# 软件技术基础

第五讲

# 上讲主要内容

- 栈的定义
- 顺序栈
- 链栈
- 栈的应用

# 本讲主要内容

队列

链队列

顺序队列

# 队列的基本概念

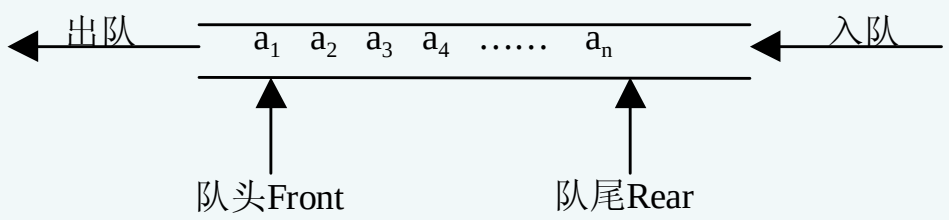
[定义]

**队列**也是一种运算受限的线性表。它只允许在表的一端进行插入，该端称为**队头**(Front)。它只允许在表的另一端进行删除，该端称为**队尾**(Rear)。

亦称作**先进先出**(First In First Out)的线性表。

当队列中没有元素时称为**空队列**。

[队列的示意图]



# 链队列的基本概念

**[定义]**

队列的链式存储结构简称为链队列，它是限制仅在表头删除和表尾插入的单链表。

类型定义

```
typedef  struct QNode {
    QElemType  data;
   struct QNode *next;
   }QNode, *QueuePtr;
typedef struct
{  QueuePtr  front;
    QueuePtr  rear;
  }LinkQueue;
LinkQueue  Q;
```

```
//元素结点


//特殊结点
//front指向头结点
//rear指向尾结点
```

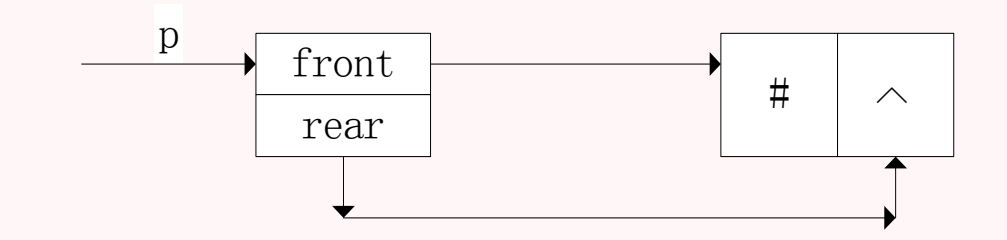
# 链队列的基本概念

实质是带头结点的线性链表；

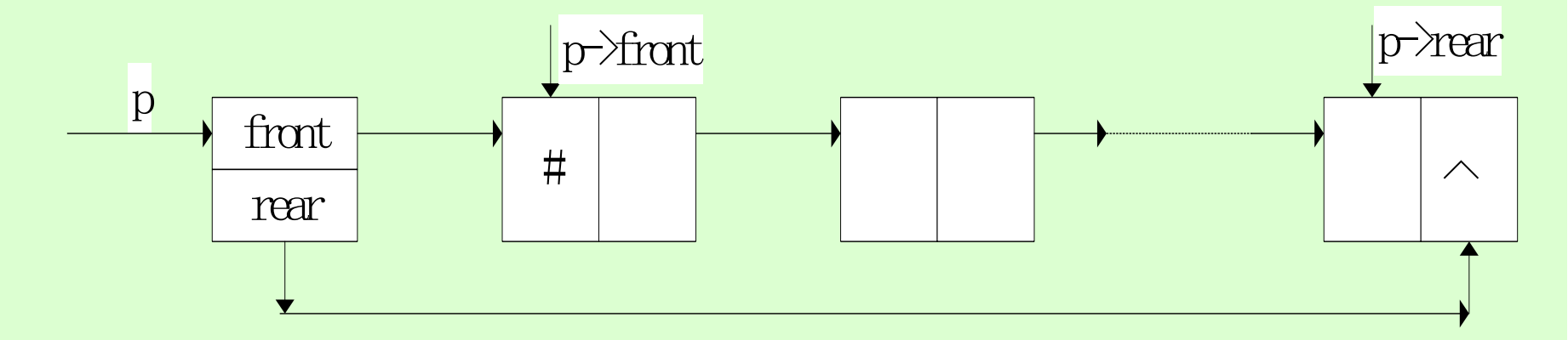
队空条件, 头指针和尾指针均指向头结点

Q.front = Q.rear

# 链队列的图示说明

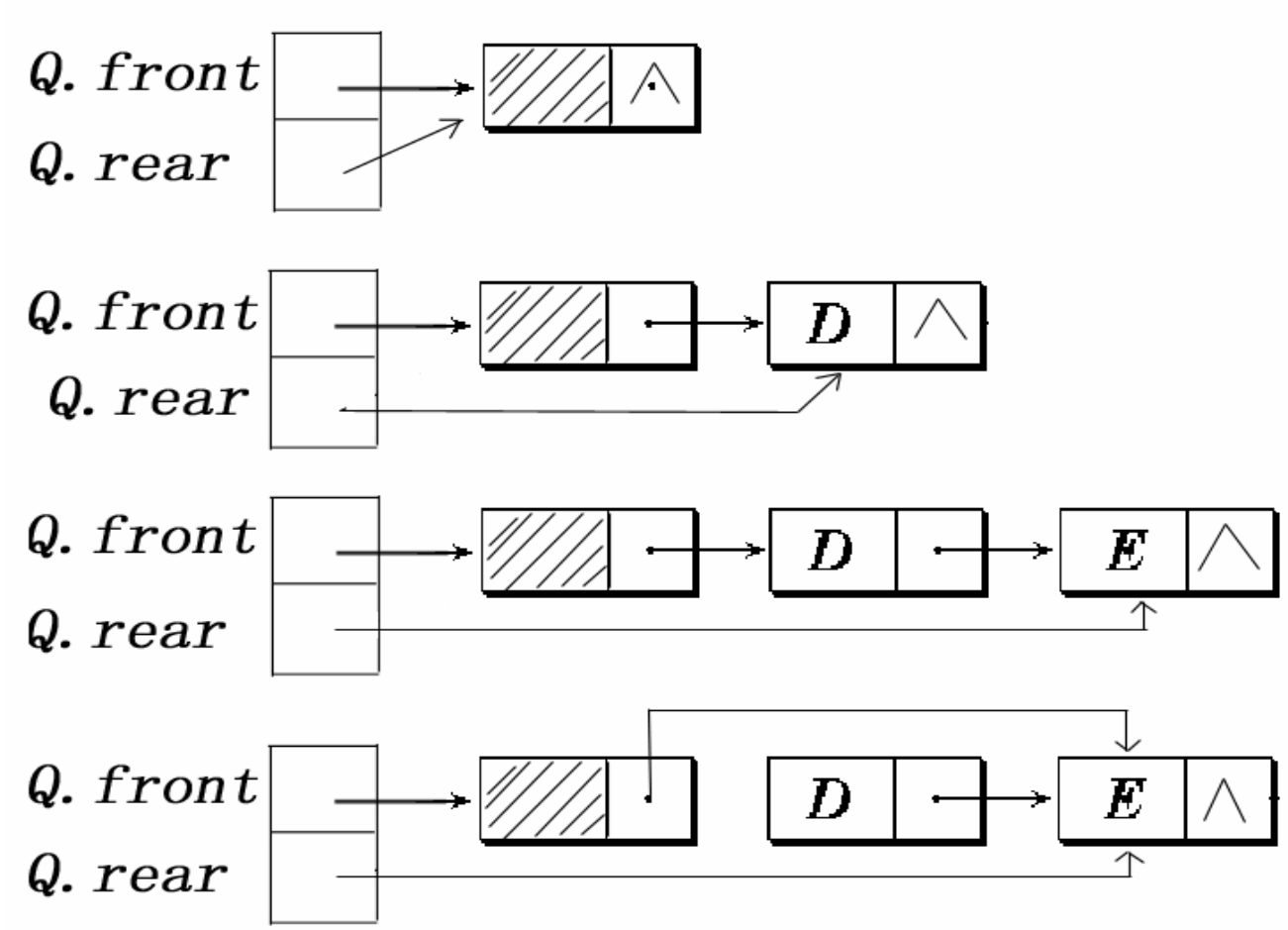



空的链队列




非空的链队列示意图

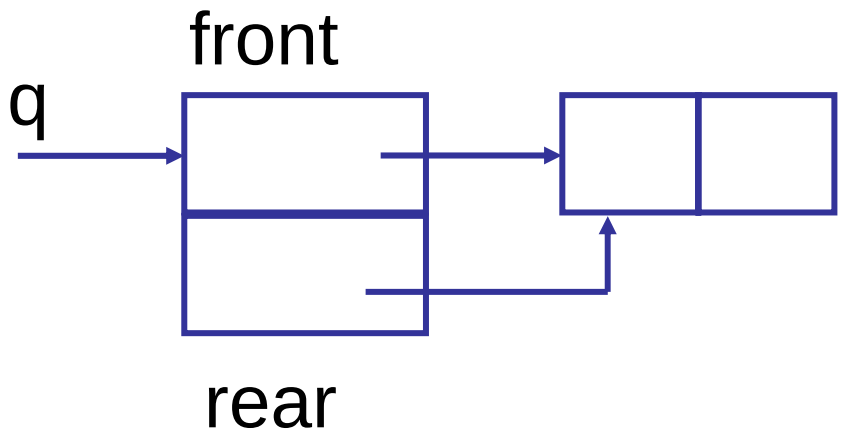
# 队列运算指针变化状况



# 基本操作与实现

- 初始化　p62
  Status InitQueue (LinkQueue &Q)
- 销毁队列　p62
  Status DestroyQueue (LinkQueue &Q)
- 入队　p62
  Status EnQueue(LinkQueue &Q, QElemType e)
- 出队　p62
  Status DeQueue(LinkQueue &Q, QElemType &e)
- 判队空
  Status QueueEmpty(LinkQueue Q)
- 取队头元素
  Status GetHead(LinkQueue Q,QElemType &e)

# 设置空队示意图




【设空队】

1. q->front=
   q->rear=(QueuePtr)malloc(sizeof(QNode));
2. q->front->next=NULL ;

# 链队列初始化

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

# 链队列的销毁

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

# 链队列的插入（入队）

```
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;
}
```

# 链队列的删除（出队）

```
Status DeQueue (LinkQueue &Q,
   ElemType &e)
{  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;
}
```

# 判断链队列是否为空

```
Status QueueEmpty(LinkQueue Q)
{
   if (Q.front==Q.rear) return TRUE;
   return FALSE;
}
```

# 取链队列的第一个元素结点

```
Status GetHead(LinkQueue
   Q,QElemType &e)
{
   if (QueueEmpty(Q)) return ERROR;
   e=Q.front->next->data;
   return OK;
}
```

# 循环队列——队列的顺序存储结构

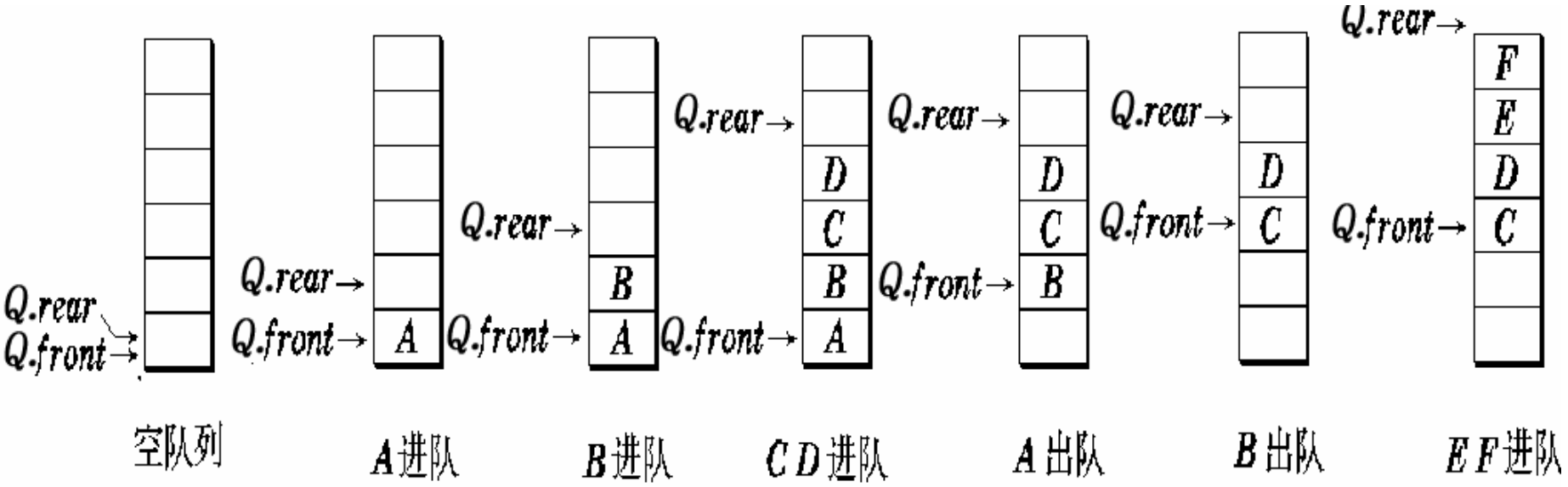
- 顺序队列：
  - 用一组地址连续的存储单元依次存放从队列头到队列尾的元素
- 设两个指针：
  - Q.front 指向队列头元素；
  - Q.rear 指向队列尾元素的下一个位置
- 初始状态（空队列）：
  - Q.front = Q.rear=0
- 队列的真满与假满

# 类型定义 p64

```
#define MAXSIZE 100
typedef struct {
   QElemType  *base;
   int front;
   int rear;
  }SqQueue;
SqQueue  Q;
```

# 队列的进队和出队

- 进队时，将新元素按Q.rear 指示位置加入，再将队尾指针增1，Q.rear = Q.rear + 1 。
- 出队时，将下标为Q.front 的元素取出，再将队头指针增1，Q.front = Q.front + 1 。
- 队满时再进队将溢出出错；队空时再出队作队空处理。上图为“假满”



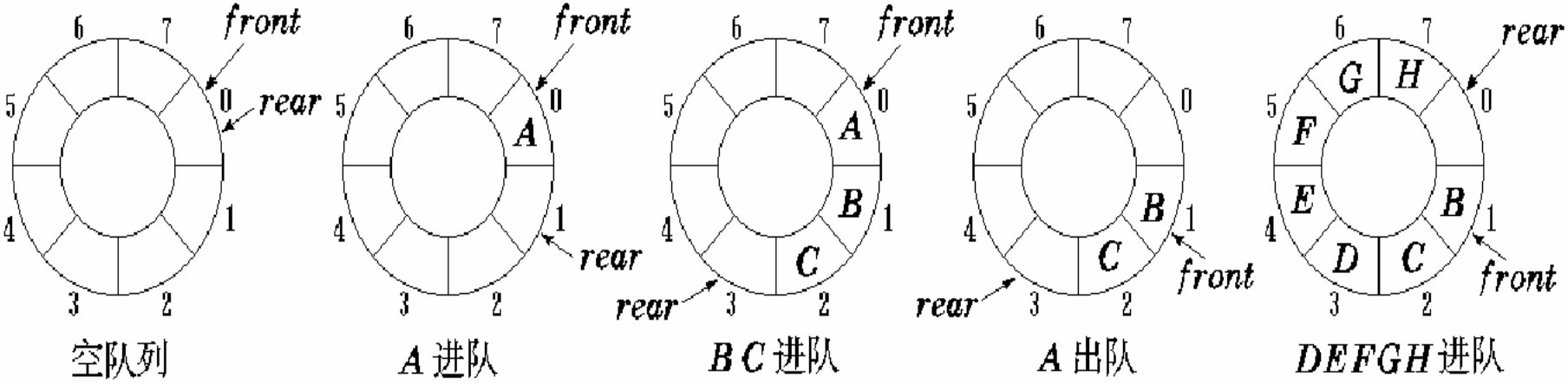
# 循环队列 (Circular Queue)

- ✓ 存储队列的数组被当作首尾相接的表处理。
- ✓ 队头、队尾指针加**1**时从***maxSize*** **−1**直接进到**0**，可用语言的取模**(**余数**)**运算实现。

  队头指针进**1:** Q.*front* = (Q.*front* + 1)% *MAXSIZE*

  队尾指针进**1:** **Q.***rear* = (Q.*rear* + 1)% *MAXSIZE*;
- ✓ 队列初始化: **Q.***front* **= Q.***rear* **= 0;**
- ✓ 队空条件: **Q.***front* **==** **Q.***rear*;
- ✓ 队满条件: (Q.*rear* + 1) % *MAXSIZE* == *Q.front*
- ✓ 队列长度：(Q.rear-Q.front+MAXSIZE)%MAXSIZE

# 循环队列的进队和出队



# 说明

- 不能用动态分配的一维数组来实现循环队列，初始化时必须设定一个最大队列长度。
- 解决 Q.front=Q.rear不能判别队列“空”还是“满”的其他办法：
  - 使用一个计数器记录队列中元素的总数（即队列长度）
  - 设一个标志变量以区别队列是空或满
- 循环队列中要有一个元素空间浪费掉 ----p63 约定队列头指针在队列尾指针的下一位置上为“满”的标志

# 基本操作

- 初始化　p64

Status InitQueue (SqQueue &Q)

- 求队列的长度　p64

int QueueLength(SqQueue Q)

- 入队　p65

Status EnQueue (SqQueue &Q, QElemType e)

- 出队　p65

Status DeQueue(SqQueue &Q, QElemType &e)

- 判队空

Status QueueEmpty(SqQueue Q)

- 取队头元素

Status GetHead(SqQueue Q,QElemType &e)

# Status InitQueue (SqQueue &Q)

```
{
   Q.base=(QElemTye
   *)malloc(MAXQSIZE*sizeof(QElemType));
   if (!Q.base) exit(OVERFLOW);
   Q.front=Q.rear=0;
   return OK;
}
```

# int QueueLength(SqQueue Q)

```
{
   return (Q.rear-
   Q.front+MAXQSIZE)%MAXQSIZE;
}
```

# Status EnQueue (SqQueue &Q, QElemType e)

```
{
   if ((Q.rear+1) % MAXQSIZE ==Q.front)
   return ERROR;
   Q.base[Q.rear]=e;
   Q.rear = (Q.rear+1)%MAXQSIZE;
   return OK;
}
```

# Status DeQueue(SqQueue &Q, QElemType &e)

```
{
   if (Q.rear==Q.front) return ERROR;
   e=Q.base[Q.front];
   Q.front=(Q.front+1)%MAXQSIZE;
   return OK;
}
```

# Status QueueEmpty(SqQueue Q)

```
{
   if (Q.front==Q.rear) return TRUE;
   return FALSE;
}
```

Status GetHead(SqQueue Q,QElemType &e)

```
{
   if QueueEmpty(Q) return ERROR;
   e=Q.base[Q.front];
   return OK;
}
```

# 非循环队列

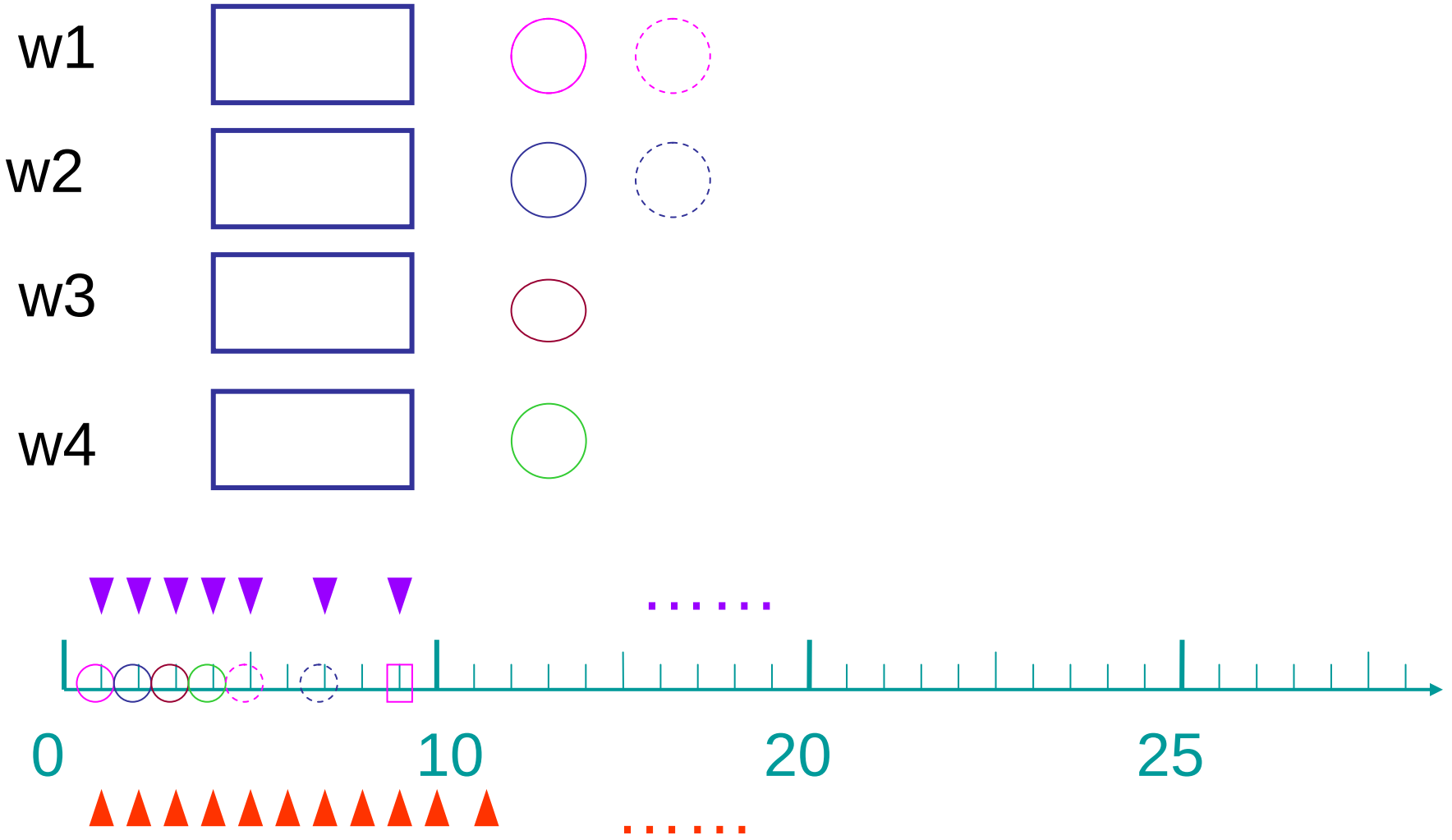
- ✓类型定义：与循环队列完全一样
- ✓关键：修改队尾/队头指针　Q.rear = Q.rear + 1; Q.front = Q.front+1;
- ✓在判断时，有**%MAXQSIZE**为循环队列，否则为非循环队列
  - ➢队空条件：　Q.front = Q.rear
  - ➢队满条件：　Q.rear>= MAXQSIZE
- ✓注意“假上溢”的处理
- ✓长度：　**Q.rear - Q.front**

# 队列的基本运算

[服务系统描述]

假设服务系统有四个窗口对外接待客户，在营业时间内不断有客户进入并要求服务。由于每个窗口只能接待一个客户，因此进入该服务系统的客户需在某一窗口前排队。如果窗口的服务员忙则进入的客户需排队等待，闲则可立即服务，服务结束则从队列中撤离。并计算一天中用户在此逗留的平均时间。

# 服务系统仿真示意图



# 模拟方式

依次根据事件来确定系统状态的变化，即**事件驱动模拟**。

依次根据时间来确定系统状态的变化，即**时间驱动模拟**。

# 离散事件仿真问题分析

## 计算用户逗留时间

用户逗留时间=用户离开时间-用户到达时间

用户离开时间=用户开始接受服务的时间+用户服务时间

用户开始接受服务的时间＝该用户所在的队列队头用户离开，且该用户为 队头用户

- 服务系统模拟的要求
  - 用户到达
  - 用户接受服务
  - 用户离开

# 离散事件仿真问题分析

- 新客户进入服务系统，该客户加入到队列最短的窗口队列中
- 四个队列中有客户服务完毕而撤离。

# 事件发生与处理过程

## [新客户到达]

- ✓ 统计的客户数目加一。
- ✓ 设定一个新的到达事件——下一客户即将到达服务系统的事件插入事件表。

  下个客户的 到达时间＝当前时间＋随机的时间间隔
- ✓ 将新到客户插入到最短队列中。队列中的信息包括本用户到达时间（当前时间）和接受服务的时间。

  注：若队列原来是空的，则插入的客户为队头元素，此时应设定一个新的事件——刚进入服务系统的客户办完业务离开服务系统的事件插入事件表。

  离队时间＝当前时间＋接受服务时间。

# 事件发生与处理过程

## [i队列客户离开达]

- ✓ 客户从队头删除。

  注：当队列非空(用户离开后)，应把新的队头客户设定为一个新的离开事件，并插入事件表。

  离队时间＝当前时间＋接受服务时间

- ✓ 并计算该客户在服务系统中逗留时间。

  逗留时间＝当前时间－用户到达时间

# 离散事件仿真的数据结构考虑

该仿真(模拟)程序中，主要应设置四个队列和一个有序事件表。

- 队列采用单链表来实现，其中单链表中的每个结点代表一个客户，应有两个数据：客户到达时间和客户的服务时间。该链队列有一个队头结点，包括的数据是队列中的客户数。
- 事件表用单链表来实现，其中的每个结点代表一个事件，包括的数据项有：事件发生时间和事件类型。事件类型为0、1、2、3、4，其中0表示客户到达事件，1、2、3和4分别表示四个窗口的客户离开事件。
- 事件表中最多有五个事件，当事件表为空时程序运行结束。

# 离散事件仿真的数据结构说明

```
struct  qneuenode
{ int  arrivetime,  duration;
  struct  queuenode  *next;
} /*队列结点*/
```

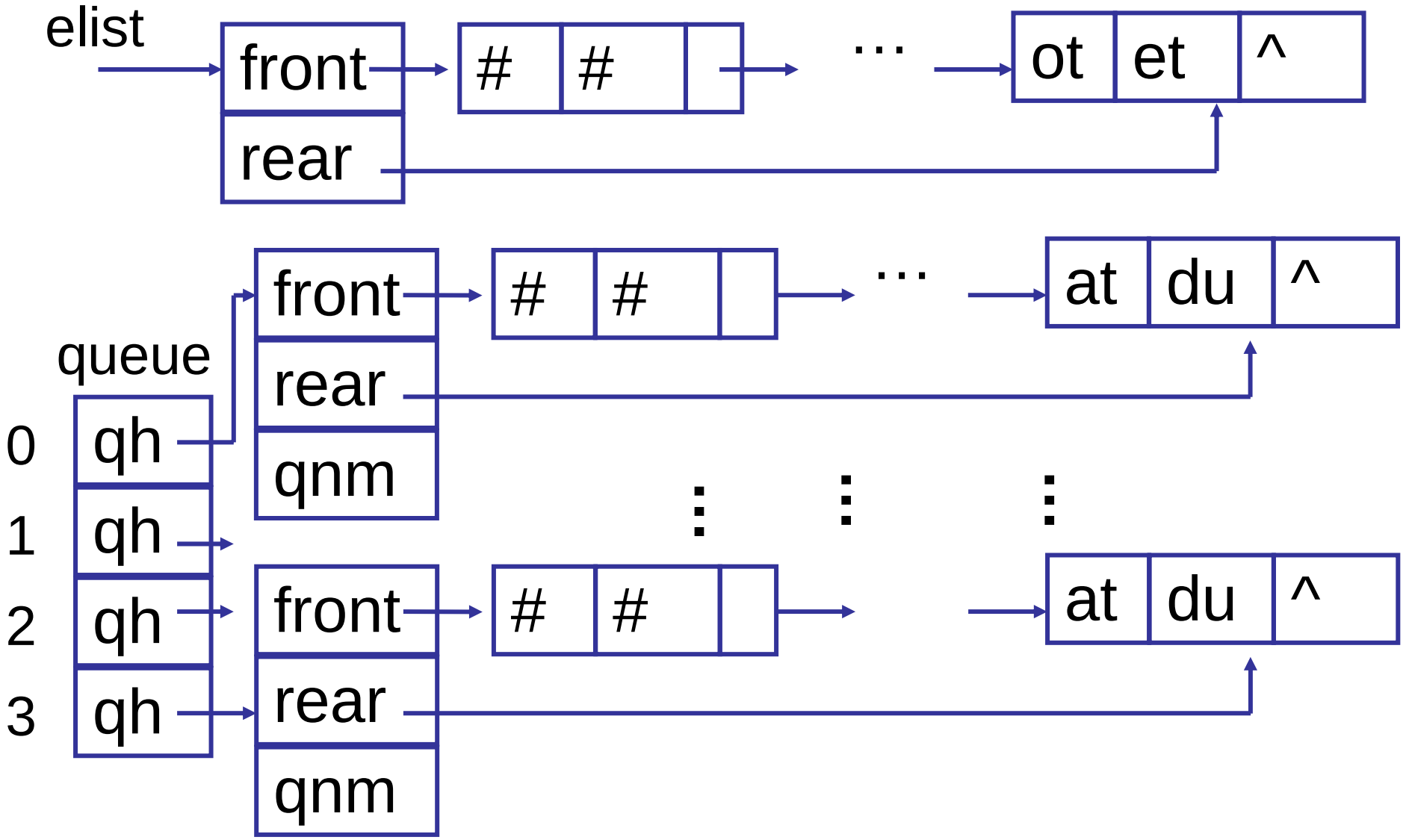
```
struct  queueheader
{struct  queuenode  *front, *rear;
 int  queuenodenum;
} /*队列*/
```

```
struct  eventnode
{ int  occurtime;
  int  eventtype;
  struct  eventnode  *next;
} /*事件结点*/
```

```
struct  eventlist
{ eventnode  *front, *rear;
  int  eventnum;
} /*事件表*/
```

```
struct  queueheader  *queue[m]; /*四个服务队列*/
struct  eventlist  *eventlst; /*事件表*/
```

# 离散事件仿真的数据结构示意图



# 小结