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：实验九 二叉树操作 | | | |
| 实验学时：2h | | 实验日期：2019.11.14 | |
| 实验目的：   1. 掌握二叉树的基本概念，链表描述方法 2. 掌握二叉树操作的实现 | | | |
| 软件开发工具：  Virtual Studio 2019 | | | |
| 1. **实验内容**    1. 创建二叉树类。二叉树的存储结构使用链表。提供操作:前序遍历、中序遍历、后序遍历、层次遍历、计算二叉树结点数目、计算二叉树高度。    2. 对建立好的二叉树，执行上述各操作，输出各操作的结果。    3. 接收键盘录入的二叉树前序序列和中序序列(各元素各不相同)，输出该二叉树的后序序列。 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. 采用链表存储二叉树。   2. 提供前序、中序、后序遍历功能。   3. 采用数组与链表配合的方式实现节点数与层数的计算功能。   4. 为了与数组配合，二叉树类提供重新设置根节点的功能。   **数据结构：**   * 1. 通过node结构体保存每个节点的信息。   2. 通过链表保存二叉树。   3. 在统计二叉树的节点数与层数中采用数组。   node结构体：  typedef struct node {  T data;  node\* left;  node\* right;  node (T data, node\* left, node\* right)  :data (data), left (left), right (right) {}  node ()  :left (nullptr), right (nullptr) {}  } node;  **算法：**   * 1. 前序、中序、后序遍历采用递归实现。   2. 层次遍历采用队列实现。   3. 计算以每个节点为根的树的节点数与层数采用数组与链表协作实现。递归地计算节点数与层数。  1. **测试结果（测试输入，测试输出）**    1. 验收展示： 本演示程序分为两部分，第一部分构建一个树，再进行前序、后序、中序、层析遍历，并输出元素数量与层数；第二部分通过前序遍历与中序遍历构建树，并输出后序遍历。       1. 对建立好的二叉树，执行上述各操作，输出各操作的结果：       2. 通过前序序列和中序序列生成后序序列：    2. 平台提交       1. 初始化一棵树，计算节点数与层数：      * + 1. 根据前序遍历序列与中序遍历序列生成二叉树并显示后续遍历序列：      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. OJ第二题一开始准备使用定长数组，后来发现题目中测试数据过大，造成数组越界，因此改用变长数组。   2. 第一题有个sizes写成了deepthes，导致结果错误。   3. 后来在计算节点数时忘记在某个条件分支处加一，导致结果错误。   4. 根据题目要求适当地变化数据结构，或进行提前计算结果，可以极大提高算法效率，这也是算法竞赛做题的常用技巧。  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 btree.h  #pragma once  #include<iostream>  #include"queue.h"  using std::cout;  using std::endl;  using std::ostream;  /\*class btree  public:  enum err; 常见错误  struct node 节点结构体  protected:  node\* \_root; 树根指针  int \_size; 树节点数量  void deleteNodes (node\* root) 删除以传入节点为根的树  node\* \_makeNodeFromPreIn (T\* pre, T\* in, int in\_length) 以输入的前序与中序序列递归生成一颗树  ostream& \_preOut (ostream& out, node\* rootin) 前序遍历  ostream& \_postOut (ostream& out, node\* rootin) 后续遍历  public:  btree () 构造函数  ~btree () 析构函数，递归删除所有节点  void clear () 清空此树，根节点置为空  void buildFromPreIn (T\* pre\_head\_in, T\* in\_head\_in, int length\_in)暴露给外部的根据前序与中序序列生成树的函数  ostream& preOut (ostream& out) 暴露给外界的前序遍历接口  ostream& postOut (ostream& out) 暴露给外界的后序遍历接口  ostream& levelOut (ostream& out) 暴露给外界的层次遍历接口  void setRoot (node\* rootin) 设置根节点  \*/  template<class T>  class btree {  public:  typedef enum {} err;  typedef struct node {  T data;  node\* left;  node\* right;  node (T data, node\* left, node\* right)  :data (data), left (left), right (right) {}  node ()  :left (nullptr), right (nullptr) {}  } node;  protected:  node\* \_root;  int \_size;  void deleteNodes (node\* root) {  if (root) {  //cout << "delete" << root->data << "\n";  if (root->left)deleteNodes (root->left);  if (root->right)deleteNodes (root->right);  delete root;  }  }  node\* \_makeNodeFromPreIn (T\* pre, T\* in, int in\_length) {  node\* new\_node;  if (in\_length == 0) {  //cout << "leaf node\n";  return nullptr;  } else {  new\_node = new node;  int len = 0;  while (pre[0] != in[len]) {  len++;  }  new\_node->data = pre[0];  //cout << pre[0] << "make in \n";  new\_node->left = \_makeNodeFromPreIn (pre + 1, in, len);  new\_node->right = \_makeNodeFromPreIn (pre + len + 1, in + len + 1, in\_length - len - 1);  return new\_node;  }  }  ostream& \_preOut (ostream& out, node\* rootin) {  if (rootin == nullptr) {  return out;  } else {  out << rootin->data << " ";  \_preOut (out, rootin->left);  \_preOut (out, rootin->right);  return out;  }  }  ostream& \_postOut (ostream& out, node\* rootin) {  if (rootin == nullptr) {  return out;  } else {  \_postOut (out, rootin->left);  \_postOut (out, rootin->right);  out << rootin->data << " ";  return out;  }  }  ostream& \_midOut (ostream& out, node\* rootin) {  if (rootin == nullptr) {  return out;  } else {  \_midOut (out, rootin->left);  out << rootin->data << " ";  \_midOut (out, rootin->right);  return out;  }  }  public:  btree () {  \_root = nullptr;  }  ~btree () {  //cout << "dis\n";  if (\_root) {  if (\_root->left)deleteNodes (\_root->left);  if (\_root->right)deleteNodes (\_root->right);  delete \_root;  }  }  void clear () {  if (\_root) {  if (\_root->left)deleteNodes (\_root->left);  if (\_root->right)deleteNodes (\_root->right);  delete \_root;  }  }  void buildFromPreIn (T\* pre\_head\_in, T\* in\_head\_in, int length\_in) {  \_root = \_makeNodeFromPreIn (pre\_head\_in, in\_head\_in, length\_in);  }  ostream& preOut (ostream& out) {  if (\_root == nullptr) {  return out;  } else {  out << \_root->data << " ";  \_preOut (out, \_root->left);  \_preOut (out, \_root->right);  return out;  }  }  ostream& postOut (ostream& out) {  if (\_root == nullptr) {  return out;  } else {  \_postOut (out, \_root->left);  \_postOut (out, \_root->right);  out << \_root->data << " ";  return out;  }  }  ostream& midOut (ostream& out) {  if (\_root == nullptr) {  return out;  } else {  \_midOut (out, \_root->left);  out << \_root->data << " ";  \_midOut (out, \_root->right);  return out;  }  }  ostream& levelOut (ostream& out) {  queue<node\* > q;  node\* t = \_root;  q.push (t);  //通过队列存储待打印元素，这样一层的数据会相邻在一起  while (!q.empty ()) {  t = q.front ();  q.pop ();  out << t->data << " ";  if (t->left != nullptr) {  q.push (t->left);  }  if (t->right != nullptr) {  q.push (t->right);  }  }  return out;  }  void setRoot (node\* rootin) {  clear ();  \_root = rootin;  return;  }  };  文件2 queue.h  /\*queue  public:  enum queue\_err { queue\_empty }; //常见的错误  private:  struct node; //结点类型  node\* \_head; //头结点指针  node\* \_end; //尾结点指针，指向NULL  int \_length; //元素个数  public:  queue (); //构造函数  ~queue (); //析构函数  void push ( const T& in ); //入队列  T front (); //获得首元素  void pop (); //弹出首元素  bool empty ()const; //是否为空  int size ()const; //获取元素个数  \*/  template<typename T>  class queue {  public:  enum queue\_err { queue\_empty };  private:  typedef struct node {  T data;  node\* next;  node () { next = nullptr; }  }node;  node\* \_head;  node\* \_end;  int \_length;  public:  queue () {  \_head = new node;  \_end = \_head;  \_length = 0;  }  ~queue () {  while (\_head->next != NULL) {  node\* temp = \_head;  \_head = \_head->next;  delete temp;  }  delete \_head;  }  void push (const T& in) {  \_length++;  node\* n\_end = new node;  n\_end->data = in;  n\_end->next = NULL;  \_end->next = n\_end;  \_end = n\_end;  }  T front () {  if (empty ()) {  throw queue\_empty;  }  return \_head->next->data;  }  void pop () {  if (empty ()) {  throw queue\_empty;  }  node\* n\_head = \_head->next;  delete \_head;  \_head = n\_head;  \_length--;  return;  }  bool empty ()const { return \_head == \_end; }  int size ()const { return \_length; }  };  文件3 main.cpp  #include<iostream>  #include"btree.h"  #define max(a,b) (a<b ? b:a)//用于获取左右子树中最大的那个层数  using namespace std;  /\*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\*/  void cal (btree<int>::node\* 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 main () {  #pragma warning(disable:4996)  freopen ("input.txt", "r", stdin);  int num = 0;  cin >> num;  /\*初始化存储节点数与层数的数组\*/  int\* sizes = new int[num + 1];  memset (sizes, 0, num + 1);  int\* deepthes = new int[num + 1];  memset (deepthes, 0, num + 1);  /\*初始化众节点\*/  btree<int> a;  btree<int>::node\*\* nodes = new btree<int>::node \* [num + 1];  for (int i = 1; i <= num; i++) {  nodes[i] = new btree<int>::node;  }  for (int i = 1; i <= num; i++) {  int l = 0;  int r = 0;  cin >> l >> r;  nodes[i]->data = i;  nodes[i]->left = l == -1 ? nullptr : nodes[l];  nodes[i]->right = r == -1 ? nullptr : nodes[r];  sizes[i] = 1;  deepthes[i] = 1;  }  a.setRoot (nodes[1]);  a.levelOut (cout);  cout << "\n";  /\*计算deepths与sizes\*/  cal (nodes[1], sizes, deepthes);  for (int i = 1; i <= num; i++) {  cout << sizes[i] << " ";  }  cout << "\n";  for (int i = 1; i <= num; i++) {  cout << deepthes[i] << " ";  }  cout << "\n";  cout << "\nmake tree from pre and in\n";  char pre[] = { 'a','b','c','d','e','f','g','h' };  char in[] = { 'c','d','b','a','g','f','e','h' };  btree<char> b;  b.buildFromPreIn (pre, in, 8);  b.postOut (cout);  return 0;  return 0;  } | | | |