



5.2

队列 Queues

郝家胜

hao@uestc.edu.cn

自动化工程学院



内容回顾

- 栈的概念
- 栈的抽象数据类型
- 栈的应用
- 栈的实现

线性表的抽象数据类型

● 数学模型

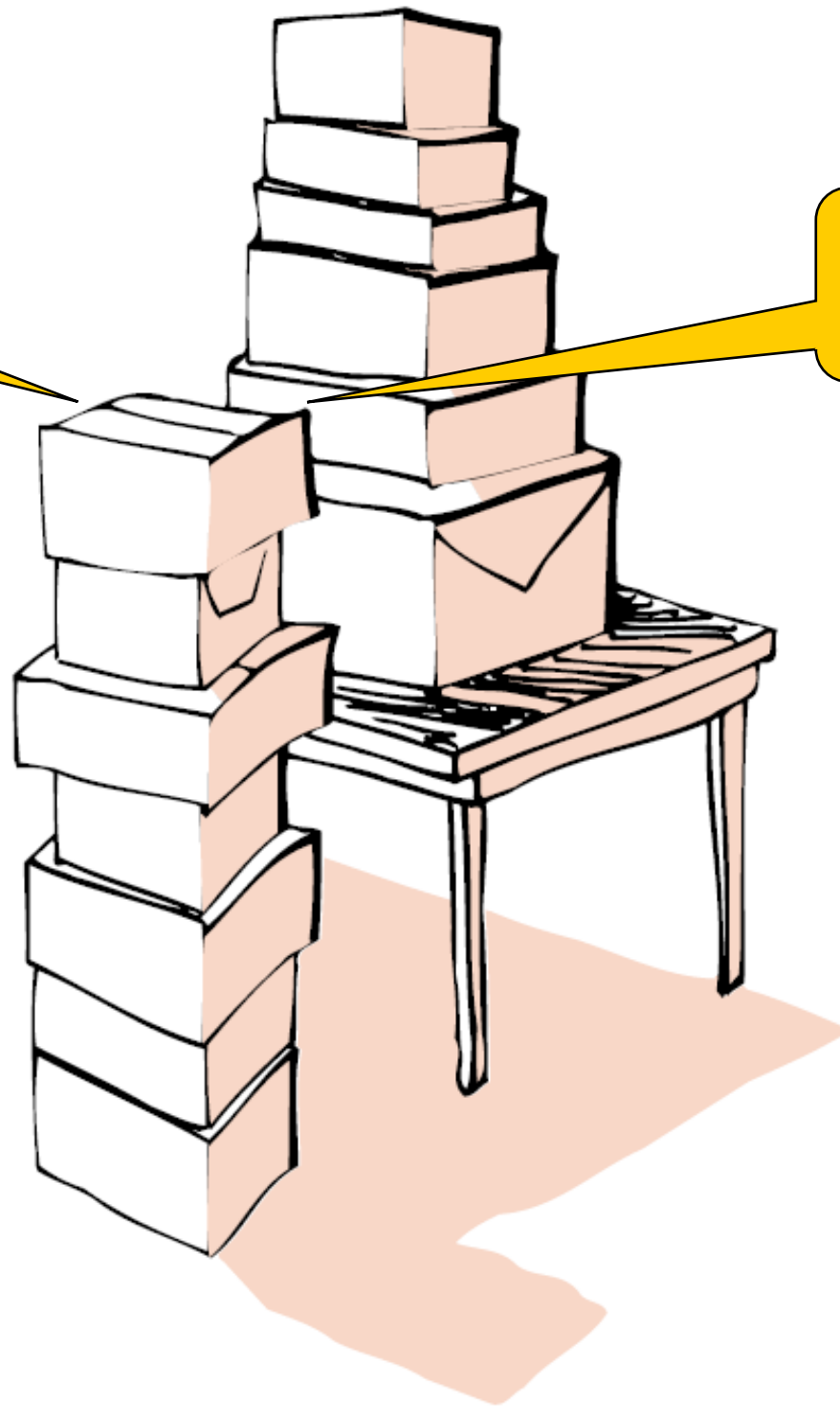
$L: (a_1, a_2, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_n)$

● 接口声明

- ▶ `LENGTH (L)`
- ▶ `GET (L, i)`
- ▶ `INSERT (L, i, x)`
- ▶ `DELETE (L, i)`

“取”操作必须在这端进行

“放”操作必须在同一端进行



在一端访问的受限线性表

线性表

栈

Insert(L, **i**, x)

$1 \leq i \leq n+1$

Delete(L, **i**)

$1 \leq i \leq n$

Insert(S, **n+1**, x)

Delete(S, **n**)

栈的基本操作

- 构造空栈：创建一个新的空栈
- 判空栈：若栈S为空返回“真”，否则返回“假”。
- 进栈：在栈的顶部压入（插入）元素x。
- 出栈：若S不空，则弹出（删除）顶部元素。
- 取栈顶：取栈顶元素，并不改变栈中内容。

栈的抽象数据类型

● 数学模型

$S: (a_1, a_2, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_n)$

● 接口声明

- ▶ $PUSH(S, x)$: //进栈
- ▶ $POP(S)$: //出栈
- ▶ $TOP(S)$: //取栈顶元素
- ▶ $IS_EMPTY(S)$: //判栈空否
- ▶ $IS_FULL(S)$: //判栈满否

栈的应用——栈与函数调用

当在一个函数的运行期间调用另一个函数时，在运行该被调用函数之前，需先完成三项任务：

- 将所有的实在参数、返回地址等信息传递给被调用函数保存；
- 为被调用函数的局部变量分配存储区；
- 将控制转移到被调用函数的入口。

从被调用函数**返回**调用函数**之前**，
应该完成下列三项任务：

- **保存**被调函数的**计算结果**；
- **释放**被调函数的**数据区**；
- 依照被调函数保存的返回地址将**控制转移**到调用函数。

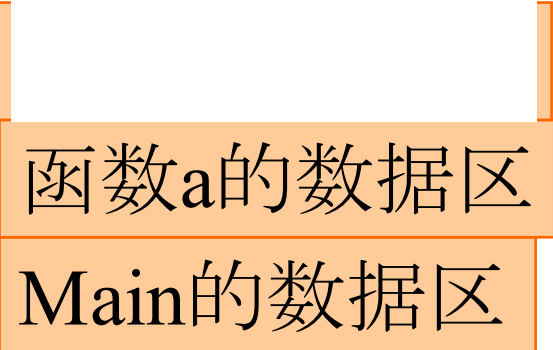
多个函数嵌套调用的规则是：

后调用先返回！

此时的内存管理实行“栈式管理”

例如：

```
void main( ){    void a( ){    void b( ){  
    ...          ...          ...  
    a( );        b( );  
    ...          ...  
} //main        } // a      } // b
```



函数a的数据区
Main的数据区

Factorials: 一个递归定义

- 数学定义:

$$n! = n \times (n - 1) \times \cdots \times 1.$$

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n - 1)! & \text{if } n > 0. \end{cases}$$

递归基
(base case)

递归步: 将一般情况
化为较简单情况
(recursive case)

```
int factorial(int n)
```

```
/* Pre:  n is a nonnegative integer.
```

```
   Post: Return the value of the factorial of n. */
```

```
{
```

```
    if (n == 0) return 1;
```

```
    else      return n * factorial(n - 1);
```

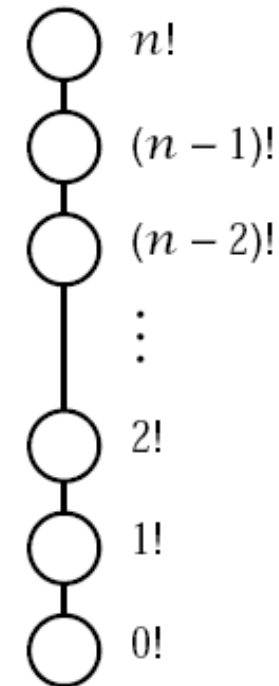
```
}
```

+ : 非常简洁, 易理解;

- : 需要保存一系列的计算结果.

Example:

$$\begin{aligned}
 \text{factorial}(5) &= 5 * \text{factorial}(4) \\
 &= 5 * (4 * \text{factorial}(3)) \\
 &= 5 * (4 * (3 * \text{factorial}(2))) \\
 &= 5 * (4 * (3 * (2 * \text{factorial}(1)))) \\
 &= 5 * (4 * (3 * (2 * (1 * \text{factorial}(0))))) \\
 &= 5 * (4 * (3 * (2 * (1 * 1)))) \\
 &= 5 * (4 * (3 * (2 * 1))) \\
 &= 5 * (4 * (3 * 2)) \\
 &= 5 * (4 * 6) \\
 &= 5 * 24 \\
 &= 120.
 \end{aligned}$$

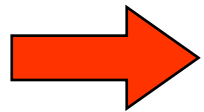


函数调用的实现

- 函数调用之间的关系: 先调用的后返回;
- 调用函数时要分配数据区存放返回地址, 实在参数以及局部变量等, 称之为活动记录;
- 数据区的特点: 先分配的数据区后释放; 所以,
- 程序运行中这种数据区的分配和释放要用栈(运行栈)来实现;
- 栈顶存放的是当前运行函数的数据: 调用函数时在栈顶为它分配数据区, 返回时释放栈顶数据区.
- 递归调用的实现同理进行: 对于每个递归调用, 将相应的(不同)活动记录入栈.

队列

- 栈



- 队列的概念
- 队列的抽象数据类型
- 队列的实现
- 队列的应用

队列的概念



队列(Queues)是生活中“排队”的抽象

- 排队只能排在尾部，排在对头的先得到服务；
- “插队”是不允许的。换言之，插入和删除只能在线性表的两头分别进行；
- 插入的一端称为队尾，删除的一端称为队头；
- 其特点是：先进先出(FIFO)

队列的概念

- **队列 (queue)** 是一种只允许在一端进行插入，而在另一端进行删除的线性表，它是一种操作受限的线性表。
 - ▶ 队列的插入操作通常称为**入队列**或**进队列**
 - ▶ 队列的删除操作则称为**出队列**或**退队列**

线性表

队列

Insert(L, **i**, x)

Insert(Q, **n+1**, x)

$1 \leq i \leq n+1$

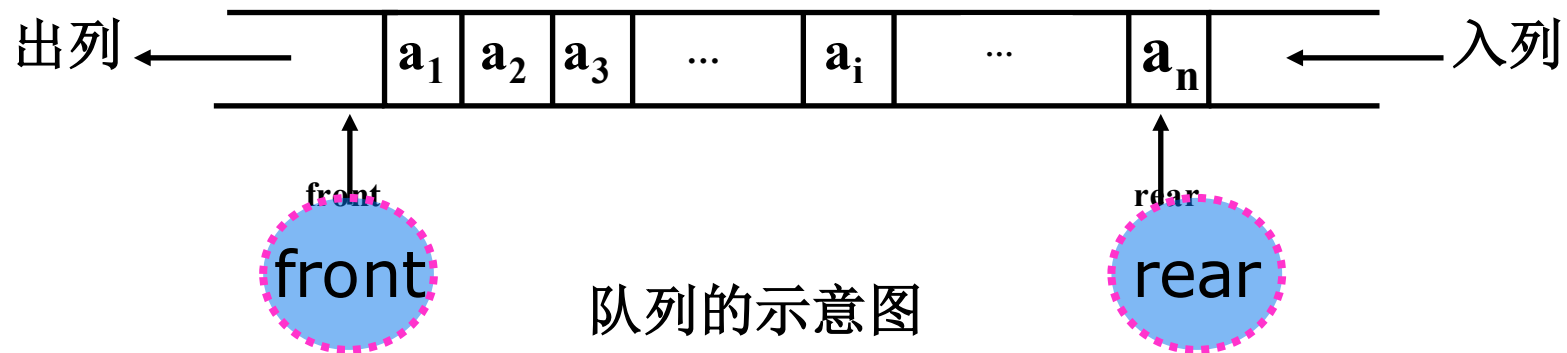
Delete(L, **i**)

Delete(Q, **1**)

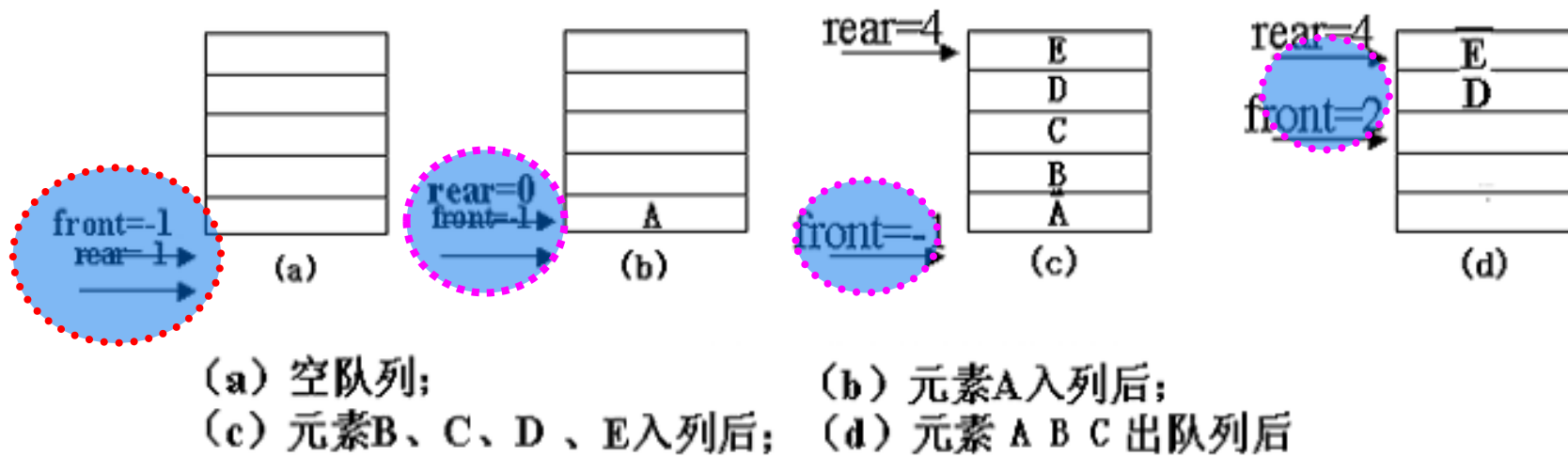
$1 \leq i \leq n$

队列的概念

假若队列 $q = \{a_1, a_2, \dots, a_n\}$ ，进队列的顺序为 a_1, a_2, \dots, a_n ，则队头元素为 a_1 ，队尾元素为 a_n 。



队列的概念



队列

- 栈

- 队列的概念

-  • 队列的抽象数据类型

- 队列的实现

- 队列的应用

队列的基本操作

- 构造队列：创建一个新的队列
- 入队列：在队尾插入新结点。
- 出队列：从队头取出并删除结点。
- 判空队列：若队列为空返回“真”，否则返回“假”
- 判满队列：若队列为空返回“真”，否则返回“假”
- 取队列头：取队头元素，并不改变队列中内容。

队列的抽象数据类型

● 数学模型

$Q: (a_1, a_2, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_n)$

● 接口声明

- ▶ `ENQUEUE(Q, x)` : //进队列
- ▶ `DEQUEUE(Q)` : //出队列
- ▶ `IS_EMPTY(Q)` : //判队列空否
- ▶ `IS_FULL(Q)` : //判队列满否
- ▶ `HEAD(Q)` : //取队列头元素

杨辉三角算法

YANGHUI-TRIANGLE (n)

```
1  ▷ 打印n阶杨辉三角
2   $Q \leftarrow \text{CREATE\_QUEUE}()$ 
3   $\text{ENQUEUE}(Q, 1)$ 
4  for  $i \leftarrow 1$  to  $n$  do
5       $s \leftarrow 0$ 
6      for  $j \leftarrow 1$  to  $i$  do
7           $t \leftarrow \text{DEQUEUE}(Q)$ 
8          print t
9           $\text{ENQUEUE}(Q, s+t)$ 
10          $s \leftarrow t$ 
11     end
12      $\text{ENQUEUE}(Q, 1)$ 
13     print '\newline'
14 end
```

队列

- 栈

- 队列的概念

- 队列的抽象数据类型

-  • 队列的实现

- 队列的应用

队列的实现

- 顺序存储
- 链接存储

顺序队列

```
struct queue_t
{
    int *data; /* 节点类型及存储缓冲区首地址 */
    int front; /* 队列头 */
    int rear; /* 队列尾 */
    int max_length; /* 最大节点个数 */
};

typedef struct queue_t *QUEUE;

/* 创建一个空的队列 */
QUEUE create_queue()
{
    QUEUE q = (QUEUE) malloc(sizeof(struct queue_t));
    q->data = (int*) malloc(sizeof(int) * MAX_LENGTH);
    q->front = q->rear = 0;
    q->max_length = MAX_LENGTH;

    return q;
}
```

顺序队列

```
bool is_queue_empty(Queue q)
```

```
{  
    return (q->front == q->rear);  
}
```

```
bool is_queue_full(Queue q)
```

```
{  
    return (q->rear == q->max_length);  
}
```

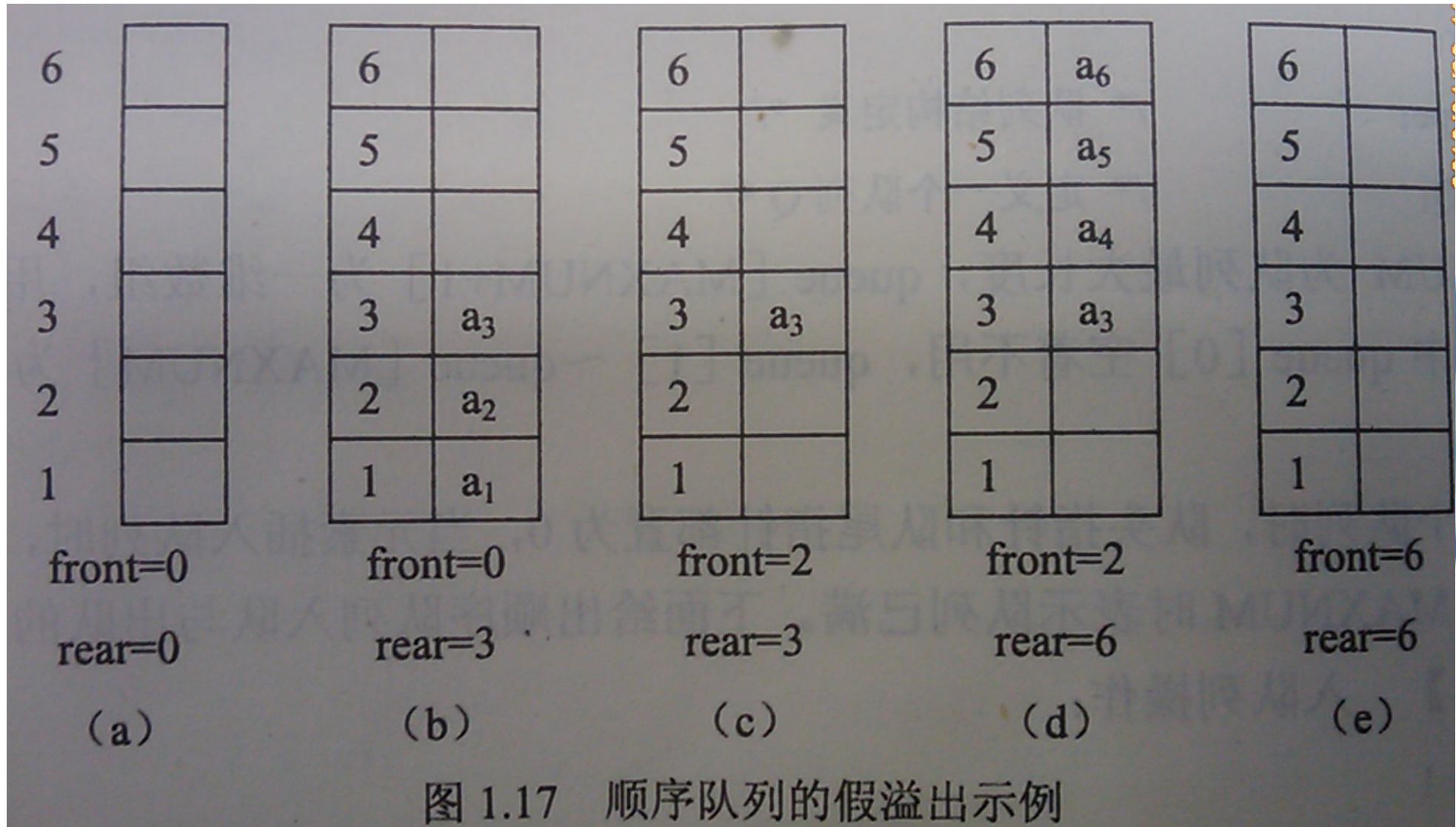
```
int enqueue(Queue q, int x)
```

```
{  
    if (is_queue_full(q))  
        return E_OVERFLOW;  
  
    q->data[q->rear] = x;  
    q->rear += 1;  
  
    return E_SUCCESS;  
}
```

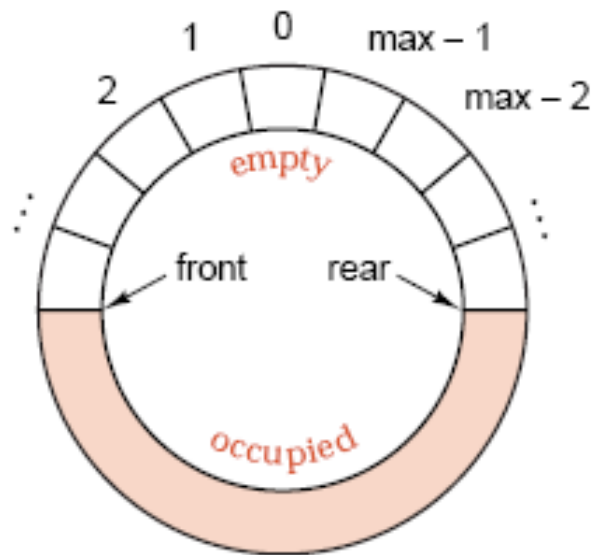
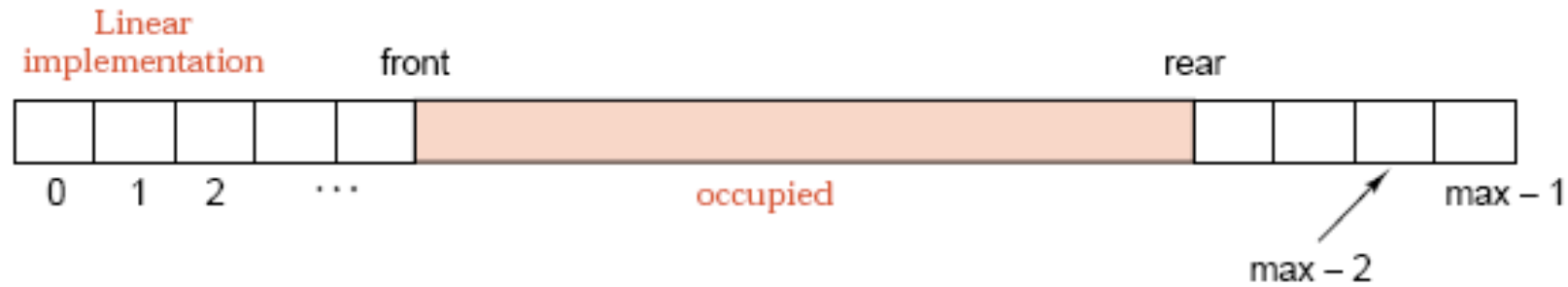
```
int dequeue(Queue q)
```

```
{  
    int x;  
  
    if (is_queue_empty(q))  
        return E_UNDERFLOW;  
  
    x = q->data[q->front];  
    q->front += 1;  
  
    return x;  
}
```

顺序队列的假溢出

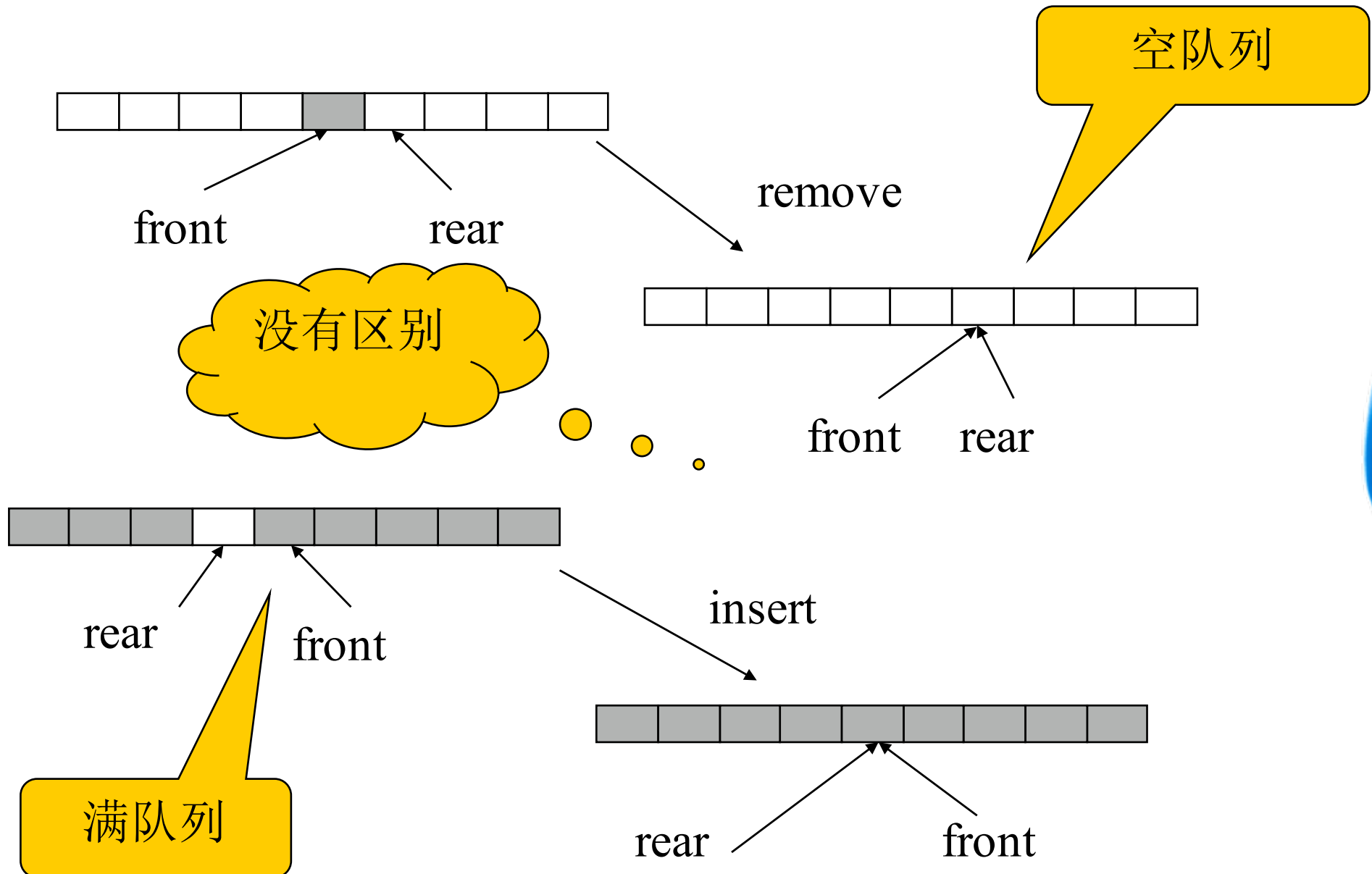


循环队列—队列的顺序表示与实现



为了有效地利用空间, 将队列想象成首尾相接的环形数组

边界条件



边界条件(Cont.)

解决的办法:

- 设一个计数器
- 另设一个标志以区别空和满;
- 数组中空一个单元, 当 $(\text{rear} + 1) \% \text{MAXSIZE} == \text{front}$ 时表示对列已满, $\text{front} == \text{rear}$ 时表示空.

顺序队列

```
bool is_queue_empty(Queue q)
```

```
{  
    return (q->front == q->rear);  
}
```

```
bool is_queue_full(Queue q)
```

```
{  
    return (q->rear == q->max_length);  
}
```

```
int enqueue(Queue q, int x)
```

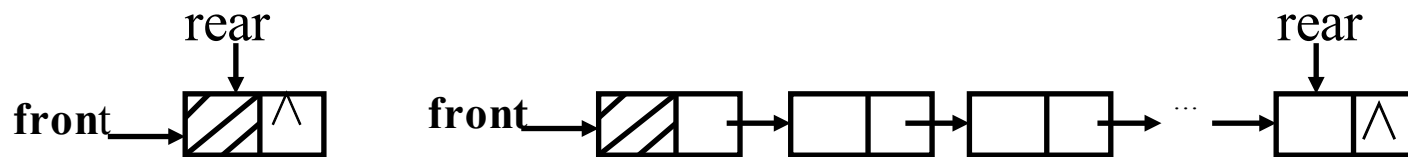
```
{  
    if (is_queue_full(q))  
        return E_OVERFLOW;  
  
    q->data[q->rear] = x;  
    q->rear += 1;  
  
    return E_SUCCESS;  
}
```

```
int dequeue(Queue q)
```

```
{  
    int x;  
  
    if (is_queue_empty(q))  
        return E_UNDERFLOW;  
  
    x = q->data[q->front];  
    q->front += 1;  
  
    return x;  
}
```


链接队列

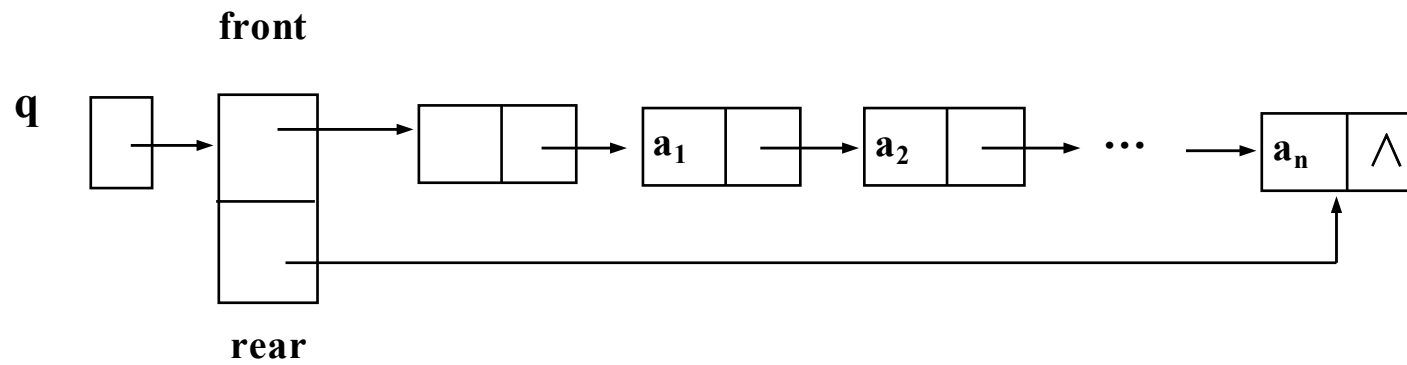
- ▶ 链队列中需设定**头指针**和**尾指针**分别指向队列的头和尾。
- ▶ 为了操作的方便，和线性链表一样，我们也给链队列添加一个头结点，并设定头指针指向头结点。
- ▶ 空队列的判定条件：头指针和尾指针都指向头结点。



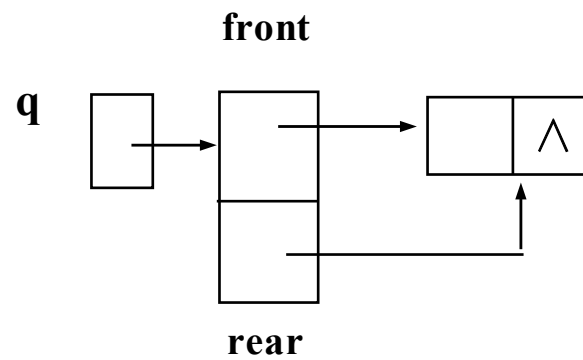
链接队列示意图

```
struct queue_t
{
    NODE front; /* 队列头 */
    NODE rear; /* 队列尾 */
};
```

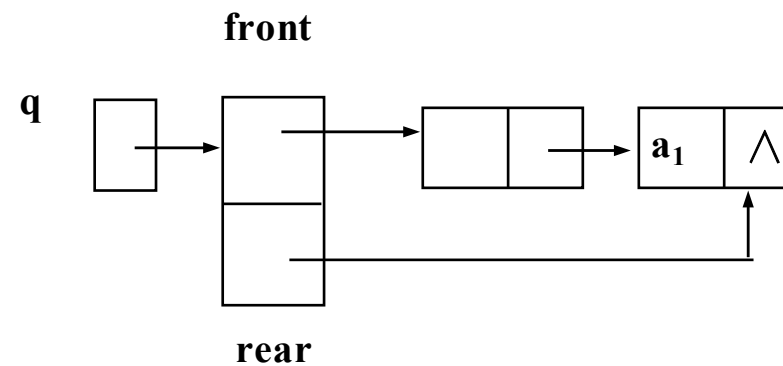
```
typedef struct node_t
{
    int data;
    struct node_t *next;
} *NODE;
```



(a) 非空队



(b) 空队



(c) 链队中只有一个元素结点

头尾指针封装在一起的链队

链接队列的实现

```
/* 创建一个空的队列 */
```

```
QUEUE create_queue()
```

```
{
```

```
    QUEUE q = (QUEUE) malloc(sizeof(struct queue_t));
```

```
    NODE head = (NODE) malloc(sizeof(struct node_t));
```

```
    head->next = NULL;
```

```
    q->front = q->rear = head;
```

```
    return q;
```

```
}
```

链接队列的实现

- **is_queue_empty(QUEUE q)**
- **is_queue_full(QUEUE q)**
- **enqueue(QUEUE q)**
- **dequeue(QUEUE q)**

队列

- 栈

- 队列的概念

- 队列的抽象数据类型

- 队列的实现

-  • 队列的应用



队列的应用

- **FIFO**

- **作业管理**

- ▶ 用户输入缓冲区
- ▶ 操作系统多任务管理机制

栈和队列是操作受限制的线性表。

- 它们具有相同的逻辑结构，即线性结构；
- 操作只能在表的两头进行：栈的插入和删除操作只能在一端进行；队列的插入和删除操作分别在两端进行。
- 它们的存储结构可以是顺序的，也可以是链式的。