# 数据结构实验报告——实验四

## 学号： 20201060330 姓名： 胡诚皓 得分：\_\_\_\_\_\_\_\_\_\_\_\_\_\_

### 一、实验目的

1. 复习线性表的逻辑结构、存储结构及基本操作；
2. 掌握顺序表和（带头结点）单链表；
3. 了解有序表。

### 二、实验内容

1. （必做题）有序线性表的构造与合并

假设有序表中数据元素类型是整型，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；

1. OrderInput(&L, int (\*compare)(a, b))

//根据有序判定函数compare，并利用有序插入函数OrderInsert，构造有序表L；

1. 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。

1. （必做题）多项式的构造与相加

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，约定系数不能等于0，指数不能小于0；
2. 两个升幂多项式的相加。
3. （选做题）约瑟夫环

问题描述：将数字1,2，...，n环形排列；按顺时针方向从1开始计数，计满k时输出该位置上的数字，同时从环中删除该数字；然后从下一位置开始重新开始计数，直到环中所有数均被输出为止。

请使用顺序表或链表实现：对输入的任意n和k，输出相应的出列序。

### 三、数据结构及算法描述

1. （必做题）有序线性表的构造与合并
2. 数据结构

此题使用了带头结点的链式存储的线性表，Lnode结构体作为每个结点，Link为指向Lnode的指针类型，其中的数据域data以int型为例，指针域next为指向下一个结点的指针。宏定义了实际为int型的Status类型，OK（实际为1）与ERROR（实际为0）作为Status类型的值。

其中各个函数参数中的int (\*compare)(Lnode a, Lnode b) 函数指针都代表“若compare的返回值不为0时，认为a应该在b的前面”其中的a、b都为Lnode类型的形参。不单纯使用返回比较两个int值的函数指针目的在于，当Lnode中的数据域的定义不只是int时，各个有compare函数指针的函数更具普适性。

1. 算法描述

程序中有三个辅助函数Lnode \*createNode(int data)、Lnode \*deepCpy(const Lnode \*src)、void displayLink(Link head)，分别用于创建以data为数据域的结点并返回指向这个新创建的结点的指针、深拷贝src指向的结点元素并返回指向拷贝出来的新结点的指针、输出以head为头结点的整个链表。

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

* 1. 判断L是否为只有头结点的空链表，若为空链表，则直接将新的结点e放到链表中；
  2. 由于之前每次添加结点时都保证以compare的规则进行排列，因此只要保证这一次将要添加的结点放在合适的位置即可。从头往尾遍历链表，直到找到某个结点，这个结点的下一个结点按照compare规则应该位于要添加的结点的后面。那么，要添加的结点的合适位置就在这个遍历到的结点的下一个，将要添加的结点插入到遍历到的结点的后面；
  3. 若没有遍历到符合②中条件的结点，说明链表中所有已有结点按照compare规则都应该处于要添加的结点的前面，就把要添加的结点放在链表的最后。

OrderInput(&L, int (\*compare)(a, b))

* 1. 让用户输入要添加结点个数；
  2. 让用户依次输入要添加的结点的数据域中int型的data，先调用createNode创建新结点，再调用OrderInsert将新结点插入到链式存储结构有序表的合适位置。

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

* 1. 判断La、Lb是否为空，若有为空的直接返回ERROR；判断Lc是否为空，因为要将合并后的有序表存在以Lc为头结点的链表中，Lc不为空直接返回ERROR
  2. 从头开始分别遍历La、Lb并按照compare规则将La、Lb中的各个元素分别放入Lc中。因为是从头往尾遍历La、Lb，先要插入Lc的是一定是靠前的元素，所以使用尾插法向Lc中插入结点。为了不影响La、Lb本身的构造，调用deepCpy拷贝各个要加入Lc的结点，将深拷贝得到的结点插入Lc中。
  3. 将La、Lb中剩余的未加入Lc中的结点元素都加入Lc中

1. （必做题）多项式的构造与相加
   1. 数据结构

此题使用Lnode作为存储多项式的某一项，Lnode中的coefficient存储该项的系数、frequency存储该项的次数，Link为Lnode类型的指针。宏定义了实际为int型的Status类型，OK（实际为1）与ERROR（实际为0）作为Status类型的值。

其中各个函数参数中的int (\*compare)(Lnode a, Lnode b) 函数指针都代表“若compare的返回值不为0时，认为a应该在b的前面”其中的a、b都为Lnode类型的形参。

本题中按照次数升序排列各项，byAscending中就是按照Lnode a, Lnode b的frequency数据域进行比较，当a.frequency < b.frequency时，byAscending返回值不为零，意味着a应该在b的前面。

* 1. 算法描述

为了方便编写其他功能，在此题中编写了Lnode \*createNode(int, int)、Lnode \*deepCpy(const Lnode \*)、void displayPoly(Link)三个辅助函数，分别用于根据两个参数创建某一项结点并返回指向多项式某一项结点的指针、深拷贝一个结点并返回一个指向拷贝出来的新结点的指针、输出多项式。同时继续沿用上一题中将结点插入链表的函数Status OrderInsert(Link, Lnode \*, int (\*)(Lnode, Lnode))

Status createPoly(Link head, int num, ...)

创建以head为头结点的多项式链表，num为想要创建的多项式的项数，之后为一个可变长度的参数列表，应为系数1、次数1、系数2、次数2......、系数num、次数num。

* + 1. 定义变量tmp用于在循环中暂时存储将要加入链表的结点，定义并初始化可变长度参数列表coeAndFre
    2. 每次循环都将num-1，一直循环到num为0，从可变长度参数列表中读取系数和次数，调用createNode创建新结点并用OrderInsert将新创建的结点插入链表
    3. 结束可变长度参数列表coeAndFre的使用

Status addPloy(Link polyA, Link polyB, Link polyC, int (\*compare)(Lnode a, Lnode b))

将以polyA、polyB为头结点的多项式相加，并把结果存储在以polyC为头结点的多项式链表中，使用compare函数作为各项前后顺序比较的规则

* + 1. polyA、polyB不能为空，polyC必须为空，不满足该条件直接返回ERROR
    2. 使pA、pB分别指向polyA与polyB的第一个实际结点元素。
    3. 由于多项式是根据compare规则从前往后排列的，若根据compare规则，pA所指向的结点应处于较前位置，则说明polyB中没有对应的项，也就无需进行相加操作，将结点深拷贝后直接加入polyC中，同时pA向后移试图在polyA中寻找与pB对应的项。
    4. 对于pB所指向的结点应处于较前位置的情况也是相同的。若根据compare规则，pB所指向的结点应处于较前位置，则说明polyA中没有对应的项，也就无需进行相加操作，将结点深拷贝后直接加入polyC中，同时pB向后移试图在polyB中寻找与pA对应的项。
    5. 若pA与pB根据compare规则是相对应的项，就调用createNode创建新结点，新结点的系数为这两个项的系数之和，次数不变。
    6. 将polyA或polyB中剩余的结点做深拷贝后加入polyC中

1. （选做题）约瑟夫环

（1）数据结构

使用循环链表来存储本题的结点，使用的是没有头结点的循环链表，即在第一个结点中直接存数据，本题在solve的循环中使用cur来指向当前判断的结点，pre为cur的前一个结点，目的是为了当cur是需要删除的结点时，可以直接让pre->next=cur->next

（2）算法描述

Status createLoopLink (Link head, int n)

用来创建以head为第一个结点的有n个人的约瑟夫环，需要一个rear指针，始终指向目前有数据的最后一个结点，在插入新结点时直接rear->next = tmp即可。

* + 1. 初始化局部变量，使得head->data = 1, rear=head
    2. i从2循环到n，创建各个结点并接入链表
    3. 将rear->next赋值为head，从而使得约瑟夫环闭合，形成循环链表

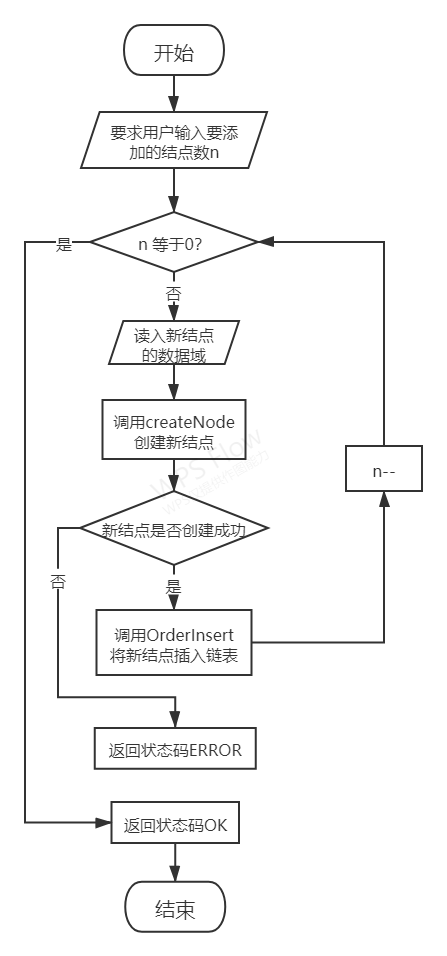
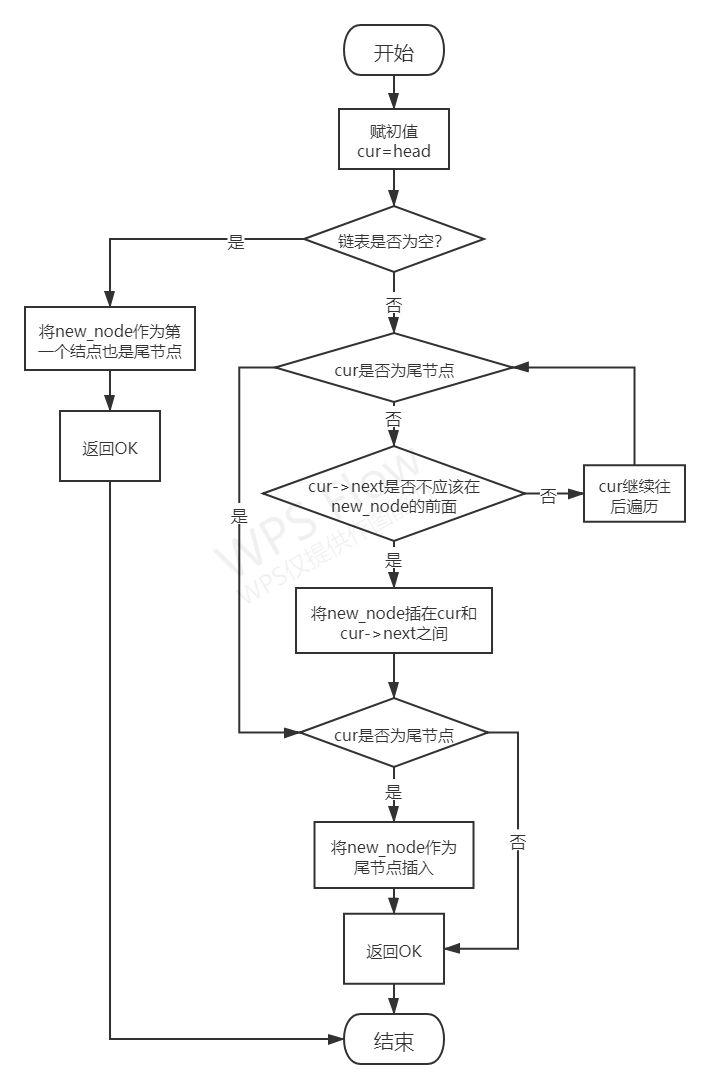
Status solve(Link head, int k)

用来解决题目描述的约瑟夫环的问题，由于head是一个循环链表，先找到head的前一个结点，也就是这个循环链表的“尾结点”，局部变量int count作为计数的临时变量，不断循环删除第k个结点。

* + 1. 初始化局部变量，使得cur=head, pre=head
    2. 不断循环cur，一边循环一边把cur往后移，直到cur的下一个元素是自己，也就是循环链表中只剩下一个元素的情况。
    3. 将count++，在使让count对k取余，若取余后为0，说明已经到了第k个，直接删除此时cur指向的结点。否则就直接往后循环，将pre、cur各自往后推。

### 详细设计

1. （必做题）有序线性表的构造与合并

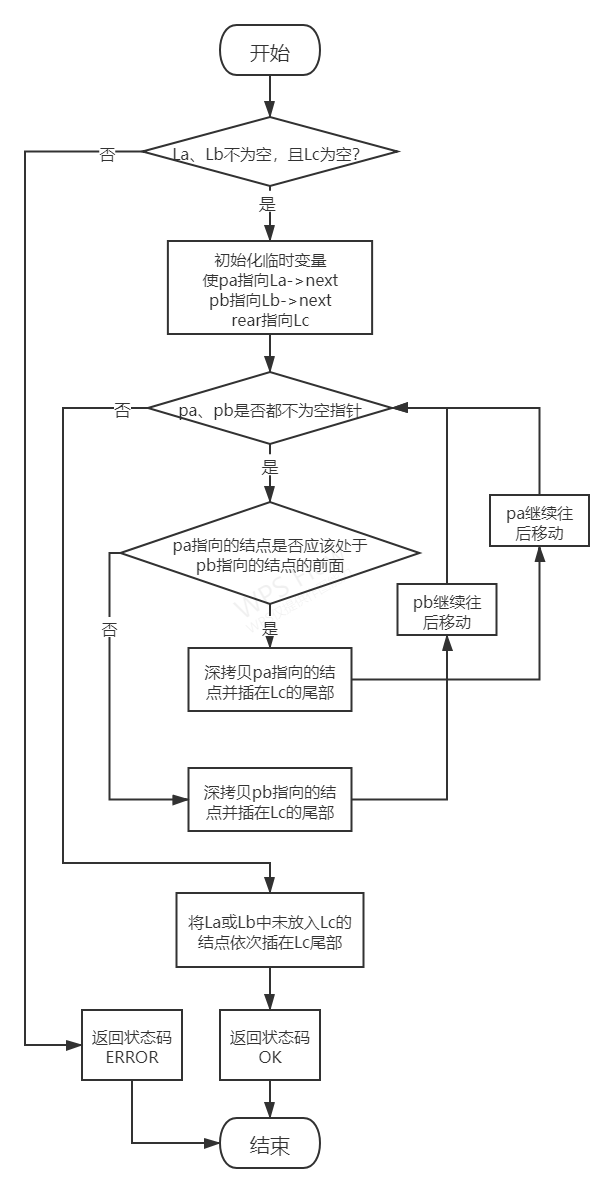


题4-1-2

OrderInput函数示意图

题4-1-1

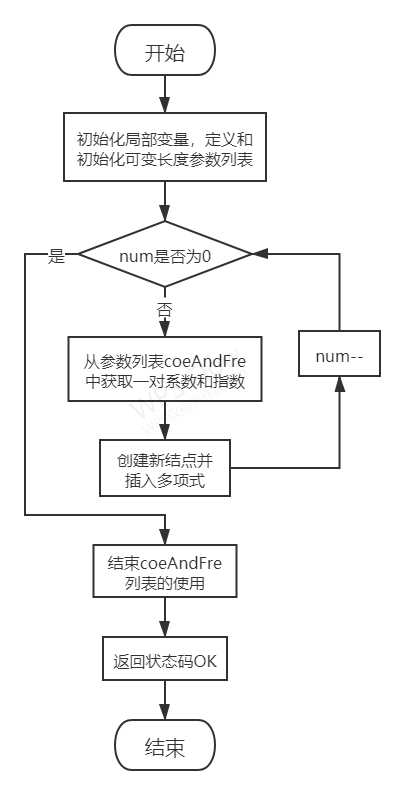
OrderInsert函数示意图



题4-1-3

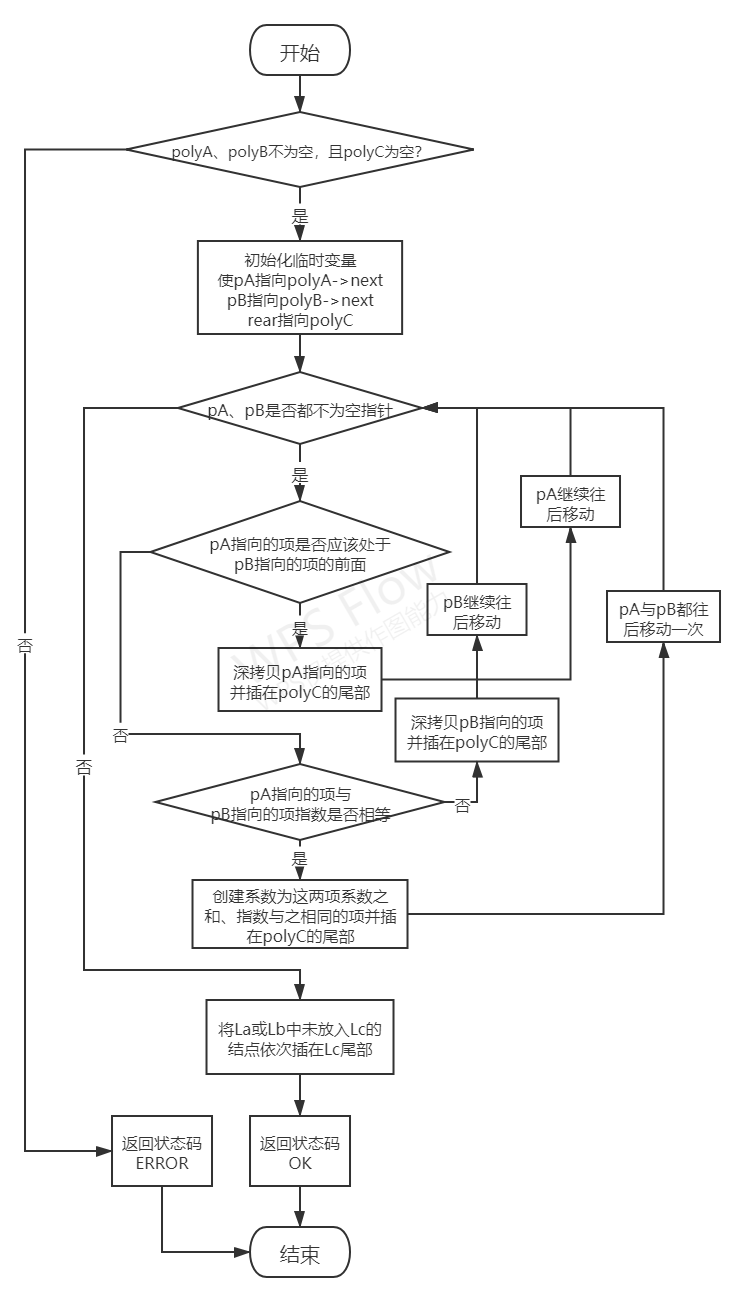
OrderMerge函数示意图

1. （必做题）多项式的构造与相加



题4-2-1

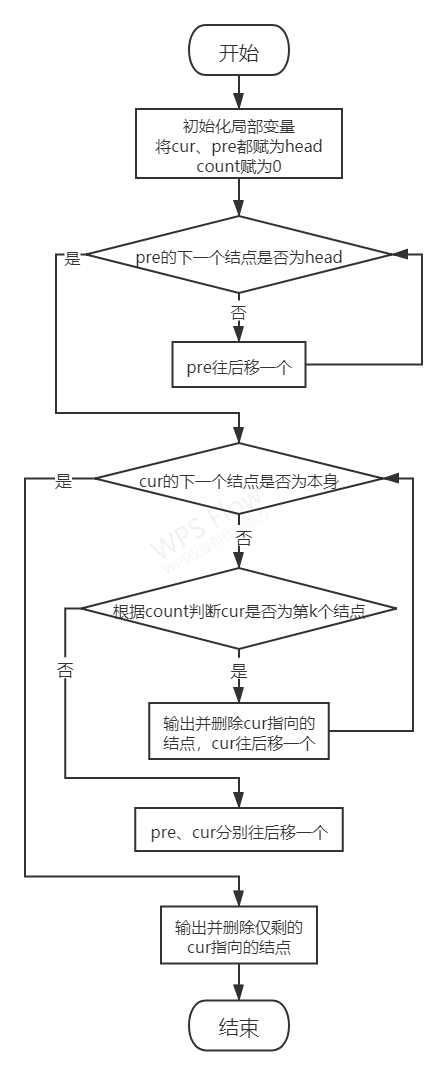
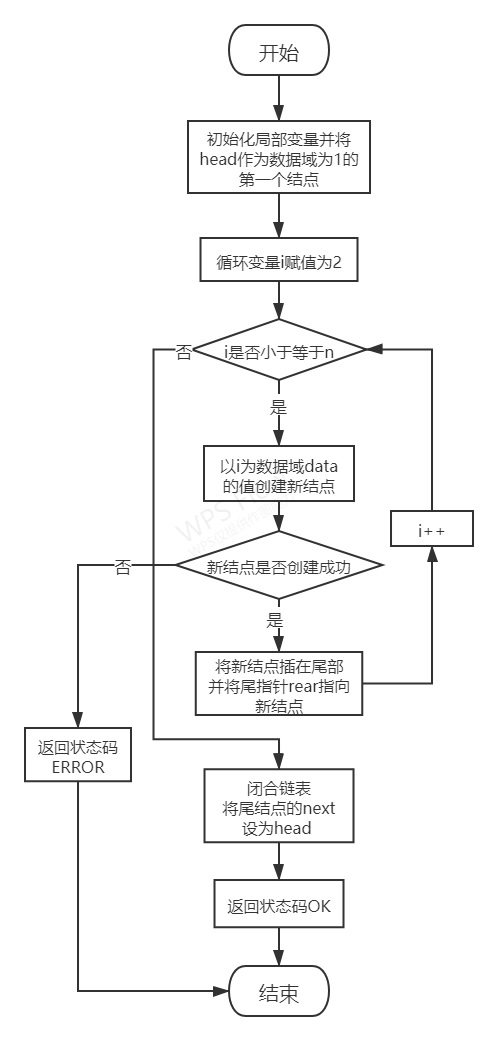
createPoly函数示意图



题4-2-2

addPloy函数示意图

1. （选做题）约瑟夫环



题4-3-2

solve函数示意图

题4-3-1

createLoopLink函数示意图

### 五、程序代码

1. （必做题）有序线性表的构造与合并



#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define Status int

#define OK 1

#define ERROR 0

typedef struct Lnode {

int data;

struct Lnode \*next;

} Lnode, \*Link;

Status OrderInsert(Link, Lnode \*, int (\*)(Lnode, Lnode));

Status OrderInput(Link head, int (\*)(Lnode, Lnode));

Status OrderMerge(Link, Link, Link, int (\*)(Lnode, Lnode));

Status OrderSort(Link head, int (\*)(Lnode, Lnode));

Lnode \*createNode(int);

Lnode \*deepCpy(const Lnode \*);

int byAscending(Lnode, Lnode);

void displayLink(Link);

int main() {

Link linkList1, linkList2;

Link mergeList;

Lnode \*tmp;

linkList1 = createNode(-1);

linkList2 = createNode(-1);

mergeList = createNode(-1);

printf("Create linkList1\n");

OrderInput(linkList1, byAscending);

printf("Create linkList2\n");

OrderInput(linkList2, byAscending);

printf("linkList1: ");

displayLink(linkList1);

printf("linkList2: ");

displayLink(linkList2);

OrderMerge(linkList1, linkList2, mergeList, byAscending);

printf("merged linklist1&2: ");

displayLink(mergeList);

printf("\n");

system("pause");

return 0;

}

//若compare的返回值不为0，认为a应该在b的前面

Status OrderInsert(Link head, Lnode \*new\_node, int (\*compare)(Lnode a, Lnode b)) {

Link cur = head;

if (head->next == NULL) {

new\_node->next = NULL;

head->next = new\_node;

return OK;

}

while (cur->next != NULL) {

if (!compare(\*(cur->next), \*new\_node)) {

new\_node->next = cur->next;

cur->next = new\_node;

break;

}

cur = cur->next;

}

if (cur->next == NULL) {

cur->next = new\_node;

new\_node->next = NULL;

}

return OK;

}

Status OrderInput(Link head, int (\*compare)(Lnode, Lnode)) {

int n, tmp\_data;

Lnode \*tmp;

printf("Input the number of nodes to add:\n");

scanf("%d", &n);

printf("Input integer data to save in each node\n");

printf("(each number ends with 'Enter' key or input in one line separating with space):\n");

while (n--) {

scanf("%d", &tmp\_data);

if ((tmp = createNode(tmp\_data)) == ERROR)

return ERROR;

OrderInsert(head, tmp, compare);

}

return OK;

}

//若compare的返回值不为0，认为a应该在b的前面

Status OrderSort(Link head, int (\*compare)(Lnode a, Lnode b)) {

//需要调整的链表是空的，直接返回ERROR

if (head->next == NULL)

return ERROR;

//p始终指向链表中还未调整位置的第一个结点

Lnode \*p = head->next->next, \*tmp;

head->next->next = NULL;

while (p) {

tmp = p->next;

OrderInsert(head, p, compare);

p = tmp;

}

return OK;

}

//若compare的返回值不为0，认为a应该在b的前面

//La与Lb必须是与将要产生的Lc一样，按照同一种compare方法构造的有序表

Status OrderMerge(Link La, Link Lb, Link Lc, int (\*compare)(Lnode a, Lnode b)) {

//Lc作为La和Lb合并后的链表的头结点，一开始需要是空的

//La、Lb也都不能是空的，不然没有合并的意义

if (Lc->next != NULL || La->next == NULL || Lb->next == NULL)

return ERROR;

//使用尾插法构造Lc

Lnode \*pa = La->next, \*pb = Lb->next, \*rear = Lc;

Lnode \*tmp;

while (pa != NULL && pb != NULL) {

//pa应该放前面

if (compare(\*pa, \*pb)) {

//深拷贝结点加入Lc中，防止La、Lb中的结点受到影响被改变

tmp = deepCpy(pa);

rear->next = tmp;

rear = tmp;

pa = pa->next;

rear->next = NULL;

} else {

tmp = deepCpy(pb);

rear->next = tmp;

rear = tmp;

pb = pb->next;

rear->next = NULL;

}

}

//剩余元素放入

while (pa != NULL) {

tmp = deepCpy(pa);

rear->next = tmp;

rear = tmp;

pa = pa->next;

rear->next = NULL;

}

while (pb != NULL) {

tmp = deepCpy(pb);

rear->next = tmp;

rear = tmp;

pb = pb->next;

rear->next = NULL;

}

return OK;

}

int byAscending(Lnode a, Lnode b) {

return a.data < b.data;

}

Lnode \*createNode(int data) {

Link p;

p = (Link) malloc(sizeof(Lnode));

p->data = data;

p->next = NULL;

return p;

}

void displayLink(Link head) {

if (head->next == NULL)

return;

Link p = head->next;

printf("%d", p->data);

p = p->next;

while (p) {

printf("->%d", p->data);

p = p->next;

}

printf("\n");

}

//返回对src结点的一份深拷贝

Lnode \*deepCpy(const Lnode \*src) {

Lnode \*res;

res = (Lnode \*) malloc(sizeof(Lnode));

res = (Lnode \*) memcpy(res, src, sizeof(Lnode));

return res;

}

1. （必做题）多项式的构造与相加



#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <stdarg.h>

#define Status int

#define OK 1

#define ERROR 0

typedef struct Lnode {

//存储系数

int coefficient;

//存储次数

int frequency;

struct Lnode \*next;

} Lnode, \*Link;

Status OrderInsert(Link, Lnode \*, int (\*)(Lnode, Lnode));

Status createPoly(Link head, int num, ...);

Status addPloy(Link, Link, Link, int (\*)(Lnode, Lnode));

Lnode \*createNode(int, int);

Lnode \*deepCpy(const Lnode \*);

int byAscending(Lnode, Lnode);

void displayPoly(Link);

int main() {

Link poly1, poly2, poly3;

poly1 = createNode(-1, -1);

poly2 = createNode(-1, -1);

poly3 = createNode(-1, -1);

createPoly(poly1, 2, 1, 2, 5, 6);

createPoly(poly2, 3, 1, 2, 1, 1, 1, 3);

printf(" ");

displayPoly(poly1);

printf("+");

displayPoly(poly2);

printf("=");

addPloy(poly1, poly2, poly3, byAscending);

displayPoly(poly3);

printf("\n");

system("pause");

return 0;

}

//创建多项式，使用以head为头结点的链表存储

//num为要创建的多项式的项数

//后面的参数依次为对应的系数、次数的交替

Status createPoly(Link head, int num, ...) {

Lnode \*tmp;

int tmpCoe, tmpFre;

//定义可变长度数组，用来接收多项式的系数和次数

va\_list coeAndFre;

//初始化参数列表

va\_start(coeAndFre, num);

while (num--) {

tmpCoe = va\_arg(coeAndFre, int);

tmpFre = va\_arg(coeAndFre, int);

tmp = createNode(tmpCoe, tmpFre);

OrderInsert(head, tmp, byAscending);

}

va\_end(coeAndFre);

return OK;

}

//将polyA与polyB两个多项式相加的结果存在polyC中

//polyA、polyB不能为空，polyC必须为空

//若compare的返回值不为0，认为a应该在b的前面

Status addPloy(Link polyA, Link polyB, Link polyC, int (\*compare)(Lnode a, Lnode b)) {

if (polyA->next == NULL || polyB->next == NULL || polyC->next != NULL)

return ERROR;

Link pA = polyA->next, pB = polyB->next;

//使用尾插法构造Lc

Link rear = polyC;

Lnode \*tmp;

while (pA != NULL && pB != NULL) {

//pA必处于较前位置，说明polyB中没有对应项，直接加入polyC中

if (compare(\*pA, \*pB)) {

//深拷贝结点

tmp = deepCpy(pA);

rear->next = tmp;

rear = tmp;

pA = pA->next;

rear->next = NULL;

}

//pA与pB次数相等，相加后加入polyC

else if (pA->frequency == pB->frequency) {

tmp = createNode(pA->coefficient + pB->coefficient, pA->frequency);

rear->next = tmp;

rear = tmp;

pA = pA->next;

pB = pB->next;

rear->next = NULL;

}

//pB必处于较前位置，说明polyA中没有对应项，直接加入polyC中

else {

tmp = deepCpy(pB);

rear->next = tmp;

rear = tmp;

pB = pB->next;

rear->next = NULL;

}

}

//剩余元素放入

while (pA != NULL) {

tmp = deepCpy(pA);

rear->next = tmp;

rear = tmp;

pA = pA->next;

rear->next = NULL;

}

while (pB != NULL) {

tmp = deepCpy(pB);

rear->next = tmp;

rear = tmp;

pB = pB->next;

rear->next = NULL;

}

return OK;

}

//若compare的返回值不为0，认为a应该在b的前面

Status OrderInsert(Link head, Lnode \*new\_node, int (\*compare)(Lnode a, Lnode b)) {

Link cur = head;

if (head->next == NULL) {

new\_node->next = NULL;

head->next = new\_node;

return OK;

}

while (cur->next != NULL) {

if (!compare(\*(cur->next), \*new\_node)) {

new\_node->next = cur->next;

cur->next = new\_node;

break;

}

cur = cur->next;

}

if (cur->next == NULL) {

cur->next = new\_node;

new\_node->next = NULL;

}

return OK;

}

//若compare的返回值不为0，认为a应该在b的前面

//若compare的返回值不为0，认为a应该在b的前面

//La与Lb必须是与将要产生的Lc一样，按照同一种compare方法构造的有序表

int byAscending(Lnode a, Lnode b) {

return a.frequency < b.frequency;

}

Lnode \*createNode(int coe, int fre) {

Link p;

p = (Link) malloc(sizeof(Lnode));

p->coefficient = coe;

p->frequency = fre;

p->next = NULL;

return p;

}

void displayPoly(Link head) {

//空链表，无需输出，直接返回

if (head->next == NULL)

return;

int tmpCoe, tmpFre;

Link p = head->next;

tmpCoe = p->coefficient;

tmpFre = p->frequency;

//首项特殊处理输出，正号无需输出

//0次方不用输出x，1次方不用输出次方号

if (tmpFre == 0) {

printf("%d", tmpCoe);

} else if (tmpFre == 1 && tmpCoe == 1) {

printf("x");

} else if (tmpFre == 1 && tmpCoe == -1) {

printf("-x");

} else if (tmpCoe == 1) {

printf("x^%d", tmpFre);

} else if (tmpCoe == -1) {

printf("-x^%d", tmpFre);

} else {

printf("%dx^%d", tmpCoe, tmpFre);

}

p = p->next;

while (p) {

tmpCoe = p->coefficient;

tmpFre = p->frequency;

if (tmpFre == 1 && tmpCoe == 1) {

printf("x");

} else if (tmpFre == 1 && tmpCoe == -1) {

printf("-x");

} else if (tmpCoe == 1) {

printf("+x^%d", tmpFre);

} else if (tmpCoe > 0) {

printf("+%dx^%d", tmpCoe, tmpFre);

} else if (tmpCoe == -1) {

printf("-x^%d", tmpFre);

} else {

printf("%dx^%d", tmpCoe, tmpFre);

}

p = p->next;

}

printf("\n");

}

//返回对src结点的一份深拷贝

Lnode \*deepCpy(const Lnode \*src) {

Lnode \*res;

res = (Lnode \*) malloc(sizeof(Lnode));

res = (Lnode \*) memcpy(res, src, sizeof(Lnode));

return res;

}

1. （选做题）约瑟夫环



#include <stdio.h>

#include <stdlib.h>

#define Status int

#define OK 1

#define ERROR 0

typedef struct Lnode {

int data;

struct Lnode \*next;

} Lnode, \*Link;

Status createLoopLink (Link, int);

Status displayLink (Link);

Status solve (Link, int);

int main() {

Link linklist;

int n, k;

linklist = (Link) malloc(sizeof(Lnode));

linklist->data = -1;

linklist->next = NULL;

printf("input n: ");

scanf("%d", &n);

printf("input k: ");

scanf("%d", &k);

createLoopLink(linklist, n);

//displayLink(linklist);

solve(linklist, k);

printf("\n");

system("pause");

return 0;

}

Status solve(Link head, int k) {

int count=0;

Link cur=head, pre=head;

Lnode \*tmp;

//找到“尾结点”作为头结点的前一个结点

while (pre->next != head)

pre = pre->next;

while (cur != cur->next) {

count++;

count %= k;

//删除第k个结点

if (count == 0) {

printf("%d\n", cur->data);

pre->next = cur->next;

tmp = cur;

cur = cur->next;

free(tmp);

} else {

pre = cur;

cur = cur->next;

}

}

printf("%d", cur->data);

free(cur);

return OK;

}

Status createLoopLink (Link head, int n) {

Lnode \*tmp;

Lnode \*rear;

head->data = 1;

rear = head;

for (int i = 2; i <= n; i++) {

tmp = (Lnode \*) malloc(sizeof(Lnode));

if (tmp == NULL)

return ERROR;

tmp->data = i;

tmp->next = NULL;

rear->next = tmp;

rear = tmp;

}

rear->next = head;

return OK;

}

Status displayLink (Link head) {

if (head == NULL)

return ERROR;

Link p=head;

printf("%d", p->data);

p = p->next;

while(p) {

printf("->%d", p->data);

p = p->next;

}

return OK;

}

### 六、测试和结果

1. （必做题）有序线性表的构造与合并

**Input:**

5

2 1 4 5 6

6

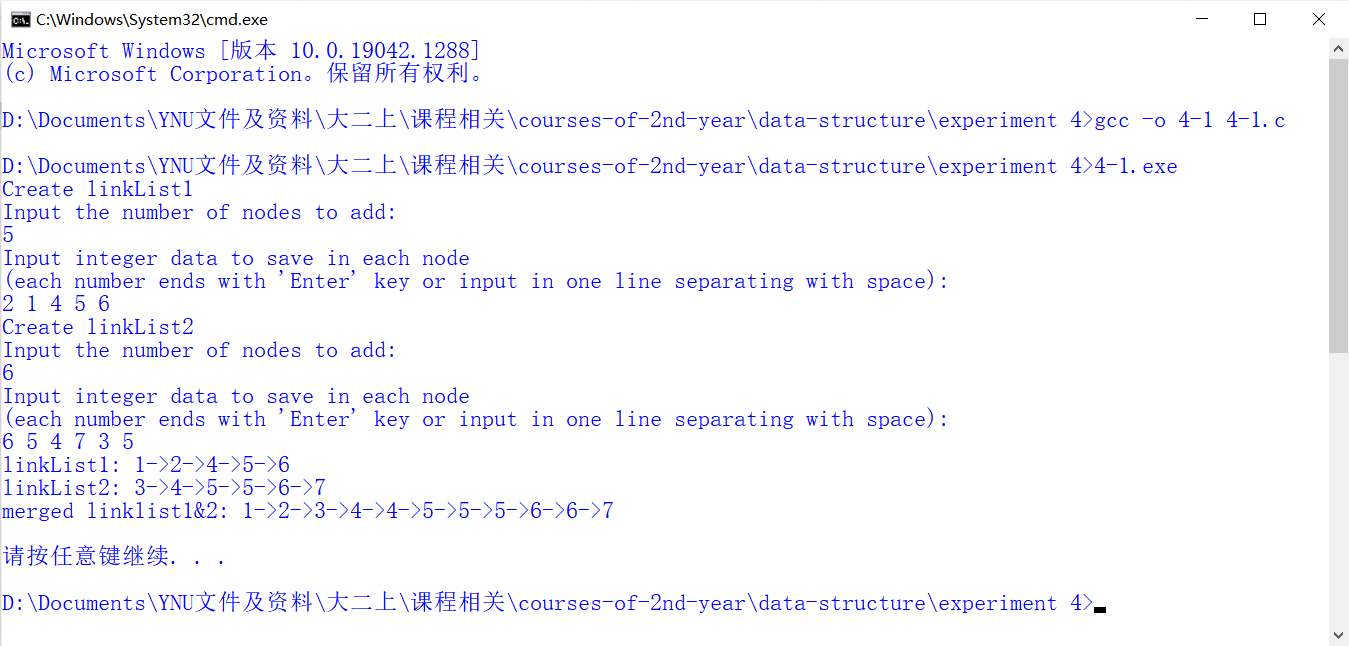
6 5 4 7 3 5

**Output:**

linkList1: 1->2->4->5->6

linkList2: 3->4->5->5->6->7

merged linklist1&2: 1->2->3->4->4->5->5->5->6->6->7



1. （必做题）多项式的构造与相加

**Input:**

createPoly(poly1, 2, 1, 2, 5, 6);

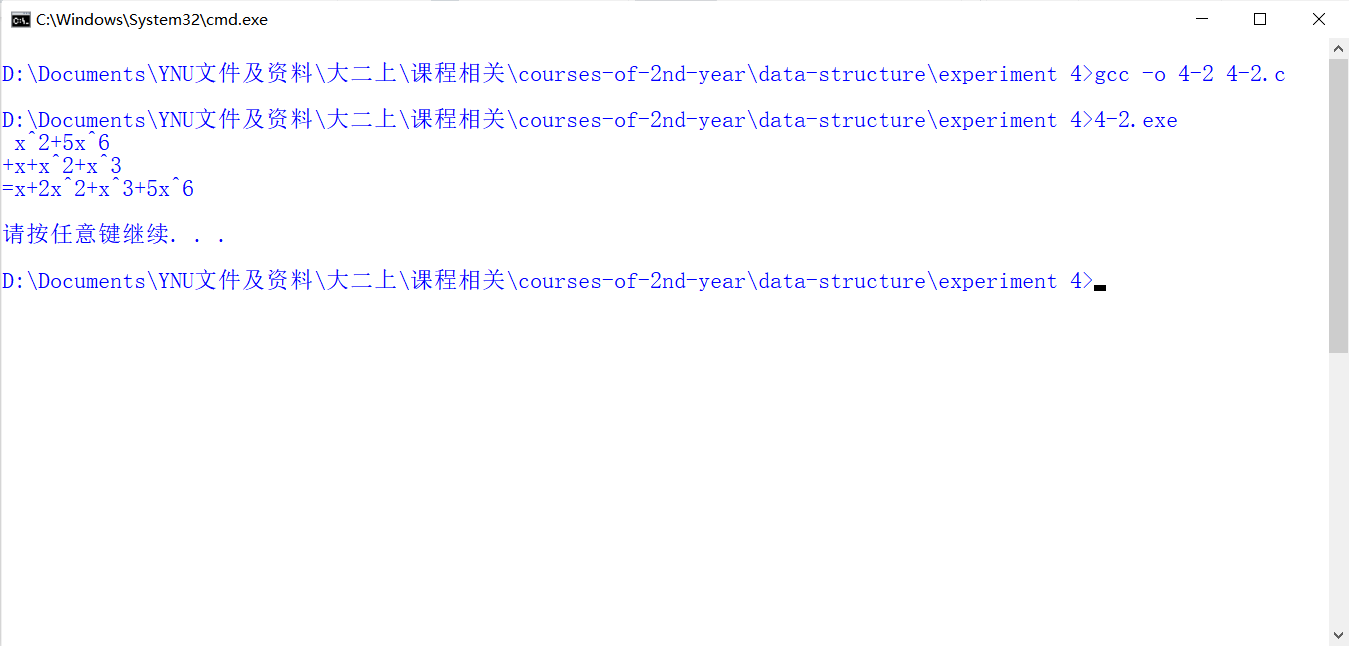
createPoly(poly2, 3, 1, 2, 1, 1, 1, 3);

**Output:**

x^2+5x^6

+x+x^2+x^3

=x+2x^2+x^3+5x^6



1. （选做题）约瑟夫环

**Input:**

10

4

**Output:**

4

8

2

7

3

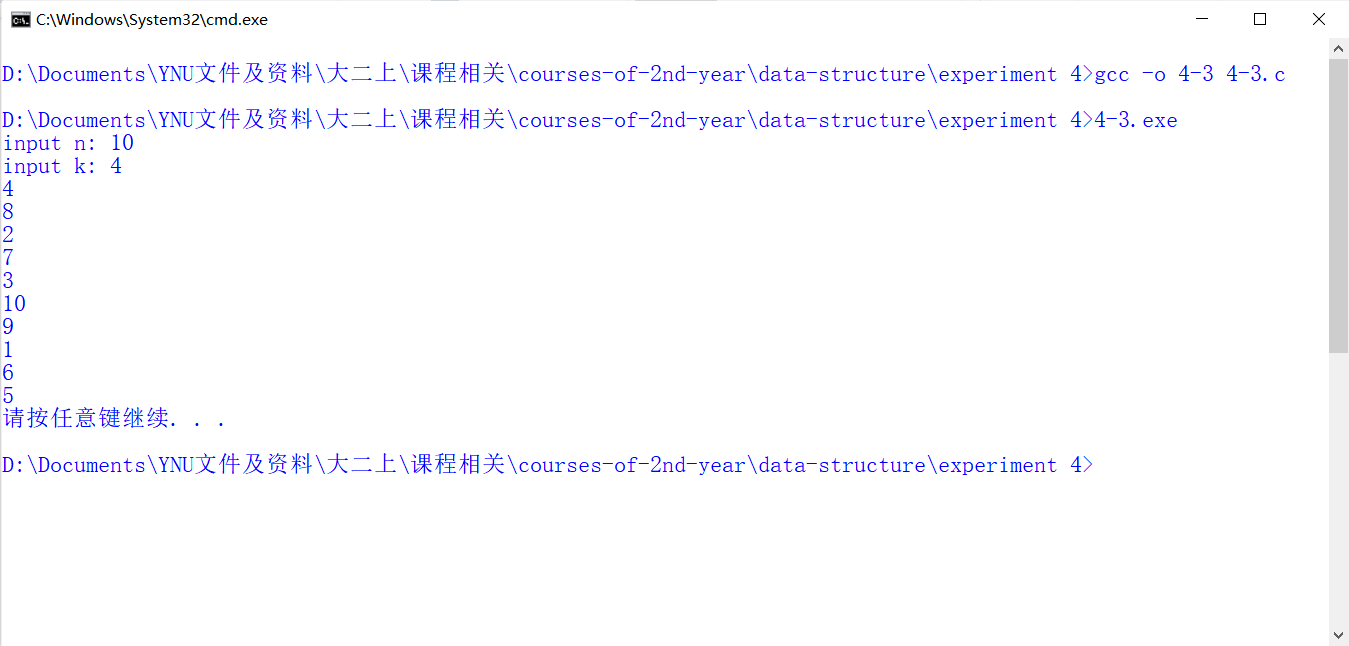
10

9

1

6

5



### 用户手册

1. （必做题）有序线性表的构造与合并

所有输入的数的范围都与int型一致。先输入要创建的第一个有序表中的元素个数，再依次输入各个元素数据域中data的值，以空格作为分隔；再输入要创建的第二个有序表中的元素个数，同样依次输入各个元素数据域中data的值，以空格作为分隔。

1. （必做题）多项式的构造与相加

此题为了函数的抽象通用性和低耦合性，将创建多项式的功能封装成了函数Status createPoly(Link head, int num, ...)，即使用以head为头结点的链表存储，参数num为要创建的多项式的项数，后面的参数依次为对应的系数、次数的交替（应有2\*num个）。

如createPoly(poly1, 2, 1, 2, 5, 6)表示创建以poly1作为头结点存储的多项式，有2项，每一项的系数和指数分别为1、2，5、6，即多项式，需要注意的是，系数、指数的参数列表中不能有指数相同的两项。

1. （选做题）约瑟夫环

输入的n、k均与int型范围保持一致。