

学生实验实习报告册

|  |  |
| --- | --- |
| 学年学期： |  |
| 课程名称： |  |
| 学生学院： |  |
| 专业班级： |  |
| 学号： |  |
| 姓名： |  |
| 联系电话： |  |

**重庆邮电大学教务处印制**

教师评阅记录表

【重要说明】

* 学生提交报告册最终版时，**必须包含此页**，否则不予成绩评定。
* 本报告册模板内容格式除确实因为填写内容改变了布局外，**不得变更其余部分的格式**，否则不予成绩评定。

|  |  |
| --- | --- |
| **报告是否符合考核规范** | ☑ 符合 □ 不符合 |
| **报告格式是否符合标准** | ☑ 符合 □ 不符合 |
| **报告是否完成要求内容** | ☑ 是 □ 否 |
| **报告评语：** | |
| **报告成绩：** | |
| **评阅人签名（签章）** | |

目 录

* 教师评阅记录表
* 实验或实习报告

实验实习综述报告

|  |  |  |  |
| --- | --- | --- | --- |
| 课程名称 |  | 课程编号 |  |
| 开课学院 |  | | |
| 指导教师 |  | | |
| 实验实习地点 |  | | |
| 学生姓名 |  | 学生学号 |  |
| 学院专业 |  | 所在班级 |  |
| **实验一：顺序表的实现及应用**  **一、实验实习目的及要求**  **实验目的：**了解和掌握线性表的顺序存储结构；掌握用C语言上机调试线性表的基本方法；掌握线性表的基本操作：插入、删除、查找以及线性表合并等运算在顺序存储结构和链接存储结构上的运算，以及对相应算法的性能分析。  **实验要求：**给定一段程序代码，程序代码所完成的功能为：  （1）建立一个线性表；  （2）依次输入数据元素1,2,3,4,5,6,7,8,9,10；  （3）删除数据元素5；  （4）依次显示当前线性表中的数据元素。假设该线性表的数据元素个数在最坏情况下不会超过100个，要求使用顺序表。  程序中有3处错误的地方，有标识，属于逻辑错误，对照书中的代码仔细分析后，要求同学们修改错误的代码，修改后上机调试得到正确的运行结果。  **二、实验实习设备（环境）及要求（软硬件条件）**  Microsoft Visual Studio Community 2019  CodeBlocks 17.12  **三、实验实习项目、内容与步骤**  项目一：改正原程序中的错误  #include <stdio.h>  #define MaxSize 100  typedef int DataType;  typedef struct  {  DataType list[MaxSize];  int size;  } SeqList;  void ListInitiate(SeqList\* L)/\*初始化顺序表L\*/  {  L->size = 0;/\*定义初始数据元素个数\*/  }  int ListLength(SeqList L)/\*返回顺序表L的当前数据元素个数\*/  {  return L.size;  }  int ListInsert(SeqList\* L, int i, DataType x)  /\*在顺序表L的位置i（0 ≤ i ≤ size）前插入数据元素值x\*/  /\*插入成功返回1，插入失败返回0\*/  {  int j;  if (L->size >= MaxSize)  {  printf("顺序表已满无法插入! \n");  return 0;  }  else if (i < 0 || i > L->size)  {  printf("参数i不合法! \n");  return 0;  }  else  {  for (j = i; j < L->size; j--)  L->list[j + 1] = L->list[j];/\*为插入做准备\*/  L->list[i] = x;/\*插入\*/  L->size++;/\*元素个数加1\*/  return 1;  }  }  int ListDelete(SeqList\* L, int i, DataType\* x)  /\*删除顺序表L中位置i（0 ≤ i ≤ size - 1）的数据元素值并存放到参数x中\*/  /\*删除成功返回1，删除失败返回0\*/  {  int j;  if (L->size <= 0)  {  printf("顺序表已空无数据元素可删! \n");  return 0;  }  else if (i < 0 || i > L->size - 1)  {  printf("参数i不合法");  return 0;  }  else  {  \*x = L->list[i];/\*保存删除的元素到参数x中\*/  for (j = i + 1; j <= L->size - 1; j++)  L->list[j - 1] = L->list[j];/\*依次前移\*/  L->size--;/\*数据元素个数减1\*/  return 1;  }  }  int ListGet(SeqList L, int i, DataType\* x)  /\*取顺序表L中第i个数据元素的值存于x中，成功则返回1，失败返回0\*/  {  if (i < 0 || i > L.size - 1)  {  printf("参数i不合法! \n");  return 0;  }  else  {  \*x = L.list[i];  return 1;  }  }  void main(void)  {  SeqList myList;  int i, x;  ListInitiate(&myList);  for (i = 0; i < 10; i++)  ListInsert(&myList, i, i + 1);  ListDelete(&myList, 4, &x);  for (i = 0; i < ListLength(myList); i++)  {  ListGet(myList, i, &x);  printf("%d ", x);  }  }  项目二：编写合并函数，将两个有序的单链表合并成一个有序单链表  #include <stdio.h>  #define MaxSize 100  typedef int DataType;  typedef struct  {  DataType list[MaxSize];  int size;  } SeqList;  void ListInitiate(SeqList\* L)/\*初始化顺序表L\*/  {  L->size = 0;/\*定义初始数据元素个数\*/  }  int ListLength(SeqList L)/\*返回顺序表L的当前数据元素个数\*/  {  return L.size;  }  int ListInsert(SeqList\* L, int i, DataType x)  /\*在顺序表L的位置i（0 ≤ i ≤ size）前插入数据元素值x\*/  /\*插入成功返回1，插入失败返回0\*/  {  int j;  if (L->size >= MaxSize)  {  printf("顺序表已满无法插入! \n");  return 0;  }  else if (i < 0 || i > L->size)  {  printf("参数i不合法! \n");  return 0;  }  else  {  for (j = i; j < L->size; j--)  L->list[j + 1] = L->list[j];/\*为插入做准备\*/  L->list[i] = x;/\*插入\*/  L->size++;/\*元素个数加1\*/  return 1;  }  }  int ListGet(SeqList L, int i, DataType\* x)  /\*取顺序表L中第i个数据元素的值存于x中，成功则返回1，失败返回0\*/  {  if (i < 0 || i > L.size - 1)  {  printf("参数i不合法! \n");  return 0;  }  else  {  \*x = L.list[i];  return 1;  }  }  int Merge(SeqList\* L1, SeqList\* L2,SeqList\* L3)  //输入两个有序表L1，L2，将其按从小到大的顺序合并为一个新的有序表L3  //若操作成功，则返回1；若操作失败，则返回0  {  //合法性检查  if (L1->size + L2->size > MaxSize)  return 0;  int i = 0, j = 0, k = 0;//分别用于表1、2、3的计数  L3->size = L1->size + L2->size;//合并表长  //按从小到大的顺序合并两个有序表1,2  while (i <= L1->size && j <= L2->size)  {  if (L1->list[i] > L2->list[j])  L3->list[k++] = L2->list[j++];  else  L3->list[k++] = L1->list[i++];  }  //将剩余数据导入表3  k--;  if (i <= L1->size)  {  while (i <= L1->size)  {  L3->list[k++] = L1->list[i++];  }  }  else  {  L3->list[k++] = L2->list[j++];  }  return 1;  }  void main(void)  {  SeqList L1, L2, L3;  int i, x;  ListInitiate(&L1);  ListInitiate(&L2);  ListInitiate(&L3);  for (i = 0; i < 10; i++)  ListInsert(&L1, i, i \* 2);  for (i = 0; i < 5; i++)  ListInsert(&L2, i, i \* 2 - 1);  Merge(&L1, &L2, &L3);  for (i = 0; i < ListLength(L3); i++)  {  ListGet(L3, i, &x);  printf("%d ", x);  }  }  **四、实验实习所得结果及分析**  **项目一：改正原程序中的错误**    实验成功。  **项目二：编写合并函数，将两个有序线性表合并为一个有序表并在主函数中加以测试**  合并操作前已设定表A内的数据为[0 2 4 6 8 10 12 14 16 18]，表B的数据为[-1 1 3 5 7]，预计合并后表C的数据为[-1 0 1 2 3 4 5 6 7 8 10 12 14 16 18]。实验结果如下图所示：    实验成功。  **五、实验实习结果分析和（或）源程序调试过程**  **项目一分析：**  错误一原因：原代码为for(j = L->size; j > i; j--)，只能实现将位置i之后的元素向后移动，而位置i的元素未移动，会出现原数据丢失的情况。  错误二原因：原代码为L->list[j] = L->list[j-1]，该段代码实现的是依次后移，不能实现依次前移的目的，应改为L->list[j-1] = L->list[j]。  错误三原因：原代码为ListGet(,i,&x)，输入参数不合法，缺少线性表L。  **项目二分析：**  运行步骤：输入两个有序线性表L1，L2，遍历L1、L2：①若L1->list[i]>L2->list[j]，则将L2->list[j]插入L3中②若L1->list[i]<L2->list[j]，则将L1->list[i]插入L3中③当L1、L2其中某个表遍历完时，将另一表中的剩余数据插入L3中。  **实验二：链表的实现及应用**  **一、实验实习目的及要求**  **实验目的：**  了解和掌握线性表的链式存储结构；掌握用C语言上机调试线性表的基本方法；掌握线性表的基本操作：插入、删除、查找以及线性表合并等运算在顺序存储结构和链接存储结构上的运算，以及对相应算法的性能分析。  **实验要求：**  给定一段程序代码，程序代码所完成的功能为：（1）建立一个线性表；（2）依次输入数据元素1,2,3,4,5,6,7,8,9,10；（3）删除数据元素5；（4）依次显示当前线性表中的数据元素。假设该线性表的数据元素个数在最坏情况下不会超过100个，要求使用单链表。  程序中有3处错误的地方，有标识，属于逻辑错误，对照书中的代码仔细分析后，要求同学们修改错误的代码，上机调试并得到正确的运行结果。  **二、实验实习设备（环境）及要求（软硬件条件）**  Microsoft Visual Studio Community 2019  CodeBlocks 17.12  **三、实验实习项目、内容与步骤**  项目一：改正原程序中的错误  #include <stdio.h>  #include <stdlib.h>  #include <malloc.h>  typedef int DataType;  typedef struct Node  {  DataType data;  struct Node\* next;  } SLNode;  void ListInitiate(SLNode\*\* head)/\*初始化\*/  {  /\*如果有内存空间，申请头结点空间并使头指针head指向头结点\*/  if ((\*head = (SLNode\*)malloc(sizeof(SLNode))) == NULL) exit(1);  (\*head)->next = NULL;  }  int ListLength(SLNode\* head) /\* 单链表的长度\*/  {  SLNode\* p = head;/\*p指向首元结点\*/  int size = 0;/\*size初始为0\*/  while (p->next != NULL)  {  p = p->next;  size++;  }  return size;  }  int ListInsert(SLNode\* head, int i, DataType x)  /\*在带头结点的单链表head的数据元素ai（0 ≤ i ≤ size）结点前\*/  /\*插入一个存放数据元素x的结点\*/  {  SLNode\* p, \* q;  int j;  p = head; /\*p指向首元结点\*/  j = -1;/\*j初始为-1\*/  while (p->next != NULL && j < i - 1)  /\*最终让指针p指向数据元素ai-1结点\*/  {  p = p->next;  j++;  }  if (j != i - 1)  {  printf("插入位置参数错！");  return 0;  }  /\*生成新结点由指针q指示\*/  if ((q = (SLNode\*)malloc(sizeof(SLNode))) == NULL) exit(1);  q->data = x;  //此段程序有一处错误  q->next = p->next;/\*给指针q->next赋值\*/  p->next = q;/\*给指针p->next重新赋值\*/  return 1;  }  int ListDelete(SLNode\* head, int i, DataType\* x)  /\*删除带头结点的单链表head的数据元素ai（0 ≤ i ≤ size - 1）结点\*/  /\*删除结点的数据元素域值由x带回。删除成功时返回1；失败返回0\*/  {  SLNode\* p, \* s;  int j;  p = head; /\*p指向首元结点\*/  j = -1;/\*j初始为-1\*/  while (p->next != NULL && p->next->next != NULL && j < i - 1)  /\*最终让指针p指向数据元素ai-1结点\*/  {  p = p->next;  j++;  }  if (j != i - 1)  {  printf("删除位置参数错！");  return 0;  }  //此段程序有一处错误  s = p->next; /\*指针s指向数据元素ai结点\*/  \*x = s->data;/\*把指针s所指结点的数据元素域值赋予x\*/  p->next = s->next;/\*把数据元素ai结点从单链表中删除\*/  free(s);/\*释放指针s所指结点的内存空间\*/  return 1;  }  int ListGet(SLNode\* head, int i, DataType\* x)  /\*取数据元素ai和删除函数类同，只是不删除数据元素ai结点\*/  {  SLNode\* p;  int j;  p = head;  j = -1;  while (p->next != NULL && j < i)  {  p = p->next; j++;  }  if (j != i)  {  printf("取元素位置参数错！");  return 0;  }  //此段程序有一处错误  \*x = p->data;  return 1;  }  void Destroy(SLNode\*\* head)  {  SLNode\* p, \* p1;  p = \*head;  while (p != NULL)  {  p1 = p;  p = p->next;  free(p1);  }  \*head = NULL;  }  void main(void)  {  SLNode\* head;  int i, x;  ListInitiate(&head);/\*初始化\*/  for (i = 0; i < 10; i++)  {  if (ListInsert(head, i, i + 1) == 0) /\*插入10个数据元素\*/  {  printf("错误! \n");  return;  }  }  if (ListDelete(head, 4, &x) == 0) /\*删除数据元素5\*/  {  printf("错误! \n");  return;  }  for (i = 0; i < ListLength(head); i++)  {  if (ListGet(head, i, &x) == 0) /\*取元素\*/  {  printf("错误! \n");  return;  }  else printf("%d ", x);/\*显示数据元素\*/  }  Destroy(&head);  }  项目二：编写合并函数，将两个有序的单链表合并成一个有序单链表  #include <stdio.h>  #include <stdlib.h>  typedef int ElemType;  typedef struct Polynode  {  ElemType Data;  struct Polynode\* next;  }Polynode, \* Polylist;  Polylist Create(int n) //尾插法创建单链表  {  Polylist Head; //头结点  Polynode\* r;  Head = (Polylist)malloc(sizeof(Polynode));  Head->next = NULL;  r = Head;  //录入数据  for (int i = 0; i < n; i++)  {  Polylist p = (Polylist)malloc(sizeof(Polynode));  printf("请输入第%d个数：", i + 1);  scanf("%d", &p->Data);  p->next = NULL;  r->next = p;  r = p;  }  return Head;  }  Polylist Merge(Polylist ListA, Polylist ListB) //链表合并  //将两个有序链表ListA、ListB按从小到大的顺序合并为一个新的有序链表ListC  {  Polylist p, q, newp, temp, s, r; //newp为结果链表的头结点，r为尾指针  newp = (Polylist)malloc(sizeof(Polynode));  p = ListA->next;  q = ListB->next;  newp->next = NULL;  r = newp;  while (p != NULL && q != NULL)  {  //链表A的数据大于链表B的数据，将B的数据存入新多项式中  if (p->Data > q->Data)  {  s = (Polylist)malloc(sizeof(Polynode));  s->Data = q->Data;  r->next = s;  r = s;  //释放多余结点内存空间  temp = q;  q = q->next;  free(temp);  }  //链表A的数据小于链表B的数据，将A的数据存入新多项式中  else  {  s = (Polylist)malloc(sizeof(Polynode));  s->Data = p->Data;  r->next = s;  r = s;  //释放多余结点内存空间  temp = p;  p = p->next;  free(temp);  }  }  //将剩余结点连至新的链表  if (p == 0) r->next = q;  else r->next = p;  return newp;  }  void PrintList(Polylist L) //打印结果链表  {  Polylist p = L->next;  while (p != NULL)  {  printf("%d ", p->Data);  p = p->next;  }  }  int main()  {  //输入数据  int n;  Polylist List1, List2, List3;  printf("输入第一个单链表的数据个数：");  scanf("%d", &n);  List1 = Create(n);  printf("输入第二个单链表的数据个数：");  scanf("%d", &n);  List2 = Create(n);  List3 = Merge(List1, List2);  printf("结果为：");  PrintList(List3);  return 0;  }  **四、实验实习所得结果及分析**  项目一：    实验成功。  项目二：  合并操作前已设定表A内的数据为[1,3,5,7,9]，表B的数据为[2,4,6,8,10]，预计合并后表C的数据为[1,2,3,4,5,6,7,8,9,10]。实验结果如下图所示：    实验成功。  **五、实验实习结果分析和（或）源程序调试过程**  **项目一分析：**  错误一原因：原代码为p->next = q->next；p->next = q，相当于两次改变p->next结点，丢失了p结点后的剩余结点，应改为q->next = p->next；p->next = q。  错误二原因：原代码为s->next = p，只能实现s结点的下一个结点为数据元素ai结点，不能实现指针s指向数据元素ai结点，应改为s=p->next。  错误三原因：原代码为\*x = p->next，等号两端数据类型不匹配，要实现获取元素的功能，应改为\*x = p->data。  **项目二分析：**  运行步骤：①输入两个有序链表ListA、ListB②遍历ListA、ListB，  若链表A的数据大于链表B的数据，则将B的数据存入新多项式中；若链表A的数据小于链表B的数据，则将A的数据存入新多项式中③将剩余结点连至新的链表。  **实验三：栈的实现及应用**  **一、实验实习目的及要求**  **实验目的：**  1.掌握栈的存储表示和实现  2.掌握栈的基本操作实现。  3.掌握栈在解决实际问题中的应用。  **实验要求：**  问题描述：设计一个程序，演示用算符优先法对算术表达式求值的过程。利用算符优先关系，实现对算术四则混合运算表达式的求值。  （1）输入的形式：表达式，例如2\*(3+4)#  包含的运算符只能有'+' 、'-' 、'\*' 、'/' 、'('、 ')'，“#”代表输入结束符；  （2）输出的形式：运算结果，例如2\*(3+4)=14； （3）程序所能达到的功能：对表达式求值并输出。  **二、实验实习设备（环境）及要求（软硬件条件）**  Microsoft Visual Studio Community 2019  CodeBlocks 17.12  **三、实验实习项目、内容与步骤**  #include<stdio.h>  #include<string.h>  #include<ctype.h>  #include<stdlib.h>  #define N 40  //中序转后序  typedef char NodeType;  typedef struct stack//栈  {  NodeType data[N];  int top;  } STACK;  void Push(STACK\* stack, NodeType data)//入栈  {  if (stack->top <= N)  memcpy(&stack->data[stack->top], &data, sizeof(NodeType));  stack->top = stack->top + 1;  printf("\nPush:%c\n", data);  }  NodeType Pop(STACK\* stack)//出栈  {  if (stack->top > 0)  {  stack->top = stack->top - 1;  return stack->data[stack->top];  }  else  {  return '0';  }  }  int isops(char a)//是否为符号  {  if (a == '+' || a == '-' || a == '\*' || a == '/' || a == '(' || a == ')')  {  return 1;  }  else  {  return 0;  }  }  int isdig(char a)//是否为数字  {  if (a >= '0' && a <= '9')  {  return 1;  }  else  {  return 0;  }  }  int opsadv(char a)//符号优先级  {  if (a == '+' || a == '-')  {  return 1;  }  if (a == '\*' || a == '/')  {  return 2;  }  }  void result(char a, char b[])  {  static int k = 0;  b[k] = a;  k++;  }  int IsEmpty(STACK s)  {  if (s.top == 0)  return 1;  else  return 0;  }  //后序表达式求值所用函数  typedef int NodeType1;  typedef struct stack1 //计算栈  {  NodeType1 data1[N];  int top1;  } STACK1;  void Push1(STACK1\* stack1, NodeType1 data1)  {  memcpy(&stack1->data1[stack1->top1], &data1, sizeof(NodeType1));  stack1->top1 = stack1->top1 + 1;  }  NodeType1 Pop1(STACK1\* stack1)  {  stack1->top1 = stack1->top1 - 1;  return stack1->data1[stack1->top1];  }  NodeType1 OpInt(int w1, int w2, char op)  {  NodeType1 resu;  switch (op)  {  case '+':  resu = w1 + w2;  break;  case '-':  resu = w1 - w2;  break;  case '\*':  resu = w1 \* w2;  break;  case '/':  resu = w1 / w2;  break;  }  return resu;  }  int Change(int q)  {  if (q >= 48 && q <= 57)  q = q - 48;  return q;  }  //用户界面  void main()  {  //转换所用  char word[N];//存储用户输入的表达式  char res[N];//用于存储转换后的表达式  char cun;  STACK stack;  stack.top = 0;  printf("Please imput an arithmetic:\n");  fgets(word, N, stdin);  int m = 0;//记录入栈次数  int m1 = 0;//记录（后入栈次数  int i, j, n, q, p;  NodeType d1 = 'A', d2 = 'A';//辅助出入栈  for (i = 0; word[i] != '\0'; i++)  {  if (isdig(word[i]))  {  result(word[i], res);  //printf("%c",word[i]);  }  else if (isops(word[i]))  {  d1 = word[i];  if (d1 == '(')  {  //d2=d1;  Push(&stack, d1);  m1++;  m--;//未改变d2的值，m会多1  }  else if (d1 == ')')  {  while (stack.data[stack.top - 1] != '(')  {  cun = Pop(&stack);  result(cun, res);  }  cun = Pop(&stack);  d2 = 'A';  m1 = 0;  }  else if (d2 == 'A')  {  d2 = d1;  Push(&stack, d1);  m++;  if (m1 != 0)  {  m1++;  }  }  else if (opsadv(d1) > opsadv(d2))//  {  Push(&stack, d1);  m++;  if (m1 != 0)  {  m1++;  }  d2 = d1;  }  else if (opsadv(d1) <= opsadv(d2))  {  for (j = 0; j < m; j++)  {  cun = Pop(&stack);  result(cun, res);  //printf("%c",cun);  }  Push(&stack, d1);  m = 1;  d2 = d1;  }  }  }  while (!IsEmpty(stack))  {  cun = Pop(&stack);  result(cun, res);  }  result('\0', res);  putchar('\n');  printf("Reverse order expression is:");  for (p = 0; res[p] != '\0'; p++)//打印存入数组中的内容  {  printf("%c ", res[p]);  }  printf("\n");  putchar('\n');  //求值所用  int s;  char calculate[N];  NodeType1 w1, w2, w3;  STACK1 stack1;  stack1.top1 = 0;  for (s = 0; res[s] != '\0'; s++) //遍历数组  {  if (isdig(res[s]))  {  w1 = Change((int)(res[s]));  Push1(&stack1, w1);//是数字则入栈  }  else  {  w2 = Pop1(&stack1);  //printf("w2=%d\n",w2);  w1 = Pop1(&stack1);//取出两个数字  //printf("w1=%d\n",w1);  w3 = OpInt(w1, w2, res[s]);  //printf("w3=%d\n",w3);//计算过程  Push1(&stack1, w3);//结果入栈  }  }  w1 = Pop1(&stack1);  printf("The result is:%d\n", w1);  }  **四、实验实习所得结果及分析**    **五、实验实习结果分析和（或）源程序调试过程**  算术表达式求值的基本思路：  ①转换为后序表达式  读入一个字符：  (1) 若为“（”，则压入栈内；若为“）”，则检查栈内是否为空。若栈非空，则依次弹出栈中元素直至遇到“（”；若栈为空，则发生错误并进行相关处理。  (2) 若为运算符，则检查栈是否为空、栈顶是否为括号。当上述条件均成立时，若栈顶元素优先级高于或等于此运算符，则将栈中元素依次弹出直至栈顶元素优先级小于此运算符，并将此运算符压入栈中；若栈顶元素优先级小于此运算符，则此运算符入栈。  (3)若为数字，则直接输出  ②求值  按顺序遍历整个表达式，若遇到操作数（假设都是二元运算符），则入栈；若遇到操作符，则连续弹出两个操作数并执行相应的计算，然后将其结果入栈。重复以上过程，直至数组遍历完，栈内只剩下一个操作数时，此操作数即为最终的运算结果，输出该结果。  **实验四：队列的实现及应用**  **一、实验实习目的及要求**  实验目的：  1.掌握队列的存储表示和实现。  2.掌握队列的基本操作实现。  3.掌握队列在解决实际问题中的应用。  实验要求：  利用队列模拟服务台前的排队现象问题。  问题描述：某银行有一个客户办理业务站，在单位时间内随机地有客户到达，设每位客户的业务办理时间是某个范围的随机值。设只有一个窗口，一位业务人员，要求程序模拟统计在设定时间内，业务人员的总空闲时间和客户的平均等待时间。假定模拟数据已按客户到达的先后顺序依次存于某个正文数据文件中，对应每位客户有两个数据：到达时间和需要办理业务的时间，文本文件内容如：10 20 23 10 45 5 55 10 58 15 65 10。  **二、实验实习设备（环境）及要求（软硬件条件）**  Microsoft Visual Studio Community 2019  CodeBlocks 17.12  **三、实验实习项目、内容与步骤**  #include<stdio.h>  #include<stdlib.h>  typedef struct {  int arrive;//到达时间  int treat;//需要占用的时间  }QNODE;  typedef struct node {  QNODE data;  struct node\* next;  }LNODE;  LNODE\* front, \* rear;  void inQueue(QNODE e)//入队  {  LNODE\* p = (LNODE\*)malloc(sizeof(LNODE));  p->data = e;  p->next = NULL;  if (front == NULL)  front = rear = p;  else  {  rear->next = p;  rear = p;  }  }  int outQueue(QNODE\* e) // 出队  {  LNODE \* s;  if (front == NULL)  return 0;//队空  \*e = front->data;  s = front;  front = front->next;  if (front == NULL)  rear = NULL;  free(s);  return 1;  }  int main(void)  {  FILE\* fp;  QNODE temp, former;  //spareTime表示业务员的总空闲时间，clock表示当前时间，waitTime表示客户的总等待时间  //count客户总数  int have = 0, spareTime = 0, clock = 0, waitTime = 0, count = 0, finishTime;  if ((fp = fopen("data.txt", "r")) == NULL)  {  printf("文件打开出错");  return 0;  }  front = rear = NULL;  have = fscanf(fp, "%d %d", &temp.arrive, &temp.treat);  do {  if (front == NULL && have - 2 == 0)  {  spareTime = spareTime + temp.arrive - clock;  clock = temp.arrive;  inQueue(temp);  have = fscanf(fp, "%d %d", &temp.arrive, &temp.treat);  }  count++;  outQueue(&former);  waitTime = waitTime + (clock - former.arrive);  finishTime = clock + former.treat;  while (have - 2 == 0 && temp.arrive <= finishTime)  {  inQueue(temp);  have = fscanf(fp, "%d %d", &temp.arrive, &temp.treat);  }  clock = finishTime;  } while (have - 2 == 0 || front != NULL);  printf("模拟总时间：%d\n业务员等待时间：%d\n客户平均等待时间：%f\n", clock, spareTime, (double)waitTime / count);  printf("客户人数：%d\n总等待时间：%d\n", count, waitTime);  return 0;  }  **四、实验实习所得结果及分析**    **五、实验实习结果分析和（或）源程序调试过程**  算法设计：   { 设置统计初值：业务员等待时间，客户总的待时间，客户总人数等     设置当前时钟clock时间为0；//用变量clock来模拟当前时间.     打开数据文件，准备读；     读入第一位客户信息于暂存变量中；   do{//约定每轮循环，处理完一位客户  if(等待队列为空，并且还有客户)  {   //等待队列为空时     累计业务员总等待时间；     时钟推进到暂存变量中的客户的到达时间；     暂存变量中的客户信息进队；     读取下一位客户信息于暂存变量；     }     从等待队列出队一位客户；     累计客户人数；     将该客户的等待时间累计到客户的总等待时间；//=当前时间-客户到达时间     设定当前客户的业务办理结束时间；//=当前时间+客户办理业务所需时间    while(下一位客户的到达时间在当前客户处理结束之前)  {  暂存变量中的客户信息进队；   读取下一位客户信息于暂存变量；      }    时钟推进到当前客户办理结束时间；   }while(还有未处理的客户)；  计算统计结果，并输出；  **实验五：二叉树操作及应用**  **一、实验实习目的及要求**  实验目的：  掌握二叉树的定义、结构特征，以及各种存储结构的特点及使用范围，各种遍历算法。掌握用指针类型描述、访问和处理二叉树的运算。账务前序或中序的非递归遍历算法。  实验要求：  有如下二叉树：  ∧  A  B  ∧  C  ∧  D  ∧  E  ∧  ∧  F  ∧  ∧  G  ∧  根  程序代码给出了该二叉树的链式存储结构的建立、前序、中序、后序遍历的算法，同时也给出了查询“E”是否在二叉树的代码。代码有三处错误，有标识，属于逻辑错误，对照书中的代码仔细分析后，请修改了在电脑里运行。  **二、实验实习设备（环境）及要求（软硬件条件）**  Microsoft Visual Studio Community 2019  CodeBlocks 17.12  **三、实验实习项目、内容与步骤**  项目一：  #include <stdlib.h>  #include <stdio.h>  typedef char DataType;  typedef struct Node  {  DataType data;/\*数据域\*/  struct Node\* leftChild;/\*左子树指针\*/  struct Node\* rightChild;/\*右子树指针\*/  }BiTreeNode;/\*结点的结构体定义\*/  /\*初始化创建二叉树的头结点\*/  void Initiate(BiTreeNode\*\* root)  {  \*root = (BiTreeNode\*)malloc(sizeof(BiTreeNode));  (\*root)->leftChild = NULL;  (\*root)->rightChild = NULL;  }  void Destroy(BiTreeNode\*\* root)  {  if ((\*root) != NULL && (\*root)->leftChild != NULL)  Destroy(&(\*root)->leftChild);  if ((\*root) != NULL && (\*root)->rightChild != NULL)  Destroy(&(\*root)->rightChild);  free(\*root);  }  /\*若当前结点curr非空，在curr的左子树插入元素值为x的新结点\*/  /\*原curr所指结点的左子树成为新插入结点的左子树\*/  /\*若插入成功返回新插入结点的指针，否则返回空指针\*/  BiTreeNode\* InsertLeftNode(BiTreeNode\* curr, DataType x)  {  BiTreeNode\* s, \* t;  if (curr == NULL) return NULL;  t = curr->leftChild;/\*保存原curr所指结点的左子树指针\*/  s = (BiTreeNode\*)malloc(sizeof(BiTreeNode));  s->data = x;  s->leftChild = t;/\*新插入结点的左子树为原curr的左子树\*/  s->rightChild = NULL;  curr->leftChild = s;/\*新结点成为curr的左子树\*/  return curr->leftChild;/\*返回新插入结点的指针\*/  }  /\*若当前结点curr非空，在curr的右子树插入元素值为x的新结点\*/  /\*原curr所指结点的右子树成为新插入结点的右子树\*/  /\*若插入成功返回新插入结点的指针，否则返回空指针\*/  BiTreeNode\* InsertRightNode(BiTreeNode\* curr, DataType x)  {  BiTreeNode\* s, \* t;  if (curr == NULL) return NULL;  t = curr->rightChild;/\*保存原curr所指结点的右子树指针\*/  s = (BiTreeNode\*)malloc(sizeof(BiTreeNode));  s->data = x;  s->rightChild = t;/\*新插入结点的右子树为原curr的右子树\*/  s->leftChild = NULL;  curr->rightChild = s;/\*新结点成为curr的右子树\*/  return curr->rightChild;/\*返回新插入结点的指针\*/  }  void PreOrder(BiTreeNode\* t, void visit(DataType item))  //使用visit(item)函数前序遍历二叉树t  {  if (t != NULL)  {  visit(t->data);  PreOrder(t->leftChild, visit);  PreOrder(t->rightChild, visit);  }  }  void InOrder(BiTreeNode\* t, void visit(DataType item))  //使用visit(item)函数中序遍历二叉树t  {  if (t != NULL)  {  InOrder(t->leftChild, visit);  visit(t->data);  InOrder(t->rightChild, visit);  }  }  void PostOrder(BiTreeNode\* t, void visit(DataType item))  //使用visit(item)函数后序遍历二叉树t  {  if (t != NULL)  {  PostOrder(t->leftChild, visit);  PostOrder(t->rightChild, visit);  visit(t->data);  }  }  void Visit(DataType item)  {  printf("%c ", item);  }  BiTreeNode\* Search(BiTreeNode\* root, DataType x)//需找元素x是否在二叉树中  {  BiTreeNode\* find = NULL;  if (root != NULL)  {  if (root->data == x)  find = root;  else  {  find = Search(root->leftChild, x);  if (find == NULL)  find = Search(root->rightChild, x);  }  }  return find;  }  void main(void)  {  BiTreeNode\* root, \* p, \* pp, \* find;  char x = 'E';  Initiate(&root);  p = InsertLeftNode(root, 'A');  p = InsertLeftNode(p, 'B');  p = InsertLeftNode(p, 'D');  p = InsertRightNode(p, 'G');  p = InsertRightNode(root->leftChild, 'C');  pp = p;  InsertLeftNode(p, 'E');  InsertRightNode(pp, 'F');  printf("前序遍历：");  PreOrder(root->leftChild, Visit);  printf("\n中序遍历：");  InOrder(root->leftChild, Visit);  printf("\n后序遍历：");  PostOrder(root->leftChild, Visit);  find = Search(root, x);  if (find != NULL)  printf("\n数据元素%c在二叉树中 \n", x);  else  printf("\n数据元素%c不在二叉树中 \n", x);  Destroy(&root);  }  项目二：  #include <stdlib.h>  #include <stdio.h>  typedef char DataType;  typedef struct Node  {  DataType data;/\*数据域\*/  struct Node\* leftChild;/\*左子树指针\*/  struct Node\* rightChild;/\*右子树指针\*/  }BiTreeNode;/\*结点的结构体定义\*/  int top = -1;  void Push(BiTreeNode\*\* s, BiTreeNode\* e)  /\*入栈\*/  {  s[++top] = e;  }  void Pop()  /\*出栈\*/  {  if (top == -1)  {  return;  }  top--;  }  BiTreeNode\* getTop(BiTreeNode\*\* s)  /\*取顶\*/  {  return s[top];  }  /\*初始化创建二叉树的头结点\*/  void Initiate(BiTreeNode\*\* root)  {  \*root = (BiTreeNode\*)malloc(sizeof(BiTreeNode));  (\*root)->leftChild = NULL;  (\*root)->rightChild = NULL;  }  void Destroy(BiTreeNode\*\* root)  {  if ((\*root) != NULL && (\*root)->leftChild != NULL)  Destroy(&(\*root)->leftChild);  if ((\*root) != NULL && (\*root)->rightChild != NULL)  Destroy(&(\*root)->rightChild);  free(\*root);  }  /\*若当前结点curr非空，在curr的左子树插入元素值为x的新结点\*/  /\*原curr所指结点的左子树成为新插入结点的左子树\*/  /\*若插入成功返回新插入结点的指针，否则返回空指针\*/  BiTreeNode\* InsertLeftNode(BiTreeNode\* curr, DataType x)  {  BiTreeNode\* s, \* t;  if (curr == NULL) return NULL;  t = curr->leftChild;/\*保存原curr所指结点的左子树指针\*/  s = (BiTreeNode\*)malloc(sizeof(BiTreeNode));  s->data = x;  s->leftChild = t;/\*新插入结点的左子树为原curr的左子树\*/  s->rightChild = NULL;  curr->leftChild = s;/\*新结点成为curr的左子树\*/  return curr->leftChild;/\*返回新插入结点的指针\*/  }  /\*若当前结点curr非空，在curr的右子树插入元素值为x的新结点\*/  /\*原curr所指结点的右子树成为新插入结点的右子树\*/  /\*若插入成功返回新插入结点的指针，否则返回空指针\*/  BiTreeNode\* InsertRightNode(BiTreeNode\* curr, DataType x)  {  BiTreeNode\* s, \* t;  if (curr == NULL) return NULL;  t = curr->rightChild;/\*保存原curr所指结点的右子树指针\*/  s = (BiTreeNode\*)malloc(sizeof(BiTreeNode));  s->data = x;  s->rightChild = t;/\*新插入结点的右子树为原curr的右子树\*/  s->leftChild = NULL;  curr->rightChild = s;/\*新结点成为curr的右子树\*/  return curr->rightChild;/\*返回新插入结点的指针\*/  }  void PreOrder(BiTreeNode\* t, void visit(DataType item))  //使用visit(item)函数前序遍历二叉树t  {  if (t != NULL)  {  visit(t->data);  PreOrder(t->leftChild, visit);  PreOrder(t->rightChild, visit);  }  }  void Visit(DataType item)  {  printf("%c ", item);  }  void NRPreOrder(BiTreeNode\* t)  //使用非递归算法实现前序遍历二叉树t，基于栈实现  {  BiTreeNode\* s[10];  BiTreeNode\* p;  Push(s, t);  while (top != -1)  {  p = getTop(s);  Pop(s);  while (p != NULL)  {  Visit(p->data);  if (p->rightChild != NULL)  Push(s,p->rightChild);  p = p->leftChild;  }  }  }  void main(void)  {  BiTreeNode\* root, \* p, \* pp, \* find;  char x = 'E';  Initiate(&root);  p = InsertLeftNode(root, 'A');  p = InsertLeftNode(p, 'B');  p = InsertLeftNode(p, 'D');  p = InsertRightNode(p, 'G');  p = InsertRightNode(root->leftChild, 'C');  pp = p;  InsertLeftNode(p, 'E');  InsertRightNode(pp, 'F');  printf("前序遍历：");  PreOrder(root->leftChild, Visit);  printf("\n\n");  printf("非递归前序遍历：");  NRPreOrder(root->leftChild);  putchar('\n');  Destroy(&root);  }  **四、实验实习所得结果及分析**  项目一：    项目二：    **五、实验实习结果分析和（或）源程序调试过程**  前序遍历：先遍历根节点，然后遍历根节点的左子树，再遍历根节点的右子树。  中序遍历：先遍历根节点的左子树，然后遍历根节点，再遍历根节点的右子树。  后序遍历：先遍历根节点的左子树，然后遍历根节点的右子树，再遍历根节点的右子树。  **实验六、图的遍历操作及应用**  **一、实验实习目的及要求**  实验目的：  掌握有向图和无向图的概念；掌握邻接矩阵和邻接链表建立图的存储结构；掌握DFS及BFS对图的遍历操作；了解图结构在人工智能、工程等领域的广泛应用。  实验要求：  采用邻接矩阵和邻接链表作为图的存储结构，完成有向图和无向图的DFS和BFS操作。本实验给出了示例程序，其中共有4处错误，错误段均有标识，属于逻辑错误。请认真理解程序，修改程序代码，并在电脑上调试运行。  **二、实验实习设备（环境）及要求（软硬件条件）**  Microsoft Visual Studio Community 2019  CodeBlocks 17.12  **三、实验实习项目、内容与步骤**  项目一：  #define \_CRT\_SECURE\_NO\_WARNINGS  #include"stdio.h"  #include"stdlib.h"  #define MaxVertexNum 100 //定义最大顶点数  typedef struct {  char vexs[MaxVertexNum]; //顶点表  int edges[MaxVertexNum][MaxVertexNum];  //邻接矩阵，可看作边表  int n, e; //图中的顶点数n和边数e  }MGraph; //用邻接矩阵表示的图的类型  //=========建立邻接矩阵=======  void CreatMGraph(MGraph\* G)  {  int i, j, k;  char a;  printf("Input VertexNum(n) and EdgesNum(e): ");  scanf("%d,%d", &G->n, &G->e); //输入顶点数和边数  scanf("%c", &a);  printf("Input Vertex string:");  for (i = 0; i < G->n; i++)  {  scanf("%c", &a);  G->vexs[i] = a; //读入顶点信息，建立顶点表  }  for (i = 0; i < G->n; i++)  for (j = 0; j < G->n; j++)  G->edges[i][j] = 0; //初始化邻接矩阵  printf("Input edges,Creat Adjacency Matrix\n");  for (k = 0; k < G->e; k++) { //读入e条边，建立邻接矩阵  scanf("%d%d", &i, &j); //输入边（Vi，Vj）的顶点序号  G->edges[i][j] = 1;  G->edges[j][i] = 1; //若为无向图，矩阵为对称矩阵；若建立有向图，去掉该条语句  }  }  //=========定义标志向量，为全局变量=======  typedef enum { FALSE, TRUE } Boolean;  Boolean visited[MaxVertexNum];  //========DFS：深度优先遍历的递归算法======  void DFSM(MGraph\* G, int i)  { //以Vi为出发点对邻接矩阵表示的图G进行DFS搜索，邻接矩阵是0，1矩阵  int j;  printf("%c", G->vexs[i]); //访问顶点Vi  visited[i] = TRUE; //置已访问标志  for (j = 0; j < G->n; j++) //依次搜索Vi的邻接点  if (G->edges[i][j] == 1 && !visited[j])  DFSM(G, j); //（Vi，Vj）∈E，且Vj未访问过，故Vj为新出发点  }  void DFS(MGraph\* G)  { //此段代码有一处错误  int i;  for (i = 0; i < G->n; i++)  visited[i] = FALSE; //标志向量初始化  for (i = 0; i < G->n; i++)  if (!visited[i]) //Vi未访问过  DFSM(G, i); //以Vi为源点开始DFS搜索  }  //===========BFS：广度优先遍历=======  void BFS(MGraph\* G, int k)  { //以Vk为源点对用邻接矩阵表示的图G进行广度优先搜索  int i, j, f = 0, r = 0;  int cq[MaxVertexNum]; //定义队列  for (i = 0; i < G->n; i++)  visited[i] = FALSE; //标志向量初始化  for (i = 0; i < G->n; i++)  cq[i] = -1; //队列初始化  printf("%c", G->vexs[k]); //访问源点Vk  visited[k] = TRUE;  cq[r] = k; //Vk已访问，将其入队。注意，实际上是将其序号入队  while (cq[f] != -1) { //队非空则执行  i = cq[f]; f = f + 1; //Vf出队  for (j = 0; j < G->n; j++) //依次Vi的邻接点Vj  if (G->edges[i][j] == 1 && !visited[j])  { //Vj未访问 \\以下三行代码有一处错误  printf("%c", G->vexs[j]); //访问Vj  visited[j] = TRUE;  r = r + 1;  cq[r] = j; //访问过Vj入队  }  }  }  //==========main=====  void main()  {  MGraph\* G;  G = (MGraph\*)malloc(sizeof(MGraph)); //为图G申请内存空间  CreatMGraph(G); //建立邻接矩阵  printf("Print Graph DFS: ");  DFS(G); //深度优先遍历  printf("\n");  printf("Print Graph BFS: ");  BFS(G, 3); //以序号为3的顶点开始广度优先遍历  printf("\n");  }  项目二：  #define \_CRT\_SECURE\_NO\_WARNINGS  #include"stdio.h"  #include"stdlib.h"  #define MaxVertexNum 50 //定义最大顶点数  typedef struct node { //边表结点  int adjvex; //邻接点域  struct node\* next; //链域  }EdgeNode;  typedef struct vnode { //顶点表结点  char vertex; //顶点域  EdgeNode\* firstedge; //边表头指针  }VertexNode;  typedef VertexNode AdjList[MaxVertexNum]; //AdjList是邻接表类型  typedef struct {  AdjList adjlist; //邻接表  int n, e; //图中当前顶点数和边数  } ALGraph; //图类型  //=========建立图的邻接表=======  void CreatALGraph(ALGraph\* G)  {  int i, j, k;  char a;  EdgeNode\* s; //定义边表结点  printf("Input VertexNum(n) and EdgesNum(e): ");  scanf("%d,%d", &G->n, &G->e); //读入顶点数和边数  scanf("%c", &a);  printf("Input Vertex string:");  for (i = 0; i < G->n; i++) //建立边表  {  scanf("%c", &a);  G->adjlist[i].vertex = a; //读入顶点信息  G->adjlist[i].firstedge = NULL; //边表置为空表  }  printf("Input edges,Creat Adjacency List\n");  for (k = 0; k < G->e; k++) { //建立边表  scanf("%d%d", &i, &j); //读入边（Vi，Vj）的顶点对序号  s = (EdgeNode\*)malloc(sizeof(EdgeNode)); //生成边表结点  s->adjvex = j; //邻接点序号为j  s->next = G->adjlist[i].firstedge;  G->adjlist[i].firstedge = s; //将新结点\*S插入顶点Vi的边表头部  s = (EdgeNode\*)malloc(sizeof(EdgeNode));  s->adjvex = i; //邻接点序号为i  s->next = G->adjlist[j].firstedge;  G->adjlist[j].firstedge = s; //将新结点\*S插入顶点Vj的边表头部  }  }  //=========定义标志向量，为全局变量=======  typedef enum { FALSE, TRUE } Boolean;  Boolean visited[MaxVertexNum];  //========DFS：深度优先遍历的递归算法======  void DFSM(ALGraph\* G, int i)  { //以Vi为出发点对邻接链表表示的图G进行DFS搜索  EdgeNode\* p;  printf("%c", G->adjlist[i].vertex); //访问顶点Vi  visited[i] = TRUE; //标记Vi已访问  p = G->adjlist[i].firstedge; //取Vi边表的头指针  while (p) { //依次搜索Vi的邻接点Vj，这里j=p->adjvex  //以下3行代码有一处错误  if (!visited[p->adjvex]) //若Vj尚未被访问  DFSM(G, p->adjvex); //则以Vj为出发点向纵深搜索  p = p->next; //找Vi的下一个邻接点  }  }  void DFS(ALGraph\* G)  {  int i;  for (i = 0; i < G->n; i++)  visited[i] = FALSE; //标志向量初始化  for (i = 0; i < G->n; i++)  if (!visited[i]) //Vi未访问过  DFSM(G, i); //以Vi为源点开始DFS搜索  }  //==========BFS：广度优先遍历=========  void BFS(ALGraph\* G, int k)  { //以Vk为源点对用邻接链表表示的图G进行广度优先搜索  int i, f = 0, r = 0;  EdgeNode\* p;  int cq[MaxVertexNum]; //定义FIFO队列  for (i = 0; i < G->n; i++)  visited[i] = FALSE; //标志向量初始化  for (i = 0; i <= G->n; i++)  cq[i] = -1; //初始化标志向量  printf("%c", G->adjlist[k].vertex); //访问源点Vk  visited[k] = TRUE;  cq[r] = k; //Vk已访问，将其入队。注意，实际上是将其序号入队  while (cq[f] != -1) { //队列非空则执行  i = cq[f]; f = f + 1; //Vi出队  p = G->adjlist[i].firstedge; //取Vi的边表头指针  while (p) { //依次搜索Vi的邻接点Vj（令p->adjvex=j）  if (!visited[p->adjvex]) { //若Vj未访问过  printf("%c", G->adjlist[p->adjvex].vertex); //访问Vj  visited[p->adjvex] = TRUE;  //以下3行代码有一处错误  r = r + 1;  cq[r] = p->adjvex; //访问过的Vj入队  }  p = p->next; //找Vi的下一个邻接点  }  }//endwhile  }  //==========主函数===========  void main()  {  int i;  ALGraph\* G;  G = (ALGraph\*)malloc(sizeof(ALGraph));  CreatALGraph(G);  printf("Print Graph DFS: ");  DFS(G);  printf("\n");  printf("Print Graph BFS: ");  BFS(G, 3);  printf("\n");  }  **四、实验实习所得结果及分析**  项目一：    项目二：    **五、实验实习结果分析和（或）源程序调试过程**  **深度优先搜索法DFS的基本思想：**从图G中某个顶点出发，首先访问，然后选择一个与相邻且没被访问过的顶点访问，再从出发选择一个与相邻且没被访问过的顶点访问，……依次继续。如果当前被访问过的顶点的所有邻接顶点都已被访问，则回退到已被访问的顶点序列中最后一个拥有未被访问的相邻顶点的顶点W，从W出发按同样方法向前遍历。直到图中所有的顶点都被访问。  **广度优先算法BFS的基本思想：**从图G中某个顶点出发，首先访问，然后访问与相邻的所有未被访问过的顶点，，……，；再依次访问与，，……，相邻的起且未被访问过的的所有顶点。如此继续，直到访问完图中的所有顶点。  项目一：项目一使用的二维矩阵对图进行储存，需要设置一个顶点数组储存图中的顶点，二维数组用于储存边。如果两个顶点直接有边连接，则将二维数组中对应的位置赋值1（无权图）或权值（有权图）。  项目二：项目二使用的是邻接表对图进行储存。该方法为每个顶点设置一条链表，用于储存该顶点所连接的点以及这个点所连接的点 实验七、查找算法的实现 **一、实验实习目的及要求**  实验目的：  掌握顺序和二分查找算法的基本思想及其实现方法。  实验要求：  问题描述：对给定的任意数组（设其长度为n），分别用顺序和二分查找方法在此数组中查找与给定值k相等的元素 。  **二、实验实习设备（环境）及要求（软硬件条件）**  Microsoft Visual Studio Community 2019  CodeBlocks 17.12  **三、实验实习项目、内容与步骤**  #define \_CRT\_SECURE\_NO\_WARNINGS  #include <stdio.h>  #include <stdlib.h>  typedef int KeyType;  typedef struct  {  KeyType key;//关键字  char rec;//记录  }SeqList[8];  //顺序查找+'  int SeqSearch(SeqList R, int n, KeyType k)  {  int i = 0;  while (i < n && R[i].key != k)  {  printf("%d ", R[i].key);  i++;  }  if (i >= n)  return -1;  else  {  printf("%d ", R[i].key);  return i;  }  }  //二分查找  int BinSearch(SeqList R, int n, KeyType k)  {  int low = 0, high = n - 1, mid, count = 0;  while (low <= high)  {  mid = (low + high) / 2;  printf("第%d次查找：在[%d,%d]中找到元素R[%d]:%d\n", ++count, low, high, mid, R[mid].key);  if (R[mid].key == k)  return mid;  if (R[mid].key > k)  high = mid - 1;  else  low = mid + 1;  }  return -1;  }  int main()  {  SeqList R;  //顺序表赋值  for (int i = 0; i < 8; i++)  {  R[i].key = i + 1;  R[i].rec = i + 65;  }  //打印顺序表存储的记录  printf("顺序表中存储的记录为：\n");  for (int i = 0; i < 8; i++)  printf("%d\t%c,%d\n", i + 1, R[i].rec, R[i].key);  //使用顺序查找，查找关键字为4的记录  printf("=======================================================\n");  printf("使用顺序查找，查找关键字为4的记录\n");  int pos\_1;  pos\_1 = SeqSearch(R, 8, 4);  printf("\n关键字为4的记录位置为：%d，该条记录为：%c，%d\n", pos\_1 + 1, R[pos\_1].rec, R[pos\_1].key);  //使用二分查找，查找关键字为7的记录  printf("=======================================================\n");  printf("使用二分查找，查找关键字为7的记录\n\n");  int pos\_2;  pos\_2 = BinSearch(R, 8, 7);  printf("\n关键字为7的记录位置为：%d，该条记录为：%c，%d\n", pos\_2 + 1, R[pos\_2].rec, R[pos\_2].key);  return 0;  }  **四、实验实习所得结果及分析**    **五、实验实习结果分析和（或）源程序调试过程**  顺序查找的基本思想：  从表中的最后一个数据元素开始，逐个同记录的关键字做比较，如果匹配成功，则查找成功；反之，如果直到表中第一个关键字查找完也没有成功匹配，则查找失败。  二分查找的基本思想：指针low和high分别指向查找表的第一个关键字和最后一个关键字，指针mid指向处于low和high指针中间位置的关键字。在查找的过程中每次都同mid指向的关键字进行比较。若待查找值大于mid，则将low 设置为当前mid位置的后一位，mid设置为新low和high的中间位置；若待查找值小于mid，则将high设置为mid位置的前一位，并将mid设置为新high和low的中间位置。重复上述步骤，即可查找出待查找值的位置，但查找表中的数据必须是有序的。  **实验八、查找算法的实现**  **一、实验实习目的及要求**  **一、实验目的**  1.掌握常用的排序方法，并掌握用高级语言实现排序算法的方法；  2.深刻理解排序的定义和各种排序方法的特点，并能加以灵活应用；  3.了解各种方法的排序过程及其时间复杂度的分析方法。  **二、实验要求**  统计成绩：给出n个学生的考试成绩表，每条信息由姓名和分数组成，试设计一个算法：  （1）按分数高低次序，打印出每个学生在考试中获得的名次，分数相同的为同一名次；  （2）按名次列出每个学生的姓名与分数。  **二、实验实习设备（环境）及要求（软硬件条件）**  Microsoft Visual Studio Community 2019  CodeBlocks 17.12  **三、实验实习项目、内容与步骤**  #define \_CRT\_NO\_SECURE\_WARNINGS  #include <stdio.h>  #include <stdlib.h>  #include <time.h>  //定义结构体  struct Student  {  char name;//姓名  int score;//分数  }stu[11];  void BubbleSort()  {  //冒泡排序  for (int i = 0; i < 10; i++)  {  for (int j = 0; j < 9 - i; j++)  {  if (stu[j].score < stu[j + 1].score)  {  stu[10] = stu[j];  stu[j] = stu[j + 1];  stu[j + 1] = stu[10];  }  }  }  for (int i = 0; i < 10; i++) {  printf("%c,%d\t", stu[i].name, stu[i].score);  }  }  void SelectionSort()  {  int maxValue = stu[0].score;  //选择排序  for (int i = 0; i < 9; i++)  {  for (int j = i; j < 10; j++)  {  if (stu[i].score < stu[j].score)  {  stu[10] = stu[j];  stu[j] = stu[i];  stu[i] = stu[10];  }  }  }  for (int i = 0; i < 10; i++) {  printf("%c,%d\t", stu[i].name, stu[i].score);  }  }  int main()  {  //生成伪随机数，导入数据  time\_t t;  srand((unsigned)time(&t));  //冒泡排序  printf("==============================冒泡排序==============================");  for (int i = 0; i < 10; i++)  {  stu[i].name = 65 + i;  stu[i].score = rand() % 100;  }  printf("\n原始数据：\n");  for (int i = 0; i < 10; i++) {  printf("%c,%d\t", stu[i].name, stu[i].score);  }  printf("\n进行冒泡排序，结果为：\n");  BubbleSort();  printf("\n\n==============================选择排序==============================");  for (int i = 0; i < 10; i++)  {  stu[i].name = 65 + i;  stu[i].score = rand() % 100;  }  printf("\n原始数据：\n");  for (int i = 0; i < 10; i++) {  printf("%c,%d\t", stu[i].name, stu[i].score);  }  printf("\n进行选择排序，结果为：\n");  SelectionSort();  putchar('\n');  }  **四、实验实习所得结果及分析**    **五、实验实习结果分析和（或）源程序调试过程**  冒泡排序基本思想：对有n个数据的表，一共最多需要进行n-1趟排序。记比较的趟次为i，则每一趟最多需要比较n-i次。冒泡排序每次比较的都是两个相邻的数，假设要得到从大到小的顺序，如果前面的数小于后面的数，就将这两个数交换位置。  选择排序基本思想：对有n个数据的表，一共最多需要进行n-1趟排序。记比较的趟次为i，则每一趟比较最多需要n-i次。对于选择排序，每一趟排序，假设要得到从大到小的顺序，且从a[i]开始，将a[i]设置为最大值，然后在选出a[i]到表尾中最大的值，如果这个值比a[i]大，就将这两个数据交换位置。  注：在本实验报告中，结构体数组末尾预留了一个空的结构体，是为了便于交换数据。 | | | |