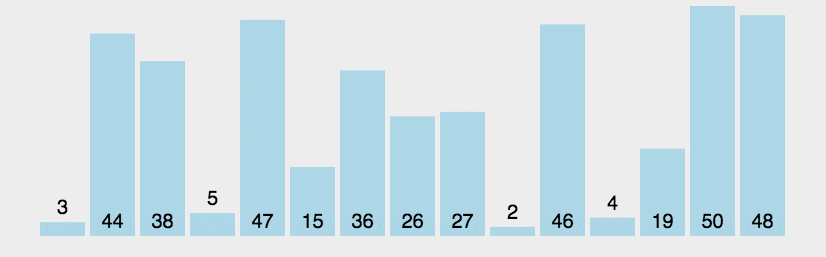
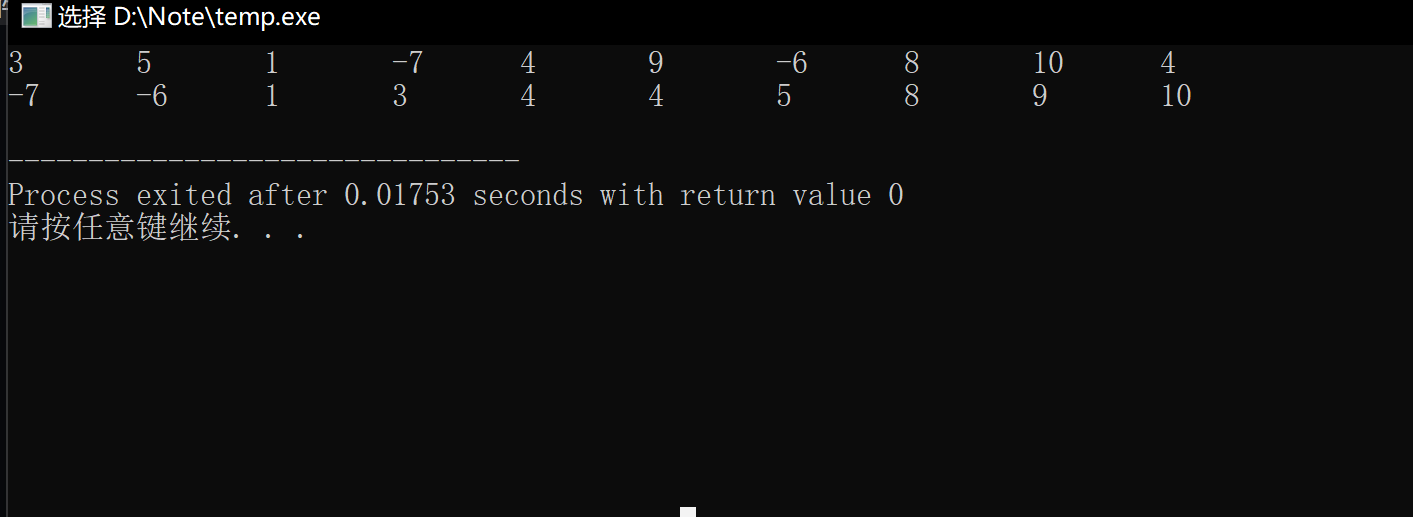
### *数据结构上机作业*

#### 1. 第一次作业：

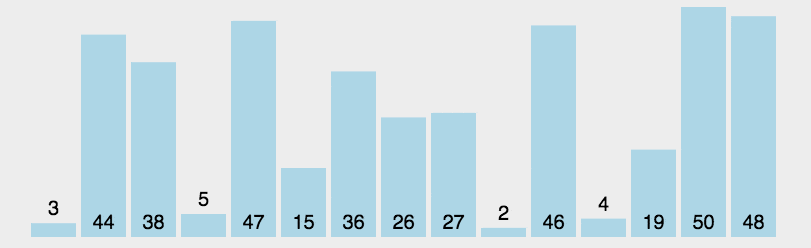
##### *（1）、实现冒泡排序*



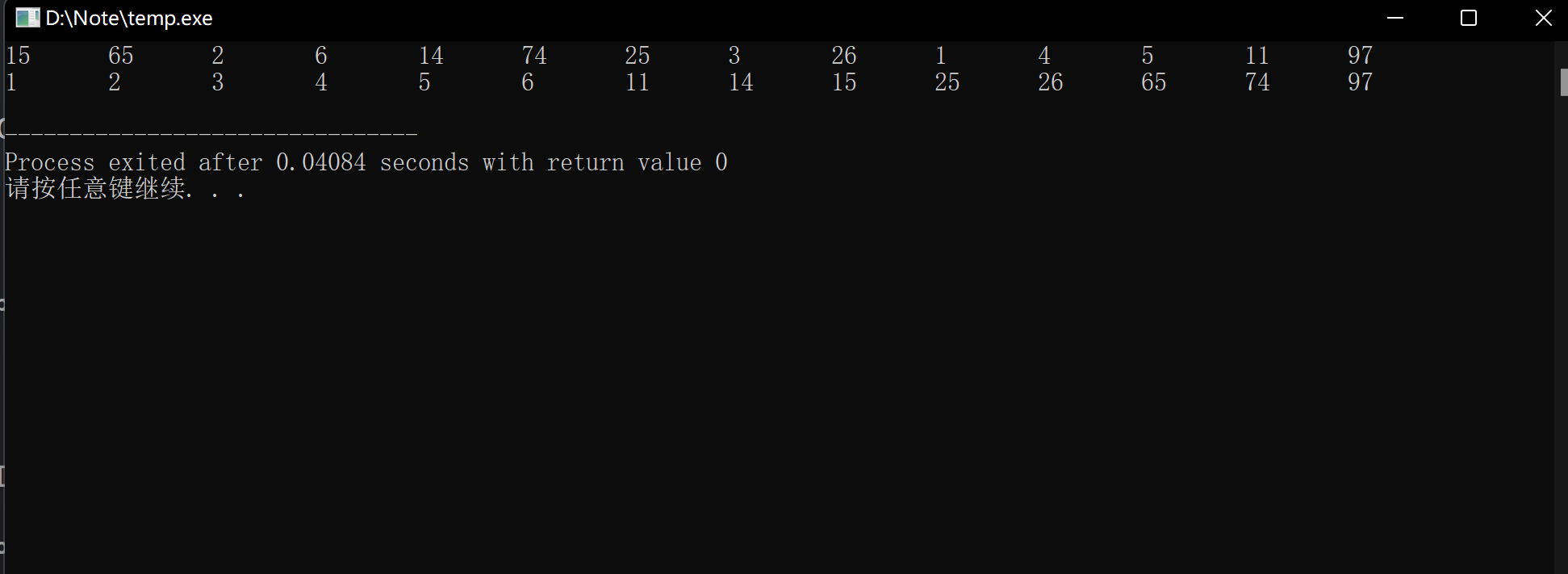
#include<stdio.h>  
#include<stdbool.h>  
  
#define ARR\_LEN 255 /\*数组长度上限\*/  
  
void bubbleSort(int arr[],int len)  
{  
 int i,j,temp;  
 \_Bool exchanged = true;  
   
 /\*exchanged值为true才有必要执行循环，否则值为false不执行循环\*/  
 for(i=0; exchanged && i<len-1; i++)   
 {  
 exchanged = false;  
 for(j=0; j<len-i-1; j++)  
 {  
 if (arr[j]>arr[j+1])  
 {  
 temp = arr[j];  
 arr[j] = arr[j+1];  
 arr[j+1] = temp;  
 exchanged = true;   
 }  
 }   
 }  
 }   
   
 int main(void){  
 int arr[ARR\_LEN]={3,5,1,-7,4,9,-6,8,10,4};  
 int len=10; // int len=sizeof(len)/sizeof(arr[0])  
 int i;  
   
 bubbleSort(arr,len);  
 for(i=0;i<len;i++)  
 printf("%d\t",arr[i]);  
 putchar('\n');  
   
 return 0;  
 }



##### *（2）、实现选择排序*



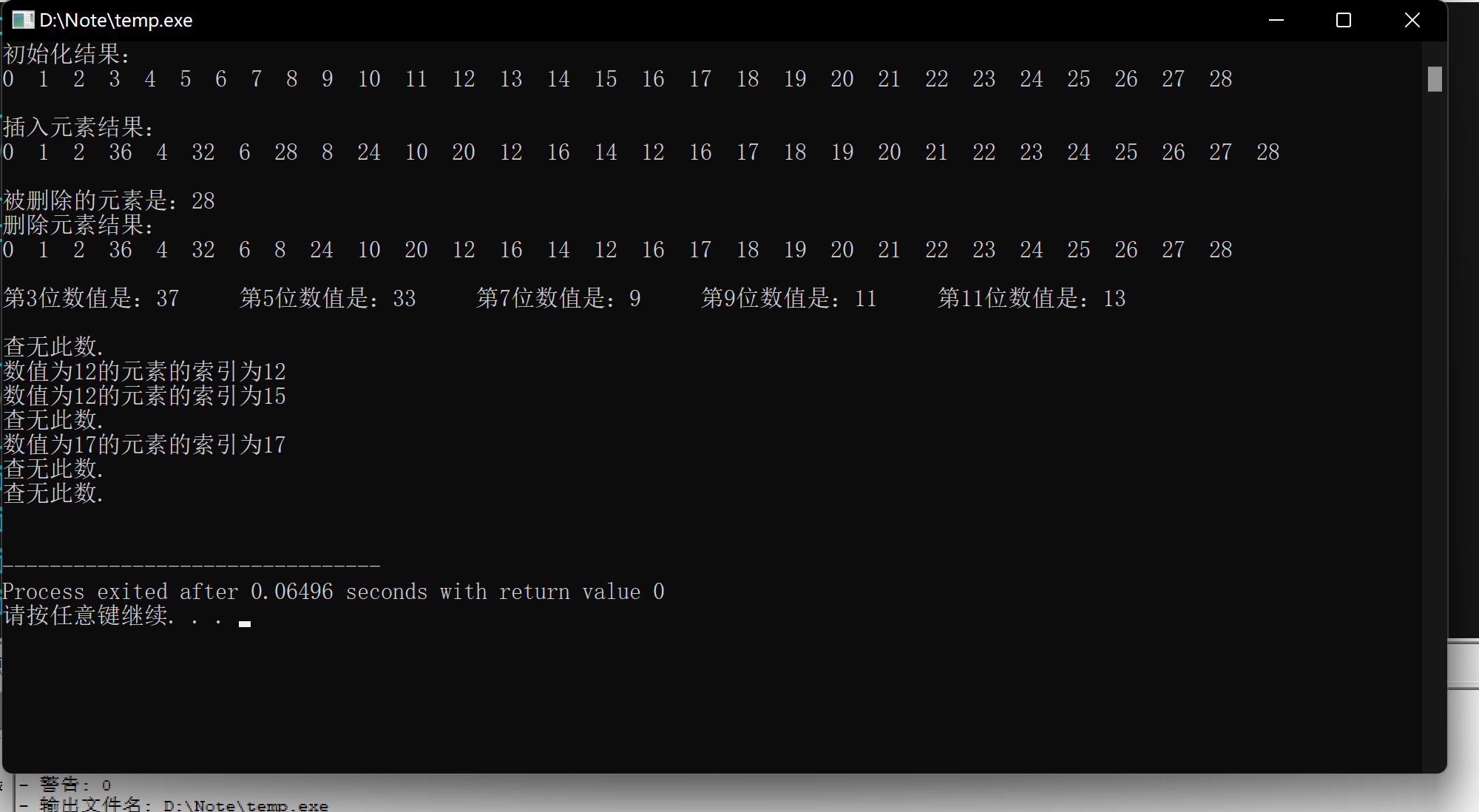
#include<stdio.h>  
  
void swap(int \*a,int \*b){  
 int temp = \*a;  
 \*a = \*b;  
 \*b = temp;  
}  
void selectionSort(int arr[],int len){  
   
 for(int i=0;i<len-1;i++){  
 int min = i;  
   
 for(int j=i+1;j<len;j++){  
 if(arr[j]<arr[min]){  
 min = j;  
 }  
 }  
 swap(&arr[min],&arr[i]);  
 }  
}  
  
int main(){  
 int arr[]={15,65,2,6,14,74,25,3,26,1,4,5,11,97};  
 int len = sizeof(arr)/sizeof(arr[0]);  
 selectionSort(arr,len);  
 for(int i=0;i<len;i++)  
 printf("%d\t",arr[i]);  
 putchar('\n');  
   
 return 0;  
}



#### 2、第二次作业：

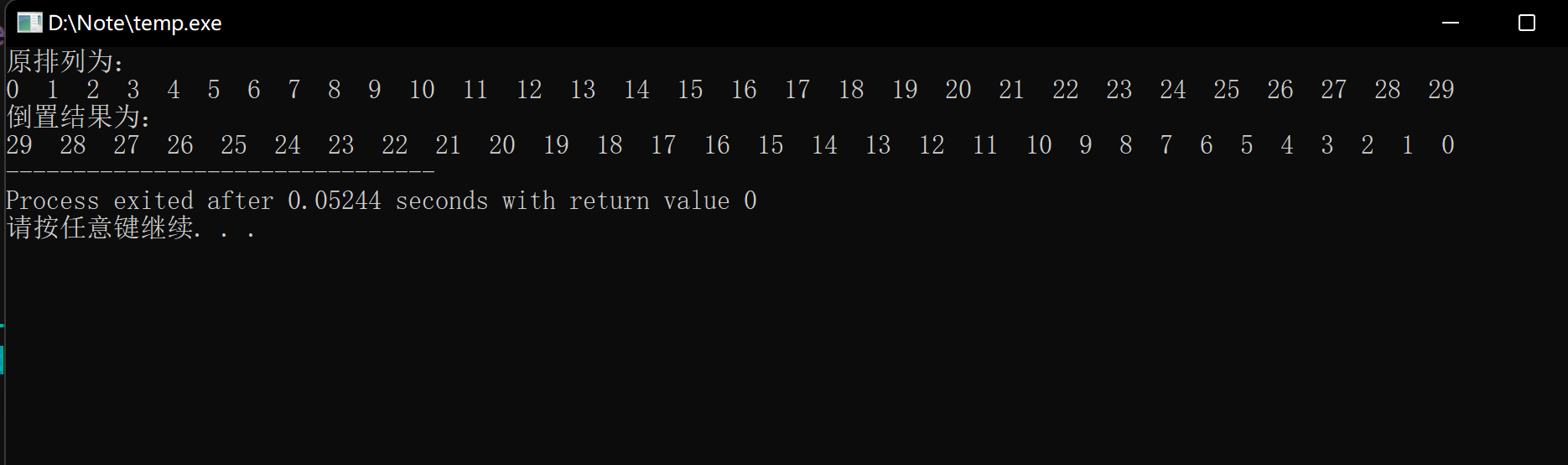
***/*有问题*/(1) 实现顺序表的基本操作：初始化顺序表、插入数据建顺序表、删除、按值查找***

#include<stdio.h>  
#include<stdlib.h>  
#include<stdbool.h>  
#define Volume 29  
typedef struct{  
 int \*data;  
 int length;  
 int capacity; //容量   
}SqList;  
  
void InitList(SqList \*L){  
 L->data = (int\*)malloc(Volume\*sizeof(int));  
 if(!L->data){  
 printf("Error.\n");  
 exit(1);  
 }  
 for(int i=0;i<Volume;i++)  
 L->data[i]=0;  
 L->length=0;  
 L->capacity=Volume;  
}  
  
void InsertList(SqList \*L,int index,int elem){  
 if(index<0 || index > L->length){  
 printf("超出范围\n");  
 return ;  
 }  
 if(L->length >= L->capacity){  
 printf("容量已满\n");  
 return ;  
 }  
 for(int i=L->length;i>index;i--)  
 L->data[i] = L->data[i-1];  
 L->data[index]=elem;  
 L->length++;  
}  
  
void DeleteList(SqList \*L,int index,int \*elem){  
 if(index<0 || index>=L->capacity){  
 printf("Error.\n");  
 exit(1);  
 }  
 //指针p指向被删除元素的位置   
 int \*p = &(L->data[index]);  
 //被删除的元素赋值给elem  
 \*elem = \*p;   
 //被删元素之后的元素左移   
 for(int i=index;i<L->length;i++){  
 L->data[i]=L->data[i+1];  
 }  
 L->length--;  
}  
  
int Find\_by\_Index(SqList \*L,int index){  
 if(index<0 || index>L->length){  
 printf("查无此数.\n");  
 exit(1);  
 }  
 int result = L->data[index];  
 return result;   
}   
  
void Find\_by\_Value(SqList \*L,int value){  
 \_Bool exchanged = true;  
 for(int i=0;i<L->length;i++){  
 exchanged = false;  
 if(value == L->data[i]){  
 printf("数值为%d的元素的索引为%d\n",value,i+1);  
 exchanged = true;  
 }  
 }  
 if(!exchanged)  
 printf("查无此数.\n");  
}  
  
int main(){  
 SqList L;   
 //初始化   
 InitList(&L);  
 printf("初始化结果：\n");  
 for(int i=0;i<L.capacity;i++){  
 L.data[i]=i;  
 L.length++;  
 printf("%d ",L.data[i]);  
 }  
 putchar('\n');  
 //插入元素  
 putchar('\n');  
 printf("插入元素结果：\n");  
 for(int i=3;i<17;i+=2)  
 L.data[i] = (-2)\*i+42;  
 for(int i=0;i<L.length;i++)  
 printf("%d ",L.data[i]);  
 putchar('\n');  
 //删除元素  
 putchar('\n');  
 int elem=0;  
 DeleteList(&L,7,&elem);  
 printf("被删除的元素是：%d\n",elem);  
 printf("删除元素结果：\n");  
 for(int i=0;i<L.length;i++)  
 printf("%d ",L.data[i]);  
 putchar('\n');  
 //按位查找   
 putchar('\n');  
 for(int i=3;i<12;i+=2){  
 printf("第%d位数值是：%d ",i,Find\_by\_Index(&L,i));  
 }  
 putchar('\n');  
 //按值查询  
 putchar('\n');  
 Find\_by\_Value(&L,7);  
 Find\_by\_Value(&L,12);  
 Find\_by\_Value(&L,17);  
 Find\_by\_Value(&L,29);  
 putchar('\n');  
   
 free(L.data);  
 return 0;  
}



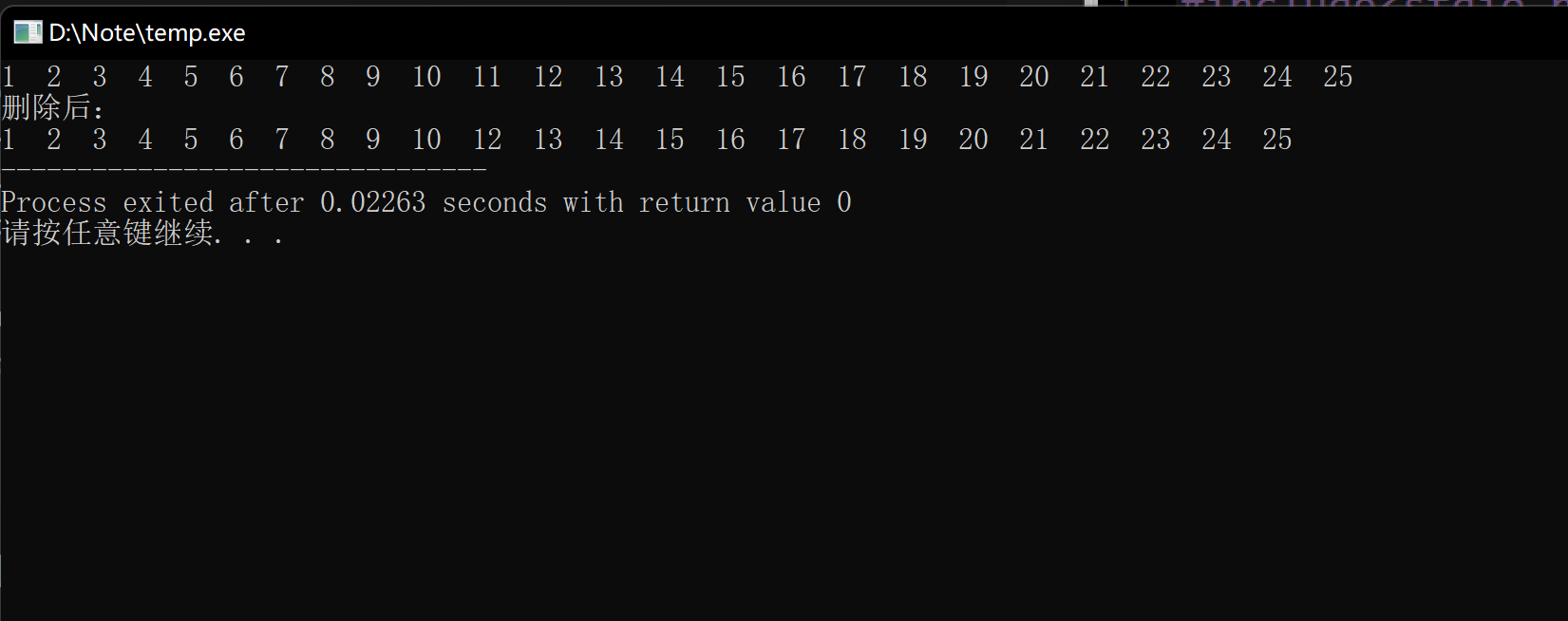
***(2) 设计一个高效的算法，将顺序表的所有元素逆置，要求算法的空间复杂度为O（1）。***

#include<stdio.h>  
#include<stdlib.h>  
  
typedef struct {  
 int \*data;  
 int length;  
} SqList;  
  
void InitList(SqList \*L, int length) {  
 L->data = (int\*)malloc(length \* sizeof(int));  
 if (!L->data) {  
 printf("Error.\n");  
 exit(1);  
 }  
 L->length = length;  
 for (int i = 0; i < L->length; i++)  
 L->data[i] = i;  
}  
  
void Reversal(SqList \*L, int length) {  
 int \*start = &L->data[0];  
 int \*end = &L->data[length - 1];  
 while (  
   
 start < end) {  
 // 倒置  
 int temp = \*start;  
 \*start = \*end;  
 \*end = temp;  
 // 移动指针  
 start++;  
 end--;  
 }  
}  
  
int main() {  
 SqList L;  
 InitList(&L, 30);  
 printf("原排列为：\n");  
 for (int i = 0; i < L.length; i++)  
 printf("%d ", L.data[i]);  
 putchar('\n');  
 Reversal(&L, 30);  
 printf("倒置结果为：\n");  
 for (int i = 0; i < L.length; i++)  
 printf("%d ", L.data[i]);  
 free(L.data);  
 return 0;  
}



***(3) 长度为n的顺序表L，编写一个时间复杂度为O（n）、空间复杂度为O（1）的算法，该算法删除线性表中所有值为x的数据元素。***

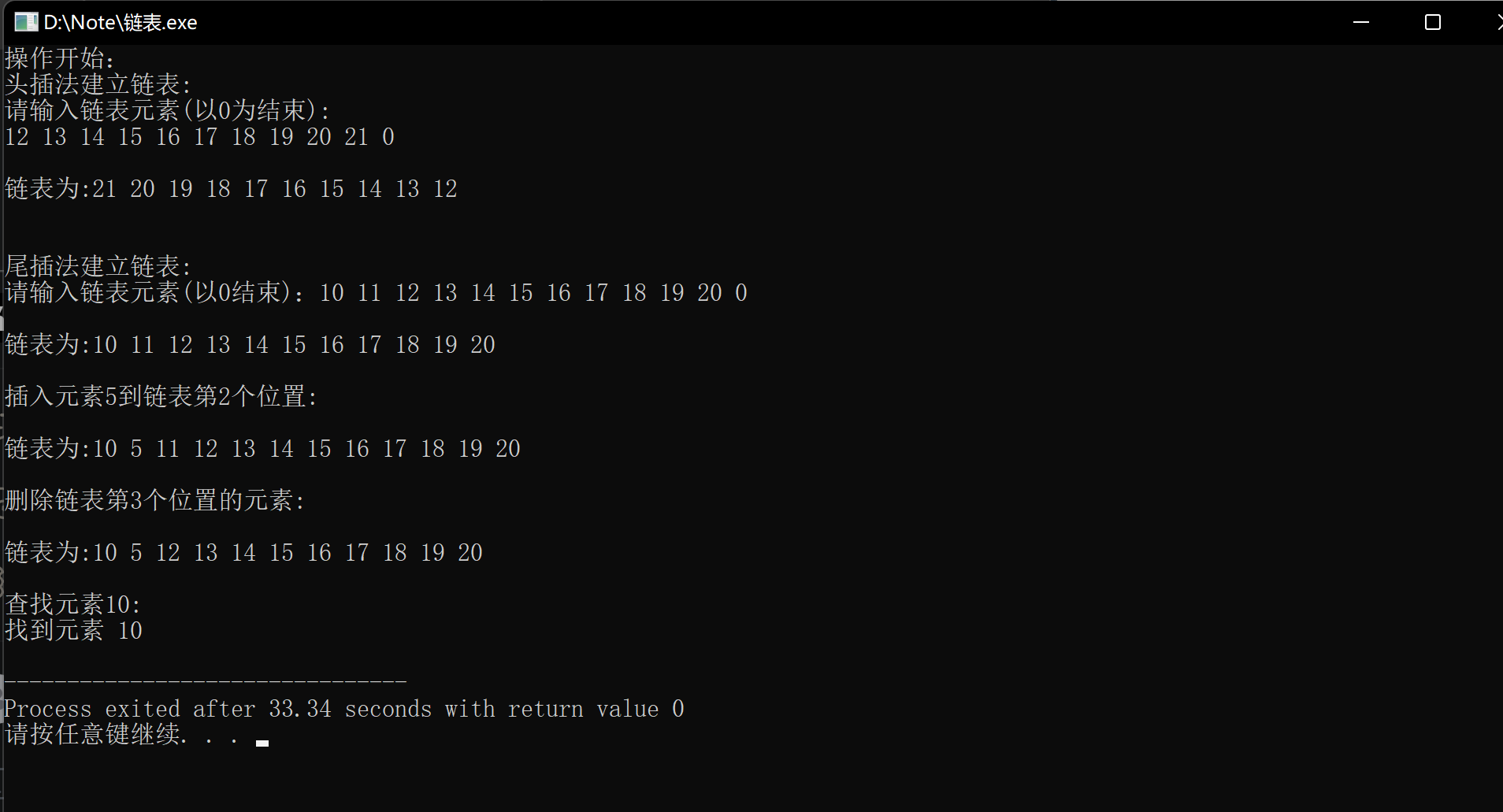
#include<stdio.h>  
#include<stdlib.h>  
#include<stdbool.h>  
typedef struct{  
 int \*data;  
 int length;  
}SqList;  
  
void InitList(SqList \*L,int n){  
 L->data=(int\*)malloc(n\*sizeof(int));  
 if(!L->data){  
 printf("Error.\n");  
 exit(1);  
 }  
 L->length=n;  
 for(int i=0;i<L->length;i++){  
 L->data[i]=i+1;  
 }  
}  
  
bool Remove(SqList \*L,int x){  
 int count=0; //用于数值等于x的元素个数计数  
 int i=0;  
 while(i<L->length){  
 if(L->data[i]==x){  
 count++;  
 }  
 else{  
 L->data[i-count]=L->data[i];   
 }  
 i++;  
 }   
 L->length=L->length-count;  
 if(count!=0)  
 return true;  
 else  
 return false;  
}   
  
int main(){  
 SqList L;  
 //初始化   
 InitList(&L,25);  
 for(int i=0;i<L.length;i++){  
 printf("%d ",L.data[i]);  
 }  
 //删除  
 printf("\n删除后：\n");   
 Remove(&L,11);  
 for(int i=0;i<L.length;i++){  
 printf("%d ",L.data[i]);  
 }  
 free(L.data);  
 return 0;  
}



#### 3、第三次作业：

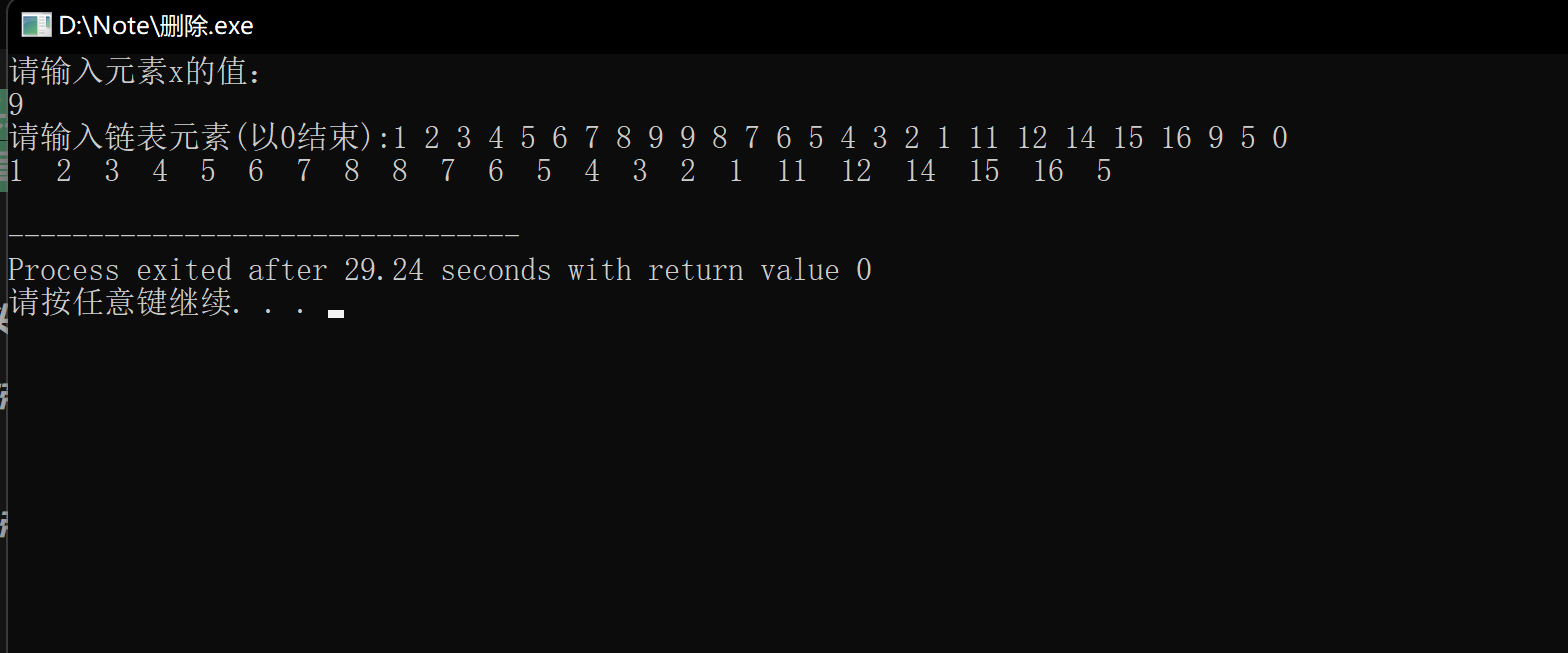
***(1) 实现带头节点链表的基本操作：初始化链表、头插法建链表、尾插法建链表、插入、删除、按值查找***

#include<stdio.h>  
#include<stdlib.h>  
#include<stdbool.h>  
  
typedef int ElemType;  
typedef struct LNode{  
 ElemType data;  
 struct LNode \*next;  
}LNode,\*LinkList;  
/\*LNode表示是个结点,LinkList表示链表\*/  
//初始化  
bool InitList(LinkList \*L){  
 \*L=(LinkList\*)malloc(sizeof(LNode));  
 if(\*L==NULL){  
 printf("Error.\n");  
 return false;  
 }  
 (\*L)->data=0;  
 (\*L)->next=NULL;  
 return true;  
}   
//头插法建链表   
LinkList HeadInsert(LinkList L){  
 if(!L){  
 return L;   
 }  
 LNode \*head=L; //头结点   
 LNode \*new;  
 int x;  
 printf("请输入链表元素(以0为结束):");  
 scanf("%d",&x);  
 while(x!=0){  
 new=(LinkList\*)malloc(sizeof(LNode));  
 if(!new){  
 return L;  
 }  
 new->data = x;  
 new->next = head->next;  
 head->next = new;  
 scanf("%d",&x);  
 }  
 return L;  
}   
//尾插法建链表   
LinkList TailInsert(LinkList L){  
 if(!L){  
 return L;  
 }  
 LNode \*tail=L; //用于追踪链表尾部   
 LNode \*new;  
 int x;  
 printf("请输入链表元素(以0结束)：");  
 scanf("%d",&x);  
 while(x!=0){  
 new=(LinkList\*)malloc(sizeof(LNode));  
 new->data=x;  
 tail->next=new;  
 tail=new;  
 scanf("%d",&x);  
 }  
 tail->next=NULL;  
 return L;  
}   
//获得长度  
int Length(LinkList L){  
 int len=0;  
 LNode \*p = L->next;  
 while(p){  
 p = p->next;  
 len++;  
 }  
 return len;  
}   
//(带头结点)插入  
bool Insert(LinkList L,int index,int elem){  
 if(index<0||index>Length(L)){  
 printf("超出范围.\n");  
 return ;  
 }  
 LNode \*s,\*p=L;  
 int j=0;  
 while(p && j<index-1){  
 p = p->next;  
 ++j;  
 }  
 s=(LinkList\*)malloc(sizeof(LNode));  
 s->data = elem;  
 s->next = p->next; //插入L中  
 p->next = s;  
  
 return true;  
}   
//(带头结点)删除  
bool Remove(LinkList L,int index){  
 if(index<1 || index>Length(L)){  
 printf("超出范围.\n");  
 return false;  
 }  
 LNode \*p=L;  
 int j=0;  
 while(p && j<index-1){  
 p = p->next;  
 ++j;  
 }  
 if(p->next==NULL){  
 return false;  
 }  
 LNode \*q = p->next;  
 p->next = q->next;  
 free(q);  
 return true;  
}   
//按值查找  
LNode \*FindElem(LinkList L,ElemType elem){  
 LNode \*p = L->next;  
 while(p && p->data!= elem){  
 p = p->next;  
 }  
 return p;  
}  
//遍历链表  
void Print(LinkList L){  
 LNode \*p=L->next;  
 if(p==NULL)   
 printf("\n这是空链表.\n");  
 else{  
 printf("\n链表为:");  
 while(p!=NULL){  
 printf("%d ",p->data);  
 p = p->next;  
 }  
 printf("\n");  
 }  
}   
  
int main(){  
 LinkList L;  
 //初始化   
 InitList(&L);  
 printf("操作开始：\n");  
 //头插法   
 printf("头插法建立链表:\n");  
 HeadInsert(L);  
 Print(L);  
 //尾插法  
 printf("\n\n尾插法建立链表:\n");  
 TailInsert(L);  
 Print(L);   
 //插入  
 printf("\n插入元素5到链表第2个位置:\n");  
 if(Insert(L, 2, 5)){  
 Print(L);  
 }  
 //删除  
 printf("\n删除链表第3个位置的元素:\n");  
 if(Remove(L, 3)){  
 Print(L);  
 }  
 //查找元素  
 printf("\n查找元素10:\n");  
 LNode \*found = FindElem(L, 10);  
 if(found != NULL){  
 printf("找到元素 %d\n", found->data);  
 } else {  
 printf("未找到元素\n");  
 }  
 //释放内存  
 LNode \*p = L,\*q;  
 while(p!=NULL){  
 q = p->next;  
 free(p);  
 p = q;  
 }   
 return 0;  
}



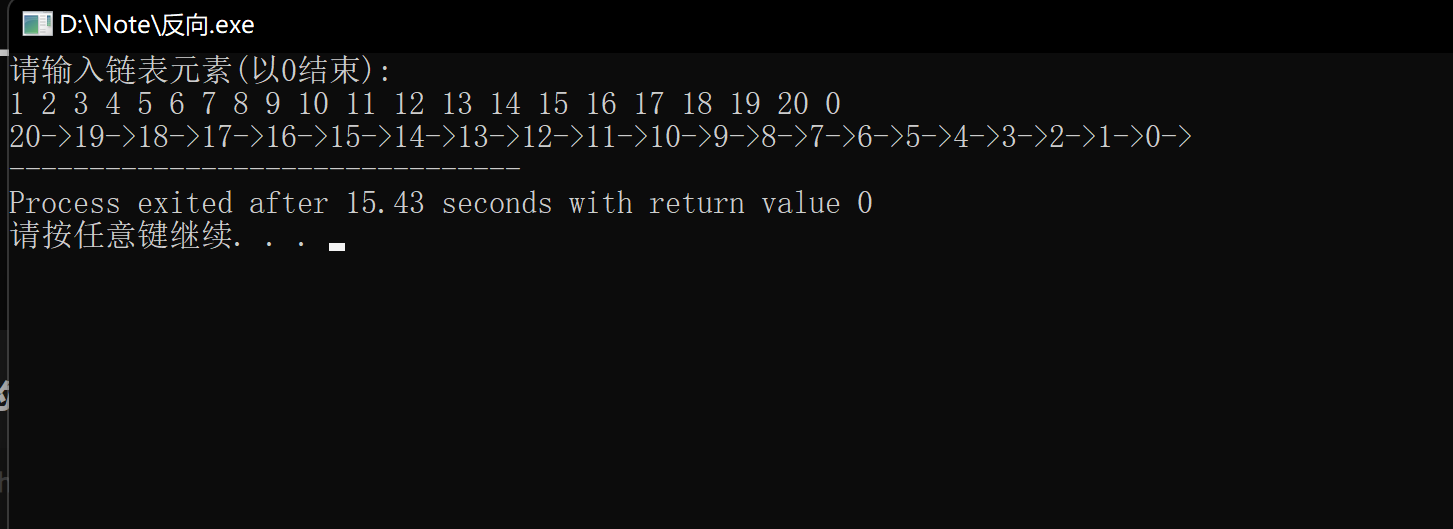
***(2) 在带头结点的单链表L中，删除所有值为x的节点，并释放其空间，假设值为x的结点不唯一，试编写算法以实现上述操作。***

#include<stdio.h>  
#include<stdbool.h>  
#include<stdlib.h>  
typedef int ElemType;  
typedef struct LNode{  
 ElemType data;  
 struct LNode \*next;  
}LNode,\*LinkList;  
//初始化  
bool InitList(LinkList \*L){  
 \*L=(LinkList\*)malloc(sizeof(LNode));  
 if(\*L==NULL){  
 printf("Error.\n");  
 return false;  
 }  
 (\*L)->data=0;  
 (\*L)->next=NULL;  
 return true;  
}   
//尾插法建链表  
LinkList TailInsert(LinkList L){  
 if(!L){  
 return L;  
 }  
 LNode \*tail=L; //用于追踪链表尾部   
 LNode \*new;  
 int x;  
 printf("请输入链表元素(以0结束):");  
 scanf("%d",&x);  
 while(x!=0){  
 new=(LinkList\*)malloc(sizeof(LNode));  
 new->data=x;  
 tail->next=new;  
 tail=new;  
 scanf("%d",&x);  
 }  
 tail->next=NULL;  
 return L;  
}   
//删除值为x的节点  
void Remove\_Elem(LinkList L,ElemType x){  
 LNode \*p=L->next;  
 while(p->next != NULL){  
 if(p->next->data == x){  
 LNode \*temp = p->next;  
 p->next = temp->next; //跳过要删除的节点  
 free(temp); //释放节点空间   
 }  
 else{  
 p = p->next;   
 }  
 }  
}  
//打印链表  
void PrintList(LinkList L){  
 LNode \*p = L->next;  
 while(p != NULL){  
 printf("%d ",p->data);  
 p = p->next;  
 }  
 printf("\n");  
}   
  
int main(){  
 LinkList L;  
 ElemType x;  
 printf("请输入元素x的值：\n");  
 scanf("%d",&x);  
   
 InitList(&L);  
 TailInsert(L);  
 Remove\_Elem(L,x);  
 PrintList(L);  
   
 // 释放链表占用的内存  
 LNode \*p = L, \*q;  
 while (p != NULL) {  
 q = p->next;  
 free(p);  
 p = q;  
 }  
 return 0;  
}



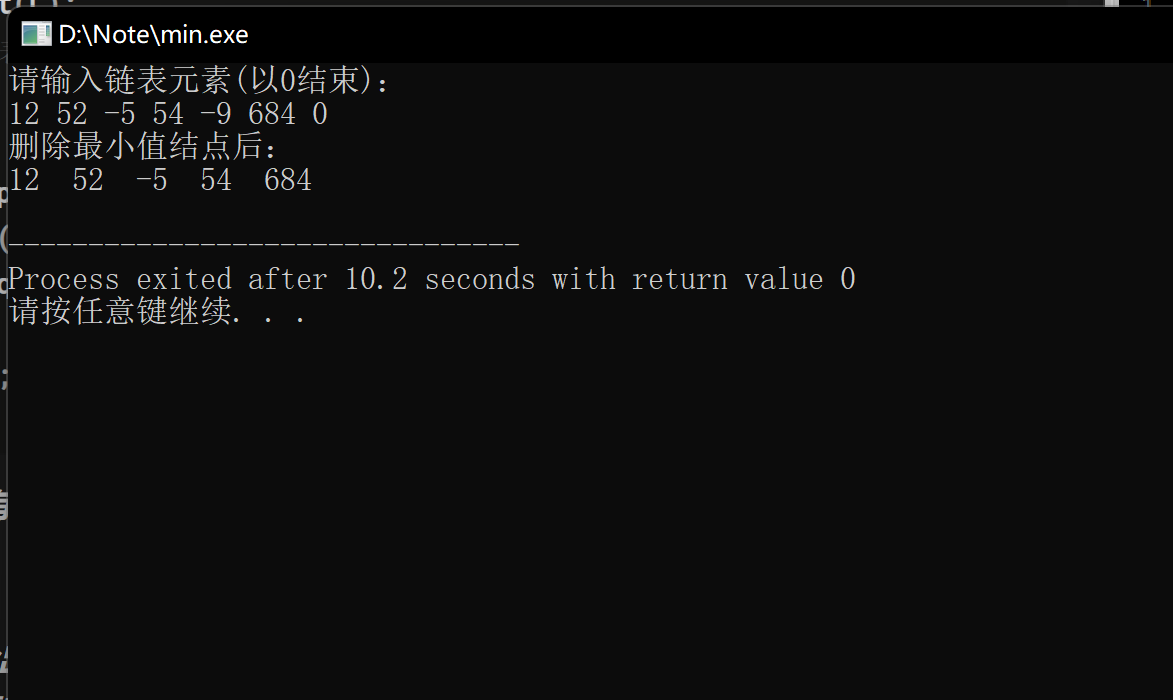
***(3) 试L为带头节点的单链表，编写算法实现从尾到头反向输出每个结点的值。***

#include<stdio.h>  
#include<stdlib.h>  
#include<stdbool.h>  
typedef int ElemType;  
typedef struct LNode{  
 ElemType data;  
 struct LNode \*next;  
}LNode,\*LinkList;  
//初始化  
bool InitList(LinkList \*L){  
 \*L=(LinkList\*)malloc(sizeof(LNode));  
 if(\*L==NULL){  
 printf("Error.\n");  
 return false;  
 }  
 (\*L)->data=NULL;  
 (\*L)->next=NULL;  
 return true;  
}  
//尾插法  
LinkList TailList(LinkList L){  
 if(!L){  
 return L;  
 }   
 LNode \*tail=L;  
 LNode \*new;  
 int x;  
 printf("请输入链表元素(以0结束):\n");  
 scanf("%d",&x);  
 while(x!=0){  
 new=(LNode\*)malloc(sizeof(LNode));  
 new->data=x;  
 tail->next=new;  
 tail=new;  
 scanf("%d",&x);  
 }   
 tail->next=NULL;  
 return L;  
}   
//反向输出  
void Reverse\_Print(LinkList L){  
 if(L->next != NULL){  
 Reverse\_Print(L->next);  
 }  
 printf("%d->",L->data);  
}   
  
int main(){  
 LinkList L;  
 InitList(&L);  
 TailList(L);  
 Reverse\_Print(L);  
 // 释放链表占用的内存  
 LNode \*p = L, \*q;  
 while (p != NULL) {  
 q = p->next;  
 free(p);  
 p = q;  
 }  
 return 0;  
}



***(4) 试编写在带头结点的单链表L中删除一个最小值结点的高效算法（假设最小值结点是唯一的）。***

#include<stdio.h>  
#include<stdlib.h>  
#include<stdbool.h>  
typedef int ElemType;  
typedef struct LNode{  
 ElemType data;  
 struct LNode \*next;  
}LNode,\*LinkList;  
//初始化  
bool InitList(LinkList \*L){  
 \*L=(LNode\*)malloc(sizeof(LNode));  
 if(\*L==NULL){  
 printf("Error.\n");  
 return false;  
 }  
 (\*L)->data=0;  
 (\*L)->next=NULL;  
 return true;  
}   
//尾插法  
LinkList TailList(LinkList L){  
 if(!L){  
 return NULL;  
 }  
 LNode \*tail=L;  
 LNode \*new;  
 int x;  
 printf("请输入链表元素(以0结束)：\n");  
 scanf("%d",&x);  
 while(x!=0){  
 new=(LNode\*)malloc(sizeof(LNode));  
 new->data=x;  
 tail->next=new;  
 tail=new;  
 scanf("%d",&x);  
 }  
 tail->next=NULL;  
 return L;  
}   
//删除最小  
void Delete\_Min(LinkList L){  
 if(L==NULL || L->next==NULL){  
 return;  
 }  
 LNode \*p=L->next; //第一个实际结点  
 LNode \*pre=L;//pre用于指向最小值结点的前一个  
 LNode \*min=p; //用于表示最小值的结点  
 while(p != NULL && p->next!=NULL){  
 if(p->next->data < min->data){  
 pre=p;  
 min=p->next;  
 }  
 p=p->next;  
 }  
 //最小值是第一个节点  
 if(min==L->next){  
 pre->next = min->next;  
 } //最小值在末尾或中间   
 else{  
 pre->next = min->next;  
 free(min);   
 }   
}   
//打印链表  
void PrintList(LinkList L){  
 LNode \*p = L->next;  
 while(p != NULL){  
 printf("%d ",p->data);  
 p = p->next;  
 }  
 printf("\n");  
}   
  
int main(){  
 LinkList L;  
 InitList(&L);  
 TailList(L);  
 Delete\_Min(L);  
 printf("删除最小值结点后：\n");  
 PrintList(L);  
 // 释放链表占用的内存  
 LNode \*p = L, \*q;  
 while (p != NULL) {  
 q = p->next;  
 free(p);  
 p = q;  
 }  
 return 0;  
}



***(5) 已知一个带有表头结点的单链表，结点结构为***

***假设该链表只给出了头指针list，在不改变链表的前提下，请设计一个尽可能高效的算法，查找链表中倒数第k个位置上的结点（k为正整数）。若查找成功，算法输出该结点的data域的值，并返回1；否则，只返回0。要求：***

***1）描述算法的基本 设计思想。***

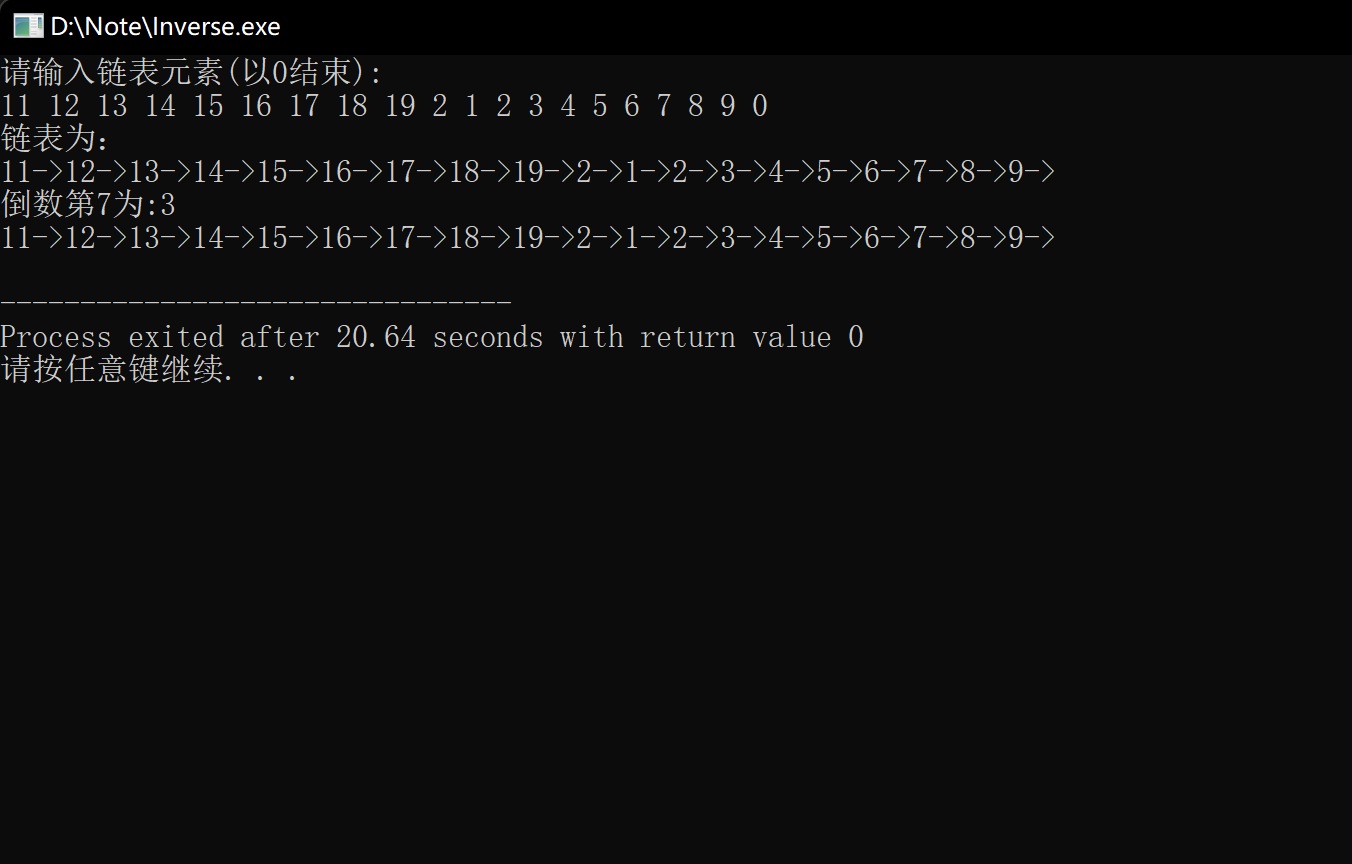
创建两个指针"fast"和"slow"，两个指针都先指向头结点。开始操作：让fast移动k步，这样两个指针间就相差了k步。两个指针接着同步移动，直至 fast->next==NULL。

***2）描述算法的详细实现步骤***

创建两个指针"fast"和"slow"，两个指针都先指向头结点。开始操作：让fast移动k步，如果链表长度小于k，那么直接返回0，因为不存在倒数第k个结点。这样两个指针间就相差了k步。两个指针接着同步移动，直至 fast->next==NULL,slow指针将指向倒数第k个结点。  
输出结果：如果fast指针能够到达链表末尾，那么slow指针指向的就是倒数第k个结点，输出该结点的data域的值，并返回1；如果fast指针在到达链表末尾之前就已经没有下一个结点了，说明链表长度小于k，返回0。

***3）根据设计思想和实现步骤，采用程序设计语言描述算法，关键之处请给出简要注释。***

#include<stdio.h>  
#include<stdlib.h>  
#include<stdbool.h>  
typedef int ElemType;  
typedef struct LNode{  
 ElemType data;  
 struct LNode \*next;  
}LNode,\*LinkList;  
//初始化  
bool InitList(LinkList \*L){  
 \*L=(LNode\*)malloc(sizeof(LNode));  
 if(\*L==NULL){  
 printf("Error.\n");  
 return false;  
 }  
 (\*L)->data=0;  
 (\*L)->next=NULL;  
 return true;  
}   
//尾插法  
LinkList TailList(LinkList L){  
 if(!L){  
 return NULL;  
 }  
 LNode \*tail=L;  
 LNode \*new;  
 int x;  
 printf("请输入链表元素(以0结束):\n");  
 scanf("%d",&x);  
 while(x!=0){  
 new=(LNode\*)malloc(sizeof(LNode));  
 new->data=x;  
 tail->next=new;  
 tail=new;  
 scanf("%d",&x);   
 }  
 tail->next=NULL;  
 return L;   
}   
//倒数第k个  
int Inverse\_Count(LinkList L,int k){  
 if(L==NULL || k<=0){  
 return 0;  
 }  
 LNode \*fast=L;  
 LNode \*slow=L;  
 //让 fast指针先走k步  
 for(int i=0;i<k;i++){  
 if(fast==NULL)  
 return 0;  
 fast = fast->next;  
 }   
 //双指针同步移动  
 while(fast != NULL){  
 fast = fast->next;  
 slow = slow->next;  
 }   
 //如果fast指针到达末尾，slow指向倒数第k个  
 printf("倒数第%d为:%d\n",k,slow->data);  
 return 1;  
}  
//打印链表  
void PrintList(LinkList L){  
 LNode \*p = L->next;  
 while(p != NULL){  
 printf("%d->",p->data);  
 p = p->next;  
 }  
 printf("\n");  
}   
int main(){  
 LinkList L;  
 InitList(&L);  
 TailList(L);  
 printf("链表为：\n");   
 PrintList(L);  
 Inverse\_Count(L,7);  
 PrintList(L);  
 // 释放链表占用的内存  
 LNode \*p = L, \*q;  
 while (p != NULL) {  
 q = p->next;  
 free(p);  
 p = q;  
 }  
 return 0;  
}



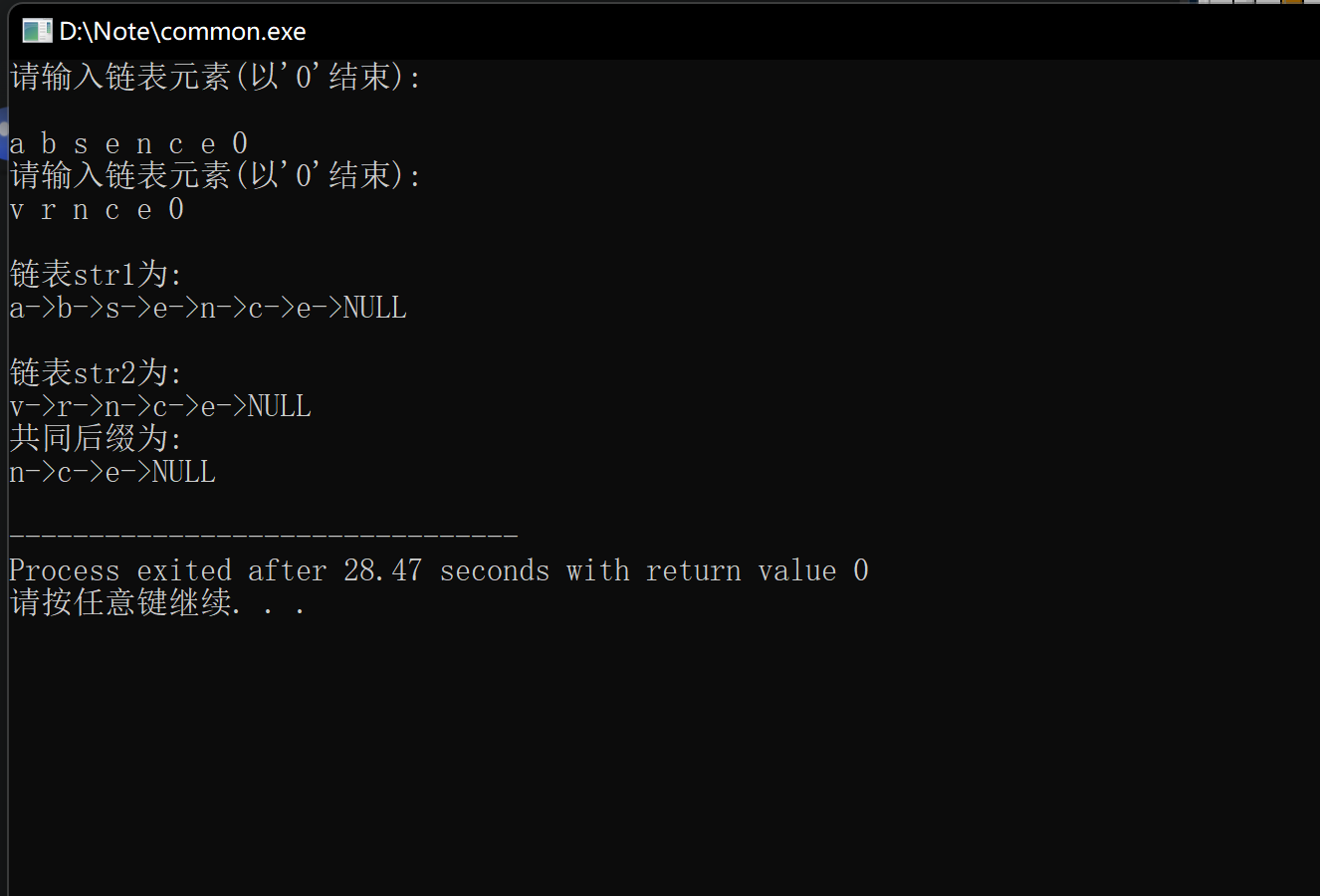
***(6) 假定采用带头结点的单链表保存单词，当两个单词有相同的后缀时，则可共享相同的后缀存储空间，设str1和str2分别指向两个单词所在单链表的头结点，链表结点结构为，请设计一个时间上尽可能高效的算法，找出由str1和str2所指向两个链表共同后缀的起始位置（如图中字符i所在结点的位置p）。要求：***

***1）描述算法的基本设计思想。***

构造一个函数Length()计算链表的长度，计算出两个链表的长度，并计算它们的差，然后让节点指针指向长度较长的链表并使该指针后移，直到与另一个较短的链表等长，最后让指针同时后移。当两个指针指向同一个地址使，该地址为所寻找的共同后缀的起始位置。

***2）根据设计思想，采用程序设计语言描述算法，关键之处请给出简要注释。***

include<stdio.h>  
#include<stdlib.h>  
#include<stdbool.h>  
#include<math.h>  
typedef char ElemType;  
typedef struct LNode{  
 ElemType data;  
 struct LNode \*next;  
}LNode,\*LinkList;  
//初始化  
bool InitList(LinkList \*L){  
 \*L=(LNode\*)malloc(sizeof(LNode));  
 if(\*L==NULL){  
 printf("Error.\n");  
 return false;  
 }  
 (\*L)->data='\0';  
 (\*L)->next=NULL;  
 return true;  
}   
//尾插法  
LinkList TailList(LinkList L){  
 if(!L){  
 return NULL;  
 }  
 LNode \*tail=L;  
 LNode \*new;  
 int x;  
 printf("请输入链表元素(以'0'结束):\n");  
 while(scanf(" %c",&x), x!='0' && x!='\n'){  
 new=(LNode\*)malloc(sizeof(LNode));  
 new->data=x;  
 tail->next=new;  
 tail=new;  
 }  
 tail->next=NULL;  
 return L;   
}   
//获得长度  
int Length(LinkList L){  
 int len=0;  
 LNode \*p = L;  
 while(p){  
 p = p->next;  
 len++;  
 }  
 return len;  
}   
//共同后缀  
LNode\* FindCommon(LinkList str1,LinkList str2){  
 LNode \*p=str1;  
 LNode \*q=str2;  
 int length1 = Length(str1);  
 int length2 = Length(str2);  
 int distance = abs(length1-length2);  
 //若p指向的链表长度大于q的长度,使p后移  
 if(length1>length2){  
 for(int i=0;i<distance;i++){  
 p=p->next;  
 }  
 }else{ //反之亦然   
 for(int i=0;i<distance;i++){  
 q=q->next;  
 }  
 }  
 while(p && q && p->next->data != q->next->data){  
 //两个指针同时后移  
 p=p->next;  
 q=q->next;   
 }  
 return p;  
}  
//打印链表  
void PrintList(LinkList L){  
 LNode \*p = L->next;  
 while(p != NULL){  
 printf("%c->",p->data);  
 p = p->next;  
 }  
 printf("NULL\n");  
}   
// 释放链表内存  
void FreeList(LinkList L) {  
 LNode \*p = L->next;  
 LNode \*q;  
 while (p != NULL) {  
 q = p->next;  
 free(p);  
 p = q;  
 }  
 free(L);  
}  
  
int main(){  
 LinkList str1,str2;  
 //初始两个链表   
 InitList(&str1);  
 InitList(&str2);  
 //尾插法  
 TailList(str1);  
 TailList(str2);   
 printf("\n链表str1为:\n");  
 PrintList(str1);  
 printf("\n链表str2为:\n");  
 PrintList(str2);  
 //共同后缀  
 LNode\* commonSuffix = FindCommon(str1, str2);  
 if(commonSuffix){  
 printf("共同后缀为:\n");  
 PrintList(commonSuffix);  
 } else {  
 printf("没有共同后缀。\n");  
 }  
 //释放内存  
 FreeList(str1);  
 FreeList(str2);  
   
 return 0;  
}



还有另外一种方法：

/\*\*  
 \* Definition for singly-linked list.  
 \* struct ListNode {  
 \* int val;  
 \* struct ListNode \*next;  
 \* };  
 \*/  
struct ListNode \*getIntersectionNode(struct ListNode \*headA, struct ListNode \*headB) {  
 if(headA == NULL || headB == NULL){  
 return NULL;  
 }  
 struct ListNode \*p = headA, \*q = headB;  
 while( p != q ){  
 p = p == NULL ? headB : p->next;  
 q = q == NULL ? headA : q->next;  
 }  
 return p;  
}

***3）说明你所设计算法的时间复杂度***

1、计算两个链表的长度，Length(str1)和Length(str2)的时间复杂度为 O(n) 。  
2、移动指针使两个链表的起始位置对齐，遍历的时间复杂度为 O(n) 。  
3、对齐后遍历比较，直到找到共同后缀，这部分的时间复杂度为 O(n) 。  
总之，函数的时间复杂度为 O(n) 。

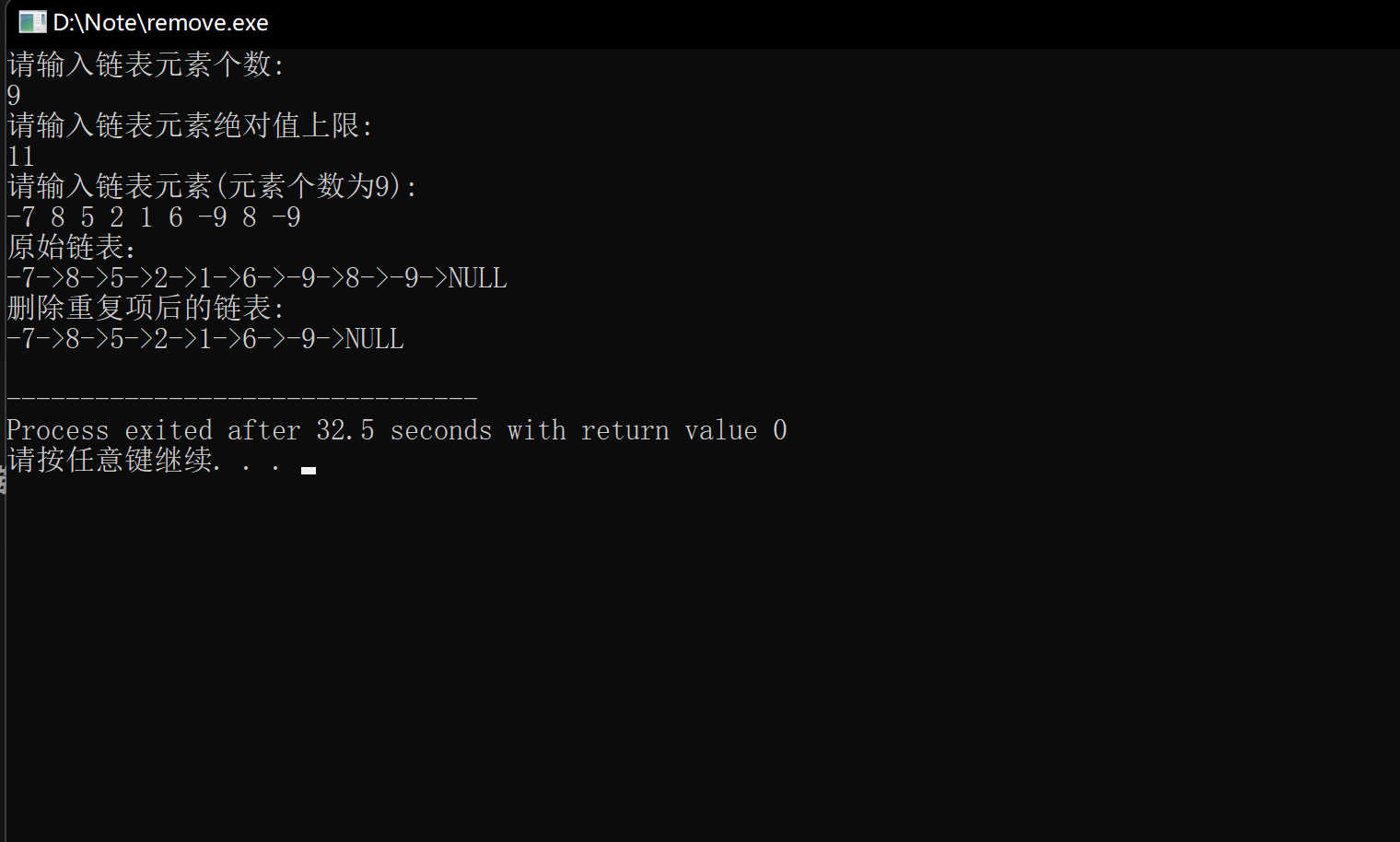
***(7) 用单链表保存m个整数，结点的结构为：data,且|data|≤n(n为正整数）。现要求设计一个时间复杂度尽可能高效的算法，对于链表中data的绝对值相等的结点，仅保留第一次出现的结点而删除其余绝对值相等的结点。***

***1）描述算法的基本 设计思想。***

1、创建一个大小为 m+1 的数组，初始化所有元素为 false。  
2、遍历链表：从头结点开始遍历链表，对于每个结点：  
 ●计算结点中 data 的绝对值。  
 ●检查出现过数组中对应索引的值：  
 如果结点中data的绝对值  
 检查出现过数组中对应索引的值：  
 如果为false，说明这是第一次遇到这个绝对值，将出现过的数组中对应索引的值设置为true，并保留当前结点。  
 如果为true，说明之前已经遇到过这个绝对值，因此需要删除当前结点。  
3、删除结点  
4、处理边界情况：  
 ●如果链表为空，直接返回。  
 ●在删除结点时，如果当前结点是头结点，需要更新头结点。  
 ●在删除结点时，如果当前结点是最后一个结点，需要确保前一个结点的指针被更新。  
5、返回结果

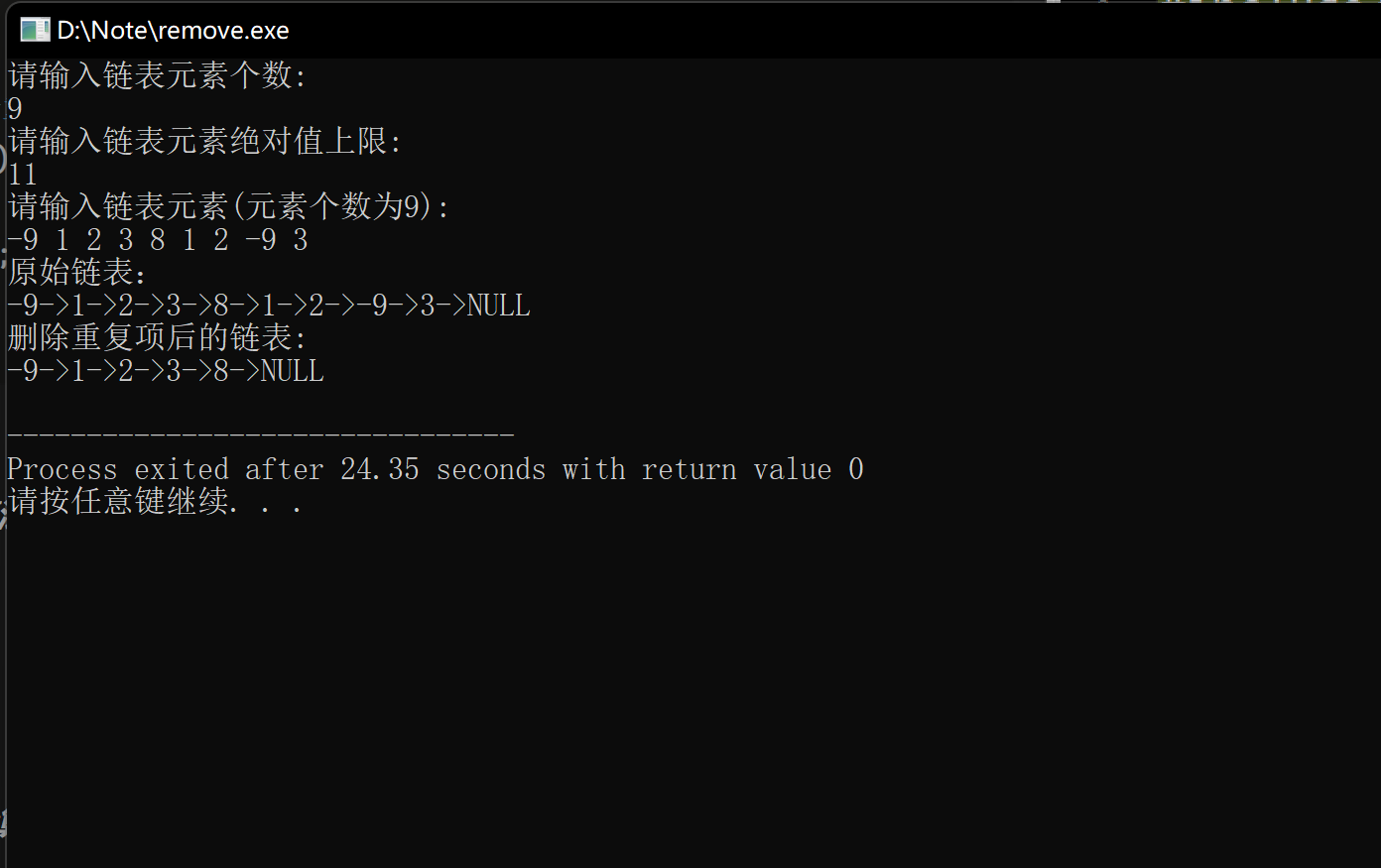
***2）使用C或者C++语言，给出单链表结点的数据类型定义。***

#include<stdio.h>  
#include<stdlib.h>  
#include<stdbool.h>  
#include<math.h>  
  
typedef int ElemType;  
typedef struct LNode{  
 ElemType data;  
 struct LNode \*next;  
}LNode,\*LinkList;  
//初始化  
bool InitList(LinkList \*L){  
 \*L=(LNode\*)malloc(sizeof(LNode));  
 if(\*L==NULL){  
 printf("Error.\n");  
 return false;  
 }  
 (\*L)->data=0;  
 (\*L)->next=NULL;  
 return true;  
}   
//尾插法  
LinkList TailList(LinkList L,int m){  
 if(m<=0){  
 return L;  
 }  
 LNode \*tail=L;  
 LNode \*new;  
 int x;  
 printf("请输入链表元素(元素个数为%d):\n",m);  
 for(int i=0;i<m;i++){  
 scanf("%d",&x);  
 new=(LNode\*)malloc(sizeof(LNode));  
 new->data=x;  
 tail->next=new;  
 tail=new;  
 }  
 tail->next=NULL;  
 return L;  
}  
  
//删除重复项   
void Remove(LinkList L,int n){  
 if(L==NULL || L->next==NULL){  
 return;  
 }  
 //追踪每个绝对值是否出现过,|data|<=n   
 bool \*Lyst=(bool\*)malloc((n+1)\*sizeof(bool));  
 for(int i=0;i<=n;i++){  
 Lyst[i]=false;  
 }  
 //current 指针用于遍历链表   
 LNode \*current = L->next;  
 //nextNode 用于临时存储current的下一个节点   
 LNode \*nextNode;  
 //prev 用于存储current的前一个节点   
 LNode \*prev = L;  
  
 while(current != NULL){  
 nextNode=current->next;  
 int Value = abs(current->data);  
 //检查当前节点的值是否已经出现  
 if(!Lyst[Value]){   
 //标记为 已出现   
 Lyst[Value]=true;  
 prev = current;  
 current = nextNode;  
 }else{  
 /\*如果已经出现过，删除节点\*/  
 //情况1：prev是头结点   
 if(prev->next==current){  
 prev->next=nextNode;  
 free(current);  
 current=nextNode;  
 } //情况2：prev不是头结点   
 else{  
 prev->next = nextNode;  
 free(current);  
 current = prev->next;  
 }  
 }  
 }  
}  
  
//打印链表  
void PrintList(LinkList L){  
 LNode \*p = L->next;  
 while(p != NULL){  
 printf("%d->",p->data);  
 p = p->next;  
 }  
 printf("NULL\n");  
}   
// 释放链表内存  
void FreeList(LinkList L) {  
 LNode \*p = L->next;  
 LNode \*q;  
 while (p != NULL) {  
 q = p->next;  
 free(p);  
 p = q;  
 }  
 free(L);  
}  
int main(){  
 LinkList L;  
 //初始化   
 InitList(&L);  
 //尾插法   
 int m,n;  
 printf("请输入链表元素个数:\n");  
 scanf("%d",&m);  
 printf("请输入链表元素绝对值上限:\n");  
 scanf("%d",&n);  
 TailList(L,m);  
 //链表输出  
 printf("原始链表：\n");   
 PrintList(L);  
 //删除重复项  
 Remove(L,n) ;  
 //输出   
 printf("删除重复项后的链表:\n");  
 PrintList(L);   
 //释放内存   
 FreeList(L);  
 return 0;  
}



***3）根据设计思想和实现步骤，采用C或者C++语言描述算法，关键之处请给出简要注释。***

#include<stdio.h>  
#include<stdlib.h>  
#include<stdbool.h>  
#include<math.h>  
  
typedef int ElemType;  
typedef struct LNode{  
 ElemType data;  
 struct LNode \*next;  
}LNode,\*LinkList;  
//初始化  
bool InitList(LinkList \*L){  
 \*L=(LNode\*)malloc(sizeof(LNode));  
 if(\*L==NULL){  
 printf("Error.\n");  
 return false;  
 }  
 (\*L)->data=0;  
 (\*L)->next=NULL;  
 return true;  
}   
//尾插法  
LinkList TailList(LinkList L,int m){  
 if(m<=0){  
 return L;  
 }  
 LNode \*tail=L;  
 LNode \*new;  
 int x;  
 printf("请输入链表元素(元素个数为%d):\n",m);  
 for(int i=0;i<m;i++){  
 scanf("%d",&x);  
 new=(LNode\*)malloc(sizeof(LNode));  
 new->data=x;  
 tail->next=new;  
 tail=new;  
 }  
 tail->next=NULL;  
 return L;  
}  
  
//删除重复项   
void Remove(LinkList L,int n){  
 if(L==NULL || L->next==NULL){  
 return;  
 }  
 //追踪每个绝对值是否出现过,|data|<=n   
 bool \*Lyst=(bool\*)malloc((n+1)\*sizeof(bool));  
 for(int i=0;i<=n;i++){  
 Lyst[i]=false;  
 }  
 //current 指针用于遍历链表   
 LNode \*current = L->next;  
 //prev 用于存储current的前一个节点   
 LNode \*prev = L;  
  
 while(current != NULL){  
 //nextNode=current->next;  
 int Value = abs(current->data);  
 //检查当前节点的值是否已经出现  
 if(!Lyst[Value]){   
 //标记为 已出现   
 Lyst[Value]=true;  
 prev = current;  
 current = current->next;  
 }else{  
 prev->next = current->next;  
 free(current);  
 current = prev->next;  
 }  
 }  
 }  
}  
  
//打印链表  
void PrintList(LinkList L){  
 LNode \*p = L->next;  
 while(p != NULL){  
 printf("%d->",p->data);  
 p = p->next;  
 }  
 printf("NULL\n");  
}   
// 释放链表内存  
void FreeList(LinkList L) {  
 LNode \*p = L->next;  
 LNode \*q;  
 while (p != NULL) {  
 q = p->next;  
 free(p);  
 p = q;  
 }  
 free(L);  
}  
int main(){  
 LinkList L;  
 //初始化   
 InitList(&L);  
 //尾插法   
 int m,n;  
 printf("请输入链表元素个数:\n");  
 scanf("%d",&m);  
 printf("请输入链表元素绝对值上限:\n");  
 scanf("%d",&n);  
 TailList(L,m);  
 //链表输出  
 printf("原始链表：\n");   
 PrintList(L);  
 //删除重复项  
 Remove(L,n) ;  
 //输出   
 printf("删除重复项后的链表:\n");  
 PrintList(L);   
 //释放内存   
 FreeList(L);  
 return 0;  
}



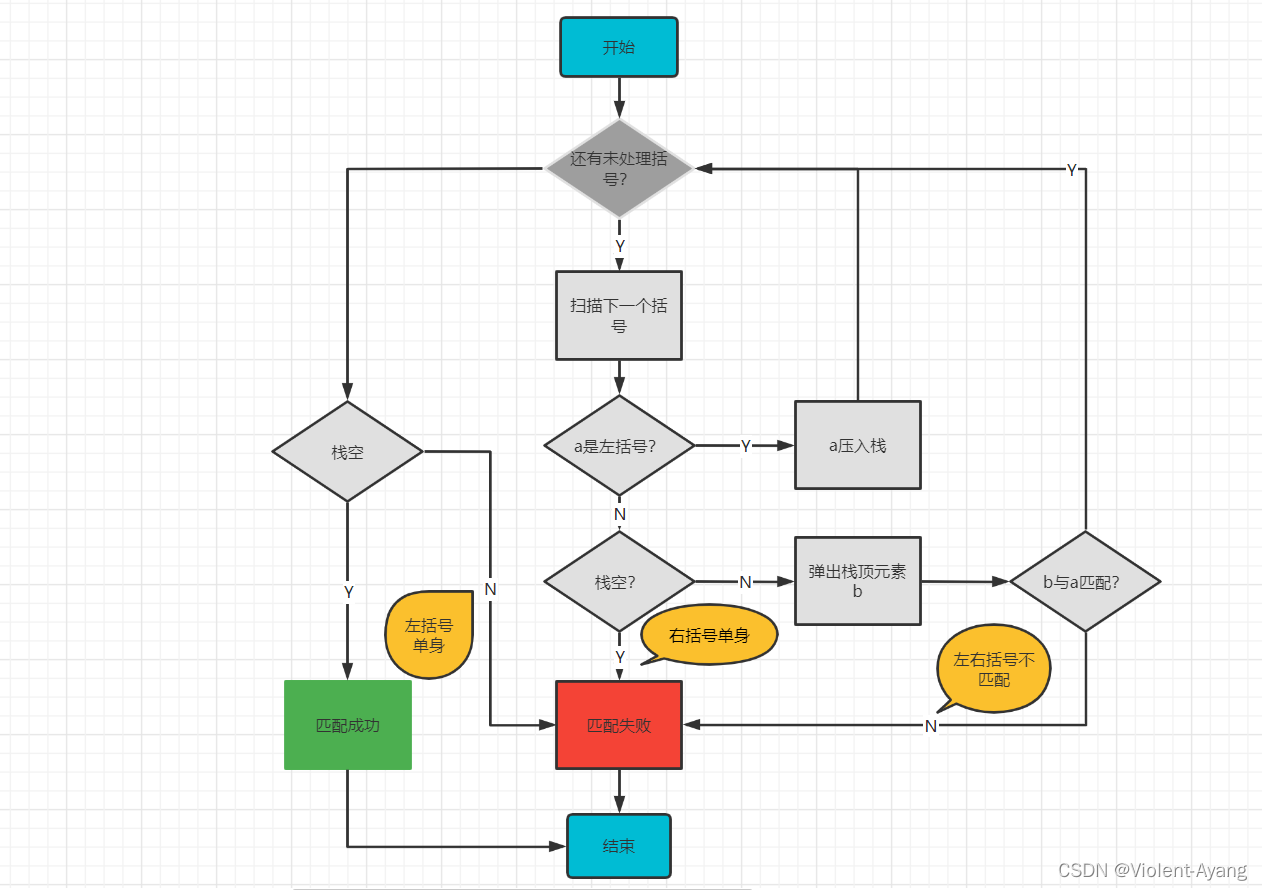
##### ***4）说明你所设计算法的时间复杂度和空间复杂度。***

#时间复杂度：  
1、初始化数组Lyst需要遍历索引，需要 O(n)。  
2、遍历链表：使用while循环遍历链表,时间复杂度也是 O(n)。  
3、检查true的时间复杂度为 O(1)。  
总之，时间复杂度为 O(n) 。  
#空间复杂度：  
1、布尔数组：需要空间 O(n)的空间  
2、几个指针变量的空间为 O(1)。  
总之，空间复杂度为 O(n) 。

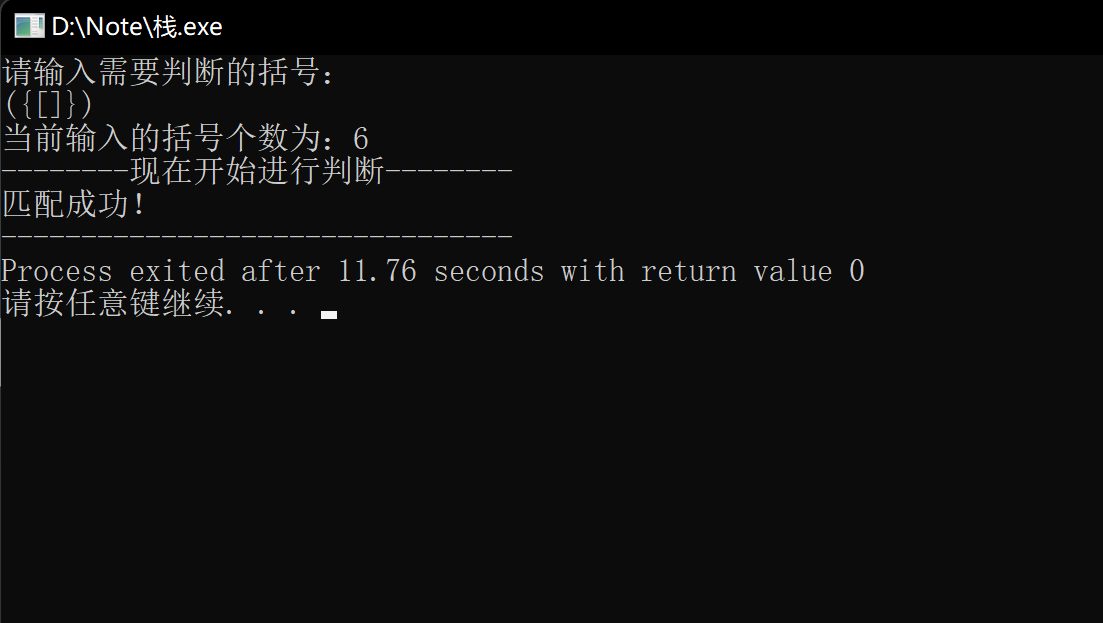
***第四次作业：***

***（1）括号匹配***

遇到左括号就入栈  
遇到有括号，就抵消一个左括号  
不匹配的情况  
遇到一个右括号，栈内弹出的左括号与之不匹配，例如 此时的右括号是 ] 而栈内的左括号是 {  
匹配到最后一个括号。栈内已经空了，说明此时多出来了括号  
处理完所有括号，栈内非空



#include<stdio.h>  
#include<stdlib.h>  
#include<stdbool.h>  
#include<string.h>   
  
#define INIT\_SIZE 10   
#define INCREASE 10  
#define MaxSize 100 //定义最大长度   
  
typedef char ElemType;  
typedef struct{  
 ElemType \*base; //栈底  
 ElemType \*top; //栈顶  
 int stacksize; //已分配存储空间   
}SqStack;  
//初始化  
bool InitStack(SqStack \*S){  
 S->base=(ElemType\*)malloc(INIT\_SIZE  
 \*sizeof(ElemType));  
 if(!S->base) //分配失败   
 return false;  
 S->top = S->base;  
 S->stacksize = INIT\_SIZE;  
 return true;  
}   
//进栈  
bool Push(SqStack \*S,ElemType e){  
 if(S->top-S->base>=S->stacksize){  
 //栈满  
 ElemType \*newBase=(ElemType\*)realloc(S->base,  
 (S->stacksize+INCREASE)\*sizeof(ElemType));  
 if (!newBase)  
 return false;  
 S->base = newBase;  
 S->top = S->base + S->stacksize;  
 S->stacksize += INCREASE;  
 }  
 \*S->top++ = e; /\*首先，\*S.top解引用top指针，获取  
 当前指向的位置。然后，将元素e的值赋给这个位置。  
 接着，S.top++将top指针向前移动一个位置，指向下一  
 个空闲的位置。\*/  
 return true;  
}   
//出栈  
bool Pop(SqStack \*S,ElemType \*e){  
 if(S->top == S->base)  
 return false;  
 \*e = \*--S->top; /\*首先，--S.top将top指针向前移动一个  
 位置，指向栈顶元素。然后，\*操作符用于解引用指针，  
 获取指向的元素值，并将其赋值给引用参数e。  
 这样，e就存储了从栈中弹出的元素。\*/  
 return true;  
}   
//取栈顶元素  
bool GetTop(SqStack \*S,ElemType \*e){  
 if(S->top == S->base)  
 return false;  
 \*e = \*(S->top - 1); /\*S.top-1将top指针向前移动一个  
 位置，指向栈顶元素。然后，\*操作符用于解引用  
 指针，获取指向的元素值，并将其赋值给引用参数e\*/  
 return true;  
}   
//判断为空  
bool StackEmpty(SqStack \*S){  
 return S->top == S->base;  
}   
// 括号匹配  
bool BracketCheck(SqStack \*S, char str[], int length) {  
 for (int i = 0; i < length; i++) {  
 if (str[i] == '(' || str[i] == '[' || str[i] == '{') {  
 Push(S, str[i]);  
 } else {  
 if (StackEmpty(S))  
 return false;  
 ElemType topElem;  
 if (!Pop(S, &topElem))  
 return false;  
 if ((str[i] == ')' && topElem != '(') ||  
 (str[i] == ']' && topElem != '[') ||  
 (str[i] == '}' && topElem != '{'))  
 return false;  
 }  
 }  
 return StackEmpty(S);  
}  
  
int main() {  
 SqStack S;  
 if (!InitStack(&S)) {  
 printf("栈初始化失败！\n");  
 return 1;  
 }  
 char s[MaxSize];  
 printf("请输入需要判断的括号：\n");  
 scanf("%99s", s); // 限制输入长度，防止溢出  
 int len = strlen(s);  
 printf("当前输入的括号个数为：%d\n", len);  
 printf("--------现在开始进行判断--------\n");  
 if (BracketCheck(&S, s, len)) {  
 printf("匹配成功！\n");  
 } else {  
 printf("匹配失败！\n");  
 }  
 // 释放栈内存  
 free(S.base);  
 return 0;  
}



***（2）算术表达式计算（两种方法都要实现：方法一先输出后缀表达式然后计算结果；方法二直接出结果***

#include<stdio.h>  
#include<stdlib.h>  
#include<ctype.h>  
#include<string.h>  
  
#define MAXSIZE 100  
  
//栈结构定义  
typedef struct{  
 char data[MAXSIZE];  
 int top;  
}Stack;  
  
//初始化栈  
void initStack(Stack \*S){  
 S->top = -1;  
}   
  
//判空  
int isEmpty(Stack \*S){  
 return S->top == -1;  
}   
  
//判断是否已满  
int isFull(Stack \*S){  
 return S->top == MAXSIZE - 1;  
}   
  
//入栈  
void Push(Stack \*S,char elem){  
 if(!isFull(S)){  
 S->data[++(S->top)] = elem;  
 }  
}   
  
//出栈  
int Pop(Stack \*S){  
 if(!isEmpty(S)){  
 return S->data[(S->top)--];  
 }  
 return -1;  
}  
  
//获取栈顶  
int Peek(Stack \*S){  
 if(!isEmpty(S)){  
 return S->data[S->top];  
 }  
 return -1;  
}   
  
//操作符优先级  
int Priority(char op){  
 switch(op){  
 case '+': case '-': return 1;//优先级是1  
 case '\*': case '/': return 2;//优先级是2  
 default: return 0;  
 }  
}   
  
//中缀转后缀  
void transform(char \*infix,char \*postfix){  
 // infix存储中缀表达式 , postfix存储后缀表达式   
 int i,k=0;   
 Stack S; //临时存储操作符和括号   
 initStack(&S);  
 for(i=0; infix[i]!='\0'; i++){  
 if(isdigit(infix[i])){  
 char num[11]; //存储数字  
 int j = 0;  
 while(isdigit(infix[i])){  
 num[j++] = infix[i++];  
 }   
 num[j] = '\0';  
 strcpy(postfix+k,num);  
 k += j;  
 postfix[k++] = ' '; //用空格分隔数字  
 i--; //因为for循环会再次增加   
 }else if(infix[i]=='('){  
 Push(&S,infix[i]);  
 //遇到左括号，压入栈   
 }else if(infix[i]==')'){  
 //遇到右括号,   
 while(!isEmpty(&S) && Peek(&S)!='('){  
 postfix[k++] = Pop(&S);  
 postfix[k++] = ' '; //空格隔开   
 }  
 if(!isEmpty(&S)){  
 Pop(&S); //删除'('   
 }   
 }  
 else{  
 /\*如果当前字符是一个操作符，则比较  
 其优先级与栈顶元素的优先级\*/   
 while( !isEmpty(&S) &&   
 Priority(Peek(&S)) >= Priority(infix[i]) ){  
 postfix[k++] = Pop(&S);   
 postfix[k++] = ' ';  
 /\*栈顶元素的优先级小于或等于当前操作符的优先级，  
 则弹出栈顶元素并添加到后缀表达式中，  
 直到遇到优先级更低的操作符或栈为空。\*/  
 }  
 Push(&S,infix[i]);  
 }  
 }  
 while(!isEmpty(&S)){  
 postfix[k++] = Pop(&S);  
 postfix[k++] = ' ';  
 /\*将它们全部弹出并添加到后缀表达式中。  
 这些通常是没有被括号包围的操作符\*/   
 }  
 postfix[k] = '\0';  
}   
  
//计算后缀表达式的值  
int evaluate(char \*postfix){  
 Stack S;  
 initStack(&S);  
 //用 strtok函数，根据" "分隔成多个字符串   
 char \*token = strtok(postfix," ");  
 while(token){  
 if(isdigit(\*token)){  
 Push(&S,atoi(token));  
 // atoi函数将字符串转换为一个整数   
 }  
 else{  
 int num1 = Pop(&S);  
 int num2 = Pop(&S);  
 int result;  
 switch(\*token){  
 case '+': result = num2 + num1; break;  
 case '-': result = num2 - num1; break;  
 case '\*': result = num2 \* num1; break;  
 case '/': if(num1==0){  
 printf("除数不为0\n");  
 exit(1);  
 }  
 result = num2 / num1;  
 break;  
 }  
 Push(&S,result);  
 }  
 token = strtok(NULL," ");  
 }  
 return Pop(&S);  
}   
  
int main(){  
 char infix[MAXSIZE], postfix[MAXSIZE];  
 printf("输入中缀表达式：\n");  
 scanf("%s",infix);  
 transform(infix,postfix);  
 printf("后缀表达式为：%s\n",postfix);  
 printf("Result:%d\n",evaluate(postfix));  
 return 0;  
}



#### 第五次作业

\* 题目描述

给定一个二维的 0-1 矩阵，其中 0 表示海洋，1 表示陆地。单独的或相邻的陆地可以形成岛屿，每个格子只与其上下左右四个格子相邻。求最大的岛屿面积。

\* 输入输出样例

输入是一个二维数组，输出是一个整数，表示最大的岛屿面积。

Input:

[ [1,0,1,1,0,1,0,1],

[1,0,1,1,0,1,1,1],

[0,0,0,0,0,0,0,1] ]

Output: 6

最大的岛屿面积为 6，位于最右侧。

#### 方法一：深度优先搜索

dfs 方法（深度优先搜索）用于计算以当前位置 (cur\_i, cur\_j) 为起点的岛屿的面积。这个方法首先检查当前位置是否越界或者是否为陆地（即单元格的值是否为1）。如果不满足条件，则返回0。然后，将当前位置标记为已访问（将 grid[cur\_i][cur\_j] 设置为0），并计算从当前位置出发，向四个方向（上、下、左、右）扩展的岛屿面积。每次递归调用 dfs 方法时，都会累加面积 ans。  
  
maxIsland 用于计算整个岛屿的最大面积。它遍历整个矩阵的每个单元格，对于每个值为1的单元格，调用 dfs 方法计算以该单元格为起点的岛屿面积，并通过 max 函数更新最大面积 ans。最后返回计算出的最大岛屿面积。

#include <stdio.h>  
#include <string.h>  
  
#define MAX\_SIZE 100 // 假设最大为100\*100  
  
// 定义一个存储二维数组的尺寸信息  
typedef struct {  
 int rows;  
 int cols;  
 int grid[MAX\_SIZE][MAX\_SIZE];  
} Island;  
  
// 深度优先搜索，计算岛屿面积  
int dfs(Island \*island, int rows, int cols, int cur\_i, int cur\_j) {  
 if (cur\_i < 0 || cur\_j < 0 || cur\_i >= rows || cur\_j >= cols || island->grid[cur\_i][cur\_j] != 1) {  
 return 0;  
 }  
 island->grid[cur\_i][cur\_j] = 0; //已检阅数组标记为0   
 int ans = 1; // 岛屿面积  
 int di[] = {0, 0, 1, -1}; // 行i的相对运动  
 int dj[] = {1, -1, 0, 0}; // 列j的相对运动  
 for (int index = 0; index < 4; ++index) {  
 int next\_i = cur\_i + di[index];  
 int next\_j = cur\_j + dj[index];  
 /\*di[0] = 0 和 dj[0] = 1：表示向右移动  
 di[1] = 0 和 dj[1] = -1：表示向左移动  
 di[2] = 1 和 dj[2] = 0：表示向下移动  
 di[3] = -1 和 dj[3] = 0：表示向上移动\*/  
 ans += dfs(island, rows, cols, next\_i, next\_j);  
 }  
 return ans;  
}  
  
// 自定义 max 函数  
int fun\_max(int a, int b) {  
 return (a > b) ? a : b;  
}  
  
// 计算最大岛屿面积  
int maxIsland(Island island) {  
 int ans = 0;  
 for (int i = 0; i < island.rows; ++i) {  
 for (int j = 0; j < island.cols; ++j) {  
 if (island.grid[i][j] == 1) {  
 ans = fun\_max(ans, dfs(&island, island.rows, island.cols, i, j));  
 island.grid[i][j] = 1; // 重置为1  
 }  
 }  
 }  
 return ans;  
}  
  
int main() {  
 int row, col;  
 printf("最大行数:\n");  
 scanf("%d", &row);  
 printf("最大列数:\n");  
 scanf("%d", &col);  
 Island island;  
 island.rows = row;  
 island.cols = col;  
 for (int i = 0; i < row; i++) {  
 for (int j = 0; j < col; j++) {  
 scanf("%d", &island.grid[i][j]);  
 }  
 putchar('\n');  
 }  
 int maxArea = maxIsland(island);  
 printf("最大面积是:%d\n", maxArea);  
 return 0;  
}

#### 方法二：广度优先

#include<stdio.h>  
#include<limits.h>  
#include<stdlib.h>  
  
// 比较两个整数，返回较大的那个  
int max(int a, int b) {  
 return (a > b) ? a : b;  
}  
  
int maxAreaOfIsland(int\*\* grid, int gridSize, int\* gridColSize) {  
 int m = gridColSize[0], n = gridSize, ans = 0; //m表示列数，n表示行数   
 int d[4][2] = {{0, -1}, {0, 1}, {-1, 0}, {1, 0}};  
 /\*{0, -1} 表示向上移动。  
 {0, 1} 表示向下移动  
 {-1, 0} 表示向左移动  
 {1, 0} 表示向右移动\*/  
 for (int i = 0; i < n; ++i) {  
 for (int j = 0; j < m; ++j) {  
 if (grid[i][j]) {  
 grid[i][j] = 0; //标记为已访问   
 //变量p用于计数连通区域的大小  
 //初始化队列que用于BFS，l和r分别表示队列的头和尾。  
 int p = 1, que[20000][2], l = 0, r = 0;  
 //将当前位置(i, j)入队。  
 que[r][0] = i, que[r++][1] = j;  
 while (l < r) { //当队列不为空时  
 //出队一个元素，得到当前要处理的位置(x, y)  
 int x = que[l][0], y = que[l++][1];  
 for (int k = 0; k < 4; k++) { //遍历四个方向（上、下、左、右）  
 //计算新位置(mx, my)，d是一个二维数组，存储了四个方向的偏移量  
 int mx = x + d[k][0], my = y + d[k][1];  
 //检查新位置是否在网格内且为连通区域的一部分  
 if (mx >= 0 && mx < n && my >= 0 && my < m && grid[mx][my]) {  
 //如果新位置有效，将其入队  
 que[r][0] = mx, que[r++][1] = my;  
 grid[mx][my] = 0; p++;  
 }  
 }  
 }  
 ans = max(ans, p);  
 }  
 }  
 }  
 return ans;  
}  
  
int main() {  
 int row, col;  
 printf("输入行数:\n");  
 scanf("%d", &row);  
 printf("输入列数:\n");  
 scanf("%d", &col);  
 int\*\* grid = (int\*\*)malloc(row \* sizeof(int\*));  
 for (int i = 0; i < row; i++) {  
 grid[i] = (int\*)malloc(col \* sizeof(int));  
 }  
 for (int i = 0; i < row; i++) {  
 for (int j = 0; j < col; j++) {  
 scanf("%d", &grid[i][j]);  
 }  
 putchar('\n');  
 }  
 int\* gridColSize = (int\*)malloc(1 \* sizeof(int));  
 gridColSize[0] = col;  
 int maxArea = maxAreaOfIsland(grid, row, gridColSize);  
 printf("最大面积是:%d\n", maxArea);  
 for (int i = 0; i < row; i++) {  
 free(grid[i]);  
 }  
 free(grid);  
 free(gridColSize);  
 return 0;  
}

### 6、第六次作业

#### 在先序遍历中建立以下二叉树，再中序遍历和后序遍历并输出结果。

#include<stdio.h>  
#include<stdlib.h>  
  
typedef struct TreeNode{  
 char data;  
 struct TreeNode \*left;  
 struct TreeNode \*right;  
} TreeNode;  
  
// 创建一个新结点  
TreeNode\* createNode(char elem){  
 TreeNode\* newNode=(TreeNode\*)malloc(sizeof(TreeNode));  
 if(newNode==NULL){  
 printf("Error.\n");  
 exit(1);  
 }  
 newNode->data = elem;  
 newNode->left = NULL;  
 newNode->right = NULL;  
 return newNode;  
}   
  
// 向二叉树插入结点  
TreeNode\* buildTree(char\* preorder, int\* index) {  
 if (preorder[\*index] == '\0') {  
 return NULL;  
 }  
   
 // 创建一个新的树节点，内容为 preorder 中当前 index 位置的字符  
 TreeNode\* root = createNode(preorder[\*index]);  
 (\*index)++;  
  
 // 如果当前节点是运算符，则需要继续构建左右子树  
 if (root->data == '+' || root->data == '-' || root->data == '\*' || root->data == '/') {  
 root->left = buildTree(preorder, index);  
 root->right = buildTree(preorder, index);  
 }  
  
 return root;  
}  
  
// 访问节点的函数  
void visit(TreeNode\* node){  
 if(node != NULL){  
 printf("%c ", node->data);  
 }  
}  
  
// 中序遍历  
void inorder(TreeNode\* root){  
 if(root != NULL){  
 inorder(root->left);  
 visit(root);  
 inorder(root->right);  
 }  
}   
  
// 后序遍历  
void postorder(TreeNode\* root){  
 if(root != NULL){  
 postorder(root->left);  
 postorder(root->right);  
 visit(root);  
 }  
}   
  
// 释放二叉树内存  
void freeTree(TreeNode\* root){  
 if(root != NULL){  
 freeTree(root->left);  
 freeTree(root->right);  
 free(root);  
 }  
}   
  
int main(){  
 char preorder[] = "-+a\*b-cd/ef";  
 int index = 0; // 指向当前处理位置   
 // 建立二叉树  
 TreeNode\* root = buildTree(preorder, &index);  
 // 中序遍历  
 printf("中序遍历结果:\n");  
 inorder(root);  
 printf("\n");  
 // 后序遍历  
 printf("后序遍历结果:\n");  
 postorder(root);  
 printf("\n");  
 // 释放内存  
 freeTree(root);  
   
 return 0;  
}

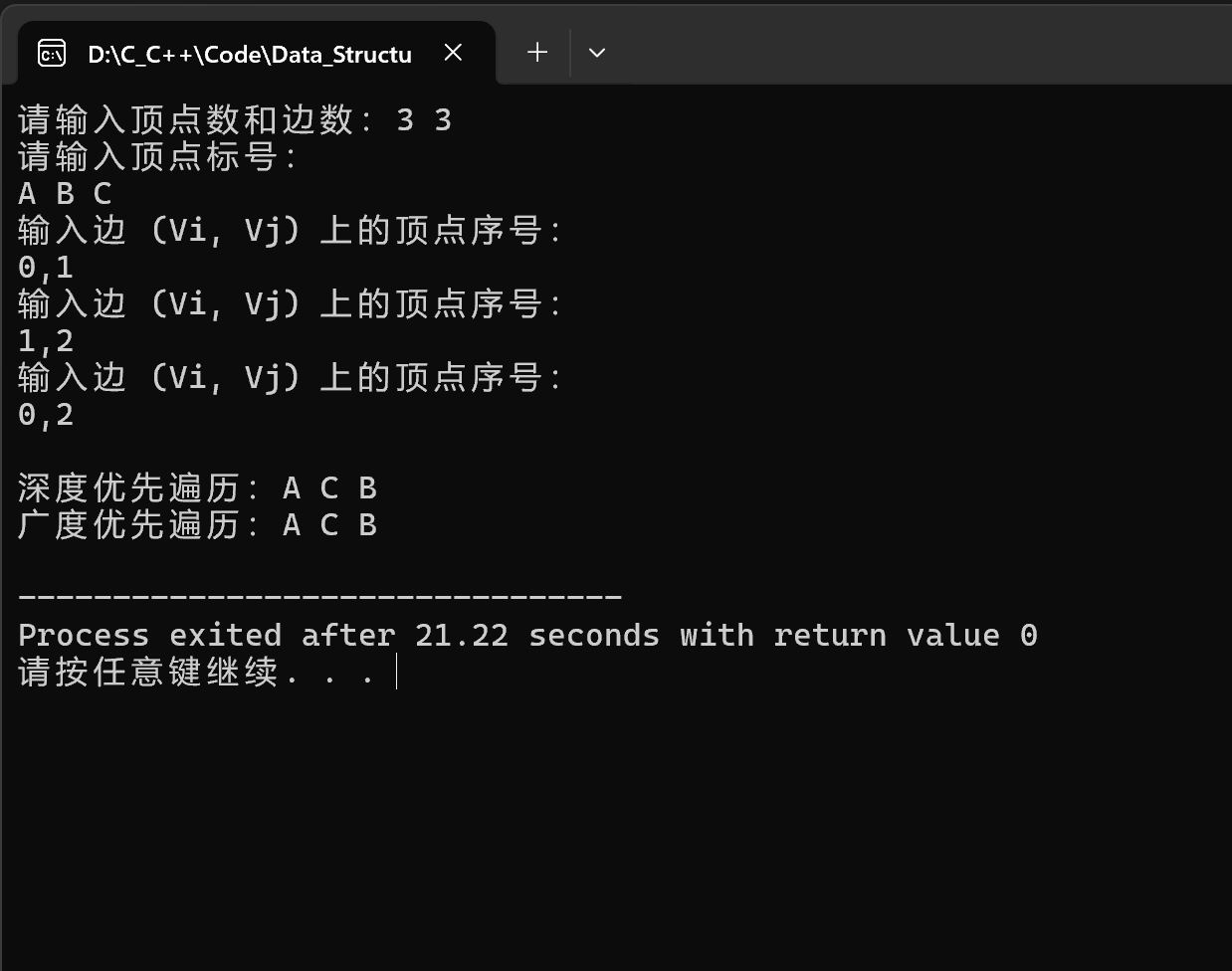
### 第八次作业

图的广度优先遍历和图的深度优先遍历

#include <stdio.h>   
#include <stdlib.h>   
   
#define MAX 20   
   
typedef struct EdgeNode {   
 int adjvex;   
 struct EdgeNode \*next;   
 int weight;   
} EdgeNode;   
   
   
typedef struct VertexNode {   
 char data;   
 EdgeNode \*firstedge;   
} VertexNode;   
   
   
typedef struct {   
 VertexNode adjlist[MAX];   
 int n, e; // 顶点数和边数   
} GraphAdjlist;   
   
int visited[MAX]; // 访问标志数组（访问过赋值为1，未访问赋值为0）   
   
// 创建邻接表   
void create(GraphAdjlist \*G) {   
 int i, j, k;   
 EdgeNode \*e;   
   
 printf("请输入顶点数和边数：");   
 scanf("%d%d", &G->n, &G->e);   
 getchar(); // 清除缓冲   
   
 printf("请输入顶点标号：\n");   
 for (i = 0; i < G->n; i++) {   
 scanf("%c", &G->adjlist[i].data); // 输入顶点编号   
 G->adjlist[i].firstedge = NULL; // 初始化边表为空   
 getchar();   
 }   
   
 for (k = 0; k < G->e; k++) {   
 printf("输入边 (Vi, Vj) 上的顶点序号：\n");   
 scanf("%d%d", &i, &j);   
   
 // 使用头插法加入边表结点   
 e = (EdgeNode \*)malloc(sizeof(EdgeNode));   
 e->adjvex = j;   
 e->next = G->adjlist[i].firstedge;   
 G->adjlist[i].firstedge = e;   
   
 // 无向图：对称加入边表结点   
 e = (EdgeNode \*)malloc(sizeof(EdgeNode));   
 e->adjvex = i;   
 e->next = G->adjlist[j].firstedge;   
 G->adjlist[j].firstedge = e;   
 }   
 printf("\n");   
}   
   
// 深度优先搜索   
void DFS(GraphAdjlist \*G, int i) {   
 EdgeNode \*p;   
 visited[i] = 1;   
 printf("%c ", G->adjlist[i].data);   
   
 p = G->adjlist[i].firstedge;   
 while (p != NULL) {   
 if (visited[p->adjvex] == 0) {   
 DFS(G, p->adjvex);   
 }   
 p = p->next;   
 }   
}   
   
// 深度优先遍历   
void DFSTraverse(GraphAdjlist \*G) {   
 for (int i = 0; i < G->n; i++) {   
 visited[i] = 0; // 初始化访问标志   
 }   
 for (int i = 0; i < G->n; i++) {   
 if (visited[i] == 0) {   
 DFS(G, i);   
 }   
 }   
}   
   
// 广度优先搜索   
void BFS(GraphAdjlist \*G, int v) {   
 EdgeNode \*p;   
 int queue[MAX], front = 0, rear = 0;   
 int w;   
   
 for (int i = 0; i < G->n; i++) {   
 visited[i] = 0; // 初始化访问标志   
 }   
   
 printf("%c ", G->adjlist[v].data);   
 visited[v] = 1;   
 rear = (rear + 1) % MAX;   
 queue[rear] = v;   
   
 while (front != rear) {   
 front = (front + 1) % MAX;   
 w = queue[front];   
   
 p = G->adjlist[w].firstedge;   
 while (p != NULL) {   
 if (visited[p->adjvex] == 0) {   
 printf("%c ", G->adjlist[p->adjvex].data);   
 visited[p->adjvex] = 1;   
 rear = (rear + 1) % MAX;   
 queue[rear] = p->adjvex;   
 }   
 p = p->next;   
 }   
 }   
 printf("\n");   
}   
   
int main() {   
 GraphAdjlist G;   
 create(&G);   
   
 printf("深度优先遍历：");   
 DFSTraverse(&G);   
   
 printf("\n广度优先遍历：");   
 BFS(&G, 0);   
   
 return 0;   
}

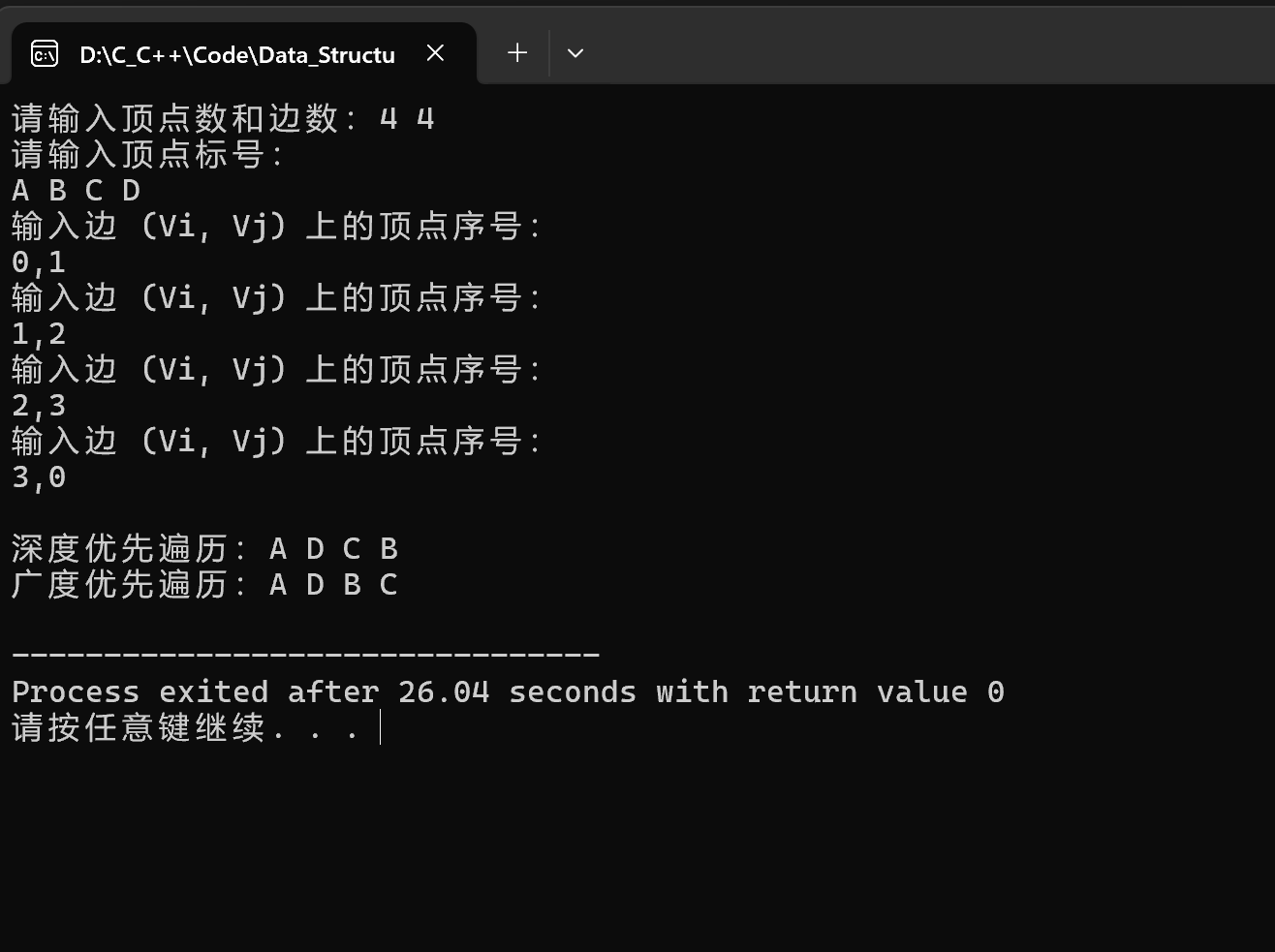
### 测试案例 1：简单无向图

* **顶点数和边数**：3个顶点，3条边
* **顶点标号**：A, B, C
* 边
* ：
  + (0, 1) 表示顶点A和顶点B之间有一条边
  + (1, 2) 表示顶点B和顶点C之间有一条边
  + (0, 2) 表示顶点A和顶点C之间有一条边



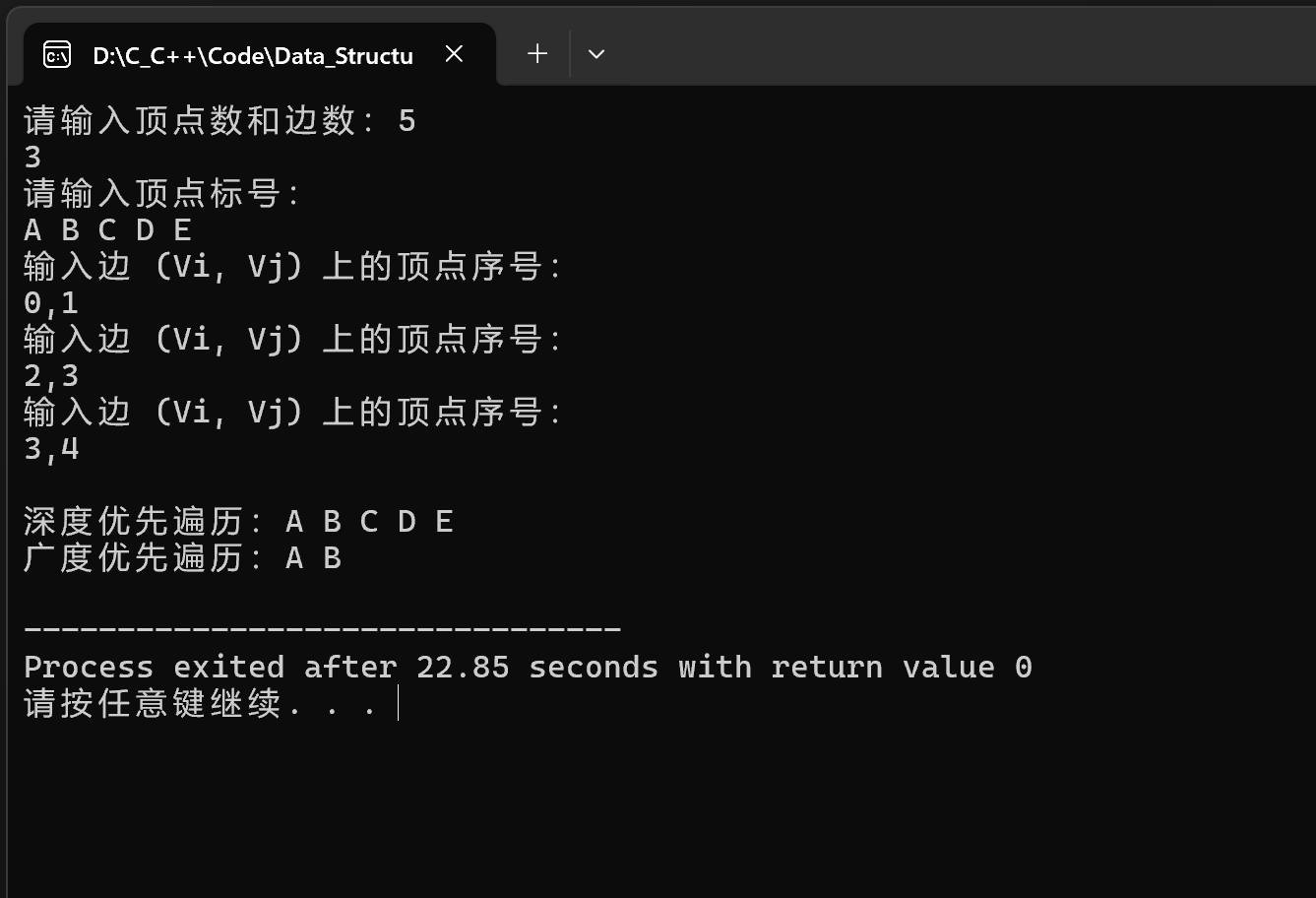
### 测试案例 2：包含环的无向图

* **顶点数和边数**：4个顶点，4条边
* **顶点标号**：A, B, C, D
* 边
* ：
  + (0, 1)
  + (1, 2)
  + (2, 3)
  + (3, 0)



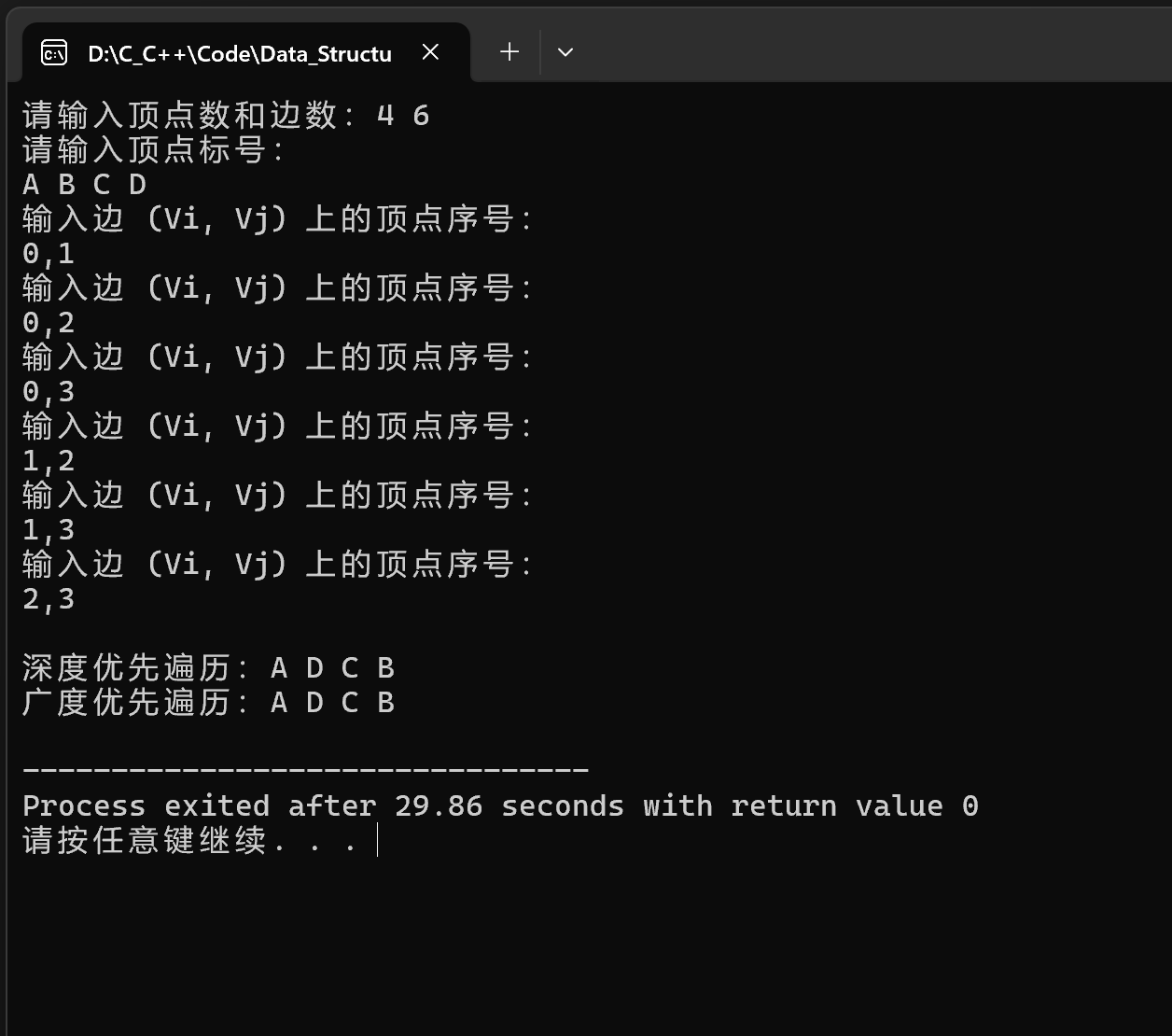
### 测试案例 3：不连通的无向图

* **顶点数和边数**：5个顶点，3条边
* **顶点标号**：A, B, C, D, E
* 边
* ：
  + (0, 1)
  + (2, 3)
  + (3, 4)



### 测试案例 4：完全图

* **顶点数和边数**：4个顶点，6条边
* **顶点标号**：A, B, C, D
* 边
* ：
  + (0, 1)
  + (0, 2)
  + (0, 3)
  + (1, 2)
  + (1, 3)
  + (2, 3)



### 测试案例 5：星形图

* **顶点数和边数**：5个顶点，4条边
* **顶点标号**：A, B, C, D, E
* 边
* ：
  + (0, 1)
  + (0, 2)
  + (0, 3)
  + (0, 4)

