山东大学 计算机科学与技术 学院

数据结构与算法 课程实验报告

|  |  |  |  |
| --- | --- | --- | --- |
| 学号：202000130198 | 姓名： 隋春雨 | | 班级： 20.4 |
| 实验题目：排序算法 | | | |
| 实验学时：2 | | 实验日期： 2021-10-21 | |
| 实验目的：  1、 掌握线性表结构、链式描述方法（链式存储结构）、链表的实现。  2、 掌握链表迭代器的实现与应用。 | | | |
| 软件开发环境：  CLION2020 | | | |
| 1. 实验内容   1、题目描述：  要求封装链表类，链表迭代器类；  链表类需提供操作：在指定位置插入元素，删除指定元素，搜索链表中是否有指定元素，原地逆置链表，输出链表；  不得使用与链表实现相关的STL。  输入输出格式：  输入：第一行两个整数 N 和 Q。  第二行 N 个整数，作为节点的元素值，创建链表。  接下来 Q 行，执行各个操作，具体格式如下：  插入操作 : 1 idx val，在链表的idx位置插入元素val;  删除操作 : 2 val，删除链表中的 val 元素。若链表中存在多个该元素，仅删除第一个。若该元素不存在，输出 -1；  逆置操作 : 3，原地逆置链表；  查询操作 : 4 val，查询链表中的val元素，并输出其索引。若链表中存在多个该元素，仅输出第一个的索引。若不存在该元素，输出 -1；  输出操作 : 5，使用链表迭代器，输出当前链表索引与元素的异或和。  2、题目描述：  要求使用题目一中实现的链表类，迭代器类完成本题；  不得使用与题目实现相关的STL；  给定两组整数序列，你需要分别创建两个有序链表，使用链表迭代器实现链表的合并，并分别输出这三个有序链表的索引与元素的异或和。  注：给定序列是无序的，你需要首先得到一个有序的链表。  输入输出格式：  输入：  第一行两个整数 N 和 M；  第二行 N 个整数，代表第一组整数序列；  第三行 M 个整数，代表第二组整数序列。  输出：  三行整数。分别代表第一组数、第二组数对应的有序链表与合并后有序链表的索引与元素的异或和。   1. 数据结构与算法描述 （整体思路描述，所需要的数据结构与算法） 2. 首先对于每一个链表结点我们需要封装一个结构体，这个类需要有保存的值和指向下一个结点的指针，即为   然后我们需要站在链表的角度对结点进行统一的管理，链表需要查询索引、删除某个值、在某个特定位置插入特定的值、输出题目要求的异或值等功能，因此我们建立链表类如下：     1. 对于迭代器，我们考虑到它是一个智能指针，需要重载++ ，\*，->等运算符，我们平时用\*iter的意思是取出它所指的元素的值，iter->意思是取出它所指的结点，因此构造如下：**值得注意的是**，书上的重载->可能是错的，它返回的是&node->element，这个在clion上只能取出来element，对于next指针就不行了。      1. 我们平时还经常使用到begin(),end()等函数，这两个函数不应该是iterator里边的函数，而应该是chain的函数，因为我们使用的时候，都是站在容器的角度来使用该容器的begin()或者end()，而不是迭代器 2. A题的插入操作：我们首先要找到需要插入的位置的前一个结点，然后更改next的值，同时考虑到，它可能没有前一个结点（就是说，我们要插入的位置是fisrtNode）,那么这种情况需要特殊处理一下，让其next直接指向头结点，然后更新头结点的值，最终算法如下：      1. 删除操作： 删除操作与插入操作类似，都是需要找到被删除结点的前一个结点，然后更新next值   同样的，如果它没有前一个结点（firstNode），那么需要更改fisrtNode的值，最终算法如下：       1. 对于rerverse操作：我们需要用三个指针记录，本算法使用的是迭代器来进行操作，p1是p2的上一个结点，p3是p2的下一个结点，每次都让p2->next指向p1指向的chainNode，然后p2与p1均往后移动，因为它们原来的next已经改了，所以用p3记录，再往后移动。最终代码如下：      1. 对于查询操作：使用迭代器遍历搜寻，当没有到end()并且没有找到就++，最后判断一下是否找到即可，代码如下：      1. 输出异或和：思路与查询类似，也是遍历搜寻，最后记录一下即可      1. 对于B题使用的数据结构，与A题大体类似，都是链表与迭代器。唯一有变动的是排序算法那里，需要增加merge函数与sort函数，我们使用的sort函数是基数排序 2. Merge函数其实就是归并排序中对左右两个区间整理有序之后，放回去的过程，时间复杂度O(n)，只要没到end，就可以继续比较，代码如下：      1. 对于基数排序，由于它的精髓就是稳定排序，因此我们增加了push\_back函数，使得对于当前这一轮中的所有箱子里，它们的相对顺序是不变的（稳定），对于每一轮操作，我们都需要提取出有效的数字，放到相应的箱子里去。然后收集的时候是从前往后收集，保持稳定。代码如下：        1. 测试结果（测试输入，测试输出）   A题输入：    输出：    B题输出    输出：       1. 分析与探讨（结果分析，若存在问题，探讨解决问题的途径） 2. 测试数据的时候发现死循环了怎么办？   解决：经过debug发现，是因为短路问题    这么写会造成死循环，因为如果这个时候iter的值是nullptr，而\*nullptr是没有定义的，正确的应该是先判断iter是否为end()，即为：     1. 对于迭代器，我们是应该将其单独作为一个类合适还是放到了chain类里边合适？   解决：应该放到类里，如果不放到类里，那么我们在使用的时候就会很麻烦，对于类型的传递就要传两次。如果放到了 chain类里，那么我们使用的时候会很方便，同时对于一些chain类的操作，也可以借助iterator来实现。   1. 对于迭代器，我们是将其作为一个成员放到chain类私有成员或者共有成员里好还是作为定义放到public里好？   解决：应该作为定义放到public里好，①首先，如果作为一个私有成员，那么用户在使用的时候就无法使用了（除非调用public函数），②其次，如果作为一个成员放到Public里，那么我们用户自行定义的时候，就必须使用这个成员，会非常令人疑惑，使用成本很大，如果作为一个定义放到了chain类里就不会有什么问题。   1. 对于边界条件的判定，我们在插入与删除函数的时候，都要找到上一个结点的位置，而如果被插入和删除的结点如果是firstNode，那么它就没有上一个结点，这个时候需要特判一下 2. 在reverse函数中，第一次调用的时候跟预期结果不一样怎么办？   解决：debug发现，是因为原来的next值被更改了，而使用的时候没注意，就发生了错误。以后在写程序的时候，一定先想好逻辑在开始。同时对于每一次的更新，都需更新一下firstNode     1. 对于迭代器，我们考虑到它是一个智能指针，需要重载++ ，\*，->等运算符，我们平时用\*iter的意思是取出它所指的元素的值，iter->意思是取出它所指的结点，因此构造如下：**值得注意的是**，书上的重载->可能是错的，它返回的是&node->element，这个在clion上只能取出来element，对于next指针就不行。正确的写法应该是：      1. 对于基数排序，需要在末尾插入，如果是一个一个Insert会很慢，因此我们更新一个变量lastNode,可以帮助我们更快的插入。 2. 自己写的时候测的样例都是对的，交到oj平台上就RE了，怎么办？   解决：RE常见情况的是数组下标越界，但是经过自己debug发现，实际情况是switch case条件没有break语句，才RE，在平时，能用switch case尽量用switch case而不是If else ，因为switch case执行的次数少。   1. 在测试样例的时候发现自己的输出值跟预期不同，怎么办？   解决：经过debug发现，在删除操作的时候，对于数组的size变量没有更新，从而导致错误。以后在写函数的时候，一定需要注意的一点就是调用更新私有变量成员。  (10)一个一个写操作很麻烦怎么办？  解决：运用面向对象的思想，将函数封装为类内函数，以后只需要调用类内函数即可进行操作。   1. 附录：实现源代码（本实验的全部源程序代码，程序风格清晰易理解，有充分的注释）    * + 1. A题 2. #include <iostream> 3. **using** **namespace** std;  6. **template** <**class** T> 7. **struct** chainNode 8. { 9. T element; 10. chainNode<T>\* next;//指向下一个结点的指针 12. chainNode() {} 13. chainNode(**const** T& element) 14. { 15. **this**->element = element; 16. } 17. chainNode(**const** T& element, chainNode<T>\* next) 18. { 19. **this**->element = element; 20. **this**->next = next; 21. } 22. };   26. **template**<**class** T> 27. **class** chain 28. { 30. **public**: 31. // constructor, copy constructor and destructor 32. chain(**int** initialCapacity = 10); 33. chain(**const** chain<T>&); 34. ~chain();  37. **void** indexOf(**const** T& val) **const**;//查询索引 38. **void** erase(T val);//删除 39. **void** insert(**int** theIndex, **const** T& theElement);//插入 40. **void** output() **const**;//输出元素异或和 41. **void** reverse();//反转操作 43. **class** iterator;//迭代器 44. iterator begin() **const** { **return** iterator(firstNode); } 45. iterator end() **const** { **return** iterator(nullptr); }   49. **class** iterator 50. { 51. **public**: 52. iterator(chainNode<T>\* theNode = nullptr) 53. { 54. node = theNode; 55. } 57. T& operator\*() **const** { **return** node->element; }//重载\* 58. chainNode<T>\* operator->() **const** { **return** node; }//重载-> 60. **bool** operator!=(**const** iterator right) **const** 61. { 62. **return** node != right.node; 63. } 64. **bool** operator==(**const** iterator right) **const** 65. { 66. **return** node == right.node; 67. } 68. iterator& operator++()  //前++ 69. { 70. node = node->next; **return** \***this**; 71. } 72. iterator operator++(**int**) // 后++ 73. { 74. iterator old = \***this**; 75. node = node->next; 76. **return** old; 77. }  80. iterator operator =(**const** chainNode<T>& c\_ptr) 81. { 82. node = c\_ptr; 83. **return** iterator(node); 84. } 86. chainNode<T>\* ptr() 87. { 88. **return** node;//返回指针 89. } 91. **protected**: 92. chainNode<T>\* node; 93. }; 95. **protected**: 97. chainNode<T>\* firstNode; 98. **int** listSize; 99. };   103. **template**<**class** T> 104. **void** chain<T>::indexOf(**const** T& val) **const** 105. { 106. **int** pos = 0;//记录索引 107. iterator iter(firstNode); 108. **while** (iter != end() && \*iter != val)//如果没到end并且没找到就++ 109. { 110. iter++; 111. pos++; 112. } 113. **if** (iter == end())//没找到 114. { 115. cout << -1 << endl; 116. } 117. **else**//找到了 118. { 119. cout << pos << endl; 120. } 121. } 123. **template**<**class** T> 124. **void** chain<T>::erase(T val) 125. { 126. iterator iter = begin(); 127. iterator pre(nullptr);//前一个结点 128. **while** (iter != end()&& \*iter != val  ) 129. { 130. pre = iter; 131. iter++; 132. } 133. **if** (iter == end())//如果没找到 134. { 135. cout << -1 << endl; 136. } 137. **else** 138. { 139. //找到 140. **if** (iter == begin()) 141. { 142. firstNode = firstNode->next; 143. listSize--; 144. } 145. **else** 146. { 147. pre->next = iter->next; 148. **delete** iter.ptr(); 149. listSize--; 150. } 151. } 152. } 154. **template**<**class** T> 155. **void** chain<T>::insert(**int** theIndex, **const** T& theElement) 156. { 157. **if** (theIndex == 0)//插入到头结点 158. firstNode = **new** chainNode<T>(theElement, firstNode); 159. **else** 160. { 161. chainNode<T>\* p = firstNode; 162. **for** (**int** i = 0; i < theIndex - 1; i++) 163. p = p->next;//找到前一个结点 165. p->next = **new** chainNode<T>(theElement, p->next); 166. } 167. listSize++; 168. } 170. **template**<**class** T> 171. **void** chain<T>::output() **const** 172. { 173. **int** pos = 0; 174. **int** ans = 0; 175. **for** (iterator iter = begin(); iter != end(); iter++, pos++)//没到end()就++ 176. { 177. ans += \*iter ^ pos; 178. } 179. cout << ans << endl; 180. }    185. **template** <**class** T> 186. **void** chain<T>::reverse() 187. { 188. //构造函数规定了至少要有一个结点 189. chain<T>::iterator p1(firstNode);//p1为p2的上一个结点 190. chain<T>::iterator p2(p1->next); 191. p1->next = nullptr;//p1是firstNode，故reverse之后一定是最后一个结点 192. **while** (p2 != nullptr) 193. { 194. chain<T>::iterator p3(p2->next); 195. p2->next = p1.ptr(); 196. p1 = p2;//记录一下p2 197. firstNode = p2.ptr();//每次都更新一下firstNode 198. p2 = p3;//移动p2 199. } 200. } 202. **template**<**class** T> 203. chain<T>::chain(**int** initialCapacity) 204. { 205. firstNode = nullptr; 206. listSize = 0; 207. }  210. **template**<**class** T> 211. chain<T>::~chain() 212. {// Chain destructor. Delete all nodes in chain. 213. chainNode<T>\* nextNode; 214. **while** (firstNode != NULL) 215. {// delete firstNode 216. nextNode = firstNode->next; 217. **delete** firstNode; 218. firstNode = nextNode; 219. } 220. }   224. **int** main() 225. { 226. **int** n, q; 227. cin >> n >> q; 228. chain<**int**>my\_chain; 229. **for** (**int** i = 0; i < n; i++) 230. { 231. **int** val; 232. cin >> val; 233. my\_chain.insert(i, val);//插入到相应的位置 234. } 235. **int** flag; 236. **int** idx, val; 237. **for** (**int** i = 0; i < q; i++) 238. { 239. cin >> flag;//标记 240. **switch** (flag) 241. { 242. **case** 1: 243. cin >> idx >> val; 244. my\_chain.insert(idx, val); 245. **break**; 246. **case** 2: 247. cin >> val; 248. my\_chain.erase(val); 249. **break**; 250. **case** 3: 251. my\_chain.reverse(); 252. **break**; 253. **case** 4: 254. cin >> val; 255. my\_chain.indexOf(val); 256. **break**; 257. **case** 5: 258. my\_chain.output(); 259. **break**; 260. } 261. } 262. **return** 0; 263. }   **(2)**   * + - 1. B题  1. #include <iostream> 2. #include <cmath> 3. #include <ctime> 5. **using** **namespace** std; 7. **template** <**class** T> 8. **struct** chainNode 9. { 10. T element; 11. chainNode<T>\* next; 13. chainNode() {} 15. chainNode(**const** T& element, chainNode<T>\* next) 16. { 17. **this**->element = element; 18. **this**->next = next; 19. } 20. };  23. **template**<**class** T> 24. **class** chain 25. { 27. **public**: 28. // constructor, copy constructor and destructor 29. chain(**int** initialCapacity = 10); 30. chain(**const** chain<T>&); 31. ~chain(); 33. // ADT methods 34. **bool** empty() **const** { **return** listSize == 0; } 35. **int** size() **const** { **return** listSize; } 37. **void** insert(**int** theIndex, **const** T& theElement); 38. **void** output() **const**; 40. **void** push\_back(T& val); 41. T& get(**int** theIndex) **const**; 43. **class** iterator; 44. iterator begin() **const** { **return** iterator(firstNode); } 45. iterator end() **const** { **return** iterator(NULL); } 46. //作业 47. **void** erase(**int** theIndex); 48. **void** merge(chain<T>& c1, chain<T>& c2); 50. **void** insertSort(); 51. **void** test() 52. { 53. iterator iter(firstNode); 54. **while** (iter != end()) 55. { 56. cout << \*iter++<<" "; 57. } 58. cout << endl; 59. } 60. **void** radixSort(**int** r, **int** d); 61. **void** sort\_by\_radix10(); 63. **class** iterator 64. { 65. **public**: 67. iterator(chainNode<T>\* theNode = nullptr) 68. { 69. node = theNode; 70. } 72. iterator(**const** T& val, chainNode<T>\* next) 73. { 74. node = **new** chainNode<T>(val, next); 76. } 77. T& operator\*() **const** { **return** node->element; } 78. chainNode<T>\* operator->() **const** { **return** node; } 80. iterator& operator++()   // preincrement 81. { 82. node = node->next; **return** \***this**; 83. } 84. iterator operator++(**int**) // postincrement 85. { 86. iterator old = \***this**; 87. node = node->next; 88. **return** old; 89. } 91. // equality testing 92. **bool** operator!=(**const** iterator right) **const** 93. { 94. **return** node != right.node; 95. } 96. **bool** operator==(**const** iterator right) **const** 97. { 98. **return** node == right.node; 99. } 101. iterator operator =(**const** chainNode<T>& c\_ptr) 102. { 103. node = c\_ptr; 104. **return** iterator(node); 105. } 107. chainNode<T>\* ptr() 108. { 109. **return** node; 110. } 112. **protected**: 113. chainNode<T>\* node; 114. };  // end of iterator class 116. **protected**: 118. chainNode<T>\* firstNode;  // pointer to first node in chain 119. chainNode<T>\* lastNode; 120. **int** listSize;             // number of elements in list 121. }; 123. **template**<**class** T> 124. chain<T>::chain(**int** initialCapacity) 125. {// Constructor. 127. firstNode = nullptr; 128. lastNode = nullptr; 129. listSize = 0; 130. }  133. **template**<**class** T> 134. chain<T>::~chain() 135. {// Chain destructor. Delete all nodes in chain. 136. chainNode<T>\* nextNode; 137. **while** (firstNode != NULL) 138. {// delete firstNode 139. nextNode = firstNode->next; 140. **delete** firstNode; 141. firstNode = nextNode; 142. } 143. } 145. **template**<**class** T> 146. **void** chain<T>::insert(**int** theIndex, **const** T& theElement) 147. {// Insert theElement so that its index is theIndex.  150. **if** (theIndex == 0) 151. { 152. firstNode = **new** chainNode<T>(theElement, firstNode); 153. lastNode = firstNode; 154. } 155. **else** 156. {  // find predecessor of new element 157. chainNode<T>\* p = firstNode; 158. **for** (**int** i = 0; i < theIndex - 1; i++) 159. p = p->next; 161. p->next = **new** chainNode<T>(theElement, p->next); 162. **if** (theIndex == listSize) 163. { 164. lastNode = p->next; 165. } 166. } 167. listSize++; 168. }  171. **template**<**class** T> 172. **void** chain<T>::output() **const** 173. { 175. **int** ans = 0; 176. **int** index = 0; 177. **for** (iterator iter = begin(); iter != end(); iter++, index++) 178. { 179. ans += index ^ \*iter; 180. } 181. cout << ans << endl; 182. }   186. **template**<**class** T> 187. **void** chain<T>::merge(chain<T>& a, chain<T>& b) 188. { 190. iterator a\_iter(a.firstNode);//a的迭代器 191. iterator b\_iter(b.firstNode);//b的迭代器 192. iterator end(nullptr); 194. **while** (a\_iter != end && b\_iter != end)//只要没到最后，就可以继续 195. { 196. push\_back(\*a\_iter <= \*b\_iter ? \*a\_iter++ : \*b\_iter++); 197. } 198. **while** (a\_iter != end)//把a剩下的元素都Push\_back进去 199. { 200. push\_back(\*a\_iter++); 201. } 202. **while** (b\_iter != end) 203. { 204. push\_back(\*b\_iter++); 205. } 207. listSize = a.listSize + b.listSize;//更新私有变量的值 208. } 210. **template**<**class** T> 211. **void** chain<T>::erase(**int** theIndex) 212. { 213. chainNode<T>\* deleteNode; 214. **if** (theIndex == 0) 215. { 216. deleteNode = firstNode; 217. firstNode = firstNode->next; 219. } 220. **else** 221. { 222. chainNode<T>\* p = firstNode; 223. **for** (**int** i = 0; i < theIndex - 1; i++) 224. p = p->next; 226. deleteNode = p->next; 227. p->next = p->next->next; // remove deleteNode from chain 228. } 229. **if** (theIndex == listSize) 230. { 231. lastNode = nullptr; 232. } 233. listSize--; 234. **delete** deleteNode; 235. } 237. **template**<**class** T> 238. T& chain<T>::get(**int** theIndex) **const** 239. { 241. chainNode<T>\* currentNode = firstNode; 242. **for** (**int** i = 0; i < theIndex; i++) 243. currentNode = currentNode->next; 245. **return** currentNode->element; 246. }  249. **template**<**class** T> 250. **void** chain<T>::insertSort() 251. { 252. **if** (listSize == 0 || listSize == 1) 253. { 254. **return**; 255. } 257. iterator pre(firstNode); 258. iterator iter(firstNode->next); 259. **while** (iter != nullptr) 260. { 261. iterator pos(firstNode);  264. **if** (\*iter <= \*pos) 265. { 266. pre->next = iter->next; 267. iter->next = firstNode; 268. firstNode = iter.ptr(); 269. iter = pre->next;//因为iter已经被更新了，所以我们要借助pre指针进行更新 270. } 271. **else** 272. { 273. **while** (pos->next->element < \*iter)//iter的存在确保了不会越界，类似于放置一个哨兵 274. { 275. pos++; 276. } 277. **if** (pos == pre) 278. { 279. pre++; 280. iter++; 281. **continue**; 282. } 283. **else** 284. { 285. pre->next = iter->next; 286. iter->next = pos->next; 287. pos->next = iter.ptr(); 288. iter = pre->next;//因为iter已经指向了头节点，所以我们要借助pre指针进行更新 289. } 290. } 291. } 292. } 294. **template**<**class** T> 295. **void** chain<T>::radixSort(**int** r, **int** d)//r=range, d=the number of loop, 296. { 297. //指针数组 298. //从后往前搜，保证了稳定性 299. chain<T>\* bin = **new** chain<T>[r]; 300. **for** (**int** i = 0; i < d; i++)//循环次数 301. { 302. **int** l = listSize;//防止listSize被破坏 303. **for** (**int** j = 0; j < l; j++) 304. { 305. //将每一位都变成箱子排序 306. **int** index;//放到哪个箱子里边 307. index = (**this**->get(0) / (**int** (pow(r, i)))) % r; 308. bin[index].push\_back( **this**->get(0)); 309. **this**->erase(0); 310. } 311. //从前往后收集 312. **for** (**int** j = 1; j < r; j++) 313. { 314. **while** (!bin[j].empty()) 315. { 316. **this**->push\_back( bin[j].get(0)); 317. bin[j].erase(0); 318. } 319. } 320. } 321. **delete**[]bin; 322. bin = nullptr; 323. }  326. **template**<**class** T> 327. **void** chain<T>::sort\_by\_radix10() 328. { 329. **if** (listSize == 0 || listSize == 1) 330. { 331. **return**; 332. } 333. iterator iter = begin(); 334. **int** \_max = -1; 335. **while** (iter != end()) 336. { 337. \_max = max(\*iter, \_max); 338. iter++; 339. } 340. **int** loop\_num = 0; 341. **while** (\_max) 342. { 343. loop\_num++; 344. \_max /= 10; 345. } 346. radixSort(10, loop\_num); 347. } 349. **template**<**class** T> 350. **void** chain<T>::push\_back(T& val) 351. { 352. **if** (listSize == 0) 353. { 354. //iterator temp(val, nullptr); 355. chainNode<T>\* temp = **new** chainNode<T>(val, nullptr); 356. lastNode = firstNode = temp; 357. } 358. **else** 359. { 360. chainNode<T>\* temp = **new** chainNode<T>(val, nullptr); 361. lastNode->next = temp; 362. lastNode = temp; 363. } 364. listSize++; 365. } 367. **int** main() 368. { 370. **int** n, m; 371. cin >> n >> m;  374. chain<**int**>my\_chain1; 375. chain<**int**>my\_chain2; 376. chain<**int**>result;//结果链表 377. **int** val; 379. **for** (**int** i = 0; i < n; i++) 380. { 381. cin >> val; 382. my\_chain1.insert(0, val); 383. }  386. **for** (**int** i = 0; i < m; i++) 387. { 388. cin >> val; 389. my\_chain2.insert(0, val); 390. }  393. my\_chain1.sort\_by\_radix10();//基数排序 394. my\_chain2.sort\_by\_radix10();//基数排序 396. result.merge(my\_chain1, my\_chain2);//合并操作 398. my\_chain1.output();//输出异或和 399. my\_chain2.output(); 400. result.output();  403. **return** 0; 404. } | | | |