3.4 队列的逻辑结构

队列的基本概念和术语

- 队列（Queue）：限定仅在表的一端进行插入，而另一端进行删除操作的线性表。
- 队尾（rear）：允许插入的一端。
- 队头（front）：允许删除的一端。
- 空队：空表。

队列操作特点：先进先出（First In First Out---FIFO）

3.4 队列的逻辑结构（续）

队列的抽象数据类型定义

ADT Queue {

数据对象：D={a_i | a_i属于ElemSet, i = 1,2, … ,n, n>=0}

数据关系：R1={< a_{i-1}, a_i >| a_{i-1}, a_i属于D, i =2,3, …, n}// a_1 为队头， a_n 队尾

基本操作：

InitQueue(&Q)

DestroyQueue(&Q)

ClearQueue(&Q)

QueueEmpty(Q)

QueueLength(Q)

GetHead(Q,&e)

初始条件：Q存在,非空；

操作结果：用e返回Q的栈顶元素

EnQueue(&Q,e) ---------- 入队（插入）

初始条件：Q存在

操作结果：插入元素e为新的队尾元素，长度加1

DelQueue (&Q, &e) ---------- 出队（删除）

初始条件：Q存在；非空

操作结果：删除Q的队头元素，e返回值，长度减1

}ADT Queue

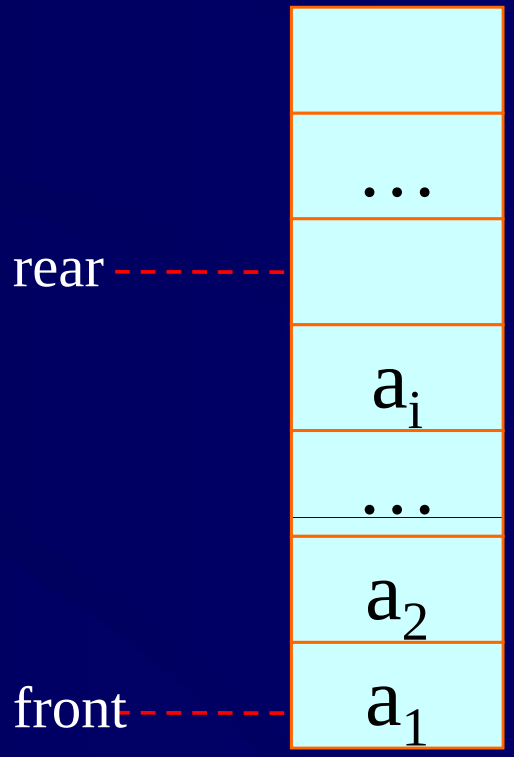
3.5 队列的存储结构

队列的顺序存储---循环队列（一）

一般顺序存储的队列特点：

- **队空条件：** front = rear
- **队满条件：** rear = MAXSIZE
- **队满条件下的队满为假满（假溢出）**

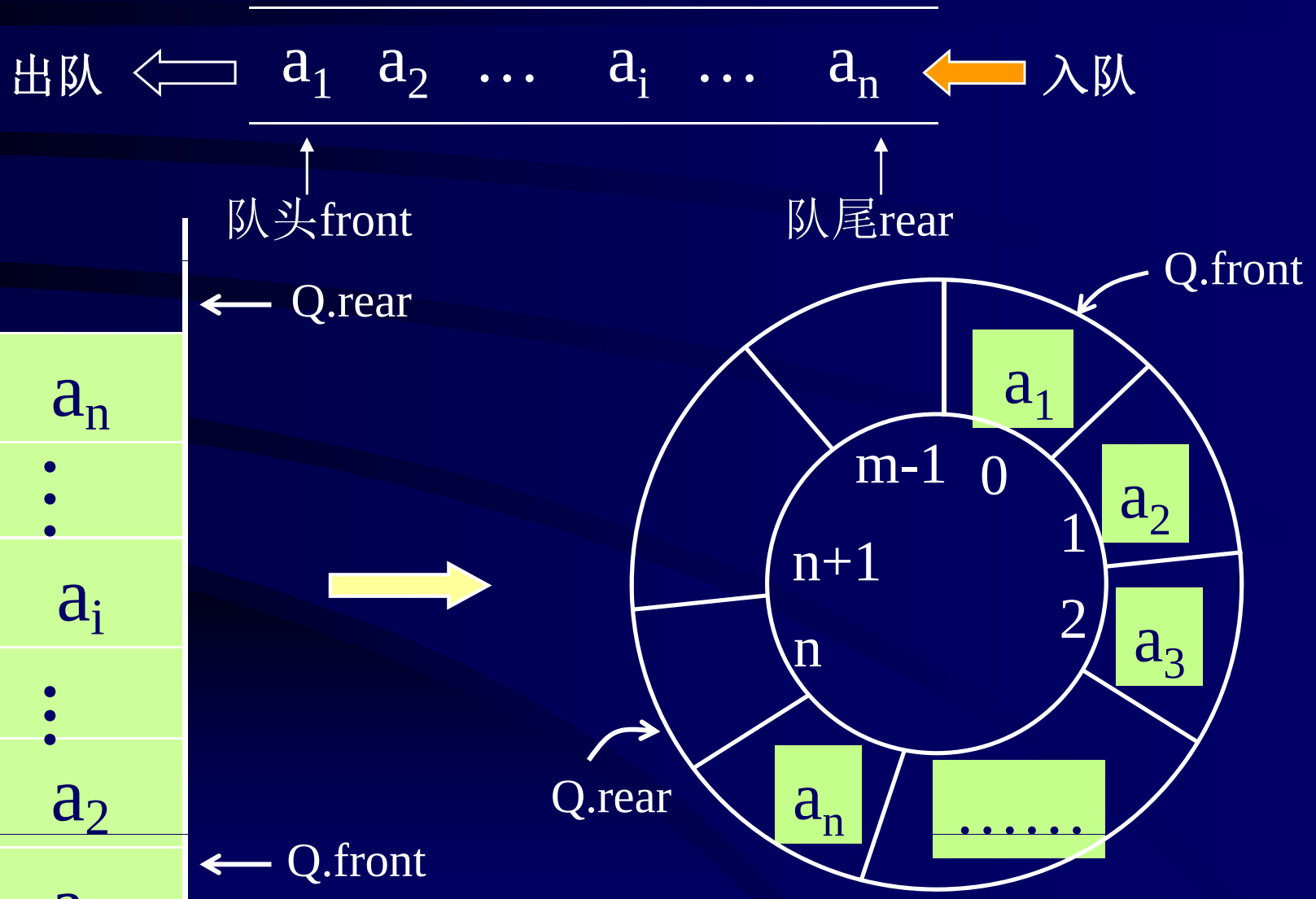
（真满时：rear = MAXSIZE, front = 0）



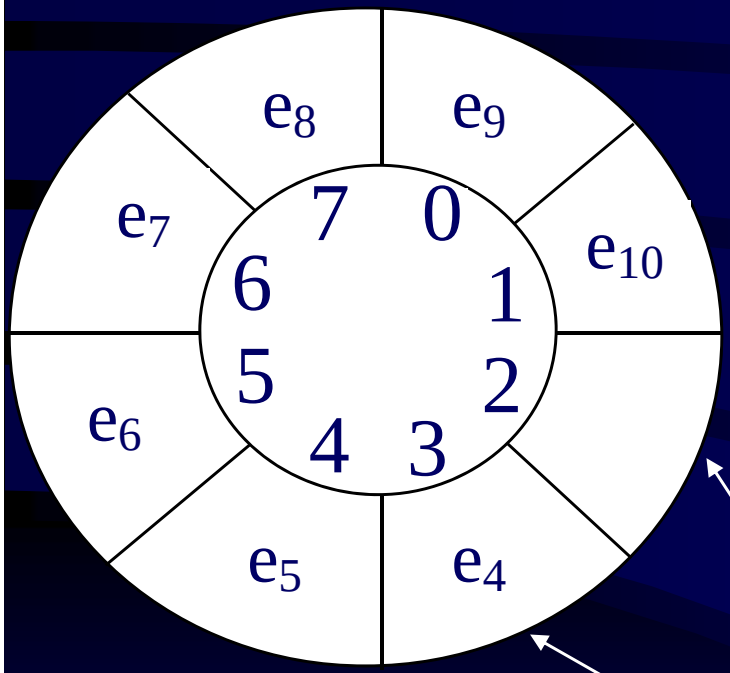
3.5 队列的存储结构（续）

队列的顺序存储---循环队列（二)

```
//----------动态分配-----------
#define MAXSIZE    100   //最大队列长度
typedef  struct {
            QElemType        *base
            int         front      //指向队头元素当前位置
            int         rear       // 指向队尾元素下一个位置
} SqQueue
```


出队
a_1 a_2 … a_i … a_n
入队
队头front
队尾rear
Q.front
Q.rear
6F36
6F26
6F02
6F00
a_n
a_i
a_2
a_1
Q.front
Q.rear
m-1
0
1
2
n
n+1
a_1
a_2
a_3
a_n
……

- 令队列空间中的一个单元闲置，使得在任何时刻，保持**Q.rear**和**Q.front**之间至少间隔一个空闲单元



MAXSIZE=8

- 队列满：
  (Q.rear+1)%MAXSIZE == Q.front
- 队列空：
  Q.rear==Q.front
- 队列长度：
  (Q.rear - Q.front+ MAXSIZE)% MAXSIZE

3.5 队列的存储结构（续）

DS/SS

队列的顺序存储---循环队列（三)

循环队列特点：

- 队空条件：

  front = rear（方式一）

  标志域 + （front = rear） （方式二）

- 队满条件：

  (rear+1) % MAXSIZE = front （方式一）

  标志域 + （front = rear） （方式二）

- 队满条件下的队满为真满

3.5 队列的存储结构（续）

DS/SS

队列的顺序存储---循环队列(四)

```
Status InitQueue(SqQueue &Q) {
    Q.base=(QElemType *)malloc(MAXSIZE * sizeof(QElemType));
    if (! Q.base) exit(overflow);
    Q.front = Q.rear = 0 ;
    return  OK;
} // InitQueue
```

3.5 队列的存储结构（续）

队列的顺序存储---循环队列(五)

```
Status EnQueue(SqQueue &Q,QElemType e)
    if((Q.rear+1) % MAXSIZE = = Q.front) return ERROR;
    Q.base[Q.rear] = e;
    Q.rear = (Q.rear +1) % MAXSIZE;
     return  OK;
} // EnQueue
```

3.5 队列的存储结构（续）

DS/SS

队列的顺序存储---循环队列(六)

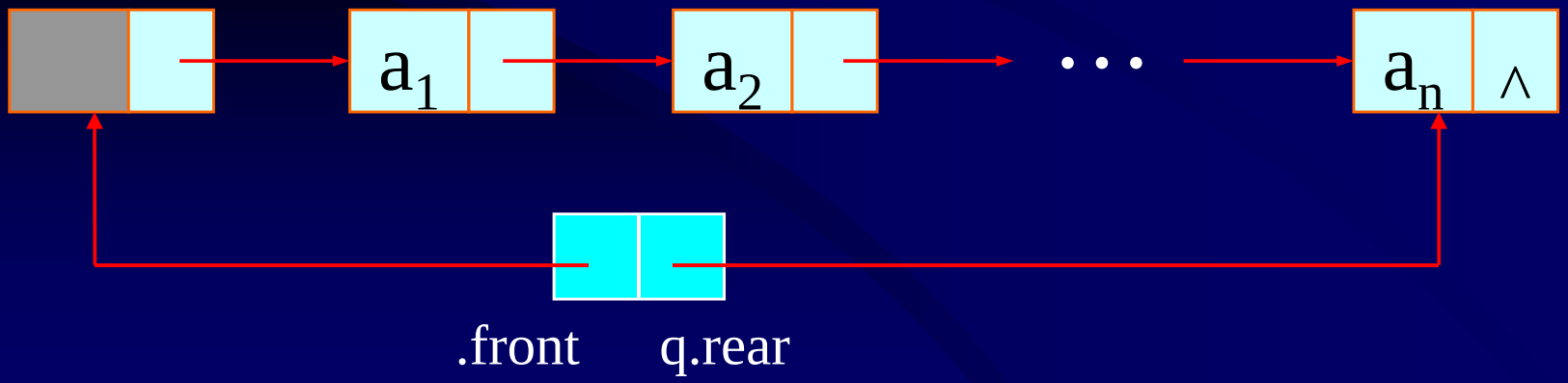
```
Status DeQueue(SqQueue &Q,QElemType &e){
    if(Q.rear= = Q.front) return ERROR;
    e = Q.base[Q.front];
    Q.front = (Q.front +1) % MAXSIZE;
     return  OK;
} // DeQueue
```

3.5 队列的存储结构（续）

队列的链式存储---链队列（一）

```
typedef  struct {
            QElemType        data
            struct QNode     *next
} QNode,* QueuePtr
```

```
typedef  struct {
            QueuePtr         rear
            QueuePtr         front
} LinkQueue
```



3.5 队列的存储结构（续）

队列的链式存储---链队列（二）

```
Status InitQueue(LinkQueue &Q) {
    Q.front = Q.rear = (QueuePtr)malloc(sizeof(QNode));
    if (! Q.front) exit(overflow);
    Q.front->next = NULL ;
    return  OK;
} // InitQueue
```

3.5 队列的存储结构（续）

队列的链式存储---链队列（三）

```
Status EnQueue(LinkQueue &Q,QElemType e)
    p = (QueuePtr) malloc (sizeof (Qnode));
    if (!p) exit(overflow);
    p->data = e;    p->next = NULL;
    Q.rear->next = p;          Q.rear = p;
     return  OK;
} // EnQueue
```

3.5 队列的存储结构（续）

队列的链式存储---链队列（四）

```
Status DeQueue(LinkQueue &Q,QElemType &e) {
    if(Q.rear= = Q.front) 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;
} // DeQueue
```