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：201900161140 | 姓名： 张文浩 | | 班级： 19级人工智能 |
| 实验题目：二叉树 | | | |
| 实验学时：2 | | 实验日期： 10.5 | |
| 实验目的：  掌握二叉搜索树结构的定义、描述方法、操作实现。 | | | |
| 软件开发工具：  Vscode | | | |
| 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）   因为二叉搜索树的元素数量和形状随着操作而改变，所以二叉搜索树要用链表来描述，所以要建立一个BinaryTreeNode树的节点类，里面包括element元素的值，两个指针分别指向左子树节点和右子树节点，还有一个为了实现按名次查找的用来描述左子树节点个数的变量leftSize。  构造二叉搜索树类binarySearchTree，成员变量包括\*root指向整个二叉搜索树的根节点的指针，和treeSize表示整个二叉搜索树一共有多少节点。方法函数包括find查找函数，insert插入函数、erase删除函数、find\_rank按排名查找函数、erase\_rank按名次删除函数、还有一个题目没有要求，但为了测试方便写的按序输出树的函数output。  **Find(theKey)查找函数：**  查找关键字为theKey的元素，先从根开始查找，如果跟为空，那么搜索树不包含任何元素，即查找失败。如果不为空，则将theKey与根的关键字相比较。如果theKey小，那么久不必在右子树中查找，只要查找左子树。如果theKey大，则正好相反，只需查找右子树。如果theKey等于根的关键字，则查找成功。在子树的查找与此类似。  **Insert(theKey)插入函数：**  首先要通过查找来确定，在树中是否存在某个元素，其关键字与theKey相同。将新元素作为搜索中断节点的孩子插入二叉搜索树。在插入完成之后，还要再次从根节点开始遍历一遍，过来更新节点的leftSize参数。  **Erase(theKey)删除操作：**  假设要删除的节点是p，删除的时候要考虑三种情况：①p是树叶。②p只有一棵非空子树。③p有两棵非空子树。  首先考查情况①：要删除的节点是叶子结点。处理的方法是释放该叶子节点的空间，若是根节点，则令根为NULL。  接下来考察情况②：要删除的节点p只有一棵子树，如果p没有父节点（即p是根节点），则p的唯一子树的根节点成为新的搜索树的根节点。如果p有父节点pp，则修改pp的指针域，使它指向p的唯一孩子，然后释放节点p。  最后考察情况③：要删除的节点p具有两棵非空子树。我们先将该节点的元素替换成它的左子树的最大元素或右子树的最小元素（我的程序选择的是左子树的最大元素），然后把替换元素的节点删除。  **Find\_rank(rank)按名词查找函数：**  整体思路与find查找函数相同，只不过在比较的过程中把比较的对象换成了leftSize和rank。  **Erase\_rank(rank)按名词删除函数：**  整体思路与erase删除函数相同，只是把比较的对象换成了每个Node节点对应的leftSize和rank比较。其他操作与erase相同。   1. 测试结果（测试输入，测试输出）   **C:\WINDOWS\TEMP\WeChat Files\8bfcedea9b847bed94596934551788d.jpg**  C:\WINDOWS\TEMP\WeChat Files\aa87c3f6508c7741c4f4e572596c774.jpg   1. 分析与探讨（结果分析，若存在问题，探讨解决问题的途径）   在本次实验中，我掌握了二叉搜索树结构的定义、描述方法、操作实现  在写erase删除函数的时候逻辑上遇到了一些困难，主要是在删除非空节点后的如何对二叉搜索树进行重建的问题，经过阅读书籍、网上查阅相关资料后学会了如何对二叉搜索树进行删除后的重建操作。方法是考虑三种情况：①p是树叶。②p只有一棵非空子树。③p有两棵非空子树。  首先考查情况①：要删除的节点是叶子结点。处理的方法是释放该叶子节点的空间，若是根节点，则令根为NULL。  接下来考察情况②：要删除的节点p只有一棵子树，如果p没有父节点（即p是根节点），则p的唯一子树的根节点成为新的搜索树的根节点。如果p有父节点pp，则修改pp的指针域，使它指向p的唯一孩子，然后释放节点p。  最后考察情况③：要删除的节点p具有两棵非空子树。我们先将该节点的元素替换成它的左子树的最大元素或右子树的最小元素（我的程序选择的是左子树的最大元素），然后把替换元素的节点删除。  (1)题目中要求的搜索树实际上是索引二叉搜索树，根据课本上的定义即可得知确定名次的方法。只需要在搜索二叉树的插入、删除步骤结束之后更新较小节点左子树的大小便可获得名次。  (2)删除搜索树中元素较难处理。我在编写代码之前对这部分的操作一直感到疑惑，为什么需要讨论删除节点父节点的情况。之后通过阅读书籍，发现了不讨论父节点情况的问题所在：由于二叉搜索树的根节点可能会在每次插入、删除操作之后发生变化，我们需要考虑到父节点是否为空，以及之后根节点是否要改变位置。  (3)这次实验调试过程较为繁琐，因为涉及到太多数据结构的使用。所以可以提前编写一个简单的输入样例，这样可以得出正确的输出结果，与程序的输出做对比即可。   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）   #include <iostream>  using namespace std;  template <class T>  struct BinaryTreeNode  {      T element;      BinaryTreeNode<T> \*leftChild, \*rightChild;      int leftSize; //璇ヨ妭鐐瑰乏瀛愭爲鑺傜偣鐨勪釜鏁帮紝鐢ㄤ簬鎸夊悕娆℃煡鎵惧拰鍒犻櫎鎿嶄綔      // BinaryTreeNode()      // {      //     leftChild = NULL;      //     rightChild = NULL;      //     leftSize = 0;      // }      BinaryTreeNode(const T &theElement)      {          element = theElement;          leftChild = NULL;          rightChild = NULL;          leftSize = 0;      }      BinaryTreeNode(const T &theElement, BinaryTreeNode \*theLeftChild, BinaryTreeNode \*theRightChild, int theleftSize)      {          element = theElement;          leftChild = theLeftChild;          rightChild = theRightChild;          leftSize = theleftSize;      }  };  template <class T>  class binarySearchTree  {  public:      binarySearchTree() //鏋勯€犲嚱锟�?      {          root = NULL;          treeSize = 0;      }      // ~binarySearchTree() { delete[] root; } //鏋愭瀯鍑芥暟      int find(const T &theKey);             //鏌ユ壘鍑芥暟      int insert(const T &theKey);           //鎻掑叆鍑芥暟      int erase(const T &theKey);            //鍒犻櫎鍑芥暟      int find\_rank(int rank);               //鎸夊悕娆℃煡鎵惧嚱锟�?      int erase\_rank(int rank);              //鎸夊悕娆″垹闄ゅ嚱鏁皊      void output(BinaryTreeNode<T> \*t);      BinaryTreeNode<T> \*getroot()      {          return root;      }  private:      BinaryTreeNode<T> \*root; //鏁扮殑鏍硅妭锟�?      int treeSize;            //鏁扮殑鑺傜偣涓暟  };  template <class T>  int binarySearchTree<T>::find(const T &theKey)  {      int res = 0;      BinaryTreeNode<T> \*p = root;      //p浠庢牴鑺傜偣寮€濮嬫悳绱紝瀵绘壘鍏抽敭瀛楃瓑浜巘heKey鐨勪竴涓厓锟�?      while (p != NULL)      {          res ^= p->element;          if (p->element == theKey)              break;          //妫€鏌ュ厓绱爌->element          if (theKey < p->element)              p = p->leftChild;          else if (theKey > p->element)              p = p->rightChild;      }      if (p == NULL)      { //娌℃壘锟�?          return 0;      }      return res;  }  template <class T>  int binarySearchTree<T>::insert(const T &theKey)  {      int res = 0;      //瀵绘壘鎻掑叆浣嶇疆      BinaryTreeNode<T> \*p = root, \*pp = NULL;      while (p != NULL)      {          res ^= p->element;          //妫€鏌ュ厓绱爌->element          pp = p;          //p绉诲姩鍒板畠鐨勪竴涓瀛愮粨锟�?          if (theKey < p->element)          {              p = p->leftChild;          }          else if (theKey > p->element)          {              p = p->rightChild;          }          else if (theKey == p->element) //宸茬粡瀛樺湪锟�?              return 0;      }      //寤虹珛涓€涓柊鑺傜偣锛岀劧鍚庝笌pp閾炬帴      BinaryTreeNode<T> \*newNode = new BinaryTreeNode<T>(theKey);      if (pp != NULL) //鏍戜笉锟�?      {          if (theKey < pp->element)              pp->leftChild = newNode;          else if (theKey > pp->element)              pp->rightChild = newNode;      }      else          root = newNode; //鎻掑叆绌烘爲      treeSize++;      //鍥犱负鎻掑叆浜嗕竴涓妭鐐癸紝鎵€浠ヨ鏇存柊leftSize      p = root;      while (p->element != theKey)      {          if (p->element < theKey)          {              p = p->rightChild;          }          else if (p->element > theKey)          {              p->leftSize++; //宸﹀瓙鏍戞暟锟�?+1              p = p->leftChild;          }      }      return res;  }  template <class T>  int binarySearchTree<T>::erase(const T &theKey)  {      int res = 0;      //鏌ユ壘鍏抽敭瀛椾负theKey鐨勮妭锟�?      BinaryTreeNode<T> \*p = root, \*pp = NULL;      while (p != NULL && p->element != theKey)      {          //绉诲姩p鍒板畠鐨勪笅涓€涓瀛愯妭锟�?          res ^= p->element;          pp = p;          if (theKey < p->element)              p = p->leftChild;          else if (theKey > p->element)              p = p->rightChild;      }      if (p == NULL)          return 0; //涓嶅瓨锟�?      res ^= p->element;      p = root;      //鍥犱负鍒犻櫎浜嗕竴涓妭鐐癸紝鎵€浠ヨ鍐嶆鏌ユ壘theKey锛屾潵鏇存柊鍥犱负鍒犻櫎鎿嶄綔鑰屽彉鍖栫殑leftSize      while (p != NULL && p->element != theKey)      {          if (theKey < p->element)          {              p->leftSize--;              p = p->leftChild;          }          else if (theKey > p->element)          {              p = p->rightChild;          }      }      //閲嶆柊缁勭粐鏍戠粨锟�?      //褰損鏈変袱涓瀛愯妭鐐规椂鐨勫锟�?      if (p->leftChild != NULL && p->rightChild != NULL)      {          //涓や釜瀛╁瓙锛岃浆鍖栦负绌烘垨鍙湁涓€涓瀛愶紝鍦╬鐨勫乏鍙冲瓙鏍戜腑瀵绘壘鏈€灏忓厓锟�?          BinaryTreeNode<T> \*s = p->rightChild, \*ps = p; //s鐨勫弻锟�?          while (s->leftChild != NULL)          {              //绉诲姩鍒版渶澶х殑鍏冪礌              s->leftSize--;              ps = s;              s = s->leftChild;          }          //灏嗘渶澶х殑鍏冪礌s绉诲姩鍒皃锛屼絾涓嶆槸绠€鍗曠殑绉诲姩          BinaryTreeNode<T> \*q = new BinaryTreeNode<T>(s->element, p->leftChild, p->rightChild, p->leftSize);          if (pp == NULL)              root = q;          else if (p == pp->leftChild)          {              pp->leftChild = q;          }          else              pp->rightChild = q;          if (ps == p)              pp = q;          else              pp = ps;          delete p;          p = s;      }      //p鏈€澶氭湁涓€涓瀛愶紝鎶婂瀛愭寚閽堝瓨鏀惧湪c      BinaryTreeNode<T> \*c;      if (p->leftChild != NULL)          c = p->leftChild;      else          c = p->rightChild;      //鍒犻櫎p      if (p == root)          root = c;      else      {          //p鏄痯p鐨勫乏瀛╁瓙杩樻槸鍙冲锟�?          if (p == pp->leftChild)              pp->leftChild = c;          else              pp->rightChild = c;      }      treeSize--;      delete p;      return res;  }  template <class T>  int binarySearchTree<T>::find\_rank(int rank)  {      int res = 0;      BinaryTreeNode<T> \*p = root; //浠庢牴鑺傜偣寮€濮嬫壘      while (p != NULL)      {          res ^= p->element;          if (p->leftSize == rank)              break;          if (rank < p->leftSize)              p = p->leftChild;          else if (rank > p->leftSize)          {              rank -= p->leftSize + 1;              p = p->rightChild;          }      }      if (p == NULL)      {          return 0;      }      return res;  }  template <class T>  int binarySearchTree<T>::erase\_rank(int rank)  {      int res = 0;      //鏌ユ壘鍚嶆涓簉ank鐨勮妭锟�?      BinaryTreeNode<T> \*p = root, \*pp = NULL;      while (p != NULL && p->leftSize != rank)      {          //绉诲姩p鍒板畠鐨勪笅涓€涓瀛愯妭锟�?          res ^= p->element;          pp = p;          if (rank < p->leftSize)              p = p->leftChild;          else if (rank > p->leftSize)          {              rank -= p->leftSize + 1;              p = p->rightChild;          }      }      if (p == NULL)          return 0; //涓嶅瓨锟�?      res ^= p->element;      //鎵惧埌瀵瑰簲rank鐨則heKey      int theKey = p->element;      p = root;      //鍥犱负鍒犻櫎浜嗕竴涓妭鐐癸紝鎵€浠ヨ鍐嶆鏌ユ壘theKey锛屾潵鏇存柊鍥犱负鍒犻櫎鎿嶄綔鑰屽彉鍖栫殑leftSize      while (p != NULL && p->element != theKey)      {          if (theKey < p->element)          {              p->leftSize--;              p = p->leftChild;          }          else if (theKey > p->element)          {              p = p->rightChild;          }      }      //閲嶆柊缁勭粐鏍戠粨锟�?      //褰損鏈変袱涓瀛愯妭鐐规椂鐨勫锟�?      if (p->leftChild != NULL && p->rightChild != NULL)      {          //涓や釜瀛╁瓙锛岃浆鍖栦负绌烘垨鍙湁涓€涓瀛愶紝鍦╬鐨勫彸瀛愭爲涓鎵炬渶灏忓厓锟�?          BinaryTreeNode<T> \*s = p->rightChild, \*ps = p; //s鐨勫弻锟�?          while (s->leftChild != NULL)          {              //绉诲姩鍒版渶澶х殑鍏冪礌              s->leftSize--;              ps = s;              s = s->leftChild;          }          //灏嗘渶澶х殑鍏冪礌s绉诲姩鍒皃锛屼絾涓嶆槸绠€鍗曠殑绉诲姩          BinaryTreeNode<T> \*q = new BinaryTreeNode<T>(s->element, p->leftChild, p->rightChild, p->leftSize);          if (pp == NULL)              root = q;          else if (p == pp->leftChild)          {              pp->leftChild = q;          }          else              pp->rightChild = q;          if (ps == p)              pp = q;          else              pp = ps;          delete p;          p = s;      }      //p鏈€澶氭湁涓€涓瀛愶紝鎶婂瀛愭寚閽堝瓨鏀惧湪c      BinaryTreeNode<T> \*c;      if (p->leftChild != NULL)          c = p->leftChild;      else          c = p->rightChild;      //鍒犻櫎p      if (p == root)          root = c;      else      {          //p鏄痯p鐨勫乏瀛╁瓙杩樻槸鍙冲锟�?          if (p == pp->leftChild)              pp->leftChild = c;          else              pp->rightChild = c;      }      treeSize--;      delete p;      return res;  }  template <class T>  void binarySearchTree<T>::output(BinaryTreeNode<T> \*t)  {      if (t != NULL)      {          output(t->leftChild);          cout << t->element << " ";          output(t->rightChild);      }      else          return;  }  int main()  {      int m, a, b;      binarySearchTree<int> tree;      cin >> m;      while (m--)      {          cin >> a >> b;          if (a == 0)              cout << tree.insert(b) << endl;          else if (a == 1)              cout << tree.find(b) << endl;          else if (a == 2)          {              cout << tree.erase(b) << endl;          }          else if (a == 3)              cout << tree.find\_rank(b - 1) << endl;          else if (a == 4)              cout << tree.erase\_rank(b - 1) << endl;          cout<<"当前搜索树的情况"<<endl;          tree.output(tree.getroot());          cout << endl;      }      system("pause");      return 0;  } | | | |