**报告名称：文件压缩**

**班级：国06**

**学号：2252075**

**姓名：刘文飞**

**完成日期：2023.6.6**

1. **设计思路与功能描述：**
2. **设计思路：本题目旨在完成一个无损文件压缩的小程序，并实现解压功能**
3. **功能描述：**
4. 压缩功能：读入文件内容，核心算法是huffman tree 输出到压缩文件中，压缩比在65%左右，压缩文件中的内容有三部分，第一部分是huffman tree的字符频率信息，第二部分是压缩文件内容，第三部分是有效位数
5. 解压缩功能：读入压缩文件的三部分信息，进行无损的还原到新的输出文件中，核心算法还是huffman tree 。是压缩的逆过程
6. **遇到的问题和解决方法**
7. **压缩：**
8. **总体思路，通过统计字符频率后建立huffman tree,然后对频率不为0的字符进行编码，再次读取文件，对应字符的编码输出到新文件中（将01串二进制转化为char类型写入）**
9. **搞懂huffman tree，本质上是一个满二叉树，通过将出现频率低的字符排在低的位置，（其相应编码更短）将每个字符进行01串的不定长编码。**
10. **如何建树？我的处理方法是根据频率大小建立一个优先列的链表，采用动态申请，频率由小到大，每个节点中存放的是对应频率和树节点的地址，每次排序后建立一个树节点，取出优先列的前两个节点放入其中**
11. **如何编码？采用递归：遍历整个二叉树，向左为0向右为1，每层节点处理下一层，复制本层的编码并添加对应方向的0或1**
12. **为了能够完成解压，我们还需要将huffmantree的信息写入，只需在文件开头顺序写入256个字符的出现频率即可，并在文件尾部写入最后一个字节有效数字的位数**
13. **解压：**
14. 总体思路：实际上是压缩的逆过程：读取压缩文件开头的对应字符频率建立huffmantree,读取文件的压缩内容转换为01串进行对应进行huffman tree的遍历，遍历到叶子节点输出对应字符到文件
15. **心得体会**
16. **文件压缩大作业总的来说是学习了一种算法，并用这种算法进行编码来处理文件的过程，完成的思路很清晰**
17. **在完成本大作业的过程中，进一步体会到了代码函数划分的重要性，比如压缩和解压缩都要建立huffman tree，要把huffman tree的建立单独提取出一个函数来**
18. **不同与之前的作业，本作业debug过程还是比较折磨的，因为数据量大，压缩和解压缩的正确性也不好确定，只有当解压缩后发现文件和原文件一样时才知道自己写的是对的（巨爽），在调试过程中进一步锻炼了debug能力，一个小tips，比如在一个大循环中想要调试是很难直接定位到变量具体是多少的，就可以写一个if（i==某某）判断语句{。。。}，将断点放到大括号语句中就可以直接定位了**
19. **源代码**
20. #define \_CRT\_SECURE\_NO\_WARNINGS
21. #include <fstream>
22. #include<iostream>
23. using namespace std;
24. struct huffman {
25. char ch = '\0';
26. int frq=0;
27. char hcode[80] = { 0 };
28. bool is\_leave = 1;
29. huffman\* left = NULL;
30. huffman\* right = NULL;
31. };/\*创立一个结构体用来作为huffman树的节点，分别存储字符极其出现频率，对应的编码以及左右孩子，以及是否是叶子节点\*/
32. void build\_tree(ifstream& in, huffman\* tree, huffman\* root,int i);
33. void huff\_coding(ifstream& in, ofstream& out, huffman\* tree);
34. void un\_coding(ifstream& in, ofstream& out, huffman\* root);
35. void free\_tree(huffman\* hf);
36. void free\_leaf\_left(huffman\* hf);
37. struct huffman\* t\_createnode();
38. }

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <fstream>

#include<iostream>

#include"file\_compression.h"

#include <chrono>//用来计时

#include <iomanip>

using namespace std;

int main(int argc, char\*\* argv) {

if (argc != 4) {

cout << "错误的指令" << endl;

return 0;

}

if (!(strcmp(argv[3], "zip"))) {

//压缩

auto start = std::chrono::high\_resolution\_clock::now();

ifstream in;

ofstream out;

struct huffman\* tree = new struct huffman[256];

huffman\* root = t\_createnode();

in.open(argv[1], ios::binary);

in.seekg(0, ios::end);

int fileSize1 = in.tellg();

in.seekg(0, std::ios::beg);

build\_tree(in, tree, root, 0);

in.close();

in.open(argv[1], ios::binary);

cout << argv[2];

out.open(argv[2], ios::binary);

huff\_coding(in, out, tree);

in.close();

out.close();

//释放动态内存

free\_leaf\_left(tree);

free\_tree(root);

auto end = std::chrono::high\_resolution\_clock::now();

auto elapsed\_time = std::chrono::duration\_cast<std::chrono::milliseconds>(end - start);

std::cout << "运行时间 " << elapsed\_time.count() << " ms\n";

in.open(argv[2], ios::binary);

in.seekg(0, ios::end);

int fileSize2 = in.tellg();

in.close();

cout << "压缩比为" << fixed << setprecision(2) << (fileSize2 + 0.0) / (fileSize1 + 0.0) \* 100.0 << "%" << endl;

}

if (!(strcmp(argv[3], "unzip"))) {

//解压\*/

auto start = std::chrono::high\_resolution\_clock::now();

ifstream in;

ofstream out;

struct huffman\* tree = new struct huffman[256];

huffman\* root = t\_createnode();

in.open(argv[1], ios::binary);

out.open(argv[2], ios::binary);

build\_tree(in, tree, root, 1);

un\_coding(in, out, root);

in.close();

out.close();

cout << "解压结束" << endl;

//释放动态内存

free\_leaf\_left(tree);

free\_tree(root);

auto end = std::chrono::high\_resolution\_clock::now();

auto elapsed\_time = std::chrono::duration\_cast<std::chrono::milliseconds>(end - start);

std::cout << "运行时间 " << elapsed\_time.count() << " ms\n";

}

return 0;

1. }
2. #define \_CRT\_SECURE\_NO\_WARNINGS
3. #include <fstream>
4. #include<iostream>
5. #include"file\_compression.h"
6. using namespace std;
7. struct line {
8. huffman\* hf;
9. line\* next;
10. };/\*创立一个列来作为优先级队列，根据huffman的frq来排序，从小到大\*/
11. //建树函数
12. void sort(struct line\* head) {
13. struct line\* t1 = NULL;
14. struct line\* t2 = NULL;
15. huffman\* mid1;
16. t1 = head;
17. while (t1->next != NULL) {
18. t2 = t1->next;
19. if ((t1->hf->frq) > (t2->hf->frq)) {
20. mid1 = t1->hf;
21. t1->hf = t2->hf;
22. t2->hf = mid1;
23. }
24. t1 = t2;
25. }
26. }
27. struct line\* l\_createnode() {
28. struct line\* pnode = new(nothrow) line;
29. if (pnode != NULL)
30. {
31. pnode->next = NULL;
32. }
33. return pnode;
34. }
35. struct huffman\* t\_createnode() {
36. struct huffman\* pnode = new(nothrow) huffman;
37. if (pnode != NULL)
38. {
39. pnode->left = NULL;
40. pnode->right = NULL;
41. pnode->is\_leave = 0;
42. }
43. return pnode;
44. }
45. /\*压缩统计\*/
46. void count(ifstream& in, huffman\* t) {
47. char buffer[1024];
48. while (in && !in.eof()) {
49. in.read(buffer, 1024);
50. int readnum = int(in.gcount());
51. for (int i = 0; i < readnum; i++) {
52. t[buffer[i] + 128].frq++;
53. t[buffer[i] + 128].ch = buffer[i];
54. t[buffer[i] + 128].left = NULL;
55. t[buffer[i] + 128].right = NULL;
56. }/\*统计了所有字符出现的次数\*/
57. }
58. }
59. /\*解压统计\*/
60. void gethuff(ifstream& in, huffman\* t) {
61. for (int i = -128; i <= 127; i++) {
62. t[i + 128].ch = char(i);
63. t[i + 128].left = NULL;
64. t[i + 128].right = NULL;
65. in >> t[i + 128].frq;
66. }
67. }
68. void h\_code(huffman\* hf) {
69. if (hf->left != NULL) {
70. char left[] = "0";
71. char right[] = "1";
72. strcat(hf->left->hcode, hf->hcode);
73. strcat(hf->right->hcode, hf->hcode);
74. strcat(hf->left->hcode, left);
75. strcat(hf->right->hcode, right);
76. h\_code(hf->left);
77. h\_code(hf->right);
78. }
79. }//该递归函数作用是将最初的256个中非零频率的字符进行编码,在最底层获得其字符对应编码
80. void build\_tree(ifstream& in, huffman\* tree, huffman\* root, int i) {
81. if (i == 0)
82. count(in, tree);
83. if (i == 1)
84. gethuff(in, tree);
85. struct line\* l\_head = NULL;
86. struct line\* t1 = NULL;
87. struct line\* t2 = NULL;
88. l\_head = l\_createnode();
89. l\_head->hf = NULL;
90. l\_head->next = NULL;
91. t1 = l\_head;
92. for (int i = 0; i < 256; i++)
93. {
94. if (tree[i].frq == 0)
95. continue;
96. if ((tree[i].frq != 0) && (l\_head->hf == NULL)) {
97. l\_head->hf = &tree[i];
98. continue;
99. }
100. t2 = l\_createnode();
101. t2->hf = &tree[i];
102. t1->next = t2;
103. t1 = t2;
104. }/\*循环结束，各种频率非0字符节点地址已经放入队列中\*/
105. //先排序（根据fre）从小到大
106. sort(l\_head);
107. //下面构建树
108. while (!(l\_head->next->next == NULL)) {
109. //生成一个新的huffman节点用于合并前两项,并将其地址添加到优先列中
110. struct huffman\* t1 = NULL;
111. struct line\* t2 = NULL;
112. t1 = t\_createnode();
113. t1->ch = '\0';
114. t1->frq = l\_head->hf->frq + l\_head->next->hf->frq;
115. t1->left = l\_head->hf;
116. t1->right = l\_head->next->hf;
117. t2 = l\_head->next->next;
118. delete l\_head->next;
119. l\_head->next = t2;
120. l\_head->hf = t1;
121. sort(l\_head);
122. }
123. //循环结束后，优先列中还剩两个节点
124. root->ch = '\0';
125. root->frq = l\_head->hf->frq + l\_head->next->hf->frq;
126. root->left = l\_head->hf;
127. root->right = l\_head->next->hf;
128. delete l\_head->next;
129. delete l\_head;//链表的动态内存同时也释放完毕
130. //至此huffman树建立完毕
131. h\_code(root);
132. }
133. //文件编码(压缩）
134. char binaryTochar(char\* str) {
135. char result = 0;
136. for (int i = 0; i < 8; i++) {
137. if (str[i] == '1') {
138. result |= 1 << (7 - i);
139. }
140. }
141. return result;
142. }
143. void huff\_coding(ifstream& in, ofstream& out, huffman\* tree) {
144. for (int i = 0; i < 256; i++) {
145. out << tree[i].frq << ' ';
146. }/\*在文本开头写入huffmantree的信息\*/
147. char buffer1[1024];
148. char buffer2[40] = {};
149. while (in && !in.eof()) {
150. in.read(buffer1, 1024);
151. int num = int(in.gcount());
152. for (int i = 0; i < num; i++) {
153. strcat(buffer2, tree[buffer1[i] + 128].hcode);
154. if (strlen(buffer2) >= 8) {
155. out << binaryTochar(buffer2);/\*按字节以char输出到文件里\*/
156. for (int j = 8; j < 30; j++) {
157. buffer2[j - 8] = buffer2[j];
158. }
159. }
160. if (in.eof() && i == num - 1) {
161. int len = int(strlen(buffer2));
162. if (strlen(buffer2) != 0) {
163. for (int j = int(strlen(buffer2)); j < 8; j++) {
164. buffer2[j] = '0';
165. }
166. out << binaryTochar(buffer2);/\*按字节以char输出到文件里\*/
167. }
168. out << len;
169. }//最后一个字节可能不足8位，我们可以补齐8位并且将最后一个字节的有效位数输出到文件中，解码时进行相应处理
170. }
171. }
172. }
173. //解压
174. void charTobinary(char c, char\* buffer) {
175. for (int i = 7; i >= 0; i--) {
176. buffer[7 - i] = (char)((c >> i) & 1) + 48; // 通过位运算输出二进制表示
177. }
178. }
179. void un\_coding(ifstream& in, ofstream& out, huffman\* root) {
180. struct huffman\* tool = root;
181. char buffer1[1024];
182. char buffer2[8];
183. int n = 0;
184. in.seekg(0, std::ios::end);
185. // 向前移动一个字符位置
186. in.seekg(-1, std::ios::cur);
187. // 读取最后一个字符
188. char c = in.get();
189. n = (c - 48);/\*获取有效位数\*/
190. in.seekg(0, ios::beg); // 将指针移动到文件开头
191. in.clear();
192. for (int i = -128; i <= 127; i++) {
193. int a;
194. in >> a;
195. }
196. in.ignore();/\*忽略掉空格\*/
197. while (in && !in.eof()) {
198. in.read(buffer1, 1024);
199. int readnum = int(in.gcount());
200. //一种特殊情况：文件没有到结尾，恰好把有效位数独立在外，那么最后一个字符的解压会出现问题
201. //加条件，判断文件指针是否到达了倒数第一个字符,peek一下
202. for (int i = 0; i < readnum; i++) {
203. if ((i == readnum - 2) && ((in.eof()) || (in.peek() == c))) {
204. charTobinary(buffer1[i], buffer2);
205. for (int k = 0; k < n; k++) {
206. if (buffer2[k] == '0') {
207. tool = tool->left;
208. if (tool->left == NULL) {
209. out << tool->ch;
210. tool = root;
211. }
212. continue;
213. }
214. if (buffer2[k] == '1') {
215. tool = tool->right;
216. if (tool->left == NULL) {
217. out << tool->ch;
218. tool = root;
219. }
220. continue;
221. }
222. }
223. break;//解压结束
224. }
225. charTobinary(buffer1[i], buffer2);
226. for (int k = 0; k < 8; k++) {
227. if (buffer2[k] == '0') {
228. tool = tool->left;
229. if (tool->left == NULL) {
230. out << tool->ch;
231. tool = root;
232. }
233. continue;
234. }
235. if (buffer2[k] == '1') {
236. tool = tool->right;
237. if (tool->left == NULL) {
238. out << tool->ch;
239. tool = root;
240. }
241. continue;
242. }
243. }
244. }
245. }
246. }
247. /\*别忘了动态内存申请的释放\*/
248. //申请的动态内存有，256个节点，huffman树的新节点，以及优先队列（单链表）
249. //链表的动态内存在使用时已经同步释放，只需关注huffman tree的节点释放
250. //先释放非叶子节点，再释放叶子
251. void free\_tree(huffman\* hf) {
252. if (hf->is\_leave) {
253. return;
254. }//终止条件
255. free\_tree(hf->left); // 先释放左子树内存
256. free\_tree(hf->right); // 再释放右子树内存
257. delete hf; // 最后释放当前节点内存
258. }
259. void free\_leaf\_left(huffman\* tree) {
260. delete[]tree;
261. }