g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：201705130113 | 姓名： 黄瑞哲 | | 班级： 计科17.3 |
| 实验题目：搜索树 | | | |
| 实验学时：2 | | 实验日期：2018.12.13 | |
| 实验目的：   1. 掌握二叉搜索树结构的定义、描述方法、操作实现 | | | |
| 软件环境：  Visual Studio Community 2017 | | | |
| 1. 实验内容（题目内容，输入要求，输出要求） 2. 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。 3. 接受键盘录入的一系列整数，输出其对应的二叉搜索树（可用文本形式输出），以及二叉搜索树的高度。 4. 对建立好的二叉搜索树，执行上述各操作（插入操作除外），输出各操作的结果（插入、删除操作输出完成后的二叉搜索树：查找操作输出查找过程中依次比较的元素）。 5. 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）   二叉搜索树满足左子树的元素值都小于根节点，右子树的元素值都大于根节点，基于这个性质，各操作方法如下。   1. 插入：如果插入元素小于根节点则进入左子树，如果左子树为空则将其插入在这个位置；如果插入元素大于根节点则进入右子树，如果右子树为空则将其插入在这个位置。 2. 查找：如果根节点等于要查找的元素值则直接返回结果，如果查找元素小于根节点则进入左子树查找，如果大于则进入右子树，如果某一节点为空则查找失败。 3. 按名次查找：每个节点维护其左子树的节点个数leftSize。如果排名rank > leftSize + 1则说明应该在其右子树中查找排名为rank-leftSize-1的元素。如果排名rank < leftSize + 1则说明应该在其左子树中查找排名为rank的元素。如果找到空节点则查找失败。 4. 删除：先查找到要删除的节点，没有找到则返回删除失败。如果待删除节点的左右子树均存在则替换为右子树中元素最小的，如果只有一个子树则直接替换为这个子树，否则直接删除这个节点。 5. 按名次删除：首先按名次查找到待删除的元素，替换操作和删除时的替换操作一致。 6. 测试结果（测试输入，测试输出，结果分析）      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. 在删除元素时如果删除元素在左子树中需要维护当前节点的leftSize值使其自减1，右子树无需维护 3. 在删除节点时也可以选择用左子树的最大值替代也可以选择用右子树的最小值替代 4. 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）   /\*binarySerachTtree.h\*/  #ifndef BINARY\_SEARCH\_TREE\_  #define BINARY\_SEARCH\_TREE\_  #include "linkedBinaryTree.h""  #include "binaryTreeNode.h"  #include <iostream>  template<typename T>  class binary\_seach\_tree : linkedBinaryTree<pair<int, T>> {  public:  typedef binaryTreeNode<pair<int, T>> node;  binary\_seach\_tree() { this->root = nullptr; }  ~binary\_seach\_tree() {}  bool empty() const override { return linkedBinaryTree<pair<int, T>>::empty(); }  int size() const override { return linkedBinaryTree<pair<int, T>>::size(); }  int height() const { return linkedBinaryTree<pair<int, T>>::height(); }  pair<int, T>\* find(const T&);  pair<int, T>\* find\_by\_rank(int);  void insert(const T& element) { insert(this->root, element); };  void erase(const T&);  void erase\_by\_rank(int);  void ascend() { this->inOrder(output); std::cout << std::endl; }  protected:  static int insert(node\*&, const T&);  static node\* erase(node\*, const T&);  static node\* erase\_by\_rank(node\*, int);  static void output(node\* t) { std::cout << t->element.second << ' '; }  };  template<typename T>  pair<int, T>\* binary\_seach\_tree<T>::find(const T& element) {  node\* rt = this->root;  while (rt != nullptr) {  cout << rt->element.second << ' ';  if (rt->element.second < element)rt = rt->rightChild;  else if (rt->element.second > element) rt = rt->leftChild;  else return &rt->element;  }  return nullptr;  }  template <typename T>  pair<int, T>\* binary\_seach\_tree<T>::find\_by\_rank(int rank) {  node\* rt = this->root;  while (rt != nullptr) {  cout << rt->element.second << ' ';  if (rt->element.first + 1 < rank) {  //find in rightChild with rank - leftSize - 1  rank -= rt->element.first + 1;  rt = rt->rightChild;  }  //find in leftChild with rank  else if (rt->element.first + 1 > rank) rt = rt->leftChild;  else return &rt->element;  }  return nullptr;  }  //if insert an element successfully, return 1, else 0  template<typename T>  int binary\_seach\_tree<T>::insert(node\*& rt, const T& element) {  if (rt == nullptr) {  rt = new node(make\_pair(0, element));  return 1;  }  if (rt->element.second < element) {  if (rt->rightChild == nullptr) {  node\* new\_node = new node(make\_pair(0, element));  new\_node->father = rt;  rt->rightChild = new\_node;  return 1;  }  return insert(rt->rightChild, element);  }  else if (rt->element.second > element) {  if (rt->leftChild == nullptr) {  node\* new\_node = new node(make\_pair(0, element));  new\_node->father = rt;  rt->leftChild = new\_node;  ++rt->element.first;  return 1;  }  int add = insert(rt->leftChild, element);  //if success, leftSize increase  rt->element.first += add;  return add;  }  return 0;  }  template<typename T>  void binary\_seach\_tree<T>::erase(const T& element) {  node\* p = erase(this->root, element);//find the element  node\* pp = nullptr;  if (p != nullptr) {  if (p->rightChild != nullptr && p->leftChild != nullptr) {//both have leftChild and rightChild  pp = p->rightChild;  while (pp->leftChild != nullptr) {//find the minium one  --pp->element.first;//lose one of their leftChild  pp = pp->leftChild;  }  if (pp->father == p) {//if p is pp's father  pp->father->rightChild = pp->rightChild;  if (pp->rightChild != nullptr) pp->rightChild->father = pp->father;  }  else {//let pp's rightChild be its father's leftChild  pp->father->leftChild = pp->rightChild;  if (pp->rightChild != nullptr) pp->rightChild->father = pp->father;  }  //implement all element in node p  pp->element.first = p->element.first;  pp->leftChild = p->leftChild;  if (pp->leftChild != nullptr) pp->leftChild->father = pp;  pp->rightChild = p->rightChild;  if (pp->rightChild != nullptr) pp->rightChild->father = pp;  pp->father = p->father;  }  //only have one child  else if (p->leftChild != nullptr) {  if (p != this->root) {  if (p->father->leftChild == p) p->father->leftChild = p->leftChild;  else p->father->rightChild = p->leftChild;  if (p->leftChild != nullptr) p->leftChild->father = p->father;  }  else {  this->root = p->leftChild;  }  delete p;  return;  }  else if (p->rightChild != nullptr) {  if (p != this->root) {  if (p->father->leftChild == p) p->father->leftChild = p->rightChild;  else p->father->rightChild = p->rightChild;  if (p->rightChild != nullptr)p->rightChild->father = p->father;  }  else {  this->root = p->rightChild;  }  delete p;  return;  }  //check whether p is the root  if (p == this->root) {  this->root = pp;  }  else {  if (p->father->leftChild == p) p->father->leftChild = pp;  else p->father->rightChild = pp;  }  delete p;  }  }  template<typename T>  binaryTreeNode<pair<int, T>>\* binary\_seach\_tree<T>::erase(node\* rt, const T& element) {  if (rt == nullptr) return nullptr;  if (rt->element.second < element) return erase(rt->rightChild, element);  else if (rt->element.second > element) {  //they will lose thier leftChild, so the leftSize will decrease  node\* tmp = erase(rt->leftChild, element);  if (tmp != nullptr) --rt->element.first;  return tmp;  }  else return rt;  }  template<typename T>  void binary\_seach\_tree<T>::erase\_by\_rank(int rank) {  node\* p = erase\_by\_rank(this->root, rank);//find the element  node\* pp = nullptr;  if (p != nullptr) {  if (p->rightChild != nullptr && p->leftChild != nullptr) {//both have leftChild and rightChild  pp = p->rightChild;  while (pp->leftChild != nullptr) {//find the minium one  --pp->element.first;//lose one of their leftChild  pp = pp->leftChild;  }  if (pp->father == p) {//if p is pp's father  pp->father->rightChild = pp->rightChild;  if (pp->rightChild != nullptr) pp->rightChild->father = pp->father;  }  else {//let pp's rightChild be its father's leftChild  pp->father->leftChild = pp->rightChild;  if (pp->rightChild != nullptr) pp->rightChild->father = pp->father;  }  //implement all element in node p  pp->element.first = p->element.first;  pp->leftChild = p->leftChild;  if (pp->leftChild != nullptr) pp->leftChild->father = pp;  pp->rightChild = p->rightChild;  if (pp->rightChild != nullptr) pp->rightChild->father = pp;  pp->father = p->father;  }  //only have one child  else if (p->leftChild != nullptr) {  if (p != this->root) {  if (p->father->leftChild == p) p->father->leftChild = p->leftChild;  else p->father->rightChild = p->leftChild;  if (p->leftChild != nullptr) p->leftChild->father = p->father;  }  else {  this->root = p->leftChild;  }  delete p;  return;  }  else if (p->rightChild != nullptr) {  if (p != this->root) {  if (p->father->leftChild == p) p->father->leftChild = p->rightChild;  else p->father->rightChild = p->rightChild;  if (p->rightChild != nullptr)p->rightChild->father = p->father;  }  else {  this->root = p->rightChild;  }  delete p;  return;  }  //check whether p is the root  if (p == this->root) {  this->root = pp;  }  else {  if (p->father->leftChild == p) p->father->leftChild = pp;  else p->father->rightChild = pp;  }  delete p;  }  }  template<typename T>  binaryTreeNode<pair<int, T>>\* binary\_seach\_tree<T>::erase\_by\_rank(node\* rt, int rank) {  if (rt == nullptr) return nullptr;  if (rt->element.first + 1 < rank) return erase\_by\_rank(rt->rightChild, rank - 1 - rt->element.first);  else if (rt->element.first + 1 > rank) {  // they will lose their leftChild, so the leftSize will decrease  node\* tmp = erase\_by\_rank(rt->leftChild, rank);  if (tmp != nullptr) --rt->element.first;  return tmp;  }  else return rt;  }  #endif //BINARY\_SEARCH\_TREE  /\*pch.h\*/  #ifndef PCH\_H  #define PCH\_H  // **TODO: 添加要在此处预编译的标头**  #include <tree/binarySearchTree.h>  #endif //PCH\_H  /\*main.cpp\*/  #include "pch.h"  #include <iostream>  using namespace std;  int main() {  binary\_seach\_tree<int> t;  int n;  cin >> n;  for (int i = 1; i <= n; ++i) {  int a;  cin >> a;  t.insert(a);  }  cout << "ascend: ";  t.ascend();  cout << "height: " << t.height() << endl;  cin >> n;  auto p = t.find(n);  cout << endl;  if (p == nullptr) cout << "failed" << endl;  cin >> n;  p = t.find\_by\_rank(n);  cout << endl;  if (p == nullptr) cout << "failed" << endl;  cin >> n;  t.erase(n);  cout << "ascend" << endl;  t.ascend();  cin >> n;  t.erase\_by\_rank(n);  cout << "ascend" << endl;  t.ascend();  return 0;  } | | | |