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：202300130183 | 姓名： 宋浩宇 | | 班级：23级人工智能班 |
| 实验题目：2024数据结构--数据/智能 实验4 链式描述线性表 | | | |
| 实验学时：2 | | 实验日期：2024/10/9 | |
| 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。 | | | |
| 软件开发工具：   1. visual studio code 2022（使用C/C++、C/C++ Extension Pack、C/C++ Themes插件） 2. mingw64工具包 | | | |
| 1. 实验内容   完成2024数据结构--数据/智能 实验4 链式描述线性表 A ： 链表实现，B ： 链表合并两道题目。  （附加要求：   1. 要求封装链表类，链表迭代器类； 2. 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； 3. 不得使用与链表实现相关的 STL。） 4. 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）   本实验全程使用了双向链表数据结构，另外还使用了快速排序算法。  首先是第一题，链表实现，就本题的题目来看并不难，主要在于链表增删查改实现的思路。链表的插入操作实现为：从链表的头开始遍历到要插入的索引处，然后将该索引处前一个元素存储的下一个元素的地址改为插入元素的地址，插入元素的上一个元素的地址改为该索引处的前一个元素的地址，插入元素的下一个元素的地址改为该索引处原本元素的地址，该索引处原本元素的前一个元素的地址改为插入元素的地址。删除操作，首先遍历到需要删除的元素的索引位置，然后将该索引的前一个元素的下一个元素的地址改为要删除的元素的下一个元素的地址，再将要删除的元素的下一个元素的前一个元素的地址改为该索引处前一个元素的地址。查找操作，从链表头开始遍历，每次更新节点指针使其指向下一个节点，比较该节点处存储的数据，并记录该节点的索引，找到需要查找的元素后返回该元素所在节点的索引，如果遍历到链表的最后一个元素还没有找到需要的元素，则返回-1。更改操作，基于查找操作，先查找到要修改的元素的索引，在查找时可以选择返回对应元素的指针，也可以选择返回对应元素的引用，在对返回值进行修改。以上为增删查改操作的实现思路，考虑到题目要求，再基于基础功能添加新的功能，首先是原地反转，实现思路为：从链表的头节点开始遍历，交换每个节点的前一个节点的地址和后一个节点的地址，再将链表数据结构的头节点的地址和尾节点的地址交换。另外我们还需要书写链表的迭代器，迭代器的设计主要为，一个指针类型的数据成员，为迭代器指向的元素，索引值，为该迭代器所指向的元素在对应容器中的索引，自增自减大于等于小于的运算符重载，用于地址的改变。然后我们可以通过迭代器自增的方式来完成对于链表数据结构的遍历（这个自增的实现为：将指针类型的数据成员改为这个指针所指向的节点的下一个节点的地址）。  然后是第二题，首先我们可以套用前一个问题所写的链表类和迭代器类，然后，对这个链表类进行拓展，首先是进行两个链表类之间加法的运算符重载，让两个链表的链接由加法的形式实现，实际实现起来也很简单，只要把两个链表进行一次深复制，再将前一个链表的尾节点的下一个节点的地址改为后一个链表的头节点的地址，再依次更新地址即可。  本题还需要完成链表的排序，并要求使用迭代器来进行排序，经过实验，使用冒泡排序会出现超时的情况，因此我们以迭代器的方式来实现快速排序，快速排序的描述与本节无关，故省略，详细内容请见附件中的quick\_sort()函数的代码。通过迭代器的方式，我们可以用通用的方法操作线性容器，即可以将所有的线性容器都视为数组，只要做好运算符的重载即可。   1. 测试结果（测试输入，测试输出）   第一题测试输入：  10 10  6863 35084 11427 53377 34937 14116 5000 49692 70281 73704  4 6863  1 2 44199  5  4 21466  1 6 11483  5  4 34937  5  4 6863  1 10 18635  输出：  0  398665  -1  410141  5  410141  0  第二题测试输入：  3 0  3 1 2  输出：  5  0  5   1. 分析与探讨（结果分析，若存在问题，探讨解决问题的途径）   从结果来看，我们成功解决了这两道算法题，且自行书写出了链表的类模板和迭代器的类模板，并且通过迭代器的方式将排序算法封装成了线性容器中的通用方法。成功掌握了线性表结构、链式描述方法（链式存储结构）、链表的实现、链表迭代器的实现与应用。  也进一步深刻理解了面向对象编程的多态的特点。   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） 2. /\*2024数据结构--数据智能 实验4 链式描述线性表 A 链表实现.cpp\*/ 3. #include<iostream> 4. **using** **namespace** std;  7. **template**<**class** T> 8. **class** list\_node 9. { 10. **private**: 11. T data; 13. **public**: 14. list\_node<T>\* front; 15. list\_node<T>\* next; 16. **bool** operator==(**const** list\_node<T>& list\_node) **const** { **return** **this**->data == list\_node.data; }; 17. T& getData() { **return** data; }; 18. list\_node<T>\* getFront() { **return** front; }; 19. list\_node<T>\* getNext() { **return** next; }; 20. list\_node(){}; 21. list\_node(**const** T& data, list\_node<T>\* front = nullptr, list\_node<T>\* next = nullptr); 22. list\_node(list\_node<T>\* **const** list\_node) { **this**->data = list\_node->data;**this**->front = list\_node->front;**this**->next = list\_node->next; }; 23. }; 25. **template**<**class** T> 26. list\_node<T>::list\_node(**const** T& data,list\_node<T>\* front,list\_node<T>\* next) 27. :data(data),front(front),next(next) 28. { 30. }   34. **template**<**class** T> 35. **class** chainlist 36. { 37. **private**: 38. list\_node<T>\* head; 39. list\_node<T>\* back; 40. unsigned **int** size; 41. **public**: 42. **void** push\_back(list\_node<T>\* operated\_list\_node); 43. **void** push\_back(**const** T& data); 44. **void** push\_front(list\_node<T>\* operated\_list\_node); 45. **void** push\_front(**const** T& data); 46. **void** pop\_back(); 47. **void** pop\_front(); 48. **void** insert(T data, **int** index); 49. **void** insert(list\_node<T>\* operated\_list\_node, **int** index); 50. **void** erase(**const** unsigned **int** index); 51. unsigned **int** getSize() { **return** size; }; 52. unsigned **int** find\_first(list\_node<T>\* operated\_list\_node); 53. unsigned **int** find\_first(**const** T& data); 54. **void** inverse(); 55. chainlist(); 56. chainlist(**const** chainlist<T>& chainlist); 57. **void** print(); 58. list\_node<T>\* getHead() { **return** head; }; 59. list\_node<T>\* getBack() { **return** back; }; 61. }; 63. **template**<**class** T> 64. **void** chainlist<T>::push\_back(list\_node<T>\* operated\_list\_node) 65. { 66. list\_node<T>\* new\_list\_node = **new** list\_node<T>(operated\_list\_node);  69. **if** (size == 0) 70. { 71. **this**->head = new\_list\_node; 72. **this**->back = new\_list\_node; 73. size++; 74. **return**; 75. } 77. **this**->back->next = new\_list\_node; 78. new\_list\_node->front = **this**->back; 79. new\_list\_node->next = nullptr; 80. **this**->back = new\_list\_node; 82. size++; 83. } 85. **template**<**class** T> 86. **void** chainlist<T>::push\_back(**const** T& data) 87. { 88. T new\_data = data; 89. list\_node<T>\* new\_list\_node = **new** list\_node<T>(new\_data); 90. **this**->push\_back(new\_list\_node); 91. **delete** new\_list\_node; 92. } 94. **template**<**class** T> 95. **void** chainlist<T>::push\_front(list\_node<T>\* operated\_list\_node) 96. { 97. list\_node<T>\* new\_list\_node = **new** list\_node<T>(operated\_list\_node); 99. **if** (size == 0) 100. { 101. **this**->back = new\_list\_node; 102. **this**->head = new\_list\_node; 103. size++; 104. **return**; 105. } 106. new\_list\_node->next = **this**->head; 107. new\_list\_node->front = nullptr; 108. **this**->head->front = new\_list\_node; 109. **this**->head = new\_list\_node; 111. size++; 112. } 114. **template**<**class** T> 115. **void** chainlist<T>::push\_front(**const** T& data) 116. { 117. T new\_data = data; 118. list\_node<T>\* new\_list\_node = **new** list\_node<T>(new\_data); 119. **this**->push\_front(new\_list\_node); 120. **delete** new\_list\_node; 121. } 123. **template**<**class** T> 124. **void** chainlist<T>::pop\_back() 125. { 126. **if** (**this**->back == nullptr) **return**; 127. list\_node<T>\* temp = **this**->back; 128. **this**->back = **this**->back->front; 129. **delete** temp; 130. **if** (size == 1) 131. { 132. **this**->head = nullptr; 133. **this**->back = nullptr; 134. } 136. size--; 137. } 139. **template**<**class** T> 140. **void** chainlist<T>::pop\_front() 141. { 142. **if** (**this**->head == nullptr) **return**; 143. list\_node<T>\* temp = **this**->head; 144. **this**->head = **this**->head->next; 145. **delete** temp; 146. **if** (size == 1) 147. { 148. **this**->head = nullptr; 149. **this**->back = nullptr; 150. } 151. size--; 152. }  155. **template**<**class** T> 156. **void** chainlist<T>::insert(list\_node<T>\* operated\_list\_node, **int** index) 157. { 158. **if** (index == 0) 159. { 160. push\_front(operated\_list\_node); 161. **return**; 162. } 163. **if** (index == size) 164. { 165. push\_back(operated\_list\_node); 166. **return**; 167. } 169. list\_node<T>\* temp = **this**->head; 170. **for** (**int** i = 0; i < index - 1; i++) 171. { 172. temp = temp->next; 173. } 174. list\_node<T>\* new\_list\_node = **new** list\_node<T>(operated\_list\_node); 175. new\_list\_node->next = temp->next; 176. temp->next->front = new\_list\_node; 177. temp->next = new\_list\_node; 178. new\_list\_node->front = temp; 179. size++; 180. } 182. **template**<**class** T> 183. **void** chainlist<T>::insert(T data, **int** index) 184. { 185. list\_node<T>\* new\_list\_node = **new** list\_node<T>(data); 186. **this**->insert(new\_list\_node, index); 187. **delete** new\_list\_node; 189. } 191. **template**<**class** T> 192. **void** chainlist<T>::erase(**const** unsigned **int** index) 193. { 194. **if** (index == 0) 195. { 196. pop\_front(); 197. **return**; 198. } 199. **if** (index == size) 200. { 201. pop\_back(); 202. **return**; 203. } 205. list\_node<T>\* temp = **this**->head; 206. **for** (**int** i = 0; i < index - 1; i++) 207. { 208. temp = temp->next; 209. } 210. list\_node<T>\* temp2 = temp->next; 211. temp->next = temp2->next; 212. temp->next->front = temp; 213. **delete** temp2; 214. size--; 215. } 217. **template**<**class** T> 218. unsigned **int** chainlist<T>::find\_first(list\_node<T>\* operated\_list\_node) 219. { 220. list\_node<T>\* temp = **this**->head; 221. unsigned **int** index = 0; 222. **while** (temp != nullptr) 223. { 224. **if** ((\*temp) == (\*operated\_list\_node)) **return** index; 225. temp = temp->next; 226. index++; 227. } 228. **return** -1; 229. } 231. **template**<**class** T> 232. unsigned **int** chainlist<T>::find\_first(**const** T& data) 233. { 234. list\_node<T>\* temp = **this**->head; 235. unsigned **int** index = 0; 236. **while** (temp != nullptr) 237. { 238. **if** (temp->getData() == data) **return** index; 239. temp = temp->next; 240. index++; 241. } 242. **return** -1; 243. } 245. **template**<**class** T> 246. **void** chainlist<T>::inverse() 247. { 248. list\_node<T>\* temp1 = **this**->head; 249. list\_node<T>\* temp2 = nullptr; 250. list\_node<T>\* temp3 = nullptr; 251. **for** (**int** i = 0; i < size; i++) 252. { 253. temp2 = temp1->next; 254. temp3 = temp1->front; 255. temp1->front = temp1->next; 256. temp1->next = temp3; 257. temp1 = temp2; 258. } 259. temp1 = **this**->head; 260. **this**->head = **this**->back; 261. **this**->back = temp1; 262. } 264. **template**<**class** T> 265. chainlist<T>::chainlist() 266. { 267. **this**->head = nullptr; 268. **this**->back = nullptr; 269. **this**->size = 0; 270. } 272. **template**<**class** T> 273. chainlist<T>::chainlist(**const** chainlist<T>& chainlist) 274. { 275. **this**->head = nullptr; 276. **this**->back = nullptr; 277. **this**->size = 0; 278. list\_node<T>\* temp = chainlist.head; 279. **while** (temp != nullptr) 280. { 281. push\_back(temp); 282. temp = temp->next; 283. } 284. } 286. **template**<**class** T> 287. **void** chainlist<T>::print() 288. { 289. list\_node<T>\* temp = **this**->head; 290. **while** (temp != nullptr) 291. { 292. cout << temp->getData() << " "; 293. temp = temp->next; 294. } 295. cout << endl; 296. }  299. **template**<**class** T> 300. **class** my\_iterator 301. { 302. **private**: 303. list\_node<T>\* current; 304. unsigned **int** index; 305. **public**: 306. my\_iterator& operator++() { **this**->current = **this**->current->next; **this**->index++; **return** \***this**; }; 307. my\_iterator& operator++(**int**) { my\_iterator temp = \***this**; **this**->current = **this**->current->next;**this**->index++; **return** temp; }; 308. my\_iterator(list\_node<T>\* current) { **this**->current = current; **this**->index = 0; }; 309. list\_node<T>\* getCurrent() { **return** **this**->current; }; 310. unsigned **int** getIndex() { **return** **this**->index; }; 311. };  314. **class** Solution 315. { 316. **public**: 317. **void** solote(); 318. }; 320. **void** Solution::solote() 321. { 322. **int** n, m; 323. cin >> n >> m; 324. chainlist<**int**> list; 325. **for** (**int** i = 0; i < n; i++) 326. { 327. **int** x; 328. cin >> x; 329. list.push\_back(x); 330. } 331. **for** (**int** i = 0; i < m; i++) 332. { 333. **int** operation; 334. cin >> operation; 335. **if** (operation == 1) 336. { 337. **int** index, value; 338. cin >> index >> value; 339. list.insert(value, index); 340. } 341. **else** **if** (operation == 2) 342. { 343. **int** value; 344. cin >> value; 345. **if** (list.find\_first(value) != -1) 346. { 347. list.erase(list.find\_first(value)); 348. } 349. **else** 350. { 351. cout << -1 << endl; 352. } 354. } 355. **else** **if** (operation == 3) 356. { 357. list.inverse(); 358. } 359. **else** **if** (operation == 4) 360. { 361. **int** value; 362. cin >> value; 363. cout << (**int**)list.find\_first(value)<<endl; 364. } 365. **else** **if** (operation == 5) 366. { 367. my\_iterator<**int**> it(list.getHead()); 368. **int** ans = 0; 369. **for** (**int** j = 0;j < list.getSize();j++) 370. { 371. ans += it.getCurrent()->getData() ^ it.getIndex(); 372. it++; 373. } 374. cout << ans << endl; 375. } 377. } 378. }  381. **int** main() 382. { 383. Solution solution; 384. solution.solote(); 385. // system("pause"); 386. **return** 0; 387. } 388. /\*2024数据结构--数据智能 实验4 链式描述线性表 B 链表合并.cpp\*/ 389. #include<iostream> 390. **using** **namespace** std;  393. **template**<**class** T> 394. **void** my\_swap(T& a, T& b) 395. { 396. T temp = a; 397. a = b; 398. b = temp; 399. } 401. **template**<**class** T> 402. **class** list\_node 403. { 404. **private**: 405. T data; 407. **public**: 408. list\_node<T>\* front; 409. list\_node<T>\* next; 410. **bool** operator==(**const** list\_node<T>& list\_node) **const** { **return** **this**->data == list\_node.data; }; 411. T& getData() { **return** data; }; 412. list\_node<T>\* getFront() { **return** front; }; 413. list\_node<T>\* getNext() { **return** next; }; 414. list\_node(){}; 415. list\_node(**const** T& data, list\_node<T>\* front = nullptr, list\_node<T>\* next = nullptr); 416. list\_node(list\_node<T>\* **const** list\_node) { **this**->data = list\_node->data;**this**->front = list\_node->front;**this**->next = list\_node->next; }; 417. }; 419. **template**<**class** T> 420. list\_node<T>::list\_node(**const** T& data,list\_node<T>\* front,list\_node<T>\* next) 421. :data(data),front(front),next(next) 422. { 424. }  427. **template**<**class** T> 428. **class** my\_iterator 429. { 430. **private**: 431. list\_node<T>\* current; 432. unsigned **int** index; 433. **public**: 434. my\_iterator& operator++() { **this**->current = **this**->current->next; **this**->index++; **return** \***this**; }; 435. my\_iterator& operator++(**int**) { my\_iterator temp = \***this**; **this**->current = **this**->current->next;**this**->index++; **return** temp; }; 436. my\_iterator& operator--() { **this**->current = **this**->current->front; **this**->index--; **return** \***this**; }; 437. my\_iterator& operator--(**int**) { my\_iterator temp = \***this**; **this**->current = **this**->current->front;**this**->index--; **return** temp; }; 438. **bool** operator==(**const** my\_iterator& my\_iterator) **const** { **return** **this**->current == my\_iterator.current; }; 439. **bool** operator!=(**const** my\_iterator& my\_iterator) **const** { **return** **this**->current != my\_iterator.current; }; 440. **bool** operator<(**const** my\_iterator& my\_iterator) **const** { **return** **this**->index < my\_iterator.index; }; 441. my\_iterator(list\_node<T>\* current, unsigned **int** index = 0) { **this**->current = current; **this**->index = index; }; 442. list\_node<T>\* getCurrent() { **return** **this**->current; }; 443. unsigned **int** getIndex() { **return** **this**->index; }; 444. **void** reset(list\_node<T>\* current) { **this**->current = current; **this**->index = 0; }; 445. }; 447. **template**<**class** T> 448. **class** chainlist 449. { 450. **private**: 451. list\_node<T>\* head; 452. list\_node<T>\* back; 453. unsigned **int** size; 454. **public**: 455. **void** push\_back(list\_node<T>\* operated\_list\_node); 456. **void** push\_back(**const** T& data); 457. **void** push\_front(list\_node<T>\* operated\_list\_node); 458. **void** push\_front(**const** T& data); 459. **void** pop\_back(); 460. **void** pop\_front(); 461. **void** insert(T data, **int** index); 462. **void** insert(list\_node<T>\* operated\_list\_node, **int** index); 463. **void** erase(**const** unsigned **int** index); 464. unsigned **int** getSize() { **return** size; }; 465. unsigned **int** find\_first(list\_node<T>\* operated\_list\_node); 466. unsigned **int** find\_first(**const** T& data); 467. **void** inverse(); 468. chainlist(); 469. chainlist(**const** chainlist<T>& chainlist); 470. **void** self\_bubble\_sort(); 471. **void** self\_quick\_sort(); 472. **void** print(); 473. list\_node<T>\* getNode(**int** index); 474. list\_node<T>\* operator[](**int** index) { list\_node<T>\* temp = **this**->head; **for** (**size\_t** i = 0; i < index; i++) { temp = temp->next; } **return** temp; }; 475. list\_node<T>\* getHead() { **return** head; }; 476. list\_node<T>\* getBack() { **return** back; }; 478. }; 480. **template**<**class** T> 481. **void** chainlist<T>::push\_back(list\_node<T>\* operated\_list\_node) 482. { 483. list\_node<T>\* new\_list\_node = **new** list\_node<T>(operated\_list\_node);  486. **if** (size == 0) 487. { 488. **this**->head = new\_list\_node; 489. **this**->back = new\_list\_node; 490. size++; 491. **return**; 492. } 494. **this**->back->next = new\_list\_node; 495. new\_list\_node->front = **this**->back; 496. new\_list\_node->next = nullptr; 497. **this**->back = new\_list\_node; 499. size++; 500. } 502. **template**<**class** T> 503. **void** chainlist<T>::push\_back(**const** T& data) 504. { 505. T new\_data = data; 506. list\_node<T>\* new\_list\_node = **new** list\_node<T>(new\_data); 507. **this**->push\_back(new\_list\_node); 508. **delete** new\_list\_node; 509. } 511. **template**<**class** T> 512. **void** chainlist<T>::push\_front(list\_node<T>\* operated\_list\_node) 513. { 514. list\_node<T>\* new\_list\_node = **new** list\_node<T>(operated\_list\_node); 516. **if** (size == 0) 517. { 518. **this**->back = new\_list\_node; 519. **this**->head = new\_list\_node; 520. size++; 521. **return**; 522. } 523. new\_list\_node->next = **this**->head; 524. new\_list\_node->front = nullptr; 525. **this**->head->front = new\_list\_node; 526. **this**->head = new\_list\_node; 528. size++; 529. } 531. **template**<**class** T> 532. **void** chainlist<T>::push\_front(**const** T& data) 533. { 534. T new\_data = data; 535. list\_node<T>\* new\_list\_node = **new** list\_node<T>(new\_data); 536. **this**->push\_front(new\_list\_node); 537. **delete** new\_list\_node; 538. } 540. **template**<**class** T> 541. **void** chainlist<T>::pop\_back() 542. { 543. **if** (**this**->back == nullptr) **return**; 544. list\_node<T>\* temp = **this**->back; 545. **this**->back = **this**->back->front; 546. **delete** temp; 547. **if** (size == 1) 548. { 549. **this**->head = nullptr; 550. **this**->back = nullptr; 551. } 553. size--; 554. } 556. **template**<**class** T> 557. **void** chainlist<T>::pop\_front() 558. { 559. **if** (**this**->head == nullptr) **return**; 560. list\_node<T>\* temp = **this**->head; 561. **this**->head = **this**->head->next; 562. **delete** temp; 563. **if** (size == 1) 564. { 565. **this**->head = nullptr; 566. **this**->back = nullptr; 567. } 568. size--; 569. } 571. **template**<**class** T> 572. **void** chainlist<T>::insert(list\_node<T>\* operated\_list\_node, **int** index) 573. { 574. **if** (index == 0) 575. { 576. push\_front(operated\_list\_node); 577. **return**; 578. } 579. **if** (index == size) 580. { 581. push\_back(operated\_list\_node); 582. **return**; 583. } 585. list\_node<T>\* temp = **this**->head; 586. **for** (**int** i = 0; i < index - 1; i++) 587. { 588. temp = temp->next; 589. } 590. list\_node<T>\* new\_list\_node = **new** list\_node<T>(operated\_list\_node); 591. new\_list\_node->next = temp->next; 592. temp->next->front = new\_list\_node; 593. temp->next = new\_list\_node; 594. new\_list\_node->front = temp; 595. size++; 596. } 598. **template**<**class** T> 599. **void** chainlist<T>::insert(T data, **int** index) 600. { 601. list\_node<T>\* new\_list\_node = **new** list\_node<T>(data); 602. **this**->insert(new\_list\_node, index); 603. **delete** new\_list\_node; 605. } 607. **template**<**class** T> 608. **void** chainlist<T>::erase(**const** unsigned **int** index) 609. { 610. **if** (index == 0) 611. { 612. pop\_front(); 613. **return**; 614. } 615. **if** (index == size) 616. { 617. pop\_back(); 618. **return**; 619. } 621. list\_node<T>\* temp = **this**->head; 622. **for** (**int** i = 0; i < index - 1; i++) 623. { 624. temp = temp->next; 625. } 626. list\_node<T>\* temp2 = temp->next; 627. temp->next = temp2->next; 628. temp->next->front = temp; 629. **delete** temp2; 630. size--; 631. } 633. **template**<**class** T> 634. unsigned **int** chainlist<T>::find\_first(list\_node<T>\* operated\_list\_node) 635. { 636. list\_node<T>\* temp = **this**->head; 637. unsigned **int** index = 0; 638. **while** (temp != nullptr) 639. { 640. **if** ((\*temp) == (\*operated\_list\_node)) **return** index; 641. temp = temp->next; 642. index++; 643. } 644. **return** -1; 645. } 647. **template**<**class** T> 648. unsigned **int** chainlist<T>::find\_first(**const** T& data) 649. { 650. list\_node<T>\* temp = **this**->head; 651. unsigned **int** index = 0; 652. **while** (temp != nullptr) 653. { 654. **if** (temp->getData() == data) **return** index; 655. temp = temp->next; 656. index++; 657. } 658. **return** -1; 659. } 661. **template**<**class** T> 662. **void** chainlist<T>::inverse() 663. { 664. list\_node<T>\* temp1 = **this**->head; 665. list\_node<T>\* temp2 = nullptr; 666. list\_node<T>\* temp3 = nullptr; 667. **for** (**int** i = 0; i < size; i++) 668. { 669. temp2 = temp1->next; 670. temp3 = temp1->front; 671. temp1->front = temp1->next; 672. temp1->next = temp3; 673. temp1 = temp2; 674. } 675. temp1 = **this**->head; 676. **this**->head = **this**->back; 677. **this**->back = temp1; 678. } 680. **template**<**class** T> 681. chainlist<T>::chainlist() 682. { 683. **this**->head = nullptr; 684. **this**->back = nullptr; 685. **this**->size = 0; 686. } 688. **template**<**class** T> 689. chainlist<T>::chainlist(**const** chainlist<T>& chainlist) 690. { 691. **this**->head = nullptr; 692. **this**->back = nullptr; 693. **this**->size = 0; 694. list\_node<T>\* temp = chainlist.head; 695. **while** (temp != nullptr) 696. { 697. push\_back(temp); 698. temp = temp->next; 699. } 700. } 702. **template**<**class** T> 703. **void** chainlist<T>::print() 704. { 705. list\_node<T>\* temp = **this**->head; 706. **while** (temp != nullptr) 707. { 708. cout << temp->getData() << " "; 709. temp = temp->next; 710. } 711. cout << endl; 712. } 714. **template**<**class** T> 715. list\_node<T>\* chainlist<T>::getNode(**int** index) 716. { 717. list\_node<T>\* temp = **this**->head; 718. **for** (**int** i = 0; i < index; i++) 719. { 720. temp = temp->next; 721. } 722. **return** temp; 723. }  726. **template**<**class** T> 727. **void** chainlist<T>::self\_bubble\_sort() 728. { 729. **for** (**size\_t** i = 0; i < size; i++) 730. { 731. **for** (**size\_t** j = 0; j < size - i - 1; j++) 732. { 733. **if** (getNode(j)->getData() > getNode(j + 1)->getData()) 734. { 735. my\_swap(getNode(j)->getData(), getNode(j + 1)->getData()); 736. } 737. } 739. } 741. } 743. **template**<**class** T> 744. **void** quick\_sort(my\_iterator<T> head, my\_iterator<T> tail) 745. { 746. **if** (head == tail||tail<head) { **return**; } 747. T pivot = head.getCurrent()->getData(); 748. my\_iterator<T> i = head; 749. my\_iterator<T> j = tail; 750. **while** (i < j) 751. { 752. **while** (j.getCurrent()->getData() >= pivot && i < j) 753. { 754. j--; 755. } 756. **if** (i<j) 757. { 758. i.getCurrent()->getData() = j.getCurrent()->getData(); 759. } 761. **while** (i.getCurrent()->getData() <= pivot && i < j) 762. { 763. i++; 764. } 765. **if** (i<j) 766. { 767. j.getCurrent()->getData() = i.getCurrent()->getData(); 768. } 769. **if** (i==j) 770. { 771. i.getCurrent()->getData() = pivot; 772. } 773. } 774. my\_iterator<T> j2 = j; 775. j2++; 776. quick\_sort(head, j); 777. quick\_sort(j2, tail); 778. } 780. **template**<**class** T> 781. **void** chainlist<T>::self\_quick\_sort() 782. { 783. my\_iterator<T> head(**this**->head); 784. my\_iterator<T> tail(**this**->back, **this**->size - 1); 785. quick\_sort(head, tail); 786. } 788. **class** Solution { 790. **public**: 791. **void** solute(); 792. **void** test(); 793. }; 795. **void** Solution::solute() 796. { 797. chainlist<**int**> list1; 798. chainlist<**int**> list2; 799. **int** a, b; 800. cin >> a >> b; 801. **for** (**size\_t** i = 0; i < a; i++) 802. { 803. **int** value; 804. cin >> value; 805. list1.push\_back(value); 806. } 807. **for** (**size\_t** i = 0; i < b; i++) 808. { 809. **int** value; 810. cin >> value; 811. list2.push\_back(value); 812. } 813. chainlist<**int**> list3; 814. my\_iterator<**int**> it2(list2.getHead()); 815. my\_iterator<**int**> it1(list1.getHead()); 816. my\_iterator<**int**> it3(list3.getHead()); 817. **while** (it1.getCurrent() != nullptr) 818. { 819. list3.push\_back(it1.getCurrent()->getData()); 820. it1++; 821. } 822. **while** (it2.getCurrent() != nullptr) 823. { 824. list3.push\_back(it2.getCurrent()->getData()); 825. it2++; 826. } 827. list1.self\_quick\_sort(); 828. list2.self\_quick\_sort(); 829. list3.self\_quick\_sort(); 830. it1.reset(list1.getHead()); 831. it2.reset(list2.getHead()); 832. it3.reset(list3.getHead()); 833. **int** ans1 = 0, ans2 = 0, ans3 = 0; 834. **while** (it1.getCurrent() != nullptr) 835. { 836. ans1 += it1.getCurrent()->getData() ^ it1.getIndex(); 837. it1++; 838. } 839. **while** (it2.getCurrent() != nullptr) 840. { 841. ans2 += it2.getCurrent()->getData() ^ it2.getIndex(); 842. it2++; 843. } 844. **while** (it3.getCurrent() != nullptr) 845. { 846. ans3 += it3.getCurrent()->getData() ^ it3.getIndex(); 847. it3++; 848. } 849. cout << ans1 << endl << ans2 << endl << ans3; 850. } 852. **void** Solution::test() 853. { 854. chainlist<**int**> list1; 855. **int** n; 856. cin >> n; 857. **for** (**size\_t** i = 0; i < n; i++) 858. { 859. **int** value; 860. cin >> value; 861. list1.push\_back(value); 862. } 863. list1.print(); 864. my\_iterator<**int**> it1(list1.getHead()); 865. my\_iterator<**int**> it2(list1.getBack(), list1.getSize() - 1); 867. quick\_sort(it1, it2); 868. list1.print(); 869. } 871. **int** main() 872. { 874. Solution solution; 876. solution.solute(); 878. **return** 0; 880. } | | | |