

《 算法与数据结构 》

实验报告本

|  |  |
| --- | --- |
| 班 级： | **计 203** |
| 学 号： | **20002462** |
| 姓 名： | **刘 子 言** |
| 指导教师： | **叶 琪** |

实验成绩：

信息科学与工程学院

2022年 6 月

**实 验 报 告 （ 1 ）**

|  |  |
| --- | --- |
| **实验名称**：线性表实验 | **实验地点**：线上 |
| **所使用的工具软件及环境：Win10/Win 7, Visual C++/Java** | |
| **一、实验目的：**  1.熟悉数据结构和编程语言的集成开发环境，掌握程序设计与实现的能力，分析算法的复杂度。  2.要求掌握线性表的基本操作：插入、删除、查找等运算在顺序存储结构和链式存储结构上的运算。  3.熟练掌握堆栈和队列的基本操作，栈在表达式求解中的应用，双端队列的应用。 | |
| **二、实验内容描述：**（填写题目内容及输入输出要求）  **1.**已知一元多项式P，设计算法计算P的导数。多项式以指数递减的方式输入，每行代表一项，每行第一个分量表示非零系数，第二个分量代表指数。输出格式同输入格式相同。  输入样例：  3 4  -5 2  6 1  -2 0  输出样例：  12 3  -10 1  6 0  **2.**给定两个链表，每个链表都已经按升序排列，设计算法实现将两个链表合并到一个升序链表中，返回合并后的链表。  输入：1 4 5  1 3 6  输出：1 1 3 4 5 6  **3.**输入一个中缀表达式，利用栈结构求解表达式的值。其中运算符包括：+、-、\*、/、（、），表达式以“=”为结尾，参与运算的数据为double类型且为正数。  输入样例：20 \* ( 4.5 – 3 ) =  输出结果：30.00  **4.**给定一个队列，利用队列的合法操作（isEmpty、AddQ、DeleteQ）实现队列中元素的从小到大排序。其中：输入第一行表示队列元素个数，第二行为队列中的元素。  注意：不允许直接访问队列中的元素。  输入样例：10  9 4 6 1 8 3 7 0 2 5  输出样例：  0 1 2 3 4 5 6 7 8 9 | |
| **三、程序运行结果（说明设计思路，解释使用的数据结构，计算时间复杂度）**  **第1题**   1. 实验运行结果截图 2. 数据结构   采用链式存储结构存储一元多项式的非零系数与指数，构造结构体如下：  typedef struct PolyNode \* PtrToPolyNode;  struct PolyNode{  int Coef; //非零系数  int Expon; //对应指数  PtrToPolyNode Next;  };  typedef PtrToPolyNode Polynomial;   1. 设计思路   自定义链表创建函数、求导运算函数。  ·先创建链表表示一元多项式：先创建链表头结点，再利用循环语句将输入内容依次存入链表的数据结点中，返回表头结点，实现创建；  ·再对已经创建好的一元多项式进行求导：链表指针从头结点开始，依次后移，每经过一个结点，进行这一项的求导运算（如果指数为零则求导后该项不存在）；  ·最后按照系数指数依次输出求导结果。   1. 时间复杂度   创建多项式链表过程有一个单循环，时间复杂度为O(n)；求导过程中有一个while循环，时间复杂度为O(n)，所以最终程序的时间复杂度为O(n)。  **第2题**   1. 实验运行结果截图      1. 数据结构   采用链式存储结构存储升序序列，构造结构体如下：  typedef struct Node \* PtrToNode;  struct Node{  int Data; //数据  PtrToNode Next;  };  typedef PtrToNode List;   1. 设计思路   自定义链表创建函数、链表合并函数。  ·先创建两个链表L1、L2：先创建链表头结点，再利用循环语句将输入的升序序列数据依次存入链表结点中，返回表头结点，实现创建；  ·再进行链表合并：将合并结果链表L3指向L1头结点，在L1基础上合并，节省存储空间；接下来进行比较与判断，如果L1结点数据较小，则将该结点接在链表L3后面，L1指针后移，如果L2结点数据较小，则将该结点接在链表L3后面，L2指针后移；以此类推，直到L1、L2中有一方遍历结束，再将L1或L2剩下的结点接在链表L3后面，即完成两个升序链表的合并；  ·最后将合并得到的链表中的数据依次输出。   1. 时间复杂度   链表创建函数中有一个单循环，时间复杂度为O(n)；链表合并过程中有一个while循环，时间复杂度为O(n)，所以最终程序的时间复杂度为O(n)。  **第3题**   1. 实验运行结果截图 2. 数据结构   采用栈结构存储、求解中缀表达式，构造结构体如下：  struct Node{  double num; //操作数  char op; //操作符  };  typedef struct Node expnode;  stack<expnode>s;   1. 设计思路   导入<stack>库；自定义操作符对应优先级函数、针对一个运算符的单次运算函数、中缀转后缀表达式的函数。  ·先输入中缀表达式：将输入的表达式串儿存入字符串类型的变量str中；  ·中缀表达式转后缀表达式：调用自定义的中缀转后缀表达式函数，设置操作符栈（栈底元素设为#）和操作数栈，依次对字符串变量str中的每一个字符进行判断：  若为数字，则将字符转化为数字存入操作数栈中，以下为数字转换的关键代码：  temp.num = str[i] - '0'; //将字符转换成数字存储  i++;  while(i<str.length() && str[i]>='0'&&str[i]<='9'){ //若整数部分出超过一位数字  temp.num = temp.num\*10 + (str[i] - '0');  i++;  }  if(str[i]=='.') { //若有小数点存在（对小数部分进行处理）  i++;  k=0.1;  while(i<str.length() && str[i]>='0'&&str[i]<='9'){  temp.num = temp.num + (str[i] - '0')\*k;  i++;  k=k\*0.1;  }  }  numS.push(temp.num); //将操作数入栈  如果不是数字，是操作符：  若为普通运算符，比较它与上一个运算符的优先级，若上一个运算符的优先级较高，则从操作数栈中取出两个操作数，用操作符栈顶元素进行单次运算，将新的运算结果压入操作数栈中，将经过运算的操作符释放，将新的运算符压入栈中；  若为左括号，直接压入操作符栈中；  若为右括号，则从操作数栈中取出两个操作数，用操作符栈顶元素进行单次运算，将新的运算结果压入操作数栈中，将经过运算的操作符释放，直到栈中左括号出栈为止；  若为等号，则表示str字符串已到末尾，可以跳出循环；  若结束循环后操作数栈仍不为空，则再从操作数栈中取出两个操作数，用操作符栈顶元素进行单次运算，将新的运算结果压入操作数栈中，将经过运算的操作符释放，直到遇到操作符栈底元素#为止，结束运算；  ·最后将后缀表达式的运算结果输出（即为操作数栈中的最后一个元素）。   1. 时间复杂度   中缀转后缀表达式的函数中涉及到循环嵌套（for+while两层），时间复杂度为O(n2)，所以最终程序的时间复杂度为O(n2)。  **第4题**   1. 实验运行结果截图      1. 数据结构   采用线性结构队列，导入<queue>库，定义队列类型变量q：  queue<int>q;   1. 设计思路   自定义队列中元素排序函数，利用队列的合法操作实现排序。  ·先输入队列中元素个数，再将队列中的元素依次入队；  ·调用自定义的排序函数，利用双重for循环，第一次取出队头两个元素进行比较，将较小的元素再入队；此后再取队头一个元素，与外面剩下的一个元素进行比较，将较小的元素再入队，最后剩下最大的元素再入队，此为一次外循环；以此类推，经过n-1次外循环以后，队中元素按照从小到大的顺序排列，双重for循环关键代码如下：  for(int i=0; i<n-1 ; i++)  {  a=q.front();  q.pop();  for(int j=0; j<n-1-i; j++)  {  b=q.front();  q.pop();  if(a<b)  {  q.push(a);  a=b;  }  else  q.push(b);  }  q.push(a);  for(int k=0; k<i; k++)  {  c=q.front();  q.pop();  q.push(c);  }  }  ·最后再利用一个for循环输出队列中的升序序列。   1. 时间复杂度   队列的输入输出均为单个的for循环，而排序函数中运用到了双重for循环，所以最终程序的时间复杂度为O(n2)。 | |

**成绩：\_\_\_\_\_\_\_\_\_\_\_\_** **任课教师签名：\_\_\_\_\_\_\_\_\_\_\_\_** 2022年 月 日

**实 验 报 告 （ 2 ）**

|  |  |
| --- | --- |
| **实验名称**：树的应用 | **实验地点**：线上 |
| **所使用的工具软件及环境：Win7, Visual C++/Java** | |
| **一、实验目的：**  1、掌握二叉树的结构特征，以及各种存储结构的特点及使用范围。  2、掌握用指针类型描述、访问和处理二叉树的运算。  3、掌握树的应用算法。 | |
| **二、实验内容描述：**（填写题目内容及输入输出要求）  **1.**编写程序判断树是否同构？其中同构是指给定两棵树T1和T2。如果T1可以通过若干次左右孩子互换就变成T2，则称两棵树是“同构”的，输出True，否则输出False。  输入：  第一行N（表示树的结点数）  第二行开始：结点数据 左孩子编号 右孩子编号（如果无孩子结点记为“-1”）  输入样例：  8  A 1 2  B 3 4  C 5 -1  D -1 -1  E 6 -1  G 7 -1  F -1 -1  H -1 -1  8  G -1 4  B 7 6  F -1 -1  A 5 1  H -1 -1  C 0 -1  D -1 -1  E 2 -1  输出样例：  True  **2、**给定一棵二叉搜索树，请按中序遍历将其重新排列为一棵递增顺序搜索树，使树中最左边的节点成为树的根节点，并且每个节点没有左子节点，只有一个右子节点。例如，将左下图的二叉搜索树转换为右下图的树。    输入样例：5 3 6 2 4 null 8 1 null null null 7 9  输出样例：1 null 2 null 3 null 4 null 5 null 6 null 7 null 8 null 9  注：样例里面的null在实验代码中均由“-1”代替。  **3.**给定一个二叉树的根节点 root ，判断其是否是一个有效的二叉搜索树。  输入样例1：  输入：2 1 3  输出：True  输入样例2：  输入：5 1 4 null null 3 6  输出：False  注：样例里面的null在实验代码中均由“-1”代替。  **4.** 给定一个二叉树，编写算法计算二叉树中任意两个结点的公共祖先。其中，输入第一行为二叉树序列，第二行和第三行分别为两个节点编号；输出：两个节点的公共祖先。例如:    输入样例1：  输入：  3 5 1 6 2 0 8 null null 7 4  5  1  输出: 3  输入样例2：  输入:  3 5 1 6 2 0 8 null null 7 4  5  4  输出: 5  注：样例里面的null在实验代码中均由“-1”代替。 | |
| **三、程序运行结果（说明设计思路，解释使用的数据结构，计算时间复杂度）**  **第1题**   1. 实验运行结果截图 2. 数据结构   定义二叉树结点的结构体类型：  struct TNode{  char data; //数据类型为字符  int left; //左孩子对应编号  int right; //右孩子对应编号  };  定义结构体数组表示输入的两棵二叉树：  TNode BT1[50],BT2[50]; //定义输入的两棵二叉树  定义一个0-1变量判断两棵二叉树是否同构：  int flag = 0; //判断两棵树是否同构，=0目前同构，=1不同构   1. 设计思路   ·按照输入格式，利用单个for循环存入两棵二叉树每个结点的数据以及左右孩子结点的编号；  ·先判断两棵树的结点数是否相等，若不等，则一定不同构，flag=1；  ·若结点数相等，再进一步利用双重for循环，依次寻找两棵二叉树中数据相等的结点，并利用“结点编号-对应数据”转换函数找到他们左右孩子结点编号对应的数据，再利用比较函数进行比较判断，若不同构则flag=1；  定义两个“结点编号-对应数据”转换函数：  //根据左右孩子结点的编号找到左右结点的data，并传回  //将无孩子结点的-1都改为字符‘0’，更方便直接比较  char getBT1\_childData(int num){  if(num != -1)  return BT1[num].data;  else  return '0';  }  char getBT2\_childData(int num){  if(num != -1)  return BT2[num].data;  else  return '0';  }  定义比较两棵树相同结点的左右孩子结点数据是否相等的函数：  void compare(char L1, char R1, char L2, char R2){  if(L1 == L2) //1左=2左  {  if(R1 != R2) //1右!=2右  flag = 1;  }  else if(L1 == R2) //1左=2右  {  if(R1 != L2) //1右!=2左  flag = 1;  }  else //1左和2的左右都不等  flag = 1;  }  双重for循环的关键代码：  for(int i=0;i<N1;i++)  {  int j=0;  for(j=0;j<N2;j++)  {  if(BT1[i].data == BT2[j].data) //找到了BT2中与BT1[i]对应的结点data  {  //比较它们俩的左右孩子结点的data  compare(getBT1\_childData(BT1[i].left),getBT1\_childData(BT1[i].right),  getBT2\_childData(BT2[j].left) , getBT2\_childData(BT2[j].right));  break;  }  }  if(j==N2 || flag){  //如果BT2中没有BT1[i]对应的那个结点，或者已经检测出flag为1了，就跳出循环  flag = 1;break;  }  }  ·最后再判断如果BT2中没有BT1[i]对应的某个结点、或者是已经检测出flag为1，就跳出循环，宣布不同构（False）；否则，若直到循环结束flag都为0，则宣布同构（True）。   1. 时间复杂度   二叉树的输入为单个for循环，而比较是否同构时运用到了双重for循环，其他部分均为判断语句及其他语句，所以最终程序的时间复杂度为O(n2)。  **第2题**   1. 实验运行结果截图 2. 数据结构   采用链式存储结构来存储二叉搜索树，构造如下结构体：  typedef struct TNode \*Position;  typedef Position BinTree;  struct TNode{  int Data;//假设数据类型为整型  BinTree Left;  BinTree Right;  };   1. 设计思路   自定义二叉搜索树的创建函数（层序遍历的方法创建）、中序遍历函数。  ·首先先利用数组TreeNode[]存放输入的二叉搜索树元素，再调用创建函数将该数组作为实参传入，进行二叉搜索树的创建；  ·再调用中序遍历函数，得到递增序列，存入新的数组TreeMdata[]；  ·然后再次调用创建函数，将数组TreeMdata[]作为实参传入创建递增顺序的二叉搜索树；  ·最后利用for循环输出递增顺序二叉树的结点数据值（结点为空则输出-1）。  二叉搜索树创建函数的关键代码：  BinTree CreatBinTree(int Treedata[])  {  int Data;  int i=0;  BinTree BT, T;  queue<BinTree>Q;  if(Treedata[i]==-1) i++; //跳过中序遍历得到的第一个空-1  Data = Treedata[i];  i++;  if(Data != -1){ //分配根节点单元，并将结点地址入队  BT = (BinTree)malloc(sizeof(struct TNode));  BT->Data = Data;  BT->Left = BT->Right = NULL;  Q.push(BT);  }  else return NULL; //否则返回树为空  while(!Q.empty()&&Treedata[i]){  T = Q.front();  Q.pop();  Data = Treedata[i]; //读入T的左孩子  i++;  if(Data == -1)  T->Left = NULL;  else{ //分配新结点，作为出队结点的左孩子；再将新结点入队  T->Left = (BinTree)malloc(sizeof(struct TNode));  T->Left->Data = Data;  T->Left->Left = T->Left->Right = NULL;  Q.push(T->Left);  }  Data = Treedata[i]; //读入T的右孩子  i++;  if(Data == -1)  T->Right = NULL;  else{ //分配新结点，作为出队结点的右孩子；再将新结点入队  T->Right = (BinTree)malloc(sizeof(struct TNode));  T->Right->Data = Data;  T->Right->Left = T->Right->Right = NULL;  Q.push(T->Right);  }  }  return BT;  }  中序遍历函数的关键代码（递归方法实现）：  void InorderTraversal(BinTree BT){  if(BT){  InorderTraversal(BT->Left);  TreeMdata[j] = BT->Data;  j++;  InorderTraversal(BT->Right);  }  else{  TreeMdata[j] = -1;  j++;  }  }   1. 时间复杂度   输入输出的时间复杂度均为O(n)；二叉搜索树创建函数的时间复杂度也为O(n)；而中序遍历使用递归实现的时间复杂度也为O(n)；所以最终程序的时间复杂度为O(n)。  **第3题**   1. 实验运行结果截图 2. 数据结构   采用链式存储结构来存储二叉搜索树，构造如下结构体：  typedef struct TNode \*Position;  typedef Position BinTree;  struct TNode{  int Data; //假设数据类型为整型  BinTree Left;  BinTree Right;  };   1. 设计思路   自定义二叉树的创建函数（层序遍历创建）、中序遍历函数，这两个函数与第二题中的创建、中序遍历函数基本相同；自定义判断函数，判断中序遍历结果是否为升序。  ·首先先利用数组TreeNode[]存放输入的二叉树元素，再调用创建函数将该数组作为实参传入，进行二叉树的创建；  ·再调用中序遍历函数，得到递增序列，存入新的数组TreeMdata[]；  ·最后调用判断函数，将数组TreeMdata[]作为实参传入，判断数组中元素是否为升序，若是，则是有效的二叉搜索树，输出True，否则输出False。  判断“是否升序”函数的关键代码：  void IsBST(int Treedata[])  {  int k=0;  while(Treedata[k+1])  {  if(Treedata[k] >= Treedata[k+1]) //不是升序  {  cout<<"False"<<endl;  break;  }  k++;  }  if(k+1 == j) //全部是升序  cout<<"True"<<endl;  }   1. 时间复杂度   输入的时间复杂度为O(n)；二叉树创建函数的时间复杂度也为O(n)；中序遍历使用递归实现的时间复杂度也为O(n)；判断函数的时间复杂度也为O(n)；所以最终程序的时间复杂度为O(n)。  **第4题**   1. 实验运行结果截图      1. 数据结构   定义二叉树结点的结构体类型：  struct Node{  int data; //数据类型为整型  int num; //结点编号  int floor; //记录数据所在树的层数  };  定义结构体数组表示输入的二叉树：  Node BT[50]; //定义输入的二叉树   1. 设计思路   自定义寻找最近公共祖先的函数。  ·先输入二叉树存入结构体数组BT[]中，同时记录每个结点元素的编号以及层数，方便后续找公共祖先时使用；  ·再输入两个待寻找的子结点的数据，存入a、b；  ·然后调用“寻找最近公共祖先”函数，将BT与a、b作为实参传入；在此函数中，先找到a、b在二叉树中的位置，记录其层数和编号，再利用while循环，通过自下而上比较父节点的方式寻找最近的祖先节点：若两个结点不在一层，则先将高层结点向上寻找，直到与另一结点同层，再一起向上寻找父节点并比较，直到两者的父节点数据相同，则输出该父节点的数据，结束循环；  ·输出的数据即为两个子结点最近的祖先节点。  “寻找最近公共祖先”函数的关键代码：  while(x.data != y.data) //通过自下而上比较父节点的方式寻找最近的祖先节点  {  if(x.floor > y.floor)  {  x.floor = x.floor - 1;  x.num = x.num/2;  x.data = BT[x.num-1].data;  }  else if(x.floor < y.floor)  {  y.floor = y.floor - 1;  y.num = y.num/2;  y.data = BT[y.num-1].data;  }  else  {  x.floor = x.floor - 1;  y.floor = y.floor - 1;  x.num = x.num/2;  y.num = y.num/2;  x.data = BT[x.num-1].data;  y.data = BT[y.num-1].data;  }  }  cout<<x.data<<endl;   1. 时间复杂度   二叉树输入的时间复杂度为O(n)；“寻找最近公共祖先”函数的时间复杂度也为O(n)；所以最终程序的时间复杂度为O(n)。 | |

**成绩：\_\_\_\_\_\_\_\_\_\_\_\_** **任课教师签名：\_\_\_\_\_\_\_\_\_\_\_\_** 2022年 月 日

**实 验 报 告 （ 3 ）**

|  |  |
| --- | --- |
| **实验名称**：图的应用 | **实验地点**：线上 |
| **所使用的工具软件及环境：Win7, Visual C++/Java** | |
| **一、实验目的：**  1、理解图的含义；  2、掌握用邻接矩阵和邻接表的方法描述图的存储结构；  3、理解并掌握深度优先遍历和广度优先遍历的存储结构；  4、掌握图的应用算法（最小生成树、最短路径、拓扑排序、关键路径计算）。 | |
| **二、实验内容描述：**（填写题目内容及输入输出要求）  **1、**编写程序实现带权图的邻接矩阵存储，输出邻接矩阵。输入第一行为结点个数（节点编号从0开始），第二行开始为边的信息（节点编号，节点编号，权重）。输出邻接矩阵。    输入样例：  6  0 1 5  0 3 7  1 2 4  2 0 8  2 5 9  3 2 5  3 5 6  4 3 5  5 0 3  5 4 1  输出样例：  0 5 0 7 0 0  0 0 4 0 0 0  8 0 0 0 0 9  0 0 5 0 0 6  0 0 0 5 0 0  3 0 0 0 1 0  **2、**利用邻接表存储有向图，编写程序实现对图的深度优先遍历。输入第一行为结点个数，第二行为结点的数据，第三行开始为边的信息。输出深度优先遍历（从第一个节点开始）结果。    输入样例：  6  a b c d e f  a b  a c  a e  b e  e d  d f  e f  输出：  a b e d f c  **3、**假如给你一个社交网络图，请你对每个节点计算符合“六度空间”理论的结点占结点总数的百分比。其中，输入第1行给出两个正整数，分别表示社交网络图的结点数N、边数M。随后的M行对应M条边，每行给出一对正整数，分别是该条边直接连通的两个结点的编号（节点从1到N编号）。输出与结点距离不超过6的结点数占结点总数的百分比，精确到小数点后2位。  输入样例：  10 9  1 2  2 3  3 4  4 5  5 6  6 7  7 8  8 9  9 10  输出样例：每行格式(编号: xx.xx%)  1: 70.00%  2: 80.00%  3: 90.00%  4: 100.00%  5: 100.00%  6: 100.00%  7: 100.00%  8: 90.00%  9: 80.00%  10: 70.00% | |
| **三、程序运行结果（说明设计思路，解释使用的数据结构，计算时间复杂度）**  **第1题**   1. 实验运行结果截图 2. 数据结构   采用邻接矩阵的存储结构，定义图的邻接矩阵相关结构体：  ·图结点的定义  typedef struct GNode \*PtrToGNode;  struct GNode{  int Nv; //顶点数  int Ne; //边数  int G[MaxVertexNum][MaxVertexNum]; //邻接矩阵  };  typedef PtrToGNode MGraph; //以邻接矩阵的方式存储的图类型  ·边的定义  typedef struct ENode \*PtrToENode;  struct ENode{  int v1,v2; //边的顶点  int w; //权重  };  typedef PtrToENode Edge;   1. 设计思路   自定义图初始化函数（创建有固定多个顶点但没有边的图）、边的插入函数、图的构建函数（函数中调用初始化函数以及边的插入函数）。  ·先调用图的构建函数：先创建有固定多个顶点但没有边的图，读入顶点数据，再利用循环语句读入边，顺序为起点、终点、权重，将边插入邻接矩阵，最后返回图；  ·再利用双重for循环，输出邻接矩阵Graph->G[i][j]。  图的构建函数关键代码如下：  MGraph BuildGraph(){  int Nv;  cin>>Nv;  MGraph Graph;  Graph = CreateGraph(Nv);  Edge E;  E = (Edge)malloc(sizeof(struct ENode));  for(int i=0; i<10; i++){  //读入边，顺序为起点、终点、权重，插入邻接矩阵  cin>>E->v1>>E->v2>>E->w;  InsertEdge(Graph, E); //插入边  Graph->Ne++;  }  return Graph;  }   1. 时间复杂度   图的初始化（双重for循环初始化邻接矩阵为0）的时间复杂度为O(n2)；插入所有边的时间复杂度为O(n)；输出邻接矩阵（双重for循环）的时间复杂度为O(n2)；所以最终程序的时间复杂度为O(n2)。  **第2题**   1. 实验运行结果截图 2. 数据结构   采用邻接表的存储结构，定义图的邻接表相关结构体：  ·边的定义  typedef struct ENode \*PtrToENode;  struct ENode{  int v1,v2; //边的顶点，无权重  };  typedef PtrToENode Edge;  ·邻接点的定义  typedef struct AdjVNode \*PtrToAdjVNode;  struct AdjVNode{  int AdjV; //邻接点下标，无边权重  PtrToAdjVNode Next; //指向下一个邻接点的指针  };  ·顶点表头结点的定义  typedef struct Vnode{  PtrToAdjVNode FirstEdge; //边表头指针  char Data; //存顶点的数据  }AdjList[MaxVertexNum];  ·图结点的定义  typedef struct GNode \*PtrToGNode;  struct GNode{  int Nv; //顶点数  int Ne; //边数  AdjList G; //邻接表  };  typedef PtrToGNode LGraph; //以邻接表的方式存储的图类型   1. 设计思路   利用邻接表存储图与邻接矩阵的区别在于：邻接矩阵基于二维数组，而邻接表基于链表。  自定义图的初始化函数（创建有固定多个顶点但没有边的图）、边的插入函数（比如插入<v1,v2>，为v2建立新的邻接点，将v2插入v1的表头）、图的构建函数（函数中调用初始化函数以及边的插入函数）、深度优先搜索函数（递归实现）。  图的构建函数等函数与第一题的构建思路相类似；  深度优先搜索函数的关键代码如下：  void DFS(LGraph Graph, int v)  {  PtrToAdjVNode f;  Visited[v] = true; //标记顶点v已经访问过了为TRUE  cout<<Graph->G[v].Data<<" "; //输出正在访问下标为v的顶点的data  for(f = Graph->G[v].FirstEdge; f; f = f->Next) //对于v的每一个邻接点f->AdjV  if(!Visited[f->AdjV]) //如果f指向的结点还没有被访问过  DFS(Graph, f->AdjV); //则递归访问它  }  ·先调用图的构建函数：先创建有固定多个顶点但没有边的图，读入顶点数据，再利用循环语句读入边，顺序为起点、终点，无权重，找到起点终点数据对应的顶点下标，再将边插入邻接表，最后返回图；  ·再调用深度优先搜索函数，选择从下标为0的顶点开始搜索，依次输出访问顶点的数据。   1. 时间复杂度   以邻接表存储图实现深度优先搜索的时间复杂度为O(n+e)，其中n和e分别为图的顶点数和边数，所以最终程序的时间复杂度为O(n+e)。  **第3题**   1. 实验运行结果截图 2. 数据结构   该题也是采用邻接表的存储方式存储图，与第2题中定义的图的邻接表相关结构体相同。   1. 设计思路   整体思路是：采用“邻接表存储图 + 广度优先搜索 + 队列”的形式实现。  自定义的图的初始化函数、边的插入函数、图的构建函数均与第2题中相同（需要注意的是，社交网络为无向图，所以插入边时需要插入<v1,v2>和<v2,v1>），以及广度优先搜索函数（利用队列实现）。  ·先调用图的构建函数：先创建有固定多个顶点但没有边的图，读入顶点数据，再利用循环语句读入边，顺序为起点、终点，无权重，将边插入邻接表，最后返回图；  ·再调用六度空间理论检验函数：利用for循环依次对图中的每个顶点都检验一遍六度空间，检验过程中需要反复初始化Visited[]的值以及调用BFS广度优先搜索（返回统计得到的与结点距离不超过6的结点数），再计算输出与结点距离不超过6的结点数占结点总数的百分比，精确到小数点后2位。  BFS广度优先搜索的关键代码如下：  int SDS\_BFS(LGraph Graph, int S)  {  queue<int>Q;  int V,Last,Tail;  PtrToAdjVNode F; //定义一个指向邻接表结点的指针  int Count, Level;  Visited[S] = true; //标记顶点v已经访问过了为TRUE  Count = 1; //统计符合“六度空间”理论的人数，从1开始  Level = 0; //起始点定义为第0层  Last = S; //该层只有S一个顶点，是该层被访问的最后一个顶点  Q.push(S); //将S入队列  while(!Q.empty())  {  V = Q.front();  Q.pop();  for(F = Graph->G[V].FirstEdge; F; F = F->Next)  { //对于V的每一个邻接点F->AdjV  if(!Visited[F->AdjV]) //如果F指向的结点还没有被访问  {  Visited[F->AdjV] = true; //标记F->AdjV已被访问  Count++; //人数加1  Tail = F->AdjV; //改变层尾  Q.push(F->AdjV); //将F->AdjV入队列  }  }  if(V == Last) //如果上一层的最后一个顶点弹出了  {  Level++; //层数递增  Last = Tail; //更新当前层尾为该层被访问的最后一个顶点  }  if(Level == 6) break; //如果6层遍历结束，退出搜索  }  if(!Q.empty()) Q.pop(); //释放队列所有元素  return Count; //返回统计距离不超过6的人数  }   1. 时间复杂度   以邻接表存储图，实现广度优先搜索的时间复杂度为O(n+e)，其中n和e分别为图的顶点数和边数，所以最终程序的时间复杂度为O(n+e)。 | |

**成绩：\_\_\_\_\_\_\_\_\_\_\_\_** **任课教师签名：\_\_\_\_\_\_\_\_\_\_\_\_** 2022年 月 日

**实 验 报 告 （4 ）**

|  |  |
| --- | --- |
| **实验名称**：排序算法 | **实验地点**：线上 |
| **所使用的工具软件及环境：Win10/Win7, Visual C++/Java** | |
| **一、实验目的：**  理解各类排序算法的设计思想，灵活应用排序方法解决实际问题。 | |
| **二、实验内容描述：**（填写题目内容及输入输出要求）  **1、**设计4种排序算法的实现，要求对数据升序排列，注意不得使用STL。输入第一行为算法编号（1堆排序，2冒泡排序，3直接插入排序，4希尔排序），输入第二行为待排序元素个数N，第三行为待排序数据，输出为排序结果。  输入样例：  1  12  57 40 38 11 13 34 48 75 6 19 9 7  输出样例：  6 7 9 11 13 19 34 38 40 48 57 75  **2、**给定N(N≤10^5)个整数，要求用[快速排序](https://so.csdn.net/so/search?q=%E5%BF%AB%E9%80%9F%E6%8E%92%E5%BA%8F&spm=1001.2101.3001.7020" \t "_blank)对数据进行升序排列，注意不得使用STL。输入第一行为N，第二行为待排序数据，输出为排序结果。  输入样例：  10  49 35 68 99 70 13 25 50 111 60  输出样例：  13 25 35 49 50 60 68 70 99 111  **3、**给定整数数组nums和整数k，请返回数组中第k个最大的元素。输入第一行为数组，-1为结束标志，第二行为k值。输出第k大个元素。  输入样例：  3 2 1 5 6 4 -1  2  输出结果：  5 | |
| **三、程序运行结果（说明设计思路，解释使用的数据结构，计算时间复杂度）**  **第1题**   1. 实验运行结果截图      1. 数据结构与设计思路   ·**主函数**中先输入选择排序方法的编号、待排序元素的个数，然后利用for循环待排序序列存入数组中；接下来运用switch选择调用对应排序算法的函数；最后再利用一个for循环将排好序的序列输出。  ·**堆排序**：先依据待排序序列建立最大堆，再利用for循环，将根结点与最后一个结点交换，再将新的堆重新调整为最大堆，经过n-1次循环以后，即可得到升序序列。  ·**冒泡排序**：利用双重for循环，每一趟冒泡找到所剩元素中最大的一个，并冒泡移动到最右端；若某次循环中未发生交换，则说明整个序列已经有序，则跳出循环；否则，则经过n-1次外循环，即可得到升序序列。  ·**直接插入排序**：利用双重for循环，先取出未排序元素中的第一个元素，再依次将其与已排序序列中的元素比较，若序列元素大于此元素，则将已排序序列元素右移，直到找到合适的位置将此元素插入；经过n-1次外循环，即可得到升序序列。  ·**希尔排序**：先自行定义一部分增量，再利用for循环判断——初始的增量Sedgewick[Si]不能超过待排序的序列长度n；接下来再利用多重for循环按照选中的增量进行多趟排序，最终得到升序序列。   1. 时间复杂度   堆排序的时间复杂度为O(nlogn)；  冒泡排序的时间复杂度为O(n2)；  直接插入排序的时间复杂度为O(n2)；  希尔排序的时间复杂度与增量的选取有很大的关系，增量序列的选取不同，时间复杂度也不尽相同，依照[5,3,1,0]增量有猜想认为平均时间复杂度大约为O(n7/6)。  **第2题**   1. 实验运行结果截图 2. 数据结构与设计思路   ·**主函数**中先输入待排序元素的个数，然后利用for循环待排序序列存入数组中；接下来调用快速排序函数进行排序；最后再利用一个for循环将排好序的序列输出。  ·**确定主元函数**：利用if判断函数以及交换函数，使序列满足A[Left]<=A[Center]<=A[Right]以后，将基准放到右边，最后返回基准。  ·**快速排序**：核心思想是利用递归实现。首先确定阈值Cutoff，若排序过程中剩下的元素个数低于阈值则直接改用简单排序，以提高程序效率；如果序列元素充分多则进入快速排序，首先调用“确定主元函数”选择基准，然后利用多重while循环将序列中比基准小的移到基准左边、大的移到右边，再将基准换到正确的位置；然后利用递归重复上述过程，处理左边和右边的序列，最终得到升序序列。   1. 时间复杂度   快速排序的时间复杂度为O(nlogn)，所以最终程序的时间复杂度也为O(nlogn)。  **第3题**   1. 实验运行结果截图 2. 数据结构与设计思路   本题选择运用冒泡排序解决。  ·**主函数**中先利用while循环将待排序序列存入数组中，同时统计一下待排序元素的个数N，再输入要求第k个最大的元素；接下来判断：若N<k则输出"输入的k值超过了整数序列的个数！"，若N>=k，则调用冒泡排序函数进行排序；  ·本题**冒泡排序的改良**：由于本题只用找到第k大的元素即可，因此冒泡排序不需要进行到底，只用进行k趟冒泡即可，改进的冒泡排序的关键代码如下：  void BubbleSort(int A[], int N, int k)  {  for(int P=N-1; P>=N-k; P--) // 冒泡循环k次以后停止  for(int i=0; i<P; i++) //一趟冒泡，每次找出一个最大元素，被交换到最右端  if(A[i] > A[i+1])  Swap(&A[i], &A[i+1]);  cout<<A[N-k]<<endl; //冒泡循环k次以后，输出A的第N-k+1个元素  }   1. 时间复杂度   原本冒泡排序的时间复杂度为O(n2)；改进以后时间复杂度为O(nk)，可见当k较小时，程序的运行效率有较好的提高。 | |

**成绩：\_\_\_\_\_\_\_\_\_\_\_\_** **任课教师签名：\_\_\_\_\_\_\_\_\_\_\_\_** 2022年 月 日