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** |

与正文无关的碎碎念

数据结构转眼就要结课了，我把这门课当作我的编程启蒙课（嘿嘿），这一学期几乎是完全独立的写出了很多算法和小实验。

我还记得我写实验1的时候，在博客上看到了很好看的八角迷宫设计图形，于是在没有参考博客代码的情况下（只参考了好看的图）写出来了实验1；写实验2哈夫曼树的时候特别好玩，我基础代码写的比较快，但是老师每讲一次课我就发现我代码的漏洞，然后一直在改，前后上了三次课，也改了三次(K叉树的处理，压缩率，文本，嘿嘿)；实验3就很有意思，我试着用堆去优化最短路径算法，但是发现优化后的比优化之前慢了20倍(.....)，后来找助教老师才知道是我的图存储结构不好，邻接矩阵是n\*n\*logn，而且信息冗余。后来没改结构，用邻接表动态建堆，然后快了20倍(嘿嘿)；实验4比较好玩的就是，emm，写着写着实验去看leecode了，哈哈哈哈。

唉，我代码其实挺弱的，大学才接触编程（或者说大学才碰电脑），之前一直是大镇做题家（现在也还是）。数据结构是真正自己动手做一个小东西，感觉挺好的，做出来特别有成就感，我相信我会进步的。

最后，我其实很遗憾数据结构作为一门考察课出现，这就意味着我要把宝贵的时间分给万恶的复变函数与积分变换，而复变考试的邻近和令人崩溃的难度这也意味着我不太可能好好复习准备数据结构的期末考试（悲）。

最后的最后，特别感谢一直为我答疑解惑的老师和助教老师。

锅大侠加油

计算学部加油

哈工大加油

呕吼



实验四 排序算法及应用

## 一、实验内容

**实验题目 1：内排序算法**

**实验内容：**

内存排序算法

实验内容：

排序是计算机科学中的常见任务，它将一组无序的数据元素按照某种规则重

新排列，以使得数据呈现有序的状态，便于后续的查找、统计和分析等操作。当

数据量较小时，将数据全部读入内存并进行排序的算法称为内存排序算法，常见

的内存排序算法有：插入排序、冒泡排序、归并排序、快速排序、堆排序、基数

排序等。本实验要求设计并实现上述内存排序算法并比较其运行速度。

实验要求：

1. 从文本文件中将两行数据读入内存，其中第一行有一个整数 n(n≤100000)，表

示待排序序列的长度，第二行有 n 个整数，用空格隔开，表示待排序序列。

2. 实现归并排序、快速排序算法，输出排序好的序列，并记录算法运行时间。

3. 实现选择排序算法或插入排序算法，并将其运行时间与归并排序、快速排序算

法比较，随机生成多个适当规模的数据进行实验并绘制折线图，反映不同算法运

行时间随着输入规模的变化趋势，并与理论分析结果进行比较。

4. 实现堆排序算法（基于优先级队列）和基数排序算法，并将其运行时

间与上述算法进行实验比较。

5. 对于序列 a1,a2,...an，若 ai>aj(1≤i<j≤n)则称(ai,aj)构成一个逆序对，请改

写归并排序算法使其在对序列排序的同时计算原序列的逆序对数量并输出

## 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，完成了算法的实现，完成了本次实验的全部任务。

**1 数据集**

完我随机生成了满足均匀分布的范围在[-99999,+99999]之间的5组数据，数据规模大小分别为9，99，999，9999，99999个

**2实现的算法**

### (1)归并排序

归并排序是一种分治算法，其基本思想是将两个已经排序的序列合并成一个排序的序列。归并排序的递归实现包括两个主要步骤：分割和合并。在分割步骤中，算法递归地将数组分成两半，直到每个子数组只包含一个元素。在合并步骤中，算法将这些有序的子数组合并成更大的有序数组，直到最终得到一个有序的完整数组。归并排序的时间复杂度为O(n log n).

### (2)归并排序输出逆序数

逆序数是指在一个序列中，前面的元素比后面的元素大的数对的数量。在归并排序的过程中可以计算逆序数。每当在合并两个有序子数组时，如果左侧数组的元素大于右侧数组的元素，就会产生逆序。通过在合并过程中跟踪这些逆序，可以计算出整个数组的逆序数。这种方法的时间复杂度仍然是O(n log n)，但额外的空间复杂度为O(n)，因为需要一个临时数组来存储合并的结果。

### (3)选择排序

选择排序是一种简单直观的排序算法。它的工作原理是每次从待排序的数据元素中选出最小（或最大）的一个元素，存放在序列的起始位置，直到全部待排序的数据元素排完。选择排序的主要优点是它对数据规模的适应性强，即使在数据量较小的情况下，选择排序也能保持较好的性能。选择排序的时间复杂度为O(n^2).在数据快速增长时，相比于其他算法，直接选择排序消耗的时间快速增长

### (4)快速排序

快速排序是一种分治算法，它通过一个划分操作将数据分为独立的两部分，其中一部分的所有数据都比另一部分的所有数据要小，然后递归地对这两部分数据分别进行排序操作。快速排序的平均时间复杂度为O(n log n)，但在最坏情况下会退化到O(n^2)。快速排序的性能通常比其他O(n log n)算法更好，源于其内部循环的快速实现

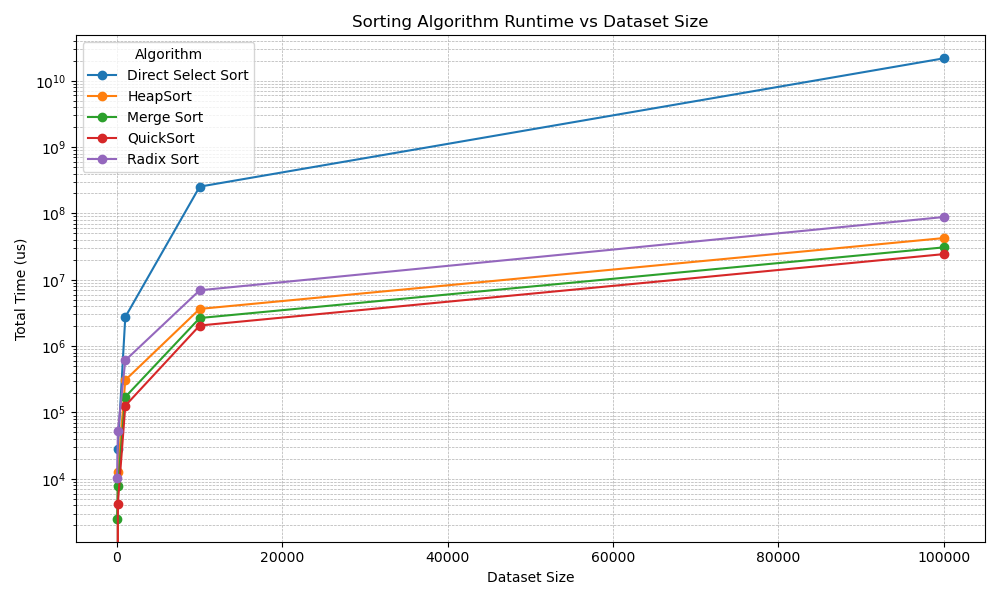
### (5)基数排序

基数排序是一种非比较型整数排序算法，其基本思想是将整数按位数切割成不同的数字，然后按每个位数分别比较。对于给定的位数，基数排序会分配一个桶给每个可能的值，然后根据当前位上的数值将元素分配到对应的桶中。当所有元素都被分配到桶中后，算法会将桶中的元素重新收集到原始数组中。这个过程会重复进行，直到所有位数都被排序。基数排序的时间复杂度为O(d(n+r))

### (6)堆排序

堆排序是利用堆这种数据结构所设计的一种排序算法。堆是一个近似完全二叉树的结构，并同时满足堆积的性质：即子节点的键值或索引总是小于（或者大于）它的父节点。堆排序的基本思想是利用堆的结构特性，将待排序的序列构造成一个大顶堆，此时，整个序列的最大值就是堆顶的根节点。将其与末尾元素进行交换，此时末尾就为最大元素。然后将剩余n-1个元素重新构造成一个大顶堆，这样会得到n个元素中第二大的元素，如此反复执行，便能得到一个有序序列。堆排序的时间复杂度为O(n log n)，其中n是数组中的元素数量。如果初建大根堆，那么最后得到的是升序序列。

1. 结果展示如下：



## 三、实验心得

* **遇到的问题和解决方法**

1. 实验验收的时候，老师指出我的链队优化的基数排序太慢，经过仔细对比，我发现STL库内置的queue（即naive方法下的）排序效果仅次于快排。我分析后发现是我的自定义的链队管理方式较差，在回收的分发的时候做了许多无效操作(....),改进之后效果好一些，但是还是比内置库差很多，大概改之前慢2，5倍，改制后慢1.5倍这样子(....)
2. 在计算逆序对的时候遭遇了溢出，后来开了long long，但是data5的数据总量太大，还是溢出，就选了一道leecode题目验证算法正确性。

* **体会和感受**

通过这次内排序实验，我深刻体会到了不同排序算法的效率和适用场景的差异。归并排序的稳定性和快速排序的高效性给我留下了深刻印象，而基数排序在处理大数据集时的优势也让我认识到算法选择的重要性。这次实践不仅加深了我对排序算法理论的理解，也锻炼了我的编程能力。

1. **源代码**
2. #include "utils.h"
3. #include "Link\_Queue.h"
4. class Sort\_Methods{
5. public:
6. std::vector<int> array;
7. Sort\_Methods(){
8. *// Read\_Data\_From\_txt("E:\\HomeWork\\lab4\\output\\Data1.txt",array);*
9. }
10. void Generate\_Random\_Data(const std::string& filename, int n = 999999, int lower\_bound = -99999, int upper\_bound = 99999) {
11. std::ofstream file(filename);
12. if (!file.is\_open()) {
13. std::cerr << "Error: Could not open "<<filename<< std::endl;
14. return;
15. }
16. std::random\_device rd;
17. std::mt19937 gen(rd());
18. std::uniform\_int\_distribution<int> dist(lower\_bound, upper\_bound);
19. file<<n<<" ";
20. for (int i = 0; i < n; ++i) {
21. file<<dist(gen);
22. if (i != n - 1) file<<" ";
23. }
24. file.close();
25. }
26. void Controller() {
27. Merge(array);
28. Save\_To\_File("Merge\_Sort\_Result.txt", array);
29. QuickSort(array);
30. Save\_To\_File("QuickSort\_Result.txt", array);
31. Direct\_Select\_Sort(array);
32. Save\_To\_File("Direct\_Select\_Sort\_Result.txt", array);
33. HeapSort(array);
34. Save\_To\_File("HeapSort\_Result.txt", array);
35. Radix\_Sort(array, 5);
36. Save\_To\_File("Radix\_Sort\_Result.txt", array);
37. Merge\_Reverse(array);
38. Save\_To\_File("Merge\_Reverse\_Result.txt", array);
39. }

42. void Read\_Data\_From\_txt(std::string addrress,std::vector<int>&array) {
43. std::ifstream file(addrress);
44. if (!file.is\_open()) {
45. std::cerr << "Error: Could not open input.txt" << std::endl;
46. return;
47. }
48. int n;
49. file >> n;
50. array.resize(n);
51. for (int i = 0; i < n; ++i) {
52. file >> array[i];
53. }
54. file.close();
55. std::cout << "Data read from input.txt. Size: " << n << std::endl;
56. *//    Show\_Array(array);*
57. }
58. void Test\_Running\_Time() {
59. *// Read\_Data\_From\_txt("output/Data5.txt");*
60. std::ofstream time\_file("All\_Algorithm\_Running\_Times.csv");
61. time\_file << "Algorithm,TotalTime us" << std::endl;
62. std::vector<std::string>Data = {"E:\\HomeWork\\lab4\\output\\Data5.txt",
63. "E:\\HomeWork\\lab4\\output\\Data4.txt",
64. "E:\\HomeWork\\lab4\\output\\Data3.txt",
65. "E:\\HomeWork\\lab4\\output\\Data2.txt",
66. "E:\\HomeWork\\lab4\\output\\Data1.txt"};
68. *// for(int k = 01 ;k<= 5;k++){*
69. std::vector<int>temp;
70. *// Read\_Data\_From\_txt(Data[k-1],temp);*
71. Read\_Data\_From\_txt(Data[4],temp);
72. *// time\_file<<(k\*10+1)\*9<<" ";*
73. long long mergeTotalTime = 0;
74. long long quickTotalTime = 0;
75. long long directSelectTotalTime = 0;
76. long long heapTotalTime = 0;
77. long long radixTotalTime = 0;
79. std::vector<int> copy = temp;
80. *// auto start1 = std::chrono::high\_resolution\_clock::now();*
81. *// for (size\_t i = 0; i < 1000; i++) {*
82. *//     copy = temp;*
83. *//     Merge(copy);*
84. *// }*
85. *// auto end1 = std::chrono::high\_resolution\_clock::now();*
86. *// mergeTotalTime = std::chrono::duration\_cast<std::chrono::microseconds>(end1 - start1).count();*
88. *// auto start2 = std::chrono::high\_resolution\_clock::now();*
89. *// for (size\_t i = 0; i < 1000; ++i) {*
90. *//     copy = temp;*
91. *//     QuickSort(copy);*
92. *// }*
93. *// auto end2 = std::chrono::high\_resolution\_clock::now();*
94. *// quickTotalTime = std::chrono::duration\_cast<std::chrono::microseconds>(end2 - start2).count();*
95. *// auto start3 = std::chrono::high\_resolution\_clock::now();*
96. *// for (size\_t i = 0; i < 1000; ++i) {*
97. *//     copy = temp;*
98. *//     Direct\_Select\_Sort(copy);*
99. *// }*
100. *// auto end3 = std::chrono::high\_resolution\_clock::now();*
101. *// directSelectTotalTime = std::chrono::duration\_cast<std::chrono::microseconds>(end3 - start3).count();*
102. *// auto start4 = std::chrono::high\_resolution\_clock::now();*
103. *// for (size\_t i = 0; i < 1000; ++i) {*
104. *//     copy = temp;*
105. *//     HeapSort(copy);*
106. *// }*
107. *// auto end4 = std::chrono::high\_resolution\_clock::now();*
108. *// heapTotalTime = std::chrono::duration\_cast<std::chrono::microseconds>(end4 - start4).count();*
109. auto start5 = std::chrono::high\_resolution\_clock::now();
110. for (size\_t i = 0; i < 1000; ++i) {
111. copy = temp;
112. Radix\_Sort(copy,5);
113. }
114. auto end5 = std::chrono::high\_resolution\_clock::now();
115. radixTotalTime = std::chrono::duration\_cast<std::chrono::microseconds>(end5 - start5).count();
116. std::cout<<radixTotalTime<<std::endl;
117. *// time\_file << "Merge Sort," << mergeTotalTime << std::endl;*
118. *// time\_file << "QuickSort," << quickTotalTime << std::endl;*
119. *// time\_file << "Direct Select Sort," << directSelectTotalTime << std::endl;*
120. *// time\_file << "HeapSort," << heapTotalTime << std::endl;*
121. *// time\_file << "Radix Sort," << radixTotalTime << std::endl;*
122. *// }*
123. *// time\_file.close();*
124. *// std::cout << "Algorithm running times have been saved to Algorithm\_Running\_Times.csv" << std::endl;*
125. }
126. void Save\_To\_File(const std::string& filename, const std::vector<int>& array) {
127. std::ofstream file(filename);
128. if (!file.is\_open()){
129. std::cerr << "Error: Could not open " << filename << std::endl;
130. return;
131. }
132. for (int num : array) {
133. file << num << " ";
134. }
135. file.close();
136. std::cout << "Saved result to file: " << filename << std::endl;
137. }
138. void HeapSort(std::vector<int>array){
139. int size = array.size();
140. int start = (size-2)/2;
141. for(int i = start ;i >= 0; i--){
142. Heapfy(array,i,size);
143. }
144. for(int k = size-1; k>0; k--){
145. std::swap(array[k],array[0]);
146. Heapfy(array,0,k);
147. }
148. *// for(int m = 0;m<size;m++) std::cout<<array[m]<<" ";*
150. }
151. void QuickSort(std::vector<int> array){
152. Quick(array,0,array.size()-1);
153. *// Show\_Array(array);*
154. }
155. void Direct\_Select\_Sort(std::vector<int> array){
156. for(int i = 0; i < array.size(); i++){
157. int min\_index = i;
158. for(int j = min\_index; j < array.size(); j++){
159. if(array[min\_index] > array[j]){
160. min\_index = j;
161. }
162. }
163. if(i != min\_index);
164. std::swap(array[min\_index],array[i]);
165. }
166. *// Show\_Array(array);*
167. }
168. void Merge(std::vector<int>array){
169. std::vector<int>temp(array.size());
170. *// std::memset(&temp,0,temp.size());*
171. Merge\_Sort(0,array.size()-1,array,temp);
172. *// Show\_Array(array);*
173. }
174. void Merge\_Reverse(std::vector<int> array){
175. std::vector<int>temp(array.size());
176. long long count = Merge\_Count(0,array.size()-1,array,temp);
177. *// Show\_Array(array);*
178. *// std::cout<<"The total num of the reverse :"<<count<<std::endl;*
179. }
180. void Radix\_Sort(std::vector<int>array,int figure){
181. std::vector<int> positives, negatives;
182. for (int num : array) {
183. if (num >= 0)
184. positives.push\_back(num);
185. else
186. negatives.push\_back(-num);
187. }
188. if (!positives.empty())
189. *// Radix\_Sort\_Update(positives,figure);*
190. Radix\_Sort\_Naive(positives,figure);
191. if (!negatives.empty()) {
192. *// Radix\_Sort\_Update(negatives,figure);*
193. Radix\_Sort\_Naive(positives,figure);
194. for (int& num : negatives)
195. num = -num; *// 恢复为负数*
196. reverse(negatives.begin(), negatives.end()); *// 翻转顺序*
197. }
198. array.clear();
199. array.insert(array.end(), negatives.begin(), negatives.end());
200. array.insert(array.end(), positives.begin(), positives.end());
201. *// Show\_Array(array);*
202. }
203. private:
204. inline int Caculate\_Figure(int k,int p){
205. int power = 1;
206. for(int i = 1;i < p;i++){
207. power \*= 10;
208. }
209. return (k%(power\*10))/power;
210. }
211. inline void Heapfy(std::vector<int>&array,int root,int end){
212. int left = 2\*root + 1;
213. int right = 2\*root +2;
214. int current = root;
215. if(left < end && array[left]>array[current]){
216. current = left;
217. }
218. if(right < end && array[right]>array[current] ){
219. current = right;
220. }
221. if(root!=current){
222. std::swap(array[current],array[root]);
223. Heapfy(array,current,end);
224. }
225. }
226. inline void Quick(std::vector<int> &array,int begin,int end){
227. *//no while here,since begin and end is actuall a constant,*
228. *//the function cannot be popped out of the stack;*
229. if(begin < end){
230. int key = array[end];
231. int i = begin - 1;
232. for(int  j= begin; j<end; j++){
233. if(array[j] < key ){
234. i++;
235. std::swap(array[i],array[j]);
236. }
237. }
238. std::swap(array[i+1],array[end]);
239. int next\_pos = i + 1;
240. Quick(array,begin,next\_pos-1);
241. Quick(array,next\_pos+1,end);
242. }
243. }
244. inline int Merge\_Sort(int l,int r,std::vector<int> &array,std::vector<int> &temp){
245. int mid = (l + r)/2;
246. if(l < r){
247. Merge\_Sort(l,mid,array,temp);
248. Merge\_Sort(mid+1,r,array,temp);
249. Merge\_Two(l,mid,r,array,temp);
250. }
251. }
252. inline void Merge\_Two(int l,int m,int r,std::vector<int>&array,std::vector<int>&temp){
253. int i = l, j = m+1, k = l;
254. while(i <= m && j <= r){
255. temp[k++] = array[i] <= array[j] ? array[i++]:array[j++];
256. }
257. while(i <= m) temp[k++] = array[i++];
258. while(j <= r) temp[k++] = array[j++];
259. for(int i = l; i <= r; i++){
260. array[i] = temp[i];
261. }
262. }
263. inline int Merge\_Count(int l,int r,std::vector<int> &array,std::vector<int> &temp){
264. int mid = (l + r)/2;
265. long long count = 0;
266. if(l < r){
267. count += Merge\_Count(l,mid,array,temp);
268. count += Merge\_Count(mid+1,r,array,temp);
269. count += Merge\_Two\_Count(l,mid,r,array,temp);
270. return count;
271. }
272. return 0;
273. }
274. inline int Merge\_Two\_Count(int l,int m,int r,std::vector<int>&array,std::vector<int>&temp){
275. int i = l,j = m + 1,k = l;
276. int count = 0;
277. while(i <= m && j <= r){
278. if(array[i] < array[j]){
279. temp[k++] = array[i++];
280. }else{
281. count += (m - i + 1);
282. temp[k++] = array[j++];
283. }
284. }
285. while(i <= m) temp[k++] = array[i++];
286. while(j <= r) temp[k++] = array[j++];
287. for(int w = l; w <= r; w++){
288. array[w] = temp[w];
289. }
290. return count;
291. }
292. inline void Show\_Array(std::vector<int>& array){
293. for(int num:array){
294. std::cout<<num<<" ";
295. }
296. std::cout<<std::endl;
297. }
298. inline void Radix\_Sort\_Update(std::vector<int>&array,int figure){
299. std::vector<Queue>QL(10);
300. Queue temp;
301. for(int num:array) temp.EnQueue(num);
303. for(int pass = 1; pass <=figure; pass++){
304. while (!temp.IsEmpty()){
305. int data = temp.DeQueue();
306. int r = Caculate\_Figure(data,pass);
307. QL[r].EnQueue(data);
308. }
309. temp.Clear();
310. for(int k = 0;k <= 9; k++){
311. while(!QL[k].IsEmpty()){
312. temp.Concentrate(QL[k]);
313. }
314. }
315. }
316. for(size\_t i = 0; i < array.size();i++) {
317. *// while(!temp.IsEmpty())*
318. *//bug covering old value*
319. array[i] = temp.DeQueue();
320. }
321. *// std::cout<<"Radix\_Sort"<<std::endl;*
322. *// Show\_Array(array);*
323. }
324. inline void Radix\_Sort\_Naive(std::vector<int>&array,int figure){
325. std::vector<std::queue<int>>QL(10);
326. for(int pass = 1; pass <=figure; pass++){
327. for(int data:array){
328. int r = Caculate\_Figure(data,pass);
329. QL[r].push(data);
330. }
331. array.clear();
332. for(int k = 0;k <= 9; k++){
333. while(!QL[k].empty()){
334. array.push\_back(QL[k].front());
335. QL[k].pop();
336. }
337. }
338. *// Show\_Array(array);*
339. }
340. *// std::cout<<"Radix\_Sort"<<std::endl;*
341. *// Show\_Array(array);*
342. }
343. };
344. int main(void){
345. Sort\_Methods solution;
346. *// solution.Generate\_Random\_Data("Data5.txt");*
347. *// solution.Controller();*
348. *// solution.Test\_Running\_Time();*
349. solution.Test\_Running\_Time();
351. return 0;
352. }
353. #pragma once
354. #include <iostream>
355. #include <vector>
356. #include <chrono>
357. #include <algorithm>
358. #include <queue>
359. #include <stdexcept>
360. #include <random>
361. #include <fstream>
362. #include <cmath>
363. *// #pragma once*
364. *// #include "utils.h"*
365. *// struct ListNode {*
366. *//     int data;*
367. *//     struct ListNode \*next;*
368. *// };*
369. *// typedef struct ListNode Node;*
370. *// class Queue{*
371. *// public:*
372. *//     Queue():front(nullptr),rear(nullptr){}*
373. *//     ~Queue(){*
374. *//         Clear();*
375. *//     }*
376. *// private:*
377. *//     Node \*front;*
378. *//     Node \*rear;*
379. *// public:*
380. *//     bool Is\_Empty()const{*
381. *//         return front == nullptr;*
382. *//     }*
383. *//     void EnQueue(int value){*
384. *//         Node \*p = new Node;*
385. *//         p->data = value;*
386. *//         if(Is\_Empty()){*
387. *//             front = rear = p;*
388. *//         }else{*
389. *//             rear->next = p;*
390. *//             rear = p;*
391. *//         }*
392. *//     }*
393. *//     int DeQueue(){*
394. *//         if(Is\_Empty()) std::cerr<<"Error";*
395. *//         Node \*temp = front;*
396. *//         int value = temp->data;*
397. *//         front = front->next;*
398. *//         if(front == nullptr) rear = front;*
399. *//         delete temp;*
400. *//         return value;*
401. *//     }*
402. *//     void Concentrate(Queue &son){*
403. *//         if(!son.Is\_Empty()){*
404. *//             if(Is\_Empty()){*
405. *//                 front = son.front;*
406. *//                 rear = son.rear;*
407. *//             }else{*
408. *//                 rear->next = son.front;*
409. *//                 rear = son.rear;*
410. *//                 rear->next = nullptr;*
411. *//             }*
412. *//         }*
413. *//     }*
414. *//     inline void Clear(){*
415. *//         while(!Is\_Empty()) DeQueue();*
416. *//     }*
417. *// };*
418. *// // int main(void){*
419. *// //     Queue Q,P;*
420. *// //     for(int i = 0;i<10;i++){*
421. *// //         Q.EnQueue(i);*
422. *// //         P.EnQueue(i+20);*
423. *// //     }*
424. *// //     Q.Concentrate(P);*
425. *// //     while(!Q.Is\_Empty())std::cout<<Q.DeQueue()<<" ";*
426. *// //     return 0;*
427. *// // }*
428. #pragma once
429. #include <iostream>
430. #include <stdexcept>
431. struct ListNode {
432. int data;
433. ListNode \*next;
434. ListNode(int val) : data(val), next(nullptr) {}
435. };
436. typedef ListNode Node;
437. class Queue{
438. public:
439. Queue() : front(nullptr), rear(nullptr) {}
440. ~Queue(){
441. Clear();
442. }
443. bool IsEmpty() const {
444. return front == nullptr;
445. }
446. void EnQueue(int value){
447. Node \*p = new Node(value);
448. if(IsEmpty()){
449. front = rear = p;
450. }else{
451. rear->next = p;
452. rear = p;
453. }
454. }
455. int DeQueue(){
456. if(IsEmpty()){
457. std::cerr << "Error: Attempt to dequeue from an empty queue." << std::endl;
458. throw std::underflow\_error("Queue is empty");
459. }
460. Node \*temp = front;
461. int value = temp->data;
462. front = front->next;
463. if(front == nullptr) rear = nullptr;
464. delete temp;
465. return value;
466. }
467. void Concentrate(Queue &son){
468. if(!son.IsEmpty()){
469. if(IsEmpty()){
470. front = son.front;
471. rear = son.rear;
472. }else{
473. rear->next = son.front;
474. rear = son.rear;
475. }
476. son.front = son.rear = nullptr;
477. }
478. }
479. void Clear(){
480. while(!IsEmpty()) DeQueue();
481. }
482. private:
483. Node \*front;
484. Node \*rear;
485. };
486. import matplotlib.pyplot as plt
487. import pandas as pd
488. import numpy as np
489. data = {
490. "Algorithm": [
491. "Merge Sort", "QuickSort", "Direct Select Sort", "HeapSort", "Radix Sort",
492. "Merge Sort", "QuickSort", "Direct Select Sort", "HeapSort", "Radix Sort",
493. "Merge Sort", "QuickSort", "Direct Select Sort", "HeapSort", "Radix Sort",
494. "Merge Sort", "QuickSort", "Direct Select Sort", "HeapSort", "Radix Sort",
495. "Merge Sort", "QuickSort", "Direct Select Sort", "HeapSort", "Radix Sort"
496. ],
497. "TotalTime\_us": [
498. 2508, 0, 0, 0, 10308,
499. 7829, 4225, 28470, 12632, 52125,
500. 171445, 126415, 2744809, 308029, 610572,
501. 2653728, 2046890, 253548867, 3641849, 6948423,
502. 30796488, 24367688, 21756034745, 42528320, 88327503
503. ]
504. }
505. df = pd.DataFrame(data)
506. dataset\_sizes = [9, 99, 999, 9999, 99999]
507. df['DatasetSize'] = np.repeat(dataset\_sizes, 5)
508. pivot\_df = df.pivot(index="DatasetSize", columns="Algorithm", values="TotalTime\_us")
509. plt.figure(figsize=(10, 6))
510. for algorithm in pivot\_df.columns:
511. plt.plot(pivot\_df.index, pivot\_df[algorithm], label=algorithm, marker='o')
512. plt.title("Sorting Algorithm Runtime vs Dataset Size")
513. plt.xlabel("Dataset Size")
514. plt.ylabel("Total Time (us)")
515. plt.yscale("log")  # Use a logarithmic scale for better visualization
516. plt.legend(title="Algorithm")
517. plt.grid(True, which="both", linestyle="--", linewidth=0.5)
518. plt.tight\_layout()
519. plt.savefig("sorting\_algorithms\_runtime.png")
520. plt.show()