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：202000130198 | 姓名： 隋春雨 | | 班级： 20.4 |
| 实验题目：二叉树操作 | | | |
| 实验学时：2 | | 实验日期： 2021-11-25 | |
| 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。 | | | |
| 软件开发环境：  CLION2020 | | | |
| 1. 实验内容 2. 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. 题目描述：   接收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。  输入输出格式：  输入：  第一行为数字n；  第二行有n个数字，表示二叉树的前序遍历；  第三行有n个数字，表示二叉树的中序遍历。  输出：  输出一行，表示该二叉树的后序遍历序列。   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）   **本次实验使用了两种方法解题，一种是递归的，一种是非递归的**   1. A题 2. 数据结构:二叉树。二叉树提供前序、中序、后序、层次遍历，同时通过遍历来计算二叉树的高度和结点的数目。对于二叉树的结点，有保存当前的元素和它的左右孩子，如果无左孩子或者右孩子，相应的设置为nullptr      1. 算法： 2. 层次遍历：我们发现，每一层中靠前的结点，他们的子树也会靠前，这是FIFO的性质，我们使用队列来存储。如果队列非空，那么我们继续。如果存在左右孩子（非空），那么我们就Push进去。      1. 计算二叉树的高度：   首先我们知道，对于所有的递归函数，都可以借助栈转换为非递归实现。因此本次实验也使用了递归和非递归来解题。  递归实现    对于每一个结点，它的高度等于max(leftchild,rightchild)+1，所以我们可以很轻松的使用递归函数来进行求解即可。  非递归函数：我们使用一个stack来保存，对于每一个结点，如果是第一次被push进去，那么我们检查一下它的左右孩子是否非空，如果非空，那么push。如果不是第一次被push进去，那么我们进行更新其父亲结点的高度，max(now,child.height+1),代码如下：     1. 计算结点数目：   递归实现：  每一个结点的size等于左右孩子的size和+1，所以我们可以使用递归函数来实现    非递归实现：  同样的，类似上面的思路，我们发现对于每一个结点，先被访问的一定后出来，所以我们使用stack来存储。每一个结点，如果没有被Push过，那么我们push其孩子结点。否则我们进行更新其父亲结点的size，代码如下：     1. B题   由前序和中序确定后序：  比如这棵二叉树：    PreOrder:        GDAFEMHZ  InOrder:            ADEFGHMZ  那么我们怎么确定它的后序遍历呢？  首先我们知道，前序、中序为什么叫前序和中序，因为这标记者拜访root结点的顺序，前序是根、左、右，中序是左、根、右。给定两个序列，我们就可以唯一确定它的后序序列，因为对于一个前序序列，它的第一个结点就是root，我们在中序序列中找到root，就可以确定它的左右子树的size。根据左右子树的size就可以确定左右子树的前序、中序遍历，递归下去，就可以求解出它的后序遍历。递归的终止条件为子树的size为0。代码如下：     1. 测试结果（测试输入，测试输出） 2. a题   输入：    输出：    提交OJ的结果：    (2)b题  输入：    输出：    提交OJ最后的结果：     1. 分析与探讨（结果分析，若存在问题，探讨解决问题的途径） 2. 如何将递归实现转换为非递归实现？   解决：对于本道题来说，递归实现是不困难的。但是我们知道，所有的递归实现都可以借助循环与stack转换为非递归实现，考虑A题的操作。计算高度的问题，我们只有知道了子树的高度才能知道root结点的高度，因此这是一个后进先出的数据结构，使用stack来存储。     1. 在提交oj的时候要把自己的测试删除，比如多输出了一个换行可能就会导致全部的判错。 2. 对于B题，不能使用string来存储。因为它有可能是两位数字，如果我们想当然的认为只有一位数字用string来存储的话，一定会RE 3. 对于BOOL数组的初始化，绝对不能想当然，每一次的定义都要对其进行初始化。在本次的实验中，因为bool数组没有初始化导致debug了很久。      1. 对于下标从1开始的数组，要多动态分配一块内存。因为索引为0的地方我们是没有访问的。      1. 对于一个只有两个私有成员的struct，我们可以直接使用pair来存储。简化了我们的代码量。同时对于返回多个值的函数，我们只能使用引用来返回。这也是为什么兴起了Go语言的一个原因。 2. 对于height等操作的计算，一定要特判是否合法。因为对于root结点来说，它的父节点是自身，不能直接增加。      1. 我们在写循环条件判断的时候，对于短路情况的判断一定要慎重，我们对于指针的使用一定要先判断是否为空，在进行取值操作，如下：      1. 对于维护私有变量的时候，要特殊情况特殊判：比如说删除结点的时候，要考虑这个是不是头结点，如果是，那么更新私有变量。如下：     (10)要注意私有成员的更新，public函数知道自己所处的状态都是靠着私有成员才知道的，如果没有及时更新，那数据就成了垃圾数据，没有任何意义。我在写实验的时候，也经历过没更新导致的Bug，最终debug查出来，就是下面这个：     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)A题**   1. #include<iostream> 2. #include <queue> 3. #include <stack> 4. using namespace std; 5. template <class T> 6. template<class T> 7. class binaryTree 8. { 9. binaryTreeNode<T>\* root; 10. int \_size; 11. public: 12. binaryTree(binaryTreeNode<T>\* root, int \_size) : root(root), \_size(\_size) {} 13. void levelOrder() 14. { 15. queue<binaryTreeNode<T>>q; 16. if (root) 17. { 18. q.push(\*root); 19. while (!q.empty())*//队列非空* 20. { 21. binaryTreeNode<T> front = q.front(); 22. q.pop(); 23. cout << front.element << " "; 24. if (front.leftChild)*//有左孩子* 25. { 26. q.push(\*front.leftChild); 27. } 28. if (front.rightChild)*//有右孩子* 29. { 30. q.push(\*front.rightChild); 31. } 32. } 33. cout << endl; 34. } 35. } 36. void size() 37. { 38. int\* cnt = new int[1 + \_size]; 39. for (int i = 1; i <= \_size; i++) 40. { 41. cnt[i] = 1;*//初始化* 42. } 43. bool\* jud\_push = new bool[1 + \_size](); 44. pair<binaryTreeNode<T>, binaryTreeNode<T>>fa\_son;*//用pair来存储* 45. stack<pair<binaryTreeNode<T>, binaryTreeNode<T>>>s; 46. s.push({ \*root,\*root });*//push进去* 47. while (!s.empty()) 48. { 49. pair<binaryTreeNode<T>, binaryTreeNode<T>>node = s.top(); 50. if (!jud\_push[node.second.element])*//如果没有被push过* 51. { 52. if (node.second.leftChild)*//存在左孩子* 53. { 54. s.push({ node.second,\*node.second.leftChild }); 55. } 56. if (node.second.rightChild)*//存在右孩子* 57. { 58. s.push({ node.second,\*node.second.rightChild }); 59. } 60. jud\_push[node.second.element] = 1;*//标记* 61. } 62. else 63. { 64. if (node.second.element != node.first.element)*//如果不是root结点* 65. { 66. cnt[node.first.element] += cnt[node.second.element]; 67. } 68. s.pop(); 69. } 70. } 71. for (int i = 1; i <= \_size; i++) 72. { 73. cout << cnt[i] << " "; 74. } 75. cout << endl; 76. delete []cnt;*//收回内存* 77. } 78. void height() 79. { 80. int\* height = new int[\_size + 1]; 81. for (int i = 1; i <= \_size; i++) 82. { 83. height[i] = 1;*//初始化* 84. } 85. bool\* jud\_push = new bool[\_size + 1]();*//初始化为0* 86. pair<binaryTreeNode<T>, binaryTreeNode<T>>fa\_son;*//父 子* 87. stack<pair<binaryTreeNode<T>, binaryTreeNode<T>>>s; 88. s.push({ \*root,\*root }); 89. while (!s.empty()) 90. { 91. pair<binaryTreeNode<T>, binaryTreeNode<T>>node = s.top(); 92. if (!jud\_push[node.second.element])*//如果没有被push过* 93. { 94. if (node.second.leftChild) 95. { 96. s.push({ node.second,\*node.second.leftChild }); 97. } 98. if (node.second.rightChild) 99. { 100. s.push({ node.second,\*node.second.rightChild }); 101. } 102. jud\_push[node.second.element] = 1; 103. } 104. else 105. { 106. if(node.first.element!=node.second.element)*//不是root结点* 107. { 108. height[node.first.element] = max(height[node.first.element], height[node.second.element] + 1); 109. } 110. s.pop(); 111. } 112. } 113. for (int i = 1; i <= \_size; i++) 114. { 115. cout << height[i] << " "; 116. } 117. cout << endl; 118. delete[]height; 119. } 120. }; 121. int main() 122. { 123. int n; 124. cin >> n; 125. binaryTreeNode<int>\* node = new binaryTreeNode<int>[1 + n]; 126. for (int i = 1; i <= n; i++) 127. { 128. node[i].element = i; 129. int l, r; 130. cin >> l >> r; 131. if (l >= 1) 132. { 133. node[i].leftChild = &node[l]; 134. } 135. else 136. { 137. node[i].leftChild = nullptr; 138. } 139. if (r >= 1) 140. { 141. node[i].rightChild = &node[r]; 142. } 143. else 144. { 145. node[i].rightChild = nullptr; 146. } 147. } 148. binaryTree<int>b(node + 1, n); 149. b.levelOrder(); 150. b.size(); 151. b.height(); 152. delete[]node; 153. return 0; 154. }   **(2)B题**   1. #include<iostream> 2. using namespace std; 3. template<class T> 4. struct binaryTreeNode{*//树节点类型* 5. T element; 6. binaryTreeNode<T>\* leftChild; 7. binaryTreeNode<T>\* rightChild; 8. binaryTreeNode(){rightChild=leftChild=NULL;} 9. binaryTreeNode(const T& theElement):element(theElement)//初始化 10. { 11. leftChild = rightChild = NULL; 12. } 13. binaryTreeNode(const T& theElement,binaryTreeNode<T>\* theLeftChild,binaryTreeNode<T>\* theRightChild) 14. { 15. element=theElement; 16. leftChild=theLeftChild; 17. rightChild=theRightChild; 18. } 19. }; 20. template<typename E> 21. binaryTreeNode<E>\* build(E\* preo,int lp,int rp,E\* ino,int li,int ri,int \*re,int n) 22. {*//前序序列preo,处理区间[lp,rp],中序序列ino,处理区间[li,ri],辅助数组re,总节点个数n;* 23. if(li>ri) return NULL; 24. binaryTreeNode<E>\* now=new binaryTreeNode<E>; 25. int pos=re[(int)preo[lp]];*//当前子树的根在中序中的位置* 26. int numl=pos-li;*//左子树的节点个数* 27. now->leftChild=build(preo,lp+1,lp+numl,ino,li,pos-1,re,n); 28. now->rightChild=build(preo,lp+numl+1,rp,ino,pos+1,ri,re,n); 29. now->element=preo[lp]; 30. return now; 31. } 32. void work(binaryTreeNode<int>\* t) 33. {*//输出后序遍历* 34. if(t==NULL) return ; 35. work(t->leftChild); 36. work(t->rightChild); 37. cout<<t->element<<' '; 38. } 39. int main() 40. { 41. int n; 42. cin>>n; 43. int\* a=new int[n+1]; 44. int\* b=new int[n+1]; 45. int\* re=new int[n+1];*//初始化re数组* 46. for(int i=1;i<=n;++i) cin>>a[i]; 47. for(int i=1;i<=n;++i) cin>>b[i]; 48. for(int i=1;i<=n;++i) re[b[i]]=i;*//记录每个数值在中序遍历中的位置* 49. work(build(a,1,n,b,1,n,re,n)); 50. return 0; 51. } | | | |
|  | | | |