3.4 队列

3.4.1 抽象数据类型队列的定义

队列（queue）是一种先进先出(first in first out，简称FIFO表)的线性表。它只允许在表的一端进行插入，而在另一端删除元素。

在队列中，允许插入的一端叫做队尾(rear)，允许删除的一端称为队头(front)。

假设队列为q=（$a_1,a_2,...a_n$），则a_1为队头元素，a_n为队尾元素。

抽象数据类型队列的定义

ADT Queue **{**

数据对象：

$D = \{a_i \mid a_i \in ElemSet, i=1,2,...,n, n \geq 0\}$

数据关系：

$R1 = \{ <a_{i-1}, a_i> \mid a_{i-1}, a_i \in D, i=2,...,n\}$

约定：其中**a_1端为队列头**，**a_n端为队列尾**

基本操作：

} **ADT** Queue

队列的基本操作：

InitQueue(&Q)

操作结果：构造一个空队列Q。

DestroyQueue(&Q)

初始条件：队列Q已存在。

操作结果：队列Q被销毁，不再存在。

QueueEmpty(Q)

初始条件：队列Q已存在。

操作结果：若Q为空队列，则返回TRUE，否则返回FALSE。

QueueLength(Q)

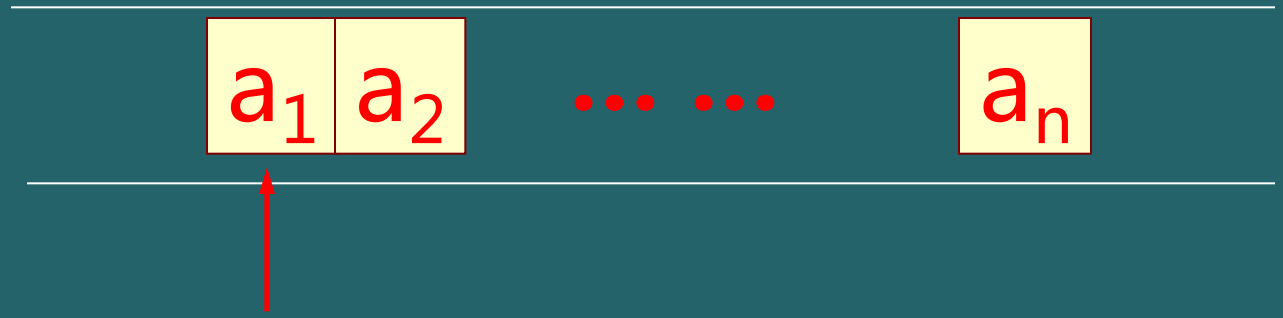
初始条件：队列Q已存在。

操作结果：返回Q的元素个数，即队列的长度。

GetHead(Q, &e)

初始条件：Q为非空队列。

操作结果：用e返回Q的队头元素。



ClearQueue(&Q)

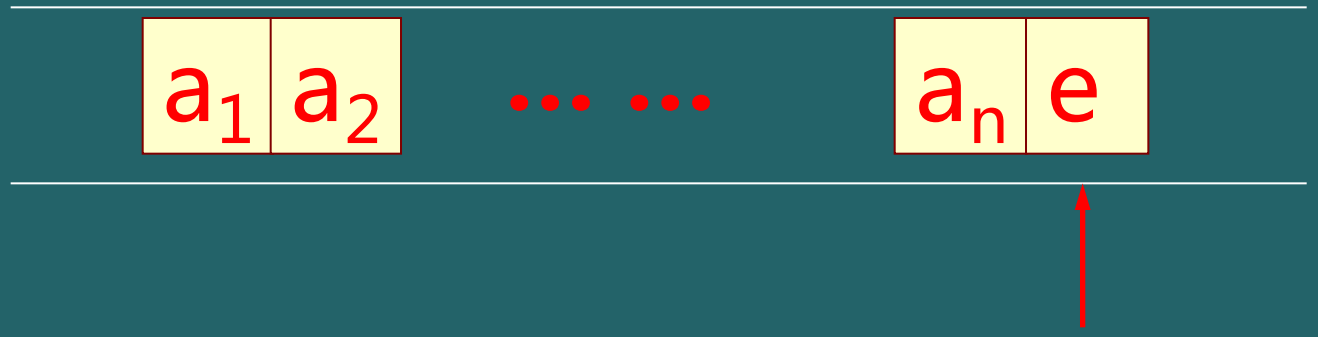
初始条件：队列Q已存在。

操作结果：将Q清为空队列。

EnQueue(&Q, e)

初始条件：队列Q已存在。

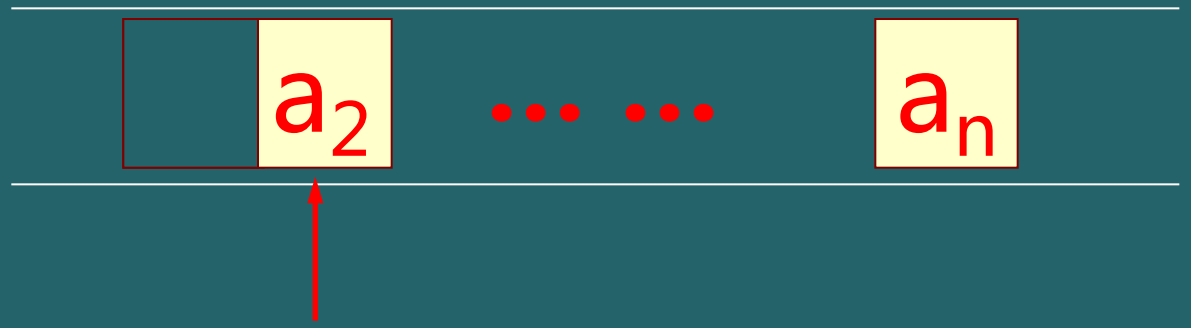
操作结果：插入元素e为Q 的新的队尾元素。



DeQueue(&Q, &e)

初始条件：Q为非空队列。

操作结果：删除Q的队头元素，并用e返回其值。



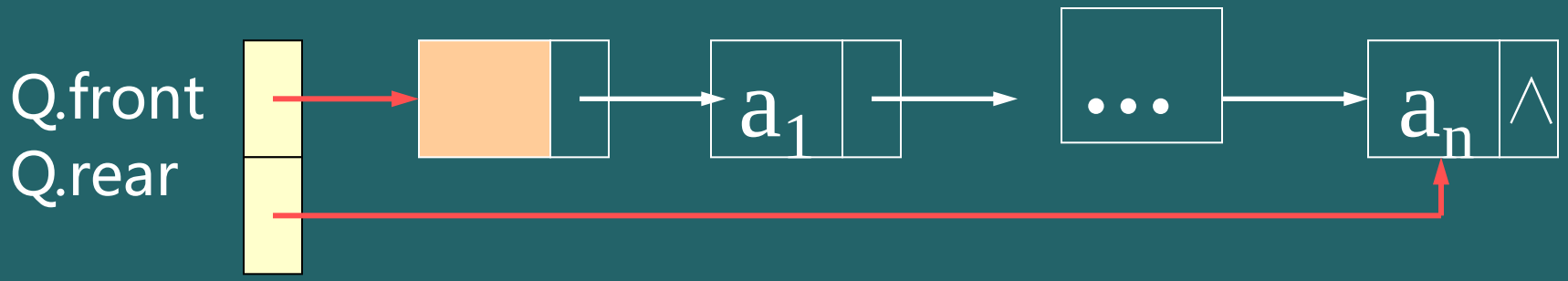
3.4.2 链队列——队列的链式表示和实现

链队列: 用链表表示的队列简称为链队列。

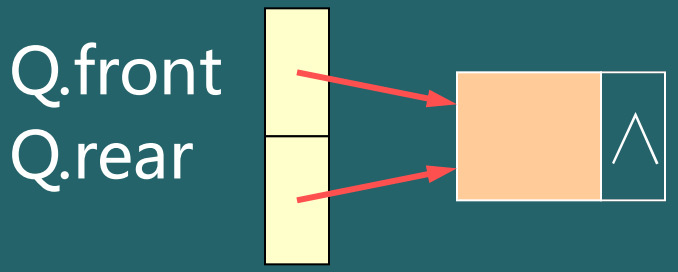
一个链队列需要两个分别指示队头和队尾的指针（分别称为头指针和尾指针）才能惟一确定。

```
typedef struct QNode {// 结点类型
  QElemType      data;
  struct QNode  *next;
 } QNode, *QueuePtr;
```

```
typedef struct { // 链队列类型
    QueuePtr  front;  // 队头指针
    QueuePtr  rear;   // 队尾指针
} LinkQueue;
```



空队列



-----基本操作的函数原型说明-----

Status InitQueue (LinkQueue **&**Q);
// 构造一个空的队列Q

Status DestroyQueue (LinkQueue **&**Q);
// 销毁队列Q，Q不再存在

Status ClearQueue (LinkQueue **&**Q);
// 重置Q为空队列

Status QueueEmpty (LinkQueue Q);
//若为空队列，则返回TRUE，否则 返回FALSE

int QueueLength (LinkQueue Q);
//返回Q的元素个数，即队列的长度

Status GetHead (LinkQueue Q , QElemType &e)
//若队列不空，则用e返回Q的队头元素，并返回OK，否则返回ERROR

Status EnQueue (LinkQueue **&**Q , QElemType e)
// 插入元素e为新的队尾元素

Status De LinkQueue (LinkQueue **&**Q , QElemType &e)
//若队列不空，则删除Q的队头元素，并用e返回其值，并返回OK，否则返回ERROR

Status Queue Traverse (LinkQueue **&**Q , visit());
//从队尾到队头依次对队列中每个元素调用函数visit()。一旦visit()失败，则操作失败

```
Status InitQueue (LinkQueue &Q) {
  // 构造一个空队列Q
  Q.front = Q.rear =
(QueuePtr)malloc(sizeof(QNode));
  if (!Q.front) exit (OVERFLOW);
//存储分配失败
  Q.front->next = NULL;
  return OK;
}
```

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

```
Status EnQueue (LinkQueue &Q,
QElemType e) {
    // 插入元素e为Q的新的队尾元素
    p = (QueuePtr) malloc (sizeof (QNode));
    if(!p) exit (OVERFLOW);  //存储分配失败
    p->data = e;    p->next = NULL;
    Q.rear->next = p;     Q.rear = p;
    return OK;
}
```

```
Status DeQueue (LinkQueue &Q,
QElemType &e) {
  // 若队列不空，则删除Q的队头元素，
  //用 e 返回其值，并返回OK；否则返回ERROR
   if (Q.front == Q.rear)    return ERROR;
   p = Q.front->next;   e = p->data;
   Q.front->next = p->next;
   if (Q.rear == p)  Q.rear = Q.front;
   free (p);      return OK;
}
```

3.4.3 循环队列——队列的顺序表示和实现

在队列的顺序存储结构中，需附设两个指针front和rear分别指示队列头元素及队列尾元素的位置。在此约定：初始化建空队列时，令front=rear=0，每当插入新的队尾元素时，‘尾指针加1’，每当删除队头元素时，‘头指针加1’。因此，在非空队列中，头指针始终指向队列头元素，而尾指针始终指向队尾元素的下一个位置。

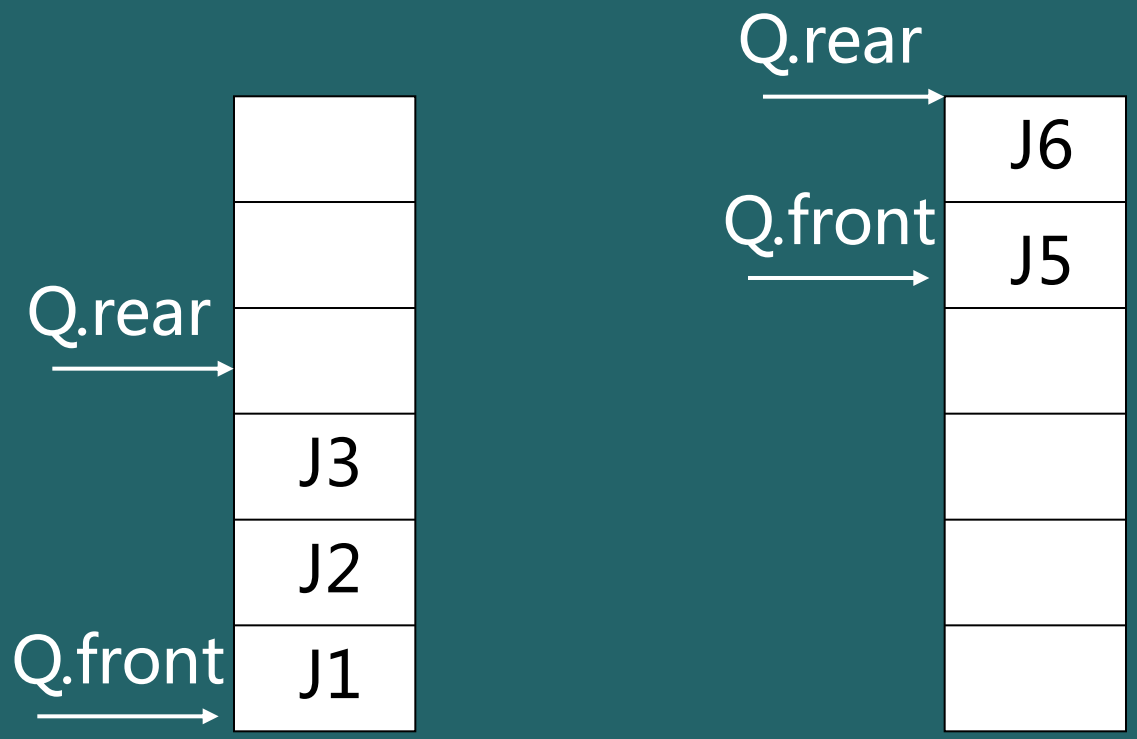


图3.12 头、尾指针和队列中元素之间的关系

循环队列类型的模块说明如下：

```
#define MAXQSIZE  100  //最大队列长度
  typedef struct {
    QElemType *base; // 动态分配存储空间
    int front;   // 头指针，若队列不空，
                 //  指向队列头元素
    int  rear;  // 尾指针，若队列不空，    //指向队列尾元素的下
一个位置
  } SqQueue;
```

```
Status InitQueue (SqQueue &Q) {
  // 构造一个空队列Q
  Q.base = (ElemType *) malloc
           (MAXQSIZE *sizeof (ElemType));
   if (!Q.base) exit (OVERFLOW);
// 存储分配失败
   Q.front = Q.rear = 0;
    return OK;
}
```

```
Status EnQueue (SqQueue &Q, ElemType e) {
    // 插入元素e为Q的新的队尾元素
    if ((Q.rear+1) % MAXQSIZE == Q.front)
        return ERROR;     //队列满
    Q.base[Q.rear] = e;
    Q.rear = (Q.rear+1) % MAXQSIZE;
    return OK;
}
```

```
Status DeQueue (SqQueue &Q, ElemType &e) {
  // 若队列不空，则删除Q的队头元素，
 // 用e返回其值，并返回OK; 否则返回ERROR
  if (Q.front == Q.rear)  return ERROR;
  e = Q.base[Q.front];
  Q.front = (Q.front+1) % MAXQSIZE;
  return OK;
}
```

二项式系数值（杨辉三角）

第 1 行　　1　　1

第 2 行　　1　　2　　1

第 3 行　　1　　3　　3　　1

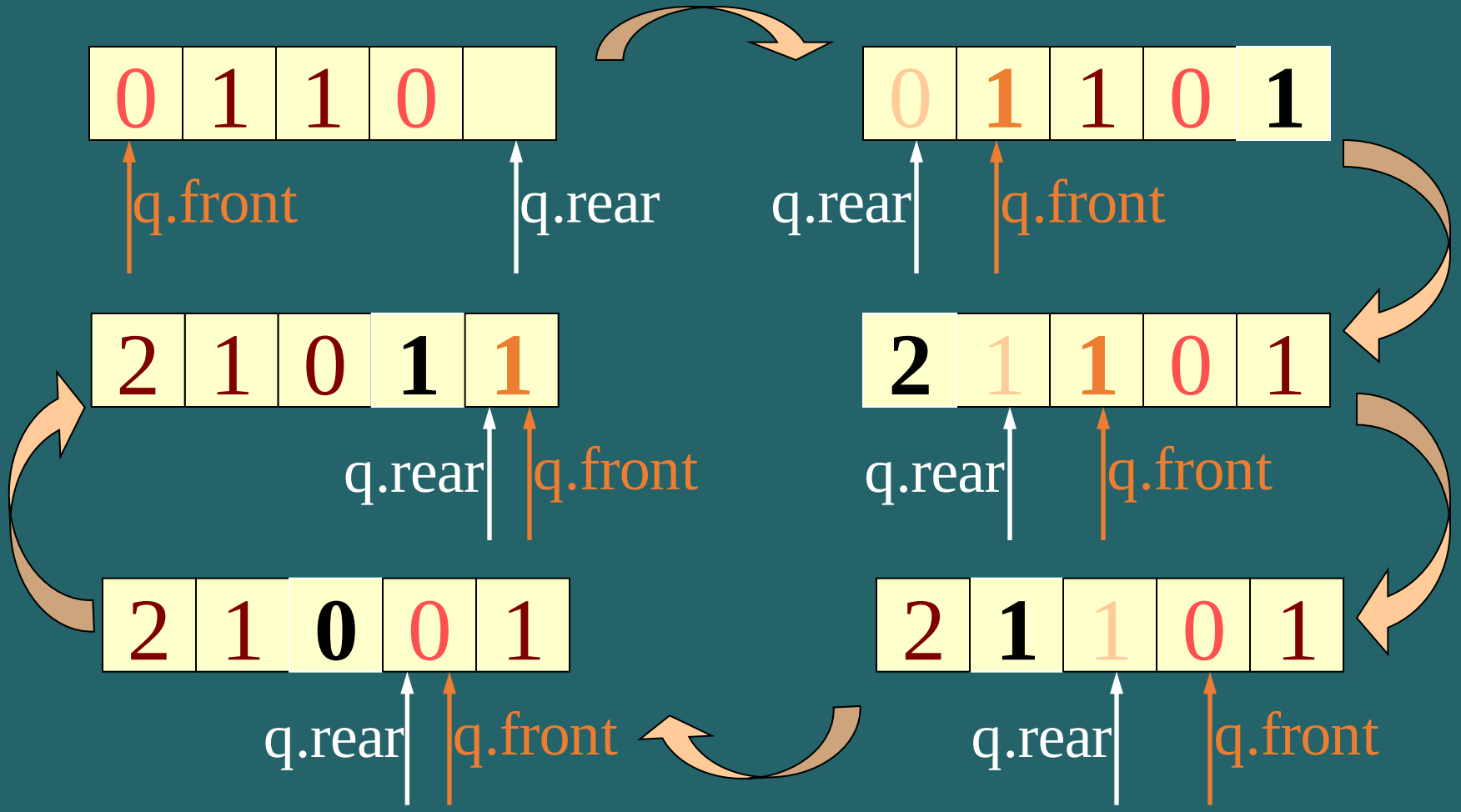
第 4 行　　1　　4　　6　　4　　1

设第 i-1行的值：(a[0]=0) a[1]..a[i] (a[i+1]=0)

则第 i 行的值：b[j] = a[j-1]+a[j], j=1,2,...,i+1

利用循环队列计算二项式的过程：

假设只计算三行，则队列的最大容量为 5。



```
do {
    DeQueue(Q, s);
    GetHead(Q, e);
    if (e!=0) printf ( "%d" , e);
    EnQueue(Q, s+e);
} while (e!=0);
```