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：2020.11.01

班级：1903002

学号：1190200523

姓名：石翔宇

|  |  |  |
| --- | --- | --- |
| 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。

以下可以不做，供思考，做了可以适当加分

1. 能否利用堆结构，优化的哈夫曼编码算法。
2. 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。
3. 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 + Intel i7-9750H + 16GB + MinGW-W64-builds-4.3.5 + VSCode

**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. 物理设计
2. 文章：包括文章长度，每个字符的频率。
3. 哈夫曼编码：包括字符，对应的编码。
4. 哈夫曼树节点：包括字符，频率总和，子节点集合。
5. 小根堆：包括堆数组，堆长度。
6. 逻辑设计
   1. 主程序流程图



图 1主程序流程图

* 1. 程序模块之间调用关系



图 2程序模块之间调用关系

**四、测试结果**

(斜体为备注内容)

1. 输入数据

4 *第一个多项式有4项*

-16/5 8 16/7 6 4/9 1 9 0 *第一个多项式各项系数和指数*

3 *第二个多项式有3项*

-4/5 8 4/7 6 -1/2 0 *第二个多项式各项系数和指数*

1 *对多项式求1阶导*

1.5 *对第一个多项式求x=1.5处的值*

3 *对第二个多项式求x=3处的值*

1. 输出数据

(-16/5)\*x^(8) + (16/7)\*x^(6) + (4/9)\*x^(1) + (9/1)\*x^(0) *第一个多项式*

(-4/5)\*x^(8) + (4/7)\*x^(6) + (-1/2)\*x^(0) *第二个多项式*

Add:

(-4/1)\*x^(8) + (20/7)\*x^(6) + (4/9)\*x^(1) + (17/2)\*x^(0) *多项式相加结果*

Minus:

(-12/5)\*x^(8) + (12/7)\*x^(6) + (4/9)\*x^(1) + (19/2)\*x^(0) *多项式相减结果*

Multiply:

(64/25)\*x^(16) + (-128/35)\*x^(14) + (64/49)\*x^(12) + (-16/45)\*x^(9) + (-28/5)\*x^(8) + (16/63)\*x^(7) + (4/1)\*x^(6) + (-2/9)\*x^(1) + (-9/2)\*x^(0) *多项式相乘结果*

Divide:

(4/1)\*x^(0) *多项式相除后商*

(4/9)\*x^(1) + (11/1)\*x^(0) *多项式相除后余数*

Get\_Derivative\_Function:

(-128/5)\*x^(7) + (96/7)\*x^(5) + (4/9)\*x^(0) *对第一个多项式求1阶导*

Get\_Derivative\_Function:

(-32/5)\*x^(7) + (24/7)\*x^(5) *对第二个多项式求1阶导*

Get\_Result:

-46.3101 *第一个多项式x=1.5处的值*

Get\_Result:

-4832.73 *第一个多项式x=3处的值*

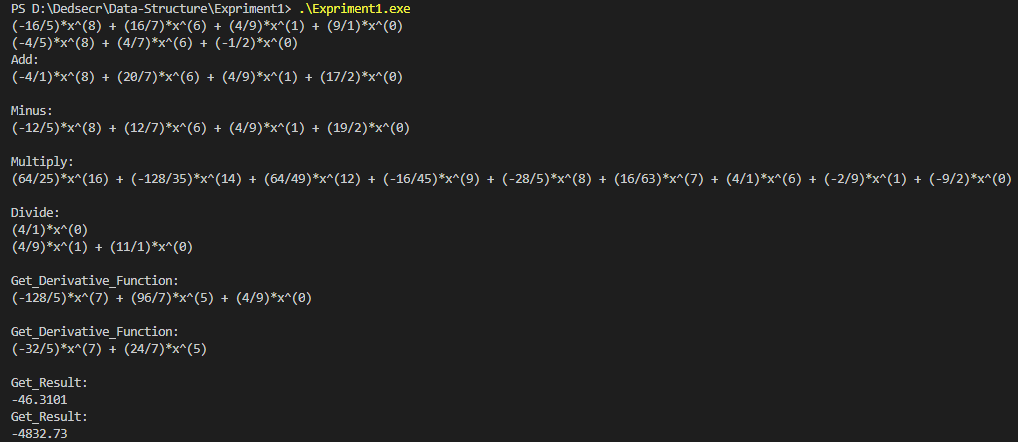


图 3运行结果截图

**五、经验体会与不足**

1. 不足
2. 输入格式对用户较不友好，可改进。
3. 未处理输入中指数重复的情况。
4. 内存空间管理能力较弱。
5. 经验体会
   1. 要考虑清楚各种情况。
   2. 运算符重定向很方便。

**六、附录：源代码（带注释）**

1. #include <iostream>
2. #include <cstdio>
3. #include <algorithm>
4. #include <string>
5. #include <cstring>
6. #include <cmath>
7. **using** **namespace** std;
8. **const** **int** MAXN = 100;
9. //分数
10. **struct** Fraction
11. {
12. **int** numerator, denominator;
13. Fraction(){}
14. Fraction(**int** \_n, **int** \_d)
15. {
16. numerator = \_n;
17. denominator = \_d;
18. Simplification();
19. }
20. Fraction(**int** \_n)
21. {
22. numerator = \_n;
23. denominator = 1;
24. }
25. //求最大公因数
26. **int** gcd(**int** x, **int** y)
27. {
28. **if** (x == 0)
29. **return** y;
30. **return** gcd(y % x, x);
31. }
32. //分数化简
33. **void** Simplification()
34. {
35. **int** \_ = gcd(abs(numerator), abs(denominator));
36. numerator /= \_;
37. denominator /= \_;
38. **if**(numerator < 0 && denominator < 0)
39. numerator = -numerator, denominator = -denominator;
40. }
41. **bool** operator < (**const** Fraction \_) **const**
42. {
43. **return** 1.0 \* numerator / denominator < 1.0 \* \_.numerator / \_.denominator;
44. }
45. **bool** operator > (**const** Fraction \_) **const**
46. {
47. **return** 1.0 \* numerator / denominator > 1.0 \* \_.numerator / \_.denominator;
48. }
49. Fraction operator \* (**const** Fraction \_) **const**
50. {
51. Fraction Res;
52. Res.numerator = numerator \* \_.numerator;
53. Res.denominator = denominator \* \_.denominator;
54. Res.Simplification();
55. **return** Res;
56. }
57. Fraction operator / (**const** Fraction \_) **const**
58. {
59. Fraction Res;
60. Res.numerator = numerator \* \_.denominator;
61. Res.denominator = denominator \* \_.numerator;
62. Res.Simplification();
63. **return** Res;
64. }
65. Fraction operator + (**const** Fraction \_) **const**
66. {
67. Fraction Res;
68. Res.numerator = numerator \* \_.denominator + denominator \* \_.numerator;
69. Res.denominator = denominator \* \_.denominator;
70. Res.Simplification();
71. **return** Res;
72. }
73. Fraction operator - (**const** Fraction \_) **const**
74. {
75. Fraction Res;
76. Res.numerator = numerator \* \_.denominator - denominator \* \_.numerator;
77. Res.denominator = denominator \* \_.denominator;
78. Res.Simplification();
79. **return** Res;
80. }
81. //分数转double
82. **double** ToDouble()
83. {
84. **return** 1.0 \* numerator / denominator;
85. }
86. };
87. //多项式项
88. **struct** EleNum
89. {
90. Fraction coef;
91. **int** expo;
92. EleNum(){}
93. EleNum(Fraction \_c, **int** \_e)
94. {
95. coef = \_c, expo = \_e;
96. }
97. EleNum operator + (**const** EleNum \_) **const**
98. {
99. **if**(expo != \_.expo && expo)
100. {
101. cerr << "Error in " << \_\_LINE\_\_;
102. exit(-1);
103. }
104. EleNum Res;
105. Res.coef = coef + \_.coef;
106. Res.expo = \_.expo;
107. **return** Res;
108. }
109. EleNum operator - (**const** EleNum \_) **const**
110. {
111. **if**(expo != \_.expo && expo)
112. {
113. cerr << "Error in " << \_\_LINE\_\_;
114. exit(-1);
115. }
116. EleNum Res;
117. Res.coef = coef - \_.coef;
118. Res.expo = \_.expo;
119. **return** Res;
120. }
121. EleNum operator \* (**const** EleNum \_) **const**
122. {
123. EleNum Res;
124. Res.coef = coef \* \_.coef;
125. Res.expo = expo + \_.expo;
126. **return** Res;
127. }
128. };
129. //多项式中的项
130. **struct** Ele
131. {
132. EleNum Elements;
133. **int** Next;
134. Ele() {}
135. Ele(EleNum \_E, **int** \_N)
136. {
137. Elements = \_E, Next = \_N;
138. }
139. };
140. //多项式
141. **struct** Polyn
142. {
143. Ele Elements[MAXN];
144. **int** Available[MAXN];
145. **int** Length, AvailableP, LastP;
146. Polyn()
147. {
148. LastP = 0;
149. Length = 0;
150. **for** (AvailableP = 1; AvailableP < MAXN; ++AvailableP)
151. Available[AvailableP] = AvailableP;
152. AvailableP--;
153. }
154. //获取新位置
155. **int** GetNewPos()
156. {
157. **int** Res = Available[AvailableP--];
158. Elements[Res].Next = 0;
159. **return** Res;
160. }
161. //插入新项
162. **void** Insert(EleNum element)
163. {
164. **int** Pos = GetNewPos();
165. Length++;
166. Elements[LastP].Next = Pos;
167. Elements[Pos].Elements = element;
168. Elements[Pos].Next = 0;
169. LastP = Pos;
170. }
171. //回收空间操作
172. **void** Recovery(**int** Pos)
173. {
174. Available[++AvailableP] = Pos;
175. Elements[Pos].Next = 0;
176. }
177. };
178. EleNum Tmp[MAXN \* MAXN], EleNum\_Zero = EleNum(Fraction(0, 1), 0);
179. //判断是否是数字
180. **bool** IsDigit(**char** x)
181. {
182. **return** x >= '0' && x <= '9';
183. }
184. //判断系数是否合法
185. **bool** CoefIsLigal(string x)
186. {
187. **int** Len = x.length();
188. **for** (**int** i = 0; i < Len; ++i)
189. {
190. **if** ((!IsDigit(x[i]) && x[i] != '/' && x[i] != '-') || (x[i] == '/' && (i == 0 || i == Len - 1)))
191. **return** 0;
192. }
193. **return** 1;
194. }
195. //判断指数是否合法
196. **bool** ExpoIsLigal(string x)
197. {
198. **int** Len = x.length();
199. **for** (**int** i = 0; i < Len; ++i)
200. {
201. **if** (!IsDigit(x[i]))
202. **return** 0;
203. }
204. **return** 1;
205. }
206. //判断是否含有分数
207. **int** IsFrac(string x)
208. {
209. **int** Len = x.length();
210. **for** (**int** i = 0; i < Len; ++i)
211. {
212. **if** (x[i] == '/')
213. **return** i + 1;
214. }
215. **return** 0;
216. }
217. //输入项
218. **bool** GetElements(Polyn &polyn)
219. {
220. string input1, input2;
221. **int** input1L, input2L, FracP;
222. **int** numerator, denominator;
223. cin >> input1 >> input2;
224. **if** (!CoefIsLigal(input1) || !ExpoIsLigal(input2))
225. **return** 0;
226. input1L = input1.length();
227. input2L = input2.length();
229. FracP = IsFrac(input1);
230. **if** (FracP)
231. {
232. FracP--;
233. **int** x = 0;
234. **if**(input1[0] == '-')
235. {
236. **for** (**int** i = 1; i < FracP; ++i)
237. {
238. x = x \* 10 + input1[i] - '0';
239. }
240. x = -x;
241. }
242. **else**
243. {
244. **for** (**int** i = 0; i < FracP; ++i)
245. {
246. x = x \* 10 + input1[i] - '0';
247. }
248. }
249. numerator = x;
250. x = 0;
251. **for** (**int** i = FracP + 1; i < input1L; ++i)
252. {
253. x = x \* 10 + input1[i] - '0';
254. }
255. denominator = x;
256. }
257. **else**
258. {
259. **int** x = 0;
260. **if**(input1[0] == '-')
261. {
262. **for** (**int** i = 1; i < input1L; ++i)
263. {
264. x = x \* 10 + input1[i] - '0';
265. }
266. x = -x;
267. }
268. **else**
269. {
270. **for** (**int** i = 0; i < input1L; ++i)
271. {
272. x = x \* 10 + input1[i] - '0';
273. }
274. }
275. numerator = x;
276. denominator = 1;
277. }
279. **int** x = 0;
280. **for** (**int** i = 0; i < input2L; ++i)
281. {
282. x = x \* 10 + input2[i] - '0';
283. }
284. polyn.Insert(EleNum(Fraction(numerator, denominator), x));
286. **return** 1;
287. }
288. **bool** Cmp(**const** EleNum &a, **const** EleNum &b)
289. {
290. **return** a.expo > b.expo;
291. }
292. //对多项式排序
293. **void** Sort(Polyn &polyn)
294. {
295. **int** Length = polyn.Length;
296. **int** P = 0;
297. **for** (**int** i = 1; i <= Length; ++i)
298. {
299. P = polyn.Elements[P].Next;
300. Tmp[i] = polyn.Elements[P].Elements;
301. }
302. sort(Tmp + 1, Tmp + 1 + Length, Cmp);
303. P = 0;
304. **for** (**int** i = 1; i <= Length; ++i)
305. {
306. P = polyn.Elements[P].Next;
307. polyn.Elements[P].Elements = Tmp[i];
308. }
309. }
310. //输入多项式
311. Polyn Input(**int** Length)
312. {
313. Polyn Res;
314. **for** (**int** i = 1; i <= Length; ++i)
315. {
316. **while** (!GetElements(Res))
317. {
318. puts("Wrong format!");
319. }
320. Sort(Res);
321. }
322. **return** Res;
323. }
324. //打印多项式
325. **void** Print(Polyn polyn)
326. {
327. **bool** First = 1;
328. **for** (**int** Pos = polyn.Elements[0].Next; Pos; Pos = polyn.Elements[Pos].Next)
329. {
330. **if**(First)
331. First = 0;
332. **else**
333. cout << " + ";
334. printf("(%d/%d)\*x^(%d)", polyn.Elements[Pos].Elements.coef.numerator,
335. polyn.Elements[Pos].Elements.coef.denominator,
336. polyn.Elements[Pos].Elements.expo);
337. }
338. puts("");
339. }
340. //多项式求和
341. Polyn Add(Polyn &A, Polyn &B)
342. {
343. Polyn Res;
344. **int** P\_A = A.Elements[0].Next, P\_B = B.Elements[0].Next;
345. **while**(P\_A && P\_B)
346. {
347. **if**(A.Elements[P\_A].Elements.expo > B.Elements[P\_B].Elements.expo)
348. {
349. Res.Insert(A.Elements[P\_A].Elements);
350. P\_A = A.Elements[P\_A].Next;
351. }
352. **else** **if**(A.Elements[P\_A].Elements.expo < B.Elements[P\_B].Elements.expo)
353. {
354. Res.Insert(B.Elements[P\_B].Elements);
355. P\_B = B.Elements[P\_B].Next;
356. }
357. **else**
358. {
359. Res.Insert(A.Elements[P\_A].Elements + B.Elements[P\_B].Elements);
360. P\_A = A.Elements[P\_A].Next;
361. P\_B = B.Elements[P\_B].Next;
362. }
363. }
364. **while**(P\_A)
365. {
366. Res.Insert(A.Elements[P\_A].Elements);
367. P\_A = A.Elements[P\_A].Next;
368. }
369. **while**(P\_B)
370. {
371. Res.Insert(B.Elements[P\_B].Elements);
372. P\_B = B.Elements[P\_B].Next;
373. }
374. //cout << Res.Length << "^^^^^^^^^\n";
375. **return** Res;
376. }
377. //多项式相减
378. Polyn Minus(Polyn &A, Polyn &B)
379. {
380. Polyn Res;
381. **int** P\_A = A.Elements[0].Next, P\_B = B.Elements[0].Next;
382. **while**(P\_A && P\_B)
383. {
384. **if**(A.Elements[P\_A].Elements.expo > B.Elements[P\_B].Elements.expo)
385. {
386. Res.Insert(A.Elements[P\_A].Elements);
387. P\_A = A.Elements[P\_A].Next;
388. }
389. **else** **if**(A.Elements[P\_A].Elements.expo < B.Elements[P\_B].Elements.expo)
390. {
391. Res.Insert(EleNum\_Zero - B.Elements[P\_B].Elements);
392. P\_B = B.Elements[P\_B].Next;
393. }
394. **else**
395. {
396. Res.Insert(A.Elements[P\_A].Elements - B.Elements[P\_B].Elements);
397. P\_A = A.Elements[P\_A].Next;
398. P\_B = B.Elements[P\_B].Next;
399. }
400. }
401. **while**(P\_A)
402. {
403. Res.Insert(A.Elements[P\_A].Elements);
404. P\_A = A.Elements[P\_A].Next;
405. }
406. **while**(P\_B)
407. {
408. Res.Insert(EleNum\_Zero - B.Elements[P\_B].Elements);
409. P\_B = B.Elements[P\_B].Next;
410. }
411. **return** Res;
412. }
413. //多项式相乘
414. Polyn Multiply(Polyn &A, Polyn &B)
415. {
416. Polyn Res;
417. **int** TmpP = 0;
418. **for** (**int** P\_A = A.Elements[0].Next; P\_A; P\_A = A.Elements[P\_A].Next)
419. **for** (**int** P\_B = B.Elements[0].Next; P\_B; P\_B = B.Elements[P\_B].Next)
420. Tmp[++TmpP] = A.Elements[P\_A].Elements \* B.Elements[P\_B].Elements;
421. sort(Tmp + 1, Tmp + 1 + TmpP, Cmp);
422. Tmp[0].expo = Tmp[1].expo + 1;
423. **for** (**int** i = 1; i <= TmpP; ++i)
424. {
425. **if**(Tmp[i].expo != Tmp[i-1].expo)
426. Res.Insert(Tmp[i]);
427. **else**
428. Res.Elements[Res.LastP].Elements = Res.Elements[Res.LastP].Elements + Tmp[i];
429. }
430. **return** Res;
431. }
432. //获取当前的商
433. Polyn Get\_This\_Quotient(EleNum A, EleNum B)
434. {
435. EleNum Now = EleNum(A.coef / B.coef, A.expo - B.expo);
436. Polyn Res;
437. Res.Insert(Now);
438. **return** Res;
439. }
440. //去掉系数为0的项
441. **void** Refresh(Polyn &A)
442. {
443. **int** LasP = 0;
444. **int** Pos = A.Elements[0].Next;
445. **while**(Pos)
446. {
447. **if**(A.Elements[Pos].Elements.coef.numerator == 0)
448. {
449. **int** Tmp = Pos;
450. A.Elements[LasP].Next = A.Elements[Pos].Next;
451. Pos = A.Elements[Pos].Next;
452. A.Recovery(Tmp);
453. }
454. **else**
455. LasP = Pos, Pos = A.Elements[Pos].Next;
456. }
458. }
459. //多项式相除
460. pair<Polyn, Polyn> Divide(Polyn A, Polyn &B)
461. {
462. pair<Polyn, Polyn> Res;
463. Polyn Quotient;
464. **while**(A.Elements[A.Elements[0].Next].Elements.expo >= B.Elements[B.Elements[0].Next].Elements.expo)
465. {
466. Polyn This\_Quotient = Get\_This\_Quotient(A.Elements[A.Elements[0].Next].Elements, B.Elements[B.Elements[0].Next].Elements);
467. Polyn tmp = Multiply(B, This\_Quotient);
468. Quotient = Add(Quotient, This\_Quotient);
469. A = Minus(A, tmp);
470. Refresh(A);
471. }
472. Res.first = Quotient;
473. Res.second = A;
474. **return** Res;
475. }
476. //多项式求导
477. Polyn Get\_Derivative\_Function(Polyn &A, **int** k)
478. {
479. Polyn Res = A;
480. **for** (**int** Pos = Res.Elements[0].Next; Pos; Pos = Res.Elements[Pos].Next)
481. {
482. **for** (**int** j = 1; j <= k; ++j)
483. {
484. Res.Elements[Pos].Elements.coef = Res.Elements[Pos].Elements.coef \* Fraction(Res.Elements[Pos].Elements.expo, 1);
485. Res.Elements[Pos].Elements.expo--;
486. }
487. }
488. Refresh(Res);
489. **return** Res;
490. }
491. //多项式求值
492. **double** Get\_Result(Polyn &A, **double** x)
493. {
494. **double** Res = 0;
495. **for** (**int** Pos = A.Elements[0].Next; Pos; Pos = A.Elements[Pos].Next)
496. {
497. Res += A.Elements[Pos].Elements.coef.ToDouble() \* pow(x, A.Elements[Pos].Elements.expo);
498. }
499. **return** Res;
500. }
501. **int** main()
502. {
503. //重定向输入输出至文件
504. freopen("Expriment1\_In.txt", "r", stdin);
505. freopen("Expriment1\_Out.txt", "w", stdout);
506. **int** n1, n2, k;
507. **double** x1, x2;
509. //输入第一个多项式的项数
510. cin >> n1;
511. //输入第一个多项式
512. Polyn test1 = Input(n1);
514. //输入第二个多项式的项数
515. cin >> n2;
516. //输入第二个多项式
517. Polyn test2 = Input(n2);
519. cin >> k;
520. cin >> x1 >> x2;
521. //输出两个多项式
522. Print(test1);
523. Print(test2);
525. //输出相加结果
526. cout << "Add:\n";
527. Print(Add(test1, test2));
528. cout << '\n';
530. //输出相减结果
531. cout << "Minus:\n";
532. Print(Minus(test1, test2));
533. cout << '\n';
535. //输出相乘结果
536. cout << "Multiply:\n";
537. Print(Multiply(test1, test2));
538. cout << '\n';
540. //输出相除结果
541. pair<Polyn, Polyn> Ans = Divide(test1, test2);
542. cout << "Divide:\n";
543. Print(Ans.first);
544. Print(Ans.second);
545. cout << '\n';
547. //输出求导结果
548. cout << "Get\_Derivative\_Function:\n";
549. Print(Get\_Derivative\_Function(test1,1));
550. cout << '\n';
552. cout << "Get\_Derivative\_Function:\n";
553. Print(Get\_Derivative\_Function(test2,1));
554. cout << '\n';
556. //输出求值结果
557. cout << "Get\_Result:\n";
558. cout << Get\_Result(test1, x1) << '\n';
560. cout << "Get\_Result:\n";
561. cout << Get\_Result(test2, x2) << '\n';
562. **return** 0;
563. }