

287.

288.     // 获取原始文件的大小

289.     streampos originalFileSize = originalFile.tellg();

290.     originalFile.close();

291.

292.     // 打开压缩后的文件

293.     ifstream compressedFile(archive, ios::binary | ios::ate);

294.     if (!compressedFile.is\_open()) {

295.         cerr << "无法打开压缩后的文件" << endl;

296.         return -1.0;

297.     }

298.

299.     // 获取压缩后文件的大小

300.     streampos compressedFileSize = compressedFile.tellg();

301.     compressedFile.close();

302.

303.     // 计算压缩率（百分比）

304.     double compressionRate = (1.0 - static\_cast<double>(compressedFileSize) / static\_cast<double>(originalFileSize)) \* 100.0;

305.

306.     return compressionRate;

307. }

308.

309. int main()

310. {

311.

312.     string content;

313.     readfile("content.txt", content);

314.     cout << "源文本：" << endl

315.          << content << endl;

316.     cout << endl;

317.     map<char, int> charCount;

318.     for (char c : content) {

319.         // 如果字符已经在map中，增加计数，

320.         //否则添加新字符并初始化计数为1

321.         if (charCount.find(c) != charCount.end()) {

322.             charCount[c]++;

323.         } else {

324.             charCount[c] = 1;

325.         }

326.     }

327.

328.     // 打印每个字符和对应的计数

329.     for (const auto& pair : charCount) {

330.         cout << "字符 '" << pair.first << "' 出现了 " << pair.second << " 次" << std::endl;

331.     }

332.     cout << endl;

333.

334.     int size = charCount.size();

335.     TreeNode\* chars[size];

336.     int i = 0;

337.     for (const auto& pair : charCount) {

338.         TreeNode\* curr = new TreeNode(pair.first, pair.second, 1);

339.         chars[i] = curr;

340.         i++;

341.     }

342.

343.     TreeNode\* examplemin = new TreeNode(-1, 99999999, 0);

344.     TreeNode\* min1;

345.     TreeNode\* min2;

346.     TreeNode\* Huffman = nullptr;

347.

348.     while (1) {

349.         if (getarrcnt(chars, size) == 1) {

350.             for (int i; i < size; i++) {

351.                 if (chars[i] != nullptr) {

352.                     Huffman = chars[i];

353.                     break;

354.                 }

355.             }

356.             break;

357.         }

358.         min1 = examplemin;

359.         min2 = examplemin;

360.         getmin(chars, size, min1, min2);

361.         TreeNode\* curr = new TreeNode(-1, (min1->count + min2->count), 0);

362.         curr->left = min1;

363.         curr->right = min2;

364.         for (int i = 0; i < size; i++) {

365.             if (chars[i] == min1) {

366.                 chars[i] = curr;

367.             } else if (chars[i] == min2) {

368.                 chars[i] = nullptr;

369.             }

370.         }

371.     }

372.     // PrintTree(Huffman, 0);

373.

374.     map<char, string> Huffmanmap;

375.

376.     Huffman\_str(Huffman, "", Huffmanmap);

377.

378.     for (const auto& pair : Huffmanmap) {

379.         cout << "字符 '" << pair.first << "' 编码：" << pair.second << std::endl;

380.     }

381.     cout << endl;

382.

383.     string archive = "";

384.     for (auto i : content) {

385.         archive += Huffmanmap[i];

386.     }

387.     cout << "哈夫曼编码：" << endl

388.          << archive << endl;

389.

390.

391.     write\_into\_Bfile("archive.bin", archive);

392.     cout << endl;

393.     string readfromBfile;

394.     readBfile("archive.bin", readfromBfile);

395.     // cout<<readfromBfile<<endl;

396.

397.     // string txt = decode(archive, Huffman);

398.     string txt = decode(readfromBfile, Huffman);

399.     cout << "解码后源文件：" << endl

400.          << txt << endl;

401.     cout << endl;

402.     double compressionRate = calculateCompressionRate("content.txt", "archive.bin");

403.     if (compressionRate >= 0.0) {

404.         std::cout << "压缩率为: " << compressionRate << "%" << std::endl;

405.     }

406.     return 0;

407. }

408.