**《数据结构实验》报告**

|  |  |  |
| --- | --- | --- |
| **第 四 次实验** | **日期：2021.11.16** | **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. 理解并学会链表、串的构建和操作

**二、实验内容**

1、假设有序表中数据元素是正整数，请采用顺序表或（带头结点）单链表实现：

（1）OrderInsert(&L, e, int(\*compare)(a,b))

//根据有序判定函数compare，在有序表L的适当位置插入元素e

（2）OrderInput(&L,mt(\*compare)(a,b))

//根据有序判定函数compare，并利用（1）中的有序插入函数OrderInsert，构造有序表L

（3）OrderMerge(&La, &Lb, &Lc, int(\*compare)())

//根据有序判定函数compare，将两个有序表La和Lb归并为一个有序表Lc

2、请实现：

（1）升幂多项式的构造，升幂多项式是指多项式的各项按指数升序有序，约定系数不等于0，指数不小于0，例如

（2）两个升幂多项式的相加。

3、请实现：求一个顺序表序列的最长连续递增子序列

例如，{1, 9, 2, 5, 7, 3, 4, 6, 8, 0}中的最长连续递增子序列为{3, 4, 6, 8}

4、请实现：求两个单链表升序集合的交集

例如，{1, 2, 5}∩{2, 4, 5, 8, 10}={2, 5}

**三、数据结构及算法设计**

1、

相比于对链表排序，在构建链表时就有序创建是更简单、效果更好的一种思路。

本题中先构建node和list两个结构体，然后在结构体中创建了OrderInsert函数，并使其调用函数compare，以确定新的数据插入的位置。

compare函数的内部逻辑是先判断是否比头节点小，是否比尾节点大，如果是的话直接将新的数据当作新的头/尾节点，返回插入位置之前的节点（这也是需要头节点的原因）。否则就遍历整个链表，一定能在链表内部找到合适的插入位置，返回插入位置前的节点。

OrderInsert接收了compare返回的节点，将新的节点插入到返回节点的后面，并将其连接成链，这样就完成了构建。

OrderMerge就是先拷贝第一个链表，然后遍历第二个链表的值，并依次调用OrderInsert即可。

2、

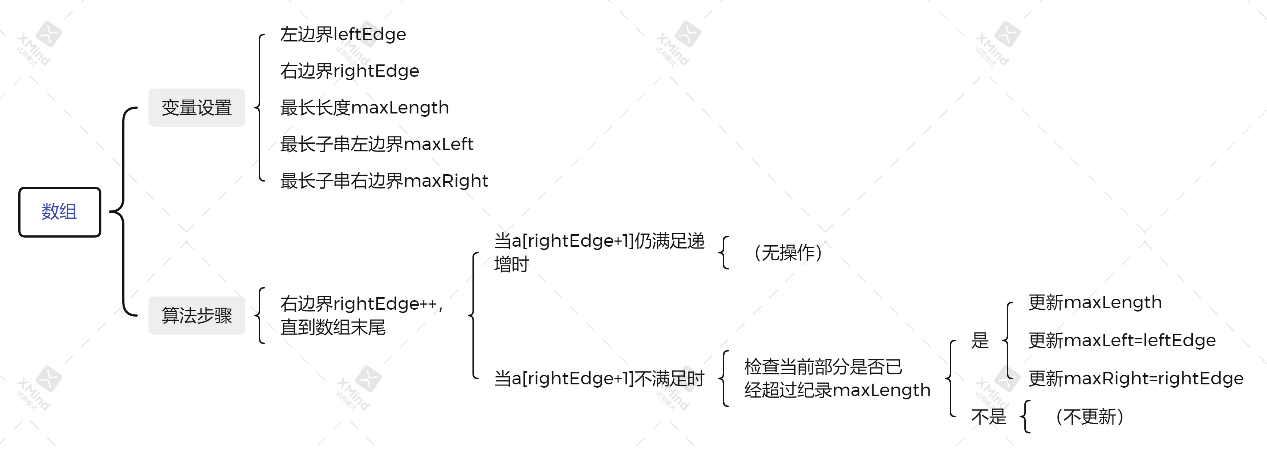
构建链表的大体思路和上题相同，只不过节点中数据变多了——需要记录系数和指数。而且对于同阶节点的重复插入，实质上是将其对应的指数相加，链表长度不变。

Merge的思路也大致相同，不过同样是当出现阶数相同的时候，不新建节点而是系数相加。

3、

首先构建的思路和第一题完全相同，此处便不作解释。

获取最长连续递增字串的思想有些类似于KMP算法和滑动窗口。也就是设置左边界右边界。让右边界遍历整个数组，当右边界的右1元素仍然满足递增时，右边界扩展；当不满足时，目前这部分即为一个递增子串。然后检查这段的长度是不是比之前记录的长，刷新纪录了就更新长度、最长左边界、最长右边界，没有则不变，然后记得要让左边界到问题处重新开始。



4、

本质和第二题相同，只不过这里只有一个值data可供比较。当出现data相同时，将其值取出存起来。

**四、核心程序代码（调用各函数中的demo函数进行展示）**

1.

|  |
| --- |
| #ifndef \_\_QUES1\_H\_\_  #define \_\_QUES1\_H\_\_  #include <stdio.h>  #include <stdlib.h>  #define ElemType int //如要修改data的数据类型，则要把ShowList的printf中类型符也进行对应的替换  typedef struct ListNode {  ElemType val = 0;  struct ListNode\* next = NULL;  }ListNode, \* NodePointer;  typedef struct LinkList {//链表  ListNode headNode;//有头节点的模式  ListNode\* head = &headNode, \* tail = &headNode;//节点指针，指向头和尾  int len = 0;//链表长度  NodePointer compare(ElemType insertVal, NodePointer comparedNode) {//在return的节点后面插入  if (insertVal <= comparedNode->val)//比最小的还小，则直接返回头指针即可。  return head;  if (insertVal >= tail->val) {//比最大的还大，则直接返回尾指针即可。  return tail;  }  //不比第一个小，不比最后一个大，则在链表内部一定能找到合适的位置。  if (insertVal > comparedNode->val && insertVal <= comparedNode->next->val) //当前节点比插入值小，但下一个比插入值大  return comparedNode;  else  return compare(insertVal, comparedNode->next);  }  void OrderInsert(ElemType val) {  ListNode\* newNode = (ListNode\*)malloc(sizeof(ListNode)); newNode->val = val;//为存放新数据而创建一个新node，并初始化  NodePointer beforeNode;//指向插入位置之前的节点的指针  if (len != 0) {//当表不为空时  beforeNode = compare(val, head);//插入位置之前的Node  if (beforeNode == tail)tail = newNode;//如果是在表尾插入，则更新tail指针，否则就不用动  }  else {  beforeNode = &headNode;  tail = newNode;  }  newNode->next = beforeNode->next;  beforeNode->next = newNode;  len++;  }  void OrderInput(ElemType data) {  OrderInsert(data);  }  void OrderMerge(LinkList& La, LinkList& Lb, LinkList& Lc) {  NodePointer pointer = La.head;  for (int i = 0; i < La.len; i++) {  pointer = pointer->next;  OrderInsert(pointer->val);  }  pointer = Lb.head;  for (int i = 0; i < La.len; i++) {  pointer = pointer->next;  OrderInsert(pointer->val);  }  }  void ShowList() {  NodePointer p = head;  printf("当前链表为：");  for (int i = 0; i < len; i++) {  p = p->next;  printf("%d", p->val);  if (i != len - 1) printf(" -> ");  }  printf("\n");  }  }  LinkList;  void ques1\_demo()  {  LinkList la,lb,lc;  la.OrderInput(4);  la.OrderInput(21);  la.OrderInput(13);  la.ShowList();  lb.OrderInput(13);  lb.OrderInput(44);  lb.OrderInput(3);  lb.ShowList();  lc.OrderMerge(la, lb, lc);  printf("合并后的链表为：\n");  lc.ShowList();  }  #endif |

2.

|  |
| --- |
| #ifndef \_\_QUES2\_H\_\_  #define \_\_QUES2\_H\_\_  #include <stdio.h>  #include <stdlib.h>  #define ParamType double //系数的数据类型  struct equationNode {//多项式每一项  int power = 0;//阶数  ParamType param = 0;//系数  equationNode\* next = NULL;  };  struct equation {//多项式  int nodeNum = 0;//项数  equationNode\* minPow = NULL, \* maxPow = NULL;//指向最低阶的节点和最高阶的节点  void InsertNode(ParamType param, int power) {  equationNode\* newNode = (equationNode\*)malloc(sizeof(equationNode)); newNode->param = param; newNode->power = power; newNode->next = NULL;  if (nodeNum == 0) {//长度为0  maxPow = newNode;  minPow = newNode;  }  else {//长度不为0  if (power < minPow->power) {//比较是否比最小值小  newNode->next = minPow;  minPow = newNode;  }  else if (power > maxPow->power) {//比较是否比最大值大  maxPow->next = newNode;  maxPow = newNode;  }  else {  for (equationNode\* pointer = minPow; pointer != NULL; pointer = pointer->next) {  if (pointer->power == power) {  pointer->param += param;  nodeNum--;//抵消后面nodeNum++的影响（因为这里实际上是同幂节点的合并）  break;  }  else if (power > pointer->power && power < pointer->next->power) {//遍历寻找合适的位置  newNode->next = pointer->next;  pointer->next = newNode;  break;  }  }  }  }  nodeNum++;  }  void ShowEquation() {  equationNode\* nodePointer = minPow;  printf("当前多项式为:\n");  for (int i = 0; i < nodeNum; i++) {  if (i != 0) printf(nodePointer->param > 0 ? " + " : " - ");  printf("%5.2lf", nodePointer->param > 0 ? nodePointer->param : -(nodePointer->param));  printf(nodePointer->power == 0 ? "" : "x^%d", nodePointer->power);  nodePointer = nodePointer->next;  }  printf("\n");  }  void MergeEquation(equation& Ea, equation& Eb, equation& Ec) {  equationNode\* pointer = Ea.minPow;  while (pointer != NULL) {//遍历多项式a，向其中添加  Ec.InsertNode(pointer->param, pointer->power);  pointer = pointer->next;  }  pointer = Eb.minPow;  while (pointer != NULL) {//遍历多项式b，向其中添加  Ec.InsertNode(pointer->param, pointer->power);  pointer = pointer->next;  }  }  };  int ques2\_demo() {  equation ea, eb, ec;  ea.InsertNode(2.13, 4);  ea.InsertNode(-14.3, 3);  ea.InsertNode(6.28, 2);  ea.InsertNode(3.14, 0);  ea.ShowEquation();  eb.InsertNode(3.14, 4);  eb.InsertNode(6.28, 3);  eb.InsertNode(-14.3, 2);  eb.InsertNode(2.13, 0);  eb.ShowEquation();  ec.MergeEquation(ea, eb, ec);  ec.ShowEquation();  return 0;  }  #endif |

3.

|  |
| --- |
| #ifndef \_\_QUES3\_H\_\_  #define \_\_QUES3\_H\_\_  #include <stdio.h>  #include <stdlib.h>  #define ElemType int  ElemType array[8] = { 2,5,6,7,8,9,3,12 };//为了便于测试，直接定义了个静态数组  void ques3\_demo() {//实现查找最长递增子序列和输出  int leftEdge = 0, rightEdge = 0;  int maxLength = 0, maxLeft = 0, maxRight;  printf("原数组为：");  for (int i = 0; i < 8; i++) {  printf("%d ", array[i]);  }  printf("\n");  for (rightEdge = 0; rightEdge < sizeof(array) / sizeof(ElemType); rightEdge++) {  if (array[rightEdge] >= array[rightEdge + 1]) {  if (rightEdge - leftEdge + 1 > maxLength) {  maxLength = rightEdge - leftEdge + 1;  maxLeft = leftEdge;  maxRight = rightEdge;  }  leftEdge = rightEdge + 1;  }  }  printf("最长连续递增字串为：\n");  for (int i = maxLeft; i <= maxRight; i++) {  printf("%d ", array[i]);  }  }  #endif |

4.

|  |
| --- |
| #ifndef \_\_QUES4\_H\_\_  #define \_\_QUES4\_H\_\_  #include <stdio.h>  #include <stdlib.h>  #define ElemType int  struct node {  ElemType val = 0;  node\* next = NULL;  };  struct list {  int len = 0;  node\* head = NULL, \* tail = NULL;  void InsertNode(ElemType val) {  node\* newNode = (node\*)malloc(sizeof(node)); newNode->val = val; newNode->next = NULL;  if (len == 0) {//长度为0  head = newNode;  tail = newNode;  }  else {//长度不为0  if (val < head->val) {  newNode->next = head;  head = newNode;  }  else if (val > tail->val) {  tail->next = newNode;  tail = newNode;  }  else {  for (node\* p = head; p != NULL; p = p->next) {  if ((val > p->val && val < p->next->val) || val == p->val) {  newNode->next = p->next;  p->next = newNode;  break;  }  }  }  }  len++;  }  void ShowList() {  node\* p = head;  printf("当前链表为：");  while (p != NULL) {  printf("%d", p->val);  if (p->next != NULL) printf(" -> ");  p = p->next;  }  printf("\n");  }  };  void ques4\_demo() {//定义链表；查找、输出公共元素  list l1, l2;  l1.InsertNode(4);  l1.InsertNode(7);  l1.InsertNode(1);  l1.InsertNode(5);  l1.ShowList();  l2.InsertNode(5);  l2.InsertNode(7);  l2.InsertNode(2);  l2.ShowList();  int len = 0;  ElemType\* res = (ElemType\*)malloc(len \* sizeof(ElemType));  for (node\* i = l1.head; i != NULL; i = i->next) {  for (node\* j = l2.head; j != NULL; j = j->next) {  if (i->val == j->val) {//出现重复的  len++;  res = (ElemType\*)realloc(res, len \* sizeof(ElemType));  res[len - 1] = i->val;  }  }  }  printf("两链表中均存在的元素：");  for (int i = 0; i < len; i++) {  printf("%d ", res[i]);  }  }  #endif |

**五、测试及结果**

1. 测试用例和结果如下：

文本

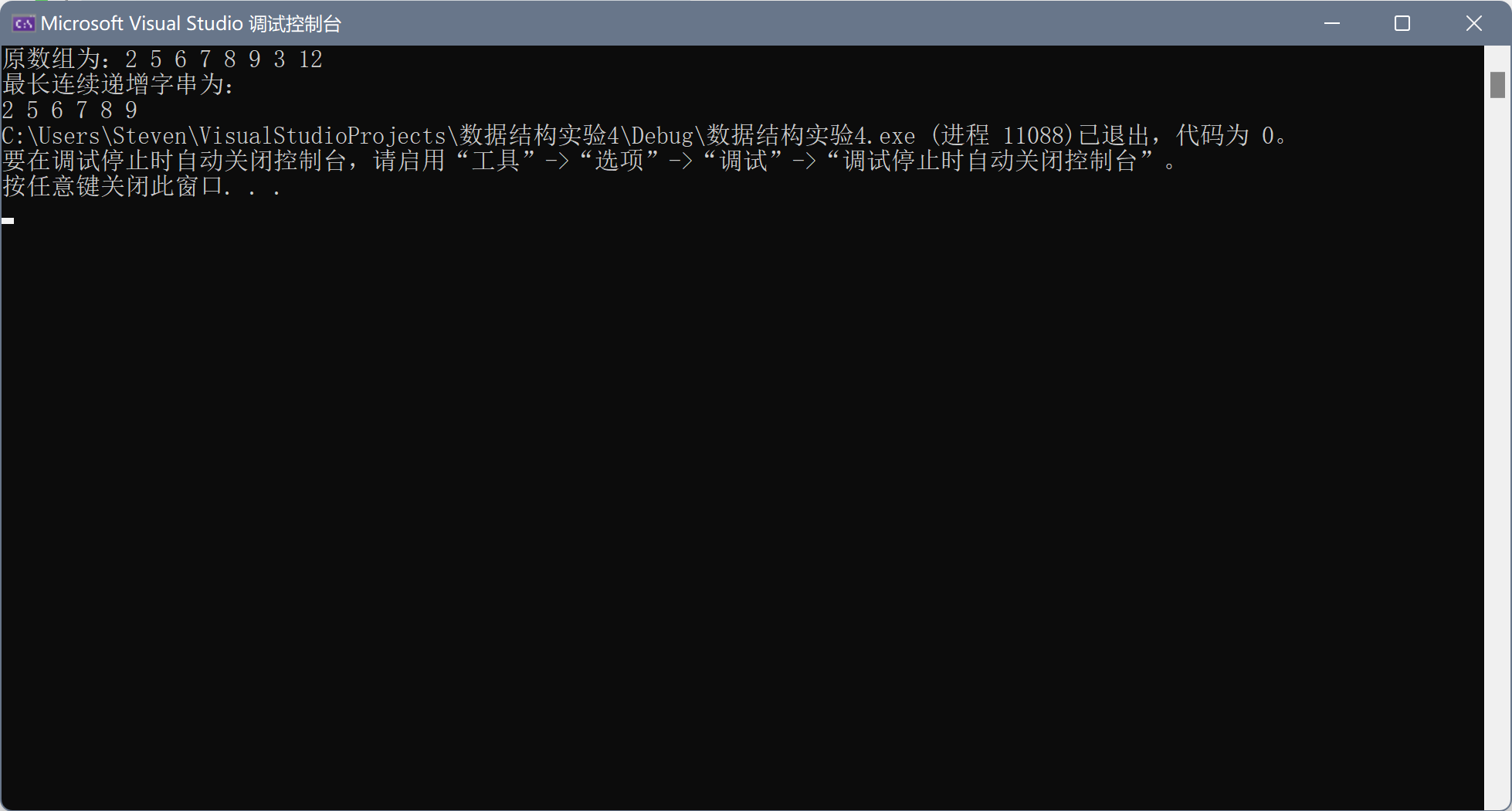
描述已自动生成

2. 测试用例和结果如下：

文本

描述已自动生成

3. 测试用例和结果如下：



4. 测试用例和结果如下：

文本

描述已自动生成