哈尔滨工业大学

**<<数据结构与算法>>**

**实验报告**

**(2024年秋季学期)**

|  |  |
| --- | --- |
| **姓名：** | **郭屹鸣** |
| **学号：** | **2023111958** |
| **学院：** | **计算机学院** |
| **教师：** | **翟德明** |

实验二 树性结构及其应用

## 一、实验内容

**实验 2 树形结构及其应用**

**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

**实验内容：**

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。

**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

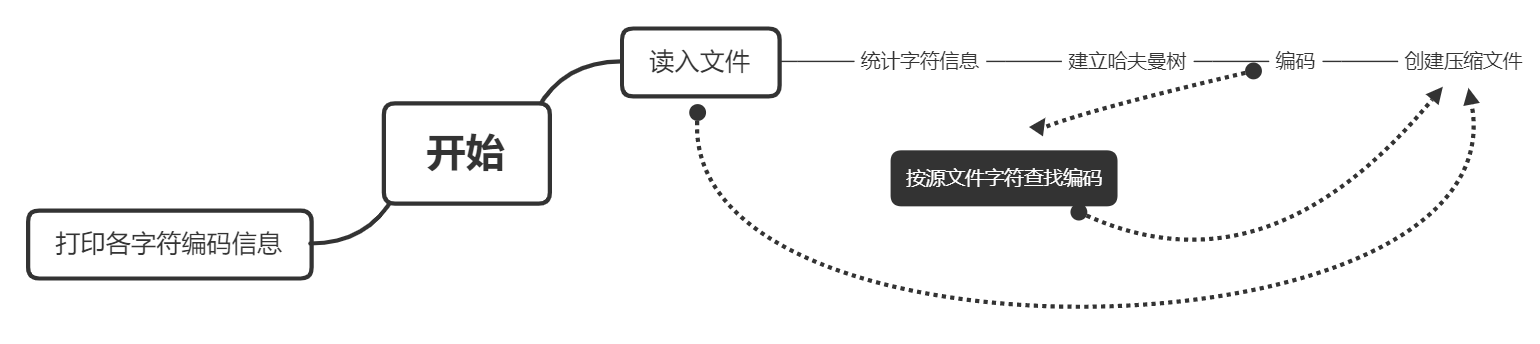
工作，讨论并比较压缩效果。

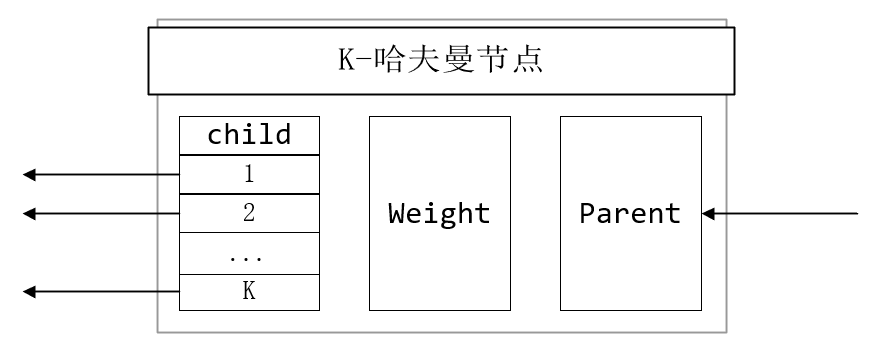
8. 上述 1-5 的编码是二进制的编码，可以采用 K 叉的哈夫曼树完成上述工作，

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。

## 二、实验过程及结果

实验采用了模块化编程，采用了面向对象编程的思想理念，完成了树形结构的设计和算法的实现，完成了本次实验的全部任务。

（1）主要流程：

（2）K叉node设计：

（3)Naive\_Huffman:  
作为最简单的二叉Huffman树，先统计字符的频率，然后不断合并构造节点并且赋予节点亲子关系，随后由叶子节点上溯到root后反向输出编码构建字符编码字典。随后生成二进制哈夫曼编码的file。读取file并且由root下探索到leaf进行encode，随后比较压缩率和输出编码字典，如图1

（4）Pieces\_Huffman是基于单词的空格的压缩，实现Pieces\_Huffman的时候，改动并不是很大，只是单纯的在Naive\_Huffman的基础之上划分空格而已，但是空格符号不好处理，我采取的方法是把空格符单独提出来，在decode的时候再加入进去，相当于以单词为划分，标点符号跟着单词走，最后再还原空格的问题，随后比较压缩率和输出编码字典，如图2

（5）K\_Ray\_Huffman .设计哈夫曼的node如图上所示，同时利用小根堆优化合并的效果，对比不同进制发现二进制的压缩率最高。随后比较压缩率和输出编码字典，如图3(以6进制为例子)

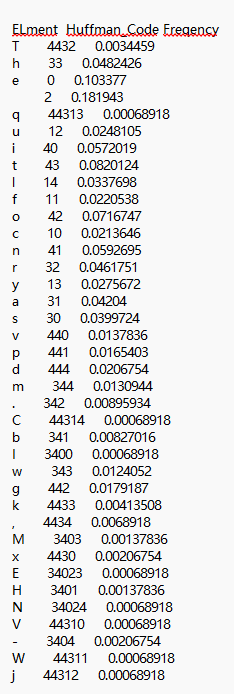
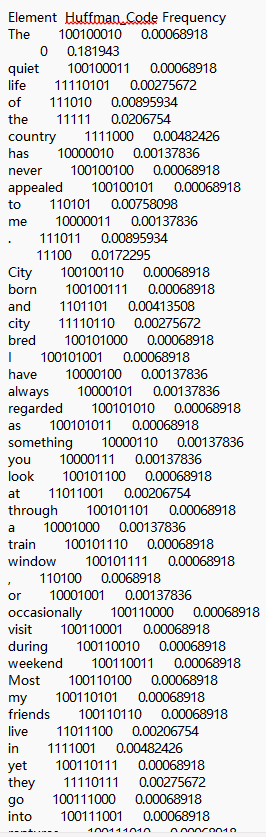
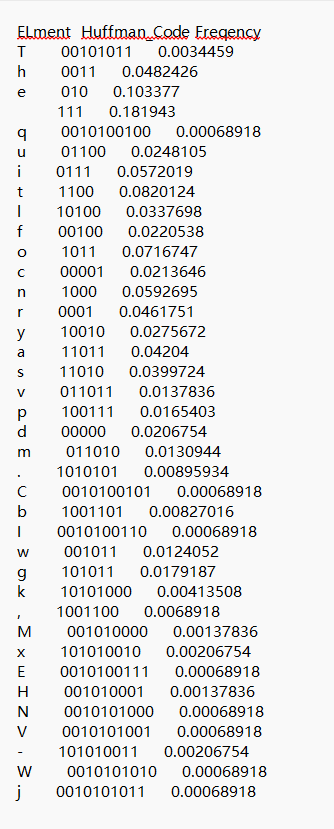
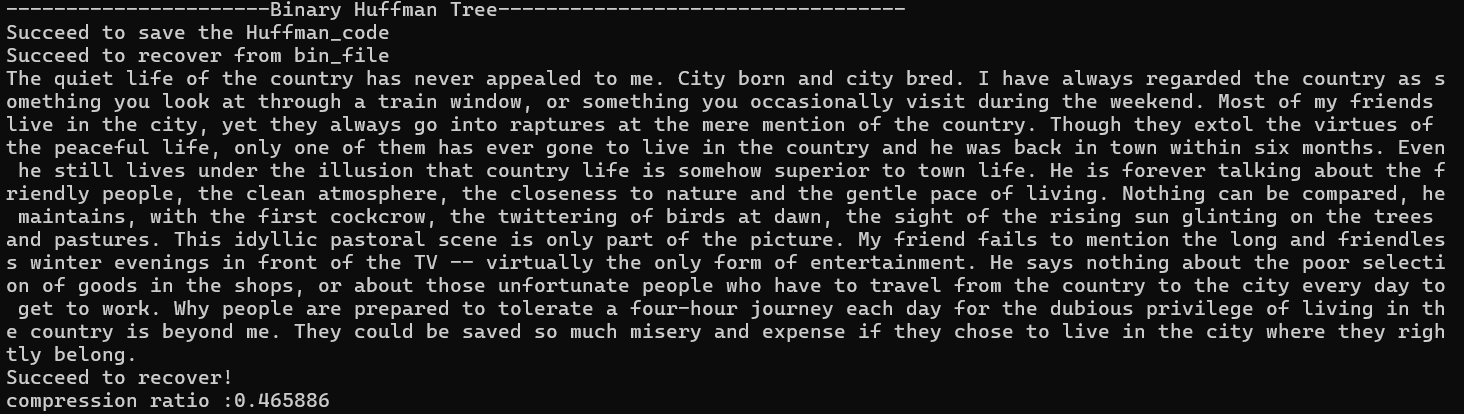
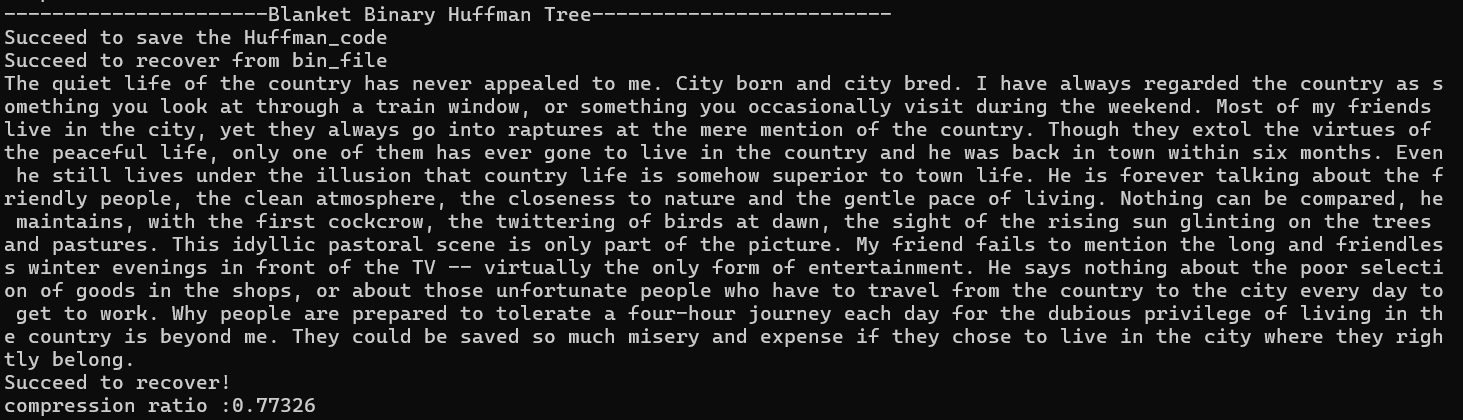


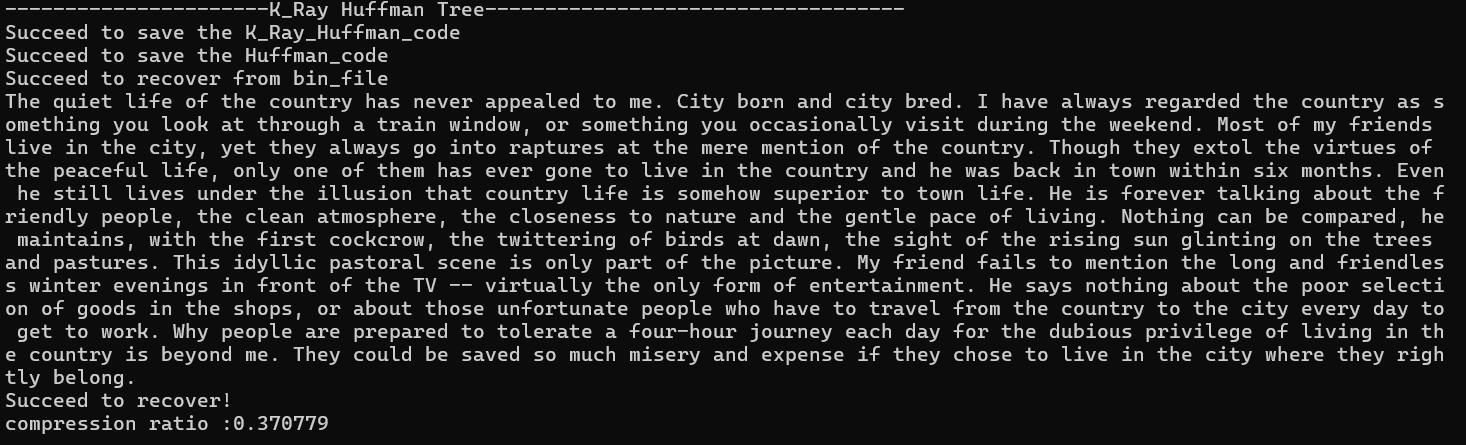
图1 图二 图三



图四：Naive



图五：Pieces



图六：K\_Ray

## 三、实验心得

* **遇到的问题和解决方法**

1. 这次实验的逻辑思维含量比较大，在最开始实现Naive\_Huffman的时候出现了encode找不到root的死循环的问题，逐步打印输出才改掉bug
2. 这次试验主要debug的时间花在K\_Ray Huffman上，在一开始建立堆的时候，我没有考虑到实际上返回的是下标，所以花了时间更改了node的结构并且记录了真实的下标情况。在encode的时候一直陷入死循环，在漫长地debug之后发现还是堆出了问题：新加入的数值下标不对，重新改。期间又遇到C++的深浅拷贝构造的问题，都一一解决了。整体来说这三个实验的思想是比较相近的。堆优化的加入极大优化了算法。
3. 在处理空格的时候，第一次处理没有看清楚题目要求，没把标点符号单独编码，recover正确但是不符合题意
4. 在处理k\_Ray的时候不正确，发现原来写的k叉Huffman的合并逻辑不太对，在root的子树不满k的时候，用0去补了，相当于root的子树小于k。上完课之后我又重新修改逻辑，一开始开始直接补了相关的叶子(权重为0)，使得整棵树正好是满k叉树。
5. 这次实验中犯了几个比较严重的错误，首先对于压缩率的理解有问题，应该是1-现有长/（原长\*8），而且现有长还涉及到补位置的问题，这个一开始弄错了，后面才修正。而且最难受的是，经过今天上课，我发现我的K进制的压码率也错了。我原来以为直接全部按照八进制字节算就好了，没想到原来还要真正压缩还原，相当于我最开始只做到了最低标准。啊啊啊啊我这个报告改了几次了，这次应该没什么bug了，我进行了真正的压缩，然后按照进制计算bit。

* **体会和感受**

通过本次实验，深刻理解了哈夫曼编码的原理及其在数据压缩中的应用。同时，使用堆结构和面向对象编程思想优化算法结构，使得编码过程更加高效。实验中，基于单词和 K 叉的哈夫曼编码进一步拓展了编码的应用场景，体会到不同编码方式在实际压缩效果上的差异。而且在这个过程中犯了很多很多很多的错误，笑死我了，不过很开心，慢慢调总能调出来的，这也是我独立完成的第二个大实验了哈哈哈哈。

很无语，我下次再也不提前发报告了，总是不知道在什么地方会出bug。

1. **源代码**
2. #include "K\_Ray\_Huffman.h"
3. #include "Pieces\_Huffman.h"
4. #include <iostream>
5. int main(void){
6. int K = 5;
7. Huffman\_Operator Binary\_Operator;
8. Pieces\_Huffman\_Operator Blanket\_Binary\_Operator;
9. K\_Ray\_Huffman\_Operator K\_Ray\_Operator(K);
10. std::cout<<"----------------------Binary Huffman Tree----------------------------------"<<std::endl;
11. Binary\_Operator.Generate\_Huffman\_Code();
12. std::cout<<"----------------------Blanket Binary Huffman Tree-------------------------"<<std::endl;
13. Blanket\_Binary\_Operator.Generate\_Huffman\_Code();
14. std::cout<<"----------------------K\_Ray Huffman Tree-----------------------------------"<<std::endl;
15. K\_Ray\_Operator.Generate\_Huffman\_Code();
16. return 0;
17. }
18. #include <iostream>
19. #include <vector>
20. #include <stdexcept>
21. #include "K\_HTNode.h"
22. *// typedef struct K\_HTNode{*
23. *//         int real\_pos;*
24. *//         int weight;*
25. *//         std::vector<int> relation;*
26. *// }K\_HTNode;*
27. void Swap(K\_HTNode &a,K\_HTNode &b) {
28. K\_HTNode temp = a;
29. a = b;
30. b = temp;
31. }
32. void heapify(std::vector<K\_HTNode> &heap,int root) {
33. int size = heap.size();
34. int small = root;
35. int lchild = root \* 2 + 1;
36. int rchild = root \* 2 + 2;
37. if (lchild < size && heap[small].weight > heap[lchild].weight) {
38. small = lchild;
39. }
40. if (rchild < size && heap[small].weight > heap[rchild].weight) {
41. small = rchild;
42. }
43. if (root != small) {
44. Swap(heap[small], heap[root]);
45. heapify(heap, small);
46. }
47. }
48. void Float\_Top(std::vector<K\_HTNode> &heap, int current) {
49. while (current > 0) {
50. int parent = (current - 1) / 2;
51. if (heap[current].weight < heap[parent].weight) {
52. Swap(heap[current], heap[parent]);
53. current = parent;
54. }else{
55. break;
56. }
57. }
58. }
59. void Build\_Heap(std::vector<K\_HTNode> &heap) {
60. for (int i = heap.size() / 2 - 1; i >= 0; i--) {
61. heapify(heap, i);
62. }
63. }
64. int Get\_Min(std::vector<K\_HTNode> &heap,int &sum) {
65. int size = heap.size();
66. if (size <= 0) {
67. throw std::logic\_error("Heap is empty!");
68. }
69. if (size == 1) {
70. heap.pop\_back();
71. return heap[0].real\_pos;
72. }
73. sum += heap[0].weight;
74. int root = heap[0].real\_pos;
75. heap[0] = heap[size - 1];
76. heap.pop\_back();
77. heapify(heap,0);
78. return root;
79. }
80. void printHeap(const std::vector<K\_HTNode>&heap) {
81. *//std::cout<<heap.size();*
82. for (size\_t i = 0; i < heap.size(); i++) {
83. std::cout << heap[i].weight << " ";
84. }
85. std::cout << std::endl;
86. }
87. void Get\_K\_Min(std::vector<K\_HTNode>heap, std::vector<std::vector<int>> &result, int k) {
88. for(size\_t i=0;i<heap.size()-1;i++){
89. heap[i] = heap[i+1];
90. }
91. heap.pop\_back();*//actually pop the front*
93. size\_t constant = heap.size() + 1;
94. Build\_Heap(heap);
95. *//std::cout<<"Inital heap"<<std::endl;*
96. *//printHeap(heap);*
97. while (heap.size() >= k) {
98. int sum = 0;
99. K\_HTNode add;
100. std::vector<int>temp;
101. for (int i = 0; i < k; i++) {
102. *//printHeap(heap);*
103. int real\_pos = Get\_Min(heap,sum);
104. temp.push\_back(real\_pos);
105. *//std::cout << min << " ";*
106. }
107. add.weight = sum;
108. add.real\_pos = constant;
109. constant++;
110. heap.push\_back(add);
111. result.push\_back(temp);
112. Float\_Top(heap, heap.size() - 1);
113. temp.clear();
114. }
115. if(heap.size() != 1) throw std::logic\_error("heap.size()");
116. }
117. */\*int main() {*
118. std::vector<K\_HTNode>A;
119. std::vector<std::vector<int>> result;
120. int K = 3;
121. for(int i = 0; i<=10;i++){
122. K\_HTNode node;
123. node.real\_pos = i;
124. node.weight = i;
125. A.push\_back(node);
126. }
127. Get\_K\_Min(A, result, K);
128. for(int i = 0;i<result.size();i++){
129. for(int j = 0;j<K;j++){
130. std::cout<<result[i][j]<<" ";
131. }
132. std::cout<<std::endl;
133. }
134. // for(int i=0;i<A.size();i++){
135. //     std::cout<<A[i].weight<<std::endl;//check bug
136. // }
137. return 0;
138. }\*/
139. #include <iostream>
140. #include <fstream>
141. #include <string>
142. #include <vector>
143. #include <stack>
144. #include <algorithm>
145. #include <sstream>
146. #include <bitset>
147. #include <cmath>
148. #include <iostream>
149. #include <vector>
150. #include <stdexcept>
151. #include "K\_HTNode.h"
152. *// typedef struct K\_HTNode{*
153. *//         int real\_pos;*
154. *//         int weight;*
155. *//         std::vector<int> relation;*
156. *// }K\_HTNode;*
157. void Swap(K\_HTNode &a,K\_HTNode &b) {
158. K\_HTNode temp = a;
159. a = b;
160. b = temp;
161. }
162. void heapify(std::vector<K\_HTNode> &heap,int root) {
163. int size = heap.size();
164. int small = root;
165. int lchild = root \* 2 + 1;
166. int rchild = root \* 2 + 2;
167. if (lchild < size && heap[small].weight > heap[lchild].weight) {
168. small = lchild;
169. }
170. if (rchild < size && heap[small].weight > heap[rchild].weight) {
171. small = rchild;
172. }
173. if (root != small) {
174. Swap(heap[small], heap[root]);
175. heapify(heap, small);
176. }
177. }
178. void Float\_Top(std::vector<K\_HTNode> &heap, int current) {
179. while (current > 0) {
180. int parent = (current - 1) / 2;
181. if (heap[current].weight < heap[parent].weight) {
182. Swap(heap[current], heap[parent]);
183. current = parent;
184. }else{
185. break;
186. }
187. }
188. }
189. void Build\_Heap(std::vector<K\_HTNode> &heap) {
190. for (int i = heap.size() / 2 - 1; i >= 0; i--) {
191. heapify(heap, i);
192. }
193. }
194. int Get\_Min(std::vector<K\_HTNode> &heap,int &sum) {
195. int size = heap.size();
196. if (size <= 0) {
197. throw std::logic\_error("Heap is empty!");
198. }
199. if (size == 1) {
200. heap.pop\_back();
201. return heap[0].real\_pos;
202. }
203. sum += heap[0].weight;
204. int root = heap[0].real\_pos;
205. heap[0] = heap[size - 1];
206. heap.pop\_back();
207. heapify(heap,0);
208. return root;
209. }
210. void printHeap(const std::vector<K\_HTNode>&heap) {
211. *//std::cout<<heap.size();*
212. for (size\_t i = 0; i < heap.size(); i++) {
213. std::cout << heap[i].weight << " ";
214. }
215. std::cout << std::endl;
216. }
217. void Get\_K\_Min(std::vector<K\_HTNode>heap, std::vector<std::vector<int>> &result, int k) {
218. for(size\_t i=0;i<heap.size()-1;i++){
219. heap[i] = heap[i+1];
220. }
221. heap.pop\_back();*//actually pop the front*
223. size\_t constant = heap.size() + 1;
224. Build\_Heap(heap);
225. *//std::cout<<"Inital heap"<<std::endl;*
226. *//printHeap(heap);*
227. while (heap.size() >= k) {
228. int sum = 0;
229. K\_HTNode add;
230. std::vector<int>temp;
231. for (int i = 0; i < k; i++) {
232. *//printHeap(heap);*
233. int real\_pos = Get\_Min(heap,sum);
234. temp.push\_back(real\_pos);
235. *//std::cout << min << " ";*
236. }
237. add.weight = sum;
238. add.real\_pos = constant;
239. constant++;
240. heap.push\_back(add);
241. result.push\_back(temp);
242. Float\_Top(heap, heap.size() - 1);
243. temp.clear();
244. }
245. if(heap.size() != 1) throw std::logic\_error("heap.size()");
246. }
247. */\*int main() {*
248. std::vector<K\_HTNode>A;
249. std::vector<std::vector<int>> result;
250. int K = 3;
251. for(int i = 0; i<=10;i++){
252. K\_HTNode node;
253. node.real\_pos = i;
254. node.weight = i;
255. A.push\_back(node);
256. }
257. Get\_K\_Min(A, result, K);
258. for(int i = 0;i<result.size();i++){
259. for(int j = 0;j<K;j++){
260. std::cout<<result[i][j]<<" ";
261. }
262. std::cout<<std::endl;
263. }
264. // for(int i=0;i<A.size();i++){
265. //     std::cout<<A[i].weight<<std::endl;//check bug
266. // }
267. return 0;
268. }\*/
269. #ifndef NAIVE\_HUFFMAN\_H
270. #define NAIVE\_HUFFMAN\_H
271. #include "ulity.h"
272. class Huffman\_Operator{
273. public:
274. typedef struct{
275. double weight;
276. int parent;
277. int lchild;
278. int rchild;
279. }HTNode;
280. typedef struct{
281. int weight;
282. char data;
283. std::string HC;
284. }Dictionary;
285. std::string buffer,huffman\_buffer,decode\_buffer,recover;
286. std::vector<Dictionary>Diction;
287. std::vector<HTNode>Huffman\_Tree;
288. int leaf\_num;
289. int node\_num;
290. int total;
291. Huffman\_Operator(){
292. }
293. ~Huffman\_Operator(){
294. }
295. virtual void Encode\_Huffman(){
296. int root = node\_num ;
297. for(int i = 1;i <= leaf\_num;i++){
298. int current = i;
299. while(current != root){
300. std::string code;
301. int parent = Huffman\_Tree[current].parent;
302. code =  Huffman\_Tree[parent].lchild == current ? "0":"1";
303. Diction[i].HC += code;
304. current = parent;
305. }
306. std::reverse(Diction[i].HC.begin(), Diction[i].HC.end());
307. }
308. */\*for(int i=1;i<=leaf\_num;i++){*
309. std::cout<<Huffman\_Tree[i].weight<<" "<<Huffman\_Tree[i].parent
310. <<" "<<Huffman\_Tree[Huffman\_Tree[i].parent].lchild<<std::endl;
311. }\*/
312. }
313. virtual void Decode(){
314. int root = node\_num;
315. size\_t i = 0;
316. while(i < decode\_buffer.size()){
317. int current = root;
318. while(Huffman\_Tree[current].lchild||Huffman\_Tree[current].rchild){
319. int child = decode\_buffer[i] == '0' ? Huffman\_Tree[current].lchild : Huffman\_Tree[current].rchild;
320. current = child;
321. i++;
322. }
323. recover += Diction[current].data;
324. }
325. std::cout<<recover<<std::endl;
326. if(recover == buffer) std::cout<<"Succeed to recover!"<<std::endl;
327. }
328. virtual void conclusion(){
329. int add = 8 - decode\_buffer.size() % 8;
330. double rate = 1- (decode\_buffer.size() + add + 8)/(8.0\*buffer.size());
331. std::cout<<"compression ratio :"<<rate<<std::endl;
332. }
333. virtual void Create\_Huffman(){
334. Inital\_Huffman();
335. if(leaf\_num <= 1) return;
336. int pos1,pos2;
337. for(int i = leaf\_num + 1; i <= node\_num; i++){
338. Select\_Two(pos1,pos2,i-1);
339. Huffman\_Tree[pos1].parent = i;
340. Huffman\_Tree[pos2].parent = i;
341. HTNode compact;
342. compact.weight = Huffman\_Tree[pos1].weight + Huffman\_Tree[pos2].weight;
343. compact.lchild = pos1;
344. compact.rchild = pos2;
345. compact.parent = 0;
346. Huffman\_Tree.push\_back(compact);
347. }
348. }
349. virtual void Generate\_Huffman\_Code(){
350. ReadFile();
351. Caculate\_Weight();
352. Create\_Huffman();
353. Encode\_Huffman();
354. Display\_Huffman\_Code("Naive\_Compression\_Dictionary.txt");
355. Display\_File\_With\_Huffman("File\_With\_Huffman.txt");
356. Save\_Bin\_File();
357. Read\_Bin\_File();
358. Decode();
359. conclusion();
360. }
361. protected:
362. virtual inline void Read\_Bin\_File(){
363. std::ifstream readFile("Naive\_Compact\_File.bin",std::ios::binary);
364. if(!readFile.is\_open()){
365. std::cerr<<"Fail to open Naive\_Compact\_File.bin"<<std::endl;
366. }
367. std::string bin\_buffer;
368. unsigned char temp;
369. int add;
370. readFile.read(reinterpret\_cast<char\*>(&temp), sizeof(temp));
371. add = static\_cast<int>(temp);
372. while(readFile.read(reinterpret\_cast<char\*>(&temp), sizeof(temp))){
373. bin\_buffer += std::bitset<8>(temp).to\_string();
374. }
375. for(int k = 0;k<add;k++){
376. bin\_buffer.pop\_back();
377. }
378. if(bin\_buffer == decode\_buffer){
379. std::cout<<"Succeed to recover from bin\_file"<<std::endl;
380. decode\_buffer.clear();
381. decode\_buffer = bin\_buffer;
382. }
383. readFile.close();
384. }
386. virtual inline void Save\_Bin\_File(){
387. Read\_Huffman\_File("File\_With\_Huffman.txt");
388. int add = 8 - decode\_buffer.size()%8;
389. for(int i = 0; i<add; i++){
390. decode\_buffer += "0";
391. }
392. std::ofstream outputFile("Naive\_Compact\_File.bin", std::ios::binary);
393. if (!outputFile.is\_open()) {
394. throw std::logic\_error("Failed to open Naive\_Compact\_File.bin");
395. }
396. std::string compact\_buffer;
397. unsigned char temp = static\_cast<unsigned char>(add);
398. outputFile.write(reinterpret\_cast<const char\*>(&temp), sizeof(temp));
399. for(size\_t i = 0;i <decode\_buffer.size(); i+=8){
400. compact\_buffer = decode\_buffer.substr(i,8);
401. unsigned char byte = static\_cast<unsigned char>(std::bitset<8>(compact\_buffer).to\_ulong());
402. outputFile.write(reinterpret\_cast<const char\*>(&byte), sizeof(byte));
403. compact\_buffer.clear();
404. }
405. for(int i = 0; i <add; i++){
406. decode\_buffer.pop\_back();
407. }
408. outputFile.close();
409. }
410. virtual inline void Display\_File\_With\_Huffman(std::string address){
411. for(int i = 0;i < total; i++){
412. for(int j = 1;j <= leaf\_num;j++){
413. if(Diction[j].data == buffer[i]){
414. huffman\_buffer = huffman\_buffer + Diction[j].HC;
415. }
416. }
417. }
418. std::ofstream outFile(address);
419. if(outFile.is\_open()){
420. outFile << huffman\_buffer;
421. outFile.close();
422. }
423. }
424. virtual inline void Display\_Huffman\_Code(std::string address){
425. std::ofstream outFile(address);
426. if(outFile.is\_open()){
427. outFile<<"ELment  Huffman\_Code Freqency "<<std::endl;
428. for(int i = 1;i <= leaf\_num ;i++){
429. outFile<<Diction[i].data<<"       "<<Diction[i].HC<<"     "
430. <<1.0\*Diction[i].weight/total<<std::endl;
431. }
432. outFile.close();
433. std::cout<<"Succeed to save the Huffman\_code"<<std::endl;
434. }
436. }
438. virtual inline void Read\_Huffman\_File(std::string address){
439. std::ifstream inputFile(address);
440. if(!inputFile) std::cerr<<"Fail to open the file"<<std::endl;
441. std::string line;
442. while(std::getline(inputFile,line)){
443. decode\_buffer += line;
444. }
445. *//std::cout<<"After read"<<decode\_buffer.size();*
446. inputFile.close();
447. }
448. virtual inline void ReadFile(){
449. std::ifstream inputFile("E:\\HomeWork\\lab2\\Life.txt");
450. if(!inputFile) std::cerr<<"Fail to open the file"<<std::endl;
451. std::string line;
452. while(std::getline(inputFile,line)){
453. buffer+=line;
454. }
455. inputFile.close();
456. total = buffer.size();
457. *//std::cout<<buffer<<std::endl;*
458. }
460. virtual inline void Caculate\_Weight() {
461. Diction.push\_back({-1,'@',""});
462. for(int i = 0;i < static\_cast<int>(buffer.size()); i++){
463. bool find = false;
464. for(std::vector<Dictionary>::iterator it = Diction.begin();it!=Diction.end();it++){
465. if(it->data == buffer[i] && it->weight != -1){
466. it->weight++;
467. find = true;
468. break;
469. }
470. }
471. if(!find) Diction.push\_back({1,buffer[i],""});
472. }
473. leaf\_num = static\_cast<int>(Diction.size())-1;
474. node\_num = 2 \* leaf\_num - 1;
475. }
476. virtual inline void Inital\_Huffman(){
477. Huffman\_Tree.push\_back({0,0,0,0});
478. for(int i = 1;i <= leaf\_num;i++){
479. HTNode node;
480. node.lchild = 0;
481. node.rchild = 0;
482. node.parent = 0;
483. node.weight = Diction[i].weight;
484. Huffman\_Tree.push\_back(node);
485. }
486. */\*for(int i = 1;i<=leaf\_num;i++){*
487. std::cout<<Huffman\_Tree[i].weight<<" ";
488. }\*/
489. }
490. virtual inline void Select\_Two(int &min1,int &min2,int upper){
491. for(int i = 1; i <= upper; i++){
492. if(Huffman\_Tree[i].parent == 0){
493. min1 = i;
494. break;
495. }
496. }
497. for(int i = 1; i <= upper; i++){
498. if(Huffman\_Tree[min1].weight>Huffman\_Tree[i].weight && Huffman\_Tree[i].parent == 0){
499. min1 = i;
500. }
501. }
502. for(int j = 1; j <= upper; j++){
503. if(Huffman\_Tree[j].parent == 0 && j!=min1){
504. min2 = j;
505. break;
506. }
507. }
508. for(int j = min2; j <= upper; j++){
509. if(Huffman\_Tree[min2].weight>Huffman\_Tree[j].weight && Huffman\_Tree[j].parent == 0 && j!=min1){
510. min2 = j;
511. }
512. }
513. }
514. };
515. #endif
516. *// int main(void)*
517. *// {*
518. *//     Huffman\_Operator hf;*
519. *//     hf.Generate\_Huffman\_Code();*
520. *//     return 0;*
521. *// }*
522. #include "ulity.h"
523. #include "Naive\_Huffman.h"
524. class Pieces\_Huffman\_Operator{
525. public:
526. typedef struct{
527. double weight;
528. int parent;
529. int lchild;
530. int rchild;
531. }HTNode;
532. typedef struct {
533. int weight;
534. std::string data;
535. std::string HC;
536. } Dictionary;
537. std::vector<Dictionary> Diction;
538. std::vector<std::string> proc\_buffer;
539. std::string buffer,huffman\_buffer,decode\_buffer,recover;
540. std::vector<HTNode>Huffman\_Tree;
541. int leaf\_num;
542. int node\_num;
543. int total;
544. int total\_word;
545. Pieces\_Huffman\_Operator(){}
546. ~Pieces\_Huffman\_Operator() {}
547. virtual void Create\_Huffman(){
548. Inital\_Huffman();
549. if(leaf\_num <= 1) return;
550. int pos1,pos2;
551. for(int i = leaf\_num + 1; i <= node\_num; i++){
552. Select\_Two(pos1,pos2,i-1);
553. Huffman\_Tree[pos1].parent = i;
554. Huffman\_Tree[pos2].parent = i;
555. HTNode compact;
556. compact.weight = Huffman\_Tree[pos1].weight + Huffman\_Tree[pos2].weight;
557. compact.lchild = pos1;
558. compact.rchild = pos2;
559. compact.parent = 0;
560. Huffman\_Tree.push\_back(compact);
561. }
562. }
563. void Encode\_Huffman() {
564. int root = node\_num;
565. for (int i = 1; i <= leaf\_num; i++) {
566. int current = i;
567. while (current != root) {
568. std::string code;
569. int parent = Huffman\_Tree[current].parent;
570. code = Huffman\_Tree[parent].lchild == current ? "0" : "1";
571. Diction[i].HC += code;
572. current = parent;
573. }
574. std::reverse(Diction[i].HC.begin(), Diction[i].HC.end());
575. }
576. }
577. void Display\_File\_With\_Huffman (std::string address){
578. for(int i = 0;i < total\_word; i++){
579. for(int j = 1;j <= leaf\_num;j++){
580. if(Diction[j].data == proc\_buffer[i]){
581. huffman\_buffer = huffman\_buffer + Diction[j].HC;
582. }
583. }
584. }
585. std::ofstream outFile(address);
586. if(outFile.is\_open()){
587. outFile << huffman\_buffer;
588. outFile.close();
589. }
590. }
591. void Decode(){
592. int root = node\_num;
593. size\_t i = 0;
594. while(i < decode\_buffer.size()){
595. int current = root;
596. while(Huffman\_Tree[current].lchild||Huffman\_Tree[current].rchild){
597. int child = decode\_buffer[i] == '0' ? Huffman\_Tree[current].lchild : Huffman\_Tree[current].rchild;
598. current = child;
599. i++;
600. }
601. recover += Diction[current].data;
602. }
603. std::cout<<recover<<std::endl;
604. if(recover == buffer) std::cout<<"Succeed to recover!"<<std::endl;
605. }
606. void Generate\_Huffman\_Code() {
607. ReadFile();
608. Caculate\_Weight();
609. Create\_Huffman();
610. Encode\_Huffman();
611. Display\_Huffman\_Code();
612. Display\_File\_With\_Huffman("E:\\HomeWork\\lab2\\output\\Pieces\_File\_With\_Huffman.txt");
613. *//Read\_Huffman\_File("Pieces\_File\_With\_Huffman.txt");*
614. Save\_Bin\_File();
615. Read\_Bin\_File();
616. Decode();
617. conclusion();
618. }
619. protected:
620. virtual inline void Read\_Bin\_File(){
621. std::ifstream readFile("Pieces\_Compact\_File.bin",std::ios::binary);
622. if(!readFile.is\_open()){
623. std::cerr<<"Fail to open Pieces\_Compact\_File.bin"<<std::endl;
624. }
625. std::string bin\_buffer;
626. unsigned char temp;
627. int add;
628. readFile.read(reinterpret\_cast<char\*>(&temp), sizeof(temp));
629. add = static\_cast<int>(temp);
630. while(readFile.read(reinterpret\_cast<char\*>(&temp), sizeof(temp))){
631. bin\_buffer += std::bitset<8>(temp).to\_string();
632. }
633. for(int k = 0;k<add;k++){
634. bin\_buffer.pop\_back();
635. }
636. if(bin\_buffer == decode\_buffer){
637. std::cout<<"Succeed to recover from bin\_file"<<std::endl;
638. decode\_buffer.clear();
639. decode\_buffer = bin\_buffer;
640. }
641. readFile.close();
642. }
644. virtual inline void Save\_Bin\_File(){
645. Read\_Huffman\_File("Pieces\_File\_With\_Huffman.txt");
646. int add = 8 - decode\_buffer.size()%8;
647. for(int i = 0; i<add; i++){
648. decode\_buffer += "0";
649. }
650. std::ofstream outputFile("Pieces\_Compact\_File.bin", std::ios::binary);
651. if (!outputFile.is\_open()) {
652. throw std::logic\_error("Failed to open Pieces\_Compact\_File.bin");
653. }
654. std::string compact\_buffer;
655. unsigned char temp = static\_cast<unsigned char>(add);
656. outputFile.write(reinterpret\_cast<const char\*>(&temp), sizeof(temp));
657. for(size\_t i = 0;i <decode\_buffer.size(); i+=8){
658. compact\_buffer = decode\_buffer.substr(i,8);
659. unsigned char byte = static\_cast<unsigned char>(std::bitset<8>(compact\_buffer).to\_ulong());
660. outputFile.write(reinterpret\_cast<const char\*>(&byte), sizeof(byte));
661. compact\_buffer.clear();
662. }
663. for(int i = 0; i <add; i++){
664. decode\_buffer.pop\_back();
665. }
666. outputFile.close();
667. }
668. inline void conclusion(){
669. int add = 8 - decode\_buffer.size() % 8;
670. double rate = 1- (decode\_buffer.size() + add + 8)/(8.0\*buffer.size());
671. std::cout<<"compression ratio :"<<rate<<std::endl;
672. }
673. inline void Display\_Huffman\_Code() {
674. std::ofstream outFile("Pieces\_Compression\_Dictionary.txt");
675. if (outFile.is\_open()) {
676. outFile << "Element  Huffman\_Code Frequency " << std::endl;
677. for (int i = 1; i <= leaf\_num; i++) {
678. outFile << Diction[i].data << "       " << Diction[i].HC << "     " << 1.0 \* Diction[i].weight / total << std::endl;
679. }
680. outFile.close();
681. std::cout << "Succeed to save the Huffman\_code" << std::endl;
682. }
683. }
684. virtual inline void Read\_Huffman\_File(std::string address){
685. std::ifstream inputFile(address);
686. if(!inputFile) std::cerr<<"Fail to open the file"<<std::endl;
687. std::string line;
688. while(std::getline(inputFile,line)){
689. decode\_buffer += line;
690. }
691. inputFile.close();
692. }
693. inline void ReadFile(){
694. std::ifstream inputFile("E:\\HomeWork\\lab2\\Life.txt");
695. if (!inputFile) std::cerr << "Fail to open the file" << std::endl;
696. std::string line;
697. while (std::getline(inputFile, line)) {
698. buffer += line;
699. }
700. inputFile.close();
701. total = buffer.size();
702. *//std::cout<<total;*
703. *//std::cout<<buffer<<std::endl;*
704. }
705. inline void Caculate\_Weight(){
706. std::string temp = "";
707. for(int i = 0;i < static\_cast<int>(buffer.size());){
708. *//int k = 0;*
709. bool word = false;
710. if(std::isalpha(buffer[i]) || i+1<static\_cast<int>(buffer.size())&&
711. !std::isalpha(buffer[i])&&std::isalpha(buffer[i+1]) && !isspace(buffer[i])){
712. temp.push\_back(buffer[i]);
713. word = true;
714. i++;
715. }
716. if(word) continue;
717. *//std::cout<<temp;*
718. proc\_buffer.push\_back(temp);
719. proc\_buffer.push\_back(std::string(1, buffer[i]));*//bug*
720. temp.clear();
721. i++;
722. }
723. Diction.push\_back({-1,"&",""});
724. for(size\_t i = 0; i< proc\_buffer.size();i++){
725. bool find = false;
726. for(size\_t k = 0; k<Diction.size(); k++){
727. if(Diction[k].data == proc\_buffer[i]){
728. Diction[k].weight++;
729. find = true;
730. break;
731. }
732. }
733. if(!find) Diction.push\_back({1,proc\_buffer[i],""});
734. }
735. *// for(size\_t i = 0; i<Diction.size() ;i++){*
736. *//     std::cout<<Diction[i].data<<"====="<<Diction[i].weight<<std::endl;*
737. *// }*
738. leaf\_num = static\_cast<int>(Diction.size()) - 1;
739. node\_num = 2 \* leaf\_num - 1;
740. total\_word = proc\_buffer.size();
741. }
742. virtual inline void Inital\_Huffman(){
743. Huffman\_Tree.push\_back({0,0,0,0});
744. for(int i = 1;i <= leaf\_num;i++){
745. HTNode node;
746. node.lchild = 0;
747. node.rchild = 0;
748. node.parent = 0;
749. node.weight = Diction[i].weight;
750. Huffman\_Tree.push\_back(node);
751. }
753. }
754. inline void Select\_Two(int &min1, int &min2, int upper) {
755. for (int i = 1; i <= upper; i++) {
756. if (Huffman\_Tree[i].parent == 0) {
757. min1 = i;
758. break;
759. }
760. }
761. for (int i = 1; i <= upper; i++) {
762. if (Huffman\_Tree[min1].weight > Huffman\_Tree[i].weight && Huffman\_Tree[i].parent == 0) {
763. min1 = i;
764. }
765. }
766. for (int j = 1; j <= upper; j++) {
767. if (Huffman\_Tree[j].parent == 0 && j != min1) {
768. min2 = j;
769. break;
770. }
771. }
772. for (int j = min2; j <= upper; j++) {
773. if (Huffman\_Tree[min2].weight > Huffman\_Tree[j].weight && Huffman\_Tree[j].parent == 0 && j != min1) {
774. min2 = j;
775. }
776. }
777. }
778. };
779. *// int main(void) {*
780. *//     Pieces\_Huffman\_Operator hf;*
781. *//     hf.Generate\_Huffman\_Code();*
782. *//     return 0;*
783. *// }*
784. #include "ulity.h"
785. #include "Naive\_Huffman.h"
786. #include "Heap\_Sort.h"
787. class K\_Ray\_Huffman\_Operator:public Huffman\_Operator{
788. public:
789. *// int leaf\_num;//直接用父亲的，不必重新定义了*
790. *// int node\_num;*
791. *// int total;*
792. int size = 0,bit\_num = 0;
793. int system;
794. int node\_num = 0;
795. int used\_leaf\_num = 0;
797. *// typedef struct{*
798. *//     int weight;*
799. *//     char data;*
800. *//     std::string HC;*
801. *// }Dictionary;*
803. *// std::string buffer,huffman\_buffer,decode\_buffer,recover;*
804. *// std::vector<Dictionary>Diction;*
805. std::vector<K\_HTNode>Huffman\_Tree;
806. std::vector<int>root;
808. K\_Ray\_Huffman\_Operator(int k):Huffman\_Operator(),system(k){}
809. ~K\_Ray\_Huffman\_Operator(){}
810. void Generate\_Huffman\_Code(){
811. Huffman\_Operator::ReadFile();
812. Huffman\_Operator::Caculate\_Weight();*//bug 子定义的Diction会隐藏父类,不要再重新定义子变量*
813. Create\_Huffman();
814. Encode\_Huffman();
815. Display\_File\_With\_Huffman();
816. Huffman\_Operator::Display\_Huffman\_Code("K\_Ray\_Compression\_Dictionary.txt");
817. *//Huffman\_Operator::Read\_Huffman\_File("E:\\HomeWork\\lab2\\output\\File\_With\_K\_Ray\_Huffman.txt");*
818. Save\_Bin\_File();
819. Read\_Bin\_File();
820. Decode();
821. conclusion();
822. }
824. virtual void Decode()override{
825. int root = node\_num;
826. size\_t i = 0;
827. while(i < decode\_buffer.size()){
828. int current = root;
829. while(Check\_Child(current)){
830. for(int j = 0; j < static\_cast<int>(Huffman\_Tree[current].relation.size()); j++){
831. if(j+'0' == decode\_buffer[i]){
832. int child = Huffman\_Tree[current].relation[j];
833. current = child;
834. i++;
835. break;
836. }
837. }
838. }
839. recover += Diction[current].data;
840. }
841. std::cout<<recover<<std::endl;
842. if(recover == buffer) std::cout<<"Succeed to recover!"<<std::endl;
843. }
844. void Display\_File\_With\_Huffman() {
845. for(int i = 0;i < total; i++){
846. for(int j = 1;j <= leaf\_num;j++){
847. if(Diction[j].data == buffer[i]){
848. huffman\_buffer = huffman\_buffer + Diction[j].HC;
849. }
850. }
851. }
852. std::ofstream outFile("File\_With\_K\_Ray\_Huffman.txt");
853. if(outFile.is\_open()){
854. outFile << huffman\_buffer;
855. outFile.close();
856. std::cout<<"Succeed to save the K\_Ray\_Huffman\_code"<<std::endl;
857. }
858. }
859. void Encode\_Huffman()override{
860. for(int i = 1;i <= leaf\_num;i++){
861. *//std::cout<<"leaf:"<<i<<" "<<std::endl;*
862. int current = i;
863. while(current != node\_num){
864. std::string code;
865. int parent = Huffman\_Tree[current].parent;
866. for(size\_t j=0;j < Huffman\_Tree[parent].relation.size();j++){
867. if(Huffman\_Tree[parent].relation[j] == current ) {
868. code = std::to\_string(j);
869. Diction[i].HC += code;
870. }
871. }
872. *//std::cout<<current<<" ";*
873. current = parent;
874. }
875. std::reverse(Diction[i].HC.begin(), Diction[i].HC.end());
876. }
877. }
878. void Create\_Huffman()override{
879. Inital\_Huffman();
880. if(leaf\_num <= 1) return;
881. int real\_pos;
882. std::vector<std::vector<int>>result;
883. Get\_K\_Min(Huffman\_Tree,result,system);
884. for(size\_t i = 0; i<result.size(); i++){
885. K\_HTNode addition;
886. addition.parent = 0;
887. addition.real\_pos = Huffman\_Tree.size();
888. *//std::cout<<"addition.real\_pos"<<addition.real\_pos;*
889. *//std::cout<<"Huffman\_Size"<<Huffman\_Tree.size();*
890. addition.weight = 0;
891. for(size\_t j=0; j<result[i].size(); j++){
892. real\_pos = result[i][j];
893. *//std::cout<<real\_pos<<count++<<" "<<Huffman\_Tree.size()<<std::endl;*
894. *//std::cout<<"\*\*\*"<<Huffman\_Tree[real\_pos].parent<<std::endl;*
895. Huffman\_Tree[real\_pos].parent = addition.real\_pos;*//bug*
896. addition.weight += Huffman\_Tree[real\_pos].weight;
897. addition.relation.push\_back(real\_pos);
898. }
899. Huffman\_Tree.push\_back(addition);
900. }
901. node\_num = Huffman\_Tree.size() - 1;
902. *//std::cout<<"node\_num"<<" "<<node\_num;*
903. }
905. protected:
906. inline void Read\_Bin\_File()override{
907. std::ifstream readFile("K\_Ray\_Compact\_File.bin", std::ios::binary);
908. if (!readFile.is\_open()){
909. std::cerr << "Fail to open K\_Ray\_Compact\_File.bin" << std::endl;
910. }
911. std::string bin\_buffer;
912. unsigned char temp;
913. int add;
915. readFile.read(reinterpret\_cast<char\*>(&temp), sizeof(temp));
916. add = static\_cast<int>(temp);
917. *//bit\_num ++;*
918. while (readFile.read(reinterpret\_cast<char\*>(&temp), sizeof(temp))){
919. *//bit\_num ++;*
920. std::string sum;
921. int value = static\_cast<int>(temp);
922. for (int i = 0; i<size; i++) {
923. sum = std::to\_string(value%system) + sum;
924. value = value/system;
925. }
926. bin\_buffer += sum;
927. }
928. *//std::cout<<"bit\_num"<<bit\_num<<std::endl;*
929. for (int k = 0; k < add; k++) {
930. bin\_buffer.pop\_back();
931. }
932. if (bin\_buffer == decode\_buffer) {
933. std::cout << "Succeed to recover from bin\_file" << std::endl;
934. decode\_buffer.clear();
935. decode\_buffer = bin\_buffer;
936. }
937. readFile.close();
938. }
939. inline void Save\_Bin\_File() override {
940. Huffman\_Operator::Read\_Huffman\_File("File\_With\_K\_Ray\_Huffman.txt");
941. size = std::floor(std::log2(256)/std::log2(system));
942. int add = size - decode\_buffer.size()%size;
943. for (int i = 0; i<add; i++){
944. decode\_buffer += "0";
945. }
946. std::ofstream outputFile("K\_Ray\_Compact\_File.bin", std::ios::binary);
947. if (!outputFile.is\_open()) {
948. throw std::logic\_error("Failed to open K\_Ray\_Compact\_File.bin");
949. }
950. unsigned char temp = static\_cast<unsigned char>(add);
951. outputFile.write(reinterpret\_cast<const char\*>(&temp), sizeof(temp));
952. bit\_num++;
953. for (size\_t i = 0; i < decode\_buffer.size(); i += size){
954. std::string compact\_buffer = decode\_buffer.substr(i,size);
955. int sum = 0;
956. for(size\_t k = 0; k<compact\_buffer.size(); k++){
957. sum = sum\*system + (compact\_buffer[k]-'0'); *// 将K进制字符串转为整数*
958. }
959. unsigned char byte = static\_cast<unsigned char>(sum);
960. outputFile.write(reinterpret\_cast<const char\*>(&byte), sizeof(byte));
961. bit\_num++;
962. }
963. for (int i = 0; i < add; i++) {
964. decode\_buffer.pop\_back();
965. }
966. outputFile.close();
967. }
969. inline bool Check\_Child(int current){
970. bool flag = false;
971. for(size\_t i = 0; i<Huffman\_Tree[current].relation.size(); i++){
972. if(Huffman\_Tree[current].relation[i]) flag = true;
973. }
974. return flag;
975. }
976. inline void Inital\_Huffman() override{
977. *//std::cout<<"The size of Diction"<<std::endl;*
978. for(int i = 0;i <= leaf\_num;i++){
979. K\_HTNode node;
980. node.weight = Diction[i].weight;
981. node.real\_pos = i;
982. node.parent = -1;
983. for(int j = 0; j <= system; j++){
984. node.relation.push\_back(0);
985. }
986. Huffman\_Tree.push\_back(node);
987. }
988. int add = 0;
989. if(system > 2){
990. add = system-leaf\_num % (system-1);
991. for(int k = leaf\_num + 1; k <= leaf\_num + add; k++){
992. K\_HTNode node;
993. node.weight = 0;
994. node.real\_pos = k;
995. node.parent = -1;
996. Huffman\_Tree.push\_back(node);
997. }
998. used\_leaf\_num = add + leaf\_num;
999. }
1000. *//std::cout<<"valid num"<<Huffman\_Tree.size()-1<<std::endl;*
1001. }
1002. inline void conclusion()override{
1003. double rate = 1-1.0\*bit\_num/(buffer.size());
1004. std::cout<<"compression ratio :"<<rate<<std::endl;
1005. }
1006. };
1007. *// int main(void){*
1008. *//     K\_Ray\_Huffman\_Operator kh(5);*
1009. *//     kh.Generate\_Huffman\_Code();*
1010. *//     return 0;*
1011. *// }*