**数据结构实验报告**

|  |  |  |  |
| --- | --- | --- | --- |
| 学号-姓名 | 桑龙龙-20030540015 | 实验时间 | 2020年 12月 12日 |
| 诚信声明 | 本实验及实验报告所写内容为本人所作 | | |
| 实验题目 | 实验六  常用的排序方法  题目一 简单排序方法  题目二 快速排序  题目三  堆排序  题目四  归并排序 | | |
| 实验过程中遇到的主要问题 | 无 | | |
| 实验小结 | 本次试验进行了常见排序算法的学习。简单排序方法有直接插入排序，选择排序，冒泡排序，他们的时间复杂度都是O(n2)，一般不不会作为我们的排序选择。快速排序、堆排序、归并排序都有优秀的时间复杂度O（nlog2n），其中归并排序是稳定的排序算法，快速排序、堆排序都不是稳定的排序算法。以前认识中，快速排序是相较其他两者较为优秀的排序算法，但是在1e6及以下数据的测试下，归并排序与其他两者相比有更低的时间消耗，相比起来较小的数据下，归并排序的额外空间消耗并不重要。在测试中，归并排序，堆排序有更为稳定的性能，不会因为极端数据导致时间复杂度降低，依旧保持在O（nlog2n），但是快速排序在极端情况下时间复杂度会降低为O(n2)。  测试数据的结果见下表。 | | |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **O（n2）排序方式** | | | | | | | | |
| 排序方式 | 数据描述 | 数据规模 | 组数 | 总耗时（ms) | 总比较次数 | 总交换次数 | 平均比较 次数 | 平均交换 次数 |
| 选择排序 | 随机数据 | 1000 | 100 | 288 | - | - | - | - |
| 直接插 入排序 | 随机数据 | 1000 | 1000 | 1028 | 8577408 | 245394856 | 8577.408 | 245394.9 |
| 正序 | 1000 | 1000 | 72 | 8977000 | 0 | 8977 | 0 |
| 逆序 | 1000 | 1000 | 1976 | 7990689 | 499489906 | 7990.689 | 499489.9 |
| 冒泡排序 | 随机数据 | 1000 | 100 | 584 | 49879385 | 24833792 | 498793.9 | 248337.9 |
| 正序 | 1000 | 100 | 4 | 99900 | 0 | 999 | 0 |
| 逆序 | 1000 | 100 | 748 | 49949900 | 49949104 | 499499 | 499491 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **快速排序** | | | | | | | |
| 数据描述 | 数据规模 | 组数 | 总耗时（ms) | 总比较次数 | 总交换次数 | 平均比较 次数 | 平均交换 次数 |
| 随机数据 | 10000 | 100 | 224 | 15246177 | 3159284 | 152461.8 | 31592.84 |
| 10000 | 100 | 220 | 16190000 | 3135100 | 161900 | 31351 |
| 10000 | 100 | 216 | 15512900 | 3143800 | 155129 | 31438 |
| 正序 | 10000 | 1 | 240 | 49995000 | 9999 | 49995000 | 9999 |
| 逆序 | 10000 | 1 | 232 | 49995000 | 9999 | 49995000 | 9999 |
| 随机数据 | 100000 | 10 | 272 | 19996200 | 3921627 | 1999620 | 392162.7 |
| 100000 | 10 | 268 | 20527240 | 3912790 | 2052724 | 391279 |
| 100000 | 10 | 288 | 19778060 | 3935520 | 1977806 | 393552 |
| 随机数据 | 1000000 | 1 | 316 | 24273048 | 4712212 | 24273048 | 4712212 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **归并排序** | | | | | |
| 数据描述 | 数据规模 | 组数 | 排序总耗时（ms) | 总比较次数 | 平均比较 次数 |
| 随机数据 | 10000 | 100 | 196 | 12047200 | 1204.72 |
| 10000 | 100 | 204 | 12045777 | 1204.5777 |
| 10000 | 100 | 188 | 12045600 | 1204.56 |
| 随机数据 | 100000 | 10 | 236 | 15361180 | 153.6118 |
| 随机数据 | 1000000 | 1 | 280 | 18673989 | 18.673989 |
| 随机数据 | 1000000 | 1 | 268 | 18674525 | 18.674525 |
| 正序 | 10000 | 100 | 136 | 6467100 | 646.71 |
| 逆序 | 10000 | 100 | 120 | 6900800 | 690.08 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **堆排序（大顶堆）** | | | | | | | |
| 数据描述 | 数据规模 | 组数 | 总耗时（ms) | 总比较次数 | 总交换次数 | 平均比较 次数 | 平均交换 次数 |
| 随机数据 | 10000 | 100 | 444 | 17275000 | 11958700 | 172750 | 119587 |
| 10000 | 100 | 440 | 17265006 | 11946978 | 172650.06 | 119469.78 |
| 10000 | 100 | 456 | 17285014 | 11957550 | 172850.14 | 119575.5 |
| 100000 | 10 | 480 | 20793880 | 13974830 | 2079388 | 1397483 |
| 100000 | 10 | 484 | 20794280 | 13974460 | 2079428 | 1397446 |
| 100000 | 10 | 484 | 20793570 | 13973830 | 2079357 | 1397383 |
| 1000000 | 1 | 588 | 25765564 | 17332930 | 25765564 | 17332930 |
| 由大到 小插入 | 10000 | 100 | 380 | 15788000 | 10669700 | 157880 | 106697 |
| 10000 | 100 | 388 | 15787100 | 10668500 | 157871 | 106685 |
| 10000 | 100 | 388 | 15787515 | 10671405 | 157875.15 | 106714.05 |
| 100000 | 10 | 484 | 20794110 | 13974830 | 2079411 | 1397483 |
| 100000 | 10 | 480 | 20793000 | 13974370 | 2079300 | 1397437 |
| 100000 | 10 | 484 | 20794350 | 13974820 | 2079435 | 1397482 |
| 1000000 | 1 | 604 | 25766214 | 17333321 | 25766214 | 17333321 |
| 由小到 大插入 | 10000 | 100 | 680 | 26684476 | 21719359 | 266844.76 | 217193.59 |
| 10000 | 100 | 668 | 26681900 | 21716600 | 266819 | 217166 |
| 10000 | 100 | 604 | 26687700 | 21726300 | 266877 | 217263 |
| 100000 | 10 | 784 | 35024745 | 28371000 | 3502474.5 | 2837100 |
| 100000 | 10 | 780 | 35024339 | 28370084 | 3502433.9 | 2837008.4 |
| 100000 | 10 | 792 | 35024100 | 28370289 | 3502410 | 2837028.9 |
| 1000000 | 1 | 988 | 43283001 | 34894619 | 43283001 | 34894619 |

|  |  |
| --- | --- |
| 数据结构  （自定义数据类型） | 无 |
| 主要算法  （或算法说明） | 1. /\* 2. 1、选择排序 3. 2、直接插入排序 4. 3、冒泡排序 5. 4、快速排序 6. \*/ 8. #include <stdio.h> 9. #include <stdlib.h> 11. **template** <**typename** T> 12. **void** selectSort(T\* arr,**int** lower,**int** upper,**bool**(\*cmp)(T&,T&)){ 13. //选择排序 14. //对[arr+lower,arr+upper)范围进行排序 15. **int** v; 16. **for**(**int** i=upper-1;i>=lower;i--){ 17. v=i; 18. **for**(**int** j=lower;j<i;j++){ 19. **if**(cmp(arr[v],arr[j])) v=j; 20. } 21. std::swap(arr[i],arr[v]); 22. } 23. } 25. **template** <**typename** T> 26. **void** insertSort(T\* arr,**int** lower,**int** upper,**bool**(\*cmp)(T&,T&)){ 27. //直接插入排序 28. //对[arr+lower,arr+upper)范围进行排序 29. **int** low,high,mid,tar; 30. T temp; 31. **for**(**int** i=lower+1;i<upper;i++){ 32. low=lower,high=i-1,tar=i,temp=arr[i]; 33. **while**(high>=low){ 34. //二分查找查找插入位置 35. mid=low+(high-low)/2; 36. **if**(cmp(arr[i],arr[mid])){ 37. tar=mid; 38. high=mid-1; 39. }**else**{ 40. low=mid+1; 41. } 42. } 43. **for**(**int** j=i-1;j>=tar;j--) arr[j+1]=arr[j]; 44. arr[tar]=temp; 45. } 46. } 48. **template** <**typename** T> 49. **void** bubbleSort(T\* arr,**int** lower,**int** upper,**bool**(\*cmp)(T&,T&)){ 50. //冒泡排序 51. //对[arr+lower,arr+upper)范围进行排序 52. **bool** flag=**true**; 53. /\* 54. 标识是否已经有序 55. 如果有序提前退出 56. \*/ 57. **for**(**int** i=upper-1;i>1 && flag;i--){ 58. flag=**false**; 59. **for**(**int** j=0;j<i;j++){ 60. **if**(cmp(arr[j+1],arr[j])) std::swap(arr[j+1],arr[j]),flag=**true**; 61. } 62. } 63. }  66. **template** <**typename** T> 67. **void** quickSort(T\* arr,**int** lower,**int** upper,**bool**(\*cmp)(T&,T&)){ 68. //快速排序 69. //排序范围[arr+lower,arr+upper) 70. //以arr[lower]作为pivot 71. **if**(lower>=upper) **return**; 72. **int** low=lower,high=upper-1; 73. **while**(high>low){ 74. **while**(high>low && !cmp(arr[high],arr[lower])) high--; 75. **while**(high>low && !cmp(arr[lower],arr[low])) low++; 76. std::swap(arr[low],arr[high]); 77. } 78. std::swap(arr[low],arr[lower]); 79. quickSort(arr,lower,low,cmp); 80. quickSort(arr,low+1,upper,cmp); 81. } 83. **int** main(){ 84. } 85. /\* 86. 5、归并排序 87. \*/ 88. #include <stdio.h> 89. #include <stdlib.h> 90. #include <string.h> 91. #define N 100005 92. **using** T=**int**; 93. T arr[N],temp[N]; 94. **void** mergeSort(T\* arr,**int** lower,**int** upper,**bool** (\*cmp)(T&,T&b)){ 95. //归并排序 96. //排序范围[arr+lower,arr+upper) 97. **if**(1+lower>=upper) **return**; 98. **int** mid=lower+(upper-lower)/2; 99. mergeSort(arr,lower,mid,cmp); 100. mergeSort(arr,mid,upper,cmp); 101. **int** i=lower,j=mid,k=lower; 102. **while**(i<mid && j<upper){ 103. **if**(cmp(arr[i],arr[j])) temp[k++]=arr[i++]; 104. **else** temp[k++]=arr[j++]; 105. } 106. **while**(i<mid) temp[k++]=arr[i++]; 107. **while**(j<upper) temp[k++]=arr[j++]; 108. memcpy(arr+lower,temp+lower,**sizeof**(T)\*(upper-lower)); 109. } 111. **int** main(){ 113. } 114. /\* 115. 6、堆排序 116. \*/ 117. #include <stdio.h> 118. #include <stdlib.h> 119. #include <time.h> 120. #include <limits.h> 121. #define N 100005 123. **template** <**class** T> 124. **class** heap{ 125. **private**: 126. **bool** (\*cmp)(T&,T&); 127. //堆排序的比较函数 128. **int** tot,capacity; 129. //tot是目前元素个数，capacity是堆排序实际大小 130. T\* arr; 131. **bool** full(){ 132. //堆空间是否已满 133. **return** tot==capacity-1; 134. } 135. **public**: 136. heap(**bool**(\*cmp)(T&,T&)){ 137. **this**->cmp=cmp; 138. arr=**new** T[N]; 139. tot=0; 140. capacity=N; 141. } 142. heap(**int** n,**bool**(\*cmp)(T&,T&)){ 143. **this**->cmp=cmp; 144. arr=**new** T[n]; 145. tot=0; 146. capacity=n; 147. } 148. ~heap(){ 149. **delete** []arr; 150. } 151. **void** clear(){ 152. //清空堆中元素 153. tot=0; 154. } 156. **bool** push(T val){ 157. /\* 158. 将val插入到堆中，插入成功返回true 159. 失败返回false 160. \*/ 161. **if**(full()) **return** **false**; 162. arr[++tot]=val; 163. **int** i=tot; 164. **while**(i!=1 && cmp(arr[i/2],arr[i])){ 165. std::swap(arr[i/2],arr[i]); 166. i/=2; 167. } 168. **return** **true**; 169. } 170. T& top(){ 171. //返回堆顶元素 172. **return** arr[1]; 173. } 174. **bool** empty(){ 175. //检查堆是否为空 176. **return** tot==0; 177. } 178. **int** size(){ 179. //返回堆大小 180. **return** tot; 181. } 182. **void** pop(){ 183. //删除堆顶元素 184. **if**(empty()) **return**; 185. std::swap(arr[1],arr[tot]); 186. tot=max(tot-1,0); 187. **int** i=1; 188. **while**(i\*2<=tot){ 189. i\*=2; 190. **if**(i+1<=tot && cmp(arr[i],arr[i+1])) i++; 191. **if**(cmp(arr[i/2],arr[i])) std::swap(arr[i/2],arr[i]); 192. **else** **break**; 193. } 194. } 195. };  198. **bool** cmp(**int** &a,**int** &b){ 199. **return** a<b; 200. } 201. **void** randTest(**int** n,heap<**int**>& Q){ 202. srand(time(0)); 203. Q.clear(); 204. printf("insert:\n"); 205. **int** v; 206. **for**(**int** i=0;i<n;i++){ 207. //建立堆 208. v=rand()%INT\_MAX; 209. printf("%d ",v); 210. Q.push(v); 211. } 212. printf("\n\nheap sorted:\n"); 213. **while**(!Q.empty()){ 214. //弹出元素，堆排序完成 215. printf("%d ",Q.top()); 216. Q.pop(); 217. } 218. } 219. **int** main(){ 220. heap<**int**> Q(10000,cmp); 221. //创建一个空间为10000的大顶堆 222. randTest(20,Q); 223. //测试 224. } |