# 《计算机软件技术基础》实验报告I—数据结构

班号：

目录

实验一：顺序表（一）……………………………….………3

实验二：顺序表（二）……………………………………….7

实验三：单链表……………………………………………...10

实验四：顺序栈……………………………………………...16

实验五：链栈………………………………………………...20

实验六：队列……………………………………………...…24

实验七：二叉树………………………………………...……28

实验八：二叉排序树………………………………………...32

实验九：查找……………………………………………...…35

实验十：排序……………………………………………...…38

实验十一：图………………………………………………...42

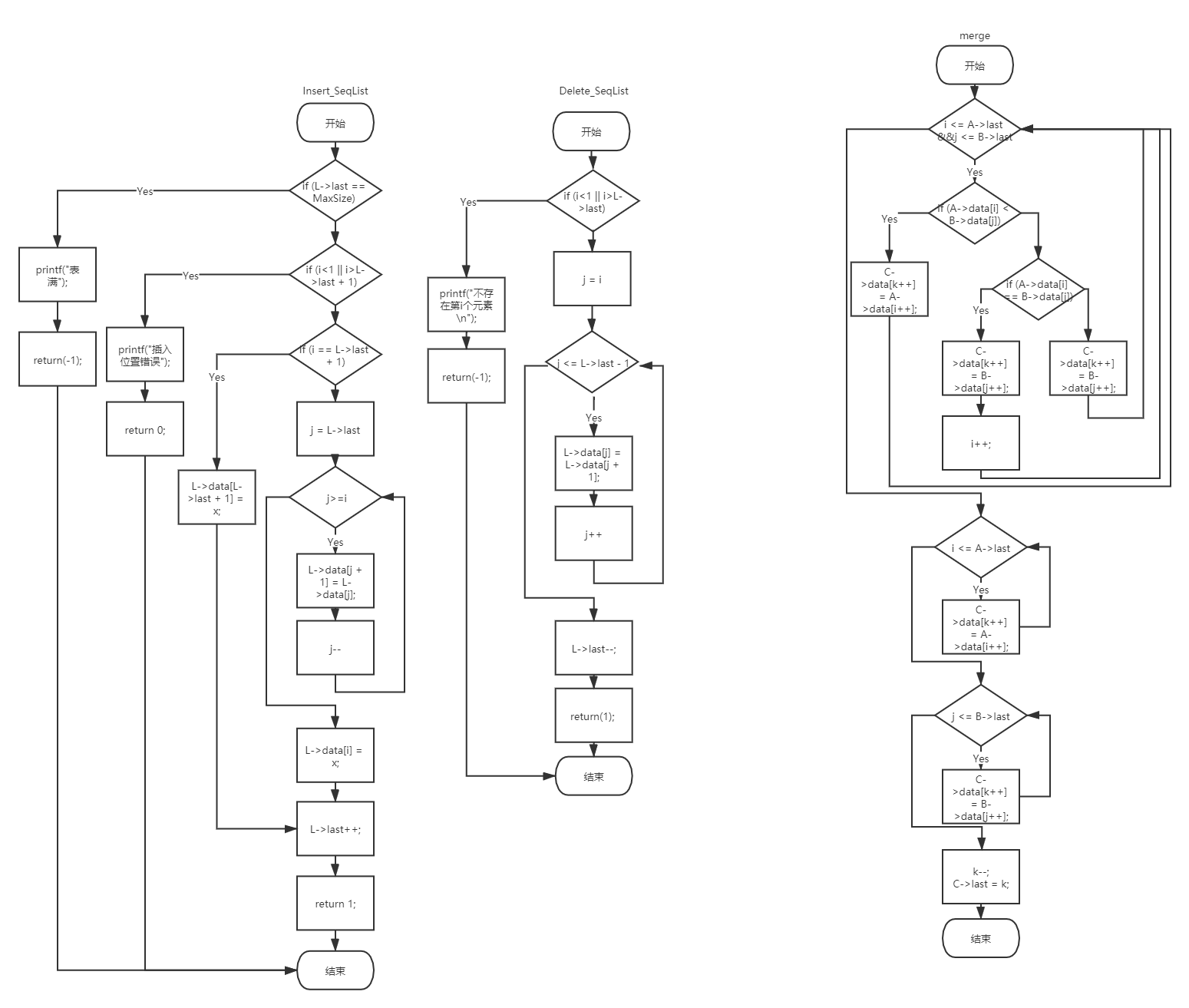
实验十二：约瑟夫斯问题求解……………………………...49

实验十四：内部排序算法的实现与比较…………………...53

**实验一**

1.对象、目的和要求以顺序表为对象，进行定义、创建、插入和删除操作，显示数据元素。

2.函数流程图



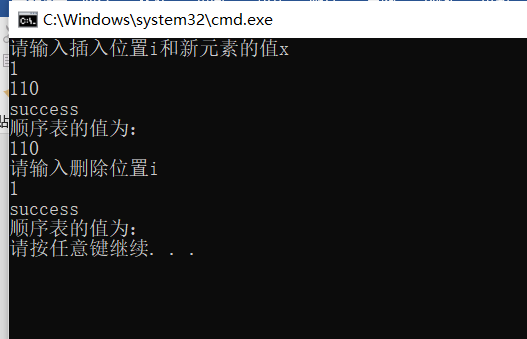
3.程序的数据输入要求、测试数据与运行结果

（1）插入操作

数据输入：输入插入数据的位置i，要插入的元素x，数据类型都为整型

测试数据：插入位置为1，插入元素为110

运行结果：成功

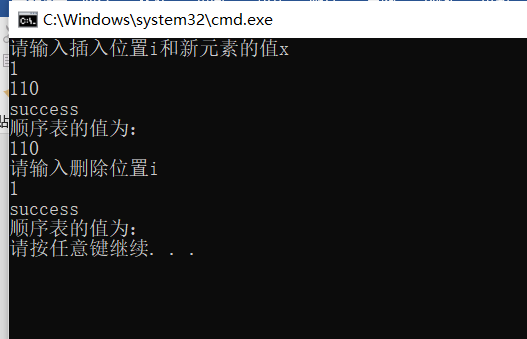


（2）删除操作

数据输入：输入要删除的数据的位置i数据类型为整型

测试数据：删除位置为1

运行结果：成功



4.带有注释的源程序

//实验一：顺序表的定义、创建、插入和删除操作，显示数据元素。

#include<stdio.h>

#define SeqList struct listtype

#define MaxSize 1000

#define ElemType int

SeqList

{

ElemType data[MaxSize + 1];

int last;

};//顺序表的定义

SeqList \*init\_SeqList()//顺序表的创建

{

SeqList \*L;

L = (SeqList\*)malloc(sizeof(SeqList));

L->last = 0;

return L;

}

int Insert\_SeqList(SeqList \*L, int i, ElemType x)// 顺序表插入操作，i是逻辑序号

{

int j;

if (L->last == MaxSize) //空间已满，不能插入

{

printf("表满");

return(-1);}

if (i<1 || i>L->last + 1) //插入位置不合法

{

printf("插入位置错误");

return 0;

}

if (i == L->last + 1)

L->data[L->last + 1] = x;

else

{

for (j = L->last; j>=i; j--)

L->data[j + 1] = L->data[j]; //结点移动

L->data[i] = x; //新元素插入

}

L->last++;//last指向最后元素

return 1; //插入成功

}

int Delete\_SeqList(SeqList \*L, int i) //顺序表的删除操作

{

int j;

if (i<1 || i>L->last) //插入位置不合法

{

printf("不存在第i个元素\n");

return(-1);

}

for (j = i; j <= L->last - 1; j++)

L->data[j] = L->data[j + 1]; //结点移动

L->last--;

return(1); //删除成功

}

int main()

{

SeqList \*L;

L = init\_SeqList();//创建顺序表

int i,flag;//flag用来存储用户选择

ElemType x;

printf("请输入插入位置i和新元素的值x\n");

scanf\_s("%d%d",&i,&x,2);

getchar();

flag=Insert\_SeqList(L,i,x);

if (flag == 1)//插入成功

printf("success\n");

printf("顺序表的值为：\n");

for (int m = 1; m <= L->last; m++)

{

printf("%d ",L->data[m]); //顺序表的输出}

printf("\n请输入删除位置i\n");

scanf\_s("%d", &i, 1);

getchar();

flag = Delete\_SeqList(L, i);

if (flag == 1)//删除成功

printf("success\n");

printf("顺序表的值为：\n");

for (int m = 1; m <= L->last; m++)

{

printf("%d ", L->data[m]); //顺序表的输出

}

return 1;

}

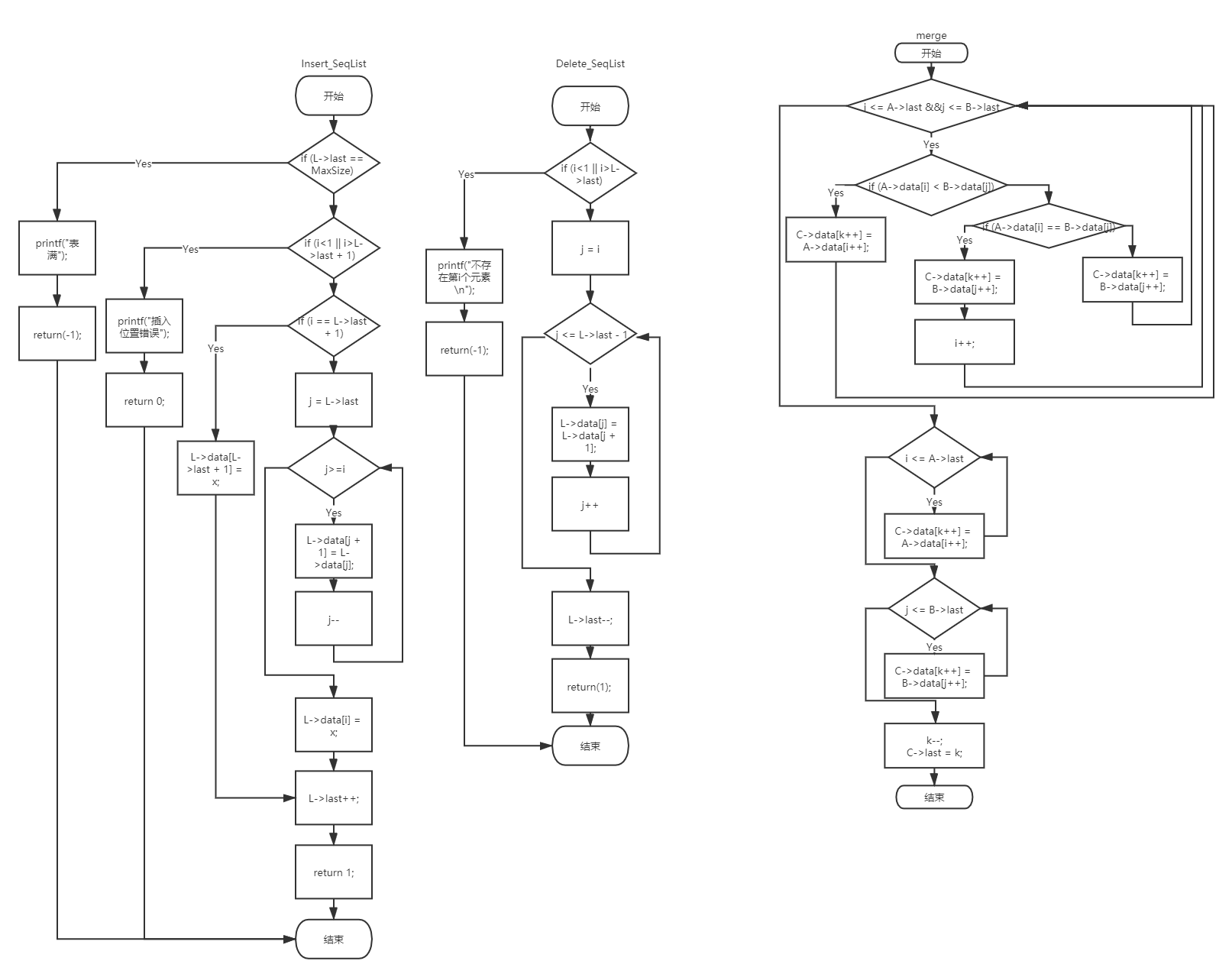
5.实验的收获：

实验中遇到的问题是插入新元素后，顺序表的元素数量总是不能正常地加一，发现问题出在L->last上，于是增加了“L->last++;”这一语句，使每次操作后，last都指向最后元素。这个方法成功解决了我的问题。

**实验二**

1.对象、目的和要求以顺序表为对象，进行顺序表的合并操作，并且使合并后的顺序表共同拥有的元素只存其一，显示数据元素。

2.函数流程图



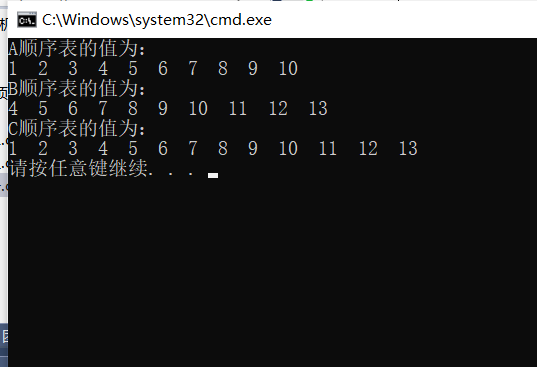
3.程序的数据输入要求、测试数据与运行结果

顺序表的合并

数据输入：A与B顺序表的元素，数据类型都为整型

测试数据：A:1 2 3 4 5 6 7 8 9 10,B:4 5 6 7 8 9 10 11 12 13

运行结果：成功



4.带有注释的源程序

//实验二：顺序表的合并（说明：将共同拥有的元素只存其一。）

#include<stdio.h>

#define SeqList struct listtype

#define MaxSize 1000

#define ElemType int

SeqList

{

ElemType data[MaxSize + 1];

int last;

}; //顺序表的定义

void merge(SeqList \*A, SeqList\*B, SeqList\*C)

{

int i, j, k;

i = 0; j = 0; k = 0;

while (i <= A->last &&j <= B->last)

if (A->data[i] < B->data[j])

C->data[k++] = A->data[i++];//将A、B中较小的元素先赋值给C的元素

else

{

if (A->data[i] == B->data[j])//若A、B中元素相等

{

C->data[k++] = B->data[j++];

i++;//赋值后A、B都跳至下一个元素的比较

}

else

C->data[k++] = B->data[j++];

}

while (i <= A->last)//若B表结束，A表未结束

C->data[k++] = A->data[i++];

while (j <= B->last) //若A表结束，B表未结束

C->data[k++] = B->data[j++];

k--;

C->last = k;}

int main()

{

SeqList \*A;

A = (SeqList\*)malloc(sizeof(SeqList));

for (int m = 1; m <= 10; m++)

{

A->data[m] = m;

}

A->last = 10;// A:1 2 3 4 5 6 7 8 9 10

printf("A顺序表的值为：\n");

for (int m = 1; m <= A->last; m++)

printf("%d ", A->data[m]);

printf("\n");

SeqList \*B;

B = (SeqList\*)malloc(sizeof(SeqList));

for (int m = 4; m <= 17; m++)

{

B->data[m-3] = m;

}

B->last = 10;// B:4 5 6 7 8 9 10 11 12 13

printf("B顺序表的值为：\n");

for (int m = 1; m <= B->last; m++)

printf("%d ", B->data[m]);

printf("\n");

SeqList \*C;

C = (SeqList\*)malloc(sizeof(SeqList));

C->last = 0;

merge(A, B, C);

printf("C顺序表的值为：\n");

for (int m = 1; m <= C->last; m++)

printf("%d ", C->data[m]);

printf("\n");

}

5.实验的收获：

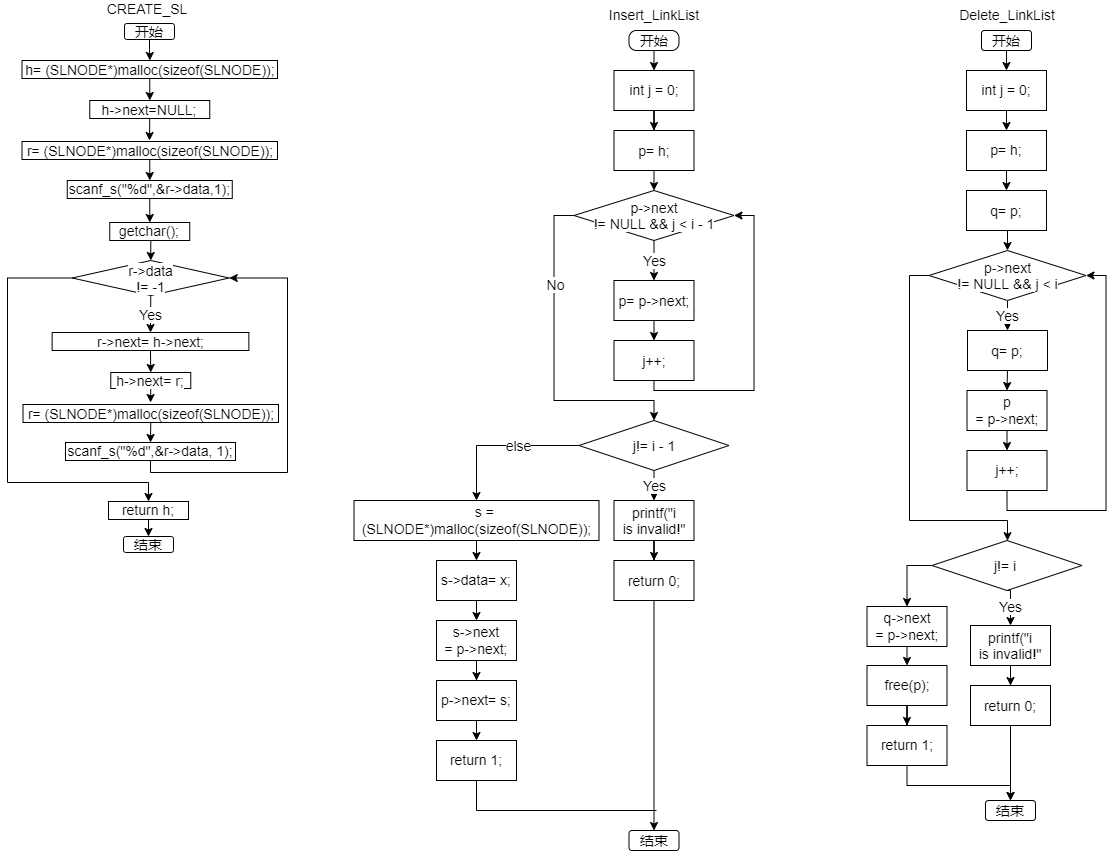
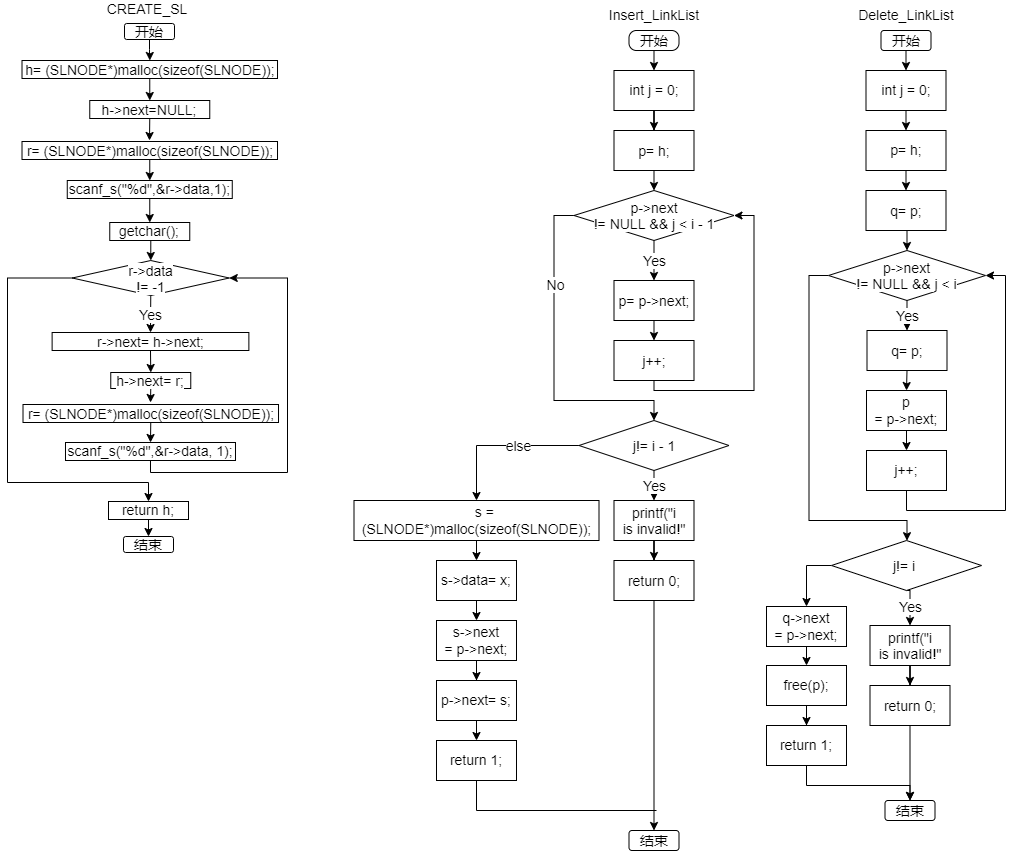
实验中遇到的问题是，将A与B合并成一个顺序表时，忽略了当遇到A、B共同拥有的元素，应该只保留一个元素的情况，即在A->data[i] == B->data[j]时，应避免合并后得到的顺序表中有重复元素。问题发现后，我增加了一段程序，在A->data[i] == B->data[j]时，将A中元素传给新顺序表C，而将B中元素不传入，并开始下一个元素的扫描。

**实验三**

1.对象、目的和要求

以单链表为对象，进行定义、创建、插入和删除操作，显示数据元素。

2.函数流程图

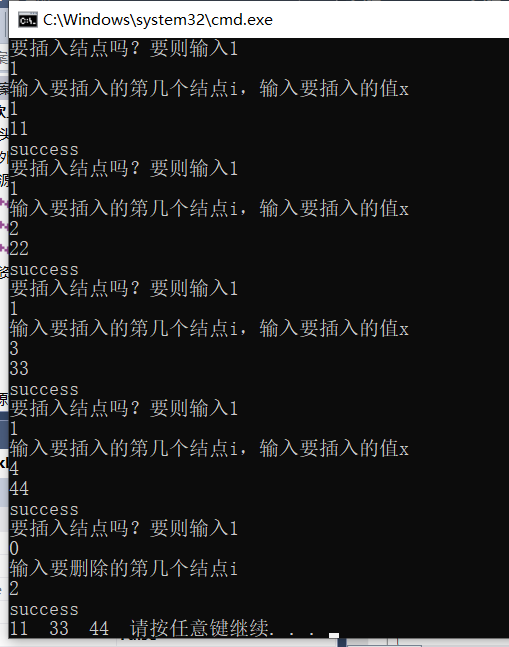
3.程序的数据输入要求、测试数据与运行结果

（1）插入操作

数据输入：输入插入数据的位置i，要插入的元素x，数据类型都为整型

测试数据：插入位置分别为1、2、3、4，插入元素分别为11、22、33、44

运行结果：成功

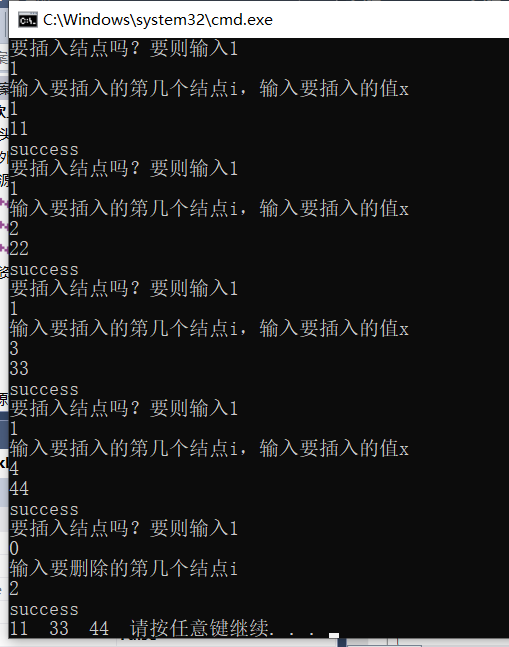


（2）删除操作

数据输入：输入要删除的第i个结点，数据类型为整型

测试数据：要删除第2个结点

运行结果：成功



4.带有注释的源程序

//实验三：单链表的定义、创建、插入和删除操作，显示数据元素。

#include<stdio.h>

#define NULL 0

#define SLNODE struct sl\_node

#define ElemType int

SLNODE{

ElemType data;

SLNODE \*next;

};

SLNODE \*CREATE\_SL(SLNODE \*h)

{

SLNODE \*r;

h = (SLNODE\*)malloc(sizeof(SLNODE));//申请内存单元

h->next=NULL;//创建一个空表

r = (SLNODE\*)malloc(sizeof(SLNODE)); //申请内存单元

scanf\_s("%d",&r->data,1);

getchar();

while (r->data != -1)//单链表结束的标志

{

r->next = h->next;

h->next = r;

r = (SLNODE\*)malloc(sizeof(SLNODE)); //申请内存单元

scanf\_s("%d", &r->data, 1);

}

return h;}

int Insert\_LinkList(SLNODE\*h, int i, ElemType x)

{ //在单链表的第i个位置上插入值为x的元素

SLNODE\*p, \*s;

int j = 0;

p = h;

while (p->next != NULL && j < i - 1)

{

p = p->next;

j++;

}

if (j != i - 1)//第i-1个不存在，不能插入

{

printf("i is invalid!");

return 0;}

else

{

s = (SLNODE\*)malloc(sizeof(SLNODE));//申请新结点

s->data = x;

s->next = p->next;

p->next = s;//p结点后面插入新结点s

return 1;}

}

int Delete\_LinkList(SLNODE\*h, int i)

{

SLNODE\*p, \*q;

int j = 0;

p = h;

q = p;

while (p->next != NULL && j < i )

{

q = p;

p = p->next;

j++;

}//查找i结点

if (j != i )

{

printf("i is invalid!");

return 0;}

else

{

q->next = p->next;

free(p);//删除

return 1;

}

}

int Show\_LinkList(SLNODE\*h)

{

SLNODE\*p;

p = h;

while (p->next != NULL)

{

p = p->next;

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

}

printf("\n");}

int main()

{

int i,flag,x;

flag = 1;

SLNODE \*h;SLNODE \*HEAD;

HEAD = (SLNODE\*)malloc(sizeof(SLNODE));

HEAD->next == NULL;

h= (SLNODE\*)malloc(sizeof(SLNODE));

h->next == NULL;

for (;;)

{

printf("要插入结点吗？要则输入1\n");

scanf\_s("%d", &flag, 1);

if (flag == 1)

{

printf("输入要插入的第几个结点i，输入要插入的值x\n");

scanf\_s("%d%d", &i, &x, 2);

flag = Insert\_LinkList(h, i, x);

if (flag == 1)

printf("success\n");

//Show\_LinkList(h);

}

else break;

}

printf("输入要删除的第几个结点i\n");

scanf\_s("%d", &i,1);

flag=Delete\_LinkList(h, i);

if (flag == 1)

printf("success\n");

Show\_LinkList(h);

return 1;

}

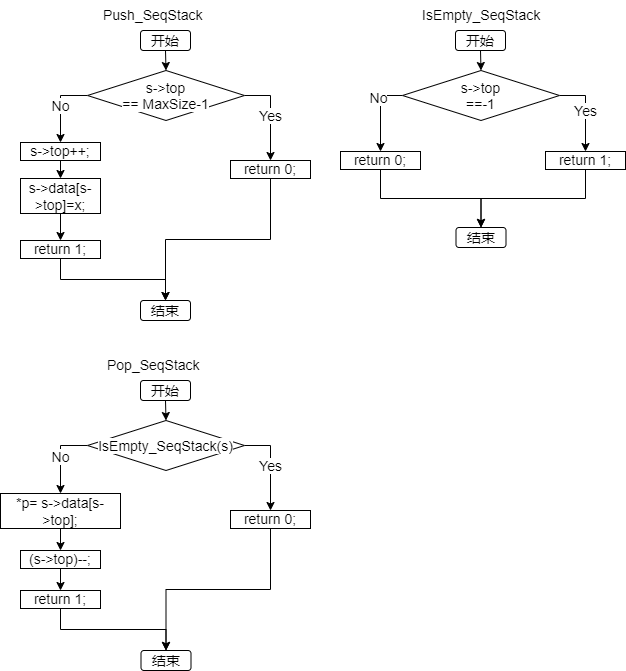
5.实验的收获：

单链表的插入有前插和后插法，前插的时间复杂度为O(n), 后插的时间复杂度为O(1)；单链表的删除分为删除前驱结点和删除后继结点，删除前驱结点的时间复杂度为O(n), 删除后继结点的时间复杂度为O(1)。因此可以建议在实验要求上增加“考虑时间效益”等条件，对同学们关于单链表操作的理解进一步考察。

**实验四**

1.对象、目的和要求

以顺序栈为对象，进行定义、创建、插入和删除操作，显示数据元素。

2.函数流程图

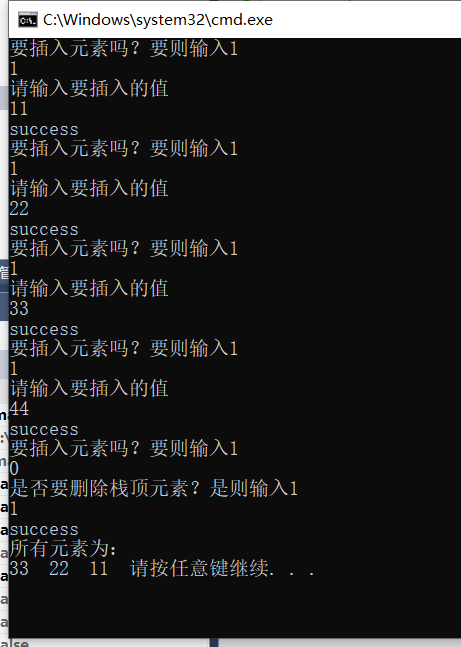
3.程序的数据输入要求、测试数据与运行结果

（1）插入操作

数据输入：输入要插入的元素x，数据类型为整型

测试数据：插入元素分别为11、22、33、44

运行结果：成功

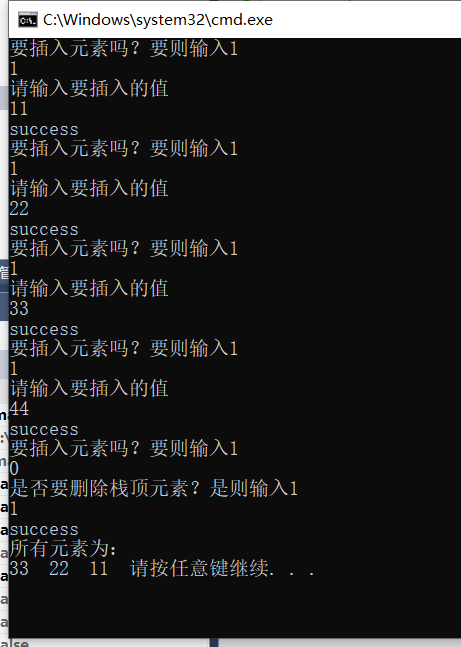


（2）删除操作

数据输入：输入是否要删除栈顶元素，是则输入1，数据类型为整型

测试数据：要删除

运行结果：成功



4.带有注释的源程序

//实验四：顺序栈的定义、创建、插入和删除操作，显示数据元素。

#include<stdio.h>

#define MaxSize 1000

#define ElemType int

typedef struct stack

{

ElemType data[MaxSize];

int top;

}SqStack;

SqStack \*Init\_SeqStack()//置空栈

{

SqStack \*s;

s = (SqStack\*)malloc(sizeof(SqStack));

s->top = -1;

return s;

}

int Push\_SeqStack(SqStack \*s, ElemType x) //入栈

{

if (s->top == MaxSize-1)

return 0;// 栈满不能入栈

else

{

s->top++;

s->data[s->top]=x;

return 1;}

}

int IsEmpty\_SeqStack(SqStack \*s)//判空栈

{

if (s->top ==-1)

return 1;

else

return 0;

}

int Pop\_SeqStack(SqStack\*s, ElemType \*p) //出栈

{

if (IsEmpty\_SeqStack(s))

return 0;// 栈空不能出栈

else

{

\*p = s->data[s->top];

(s->top)--;

return 1;

}//栈顶元素存入x，返回

}

int main()

{

SqStack \*s;

s=Init\_SeqStack();

int flag;

ElemType x;

ElemType \*p;

p = &x;

for (;;)

{

printf("要插入元素吗？要则输入1\n");

scanf\_s("%d", &flag, 1);

if (flag == 1)

{

printf("请输入要插入的值\n");

scanf\_s("%d", &x, 1);

flag = Push\_SeqStack(s, x);

if (flag == 1)

printf("success\n");}

else break;

}

printf("是否要删除栈顶元素？是则输入1\n");

scanf\_s("%d", &flag, 1);

if (flag == 1)

{

flag = Pop\_SeqStack(s, p);

if (flag == 1)

printf("success\n");}

printf("所有元素为：\n");

for (;;)

{

flag = Pop\_SeqStack(s, p);

if (flag == 1)

printf("%d ", \*p);

else break;

}

return 1;

}

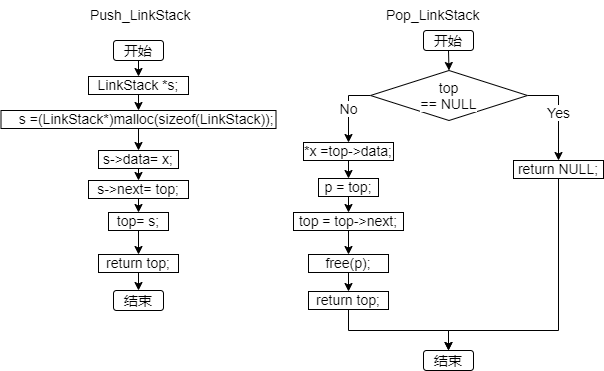
5.实验的收获：

实验中遇到的疑惑是出栈与删除栈顶元素用的是同一个算法，这意味着如果在程序中想要输出所有元素，则实际效果等同于删除所有元素，所以必须在输出所有元素完成所有编辑操作，并且输出所有元素是一次性的。

**实验五**

1.对象、目的和要求

以链栈为对象，进行定义、创建、插入和删除操作，显示数据元素。

2.函数流程图

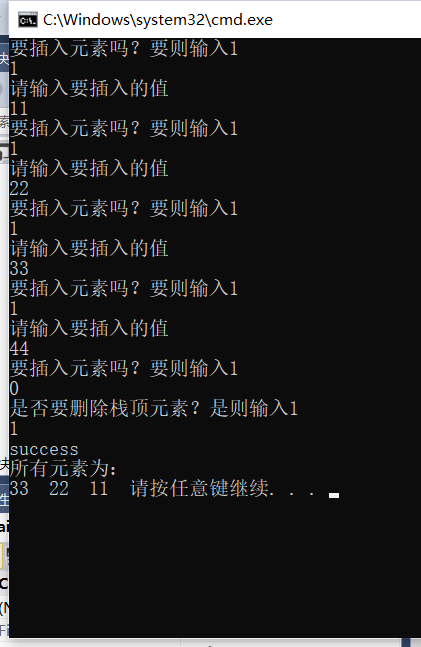
3.程序的数据输入要求、测试数据与运行结果

（1）插入操作

数据输入：输入要插入的元素x，数据类型为整型

测试数据：插入元素分别为11、22、33、44

运行结果：成功

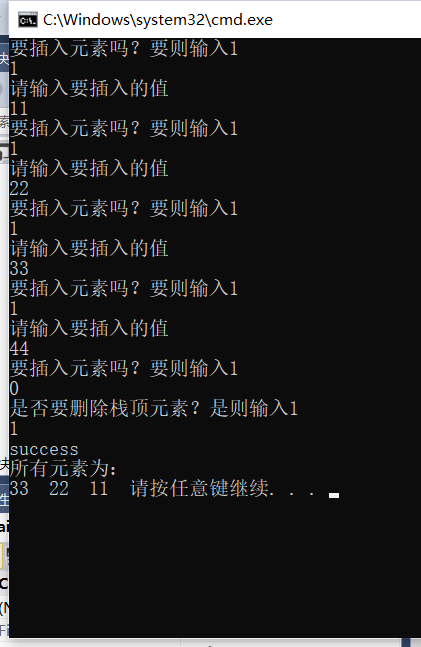


（2）删除操作

数据输入：输入是否要删除栈顶元素，是则输入1，数据类型为整型

测试数据：要删除

运行结果：成功



4.带有注释的源程序

//实验五：链栈的定义、创建、插入和删除操作，显示数据元素。

#include<stdio.h>

#define MaxSize 1000

#define ElemType int

#define LinkStack struct linkstack

LinkStack

{

ElemType data;

LinkStack \*next;

};

LinkStack \*Init\_LinkStack()//初始化

{

LinkStack \*top;

top = (LinkStack\*)malloc(sizeof(LinkStack));//分配内存单元

top = NULL;

return top;

}

LinkStack \*Push\_LinkStack(LinkStack\*top,ElemType x)//入栈

{

LinkStack \*s;

s = (LinkStack\*)malloc(sizeof(LinkStack)); //分配内存单元

s->data = x;

s->next = top;

top = s;

return top;

}//不需要判空

LinkStack \*Pop\_LinkStack(LinkStack \*top, ElemType \*x)//出栈

{

LinkStack \*p;

if (top == NULL)

return NULL;

else{

\*x = top->data;

p = top;

top = top->next;

free(p);

return top;

}

}//需要判空

int main()

{

LinkStack \*top;

top=Init\_LinkStack();

int flag;

ElemType x;

ElemType \*px;

px = &x;

for (;;)

{

printf("要插入元素吗？要则输入1\n");

scanf\_s("%d", &flag, 1);

if (flag == 1)

{

printf("请输入要插入的值\n");

scanf\_s("%d", &x, 1);

top = Push\_LinkStack(top,x);

}

else break;

}

printf("是否要删除栈顶元素？是则输入1\n");

scanf\_s("%d", &flag, 1);

if (flag == 1)

{

top = Pop\_LinkStack(top, px);

flag = top;

if (flag != NULL)

printf("success\n");

if (flag == NULL) printf("error\n");

}

printf("所有元素为：\n");

for (;;)

{

if (top != NULL)

{

top = Pop\_LinkStack(top, px);

printf("%d ", \*px);

}

else break;

}

return 1;

}

5.实验的收获：

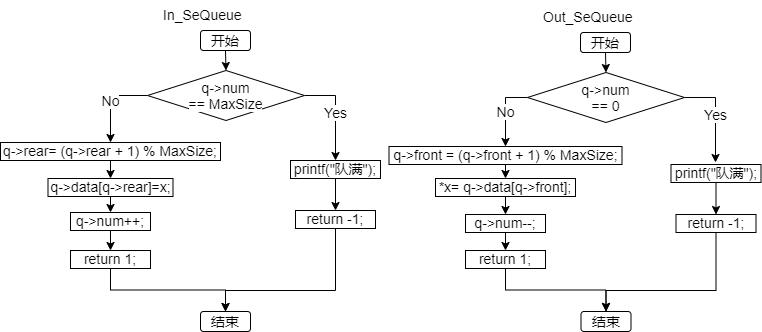
链栈实验中遇到的疑惑与顺序栈情况相似，即如果在程序中想要输出所有元素，则实际效果等同于删除所有元素，所以必须在输出所有元素完成所有编辑操作，并且输出所有元素是一次性的。

**实验六**

1.对象、目的和要求

以队列为对象，进行定义、创建、插入和删除操作，显示数据元素。

2.函数流程图



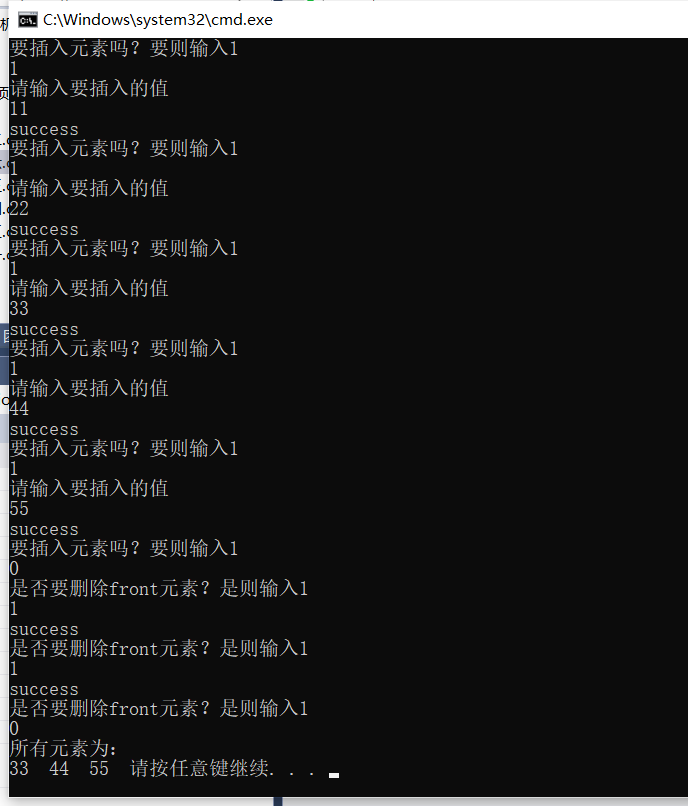
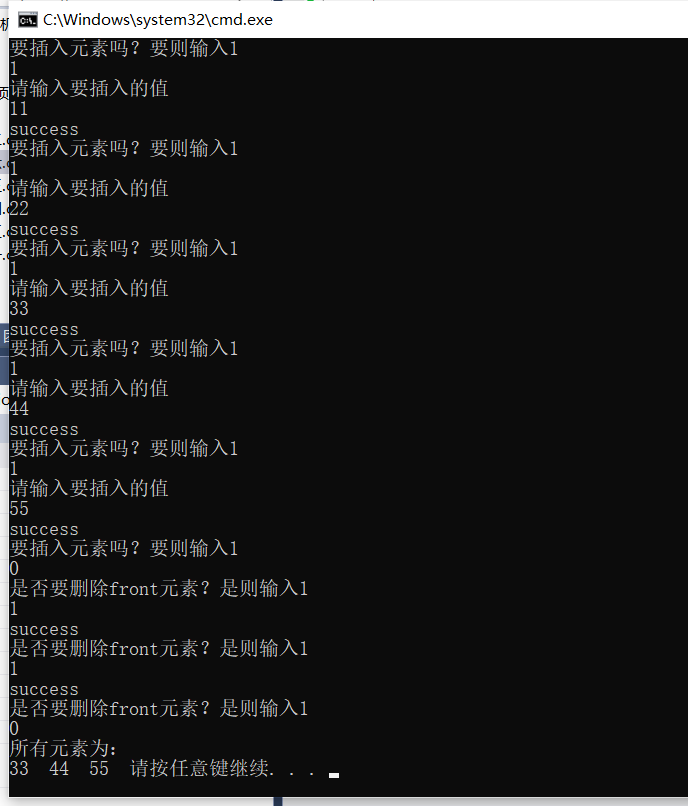
3.程序的数据输入要求、测试数据与运行结果

（1）插入操作

数据输入：输入要插入的元素x，数据类型为整型

测试数据：插入元素分别为11、22、33、44、55

运行结果：成功

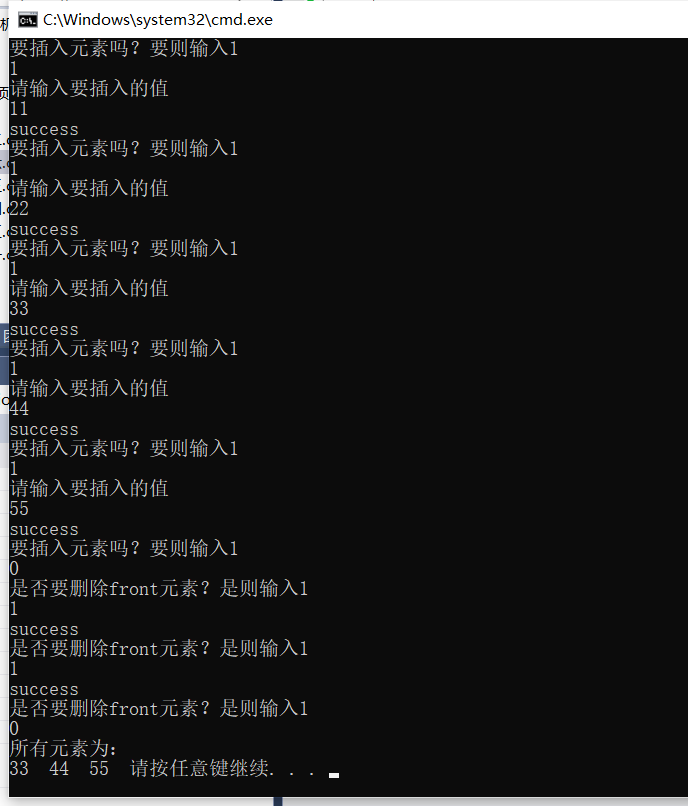


（2）删除操作

数据输入：输入是否要删除front元素，是则输入1，数据类型为整型

测试数据：要删除

运行结果：成功



4.带有注释的源程序

//实验六：队列的定义、创建、插入和删除操作，显示数据元素。

#include<stdio.h>

#define MaxSize 1000//队列的最大容量

#define ElemType int

typedef struct c\_queue

{

ElemType data[MaxSize]; //队列的存储空间

int front, rear; //队列的队头队尾指针

int num; //队列的队中元素个数

}c\_SeQueue;

c\_SeQueue\* Init\_SeQueue()//置空队

{

c\_SeQueue\*q;

q = (c\_SeQueue\*)malloc(sizeof(c\_SeQueue));

q->front = q->rear = MaxSize - 1;

q->num = 0;

return q;

}

int In\_SeQueue(c\_SeQueue \*q, ElemType x)//入队

{

if (q->num == MaxSize)//队满不能入队

{

printf("队满");

return -1;

}

else

{

q->rear = (q->rear + 1) % MaxSize;

q->data[q->rear] = x;

q->num++;

return 1;

} //入队完成

}

int Out\_SeQueue(c\_SeQueue \*q, ElemType \*x)//出队

{

if (q->num == 0)

{

printf("队空");

return -1;

}//队空不能出队

else

{

q->front = (q->front + 1) % MaxSize;

\*x = q->data[q->front];//读出队头元素

q->num--;

return 1;

}//出队完成

}

int main()

{

c\_SeQueue\*q;

q = Init\_SeQueue();//创建

ElemType x;

int flag;

ElemType \*px;

px = &x;

for (;;)

{

printf("要插入元素吗？要则输入1\n");

scanf\_s("%d", &flag, 1);

if (flag == 1)

{

printf("请输入要插入的值\n");

scanf\_s("%d", &x, 1);

flag = In\_SeQueue(q, x);

if (flag == 1)

printf("success\n");

if (flag != 1)

printf("error\n");

}

else break;

}

for (;;)

{

printf("是否要删除front元素？是则输入1\n");

scanf\_s("%d", &flag, 1);

if (flag == 1)

{

flag = Out\_SeQueue(q, px);

if (flag == 1)

printf("success\n");

if (flag != 1)

printf("error\n");

}

else break;

}

printf("所有元素为：\n");

for (;;)

{

if (q->num != 0)

{

flag = Out\_SeQueue(q, px);

printf("%d ", \*px);

}

else break;

}

return 1;

}

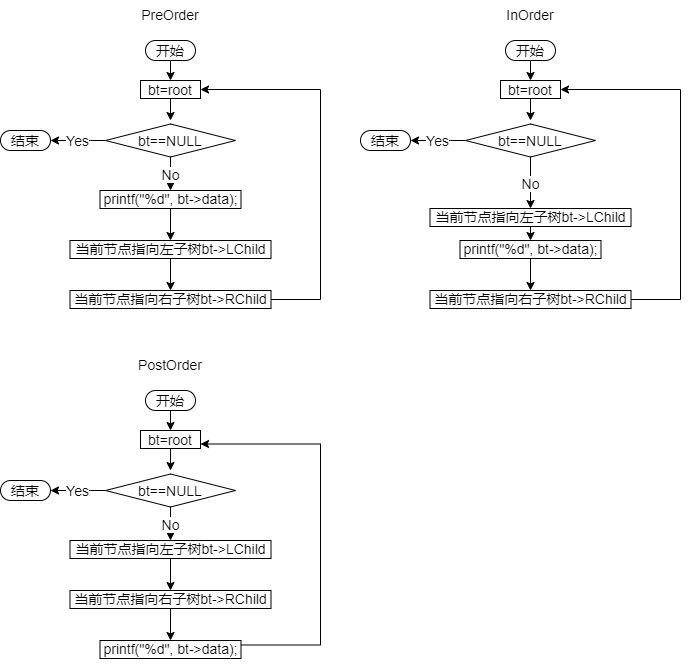
5.实验的收获：

队列实验中遇到的疑惑与链栈、顺序栈情况相似，即出队与删除front元素用的是同一个算法，但区别是如果在循环队列中想要输出所有元素，并不意味着删除所有元素，所以输出所有元素不是一次性的。

**实验七**

1.对象、目的和要求

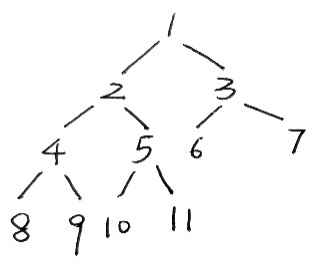
以二叉树的链式存储结构为对象，进行的定义、创建、先序 / 中序 / 后序遍历，并输出结果序列。

2.函数流程图

3.程序的数据输入要求、测试数据与运行结果

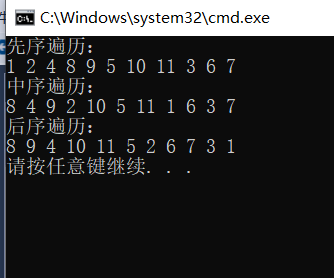
（1）插入操作

数据输入：无数据输入

****测试数据：

二叉树

运行结果：成功



4.带有注释的源程序

//实验七：二叉树的链式存储结构的定义、创建、先序 / 中序 / 后序遍历，并输出结果序列。

#include<stdio.h>

#define MaxSize 1000

#define ElemType int

#define BiTree struct bitree

BiTree

{

ElemType data;

BiTree \*LChild, \*RChild;

};

int Initiate()//建立一棵空二叉树

{

BiTree \*bt;//初始化建立二叉树 \* bt的头结点

if ((bt = (BiTree \*)malloc(sizeof(BiTree))) == NULL)

return 0;

bt->LChild = NULL;

bt->RChild = NULL;

return bt;

}

BiTree \*Create(ElemType x,BiTree \*lbt,BiTree \*rbt)

//建立一棵以x为根结点的数据域信息

{

BiTree \*p;

if ((p = (BiTree \*)malloc(sizeof(BiTree))) == NULL)

return NULL;

p->data = x;

p->LChild = lbt;

p->RChild = rbt;

return p;

}

void PreOrder(BiTree \*bt)

{

//先序遍历二叉树bt

if (bt == NULL)

return; //递归调用的结束条件

printf("%d ", bt->data); //访问结点的数据域

PreOrder(bt->LChild);//先序递归遍历bt的左子树

PreOrder(bt->RChild);//先序递归遍历bt的右子树

}

void InOrder(BiTree \*bt)

{

//中序遍历二叉树bt

if (bt == NULL)

return; //递归调用的结束条件

InOrder(bt->LChild);//中序递归遍历bt的左子树

printf("%d ", bt->data); //访问结点的数据域

InOrder(bt->RChild); //中序递归遍历bt的右子树

}

int PostOrder(BiTree \*bt)

{

//后序遍历二叉树bt

if (bt == NULL)

return 0; //递归调用的结束条件

PostOrder(bt->LChild); //后序递归遍历bt的左子树

PostOrder(bt->RChild); //后序递归遍历bt的右子树

printf("%d ", bt->data); //访问结点的数据域

}

int main()

{

BiTree \*p;

BiTree \*lbt[16] = { NULL };//左子树指针数组

BiTree \*rbt[16] = { NULL };//右子树指针数组

p=Initiate();

int max = 15;//满足max=2max+1

BiTree \*q[16]; //头结点指针数组

int i = max;

int a, b;

for (; i >=3;i=i-2)//循环建立特定二叉树

{

a = i - 1;

b = i;

if (i>=max/2)

{

q[a] = Create(a, lbt[a], rbt[a]);

q[b] = Create(b, lbt[b], rbt[b]);

}

q[i/2] = Create((i/2), q[a], q[b]);

}

p = q[1];

printf("先序遍历：\n");

PreOrder(p);

printf("\n");

printf("中序遍历：\n");

InOrder(p);

printf("\n");

printf("后序遍历：\n");

PostOrder(p);

printf("\n");

return 1;}

5.实验的收获：

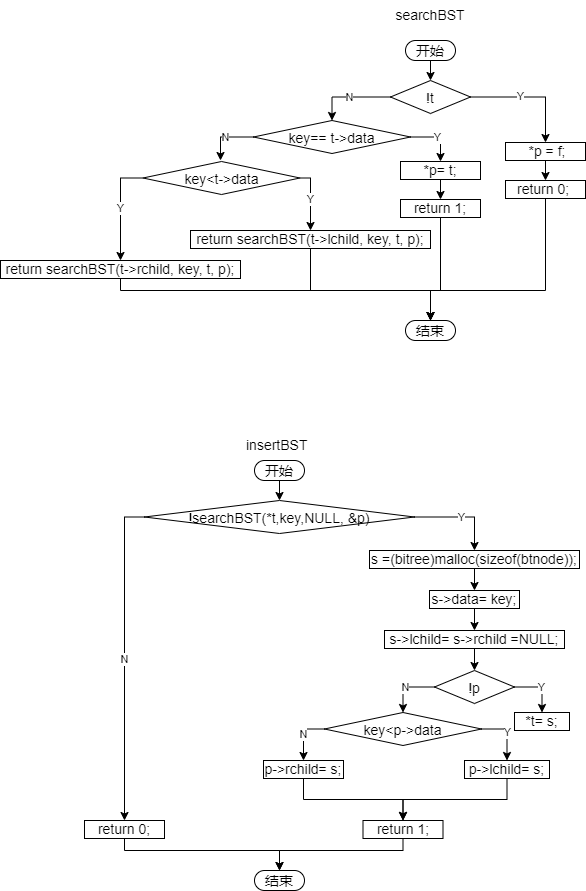
实验中遇到的问题是main函数中，通过循环建立特定二叉树时，该二叉树的头结点指针p最终并非先前调用“p=Initiate();//初始化建立二叉树的头结点”所获得的地址，而是数据为1 的一个新指针；但又发现，头结点的改变并不会导致程序运行失败，并且我和其他同学交流的时候，他们也反映实际上用不到“Initiate()//建立一棵空二叉树”这个函数。

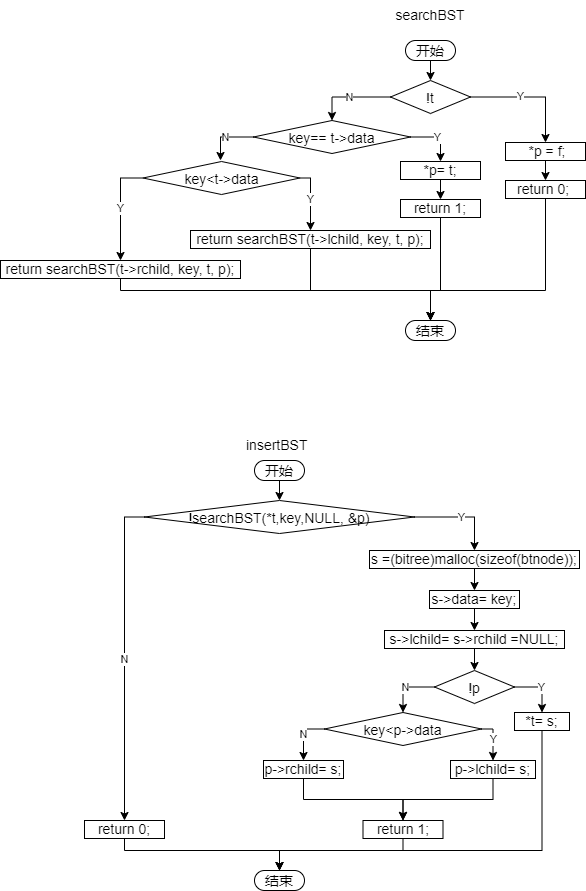
**实验八**

1.对象、目的和要求

以二叉排序树为对象，进行创建和输出。

2.函数流程图



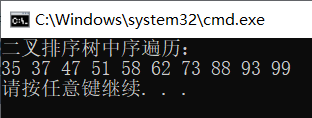


3.程序的数据输入要求、测试数据与运行结果

数据输入：无数据输入

测试数据：a[10] = { 62, 88, 58, 47, 35, 73, 51, 99, 37, 93 };程序将其比较、排序，建立成一个二叉排序树

运行结果：成功



4.带有注释的源程序

//实验八：二叉排序树的创建和输出。

#include<stdio.h>

#include<malloc.h>

typedef int elemtype;

typedef struct btnode

{

elemtype data;//关键字域

struct btnode \*lchild, \*rchild;

}btnode, \*bitree;

int searchBST(bitree t, elemtype key, bitree f, bitree \*p)

//递归查找二叉排序树t是否存在key，指针f指向t的双亲，其初始调用值为NULL，若查找成功,则指针p指向该数据元素结点并返回1，否则指针p指向查找路径上访问的最后一个结点并返回0

{

if (!t)

{ \*p = f;

return 0; }//查找不成功

else if (key == t->data)//查找成功

{

\*p = t;

return 1; }

else if (key<t->data)

return searchBST(t->lchild, key, t, p);//在左子树继续查找

else

return searchBST(t->rchild, key, t, p);//在右子树继续查找}

int insertBST(bitree \*t, elemtype key)

//当二叉排序树T中不存在关键字等于key的数据元素时，插入key返回二叉排序树的根结点。

{

bitree p, s;

if (!searchBST(\*t, key, NULL, &p))//查找不成功

{

s = (bitree)malloc(sizeof(btnode));

s->data = key;

s->lchild = s->rchild = NULL;

if (!p)

\*t = s;//插入s为新的根结点

else if (key< p->data)

p->lchild = s;//插入s为左孩子

else

p->rchild = s;//插入s为右孩子

return 1;

}

else

return 0;

}

void inorder(bitree bt) //中序遍历

{

if (bt == NULL) return;

else{

inorder(bt->lchild);

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

inorder(bt->rchild);

}

}

int main()

{

int i;

bitree f = NULL, p;

int a[10] = { 62, 88, 58, 47, 35, 73, 51, 99, 37, 93 };

bitree t = NULL;

for (i = 0; i<10; i++)

insertBST(&t, a[i]);//插入

printf("二叉排序树中序遍历：\n");

inorder(t);

printf("\n");

return 0;

}

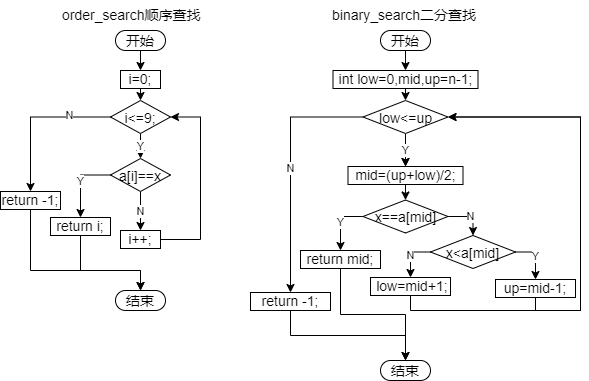
5.实验的收获：

这一实验花费了我很长时间，参考了网络资源才运行成功。首先，我对二叉排序树这一概念不太熟悉；其次，与二叉树不同的是，二叉排序树需要先递归查找二叉排序树是否存在重复元素，并将给定元素排序后再按一定顺序存入存储空间。二叉排序树的算法对我来说比较有挑战性，但是最终解决时也很有成就感。

**实验九**

1.对象、目的和要求以数组为对象，对其中某一元素进行顺序查找、二分查找。

2.函数流程图



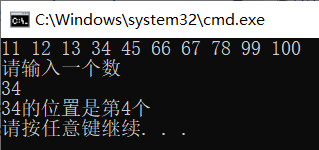
3.程序的数据输入要求、测试数据与运行结果

（1）顺序查找

数据输入：输入要查找的元素x，数据类型为整型

测试数据：a[10]={11,12,13,34,45,66,67,78,99,100};

运行结果：成功

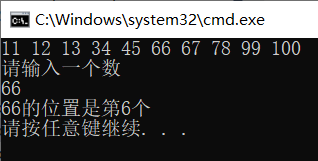


（2）二分查找

数据输入：输入要查找的元素x，数据类型为整型

测试数据：a[10]={11,12,13,34,45,66,67,78,99,100};

运行结果：成功



4.带有注释的源程序

（1）顺序查找

//实验九：顺序查找

#include<stdio.h>

int main ()

{

int order\_search(int a[],int n,int x);

int a[10]={11,12,13,34,45,66,67,78,99,100};

int i=0,x,p;

for(i=0;i<=9;i++)

printf("%d ",a[i]);

printf("\n");

printf("请输入一个数\n");

scanf\_s("%d",&x,1);

p = order\_search(a, 10, x);

printf("%d的位置是第%d个\n", x, p + 1);

return 0;

}

int order\_search(int a[], int n, int x)

{

int i;

for(i=0;i<=9;i++)

if(a[i]==x)//数组中找到了x

return i;

return -1; //数组中没有x

}

（2）二分查找

//实验九：二分查找。

#include<stdio.h>

int main()

{

int binary\_search(int a[],int n,int x);

int a[10]={11,12,13,34,45,66,67,78,99,100};

int i,x,p;

for(i=0;i<=9;i++)

printf("%d ",a[i]);

printf("\n");

printf("请输入一个数\n");

scanf\_s("%d",&x,1);

p = binary\_search(a, 10, x);

printf("%d的位置是第%d个\n", x, p + 1);

return 0;

}

int binary\_search(int a[], int n, int x)

{

int low=0,mid,up=n-1;

while(low<=up)

{

mid=(up+low)/2;

if(x==a[mid])//查找成功

return mid;

else if(x<a[mid])//待查找值位于数组的前一半

up=mid-1;

else

low=mid+1; //待查找值位于数组的后一半

}

return -1;

}

5.实验的收获：

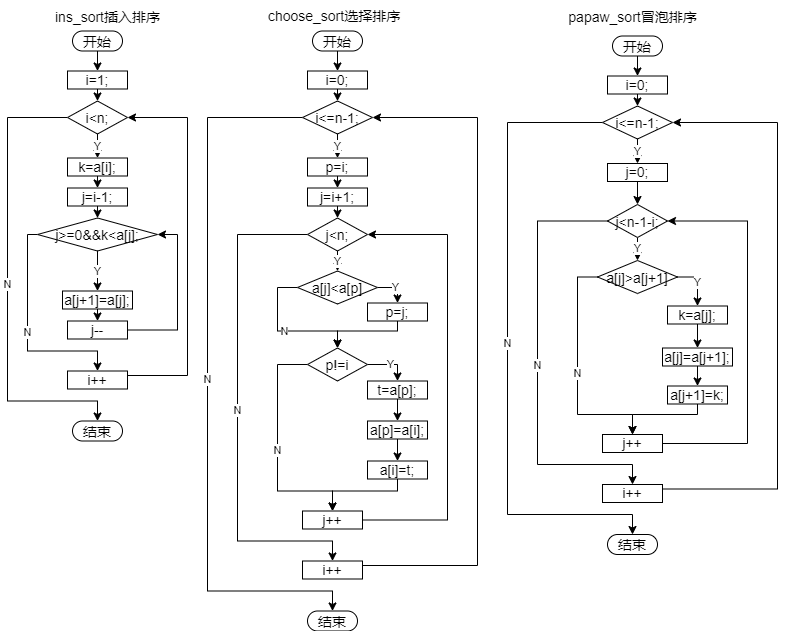
在这次实验中，我基本上掌握了顺序查找、二分查找的基本思想和实现方法，体会到同样的数据在不同的查询方法下有着不同的查询效率，例如用顺序查找法在n个数据中查找一个关键字，需要的查找的次数为(n+1)/2次；但是当线性表采用顺序存储结构，而且表中元素按关键字有序排列时，用二分查找法只要log2n次，由此可以看出，我们在查找时不仅要考虑查找的实现，还要考虑查找的效率和查找所用的时间。

**实验十**

1.对象、目的和要求

以数组为对象，对其中某一元素进行插入排序、选择排序、冒泡排序。

2.函数流程图



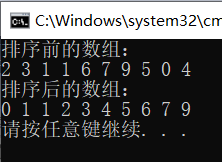
3.程序的数据输入要求、测试数据与运行结果

（1）插入排序

数据输入：无

测试数据：a[10] = { 2, 3, 1, 1, 6, 7, 9, 5, 0 ,4};

运行结果：成功

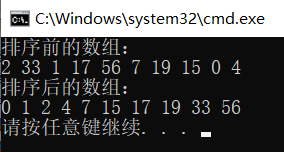


（2）选择排序

数据输入：无

测试数据：a[10] = { 2, 33, 1, 17, 56, 7, 19, 15, 0 ,4};

运行结果：成功

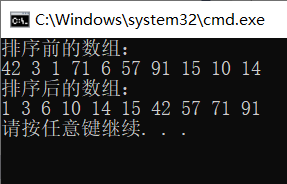


（3）冒泡排序。

数据输入：无

测试数据：a[10]={ 42, 3, 1, 71, 6, 57, 91, 15, 10 ,14};

运行结果：成功



4.带有注释的源程序

（1）插入排序

//实验十：插入排序

#include<stdio.h>

int main()

{

void ins\_sort(int a[],int n);

int a[10] = { 2, 3, 1, 1, 6, 7, 9, 5, 0 ,4};

int i;

printf("排序前的数组：\n");

for(i=0;i<=9;i++)

printf("%d ",a[i]);

printf("\n");

ins\_sort(a,10);

printf("排序后的数组：\n");

for(i=0;i<=9;i++)

printf("%d ",a[i]);

printf("\n");

return 0;

}

void ins\_sort(int a[],int n)//后插法

{

int i,j,k;

for(i=1;i<n;i++)

{

k=a[i];

for(j=i-1;j>=0&&k<a[j];j--)//将比k大的 元素依次右移一个位置

a[j+1]=a[j];

a[j+1]=k;

}

}

（2）选择排序

//实验十：选择排序

#include<stdio.h>

int main()

{

void choose\_sort(int a[],int n);

int a[10] = { 2, 33, 1, 17, 56, 7, 19, 15, 0 ,4};

int i;

printf("排序前的数组：\n");

for(i=0;i<=9;i++)

printf("%d ",a[i]);

printf("\n");

choose\_sort(a, 10);

printf("排序后的数组：\n");

for(i=0;i<=9;i++)

printf("%d ",a[i]);

printf("\n");

return 0;

}

void choose\_sort(int a[], int n)

{

int i,j,p,t;

for(i=0;i<=n-1;i++)

{

p=i;

for(j=i+1;j<n;j++)

if(a[j]<a[p])

p=j;

if(p!=i)

{

t=a[p];

a[p]=a[i];

a[i]=t;

}

}

}

（3）冒泡排序。

//实验十：冒泡排序。

#include<stdio.h>

int main()

{

void papaw\_sort(int a[],int n);

int a[10]={ 42, 3, 1, 71, 6, 57, 91, 15, 10 ,14};

int i;

printf("排序前的数组：\n");

for(i=0;i<=9;i++)

printf("%d ",a[i]);

printf("\n");

papaw\_sort(a, 10);

printf("排序后的数组：\n");

for(i=0;i<=9;i++)

printf("%d ",a[i]);

printf("\n");

return 0;

}

void papaw\_sort(int a[], int n)

{

int i,j,k;

for(i=0;i<=n-1;i++)

for(j=0;j<n-1-i;j++)

if(a[j]>a[j+1])//一旦a[j]大于a[j+1]的值，则交换二者的值

{

k=a[j];

a[j]=a[j+1];

a[j+1]=k;

}

}

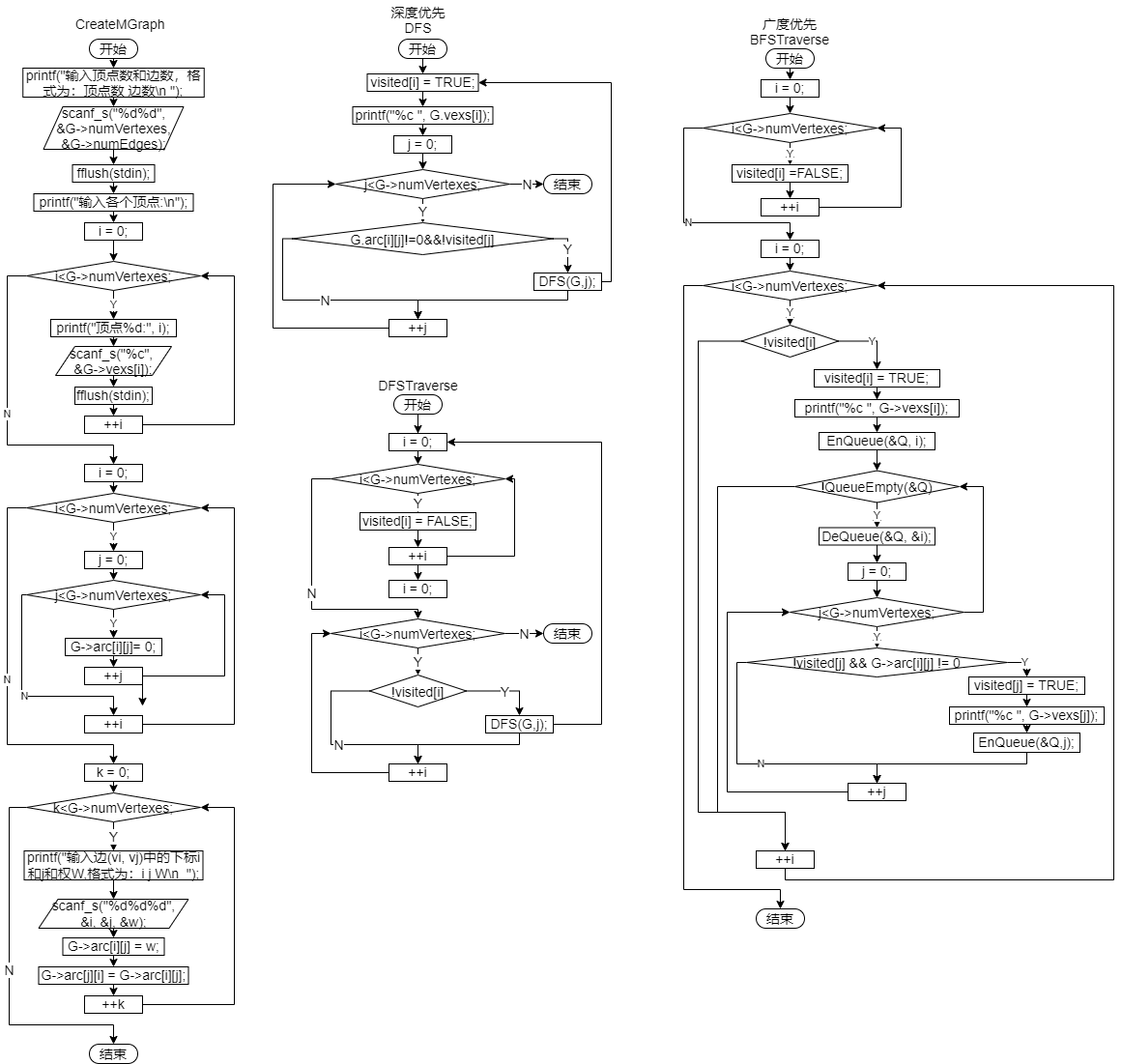
5.实验的收获：

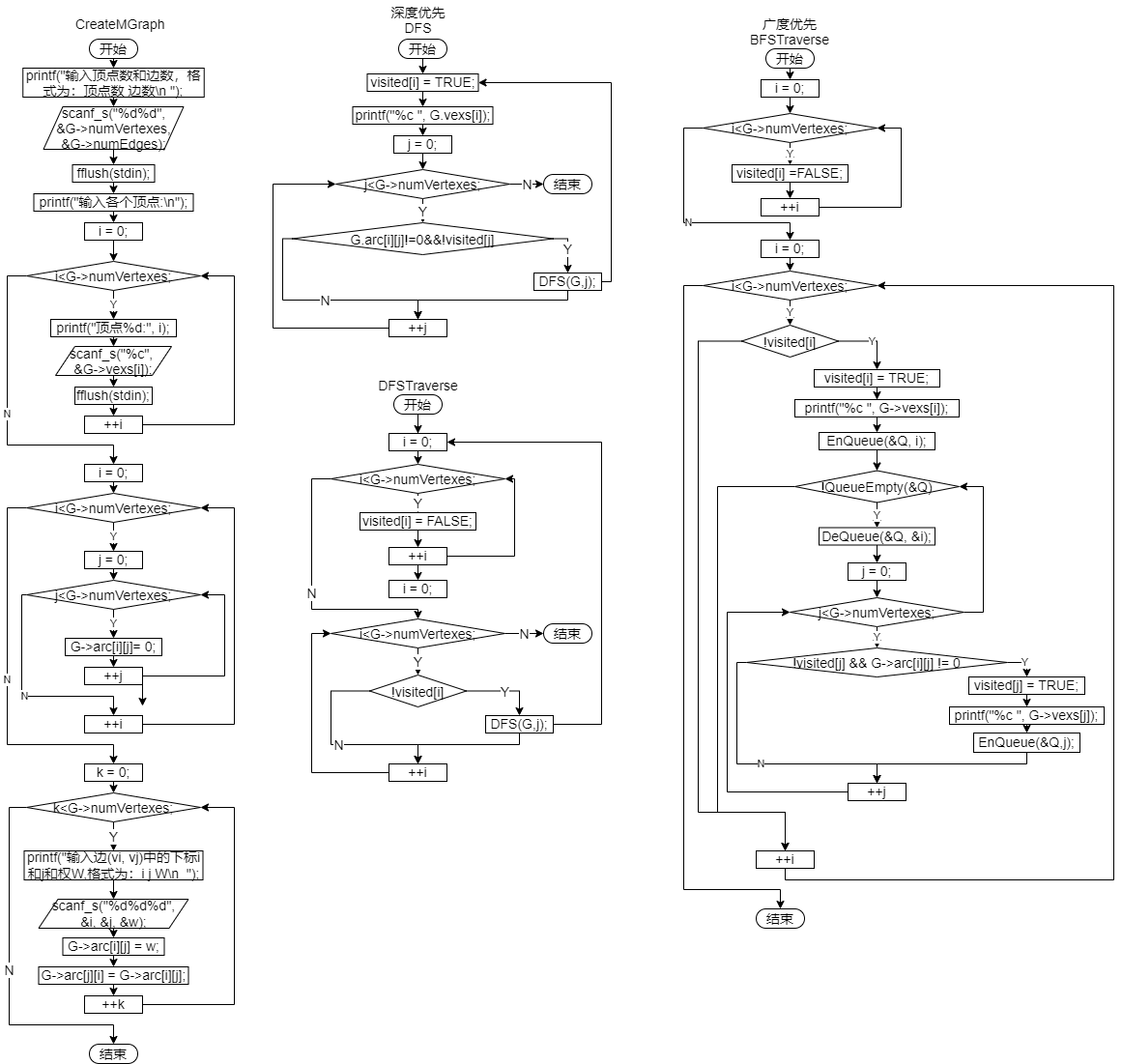
本实验总共使用了三种排序方法，而这三种编程方法放在一起进行编写时，很容易就让我们对其难易程度有了更深刻的了解。首先，对于插入排序，每次都要对一段序列进行检索，每排一次，所要检索的序列长度减一；其次，对于选择排序，思路简单，易于进行；最后，就是冒泡排序，我个人认为它的逻辑是最难理解的，它重复地走访过要排序的元素列，依次比较两个相邻的元素，如果顺序（如从大到小、首字母从Z到A）错误就把他们交换过来。这三个算法的时间复杂度一样，都是O(n²)。

**实验十一**

1.对象、目的和要求以图的邻接矩阵为对象，进行定义、创建，并完成图的深度优先遍历、广度优先遍历。

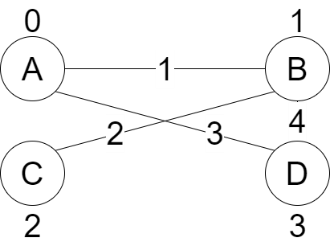
2.函数流程图



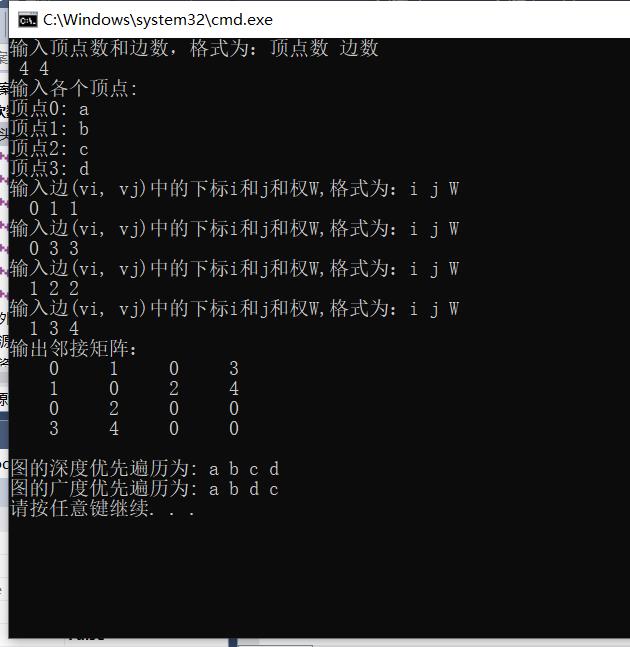


3.程序的数据输入要求、测试数据与运行结果

数据输入：输入顶点数和边数，格式为:顶点数 边数；输入各个顶点；输入边(vi，vj)中的下标i和j和权W，格式为:i j W。其中顶点数和边数、边(vi，vj)中的下标i和j和权W的数据类型都为整型，各个顶点为字符型。

测试数据：

运行结果：成功



4.带有注释的源程序

//实验十一：图的邻接矩阵的定义、创建；图的深度优先遍历、广度优先遍历。

#include <stdio.h>

#define MaxVex 100//最大顶点数

#define TRUE 1

#define FALSE 0

typedef char VertexType;//顶点类型

typedef int EdgeType;//权值类型

typedef int Bool;

Bool visited[MaxVex];

typedef struct

{

VertexType vexs[MaxVex];//顶点数组

EdgeType arc[MaxVex][MaxVex];//邻接矩阵

int numVertexes, numEdges; //当前图中的结点数以及边数

}MGraph;

typedef struct {

int data[MaxVex];

int front, rear;

}Queue;//广度优先遍历需要的循环队列

void InitQueue(Queue \*Q)//队列初始化

{

Q->front = Q->rear = 0;

}

void EnQueue(Queue \*Q, int e)//入队

{

if ((Q->rear + 1) % MaxVex == Q->front)

return;

Q->data[Q->rear] = e;

Q->rear = (Q->rear + 1) % MaxVex;

}

Bool QueueEmpty(Queue \*Q)//判空

{

if (Q->front == Q->rear)

return TRUE;

else

return FALSE;

}

void DeQueue(Queue \*Q, int \*e)//出队

{

if (Q->front == Q->rear)

return;

\*e = Q->data[Q->front];

Q->front = (Q->front + 1) % MaxVex;

}

void CreateMGraph(MGraph \*G)//建立图的邻接矩阵

{

int i, j, k, w;

printf("输入顶点数和边数，格式为：顶点数 边数\n ");

scanf\_s("%d%d", &G->numVertexes, &G->numEdges);

fflush(stdin);

printf("输入各个顶点:\n");

for (i = 0; i<G->numVertexes; ++i)

{

printf("顶点%d: ", i);

scanf\_s("%c", &G->vexs[i]);

fflush(stdin);

}

for (i = 0; i<G->numVertexes; ++i)

for (j = 0; j<G->numVertexes; ++j)

G->arc[i][j] = 0;

for (k = 0; k<G->numEdges; ++k)

{

printf("输入边(vi, vj)中的下标i和j和权W,格式为：i j W\n ");

scanf\_s("%d%d%d", &i, &j, &w);

G->arc[i][j] = w;

G->arc[j][i] = G->arc[i][j];

}

}

void DisMGraph(MGraph \*G)//输出

{

int i, j, k;

k = G->numVertexes;

for (i = 0; i<k; ++i)

{

for (j = 0; j<k; ++j)

{

printf("%5d ", G->arc[i][j]);

}

putchar('\n');

}

}

void DFS(MGraph G, int i)//图的深度优先遍历

{

int j;

visited[i] = TRUE;

printf("%c ", G.vexs[i]);

for (j = 0; j<G.numVertexes; ++j)

if (G.arc[i][j] != 0 && !visited[j])

DFS(G, j);

}

void DFSTraverse(MGraph G)

{

int i;

for (i = 0; i<G.numVertexes; ++i)

visited[i] = FALSE;

for (i = 0; i<G.numVertexes; ++i)

if (!visited[i])

DFS(G, i);

}

void BFSTraverse(MGraph \*G)//图的广度优先遍历

{

int i, j;

Queue Q;

for (i = 0; i<G->numVertexes; ++i)

visited[i] = FALSE;

InitQueue(&Q);

for (i = 0; i<G->numVertexes; ++i)

{

if (!visited[i])

{

visited[i] = TRUE;

printf("%c ", G->vexs[i]);

EnQueue(&Q, i);

while (!QueueEmpty(&Q))

{

DeQueue(&Q, &i);

for (j = 0; j<G->numVertexes; ++j)

{

if (!visited[j] && G->arc[i][j] != 0)

{

visited[j] = TRUE;

printf("%c ", G->vexs[j]);

EnQueue(&Q, j);

}

}

}

}

}

}

int main()

{

MGraph G;

CreateMGraph(&G);

printf("输出邻接矩阵：\n");

DisMGraph(&G);

printf("\n图的深度优先遍历为: ");

DFSTraverse(G);

printf("\n图的广度优先遍历为: ");

BFSTraverse(&G);

printf("\n");

return 0;

}

5.实验的收获：

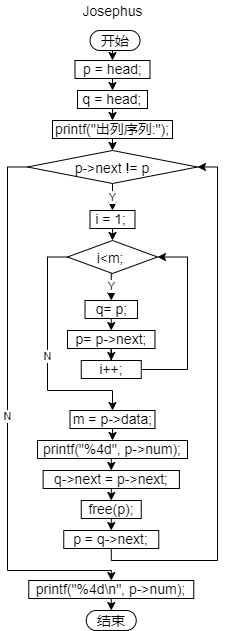
本实验我选做了邻接矩阵储存图、并进行深度优先广度优先处理的部分，程序编译时发现不同的插入方法会影响深度优先、广度优先的运行结果，例如在本实验的测试数据中，当使用头插法时，深度优先的结果会变为adbc。因此，本实验可以要求同学们在学有余力的情况下探究不同插入方法的影响。

**实验十二：约瑟夫斯问题求解**

1.对象、目的和要求

以约瑟夫问题为对象，利用单向循环链表存储结构模拟此过程，按照出列的顺序印出各人的编号。

2.函数流程图

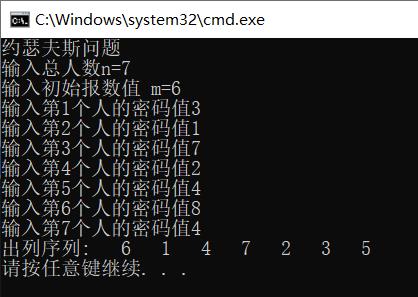


3.程序的数据输入要求、测试数据与运行结果

数据输入：依次输入总人数、初始报数值、第n个人的密码值，数据类型都为整型

测试数据：n=7，7个人的密码依次为：3，1，7，2，4，8，4；m初值为6

运行结果：成功



4.带有注释的源程序

//实验十二：约瑟夫斯问题求解

#include <stdio.h>

#include <malloc.h>

#include <stdlib.h>

#include <time.h>

typedef int ElemType;

typedef struct node

{

ElemType data;

ElemType num;

struct node \*next;

}SLNODE;//单链表的定义

struct node \*create\_sl(int n)

{

SLNODE\*p, \*s, \*head; ElemType x;

int a;

head = (SLNODE \*)malloc(sizeof(SLNODE));

p = head;

head->next = head;

for (a = 1; a <= n; a++)//循环直到输入n个密码值跳出循环

{

s = (SLNODE \*)malloc(sizeof(SLNODE));

printf("输入第%d个人的密码值", a);

scanf\_s("%d", &x, 1);

s->data = x;

s->num = a;

if (head->next == head)

head = s;

else

p->next = s;

p = s;

}

p->next = head;

return head;

}

void Josephus(SLNODE \*head, int n, int m)//约瑟夫斯问题求解

{

SLNODE \*p;

SLNODE \*q;

int i;

p = head;

q = head;

printf("出列序列:");

while (p->next != p)

{

for (i = 1; i<m; i++)//程序运行到第m个人跳出循环

{

q = p;

p = p->next;

}

m = p->data;//读取新的密码值

printf("%4d", p->num);

q->next = p->next;//删除p节点

free(p);

p = q->next;

}

printf("%4d \n", p->num);

}

int main()//主函数

{

SLNODE \*head;

int n, m;

head = (SLNODE \*)malloc(sizeof(SLNODE));

printf("约瑟夫斯问题\n");

printf("输入总人数n=");

scanf\_s("%d", &n, 1);

printf("输入初始报数值 m=");

scanf\_s("%d", &m, 1);

head = create\_sl(n);//创建单链表Josephus(head, n, m);

Josephus(head,n,m);

return 0;}

5.实验的收获：

实验中遇到的问题是，在vs2013中编译时，程序总是报错约瑟夫斯问题求解函数Josephus()中变量q未赋初值，但是在for循环中“q = p;”语句已经给q赋了初值。反复检查之后没有发现逻辑上的错误，增加了“q = head;”语句后不再报错，但是不了解最初报错的缘由。

**实验十四：内部排序算法的实现与比较**

1.对象、目的和要求

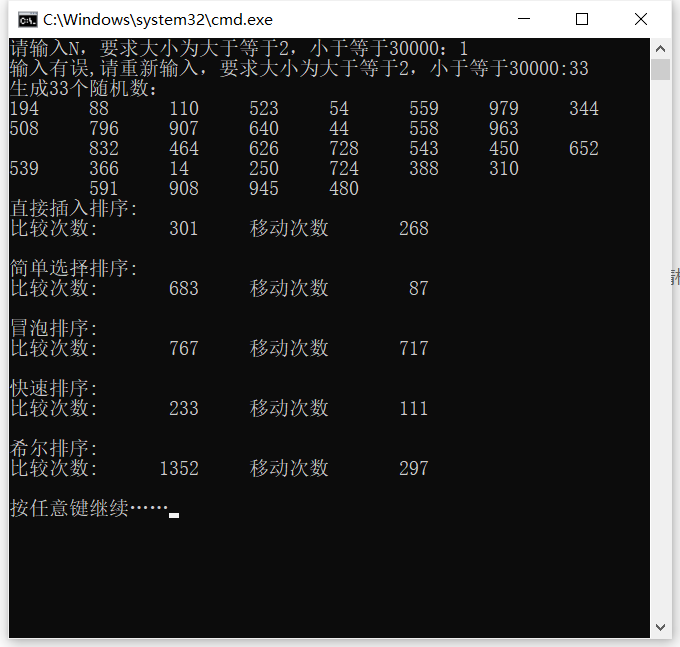
对常用的内部排序算法进行比较：直接插入排序、简单选择排序、冒泡排序、快速排序、希尔排序，比较关键字参加的比较次数和关键字的移动次数（关键字交换计为3次移动）， 对结果作出简要分析。

2.程序的数据输入要求、测试数据与运行结果

数据输入：输入参加排序的整数个数N，要求N大于等于2，小于等于30000，数据类型都为整型

测试数据：N=1，N=33

运行结果：成功



3.带有注释的源程序

//实验十四：内部排序算法的实现与比较

# include <stdio.h>

# include <stdlib.h>

# include <time.h>

# define N 30000

# define SR 1001

int A[N], B[N], C[N], D[N], E[N], F[N];

int a, aa, b, bb, d, dd, e, ee, num;

//单个字母的为记录的比较次数-移动的次数，双字母为关键字的移动次数

long long c, cc;

void charu(int A[], int n);//对n个元素直接插入排序

void xuanzepai(int A[], int n);//对n个元素选择升序

void maopao(int A[], int n); //对n个元素冒泡排序

void Quicksort(int A[], int L, int R);//对[L,R]个元素快速升序

void shell(int A[], int n);//对n个元素shell升序

void GBPX(int S[], int L, int R, int T[]);//归并排序

int GB(int S[], int L, int M, int R, int T[]);//数组归并操作

int gainint(int \*p, int min, int max);//防输错N而建的函数，\*p的范围[min,max]

int change(int \*a, int \*b);//交换函数,交换a b的值

void charu(int A[], int n)//返回循环比较的次数

{

int i, j, temp;

for (i = 1, a++; i<n; i++, a++)

if (A[i]<A[i - 1])

{

temp = A[i];

for (a++, aa++, j = i - 1; j >= 0 && A[j]>temp; j--, a++, aa++)

A[j + 1] = A[j];

A[j + 1] = temp;

}

}

void xuanzepai(int A[], int n)//以A[0]为比较依据 升序

{

int i, j, k;

for (i = 0, b++; i<n - 1; i++, b++)

{

k = i;

for (j = i + 1, b++; j<n; j++, b++)

if (A[j]<A[k])

{

k = j;

b++;

}

if (k != i)

bb += change(&A[i], &A[k]);

}

}

void maopao(int A[], int n)

{

int i, j;

for (i = n - 1; i >= 0; i--)

{

for (j = 0; j<i; j++)

{

if (A[j]>A[j + 1])

cc += change(&A[j], &A[j + 1]);

}

}

}

void Quicksort(int A[], int L, int R)//快速排序，升序

{

int i = L, j = R, T = A[L]; //T为基准数

if (L>R)

return;

while (i != j) //当数组左右两边没有相遇

{

while (A[j] >= T&&i<j){ j--; d++; } //从右向左找

while (A[i] <= T&&i<j){ i++; d++; } //从左向右找

if (i<j)

dd += change(&A[i], &A[j]); //交换两数

}

if (L != i)

dd += change(&A[L], &A[i]); //基准数归位

Quicksort(A, L, i - 1); //递归左

Quicksort(A, i + 1, R); //递归右

}

void shell(int A[], int n)//希尔排序，增量为2

{

int i, j, k;

for (k = n >> 1, e++; k>0; k = k >> 1, e++)

{

for (i = k, e++; i<n; i++, e++)

{

for (j = i - k, e++; j >= 0; j -= k, e++)

if (A[j]>A[j + k])

ee += change(&A[j], &A[j + k]);

}

}

}

int gainint(int \*p, int min, int max)//输入int \*p直至满足(a,b)输入结束，并返回\*p的位数

{

do{

\*p = min - 1; //此处是为了减少意外情况的发生 虽然那种意外情况不常见

scanf\_s("%d", p,1);

while (getchar() != '\n');

if (\*p>max || \*p<min)

printf("输入有误,请重新输入，要求大小为大于等于%d，小于等于%d:", min, max);

} while (\*p>max || \*p<min);

return \*p;

}

int change(int \*a, int \*b)//交换函数，交换a b的值

{

int c = \*a;

\*a = \*b;

\*b = c;

return 3;

}

int main(){

int i, t;

srand(time(0));

printf("请输入N，要求大小为大于等于2，小于等于%d：", N);

gainint(&num, 2, N);

printf("生成%d个随机数：\n",num);

for (i = 0; i<num; i++)

printf("%d\t", A[i] = B[i] = C[i] = D[i] = E[i] = F[i] = rand() % SR);

charu(A, num);

printf("\n直接插入排序:\n比较次数:%10d\t移动次数%10d\n\n", a, aa);

xuanzepai(B, num);

printf("简单选择排序:\n比较次数:%10d\t移动次数%10d\n\n", b + bb / 3, bb);

c = (num - 1)\*num / 2;

maopao(C, num);

printf("冒泡排序:\n比较次数:%10lld\t移动次数%10lld\n\n", c + cc / 3, cc);

Quicksort(D, 0, num - 1);

printf("快速排序:\n比较次数:%10d\t移动次数%10d\n\n", d + dd / 3, dd);

shell(E, num);

printf("希尔排序:\n比较次数:%10d\t移动次数%10d\n\n", e + ee / 3, ee);

printf("按任意键继续……");

getchar();

return 0;

}

4.实验的收获：

实验中，一开始想制作一个小型系统，可以在初始界面选择不同排序算法，并显示其关键字比较次数和关键字移动次数，但是由于这样对不同算法比较不够直观、实验调试时间过长，便放弃了这一想法。建议本实验可以要求学有余力的同学进一步完善制作系统。

此外，我对快速排序、希尔排序的算法不太熟悉，漏掉一些关键语句，如快速排序中的“if (L != i) dd += change(&A[L], &A[i]); //基准数归位”，在学习了网络资源才完成了这个实验。