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

课程类型：必修

试验项目名称：搜索与排序

实验题目：二叉搜索树

班级：1603005

学号：1160300524

姓名：冯运

1. 实验目的

通过建立和维护一棵二叉搜索树，熟悉二叉搜索树的插入，删除，搜索的功能，相较于遍历查找，二叉搜索树的搜索效率很高。

1. 实验要求及实验环境

由用户输入想要插入的数据，根据输入的数据建立二叉搜索树，可以删除节点和查询关键字，允许有相同的关键字。

实验环境：寝室，机房，Clion集成开发环境

1. 设计思想

我的二叉搜索树采用了平衡插入的方式，在每插入一个节点时，需要判断这棵树的平衡因子，如果发现最靠近叶节点的节点出现不平衡，通过左旋和右旋的方式将树再平衡，平衡的目的主要是要尽量降低树的层数，使得搜索的时间复杂度平均和最坏情况都是Olog(n),比不平衡的最坏情况O(n)要好得多。

具体的实现方式是：递归的寻找要插入的具体位置，如果遇到已经存的的关键字，就将他的计数加1，否则在该节点处插入一个新的节点，然后将bool 变量 addFloor 设为true, 这个变量将用来重新调整其他节点的平衡因子。在递归返回的过程中，同时修改平衡因子，如果遇到不平衡的，就将其通过具体的判断，采用（LL,LR,RL,RR）之一的方式将其重新调整为平衡。

由于在实现删除时，时间比较局促，没有实现删除时的再平衡，只是找到节点并删除，如果后序还有时间，我会将其实现。

主要的数据类型

定义了一个class node 表示二叉搜索树的一个节点

1. **class** node{
2. #define infinite 0x0fffffff
3. **public**:
4. **int** balancedFactor;  //AVL的平衡因子
5. **int** data;    //节点的数据
6. **int** sameNum;  //表示与该节点有相同关键字的个数
7. node \*lchild , \*rchild;
8. node()   //将节点的数据初始化
9. {
10. lchild = nullptr;
11. rchild = nullptr;
12. data=infinite;
13. sameNum=1;
14. balancedFactor=0;
15. }
16. };
17. **class** AVL {
18. #define LH 1   //表示左面的树高
19. #define EH 0   //表示两面的树一样高
20. #define RH -1  //表示右面的树高
21. **public**:
22. **class** node{
23. ……
24. };
26. node \*root;
27. **int** size;
29. AVL()   //初始化树的节点
30. {
31. root = **new** node;
32. size = 0;
33. }

类内包含了插入，删除，搜索，以及平衡树的几个函数

重要函数的解释：

这个函数是用于外部调用的插入函数，还有一个private属性的内部递归调用的相似函数（node \*insertRecursive(node \*tree, int insertNum, bool \*addFloor)），实现方式相近，只是传入的参数较多，不适合类外调用。

1. **void** insert(**int** insertNum) {
2. size++;
3. **bool** addFloor = **false**;
4. **if** (root->data == infinite)    //在第一次插入节点时，需要特殊处理, 如果是根节点没有数据,则他的值是宏 infinite
5. {
6. root->data = insertNum;
7. **return**;
8. }
9. **if** (root->data == insertNum)   //如果有相同的关键字，就在该节点的记录重复的数据上加1
10. {
11. root->sameNum++;
12. **return**;
13. } **else** **if** (insertNum < root->data) {    //插入的节点在该节点的左子树上
14. root->lchild = insertRecursive(root->lchild, insertNum, &addFloor);
15. **if** (addFloor) {
16. **switch** (root->balancedFactor)        //addFloor  是表示插入新节点导致层数增加，根据原来的状态来判断现在的平衡因子
17. {
18. **case** LH:    //原本左子树比右子树高，需要作左平衡处理
19. root = leftBalanced(root);    //进行左平衡
20. addFloor = **false**;
21. **break**;
22. **case** EH:   //原本左、右子树等高，现因左子树增高而使树增高
23. root->balancedFactor = LH;
24. addFloor = **true**;
25. **break**;
26. **case** RH:   //原本右子树比左子树高，现左、右子树等高
27. root->balancedFactor = EH;
28. addFloor = **false**;
29. **break**;
30. }
31. }
32. } **else** {
33. root->rchild = insertRecursive(root->rchild, insertNum, &addFloor);
34. **if** (addFloor) {
35. **switch** (root->balancedFactor) {
36. **case** LH:    //原本左子树比右子树高，现左、右子树等高
37. root->balancedFactor = EH;
38. addFloor = **false**;
39. **break**;
40. **case** EH:    //原本左、右子树等高，现因右子树增高而使树增高
41. root->balancedFactor = RH;
42. addFloor = **true**;
43. **break**;
44. **case** RH:   //原本右子树比左子树高，需要作右平衡处理
45. root = rightBalanced(root);   //进行右平衡
46. addFloor = **false**;
47. **break**;
48. }
49. }
50. }
51. }

这个函数负责调整左子树上的不平衡现象

1. node \*leftBalanced(node \*tree) {
2. node \*l = tree->lchild, \*lr = l->rchild;    //l表示tree的左子树 lr表示左子树的右子树
3. **switch** (l->balancedFactor)             //根据l的平衡因子来判断  是LL旋转还是LR旋转的方式来调节树的平衡
4. {
5. **case** LH:
6. rightRotate(&tree);       //是左子树的左子树导致不平衡，是LL的旋转方式
7. tree->balancedFactor = l->balancedFactor = EH;
8. **break**;
9. **case** RH:
10. **switch** (lr->balancedFactor)   //调节tree和l的平衡因子
11. {
12. **case** LH:
13. l->balancedFactor = EH;
14. tree->balancedFactor = RH;
15. **break**;
16. **case** EH:
17. l->balancedFactor = tree->balancedFactor = EH;
18. **break**;
19. **case** RH:
20. l->balancedFactor = LH;
21. tree->balancedFactor = EH;
22. **break**;
23. }
24. lr->balancedFactor = EH;
25. leftRotate(&l);    //是LR的情况
26. tree->lchild = l;
27. rightRotate(&tree);
29. }
30. **return** tree;
31. }

这个函数负责右子树上的不平衡现象，与上面的函数基本相似，只是相对称

1. node \*rightBalanced(node \*tree) {
2. node \*r = tree->rchild, \*rl = r->lchild;
3. **switch** (r->balancedFactor) {
4. **case** RH:
5. leftRotate(&tree);
6. tree->balancedFactor = r->balancedFactor = EH;     //是RR的情况
7. **break**;
8. **case** LH:
9. **switch** (rl->balancedFactor) {
10. **case** LH:
11. tree->balancedFactor = EH;
12. r->balancedFactor = RH;
13. **break**;
14. **case** EH:
15. tree->balancedFactor = r->balancedFactor = EH;
16. **break**;
17. **case** RH:
18. tree->balancedFactor = LH;
19. r->balancedFactor = EH;
20. **break**;
21. }
22. rl->balancedFactor = EH;
23. rightRotate(&r);
24. tree->rchild = r;   //是RL的旋转调节平衡方式
25. leftRotate(&tree);
26. }
27. **return** tree;
28. }

这个删除函数暂时为对非平衡的情况处理

1. //负责删除节点的函数
2. **void** Delete(node \*\*tree, **int** deleteNum) {
3. **if** (!\*tree)
4. **return**;
5. node \*pr, \*p;
6. **if** (empty())
7. **return**;
8. **if** ((\*tree)->data == deleteNum) {
9. node \*temp;
10. size--;
11. **if** ((\*tree)->sameNum > 1) {    //如果遇到该关键词有多个相同的,需要将count减1
12. (\*tree)->sameNum--;
13. } **else** {
14. **if** (!(\*tree)->lchild && !(\*tree)->rchild) {    //如果该节点是叶节点的情况,直接删除节点
15. **if** ((\*tree) == root)
16. (\*tree)->data = infinite;
17. **else** {
18. temp = \*tree;
19. **delete** (temp);
20. \*tree = nullptr;
21. }
22. } **else** **if** (!(\*tree)->lchild) {      //如果是只有左子树,没有右子树
23. temp=\*tree;
24. \*tree=(\*tree)->rchild;
25. **delete** (temp);
26. } **else** **if** (!(\*tree)->rchild) {    //只有右子树,没有左子树
27. temp = \*tree;
28. \*tree=(\*temp).lchild;
29. **delete** (temp);
30. } **else** {    //既有左子树,又有右子树,需要找到右子树中最左的节点,将其删除
31. pr = p = (\*tree)->rchild;
32. **while** (p->lchild) {
33. pr = p;
34. p = p->lchild;
35. }
36. (\*tree)->data = p->data;
37. (\*tree)->sameNum = p->sameNum;
38. **if** (pr != p) {       //递归左子树,寻找匹配的节点
39. pr->lchild = p->rchild;
40. } **else** {       //递归左子树,寻找匹配的节点
41. (\*tree)->rchild = p->rchild;
42. }
43. **delete** (p);
44. }
45. }
46. } **else** **if** (deleteNum < (\*tree)->data) {
47. Delete(&(\*tree)->lchild, deleteNum);
48. } **else** {
49. Delete(&(\*tree)->rchild, deleteNum);
50. }
51. }
52. 实验结果

对于建立的树是否平衡，我是通过设置断点，通过debug模式查看内存，然后画在智商验证树的结构是否平衡。

通过多种数据验证，我建立的搜索树是平衡的

图4-1是输入一段乱序的数字，通过中序遍历二叉搜索树得到排序后的数字（该测试用例已经包含了相同关键词的情况）

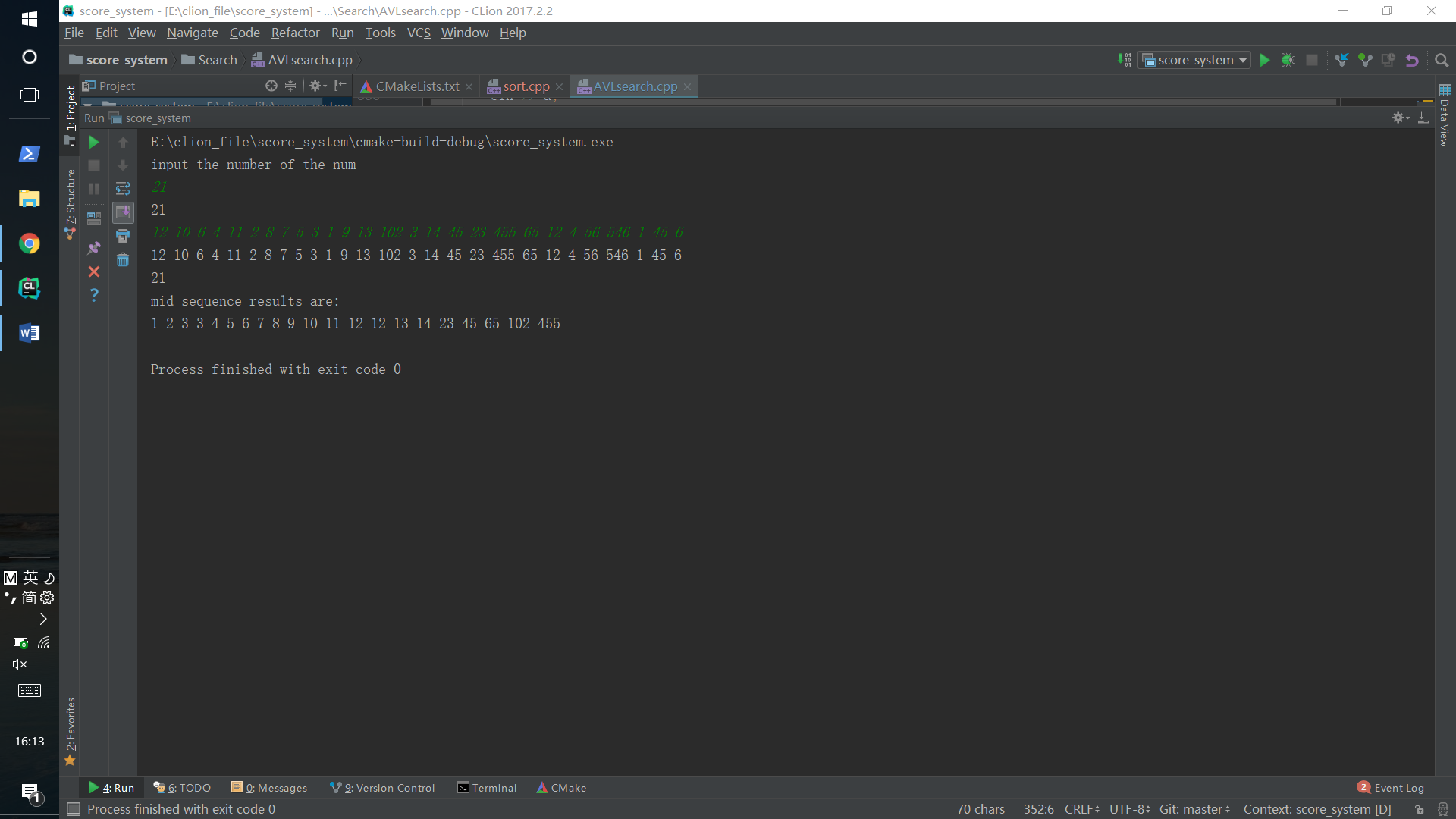


图4-1

1. 系统不足和经验体会

不足:

1、输入的交互不完善，只能验证程序的各个函数是正确的，不能作为工具来使用。

2、相同的关键词没有用链表来存储，输入的信息不完善，只有数字，不支持多种类型。

3、输的删除没有平衡，导致删除的过程可能使得树成为一条单链，使得树的搜索变慢

经验体会：在使用delete()删除节点时，需要考虑他的父节点的指针是否有正确的指向，否则将会出现bug.

平衡树的平衡过程过于复杂，维护和开发过程会很艰难，最好选用一些有相同效果的其他树，比如红黑树，可以提高开发效率。

1. 源代码
2. //
3. // Created by Fitz on 2017/12/14.
4. //
5. /\*
6. \* 在构建平衡二叉树的过程中，每当插入一个结点时，先检查是否因插入而破坏了树的平衡性，若是，则找出最小不平衡子树(通过递归的方式)，在保持二叉排序树特性的前提下，调整关系。
7. 这句话意味着：只要破坏了平衡性，就马上修改使得二叉树重新平衡，意思就是只要修改了最小不平衡树就可以使得整个二叉树重新平衡.
8. \*/
9. #include <iostream>
10. #include <cstdlib>
11. #include <fstream>
13. **using** **namespace** std;
14. #define nullptr NULL

17. **class** AVL {
18. #define LH 1
19. #define EH 0
20. #define RH -1
21. **public**:
22. **class** node {
23. #define infinite 0x0fffffff
24. **public**:
25. **int** balancedFactor;  //AVL的平衡因子
26. **int** data;
27. **int** sameNum;  //表示与该节点有相同关键字的个数
28. node \*lchild, \*rchild;
30. node() {
31. lchild = nullptr;
32. rchild = nullptr;
33. data = infinite;
34. sameNum = 1;
35. balancedFactor = 0;
36. }
37. };
39. node \*root;
40. **int** size;
42. AVL() {
43. root = **new** node;
44. size = 0;
45. }
47. //test if the AVL tree is empty if so , return true
48. **bool** empty() {
49. **return** !size;
50. }
52. //通过中序遍历建立的搜索树来得到排序后的结果
53. **void** MidRecursive(node \*root) {
54. **if** (root != nullptr) {
55. MidRecursive(root->lchild);
56. **for** (**int** i = 0; i < root->sameNum; i++)
57. cout << root->data << " ";
58. MidRecursive(root->rchild);
59. }
60. }
62. //search the specific data
63. node \*search(**int** searchNum) {
64. **if** (searchNum == root->data)
65. **return** root;
66. **else** **if** (searchNum < root->data) {
67. **if** (root->lchild)
68. **return** searchRecursive(searchNum, root->lchild);
69. **else**
70. **return** nullptr;
71. } **else** {
72. **if** (root->rchild)
73. **return** searchRecursive(searchNum, root->rchild);
74. **else**
75. **return** nullptr;
76. }
77. }
79. **void** insert(**int** insertNum) {
80. size++;
81. **bool** addFloor = **false**;
82. **if** (root->data == infinite)    //在第一次插入节点时，需要特殊处理
83. {
84. root->data = insertNum;
85. **return**;
86. }
87. **if** (root->data == insertNum)   //如果有相同的关键字，就在该节点的记录重复的数据上加一
88. {
89. root->sameNum++;
90. **return**;
91. } **else** **if** (insertNum < root->data) {
92. root->lchild = insertRecursive(root->lchild, insertNum, &addFloor);
93. **if** (addFloor) {
94. **switch** (root->balancedFactor)        //addFloor  是表示插入新节点导致层数增加，根据原来的状态来判断现在的平衡因子
95. {
96. **case** LH:    //原本左子树比右子树高，需要作左平衡处理
97. root = leftBalanced(root);
98. addFloor = **false**;
99. **break**;
100. **case** EH:   //原本左、右子树等高，现因左子树增高而使树增高
101. root->balancedFactor = LH;
102. addFloor = **true**;
103. **break**;
104. **case** RH:   //原本右子树比左子树高，现左、右子树等高
105. root->balancedFactor = EH;
106. addFloor = **false**;
107. **break**;
108. }
109. }
110. } **else** {
111. root->rchild = insertRecursive(root->rchild, insertNum, &addFloor);
112. **if** (addFloor) {
113. **switch** (root->balancedFactor) {
114. **case** LH:    //原本左子树比右子树高，现左、右子树等高
115. root->balancedFactor = EH;
116. addFloor = **false**;
117. **break**;
118. **case** EH:    //原本左、右子树等高，现因右子树增高而使树增高
119. root->balancedFactor = RH;
120. addFloor = **true**;
121. **break**;
122. **case** RH:   //原本右子树比左子树高，需要作右平衡处理
123. root = rightBalanced(root);
124. addFloor = **false**;
125. **break**;
126. }
127. }
128. }
129. }
131. //负责删除节点的函数
132. **void** Delete(node \*\*tree, **int** deleteNum) {
133. **if** (!\*tree)
134. **return**;
135. node \*pr, \*p;
136. **if** (empty())
137. **return**;
138. **if** ((\*tree)->data == deleteNum) {
139. node \*temp;
140. size--;
141. **if** ((\*tree)->sameNum > 1) {    //如果遇到该关键词有多个相同的,需要将count减1
142. (\*tree)->sameNum--;
143. } **else** {
144. **if** (!(\*tree)->lchild && !(\*tree)->rchild) {    //如果该节点是叶节点的情况,直接删除节点
145. **if** ((\*tree) == root)
146. (\*tree)->data = infinite;
147. **else** {
148. temp = \*tree;
149. **delete** (temp);
150. \*tree = nullptr;
151. }
152. } **else** **if** (!(\*tree)->lchild) {      //如果是只有左子树,没有右子树
153. temp=\*tree;
154. \*tree=(\*tree)->rchild;
155. **delete** (temp);
156. } **else** **if** (!(\*tree)->rchild) {    //只有右子树,没有左子树
157. temp = \*tree;
158. \*tree=(\*temp).lchild;
159. **delete** (temp);
160. } **else** {    //既有左子树,又有右子树,需要找到右子树中最左的节点,将其删除
161. pr = p = (\*tree)->rchild;
162. **while** (p->lchild) {
163. pr = p;
164. p = p->lchild;
165. }
166. (\*tree)->data = p->data;
167. (\*tree)->sameNum = p->sameNum;
168. **if** (pr != p) {       //递归左子树,寻找匹配的节点
169. pr->lchild = p->rchild;
170. } **else** {       //递归左子树,寻找匹配的节点
171. (\*tree)->rchild = p->rchild;
172. }
173. **delete** (p);
174. }
175. }
176. } **else** **if** (deleteNum < (\*tree)->data) {
177. Delete(&(\*tree)->lchild, deleteNum);
178. } **else** {
179. Delete(&(\*tree)->rchild, deleteNum);
180. }
181. }
183. **private**:
184. //this is for recursive search use and the other users can't use and see it and parameter is complex ,the search() is more easy for use
185. node \*searchRecursive(**int** searchNum, node \*newTree) {
186. **if** (searchNum == newTree->data)
187. **return** newTree;
188. **else** **if** (searchNum < newTree->data) {
189. **if** (newTree->lchild)
190. **return** searchRecursive(searchNum, newTree->lchild);
191. } **else** {
192. **if** (newTree->rchild)
193. **return** searchRecursive(searchNum, newTree->rchild);
194. **else**
195. **return** nullptr;
196. }
197. }
199. //这个函数用于递归调用，与上面public属性的区别在于上面的函数传入的参数更少，更符合restful api 的设计
200. node \*insertRecursive(node \*tree, **int** insertNum, **bool** \*addFloor) {
201. **if** (!tree)   //如果是空，表示搜索树中没有该节点
202. {
203. tree = **new** node;
204. tree->data = insertNum;
205. \*addFloor = **true**;
206. **return** tree;
207. }
208. **if** (tree->data == insertNum) {
209. tree->sameNum++;
210. \*addFloor = **false**;
211. } **else** **if** (insertNum < tree->data) {
212. tree->lchild = insertRecursive(tree->lchild, insertNum, addFloor);
213. **if** (\*addFloor) {
214. **switch** (tree->balancedFactor) {
215. **case** LH:
216. tree = leftBalanced(tree);
217. \*addFloor = **false**;
218. **break**;
219. **case** EH:
220. tree->balancedFactor = LH;
221. \*addFloor = **true**;
222. **break**;
223. **case** RH:
224. tree->balancedFactor = EH;
225. \*addFloor = **false**;
226. **break**;
227. }
228. }
229. } **else** {
230. tree->rchild = insertRecursive(tree->rchild, insertNum, addFloor);
231. **if** (\*addFloor) {
232. **switch** (tree->balancedFactor) {
233. **case** LH:
234. tree->balancedFactor = EH;
235. \*addFloor = **false**;
236. **break**;
237. **case** EH:
238. tree->balancedFactor = RH;
239. \*addFloor = **true**;
240. **break**;
241. **case** RH:
242. tree = rightBalanced(tree);
243. \*addFloor = **false**;
244. **break**;
245. }
246. }
247. }
248. **return** tree;
249. }
251. //单次向左旋转
252. **void** leftRotate(node \*\*tree) {
253. node \*newNode;
254. newNode = (\*\*tree).rchild;
255. (\*\*tree).rchild = newNode->lchild;
256. newNode->lchild = \*tree;
257. \*tree = newNode;
258. }
260. //单次向右旋转
261. **void** rightRotate(node \*\*tree) {
262. node \*newNode;
263. newNode = (\*\*tree).lchild;
264. (\*\*tree).lchild = newNode->rchild;
265. newNode->rchild = \*tree;
266. \*tree = newNode;
267. }
269. node \*leftBalanced(node \*tree) {
270. node \*l = tree->lchild, \*lr = l->rchild;    //l表示tree的左子树 lr表示左子树的右子树
271. **switch** (l->balancedFactor)             //根据l的平衡因子来判断  是LL旋转还是LR旋转的方式来调节树的平衡
272. {
273. **case** LH:
274. rightRotate(&tree);       //是左子树的左子树导致不平衡，是LL的旋转方式
275. tree->balancedFactor = l->balancedFactor = EH;
276. **break**;
277. **case** RH:
278. **switch** (lr->balancedFactor)   //调节tree和l的平衡因子
279. {
280. **case** LH:
281. l->balancedFactor = EH;
282. tree->balancedFactor = RH;
283. **break**;
284. **case** EH:
285. l->balancedFactor = tree->balancedFactor = EH;
286. **break**;
287. **case** RH:
288. l->balancedFactor = LH;
289. tree->balancedFactor = EH;
290. **break**;
291. }
292. lr->balancedFactor = EH;
293. leftRotate(&l);    //是LR的情况
294. tree->lchild = l;
295. rightRotate(&tree);
297. }
298. **return** tree;
299. }
301. node \*rightBalanced(node \*tree) {
302. node \*r = tree->rchild, \*rl = r->lchild;
303. **switch** (r->balancedFactor) {
304. **case** RH:
305. leftRotate(&tree);
306. tree->balancedFactor = r->balancedFactor = EH;     //是RR的情况
307. **break**;
308. **case** LH:
309. **switch** (rl->balancedFactor) {
310. **case** LH:
311. tree->balancedFactor = EH;
312. r->balancedFactor = RH;
313. **break**;
314. **case** EH:
315. tree->balancedFactor = r->balancedFactor = EH;
316. **break**;
317. **case** RH:
318. tree->balancedFactor = LH;
319. r->balancedFactor = EH;
320. **break**;
321. }
322. rl->balancedFactor = EH;
323. rightRotate(&r);
324. tree->rchild = r;   //是RL的旋转调节平衡方式
325. leftRotate(&tree);
326. }
327. **return** tree;
328. }
329. };
331. **int** main() {
332. AVL atree;
333. **int** n, a;
334. cout << "input the number of the num" << endl;
335. cin >> n;
336. **for** (**int** i = 0; i < n; i++) {
337. cin >> a;
338. atree.insert(a);
339. }
340. **for** (**int** i = 1; i < 9; i++) {
341. atree.Delete(&atree.root, i);
342. }
343. cout << atree.size << endl;
344. cout << "mid sequence results are:" << endl;
345. atree.MidRecursive(atree.root);
346. cout << endl;
347. **return** 0;
348. }
349. //13 12 10 6 4 11 2 8 7 5 3 1 9 13
350. //1,5,3,7,6,2,4,8,9,0
351. //21 12 10 6 4 11 2 8 7 5 3 1 9 13 102 3 14 45 23 455 65 12 4 56 546 1 45 6

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

课程类型：必修

试验项目名称：搜索与排序

实验题目：快速排序的优化

班级：1603005

学号：1160300524

姓名：冯运

* 1. 实验目的

通过优化快速排序的效率，深入理解快速排序的时间和空间特性，学会优化快速排序的几大方法

* 1. 实验要求
     1. 首先实现一个最基本的快速排序
     2. 在基本版本的基础上，通过一些优化手段，提升快速排序的速度

实验环境：机房 编程软件：clion

* 1. 设计思想

“四大“优化策略：

* + 1. 三点取中的优化
    2. 在小区间使用插入排序替代快速排序
    3. 有相同元素时，聚中减少递归次数
    4. （伪）尾递归，本来认为此法有效果，后来实际实验后发现无论效率还是空间都没有明显的提升，原因是在快排的算法中，尾递归优化是不存在的。

对快排算法本身，如果使用最基本的固定枢轴的方法，在排序数据是高度乱序（随机度较高）的情况下，时间复杂度稳定在O(logn),但是测试时数据若是已经有序的，或是重复的数字较多，时间复杂度接近于O(n2)

所以优化针对的就是已经有序的和重复数字较多的情况。

* 1. 三点取中的方法：

思路：选取枢轴时，在左端点和右端点以及中间节点三个值中，选择一个中间值作为枢轴，这样可以有效避免已经有序的输入情况。

* 1. 插入排序的方式：

经过时间证明在排序的数量少于16时，插入排序的效率要高于其他所有的排序，所以可以使用插入排序优化速度

* 1. 聚中的方式

当有相同元素时，扫描区间时，遇到与枢轴相同的元素时，将元素放到区间的两端，当扫描结束后，将枢轴的位置调节为扫描标记停止的位置，然后将两边的与枢轴值相同的元素调整到枢轴的周围。在进行递归时，枢轴周围相同的元素将被跳过，不再进行递归。

这样可以避免相同元素对快排效率的影响。

* 1. 关于尾递归优化的认知

在查资料时看到尾递归优化的方式，于是查看了尾递归的含义：

递归函数在一个函数返回的位置，由于是函数的结束，之前函数栈帧的内容将不再进行保存，只保存当前递归函数的内容，这样无论递归多少次对不会发成stack overflow 的错误，而且可以减少对栈空间的使用。

查到快排的这样一个尾递归方式：

1. **public** **void** QuickSort(**int**[] num, **int** low, **int** high)
2. {
3. **int** pivot;
4. **while** (low < high)
5. {
6. pivot = partition(num, low, high); ;
7. QuickSort(num, low, pivot - 1); // 对低子表递归排序
8. low = pivot + 1; // 尾递归
9. }
10. }

我自己写了一个类似的实现，多次实验后发现并没有明显的效果，我开始分析原因：

这个实现是将

1. quickSort(list,pivot+1,high);

替换为了

1. low = pivot + 1; // 尾递归

并且通过循环的方式，每一次循环改变了low(左端点)，和pivot(枢点)，然后进入下一次递归。

发现这样的写法和直接进行右递归是基本一样的实现方式，首先递归的次数一次都没有减少，不存在效率上的优化。

其次，尾递归是在调用递归函数后，父函数结束，不再保存其栈帧，然而上面的循环写法明显在调用递归函数后，父函数没有结束运行，栈帧的信息依旧保存，并没有减少，所以这种优化就是没有作用，还会增加代码的复杂度，使得代码的可读性下降。快排算法不存在尾递归的优化。

主要数据结构的定义：

1. **int** randomNum1[Num],randomNum2[Num],temp;   //用于保存用于排序的数值

主要函数的解析：

1. **void** quickSort(**int** \*num,**int** left,**int** right)
2. {
3. **int** i,j,t,temp,l,r,lLen=0,rLen=0;
4. **if**(left>=right)
5. **return**;
6. **if**(right-left<16)    // 当新区键的大小小于16时，改用插入排序的效率更高  经过多次测试，发现参数是16时用时最少
7. {
8. insertSort(num,left,right);
9. **return** ;
10. }
11. **int** pivot=findKey(num,left,right);
12. temp=num[pivot]; //temp中存的就是基准数
13. i=left;
14. j=right;
15. l=left;
16. r=right;
17. **while**(i<j)
18. {
19. //顺序很重要，要先从右边开始找
20. **while**(num[j]>=temp && i<j)
21. {
22. **if**(num[j]==temp&&j!=pivot)   //这是聚中的优化方式，遇到与枢轴值相同时，将其放到两边
23. {
24. **if**(pivot==r)     //这是在调试时遇到的一个bug,  如果pivot的位置与r相同，交换后会对后面的pivot位置调整产生影响
25. pivot=j;
26. swap(num[j],num[r]);
27. r--;   //r指示的是右面相同元素的位置
28. rLen++;  //右面相同元素的个数
29. }
30. j--;
31. }
32. //再找左边的
33. **while**(num[i]<=temp && i<j)
34. {
35. **if**(num[i]==temp&&i!=pivot)   //与上面的相对称
36. {
37. **if**(pivot==l)
38. pivot=i;
39. swap(num[i],num[l]);
40. l++;
41. lLen++;
42. }
43. i++;
44. }
45. //交换两个数在数组中的位置
46. **if**(i<j)
47. {
48. t=num[i];
49. num[i]=num[j];
50. num[j]=t;
51. }
52. }
53. //最终将基准数归位
54. **if**(pivot<=i||num[i]>num[pivot])
55. {
56. num[pivot]=num[i];
57. num[i]=temp;
58. }
59. **else** **if**(i+1<=right)
60. {
61. num[pivot]=num[i+1];
62. num[i+1]=temp;
63. i++;
64. }
65. **else**
66. {
67. num[pivot]=num[i+1];
68. num[i+1]=temp;
69. }
71. //开始将相同的元素从两面向中间聚拢
72. **int** x=1;
73. **if**(lLen>0)
74. l--;
75. **int** a=lLen;
76. **while**(a>0)
77. {
78. swap(num[l],num[i-x]);
79. x++;
80. l--;
81. a--;
82. }
83. a=rLen;
84. x=1;
85. **if**(rLen>0)
86. r++;
87. **while**(a>0)
88. {
89. swap(num[r],num[i+x]);
90. x++;
91. r++;
92. a--;
93. }
94. quickSort(num,i+1+rLen,right);
95. quickSort(num,left,i-1-lLen);
96. }
97. //寻找枢值的函数  三点取中
98. **int** findKey(**int**\* num,**int** a,**int** b)
99. {
100. **int** mid;
101. mid=(a+b)>>1;   //中点的下标
102. **if**(num[mid]>num[b])
103. swap(num[mid],num[b]);
104. **if**(num[a]>num[mid])
105. swap(num[a],num[mid]);
106. **if**(num[mid]>num[b])
107. swap(num[mid],num[b]);
108. **return** mid;    //此时mid的值就是三点的中值
109. }
     1. 测试结果

我的测试直接是与<algorithm>库中的sort()快排函数的效率进行比较的

在数到的规模为10,000,000 的前提下，有三种情况：

* + 1. 全部是随机生成的数（随机范围为30000000）

测试结果如图4-1所示

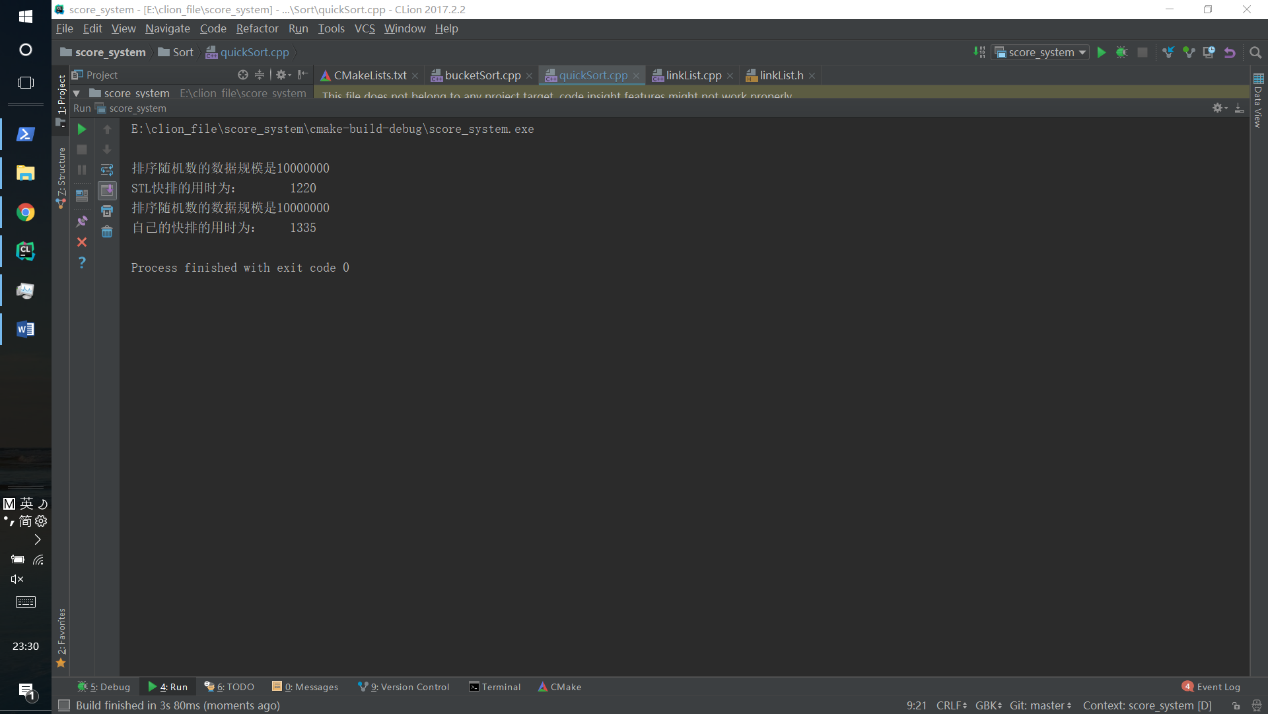


图4-1

上面的是STL快排的用时，下面是我的快排的用时，单位是 ms,没有太大的差距

* + 1. 有大量的相同数字排序情况（总的数字不变，将随机值的范围调整为（0到100））

测试结果如图4-2所示

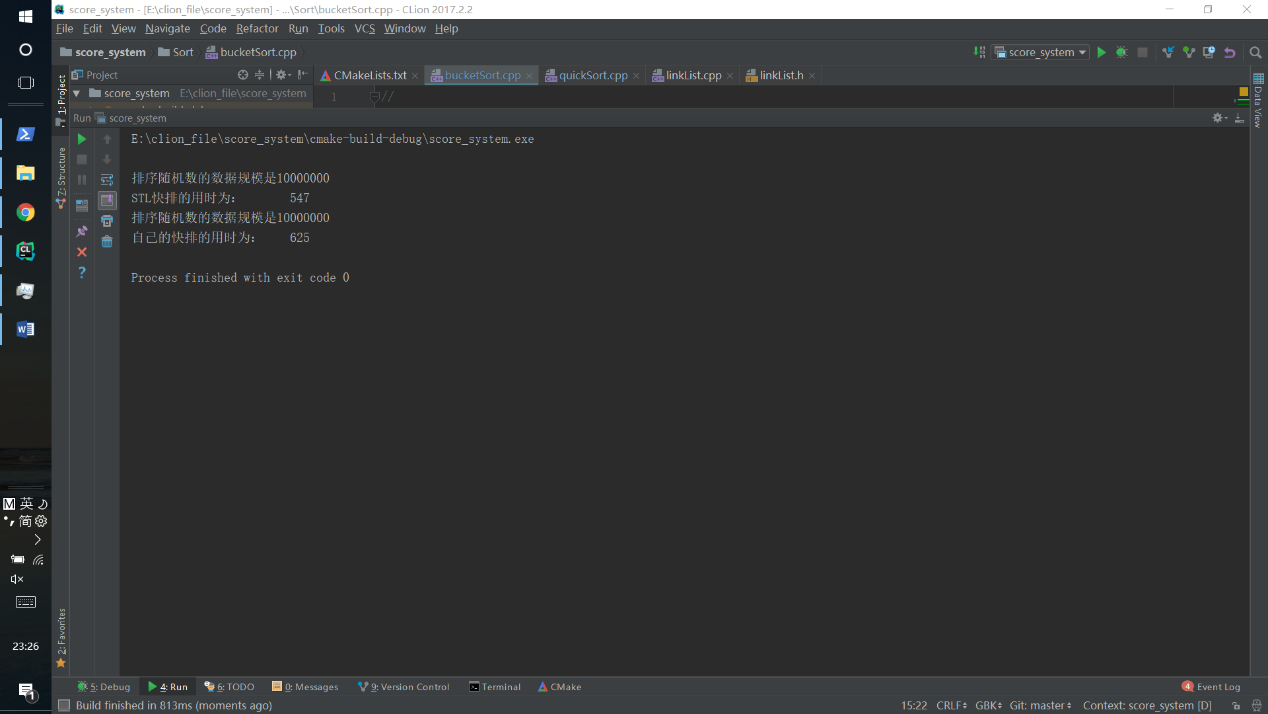


图4-2

可以看出速度和上面的差不多，可见聚中的优化方式起到了明显的效果。

* + 1. 已经有序的数字排序，测试结果如图4-3所示

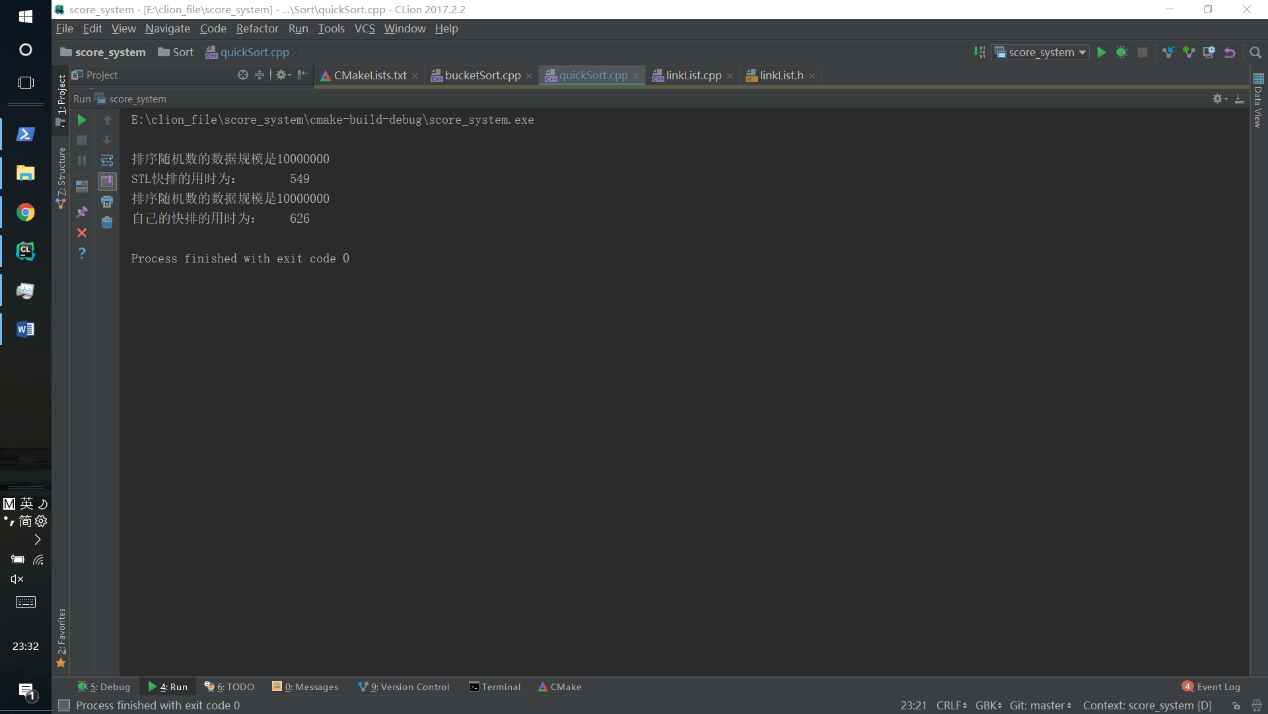


图4-3

可见与系统的快排的差距小于 100ms,说明三点取中的优化有效。

* 1. 系统的不足与体会

系统的不足：  
在做效率比较时，未与原来未优化的时候作比较，可能使得优化的效果不容易直观的看出来。

与系统的快排函数在效率方面总有小于100ms的差距，以后希望能通过一些优化手段和系统的函数效率持平甚至超过其效率

体会：通过这次实际的动手操作，深入的理解体会了优化算法的全过程，对算法的效率分析和优化增加了不少经验，收获了很多的知识

* 1. 源代码

1. #include <iostream>
2. #include <cstdlib>
3. #include <algorithm>
4. #include <ctime>
5. #define Num 10000000
6. #define Max 10
7. #define Min 1
8. **int** randomNum1[Num],randomNum2[Num],temp;   //用于保存用于排序的数值
9. **using** **namespace** std;
10. //寻找枢值的函数  三点取中
11. **int** findKey(**int**\* num,**int** a,**int** b)
12. {
13. **int** mid;
14. mid=(a+b)>>1;   //中点的下标
15. **if**(num[mid]>num[b])
16. swap(num[mid],num[b]);
17. **if**(num[a]>num[mid])
18. swap(num[a],num[mid]);
19. **if**(num[mid]>num[b])
20. swap(num[mid],num[b]);
21. **return** mid;    //此时mid的值就是三点的中值
22. }
24. //插入排序
25. **void** insertSort(**int** num[],**int** left,**int** right)
26. {
27. **int** j,temp,k;
28. **for**(**int** i=left+1;i<=right;i++)
29. {
30. temp=num[i];
31. j=i;
32. **while**(temp<=num[j]&&j>left)
33. {
34. j--;
35. }
36. k=i;
37. **while**(k!=j+1)
38. {
39. num[k]=num[k-1];
40. k--;
41. }
42. **if**(temp<num[j])
43. {
44. num[j+1]=num[j];
45. num[j]=temp;
46. }
47. **else**
48. num[j+1]=temp;
49. }
50. }

53. **void** quickSort(**int** \*num,**int** left,**int** right)
54. {
55. **int** i,j,t,temp,l,r,lLen=0,rLen=0;
56. **if**(left>=right)
57. **return**;
58. **if**(right-left<16)    // 当新区键的大小小于16时，改用插入排序的效率更高  经过多次测试，发现参数是16时用时最少
59. {
60. insertSort(num,left,right);
61. **return** ;
62. }
63. **int** pivot=findKey(num,left,right);
64. temp=num[pivot]; //temp中存的就是基准数
65. i=left;
66. j=right;
67. l=left;
68. r=right;
69. **while**(i<j)
70. {
71. //顺序很重要，要先从右边开始找
72. **while**(num[j]>=temp && i<j)
73. {
74. **if**(num[j]==temp&&j!=pivot)   //这是聚中的优化方式，遇到与枢轴值相同时，将其放到两边
75. {
76. **if**(pivot==r)     //这是在调试时遇到的一个bug,  如果pivot的位置与r相同，交换后会对后面的pivot位置调整产生影响
77. pivot=j;
78. swap(num[j],num[r]);
79. r--;   //r指示的是右面相同元素的位置
80. rLen++;  //右面相同元素的个数
81. }
82. j--;
83. }
84. //再找左边的
85. **while**(num[i]<=temp && i<j)
86. {
87. **if**(num[i]==temp&&i!=pivot)   //与上面的相对称
88. {
89. **if**(pivot==l)
90. pivot=i;
91. swap(num[i],num[l]);
92. l++;
93. lLen++;
94. }
95. i++;
96. }
97. //交换两个数在数组中的位置
98. **if**(i<j)
99. {
100. t=num[i];
101. num[i]=num[j];
102. num[j]=t;
103. }
104. }
105. //最终将基准数归位
106. **if**(pivot<=i||num[i]>num[pivot])
107. {
108. num[pivot]=num[i];
109. num[i]=temp;
110. }
111. **else** **if**(i+1<=right)
112. {
113. num[pivot]=num[i+1];
114. num[i+1]=temp;
115. i++;
116. }
117. **else**
118. {
119. num[pivot]=num[i+1];
120. num[i+1]=temp;
121. }
123. //开始将相同的元素从两面向中间聚拢
124. **int** x=1;
125. **if**(lLen>0)
126. l--;
127. **int** a=lLen;
128. **while**(a>0)
129. {
130. swap(num[l],num[i-x]);
131. x++;
132. l--;
133. a--;
134. }
135. a=rLen;
136. x=1;
137. **if**(rLen>0)
138. r++;
139. **while**(a>0)
140. {
141. swap(num[r],num[i+x]);
142. x++;
143. r++;
144. a--;
145. }
146. quickSort(num,i+1+rLen,right);
147. quickSort(num,left,i-1-lLen);
148. }
150. **int** cmp( **const** **void** \*a , **const** **void** \*b)
151. {
152. **return** \*(**int** \*)a - \*(**int** \*)b;
153. }
155. **int** main()
156. {
157. **for**(**int** i=0;i<Num;i++)
158. {
159. temp=Min+rand()%Max;
160. temp=Max-i;
161. randomNum1[i]=temp;
162. randomNum2[i]=temp;
163. }
164. cout<<endl;
165. **double** time;
166. time=clock();
167. //    sort(randomNum1,&randomNum1[Num],greater<int>());  //从大到小排序
168. qsort(randomNum1,Num, **sizeof**(**int**),cmp);   //从小到大排序
169. //    sort(randomNum1,&randomNum1[Num]);
170. time=clock()-time;
171. cout<<"排序随机数的数据规模是"<<Num<<endl<<"STL快排的用时为：\t";
172. cout<<time<<endl;
173. time=clock();
174. quickSort(randomNum2,0,Num-1);  //自己的快排函数
175. time=clock()-time;
176. cout<<"排序随机数的数据规模是"<<Num<<endl<<"自己的快排的用时为：\t";
177. cout<<time<<endl;
178. **for**(**int** i=0;i<Num;i++)
179. {
180. **if**(randomNum1[i]!=randomNum2[i])
181. {
182. cout<<i<<endl;
183. cout<<randomNum1[i]<<"  "<<randomNum2[i]<<endl;
184. }
185. }
186. **return** 0;
187. }

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

课程类型：必修

试验项目名称：搜索与排序

实验题目：线性排序的比较

班级：1603005

学号：1160300524

姓名：冯运

一 、实验目的

通过实际的编程实现三种常见的线性排序，然后比较各自的特性以及优缺点，以达到深入理解的目的

二 、实验要求及环境

* + 1. 实际实现三种常见的先行排序的函数，比较各自的特性以及优缺点

实验环境：机房 编程软件：clion

版本管理工具：git github

三、设计思想

三种线性排序的理解：

1. 桶排序 是设置一些区间，通过将要排序的数字通过一定的规则划分进这些区间（桶），然后使用别的方法将他们排序
2. 计数排序 是使用一个额外的空间，其中存储的是要排序数出现的次数。
3. 基数排序 是按照不同的关键词进行排序的方式

共同点：1、都是通过空间换取时间的方式实现高效率

* + 1. 都是利用了数值的空间局部性的特点，实现高效的排序
    2. 都不是通过比较数值大小，而是使用数值作为下标直接映射到某一空间或者记录出现的次数，超过了用单纯比较方式排序的下限

不同点：1、其中的桶排序在桶内的排序是随意地，所以桶排序不稳定，其效率既取决于给的数据是否能够均匀分布在这些桶内，还取决于桶内排序算法是否高效，同时还有桶的个数也其关键作用，桶的个数越多，每个桶内的排序元素数量就会减少。 桶排序可以对浮点数进行排序 桶排序一般情况下是三个排序中最慢的，但是计数排序是一种特殊的桶排序。

2、计数排序在三种排序中效率最高，但是对内存的消耗最大，对输入的数据范围也有要求 ，如果输入的数据太大，会浪费大量的空间，同时无法对浮点数进行排序，也是一大缺点

3、基数排序兼具效率空间的优良特性，在效率上，最好可以达到O(n)的效率，空间上的开销也是可以控制的，通过调节基数的大小，达到平衡空间和时间需求的目的： 基数越大，基数排序越接近于计数排序，空间开销增大，但是时间复杂性会大大降低， 基数越小，空间的开销越小，但是时间复杂性越高，因为需要排序的次数增大 ，复杂度为O(K\*N),随着基数的增大，K会增大。 K是根据基数确定的最大的位数。

在三种排序中，基数排序的时间和空间相对平衡，也比较稳定，可以与快速排序相媲美。

主要数据结构的定义

1. #define Max 60000000   //需要排序数组的最大范围
2. #define bucketNum 5000000    //桶的个数
3. #define madix 10      //基数排序中基数的大小
4. #define MaxRandom 10000000     //生成随机数时最大随机范围的定义
5. **int** randomNum1[Max], randomNum2[Max], randomNum3[Max], randomNum4[Max];    //分别用于四种排序
6. **int** temp1[Max + 1];   //用于计数排序的辅助数组
7. **int** temp2[Max + 1];   //用于计数排序的辅助数组
8. linkList<**int**> bucket[bucketNum];   //桶内的链表结构，使用方法详见 linkList.h

关键函数的具体解释

桶排序

1. //桶排序的函数，num接收一个数组，length表示数组的长度
2. **void** bucketSort(**int** num[], **int** length) {
3. **int** flag;
4. //在每个桶内使用的是插入排序法
5. **for** (**int** i = 0; i < length; i++) {
6. **if** (bucket[num[i] >> 1].size == 0)  //如果原来链表为空，就插入一个新元素
7. bucket[num[i] >> 1].insert(0, num[i]);
8. **else** {
9. flag = 0;
10. **for** (**int** j = 1; j <= bucket[num[i] >> 1].size; j++) {    //从链表的开头遍历找到新节点合适的插入位置
11. **if** (bucket[num[i] >> 1].inquire(j)->data1 > num[i]) {
12. bucket[num[i] >> 1].insert(j - 1, num[i]);
13. flag = 1;
14. **break**;
15. }
16. }
17. **if** (flag == 0)
18. bucket[num[i] >> 1].insert(bucket[num[i] >> 1].size, num[i]);
19. }
20. }
21. **int** count = 0;
22. //按照桶的顺序遍历每个桶，将桶内的有序元素赋给要排序的数组，同时释放内存
23. **for** (**int** i = 0; i < bucketNum; i++) {
24. **for** (**int** j = 1; j <= bucket[i].size; j++) {
25. num[count] = bucket[i].inquire(j)->data1;
26. bucket[i].pop(j);    //释放内存，防止内存泄漏
27. count++;
28. }
29. }
30. }

计数排序实现1

1. //一种计数排序的省内存，效率高，但是不能用于基数排序的版本
2. **void** countSortEasy(**int** num[], **int** length, **int** maxNum) {
3. memset(temp1, 0, **sizeof**(**int**) \* (maxNum + 1));
4. **for** (**int** i = 0; i < length; i++)
5. temp1[num[i]]++;
6. **int** c = 0;
7. **for** (**int** i = 0; i <= maxNum; i++) {
8. **for** (**int** j = 0; j < temp1[i]; j++) {
9. num[c] = i;
10. c++;
11. }
12. }
13. }

计数排序实现2

1. //这种是空间消耗较大，效率相对较低的版本，但是可以用于多关键词比较
2. **void** countSortComplex(**int** num[], **int** length, **int** maxNum) {
3. **int** better1, better2, better3;   //用于循环展开的优化方式
4. memset(temp1, 0, **sizeof**(**int**) \* (maxNum + 1));
5. **for** (**int** i = 0; i < length; i++)
6. temp1[num[i]]++;
7. **int** x;
8. //这个函数本来是迭代的循环，根据计算机系统的原理，我使用3x1循环展开的方式来优化效率，提高并行执行的能力，可以达到提升效率的目的
9. **for** (x = 0; x < length - 4; x += 3) {
10. better1 = temp1[x] + temp1[x + 1];
11. better2 = temp1[x] + (temp1[x + 1] + temp1[x + 2]);
12. temp1[x + 1] = better1;
13. better3 = better1 + (temp1[x + 2] + temp1[x + 3]);
14. temp1[x + 2] = better2;
15. temp1[x + 3] = better3;
16. }
17. //处理不能整除的剩余部分
18. **for** (; x < maxNum - 1; x++) {
19. temp1[x + 1] = temp1[x] + temp1[x + 1];
20. }
22. //这是不用循环展开优化的版本
23. //    for(int i=0;i<maxNum-1;i++)
24. //    {
25. //        temp1[i+1]=temp1[i]+temp1[i+1];
26. //    }
28. //定位排序后的元素位置
29. **for** (**int** i = length - 1; i >= 0; i--) {
30. temp2[temp1[num[i]] - 1] = num[i];
31. temp1[num[i]]--;
32. }
33. //将修改后的值赋给需要排序的数组
34. **for** (**int** i = 0; i < length; i++) {
35. num[i] = temp2[i];
36. }
37. }

基数排序

1. //基数排序的实现  ，内部使用了计数排序的复杂版本
2. **void** madixSort(**int** num[], **int** length, **int** max) {
3. //    原本用来寻找传入数组的最大值
4. //    int max=num[0];
5. //    for(int i=0;i<length-1;i++)
6. //    {
7. //        if(max<num[i+1])
8. //            max=num[i+1];
9. //    }
10. **int** temp = 1;
11. **int** stash;   //这里存储一个常用值，来优化效率
12. memset(temp1, 0, **sizeof**(**int**) \* (max + 1));
13. //循环，依次按照个位，十位……排序
14. **while** (max / temp) {
15. //接下来的都是计数排序的内容
16. //按照某位,  暂存数组存的是该位出现的次数
17. **for** (**int** i = 0; i < length; i++)
18. temp1[(num[i] / temp) % madix]++;
19. //暂存数组temp1  中存储小于下标值的出现次数
20. **for** (**int** i = 0; i < madix; i++) {
21. temp1[i + 1] = temp1[i] + temp1[i + 1];
22. }
23. //定位按照某位排序后关键字存储的位置，暂存于 temp2中
24. **for** (**int** i = length - 1; i >= 0; i--) {
25. stash = (num[i] / temp) % madix;   //存储的是num[i]的某位的值
26. temp2[temp1[stash] - 1] = num[i];
27. temp1[stash]--;
28. }
29. memset(temp1, 0, **sizeof**(**int**) \* (max + 1));//初始化，避免下次循环出现Bug
30. //将temp2的内容复制到num数组中
31. **for** (**int** i = 0; i < length; i++) {
32. num[i] = temp2[i];
33. }
34. temp \*= madix;
35. }
36. }

四、测试结果

测试数据的规模为 6000,0000

测试的排序结果都在主函数中与系统快排的结果一一比对，如果有错误，就会报错

1、随机数的随机范围为 10,000,000

测试结果如图4-1所示

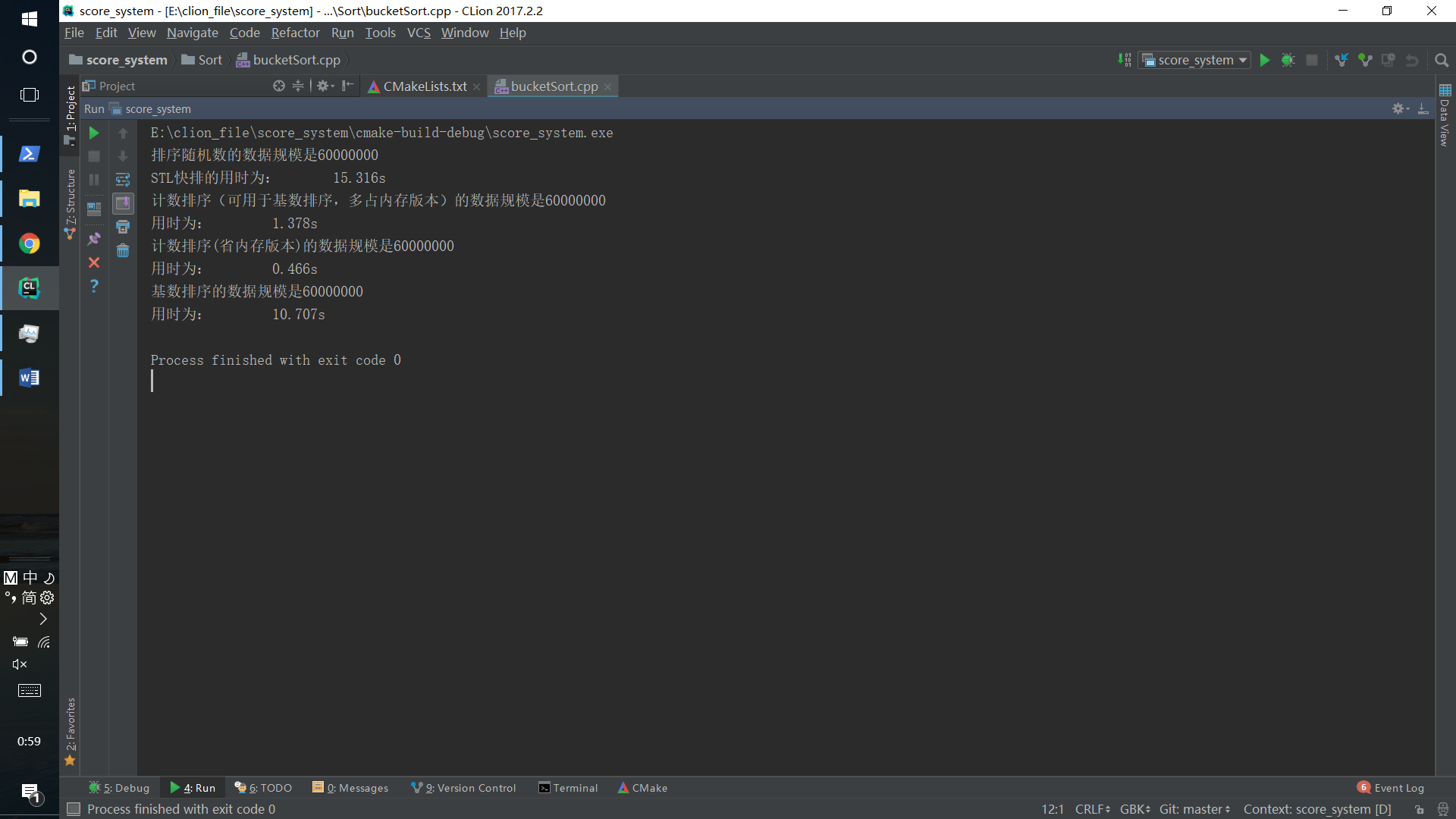


图4-1

结果分析：

可以看到快排是其中最慢的排序，其次是基数排序，最快的是计数排序。（桶排序由于在如此大的数据规模下，无法存储，无法进行测试）

2、在数据已经有序的情况下，测试结果如图4-2所示

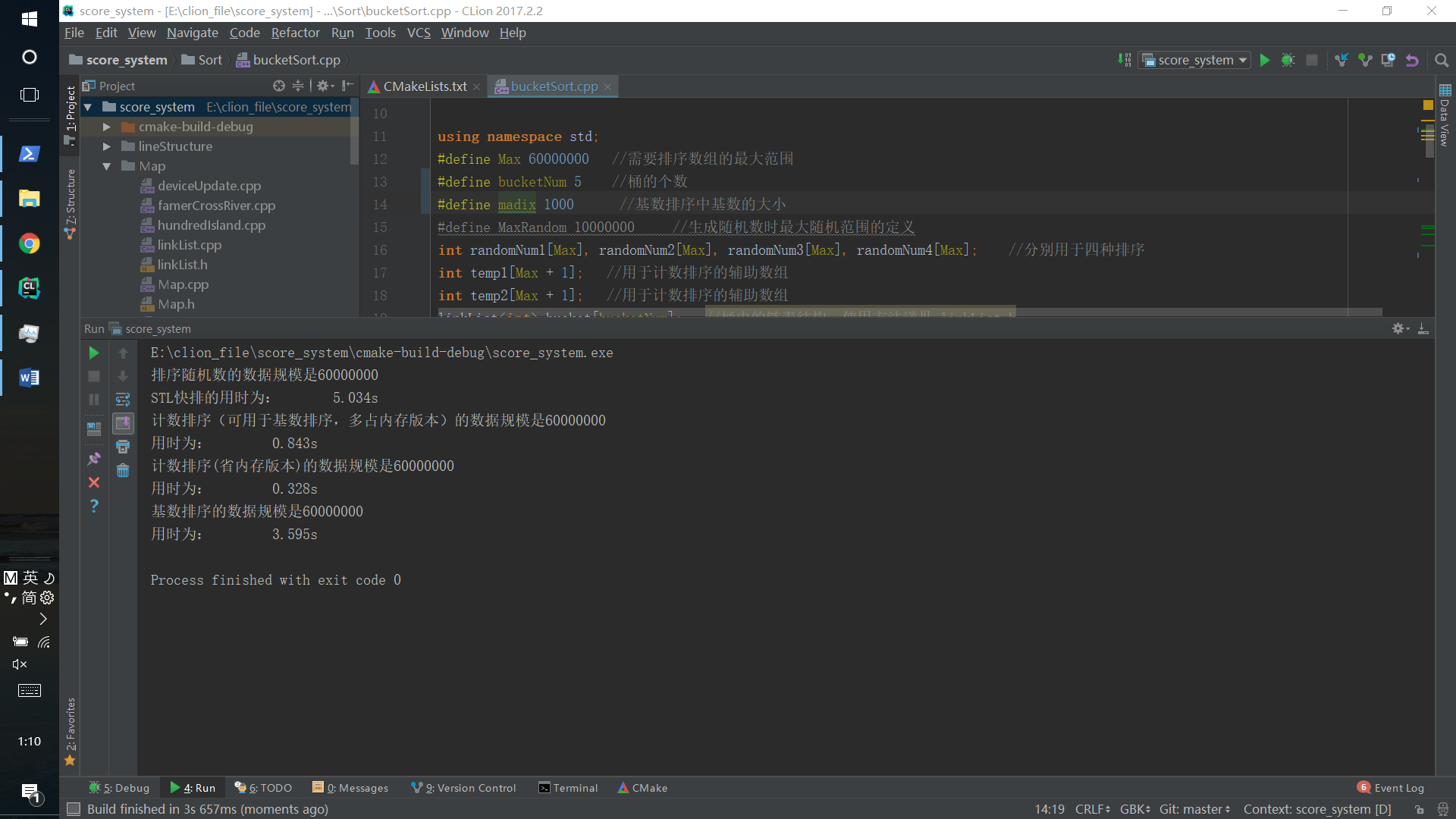


图4-2

此条件下，基数排序的基数设置为 1000

可以看出计数排序最快，基数次之，快排最慢，桶排序依旧因为内存限制无法测试

3、同2的条件，只是将基数设置为了10，测试结果如图4-3所示

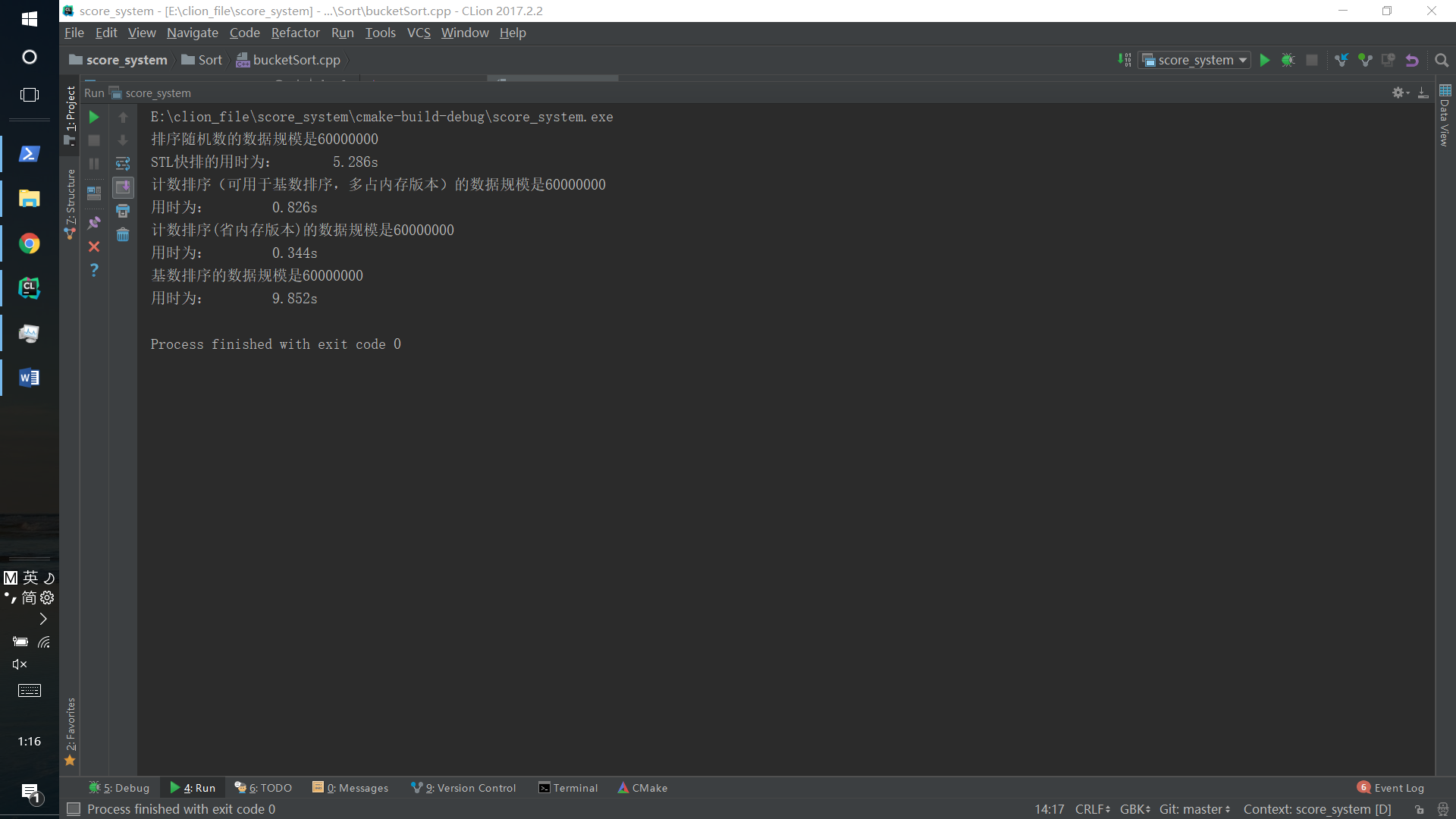


图4-3

结果分析：同2相比，只有基数的值变成了10，基数排序的效率下降很多，因为基数排序的时间复杂度为O(K\*N),基数增大，K会增大，时间就会增大

4、在减小数据规模（1000，0000）后，测试得到桶排序的结果，如图4-4

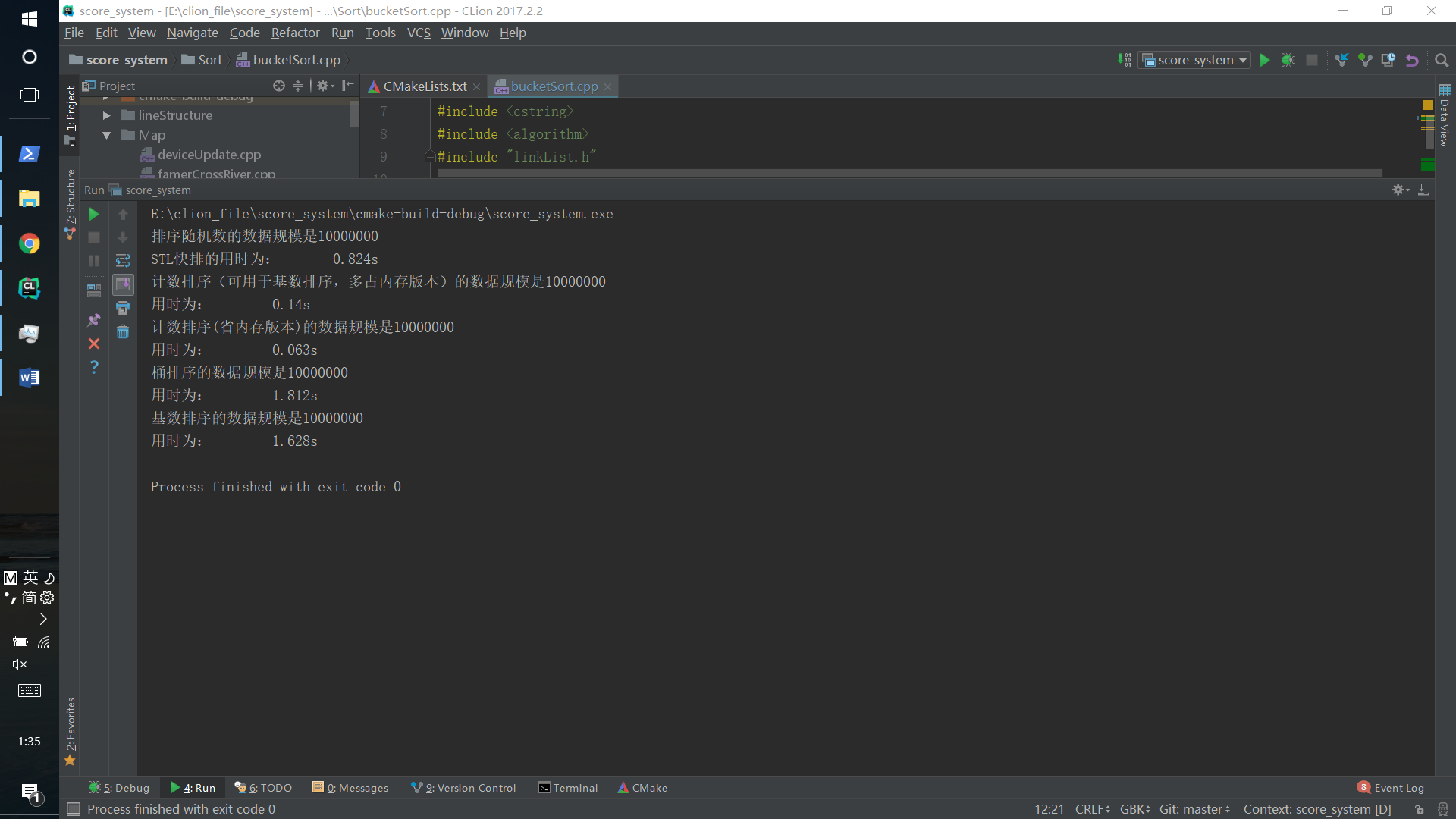


图4-4

结果分析：

可以看出，桶排序的速度最慢，分析原因是：在桶排序时，用了链表，链表需要频繁插入，以及最后还需要遍历一次释放内存，导致效率不太高。

五、系统的不足与经验体会

经验体会：对线性排序的优缺点和使用特点有了深入的了解，对以后的编程开发很有帮助

六、源代码

1. //
2. // Created by Fitz on 2017/12/17.
3. //
4. #include <iostream>
5. #include <ctime>
6. #include <cstdlib>
7. #include <cstring>
8. #include <algorithm>
9. #include "linkList.h"
11. **using** **namespace** std;
12. #define Max 10000000   //需要排序数组的最大范围
13. #define bucketNum 50000000    //桶的个数
14. #define madix 10      //基数排序中基数的大小
15. #define MaxRandom 10000000     //生成随机数时最大随机范围的定义
16. **int** randomNum1[Max], randomNum2[Max], randomNum3[Max], randomNum4[Max], randomNum5[Max];   //分别用于五种排序
17. **int** temp1[Max + 1];   //用于计数排序的辅助数组
18. **int** temp2[Max + 1];   //用于计数排序的辅助数组
19. linkList<**int**> bucket[bucketNum];   //桶内的链表结构，使用方法详见 linkList.h
21. //桶排序的函数，num接收一个数组，length表示数组的长度
22. **void** bucketSort(**int** num[], **int** length) {
23. **int** flag;
24. //在每个桶内使用的是插入排序法
25. **for** (**int** i = 0; i < length; i++) {
26. **if** (bucket[num[i] >> 1].size == 0)  //如果原来链表为空，就插入一个新元素
27. bucket[num[i] >> 1].insert(0, num[i]);
28. **else** {
29. flag = 0;
30. **for** (**int** j = 1; j <= bucket[num[i] >> 1].size; j++) {    //从链表的开头遍历找到新节点合适的插入位置
31. **if** (bucket[num[i] >> 1].inquire(j)->data1 > num[i]) {
32. bucket[num[i] >> 1].insert(j - 1, num[i]);
33. flag = 1;
34. **break**;
35. }
36. }
37. **if** (flag == 0)
38. bucket[num[i] >> 1].insert(bucket[num[i] >> 1].size, num[i]);
39. }
40. }
41. **int** count = 0;
42. //按照桶的顺序遍历每个桶，将桶内的有序元素赋给要排序的数组，同时释放内存
43. **for** (**int** i = 0; i < bucketNum; i++) {
44. **for** (**int** j = 1; j <= bucket[i].size; j++) {
45. num[count] = bucket[i].inquire(j)->data1;
46. count++;
47. }
48. }
50. //单独循环释放内存
51. **for** (**int** i = 0; i < bucketNum; i++) {
52. **while** (!bucket[i].empty())
53. bucket[i].pop(1);
54. }
56. }
58. //一种计数排序的省内存，效率高，但是不能用于基数排序的版本
59. **void** countSortEasy(**int** num[], **int** length, **int** maxNum) {
60. memset(temp1, 0, **sizeof**(**int**) \* (maxNum + 1));
61. **for** (**int** i = 0; i < length; i++)
62. temp1[num[i]]++;
63. **int** c = 0;
64. **for** (**int** i = 0; i <= maxNum; i++) {
65. **for** (**int** j = 0; j < temp1[i]; j++) {
66. num[c] = i;
67. c++;
68. }
69. }
70. }
72. //这种是空间消耗较大，效率相对较低的版本，但是可以用于多关键词比较
73. **void** countSortComplex(**int** num[], **int** length, **int** maxNum) {
74. **int** better1, better2, better3;   //用于循环展开的优化方式
75. memset(temp1, 0, **sizeof**(**int**) \* (maxNum + 1));
76. **for** (**int** i = 0; i < length; i++)
77. temp1[num[i]]++;
78. **int** x;
79. //这个函数本来是迭代的循环，根据计算机系统的原理，我使用3x1循环展开的方式来优化效率，提高并行执行的能力，可以达到提升效率的目的
80. **for** (x = 0; x < length - 4; x += 3) {
81. better1 = temp1[x] + temp1[x + 1];
82. better2 = temp1[x] + (temp1[x + 1] + temp1[x + 2]);
83. temp1[x + 1] = better1;
84. better3 = better1 + (temp1[x + 2] + temp1[x + 3]);
85. temp1[x + 2] = better2;
86. temp1[x + 3] = better3;
87. }
88. //处理不能整除的剩余部分
89. **for** (; x <= maxNum - 1; x++) {
90. temp1[x + 1] = temp1[x] + temp1[x + 1];
91. }
93. //这是不用循环展开优化的版本
94. //    for(int i=0;i<maxNum-1;i++)
95. //    {
96. //        temp1[i+1]=temp1[i]+temp1[i+1];
97. //    }
99. //定位排序后的元素位置
100. **for** (**int** i = length - 1; i >= 0; i--) {
101. temp2[temp1[num[i]] - 1] = num[i];
102. temp1[num[i]]--;
103. }
104. //将修改后的值赋给需要排序的数组
105. **for** (**int** i = 0; i < length; i++) {
106. num[i] = temp2[i];
107. }
108. }
110. //基数排序的实现  ，内部使用了计数排序的复杂版本
111. **void** madixSort(**int** num[], **int** length, **int** max) {
112. //    原本用来寻找传入数组的最大值
113. //    int max=num[0];
114. //    for(int i=0;i<length-1;i++)
115. //    {
116. //        if(max<num[i+1])
117. //            max=num[i+1];
118. //    }
119. **int** temp = 1;
120. **int** stash;   //这里存储一个常用值，来优化效率
121. memset(temp1, 0, **sizeof**(**int**) \* (max + 1));
122. //循环，依次按照个位，十位……排序
123. **while** (max / temp) {
124. //接下来的都是计数排序的内容
125. //按照某位,  暂存数组存的是该位出现的次数
126. **for** (**int** i = 0; i < length; i++)
127. temp1[(num[i] / temp) % madix]++;
128. //暂存数组temp1  中存储小于下标值的出现次数
129. **for** (**int** i = 0; i < madix; i++) {
130. temp1[i + 1] = temp1[i] + temp1[i + 1];
131. }
132. //定位按照某位排序后关键字存储的位置，暂存于 temp2中
133. **for** (**int** i = length - 1; i >= 0; i--) {
134. stash = (num[i] / temp) % madix;   //存储的是num[i]的某位的值
135. temp2[temp1[stash] - 1] = num[i];
136. temp1[stash]--;
137. }
138. memset(temp1, 0, **sizeof**(**int**) \* (max + 1));//初始化，避免下次循环出现Bug
139. //将temp2的内容复制到num数组中
140. **for** (**int** i = 0; i < length; i++) {
141. num[i] = temp2[i];
142. }
143. temp \*= madix;
144. }
145. }
147. **int** main() {
148. **int** temp;
149. **for** (**int** i = 0; i < Max; i++) {
150. //        temp = rand() % MaxRandom;
151. temp = Max - i;
152. randomNum1[i] = temp;
153. randomNum2[i] = temp;
154. randomNum3[i] = temp;
155. randomNum4[i] = temp;
156. randomNum5[i] = temp;
157. }
158. **double** time;
159. time = clock();
160. //    sort(randomNum1,&randomNum1[Num],greater<int>());  //从大到小排序
161. //    qsort(randomNum1,Max, sizeof(int),cmp);   //从小到大排序
162. sort(randomNum1, &randomNum1[Max]);
163. time = clock() - time;
164. cout << "排序随机数的数据规模是" << Max << endl << "STL快排的用时为：\t";
165. cout << time / 1000 << "s" << endl;
166. time = clock();
167. countSortComplex(randomNum2, Max, Max);
168. time = clock() - time;
169. cout << "计数排序（可用于基数排序，多占内存版本）的数据规模是" << Max << endl << "用时为：\t";
170. cout << time / 1000 << "s" << endl;
171. time = clock();
172. countSortEasy(randomNum3, Max, Max);
173. time = clock() - time;
174. cout << "计数排序(省内存版本)的数据规模是" << Max << endl << "用时为：\t";
175. cout << time / 1000 << "s" << endl;
176. time = clock();
177. bucketSort(randomNum4, Max);
178. time = clock() - time;
179. cout << "桶排序的数据规模是" << Max << endl << "用时为：\t";
180. cout << time / 1000 << "s" << endl;
181. time = clock();
182. madixSort(randomNum5, Max, Max);
183. time = clock() - time;
184. cout << "基数排序的数据规模是" << Max << endl << "用时为：\t";
185. cout << time / 1000 << "s" << endl;
186. **for** (**int** i = 0; i < Max; i++) {
187. **if** (randomNum1[i] != randomNum2[i]) {
188. cout << i << endl;
189. cout << randomNum1[i] << "  " << randomNum2[i] << endl;
190. }
191. **if** (randomNum1[i] != randomNum3[i]) {
192. cout << i << endl;
193. cout << randomNum1[i] << "  " << randomNum2[i] << endl;
194. }
195. **if** (randomNum1[i] != randomNum4[i]) {
196. cout << i << endl;
197. cout << randomNum1[i] << "  " << randomNum2[i] << endl;
198. }
199. **if** (randomNum1[i] != randomNum5[i]) {
200. cout << i << endl;
201. cout << randomNum1[i] << "  " << randomNum2[i] << endl;
202. }
203. }
204. **return** 0;
205. }