1. 若按教科书 3.1.1 节中图 3.1(b)所示铁道进行车厢调度（注意：两侧铁道均为单向行驶道），则请回答：
   (1) 如果进站的车厢序列为 123，则可能得到的出站车厢序列是什么？
   (2) 如果进站的车厢序列为 123456，则能否得到 435612 和 135426 的出站序列，并请为什么不能得到或者如何得到？（即写出以‘S’表示进栈和‘X’表示出栈的栈操作序列）。

解：

(1) 123,132,213,231,321,

123 对应的操作序列为：SXSXSX；132 对应的操作序列为：SXSSXX；213 对应的操作序列为：SSXXSX；231 对应的操作序列为：SSXSXX；321 对应的操作序列为：SSSXXX；

(2) 不能得到 435612，因为，首先出站的车厢为 4 意味着 123 号车厢已经入栈，则这两节车厢出站的相对顺序一定是 3,2,1，而该序列中，1 号车厢在 2 号车厢之前出站，这不符合操作规则。

能得到 135426，操作序列为：SXSSXSSXXXSX

2. 写出下列程序段的输出结果（栈的元素类型 SElemType 为 char）

```
void main(){
  Stack S;
  char   x, y;

  InitStack(S);
  x='c'; y='k';
  Push(S, x); Push(S, 'a'); Push(S, y);
  Pop(S, x); Push(S, 't'); Push(S, x);
  Pop(S, x); Push(S, 's');
  while(!StackEmpty(S)){Pop(S, y); printf(y);}
  printf(x);
}
```

解：

根据操作序列，栈 S 为：

(1) InitStack(S): S()

(2) Push(S, x); Push(S, 'a'); Push(S, y)：S(‘c’,’a’,’k’)

(3) Pop(S, x); Push(S, 't'); Push(S, x)：x=’k’, S(‘c’,’a’,’t’,’k’)

(4) Pop(S, x); Push(S, 's'): x=’k’, S(‘c’,’a’,’t’,’s’)

(5) while(!StackEmpty(S)){Pop(S, y); printf(y);} printf(x);输出：stack

所以：输出结果为：stack

3. 按照四则运算加减乘除和幂运算（^）优先关系的惯例，并仿照教科书 3.2 节例 3-2 的格式，画出对下列算数表达式求值时操作数栈和运算符栈的变化过程。

A-B×C/D+E^F

序号	OPTR	OPND	当前字符	备注
1	#		A-B×C/D+E^F	push(OPND,’A’)
2	#	A	A-B×C/D+E^F	push(OPTR,’-’)
3	#-	A	A-B×C/D+E^F	push(OPND,’B’)

4	#-	AB	A-B×C/D+E^F	push(OPTR,'×')
5	#-×	AB	A-B×C/D+E^F	push(OPND,'C')

6	#-×	ABC	A-B×C/D+E^F	pop(ORND,right)//C
				pop(OPND,left)//B
				pop(OPTR,optr)//×
				T1=exe(left,optr,right)
				//T1=B×C
				push(OPND,T1)
				push(OPTR,'/')

7	#-/	AT1	A-B×C/D+E^F	push(OPND,'D')

8	#-/	AT1D	A-B×C/D+E^F	pop(ORND,right)//D
				pop(OPND,left)//T1
				pop(OPTR,optr)//除/
				T2=exe(left,optr,right)
				//T2=T1/D
				push(OPND,T2)
				pop(ORND,right)//T2
				pop(OPND,left)//A
				pop(OPTR,optr)//-
				T3=exe(left,optr,right)
				//T3=A-T2
				push(OPND,T3)
				push(OPTR,'+')

9	#+	T3	A-B×C/D+E^F	push(OPND,'E')
10	#+	T3E	A-B×C/D+E^F	push(OPTR,'^')
11	#+^	T3E	A-B×C/D+E^F	push(OPND,'F')

11	#+^	T3EF	A-B×C/D+E^F	pop(ORND,right)//F
				pop(OPND,left)//E
				pop(OPTR,optr)//^
				T4=exe(left,optr,right)
				//T4=E^F
				push(OPND,T4)
				pop(ORND,right)//T4
				pop(OPND,left)//T3
				pop(OPTR,optr)//+
				T5=exe(left,optr,right)
				//T5=T3+T4

4. 试推导求解 n 阶汉诺塔问题至少要执行的 move 操作次数。

解：设a_i表示 i 阶汉诺塔问题至少要执行的 move 操作次数，则可知：$a_i = 2a_{i-1} + 1$，且$a_1 = 1$，可推导：

$$a_1 = 1$$

$$a_2 = 2a_1 + 1 = 2 + 1$$

$$a_3 = 2a_2 + 1 = 2^2 + 2^1 + 2^0$$

...

$$a_i = \sum_{k=0}^{i-1} 2^k = \frac{2^i - 1}{2 - 1} = 2^i - 1$$

所以，n 阶汉诺塔问题至少要执行的 move 操作次数为$2^n - 1$

5. 假如以顺序存储结构实现一个双向栈，即在一维数组的存储空间中存在着两个栈，它们的栈底分别设在数组的两个端点。试编写实现这个双向栈 tws 的三个操作：初始化 `Status InitStack(tws *t)`，入栈 `Status Push(tws *t, int i, SElemType x)`和出栈 `Status Pop(tws *t, int i, SElemType *x)`算法，其中 i 为 0 或 1，用以分别指示在数组两端的两个栈。

解：

```
#define STACK_INIT_SIZE (4)
#define STACK_INCREAMENT (10)
#define OK (0)
#define ERROR (1)
#define ERROR_OVERFLOW (2)
#define ERROR_EMPTY (3)
#define NULL (0)

typedef int SElemType;
typedef int Status;

typedef struct _tws
{/*顺序存储结构实现的双向栈*/
    SElemType *base;//数组基地址
    SElemType *top[2];//栈顶指针
    int stacksize;  //最大容量，按元素个数计数
}tws;

Status InitStack(tws *t) {
    t->base = (SElemType *)malloc(STACK_INIT_SIZE * sizeof(SElemType));
    if (!t->base) return ERROR_OVERFLOW;
    t->stacksize = STACK_INIT_SIZE;
    t->top[0] = t->base;//栈0的栈底和栈顶指针
    t->top[1] = t->base + STACK_INIT_SIZE-1;//栈1的栈底和栈顶指针
    return OK;
}

Status Push(tws *t, int i, SElemType x) {
    int step = (i == 0) ? 1 : -1;//0号栈，栈顶指针向数组末端移动；1号栈，栈顶指针向数组起始方向移动

    if (t->top[0] > t->top[1])//栈满了
```

```
        return ERROR_OVERFLOW;
    *(t->top[i]) = x;
    t->top[i] += step;
    return OK;
}

Status Pop(tws *t, int i, SElemType *x) {
    SElemType *base = (i == 0) ? t->base : t->base + t->stacksize - 1;//0号栈，栈底为数组头；1号栈，栈底为数组尾
    int step = (i == 0) ? 1 : -1;//0号栈，栈顶指针向数组末端移动；1号栈，栈顶指针向数组起始方向移动

    if (t->top[i] == base)//栈空了
        return ERROR_EMPTY;
    t->top[i] -= step;
    *x = *(t->top[i]);
    return OK;
}

Status DestroyStack(tws *t) {
    if (!t->base) return ERROR;
    free(t->base);
    t->top[0] = t->top[1] = NULL;
    return OK;
}
```

6. 编写算法删除单链表中"多余"的数据元素，即使操作之后的单链表中所有元素的值都不相同。

```
void purge_L(LinkList &L )
  { // 删除单链表 L 中的冗余元素，即使操作之后的单链表中只保留
    // 操作之前表中所有值都不相同的元素
    p = L->next;      L->next = NULL;   // 设新表为空表
    while ( p )                    // 顺序考察原表中每个元素
    {    succ = p->next;             // 记下结点 *p 的后继
      q = L->next;                 // q 指向新表的第一个结点
      while( q && p->data!=q->data )
            q = q->next;   //在新表中查询是否存在与 p->data 相同的元素
      if ( !q ){                   // 将结点 *p 插入到新的表中
              p->next = L->next;L->next = p;}
      else delete p;           // 释放结点 *p
      p = succ;
    } // for
  } // purge_L
```

```
Status LinkListInsert(LinkList &L){
    //新链表
    p = L->next; L->next = NULL;
    while(p){
        q = L->next; pre = L;
        //移除一个元素，若在新链表中不存在，则插入，否则删除
        while(q && q->data != p->data){
            pre = q; q = q->next;
        }
        if(q){//q->data == p->data，应该删除该节点
            q = p; p = p->next; free(q);
        }else{//应该插入该节点
            pre->next = p; p = p->next; pre->next->next= NULL;
        }
    }
    return OK;
}
```