** **

**课程实验报告**

课程名： 数据结构实验

学 院： 数学与计算机学院 系 计算机科学与技术系

专 业： 计算机科学与技术

班 级： 计算机222班

学 号： 5418122020

姓 名： 马星

任课教师： 任 燕

授课学期： 2022 年~~~~ 2023 年 1 学期

**目 录**

[**实验一 线性表实验 (顺序表的实现)** 3](#_Toc155012170)

[**实验二 线性表实验(非循环单链表的实现)** 11](#_Toc155012171)

[**实验三 栈实验(链栈的实现)** 23](#_Toc155012172)

[**实验四 多维数组实验(三元组表的实现)** 29](#_Toc155012173)

[**实验五 树(二叉树)实验(采用二叉树链表存储)** 38](#_Toc155012174)

[**实验六 排序实验(主要排序算法的实现)** 47](#_Toc155012175)

**实验一 线性表实验 (顺序表的实现)**

学生姓名： 马星 学 号： 5418122020 专业班级： 计算机222班

实验类型：□ 验证 □ 综合 □ 设计 □ 创新 实验日期： 2023-10-13 实验成绩：

1. 实验项目名称

线性表实验 (顺序表的实现)

1. 实验目的
2. 采用顺序表实现线性表ADT的操作。
3. 现有两个顺序表 la和lь，此两个表本身无相同元素存在，la表递增有序，lb表递减有序，且lb表中的元素包含在la表连结续的某一部分中，算法要找到这部份元素在la表中的位置，并将它们逆置，使之与lb表中白的序相同。
4. 已知3个顺序表A、B和C中的结点均依元素值自小至大非递减排列(可能存在两个以上值相同的结点)，编写算法对A表进行如下操作:使操作后的表A中仅留下3个表中均包含的数据元素的结点，且没有值相同的结点，并释放所有无用结点。限定算法的时间复杂度为0(m+n+p)，其中m、n和p分别为3个表的长度。
5. 实验基本原理

基于数组的顺序表操作

1. 主要仪器设备及耗材

Windows电脑,编程软件Clion

1. 实验步骤
2. 采用顺序表实现线性表ADT的操作。

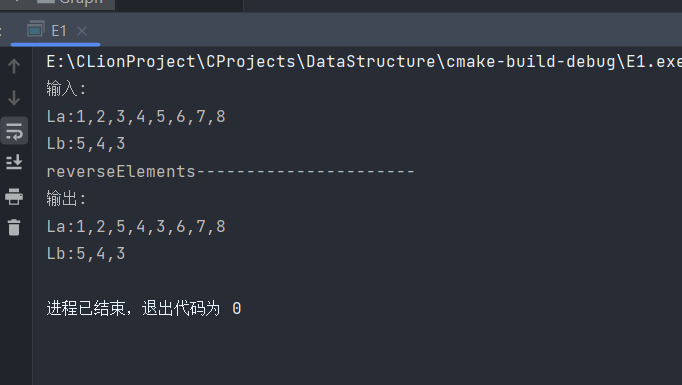
*#include* "../Status.h"  
*#include* "stdlib.h"  
*#include* "stdbool.h"  
  
  
*typedef struct* SqList {  
 *int* \*elem;  
 *int* size;  
} SqList;*//顺序表  
  
/\*\*  
 \* 初始化顺序表  
 \*/*Status InitSqList(SqList \*L) {  
 L->elem = malloc(*sizeof*(*int*) \* DEFAULT\_SIZE);  
 *if* (!L->elem)exit(ERROR);  
 L->size = 0;  
 *return* OK;  
}  
*/\*\*  
 \* 判断顺序表是否已满  
 \*/*bool IsFull(SqList L) {  
 *return* L.size == DEFAULT\_SIZE;  
}  
*/\*\*  
 \* 判断顺序表是否为空  
 \*/*bool IsEmpty(SqList L) {  
 *return* L.size == 0;  
}  
*/\*\*  
 \* 添加元素到顺序表尾  
 \*/*Status Add(SqList \*L, *int* e) {  
 *if* (IsFull(\*L)) {  
 exit(ERROR);  
 }  
 L->elem[L->size++] = e;  
 *return* OK;  
}  
*/\*\*  
 \* 删除顺序表指定索引的元素并返回  
 \*/*Status Delete(SqList \*L, *int* index, *int* \*e) {  
 *if* (IsEmpty(\*L)) {  
 exit(ERROR);  
 }  
 *if* (index < 0 || index > L->size) {  
 exit(ERROR);  
 }  
 \*e = L->elem[index];  
  
 *for* (*int* i = index; i < L->size - 1; i++) {  
 L->elem[i] = L->elem[i + 1];  
 }  
 L->elem[L->size - 1] = 0;  
 L->size--;  
 *return* OK;  
}  
*/\*\*  
 \* 插入元素到顺序表指定索引  
 \*/*Status Insert(SqList \*L, *int* index, *int* e) {  
 *if* (IsFull(\*L)) {  
 exit(ERROR);  
 }  
 *if* (index < 0 || index > L->size) {  
 exit(ERROR);  
 }  
 *for* (*int* i = L->size; i > index; i--) {  
 L->elem[i] = L->elem[i - 1];  
 }  
 L->elem[index] = e;  
 L->size++;  
 *return* OK;  
}  
*/\*\*  
 \* 获取顺序表当前元素个数  
 \*/  
int* GetLength(SqList L) {  
 *return* L.size;  
}  
*/\*\*  
 \* 清空顺序表  
 \*/*Status Clear(SqList \*L) {  
 L->size = 0;  
 *return* OK;  
}  
*/\*\*  
 \* 销毁顺序表  
 \*/*Status Destroy(SqList \*L) {  
 free(L->elem);  
 *return* OK;  
}  
*/\*\*  
 \* 获取顺序表指定索引元素  
 \*/  
int* GetElem(SqList L, *int* index) {  
 *if* (index < 0 || index >= L.size) {  
 exit(ERROR);  
 }  
 *return* L.elem[index];  
}  
*/\*\*  
 \* 查找元素在顺序表中的索引,找不到返回-1  
 \*/  
int* FindElem(SqList L, *int* e) {  
 *for* (*int* i = 0; i < L.size; i++) {  
 *if* (L.elem[i] == e) {  
 *return* i;  
 }  
 }  
 *return* -1;  
}  
*/\*\*  
 \* 遍历顺序表  
 \*/*Status Travel(SqList L, *int* \*visit(*int*)) {  
 *for* (*int* i = 0; i < L.size; i++) {  
 visit(L.elem[i]);  
 }  
 *return* OK;  
}

1. 现有两个顺序表 la和lь，此两个表本身无相同元素存在，la表递增有序，lb表递减有序，且lb表中的元素包含在la表连结续的某一部分中，算法要找到这部份元素在la表中的位置，并将它们逆置，使之与lb表中白的序相同。

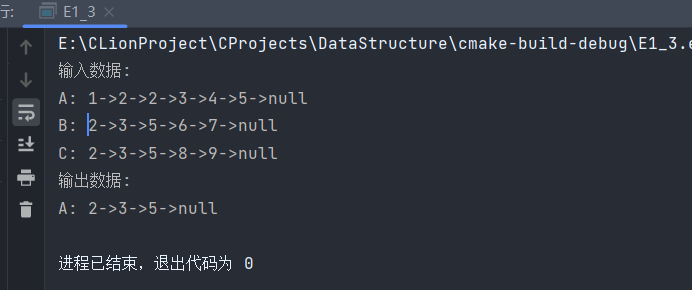
*#include* <stdio.h>  
*#include* "SqList.c"  
  
*/\*\*  
 \* 将lb在la中的部分倒置  
 \* @param la 递增  
 \* @param lb 递减  
 \* @param laSize  
 \* @param lbSize  
 \*/  
void* reverseElements(SqList \*la, *const* SqList lb) {  
 *int* p1 = 0, p2 = 0;  
 *int* index = -1;  
  
 *while* (p1 < la->size && p2 < lb.size) {*//查找a中第一个在b中的元素  
 if* (la->elem[p1] < lb.elem[p2]) {  
 p1++;  
 } *else if* (la->elem[p1] > lb.elem[p2]) {  
 p2++;  
 } *else* {  
 index = p1;  
 *break*;  
 }  
 }  
 *if* (index == -1) {*//没有找到相同元素  
 return*;  
 }  
 *int* i, j;  
 *for* (i = index - lb.size + 1, j = index; i < j; i++, j--) {*//将表a对应区域逆置  
 int* temp = la->elem[i];  
 la->elem[i] = la->elem[j];  
 la->elem[j] = temp;  
 }  
}  
  
*int* main() {  
 *//数据输入* SqList la, lb;  
 InitSqList(&la);  
 InitSqList(&lb);  
 *int* a[] = {1, 2, 3, 4, 5, 6, 7, 8};  
 AddAll(&la, a, *sizeof*(a) / *sizeof*(*int*));  
 *int* b[] = {5, 4, 3};  
 AddAll(&lb, b, *sizeof*(b) / *sizeof*(*int*));  
 printf("输入:\n");  
 printf("La:");  
 printSqList(la);  
 printf("Lb:");  
 printSqList(lb);  
 *//运行测试* printf("reverseElements----------------------\n");  
 reverseElements(&la, lb);  
 *//数据输出* printf("输出:\n");  
 printf("La:");  
 printSqList(la);  
 printf("Lb:");  
 printSqList(lb);  
 *return* 0;  
}

1. 已知3个顺序表A、B和C中的结点均依元素值自小至大非递减排列(可能存在两个以上值相同的结点)，编写算法对A表进行如下操作:使操作后的表A中仅留下3个表中均包含的数据元素的结点，且没有值相同的结点，并释放所有无用结点。限定算法的时间复杂度为0(m+n+p)，其中m、n和p分别为3个表的长度。
2. *#include* "../../Structures/LinkedList/LinkedList.c"  
   *#include* <stdio.h>  
     
   */\*\*将三个递增链表保留3个相同的元素到表A\*/  
   void* retainCommonNodes(LinkedList \*A, LinkedList B, LinkedList C) {  
    LNode \*prev = \*A;  
    LNode \*currA = (\*A)->next, \*currB = B->next, \*currC = C->next;  
    *int* count = 0;  
    *while* (currA && currB && currC) {  
    *if* (currA->data == currB->data && currA->data == currC->data) {*//三者相等,满足添加*count++;  
    *if* (count == 3) {*//已经有三个,可以退出* FreeList(&currA->next);*//将后面的无用节点释放* }  
    prev = currA;  
    currA = currA->next;  
    currB = currB->next;  
    currC = currC->next;  
    } *else* {*//三者不相等  
    if* (currA->data < currB->data || currA->data < currC->data) {*//a的当前节点不满足要求,删除节点* prev->next = currA->next;  
    currA = prev->next;  
    } *else* {*//bc表的节点值至少有一个小于a表当前节点,移动b/c表的指针  
    if* (currB->data < currC->data) {  
    currB = currB->next;  
    } *else if* (currB->data > currC->data) {  
    currC = currC->next;  
    } *else* {  
    currB = currB->next;  
    currC = currC->next;  
    }  
    }  
    }  
    }  
   }  
     
     
   *int* main() {  
    *//测试* LinkedList A = NULL, B = NULL, C = NULL;  
    InitLinkedList(&A);  
    InitLinkedList(&B);  
    InitLinkedList(&C);  
    *int* a[] = {1, 2, 2, 3, 4, 5};  
    AddAll(&A, a, *sizeof*(a) / *sizeof*(*int*));  
    *int* b[] = {2, 3, 5, 6, 7};  
    AddAll(&B, b, *sizeof*(b) / *sizeof*(*int*));  
    *int* c[] = {2, 3, 5, 8, 9};  
    AddAll(&C, c, *sizeof*(c) / *sizeof*(*int*));  
    printf("输入数据:\n");  
    printf("A: ");  
    PrintLinkedList(A);  
    printf("B: ");  
    PrintLinkedList(B);  
    printf("C: ");  
    PrintLinkedList(C);  
     
    retainCommonNodes(&A, B, C);  
      
    printf("输出数据:\n");  
    printf("A: ");  
    PrintLinkedList(A);  
     
    FreeList(&A);  
    FreeList(&B);  
    FreeList(&C);  
     
    *return* 0;  
   }
3. 实验数据及处理结果

2运行结果



3运行结果:



1. 思考讨论题或体会或对改进实验的建议

可以加入一些异常处理,比如链表未初始化、不足3个等

1. 参考资料

数据结构

**实验二** **线性表实验(非循环单链表的实现)**

学生姓名： 马星 学 号： 5418122020 专业班级： 计算机222班

实验类型：□ 验证 □ 综合 □ 设计 □ 创新 实验日期： 2023-10-20 实验成绩：

1. 实验项目名称

线性表实验(非循环单链表的实现)

1. 实验目的

1、 非循环单链表实现

2、 判断回文链表

3、 分隔链表

三、实验基本原理

非循环单链表

1. 主要仪器设备及耗材

Windows电脑,编程软件Clion

五、实验步骤

1.非循环单链表实现

*#include*<stdio.h>  
*#include*<stdlib.h>  
*#include* <stdbool.h>  
*#include* "../../Structures/Status.h"  
  
*typedef struct* LNode { *//结点类  
 int* data;  
 *struct* LNode \*next;  
} LNode, \*LinkedList;  
  
*/\*\*初始化链表\*/*Status InitLinkedList(LinkedList \*L) {  
 \*L = malloc(*sizeof*(LNode));  
 *if* (!\*L) exit(OVERFLOW);  
 (\*L)->next = NULL;  
 *return* OK;  
}  
  
*/\*\*获取链表长度\*/  
int* GetLength(LinkedList L) {  
 LinkedList p = L->next;  
 *int* len = 0;  
 *while* (p) {  
 len++;  
 p = p->next;  
 }  
 *return* len;  
}  
  
*/\*\*添加一个元素到链表尾\*/*Status Append(LinkedList \*L, *int* e) {  
 LNode \*p = (\*L);  
 *while* (p->next) {  
 p = p->next;  
 }  
 LNode \*added = malloc(*sizeof*(LNode));  
 added->data = e;  
 added->next = NULL;  
 p->next = added;  
 *return* OK;  
}  
  
*/\*\*添加一个数组的所有元素到链表尾\*/*Status AddAll(LinkedList \*L, *int* \*arr, *int* len) {  
 LNode \*p = (\*L);  
 *while* (p->next) {  
 p = p->next;  
 }  
 *for* (*int* i = 0; i < len; i++) {  
 LNode \*added = malloc(*sizeof*(LNode));  
 added->data = arr[i];  
 added->next = NULL;  
 p->next = added;  
 p = p->next;  
 }  
 *return* OK;  
  
}  
  
*/\*\*  
 \* 按索引插入元素到链表  
 \*/*Status Insert(LinkedList \*L, *int* n, *int* e) {  
 LinkedList p = (\*L);  
 *for* (*int* i = 0; i < n; i++) {  
 p = p->next;  
 *if* (!p) *return* ERROR;  
 }  
 LinkedList s = malloc(*sizeof*(LNode));  
 s->data = e;  
 s->next = p->next;  
 p->next = s;  
 *return* OK;  
}  
  
*/\*\*获取指定索引的元素\*/*Status GetElem(LinkedList L, *int* n, *int* \*e) {  
 LinkedList p = L->next;  
 *int* i;  
 *for* (i = 1; i < n; i++)  
 p = p->next;  
 *if* (!p || i > n) *return* ERROR;  
  
 \*e = p->data;  
 *return* OK;  
}  
  
*/\*\*  
 \* 初始化一个含n个节点的链表,节点值在控制台输入,节点顺序与输入顺序相反  
 \*/  
void* CreateLinkedList(LinkedList \*L, *int* n) {  
 (\*L) = malloc(*sizeof*(LNode));  
 *if* (!\*L) exit(OVERFLOW);  
 (\*L)->next = NULL;  
  
 *for* (*int* i = n; i > 0; i--) {  
 LinkedList p = malloc(*sizeof*(LNode));  
 scanf\_s("%d", &p->data);  
 p->next = (\*L)->next;  
 (\*L)->next = p;  
 }  
}  
  
*/\*\*清空链表\*/  
void* Clear(LinkedList \*L) {  
 LNode \*p = (\*L)->next;  
 *while* (p != NULL) {  
 LinkedList q = p;  
 p = p->next;  
 free(q);  
 }  
 (\*L)->next = NULL;  
}  
*/\*\*判断链表是否为空\*/*bool IsEmpty(LinkedList L) {  
 *return* !L->next;  
}  
  
*int* LocateElem(LinkedList L, *int* e, bool(\*compare)(*int*, *int*)) {  
 LinkedList p = L->next;  
 *int* i = 1;  
 *while* (p) {  
 *if* (compare(e, p->data))  
 *return* i;  
 i++;  
 p = p->next;  
 }  
 *return* ERROR;  
}  
  
*/\*\*  
 \* 获取前驱  
 \*/*Status PriorElem(LinkedList L, *int* cur\_e, *int* \*pre\_e) {  
 LinkedList p = L->next;  
 LinkedList q = L;  
 *while* (p && p->data != cur\_e) {  
 q = q->next;  
 p = p->next;  
 }  
 *if* (!p) exit(ERROR);  
 \*pre\_e = q->data;  
 *return* OK;  
}  
  
*/\*\*  
 \* 获取后继  
 \*/*Status NextElem(LinkedList L, *int* cur\_e, *int* \*nex\_e) {  
 LinkedList p = L;  
 *while* (p->next && p->data != cur\_e)  
 p = p->next;  
 *if* (!p->next) exit(ERROR);  
 \*nex\_e = p->next->data;  
 *return* OK;  
}  
  
*/\*\*  
 \* 插入节点到链表元素值为e的节点前  
 \*/*Status InsertByVal(LinkedList \*L, *int* e, *int* insertVal) {  
 LinkedList p = (\*L)->next;  
 LinkedList prev = \*L;  
 *while* (p && p->data != e) {  
 p = p->next;  
 prev = prev->next;  
 }  
 *if* (!p) exit(ERROR);  
 LinkedList s = malloc(*sizeof*(LNode));  
 s->data = insertVal;  
 s->next = p;  
 prev->next = s;  
 *return* OK;  
}  
  
*/\*\*  
 \* 按值删除元素  
 \*/*Status Delete(LinkedList \*L, *int* e) {  
 LinkedList p = (\*L)->next;  
 LinkedList prev = \*L;  
  
 *while* (p && p->data != e) {*//找到删除节点以及它的前驱* p = p->next;  
 prev = prev->next;  
 }  
 *if* (!p) exit(ERROR);  
 prev->next = p->next;  
 free(p);  
 *return* OK;  
}  
  
*/\*\*  
 \* 合并升序链表Lb到升序链表La  
 \*/*Status Merge(LinkedList \*La, LinkedList \*Lb) { *//用于非递减链表* LinkedList p = \*La;  
 LinkedList Pa = (\*La)->next;  
 LinkedList Pb = (\*Lb)->next;  
 *while* (Pa && Pb) {  
 *if* (Pa->data > Pb->data) {  
 p->next = Pb;  
 p = Pb;  
 Pb = Pb->next;  
 } *else* {  
 p->next = Pa;  
 p = Pa;  
 Pa = Pa->next;  
 }  
 }  
 p->next = Pa ? Pa : Pb; *// 插入剩余段* free(Lb); *//释放Lb的头节点  
 return* OK;  
}  
  
*/\*\*  
 \* 链表反转  
 \*/*Status Reverse(LinkedList \*L) {  
 LinkedList curr = (\*L)->next;  
 LinkedList prev = NULL, next = NULL;  
 *while* (curr) {  
 next = curr->next;  
 curr->next = prev;  
 prev = curr;  
 curr = next;  
 }  
 (\*L)->next = prev;  
 *return* OK;  
}  
  
*/\*\*打印链表\*/  
void* PrintLinkedList(LinkedList L) {  
 LinkedList p = L->next;  
 *while* (p != NULL) {  
 printf("%d->", p->data);  
 p = p->next;  
 }  
 printf("null\n");  
}  
  
*/\*销毁链表链表\*/  
void* FreeList(LinkedList \*L) {  
 LNode \*p = \*L;  
 *while* (p != NULL) {  
 LNode \*next = p->next;  
 free(p);  
 p = next;;  
 }  
}

1. 判断回文链表

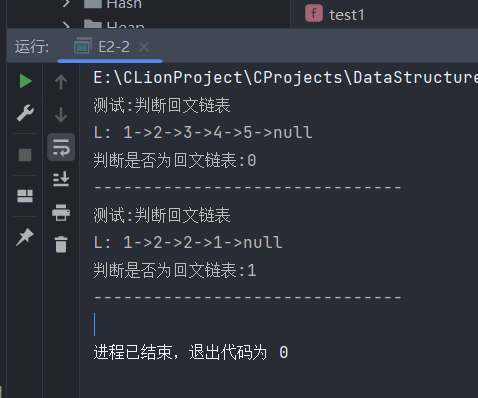
*#include* <stdio.h>  
*#include* <stdbool.h>  
*#include* "../../Structures/Status.h"  
*#include* "../../Structures/LinkedList/LinkedList.c"  
  
  
*/\*\*  
 \* <h1>判断回文链表</h1>  
 \* 快慢指针法反转前半部分链表  
 \* 然后从前半部分链表的新头节点与后半部分链表头开始依次比较  
 \* @return 是回文链表返回true(1),不是则返回false(0)  
 \*/*bool isPlalindrome(LinkedList L) {  
 LNode \*p = NULL;  
 LNode \*slow = L->next;*//快慢指针* LNode \*fast = L->next;  
 *while* (fast) {  
 fast = fast->next;  
 *if* (!fast) {  
 slow = slow->next;  
 *break*;  
 }  
 fast = fast->next;  
 *//链表反转* LNode \*temp = slow->next;  
 slow->next = p;  
 p = slow;  
  
 slow = temp;  
 }  
 *while* (slow && p) {  
 *if* (slow->data != p->data) {  
 *return* false;  
 }  
 slow = slow->next;  
 p = p->next;  
 }  
 *return* true;  
}  
  
  
*void* test1() {  
 printf("测试:判断回文链表\n");  
 LinkedList L;  
 InitLinkedList(&L);  
 *int* data[] = {1, 2, 3, 4, 5};  
 AddAll(&L, data, *sizeof*(data) / *sizeof*(*int*));  
 printf("L: ");  
 PrintLinkedList(L);  
 printf("判断是否为回文链表:%d\n", isPlalindrome(L));  
 printf("-------------------------------\n");  
}  
  
*void* test2() {  
 printf("测试:判断回文链表\n");  
 LinkedList L;  
 InitLinkedList(&L);  
 *int* data[] = {1, 2, 2, 1};  
 AddAll(&L, data, *sizeof*(data) / *sizeof*(*int*));  
 printf("L: ");  
 PrintLinkedList(L);  
 printf("判断是否为回文链表:%d\n", isPlalindrome(L));  
 printf("-------------------------------\n");  
}  
  
*int* main() {  
 test1();  
 test2();  
}

3.分隔链表

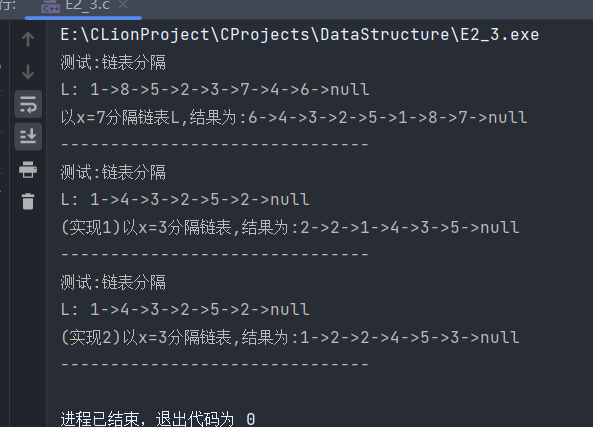
1. *#include* <stdio.h>  
   *#include* "LinkedList.c"  
     
   */\*\*  
    \* 分隔链表:实现1  
    \* 头插法  
    \*/  
   void* split(LinkedList \*L, *int* x) {  
    LNode \*prev = \*L;  
    LNode \*p = (\*L)->next;  
    *while* (p) {  
    *if* (p->data < x) {  
    *//将节点插入到表头* LNode \*next = p->next;  
    prev->next = p->next;  
    p->next = (\*L)->next;  
    (\*L)->next = p;  
    p = next;  
    *if* (prev == \*L) {*//前驱为头节点需要特殊处理* prev = prev->next;  
    }  
    } *else* {  
    prev = p;  
    p = p->next;  
    }  
    }  
   }  
     
   */\*\*  
    \*<h1>分隔链表:实现2</h1>  
    \* 交换值法  
    \*/  
   void* split2(LinkedList \*L, *int* x) {  
    *if* (!(\*L)->next) {  
    *return*;  
    }  
    LNode \*p1 = (\*L)->next;  
    LNode \*p2 = (\*L)->next;  
    *while* (p1 && p2) {  
    *//p2寻找大于等于x的节点  
    while* (p2 && p2->data < x) {  
    p2 = p2->next;  
    }  
    *//p1从第一个大节点开始,寻找小于x的节点* p1 = p2;  
    *while* (p1 && p1->data >= x) {  
    p1 = p1->next;  
    }  
    *if* (p1 && p2) {*//交换节点值  
    int* temp = p1->data;  
    p1->data = p2->data;  
    p2->data = temp;  
    }  
    }  
   }  
     
   *void* test1() {  
    printf("测试:链表分隔\n");  
    LinkedList L;  
    InitLinkedList(&L);  
    *int* data[] = {1, 8, 5, 2, 3, 7, 4, 6};  
    AddAll(&L, data, *sizeof*(data) / *sizeof*(*int*));  
    printf("L: ");  
    PrintLinkedList(L);  
    split(&L, 7);  
    printf("以x=7分隔链表L,结果为:");  
    PrintLinkedList(L);  
    printf("-------------------------------\n");  
   }  
     
   *void* test2() {  
    printf("测试:链表分隔\n");  
    LinkedList L;  
    InitLinkedList(&L);  
    *int* data[] = {1, 4, 3, 2, 5, 2};  
    AddAll(&L, data, *sizeof*(data) / *sizeof*(*int*));  
    printf("L: ");  
    PrintLinkedList(L);  
    split(&L, 3);  
    printf("(实现1)以x=3分隔链表,结果为:");  
    PrintLinkedList(L);  
    printf("-------------------------------\n");  
   }  
     
   *void* test3() {  
    printf("测试:链表分隔\n");  
    LinkedList L;  
    InitLinkedList(&L);  
    *int* data[] = {1, 4, 3, 2, 5, 2};  
    AddAll(&L, data, *sizeof*(data) / *sizeof*(*int*));  
    printf("L: ");  
    PrintLinkedList(L);  
    split2(&L, 3);  
    printf("(实现2)以x=3分隔链表,结果为:");  
    PrintLinkedList(L);  
    printf("-------------------------------\n");  
   }  
     
   *int* main() {  
    test1();  
    test2();  
    test3();  
   }

六、实验数据及处理结果

2运行结果:



3运行结果:



七、思考讨论题或体会或对改进实验的建议

分隔链表题目可以改为大于x的放右边,小于x的放左边

八、参考资料

数据结构

**实验三 栈实验(链栈的实现)**

学生姓名： 马星 学 号： 5418122020 专业班级： 计算机222班

实验类型：□ 验证 □ 综合 □ 设计 □ 创新 实验日期：2023-11-03 实验成绩：

1. 实验项目名称

栈实验(链栈的实现)

1. 实验目的

1.链栈实现

2.检测出栈顺序:输入两个整数序列表示入栈序列和出栈序列

3.删除最外层括号

1. 实验基本原理

单链表与栈

1. 主要仪器设备及耗材

Windows电脑,编程软件Clion

1. 实验步骤

1、链栈实现

*#include* <stdio.h>  
*#include* "../../Structures/Status.h"  
*#include* "stdbool.h"  
*#include* "stdlib.h"  
  
*typedef struct* Node {  
 *int* data;  
 *struct* Node \*next;  
} Node, \*LinkedStack;*//链栈(非循环单链表)  
/\*\*初始化链栈\*/*Status InitLinkedStack(LinkedStack \*L) {  
 \*L = malloc(*sizeof*(Node));  
 *if* (!\*L)exit(ERROR);  
 (\*L)->data = -1;  
 (\*L)->next = NULL;  
 *return* OK;  
}  
*/\*\*判断栈是否为空\*/*bool IsEmpty(LinkedStack L) {  
 *return* L->next == NULL;  
}  
*/\*\*添加元素到栈顶\*/*Status Push(LinkedStack L, *int* e) {  
 Node \*added = malloc(*sizeof*(Node));  
 added->data = e;  
 added->next = L->next;  
 L->next = added;  
 *return* OK;  
}  
  
*/\*\*弹出栈顶元素\*/*Status Pop(LinkedStack \*L, *int* \*e) {  
 *if* (IsEmpty(\*L)) {  
 exit(ERROR);  
 }  
 Node \*p = (\*L)->next;  
 \*e = p->data;  
 (\*L)->next = p->next;  
 free(p);  
 *return* OK;  
}  
*/\*\*获取栈顶元素\*/*Status GetTop(LinkedStack \*L, *int* \*e) {  
 *if* (IsEmpty(\*L)) {  
 exit(ERROR);  
 }  
 Node \*p = (\*L)->next;  
 \*e = p->data;  
 *return* OK;  
}  
*/\*\*获取栈元素个数\*/  
int* GetLength(LinkedStack L) {  
 Node \*p = L->next;  
 *int* len = 0;  
 *while* (p) {  
 len++;  
 p = p->next;  
 }  
 *return* len;  
}  
  
*/\*\*清空栈\*/*Status Clear(LinkedStack \*L) {  
 Node \*p = (\*L)->next;  
 *while* (p) {  
 Node \*n = p->next;  
 free(p);  
 p = n;  
 }  
 (\*L)->next = NULL;  
 *return* OK;  
}  
  
*/\*\*销毁栈\*/*Status Destroy(LinkedStack \*L) {  
 Clear(L);  
 free(L);  
 *return* OK;  
}

1. 检测出栈顺序:输入两个整数序列表示入栈序列和出栈序列

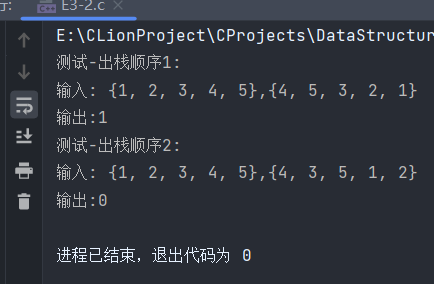
*#include* <stdio.h>  
*#include* <stdbool.h>  
*#include* "LinkedStack.c"  
*/\*\*  
 \* 检测出栈顺序  
 \* @param seq1 入栈顺序  
 \* @param seq2 待检测出栈顺序  
 \* @param len 数据量  
 \* @return seq2是否是可能出栈顺序  
 \*/*bool popSeq(*int* \*seq1, *const int* \*seq2, *int* len) {  
 *int* k = 0;  
 LinkedStack stack;  
 InitLinkedStack(&stack);  
 *for* (*int* i = 0; i < len; i++) {  
 Push(&stack, seq1[i]);*//seq1[i]入栈  
 while* (!IsEmpty(stack)) {*//检测seq2[k]是否需要出栈  
 int* e;  
 GetTop(stack, &e);  
 *if* (e == seq2[k]) {*//需要出栈  
 int* \_;  
 Pop(&stack, &\_);  
 k++;*//检测下一个* } *else* {  
 *break*;  
 }  
 }  
 }  
 *//退出后一定是全部出栈的,否则有元素没有出栈说明不是正确的出栈序列  
 return* IsEmpty(stack);  
}  
  
*int* main() {  
 printf("测试-出栈顺序1:\n");  
 *int* seq[] = {1, 2, 3, 4, 5};  
 *int* seq1[] = {4, 5, 3, 2, 1};  
 bool result = popSeq(seq, seq1, 5);  
 printf("输入: {1, 2, 3, 4, 5},{4, 5, 3, 2, 1}\n");  
 printf("输出:%d\n", result);*//1 -> true* printf("测试-出栈顺序2:\n");  
 *int* seq2[] = {4, 3, 5, 1, 2};  
 bool result2 = popSeq(seq, seq2, 5);  
 printf("输入: {1, 2, 3, 4, 5},{4, 3, 5, 1, 2}\n");  
 printf("输出:%d\n", result2);*//0 -> false*}

3、删除最外层括号

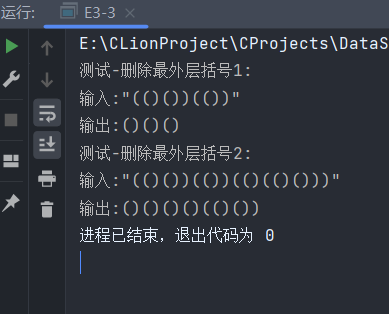
*#include* <stdio.h>  
*#include* <stdlib.h>  
*#include* "LinkedStack.c"  
  
*/\*\*删除最外层括号\*/  
char* \*deleteOuter(*const char* \*s, *int* len, *int* \*resLen) {  
 *char* \*result = malloc(*sizeof*(*char*) \* len);  
 \*resLen = len;*//初始设为len,每次删除最外层括号时-2  
 int* k = 0;*//当前result的填入位置* LinkedStack stack;*//存储括号* InitLinkedStack(&stack);  
 *int* left = 0, right = 0;*//统计当前左右括号数量  
 for* (*int* i = 0; i < len; i++) {  
 *char* ch = s[i];  
 Push(&stack, ch);*//括号入栈  
 if* (ch == '(') left++; *else* right++;*//计数  
 if* (left == right) {*//左右括号数量相等,说明是一个完整的项  
 //取出中间部分  
 int* stackSize = GetLength(stack) - 1;  
 *for* (*int* j = 0; j < stackSize; j++) {  
 *int* r;*//由于存的是整数,需要获取后将r转为字符型* Pop(&stack, &r);  
 result[k + stackSize - j - 1] = (*char*) r;*//出栈顺序是倒着的,需要在k位置倒序填入* }  
 k += stackSize - 1;*//k跳到下一个起点位置  
 int* e;  
 Pop(&stack, &e);*//弹出前面的左括号* \*resLen -= 2;*//去除了一对括号* }  
 }  
 *return* result;  
}  
*int* main() {  
 printf("测试-删除最外层括号1:\n");  
 *char* \*s1 = "(()())(())";  
 *int* resLen1;  
 *char* \*res1 = deleteOuter(s1, 10, &resLen1);  
 printf("输入:\"(()())(())\"\n");  
 printf("输出:");  
 *for* (*int* i = 0; i < resLen1; i++) {  
 printf("%c", res1[i]);*//()()()* }  
 printf("\n");  
  
 printf("测试-删除最外层括号2:\n");  
 *char* \*s2 = "(()())(())(()(()()))";  
 *int* resLen2;  
 *char* \*res2 = deleteOuter(s2, 20, &resLen2);  
 printf("输入:\"(()())(())(()(()()))\"\n");  
 printf("输出:");  
 *for* (*int* i = 0; i < resLen2; i++) {  
 printf("%c", res2[i]);*//()()()()(()())* }  
}

1. 实验数据及处理结果

2运行结果:



3运行结果:



1. 思考讨论题或体会或对改进实验的建议

无

1. 参考资料

数据结构

**实验四 多维数组实验(三元组表的实现)**

学生姓名： 马星 学 号： 5418122020 专业班级： 计算机222班

实验类型：□ 验证 □ 综合 □ 设计 □ 创新 实验日期：2023-11-24 实验成绩：

1. 实验项目名称

多维数组实验(三元组表的实现)

1. 实验目的

1、采用三元组表实现多为数组的压缩存储。

三、实验基本原理

多维数组三元表

1. 主要仪器设备及耗材

Windows电脑,编程软件Clion

五、实验步骤

1. 采用三元组表实现多为数组的压缩存储。

*#include* "../Status.h"  
*#include* <stdio.h>  
*#include* <stdlib.h>  
  
*typedef struct* {  
 *int* i, j; *//该非零元的行下标和列下标  
 int* e;  
} Triple; *//Triple:三倍的,三重的. 在这里指三元组  
  
typedef struct* {  
 *int* mu, nu, tu; *//矩阵的行数, 列数和非零元个数* Triple \*data;*// 书上的写法:Triple data[MAXSIZE + 1];*} SMatrix;  
  
*/\*\*创建多维数组\*/*Status CreateSMatrix(SMatrix \*M) {  
 printf("请输入矩阵的行数,列数,");  
 printf("以及要存入的非零元的个数(以空格分隔):\n");  
 fflush(stdout);  
  
 scanf("%d %d %d", &M->mu, &M->nu, &M->tu);  
 getchar(); *// 读取换行符* M->data = malloc(*sizeof*(Triple) \* M->tu); *// 分配内存  
 if* (!M->data) exit(ERROR); *//内存分配失败  
  
 for* (*int* k = 0; k < M->tu; k++) { *//将值输入到数组中* printf("请依次输入第%d个元素的行下标", k + 1);  
 printf(",列下标和值(以空格分隔):\n");  
 fflush(stdout);  
 scanf("%d %d %d", &M->data[k].i, &M->data[k].j, &M->data[k].e);  
 getchar(); *// 读取换行符* }  
 *return* OK;  
}  
*/\*\*销毁多维数组\*/*Status DestroySMatrix(SMatrix \*M) {  
 free(M->data);  
 *return* OK;  
}  
*/\*\*拷贝多维数组\*/*Status CopySMatrix(SMatrix M, SMatrix \*T) {  
 T->tu = M.tu;  
 T->nu = M.nu;  
 T->mu = M.mu;  
 *for* (*int* k = 0; k < M.tu; k++) {  
 T->data[k].e = M.data[k].e;  
 T->data[k].j = M.data[k].j;  
 T->data[k].i = M.data[k].i;  
 }  
 *return* OK;  
}  
*/\*\*快速转置\*/*Status FastTransPoseSMatrix(SMatrix M, SMatrix \*T) {  
 *//先清空矩阵T,再为矩阵T分配内存空间  
 if* (T->data) free(T->data);  
 T->data = malloc(*sizeof*(Triple) \* M.tu);  
 *if* (!T->data) exit(ERROR);  
 *//对矩阵T的行数,列数,及非零元素个数进行赋值* T->tu = M.tu;  
 T->nu = M.mu;  
 T->mu = M.nu;  
  
 *if* (T->tu) { *// 矩阵T不为空  
 return* OK;  
 }  
 *//求M中每一列含非零元的个数  
 int* num[M.nu + 1];  
 *for* (*int* i = 0; i < M.nu; i++) { *//初始化数组(变长数组不支持声明的时候初始化)* num[i] = 0;  
 }  
 *// 开始记录  
 for* (*int* i = 0; i < M.tu; i++) {  
 num[M.data[i].j]++;  
 }  
 *// 求第col列中第一个非零元在T->base中的序号  
 int* cpot[M.nu + 1];  
 cpot[1] = 1;  
 *for* (*int* i = 2; i <= M.nu; i++) {  
 cpot[i] = cpot[i - 1] + num[i - 1];  
 }  
  
 *for* (*int* p = 0; p < M.tu; p++) {  
 *int* col = M.data[p].j;  
 *int* q = cpot[col] - 1; *// 根据p所指向的列数和cpot数组可以推出对应在矩阵T->data中的序号* T->data[q].j = M.data[p].i;  
 T->data[q].i = M.data[p].j;  
 T->data[q].e = M.data[p].e;  
 cpot[col]++; *//M矩阵中同一列有多个元素时,使之更新到下一个* } *//for  
 return* OK;  
}  
*/\*\*多维数组相乘\*/*Status MultSMatrix(SMatrix A, SMatrix B, SMatrix \*C) {  
 *if* (A.nu != B.mu) *return* ERROR; *//如果A的列数不等于B的行数，不能相乘  
 if* (!C->data) free(C->data);  
 C->data = malloc(*sizeof*(Triple) \* A.mu \* B.nu);  
 *if* (!C->data) exit(ERROR);  
 C->mu = A.mu;  
 C->nu = B.nu;  
 C->tu = 0; *//初始化结果矩阵  
 if* (A.tu \* B.tu) { *//非零元素个数都不为0  
 for* (*int* a = 0; a < A.tu; ++a) { *//遍历A中的非零元  
 for* (*int* b = 0; b < B.tu; ++b) { *//遍历B中的非零元  
 if* (A.data[a].j == B.data[b].i) { *//如果A中元素的列下标等于B中元素的行下标  
 int* c;  
 *for* (c = 0; c < C->tu; ++c) { *//在C中查找是否已有对应位置的元素  
 if* (C->data[c].i == A.data[a].i && C->data[c].j == B.data[b].j) {  
 C->data[c].e += A.data[a].e \* B.data[b].e; *//如果有，累加上乘积  
 break*;  
 }  
 }  
 *if* (c == C->tu) { *//如果没有找到对应位置的元素* C->data[C->tu].i = A.data[a].i;  
 C->data[C->tu].j = B.data[b].j;  
 C->data[C->tu].e = A.data[a].e \* B.data[b].e; *//新建一个元素* C->tu++; *//非零元个数增加* }  
 }  
 }  
 }  
 }  
 *return* OK;  
}  
*/\*\*多维数组相加\*/*Status AddSMatrix(SMatrix M, SMatrix N, SMatrix \*Q) {  
 free(Q->data);  
 Q->data = malloc(*sizeof*(Triple) \* (M.tu + N.tu)); *//分配内存* Q->tu = 0, Q->mu = M.mu, Q->nu = M.nu;  
 *int* p = 0, q = 0;  
  
 *while* (p < M.tu && q < N.tu) {  
 *int* kM = (M.data[p].i - 1) \* M.nu + M.data[p].j; *//计算此时M,N中非零元的位序,方便比较  
 int* kN = (N.data[q].i - 1) \* N.nu + N.data[q].j;  
 *if* (kM == kN) { *//如果相等,就让两个非零元相加,相加值不为零则加入Q  
 int* e = M.data[p].e + N.data[q].e;  
 *if* (e != 0) {  
 Q->data[Q->tu].e = e;  
 Q->data[Q->tu].i = M.data[p].i;  
 Q->data[Q->tu].j = M.data[p].j;  
 Q->tu++;  
 }  
 p++;  
 q++;  
 } *else if* (kM < kN) {  
 Q->data[Q->tu].e = M.data[p].e;  
 Q->data[Q->tu].i = M.data[p].i;  
 Q->data[Q->tu].j = M.data[p].j;  
 Q->tu++;  
 p++;  
 } *else if* (kM > kN) {  
 Q->data[Q->tu].e = N.data[q].e;  
 Q->data[Q->tu].i = N.data[q].i;  
 Q->data[Q->tu].j = N.data[q].j;  
 Q->tu++;  
 q++;  
 }  
 }  
 *while* (p < M.tu) {  
 Q->data[Q->tu].e = M.data[p].e;  
 Q->data[Q->tu].i = M.data[p].i;  
 Q->data[Q->tu].j = M.data[p].j;  
 Q->tu++;  
 p++;  
 }  
 *while* (q < N.tu) {  
 Q->data[Q->tu].e = N.data[q].e;  
 Q->data[Q->tu].i = N.data[q].i;  
 Q->data[Q->tu].j = N.data[q].j;  
 Q->tu++;  
 q++;  
 }  
 *return* OK;  
}  
*/\*\*打印多维数组\*/*Status PrintSMatrix(SMatrix M) {  
 *int* k = 0;  
 *for* (*int* m = 0; m < M.mu; m++) {  
 *for* (*int* n = 0; n < M.nu; n++) {  
 *if* (M.data[k].i == m + 1 && M.data[k].j == n + 1) *//为非零元素* printf("%d\t", M.data[k++].e);  
 *else //为零元素* printf("0\t");  
 }  
 printf("\n");  
 }  
 printf("\n");  
 *return* OK;  
}

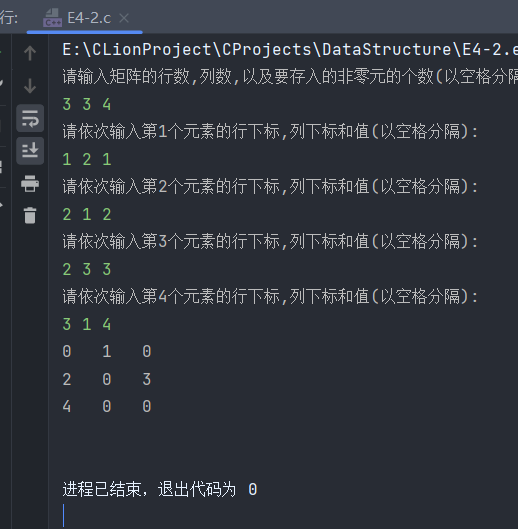
1. 若一个稀疏矩阵A中的非零元素aij(1≤i≤n,1≤j≤m)，以三元组的形式按照行优先的次序存储在三元组顺序表中，则打印输出对应的稀疏矩阵。

*#include* <stdio.h>  
*#include* "SMatrix.c"  
  
Status PrintSMatrix(SMatrix M) {  
 *int* k = 0;  
 *for* (*int* m = 0; m < M.mu; m++) {  
 *for* (*int* n = 0; n < M.nu; n++) {  
 *if* (M.data[k].i == m + 1 && M.data[k].j == n + 1) *//为非零元素* printf("%d\t", M.data[k++].e);  
 *else //为零元素* printf("0\t");  
 }  
 printf("\n");  
 }  
 printf("\n");  
 *return* OK;  
}  
  
*int* main() {  
 SMatrix A;  
 CreateSMatrix(&A);  
 PrintSMatrix(A);  
 */\*  
 \* 0 1 0  
 \* 2 0 3  
 \* 4 0 0  
 \*/*}

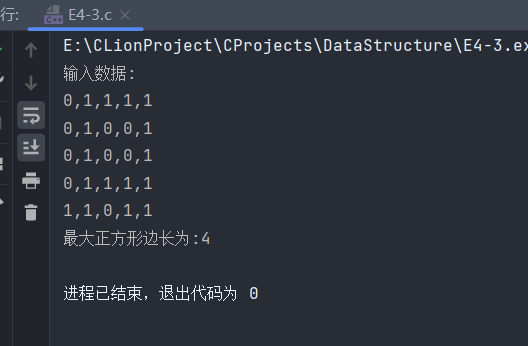
1. 给定一个N\*N的矩阵matrix,在此矩阵中,只有0和1两种值,返回边框全是1的最大正方形的边长长度。如图给定的矩阵中,边框全是1的最大正方形的大小是4\*4，返回4。
2. *#include* <stdio.h>  
   *#include* <stdbool.h>  
   *#include* "../../Algorithm/Sort/ArrayUtil.h"  
     
   *#define* N 5  
   */\*\*以(i,j)为左上角判断边长为len是否为边框全1的正方形\*/*bool isSquare(*int* matrix[N][N], *int* i, *int* j, *int* len) {  
    *for* (*int* k = 0; k < len; k++) {  
    *//边框为1  
    if* (matrix[i][k + j] == 0)*return* false;  
    *if* (matrix[k + i][j] == 0)*return* false;  
    *if* (matrix[i + len - 1][k + j] == 0)*return* false;  
    *if* (matrix[k + i][j + len - 1] == 0)*return* false;  
    }  
    *return* true;  
   }  
     
   */\*\*  
    \* 搜索矩阵中边框为1的最大正方形边长  
    \* @param matrix 二维数组矩阵  
    \* @return 最大正方形边长  
    \*/  
   int* maxLength(*int* matrix[N][N]) {  
    *int* max = 0;  
    *//枚举左上角(i,j)  
    for* (*int* i = 0; i < N; i++) {  
    *for* (*int* j = 0; j < N; j++) {  
    *if* (matrix[i][j] == 0) {  
    *continue*;  
    }  
    *//枚举边长  
    for* (*int* k = i + 1; k <= N; k++) {  
    *int* len = k - i;  
    *if* (len <= max) {  
    *//没有成功当前最大边长,不用进行后续判断  
    continue*;  
    }  
    *if* (isSquare(matrix, i, j, len)) {*//判断是否为正方形* max = len;  
    }  
    }  
    }  
    }  
    *return* max;  
   }  
     
     
   *int* main() {  
    *int* matrix[N][N] = {  
    {0, 1, 1, 1, 1},  
    {0, 1, 0, 0, 1},  
    {0, 1, 0, 0, 1},  
    {0, 1, 1, 1, 1},  
    {1, 1, 0, 1, 1}  
    };  
    printf("输入数据:\n");  
    *for* (*int* i = 0; i < N; i++) {  
    printArr(matrix[i], N);  
    }  
    printf("最大正方形边长为:%d\n", maxLength(matrix));  
   }

六、实验数据及处理结果

2运行结果:



3运行结果:



七、思考讨论题或体会或对改进实验的建议

无

八、参考资料

数据结构

**实验五 树(二叉树)实验(采用二叉树链表存储)**

学生姓名： 马星 学 号： 5418122020 专业班级： 计算机222班

实验类型：□ 验证 □ 综合 □ 设计 □ 创新 实验日期： 2023-12-01 实验成绩：

1. 实验项目名称

树(二叉树)实验(采用二叉树链表存储)

1. 实验目的

1、二叉树(链式)实现

2、一颗二叉树是不是对称的。注意，如果一个二叉树同此二叉树的镜像是同样的，定义其为对称的。

1. 打印出二叉树中结点值的和为输入整数的所有路径。路径定义为从树的根结点开始往下一直到叶结点所经过的结点形成一条路径。
2. 实验基本原理

二叉树链式存储

1. 主要仪器设备及耗材

Windows电脑,编程软件Clion

五、实验步骤

1、二叉树(链式)实现

*#include* <stdio.h>  
*#include* <stdbool.h>  
*#include* <math.h>  
*#include* <malloc.h>  
  
*typedef struct* TreeNode {  
 *int* val;  
 *struct* TreeNode \*left;  
 *struct* TreeNode \*right;  
} TreeNode, \*Tree;  
  
*//前序遍历  
void* PreOrderTraverse(Tree T) {  
 *if* (T == NULL) {  
 *return*;  
 }  
 printf("%c ", T->val);  
 PreOrderTraverse(T->left);  
 PreOrderTraverse(T->right);  
}  
  
*//中序遍历  
void* InOrderTraverse(Tree T) {  
 *if* (T == NULL) {  
 *return*;  
 }  
 PreOrderTraverse(T->left);  
 printf("%c ", T->val);  
 PreOrderTraverse(T->right);  
}  
  
*//后序遍历  
void* PostOrderTraverse(Tree T) {  
 *if* (T == NULL) {  
 *return*;  
 }  
 PreOrderTraverse(T->left);  
 PreOrderTraverse(T->right);  
 printf("%c ", T->val);  
}  
  
  
*void* CreateTreeByScanf(Tree \*T) { *//注意 这里使用二级指针，因为我们要修改指针的值  
 char* ch;  
 scanf("%c", &ch);  
 getchar();  
 *if* (ch == ' ') {  
 \*T = NULL;  
 *return*;  
 }  
 \*T = (TreeNode \*) malloc(*sizeof*(TreeNode));  
 (\*T)->val = ch;  
 CreateTreeByScanf(&(\*T)->left);  
 CreateTreeByScanf(&(\*T)->right);  
}  
  
*/\*\*  
 \* 递归建树  
 \* @param node 当前节点  
 \* @param arr 数据源  
 \* @param curr 当前索引  
 \* @param length 总数据长度  
 \*/  
void* create(Tree \*node, *char* \*arr, *int* \*curr, *int* length) {  
 *char* ch = arr[(\*curr)++];  
 *if* (\*curr == length || !('0' <= ch && ch <= '9')) {  
 node = NULL;  
 *return*;  
 }  
 \*node = (TreeNode \*) malloc(*sizeof*(TreeNode));  
 (\*node)->val = ch;  
 (\*node)->left = NULL;  
 (\*node)->right = NULL;  
 create(&(\*node)->left, arr, curr, length);  
 create(&(\*node)->right, arr, curr, length);  
}  
  
*/\*\*  
 \* 建树  
 \* @param T Tree  
 \* @param arr 数据源  
 \* @param length 总数据长度  
 \*/  
void* CreateTree(Tree \*T, *char* \*arr, *int* length) {  
 *int* curr = 0;  
 create(T, arr, &curr, length);*//递归函数*}

2、一颗二叉树是不是对称的。注意，如果一个二叉树同此二叉树的镜像是同样的，定义其为对称的。

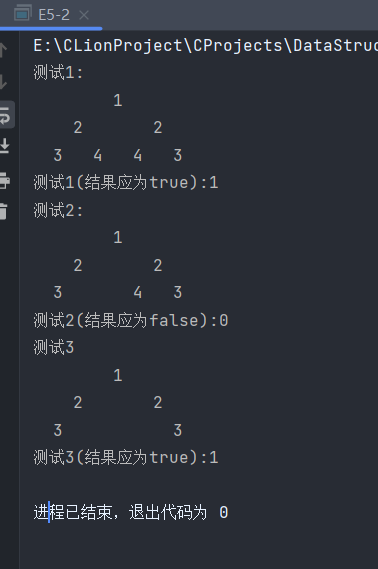
*#include* <stdio.h>  
*#include* <stdbool.h>  
*#include* "Tree.c"  
*#include* <string.h>  
*/\*\*  
 \* 判断两颗子树是否镜像相同  
 \* @param leftNode 左子树根节点  
 \* @param rightNode 右子树根节点  
 \*/*bool isSame(TreeNode \*leftNode, TreeNode \*rightNode) {  
 *if* (leftNode == NULL && rightNode == NULL) {  
 *return* true;  
 }  
 *if* (leftNode == NULL || rightNode == NULL) {  
 *return* false;  
 }  
 *if* (leftNode->val != rightNode->val) {  
 *return* false;  
 }  
 *// 左子树的左子树应该与右子树的右子树镜像相同  
 // 左子树的右子树应该与右子树的左子树镜像相同  
 return* isSame(leftNode->left, rightNode->right) && isSame(leftNode->right, rightNode->left);  
  
}  
*/\*\*  
 \* 判断树是否为对称二叉树  
 \* @param node 根节点  
 \*/*bool isSymmetry(TreeNode node) {  
 *return* isSame(node.left, node.right);  
}  
  
*int* main() {  
 printf("测试1:\n 1\n 2 2\n 3 4 4 3\n");  
 Tree T1;  
 *char* \*arr1 = "123 4 24 3 ";  
 CreateTree(&T1, arr1, strlen(arr1));  
 printf("测试1(结果应为true):%d\n", isSymmetry(\*T1));  
  
 printf("测试2:\n 1\n 2 2\n 3 4 3\n");  
 Tree T2;  
 *char* \*arr2 = "123 24 3 ";  
 CreateTree(&T2, arr2, strlen(arr2));  
 printf("测试2(结果应为false):%d\n", isSymmetry(\*T2));  
  
 printf("测试3\n 1\n 2 2\n 3 3\n");  
 Tree T3;  
 *char* \*arr3 = "123 2 3 ";  
 CreateTree(&T3, arr3, strlen(arr3));  
 printf("测试3(结果应为true):%d\n", isSymmetry(\*T3));  
  
}

1. 打印出二叉树中结点值的和为输入整数的所有路径。路径定义为从树的根结点开始往下一直到叶结点所经过的结点形成一条路径。

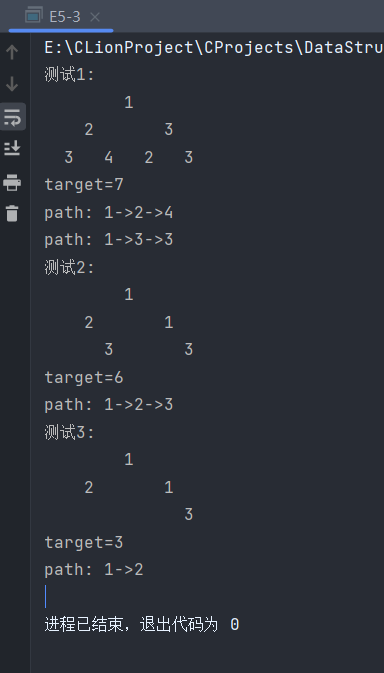
*#include* <stdio.h>  
*#include* <stdbool.h>  
*#include* "Tree.c"  
*#include* <string.h>  
*#include* "../../Structures/Status.h"  
  
*typedef struct* {  
 TreeNode \*base[DEFAULT\_SIZE];  
 *int* size;  
 *int* capacity;  
} Stack;  
  
*int* GetLength(Stack stack) {  
 *return* stack.size;  
}  
  
Status InitStack(Stack \*stack) { *//初始化栈* stack->capacity = DEFAULT\_SIZE;  
 stack->size = 0;  
 *return* OK;  
}  
  
bool IsFull(*const* Stack stack) {  
 *return* stack.size == stack.capacity;  
}  
  
  
TreeNode \*Pop(Stack \*stack) { *//弹栈  
 if* (IsFull(\*stack))exit(OVERFLOW);  
 stack->size--;  
 *return* stack->base[stack->size];  
}  
  
Status Push(Stack \*stack, TreeNode \*e) {  
 *if* (IsFull(\*stack)) {  
 exit(OVERFLOW); *// 栈满, 退出* }  
 stack->base[stack->size] = e;  
 stack->size++;  
 *return* OK;  
}  
  
  
*/\*\*  
 \* 打印栈中存储的路径  
 \* @param stack 栈  
 \*/  
void* printStack(Stack stack) {  
 printf("path: ");  
 *int* len = GetLength(stack);  
 *for* (*int* i = 0; i < len; i++) {  
 printf("%c", stack.base[i]->val);  
 *if* (i != len - 1) {  
 printf("->");  
 }  
 }  
 printf("\n");  
}  
  
*/\*\*  
 \* 向左右搜索  
 \* @param node 当前节点  
 \* @param target 目标值  
 \* @param stack 用栈存储路径  
 \*/  
void* search(TreeNode \*node, *int* target, Stack \*stack) {  
 *//当node为叶子节点,且target为0时说明找到路径  
 if* (node->left == NULL && node->right == NULL && target == 0) {  
 printStack(\*stack);  
 *return*;  
 }  
 *//如果node为叶子节点,而target不为0  
 //或者target小于等于0,而node不是叶子节点  
 //出栈回溯  
 if* (node->left == NULL && node->right == NULL || target <= 0) {  
 *return*;  
 }  
 *//搜索左边  
 if* (node->left != NULL) {  
 Push(stack, node->left);  
 search(node->left, target - node->left->val + '0', stack);  
 Pop(stack);*//回溯* }  
 *//搜索右边  
 if* (node->right != NULL) {  
 Push(stack, node->right);  
 search(node->right, target - node->right->val + '0', stack);  
 Pop(stack);*//回溯* }  
  
}  
  
*/\*\*  
 \* 打印和为target的路径  
 \* @param T 树  
 \* @param target 目标值  
 \*/  
void* printPath(TreeNode T, *int* target) {  
 Stack stack;*//用栈存储路径* InitStack(&stack);  
 Push(&stack, &T);*//根节点必须入栈* search(&T, target - T.val + '0', &stack);*//搜索左右,+'0'是因为存储的数据为字符类型,要转为数字*}  
  
*int* main() {  
 Tree T1;  
 *char* \*data1 = "123 4 32 3 ";  
 CreateTree(&T1, data1, strlen(data1));  
 printf("测试1:\n 1\n 2 3\n 3 4 2 3\n");  
 printf("target=7\n");  
 *// ans:1->2->4 1->3->3* printPath(\*T1, 7);  
  
 Tree T2;  
 *char* \*data2 = "12 3 1 3 ";  
 CreateTree(&T2, data2, strlen(data2));  
 printf("测试2:\n 1\n 2 1\n 3 3\n");  
 printf("target=6\n");  
 *// ans:1->2->3* printPath(\*T2, 6);  
  
 Tree T3;  
 *char* \*data3 = "12 1 3 ";  
 CreateTree(&T3, data3, strlen(data3));  
 printf("测试3:\n 1\n 2 1\n 3\n");  
 printf("target=3\n");  
 *// ans:1->2* printPath(\*T3, 3);  
}

六、实验数据及处理结果

2运行结果:



3运行结果:



七、思考讨论题或体会或对改进实验的建议

无

八、参考资料

数据结构

**实验六 排序实验(主要排序算法的实现)**

学生姓名： 马星 学 号： 5418122020 专业班级： 计算机222班

实验类型：□ 验证 □ 综合 □ 设计 □ 创新 实验日期： 2023-12-15 实验成绩：

1. 实验项目名称

排序实验(主要排序算法的实现)

1. 实验目的

采用顺序表，实现以下排序算法

（1）直接插入排序

（2）希尔排序；

（3）冒泡排序

（4）快速排序；

（5）直接选择排序

（6）堆排序

（7）归并排序

三、实验基本原理

排序算法

四、主要仪器设备及耗材

Windows电脑,编程软件Clion

五、实验步骤

1. 插入排序

*#include* "ArrayUtil.h"  
  
*/\*\*  
 \* 插入排序  
 \* 将数组分为两部分,已排序[0,low-1]和未排序[low,length-1]  
 \* 每次从未排区取出low位置的元素,寻找插入位置  
 \* @param a 待排序数组  
 \* @param len 数组长度  
 \*/  
void* insertSort(*int* \*a, *int* len) {  
 *for* (*int* low = 1; low < len; low++) {  
 *int* t = a[low];  
 *int* i = low - 1;  
 *//寻找插入位置  
 while* (0 <= i && t < a[i]) {  
 a[i + 1] = a[i];*//比待插入值大的右移,空出一个位置* i--;  
 }  
 *if* (i != low - 1) {*//找到插入位置* a[i + 1] = t;  
 }  
 printArr(a, len);  
 }  
}  
  
*int* main() {  
 *int* arr1[] = {3, 5, 7, 4, 6, 1, 2};  
 *int* len1 = *sizeof*(arr1) / *sizeof*(*int*);  
 insertSort(arr1, len1);  
 printf("排序1结果:");  
 printArr(arr1, len1);  
  
 *int* arr2[] = {4, 1, 6, 2};  
 *int* len2 = *sizeof*(arr2) / *sizeof*(*int*);  
 insertSort(arr2, len2);  
 printf("排序2结果:");  
 printArr(arr2, len2);  
}

1. 希尔排序

*#include* "ArrayUtil.h"  
  
*/\*\*  
 \* 希尔排序  
 \* 数据分组,每组间隙为gap(例如8个元素分4组,间隙为4,索引0与4 1与5 2与6 3与7)  
 \* 每组内插入排序,排序后gap变小,直到gap为1完成排序(每次除2)  
 \*/  
void* shellSort(*int* \*a, *int* len) {  
 *//printf("开始排序:\n");  
 for* (*int* gap = len >> 1; gap >= 1; gap = gap >> 1) {*//使用位运算代替除法  
 //插入排序,但是不比较相邻元素,而是相隔gap的元素  
 for* (*int* low = gap; low < len; low++) {*//low=gap,第一个元素视为已排;low++,排下一组  
 int* t = a[low];*//当前未排序元素  
 int* i = low - gap;*//同组的最右侧已排序元素索引  
 while* (0 <= i && t < a[i]) {*//寻找插入位置* a[i + gap] = a[i];*//比待插入值大的右移,空出一个位置* i -= gap;*//同组的元素* }  
 *if* (i != low - gap) {*//找到插入位置* a[i + gap] = t;  
 }  
 }  
 printArr(a, len);  
 }  
}  
  
*int* main() {  
 *int* arr1[] = {3, 5, 7, 4, 6, 1, 2};  
 *int* len1 = *sizeof*(arr1) / *sizeof*(*int*);  
 shellSort(arr1, len1);  
 printf("排序1结果:");  
 printArr(arr1, len1);  
  
 *int* arr2[] = {4, 1, 6, 2};  
 *int* len2 = *sizeof*(arr2) / *sizeof*(*int*);  
 shellSort(arr2, len2);  
 printf("排序2结果:");  
 printArr(arr2, len2);  
}

1. 冒泡排序

*#include* <stdio.h>  
*#include* "ArrayUtil.h"  
  
*/\*\*  
 \* 冒泡排序,每轮比较相邻元素  
 \* 如果前一个元素大于后一个元素,则交换  
 \*/  
void* bubbleSort1(*int* \*a, *int* len) {  
 *for* (*int* i = 0; i < len - 1; i++) {  
 *for* (*int* j = 0; j < len - i - 1; j++) {  
 *if* (a[j] > a[j + 1]) {  
 *//交换元素值* swap(a, j, j + 1);  
 }  
 }  
 printArr(a, len);  
 }  
}  
  
*/\*\*  
 \* 冒泡排序优化  
 \* 加入一个指针x,记录本轮的最后一次交换位置  
 \* 减少了比较次数  
 \*/  
void* bubbleSort2(*int* \*a, *int* len) {  
  
 *int* j = len - 1;  
 *do* {  
 *int* x = 0;  
 *for* (*int* i = 0; i < j; i++) {*//遍历0~j的元素  
 if* (a[i] > a[i + 1]) {  
 swap(a, i, i + 1);  
 x = i;*//x记录一次遍历中最后一次交换的位置(无序的右边界)* }  
 }  
 j = x;  
 printArr(a, len);  
 } *while* (j != 0);  
}  
  
*int* main() {  
 *int* arr1[] = {3, 5, 7, 4, 6, 1, 2};  
 *int* len1 = *sizeof*(arr1) / *sizeof*(*int*);  
 printf("开始排序:\n");  
 bubbleSort1(arr1, len1);  
 printf("排序1结果:");  
 printArr(arr1, len1);  
  
 *int* arr2[] = {4, 1, 6, 2};  
 *int* len2 = *sizeof*(arr2) / *sizeof*(*int*);  
 printf("开始排序:\n");  
 bubbleSort2(arr2, len2);  
 printf("排序2结果:");  
 printArr(arr2, len2);  
}

1. 快速排序

*// Author: 绝迹的星   
// Created on 2023/12/15  
  
  
#include* <stdio.h>  
*#include* <stdbool.h>  
*#include* <stdlib.h>  
*#include* "ArrayUtil.h"  
  
  
*/\*\*  
 \* 随机基准点分区  
 \*/  
int* partition(*int* \*a, *int* left, *int* right) {  
 *int* index = rand() % (right - left + 1) + left;  
 swap(a, index, left);  
 *int* pv = a[left];*//基准点元素值  
 int* i = left;  
 *int* j = right;  
 *while* (i < j) {  
 *while* (i < j && a[j] > pv) {  
 *//寻找小的* j--;  
 }  
 *while* (i < j && pv >= a[i]) {*//内层循环也要判断i<j  
 //寻找大的* i++;  
 }  
 *//找到则交换* swap(a, i, j);  
 }  
 swap(a, left, i);  
 *return* i;  
}  
  
*void* quick(*int* \*a, *int* len, *int* left, *int* right) {  
 *//递归主函数  
 if* (left >= right) {  
 *return*;  
 }  
 *//快排核心方法:分区  
 // 找基准点,小的放左边,大的放右边  
 int* p = partition(a, left, right);  
 printf("分区元素的元素值:%d 分区后数组:",p);  
 printArr(a, len);  
 quick(a, len, left, p - 1);*//对两个区域再次分区排序* quick(a, len, p + 1, right);  
}  
  
  
*void* quickSort(*int* \*a, *int* len) {  
   
 quick(a, len, 0, len - 1);  
  
}  
  
*/\*\*  
 \* 选择排序  
 \* 每次在未排序区域选择最小元素放在数组排序区域的末端  
 \*/  
void* selectSort(*int* \*a, *int* len) {  
 printf("开始排序:\n");  
 *for* (*int* i = 0; i < len - 1; i++) {  
 *//在未排序区寻找最小元素  
 int* min = i;  
 *for* (*int* j = i + 1; j < len; j++) {  
 *if* (a[j] < a[min]) {  
 min = j;  
 }  
 }  
 swap(a, i, min);  
 printArr(a, len);  
 }  
}  
  
*int* main() {  
 *int* arr1[] = {3, 5, 7, 4, 6, 1, 2};  
 *int* len1 = *sizeof*(arr1) / *sizeof*(*int*);  
 printf("开始排序:\n");  
 quickSort(arr1, len1);  
 printf("排序1结果:");  
 printArr(arr1, len1);  
  
 *int* arr2[] = {4, 1, 6, 2};  
 *int* len2 = *sizeof*(arr2) / *sizeof*(*int*);  
 printf("开始排序:\n");  
 quickSort(arr2, len2);  
 printf("排序2结果:");  
 printArr(arr2, len2);  
}

5.选择排序

*#include* <stdio.h>  
*#include* "ArrayUtil.h"  
  
*/\*\*  
 <div color=rgb(155,200,80)>  
 <h1>选择排序</h1>  
 </div>  
 \*/  
void* selectSort(*int* \*arr, *int* len) {  
 *// 选择轮数:length-1  
 // 交换位置(right) 初始length-1,每次递减  
 for* (*int* right = len - 1; right > 0; right--) {  
 *int* max = right;*//循环前面未排序区域,找到最大值的索引  
 for* (*int* i = 0; i < right; i++) {  
 *if* (arr[i] > arr[max]) {  
 max = i;  
 }  
 }  
 *if* (max != right) {  
 swap(arr, max, right);*//交换* }  
 printArr(arr, len);  
 }  
}  
  
*int* main() {  
 *int* arr1[] = {3, 5, 7, 4, 6, 1, 2};  
 *int* len1 = *sizeof*(arr1) / *sizeof*(*int*);  
 printf("开始排序:\n");  
 selectSort(arr1, len1);  
 printf("排序1结果:");  
 printArr(arr1, len1);  
  
 *int* arr2[] = {4, 1, 6, 2};  
 *int* len2 = *sizeof*(arr2) / *sizeof*(*int*);  
 printf("开始排序:\n");  
 selectSort(arr2, len2);  
 printf("排序2结果:");  
 printArr(arr2, len2);  
}

6.堆排序

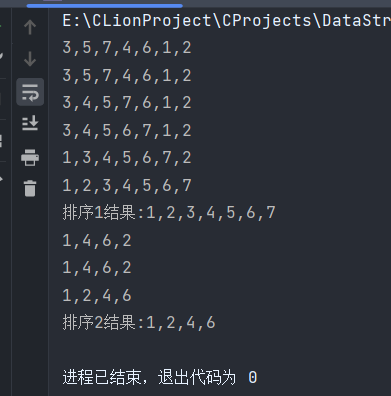
*#include* "ArrayUtil.h"  
  
*//下潜  
void* down(*int* \*array, *int* parent, *int* size) {*//小的元素下潜  
 int* left = 2 \* parent + 1;  
 *int* right = left + 1;  
 *int* max = parent;*//寻找 父,左,右 三者较大的  
 if* (left < size && array[left] > array[max]) {  
 max = left;  
 }  
 *if* (right < size && array[right] > array[max]) {  
 max = right;  
 }  
 *if* (max != parent) {*//如果子节点比父节点优先级大* swap(array, parent, max);*// 交换* down(array, max, size);*//递归,直到max==parent,父节点大于左右子节点时停止* }  
}  
  
*//建堆  
void* heapify(*int* \*array, *int* size) {  
 *for* (*int* i = size / 2 - 1; i >= 0; i--) {  
 down(array, i, size);*//数组前一半的元素进行下潜* }  
}  
  
*void* heapSort(*int* \*arr, *int* len) {  
 heapify(arr, len);*//建堆  
 for* (*int* right = len - 1; right > 0; right--) {  
 *//交换第一个元素(大顶堆第一个元素为最大元素)与未排序区域最右侧元素  
 // 并重新下潜排序(范围到未排序区域的最右侧)* swap(arr, 0, right);  
 down(arr, 0, right);  
 printArr(arr,len);  
 }  
}  
  
*int* main() {  
 *int* arr1[] = {3, 5, 7, 4, 6, 1, 2};  
 *int* len1 = *sizeof*(arr1) / *sizeof*(*int*);  
 printf("开始排序:\n");  
 heapSort(arr1, len1);  
 printf("排序1结果:");  
 printArr(arr1, len1);  
  
 *int* arr2[] = {4, 1, 6, 2};  
 *int* len2 = *sizeof*(arr2) / *sizeof*(*int*);  
 printf("开始排序:\n");  
 heapSort(arr2, len2);  
 printf("排序2结果:");  
 printArr(arr2, len2);  
}

7.归并排序

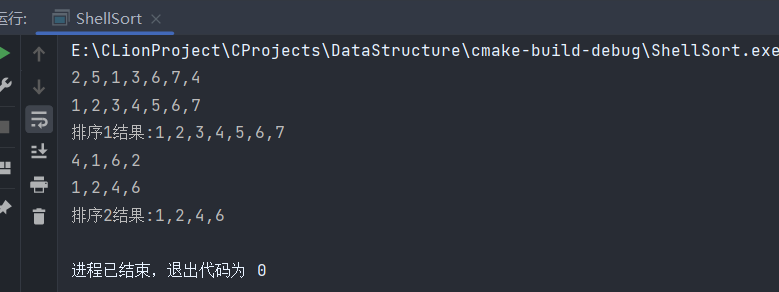
*#include* <stdlib.h>  
*#include* "ArrayUtil.h"  
  
*/\*\*  
 合并两个有序数组  
  
 @param a1 数据源  
 @param i 第一个数组的起始索引  
 @param iEnd 第一个数组的结束索引  
 @param j 第二个数组的起始索引  
 @param jEnd 第二个数组的结束索引  
 @param a2 临时数组  
 \*/  
void* merge(*int* \*a1, *int* i, *int* iEnd, *int* j, *int* jEnd, *int* \*a2) {  
 *int* k = 0;  
 *while* (i <= iEnd && j <= jEnd) {  
 *if* (a1[i] < a1[j]) {  
 a2[k] = a1[i];  
 i++;  
 } *else* {  
 a2[k] = a1[j];  
 j++;  
 }  
 k++;  
 }  
 *//处理剩余没放入a2的部分  
 if* (i > iEnd) {  
 *for* (*int* l = j; l <= jEnd; l++) {  
 a2[k++] = a1[l];  
 }  
  
 }  
 *if* (j > jEnd) {  
 *for* (*int* l = i; l <= iEnd; l++) {  
 a2[k++] = a1[l];  
 }  
 }  
  
}  
  
*/\*\*  
 \* 分治  
 \*/  
void* split(*int* \*a1, *int* left, *int* right, *int* \*a2, *int* len) {  
  
 *// 2.治  
 if* (left == right) {  
 *return*;  
 }  
 *// 1.分  
 int* m = (left + right) >> 1;*//从中间分隔* split(a1, left, m, a2, len);  
 split(a1, m + 1, right, a2, len);  
  
 *// 3.合* merge(a1, left, m, m + 1, right, a2);*//合并两个有序区域到a2  
 for* (*int* i = 0; i < right - left + 1; i++) {  
 a1[i + left] = a2[i];*//将合并后的再存入a1* }  
 printArr(a1, len);  
}  
  
*void* mergeSort(*int* \*a1, *int* len) {  
 *int* \*a2 = malloc(*sizeof*(*int*) \* len);*//临时数组,在合并时将数据放入* split(a1, 0, len - 1, a2, len);  
 free(a2);  
}  
  
*int* main() {  
 *int* arr1[] = {3, 5, 7, 4, 6, 1, 2};  
 *int* len1 = *sizeof*(arr1) / *sizeof*(*int*);  
 printf("开始排序:\n");  
 mergeSort(arr1, len1);  
 printf("排序1结果:");  
 printArr(arr1, len1);  
  
 *int* arr2[] = {4, 1, 6, 2};  
 *int* len2 = *sizeof*(arr2) / *sizeof*(*int*);  
 printf("开始排序:\n");  
 mergeSort(arr2, len2);  
 printf("排序2结果:");  
 printArr(arr2, len2);  
}

六、实验数据及处理结果

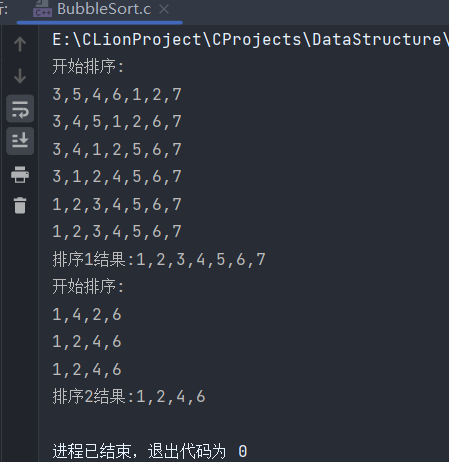
1. 插入排序结果:



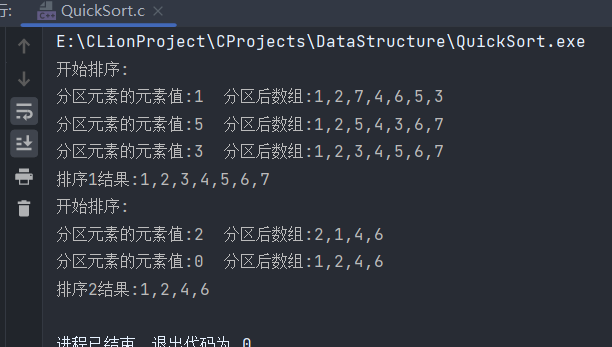
1. 希尔排序结果:



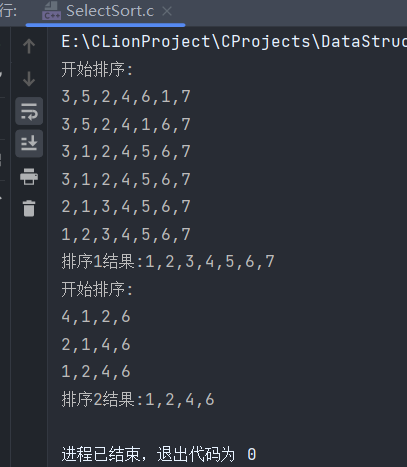
1. 冒泡排序结果:



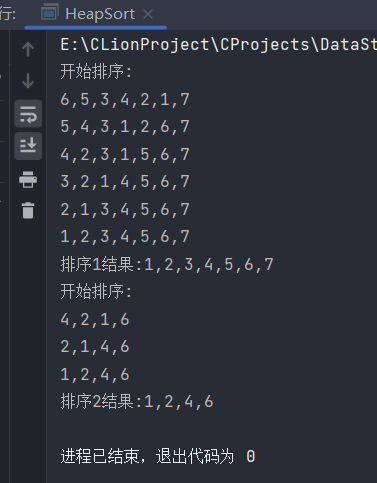
4.快速排序结果:



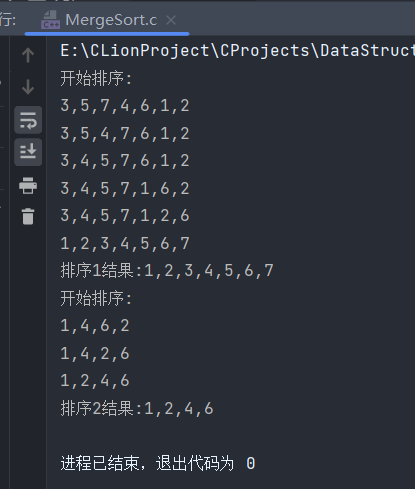
5.选择排序结果:



6.堆排序结果:



7.归并排序结果:



七、思考讨论题或体会或对改进实验的建议

某些排序算法仍有优化的空间,比如归并排序,进行分治的时候不需要使区域只剩一个元素,当元素个数恰当时就可以使用其他排序算法进行排序

八、参考资料

数据结构