山东大学 软件 学院

数据结构 课程实验报告

|  |  |  |  |
| --- | --- | --- | --- |
| 学号：201400301034 | 姓名： 石兴帮 | | 班级： 软件八班 |
| 实验题目：二叉树的操作 | | | |
| 实验学时：4h | | 实验日期： 2015-12-13 | |
| 实验目的：  掌握二叉树的基本概念，链表描述方法；遍历方法 | | | |
| 硬件环境：  MacBook Pro  OS X Yosemite 10.10.3 | | | |
| 软件环境：  Xcode 6.4 | | | |
| 实验内容与设计：   1. 实验内容（题目内容，输入要求，输出要求）     1.创建二叉树类。二叉树的存储结构使用链表。  2.提供操作：前序遍历，中序遍历，后序遍历，层次遍历，计算二叉树节点数目，计算二叉树高度。  3.对建立好的二叉树，执行上述各操作。  4.接受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）  创建二叉树类：节点类，一个二叉树节点中包含本身元素，两个子节点。二叉树类包含一个根节点，以及MakeTree，BreekTree，PreOrder，InOrder，PostOrder等方法。  前序遍历：先访问节点，再分别遍历左节点和右节点。  中序遍历：先遍历左节点，再访问节点，再遍历右节点。  后序遍历：先访问左节点，再访问右节点，再访问节点。  计算二叉树节点树木：遍历时count++；  计算二叉树高度：递归地统计左右节点高度，取其大者。  接受前序和中序，输出后序：在中序中找到前序的第一个元素，由其作为新树根节点，左右部分分别用于构建左右子树，在前序中找到相应部分，用同样方法递归构造。  3.测试结果（测试输入，测试输出）    **前序输出:**  **1 2 4 8 9 5 10 11 3 6 12 13 7 14 15**  **中序输出:**  **8 4 9 2 10 5 11 1 12 6 13 3 14 7 15**  **后序输出:**  **8 9 4 10 11 5 2 12 13 6 14 15 7 3 1**  **层次遍历:**  **1 2 3 4 5 6 7 8 9 10 11 12 13 14 15**  **树的高度:**  **4**  **树的节点数:**  **15**  **输入序列长度:**  4  **输入前序:**  1 2 3 4  **输入中序:**  2 3 1 4  **后序如下:**  **3 2 4 1**    4.实现源代码（程序风格清晰易理解，有充分的注释）  //  //  shiyan6.h  //  c++初体验  //  //  Created by SXB on 15/11/21.  //  Copyright (c) 2015年 apple. All rights reserved.  //  #ifndef c\_\_\_\_\_\_shiyan6\_h  #define c\_\_\_\_\_\_shiyan6\_h  #include<iostream>  #include<stdlib.h>  using namespace std;  static int xulieSize;  template<class T>  struct Node {      T data;      Node \*link;  };  template<class T>  class LinkedQueue {  //    FIFO 对象  public:      LinkedQueue() {front = rear = 0;} //  构造函数      ~LinkedQueue(); //  析构函数      bool IsEmpty() const      {          return ((front) ? false : true);      }      bool IsFull() const;      T First() const;                //  返回第一个元素      T Last() const;                 //  返回最后一个元素      LinkedQueue<T> &Add(const T& x );      LinkedQueue<T> &Delete( T& x );    private:      Node<T> \*front;                 //  指向第一个节点      Node<T> \*rear;                  //  指向最后一个节点  };  template<class T>  LinkedQueue<T>::~LinkedQueue() {  //    队列析构函数，删除所有节点      Node<T> \*next;      while (front) {          next = front->link;          delete front;          front = next;      }  }  template<class T>  bool LinkedQueue<T>::IsFull() const  {  //    判断队列是否已满      Node<T> \*p;  //    链表怎么会满呢。。直接return true 好了。。      return true;  }  template<class T>  T LinkedQueue<T>::First() const  {  //    返回队列的第一个元素  //    如果队列为空， 则引发异常。烦死了。      return front->data;  }  template<class T>  T LinkedQueue<T>::Last() const  {  //    返回队列的最后一个元素  //    如果队列为空，则爆炸      return rear->data;  }  template<class T>  LinkedQueue<T>& LinkedQueue<T>::Add(const T& x )  {  //    把 x添加到队列的尾部  //    不捕获可能由new引发的NoMem 异常    //    为新元素创建链表节点      Node<T> \*p = new Node<T>;      p->data = x;      p->link = 0;    //    在队列尾部添加新节点      if (front) {          rear->link = p;     //  队列不为空      } else {          front = p;          //  队列为空      }      rear = p;        return \*this;  }  template<class T>  LinkedQueue<T>& LinkedQueue<T>::Delete(T& x )  {  //    删除第一个元素，并将其放入 x  //    如果队列为空， 则引发异常OutOfBounds        if (IsEmpty()) {  //        。      }    //    保存第一个节点中的元素      x = front->data;    //    删除第一个节点      Node<T> \*p = front;      front = front->link;      delete p;        return \*this;  }  template<class T>  class BinaryTreeNode {      friend void Visit(BinaryTreeNode<T> \*);      friend void InOrder(BinaryTreeNode<T> \*);      friend void PreOrder(BinaryTreeNode<T> \*);      friend void PostOrder(BinaryTreeNode<T> \*);      friend void LevelOrder(BinaryTreeNode<T> \*);      friend int main();    public:      BinaryTreeNode() {LeftChild = RightChild = 0;}      BinaryTreeNode(const T& e)      {          data = e;          LeftChild = RightChild = 0;      }      BinaryTreeNode(const T& e, BinaryTreeNode \*l, BinaryTreeNode \*r )      {          data = e;          LeftChild = l;          RightChild = r;      }        BinaryTreeNode<T>& copy() const;  //private:      T data;      BinaryTreeNode<T> \*LeftChild, \*RightChild;  };  template<class T>  void Infix(BinaryTreeNode<T> \*t )  {  //    输出表达式的中缀形式      if (t) {          cout << '(';          Infix(t->LeftChild);        //  左操作数          cout << t->data;            //  操作符          Infix(t->RightChild);       //  右操作数          cout << ')';                //      }  }  template<class T>  class BinaryTree {  public:      BinaryTree() { root = 0; }      ~BinaryTree() {};      bool IsEmpty() const {          return ((root) ? false : true);      }      bool Root(T& x ) const;      void MakeTree(const T& element, BinaryTree<T>& left, BinaryTree<T>& right );      void BreakTree(T& element, BinaryTree<T>& left, BinaryTree<T>& right );      void PreOrder(void(\*Visit) (BinaryTreeNode<T> \*u ))      {          PreOrder(Visit, root );      }      void InOrder(void(\*Visit) (BinaryTreeNode<T> \*u ))      {          InOrder(Visit, root );      }      void PostOrder(void(\*Visit) (BinaryTreeNode<T> \*u ))      {          PostOrder(Visit, root );      }      void LevelOrder(void(\*Visit) (BinaryTreeNode<T> \*u ));      int Height(BinaryTreeNode<T> \*t ) const;        BinaryTreeNode<T> \*root;      int treeSize=0;      void PreOrder(void(\*Visit)(BinaryTreeNode<T> \*u ), BinaryTreeNode<T> \*t );      void InOrder(void(\*Visit)(BinaryTreeNode<T> \*u ), BinaryTreeNode<T> \*t );      void PostOrder(void(\*Visit)(BinaryTreeNode<T> \*u ), BinaryTreeNode<T> \*t );        bool compareTree(BinaryTree<T> \*X );      bool compareTree(BinaryTreeNode<T> \*x1, BinaryTreeNode<T> \*x2);  };  template<class T>  bool BinaryTree<T>::Root(T& x ) const  {  //    取根节点的data 域， 放入x  //    如果没有根节点， 则返回false      if (root ) {          x = root->data;          return true;      }      else return false;  //  没有根节点  }  template<class T>  void BinaryTree<T>::MakeTree(const T& element, BinaryTree<T>& left, BinaryTree<T>& right)  {  //    将left，right 和 element 合并成一棵新树  //    left,right 和 this必须是不同的树  //    创建新树      root = new BinaryTreeNode<T>(element, left.root, right.root );    //    阻止访问left 和 right      left.root = right.root = 0;  }  template<class T>  void BinaryTree<T>::BreakTree(T& element, BinaryTree<T>& left, BinaryTree<T>& right)  {  //    left，right和this必须是不同的树  //    检查树是否为空      if (!root) {          return;      }    //    分解树      element = root->data;      left.root = root->LeftChild;      right.root = root->RightChild;        delete root;      root = 0;  }  template<class T>  void BinaryTree<T>::PreOrder(void(\*Visit)(BinaryTreeNode<T> \*u ), BinaryTreeNode<T> \*t)  {  //    前序遍历      if (t) {          Visit(t);          treeSize++;          PreOrder(Visit, t->LeftChild);          PreOrder(Visit, t->RightChild);      }  }  template<class T>  void BinaryTree<T>::InOrder(void(\*Visit)(BinaryTreeNode<T> \*u ), BinaryTreeNode<T> \*t)  {  //    中序遍历      if (t) {          InOrder(Visit, t->LeftChild);          Visit(t);  //        treeSize++;          InOrder(Visit, t->RightChild);      }  }  template<class T>  void BinaryTree<T>::PostOrder(void(\*Visit)(BinaryTreeNode<T> \*u ), BinaryTreeNode<T> \*t )  {  //    后序遍历      if (t) {          PostOrder(Visit, t->LeftChild);          PostOrder(Visit, t->RightChild);          Visit(t);  //        treeSize++;      }  }  template<class T>  void BinaryTree<T>::LevelOrder(void(\*Visit)(BinaryTreeNode<T> \*u ))  {  //    逐层遍历      LinkedQueue<BinaryTreeNode<T> \*> Q;      BinaryTreeNode<T> \*t;      t = root;      while (t ) {          Visit(t);  //        treeSize++;          if (t->LeftChild) {              Q.Add(t->LeftChild);          }          if (t->RightChild) {              Q.Add(t->RightChild);          }            if (Q.IsEmpty()) {              return ;          }          Q.Delete(t);      }  }  template<class T>  int BinaryTree<T>::Height(BinaryTreeNode<T> \*t ) const  {  //    返回树 \*t的高度      if(!t ) return 0;      int hl = Height(t->LeftChild);      int hr = Height(t->RightChild);      if (hl > hr) {          return ++hl;      } else          return ++hr;  }  template<class T>  BinaryTreeNode<T> \*createBinaryTreeNode(T e) {      BinaryTreeNode<T> \*nd = new BinaryTreeNode<T>;      nd->data = e;      nd->LeftChild = NULL;      nd->RightChild = NULL;      return nd;  }  template<class T>  int findData(const T \*str, int s1, int s2, T t ) {      if (!str || s2<s1 || s1<0 || s2>xulieSize-1) {          return -1;      }      for (int i=s1; i <= s2; i++) {          if (str[i] == t ) {              return i;          }      }      return -1;  }  template<class T>  BinaryTreeNode<T> \*getRoot(T \*pre, int p1, int p2, T \*inn, int i1, int i2) {      T roott = pre[p1];      if (!pre || p2<p1 || p1<0 || p2>=xulieSize || !inn || i2<i1 || i1<0 || i2>=xulieSize) {          return NULL;      }      int tmp = findData(inn, i1, i2, roott);      if (tmp < 0) {          return NULL;      }      BinaryTreeNode<T> \*nd = createBinaryTreeNode(roott);      nd->LeftChild = getRoot(pre, p1+1, p1+tmp-i1, inn, i1, tmp-1);      nd->RightChild = getRoot(pre, p1+tmp-i1+1, p2, inn, tmp+1, i2);      return nd;  }  template<class T>  void printPostOrder(BinaryTreeNode<T> \*t){      if (!t) {          return;      }      printPostOrder(t->LeftChild);      printPostOrder(t->RightChild);      cout << t->data << " ";  }  template<class T>  void ct(BinaryTreeNode<T> \*t)  {      if (t) {          cout << t->data << " ";      }  }  #endif  //  MARK:  main  //#include "shiyan6.h"  //  //int count = 0;  //BinaryTree<int> a,x,y,z,m,n;  //  //  //int main()  //{  //    x.MakeTree(8, a , a );  //    y.MakeTree(9, a , a );  //    z.MakeTree(4 ,x , y );  //    x.MakeTree(10, a , a);  //    y.MakeTree(11, a , a);  //    m.MakeTree(5, x , y );  //    x.MakeTree(2, z , m );  //    z.MakeTree(12, a , a);  //    y.MakeTree(13, a , a);  //    m.MakeTree(6, z , y );  //    z.MakeTree(14, a , a);  //    y.MakeTree(15, a , a);  //    n.MakeTree(7, z , y );  //    y.MakeTree(3, m , n );  //    z.MakeTree(1, x , y );  //  //  //  //    cout << "前序输出: " << endl;  //    z.PreOrder(ct );  //    cout << endl;  //  //    cout << "中序输出: " << endl;  //    z.InOrder(ct );  //    cout << endl;  //  //    cout << "后序输出: " << endl;  //    z.PostOrder(ct );  //    cout << endl;  //  //    cout << "层次遍历: " << endl;  //    z.LevelOrder(ct );  //    cout << endl;  //  //    cout << "树的高度: " << endl;  //    cout << z.Height(z.root ) << endl;;  //  //    cout << "树的节点数: " << endl;  //    cout << z.treeSize << endl;  //  //    cout << "输入序列长度: " << endl;  //    //    int xulieSize;  //    cin >> xulieSize;  //  //    //    int qianxu[xulieSize];  //    //    int zhongxu[xulieSize];  //  //    int \*qianxu, \*zhongxu;  //    qianxu = new int[xulieSize];  //    zhongxu = new int[xulieSize];  //  //    cout << "输入前序: " << endl;  //    for (int i=0; i<xulieSize; i++) {  //        cin >> qianxu[i];  //    }  //    cout << "输入中序: " << endl;  //    for (int i=0; i<xulieSize; i++) {  //        cin >> zhongxu[i];  //    }  //  //    BinaryTreeNode<int> \*bTree;  //    bTree = getRoot(qianxu, 0, xulieSize-1, zhongxu, 0, xulieSize-1);  //    cout << "后序如下: " << endl;  //    printPostOrder(bTree);  //  //} | | | |
| 结论分析与体会：  自己设计算法然后debug确实很好玩 | | | |