**《数据结构实验》报告**

|  |  |  |
| --- | --- | --- |
| **第 六 次实验** | **日期：2021.12.10** | **得分：** |
| **学号：** | **姓名：*Steven*** | **专业：智能科学与技术** |

**一、实验目的**

1. 练习C语言函数的使用方法。

2. 练习使用malloc、realloc、free等函数动态分配内存的方法。

3. 练习指针变量的使用，了解其用法。

4. 理解并熟练结构体的使用方法。

5. 理解并学会二叉链表、顺序二叉树的构建和操作

**二、实验内容**

1 、假设二叉树中数据元素类型是字符型， 请采用二叉链表实现二叉树的以下基本操作：

(1) 根据二叉树的先序序列和中序序列构造二叉树；

(2) 根据先序遍历二叉树；

(3) 根据中序遍历二叉树；

(4) 根据后序遍历二叉树。

测试数据包括如下错误数据：

先序：1234；中序：12345

先序：1234；中序：1245

先序：1234；中序：4231

2、对于一棵二叉树， 请实现：

(1)计算二叉树的叶子数目；

(2)计算二叉树的深度。

3、给定n个权值，请构造它们的最优二叉树(赫夫曼树)。

4、假设二叉树中数据元素是正整数，请采用顺序存储实现二叉树，其中值为0的结点代表空结点。请求解编号为i和j的两个结点的最近公共祖先结点的编号p和值v，如果i或j是空结点，则输出错误提示。

5、农夫要修理牧场的一段栅栏，他测量了栅栏，发现需要N块木头，每块木头长度为整数Li个长度单位，于是他购买了一条很长的、能锯成N块的木头，即该木头的长度是Li的总和。但是农夫自己没有锯子，请人锯木的酬金跟这段木头的长度成正比。为简单起见，不妨设酬金等于所锯木头的长度。例如，要将长度为20的木头锯成长度为8、7和5的三段，如果第一次将长度为20的木头锯成12和8，花费20；第二次将长度为12的木头锯成7和5，花费12，总花费为32。如果第一次将长度为20的木头锯成巧和5，第二次将长度为巧的木头锯成8和7，总花费为35。请帮助农夫计算将木头锯成N块的最少花费。

**三、数据结构及算法设计**

1、

按照深度优先的规则递归地构建二叉树，并在构建好后，分别使用前序、中序、后序遍历，得到遍历结果。

2、

(1) 计算叶子节点：

按照“根-左-右”即前序遍历的顺序进行遍历，当遍历到的节点的左右子树都为NULL的时候，叶子节点数目++。

(2) 计算整棵树的深度：

首先对于任一节点，其深度要继承左右子树的深度的较大值再加1。根据这一规律递归。一直递归到叶子节点，随后每一层的返回值都是其左右子树的深度的较大值+1。最后直到开始调用的那一层时，已经累加到最大层数了。

3、

构建哈夫曼树比较麻烦。首先将每个节点都存到一个数组中，该数组视为存放根节点的数组。随后遍历这些节点，将其中的两个区域作为存放要归并成树的部分；再之后再遍历整个数组，从中找到最小的两个根节点作为归并对象，malloc一个新的根节点，作为这两个的父节点。构建好之后，将这两个节点取出，一个改为NULL，一个让新的父节点存进去。重复该过程。

4、

由于本题中是用顺序存储二叉树，所以可以直接通过index计算出夫节点和左右子节点，也就是节点的index中暗含了父子关系。对于两个节点，首先判断其中是否有无效节点（值为0的节点），确定没有之后再进行查找祖先。因为任两个有效节点一定存在公共祖先，而两者通过不断整除2可以逐层获得各自的祖先。所以只要对每个节点迭代地找祖先节点，并且按照趋近原则进行查找即可。

5、

其实本质上就是利用的哈夫曼树，因为在切出某段的前几次切割中，该段的长度产生的费用是累加的——这和带权路径长度特性是一致的，所以按照哈夫曼树的特点，让权值较大的段优先切割即可。

**四、核心程序代码**

1.

|  |
| --- |
| BiTree.h |
| #ifndef \_BITREE\_H\_  #define \_BITREE\_H\_  #include <stdio.h>  #include <stdlib.h>  #include <windows.h>  #include <string.h>  #include <math.h>  #define DataType char  #define PREORDER 'A' //通过前序序列构建  #define POSTORDER 'B' //通过后序序列构建  //链二叉树设计  struct BiTreeNode {  DataType data;  BiTreeNode\* lChild = NULL;  BiTreeNode\* rChild = NULL;  };  //链二叉树的基本操作  struct LinkBiTree {  BiTreeNode\* root = NULL; //根节点(一切的开始)  int CreateSituation = FALSE; //记录二叉树的构建情况  int nodeNum = 0; //统计节点数  int leavesNum = 0; //统计叶子节点数  //创建链二叉树的函数  int CreateBiTree(char buildMethod, char\* midStr, char\* anoStr) {  //参数分别为：构建二叉树的方式、中序序列、前/后序序列  //以下步骤为初步检查两个字符串  if (strlen(anoStr) != strlen(midStr)) {  printf("您输入的序列长度不一致，无法构建");  return ERROR;  }  if (strlen(anoStr) == 0) {  printf("该树为空树");  root->data = 0;  root->lChild = NULL;  root->rChild = NULL;  }  for (int i = 0; i < strlen(midStr); i++) {  if (!strchr(anoStr, midStr[i])) {  printf("您输入的两组序列中，字符不完全一致，无法构建");  return ERROR;  }  }  nodeNum = strlen(midStr);  //初步检查完毕，可以构建了  if (buildMethod == PREORDER) {  root = PreOrderBuild(anoStr, midStr, 0, nodeNum - 1, 0, nodeNum - 1);  }  else if (buildMethod == POSTORDER) {  root = PostOrderBuild(anoStr, midStr, 0, nodeNum - 1, 0, nodeNum - 1);  }  CreateSituation = TRUE;  return TRUE;  }  //通过前序序列和中序序列构建二叉树（实际构建的函数）  BiTreeNode\* PreOrderBuild(char\* preStr, char\* inStr, int preLEdge, int preREdge, int inLEdge, int inREdge) {  if (preLEdge > preREdge || inLEdge > inREdge) { return NULL; }  BiTreeNode\* newRoot = (BiTreeNode\*)malloc(sizeof(BiTreeNode)); newRoot->data = preStr[preLEdge]; newRoot->lChild = NULL; newRoot->rChild = NULL;  int lCMidLen = (strchr(inStr, preStr[preLEdge]) - inStr) - inLEdge;  //找到当前根节点的左子节点的中序序列的长度  int rCMidLen = inREdge - (strchr(inStr, preStr[preLEdge]) - inStr);  //找到当前根节点的右子节点的中序序列的长度  preLEdge++;//++表示前序左边界逐步右移，即遍历前序序列，这一步是为构建子树做准备  //接下来这几行是为了检查是否能构建成二叉树  DataType lC = preStr[preLEdge], rC = preStr[preREdge - rCMidLen + 1];  if ((strchr(preStr, lC) - strchr(preStr, rC)) \* (strchr(inStr, lC) - strchr(inStr, rC)) < 0) {//小于0说明异号，也就是两者的相对位置不同  printf("您输入的两个序列不匹配，无法构建成二叉树\n");  exit(ERROR);  }  newRoot->lChild = PreOrderBuild(preStr, inStr, preLEdge, preLEdge + lCMidLen - 1, inLEdge, lCMidLen + inLEdge - 1);  newRoot->rChild = PreOrderBuild(preStr, inStr, preREdge - rCMidLen + 1, preREdge, inREdge - rCMidLen + 1, inREdge);  return newRoot;  }  //通过后序序列和中序序列构建二叉树（实际构建的函数）  BiTreeNode\* PostOrderBuild(char\* postStr, char\* inStr, int postLEdge, int postREdge, int inLEdge, int inREdge) {  if (postLEdge > postREdge || inLEdge > inREdge) { return NULL; }  BiTreeNode\* newRoot = (BiTreeNode\*)malloc(sizeof(BiTreeNode)); newRoot->data = postStr[postREdge]; newRoot->lChild = NULL; newRoot->rChild = NULL;  int lCMidLen = int(strchr(inStr, postStr[postREdge]) - inStr) - inLEdge;  //找到当前根节点的左子节点的中序序列的长度  int rCMidLen = inREdge - int(strchr(inStr, postStr[postREdge]) - inStr);  //找到当前根节点的右子节点的中序序列的长度  //接下来这几行是为了检查所给字符串是否能构建成二叉树  DataType lC = postStr[postLEdge + lCMidLen - 1], rC = postStr[postREdge - 1];  if ((strchr(postStr, lC) - strchr(postStr, rC)) \* (strchr(inStr, lC) - strchr(inStr, rC)) < 0) {//小于0说明异号，也就是两者的相对位置不同  printf("您输入的两个序列不匹配，无法构建成二叉树\n");  exit(ERROR);  }  newRoot->lChild = PostOrderBuild(postStr, inStr, postLEdge, postLEdge + lCMidLen - 1, inLEdge, lCMidLen + inLEdge - 1);  postREdge--;  newRoot->rChild = PostOrderBuild(postStr, inStr, postLEdge - rCMidLen + 1, postREdge, inREdge - rCMidLen + 1, inREdge);  return newRoot;  }  //前序遍历二叉树  int PreOrderTraverse(BiTreeNode\* root) {  if (root && CreateSituation) {  printf("%c", root->data);  PreOrderTraverse(root->lChild);  PreOrderTraverse(root->rChild);  }  return TRUE;  }  //中序遍历二叉树  int InOrderTraverse(BiTreeNode\* root) {  if (root && CreateSituation) {  InOrderTraverse(root->lChild);  printf("%c", root->data);  InOrderTraverse(root->rChild);  }  return TRUE;  }  //后序遍历二叉树  int PostOrderTraverse(BiTreeNode\* root) {  if (root && CreateSituation) {  PostOrderTraverse(root->lChild);  PostOrderTraverse(root->rChild);  printf("%c", root->data);  }  return TRUE;  }  //返回二叉树深度  int GetDepth(BiTreeNode\* root) {  int i, j;  if (!root)  return 0;  if (root->lChild)  i = GetDepth(root->lChild); //左子树深度  else  i = 0;  if (root->rChild)  j = GetDepth(root->rChild); //右子树深度  else  j = 0;  return i > j ? i + 1 : j + 1;  }  //返回叶节点数  int LeavesNum(BiTreeNode\* root) {  //只可执行1次，因为它是叶子数++计数，如果该函数需要多次调用的话需要将leavesNum清零再开始  if (root) {  if (root->lChild == NULL && root->rChild == NULL) { leavesNum++; }  LeavesNum(root->rChild);  LeavesNum(root->lChild);  }  return leavesNum;  }  };  #endif |

2.

|  |
| --- |
| BiTree.h（这两个函数是属于BiTree.struct LinkBiTree中的） |
| //返回二叉树深度  int GetDepth(BiTreeNode\* root) {  int i, j;  if (!root)  return 0;  if (root->lChild)  i = GetDepth(root->lChild); //左子树深度  else  i = 0;  if (root->rChild)  j = GetDepth(root->rChild); //右子树深度  else  j = 0;  return i > j ? i + 1 : j + 1;  }  //返回叶节点数  int LeavesNum(BiTreeNode\* root) {  //只可执行1次，因为它是叶子数++计数，如果该函数需要多次调用的话需要将leavesNum清零再开始  if (root) {  if (root->lChild == NULL && root->rChild == NULL) { leavesNum++; }  LeavesNum(root->rChild);  LeavesNum(root->lChild);  }  return leavesNum;  } |

3.

|  |
| --- |
| HuffmanTree.h |
| #ifndef \_HUFFMANTREE\_H\_  #define \_HUFFMANTREE\_H\_  #include<stdio.h>  #include<stdlib.h>  struct HuffmanNode {  int data; //树中节点的值  HuffmanNode\* left;  HuffmanNode\* right;  };  HuffmanNode\* createTree(int arr[], int length) {//输入数值节点和数组长度（实际的数值个数）  HuffmanNode\*\* ptrArr = (HuffmanNode\*\*)malloc(length \* sizeof(HuffmanNode\*));//存放节点  HuffmanNode\* ptr; HuffmanNode\* newRoot = NULL;//指针，分别作为创建时的遍历指针和每层创建的根指针  for (int i = 0; i < length; i++) { //依次构建每一个节点，并用数组存放所有节点  ptr = (HuffmanNode\*)malloc(sizeof(HuffmanNode));  ptr->data = arr[i];  ptr->left = ptr->right = NULL;  ptrArr[i] = ptr;  }  for (int i = 1; i < length; i++) { //进行 n-1 次循环建立哈夫曼树  //min1Index表示森林中具有最小权值的根结点的下标,min2index为第二小的下标  int min1Index = -1, min2Index;  for (int j = 0; j < length; j++) {//先直接取第一下标和第二下标  if (ptrArr[j] != NULL && min1Index == -1) {  min1Index = j;  continue;  }  if (ptrArr[j] != NULL) {  min2Index = j;  break;  }  }  for (int j = min2Index; j < length; j++) {  if (ptrArr[j] != NULL) {  if (ptrArr[j]->data < ptrArr[min1Index]->data) {//依次更新，保证min1Index始终指向最小值，min2Index指向次小值  min2Index = min1Index;  min1Index = j;  }  else if (ptrArr[j]->data < ptrArr[min2Index]->data) {  min2Index = j;  }  }  }  //由最小权值树和次最小权值树建立一棵新树,pRoot为新根  newRoot = (HuffmanNode\*)malloc(sizeof(HuffmanNode));  newRoot->data = ptrArr[min1Index]->data + ptrArr[min2Index]->data;  newRoot->left = ptrArr[min1Index];  newRoot->right = ptrArr[min2Index];  ptrArr[min1Index] = newRoot; //将指向新树的指针赋给ptrArr指针数组中min1Index  ptrArr[min2Index] = NULL; //min2Index位置为空  }  return newRoot;  }  int calcWeightLength(HuffmanNode\*& rootPtr, int len) {  if (rootPtr == NULL) { //空树返回0  return 0;  }  else {  if (rootPtr->left == NULL && rootPtr->right == NULL) { //访问到叶子节点  return rootPtr->data \* len;  }  else {  return calcWeightLength(rootPtr->left, len + 1) + calcWeightLength(rootPtr->right, len + 1); //向下递归计算  }  }  }  void InOrderTraverse(HuffmanNode\*& rootPtr) {  if (rootPtr == NULL) {  return;  }  else {  InOrderTraverse(rootPtr->left);  printf("%d ", rootPtr->data);  InOrderTraverse(rootPtr->right);  }  }  void PreOrderTraverse(HuffmanNode\*& rootPtr) {  if (rootPtr == NULL) {  return;  }  else {  printf("%d ", rootPtr->data); //依次打印哈夫曼树中各个节点的孩子节点  PreOrderTraverse(rootPtr->left);  PreOrderTraverse(rootPtr->right);  }  }  #endif |

4.

|  |
| --- |
| BiTree.h |
| #ifndef \_BITREE\_H\_  #define \_BITREE\_H\_  #include <stdio.h>  #include <stdlib.h>  #include <windows.h>  #include <string.h>  #include <math.h>  //顺序二叉树  struct SqBiTree {  char\* sqTree;//存放顺序二叉树  void PreOrderBuild(DataType\* preStr, DataType\* inStr, int preLEdge, int preREdge, int inLEdge, int inREdge, int index) {  //参数列表依次是：前序序列，中序序列，前序左边界，前序右边界，中序左边界，中序右边界，原数据数组，当前节点的index  if (preLEdge > preREdge || inLEdge > inREdge) { if (index < strlen(sqTree))sqTree[index] = 0; return; }  sqTree[index] = preStr[preLEdge];  int lCMidLen = (strchr(inStr, preStr[preLEdge]) - inStr) - inLEdge;//找到当前根节点的左子节点的中序序列的长度  int rCMidLen = inREdge - (strchr(inStr, preStr[preLEdge]) - inStr);//找到当前根节点的右子节点的中序序列的长度  preLEdge++;//++表示前序左边界逐步右移，即遍历前序序列，这一步是为构建子树做准备  //接下来这几行是为了检查是否能构建成二叉树  DataType lC = preStr[preLEdge], rC = preStr[preREdge - rCMidLen + 1];  if ((strchr(preStr, lC) - strchr(preStr, rC)) \* (strchr(inStr, lC) - strchr(inStr, rC)) < 0) {//小于0说明异号，也就是两者的相对位置不同  printf("您输入的两个序列不匹配，无法构建成二叉树\n");  exit(ERROR);  }  PreOrderBuild(preStr, inStr, preLEdge, preLEdge + lCMidLen - 1, inLEdge, lCMidLen + inLEdge - 1, 2 \* index);  PreOrderBuild(preStr, inStr, preREdge - rCMidLen + 1, preREdge, inREdge - rCMidLen + 1, inREdge, 2 \* index + 1);  }  void FindAncestor(int index1, int index2, int& index, char& val) {  if (sqTree[index1] == 0 || sqTree[index2] == 0) {  printf("ERROR\n");  index = -1; val = 0;  return;  }  while (index1 != index2) {  index1 > index2 ? index1 /= 2 : index2 /= 2;  }  index = index1; val = sqTree[index1];  }  };  #endif |

5.

|  |
| --- |
| HuffmanTree.h |
| （和第3题代码一致） |

**五、测试及结果**

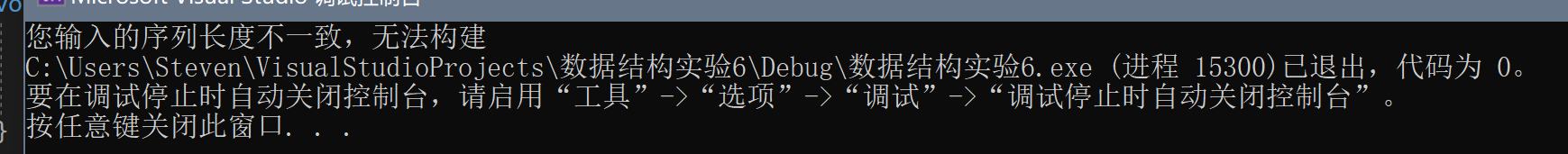
1. 测试用例：

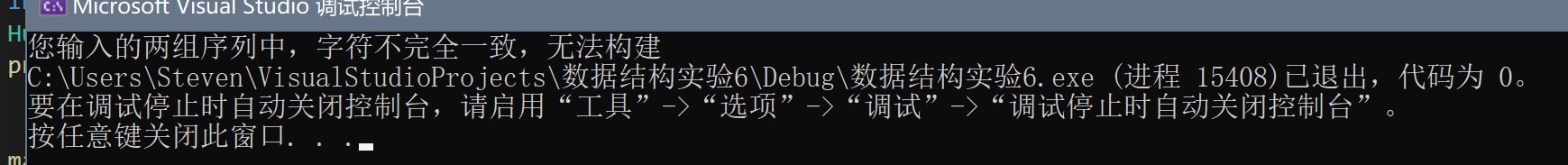
按照题给测试用例，分别以以下用例进行测试，结果依次如图

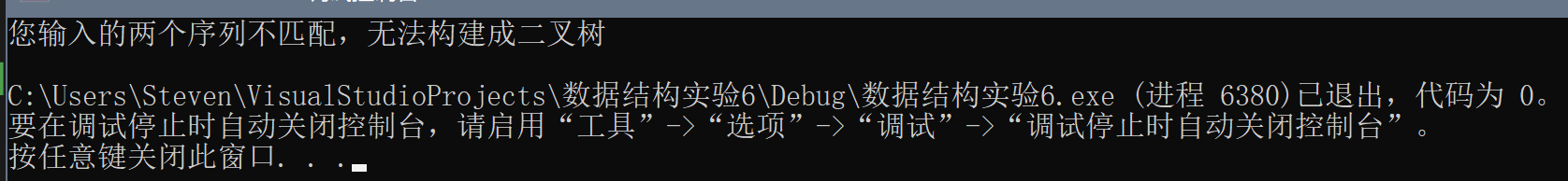
①前序：1234，中序：12345

②前序：1234，中序：1245

③前序：4231，中序：1234







2. 测试用例：

前序序列：abdce，中序序列：dbace

展示其深度和叶子节点个数

图示

中度可信度描述已自动生成 图片包含 游戏机, 电路, 监控

描述已自动生成

3. 测试用例：3，9，5，12，6，15

日程表

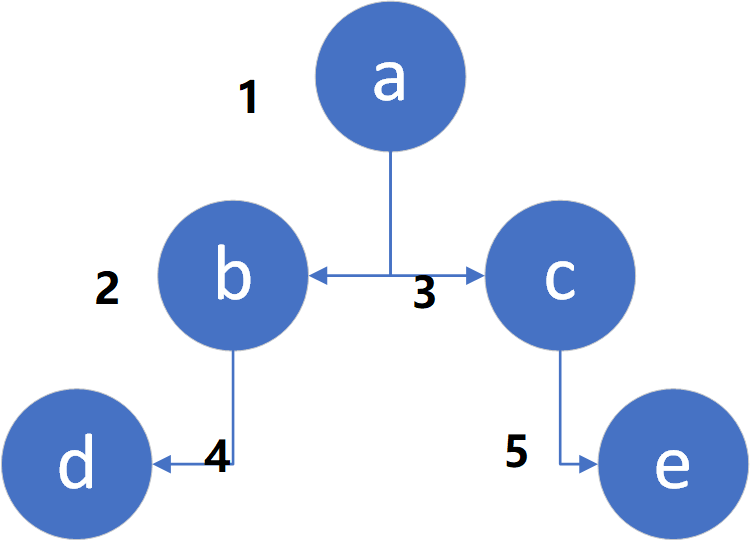
描述已自动生成手机屏幕截图

描述已自动生成

4. 测试用例：

前序序列：abdce，中序序列：dbace

求公共祖先的节点：tree[2]和tree[1]

文本

描述已自动生成

5. 测试用例：Li={8,7,5}

文本

描述已自动生成