第2章：线性数据结构

1、实现顺序栈的判空操作

```
/**********
【题目】试写一算法，实现顺序栈的判空操作
StackEmpty_Sq(SqStack S)。
顺序栈的类型定义为：
typedef struct {
  ElemType *elem; // 存储空间的基址
  int top;        // 栈顶元素的下一个位置，简称栈顶位标
  int size;       // 当前分配的存储容量
  int increment;  // 扩容时，增加的存储容量
} SqStack;        // 顺序栈
***********/
Status StackEmpty_Sq(SqStack S)
/* 对顺序栈S判空。                          */
/* 若S是空栈，则返回TRUE；否则返回FALSE */
{
    if(S.top != 0) return FALSE;
    else return TRUE;
}
```

2、实现顺序栈的取栈顶元素操作

```
/**********
【题目】试写一算法，实现顺序栈的取栈顶元素操作
GetTop_Sq(SqStack S, ElemType &e)。
顺序栈的类型定义为：
typedef struct {
  ElemType *elem;  // 存储空间的基址
  int top;         // 栈顶元素的下一个位置，简称栈顶位标
  int size;        // 当前分配的存储容量
  int increment;   // 扩容时，增加的存储容量
} SqStack;         // 顺序栈
***********/
Status GetTop_Sq(SqStack S, ElemType &e)
/* 取顺序栈S的栈顶元素到e，并返回OK；  */
/* 若失败，则返回ERROR。               */
{
   if(S.top == 0) return ERROR;
   e = S.elem[S.top - 1];
   return OK;
}
```

3、实现顺序栈的出栈操作

```
/**********
【题目】试写一算法，实现顺序栈的出栈操作
Pop_Sq(SqStack &S, ElemType &e)。
顺序栈的类型定义为：
```

```
typedef struct {
  ElemType *elem;  // 存储空间的基址
  int top;         // 栈顶元素的下一个位置，简称栈顶位标
  int size;        // 当前分配的存储容量
  int increment;   // 扩容时，增加的存储容量
} SqStack;         // 顺序栈
***********/
Status Pop_Sq(SqStack &S, ElemType &e)
/* 顺序栈S的栈顶元素出栈到e，并返回OK；*/
/* 若失败，则返回ERROR。                */
{
    if(S.top == 0) return ERROR;
    //这里--S.top既可以找到栈顶元素
    //也可以实现栈顶位标减1
    e = S.elem[--S.top];
    return OK;
}
```

4、重新定义顺序栈，构建初始容量和扩容增量分别为size和inc的空顺序栈S

```
/**********
【题目】若顺序栈的类型重新定义如下。试编写算法，
构建初始容量和扩容增量分别为size和inc的空顺序栈S。
typedef struct {
  ElemType *elem; // 存储空间的基址
  ElemType *top;  // 栈顶元素的下一个位置
  int size;       // 当前分配的存储容量
  int increment;  // 扩容时，增加的存储容量
} SqStack2;
***********/
Status InitStack_Sq2(SqStack2 &S, int size, int inc)
/* 构建初始容量和扩容增量分别为size和inc的空顺序栈S。*/
/* 若成功，则返回OK；否则返回ERROR。                  */
{
    //考虑size、inc值不合理的情况
    if(size <= 0 || inc <= 0) return ERROR;
    //这里开辟的空间是ElemType而不是SqStack2
    S.elem = S.top = (ElemType *)malloc(sizeof(ElemType)*size);
    if(S.elem == NULL) return ERROR;
    S.size = size;
    S.increment = inc;
    return OK;
}
```

5、重新定义顺序栈，实现顺序栈的判空操作

```
/**********
【题目】若顺序栈的类型重新定义如下。试编写算法，
实现顺序栈的判空操作。
typedef struct {
  ElemType *elem; // 存储空间的基址
  ElemType *top;  // 栈顶元素的下一个位置
  int size;       // 当前分配的存储容量
  int increment;  // 扩容时，增加的存储容量
```

```
} SqStack2;
***********/
Status StackEmpty_Sq2(SqStack2 S)
/* 对顺序栈S判空。                          */
/* 若S是空栈，则返回TRUE；否则返回FALSE */
{
     //同地址为空栈
     if(S.elem != S.top)return FALSE;
     else return TRUE;
}
```

6、重新定义顺序栈，实现顺序栈的入栈操作

```
/**********
【题目】若顺序栈的类型重新定义如下。试编写算法，
实现顺序栈的入栈操作。
typedef struct {
  ElemType *elem; // 存储空间的基址
  ElemType *top;  // 栈顶元素的下一个位置
  int size;       // 当前分配的存储容量
  int increment;  // 扩容时，增加的存储容量
} SqStack2;
***********/
Status Push_Sq2(SqStack2 &S, ElemType e)
/* 若顺序栈S是满的，则扩容，若失败则返回ERROR。*/
/* 将e压入S，返回OK。                          */
{
    int i = 0;
    ElemType *temp = S.elem;
    while(temp != S.top)/*找到两者地址相等，此时i表示有多少个元素，类似顺序栈1中top的作用
*/
    {
        i++;
        temp ++;
    }
    temp = S.elem;
    if(i == S.size){
        /*内存重新分配，之所以用另一个变量来存储有以下原因：内存分配有几种情况(这个函数不太安
全)*/
        /*
            1、若已有内存后面还存在increment个连续地址，则把后面的increment内存纳入数组
中，此时temp的地址跟S.elem一致；
            2、如果原来的内存后面没有足够的空闲空间用来分配，那么会在栈中重新找一块newSize大
小的内存，然后把已有内存的数据复制到新内存中(数据已移动)，把新内存的地址返回给temp，而老块内存
重新放回栈中
            3、如果没有足够可用的内存用来完成重新分配（扩大原来的内存块或者分配新的内存块），
则返回null而原来的内存块保持不变。
            4、如果S.elem为null，则realloc()和malloc()类似。分配一个newsize的内存块，返
回一个指向该内存块的指针。
            5、如果newsize大小为0，那么释放S.elem指向的内存，并返回null。

            综上，这就是为什么重新分配内存时要定义一个临时指针，并最后把该指针指向NULL
         */
       temp = (ElemType *)realloc(S.elem,sizeof(ElemType)*(S.size
       +S.increment));
       if(temp == NULL)return ERROR;
```

```
        S.elem = temp;
        /*注意，之前这里如果没有下面这行代码，
          若如果原本已有六个元素(size)，
          则测试不通过，经调试发现：
          那个temp在内存重新分配后不一定是指向S.elem的值
          所以这时候S.top跟S.elem的地址差距甚大
          */
        S.top = S.elem + i;
        S.size += S.increment;
        temp = NULL;
    }
    *(S.top ++) = e;
    return OK;
}
```

7、重新定义顺序栈，实现顺序栈的出栈操作

```
/**********
【题目】若顺序栈的类型重新定义如下。试编写算法，
实现顺序栈的出栈操作。
typedef struct {
  ElemType *elem; // 存储空间的基址
  ElemType *top;  // 栈顶元素的下一个位置
  int size;       // 当前分配的存储容量
  int increment;  // 扩容时，增加的存储容量
} SqStack2;
***********/
Status Pop_Sq2(SqStack2 &S, ElemType &e)
/* 若顺序栈S是空的，则返回ERROR；     */
/* 否则将S的栈顶元素出栈到e，返回OK。*/
{
    if(S.elem == S.top) return ERROR;
    e = *(--S.top);
    return OK;
}
```

8、借助辅助栈，复制顺序栈S1得到S2

```
/**********
【题目】试写一算法，借助辅助栈，复制顺序栈S1得到S2。
顺序栈的类型定义为：
typedef struct {
  ElemType *elem; // 存储空间的基址
  int top;        // 栈顶元素的下一个位置，简称栈顶位标
  int size;       // 当前分配的存储容量
  int increment;  // 扩容时，增加的存储容量
} SqStack;        // 顺序栈
可调用顺序栈接口中下列函数：
Status InitStack_Sq(SqStack &S, int size, int inc); // 初始化顺序栈S
Status DestroyStack_Sq(SqStack &S); // 销毁顺序栈S
Status StackEmpty_Sq(SqStack S);    // 栈S判空，若空则返回TRUE，否则FALSE
Status Push_Sq(SqStack &S, ElemType e); // 将元素e压入栈S
Status Pop_Sq(SqStack &S, ElemType &e); // 栈S的栈顶元素出栈到e
***********/
Status CopyStack_Sq(SqStack S1, SqStack &S2)
/* 借助辅助栈，复制顺序栈S1得到S2。     */
```

```
/* 若复制成功，则返回TRUE；否则FALSE。 */
{
     //先初始化
     Status status = InitStack_Sq(S2,S1.size,S1.increment);
     if(status == OVERFLOW)return FALSE;
     if(S1.top == 0) return TRUE;
     for(int i = 0;i < S1.top;i ++){
        Push_Sq(S2,S1.elem[i]);
     }
     DestroyStack_Sq(S1);
     return OK;
}
```

9、求循环队列的长度

```
/**********
【题目】试写一算法，求循环队列的长度。
循环队列的类型定义为：
typedef struct {
  ElemType *base;  // 存储空间的基址
  int front;       // 队头位标
  int rear;        // 队尾位标，指示队尾元素的下一位置
  int maxSize;     // 最大长度
} SqQueue;
***********/
int QueueLength_Sq(SqQueue Q)
/* 返回队列Q中元素个数，即队列的长度。 */
{
    //保留一个内存空间不用的方案
    //有两种情况：rear在front后；rear在front前
    return (Q.rear-Q.front+Q.maxSize)%Q.maxSize;
}
```

10、设置一个标志域tag标记队列的空或满，编写与此结构相应的入队列和出队列的算法

```
/**********
【题目】如果希望循环队列中的元素都能得到利用，
则可设置一个标志域tag，并以tag值为0或1来区分尾
指针和头指针值相同时的队列状态是"空"还是"满"。
试编写与此结构相应的入队列和出队列的算法。
本题的循环队列CTagQueue的类型定义如下：
typedef struct {
  ElemType elem[MAXQSIZE];
  int tag;
  int front;
  int rear;
} CTagQueue;
**********/
/*方案一：利用出队时顺便把出队的那个存储空间清0，然后在入队函数那里判断是否队列满；该方法有局限性，如遇到那种不能清零的存储空间比如非基本类型就不能用了，还是方案二简单易懂*/

Status EnCQueue(CTagQueue &Q, ElemType x)
/* 将元素x加入队列Q，并返回OK；*/
/* 若失败，则返回ERROR。        */
```

```
{
    if(Q.rear == Q.front && Q.tag == 1){//队列满
        return ERROR;
    }
    Q.elem[Q.rear] = x;
    Q.rear = (Q.rear + 1)%MAXQSIZE;
    if(Q.elem[Q.rear] && Q.rear == Q.front)
    {//rear等于front且该值不为0（因为在出队时设置出队的元素为0）
     //如果该值不为0说明该地方有数据，即队列满
        Q.tag = 1;
    }else{
        Q.tag = 0;
    }
    return OK;
}

Status DeCQueue(CTagQueue &Q, ElemType &x)
/* 将队列Q的队头元素退队到x，并返回OK；*/
/* 若失败，则返回ERROR。                */
{
    if(Q.rear == Q.front && Q.tag == 0){//队列空
        return ERROR;
    }
    x = Q.elem[Q.front];
    Q.elem[Q.front] = 0;//值清零，为了入队时判断是否满
    /*if(!Q.elem[(Q.front+1)%MAXQSIZE]){//队列的Q.elem[Q.rear]为0?
        Q.tag = 0;
    }else Q.tag = 1; */
    Q.front = (Q.front+1)%MAXQSIZE;
    return OK;
}

/*方案二：入队后,如果rear == front,那么说明队列满；出队后,如果rear == front,那么该队列为空
*/

Status EnCQueue(CTagQueue &Q, ElemType x)
/* 将元素x加入队列Q，并返回OK；*/
/* 若失败，则返回ERROR。        */
{
    if(Q.rear == Q.front && Q.tag == 1){//队列满
        return ERROR;
    }
    Q.elem[Q.rear] = x;
    Q.rear = (Q.rear + 1)%MAXQSIZE;
    //入队后,如果rear == front,那么说明队列满
    if(Q.rear == Q.front) Q.tag = 1;
    return OK;
}

Status DeCQueue(CTagQueue &Q, ElemType &x)
/* 将队列Q的队头元素退队到x，并返回OK；*/
/* 若失败，则返回ERROR。                */
{
    if(Q.rear == Q.front && Q.tag == 0){//队列空
        return ERROR;
    }
    x = Q.elem[Q.front];
    Q.front = (Q.front+1)%MAXQSIZE;
```

```
    //出队后,如果rear == front,那么该队列为空
    if(Q.rear == Q.front) Q.tag = 0;
    return OK;
}
```

11、设一个长度域length记录个数，实现相应的入队列和出队列的算法，

```
/**********
【题目】假设将循环队列定义为：以域变量rear
和length分别指示循环队列中队尾元素的位置和内
含元素的个数。试给出此循环队列的队满条件，并
写出相应的入队列和出队列的算法（在出队列的算
法中要返回队头元素）。
本题的循环队列CLenQueue的类型定义如下：
typedef struct {
  ElemType elem[MAXQSIZE];
  int length;
  int rear;
} CLenQueue;
**********/
Status EnCQueue(CLenQueue &Q, ElemType x)
  /* 将元素x加入队列Q，并返回OK；*/
  /* 若失败，则返回ERROR。        */
{
    if(Q.length == MAXQSIZE)return ERROR;//队列满
    Q.rear = (Q.rear+1)%MAXQSIZE;//改变rear
    /*注意，这里rear是指队尾元素，不是队尾位标*/
    Q.elem[Q.rear] = x;
    Q.length ++;//记得自增
    return OK;
}
Status DeCQueue(CLenQueue &Q, ElemType &x)
  /* 将队列Q的队头元素退队到x，并返回OK；*/
  /* 若失败，则返回ERROR。                */
{
    if(Q.length == 0) return ERROR;//队列空
    /*队头元素考虑队尾在队头后面的情况哦
      Q.rear + 1是为了得到队尾位标*/
    x = Q.elem[(Q.rear + 1 - Q.length + MAXQSIZE) % MAXQSIZE];
    Q.length --;//记得自减
    return OK;
}
```

12、用循环队列编写求k阶斐波那契序列中第n+1项f_n的算法

```
/**********
【题目】已知k阶斐波那契序列的定义为:
    f(0)=0, f(1)=0, ..., f(k-2)=0, f(k-1)=1;
    f(n)=f(n-1)+f(n-2)+...+f(n-k), n=k,k+1,...
试利用循环队列编写求k阶斐波那契序列中第
n+1项fn的算法。

本题的循环队列的类型定义如下：
```

```
typedef struct {
  ElemType *base; // 存储空间的基址
  int front;      // 队头位标
  int rear;       // 队尾位标，指示队尾元素的下一位置
  int maxSize;    // 最大长度
} SqQueue;
**********/
long Fib(int k, int n)
/* 求k阶斐波那契序列的第n+1项fn */
{
     if(k < 2 || n < -1) return ERROR;//k,n不合理
     long result = 0;
     //这里求的n+1项貌似是从1算起的
     if(n < k-1) return result;
     if(n == k-1){
        result = 1;
        return result;
     }
     SqQueue Q;
     int i,j = 0;
     /* 下面代码这样写可以，但是这样根本没有体现出队列的性质
        更不用说循环队列了，根据循环队列根本不需要开辟n+1个空间的，
        只需要开辟k个空间，先存储0~k-1的，然后
        最后第k项入队，队头元素出队
     Q.maxSize = n + 1;
     Q.base = (ElemType *)malloc(sizeof(ElemType)*Q.maxSize);
     if(NULL == Q.base) return OVERFLOW;
     Q.rear = Q.front = 0;
     for(i = 0;i <= k - 2;i ++){
         Q.base[i] = 0;
     }
     Q.base[i] = 1;
     for(i = k;i <= n;i ++){
        //判断条件不是j<k哦，是j<i才对
        for(j = i - k;j < i;j ++){
            result += Q.base[j];
        }
        Q.base[i] = result;
        result = 0;//记得清零
     }
     return Q.base[n];
     */
     Q.maxSize = k + 1;//用队尾位标与队头元素差一个空间的方案
     Q.base = (ElemType *)malloc(sizeof(ElemType)*Q.maxSize);
     if(NULL == Q.base) return OVERFLOW;
     Q.rear = Q.front = 0;
     for(i = 0;i <= k -2;i ++){
        Q.base[Q.rear ++] = 0;
        Q.rear %= Q.maxSize;
     }
     Q.base[Q.rear ++] = 1;
     for(i = k;i <= n;i ++){
        result = 0;
        for(j = 0;j < k;j ++){//这里j就没必要初始化为j=i-k了
            result +=  Q.base[(Q.front + j)%Q.maxSize];
        }
        //移除当前队头元素，因为以后用不到了
        //之所以在赋值给rear之前移除是为了防止出现队列满的情况
```

```
        //移除前rear与front只差一个空间了
        Q.front = (Q.front+1)%Q.maxSize;
        if((Q.rear+1)%Q.maxSize == Q.front) return ERROR;//队列满
        Q.base[Q.rear ++] = result;
        Q.rear %= Q.maxSize;
    }
    /*返回队尾元素或者result，注意队尾元素时如果rear已经取余了
      那这时候rear-1会出现问题的*/
    //return Q.base[Q.rear-1];
    //用队尾元素返回，不能free内存了，建议用result返回
    //return Q.base[(Q.rear-1+Q.maxSize)%Q.maxSize];
    free(Q.base);
    return result;
}
```

13、比较两个有序顺序表

```
/**********
【题目】设A=(a1,…,am)和B=(b1,…,bn)均为有序顺序表，
A'和B'分别为A和B中除去最大共同前缀后的子表（例如，
A=(x,y,y,z,x,z)，B=(x,y,y,z,y,x,x,z)，则两者中最大
的共同前缀为(x,y,y,z)， 在两表中除去最大共同前缀后
的子表分别为A'=(x,z)和B'=(y,x,x,z)）。若A'=B'=空表，
则A=B；若A'=空表，而B'≠ 空表，或者两者均不为空表，
且A'的首元小于B'的首元，则A<B；否则A>B。试写一个比
较A和B大小的算法。（注意：在算法中，不要破坏原表A
和B，也不一定先求得A'和B'才进行比较）。
顺序表类型定义如下：
typedef struct {
  ElemType *elem;
  int       length;
  int       size;
  int       increment;
} SqList;
**********/
char Compare(SqList A, SqList B)
/* 比较顺序表A和B，       */
/*   返回'<', 若A<B;     */
/*       '=', 若A=B;     */
/*       '>', 若A>B      */
{
    char result = '=';
    //如果没有前缀
    if(A.elem[0] > B.elem[0]) result = '>';
    else if(A.elem[0] < B.elem[0]) result = '<';
    //有前缀
    else{
        //i用于遍历，j用于表示有多少个前缀相同
        //都从1开始哦，因为首元相同了
        int i = 1,j = 1;
        //这里的判断条件是==而不是!=,
        //但是要考虑如果A的所有元素全部等于B的时，
        //该循环无法退出，所有还要加点条件
        while(A.elem[i] == B.elem[i]){
            if(j == A.length || j == B.length) break;
            i ++;
```

```
            j ++;
        }
        //A'=B'=空表的情况不用讨论，因为result默认值为‘=’
        if(A.length == j){
            //A'=空表，而B'≠ 空表
            if(j != B.length) result = '<';
        }else{
            if(j == B.length) result = '>';
            else{
                //A'的首元大于B'的首元
                if(A.elem[i] > B.elem[i]) result = '>';
                else result = '<';
            }
        }
    }
    return result;
}
```

14、实现顺序表的就地逆置

```
/**********
【题目】试写一算法，实现顺序表的就地逆置，
即利用原表的存储空间将线性表(a1,a2,…,an)
逆置为(an,an-1,…,a1)。
顺序表类型定义如下：
typedef struct {
  ElemType *elem;
  int       length;
  int       size;
  int       increment;
} SqList;
**********/
void Inverse(SqList &L)
{
   if(L.length <= 1) return ;//0或1个元素
   int i;//遍历
   ElemType temp;
   for(i = 0;i < L.length / 2;i ++){//对半交换
        temp = L.elem[i];
        L.elem[i] = L.elem[L.length-1-i];
        L.elem[L.length-1-i] = temp;
   }
   return ;
}
```

15、计算一元稀疏多项式的值

```
/**********
【题目】试对一元稀疏多项式Pn(x)采用存储量同多项式
项数m成正比的顺序存储结构，编写求Pn(x0)的算法（x0
为给定值）。

一元稀疏多项式的顺序存储结构：
typedef struct {
  int  coef;  // 系数
  int   exp;  // 指数
```

```
} Term;

typedef struct {
  Term  *elem;   // 存储空间基址
  int    length; // 长度（项数）
} Poly;
**********/
float Evaluate(Poly P, float x)
/* P.elem[i].coef 存放ai，                          */
/* P.elem[i].exp存放ei (i=1,2,...,m)               */
/* 本算法计算并返回多项式的值。不判别溢出。        */
/* 入口时要求0≤e1<e2<...<em，算法内不对此再作验证 */
{
    float sum = 0,temp = 1;
    int i,j;
    //根据题目，i应该从1开始；但是从i=1开始会出错...
    for(i = 0;i <= P.length;i ++){
       for(j = 1;j <= P.elem[i].exp; j ++) temp *= x;
       sum += temp * P.elem[i].coef;
       temp = 1;
    }
    return sum;
}
```

16、两个线性表(集合)的并

```
/**********
【题目】假设有两个集合A和B分别用两个线性表LA和LB
表示(即：线性表中的数据元素即为集合中的成员），
试写一算法，求并集A＝A∪B。
顺序表类型定义如下
typedef struct {
  ElemType *elem;     // 存储空间的基址
  int length;    // 当前长度
  int size;      // 存储容量
  int increment; // 空间不够增加空间大小
} SqList;  // 顺序表
可调用顺序表的以下接口函数：
Status InitList_Sq(SqList &L, int size, int inc); // 初始化顺序表L
int ListLength_Sq(SqList L);  // 返回顺序表L中元素个数
Status GetElem_Sq(SqList L, int i, ElemType &e);
// 用e返回顺序表L中第i个元素的值
int Search_Sq(SqList L, ElemType e);
// 在顺序表L顺序查找元素e，成功时返回该元素在表中第一次出现的位置，否则返回-1
Status Append_Sq(SqList &L, ElemType e);  // 在顺序表L表尾添加元素e
**********/
void Union(SqList &La, SqList Lb)
{
   int i = 0; //i遍历Lb
   ElemType e,*temp;//e表示Lb所取出的元素
   //这个GetElem_Sq是从1开始的
   for(i = 1;i <= Lb.length;i ++){
       /* 按理说Append_Sq函数会有以下代码的
       if(La.length == La.size){
           temp = (ElemType *)realloc(La.elem,
(La.size+La.increment)*sizeof(ElemType));
           if(NULL == temp) return;
```

```
            La.elem = temp;
            La.size += La.increment;
        }*/

        GetElem_Sq(Lb,i,e);
        //这里不要直接把GetElem_Sq(Lb,i,e)作Search_Sq的参数
        //因为该函数有Status类型返回值的，所以也不要用e=函数
        if(Search_Sq(La,e) == -1){
             Append_Sq(La,e);
        }
    }
}
```

17、实现链栈的判空操作

```
/**********
【题目】试写一算法，实现链栈的判空操作。
链栈的类型定义为：
typedef struct LSNode {
  ElemType data;        // 数据域
  struct LSNode *next; // 指针域
} LSNode, *LStack;     // 结点和链栈类型
***********/
Status StackEmpty_L(LStack S)
/* 对链栈S判空。若S是空栈，则返回TRUE；否则返回FALSE */
{
     //如果S->next为空且数据域也为空，那么链栈空
     //data的默认值视情况而定
     if(S->next == NULL && S->data == 0) return TRUE;
     return FALSE;
}
```

18、实现链栈的取栈顶元素操作

```
/**********
【题目】试写一算法，实现链栈的取栈顶元素操作。
链栈的类型定义为：
typedef struct LSNode {
  ElemType data;        // 数据域
  struct LSNode *next; // 指针域
} LSNode, *LStack;     // 结点和链栈类型
***********/
Status GetTop_L(LStack S, ElemType &e)
/* 取链栈S的栈顶元素到e，并返回OK； */
/* 若S是空栈，则失败，返回ERROR。   */
{
    if(S->next == NULL && S->data == 0) return ERROR;
    //S就是栈顶元素
    e = S->data;
    return OK;
}
```

19、实现链队列的判空操作

```
/**********
【题目】试写一算法，实现链队列的判空操作。
链队列的类型定义为：
typedef struct LQNode {
  ElemType  data;
  struct LQNode  *next;
} LQNode, *QueuePtr; // 结点和结点指针类型
typedef struct {
  QueuePtr  front;  // 队头指针
  QueuePtr  rear;   // 队尾指针
} LQueue;  // 链队列类型
***********/
Status QueueEmpty_LQ(LQueue Q)
/* 判定链队列Q是否为空队列。              */
/* 若Q是空队列，则返回TRUE，否则FALSE。*/
{
     //注意，如果是只有一个结点的空队列，
     //那么此时Q.front == Q.rear，然而非空
     //所以，如果初始化队头元素为NULL就比较容易判断
     if(Q.front == NULL ) return TRUE;
     return FALSE;
}
```

20、实现链队列的求队列长度操作

```
/**********
【题目】试写一算法，实现链队列的求队列长度操作。
链队列的类型定义为：
typedef struct LQNode {
  ElemType  data;
  struct LQNode  *next;
} LQNode, *QueuePtr; // 结点和结点指针类型
typedef struct {
  QueuePtr  front;  // 队头指针
  QueuePtr  rear;   // 队尾指针
} LQueue;  // 链队列类型
***********/
int QueueLength_LQ(LQueue Q)
/* 求链队列Q的长度并返回其值 */
{
     if(Q.front == NULL) return 0;//队列空
     //下面的循环当长度为3时实际上只循环了两次
     //所以i的初始值应该设置为1，即只有一个元素
     int i = 1;
     QueuePtr temp = Q.front;
     while(temp != Q.rear){
        i++;
        temp = temp->next;
     }
     return i;
}
```

21、以带头结点的循环链表表示队列，编写相应的队列初始化、入队列和出队列的算法

```
/**********
【题目】假设以带头结点的循环链表表示队列，并且
只设一个指针指向队尾元素结点(注意不设头指针)，
试编写相应的队列初始化、入队列和出队列的算法。
带头结点循环链队列CLQueue的类型定义为：
typedef struct LQNode {
  ElemType data;
  struct LQNode *next;
} LQNode, *CLQueue;
**********/
Status InitCLQueue(CLQueue &rear) // 初始化空队列
{
     //初始化头结点
     rear = (LQNode *)malloc(sizeof(LQNode));
     if(NULL == rear)return OVERFLOW;
     rear->next = rear; //记得这里形成回环哦
     return OK;
}

Status EnCLQueue(CLQueue &rear, ElemType x) // 入队
{
     //head是头结点,尾结点的next指向它
     CLQueue temp,head = rear->next;
     temp = (LQNode *)malloc(sizeof(LQNode));
     if(temp == NULL) return OVERFLOW;
     temp->data = x;
     temp->next = head;//指向头结点
     rear->next = temp;
     rear = temp;//这里要把尾结点更新
     //free(temp);//这里不能释放该空间
     head = NULL; //head赋值为空，但是不能free
     return OK;
}

Status DeCLQueue(CLQueue &rear, ElemType &x) // 出队
{
     CLQueue front,head = rear->next;
     //注意，这里的判断队头元素为空不可以为
     //head->next == NULL，因为当为空时head->next=rear
     if(head->next == rear) return ERROR;//空队列
     front = head->next;//队头元素为头结点的next
     x = front->data;
     head->next = front->next;//重新定义队头元素
     free(front);//释放该空间
     head = NULL;//head归空
     return OK;
}
```

22、实现带头结点单链表的判空操作

```
/**********
【题目】试写一算法，实现带头结点单链表的判空操作。

单链表的类型定义为：
typedef struct LNode {
  ElemType  data;
  struct LNode  *next;
```

```
} LNode, *LinkList; // 结点和结点指针类型
***********/
Status ListEmpty_L(LinkList L)
/* 判定带头结点单链表L是否为空链表。    */
/* 若L是空链表，则返回TRUE，否则FALSE。*/
{
    if(L->next == NULL)return TRUE;
    else return FALSE;
}
```

23、实现带头结点单链表的销毁操作

```
/**********
【题目】试写一算法，实现带头结点单链表的销毁操作。
单链表的类型定义为：
typedef struct LNode {
  ElemType  data;
  struct LNode  *next;
} LNode, *LinkList; // 结点和结点指针类型
***********/
Status DestroyList_L(LinkList &L)
/* 销毁带头结点单链表L，并返回OK。*/
{
    LinkList temp = L,p = NULL;
    //这里如果写成temp->next != NULL
    //那么最后还要销毁temp
    while(temp != NULL){
        p = temp;
        temp = p->next;//把当前节点的下个节点赋值给temp
        free(p);//销毁当前节点
    }
    //temp = p = NULL;//这行没必要，temp退出循环条件为null
    free(L);//销毁头结点
    /*不知道为什么temp=L，明明在最开始时由循环free了，为何
    最后free(L)代码不写会全部错误*/
    return OK;
}
```

24、实现带头结点单链表的清空操作

```
/**********
【题目】试写一算法，实现带头结点单链表的清空操作。

单链表的类型定义为：
typedef struct LNode {
  ElemType  data;
  struct LNode  *next;
} LNode, *LinkList; // 结点和结点指针类型
***********/
Status ClearList_L(LinkList &L)
/* 将带头结点单链表L置为空表，并返回OK。*/
/* 若L不是带头结点单链表，则返回ERROR。  */
{
    if(L == NULL)return ERROR;//L不是带头结点单链表
    /*清空与销毁的区别是：前者不销毁头结点*/
    if(L->next == NULL) return OK;
```

```
    LinkList temp = L->next,p = NULL;
    while(temp != NULL){
        p = temp;
        temp = p->next;
        free(p);
    }
    L->next = NULL;//最后记得说明这行代码
    return OK;
}
```

25、实现带头结点单链表的求表长度操作

```
/**********
【题目】试写一算法，实现带头结点单链表的求表长度操作。
单链表的类型定义为：
typedef struct LNode {
  ElemType  data;
  struct LNode  *next;
} LNode, *LinkList; // 结点和结点指针类型
***********/
int ListLength_L(LinkList L)
/* 求带头结点单链表L的长度，并返回长度值。*/
/* 若L不是带头结点单链表，则返回-1。       */
{
    if(L == NULL)return -1;//L不是带头结点单链表
    int i = 0;//i表示长度
    LinkList temp = L->next;//记得从L->next出发，要不然长度多1个
    while(temp != NULL){
        i ++;
        temp = temp -> next;
    }
    return i;
}
```

26、在带头结点单链表L插入第i元素e

```
/**********
【题目】试写一算法，在带头结点单链表L插入第i元素e。
带头结点单链表的类型定义为：
typedef struct LNode {
  ElemType      data;
  struct LNode *next;
} LNode, *LinkList;
**********/
Status Insert_L(LinkList L, int i, ElemType e)
/* 在带头结点单链表L插入第i元素e，并返回OK。*/
/* 若参数不合理，则返回ERROR。               */
{
    if(i < 1) return ERROR;//参数i不合理,应从1开始
    int j = 0;//j表示链表长度
    LinkList temp = L->next,p = NULL;//p表示插入元素
    while(temp != NULL){
        j ++;
        temp = temp->next;
    }
    temp = L;
```

```
    /*链表长度+1是考虑i表示想插入表尾元素后面时*/
    if(i > j + 1) return ERROR;//比现有链表长度+1大，不合理
    for(j = i - 1;j > 0;j --){
         temp = temp->next;//当temp指向第i-1个元素时退出
         /*第0个元素即头结点*/
    }
    p = (LNode *)malloc(sizeof(LNode));
    if(p == NULL)return OVERFLOW;
    p->data = e;
    p->next = temp->next;
    temp->next = p;
    return OK;
}
```

27、在带头结点单链表删除第i元素到e

```
/**********
【题目】试写一算法，在带头结点单链表删除第i元素到e。
带头结点单链表的类型定义为：
typedef struct LNode {
  ElemType      data;
  struct LNode *next;
} LNode, *LinkList;
**********/
Status Delete_L(LinkList L, int i, ElemType &e)
/* 在带头结点单链表L删除第i元素到e，并返回OK。*/
/* 若参数不合理，则返回ERROR。                  */
{
    if(i < 1)return ERROR;
    int j = 0;
    LinkList temp = L->next,p = NULL;//p指向删除结点
    while(temp != NULL){
        j ++;
        temp = temp->next;
    }
    temp = L;
    /*这个不能大于j+1了哦*/
    if(i > j)return ERROR;
    for(j = i - 1;j > 0;j --){
        temp = temp -> next;//退出条件是temp指向第i-1个元素
    }
    /*不用另外考虑只有1个结点的情况*/
    /* 这里的代码有错，因为free(temp->next)中此时
       temp->next不指向删除结点
    e = temp->next->data;
    temp->next = temp->next->next;
    free(temp->next);//释放该结点空间
    */
    p = temp->next;
    e = p->data;
    temp->next = p->next;
    free(p);//释放该结点空间
    return OK;
}
```

28、在带头结点单链表的第i元素起的所有元素从链表移除，并构成一个带头结点的新链表

```
/**********
【题目】试写一算法，在带头结点单链表的第i元素起的
所有元素从链表移除，并构成一个带头结点的新链表。
带头结点单链表的类型定义为：
typedef struct LNode {
  ElemType      data;
  struct LNode *next;
} LNode, *LinkList;
**********/

/*方案1*/
Status Split_L(LinkList L, LinkList &Li, int i)
/* 在带头结点单链表L的第i元素起的所有元素 */
/* 移除，并构成带头结点链表Li，返回OK。    */
/* 若参数不合理，则Li为NULL，返回ERROR。   */
{
    /*经过多轮测试，发现当i出错时，Li应该为NULL*/
    if(i < 1){
        Li = NULL;
        return ERROR;
    }
    int j = 0;
    /*注意，Li还没创建头结点的*/
    LinkList temp = L->next,p = NULL;//p指向被移除元素
    while(temp != NULL){
        j ++;
        temp = temp->next;
    }
    temp = L;//重置temp
    if(i > j){
        Li = NULL;
        return ERROR;
    }//参数不合理
    for(j = i-1;j > 0;j --){
        temp = temp->next;//temp指向第i-1个元素且保持不变
    }
    p = temp->next;//第i个元素
    /*不用循环那么麻烦的
    while(p != NULL){
        q->next = p;
        q = p;
        temp->next = p->next;//原链表的第i个元素递增
        p = p->next;
    }*/
    if(p != NULL){//L不为空链表的情况
        Li = (LNode *)malloc(sizeof(LNode));//不为空才创建
        if(Li == NULL)return OVERFLOW;
        Li->next = p;
        temp->next = NULL;
    }
    return OK;
}
```

```
/*方案二:优化，减少一个循环*/
Status Split_L(LinkList L, LinkList &Li, int i)
/* 在带头结点单链表L的第i元素起的所有元素 */
/* 移除，并构成带头结点链表Li，返回OK。    */
/* 若参数不合理，则Li为NULL，返回ERROR。   */
{
    /*经过多轮测试，发现当i出错时，Li应该为NULL*/
    if(i < 1){
        Li = NULL;
        return ERROR;
    }
    int j = 0;
    /*注意，Li还没创建头结点*/
    LinkList temp = L,p = NULL,q = NULL;//p指向被移除元素
    while(temp->next != NULL && j < i){//当j=i时会退出，此时q为第i-1个元素
        j ++;
        q = temp;
        temp = temp->next; //temp为第i个元素
    }
    if(i > j){//判断是否是temp->next为空而退出
        Li = NULL;
        return ERROR;
    }//参数不合理

    if(temp != NULL){//L不为空链表的情况
        Li = (LNode *)malloc(sizeof(LNode));//不为空才创建
        if(Li == NULL)return OVERFLOW;
        Li->next = temp;
        q->next = NULL;
    }
    return OK;
}
```

29、在带头结点单链表删除第i元素起的所有元素。

```
/**********
【题目】试写一算法，在带头结点单链表删除第i元素
起的所有元素。
带头结点单链表的类型定义为:
typedef struct LNode {
  ElemType      data;
  struct LNode *next;
} LNode, *LinkList;
**********/
Status Cut_L(LinkList L, int i)
/* 在带头结点单链表L删除第i元素起的所有元素，并返回OK。*/
/* 若参数不合理，则返回ERROR。                          */
{
    if(i < 1) return ERROR;
    int j = 0;
    LinkList temp = L,p = NULL;
    while(temp->next != NULL && j < i){
        j ++;
        p = temp;
        temp = temp->next;
    }
```

```
    if(i > j){
        return ERROR;
    }
    p->next = NULL;//第i-1个元素，此后元素删除
    while(temp != NULL){
        p = temp;
        temp = p->next;
        free(p);
    }
    return OK;
}
```

30、删除带头结点单链表中所有值为x的元素，并释放被删结点空间。

```
/**********
【题目】试写一算法，删除带头结点单链表中所有值
为x的元素，并释放被删结点空间。
单链表类型定义如下：
typedef struct LNode {
  ElemType      data;
  struct LNode *next;
} LNode, *LinkList;
**********/
Status DeleteX_L(LinkList L, ElemType x)
/* 删除带头结点单链表L中所有值为x的元素，        */
/* 并释放被删结点空间，返回实际删除的元素个数。*/
{
   int result = 0;//初始化删除个数为0
   if(L->next == NULL) return result;
   LinkList temp = L,p = NULL;//p指向被删除元素
   while(temp->next != NULL){
        if(temp->next->data == x){//已包含表尾元素的情况
            p = temp->next;
            temp->next = p->next;
            free(p);
            result ++;
            /*注意这行代码，因为此时temp->next已改变
            如果继续执行temp = temp->next,那么会跳过一个元素，
            因此如果是连续多个元素相同，则会出错*/
            continue;
        }
        temp = temp->next;
   }
   return result;
}
```

31、删除带头结点单链表中所有值小于x的元素，并释放被删结点空间。

```
/**********
【题目】试写一算法，删除带头结点单链表中所有值
小于x的元素，并释放被删结点空间。
单链表类型定义如下：
typedef struct LNode {
```

```
  ElemType      data;
  struct LNode *next;
} LNode, *LinkList;
**********/
Status DeleteSome_L(LinkList L, ElemType x)
/* 删除带头结点单链表L中所有值小于x的元素，     */
/* 并释放被删结点空间，返回实际删除的元素个数。*/
{
    int result = 0;//初始化删除个数为0
    //if(L->next == NULL) return result; 无必要
    LinkList temp = L,p = NULL;
    while(temp->next != NULL){
        if(temp->next->data < x){
            p = temp->next;
            temp->next = p->next;
            free(p);
            result ++;
            continue;
        }
        temp = temp->next;
    }
    return result;
}
```