



Beijing Normal University
School of Artificial Intelligence

第3章 栈与队列

郑新、徐鹏飞、李健
北京师范大学人工智能学院
2025-2026学年 第一学期

考核要点

■ 考核大纲

- 栈和队列的基本概念
- 栈和队列的顺序存储和链式存储
- 栈和队列的应用

■ 复习要点

- 理解栈的基本概念 (A) ; 熟练掌握栈的顺序实现和链接实现 (A)
- 灵活运用栈求解经典问题 (B)
- 灵活运用单调栈求解问题 (B)
- 理解队列的基本概念 (A) ; 熟练掌握队列的顺序实现和链接实现 (A)
- 灵活运用队列求解经典问题 (B)
- 了解单调队列的实现方法及应用 (B)

- 3.1 问题引入：超市货架管理
- 3.2 栈的定义与结构
- 3.3 队列的定义与结构
- 3.4 栈与队列的应用
- 3.5 拓展延伸
- 3.6 应用场景：消息队列

3.1 问题引入：超市货架

■ 商品陈列

- 商品分类陈列原则：按照商品的分类层次，大区域 → 中区域 → 小区域
- 价格按序陈列原则：由上至下、由左向右，价格由低到高陈列
- 按吸引力陈列原则：应季、紧俏、特价等特点
- 按方便性陈列原则：.....

■ 数据结构角度

- 对商品信息（数据）进行合理的组织（商品分类）、存储（商品陈列）

■ 问题：如何进行补货或选购？

3.1 问题引入：超市货架

■ 超市货架操作

- 补货：将特定商品**插入**到货架的某个位置
- 选购：从货架的某个位置**删除**选定的商品

■ 根据货架规格特点，形成不同的选购补货策略

- 选购补货统一（一端操作）：单门式冰柜因其构造特点，理货人员补货/顾客选购商品往往都是打开冰柜门直接存/取最前面的物品，这样使得商品的补架和选购均在冰柜门一端进行
- 选购补货分离（两端操作）：装备面向顾客的前门和面向理货人员的后门的大型冰柜，补架在后门进行，选购则在另一端前门进行

两种典型的商品陈列和商品补架的方式
栈 vs 队列

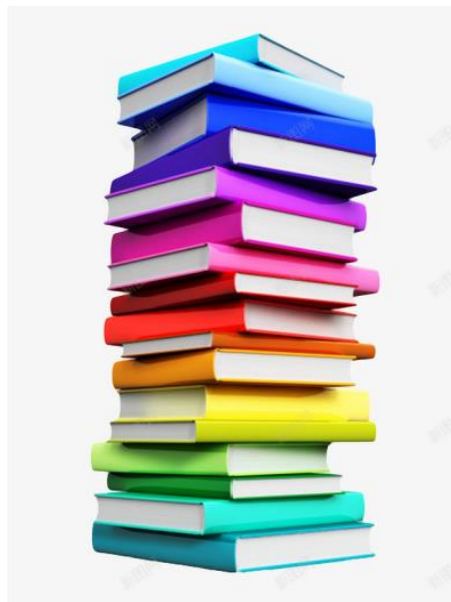
3.1 问题引入：超市货架

- 操作受限的线性表
 - 栈 (Stack)：运算只在表的一端进行
 - 队列 (Queue)：运算只在表的两端进行

- 3.1 问题引入：超市货架管理
- 3.2 栈的定义与结构
- 3.3 队列的定义与结构
- 3.4 栈与队列的应用
- 3.5 拓展延伸
- 3.6 应用场景：消息队列

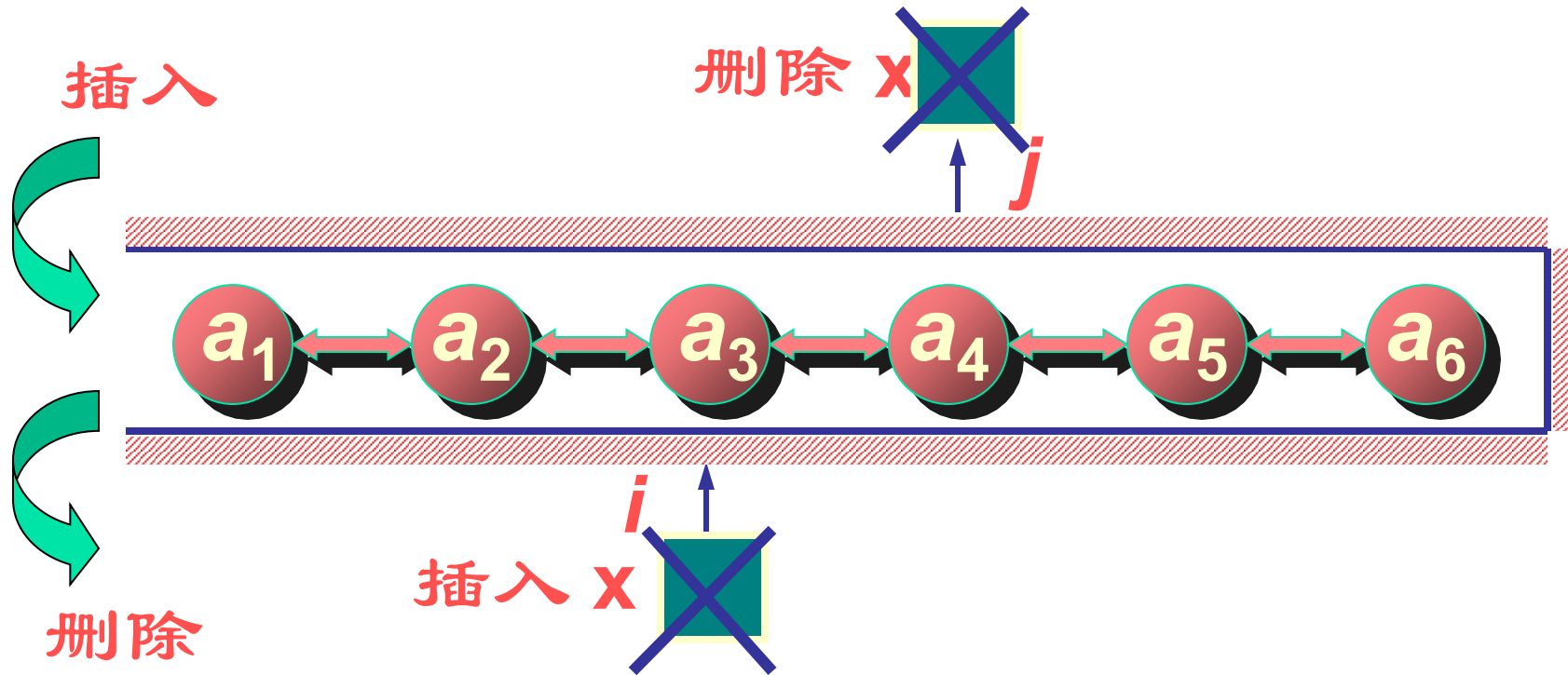
3.2.1 栈的定义

- 堆放盘子、书籍、羽毛球共同点是什么？



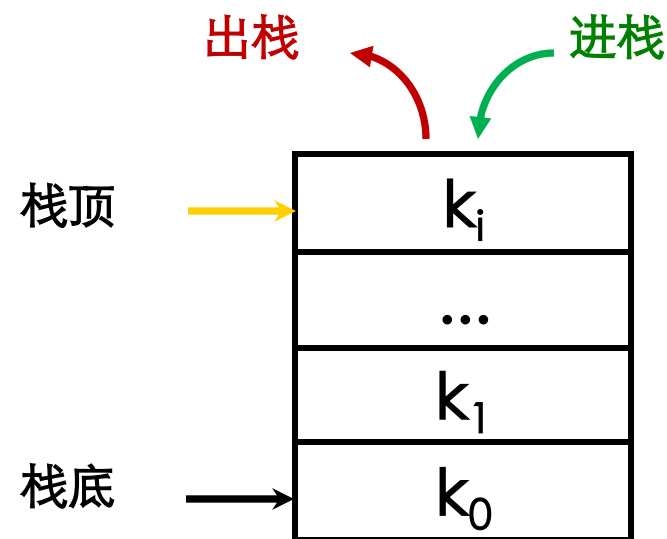
- 一端操作；后进先出（Last In First Out, LIFO）

3.2.1 栈的定义



3.2.1 栈的定义

- **定义：**栈（Stack） 又称堆栈，是只允许在一端进行插入和删除操作的受限的线性表。
 - 一种限制访问端口的线性表
 - 栈存储和删除元素的顺序与元素插入顺序相反
 - 也称为“下推表”
- **特点：**后进先出（Last In First Out, LIFO）
- **主要元素**
 - **栈顶**（top）元素：栈的唯一可访问元素
 - ◆ 元素插入栈称为“入栈”或“压栈”（push）
 - ◆ 删除元素称为“出栈”或“弹出”（pop）
 - 栈底（bottom）：另一端



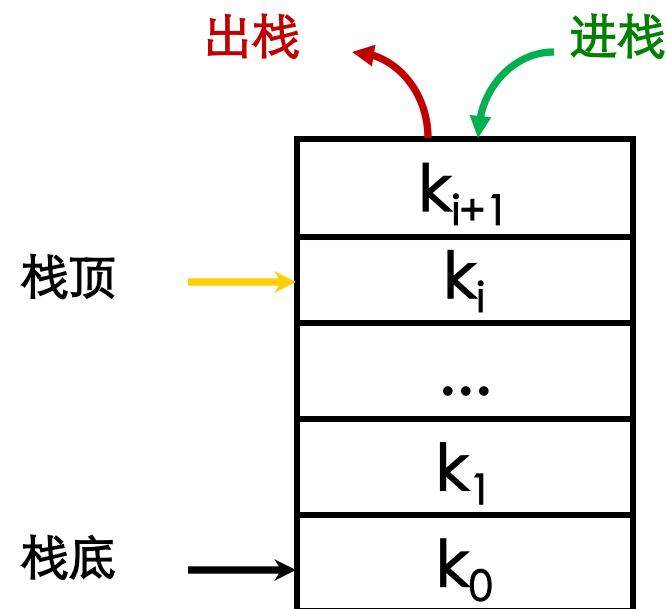
3.2.1 栈的定义

■ 栈的图示

- 每次取出（被删除）的总是刚压进的元素，而最先压入的元素则位于栈的底部
- 空栈：没有元素的栈

■ 应用

- 表达式求值
- 函数调用
- 深度优先搜索



3.2.1 栈的定义

■ 栈的抽象数据类型描述

ADT Stack{

数据对象：一个包含0个或多个元素的有穷线性表。 $D = \{a_i \mid a_i \in \text{Elemset}, i \in \mathbb{N}^+, i \leq n\}$

数据关系： $R = \{ \langle a_i, a_{i+1} \rangle \mid a_i, a_{i+1} \in D, i \leq n \}$

基本操作：

Stack InitStack(&S)：初始化一个空堆栈。

Stack DestoryStack (&S)：销毁栈，并释放栈占用的空间。

bool StackEmpty(S)：判断堆栈S是否为空，返回True|False。

bool StackFull(S)：判断堆栈S是否已满，返回True|False。

bool Push(&S, x)：进栈，若栈未满，则将元素x压入堆栈，x变为新栈顶。

ElementType PoP (&S, &x)：出栈，若栈非空，则弹出栈顶元素，并返回x。

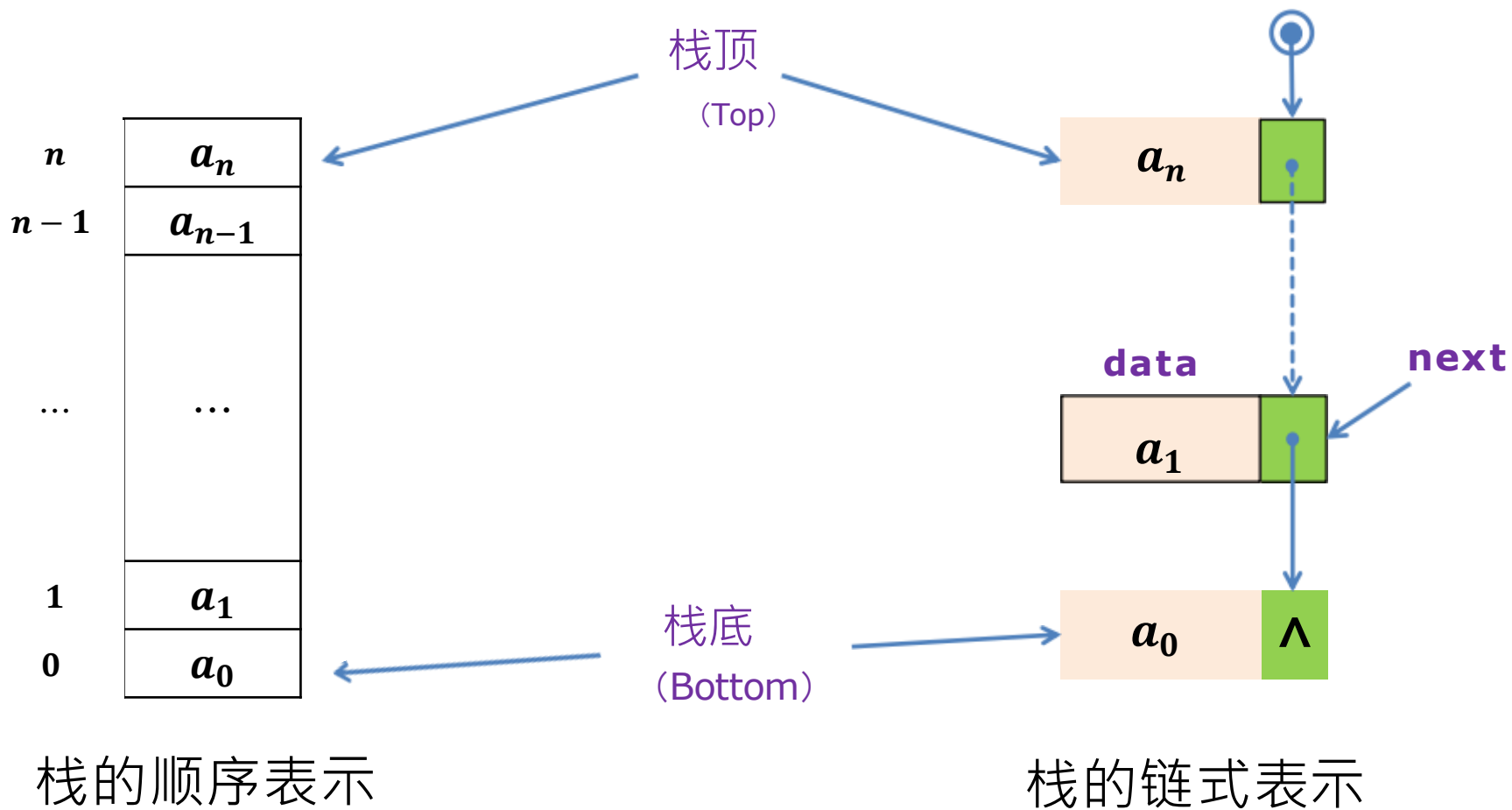
ElementType Top (S, &x)：读栈顶元素，若栈非空，则用x返回栈顶元素。

} ADT Stack

注意区别

3.2.1 栈的定义

■ 栈的顺序表示和链式表示



3.2.1 栈的定义

- 顺序栈 (Array-based Stack)
 - ✓ 采用向量实现, 本质上顺序表的简化版
 - ◆ 栈的大小
 - ✓ 关键: 确定哪一端作为栈顶
 - ✓ 上溢/下溢问题
- 链式栈 (Linked Stack)
 - ✓ 采用单链表方式存储, 指针的方向是从栈顶向下链接

3.2.1 栈的定义

■ 栈的混洗/出栈序列的合法性

- “混洗”原意是重新洗牌。用在本领域，问题的提法是：当进栈元素的编号为 $1, 2, \dots, n$ 时，可能的出栈序列有多少种？
- 当进栈序列为 $1, 2$ 时，可能的出栈序列有2种： $1, 2$ （进1出1进2出2）和 $2, 1$ （进1进2出2出1）；
- 当进栈序列为 $1, 2, 3$ 时，可能的出栈序列有5种：
 - $1, 2, 3$ （进1出1进2出2进3出3）
 - $1, 3, 2$ （进1出1进2进3出3出2）
 - $2, 1, 3$ （进1进2出2出1进3出3）
 - $2, 3, 1$ （进1进2出2进3出3出1）
 - $3, 2, 1$ （进1进2进3出3出2出1）
- 注意， **$3, 1, 2$ 是不可能的出栈序列**，因为若3第1个出栈，栈内一定是1压在2的下面，1不可能先于2出栈。

3.2.1 栈的定义

■ 栈的混洗/出栈序列的合法性

■ 一般情形如何呢？若设进栈序列为 $1, 2, \dots, n$ ，可能的出栈序列有 m_n 种，则

■ $n=0$ 时， $m_0 = 1$: 出栈序列为 $\{\}$ 。

■ $n=1$ 时， $m_1 = 1$: 出栈序列为 $\{1\}$ 。

■ $n=2$ 时， $m_2 = 2$:

① 出栈序列中 1 在首位，1 左侧有 0 个数，右侧有 1 个数，有 $m_0 * m_1 = 1$ 种出栈序列： $\{1, 2\}$

② 出栈序列中 1 在末位，1 左侧有 1 个数，右侧有 0 个数，有 $m_1 * m_0 = 1$ 种出栈序列： $\{2, 1\}$ 。

■ 可能出栈序列有 $m_0 * m_1 + m_1 * m_0 = 2$ 种。

3.2.1 栈的定义

■ 栈的混洗/出栈序列的合法性

■ 一般情形如何呢？若设进栈序列为 $1, 2, \dots, n$ ，可能的出栈序列有 m_n 种，则

■ $n = 3$ 时， $m_3 = 5$ ：

- ① 出栈序列中1在首位，1左侧有0个数，右侧有2个数，有 $m_0 * m_2 = 2$ 种出栈序列： $\{1, 2, 3\}$ 和 $\{1, 3, 2\}$ 。
- ② 出栈序列中1在第2位，1左侧1个数，右侧1个数，有 $m_1 * m_1 = 1$ 种出栈序列： $\{2, 1, 3\}$ 。
- ③ 出栈序列中1在第3位。1左侧2个数，右侧0个数，有 $m_2 * m_0 = 2$ 种出栈序列： $\{2, 3, 1\}$ 和 $\{3, 2, 1\}$ 。

■ 可能的出栈序列有 $m_0 * m_2 + m_1 * m_1 + m_2 * m_0 = 2 + 1 + 2 = 5$ 种。

3.2.1 栈的定义

■ 栈的混洗/出栈序列的合法性

■ 一般情形如何呢？若设进栈序列为 $1, 2, \dots, n$ ，可能的出栈序列有 m_n 种，则

■ $n = 4$, $m_4 = 14$:

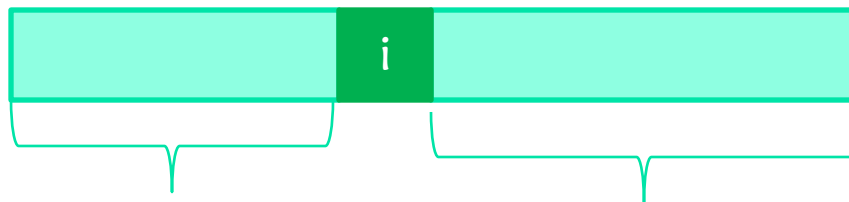
- ① 出栈序列中 1 在第 1 位。1 左侧 0 个数，右侧 3 个数，有 $m_0 * m_3 = 5$ 种出栈序列：
 $\{1, 2, 3, 4\}, \{1, 2, 4, 3\}, \{1, 3, 2, 4\}, \{1, 3, 4, 2\}, \{1, 4, 3, 2\}$ 。
 - ② 出栈序列中 1 在第 2 位。1 左侧 1 个数，右侧 2 个数，有 $m_1 * m_2 = 2$ 种出栈序列：
 $\{2, 1, 3, 4\}, \{2, 1, 4, 3\}$ 。
 - ③ 出栈序列中 1 在第 3 位。1 左侧 2 个数，右侧 1 个数，有 $m_2 * m_1 = 2$ 种出栈序列：
 $\{2, 3, 1, 4\}, \{3, 2, 1, 4\}$ 。
 - ④ 出栈序列中 1 在第 4 位。1 左侧 3 个数，右侧 0 个数，有 $m_3 * m_0 = 5$ 种出栈序列：
 $\{2, 3, 4, 1\}, \{2, 4, 3, 1\}, \{3, 2, 4, 1\}, \{3, 4, 2, 1\}, \{4, 3, 2, 1\}$ 。
- 可能出栈序列 $m_0 * m_3 + m_1 * m_2 + m_2 * m_1 + m_3 * m_0 = 5 + 2 + 2 + 5 = 14$ 种。

3.2.1 栈的定义

■ 栈的混洗/出栈序列的合法性

- 一般地，有 n 个元素按序号 $1, 2, \dots, n$ 进栈，轮流让 1 在出栈序列的第 $1, 2, \dots, n$ 位，则可能的出栈序列数为：

$$\sum_{i=0}^{n-1} m_i * m_{n-i-1} = m_0 * m_{n-1} + m_1 * m_{n-2} + \dots + m_{n-1} * m_0$$



i 个元素

$n - i - 1$ 个元素

- 推理结果（卡特兰数 C_n ）

$$\sum_{i=0}^{n-1} m_i * m_{n-i-1} = \frac{1}{n+1} C_{2n}^n$$

3.2.1 栈的定义

■ 栈的混洗/出栈序列的合法性

- 例题1: 元素a, b, c, d, e依次进入初始为空的栈中, 若元素进栈后可停留、可出栈, 直到所有元素都出栈, 则在所有可能的出栈序列中, 以元素d开头的序列个数是

A. 3 B. 4 C. 5 D. 6

- 解答: 选B。如果d在出栈序列中排在开头, a, b, c必须在d出栈前压在栈内, d出栈后它们才可出栈, 出栈顺序必须是c, b, a。e可夹在它们之间进栈和出栈。因此, 以d开头的出栈序列只可能是

d, e, c, b, a d, c, e, b, a d, c, b, e, a d, c, b, a, e

3.2.1 栈的定义

- 设 $f(m, n)$ 表示 “栈内有 m 个元素，待进栈有 n 个元素（按顺序）” 时的出栈序列数

- 递归公式: $f(m, n) = f(m-1, n) + f(m+1, n-1)$

(解释: 要么弹出栈顶元素, 此时栈内剩 $m-1$ 个; 要么压入下一个待进栈元素, 此时栈内变为 $m+1$ 个, 待进栈剩 $n-1$ 个)

- 边界条件

- 若 $m=0$ 且 $n=0$: 只有 1 种序列 (空序列);
 - 若 $n=0$ (无待进栈元素): 只能按栈顶到栈底顺序出栈, 仅 1 种序列, 即 $f(m, 0) = 1$;
 - 若 $m=0$ (栈为空): 等价于 n 个元素的出栈序列数 (卡特兰数), 即 $f(0, n) = C_n$, C_n 为第 n 个卡特兰数。
- 例题1: 元素 a, b, c, d, e 依次进入初始为空的栈中, 若元素进栈后可停留、可出栈, 直到所有元素都出栈, 则在所有可能的出栈序列中, 以元素 d 开头的序列个数是

$$\begin{aligned} f(3, 1) &= f(2, 1) + f(4, 0) \\ &= f(1, 1) + f(3, 0) + f(4, 0) \\ &= f(0, 1) + f(2, 0) + f(3, 0) + f(4, 0) = 4 \end{aligned}$$

- 扩展: 元素 a, b, c, d, e, f 依次进入初始为空的栈中, 若元素进栈后可停留、可出栈, 直到所有元素都出栈, 则在所有可能的出栈序列中, 以元素 d 开头的序列个数 $f(3, 2)$

3.2.1 栈的定义

■ 栈的混洗/出栈序列的合法性

- 例题2: 若元素的进栈序列为ABCDE, 运用栈操作, 能否得到出栈序列BCAED和DBACE? 为什么?

- 解答:

- 可以得到BCAED, 其顺序为A进, B进, B出, C进, C出, A出, D进, E进, E出, D出。
- 无法得到DBACE。D是第一个出来的, 说明ABC是按顺序进站。所以, 出栈就只能是CBA。

因此题目给出的BAC是错误的。

解题技巧:

1. 从输出开始, 放入堆栈并依次对比
2. 每个输出执行前, 确保前面的元素都入栈
3. 按照LIFO的原则, 检查是否能完成所有元素的输出

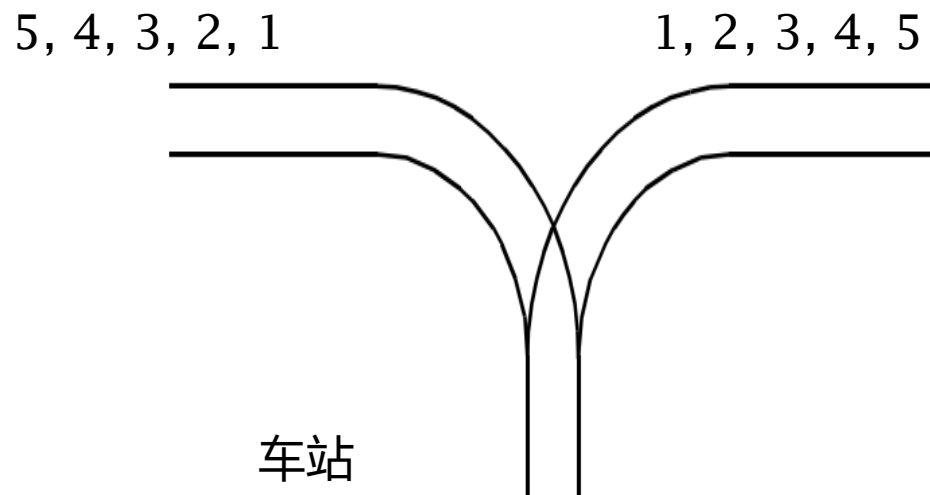
3.2.1 栈的定义

■ 栈的混洗/出栈序列的合法性

- 例题3: 判断火车的出栈顺序是否合法

- <http://poj.org/problem?id=1363>

- 编号为1, 2, ..., n 的 n 辆火车依次进站, 给定一个 n 的排列, 判断是否为合法的出站顺序?

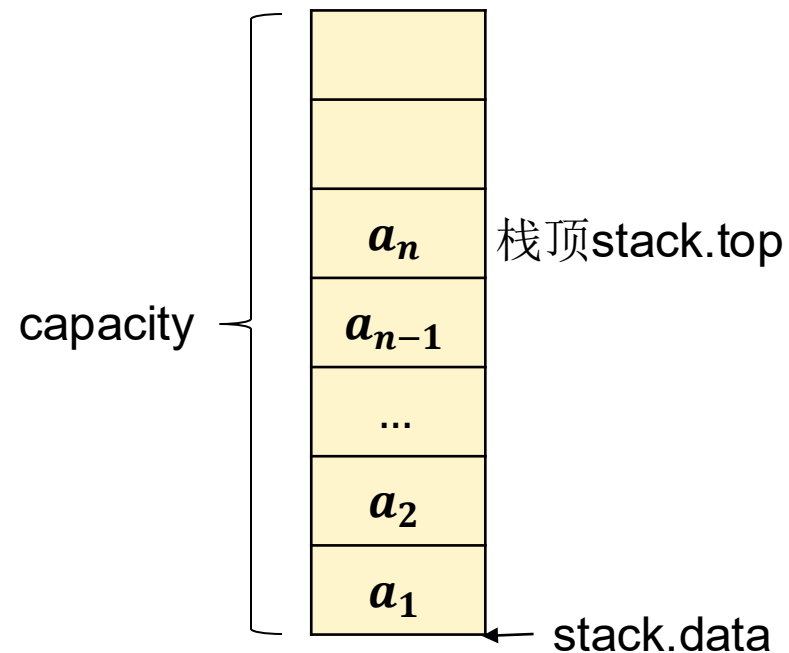


3.2.2 栈的顺序存储实现

■ 顺序栈：采用顺序存储结构的栈

```
typedef enum {false, true} bool;
typedef int SElemSet; /* 栈元素类型为int型 */
#define NIL -1

typedef int Position; /* 整型下标，表示元素的位置 */
typedef struct StackNode *Stack;
struct StackNode {
    int capacity; /* 顺序栈的容量 */
    Position top; /* 顺序栈的栈顶指针，初始化为-1 */
    SElemSet *data; /* 存储数据的数组 */
};
```



3.2.2 栈的顺序存储实现

■ 顺序栈的初始化

代码：顺序实现的栈的初始化 `InitStack (stack, kSize)`

输入：栈 `stack` 和正整数 `kSize`

输出：一个大小为 `kSize` 的顺序栈

1. `stack.capacity` \leftarrow `kSize`
2. `stack.top` \leftarrow -1
3. `stack.data` \leftarrow **new** ElemSet[`kSize`]

```
void InitStack(Stack stack, int kMaxSize)
{ /* 初始化一个大小为kMaxSize的顺序栈 */
    stack->capacity = kMaxSize;
    stack->top = -1;
    stack->data = (SElemSet *)malloc(sizeof(SElemSet)*kMaxSize);
}
```

3.2.2 栈的顺序存储实现

■ 顺序栈的入栈操作

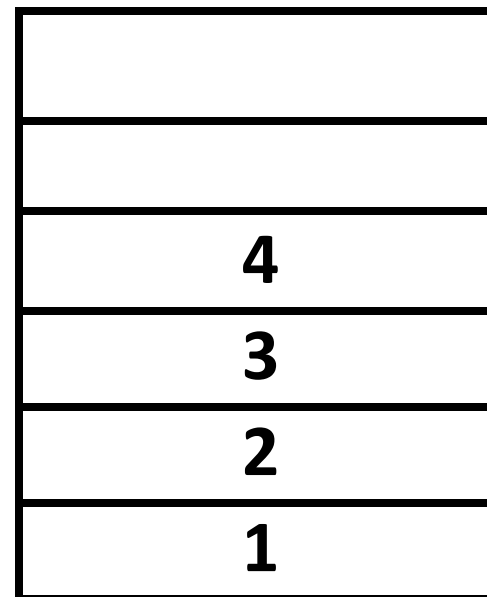
算法：顺序栈的入栈操作 $\text{Push}(stack, x)$

输入： 栈 $stack$ 和待压入的元素 x

输出： 压入 x 后的顺序栈；若栈满，则退出

1. **if** $\text{IsFull}(stack) = \text{true}$ **then**
2. | 栈满，退出
3. **else**
4. | $stack.top \leftarrow stack.top + 1$
5. | $stack.data[stack.top] \leftarrow x$
6. **end**

栈顶



3.2.2 栈的顺序存储实现

■ 顺序栈的入栈操作

```
bool IsFull(Stack stack)
{ /* 判断栈是否已满 */
    if ((stack->top+1) == stack->capacity)
        return true;
    else
        return false;
}

/* 算法3-1: 顺序栈的入栈操作 Push (stack, x) */
void Push (Stack stack, SElemSet x)
{
    if (IsFull(stack)) {
        printf("错误: 栈已满。 \n");
    }
    else {
        stack->top++;
        stack->data[stack->top] = x;
    }
}
```

3.2.2 栈的顺序存储实现

■ 顺序栈的取栈顶操作

算法：顺序栈的取顶操作 $\text{Top}(stack)$

输入：栈 $stack$

输出：栈顶元素；若栈空，则输出NIL

1. **if** $\text{IsEmpty}(stack) = \text{true}$ **then**
2. | **return** NIL
3. **else**
4. | **return** $stack.data[stack.top]$
5. **end**

3.2.2 栈的顺序存储实现

■ 顺序栈的取栈顶操作

```
bool IsEmpty(Stack stack)
{ /* 判断栈是否为空 */
    if (stack->top == -1)
        return true;
    else
        return false;
}

SElemSet Top (Stack stack)
{
    if (IsEmpty(stack)) {
        printf("错误： 栈为空。 \n");
        return NIL;
    }
    else {
        return stack->data[stack->top];
    }
}
```

3.2.2 栈的顺序存储实现

■ 顺序栈的出栈操作

算法：顺序栈的出栈操作 Pop (*stack*)

输入：栈*stack*

输出：删除栈顶元素后的顺序栈；若栈空，则退出

1. **if** IsEmpty(*stack*)=**true** **then**
2. | 栈空, 退出
3. **else**
4. | $stack.top \leftarrow stack.top - 1$
5. **end**

3.2.2 栈的顺序存储实现

■ 顺序栈的出栈操作

```
/* 算法3-3: 顺序栈的出栈操作 Pop (stack) */  
void Pop (Stack stack)  
{  
    if (IsEmpty(stack)) {  
        printf("错误: 栈为空。 \n");  
    }  
    else {  
        stack->top--;  
    }  
}
```

3.2.2 栈的顺序存储实现

■ 不同栈顶设置

- 表的首位置为栈顶：Push、Pop，时间复杂度为 $O(n)$
- 表尾元素为栈顶：Push、Pop，时间复杂度为 $O(1)$

■ 顺序栈的溢出

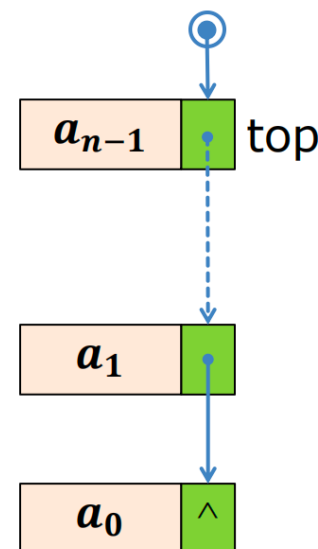
- 上溢 (Overflow)
 - 栈中已有maxsize个元素时，再做进栈运算时产生的现象
- 下溢 (Underflow)
 - 对空栈进行出栈运算时所产生的现象

3.2.3 栈的链式存储实现

■ 链式栈（LinkStack）：栈的链式存储结构实现

- 一种受限的单链表，称为链栈、链式栈。
- 栈顶元素设置为链表头（插入、删除更加高效）。
- 在链栈中，结点的插入（入栈）和删除（出栈）操作只能在栈顶进行。
- 有效地提高了存储效率，且不存在栈满上溢的问题。

```
typedef struct StackNode *Position; /* 指针即结点位置 */
struct StackNode {
    SElemSet data; /* 存储数据 */
    Position next; /* 链式栈中下一个元素的位置 */
};
typedef struct StackHeadNode *Stack;
struct StackHeadNode {
    int size; /* 链式栈中当前元素个数 */
    Position top; /* 链式栈的栈顶指针，初始化为NULL */
};
```



3.2.3 栈的链式存储实现

■ 链式栈的初始化

代码：链式栈的初始化 InitStack (*stack*)

输入：无

输出：一个空的链式栈

1. $stack.size \leftarrow 0$
2. $stack.top \leftarrow NIL$

```
stack = (Stack)malloc(sizeof(struct StackHeadNode));
```

```
void InitStack(Stack stack)
{ /* 初始化一个空的链式栈 */
    stack->size = 0;
    stack->top = NULL;
}
```

3.2.3 栈的链式存储实现

■ 链式栈的入栈Push

算法：链式栈的入栈操作 $\text{Push}(stack, x)$

输入：栈 $stack$ 和待压入的元素 x

输出：压入元素的链式栈

1. $\text{new_node} \leftarrow \text{new StackNode}$
2. $\text{new_node.data} \leftarrow x$
3. $\text{new_node.next} \leftarrow \text{stack.top}$
4. $\text{stack.top} \leftarrow \text{new_node}$
5. $\text{stack.size} \leftarrow \text{stack.size} + 1$

链表首结点前插入新节点, $O(1)$

3.2.3 栈的链式存储实现

■ 链式栈的入栈Push

```
/* 算法3-4: 链式栈的入栈操作 Push (stack, x) */  
void Push (Stack stack, SElemSet x)  
{  
    Position new_node;  
    new_node = (Position)malloc(sizeof(struct StackNode));  
    new_node->data = x;  
    new_node->next = stack->top;  
    stack->top = new_node;  
    stack->size++;  
}
```

3.2.3 栈的链式存储实现

■ 链式栈的取栈顶Top

算法：链式栈的取顶操作 Top (*stack*)

输入：栈*stack*

输出：栈顶元素；若栈空，则输出NIL

```
1. if IsEmpty(stack)=true then
2. | return NIL
3. else
4. | return stack.top.data
5. end
```

取链表首结点， $O(1)$

3.2.3 栈的链式存储实现

■ 链式栈的取栈顶Top

```
bool IsEmpty(Stack stack)
{ /* 判断栈是否为空 */
    if (stack->size == 0)
        return true;
    else
        return false;
}
```

```
/* 算法3-5: 链式栈的取顶操作 Top (stack) */
SElemSet Top (Stack stack)
{
    if (IsEmpty(stack)) {
        printf("错误: 栈为空.\n");
        return NIL;
    }
    else {
        return stack->top->data;
    }
}
```

3.2.3 栈的链式存储实现

■ 链式栈的出栈Pop

算法：链式栈的出栈操作 Pop (*stack*)

输入：栈*stack*

输出：删除栈顶元素后的链式栈；若栈空，则退出

1. **if** IsEmpty(*stack*)=**true** **then**
2. | 栈空，退出
3. **else**
4. | $temp \leftarrow stack.top$
5. | $stack.top \leftarrow stack.top.next$
6. | **delete** *temp*
7. | $stack.size \leftarrow stack.size - 1$
8. **end**

删除链表首结点， $O(1)$

3.2.3 栈的链式存储实现

■ 链式栈的出栈Pop

```
/* 算法3-6: 链式栈的出栈操作 Pop (stack) */  
void Pop (Stack stack)  
{  
    Position temp;  
    if (IsEmpty(stack)) {  
        printf("错误: 栈为空。 \n");  
    }  
    else {  
        temp = stack->top;  
        stack->top = stack->top->next;  
        free(temp);  
        stack->size--;  
    }  
}
```

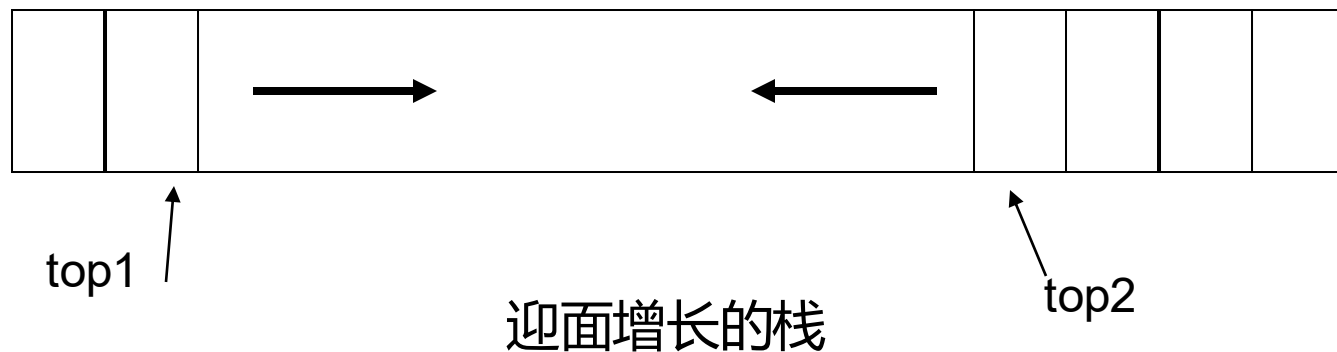
3.2.3 栈的链式存储实现

- 实际中顺序栈应用更广泛些
 - ✓ 存储开销低
 - ✓ 根据栈顶位置的相对位移，快速定位读取栈的内部元素
 - 顺序栈 读取内部元素 的时间为 $O(1)$ ，而链式栈则需要沿指针链游走，读取第 k 个元素需要时间为 $O(k)$
 - 尽管一般来说，栈不允许“读取内部元素”，只能在栈顶操作

3.2.4 栈的变种

■ 两个独立的栈

- 共享栈：迎面增长
- 双栈：底部相连



- 3.1 问题引入：超市货架管理
- 3.2 栈的定义与结构
- 3.3 队列的定义与结构
- 3.4 栈与队列的应用
- 3.5 拓展延伸
- 3.6 应用场景：消息队列

3.3.1 队列的定义

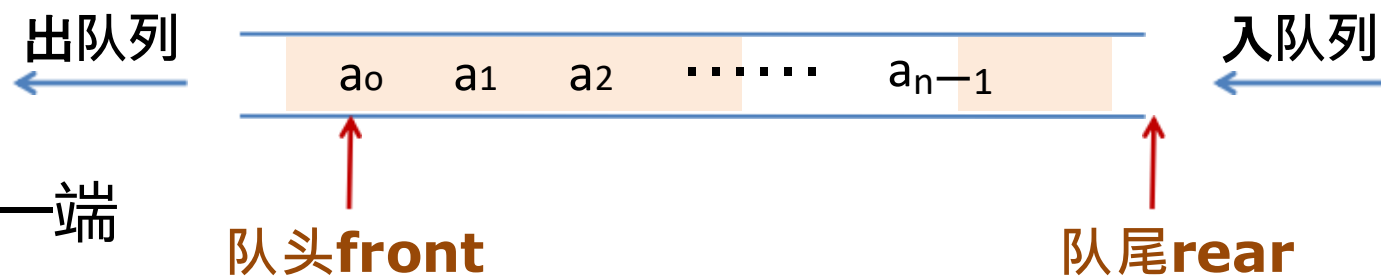
■ 先进先出 (First In First Out)

- 按照到达的顺序来释放元素
- 限制访问点的线性表
 - 所有的插入在表的一端进行, 所有的删除都在表的另一端进行
 - 特例: 空队列



■ 主要元素

- 队头 (front) : 允许删除的一端
- 队尾 (rear) : 允许插入的一端



3.3.1 队列的定义

- 队列的主要操作
 - 入队 (enqueue) (插入)
 - 出队 (dequeue) (删除)
 - 取队首 (getFront)
 - 判断队列是否为空 (isEmpty)

3.3.1 队列的定义

■ 队列的抽象数据类型

代码：队列的抽象数据类型定义

ADT Queue {

数据对象： $\{ ai | ai \in \text{ElemSet}, i=1,2,\dots,n, n > 0 \}$ 或 Φ ，即空表；ElemSet为元素集合。

数据关系： $\{ \langle ai, ai+1 \rangle | ai, ai+1 \in \text{ElemSet}, i=0,1,\dots,n-2 \}$ 。

基本操作：

InitQueue(queue): 初始化一个空的队列queue。

DestroyQueue(queue): 释放队列queue占用的所有空间。

Clear(queue): 清空队列queue。

IsEmpty(queue): 队列queue为空返回真，否则返回假。

IsFull(queue): 队列queue满返回真，否则返回假。

GetFront(queue): 返回队列queue的队首结点，队首结点不变。

EnQueue(queue, x): 将结点x插入队列queue，使其成为新的队尾。

DeQueue(queue): 将队首结点从队列queue删除。

}

3.3.1 队列的定义

■ 队列的实现方式

■ 顺序队列

- 关键：如何防止假溢出
- 链式队列：采用单链表方式存储，每个元素对应链表中一个结点

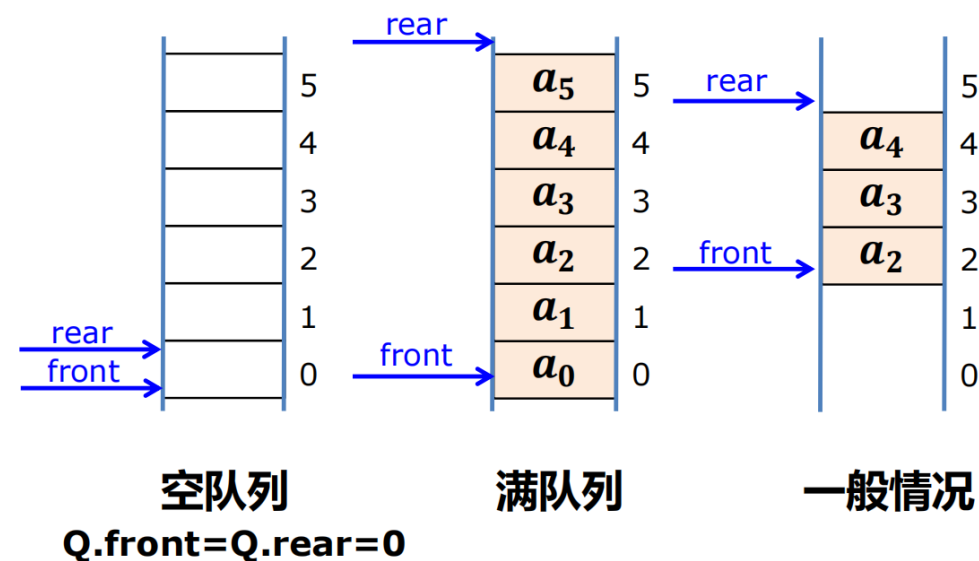
■ 队列的溢出

- 上溢：当队列满时，入队操作所出现的现象
- 下溢：当队列空时，出队操作所出现的现象
- 假溢出
 - 顺序队列可能出现的一种现象：当队尾指针达到最大值（ $\text{rear} = \text{mSize}$ ）时，再做入队运算产生溢出，但此时队列前端可能尚有空闲位置

3.3.2 队列的顺序存储实现

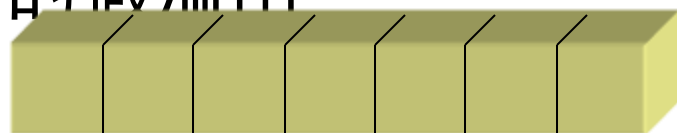
- 队列（Queue）的顺序存储结构是由一块连续的存储单元存放队列的**一维数组**，它附设两个指针，**队头指针（front）**指向**队头元素**，**队尾指针（rear）**指向**队尾元素的下一个位置**。
- 队列与栈的共性在于它们都是**限制了存取位置的线性表**；区别在于存取位置有所不同。

```
typedef int Position; /* 整型下标，表示元素的位置 */
typedef struct QueueNode *Queue;
struct QueueNode {
    int capacity; /* 顺序队列的容量 */
    Position front; /* 顺序队列的队首指针，初始化为0 */
    Position rear; /* 顺序队列的队尾指针，初始化为0 */
    QElemSet *data; /* 存储数据的数组 */
};
```

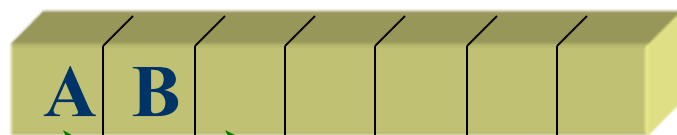


3.3.2 队列的顺序存储实现

■ 顺序队列的假溢出



front rear **空队列**



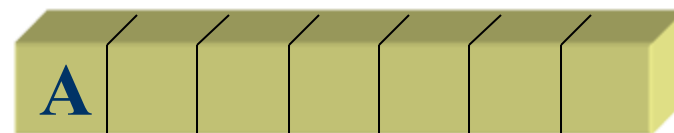
front rear **B 进队**



front rear **A 退队**



front rear **E, F, G 进队**



front rear **A 进队**



front rear **C, D 进队**



front rear **B 退队**



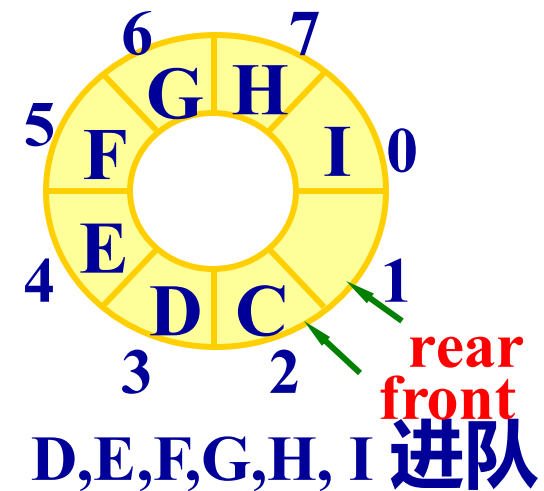
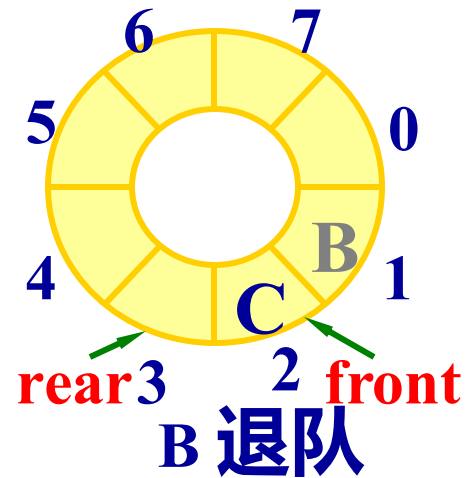
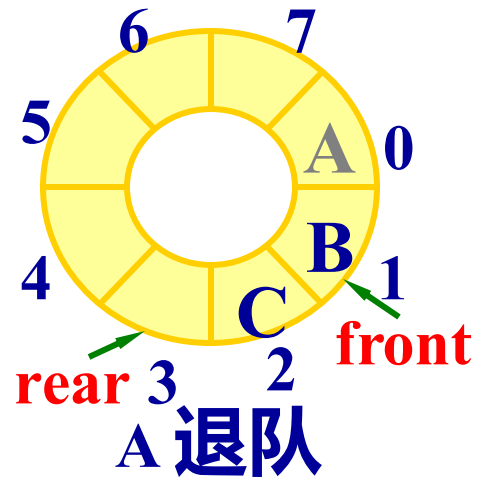
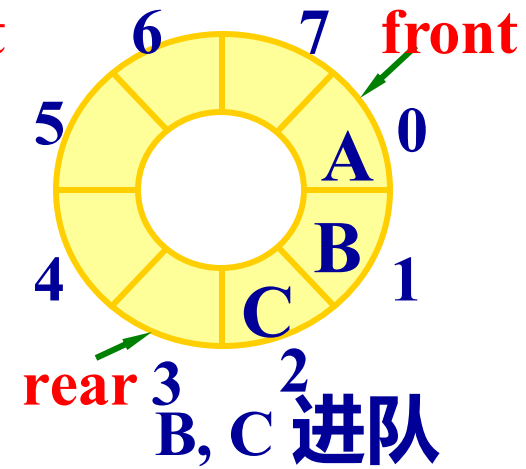
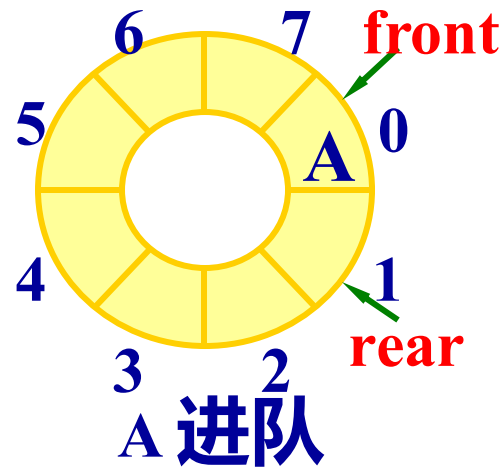
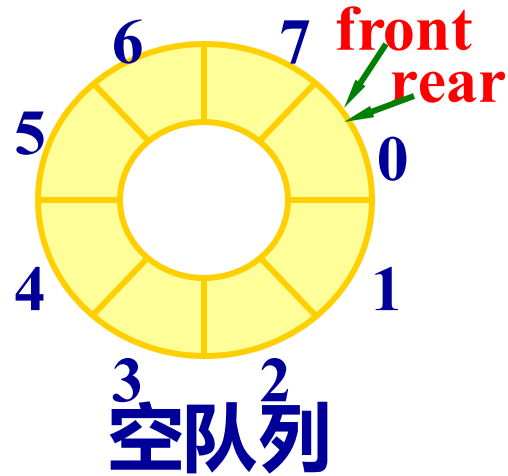
front rear **H 进队, 溢出**

3.3.2 队列的顺序存储实现

- 顺序队列的假溢出：尚有空闲位置发生的溢出
- 如何解决顺序队列的假溢出？
 - 将顺序表在逻辑上视为一个环：下标编号最小的位置0视为最大位置 ($capacity - 1$) 的直接后继
 - 位置 x 的直接后继位置是 $(x + 1) \% capacity$

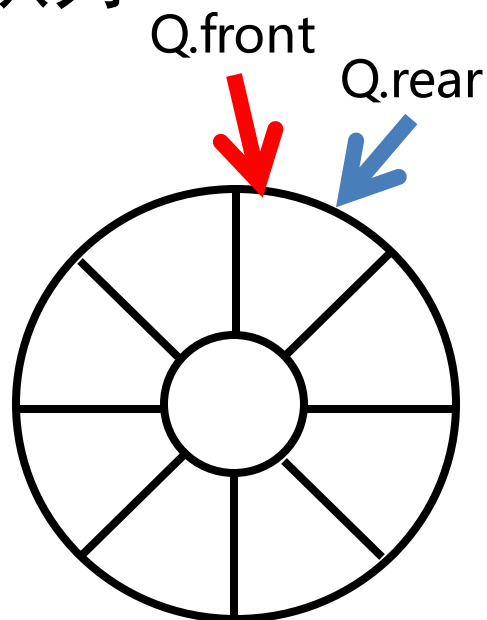
3.3.2 队列的顺序存储实现

■ 循环队列



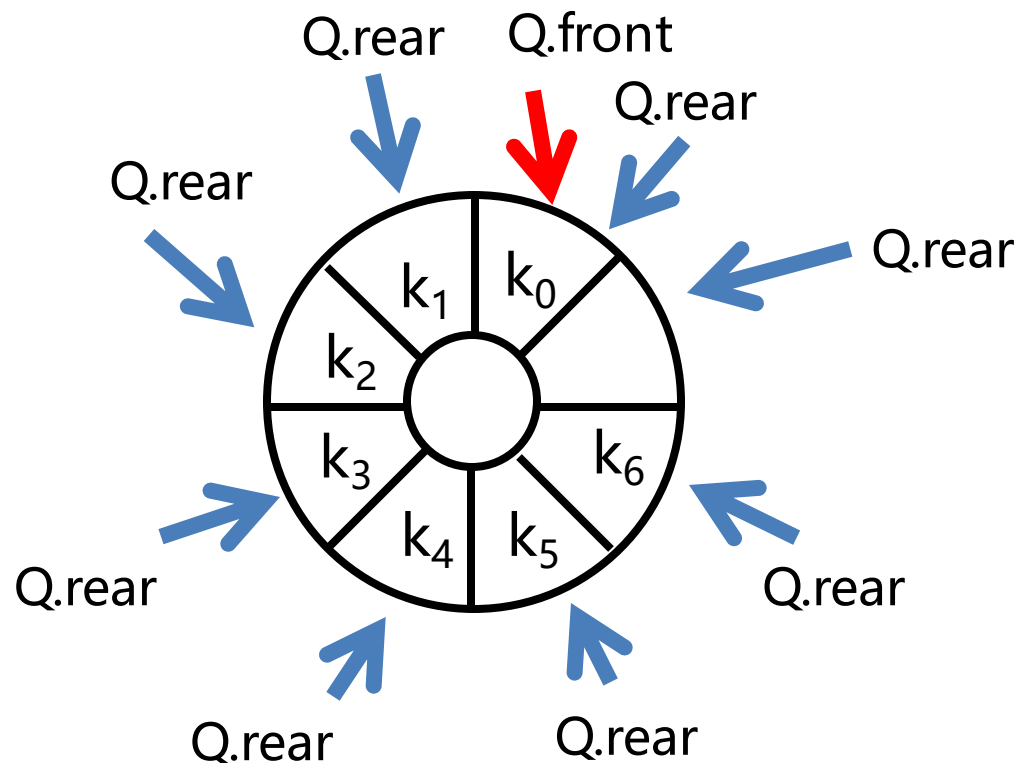
3.3.2 队列的顺序存储实现

■ 循环队列



队列空:

$$Q.rear == Q.front$$



队列满:

$$(Q.rear + 1) \bmod (n + 1) == Q.front$$

3.3.2 队列的顺序存储实现

- 队列存放数组被当作首尾相接的环形表处理。
- 队头、队尾指针加 1 时从capacity-1直接进到0，可用语言的取模 (%) 运算实现。
 - 队头指针进1: $\text{front} = (\text{front} + 1) \% \text{capacity};$
 - 队尾指针进1: $\text{rear} = (\text{rear} + 1) \% \text{capacity};$
 - 队列初始化: $\text{front} = \text{rear} = 0;$
 - 队空条件: $\text{front} == \text{rear};$
 - 队满条件: $(\text{rear} + 1) \% \text{capacity} == \text{front}.$
- 注意，进队和出队时指针都是顺时针前进。
- 使用大小为 $n + 1$ 的顺序表存储具有 n 个元素的队列
 - 从而区分队尾的 $n + 1$ 种情况; $\text{capacity} = \text{kMaxsize} + 1$

3.3.2 队列的顺序存储实现

■ 循环队列的初始化

代码：顺序队列的初始化 `InitQueue(queue, kSize)`

输入：队列 `queue` 和正整数 `kSize`

输出：一个大小为 `kSize` 的顺序队列

1. `queue.capacity` \leftarrow `kSize+1`
 2. `queue.data` \leftarrow `new ElemSet[kSize+1]` // 浪费一个存储空间以区别空和满
 3. `queue.front` \leftarrow 0
 4. `queue.rear` \leftarrow 0
 5. _____
-

3.3.2 队列的顺序存储实现

■ 循环队列的初始化

```
void InitQueue(Queue queue, int kMaxSize)
{ /* 初始化一个大小为kMaxSize的顺序队列 */
    queue->capacity = kMaxSize+1;
    /* 浪费一个存储空间以区别空和满 */
    queue->data = (QElemSet *)malloc(sizeof(QElemSet)*(kMaxSize+1));
    queue->front = 0;
    queue->rear = 0;
}
```

3.3.2 队列的顺序存储实现

■ 判断队列是否满或者空

```
bool IsFull(Queue queue)
{ /* 判断队列是否已满 */
    if ((queue->rear+1)%queue->capacity == queue->front)
        return true;
    else
        return false;
}
```

```
bool IsEmpty(Queue queue)
{ /* 判断队列是否为空 */
    if (queue->front == queue->rear)
        return true;
    else
        return false;
}
```

3.3.2 队列的顺序存储实现

■ 循环队列的入队操作

算法3-7: 顺序队列的入队操作 $\text{EnQueue}(\text{queue}, x)$

输入: 队列 queue 和待入队的元素 x

输出: x 入队后的顺序队列; 若队列满, 则退出

1. if $\text{IsFull}(\text{queue}) = \text{true}$ then
2. | 队列满, 退出
3. else
4. | $\text{queue.data}[\text{queue.rear}] \leftarrow x$
5. | $\text{queue.rear} \leftarrow (\text{queue.rear} + 1) \% \text{queue.capacity}$ // 循环后继
6. end

3.3.2 队列的顺序存储实现

■ 循环队列的入队操作

```
/* 算法3-7: 顺序队列的入队操作 EnQueue(queue, x) */  
void EnQueue(Queue queue, QElemSet x)  
{  
    if (IsFull(queue)) {  
        printf("错误: 队列已满. \n");  
    }  
    else {  
        queue->data[queue->rear] = x;  
        /* 循环后继 */  
        queue->rear = (queue->rear+1)%queue->capacity;  
    }  
}
```

3.3.2 队列的顺序存储实现

■ 循环队列的查看队首操作

算法：顺序队列的查看队首操作 $\text{GetFront}(\text{queue})$

输入：队列 queue

输出：队列的头元素；若队列空，则输出NIL

1. if $\text{IsEmpty}(\text{queue}) = \text{true}$ then
2. | return NIL
3. else
4. | return $\text{queue.data}[\text{queue.front}]$
5. end

3.3.2 队列的顺序存储实现

■ 循环队列的查看队首操作

```
/* 算法3-8: 顺序队列的查看队首操作 GetFront(queue) */
QElemSet GetFront(Queue queue)
{
    if (IsEmpty(queue)) {
        printf("错误: 队列为空.\n");
        return NIL;
    }
    else {
        return queue->data[queue->front];
    }
}
```

3.3.2 队列的顺序存储实现

■ 循环队列的出队操作

算法：顺序队列的出队操作 $\text{DeQueue}(queue)$

输入：队列 $queue$

输出：删除队列头元素后的顺序队列；若队列空，则退出

1. if $\text{IsEmpty}(queue) = \text{true}$ then
2. | 队列空，退出
3. else
4. | $queue.front \leftarrow (queue.front + 1) \% queue.capacity$
5. end

3.3.2 队列的顺序存储实现

■ 循环队列的出队操作

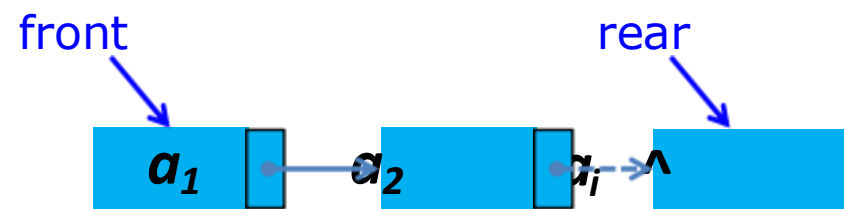
```
/* 算法3-9: 顺序队列的出队操作 DeQueue(queue) */  
void DeQueue(Queue queue)  
{  
    if (IsEmpty(queue)) {  
        printf("错误: 队列为空。 \n");  
    }  
    else {  
        queue->front = (queue->front+1)%queue->capacity;  
    }  
}
```

3.3.3 队列的链式存储实现

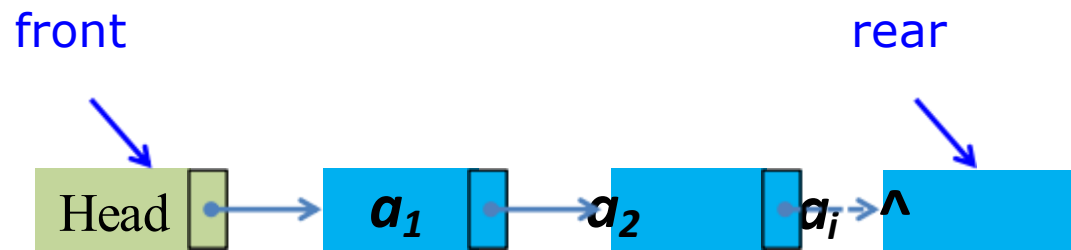
- 队列的链式存储结构称为链式队列（Link Queue），它是一个同时带头指针和尾指针的单链表。头指针（front）指向队头结点（或头结点），尾指针（rear）指向队尾结点（顺序存储通常指向对尾结点的下一结点）。

```
typedef struct QueueNode *Position; /* 指针即结点位置 */
struct QueueNode {
    QElemSet data; /* 存储数据 */
    Position next; /* 链式队列中下一个元素的位置 */
};

typedef struct QueueHeadNode *Queue;
struct QueueHeadNode {
    int size; /* 链式队列中当前元素个数 */
    Position front; /* 链式队列的队首指针 */
    Position rear; /* 链式队列的队尾指针 */
};
```



不带头结点的链队列



带头结点的链队列

3.3.3 队列的链式存储实现

■ 链式队列的初始化

代码：链式队列的初始化 `InitQueue(queue)`

输入：无

输出：一个空的链式队列

1. `queue.size` \leftarrow 0
2. `queue.front` \leftarrow NIL
3. `queue.rear` \leftarrow NIL

```
void InitQueue(Queue queue)
{ /* 初始化一个空的链接队列 */
    queue->size = 0;
    queue->front = NULL;
    queue->rear = NULL;
}
```

3.3.3 队列的链式存储实现

■ 链式队列的入队操作（插入带尾指针的单向链表尾部）

算法：链式队列的入队操作 $\text{EnQueue}(\text{queue}, x)$

输入：队列 queue 和待入队的元素 x

输出： x 入队后的链式队列

1. $\text{new_node} \leftarrow \text{new QueueNode}$
2. $\text{new_node.data} \leftarrow x$
3. $\text{new_node.next} \leftarrow \text{NIL}$
4. if $\text{IsEmpty}(\text{queue}) = \text{true}$ then //特殊处理插入空队列的情况
5. | $\text{queue.rear} \leftarrow \text{new_node}$
6. | $\text{queue.front} \leftarrow \text{new_node}$
7. else
8. | $\text{queue.rear.next} \leftarrow \text{new_node}$
9. | $\text{queue.rear} \leftarrow \text{queue.rear.next}$
10. end
11. $\text{queue.size} \leftarrow \text{queue.size} + 1$

3.3.3 队列的链式存储实现

■ 链式队列的入队操作

```
/* 算法3-10: 链式队列的入队操作 EnQueue(queue, x) */  
void EnQueue(Queue queue, QElemSet x)  
{  
    Position new_node;  
    new_node = (Position)malloc(sizeof(struct QueueNode));  
    new_node->data = x;  
    new_node->next = NULL;  
    if (IsEmpty(queue)) { /* 特殊处理插入空队列的情况 */  
        queue->rear = new_node;  
        queue->front = new_node;  
    }  
    else {  
        queue->rear->next = new_node;  
        queue->rear = queue->rear->next;  
    }  
    queue->size++;  
}
```

3.3.3 队列的链式存储实现

■ 链式队列的查看队首操作

算法3-11: 链式队列的查看队首操作 $\text{GetFront}(\text{queue})$

输入: 队列 queue

输出: 队首元素; 若队列空, 则输出NIL

1. if $\text{IsEmpty}(\text{queue})=\text{true}$ then
2. | return NIL
3. else
4. | return queue.front.data
5. end

3.3.3 队列的链式存储实现

■ 链式队列的查看队首操作

```
/* 算法3-11: 链式队列的查看队首操作 GetFront(queue) */
QElemSet GetFront(Queue queue)
{
    if (IsEmpty(queue)) {
        printf("错误: 队列为空.\n");
        return NIL;
    }
    else {
        return queue->front->data;
    }
}
```

3.3.3 队列的链式存储实现

■ 链式队列的出队操作（删除单向链表首结点）

算法3-12：链式队列的出队操作 DeQueue(queue)

输入：队列queue

输出：删除队首元素后的链式队列；若队列空，则退出

```
1.  if IsEmpty(queue)=true then
2.  |  队列空，退出
3.  else
4.  |  temp ← queue.front
5.  |  queue.front ← queue.front.next
6.  |  delete temp
7.  |  queue.size ← queue.size - 1
8.  |  if queue.front = NIL then // 特殊处理删除后变为空的队列
9.  |  |  queue.rear ← NIL
10. |  end
11. end
```

3.3.3 队列的链式存储实现

■ 链式队列的出队操作

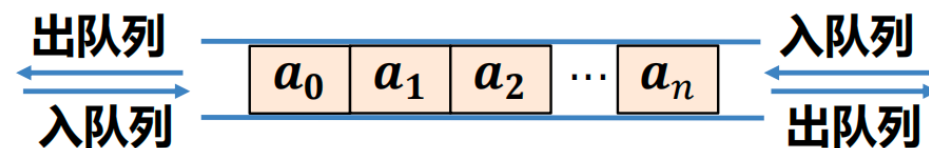
/* 算法3-12: 链式队列的出队操作 DeQueue(queue) */

```
void DeQueue(Queue queue)
{
    Position temp;
    if (IsEmpty(queue)) {
        printf("错误: 队列为空。\\n");
    }
    else {
        temp = queue->front;
        queue->front = queue->front->next;
        free(temp);
        queue->size--;
        if (queue->front == NULL) { /* 特殊处理删除后变为空的队列 */
            queue->rear = NULL;
        }
    }
}
```

3.3.4 队列的变种

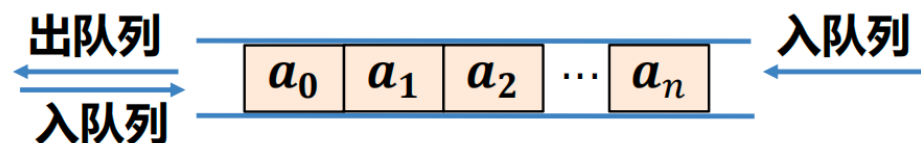
■ 双端队列

- 限制插入和删除在线性表的**两端**进行
- 两个底部相连的栈



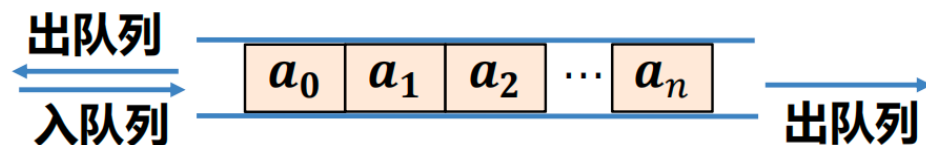
■ 超队列

- 一种**删除受限**的双端队列：**删除只允许在一端进行**，而插入可在两端进行



■ 超栈

- 一种**插入受限**的双端队列，**插入只限制在一端**而删除允许在两端进行



- 3.1 问题引入：超市货架管理
- 3.2 栈的定义与结构
- 3.3 队列的定义与结构
- 3.4 栈与队列的应用
- 3.5 拓展延伸
- 3.6 应用场景：消息队列

3.4 栈与队列的应用

- 栈的特点：后进先出
- 常用来处理具有递归结构的数据
 - 表达式求值
 - 后缀表达式求值
 - 中缀表达式转后缀表达式
 - 中缀表达式求值
 - 递归实现
 - 括号匹配
 - 数制转换
 - 深度优先搜索
 - 行编辑处理

3.4.1 表达式求值

- 表达式求值是一种典型的栈的应用。
- 一个表达式由操作数(亦称运算对象)、操作符 (亦称运算符) 和分界符组成。
- 算术表达式有三种表示:
 - 中缀(infix)表示
 <操作数> <操作符> <操作数>, 如 $A+B$;
 - 前缀(prefix)表示
 <操作符> <操作数> <操作数>, 如 $+AB$;
 - 后缀(postfix)表示
 <操作数> <操作数> <操作符>, 如 $AB+$;

3.4.1 表达式求值

■ 表达式示例

- 中缀表达式 $a + b * (c - d) - e / f$
- 后缀表达式 $a b c d - * + e f / -$

■ 表达式中相邻两个操作符的计算次序为：

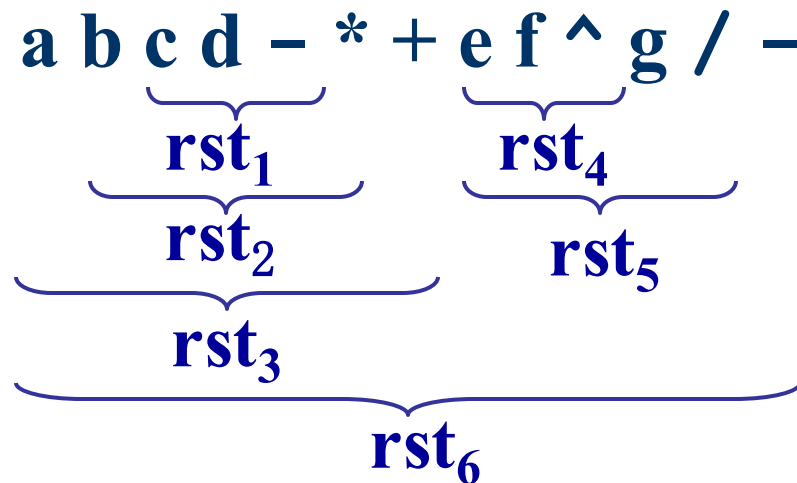
- a) 优先级高的先计算；
- b) 优先级相同的自左向右计算；
- c) 当使用括号时从最内层括号开始计算。

- 当使用中缀表达式计算时，需要同时使用两个栈辅助求值；而使用后缀表达式求值，则只需要一个栈，相对简单一些。

3.4.1 表达式求值

■ 应用后缀表示计算表达式的值

- 从左向右顺序地扫描表达式，并用一个栈暂存扫描到的操作数或计算结果。
- 在后缀表达式的计算顺序中已隐含了加括号的优先次序，括号在后缀表达式中不出现。
- 例 $a b c d - * + e f ^ g / -$ (计算顺序见下标)



3.4.1 表达式求值

■ 应用后缀表示计算表达式的值

- 顺序扫描表达式的每一项，根据它的类型做如下相应操作：
 - a) 若该项是操作数，则将其压栈；
 - b) 若该项是操作符 $\langle op \rangle$ ，则连续从栈中退出两个操作数Y和X，形成运算指令 $X\langle op \rangle Y$ ，并将计算结果重新压栈。
- 当表达式的所有项都扫描并处理完后，栈顶存放的就是最后的计算结果。
- 举例 $a\ b\ c\ d\ -\ *\ +\ e\ f\ ^\ g\ /\ -$

3.4.1 表达式求值

■ 应用后缀表示计算表达式的值

■ 举例 $a\ b\ c\ d\ -\ *\ +\ e\ f\ ^\ g\ /\ -$

步	输入	类型	动 作	栈内容
1			置空栈	空
2	a	操作数	进栈	a
3	b	操作数	进栈	a b
4	c	操作数	进栈	a b c
5	d	操作数	进栈	a b c d
6	-	操作符	d、c 退栈, 计算 $c-d$, 结果 r_1 进栈	a b r_1
7	*	操作符	r_1 、b退栈, 计算 $b*r_1$, 结果 r_2 进栈	a r_2

3.4.1 表达式求值

■ 应用后缀表示计算表达式的值

■ 举例 $a\ b\ c\ d\ -\ *\ +\ e\ f\ ^\ g\ /\ -$

8	+	操作符	r_2 、 a 退栈, 计算 $a+r_2$, 结果 r_3 进栈	r_3
9	e	操作数	进栈	$r_3\ e$
10	f	操作数	进栈	$r_3\ e\ f$
11	$^$	操作符	f 、 e 退栈, 计算 e^f , 结果 r_4 进栈	$r_3\ r_4$
12	g	操作数	进栈	$r_3\ r_4\ g$
13	$/$	操作符	g 、 r_4 退栈, 计算 r_4/g , 结果 r_5 进栈	$r_3\ r_5$
14	$-$	操作符	r_5 、 r_3 退栈, 计算 r_3-r_5 , 结果 r_6 进栈	r_6

3.4.1 表达式求值

- 利用栈将中缀表示转换为后缀表示
 - 使用栈可将表达式的中缀表示转换成它的前缀表示和后缀表示。
 - 为了实现这种转换，需要考虑各操作符的优先级。

C/C++中操作符的优先级

优先级	1	2	3	4	5	6	7
操作符	单目 -, !	*, /, %	+, -	<, <=, >, >=	==, !=	&&	

3.4.1 表达式求值

- 利用栈将中缀表示转换为后缀表示

各个算术操作符的优先级

操作符ch	#	(^	*, /, %	+, -)
isp (栈内)	0	1	7	5	3	8
icp (栈外)	0	8	6	4	2	1

- isp 叫做栈内 (in stack priority) 优先数
- icp 叫做栈外 (in coming priority) 优先数。
- 操作符优先数相等的情况只出现在括号配对或栈底的“#”号与输入流最后的“#”号配对时。
- 在把中缀表达式转换为后缀表达式的过程中，需要检查算术运算符的优先级，以实现运算规则。

3.4.1 表达式求值

■ 中缀表达式转换为后缀表达式

- 操作符栈初始化，将结束符 ‘#’ 进栈。然后读入中缀表达式字符流的首字符 **ch**。
- 重复执行以下步骤，直到 **ch = ‘#’**，同时栈顶的操作符也是 ‘#’，停止循环。
 - a) 若 **ch** 是操作数，直接输出，读入下一个字符 **ch**。
 - b) 若 **ch** 是操作符，判断 **ch** 的优先级 icp 和位于栈顶的操作符 op 的优先级 isp ：
 - 若 $icp(ch) > isp(op)$ ，令 **ch** 进栈，读入下一个字符 **ch**。（看后面是否有更高的）
 - 若 $icp(ch) < isp(op)$ ，退栈并输出。（执行先前保存在栈内的优先级高的操作符）
 - 若 $icp(ch) == isp(op)$ ，退栈但不输出，若退出的是 “(” 号读入下一个字符 **ch**。（销括号）
- 算法结束，输出序列即为所需的后缀表达式。

3.4.1 表达式求值

- 利用栈将中缀表示转换为后缀表示

- 举例，将中缀表达式

$a + b * (c - d) - e / f \#$

转换为后缀表达式

$a b c d - * + e f / -$

步	输入	栈内容	语义	输出	动作
1		#			栈初始化
2	a	#		a	操作数 a 输出, 读字符
3	+	#	+>#		操作符+进栈, 读字符
4	b	#+		b	操作数 b 输出, 读字符
5	*	#+	*>+		操作符*进栈, 读字符
6	(#+*	(>*		操作符(进栈, 读字符
7	c	#+*(c	操作数 c 输出, 读字符
8	-	#+*(->(操作符-进栈, 读字符
9	d	#+*(-		d	操作数 d 输出, 读字符
10)	#+*(-)<-	-	操作符-退栈输出
11		#+*()=((退栈, 消括号, 读字符

步	输入	栈内容	语义	输出	动作
12	-	#+*	-<*	*	操作符*退栈输出
13		#+	-<+	+	操作符+退栈输出
14		#	->#		操作符-栈, 读字符
15	e	#-		e	操作数 e 输出, 读字符
16	/	#-	/>-		操作符/进栈, 读字符
17	f	#-/		f	操作数 f 输出, 读字符
18	#	#-/	#</	/	操作符/退栈输出
19		#-	#<-	-	操作符-退栈输出
20		#	#=#		#配对, 转换结束

3.4.1 表达式求值

- 应用中缀表示计算表达式的值

$$\begin{array}{c} a + b * (c - d) - e / f \\ \underbrace{\hspace{10em}}_{\text{rst5}} \\ \underbrace{\hspace{12em}}_{\text{rst3}} \\ \underbrace{\hspace{14em}}_{\text{rst2}} \\ \underbrace{\hspace{16em}}_{\text{rst1}} \quad \underbrace{\hspace{4em}}_{\text{rst4}} \end{array}$$

- 使用两个栈，操作符栈OPTR (operator)，操作数栈OPND (operand)
- 为了实现这种计算，仍需要考虑各操作符的优先级，参看前面给出的优先级表。

3.4.1 表达式求值

■ 中缀算术表达式求值

□ 对中缀表达式求值的一般规则：

1. 建立并初始化OPTR栈和OPND栈，然后在OPTR栈中压入一个“#”
2. 扫描中缀表达式，取一字符送入ch。
3. 当ch != ‘#’ 或OPTR栈的栈顶 != ‘#’ 时，执行以下工作，否则结束算法。在OPND栈的栈顶得到运算结果。
 - ① 若ch是操作数，进OPND栈，从中缀表达式取下一字符送入ch；

3.4.1 表达式求值

■ 中缀算术表达式求值

□ 对中缀表达式求值的一般规则：

② 若ch是操作符，比较 $icp(ch)$ 的优先级和 $isp(OPTR)$ 的优先级：

- 若 $icp(ch) > isp(OPTR)$ ，则ch进OPTR栈，从中缀表达式取下一字符送入ch；
- 若 $icp(ch) < isp(OPTR)$ ，则从OPND栈退出a2和a1，从OPTR栈退出 θ ，形成运算指令 $(a1)\theta(a2)$ ，结果进OPND栈；
- 若 $icp(ch) == isp(OPTR)$ 且 $ch == '('$ ，则从OPTR栈退出 $'('$ ，对消括号，然后从中缀表达式取下一字符送入ch；

步	输入	OPND	OPTR	语义	动作
1			#		栈初始化
2	A	A	#		操作数 A 进栈, 读字符
3	+	A	#+	+ > #	操作符 + 进栈, 读字符
4	B	A B	#+		操作数 B 进栈, 读字符
5	*	A B	#+*	* > +	操作符 * 进栈, 读字符
6	(A B	#+*((> *	操作符 (进栈, 读字符
7	C	A B C	#+*(操作数 C 进栈, 读字符
8	-	A B C	#+*(-	- > (操作符 - 进栈, 读字符
9	D	A B C D	#+*(-		操作数 D 进栈, 读字符
10)	AB r ₁	#+*() < -	D、C、- 退栈, 计算 C-D, 结果 r ₁ 进栈

步	输入	OPND	OPTR	语义	动作
10)	AB r_1	#+* () < -	D、C、- 退栈, 计算 C-D, 结果 r_1 进栈
11		A B r_1	#+*) = ((退栈, 消括号, 读字符
12	-	A r_2	#+	- < *	r_1 、B、* 退栈, 计算 $B*r_1$, 结果 r_2 进栈
13		r_3	#	- < +	r_2 、A、+ 退栈, 计算 $A+r_2$, 结果 r_3 进栈
14		r_3	#-	- > #	操作符 - 进栈, 读字符
15	E	r_3 E			操作数E进栈, 读字符
16	/	r_3 E	#-/	/ > -	操作符/进栈, 读字符
17	F	r_3 E F	#-/		操作数F进栈, 读字符

步	输入	OPND	OPTR	语义	动作
17	F	r ₃ E F	# - /		操作数F进栈, 读字符
18	#	r ₃ r ₄	# -	# < /	F、E、/退栈, 计算E/F, 结果 r ₄ 进栈
19		r ₅	#	# < -	r ₄ 、r ₃ 、-退栈, 计算r ₃ -r ₄ , 结果 r ₅ 进栈
20		r ₅	#	# = #	算法结束, 结果在OPND

中缀表达式求值 = 中缀表达式转换为后缀表达式 + 后缀表达式求值

3.4.2 递归实现与系统运行栈

- 递归的定义：直接或间接调用自己的函数或过程
 - 递归步骤：将规模较大的原问题分解为一个或多个规模更小、但具有类似于原问题特性的子问题
 - 即，较大的问题递归地用较小的子问题来描述，解原问题的方法同样可用来解这些子问题
 - 递归出口：确定一个或多个无须分解、可直接求解的最小子问题（称为递归的终止条件）
- 相关概念
 - 迭代/递推
 - 归纳

3.4.2 递归实现与系统运行栈

- 阶乘 $n!$ 的递归定义如下:

$$\text{factorial}(n) = \begin{cases} 1, & \text{if } n \leq 1 \\ n \times \text{factorial}(n-1), & \text{if } n > 1 \end{cases}$$

- 递归定义由两部分组成
 - ✓ 递归基础 (递归出口)
 - ✓ 递归规则
- 正确性证明?

3.4.2 递归实现与系统运行栈

■ 递归示例：阶乘函数的程序实现

算法：阶乘的递归实现 $\text{Factorial}(n)$

输入：整数 $n \geq 0$

输出：整数 n 的阶乘

1. if $n \leq 0$ then
2. | return 1
3. else
4. | return $n \times \text{Factorial}(n-1)$
5. end

3.4.2 递归实现与系统运行栈

- 递归示例：阶乘函数的程序实现



3.4.2 递归实现与系统运行栈

● 函数调用

- ✓ **静态分配**：数据区的分配在程序**运行前**进行，整个程序运行结束才释放。函数的调用和返回处理比较简单，不需每次分配和释放被调函数的数据区
- ✓ **动态分配**：递归调用时，被调函数的局部变量等数据不能预先分配到固定单元，而须**每调用一次就分配一份**，以存放运行当前所用的数据，当返回时随即释放。故其存储分配只能在执行调用时才能进行

● 程序运行时环境

- ✓ 目标计算机上用来管理存储器并保存执行过程所需的信息的寄存器及存储器的结构

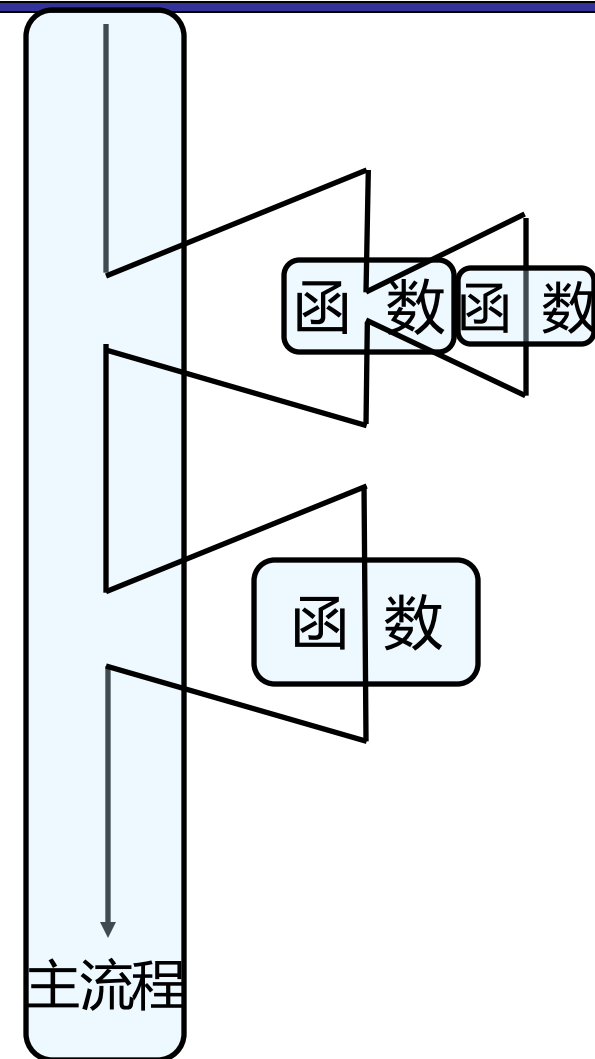
3.4.2 递归实现与系统运行栈

● 调用

- ✓ 保存调用信息（参数，返回地址）
- ✓ 分配数据区（局部变量）
- ✓ 控制转移给被调函数的入口

● 返回

- ✓ 保存返回信息
- ✓ 释放数据区
- ✓ 控制转移到上级函数（主调用函数）



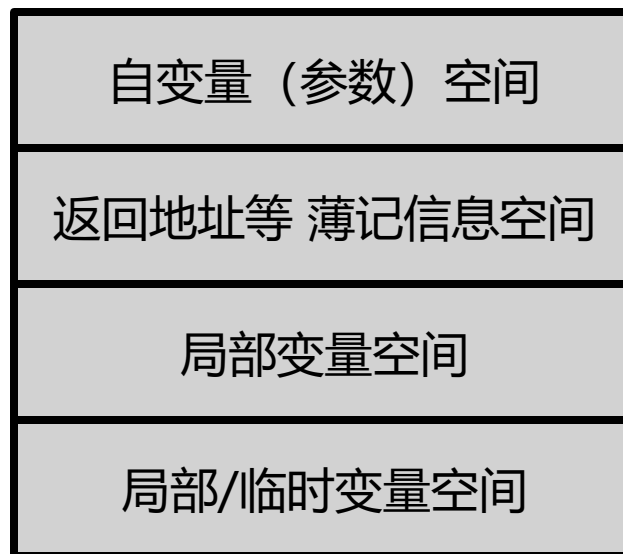
3.4.2 递归实现与系统运行栈

- 函数运行时的动态存储分配
 - **运行栈 (stack)** 用于分配**后进先出LIFO**的数据
 - ✓ e.g., 函数调用
 - **堆 (heap)** 用于不符合LIFO的数据
 - ✓ e.g., 指针所指向空间的分配



3.4.2 递归实现与系统运行栈

- 运行栈中的活动记录
 - **函数活动记录** (activation record) : 动态存储分配机制中一个重要单元
 - 当**调用或激活**一个函数时, 相应的**活动记录**包含了为其局部数据分配的存储空间



3.4.2 递归实现与系统运行栈

■ 运行栈中的活动记录

● 运行栈随程序执行时的调用链而生长/缩小

- ✓ 每调用一个函数，执行一次进栈操作，将被调函数的活动记录入栈，即每个新的活动记录都分配在栈的顶部
- ✓ 每次从函数返回时，执行一次出栈操作，释放本次活动记录，恢复到上次调用所分配的数据区
- ✓ 被调函数的变量地址全部采用相对于栈顶的相对地址来表示

● 一个函数可有多不同的活动记录，各代表 1 次调用

- ✓ 递归的深度决定递归函数在运行栈中活动记录的数目
- ✓ 递归函数中同一个局部变量，在不同的递归层次被分配不同的存储空间，放在内部栈的不同位置

3.4.2 递归实现与系统运行栈

- 递归算法的非递归实现
 - 以阶乘为例，非递归方式
 - ✓ 建立迭代
 - ✓ 递归转换为非递归

// 使用循环迭代方法，计算阶乘n!的一种程序

```
long fact (long n) {  
    int m = 1;  
    int i;  
    if (n > 1)  
        for (i = 1; i <= n; i++)  
            m = m * i;  
    return m;  
}
```

3.4.2 递归实现与系统运行栈

- 阶乘的一种非递归实现

```
long fact(long n) {           // 使用栈方法, 计算阶乘n!的程序
    Stack s;
    int m = 1;

    while (n > 1)              // ?
        s.push(n--);
    while (!isEmpty(s))        // ?
        m *= s.pop(s);
    return m;
}
```

3.4 栈与队列的应用

- 满足**先来先服务特性**的应用均可采用队列作为其数据组织方式或中间数据结构
- **调度或缓冲**
 - 消息缓冲器
 - 邮件缓冲器
 - 数据缓冲
 - 计算机的硬件设备间通信也需要队列作为数据缓冲
 - 操作系统的资源管理
- **宽度优先搜索**

3.4.3 火车车厢重排

■ 问题描述

一列挂有 n 节车厢的货运列车途径 n 个车站，计划在行车途中将各节车厢停放在不同的车站。假设 n 个车站的编号从 1 到 n ，货运列车按照从第 n 站到第 1 站的顺序经过这些车站，且将与车站编号相同的车厢卸下。

货运列车的各节车厢以随机顺序入轨，为方便列车在各个车站卸掉相应的车厢，则须重排这些车厢，使得各车厢从前往后依次编号为 1 到 n ，这样在每个车站只需卸掉当前最后一节车厢即可。

车厢重排可通过转轨站完成。一个转轨站包含 1 个入轨 (I)， 1 个出轨 (O) 和 k 个位于入轨和出轨之间缓冲轨 (H_i)。请设计合适的算法来实现火车车厢的重排。

3.4.3 火车车厢重排

- 本质：将一个无序序列转换成一个以队列方式组织的有序序列
 - 转换过程采用缓冲轨存储尚未确定输出次序的车厢，满足递增或增减的特性即可，以栈或队列组织均可
- 若将每个缓冲轨看成一个队列
 - 转化为：将一个长度为 n 的随机序列（车厢进入入轨），通过 k 个缓冲队列，输出到一个队列（出轨）
 - 重排规则：
 1. 一个车厢从入轨 移至 出轨 或 缓冲轨；
 2. 一个车厢 只有在**其编号恰是下一个待输出编号**时，可 移到出轨；
 3. 一个车厢移到某个缓冲轨，仅当其编号大于该缓冲轨中队尾车厢的编号，若多个缓冲轨满足这一条件，则选择队尾车厢编号最大的，否则选择一个空缓冲轨，若无空缓冲轨则无法重排

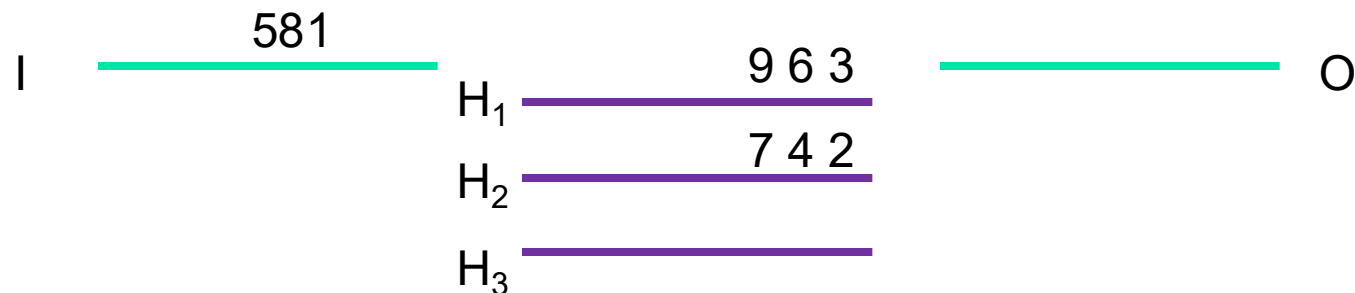
3.4.3 火车车厢重排

示例： 9节车厢的货车，使用3个容量为3的缓冲轨

初始状态： 9节车厢均在入轨排队

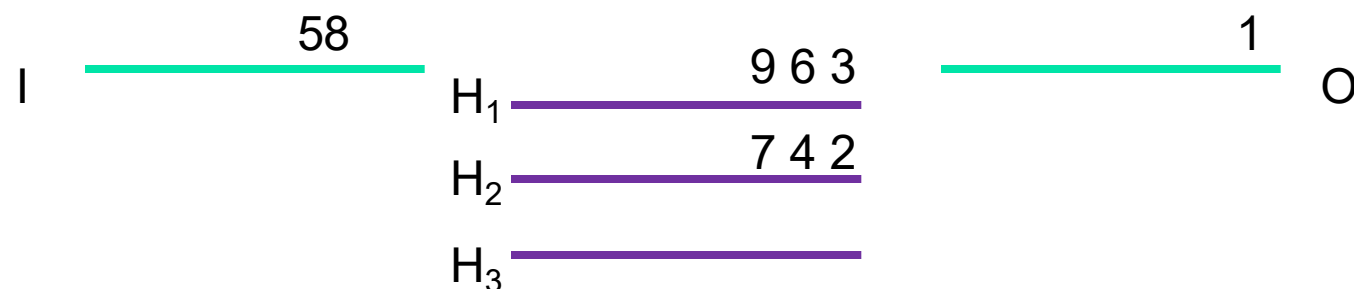


入轨的车厢按序分别进入缓冲轨 H_1 、 H_2 、 H_3

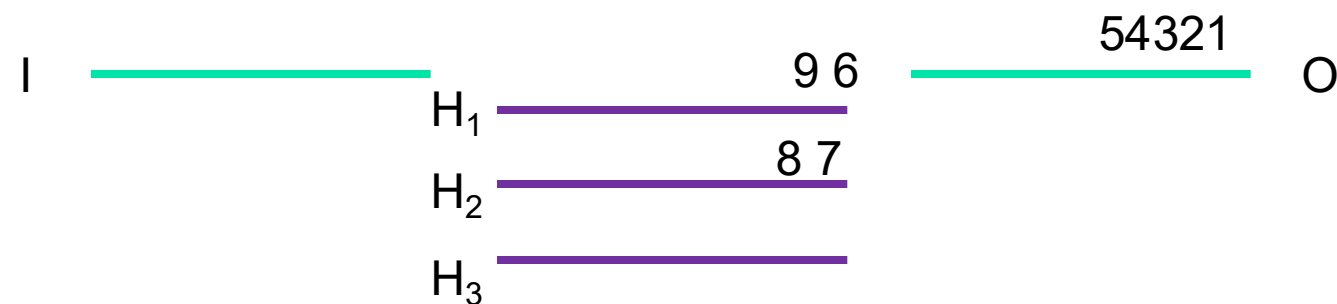


3.4.3 火车车厢重排

车厢1从 H_3 出队进入出轨



车厢2、3、4按序分别从 H_2 、 H_1 、 H_2 出队进入出轨；
车厢8进入缓冲轨 H_2 ，车厢5直接进入出轨



3.4.3 火车车厢重排

各缓冲轨中车厢按编号进入出轨



3.4.3 火车车厢重排

算法: 车厢重排 $\text{TrainCarriageScheduling}(in_track, out_track, n, k)$

输入: 入轨的车厢序列 in_track ; 车厢数量 $n \geq 0$; 缓冲轨数量 $k > 0$

输出: 按序重排车厢的出轨序列 out_track ; 若任务不可能完成则退出

```
1. for  $i \leftarrow 1$  to  $k$  do
2. | InitQueue(buffer[i])
3. end
4. InitQueue(out_track)
5. next_out  $\leftarrow 1$ 
6. for  $i \leftarrow 1$  to  $n$  do
7. | if  $in\_track[i] = next\_out$  then
8. | | EnQueue(out_track, i)
9. | | next_out  $\leftarrow next\_out + 1$ 
10. | else
11. | | for  $j \leftarrow 1$  to  $k$  do // 考察每一缓冲轨队列
12. | | | front_crg  $\leftarrow$  GetFront(buffer[j]) // 查看队列j的头元素
13. | | | if front_crg  $\neq$  NIL 且  $in\_track[front\_crg] = next\_out$  then
14. | | | | EnQueue(out_track, front_crg)
15. | | | | DeQueue(buffer[j])
16. | | | | next_out  $\leftarrow next\_out + 1$ 
17. | | | | break
18. | | | end
19. | | end
```

3.4.3 火车车厢重排

```
20. | | if  $j > k$  then // 若入轨和缓冲轨的队首元素中没有编号为 $next\_out$ 的车厢
21. | | |  $max\_rear \leftarrow 0$ 
22. | | |  $max\_buffer \leftarrow -1$ 
23. | | | for  $j \leftarrow 1$  to  $k$  do // 考察每一缓冲轨队列的队尾
24. | | | |  $rear\_crg \leftarrow \text{GetRear}(buffer[j])$ 
25. | | | | if  $rear\_crg \neq \text{NIL}$  且  $in\_track[i] > in\_track[rear\_crg]$  then
26. | | | | | if  $in\_track[rear\_crg] > max\_rear$  then
27. | | | | | |  $max\_rear \leftarrow in\_track[rear\_crg]$  // 最大队尾元素值
28. | | | | | |  $max\_buffer \leftarrow j$  // 最大队尾元素所在的队列编号
29. | | | | | end
30. | | | | end
31. | | | end
32. | | | if  $max\_buffer \neq -1$  then
33. | | | |  $\text{EnQueue}(buffer[max\_buffer], i)$ 
34. | | | else
35. | | | | for  $j \leftarrow 1$  to  $k$  do
36. | | | | | if  $\text{IsEmpty}(buffer[j]) = \text{true}$  then
37. | | | | | | break
38. | | | | | end
39. | | | | end
```

3.4.3 火车车厢重排

```
40. | | | | if  $j \leq k$  then
41. | | | | | EnQueue(buffer[j], i)
42. | | | | else
43. | | | | | 任务不可能完成, 退出
44. | | | | end
45. | | | end
46. | | end
47. | end
48. end
49. while next_out ≤ n do
50. | for  $j \leftarrow 1$  to k do // 考察每一缓冲轨队列
51. | | if front_crg ≠ NIL 且 in_track[front_crg] = next_out then
52. | | | EnQueue(out_track, front_crg)
53. | | | DeQueue(buffer[j])
54. | | | next_out ← next_out + 1
55. | | | break
56. | | end
57. | end
58. end
59. for  $i \leftarrow 1$  to k do
60. | DestroyQueue(buffer[i])
61. end
```

- 3.1 问题引入：超市货架管理
- 3.2 栈的定义与结构
- 3.3 队列的定义与结构
- 3.4 栈与队列的应用
- 3.5 拓展延伸
- 3.6 应用场景：消息队列

3.5.1 单调栈

- 栈中元素具有**单调性**的栈
 - ✓ 单调递增栈：元素从栈顶到栈底若从栈顶到栈底的元素单调递增
 - ✓ 单调递减栈：若栈中元素单调递减
- 适用于涉及到序列中元素大小的问题的求解

一组数 8, 2, 7, 3, 10

栈为空, 8 入栈

8

$2 < 8$, 直接 入栈

8 2

$7 > 2$, 弹出2, 此时 $7 < 8$, 入栈

8 7

$3 < 7$, 直接入栈

8 7 3

$10 > 3$ 及栈中所有元素, 依此出栈后, 10入栈

10

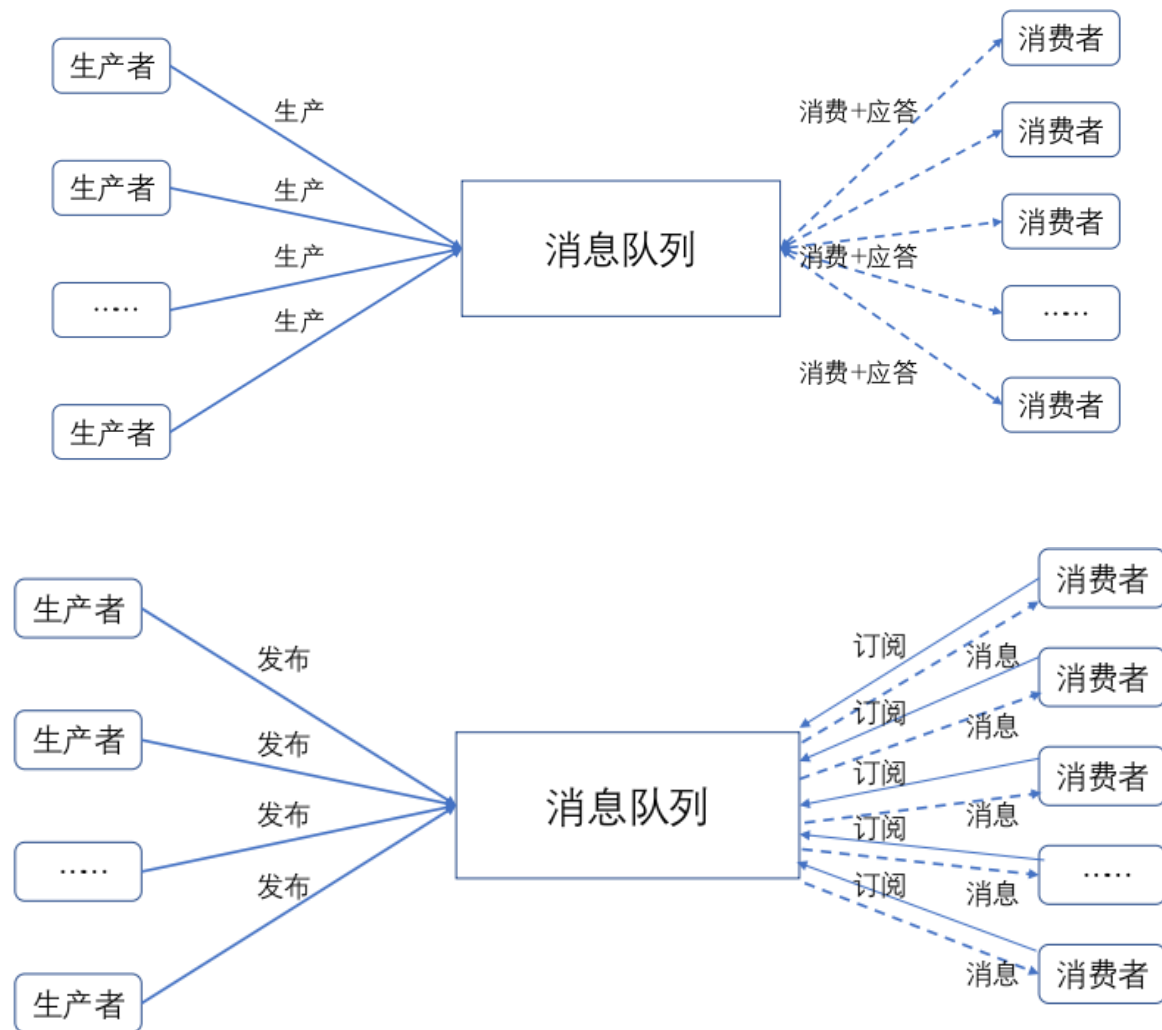
3.5.2 单调队列

- 一种元素具有严格单调性的队列，满足两个约束：
 - 队列元素从头到尾的严格单调性。
 - 队列元素先进先出，当然队首元素比尾部元素要先进。
- 根据元素的递增或递减可组织为
 - 单调递增队列
 - 单调递减队列
- 单调队列的主要操作基本与队列相同，**区别**在于单调队列的元素入队时，需要**单调性检查**，通过删除队列中不满足单调性的元素来保持队列的单调性
- 单调队列的元素间兼有先进先出的公平性和单调性，可用于解决滑动窗口类问题

- 3.1 问题引入：超市货架管理
- 3.2 栈的定义与结构
- 3.3 队列的定义与结构
- 3.4 栈与队列的应用
- 3.5 拓展延伸
- 3.6 应用场景：消息队列

3.6 消息队列

- 进程或线程之间通信的一种常用方式
- 为进程/线程提供一个临时存储消息的轻量级缓冲区，通常采用先进先出的存储方式，包括
 - 请求、恢复、错误消息、明文信息或控制权等
- 一个消息包含
 - 消息头：用于存储消息类型、目的地id、源id、消息长度和控制信息等信息
 - 消息体



小结

- 栈是限定仅在表尾（栈顶）进行插入或删除的线性表
- 栈与线性表相同，数据元素间具有1对1的关系
- 具有后进先出、先进后出的特点
- 栈有两种存储表示，分别是顺序栈（顺序结构）和链栈（链式结构）
- 栈最主要的操作是进站和出栈，在进行进站和出栈时注意判满和判空
- 在顺序栈中，共享栈可以互相调剂两个栈的空间，灵活性强；只有在整个存储空间被占满时，才会发生上溢现象
- 链栈使用指针连接栈结点，可以同时实现多个栈共享存储空间，且不存在栈满上溢的问题。

小结

- 队列是限定仅在一端进行插入（队尾），在另一端进行删除（队头）的线性表。
- 队列具有**先进先出**的特点
- **循环队列**可以解决标准队列**假溢出**问题，它通过对MaxSize求模实现队头队尾指针的连接
- 队列和栈都是都是**受限制**的线性表，数据元素间具有**1对1**的关系
- 顺序存储的栈和队列都需要**预先分配存储空间**，可能会出现**空间闲置或栈满溢出**的现象，且数据元素无法自由扩充
- 链式存储的栈和队列采用**动态分配内存空间**，**不会出现空间闲置或栈满溢出**的现象，且数据元素可自由扩充



栈与队列高频必刷题 (LeetCode)

分类	题号	题目	考察点	难度
栈 - 基础应用	20	Valid Parentheses	括号匹配, 栈的基本应用	Easy
	32	Longest Valid Parentheses	栈/DP求最长合法括号子串	Hard
栈 - 单调栈	739	Daily Temperatures	单调递减栈, 找下一个更大元素	Medium
	496	Next Greater Element I	单调栈模板	Easy
	84	Largest Rectangle in Histogram	直方图最大矩形面积, 经典难题	Hard
栈 - 结构互实现	225	Implement Stack using Queues	用队列实现栈	Easy
	232	Implement Queue using Stacks	用栈实现队列	Easy
栈 - 表达式计算	150	Evaluate Reverse Polish Notation	逆波兰表达式求值	Medium
队列 - BFS	102	Binary Tree Level Order Traversal	层序遍历	Medium
	200	Number of Islands	BFS/DFS 图搜索	Medium
	994	Rotting Oranges	BFS多源扩散	Medium
	207	Course Schedule	拓扑排序, 队列实现	Medium
队列 - 单调队列	239	Sliding Window Maximum	单调队列维护滑动窗口最大值	Hard
栈/队列混合	155	Min Stack	O(1) 获取最小值的栈	Medium



线性表高频必刷题 (LeetCode)

分类	题号	题目	考察点	难度
数组基础	1	Two Sum	哈希表 + 数组遍历	Easy
	26	Remove Duplicates from Sorted Array	双指针去重	Easy
	27	Remove Element	双指针删除元素	Easy
	88	Merge Sorted Array	双指针合并两个有序数组	Easy
	283	Move Zeroes	双指针移动元素	Easy
链表基础	206	Reverse Linked List	单链表反转	Easy
	141	Linked List Cycle	快慢指针判环	Easy
	142	Linked List Cycle II	找链表入环点 (快慢指针+数学)	Medium
	21	Merge Two Sorted Lists	合并两个有序链表	Easy
	19	Remove Nth Node From End of List	双指针删除节点	Medium
	160	Intersection of Two Linked Lists	双指针找链表相交点	Easy
	207	Course Schedule	拓扑排序, 队列实现	Medium