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：201700130009 | 姓名： 张愈博 | | 班级： 计科17.3 |
| 实验题目：二叉树操作 | | | |
| 实验学时：2 | | 实验日期： 2018.11.22 | |
| 实验目的：  掌握二叉树的基本概念，链表描述方法；二叉树操作的实现。 | | | |
| 软件环境：  Windows 10  Dev C++ 5.11 | | | |
| 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。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。   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）   (1)针对前两小题，只需创建二叉树类即可完成。题目中要求的操作基本都使用递归函数实现，其思想是把根节点、左子树和右子树分开处理，最后再合并结果返回即可。所以函数结构都大同小异。层次遍历操作较为特殊，它需要按从左到右、从根节点由近到远地访问节点，这个巧妙的思想可以由队列结构来实现。首先把根节点压入队列，当队列的首节点不为空时依次压入它的左右节点，然后把该节点弹出访问下一个节点，直至访问完所有元素（队列为空）  (2)针对第3小题，函数void changeIntoPost(T \*pre,T \*in,T \*post,const int n)依旧采用递归思想，我们首先找到根节点元素并把它复制到post的末位，接着搜索root在in中的位置，然后可以以此计算出左右子树的长度，接着对左右子树依次递归，直到子树长度为0为止。此函数也可以声明为静态函数。   1. 测试结果（测试输入，测试输出，结果分析）     如图所示，给出了要求的各种操作，且经验证输出均正确。  但是利用函数指针的二叉树由于链接错误受限于编译器暂未实现，不过在同学的visual studio 2017环境下可以正常运行。   1. 分析与探讨（结果分析，若存在问题，探讨解决问题的途径）   (1)在实验中，有许多方法需要传入参数是根节点，但是由于根节点是二叉树类的私有变量，在类外不可引用，为了解决这个方法，我们可以重写一个专用的无参对外接口函数，然后把参数是根节点的计算函数声明为私有并在接口函数用调用，这样既保证了数据的安全性又保证了用户友好性。  (2)教材上所给出的代码是利用函数指针实现各种遍历删除等操作，这样写的好处是部分函数可以反复利用，提高通用性，这对于我们的模板类来说非常重要。比如postOrder函数可以用来进行遍历或者删除操作，因为visit可以指向不同的函数。但是在本实验中也给我们带来了麻烦，比如一个简单的前序遍历操作就需要我们写preOutput, preOrder和static preOrder三个函数。   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）   Main.cpp  #include<iostream>  #include"二叉树.h"  //#include"二叉树(函数指针版).h"  using namespace std;  int main()  {  char \*pre="ABHFDECKG";  char \*in="HBDFAEKCG";  linkedBinaryTree<char> BT;  BT.makeTree(pre,in,9);    cout<<"preOrder is:"; BT.preOrder(); cout<<endl;  cout<<"inOrder is:"; BT.inOrder(); cout<<endl;  cout<<"postOrder is:"; BT.postOrder(); cout<<endl;  cout<<"levelOrder is:"; BT.levelOrder(); cout<<endl;  cout<<"tree size is:"<<BT.size(); cout<<endl;  cout<<"tree height is:"<<BT.height(); cout<<endl;    char \*post=new char[10];  linkedBinaryTree<char>::changeIntoPost(pre,in,post,9);  post[9]='\0';  cout<<"由前后序转来的后序序列为："<<post<<endl;  return 0;  }  二叉树.h  #include<iostream>  #include"arrayQueue.h"  #include<string.h>  using namespace std;  template<class T>  class binaryTreeNode  {  public:  T element; //节点中的元素  binaryTreeNode<T> \*left,\*right; //左右子树    binaryTreeNode() //无参数  { left=right=NULL; }  binaryTreeNode(const T& ele)  {//只有数据参数  element=ele;  left=right=NULL;  }  binaryTreeNode(const T& ele,binaryTreeNode\* theLeft,binaryTreeNode\* theRight)  {//数据加指针  element=ele;  left=theLeft;  right=theRight;  }  };  template<class T>  class linkedBinaryTree  {  public:  linkedBinaryTree()  {  root=NULL;  treeSize=0;  }  ~linkedBinaryTree()  {  //erase();  }  bool empty() const  {  return treeSize;  }  void preOrder()  {  preOrder(root); //这里调用的是静态成员函数  }  void inOrder()  {  inOrder(root);  }  void postOrder()  {  postOrder(root);  }  void makeTree(const T& element,linkedBinaryTree<T>& left,linkedBinaryTree<T> &right)  {//将left,right和element合并成一颗新树  //left，right和this必须不同！！    root=new binaryTreeNode<T>(element,left.root,right.root);  treeSize=left.treeSize+right.treeSize+1;    //阻止访问left和right  left.root=right.root=NULL;  left.treeSize=right.treeSize=0;  }  Void makeTree(char \*pre,char \*in,int n)  {  root=Sub\_makeTree(pre,in,n);  // cout<<"this is "<<root<<endl;  }  int height()  {  return Sub\_height(root);  }  int size()  {  //return treeSize;  return size(root);  }  int size(binaryTreeNode<T> \*t)  {//返回根为\*t的树的节点数。对某根点求它左右子树的元素数，相加再加上它本身即为总节点数。  if(t==NULL) return 0;  else  {  int ls=size(t->left); //左子树元素数  int rs=size(t->right); //右子树元素数  return ls+rs+1; //加上根节点  }  //复杂度O(n)  }  void levelOrder()  {  Sub\_levelOrder(root);  }  int max\_width(binaryTreeNode<T> \*t)  {//求二叉树的每层最大节点数  arrayQueue< binaryTreeNode<T>\* > q;  binaryTreeNode<T> \*last=root; //last指向每层最右端的元素  int level=1; //层数  int width[20]; //保存每层节点数  while(t!=NULL)  {  if(t->left!=NULL)  q.push(t->left.element);  if(t->right!=NULL)  q.push(t->right.element);    //下一个t  if(q.empty()==true) break; //队列为空，结束循环  t=q.front();  q.pop();  if(t==last)  {//已经扫完当前层,此时队列里全是下一层的节点  width[++level]=q.size();  last=q.back(); //更新last  }  }  //从层数组中找出最大节点数  int max=0;  for(int i=1;i<level;i++)  {  if(width[i]>max)  max=width[i];  }    return max;  }  static bool compare(binaryTreeNode<T> \*root1,binaryTreeNode<T> \*root2)  {//先判断两棵树根节点是否有空，若均不为空就比较其内的元素值  //再判断左右子树 ------O(N)  if( (root1==NULL&&root2!=NULL) || (root1!=NULL&&root2==NULL) )  return false; //树的结构不一样  else  {  if(root1==NULL && root2==NULL) return true; //根节点均为空，递归到终点  if( \*(root1) != \*(root2) ) return false; //根节点均不为空且值不同  else  {//根节点不为空且值相同，比较左右子树  bool lt=compare(root1->left,root2->left);  bool rt=compare(root1->right,root2->right);  return lt&&rt;  }  }  }  static void changeIntoPost(T \*pre,T \*in,T \*post,const int n)  {  if(n<1) return;  T root=pre[0];  post[n-1]=root;  T \*p;  int mid; //mid是root在in中的索引    for(p=in;p<in+n;p++)  {  if( \*(p)==root )  break;  }  mid=p-in;  changeIntoPost(pre+1,in,post,mid);  changeIntoPost(pre+mid+1,p+1,post+mid,n-1-mid);  }  void func(binaryTreeNode<T> \*X)  {    if(X->left==NULL) cout<<X->element<<"'s left is null"<<endl;  else{  cout<<X->element<<"'s left is not null"<<endl;  func(X->left);  }    if(X->right==NULL) cout<<X->element<<"'s right is null"<<endl;  else{  cout<<X->element<<"'s right is not null"<<endl;  func(X->right);  }  }  private:  binaryTreeNode<T> \*root; //指向根节点的指针  int treeSize; //树的节点数  static void visit(binaryTreeNode<T> \*X)  {//访问函数  cout<< X->element<<' ';  }  void preOrder(binaryTreeNode<T> \*t)  {//前序遍历  if(t!=NULL)  {  visit(t);  preOrder(t->left);  preOrder(t->right);  }  }  static void inOrder(binaryTreeNode<T> \*t)  {//中序遍历  if(t!=NULL)  {  inOrder(t->left);  visit(t);  inOrder(t->right);  }  }  static void postOrder(binaryTreeNode<T> \*t)  {//后序遍历  if(t!=NULL)  {  postOrder(t->left);  postOrder(t->right);  visit(t);  }  }  static void dispose(binaryTreeNode<T> \*t)  {  delete t;  }  int Sub\_height(binaryTreeNode<T> \*t)  {//返回根为\*t的树的高度。对根节点求它的左右子树的高度，取大者作为总树的高度  if(t==NULL)  return 0;  else  {  int hl=Sub\_height(t->left);  int hr=Sub\_height(t->right);  if(hl>hr)  return ++hl;  else  return ++hr;  }  //复杂度O(n)，每一个元素都比较了一遍  }  binaryTreeNode<T> \*Sub\_makeTree(char \*pre,char \*in,const int n)  {//由前序和中序序列构造一颗二叉树，n是二叉树的节点总数  if(n<1) return NULL;  char Root=pre[0]; //cout<<"Root="<<Root<<"\t";  binaryTreeNode<T> \*b=new binaryTreeNode<T>(Root);  treeSize++;    char \*p; int mid=0; //根节点在in中的索引  for(p=in;p<in+n;p++)  {//在中序序列中找到根节点  if( (\*p)==Root )  break;  mid++;  }  //cout<<"mid="<<mid<<endl;  b->left=Sub\_makeTree(pre+1,in,mid); //对左子树递归  b->right=Sub\_makeTree(pre+1+mid,in+mid+1,n-mid-1); //对右子树递归    return b;  }  void Sub\_levelOrder(binaryTreeNode<T> \*t)  {//层次遍历二叉树\*t  arrayQueue< binaryTreeNode<T>\* > q; //这里先进先出所以用队列  while(t!=NULL)  {  visit(t); //访问 t    //将t的孩子插入队列，先左后右  if(t->left!=NULL)  q.push(t->left);  if(t->right!=NULL)  q.push(t->right);    //访问下一个节点  if(q.empty()==true) return; //已访问完所有元素  t=q.front();  q.pop();  }  }    };  arrayQueue.h  #pragma once  #include<iostream>  #include<cstdlib>  #include<algorithm>  template<class T>  class arrayQueue  {  public:  arrayQueue(int initialCapacity = 10)  {//构造函数  if(initialCapacity<0) std::cerr<<"队列长度必须大于0！"<<std::endl;  else{  Queue=new T[initialCapacity];  arrayLength=initialCapacity;  qFront=qBack=0; //这里是从Queue[1]开始插入元素  }  }  ~arrayQueue() {delete [] Queue;}  bool empty() const  {  if(qFront==qBack) return true;  else return false;  }  int size() const  {  return (arrayLength+qBack-qFront)%arrayLength;  }  T& front()  {  if(empty()!=true)  return Queue[(qFront+1)%arrayLength];  else  { std::cerr<<"队列为空"<<std::endl; exit(1); }  }  T& back()  {  if(empty()!=true)  return Queue[qBack];  else  { std::cerr<<"队列为空"<<std::endl; exit(1); }  }  T pop()  {//从队首删除元素  T \*p=&front(); //这里已经判断了队列是否为空  T temp=(\*p);  qFront=(qFront+1)%arrayLength;  (\*p).~T(); //析构首元素好像不能表示int的删除...，就是无法恢复到初始化以前的状态  return temp;  }  void push(const T& ele)  {//从队尾添加元素  if( (qBack+1)%arrayLength==qFront )  {//队列将满，加倍数组长度  T \*newQueue=new T[2\*arrayLength];  int start=(qFront+1)%arrayLength;  if(start==0||start==1)  {//未形成环  std::copy(Queue+start,Queue+qBack+1,newQueue);  }  else  {//形成了环  std::copy(Queue+start,Queue+arrayLength,newQueue);  //复制第2段(start,队列末端，新队列起点）  std::copy(Queue,Queue+qBack+1,newQueue+(arrayLength-start));  //复制第1段（原队列首端，qback,新队列第arraylength-start个位置）  }  qFront=(arrayLength)\*2-1;  qBack=arrayLength-1-1; //重新设置首尾游标  arrayLength=arrayLength\*2;  delete [] Queue;  Queue=newQueue;  }  //把元素插入队列的尾部  qBack=(qBack+1)%arrayLength;  Queue[qBack]=ele;    }  void output()  {  for(int i=qFront;i<qBack;i++)  std::cout<<Queue[i];  std::cout<<std::endl;  }  private:  int qFront; //队列中第一个元素的前一个未知  int qBack; //队列最后一个元素的位置  int arrayLength; //队列的容量  T \*Queue; //队列元素  };  二叉树（函数指针版）.h  #pragma once  #include<iostream>  #include"arrayQueue.h"  #include<string.h>  using namespace std;  template<class T>  class binaryTreeNode  {  public:  T element; //节点中的元素  binaryTreeNode<T> \*left,\*right; //左右子树    binaryTreeNode() //无参数  { left=right=NULL; }  binaryTreeNode(const T& ele)  {//只有数据参数  element=ele;  left=right=NULL;  }  binaryTreeNode(const T& ele,binaryTreeNode\* theLeft,binaryTreeNode\* theRight)  {//数据加指针  element=ele;  left=theLeft;  right=theRight;  }  };  template<class T>  class linkedBinaryTree  {  public:  linkedBinaryTree()  {  root=NULL;  treeSize=0;  }  ~linkedBinaryTree()  {  //erase();  }  bool empty() const  {  return treeSize;  }  void preOutput()  {  preOrder(&linkedBinaryTree<T>::output); cout<<endl;  }  void inOutput()  {  inOrder(&linkedBinaryTree<T>::output); cout<<endl;  }  void postOutput()  {  postOrder(&linkedBinaryTree<T>::output); cout<<endl;  }  void preOrder(void (\*theVisit) (binaryTreeNode<T>\*) )  {  visit=theVisit;  preOrder(root); //这里调用的是静态成员函数  }  void inOrder(void (\*theVisit) (binaryTreeNode<T>\*))  {  inOrder(root);  }  void postOrder(void (\*theVisit) (binaryTreeNode<T>\*))  {  postOrder(root);  }  void makeTree(const T& element,linkedBinaryTree<T>& left,linkedBinaryTree<T> &right)  {//将left,right和element合并成一颗新树  //left，right和this必须不同！！    root=new binaryTreeNode<T>(element,left.root,right.root);  treeSize=left.treeSize+right.treeSize+1;    //阻止访问left和right  left.root=right.root=NULL;  left.treeSize=right.treeSize=0;  }  void makeTree(const char \*pre,const char \*in,const int n)  {  root=Sub\_makeTree(pre,in,n);  // cout<<"this is "<<root<<endl;  }  int height()  {  return Sub\_height(root);  }  int size()  {  //return treeSize;  return size(root);  }  int size(binaryTreeNode<T> \*t)  {//返回根为\*t的树的节点数。对某根点求它左右子树的元素数，相加再加上它本身即为总节点数。  if(t==NULL) return 0;  else  {  int ls=size(t->left); //左子树元素数  int rs=size(t->right); //右子树元素数  return ls+rs+1; //加上根节点  }  //复杂度O(n)  }  void levelOrder()  {  Sub\_levelOrder(root);  }  int max\_width(binaryTreeNode<T> \*t)  {//求二叉树的每层最大节点数  arrayQueue< binaryTreeNode<T>\* > q;  binaryTreeNode<T> \*last=root; //last指向每层最右端的元素  int level=1; //层数  int width[20]; //保存每层节点数  while(t!=NULL)  {  if(t->left!=NULL)  q.push(t->left.element);  if(t->right!=NULL)  q.push(t->right.element);    //下一个t  if(q.empty()==true) break; //队列为空，结束循环  t=q.front();  q.pop();  if(t==last)  {//已经扫完当前层,此时队列里全是下一层的节点  width[++level]=q.size();  last=q.back(); //更新last  }  }  //从层数组中找出最大节点数  int max=0;  for(int i=1;i<level;i++)  {  if(width[i]>max)  max=width[i];  }    return max;  }  static bool compare(binaryTreeNode<T> \*root1,binaryTreeNode<T> \*root2)  {//先判断两棵树根节点是否有空，若均不为空就比较其内的元素值  //再判断左右子树 ------O(N)  if( (root1==NULL&&root2!=NULL) || (root1!=NULL&&root2==NULL) )  return false; //树的结构不一样  else  {  if(root1==NULL && root2==NULL) return true; //根节点均为空，递归到终点  if( \*(root1) != \*(root2) ) return false; //根节点均不为空且值不同  else  {//根节点不为空且值相同，比较左右子树  bool lt=compare(root1->left,root2->left);  bool rt=compare(root1->right,root2->right);  return lt&&rt;  }  }  }  static void changeIntoPost(const T \*pre,const T \*in,T \*post,const int n)  {  if(n<1) return;  T root=pre[0];  post[n-1]=root;  int mid; //mid是root在in中的索引    for(mid=0;mid<n;mid++)  {//在中序序列中找到根节点  if(in[mid]==root )  break;  }  changeIntoPost(pre+1,in,post,mid);  changeIntoPost(pre+mid+1,in+mid+1,post+mid,n-1-mid);  }    private:  binaryTreeNode<T> \*root; //指向根节点的指针  int treeSize; //树的节点数  static void output(binaryTreeNode<T> \*X)  {//访问函数  cout<< X->element<<' ';  }  static void (\*visit) (binaryTreeNode<T> \*); //函数指针  static void preOrder(binaryTreeNode<T> \*t)  {//前序遍历  if(t!=NULL)  {  linkedBinaryTree<T>::visit(t);  preOrder(t->left);  preOrder(t->right);  }  }  static void inOrder(binaryTreeNode<T> \*t)  {//中序遍历  if(t!=NULL)  {  linkedBinaryTree<T>::inOrder(t->left);  visit(t);  inOrder(t->right);  }  }  static void postOrder(binaryTreeNode<T> \*t)  {//后序遍历  if(t!=NULL)  {  postOrder(t->left);  postOrder(t->right);  linkedBinaryTree<T>::visit(t);  }  }  static void dispose(binaryTreeNode<T> \*t)  {  delete t;  }  int Sub\_height(binaryTreeNode<T> \*t)  {//返回根为\*t的树的高度。对根节点求它的左右子树的高度，取大者作为总树的高度  if(t==NULL)  return 0;  else  {  int hl=Sub\_height(t->left);  int hr=Sub\_height(t->right);  if(hl>hr)  return ++hl;  else  return ++hr;  }  //复杂度O(n)，每一个元素都比较了一遍  }  binaryTreeNode<T> \*Sub\_makeTree(const char \*pre,const char \*in,const int n)  {//由前序和中序序列构造一颗二叉树，n是二叉树的节点总数  if(n<1) return NULL;  char Root=pre[0]; cout<<"Root="<<Root<<"\t";  binaryTreeNode<T> \*b=new binaryTreeNode<T>(Root);  treeSize++;    int mid=0; //根节点在in中的索引  for(mid=0;mid<n;mid++)  {//在中序序列中找到根节点  if(in[mid]==Root )  break;  }  cout<<"mid="<<mid<<endl;  b->left=Sub\_makeTree(pre+1,in,mid); //对左子树递归  b->right=Sub\_makeTree(pre+1+mid,in+mid+1,n-mid-1); //对右子树递归    return b;  }  void Sub\_levelOrder(binaryTreeNode<T> \*t)  {//层次遍历二叉树\*t  arrayQueue< binaryTreeNode<T>\* > q; //这里先进先出所以用队列  while(t!=NULL)  {  output(t); //访问 t    //将t的孩子插入队列，先左后右  if(t->left!=NULL)  q.push(t->left);  if(t->right!=NULL)  q.push(t->right);    //访问下一个节点  if(q.empty()==true) return; //已访问完所有元素  t=q.front();  q.pop();  }  }    };  template<> void(\*(linkedBinaryTree<char> ::visit))(binaryTreeNode<char>\*); | | | |