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：201900161140 | 姓名： 张文浩 | | 班级： 19级人工智能 |
| 实验题目：二叉树操作 | | | |
| 实验学时：2 | | 实验日期： 10.3 | |
| 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。 | | | |
| 软件开发工具：  Vscode | | | |
| 1. 实验内容   **题目描述**：  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。  **输入输出格式**：  **输入**：  第一行为一个数字n (10<=n<=100000)，表示有这棵树有n个节点，编号为1~n。之后n行每行两个数字，第 i 行的两个数字a、b表示编号为 i 的节点的左孩子节点为 a，右孩子节点为 b，-1表示该位置没有节点。保证数据有效，根节点为1。  **输出**：  第一行，n个数字，表示该树的层次遍历。  第二行，n个数字，第i个数字表示以 i 节点为根的子树的节点数目。  第三行，n个数字，第i个数字表示以 i 节点为根的子树的高度。   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.**  为实现层次遍历的操作，要使用队列，自定义一个Queue的类，定义方式与前面那个实现队列的实验相同。  抽象数据类型binaryTree  定义节点结构体binaryTreeNode  定义binaryTree的派生类linkedBinaryTree,节点类型是bianryTreeNode。  成员变量包括\*root指向根节点的指针，treeSize数的节点个数，visit访问函数，以及一些静态函数，包括preOrder先序遍历，inOrder中序遍历，postOrder后序遍历，dispose删除节点函数。  定义linkedBinaryTree的方法函数，包括preOrder先序遍历，inOrder中序遍历，postOrder后序遍历，levelOrder层次遍历，erase删除所有节点函数。和根据先序、中序、后序层次遍历输出二叉树的方法。还有makeTree构建树的函数。  preOrder先序遍历：  递归实现，每次先visit根节点，再visit左子树，再visit右子树。  inOrder中序遍历：  递归实现，每次先visit左子树，再visit根节点，再visit右子树  postOrder后序遍历：  每次先visit左子树，再visit右子树，再visit根节点  levelOrder层次遍历：  每次先visit根节点，再将左子树入队，再将右子树入队，再从队列中获得下一个要处理的节点。  Height(root)求以root为根节点的二叉树的高度，递归实现，当前root根节点的树高总是等于左右子树树高的最大值+1，递归的终止条件是左右子树都为空，即叶子结点。叶子节点的树高就为1.   1. 分析与探讨（结果分析，若存在问题，探讨解决问题的途径）   通过本次实验，我实现了二叉树的定义以及包括前序遍历、中序遍历、后序遍历、层次遍历等方法函数。在定义linkedBinaryTree的时候用到了static定义静态函数，了解了静态函数的性质和使用方式，static函数与普通函数作用域不同,仅在本文件。只在当前源文件中使用的函数应该说明为内部函数(static修饰的函数)，内部函数应该在当前源文件中说明和定义。对于可在当前源文件以外使用的函数，应该在一个头文件中说明，要使用这些函数的源文件要包含这个头文件.  static函数在内存中只有一份，普通函数在每个被调用中维持一份拷贝。  在建立二叉树的时候，将整个树的创建拆分为节点之间的连接会使得问题的解决更加地简单。而在其他操作，如前序遍历、中序遍历、后序遍历、统计子树节点数目、计算子树高度中，使用递归的思想解决问题可以使得代码得到很大的简化。而在题目要求2的输出后序遍历中，要注意到输出元素的顺序：左节点，右节点，根节点，并在函数内部安排好子树的划分。  编写题目1的代码时，我起初是将每个函数分开，实现各自的功能的，但在之后发现重复的部分过多，影响了代码的简介性，于是之后采用了重载函数的方式，通过传入某个特定函数的指针，利用同一段代码为遍历赋予不同的功能。在类中定义函数指针之后，类外也需要再定义一次，否则会出现编译失败的问题。函数指针的写法略微有些复杂，尤其是涉及到在其他函数中的调用时，在网络上查询到写法之后我花费了一段时间才将它看懂。编写题目2的代码时，我最开始是先根据前序、中序遍历的顺序建立好二叉树，再进行后序遍历获得结果的，但之后在csdn上看到了直接输出后序排序序列的方法，也就是直接利用递归获取答案。因此我认为学习过程中信息的查询是非常重要的，可以让我们学到更多有用的知识。   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）   **1.**  #include <iostream>  using namespace std;  //为实现层次遍历的操作，要使用队列，自定义一个Queue的类  template <class T>  class Queue  {  public:      Queue(int initialCapacity = 16)      {          length = initialCapacity;          queue = new T[length];          thefront = 0;          theback = 0;      }      ~Queue() { delete[] queue; }      bool empty() const { return thefront == theback; }      int size() const      {          return (theback - thefront + length) % length;      }      T &front()      {          return queue[(thefront + 1) % length];      }      T &back()      {          return queue[theback];      }      void pop()      {          thefront = (thefront + 1) % length;          queue[thefront].~T();      }      void push(const T &x)      {          //如果需要，则增加数组长度          if ((theback + 1) % length == thefront) //加倍数组长度          {              //分配新的数组空间              T \*newqueue = new T[2 \* length];              //把原数组复制到新数组              int start = (thefront + 1) % length;              if (start < 2)              {                  //没有形成环                  copy(queue + start, queue + start + length - 1, newqueue);              }              else              {                  //队列形成环                  copy(queue + start, queue + length, newqueue);                  copy(queue, queue + theback + 1, newqueue + length - start);              }              //这是新的队列的首和尾的元素位置              thefront = 2 \* length - 1;              theback = length - 2;              length \*= 2;              delete[] queue;              queue = newqueue;          }          theback = (theback + 1) % length;          queue[theback] = x;      }  private:      int thefront;      int theback;      int length;      T \*queue;  };  template <class T>  class binaryTree  {  public:      virtual ~binaryTree() {}      virtual bool empty() const = 0;      virtual int size() const = 0;      virtual void preOrder(void (\*)(T \*)) = 0;      virtual void inOrder(void (\*)(T \*)) = 0;      virtual void postOrder(void (\*)(T \*)) = 0;      virtual void levelOrder(void (\*)(T \*)) = 0;  };  template <class T>  struct BinaryTreeNode  {      T element;      BinaryTreeNode<T> \*leftChild, \*rightChild;      BinaryTreeNode()      {          leftChild = NULL;          right = NULL;      }      BinaryTreeNode(const T &theElement)      {          element = theElement;          leftChild = NULL;          rightChild = NULL;      }      BinaryTreeNode(const T &theElement, BinaryTreeNode \*theLeftChild, BinaryTreeNode \*theRightChild)      {          element = theElement;          leftChild = theLeftChild;          rightChild = theRightChild;      }  };  template <class E>  class linkedBinaryTree : public binaryTree<BinaryTreeNode<E>>  {  public:      linkedBinaryTree()      {          root = NULL;          treeSize = 0;      }      ~linkedBinaryTree() { erase(); }      bool empty() const { return treeSize == 0; }      int size() const { return treeSize; }      void preOrder(void (\*theVisit)(BinaryTreeNode<E> \*))      {          visit = theVisit;          preOrder(root);      }      void inOrder(void (\*theVisit)(BinaryTreeNode<E> \*))      {          visit = theVisit;          inOrder(root);      }      void postOrder(void (\*theVisit)(BinaryTreeNode<E> \*))      {          visit = theVisit;          postOrder(root);      }      void levelOrder(void (\*theVisit)(BinaryTreeNode<E> \*))      {          visit = theVisit;          levelOrder(root);      }      void erase()      {          dispose(root);          root = NULL;          treeSize = 0;      }      int height() const { return height(root); }      void preOrderOutput()      {          preOrder(output);          cout << endl;      }      void inOrderOutput()      {          inOrder(output);          cout << endl;      }      void postOrderOutput()      {          postOrder(output);          cout << endl;      }      void levelOrderOutput()      {          levelOrder(output);          cout << endl;      }      void makeTree(const E &element, linkedBinaryTree<E> &l, linkedBinaryTree<E> &r)      {          root = new BinaryTreeNode<E>(element, l.root, r.root);          treeSize = l.treeSize + r.treeSize + 1;      }  private:      BinaryTreeNode<E> \*root;                   //指向根的指针      int treeSize;                              //数的节点数      static void (\*visit)(BinaryTreeNode<E> \*); //访问函数      static void preOrder(BinaryTreeNode<E> \*t);      static void inOrder(BinaryTreeNode<E> \*t);      static void postOrder(BinaryTreeNode<E> \*t);      static void levelOrder(BinaryTreeNode<E> \*t);      static void dispose(BinaryTreeNode<E> \*t) { delete t; }      static int height(BinaryTreeNode<E> \*t);      static void output(BinaryTreeNode<E> \*t)      {          if (t != NULL)          {              cout << t->element << " ";          }      }  };  template <class T>  void (\*linkedBinaryTree<T>::visit)(BinaryTreeNode<T> \*);  template <class T>  void linkedBinaryTree<T>::preOrder(BinaryTreeNode<T> \*t)  {      if (t != NULL)      {          linkedBinaryTree<T>::visit(t);          preOrder(t->leftChild);          preOrder(t->rightChild);      }  }  template <class T>  void linkedBinaryTree<T>::inOrder(BinaryTreeNode<T> \*t)  {      if (t != NULL)      {          inOrder(t->leftChild);          linkedBinaryTree<T>::visit(t);          inOrder(t->rightChild);      }  }  template <class T>  void linkedBinaryTree<T>::postOrder(BinaryTreeNode<T> \*t)  {      if (t != NULL)      {          postOrder(t->leftChild);          postOrder(t->rightChild);          linkedBinaryTree<T>::visit(t);      }  }  template <class T>  void linkedBinaryTree<T>::levelOrder(BinaryTreeNode<T> \*t)  {      Queue<BinaryTreeNode<T> \*> q;      while (t != NULL)      {          linkedBinaryTree<T>::visit(t);          //将t的孩子插入队列          if (t->leftChild != NULL)              q.push(t->leftChild);          if (t->rightChild != NULL)              q.push(t->rightChild);          //提取下一个要访问的节点          if (q.empty())              return;          t = q.front();          q.pop();      }  }  template <class T>  int linkedBinaryTree<T>::height(BinaryTreeNode<T> \*t) //返回根为\*t的树的高度  {      if (t == NULL)          return 0;                   //空树      int hl = height(t->leftChild);  //左树高      int hr = height(t->rightChild); //右树高      if (hl > hr)          return ++hl;      else          return ++hr;  }  int a[10001], b[10001], h[10001], s[10001];  linkedBinaryTree<int> t[10001];  int main()  {      linkedBinaryTree<int> non;      int n;      cin >> n;      for (int i = 1; i <= n; i++)      {          cin >> a[i] >> b[i];      }      for (int i = n; i > 0; i--)      {          if (a[i] != -1 && b[i] != -1)          {              t[i].makeTree(i, t[a[i]], t[b[i]]);          }          if (a[i] != -1 && b[i] == -1)          {              t[i].makeTree(i, t[a[i]], non);          }          if (a[i] == -1 && b[i] != -1)          {              t[i].makeTree(i, non, t[b[i]]);          }          if (a[i] == -1 && b[i] == -1)          {              t[i].makeTree(i, non, non);          }          h[i]=t[i].height();          s[i]=t[i].size();      }      t[1].levelOrderOutput();      for (int i = 1; i <= n; i++)      {          cout << t[i].size() << " ";      }      cout << endl;      for (int i = 1; i <= n; i++)      {          cout << t[i].height() << " ";      }      cout << endl;      system("pause");      return 0;  } | | | |