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：搜索树 | | | |
| 实验学时：2 | | 实验日期： 2021-12-9 | |
| 实验目的：  掌握二叉搜索树结构的定义、描述方法、操作实现。 | | | |
| 软件开发环境：  CLION2020 | | | |
| 1. 实验内容   1、题目描述：  创建带索引的二叉搜索树类。存储结构使用链表，提供操作:插入、删除、按名次删除、查找、按名次查找、升序输出所有元素。  输入输出格式：  输入：  输入第一行一个数字m (m<=1000000)，表示有m个操作。  接下来m行，每一行有两个数字a，b；  当输入的第一个数字 a 为 0 时，输入的第二个数字 b 表示向搜索树中插入 b；  当输入的第一个数字 a 为 1 时，输入的第二个数字 b 表示向搜索树中查找 b；  当输入的第一个数字 a 为 2 时，输入的第二个数字 b 表示向搜索树中删除 b；  当输入的第一个数字 a 为 3 时，输入的第二个数字 b 表示查找搜索树中名次为 b 的元素；  当输入的第一个数字 a 为 4 时，输入的第二个数字 b 表示删除搜索树中名次为 b 的元素；  输出：  对于输入中的每一种操作，输出执行操作的过程中依次比较的元素值的异或值。   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：选择二叉搜索树。二叉搜索树的特征：任取一个结点，如果存在左孩子，则左孩子的值一定小于等于该结点。如果存在右孩子，那么右孩子的值一定大于等于该结点。如下图      1. 算法： 2. 查找操作：类似于二分查找。其实二叉搜索树本质上也就是二分查找。我们将要查找的值和当前结点的值进行比较，如果相等，那么成功找到。如果不等，向左子树或者右子树继续查找。（这取决于大小关系），如果最后发现当前指向的结点是空，那么就查找失败。代码如下：      1. 插入操作：首先，我们需要判断一下，要插入的值是否在这棵树之中。如果在，那么我们就不插入了。如果不在，那么我们需要找到插入的位置。我们需要记录一下其父亲结点的位置，然后用一个指针去绑定他们俩。最后通过一下比较大小关系，决定是放在左子树还是右子树上。同时最后，需要更新一下index的值。代码如下：        1. 删除操作：对于删除操作，我们首先需要判断一下这棵树中是否有这个元素。如果没有，则返回。否则，我们需要删除这个元素，并且重新组织这棵树的结构。我们需要记录一下其父亲结点，然后分别考虑要删除的结点有几个孩子，如果有0个，也就是一个叶子结点，那么我们直接删除就可以了，将父亲结点的孩子指向了空。如果有1个那么我们直接让其代替被删除的结点在这棵树中的位置，并且更新一下index即可。如果有两个，我们选择右子树中最小的元素/左子树中最大的元素来替换其位置即可。代码如下：          1. 测试结果（测试输入，测试输出） 2. a题   输入：    输出：    提交OJ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. 在执行完函数体的时候，要更新Index，因为我们删除了一个元素，需要更新它所在子树的index值，把大于它的结点的index--。代码如下：      1. 我们在使用指针的时候，一定要防止短路。比如这段，一定要先判断其是否为空，再判断是否等于index，否则会导致访问错误而RE      1. 我们在写完类内函数的时候，要考虑全面，比如本实验中的孩子结点的判断，需要分成3种情况：0个、1个、2个，对于0个的我们直接删除，父节点孩子指针指向空，对于1个，我们直接代替。对于2个，我们寻找左子树中的最大元素/右子树中的最小元素来代替。同时需要注意的是，寻找最大/最小元素的时候，对于孩子指针的判断，需要特判一下左子树还是右子树，因为执行次数是不是1会导致结果的不同。 2. 在提交oj的时候要把自己的测试删除，比如多输出了一个换行可能就会导致全部的判错。 3. 对于BOOL数组的初始化，绝对不能想当然，每一次的定义都要对其进行初始化。在本次的实验中，因为bool数组没有初始化导致debug了很久。      1. 对于下标从1开始的数组，要多动态分配一块内存。因为索引为0的地方我们是没有访问的。      1. 对于一个只有两个私有成员的struct，我们可以直接使用pair来存储。简化了我们的代码量。同时对于返回多个值的函数，我们只能使用引用来返回。这也是为什么兴起了Go语言的一个原因。 2. 对于height等操作的计算，一定要特判是否合法。因为对于root结点来说，它的父节点是自身，不能直接增加。      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：     (10)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. 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. template <class T> 3. class binarySearchTree //二叉搜索树 4. { 5. private: 6. binaryTreeNode<T>\* root;  *//根* 7. int treeSize;  *//树中元素个数* 9. public: 10. binarySearchTree() 11. { 12. root = NULL; 13. treeSize = 0; 14. } 15. int find(const T& theElement); 16. int get(int theIndex); 17. int insert(const T& theElement); 18. int erase(const T& theElement); 19. int deleteByIndex(int theIndex); 21. }; 22. template <class T> 23. int binarySearchTree<T>::find(const T& theElement) 24. {*//根据元素值去查找节点* 25. binaryTreeNode<T>\* p = root; 26. int answer = 0; 27. while (p != NULL && p->element != theElement) 28. {*//检察元素p->element* 29. answer ^= p->element; 30. if (p->element > theElement) 31. p = p->leftChild; 32. else if (p->element < theElement) 33. p = p->rightChild; 34. } 35. if (p == NULL) 36. return 0; 37. else 38. {*//找到匹配的元素* 39. answer ^= p->element; 40. return answer; 41. } 42. } 44. template <class T> 45. int binarySearchTree<T>::get(int theIndex) 46. {*//根据索引去查找节点* 47. binaryTreeNode<T>\* p = root; 48. int answer = 0; 49. while (p != NULL && p->leftSize != theIndex) 50. { *//检察p->leftSize* 51. answer ^= p->element; 52. if (p->leftSize > theIndex) *//index小，指向左子树* 53. p = p->leftChild; 54. else if (p->leftSize < theIndex) *//index大，指向右子树* 55. { 56. theIndex = theIndex - (p->leftSize + 1);  *//减去leftSize+1，书P348* 57. p = p->rightChild; 58. } 59. } 60. if (p == NULL) 61. return 0; 62. else 63. {*//找到匹配的元素* 64. answer ^= p->element; 65. return answer; 66. } 67. } 68. template <class T> 69. int binarySearchTree<T>::insert(const T& theElement) 70. {*//插入元素值为theElement的元素* 71. binaryTreeNode<T>\* p = root; 72. binaryTreeNode<T>\* pp = NULL; *//用于记录父节点* 73. int answer = 0; 74. while (p != NULL) 75. {*//检察元素p->element* 76. answer ^= p->element; 77. pp = p;*//p移到它的一个孩子节点* 78. if (p->element < theElement) 79. { 80. p = p->rightChild; 81. } 82. else if (p->element > theElement) 83. { 84. p = p->leftChild; 85. } 86. else if (p->element == theElement) 87. { 88. *//如果是pair类型，这里就需要覆盖旧值* 89. return 0; 90. } 91. } 92. *//为theElement建立一个节点，然后与pp连接* 93. binaryTreeNode<T>\* newNode = new binaryTreeNode<T>(theElement); 94. if (pp != NULL)   *//树不空* 95. { 96. if (theElement > pp->element) 97. pp->rightChild = newNode; 98. else if (theElement < pp->element) 99. pp->leftChild = newNode; 100. } 101. else 102. { 103. root = newNode; *//插入空树* 104. } 105. treeSize++; 106. *//对节点名次进行计算* 107. p = root; 108. while (p->element != theElement) 109. { 110. if (p->element < theElement) 111. { 112. p = p->rightChild; 113. } 114. else if (p->element > theElement) 115. { 116. p->leftSize++; 117. p = p->leftChild; 118. } 119. } 120. return answer; 121. } 123. template <class T> 124. int binarySearchTree<T>::erase(const T& theElement) 125. {*//删除元素值为theElement的元素* 126. *//删除操作中，如果当前元素有两个孩子，替换的为 右子树中最小的，* 127. *//如果只有一个孩子，直接用该孩子替换当前元素，如果没有孩子，直接删除* 128. binaryTreeNode<T>\* p = root; 129. binaryTreeNode<T>\* pp = NULL; 130. int answer = 0; 131. while (p != NULL && p->element != theElement) 132. {*//p移到它的一个孩子节点* 133. answer ^= p->element; 134. pp = p; 135. if (p->element < theElement) 136. p = p->rightChild; 137. else if (p->element > theElement) 138. p = p->leftChild; 139. } 140. if (p == NULL) 141. return 0;*//不存在与关键值theElement匹配的元素* 143. *//重新组织树结构* 144. answer ^= p->element; 145. p = root; 146. while (p!= NULL && p->element != theElement) 147. { 148. if (p->element < theElement) 149. p = p->rightChild; 150. else if (p->element > theElement) 151. { 152. p->leftSize--; 153. p = p->leftChild; 154. } 155. } 156. *//当p有两个孩子时的处理* 157. *//转化为空或只有一个孩子* 158. *//在p的右子树中寻找最小元素* 159. if (p->leftChild != NULL && p->rightChild != NULL) 160. { 161. binaryTreeNode<T>\* s = p->rightChild; 162. binaryTreeNode<T>\* ps = p; 163. while (s->leftChild != NULL) 164. {*//移到最小的元素* 165. s->leftSize--; 166. ps = s; 167. s = s->leftChild; 168. } 170. binaryTreeNode<T>\* q = new binaryTreeNode<T>(s->element, p->leftChild, p->rightChild, p->leftSize); 171. if (pp == NULL) 172. root = q; 173. else if (p == pp->leftChild) 174. pp->leftChild = q; 175. else 176. pp->rightChild = q; 178. if (ps == p)  pp = q; 179. else  pp = ps; 181. delete p; 182. p = s; 183. } 185. *//p最多有一个孩子，把孩子指针存放在c* 186. binaryTreeNode<T>\* c; 187. if (p->leftChild != NULL) 188. c = p->leftChild; 189. else 190. c = p->rightChild; 191. *//删除p* 192. if (p == root) 193. root = c; 194. else 195. {*//p是pp的左孩子还是右孩子* 196. if (p == pp->leftChild) 197. pp->leftChild = c; 198. else 199. pp->rightChild = c; 200. } 201. treeSize--; 202. delete p; 203. return answer; 204. }  207. template <class T> 208. int binarySearchTree<T>::deleteByIndex(int theIndex) 209. { 210. binaryTreeNode<T>\* p = root; 211. binaryTreeNode<T>\* pp = NULL; 212. int answer = 0; 214. while (p != NULL && p->leftSize != theIndex) 215. {*//p移到它的一个孩子节点* 216. answer ^= p->element; 217. pp = p; 218. if (p->leftSize > theIndex) 219. p = p->leftChild; 220. else if (p->leftSize < theIndex) 221. { 222. theIndex = theIndex - p->leftSize - 1; 223. p = p->rightChild; 224. } 225. } 226. if (p == NULL) 227. return 0; 229. *//重新组织树结构* 230. answer ^= p->element; 231. int theElement = p->element;*//转换到用element去比较* 232. p = root; 233. while (p!= NULL && p->element != theElement) 234. { 235. if (p->element < theElement) 236. p = p->rightChild; 237. else if (p->element > theElement) 238. { 239. p->leftSize--; 240. p = p->leftChild; 241. } 242. } 243. *//当p有两个孩子时的处理* 244. if (p->leftChild != NULL && p->rightChild != NULL) 245. { 246. binaryTreeNode<T>\* s = p->rightChild; 247. binaryTreeNode<T>\* ps = p; 248. while (s->leftChild != NULL) 249. { 250. s->leftSize--; 251. ps = s; 252. s = s->leftChild; 253. } 255. binaryTreeNode<T>\* q = new binaryTreeNode<T>(s->element, p->leftChild, p->rightChild, p->leftSize); 256. if (pp == NULL) 257. root = q; 258. else if (p == pp->leftChild) 259. pp->leftChild = q; 260. else 261. pp->rightChild = q; 263. if (ps == p)  pp = q; 264. else  pp = ps; 266. delete p; 267. p = s; 268. } 270. *//p最多有一个孩子，把孩子指针存放在c* 271. binaryTreeNode<T>\* c; 272. if (p->leftChild != NULL) 273. c = p->leftChild; 274. else 275. c = p->rightChild; 276. *//删除p* 277. if (p == root) 278. root = c; 279. else 280. {*//p是pp的左孩子还是有孩子* 281. if (p == pp->leftChild) 282. pp->leftChild = c; 283. else 284. pp->rightChild = c; 285. } 286. treeSize--; 287. delete p; 288. return answer; 289. }   293. int main() 294. { 295. binarySearchTree<int> BSTree; 297. int m, a, b; 298. cin >> m; 299. for (int i = 0; i < m; i++) 300. { 301. cin >> a >> b; 303. switch (a) 304. { 305. case 0:cout << BSTree.insert(b) << endl; break; 306. case 1:cout << BSTree.find(b) << endl; break; 307. case 2:cout << BSTree.erase(b) << endl; break; 308. case 3: 309. b = b - 1;*//从0开始* 310. cout << BSTree.get(b) << endl; 311. break; 312. case 4: 313. b = b - 1;*//从0开始* 314. cout << BSTree.deleteByIndex(b) << endl; 315. break; 316. } 317. } 319. } | | | |
|  | | | |