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：201700140056 | 姓名：李港 | | 班级：跟18.2（17.4） |
| 实验题目：实验十二 图 | | | |
| 实验学时：4h | | 实验日期：2019.12.9 | |
| 实验目的：   * 1. 掌握图的基本概念，图的描述方法；图上的操作方法实现。   2. 掌握图结构的应用。 | | | |
| 软件开发工具：  Virtual Studio 2019 | | | |
| 1. **实验内容**    1. 创建无向图类。存储结构分别使用邻接矩阵和邻接链表。提供操作：       1. 插入一条边       2. 删除一条边       3. 遍历：BFS、DFS。    2. 键盘输入图中顶点的个数n和边的数目e，以顶点对（i，j）形式依次输入图的每一条边或随机生成含e条边的图，其中（i,j）表示顶点i和顶点j之间有边相连，建立图。    3. 判断图是否连通。若不连通，输出该图的连通分量的个数及每个连通分量中的顶点；    4. 对建立好的连通图，键盘输入一顶点，输出从该顶点开始的一个DFS序列和BFS序列；一个DFS生成树和BFS生成树（树可以文本形式输出）    5. 键盘输入两顶点，输出两顶点之间的最短路径。 2. **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）**   **总体思路：**   * 1. 由于二叉搜索树左子树小于根节点，右子树大于等于根节点，所以二叉搜索树的升序输出即为二叉树前序遍历。   2. 计算二叉树的高度采用之前二叉树实验中的数组法。   3. 为了降低按名词查询的时间复杂度，在结点结构体中添加一个整数成员，记录左子树大小。   **数据结构：**   * 1. 设计节点类，注意要有简化名次查询时间复杂度的“左子树大小”成员。   template<class T>  struct node {  T data;  int left\_size; //代表结点在以自己为根的树中的名次  node<T>\* left; //指向左子树的指针  node<T>\* right; //指向右子树的指针  //空构造函数  node ()  :left(nullptr),right(nullptr), left\_size(0){}  //带子节点构造函数  node (const T& data\_in, node<T>\* left\_in, node<T>\* right\_in)  :left\_size (0), data (data\_in), left (left\_in), right (right\_in) {}  //有数据构造函数  node (const T& data\_in)  :data (data\_in),left (nullptr),right ( nullptr), left\_size (0) { }  node (const node\* node\_in)  :data (node\_in->data), left (node\_in->left),  right (node\_in->right), left\_size (node\_in->left\_size) {}  };  **算法：**   * 1. 搜索      1. 使用p保存当前结点      2. 将根节点设置为p结点      3. 开始循环         1. 若p结点比目标结点大，则p结点指向p的左子节点，重新进行对比         2. 若p结点比目标结点小，则p结点指向p的右子节点，重新进行对比         3. 否则，p结点就是要寻找的结点   2. 删除      1. 查询找到目标结点后，分为四种不同的情况         1. 有左右子树：找到右子树的最小元素代替当前结点。         2. 无左右子树：直接删除该元素，其父结点指向目标结点的指针置为空。         3. 有右子树无左子树：使用右子结点替换当前结点。         4. 有左子树无右子树：使用左子结点替换当前结点。   3. 插入      1. 对于任何一个插入值和任何一个“当前结点”，它要么大于等于当前结点，要么小于当前结点，所以只需查找对比到树的叶子处再进行空指针的替换操作即可。   4. 按名次查询      1. 保存名次信息，当按照名次查询时，使用名次代替数据作为比较关键词。      2. 注意，这里使用“左子树元素数”来作为一个节点在以自己为根的树中的名次，所以每次向右子树进行查找的时候都要用“目标名次-父节点的左子树的左子树元素数-1”来代替目标名次。   5. 按名次删除      1. 可以通过包装名次查询与普通删除的操作来实现按名次删除。  1. **测试结果（测试输入，测试输出）**    * 1. 验收展示：   测试初始化与插入功能：    测试删除操作：    测试查找功能：    如图可见，程序输出正确。   * 1. 平台提交      1. **分析与探讨（结果分析，若存在问题，探讨解决问题的途径）**   本实验最终结果正确，在实验过程中有以下问题或心得：   * 1. 二叉搜索树的删除操作较为复杂，需要考虑四种情况，即被删除的结点没有子节点，有左右子节点，有左无右子节点，有右无左子节点。   2. 在编程中可以通过添加某些临时变量的方式简化运算，比如按照名次搜索元素的功能需要用到名次信息，如果每次都要计算名次则大大增加了时间复杂度。本实验可以通过提高空间复杂度的方式来降低时间复杂度。   3. 链表之类的程序中，用哨兵变量来进行父节点的保存是常见的思路。  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）**   文件1 main.h  #include<iostream>  #include"bstree.h"  using namespace std;  #define fori(i,n) for(int i=0;i<(int)(n);i++)  #define LGRAND(min,max) ((rand()%(max-min+(int)1))+(int)min )  int main () {  ios::sync\_with\_stdio (false);  cin.tie (0);  cout.tie (0);  //#pragma warning(disable:4996)  // freopen ("input.txt", "r", stdin);  bsTree<int> b;  int num = 10;  int nums[] = { 6, 8, 9, 7,1, 5, 4, 3,11,15};  //初始化二叉搜索树，随机1-100  fori (i,num) {  b.insert (nums[i]);  }  cout << "树内容为：" << b;  cout << "树高度为：" << calF (b)<<"\n";  cout << "\n\n测试各种操作：插入、删除、按名次删除、查找、按名次查找\n";  cout << "\n\n测试插入 99\n";  b.insert (99);  cout << "树内容为：" << b;  cout << "\n\n测试删除 99\n";  b.erase (99);  cout << "树内容为：" << b;  cout << "\n\n测试按名次删除 1\n";  b.eraseByIndes (1);  cout << "树内容为：" << b;  cout << "\n\n测试查找 99 7\n";  cout << b.search (99)<<"\n\n";  cout << b.search (7);  cout << "\n\n测试按名次查找 b[4]==7\n";  cout<<b.getPByIndex (4);  }  文件2 bstree.cpp  #pragma once  #include<iostream>  using std::ostream;  using std::cout;  template<class T>  struct node {  T data;  int left\_size; //代表结点在以自己为根的树中的名次  node<T>\* left; //指向左子树的指针  node<T>\* right; //指向右子树的指针  //空构造函数  node ()  :left (nullptr), right (nullptr), left\_size (0) {}  //带子节点构造函数  node (const T& data\_in, node<T>\* left\_in, node<T>\* right\_in)  :left\_size (0), data (data\_in), left (left\_in), right (right\_in) {}  //有数据构造函数  node (const T& data\_in)  :data (data\_in), left (nullptr), right (nullptr), left\_size (0) {}  node (const node\* node\_in)  :data (node\_in->data), left (node\_in->left), right (node\_in->right), left\_size (node\_in->left\_size) {}  };  template<class T>  class bstree {  protected:  node<T>\* \_root;  T \_size;  void \_inOrder (node<T>\* t, ostream& out);  void friend cal (node<int>\* rootin, int\* sizes, int\* deepthes);  int friend calF (bstree<int>& bst);  public:  int xor\_result;  bstree () :\_root (nullptr), \_size (0) {}  bool insert (T data\_in);  node<T>\* search (T data\_in);  node<T>\* getPByIndex (int index);//索引从零开始  bool erase (T data\_in);  bool eraseByIndes (int index);  ostream& inOrder (ostream& out);  };  template<class T>  void bstree<T>::\_inOrder (node<T>\* t, ostream& out) {  if(t->left)\_inOrder (t->left,out);  out << t->data<<" ";  if (t->right)\_inOrder (t->right,out);  }  template<class T>  ostream& bstree<T>::inOrder (ostream& out) {  \_inOrder (\_root,out);  return out;  }  template<class T>  ostream& operator << (ostream& out, bstree<T> b) {  b.inOrder(out);  return out;  }  template<class T>  bool bstree<T>::insert (T data\_in) {  //检查目标结点是否存在  node<T>\* temp = search (data\_in);  if (temp != nullptr && temp->data == data\_in) {  //找不到则返回false  return false;  }  xor\_result = 0; //初始化异或值  node<T>\* p = \_root; //p是用来寻找目标位置的指针  node<T>\* pp = nullptr; //pp是为了避免p指向空叶子结点而造成目标丢失  /\*循环寻找目标位置\*/  while (p != nullptr) {  pp = p;  xor\_result = xor\_result ^ p->data;  if (data\_in < p->data) {  p->left\_size++;  p = p->left;  } else {  p = p->right;  }  }  node<T>\* new\_node = new node<T> (data\_in); //构造新的结点  if (\_root != nullptr) { //若树不为空则插入到合适位置  if (data\_in < pp->data) {  pp->left = new\_node;  } else {  pp->right = new\_node;  }  } else { //若树为空则直接代替root  \_root = new\_node;  }  \_size++;  return true;  }  /\*寻找目标值元素，返回指针\*/  template<class T>  node<T>\* bstree<T>::search (T data\_in) {  xor\_result = 0;  node<T>\* p = \_root;  while (p != nullptr) {  xor\_result = xor\_result ^ p->data;  if (data\_in < p->data) {  cout << "对比："<< data\_in << " < " << p->data<<"\n";  p = p->left;  } else {  if (data\_in > p->data) {  cout << "对比："<< data\_in << " > " << p->data << "\n";  p = p->right;  } else {  return p;//找到目标便返回  }  }  }  return nullptr;  }  template<class T>  node<T>\* bstree<T>::getPByIndex (int index) {  if (index < 0 || index >= \_size) {  return nullptr;  }  xor\_result = 0;  node<T>\* p = \_root;  while (p != nullptr) {  xor\_result = xor\_result ^ p->data;  if (p->left\_size == index) {  cout << "对比：" << p->left\_size << " == " << index << "\n";  return p;//直接返回  } else {  if (p->left\_size > index) {  cout << "对比："<< p->left\_size << " > " << index << "\n";  p = p->left;  } else {  index = index - p->left\_size - 1;//这里注意，向右下方寻找时记得修改目标索引值  p = p->right;  }  }  }  return nullptr;  }  template<class T>  bool bstree<T>::erase (T data\_in) {  node<T>\* temp = search (data\_in);  if (temp == nullptr) {  return false;  }  //p是目标元素，pp是其父元素  xor\_result = 0;  node<T>\* p = \_root;  node<T>\* pp = nullptr;  while (p->data != data\_in && p != nullptr) {  xor\_result = xor\_result ^ p->data;  pp = p;  if (data\_in < p->data) {  p->left\_size--;  p = p->left;  } else {  p = p->right;  }  }  xor\_result = xor\_result ^ data\_in;  if (p == nullptr) {  return false;  }  //寻找可代替p的s结点  if (p->left != nullptr && p->right != nullptr) {  node<T>\* s = p->right;  node <T>\* ps = p;  while (s->left != nullptr) {  ps = s;  s->left\_size--;  s = s->left;  }  node<T>\* q = new node<T> (p);  q->data = s->data;  if (pp == nullptr) {  \_root = q;  } else if (p == pp->left) {  pp->left = q;  } else {  pp->right = q;  }  if (ps == p) {  pp = q;  } else {  pp = ps;  }  delete p;  p = s;  }  node<T>\* c = nullptr;  if (p->left != nullptr) {  c = p->left;  } else {  c = p->right;  }  if (p == \_root) { \_root = c; } else {  if (p == pp->left) {  pp->left = c;  } else {  pp->right = c;  }  }  \_size--;  delete p;  return true;  }  template<class T>  bool bstree<T>::eraseByIndes (int index) {  if (index < 0 || index >= \_size) {  return false;  }  xor\_result = 0;  node<T>\* temp = getPByIndex (index);  if (temp == nullptr) {  return false;  }  erase (temp->data);  return true;  }  /\*本题目第二三个小任务的核心函数，计算每个节点为根的树的层数与节点数\*/  #define max(a,b) (a<b ? b:a)//用于获取左右子树中最大的那个层数  void cal (node<int>\* rootin, int\* sizes, int\* deepthes) {  //如果传入空则直接返回  if (rootin) {  //首先递归计算节点为根的树的节点数与层数，所有子节点全部计算完毕后再计算当前元素  cal (rootin->left, sizes, deepthes);  cal (rootin->right, sizes, deepthes);  //对于该节点  //若当前元素没有子节点，则以其为根的树的节点数与层数均为1  if (rootin->left == nullptr && rootin->right == nullptr) {  sizes[rootin->data] = 1;  deepthes[rootin->data] = 1;  //cout << deepthes[rootin->data] << " " << sizes[rootin->data] << "都无\n";  //若当前元素有左子树，则以其为根的树的节点数与层数为左子树相应数据加1  } else if (rootin->left != nullptr && rootin->right == nullptr) {  sizes[rootin->data] = sizes[rootin->left->data] + 1;  deepthes[rootin->data] = deepthes[rootin->left->data] + 1;  //cout << deepthes[rootin->data] << " " << sizes[rootin->data] << "左有\n";  //若当前元素有右子树，则以其为根的树的节点数与层数为右子树相应数据加1  } else if (rootin->left == nullptr && rootin->right != nullptr) {  sizes[rootin->data] = sizes[rootin->right->data] + 1;  deepthes[rootin->data] = deepthes[rootin->right->data] + 1;  //cout << deepthes[rootin->data] << " " << sizes[rootin->data] << "右都有\n";  //若当前元素有左右子树，则以其为根的树的节点数与层数为左右子树相应数据相加再加1  } else if (rootin->left != nullptr && rootin->right != nullptr) {  sizes[rootin->data] = sizes[rootin->right->data] + sizes[rootin->left->data] + 1;  deepthes[rootin->data] = max (deepthes[rootin->right->data], deepthes[rootin->left->data]) + 1;  //cout << deepthes[rootin->data] <<" "<< sizes[rootin->data] << "左右都有\n";  }  }  return;  }  int calF (bstree<int>& bst) {  /\*初始化存储节点数与层数的数组\*/  int\* sizes = new int[100 + 1];  memset (sizes, 0, 100 + 1);  int\* deepthes = new int[100 + 1];  memset (deepthes, 0, 100 + 1);  cal (bst.\_root,sizes,deepthes);  return deepthes[bst.\_root->data];  } | | | |