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：  掌握稀疏矩阵结构的描述及操作的实现。 | | | |
| 软件开发环境：  CLION2020 | | | |
| 1. 实验内容   1、题目描述：  创建稀疏矩阵类（参照课本MatrixTerm三元组定义） ,采用行主顺序把稀疏矩阵非0元素映射到一维数组中，实现操作：两个稀疏矩阵相加、两个稀疏矩阵相乘、稀疏矩阵的转置、输出矩阵。  重置矩阵：操作1，即重置矩阵 P 的尺寸为 n 行 m 列,且随后按行优先顺序输入矩阵 P 的各个元素。  矩阵乘法：操作2，t 行非零元素已按行优先顺序给出，矩阵中非零元素的表示为 x y v，其中 x 表示行序号，y 表示列序号，v 表示非零元素值，行列序号从 1 开始。设输入的矩阵为Q，若PxQ运算合法,则将PxQ的结果矩阵赋给 P,若不合法,则将Q赋给P，同时输出-1。  矩阵加法：操作3，t 行非零元素已按行优先顺序给出，矩阵中非零元素的表示为 x y v，其中 x 表示行序号，y 表示列序号，v 表示非零元素值，行列序号从 1 开始。设输入的矩阵为 Q,若 P+Q 运算合法,则将 P+Q 的结果矩阵赋给 P,若不合法,则将 Q 赋给 P,同时输出 -1。  输出操作：操作4，设当前矩阵 P 的尺寸为 n 行 m 列,第一行输出矩阵 P 的行数和列数，随后 n 行按行优先顺序输出矩阵 P,每行 m 个数字,来表示当前的矩阵内容，每行数字之间用空格分隔。  转置操作：操作5，设当前矩阵 P 的尺寸为 n 行 m 列，将其转置为 m 行 n 列的矩阵，无需输出。  输入输出格式：  输入：  第一行一个w代表操作个数，接下来若干行是各个操作，其中保证第一个操作一定为重置矩阵。  输出：  当执行操作4时，输出矩阵P；当执行操作2或3时，若对应运算不合法，则输出-1。   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. 本次实验用到的数据结构是稀疏矩阵，当矩阵中的0元素非常多的时候，我们使用稀疏矩阵来保存矩阵的信息。稀疏矩阵结构如下：      1. 转置操作: 对于题目要求的转置操作，我们需要把第一列的每一个非0元素转换为第一行的非0元素，即为原来在矩阵中的(i,j)非0元素，变成新的矩阵中的(j,i)的非0元素，且元素值保持不变。那么我们第一反应就是for循环搜索，其中第i次搜索，收集第i列的元素。但是这样时间复杂度就变成了O(cols\*n)，其中n是非0元素个数，显然时间复杂度过高了。于是我们又想到了，其实第一轮的搜索结束后，我们就知道了每个元素的列号，如果我们将它们的相关信息保存下来，是不是就可以降低时间复杂度了？于是想到了用数组保存每一列的非0元素在arrayList中的下标，这样在转换时候可以O(1)访问，非常快捷。于是在收集完下标之后，我们就可以O（n）时间复杂度的转换了。代码如下：      1. Add操作:首先要分成两种情况讨论，一个是可以相加，一个是不可以相加。可以相加的需要保存最后的结果并赋值给\*this，不可以相加的需要将新输入的矩阵赋值给\*this。对于可以相加的，我们结合归并排序的思想，将矩阵映射到一个一维数组，以到原点的距离大小关系为判断依据，哪个到原点的距离小我们就在result向量中Push进去，如果相等，那么需要判断一下相加是否为0，如果是，那么Push。如果不是，那么我们什么都不执行，最后更新私有变量。            1. Multi操作：对于两个稀疏矩阵相乘，其实类似转置操作，我们首先需要判断是否可以相乘(第一个矩阵的列数等于第二个矩阵的行数),如果可以，那么进行下一个操作。A矩阵的第i行向量乘以B矩阵的第j列，得到的为结果矩阵的第(i,j)个元素。同样的，我们需要记录一下第index列的非0元素在array数组中的第几个，然后进行相乘，最后求和。类似与归并排序              1. Output操作:对于output操作，我们只需要判断一下输出的是不是保存的array数组中的第一个元素就行，如果是那么直接输出迭代器指向的元素，如果不是，那么输出0。      1. Reset操作：对于reset操作我们只需要记录一下非0元素的信息即可。然后将非0元素组成的数组回写到array中去，并且记得更新私有变量。        1. 测试结果（测试输入，测试输出）   输入：    输出：    提交OJ最后的结果：     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. 值得注意的是，一定一定要记得更新私有成员，因为public接口的函数是没有记忆性的，只有靠着私有成员才能判断自身的状态。并且更新私有成员尽量在成员函数的末尾进行，以防止不必要的思维定式导致的bug，比如说第一次提交的时候，在multi函数中忘记更新不能相乘时候的私有变量的值，导致了错误。经过debug发现后才一次AC，就是下面这个图      1. 想要记录每一列的非0元素在原来array数组中的下标，那么我们需要开的数组大小是矩阵的列数+1，而不是列数，否则会RE 2. 学会了二维vecotr的创建，即为：vector保存的每一个元素都是一个vecotr，比如在记录每一列的非0元素的时候开的这么一个vector      1. 进行矩阵相乘的时候，我们考虑的应该全面。比如说，我们进入矩阵的时候，可能按照行优先保存的第一个的非0元素就是第4行的，那么这个时候我们就应该特判，如果不是我们当前正在进行的行，就continue。同时，我们还需要定位一下这一行的最后一个元素的位置，以达到O（1）判断是否到了终点。 2. 我们应该有防止短路的意识，尤其是while循环的时候，一定要判断是否到了边界。并且判断是否到了边界一定要放在第一位，如果放到了后面很容易短路(尤其是涉及到空指针的情况下).比如：     空指针造成错误的情况：    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add操作，我们需要特判一下相加的最后的值是否为0再push进去。 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. 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. #include <iostream> 3. #include <vector> 4. **using** **namespace** std; 6. **template**<**class** T> 8. **class** arrayList 9. { 10. **public**: 12. arrayList(**int** initialCapacity = 10); 13. arrayList(**const** arrayList<T>&); 14. ~arrayList() { **delete**[] element; } 16. T& operator[](**int** index); 17. // ADT methods 18. **int** size() **const** { **return** listSize; } 20. **class** iterator; 21. iterator begin() { **return** iterator(element); } 22. iterator end() { **return** iterator(element + listSize); }  25. **class** iterator 26. { 27. **public**: 29. // constructor 30. iterator(T\* thePosition = 0) { position = thePosition; } 32. // dereferencing operators 33. T& operator\*() **const** { **return** \*position; } 34. T\* operator->() **const** { **return** &\*position; }  37. // increment 38. iterator& operator++()   // preincrement 39. { 40. ++position; **return** \***this**; 41. } 42. iterator operator++(**int**) // postincrement 43. { 44. iterator old = \***this**; 45. ++position; 46. **return** old; 47. } 49. // equality testing 50. **bool** operator!=(**const** iterator right) **const** 51. { 52. **return** position != right.position; 53. } 54. **bool** operator==(**const** iterator right) **const** 55. { 56. **return** position == right.position; 57. } 58. **protected**: 59. T\* position; 60. };  // end of iterator class 62. **public**:  // additional members of arrayList 64. T\* element;            // 1D array to hold list elements 65. **int** arrayLength;       // capacity of the 1D array 66. **int** listSize;          // number of elements in list 67. }; 69. **template**<**class** T> 70. arrayList<T>::arrayList(**int** initialCapacity) 71. {// Constructor. 73. arrayLength = initialCapacity; 74. element = **new** T[arrayLength]; 75. listSize = 0; 76. } 78. **template**<**class** T> 79. arrayList<T>::arrayList(**const** arrayList<T>& theList) 80. {// Copy constructor. 81. arrayLength = theList.arrayLength; 82. listSize = theList.listSize; 83. element = **new** T[arrayLength]; 84. copy(theList.element, theList.element + listSize, element); 85. }  88. **template**<**class** T> 89. T& arrayList<T>::operator[](**int** index) 90. { 91. **return** element[index]; 92. }  95. **template**<**class** T> 96. **struct** MatrixTerm { 97. **public**: 98. MatrixTerm<T>(**int** i,**int** j,T val):row(i),col(j),val(val){} 99. **int** row; 100. **int** col; 101. T val; 102. MatrixTerm<T>(){} 103. }; 105. **template**<**class** T> 106. **class** sparseMatrix 107. { 108. **int** rows; 109. **int** cols; 110. arrayList<MatrixTerm<T> >array; 111. **public**: 112. sparseMatrix(); 113. **void** reset(); 114. **void** output(); 115. **void** add(); 116. **void** transpose(); 117. **void** multi(); 118. **void** change\_rows\_cols(**int** to\_rows,**int** to\_cols); 119. }; 121. **template**<**class** T> 122. **void** sparseMatrix<T>::reset() 123. { 124. **int** to\_rows; 125. **int** to\_cols; 126. cin >> to\_rows >> to\_cols; 127. vector<MatrixTerm<T> >temp; 128. **for** (**int** i = 1; i <= to\_rows; i++) 129. { 130. **for** (**int** j = 1; j <= to\_cols; j++) 131. { 132. T val; 133. cin >> val; 134. **if** (val != 0) 135. { 136. temp.push\_back(MatrixTerm<T>(i, j, val));//压进去 137. } 138. } 139. } 140. **delete** [] array.element; 141. array.listSize = temp.size(); 142. array.arrayLength = temp.size(); 143. array.element = **new** MatrixTerm<T>[temp.size()]; 144. **for** (**int** i = 0; i < temp.size(); i++) 145. {//一个一个读出来 146. array[i] = temp[i]; 147. } 148. rows = to\_rows; 149. cols = to\_cols; 150. } 152. // T是保存的数据类型 153. **template**<**class** T> 154. **void** sparseMatrix<T>::output() 155. { 156. cout << rows << " " << cols << endl; 157. **typename** arrayList<MatrixTerm<T> >::iterator iter = array.begin(); 158. **for** (**int** i = 1; i <= rows; i++) 159. { 160. **for** (**int** j = 1; j <= cols; j++) 161. { 162. T val = (\*iter).row == i && (\*iter).col == j ? (\*(iter++)).val : 0; 163. cout << val << " "; 164. } 165. cout << endl; 166. } 167. } 169. **template**<**class** T> 170. **void** sparseMatrix<T>::add() 171. { 172. **int** new\_rows; 173. **int** new\_cols; 174. **int** t;//非0元素个数 175. cin >> new\_rows >> new\_cols >> t; 177. vector<MatrixTerm<T> >temp;//记录输入的数据 178. **for** (**int** i = 0; i < t; i++) 179. { 180. **int** row, col, val; 181. cin >> row >> col >> val; 182. temp.push\_back(MatrixTerm<T>(row, col, val)); 183. } 184. **if** (new\_rows != rows || new\_cols != cols) 185. {//更新私有成员 186. **delete** []array.element; 187. array.element = **new** MatrixTerm<T>[t]; 188. array.arrayLength = t; 189. array.listSize = t; 190. **for** (**int** i = 0; i < t; i++) 191. { 192. array[i] = temp[i]; 193. } 194. cout << -1 << endl; 195. rows=new\_rows; 196. cols=new\_cols; 197. } 198. **else** 199. { 200. //归并 201. vector<MatrixTerm<T> >result; 202. auto array\_iter = array.begin(); 203. //vector<MatrixTerm<T> >temp;//记录输入的数据 204. auto temp\_iter = temp.begin(); 205. //类似归并排序 206. **while** (array\_iter != array.end() && temp\_iter != temp.end()) 207. { 208. **int** array\_index = ((\*array\_iter).row - 1) \* cols + (\*array\_iter).col - 1;//记录到原点的距离 209. **int** temp\_index = ((\*temp\_iter).row - 1) \* new\_cols + (\*temp\_iter).col - 1; 210. **if** (array\_index < temp\_index) 211. { 212. result.push\_back(MatrixTerm<T>(array\_iter->row, array\_iter->col, array\_iter->val)); 213. array\_iter++; 214. } 215. **else** **if** (array\_index == temp\_index) 216. { 217. **if** (array\_iter->val + (\*temp\_iter).val != 0)//需要特判相加是否等于0 218. { 219. result.push\_back(MatrixTerm<T>(array\_iter->row, array\_iter->col, array\_iter->val + (\*temp\_iter).val)); 220. } 221. array\_iter++; 222. temp\_iter++; 223. } 224. **else** 225. { 226. result.push\_back(MatrixTerm<T>((\*temp\_iter).row, (\*temp\_iter).col, (\*temp\_iter).val)); 227. temp\_iter++; 228. } 229. } 230. **while** (array\_iter != array.end()) 231. { 232. result.push\_back(MatrixTerm<T>(array\_iter->row, array\_iter->col, array\_iter->val)); 233. array\_iter++; 234. } 236. **while** (temp\_iter != temp.end()) 237. { 238. result.push\_back(MatrixTerm<T>((\*temp\_iter).row, (\*temp\_iter).col, (\*temp\_iter).val)); 239. temp\_iter++; 240. } 242. //回写,并且更新私有变量 243. **delete** [] array.element; 244. array.listSize=result.size(); 245. array.arrayLength=result.size(); 246. array.element=**new** MatrixTerm<T>[result.size()]; 247. **for** (**int** i = 0; i < result.size(); i++) 248. { 249. array[i] = result[i]; 250. } 252. } 253. } 255. **template**<**class** T> 256. **void** sparseMatrix<T>::transpose() 257. { 258. vector<vector<**int**>>next(cols + 1);//如果到cols那么需要多开一个 259. **for** (**int** i = 0; i < array.size(); i++) 260. { 261. next[array[i].col].push\_back(i);//next[i]保存的是第i列的每一个元素在array数组中的id 262. } 263. //防止被破坏,先收集到vector里边 264. vector<MatrixTerm<T>>result; 265. **for** (**int** i = 1; i <= cols; i++)//收集vector并且回写 266. {//next[i][j]存放的是原来的矩阵第i列的非0元素在array中的id 267. **for** (**int** j = 0; j < next[i].size(); j++) 268. { 269. result.push\_back(MatrixTerm<T>(i,array[next[i][j]].row,array[next[i][j]].val)); 270. } 271. } 272. **for** (**int** i = 0; i < result.size(); i++) 273. { 274. array[i] = result[i]; 275. } 276. std::swap(rows,cols); 277. } 279. **template**<**class** T> 280. **void** sparseMatrix<T>::multi() 281. { 282. //前期准备 283. **int** new\_rows; 284. **int** new\_cols; 285. **int** t;//新矩阵非0元素的个数 286. cin >> new\_rows >> new\_cols >> t; 287. vector<MatrixTerm<T>>temp;//新的矩阵 288. **for** (**int** i = 0; i < t; i++) 289. { 290. **int** row, col, val; 291. cin >> row >> col >> val; 292. temp.push\_back(MatrixTerm<T>(row, col, val)); 293. } 294. **if** (cols != new\_rows)//不能相乘 295. { 296. **delete**[]array.element; 297. array.listSize = t; 298. array.arrayLength = t; 299. array.element = **new** MatrixTerm<T>[t]; 300. **for** (**int** i = 0; i < t; i++) 301. { 302. array[i] = temp[i]; 303. } 304. rows=new\_rows; 305. cols=new\_cols; 306. cout << -1 << endl; 308. } 309. **else** 310. { 311. //temp是新的矩阵 312. vector<vector<**int**>>next(new\_cols + 1); 313. vector<MatrixTerm<**int**>>result; 314. **for** (**int** i = 0; i < t; i++) 315. { 316. next[temp[i].col].push\_back(i);//按照行优先的顺序输入了id 317. } 318. **int** first = 0; 319. **int** last = 0; 320. **for** (**int** i = 1; i <= rows; i++)//遍历每一行 321. { 322. //定位原矩阵的首尾 323. //i是想要计算的行 324. **if** (array[first].row > i)//先判断是否存在第i行  这里有问题 325. { 326. **continue**; 327. } 328. //一定存在第i行 329. **while** (last<array.size() &&array[last].row == i)//防止短路 330. { 331. last++; 332. } 333. last = last - 1;//回到尾节点 334. **for**(**int** j=1;j<=new\_cols;j++) 335. { 336. //类似归并 337. **int** ori\_pos = first; 338. **int** new\_pos = 0;//next[i]下标 339. //矩阵乘法 340. **int** ans=0; 341. **while** (ori\_pos <= last && new\_pos < next[j].size())//有问题 342. { 343. **if** (array[ori\_pos].col < temp[next[j][new\_pos]].row) 344. { 345. ori\_pos++; 346. } 347. **else** **if** (array[ori\_pos].col == temp[next[j][new\_pos]].row) 348. { 349. ans+= array[ori\_pos].val \* temp[next[j][new\_pos]].val; 350. ori\_pos++; 351. new\_pos++; 352. } 353. **else** 354. { 355. new\_pos++; 356. } 357. } 358. **if**(ans!=0) 359. { 360. result.push\_back(MatrixTerm<**int**>(i,j,ans)); 361. } 362. } 363. last++; 364. first = last; 365. } 366. array.arrayLength=result.size(); 367. array.listSize=result.size(); 368. **delete** []array.element; 369. array.element=**new** MatrixTerm<T>[result.size()]; 370. **this**->change\_rows\_cols(rows,new\_cols); 371. **for**(**int** i=0;i<result.size();i++) 372. { 373. array[i]=result[i]; 374. } 375. } 376. } 378. **template**<**class** T> 379. sparseMatrix<T>::sparseMatrix() {} 381. **template**<**class** T> 382. **void** sparseMatrix<T>::change\_rows\_cols(**int** to\_rows, **int** to\_cols) 383. { 384. rows=to\_rows; 385. cols=to\_cols; 386. }  389. **int** main() 390. { 391. **int** w; 392. cin.tie(0); 393. cout.tie(0); 394. cin >> w; 395. sparseMatrix<**int**>matrix; 396. **for** (**int** i = 0; i < w; i++) 397. { 398. **int** flag; 399. cin >> flag; 400. **switch** (flag) { 401. **case** 1: 402. matrix.reset(); 403. **break**; 404. **case** 2: 405. matrix.multi(); 406. **break**; 407. **case** 3: 408. matrix.add(); 409. **break**; 410. **case** 4: 411. matrix.output(); 412. **break**; 413. **case** 5: 414. matrix.transpose(); 415. **break**; 416. } 417. } 418. **return** 0; 419. } | | | |