3.3 栈与递归

1. 什么是递归

递归是算法设计中最常用的手段，它通常把一个大型复杂问题的描述和求解过程变得简洁和清晰。因此递归算法常常比非递归算法更易设计。

递归定义：

简单地说，一个用自己定义自己的概念，称为递归定义。

例 n!= 1*2*3*4*(n-1)*n

n!递归定义 { n!= 1 当 n=1时

n!= n (n-1)! 当n> 1时

用(n-1)!定义n!

递归函数：一个直接调用自己或通过一系列调用间接调用自己的函数称为递归函数。

```
A( ){
   …
   A( ) ;
   …
}
```

A 直接调用自己

```
B( ) {
   …
   C( );
   …
}
```

```
C( ) {
   …
   B( );
   …
}
```

B间接调用自己

2．递归算法的编写

1）将问题用递归的方式描述（定义）

2）根据问题的递归描述（定义）编写递归算法

问题的递归描述（定义）

递归定义包括两项

递归项：

将问题分解为与原问题性质相同，但规模较小的问题；

基本项（终止项）：描述递归终止时问题的求解；

例 n!的递归定义

基本项：　　n!=1　　当 n=1

递归项：　　n!=n（n-1)!　　当 n＞1

例1　编写求解 阶乘n! 的递归算法

首先给出阶乘n! 的递归定义

n!的递归定义

基本项：　n!=1　　当 n=1

递归项：　n!=n (n-1)!　　当 n> 1

有了问题的递归定义，很容易写出对应的递归算法：

```
int fact (int n) {
    //算法功能是求解并返回n的阶乘
    if（n==1）  return（1）；
    else return（n*fact (n-1)）；
}
```

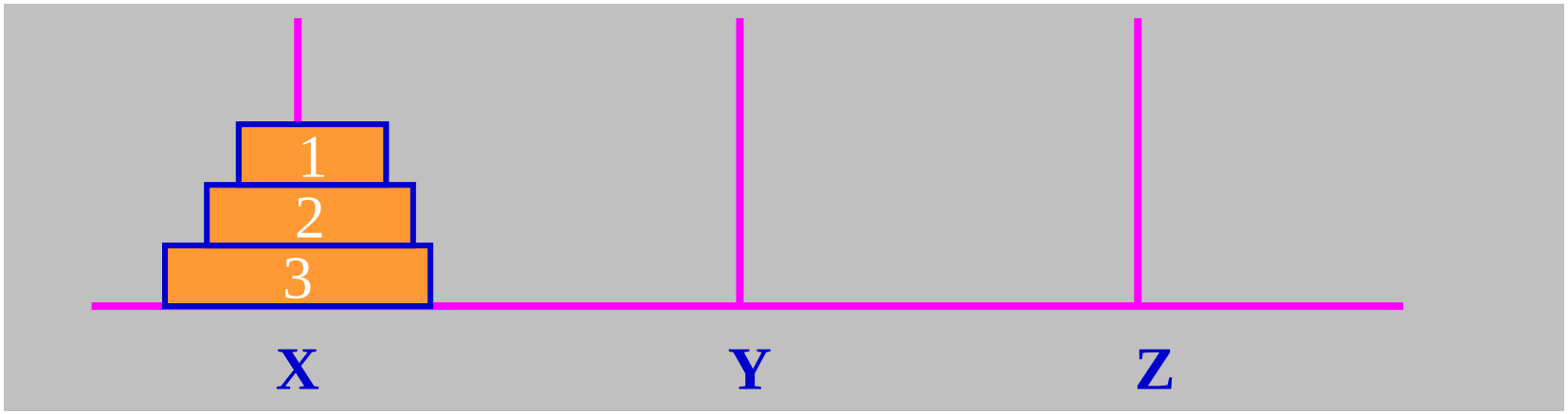
例2. 编写求解Hanoi塔问题的递归算法

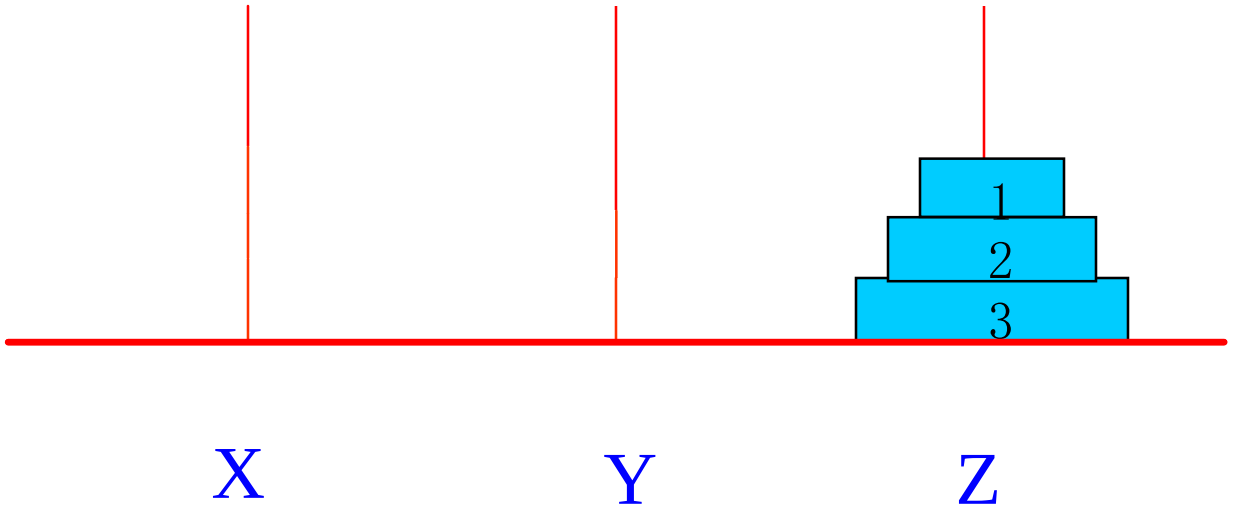
有三个各为X，Y，Z的塔座，在X上有n个大小不同，依小到大编号为1，2…n的圆盘。 现要求将X上的n个圆盘移至Z上，并仍以同样顺序叠放， 圆盘移动时必须遵守下列原则：

1）每次移动一个盘子；

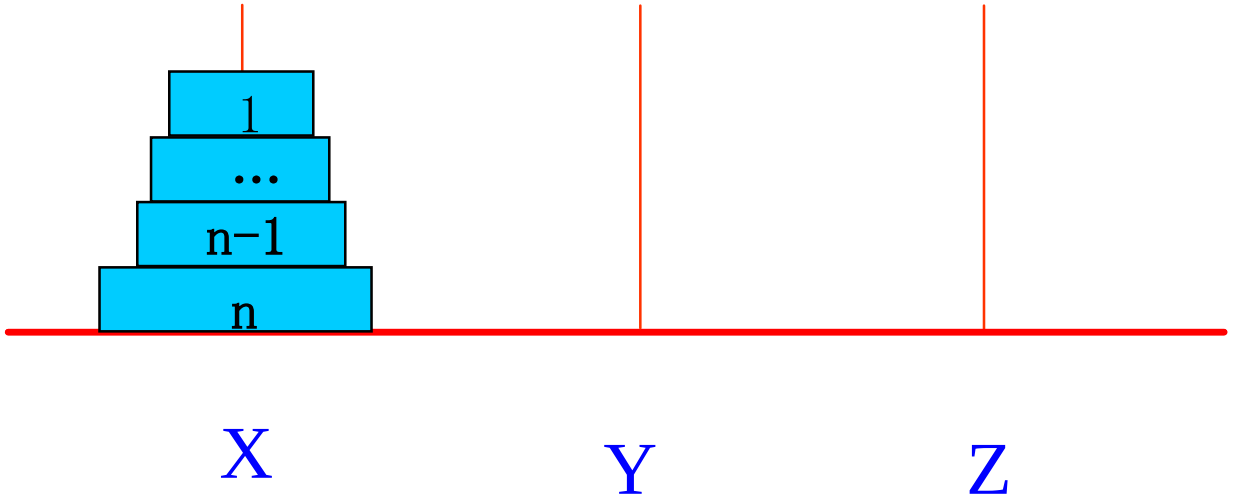
2）圆盘可以放在X，Y，Z中的任一塔座上；

3）任何时刻都不能将较大的圆盘压放在较小圆盘之上；



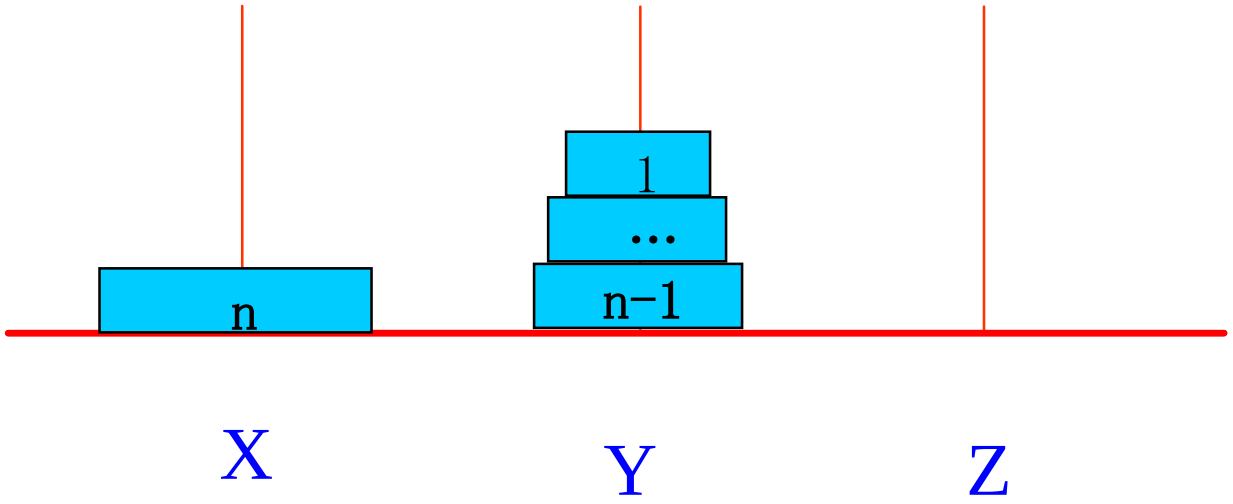

1
2
3
X
Y
Z


n
X
Y
Z

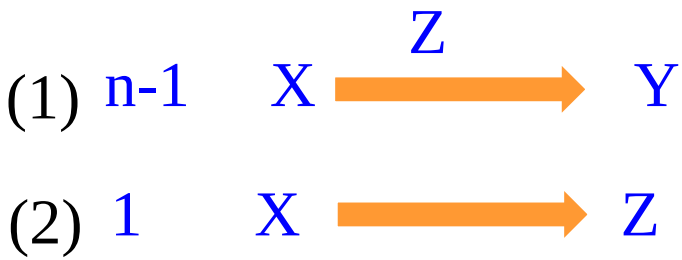

1
...
n−1
n
X
Y
Z

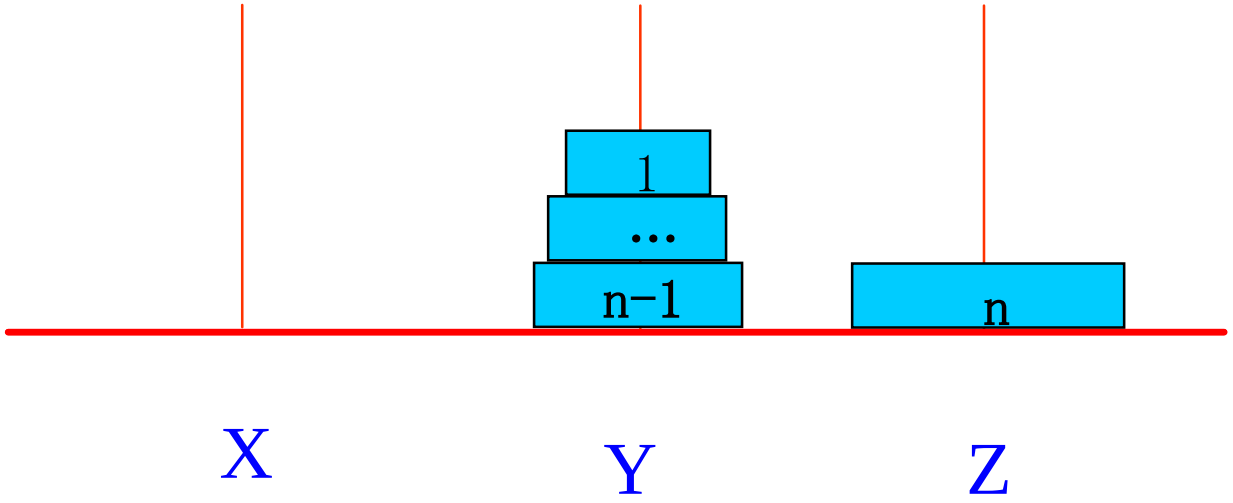
n X Y Z

(1) n-1 X Z Y

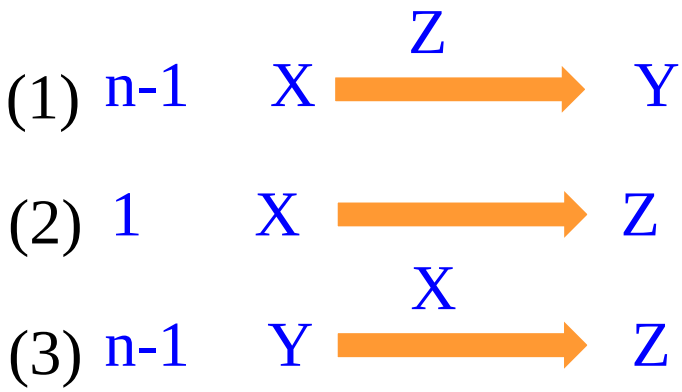


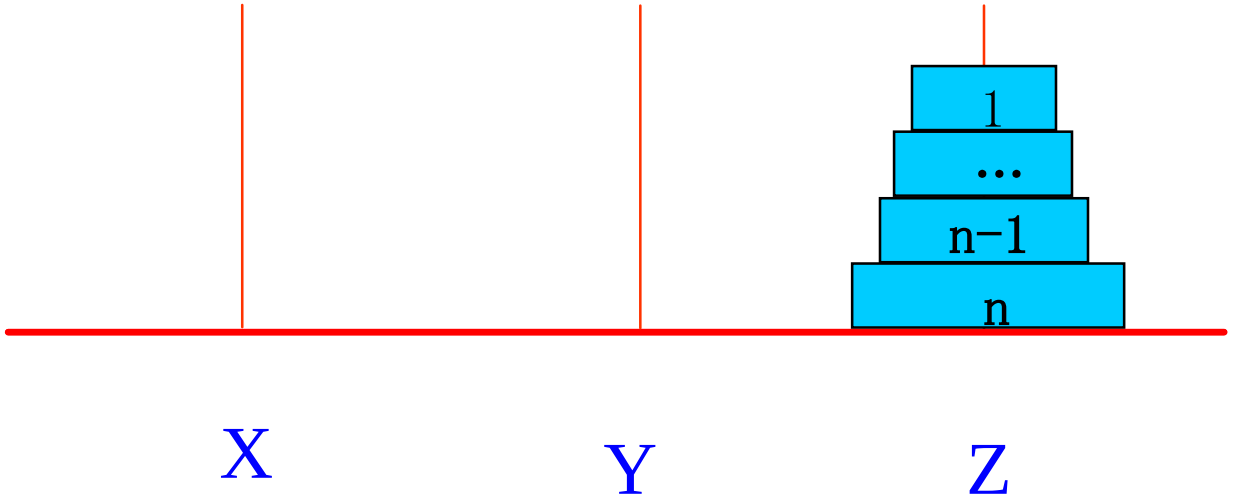

n
X
Y
Z


(1) n-1
X
Z
Y
(2) 1
X
Z


1
...
n−1
n
X
Y
Z


n
X
Y
Z


(1) n-1 X Z Y
(2) 1 X Z
(3) n-1 X Y Z


1
...
n−1
n
X
Y
Z

首先给出求解Hanoi塔问题的递归定义

基本项： n=1时，将n号圆盘从X移至Z；

递归项： n>1时，

将X上1— n-1号圆盘移至Y；

将X上的n号圆盘从X移至Z；

将Y上1— n-1号圆盘从Y移至Z；

将规模为n的问题的求解分解为规模为n-1的问题的求解

有了问题的递归定义，很容易写出对应的递归算法：

```
void hanoi (int n, char x, char y, char z)
```

/*将塔座x上按直径由小到大且自上而下编号为1至n的n个圆盘按规则搬到塔座z上，y可用作辅助塔座。

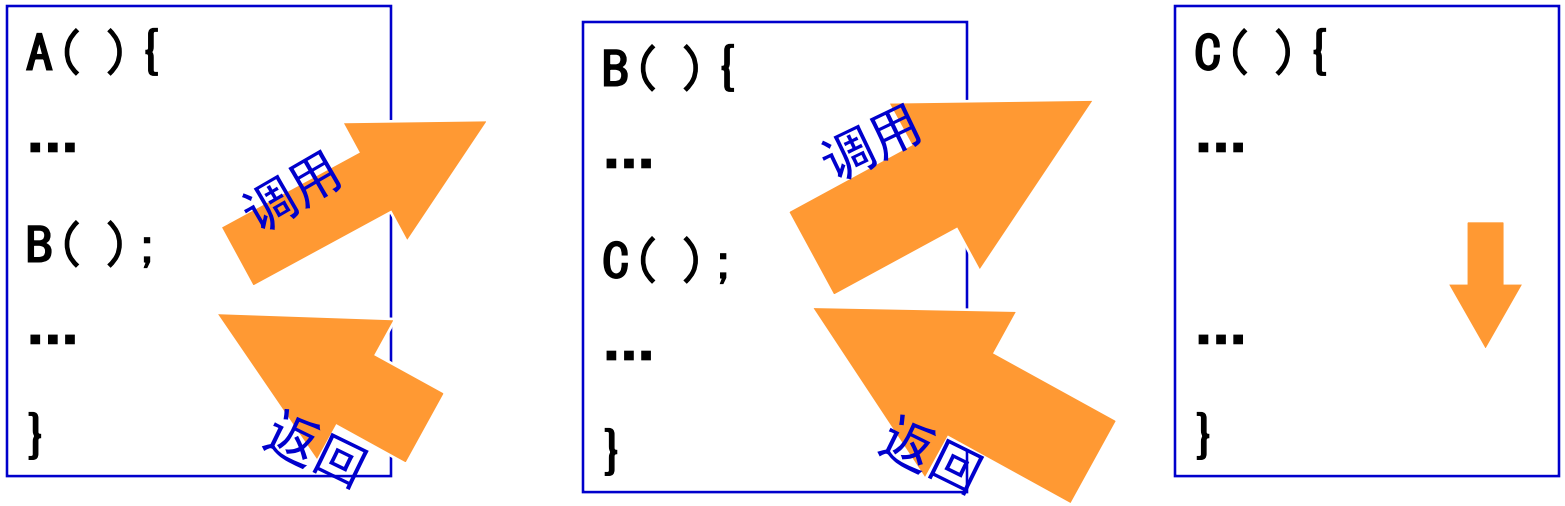
搬动操作move(x, n, z)可定义为(c是初值为0的全局变量，对搬动计数)：
printf("%d Move disk %di from %c to %c\n", ++c, n, x, z);*/

```
{
    if (n= =1)
            move(x, 1, z);    //将编号为1的圆盘从x移动z
    else {
      hanoi(n-1, x, z, y);    //将x上编号为1至n-1的圆盘移到y, z作辅助塔
      move(x, n, z);          //将编号为 n的圆盘从x移到z
      hanoi(n-1, y, x, z);    //将y上编号为1至n-1的圆盘移到z，x作辅助塔
    }
}
```

算法3.5 Hanci塔问题

3 递归函数的实现

在递归函数的执行中， 需多次自己调用自己，递归函数是如何执行的？先看一般函数的调用机制如何实现的。



函数调用顺序 A ⟶ B ⟶ C

函数返回顺序 C ⟶ B ⟶ A

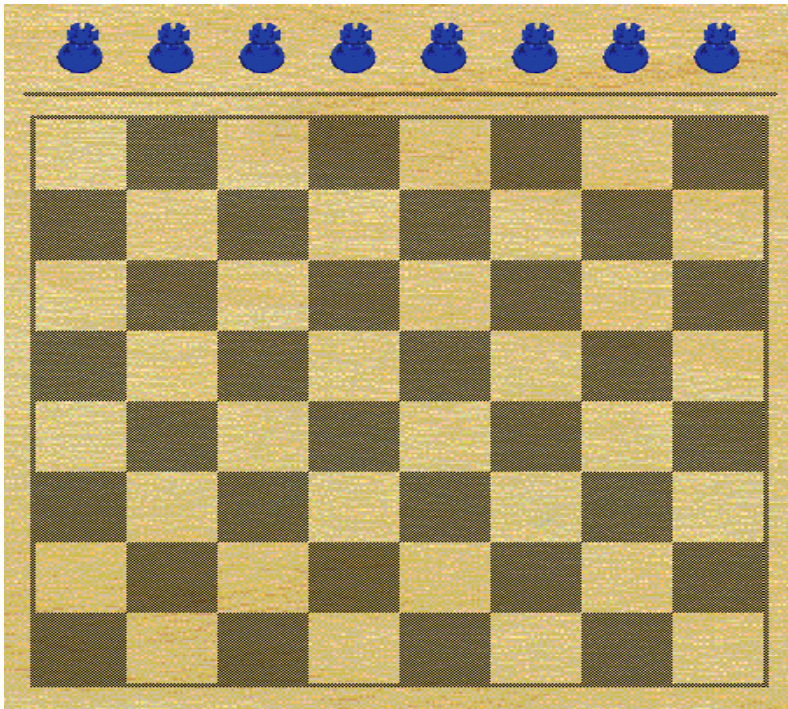
后调用的函数先返回

函数调用机制可通过栈来实现

计算机正是利用栈来实现函数的调用和返回的，函数之间的信息传递和控制转移通过“栈”来实现

八皇后问题

设在初始状态下在国际象棋棋盘上没有任何棋子(皇后)。然后顺序在第1行，第2行，…。第8行上布放棋子。在每一行中有8个可选择位置，但在任一时刻，棋盘的合法布局都必须满足3个限制条件，即任何两个棋子不得放在棋盘上的同一行、或者同一列、或者同一斜线上。试编写一个算法，求解并输出此问题的所有合法布局。



背包问题

设有一个背包可以放入的物品的重量为s，现有n件物品，重量分别为w[1]，w[2]，…，w[n]。问能否从这n件物品中选择若干件放入此背包中，使得放入的重量之和正好为s。

如果存在一种符合上述要求的选择，则称此背包问题有解(或称其解为真)；否则称此背包问题无解(或称其解为假)。试设计求解背包问题的算法

上面看到：栈结构后进先出的特征在程序设计中的应用，栈在实现函数递归调用中的作用；以及如何编写递归算法（递归函数）。

在后面的章节中，还将利用递归函数实现树和图的基本操作，这里同学们要好好理解如何编写递归算法（递归函数）。

小 结

1. 栈是限定仅能在表尾一端进行插入、删除操作的线性表；

2. 栈的元素具有后进先出的特点；

3. 栈顶元素的位置由一个称为栈顶指针的变量指示， 进栈、出栈操作要修改栈顶指针；

第三章　　习题一

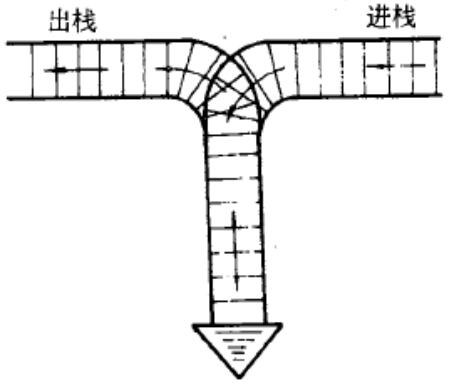
1、P21-24　　3.1，3.4，3.17

P21.

3.1　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)，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，并请说明为什么不能得到或者如何得到（即写出以’S’表示进栈和以’X’表示出栈的栈操作序列）。



3.4 简述以下算法的功能

（其中栈的元素类型SElemType为int）：

1.

```
status algo1 (Stack S){
    int i, n, A[255];
     n=0;
     while(!EmptyStack(S))   {n++;  Pop(S, A[n]); }
      for(i=1;  i<=n;  i++)    Push(S,  A[i]);
       }
```

2.

```
status algo2(Stack S,  int e){
   Stack T;  int d;
   InitStack(T);
      while(!EmptyStack(S)){
         pop(S,  d);
         if (d!=e)  Push( T,  d);
       }
      while(!EmptyStack(T))
     { Pop(T,  d);
       Push( S,  d);
     }
}
```

3. 17. 试写一个算法，判断依次读入的一个以@为结束符的字母序列，是否为形如‘序列1&序列2’模式的字符序列。其中序列1和序列2中都不含字符’&’，且序列2 是序列1的逆序列。例如，‘a+b&b+a’是属该模式的字符序列，而‘1+3&3-1’则不是。

[提示]:

1. 边读边入栈，直到&

2. 边读边出栈边比较，直到……

队 列

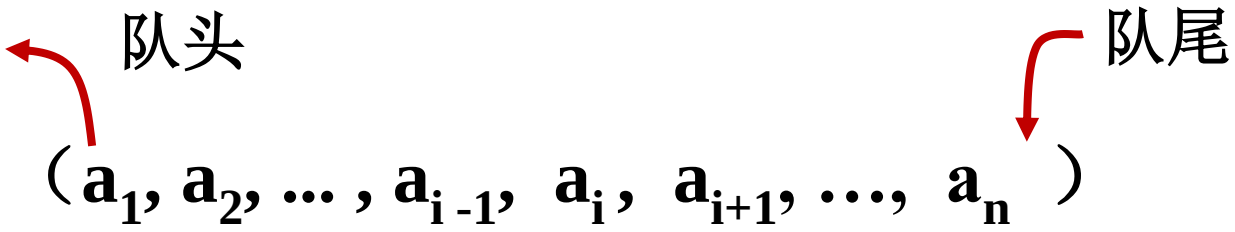
3.4 队列的定义和链队列

3.5 循环队列

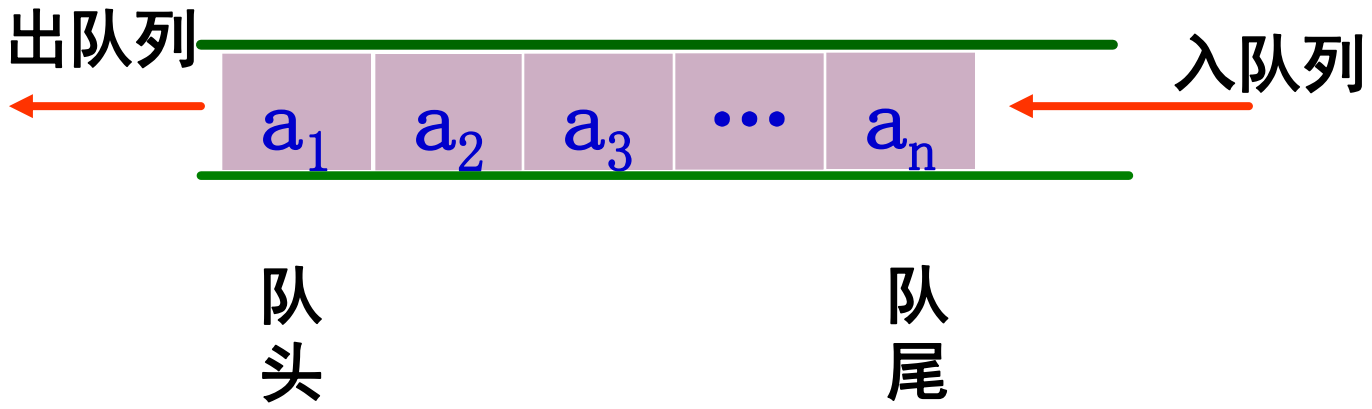
3.4　队列的定义和链队列

1.　队列的定义

队列(Queue)：　只允许在表的一端进行插入，而在另一端进行删除的线性表。允许插入的一端称为队尾(rear)，允许删除的一端称为队头(front)　。



- 当队列中没有元素时称为空队列。在空队列中依次加入元素$a_1,a_2,\ldots a_n$之后，a_1是队头元素，a_n是队尾元素。
- 队列也称为先进先出(First In First Out)线性表。



队列的基本操作:

- 初始化队列 InitQueue(&Q)：构造一个空队列Q。
- 销毁队列 DestroyQueue(&Q)：释放队列Q占用的内存空间。
- 清空队列 ClearQueue （ &Q ）：将队列Q重置为空。
- 判断队列是否为空 QueueEmpty(S)：若队列Q为空，则返回TRUE，否则FALSE。
- 求队列的长度 QueueLength(Q)：返回队列Q中元素个数。

- 取队头元素 GetHead(Q, &e)：用e返回Q的队头元素。
- 入队 EnQueue(&Q, e)：插入元素e为新的队尾元素。
- 出队 DeQueue(&Q, &e)：删除Q的队头元素，并用e返回其值。
- 遍历队列 QueueTraverse(Q)：从队头到队尾依次对Q的每个元素进行访问。

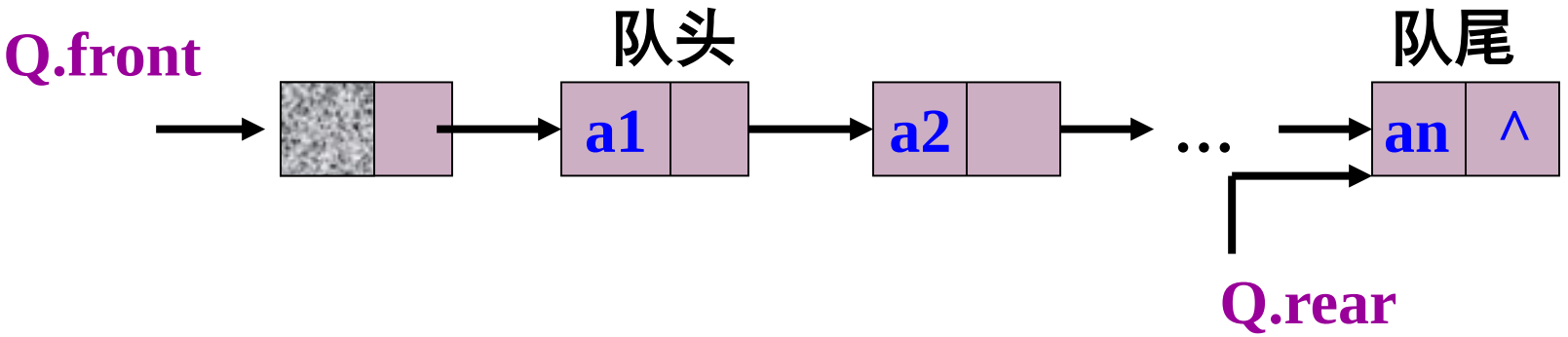
队列的存储结构

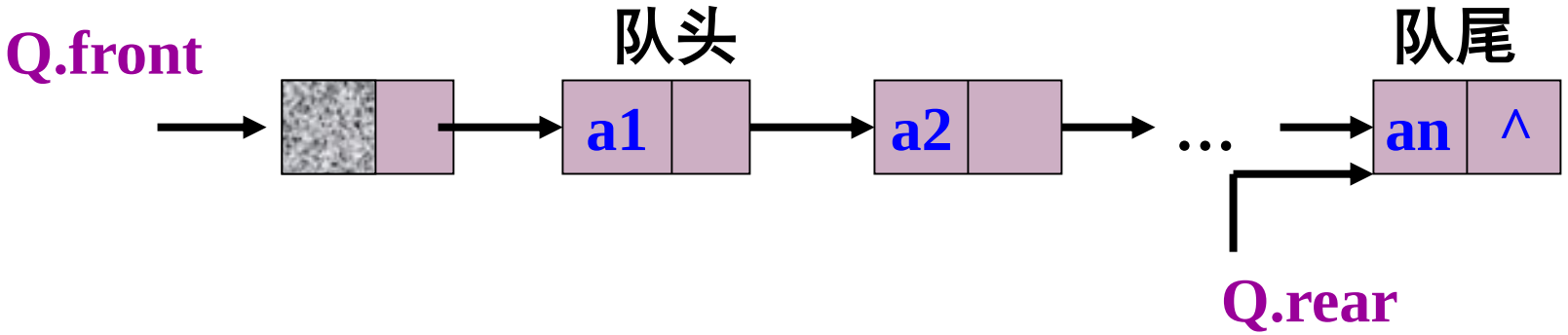
(1). 链队列—队列的链式表示

(2). 循环队列—队列的顺序表示

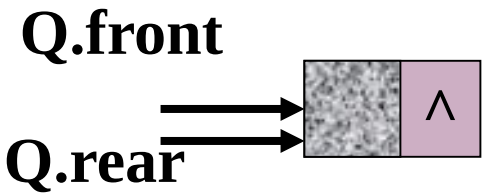
2 链队列—队列的链式表示和实现

用链表表示的队列简称为**链队列**。它是限制仅在**表头删除**和**表尾插入**的单链表。





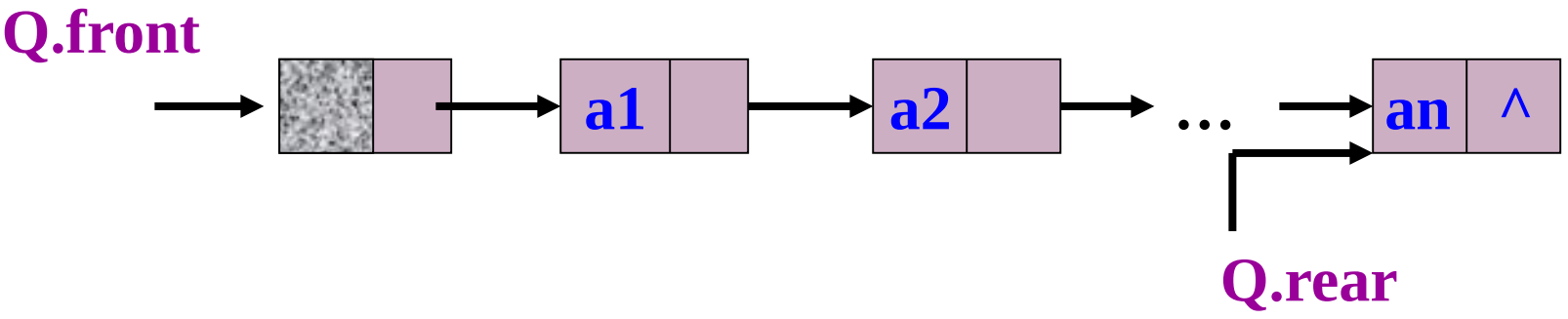
非空链队列



空队列

链队列的类型声明如下：

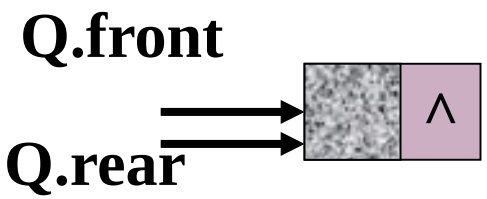
```
typedef struct QNode{    //链队列结点的类型
        QElemType   data;
        struct QNode   *next;
}QNode,*QueuePtr;

typedef struct{          //链队列的类型
      QueuePtr   front;    // 队头指针，指向链表的头结点
      QueuePtr   rear;     // 队尾指针，指向队尾结点
}LinkQueue;
```



链队列基本操作的实现

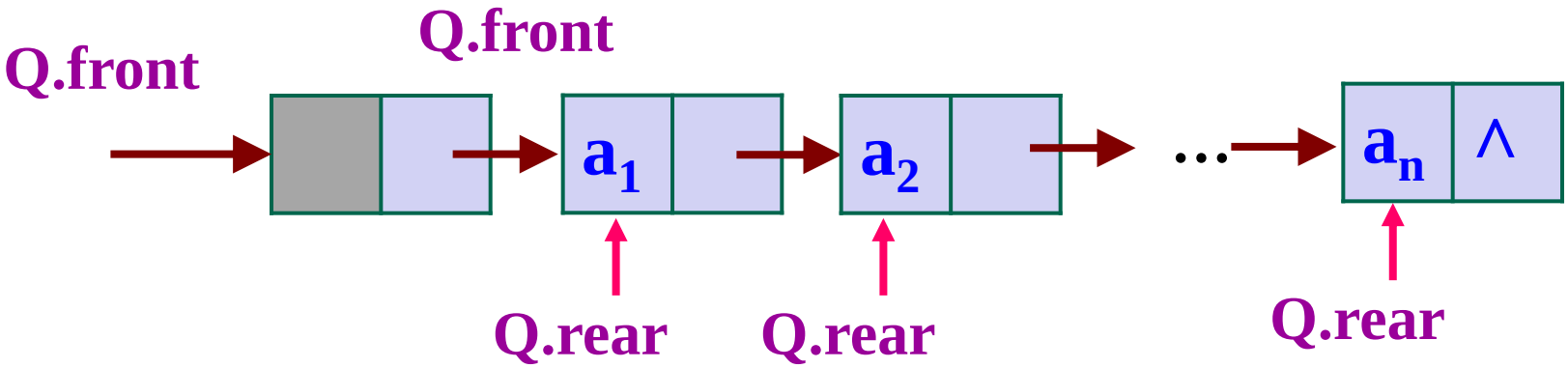
1. 初始化 InitQueue (&Q)

```
Status InitQueue(LinkQueue &Q)
{ //构造一个空队列Q
    Q.front=Q.rear=(QueuePtr)malloc(sizeof(QNode));
      //生成新结点作为头结点，队头和队尾指针都指向此结点
   Q.front->next=NULL;
  return OK;
}
```



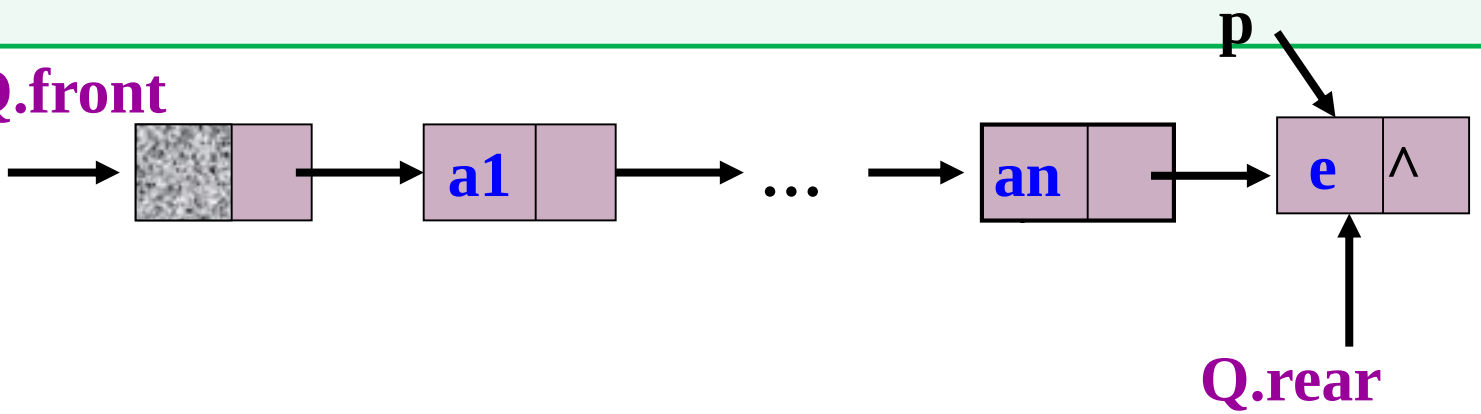
2. 销毁队列 DestroyQueue(&Q)

```
Status DestroyQueue_L(LinkQueue &Q)
{  //销毁队列Q
      while(Q.front)
       {
          Q.rear=Q.front->next;
          free(Q.front);
          Q.front=Q.rear
       }
    return OK;
}
```



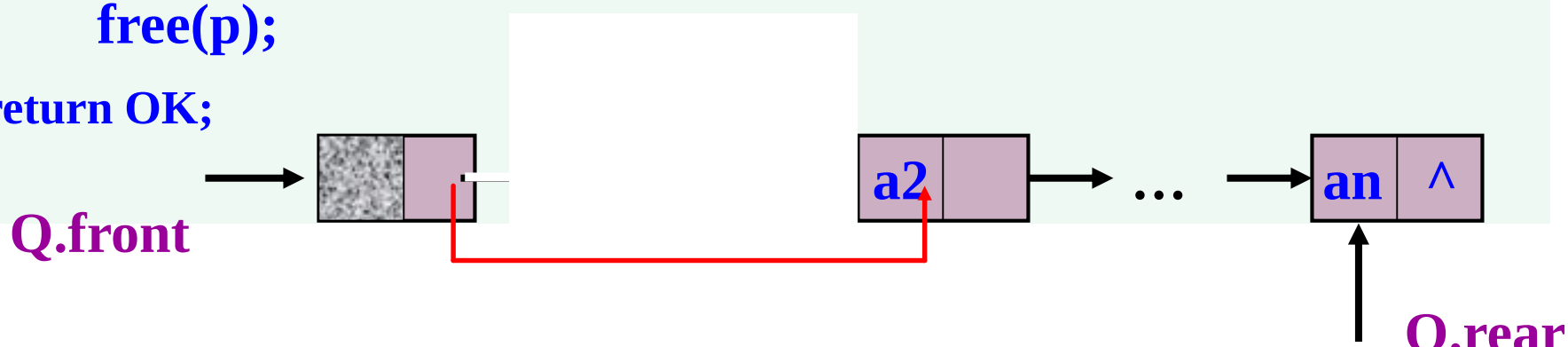
3. 入队 EnQueue(&Q, e)

```
void EnQueue(LinkQueue &Q,QElemType e)
 {//插入元素e为Q的新的队尾元素
     p=(QueuePtr)malloc(sizeof(QNode)); //为e分配结点空间
     p->data=e;     //将新结点数据域置为e
     p->next=NULL;
     Q.rear->next=p;                    //将新结点插入到队尾
     Q.rear=p;                          //修改队尾指针
}
```



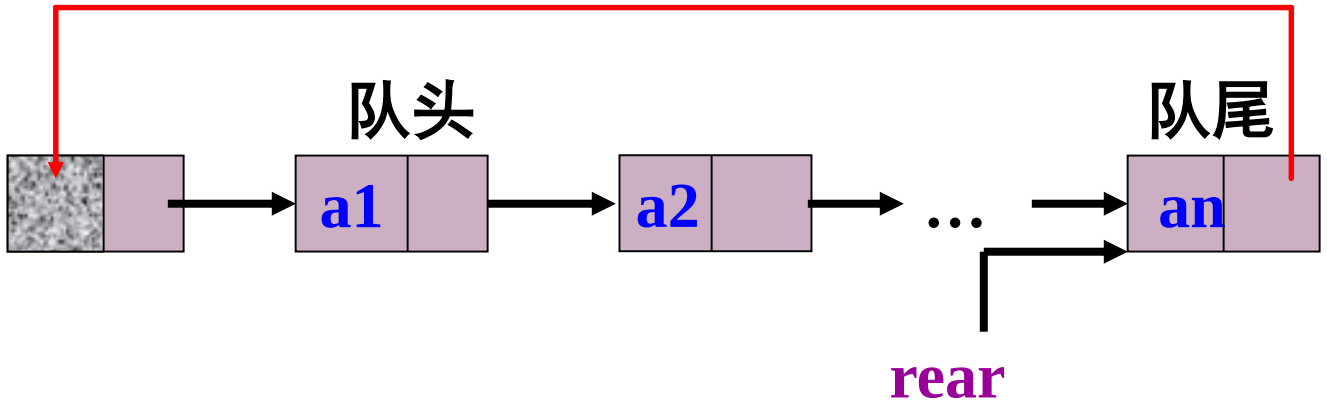
4. 出队 DeQueue(&Q, &e)

```
Status DeQueue(LinkQueue &Q, QElemType &e)
 {//删除Q的队头元素,用e返回其值
       if(Q.front==Q.rear)return ERROR; //若队列空，返回ERROR
       p=Q.front->next;              // p指向队头元素结点
       e=p->data;
        Q.front->next=p->next;     // 修改链队列头结点指针
        if(Q.rear==p)   Q.rear=Q.front;
        // 对于链队列只有一个元素结点的情况要同时修改队尾指针
        free(p);
 return OK;
}
```



思考题：

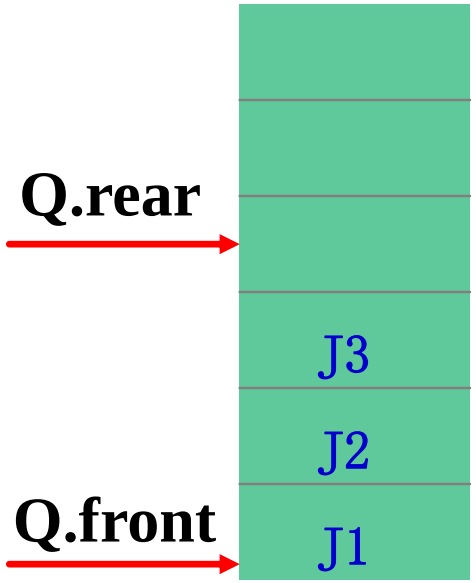
假设以带头结点的循环链表表示队列，并且只设一个指针指向队尾元素结点（注意不设头指针），相应的队列的操作，如初始化、入队、出队等如何来具体实现？



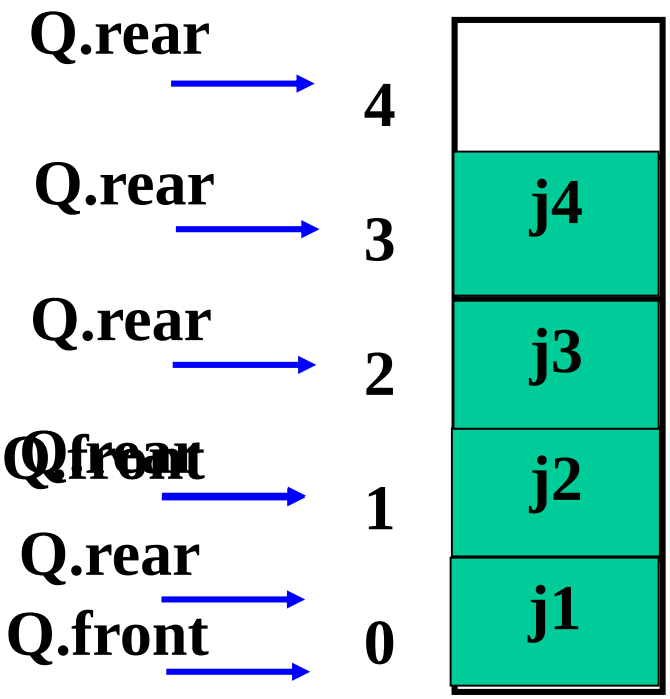

```
typedef struct QNode{    //链队列结点的类型
        QElemType   data;
        struct QNode   *next;
}QNode,*QueuePtr;
```

4.5 循环队列—队列的顺序表示和实现

队列的顺序存储结构称为顺序队列，是用一组地址连续的存储单元依次存放从队列头到队列尾的元素。

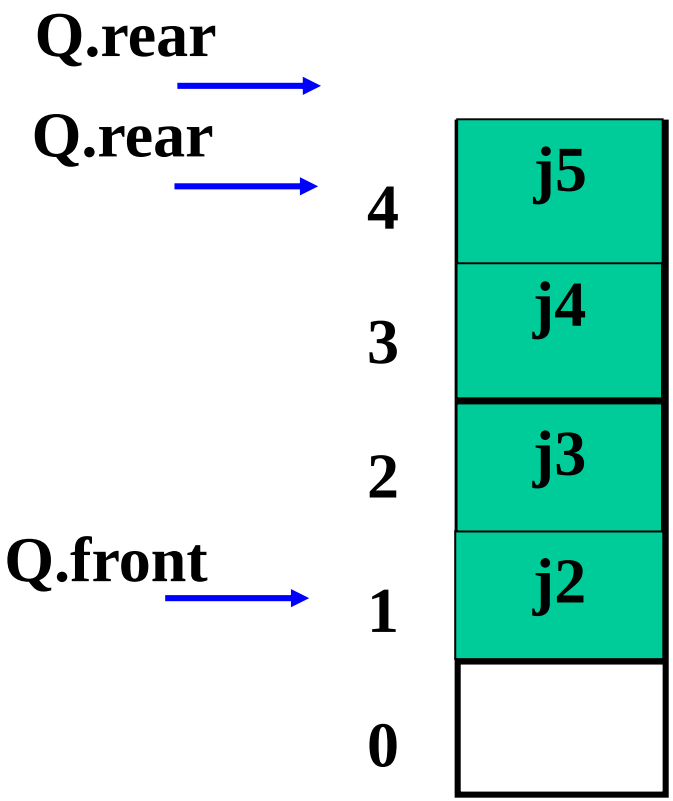


由于队列的队头和队尾元素的位置是变化的，因而要设两个指针front和rear分别指示队头和队尾元素的位置。



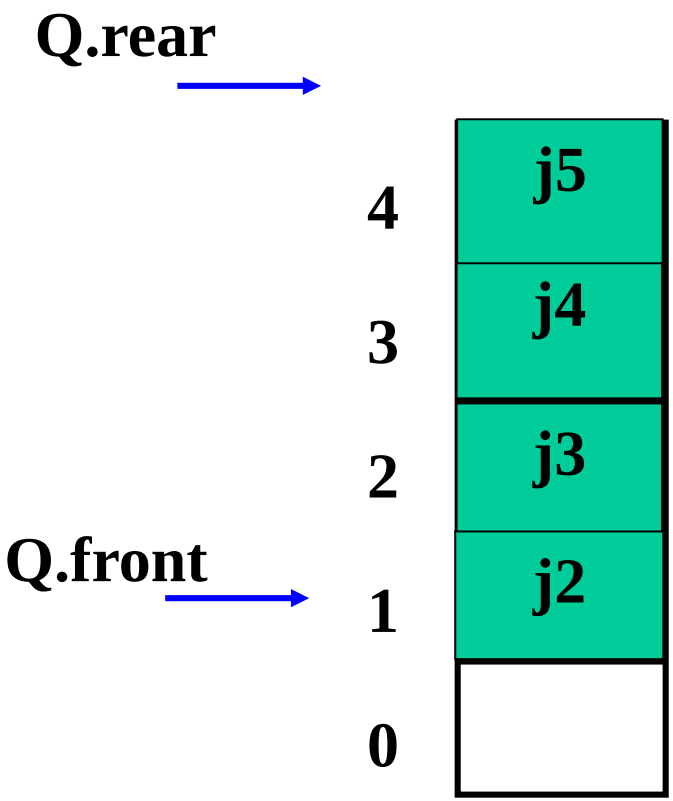
顺序队列中头、尾指针和元素之间的关系

- 初始化建空队列时，令front=rear=0。
- 每当插入新的队列尾元素时，尾指针加1;
- 每当删除队列头元素时，头指针加1。



顺序队列中头、尾指针和元素之间的关系

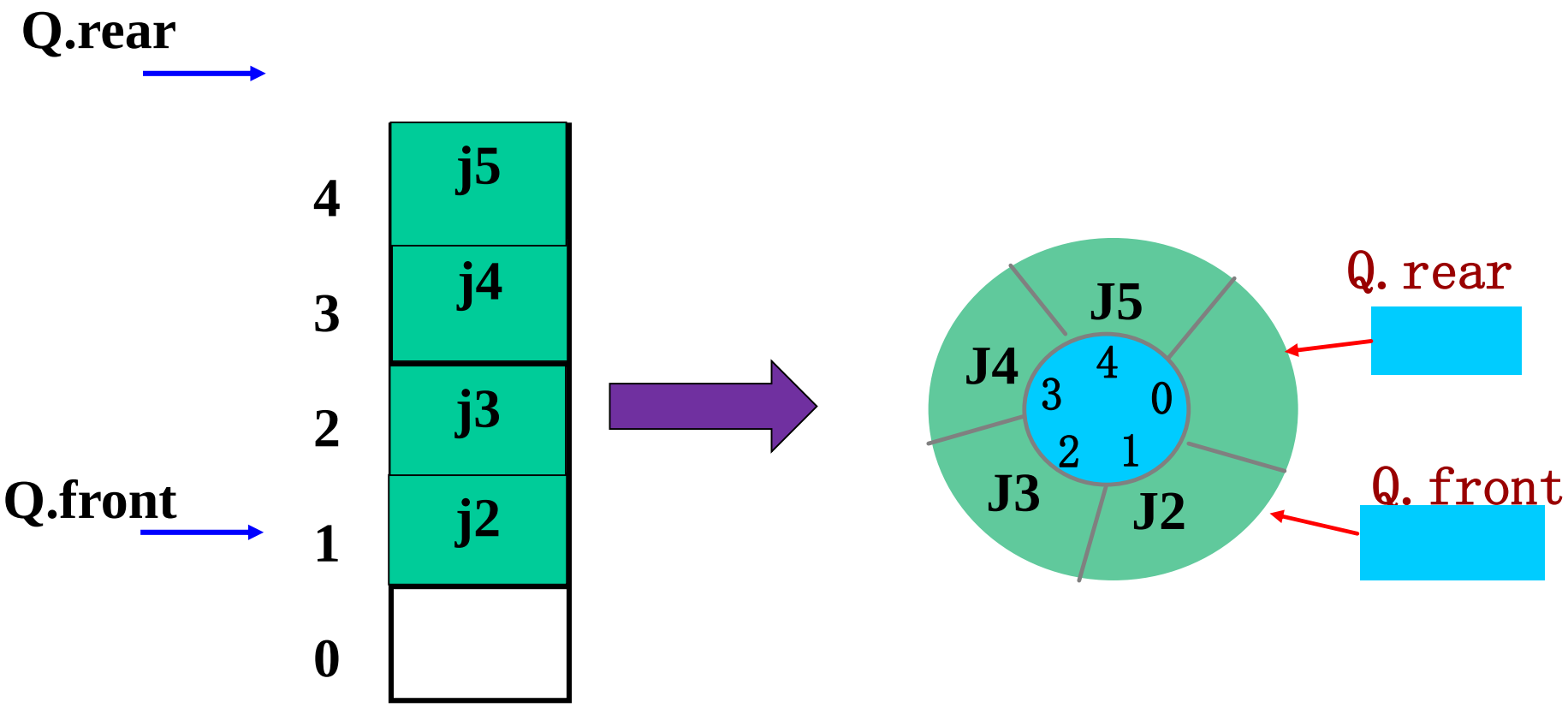
在非空队列里，头指针始终指向队头元素，而尾指针始终指向队尾元素的下一位置。



顺序队列中头、尾指针和元素之间的关系

假溢出：从图示中可看出，当J5 入队后，队尾指针Q.rear越界，不可能再插入新的队尾元素，但是另一方面，队列的实际可用空间并未占满。

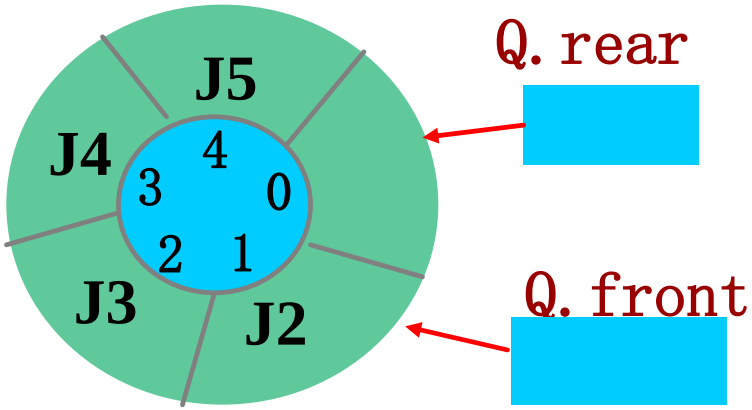
解决办法：将顺序队列设想为首尾相连的环状空间，如图，当Q.rear值超出队列空间的最大位置时，令Q.rear= 0 ，使队列空间能“循环”使用。这种循环使用空间的队列称为循环队列。

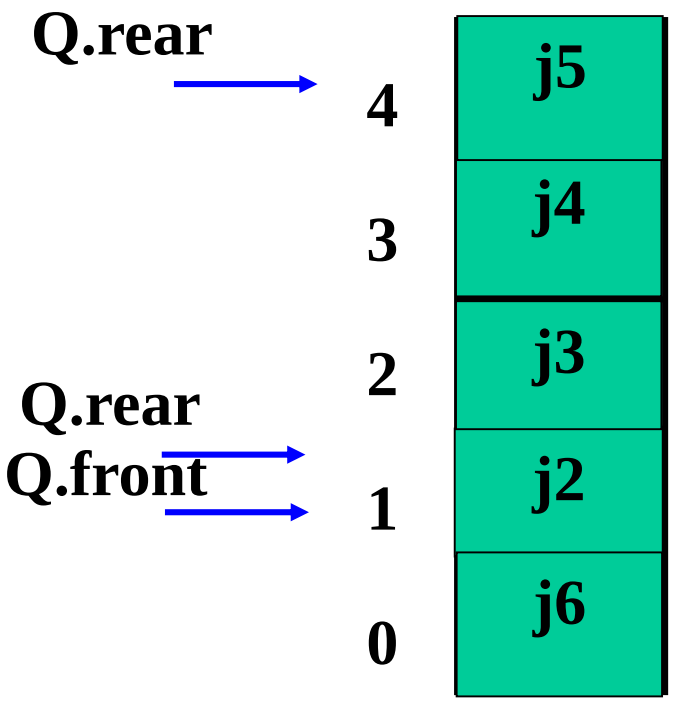


这种循环意义下的加1操作可以描述为：

```
if(i+1== MAXSIZE)
      i=0;
else
      i++;
```

利用模运算可简化为：　i=(i+1)% MAXSIZE



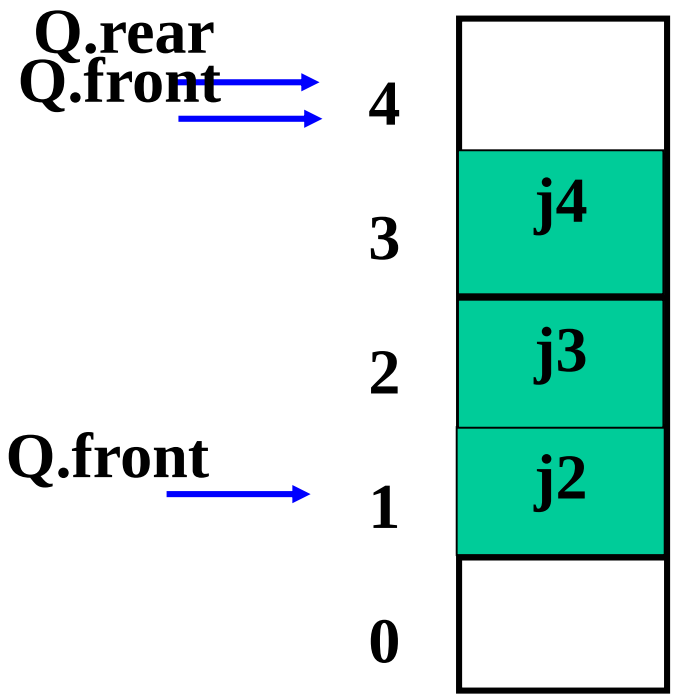


循环队列中头、尾指针和元素之间的关系

情况1：

(1). 初始情况，j2、 j3、 j4在队列中

(2). J5、J6入队后，队满：Q.front==Q.rear



情况2：

循环队列中头、尾指针和元素之间的关系

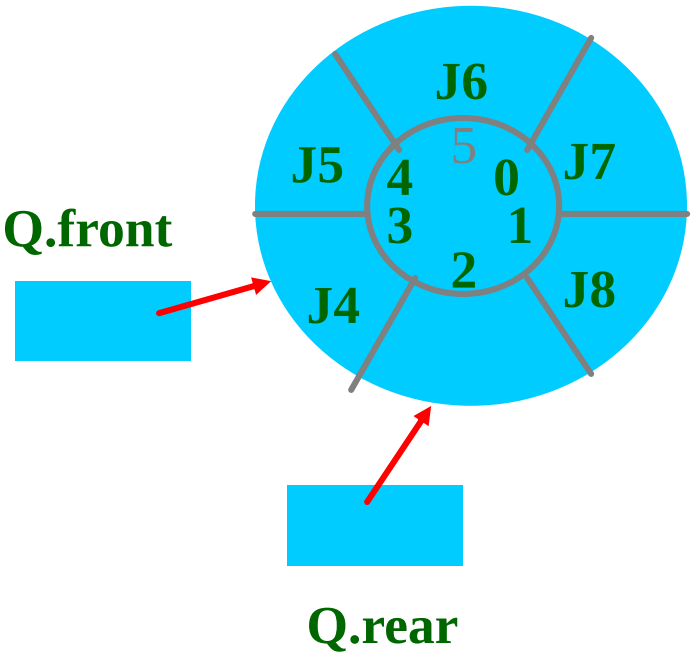
(1). 初始情况，j2、 j3、 j4在队列中

(2). J2、J3、J4出队后，队空：Q.front==Q.rear

队空和队满时头尾指针均相等。因此，我们无法通过是否front=rear来判断队列“空”还是“满”。

解决此问题的方法：

一、少用一个元素的空间。当尾指针在循环意义下加1后如等于头指针，则认为队满（注意：rear所指的单元始终为空）；

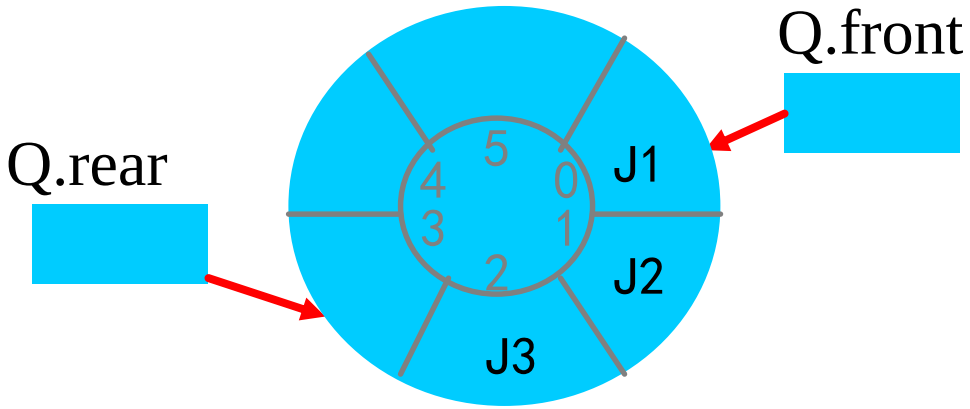


二、另设一个标志位以区分队空、队满；

三、使用一个计数器记录队列中元素的总数。

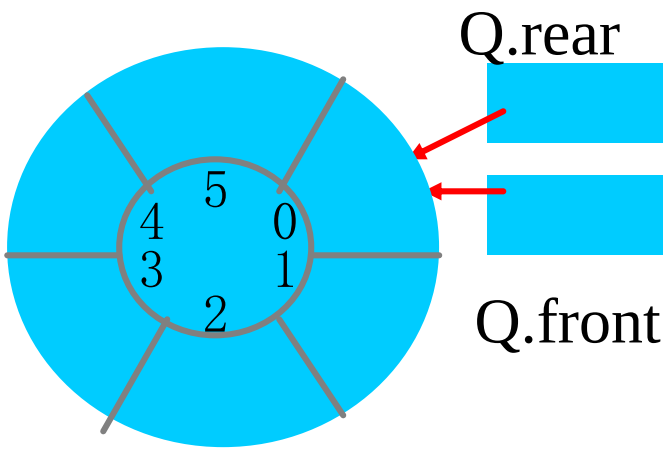
循环队列的表示：

```
# define MAXSIZE 100   //最大队列长度
typedef struct {
    QElemType  *base; // 动态分配存储空间
 QElemType *front; // 头指针，指示队列头元素的位置
QElemType  *rear; // 尾指针，指示队列尾元素的下一个位置
}SqQueue;
```



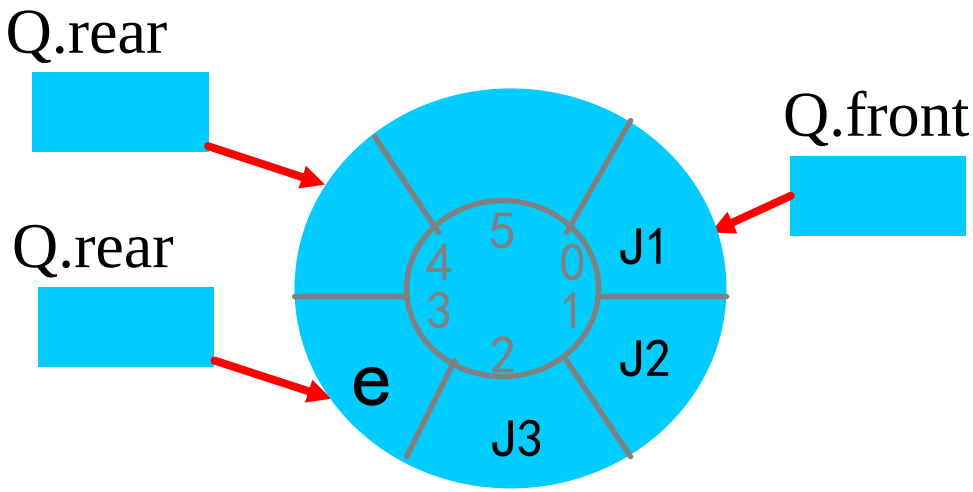
循环队列基本操作的实现

1. 初始化 InitQueue (&Q)



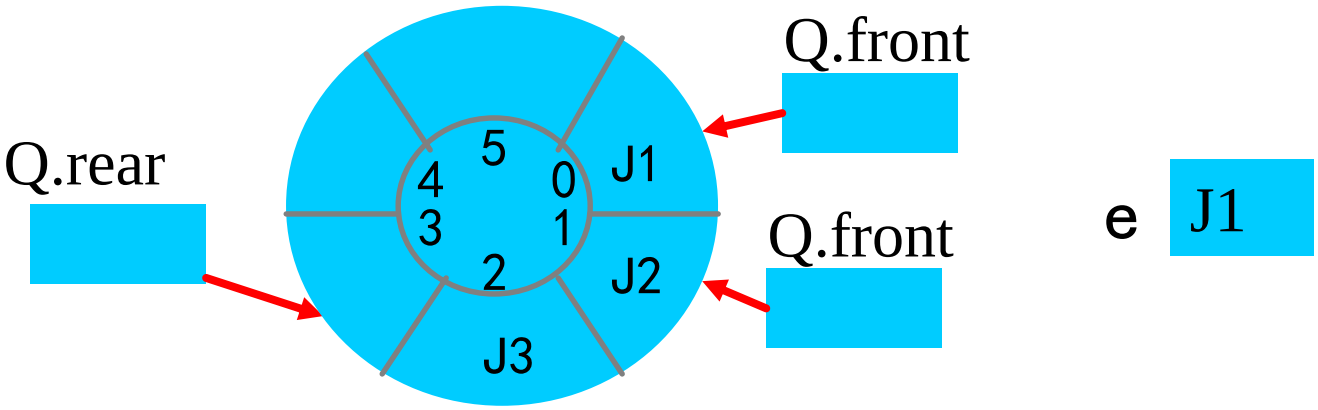

```
Status InitQueue_Sq(SqQueue &Q) {
    //构造一个空队列Q
    Q.base=(QElemType * )malloc (MAXQSIZE *sizeof (QElemType));
        //分配一个最大容量为MAXSIZE的数组空间
    if (!Q.base) exit (OVERFLOW);  //存储分配失败
    Q.front = Q.rear=0;  //队列为空，头指针和尾指针都为零
    return OK;
}
```

2. 入队EnQueue (&Q, e)




```
Status EnQueue (SqQueue  &Q, QElemType  e )
   if ((Q.rear+1)%MAXSIZE= =Q.front)  return ERROR ;
   Q.base[Q.rear] = e ;      // 将元素e插入队尾
  Q.rear= (Q.rear+1)%MAXSIZE;    // 修改队尾指针

return OK;
}
```

3. 出队 DeQueue(&Q, &e)




```
Status DeQueue (SqQueue  &Q, QElemType &e )
      //删除队头元素，用e返回其值
      if ((Q.rear= =Q.front) return ERROR ;
      e =Q.base[Q.front] ;
      Q.front= (Q.front+1)%MAXSIZE;        // 修改队头指针
      return OK;
}
```

队列的应用

队列与栈一样，也是程序设计中经常使用的数据结构，凡是符合先进先出原则的数学模型，都可以用队列。例如：

在操作系统中用来

- 解决计算机主机与外设不匹配的问题
- 解决由于多用户引起的资源竞争问题

在实际应用中用来

- 模拟各种排队现象

广度优先搜索

判断题：

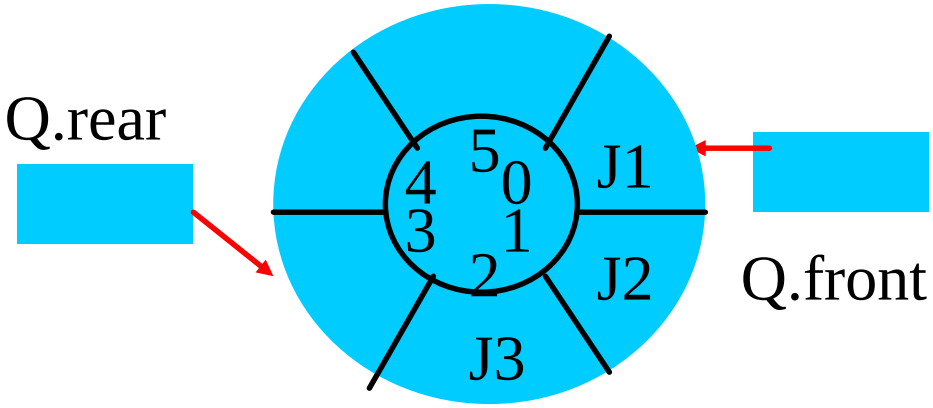
1.为解决计算机主机与打印机之间速度不匹配问题，通常设置一个打印数据缓冲区，主机将要输出的数据依次写入该缓冲区，而打印机则依次从该缓冲区中取出数据。该缓冲区的逻辑结构应该是（　　）。

A. 栈　　B. 队列　　C. 树　　D. 图

第三章　　习 题 二

1. P25　　3.29　　3.30

3．29如果希望循环队列中的元素都能得到利用，则需设置一个标志域tag，并以tag的值为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，并从时间和空间角度讨论设标志和不设标志这两种方法的使用范围(如当循环队列容量较小而队列中每个元素占的空间较多时，哪一种方法较好)。



3.30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。