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：散列表 | | | |
| 实验学时：2 | | 实验日期： 2021-11-25 | |
| 实验目的：  1、掌握队列结构的定义与实现；  2、掌握队列结构的使用。 | | | |
| 软件开发环境：  CLION2020 | | | |
| 1. 实验内容 2. 题目描述：   给定散列函数的除数D和操作数m，输出每次操作后的状态。  有以下三种操作：  插入x，若散列表已存在x，输出“Existed”，否则插入x到散列表中，输出所在的下标。  查询x，若散列表不含有x，输出“-1”，否则输出x对应下标。  删除x，若散列表不含有x，输出“Not Found”，否则输出删除x过程中移动元素的个数。  输入输出格式：  输入：  第一行两个整数D，m。分别代表散列函数的除数D和操作数m。  接下来m行，每行两个整数opt和x，分别代表操作类型和操作数。  若opt为0，代表插入x；  若opt为1，代表查询x；  若opt为2，代表删除x。  输出：  按需输出。   1. 题目描述：   给定散列函数的除数D和操作数m，输出每次操作后的状态。  有以下三种操作：  插入x，若散列表已存在x，输出"Existed"；  查询x，若散列表不含有x，输出"Not Found"，否则输出x所在的链表长度；  删除x，若散列表不含有x，输出"Delete Failed"，否则输出x所在链表删除x后的长度；  输入输出格式：  输入：  第一行两个整数D(1<=D<=3000)和m(1<=m<=3000)，其中D为散列函数的除数，m为操作数。  接下来的m行，每行两个整数opt和x，分别代表操作类型和操作数。  若opt为0，则代表向散列表中插入x；  若opt为1，代表查询散列表中x是否存在；  若opt为2，(如果散列表中含有x)，删除x。  输出：  按需输出。   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. 数据结构：使用散列表。我们知道散列表的核心就是将一个数据范围比较大的一个序列映射到一个比较小的区间上，（也就是离散化的思想），常用的方法有取模等。同时实现散列表的方式也有很多种，常见的比如线性开型寻址和链表寻址（也有的用静态链表来模拟） 3. 算法：对于第1题 4. Erase操作：   首先，对于删除操作，我们需要先知道散列表中是否含有该元素。因此我们使用find函数进行搜索，如果返回的不是-1，则存在，那么我们接下来进行删除操作，否则输出“Not Found”。对于接下来的删除步骤，我们的思路与原则就是，让从被删除的位置开始，到其后最近的空位置为止，每一个元素都尽可能的靠近其哈希值的位置。因此可以移动的情况就会分成三种，如下图所示：    此时自然可以将新值向前移动，靠近了哈希位置，方便了我们之后的搜寻。    我们思考，为什么哈希位置会在读取到的新值前面呢？因为哈希位置和它后面的位置都被填满了，所以使用循环队列往前面插。    还有这种情况，也是更靠近了哈希位置。对于以上的三种情况，我们均需要将新值移动，经过验证，其余的三种情况不可以移动。代码如下：       1. Insert操作：题目要求     我们使用search函数，如果返回的是一个空指针的位置，那么说明散列表中无该值。如果返回的index的值等于输入，则说明已经找到。值得注意的是，我们不能使用find函数，因为find函数对search函数进行了简化，它返回的是是否找到了该输入，而没有返回是否还有空余地方的情况。代码如下：     1. Search函数：对于每一个输入，经过哈希后，映射到一个地址，返回等于输入的值的位置或者一个nullptr的位置。值得注意的是，可能存在这么一种情况，散列表中存在的每一个值都不等于输入，且表满，这个时候我们要特判一下，循环次数不能等于自身的size。代码如下：      1. Find函数：检查search的返回值，如果返回的地址的值是我们的输入，那么将该索引返回。否则返回-1 2. 使用链表维护的散列表算法：首先将链表进行封装，对于每一个输入，我们都将其映射到相应的链表中存储，对于每一个操作，让链表来实现。   链表的erase操作：首先我们检查一下链表中是否含有该元素，如果有，那么删除。否则输出"Delete Failed"  代码如下：    链表实现的find：交给链表来实现，同时通过返回的标记判断是否成功。代码如下：     1. 测试结果（测试输入，测试输出） 2. a题   输入：    (1)b题  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. 对于hashTable维护的散列表，需要使用指针来维护，因为对于空的情况，我们无论使用哪个整数来标记都不合适，因为我们完全有可能插入这个值。     (2 )Search操作：对于每一个输入，经过哈希后，映射到一个地址，返回等于输入的值的位置或者一个nullptr的位置。值得注意的是，可能存在这么一种情况，散列表中存在的每一个值都不等于输入，且表满，这个时候我们要特判一下，循环次数不能等于自身的size。代码如下：     1. 我们在写循环条件判断的时候，对于短路情况的判断一定要慎重，我们对于指针的使用一定要先判断是否为空，在进行取值操作，如下：      1. 对于情况的考虑我们应该全面，比如说这个，循环队列很有可能theBack在队头，front在队尾，那么我们加上一个arrayLength再取模就行了，在结合点和线的映射关系，就可以计算出来      1. 对于维护私有变量的时候，要特殊情况特殊判：比如说删除结点的时候，要考虑这个是不是头结点，如果是，那么更新私有变量。如下：      1. 我们在使用函数的时候，应该充分的考虑异常情况，比如说调用top函数的时候或者Pop的时候，都应该考虑一下是否为空，如果是空，那么就抛出异常      1. 要注意私有成员的更新，public函数知道自己所处的状态都是靠着私有成员才知道的，如果没有及时更新，那数据就成了垃圾数据，没有任何意义。我在写实验的时候，也经历过没更新导致的Bug，最终debug查出来，就是下面这个：      1. Erase: 首先，对于删除操作，我们需要先知道散列表中是否含有该元素。因此我们使用find函数进行搜索，如果返回的不是-1，则存在，那么我们接下来进行删除操作，否则输出“Not Found”。对于接下来的删除步骤，我们的思路与原则就是，让从被删除的位置开始，到其后最近的空位置为止，每一个元素都尽可能的靠近其哈希值的位置。因此可以移动的情况就会分成三种.相对应的代码如下：        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**   1. #include <iostream> 3. **using** **namespace** std; 5. **template**<**class** K> 6. **class** hashTable 7. { 8. **public**: 9. hashTable(**int** theDivisor ); 10. ~hashTable(){**delete** [] table;}//析构 11. **bool** empty() **const** {**return** dSize == 0;} 12. **int** size() **const** {**return** dSize;} 13. **int** find(**const** K&) **const**;//查找 14. **void** insert(**const** K& val);//插入 15. **void** erase(**const** K&val);//删除 16. **protected**: 17. **int** hash(**const** **int** &index)**const**; 18. **int** search(**const** K&) **const**; 19. K\*\* table;//必须要用指针来处理 20. **int** dSize;                 // number of pairs in dictionary 21. **int** divisor;               // hash function divisor 22. }; 24. **template**<**class** K> 25. hashTable<K>::hashTable(**int** theDivisor) 26. { 27. divisor = theDivisor; 28. dSize = 0; 29. table=**new** K\*[theDivisor]();//初始化赋值为nullptr 30. } 32. **template**<**class** K> 33. **int** hashTable<K>::search(**const** K& theKey) **const** 34. { 35. **int** i = hash(theKey); 36. **int** j = i; 37. **do** 38. { 39. **if** (table[j] == nullptr || \*table[j] == theKey) 40. **return** j; 41. j = (j + 1) % divisor; 42. } **while** (j != i); 44. **return** j;  // 表满了 45. } 47. **template**<**class** K> 48. **int** hashTable<K>::find(**const** K& theKey) **const** 49. { 50. **int** b = search(theKey); 52. **if** (table[b] == nullptr || \*table[b]!= theKey) 53. **return** -1;           // no match 54. **return** b;//返回了数组的下标 55. } 57. **template**<**class** K> 58. **void** hashTable<K>::insert(**const** K& val) 59. { 60. **int** b=search(val);//不能使用find，因为find不会返回空指针的信息 61. **if**(table[b]== nullptr) 62. { 63. table[b]=**new** K(val); 64. dSize++; 65. cout<<b<<endl; 66. } 67. **else** **if**(\*table[b]==val)//找到了 68. { 69. cout<<"Existed"<<endl; 70. } 71. } 73. **template**<**class** K> 74. **void** hashTable<K>::erase(**const** K &val) 75. { 76. **int** loc=find(val); 77. **if**(loc==-1)//没找到 78. { 79. cout<<"Not Found"<<endl; 80. } 81. **else**//找到了 82. {//有问题 83. //直到碰到了空或者又回到了该位置 84. table[loc]= nullptr; 85. **int** pos= (loc + 1) % divisor; 86. **int** cnt=0; 87. **int** loc\_of\_nullptr=loc;//记录空指针的位置 88. **do**{ 89. **if**(!table[pos]) 90. { 91. **break**; 92. } 93. //pos==10 loc\_of\_nullptr==9 hash==9 94. **if**((loc\_of\_nullptr < pos &&pos< hash(\*table[pos])) || 95. (pos < hash(\*table[pos])&&hash(\*table[pos]) <= loc\_of\_nullptr )|| 96. (hash(\*table[pos]) <= loc\_of\_nullptr&& loc\_of\_nullptr< pos) ) 97. { 98. table[loc\_of\_nullptr]=table[pos]; 99. table[pos]= nullptr; 100. loc\_of\_nullptr=pos; 101. cnt++; 102. } 103. pos= (pos + 1) % divisor; 104. } **while** (pos != loc); 105. cout<<cnt<<endl; 106. } 107. } 109. **template**<**class** K> 110. **int** hashTable<K>::hash(**const** **int** &index) **const** 111. { 112. **return** index%divisor; 113. }  116. **int** main() 117. { 118. **int** D,m; 119. cin>>D>>m; 120. hashTable<**int**>hash(D); 121. **for**(**int** i=0;i<m;i++) 122. { 123. **int** flag,x; 124. cin>>flag>>x; 125. /\*  if(i==25) 126. { 127. cout<<"test"<<endl; 128. }\*/ 129. **switch** (flag) { 130. **case** 0: 131. hash.insert(x); 132. **break**; 133. **case** 1: 134. cout<<hash.find(x)<<endl; 135. **break**; 136. **case** 2: 137. hash.erase(x); 138. **break**; 139. **default**://异常处理 140. **throw** "please check your input"; 141. } 143. } 144. **return** 0; 145. }   (2)B题   1. #include<iostream> 2. **using** **namespace** std;  5. //链表结构 6. **template**<**class** K, **class** E> 7. **class** sortedChain 8. { 9. **public**: 10. sortedChain() { firstNode = NULL; dSize = 0; } 11. ~sortedChain();//析构 13. **bool** empty() **const** { **return** dSize == 0; } 14. **int** size() **const** { **return** dSize; } 15. pair<**const** K, E>\* find(**const** K&) **const**;//查找 16. **void** erase(**const** K&);//删除 17. **void** insert(**const** pair<**const** K, E>&);//插入  20. **protected**: 21. pairNode<K, E>\* firstNode;  // 头结点 22. **int** dSize;                 // size 23. }; 25. **template**<**class** K, **class** E> 26. sortedChain<K, E>::~sortedChain()//析构 27. { 28. **while** (firstNode != NULL) 29. {// delete firstNode 30. pairNode<K, E>\* nextNode = firstNode->next; 31. **delete** firstNode; 32. firstNode = nextNode; 33. } 34. } 36. **template**<**class** K, **class** E> 37. pair<**const** K, E>\* sortedChain<K, E>::find(**const** K& theKey) **const**//查找操作 38. { 39. pairNode<K, E>\* currentNode = firstNode; 41. **while** (currentNode != NULL &&//值不等于输入，且不越界，对于NULL值的判断要放到前边 42. currentNode->element.first != theKey) 43. currentNode = currentNode->next; 45. **if** (currentNode != NULL && currentNode->element.first == theKey) 46. // yes, found match 47. **return** ¤tNode->element; 49. // no match 50. **return** NULL; 51. } 53. **template**<**class** K, **class** E> 54. **void** sortedChain<K, E>::insert(**const** pair<**const** K, E>& thePair) 55. {// Insert thePair into the dictionary. Overwrite existing 56. // pair, if any, with same key. 57. pairNode<K, E>\* p = firstNode, 58. \* tp = NULL; // tp trails p 60. // move tp so that thePair can be inserted after tp 61. **while** (p != NULL && p->element.first < thePair.first) 62. { 63. tp = p; 64. p = p->next; 65. } 67. // check if there is a matching pair 68. **if** (p != NULL && p->element.first == thePair.first) 69. {// replace old value 70. p->element.second = thePair.second; 71. **return**; 72. } 74. // no match, set up node for thePair 75. pairNode<K, E>\* newNode = **new** pairNode<K, E>(thePair, p); 77. // insert newNode just after tp 78. **if** (tp == NULL) firstNode = newNode; 79. **else** tp->next = newNode; 81. dSize++; 82. **return**; 83. } 85. **template**<**class** K, **class** E> 86. **void** sortedChain<K, E>::erase(**const** K& theKey) 87. {// Delete the pair, if any, whose key equals theKey. 88. pairNode<K, E>\* p = firstNode, 89. \* tp = NULL; // tp trails p 91. // search for match with theKey 92. **while** (p != NULL && p->element.first < theKey) 93. { 94. tp = p; 95. p = p->next; 96. } 98. // verify match 99. **if** (p != NULL && p->element.first == theKey) 100. { 101. **if** (tp == NULL) firstNode = p->next;//头结点特殊处理 102. **else** tp->next = p->next; 103. **delete** p;//删除 104. dSize--; 105. } 106. }       114. **template**<**class** K, **class** E> 115. **class** hashChains 116. { 117. **public**: 118. hashChains(**int** theDivisor = 11) 119. { 120. divisor = theDivisor; 121. dSize = 0; 122. table = **new** sortedChain<K, E>[divisor]; 123. } 125. ~hashChains() { **delete**[] table; } 127. **bool** empty() **const** { **return** dSize == 0; } 128. **int** size() **const** { **return** dSize; } 129. **int** hash(**const** K& val) { **return** val % divisor; } 130. **int** find(**const** K& theKey) 131. { 132. **if** (!table[hash(theKey)].find(theKey))//没找到 133. { 134. cout << "Not Found" << endl; 135. **return** 0; 136. } 137. **else** 138. { 139. cout << table[hash(theKey)].size()<<endl;//输出当前的size 140. **return** 1;//返回成功找到的标记 141. } 142. } 144. **void** insert(**const** K&val) 145. { 146. pair<K, E> thePair(val, val); 147. **int** homeBucket = (**int**)hash(thePair.first) % divisor; 148. **int** homeSize = table[homeBucket].size(); 149. table[homeBucket].insert(thePair); 150. **if** (table[homeBucket].size() == homeSize)//如果没有更新，那就说明了已经存在 151. { 152. cout << "Existed" << endl; 153. } 154. } 156. **void** erase(**const** K& theKey)//删除操作 157. { 158. **if** (table[hash(theKey)].find(theKey))//找到了 159. { 160. table[hash(theKey) % divisor].erase(theKey);//进行删除 161. cout << table[hash(theKey)].size() << endl;//输出size 162. } 163. **else** 164. { 165. cout << "Delete Failed" << endl; 166. } 167. }    172. **protected**: 173. sortedChain<K, E>\* table;  // hash table 174. **int** dSize;                 // number of elements in list 175. **int** divisor;               // hash function divisor 176. };    181. **int** main() 182. { 183. **int** D, m; 184. cin >> D >> m; 185. hashChains<**int**, **int**>hash(D); 186. **for** (**int** i = 0; i < m; i++) 187. { 188. **int** opt, x; 189. cin >> opt >> x; 190. **switch** (opt) 191. { 192. **case** 0: 193. hash.insert(x); 194. **break**; 195. **case** 1: 196. hash.find(x); 197. **break**; 198. **case** 2: 199. hash.erase(x); 200. **break**; 201. **default**://异常处理 202. **throw** "please check your input"; 203. } 204. } 205. **return** 0; 206. } | | | |