1.设计对循环链表实现如下算法：

（1） 合并两个表

（2） 两表的交运算

（3） 删去第 i 个结点的前驱结点

（4） 把第 i 个结点向前移动 n 个位置

思路：单循环链表优点便是连接 2 个链表容易，一般只要知道尾指针；

LinkList l1,l2；LinList head1=l1->tail->next；l1->tail->next=l2->tail->next；l2->tail->next=head1；

第二小问的算法应是把 LA 中的那些在 LB 中不存在的元素删除。

第三小问得先找到第 i-1 个结点，并将之前驱与后继相连。

最后一问得先将 P 指向第 i 个结点前 n 个位置结点的前驱，再用基本删除操作处理即可。

典型错误：没有考虑到若初试链表为空的情况，如 1 问中的，应加入 If(L1=NULL) return （L2）等；对于交运算，没有考虑到完整的思路，即没有想到是把 LA 中的那些在 LB 中不存在的元素删除。3 问中没有考虑到应当先遍历到第 i-1 个结点，之后进行的是基本的删除结点操作，对于基础题应加强练习；最后一问，很多同学没有考虑到当 i-n<0 时会出现错误的情况，首先应当要说明这一点。

参考代码：

```
（1）void link(linklist &L1，linklist &L2)
    {      p=L1;   if (L1=NULL) return (L2);
       While (p—>next!=L1) p=p—>next;//p 指向 L1 表尾
         q=L2;   if (L2=NULL) return (L1);
      While (q—>next!=L1) q=q—>next;
       p—>next=L2;
       q—>next=L1;
```

```
        }
```

(2) void intersection(LinkList L,LinkList M)

```
{
        int i=0,j=0;
        p=L->next;
        s=L->next;
        q=M->next;
        while(p!=L)
        {
                while(q->next!=M)
                {
                        if(p->data==q->data)
                        {
                                i++;
                                s->data=p->data;
                                s=s->next;
                        }
                        q=q->next;
                }
                q=M->next;
                p=p->next;
        }
        s=L;
        for(j=0;j<i;j++)
        {
                s=s->next;
        }
        s->next=L;
}
```

(3) void delprev (linklist &L)

```
 {   p=L; int j, r ;

     for(j=1;j<i%L.length;j++)   { r=p;p=p->next;}

     r->next=p->next;

     free(p);
 }
```

```
(4)void Elemmove ( linklist L)
{              link *p,*q,*r;
              if(i-n<0) return 0;
              else
               {     p=L->next; for(int j=0;j<i-n-1;j++) p= p->next;
                     For(q=0;j<i-1;j++)q=q->next;
                     r=q->next; q->next=r->next;
                     r->next=p->next;
                     p->next=r;
               }
}
```

2.设 L 为顺序存储结构线性表，删除其中所有的零元素。

思路：主要考虑到删除 0 元素后，非 0 元素向前移动的问题，因此，设立一个标记 j,在遍历线性表过程中记录 0 元素的个数，每当遇到非 0 元素时，则用 L.data[i-j]=L.data[i]即可，最后记得将表长修改为 L.length=L.length-j.

典型错误：仍然是思路不清，考虑不全面，有的同学写的将非 0 元素移动的算法经不起推敲，两个工作指针没有一致移动；应该记录下 0 元素的个数，最终的表长变成 L.length=L.length-j，这也是很多同学所忽略的。

参考代码：

```
    Void delete_0(L)
{       int i=0;j=0;
        While (i<L.length)
        {       if (L.data[i]=0) j++;
                Else    L.data[i-j]=L.data[i];i++;
         }
        L.length=L.length-j;
}
```

3. 写出如下中缀表达式的后缀形式：

(1)A*B*C；

(2)-A+B-C+D；

(3)(A+B)*D+E/(F+A*D)+C；

比较简单，有兴趣的同学可以写一个算法，将中缀表达式转换成后缀形式，这里给出算法思想，可以加 ssxbily@163.com 与我讨论。

将中缀表达式转换为后缀表达式的算法步骤：

(1)设置一个运算符栈，初始时将栈顶运算符置为"#"；

(2)按顺序扫描中缀表达式，当输入为操作数时就将其输出到后缀表达式中。

(3)当输入为运算符时，则比较输入运算符和栈顶元素的优先级。若输入运算符的优先级高于栈顶元素的优先级，则将输入运算符入栈。否则，栈顶运算符的优先级高于或者等于输入运算符的优先级，弹出栈顶运算符至后缀表达式，然后重新比较输入运算符和更新后的栈顶运算符的优先级。

(4)当输入运算符为(时，直接将(入栈

(5)当输入运算符为")"时，将栈顶运算符出栈至后缀表达式，直至栈顶为(时，将(出栈并抛弃，同时)也抛弃

(6)中缀表达式扫描完毕后，将运算符依次出栈至后缀表达式直至栈顶为#时停止。

答案：（1）AB*C*

（2）A-B+C-D+

（3）AB+D*EFAD*+/+C+

4. **设用循环链表表示队列，给出进队，出队，判断队空的算法**

思路：需要设立一个尾指针指向队尾元素结点，接着按基本操作来，注意是循环链表。

典型错误：有的同学忽略了插入前需要“判满”，同时也忽略了这里是用**循环链表**表示队列，别的都没什么问题。

参考代码：

```
void InitCiQueue(CiQueue &Q)//初始化循环链表表示的队列 Q
{
  Q=(CiLNode*)malloc(sizeof(CiLNode));
  Q->next=Q;
}//InitCiQueue

void EnCiQueue(CiQueue &Q,int x)//把元素 x 插入循环链表表示的队列 Q,Q 指向队尾元素,Q->next 指向头结点,Q->next->next 指向队头元素
{
  p=(CiLNode*)malloc(sizeof(CiLNode));
  p->data=x;
  p->next=Q->next; //直接把 p 加在 Q 的后面
  Q->next=p;
  Q=p;  //修改尾指针
}

Status DeCiQueue(CiQueue &Q,int x)//从循环链表表示的队列 Q 头部删除元素 x
{
  if(Q==Q->next) return INFEASIBLE; //队列已空
  p=Q->next->next;
  x=p->data;
  Q->next->next=p->next;
  free(p);
```

```
  return OK;
}//DeCiQueue
```

5. 编写用两个栈实现队列的算法

思路：栈的特点是后进先出，队列的特点是先进先出。所以，用两个栈 s1 和 s2 模拟一个队列时，s1 作输入栈，逐个元素压栈，以此模拟队列元素的入队。当需要出队时，将栈 s1 退栈并逐个压入栈 s2 中，s1 中最先入栈的元素，在 s2 中处于栈顶。s2 退栈，相当于队列的出队，实现了先进先出。显然，只有栈 s2 为空且 s1 也为空，才算是队列空。

典型错误：基本都没做出来，做出来的考虑也不够全面，需要注意的是出队从栈 s2 出，当 s2 为空时，若 s1 不空，则将 s1 倒入 s2 再出栈。入队在 s1，当 s1 满后，若 s2 空，则将 s1 倒入 s2，之后再入队。因此队列的容量为两栈容量之和。元素从栈 s1 倒入 s2，必须在 s2 空的情况下才能进行，即在要求出队操作时，若 s2 空，则不论 s1 元素多少（只要不空），就要全部倒入 s2 中。

参考代码：

```
#include <stdlib.h>
#include <stdio.h>

typedef struct
{
   char *base;
   char *top;
   int stack_size;
}stack;
```

```
void init_stack(stack *s)
{
   s->base=(char *)malloc(50*sizeof(char));
   if(!s->base)return;

   s->top=s->base;
   s->stack_size=50;
}

void push(stack *s,char e)
{
   if(s->top-s->base>=s->stack_size)
   {
     s->base=(char *)realloc(s->base,(s->stack_size+50)*sizeof(char));
     if(!s->base)return;
     s->top=s->base+s->stack_size;
     s->stack_size+=50;
   }

   *(s->top)=e;
   s->top++;

}

void pop(stack *s,char *e)
{
   s->top--;
   *e=*(s->top);

}
int stack_empty(stack *s)
{
    if(s->top==s->base)return 1;
    return 0;
}
int queue_empty(stack *s1,stack *s2)
{
    if(stack_empty(s1)&&stack_empty(s2))return 1;
    return 0;
}
void dequeue(stack *s1,stack *s2,char *a)     /*s1 负责入队，s2 负责出队*/
{                                    /*出队之前  */
    char e;                            /*先把 s1 倒灌到 s2 里面*/
    if(!queue_empty(s1,s2))                /*这样把最先入队的暴露在最外面*/
    {
```

```
        while(!stack_empty(s1))
        {
            pop(s1,&e);
            push(s2,e);
        }
        pop(s2,a);
    }
}

void enqueue(stack *s1,stack *s2,char a)
{
    char e;
    while(!stack_empty(s2))
    {
        pop(s2,&e);
        push(s1,e);
    }
    push(s1,a);
}

main()
{
   char e,*a="good good study day day up";
   int i=0;
   stack s1,s2;
   init_stack(&s1);
   init_stack(&s2);

   while(a[i])
   {
      enqueue(&s1,&s2,a[i]);
      i++;
   }
   while(!queue_empty(&s1,&s2))
   {
      dequeue(&s1,&s2,&e);
      printf("%c",e);
   }

   getch();
}
```

6. 编写一个判断表达式中“（ ）”、“[ ]”是否配对的算法。

算法思想：判断表达式中括号是否匹配，可通过栈，简单说是左括号时进栈，右括号时退栈。退栈时，若栈顶元素是左括号，则新读入的右括号与栈顶左括号就可消去。如此下去，输入表达式结束时，栈为空则正确，否则括号不匹配。

算法：这里设表达式以字符形式已存入数组E[n]中，‘#’为表达式的结束符

```
 int EXYX (char E[]){
 /*E[]存放字符串表达式，以‘#’结束*/
 char s[30];                /*s是一维数组，容量足够大，用作存放括号的栈*/
 int top=0,i;                /*top用作栈顶指针*/
 s[top]='#';            /*‘#’先入栈，用于和表达式结束符号‘#’匹配*/
 i=0;                    /*字符数组E的工作指针*/
 while(E[i]!='#')    /*逐字符处理字符表达式的数组*/
  switch (E[i])
    {case '(': s[++top]= '('; i++ ;    break ;
     case ')':    if(s[top]=='('){top--; i++; break;}
                        else{printf("括号不配对");exit(0);}
     case '#':    if(s[top]=='#'){printf("括号配对\n");return (1);}
                        else {printf("  括号不配对\n");return (0);} /*括号不配对*/
     default :    i++;       /*读入其它字符，不作处理*/
    }
}/*算法结束*/
```