



计算机领域本科教育教学改革试点
工作计划（“101计划”）研究成果

数据结构

授课教师：屈卫兰

湖南大学 信息科学与工程学院

第 3 章

栈与队列

提纲

- 3.1 问题引入
- 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



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：
- 问题

如何对商品信息（数据）进行合理的组织（商品分类）、存储（商品陈列）、以及提供必须的操作（商品补架、下架及查找商品）？



如何兼顾商品的特点与商品的补货和选购呢



超市货架

补货是将特定商品 **插入** 到货架的某个位置
顾客选购则是从货架的某个位置 **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：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：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



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



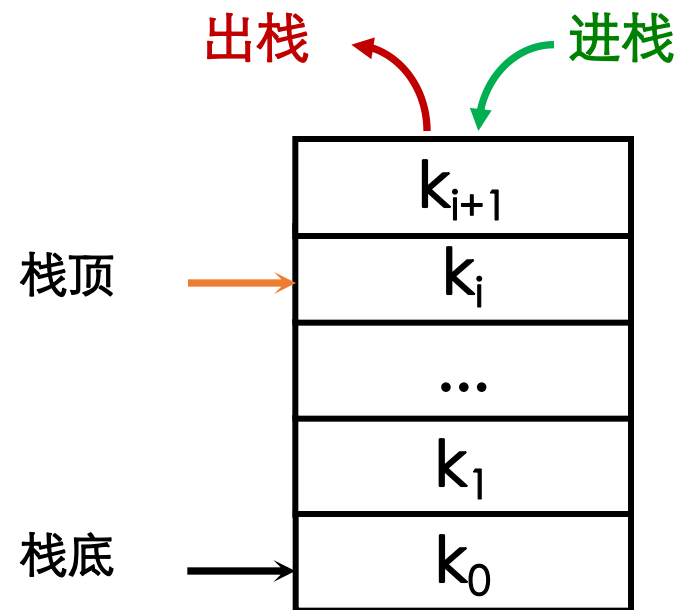
栈

- 后进先出**LIFO** (**L**ast **I**n **F**irst **O**ut)
 - 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”
- 栈的主要元素
 - **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：另一端



栈的图示

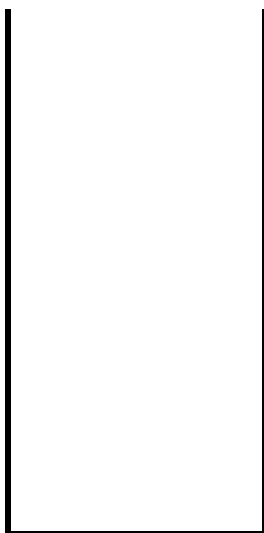
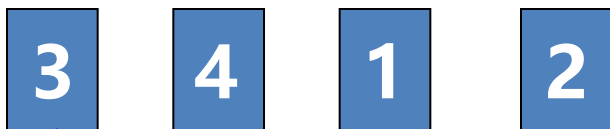
- 每次取出（被删除）的总是刚压进的元素，而最先压入的元素则位于栈的底部
- 空栈：没有元素的栈
- 应用
 - 表达式求值
 - 函数调用
 - 深度优先搜索





示例：火车进出栈问题

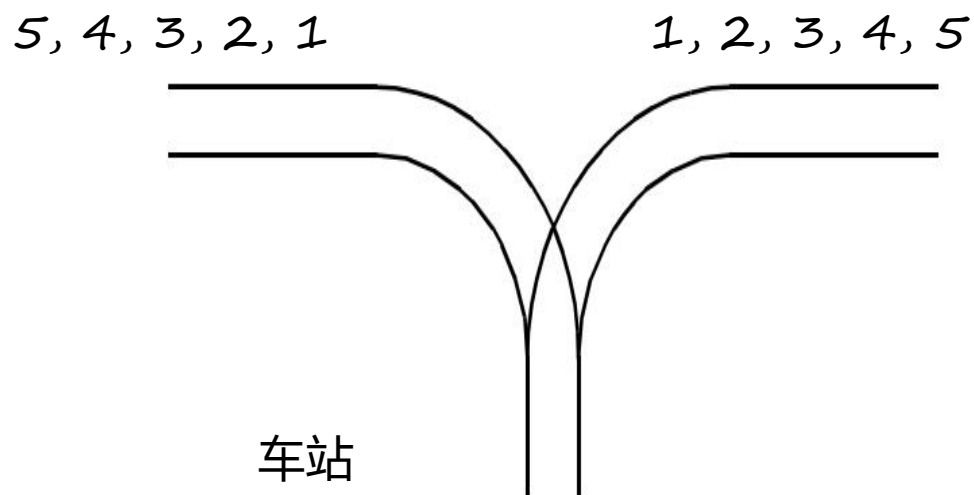
- 利用 合法的重构 发现 冲突





示例：火车进出栈问题

- 判断火车的出栈顺序是否合法
- 编号为 $1, 2, \dots, n$ 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？





思考



• 若入栈顺序为1,2,3,4的话, 则出栈的顺序可以有哪些?

- 1234
- 1243
- 1324
- 1342
- 1423
- 1432
- 2134
- 2143
-



思考



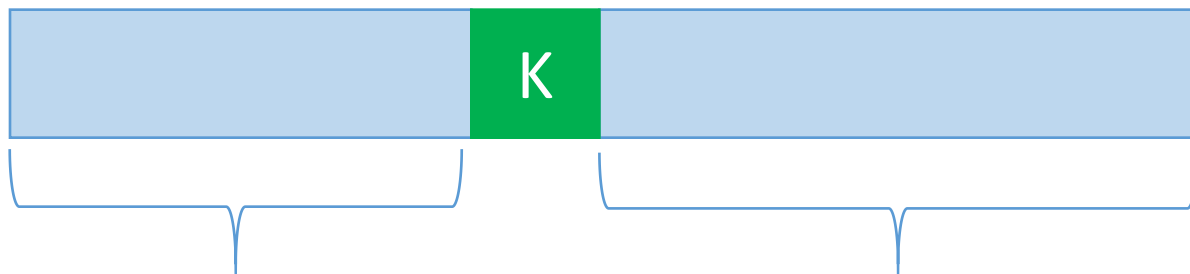
- 给定一个入栈序列，和一个出栈序列，请你写出一个程序，判定出栈序列是否合法？
- 给定一个入栈序列，序列长度为 N ，请计算有多少种出栈序列？



思考



- 给定一个入栈序列，序列长度为N，请问有多少种出栈序列？



K个元素

N-K-1 个元素

$$f(N) = \sum_{k=0}^{n-1} f(k) \times f(N-1-k) = \frac{1}{N+1} \times C_{2N}^N$$



栈的抽象数据类型

代码3-1：栈的抽象数据类型定义

ADT Stack {

数据对象：

$\{ ai \mid ai \in \text{ElemSet}, i=1,2,\dots,n, n > 0 \}$ 或 \emptyset ，即空表；ElemSet为元素集合。

数据关系：

$\{ \langle ai, ai+1 \rangle \mid ai, ai+1 \in \text{ElemSet}, i=0,1,\dots,n-2 \}$ 。

基本操作：

InitStack (*stack*):

初始化一个空的栈*stack*。

DestroyStack (*stack*):

释放栈*stack*占用的所有空间。

Clear (*stack*):

清空栈*stack*。

IsEmpty (*stack*):

栈*stack*为空返回真，否则返回假。

IsFull (*stack*):

栈*stack*满返回真，否则返回假。

Top (*stack*):

返回栈*stack*的栈顶结点，栈顶结点不变。

Push (*stack*, *x*):

将结点*x*压入栈*stack*，使其成为新的栈顶。

Pop (*stack*):

将栈顶结点弹出栈*stack*。

}



栈的实现方式

- 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



顺序栈

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

输入：栈*stack* 和正整数kSize

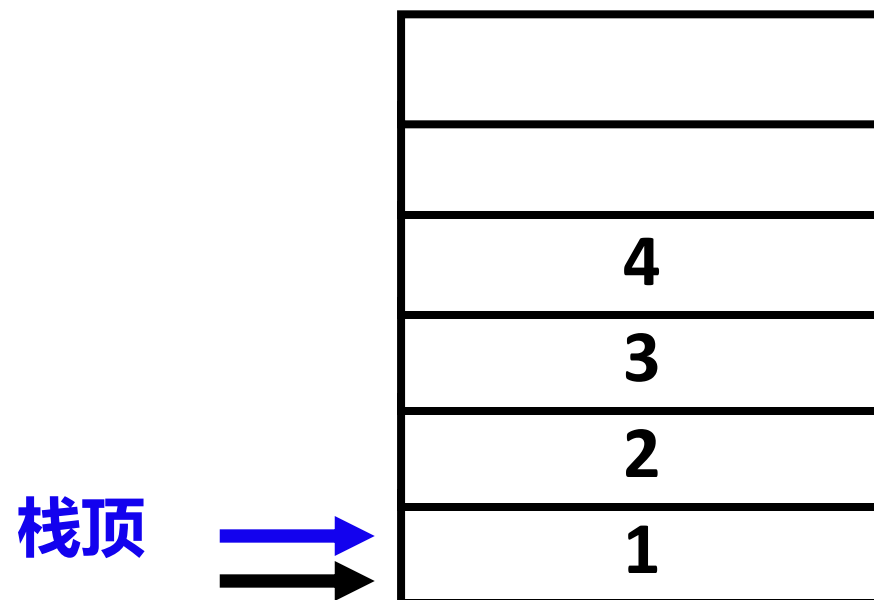
输出：一个大小为kSize的顺序栈

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



顺序栈示例

- 按1, 2, 3, 4的次序入栈，最后压入的元素编号为4





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



压栈操作

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



出栈操作

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**



读栈操作

算法：顺序栈的出栈操作 $\text{Pop}(stack)$

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



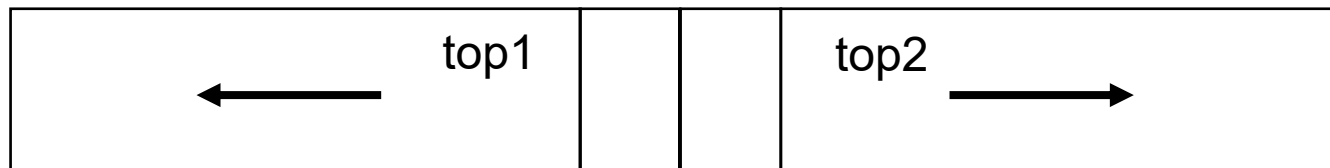
栈的变种

- 两个独立的栈

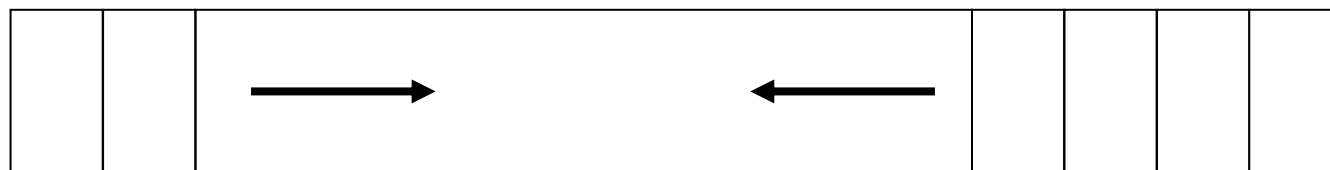
- ✓ 底部相连：双栈

- ✓ 迎面增长

- ✓ 各自适用的场景？



底部相连的栈

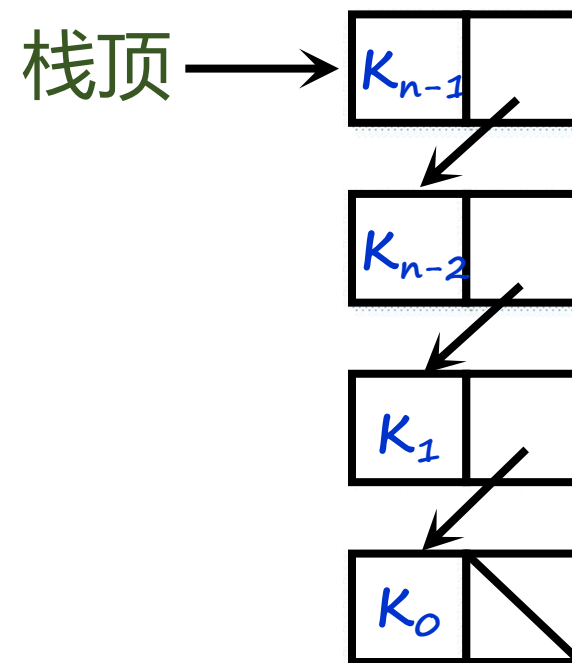


迎面增长的栈



链式栈

- 采用单链表
- 关键：栈顶
 - ✓ 指针的方向从栈顶向下链接





链式栈的创建

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. $stack.top \leftarrow \text{NIL}$



压栈操作

算法：链式栈的取顶操作 $\text{Top}(\text{stack})$

输入：栈 stack

输出：栈顶元素；若栈空，则输出NIL

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



出栈操作

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**



顺序栈 vs 链式栈

- 时间效率
 - ✓ 入栈/出栈操作均只需 常数时间
 - ✓ 时间效率上难分伯仲
- 空间效率
 - ✓ 顺序栈结构紧凑，但须事先确定长度
 - ✓ 链式栈长度可变，增加结构性开销



顺序栈 vs 链式栈

- 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



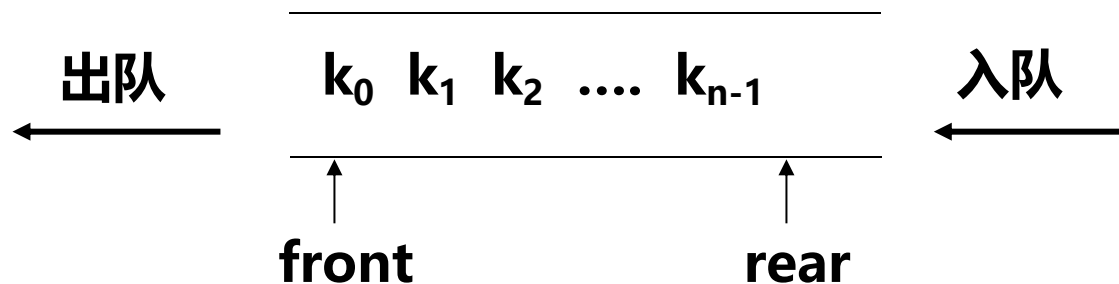
队列

- 先进先出**FIFO** (**F**irst **I**n **F**irst **O**ut)

- 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





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)



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。

}



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



顺序队列

- 使用顺序表来实现队列
 - 用数组存储队列元素，用两个变量分别指向队列的队头（前端）和队尾（尾端）
 - **Q.front**
 - **Q.rear**

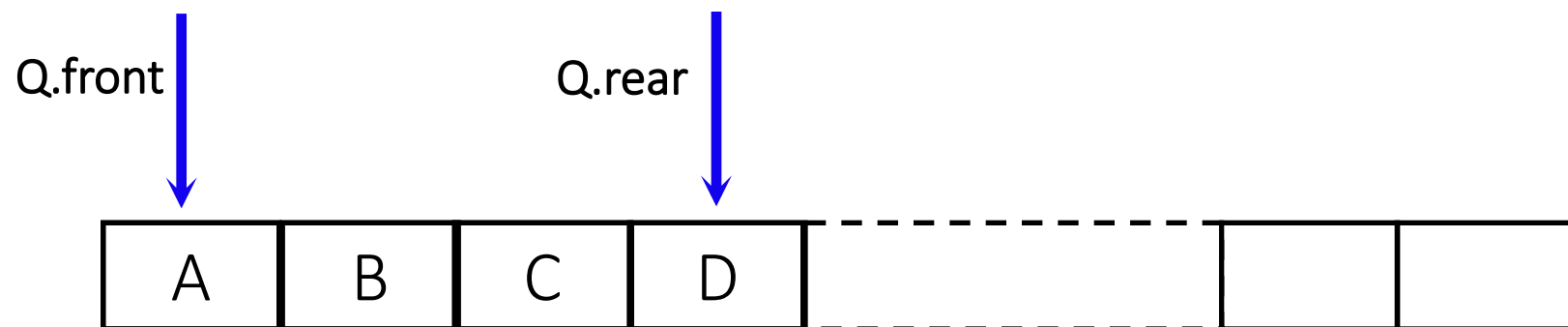
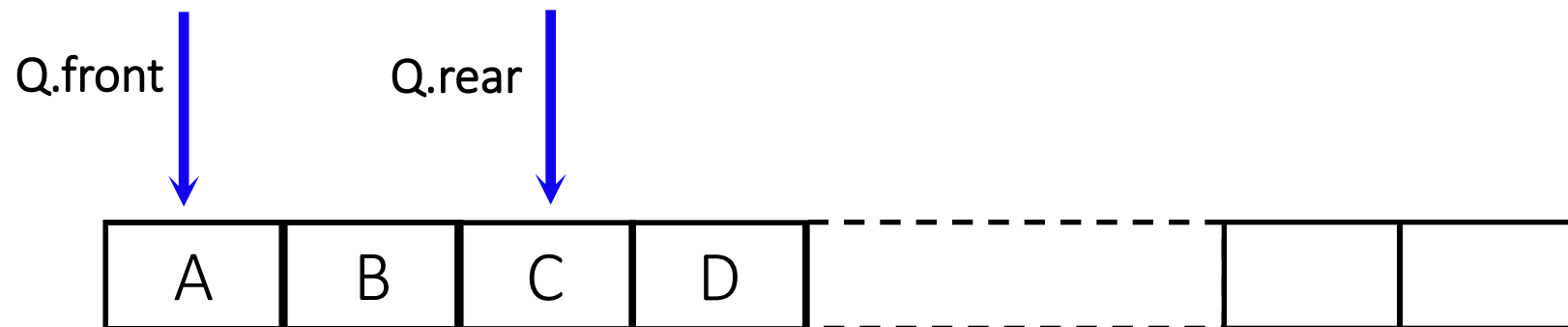


顺序队列的维护

- 尾指针Q.rear 的处理

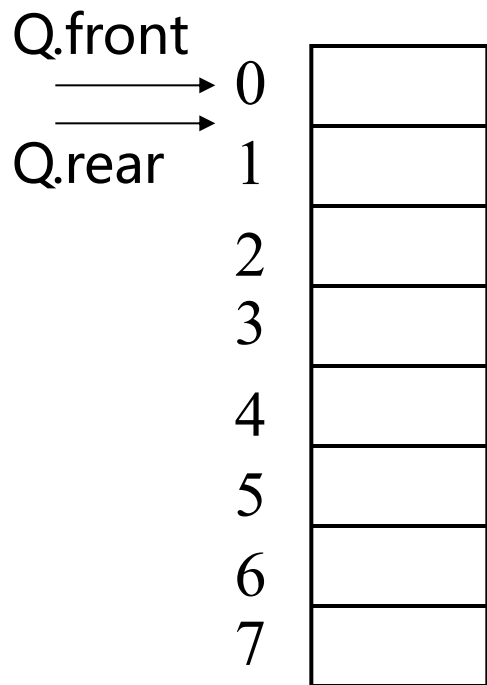
- ✓ 实指

- ✓ 虚指

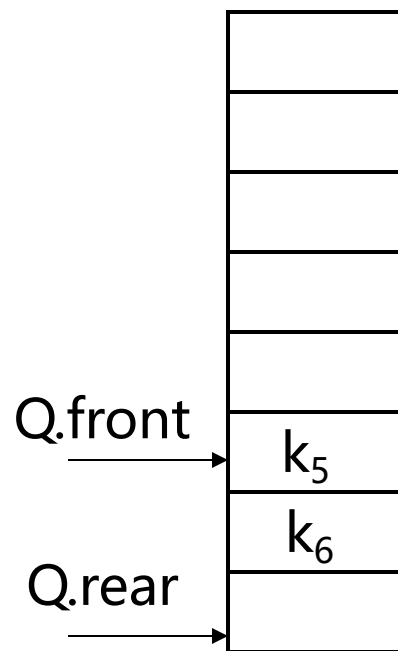
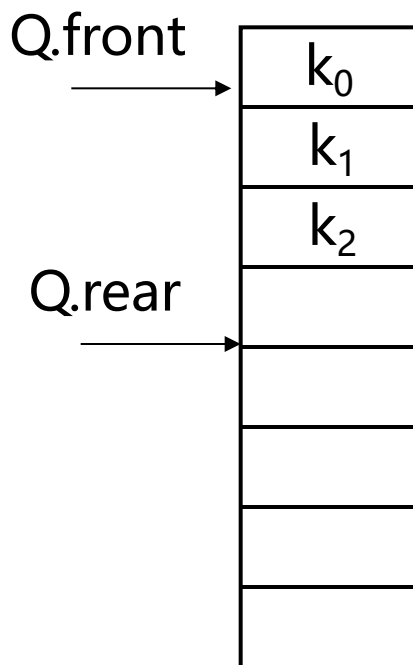




队列示意：普通



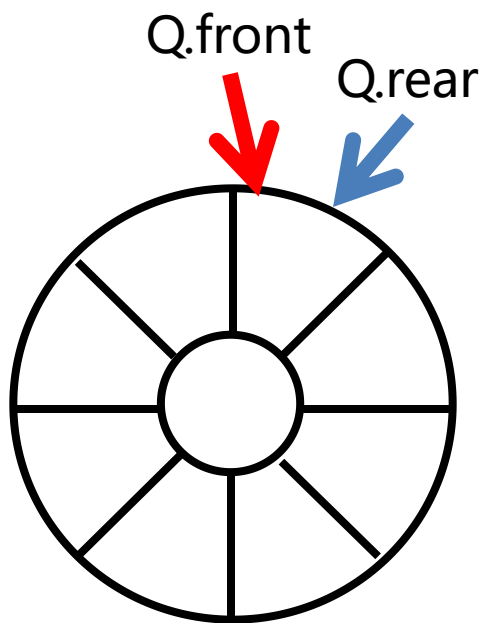
队列空



再进队一个元素如何?

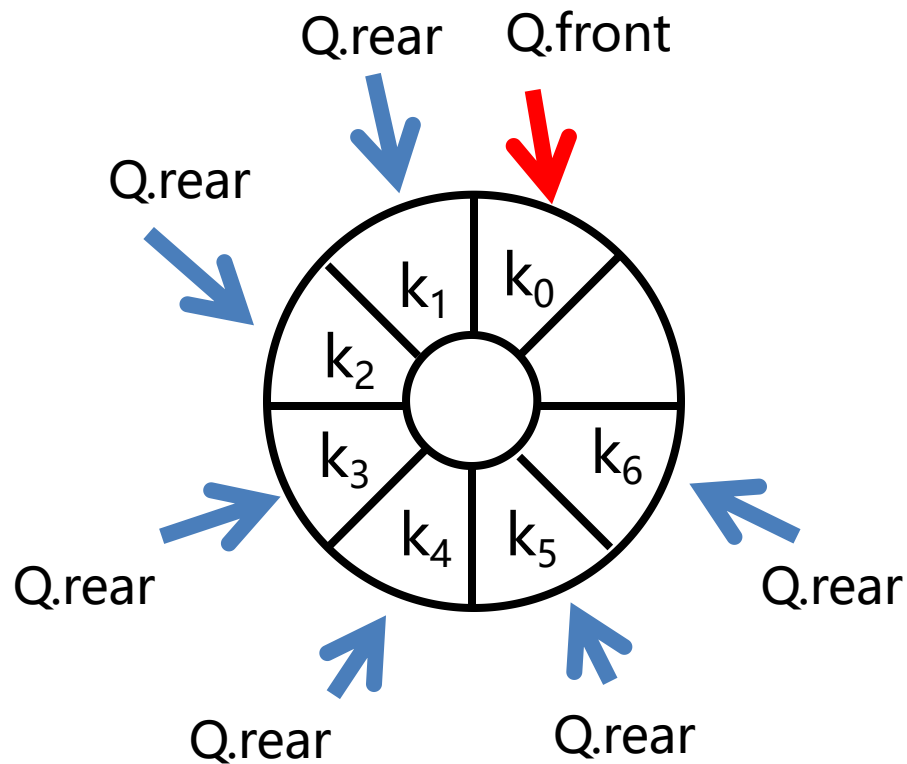


队列示意：环形



队列空：

$Q.rear == Q.front$



队列满：

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



思考

1. 只是用 front, rear 两个变量, 长度为 $mSize = n$ 的队列, 可以容纳的**最大**元素个数为多少? 请给出详细的推导过程。
2. 若不想浪费队列的存储单元, 还可以采用什么方法?
3. 采用实指和虚指方法实现队尾指针(rear指向队尾元素的下一个元素, 和实指相比后移一位), 在具体实现上有何异同? 哪一种更好?



顺序队列的类定义

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



顺序队列的实现

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

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



顺序队列的实现

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
-

算法：顺序队列的出队操作 $\text{DeQueue}(\text{queue})$

输入：队列 queue

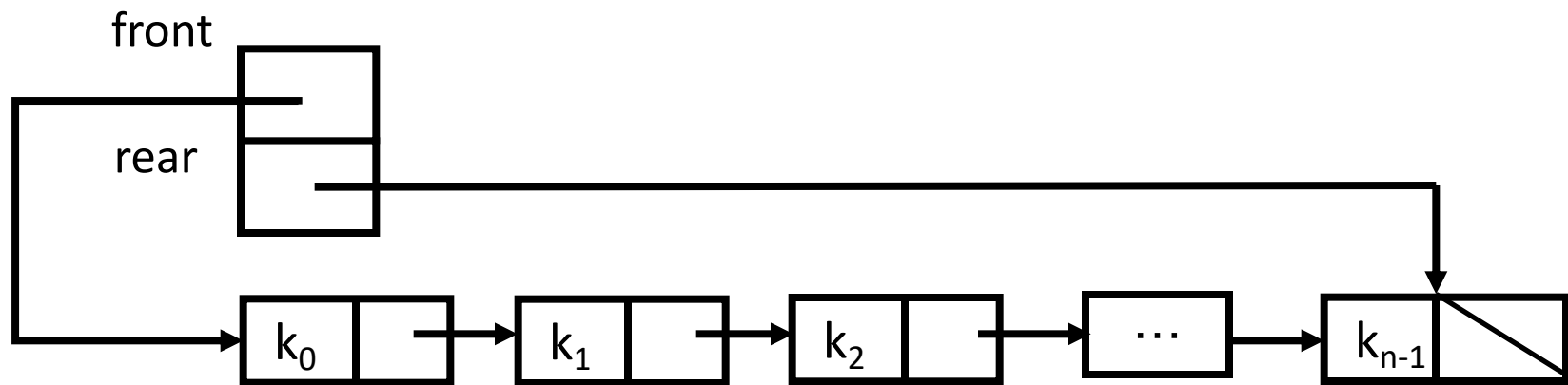
输出：删除队列头元素后的顺序队列；若队列空，则退出

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



链式队列

- 单链表队列
- 链接指针的方向是从队头到队尾





链式队列的类定义

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



链式队列的入队

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. $\text{queue.size} \leftarrow \text{queue.size} + 1$



链式队列的出队

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



链式队列的出队

算法3-12：链式队列的出队操作 $\text{DeQueue}(\text{queue})$

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



顺序队列 vs 链式队列

- **顺序队列**
 - 固定的存储空间
- **链式队列**
 - 可以满足浪涌大小无法估计的情况
- 不允许访问队列内部元素



变种的栈和队列结构

- **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



思考



- 是否可以用**栈**来**模拟队列**？若可以的话，需要几个栈，如何来模拟？
- 试利用非数组变量，按下述条件各设计一个相应的算法以使队列中的元素有序：
 - a) 使用两个辅助的队列；
 - b) 使用一个辅助的队列。



栈的应用

- 栈的特点： 后进先出
 - 体现了元素间的透明性
- 常用来处理具有递归结构的数据
 - 深度优先搜索
 - 数制转换
 - 表达式求值
 - 行编辑处理
 - 子程序 / 函数调用的管理
 - 消除递归



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



数制转换

- 非负十进制数转换成其它进制的一种方法

- ✓ 例, 十进制转换成八进制: $(66)_{10} = (102)_8$

- $66/8 = 8$ 余 2
 - $8/8 = 1$ 余 0
 - $1/8 = 0$ 余 1

结果为余数的**逆序**: 102

- 特点: 最先求得的**余数**在输出结果中最后写出, 最后求出的**余数**最先写出, 符合**栈的先入后出**性质, 故可用**栈**来实现数制转换

- ✓ 同理, 一个非负十进制整数 N 转换为另一个等价的 B 进制数, 可以采用“**除B取余法**”来解决



表达式计算

- 表达式定义

- ✓ 基本**符号集**: $\{0, 1, \dots, 9, +, -, *, /, (,)\}$
- ✓ 语法**成分集**: $\{<\text{表达式}>, <\text{项}>, <\text{因子}>, <\text{常数}>, <\text{数字}>\}$
- ✓ 语法**公式集**

- 中缀表达式

$23 + (34 * 45) / (5 + 6 + 7)$

- 后缀表达式

$23\ 34\ 45\ * \ 5\ 6\ + \ 7\ + \ / \ +$



中缀表达式的语法公式

$\langle \text{表达式} \rangle ::= \langle \text{项} \rangle + \langle \text{项} \rangle$
 $\quad \quad \quad | \langle \text{项} \rangle - \langle \text{项} \rangle$
 $\quad \quad \quad | \langle \text{项} \rangle$

$\langle \text{项} \rangle ::= \langle \text{因子} \rangle * \langle \text{因子} \rangle$
 $\quad \quad \quad | \langle \text{因子} \rangle / \langle \text{因子} \rangle$
 $\quad \quad \quad | \langle \text{因子} \rangle$

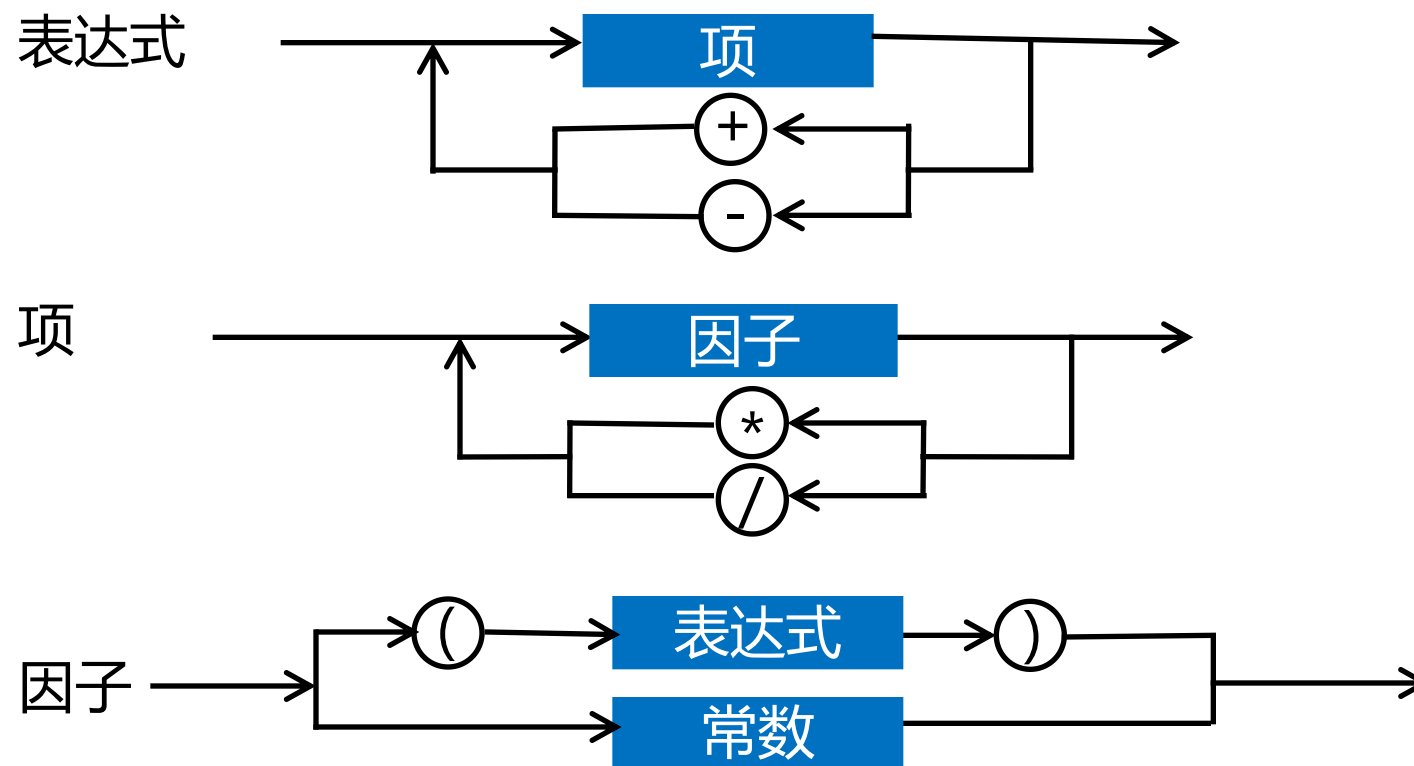
$\langle \text{因子} \rangle ::= \langle \text{常数} \rangle$
 $\quad \quad \quad | (\langle \text{表达式} \rangle)$

$\langle \text{常数} \rangle ::= \langle \text{数字} \rangle$
 $\quad \quad \quad | \langle \text{数字} \rangle \langle \text{常数} \rangle$

$\langle \text{数字} \rangle ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$



中缀表达式的递归图示



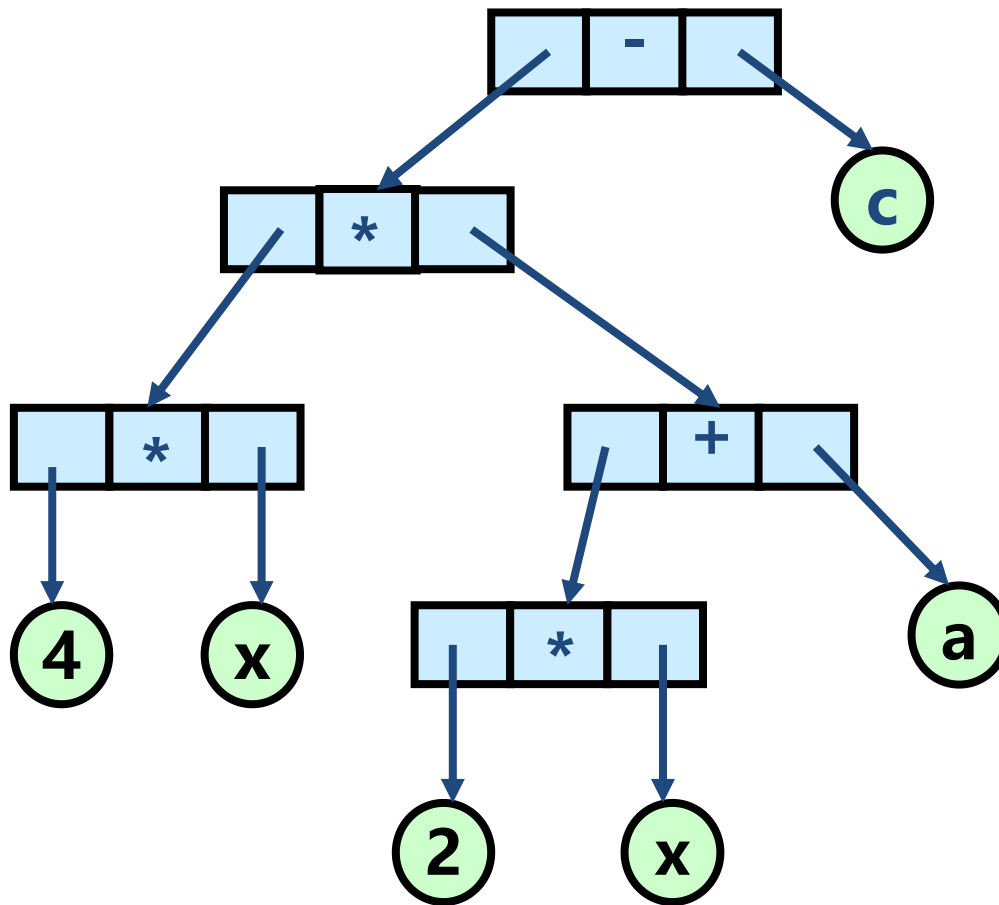


中缀表达式

- 特点

- ✓ 运算符在中间
- ✓ 运算优先级可由括号改变

$4 * x * (2 * x + a) - c$





中缀表达式的计算

1. 若有**括号**，先执行**括号内**的计算，后执行**括号外**的计算。具有多层括号时，按层次由内而外反复地脱括号，**左右括号必须配对**
2. **无括号**或同层括号时，先乘(*)、除(/)，后加(+)、减(-)
3. **同一个层次**，若有多个乘除(*、/)或加减(+、-)的运算，按**自左至右次序**执行

$$23 + (34 * 45) / (5 + 6 + 7) = ?$$

计算特点？



后缀表达式的语法公式

$\langle \text{表达式} \rangle ::= \langle \text{项} \rangle \langle \text{项} \rangle +$
 $\quad \quad \quad | \langle \text{项} \rangle \langle \text{项} \rangle -$
 $\quad \quad \quad | \langle \text{项} \rangle$

$\langle \text{项} \rangle ::= \langle \text{因子} \rangle \langle \text{因子} \rangle *$
 $\quad \quad \quad | \langle \text{因子} \rangle \langle \text{因子} \rangle /$
 $\quad \quad \quad | \langle \text{因子} \rangle$

$\langle \text{因子} \rangle ::= \langle \text{常数} \rangle$

$\langle \text{常数} \rangle ::= \langle \text{数字} \rangle$
 $\quad \quad \quad | \langle \text{数字} \rangle \langle \text{常数} \rangle$

$\langle \text{数字} \rangle ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$

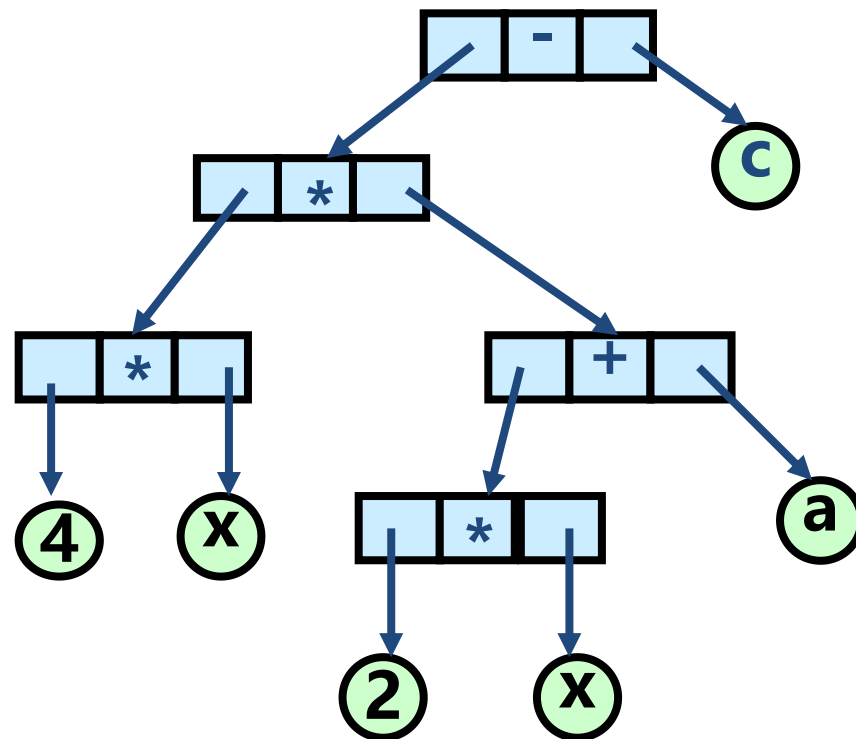


后缀表达式

- 特点

- 运算符 在 操作数 后面
- 完全不需要 括号

4 x * 2 x * a + * c -





后缀表达式求值

1. 循环：**依序读入**表达式的符号序列（以 **=** 作为输入序列的结束），根据读入元素符号逐一分析处理：
 - **操作数**，将其压入栈中；
 - **运算符**，从栈中两次取出栈顶，按照运算符对这两个操作数进行相应计算，并将计算结果压回栈；
2. 如此继续，直到遇到符号 **=**，此时栈顶元素即为输入表达式的值

23 34 45 * 5 6 + 7 + / + =?

计算特点？

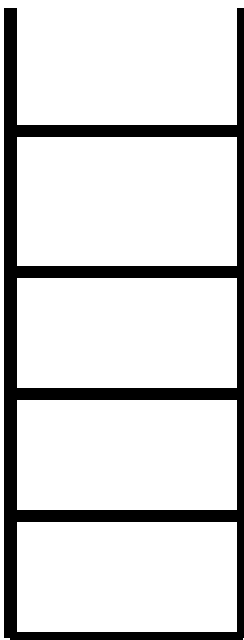


后缀表达式求值

待处理的后缀表达式

23 34 45 * 5 6 + 7 + / +

栈状态的变化



计算

结果



后缀表达式求值算法

算法3-13: 后缀表达式求值 $\text{PostFixEval}(expr)$

输入: 一个后缀表达式 $expr$

输出: 后缀表达式 $expr$ 的值。若表达式不规范, 则输出错误码 (ErrorCode)

```
1.  $token \leftarrow \text{GetToken}(expr)$  // 从表达式中取出一个元素
2. while  $token \neq$  表达式结尾 do
3. | if  $\text{IsOperand}(token)$  then // 如果该元素是操作数
4. | |  $\text{Push}(stack, token)$  // 则入栈
5. | else // 如果该元素是操作符
6. | |  $operand2 \leftarrow \text{Top}(stack)$ 
7. | |  $\text{Pop}(stack)$ 
8. | |  $operand1 \leftarrow \text{Top}(stack)$ 
9. | |  $\text{Pop}(stack)$ 
10. | |  $result \leftarrow \text{Calculate}(operand1, token, operand2)$  // 计算  $operand1$   $token$   $operand2$  的值
11. | |  $\text{Push}(stack, result)$  // 计算结果入栈
12. | end
13. |  $token \leftarrow \text{GetToken}(expr)$  // 从表达式中取下一个元素
14. end
15.  $result \leftarrow \text{Top}(stack)$ 
16.  $\text{Pop}(stack)$ 
17. if  $\text{IsEmpty}(stack) = \text{false}$  then // 表达式不规范
18. |  $result \leftarrow \text{ErrorCode}$ 
19. end
20.  $\text{DestroyStack}(stack)$ 
21. return  $result$ 
```



中缀表达式 vs 后缀表达式

- 中缀和后缀表达式的主要异同？

$$23 + (34 * 45) / (5 + 6 + 7) = ?$$

$$23 \ 34 \ 45 * \ 5 \ 6 + 7 + / + = ?$$

用 同 化 异 ？



中缀表达式 to 后缀表达式

- **从左到右扫描中缀表达式**：用 **栈** 存放 表达式中的**操作符**、**开括号** 以及**开括号**后 **暂不确定计算次序** 其他符号
 - (1) **操作数**，直接输出到后缀表达式序列；
 - (2) **开括号**，将其入栈；
 - (3) **闭括号**时，先判断栈是否为空，**若为空**（**括号不匹配**），应作为错误进行异常处理，清栈退出；**若非空**，依次弹出栈中元素，并输出到后缀表达式序列中，直到遇到**一个开括号**为止，若没有遇到开括号，说明**括号不配对**，做异常处理，清栈退出；



中缀表达式 to 后缀表达式

- (4) **运算符**op (四则运算 + - * / 之一) 时
- (a) 循环 当 (**栈非空 and 栈顶不是开括号 and 栈顶运算符的优先级不低于输入的运算符的优先级**) 时, 反复
 将栈顶元素弹出, 放到后缀表达式序列中;
 - (b) 将输入的**运算符**压入栈内;
- (5) 读完全部输入符号, 若栈内仍有元素, 将其全部依次弹出, 置于后缀表达式序列的尾部。若弹出元素中有开括号出现, 则说明**括号不匹配**, 做异常处理, 清栈退出

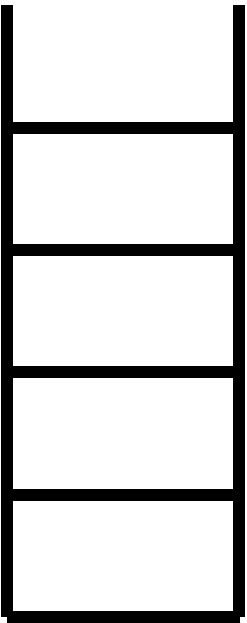


中缀表达式 to 后缀表达式

待处理中缀表达式

23 + (34 * 45) / (5 + 6 + 7)

栈状态的变化



输出的后缀表达式



中缀表达式求值

- 中缀表达式 \rightarrow 后缀表达式
- 直接计算 how to ?



栈与递归

- 函数的递归定义
- 主程序和子程序的参数传递
- 栈在实现函数递归调用中的作用



递归

- 作为数学和计算机学科的基础，递归是解决**复杂问题**的一个有力手段，可使问题的描述和求解变得 **简洁、清晰、易懂**
 - ✓ 符合人类解决问题的思维方式，能够描述和解决很多问题，许多程序设计语言提供支持机制
 - ✓ 往往比非递归算法更易设计，尤当问题本身或所涉及的数据结构就是递归定义时
- 计算机（编译程序）是如何将程序设计中的便于人类理解的递归程序转换为计算机可处理的非递归程序的？
 - ✓ 计算机只能按照指令集顺序执行



递归

- **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决这些子问题
- ✓ 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



递归示例： 阶乘函数

- 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？



递归示例：阶乘函数的程序实现

算法：阶乘的递归实现 $\text{Factorial}(n)$

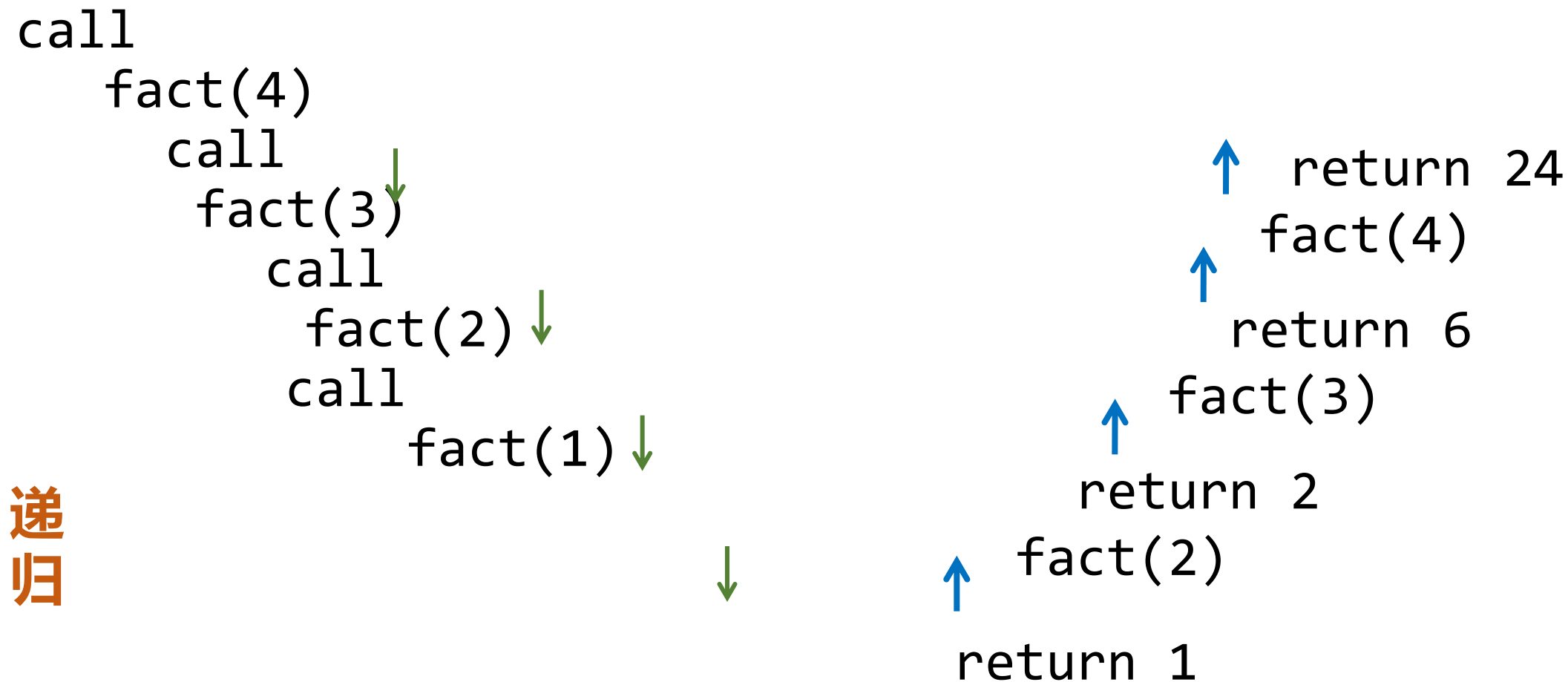
输入：整数 $n \geq 0$

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



函数执行过程图解：4!



递归

回归



函数调用与递归的实现

● 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



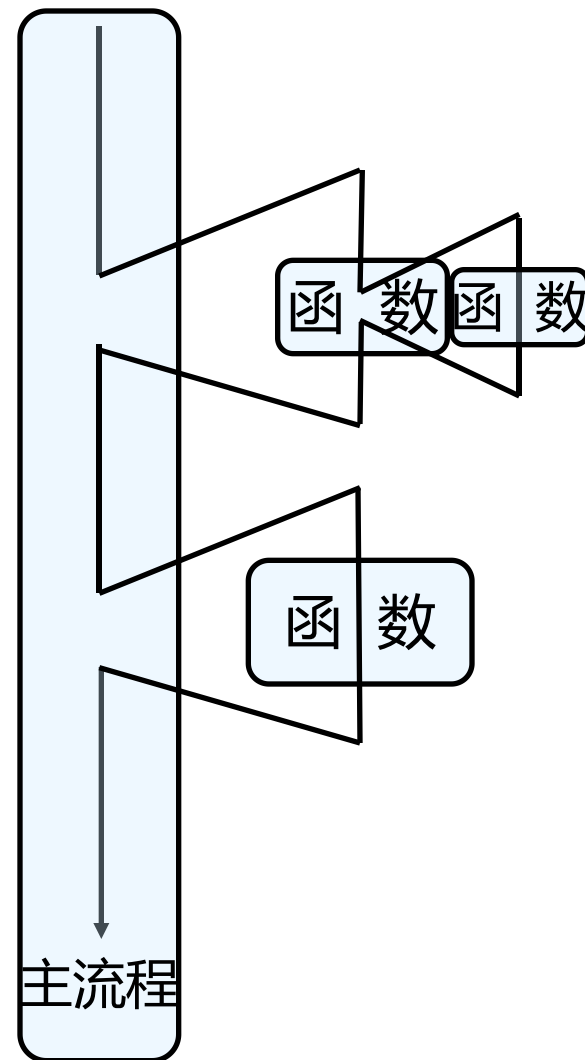
函数调用及返回的步骤

● 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

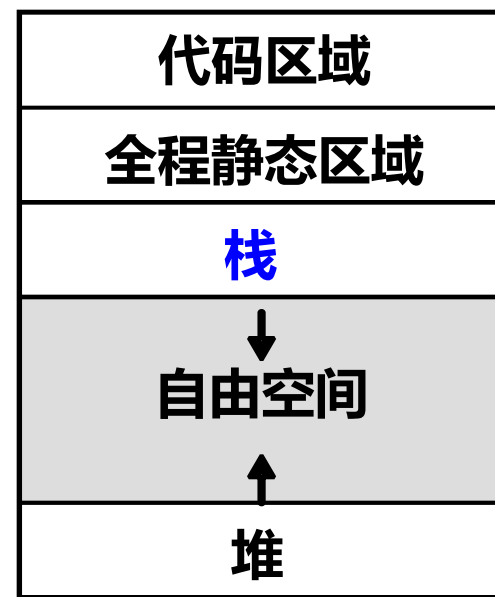
- ✓ 保存返回信息
- ✓ 释放数据区
- ✓ 控制转移到上级函数（主调用函数）





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





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

自变量 (参数) 空间
返回地址等 簿记信息空间
局部变量空间
局部/临时变量空间



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

- **一个函数可有多个不同的活动记录，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



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



阶乘的迭代实现

// 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;  
}
```




递归的再思考

- 递归出口
- 递归规则

意味?

意味?



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;
}
```



非递归程序的实现原理

- 与函数调用原理相同，只不过将由系统负责的保存工作信息变为由程序自己保存
 - ✓ 减少数据的冗余(节省局部变量的空间)，提高**存储效率**
 - ✓ 减少函数调用的处理以及冗余的重复计算，提高**时间效率**
- 程序/进程要完成的工作分成**两类**
 - ✓ **手头工作** 正在做的工作，须有其结束条件，不能永远做下去
 - ✓ **待完成工作** 某些不能一步完成的工作，必须 **暂缓完成**，保存在栈等数据结构中，须含有完成该项工作的所有必要信息
- 程序/进程须**有序**地完成各项工作
 - ✓ **手头工作**和**待完成工作**可**互相切换**
 - ✓ **待完成工作**必须转换成**手头工作**才能处理



队列应用：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。



队列应用：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



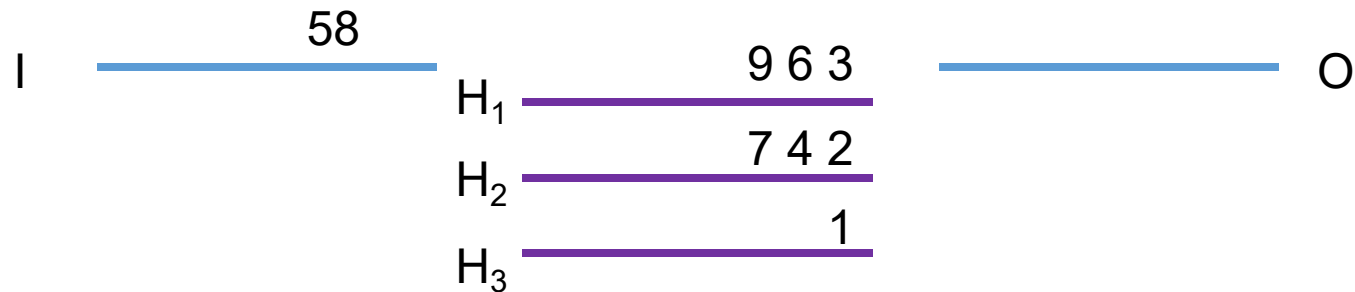
队列应用：车厢重排

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



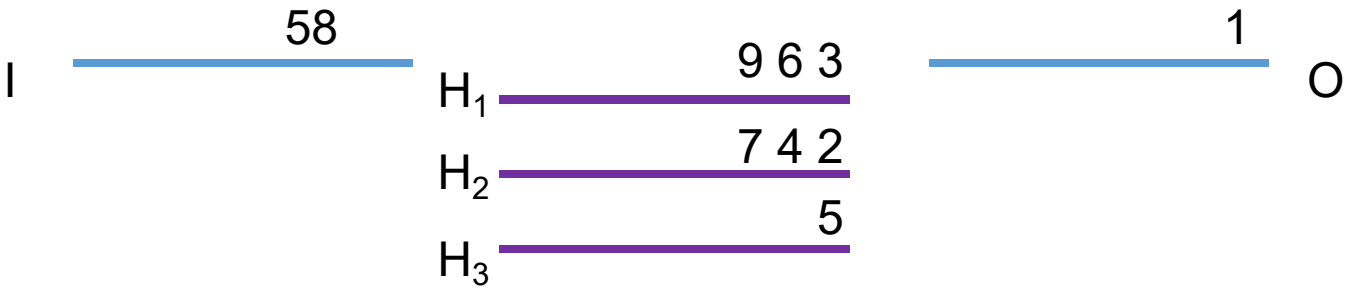
入轨的车厢按序分别进入缓冲轨 H1、H2、H3



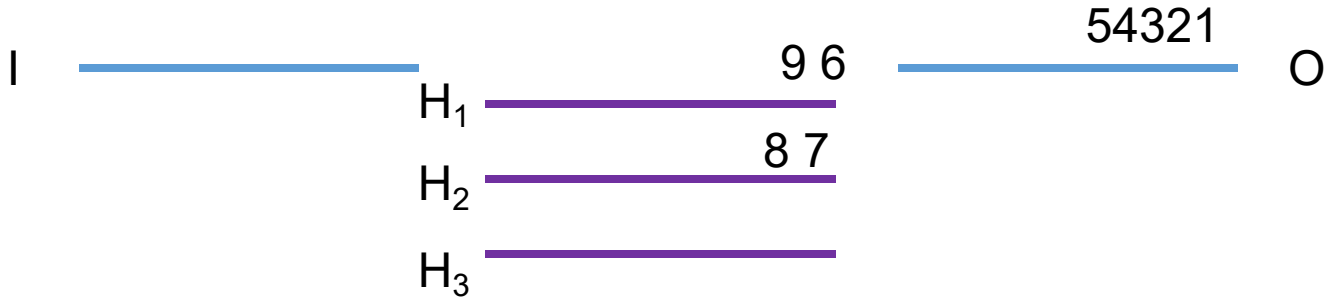


队列应用：车厢重排

车厢1从H₃出队进入出轨



车厢2、3、4按序分别从H₂、H₁、H₂出队进入出轨；
车厢8进入缓冲轨H₂，车厢5直接进入出轨





队列应用：车厢重排

各缓冲轨中车厢按编号进入出轨





队列应用：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. | | |  $front\_crg \leftarrow \text{GetFront}(buffer[j])$  // 查看队列  $j$  的头元素
13. | | | if  $front\_crg \neq \text{NIL}$  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
```



队列应用：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
```



队列应用：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
```



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



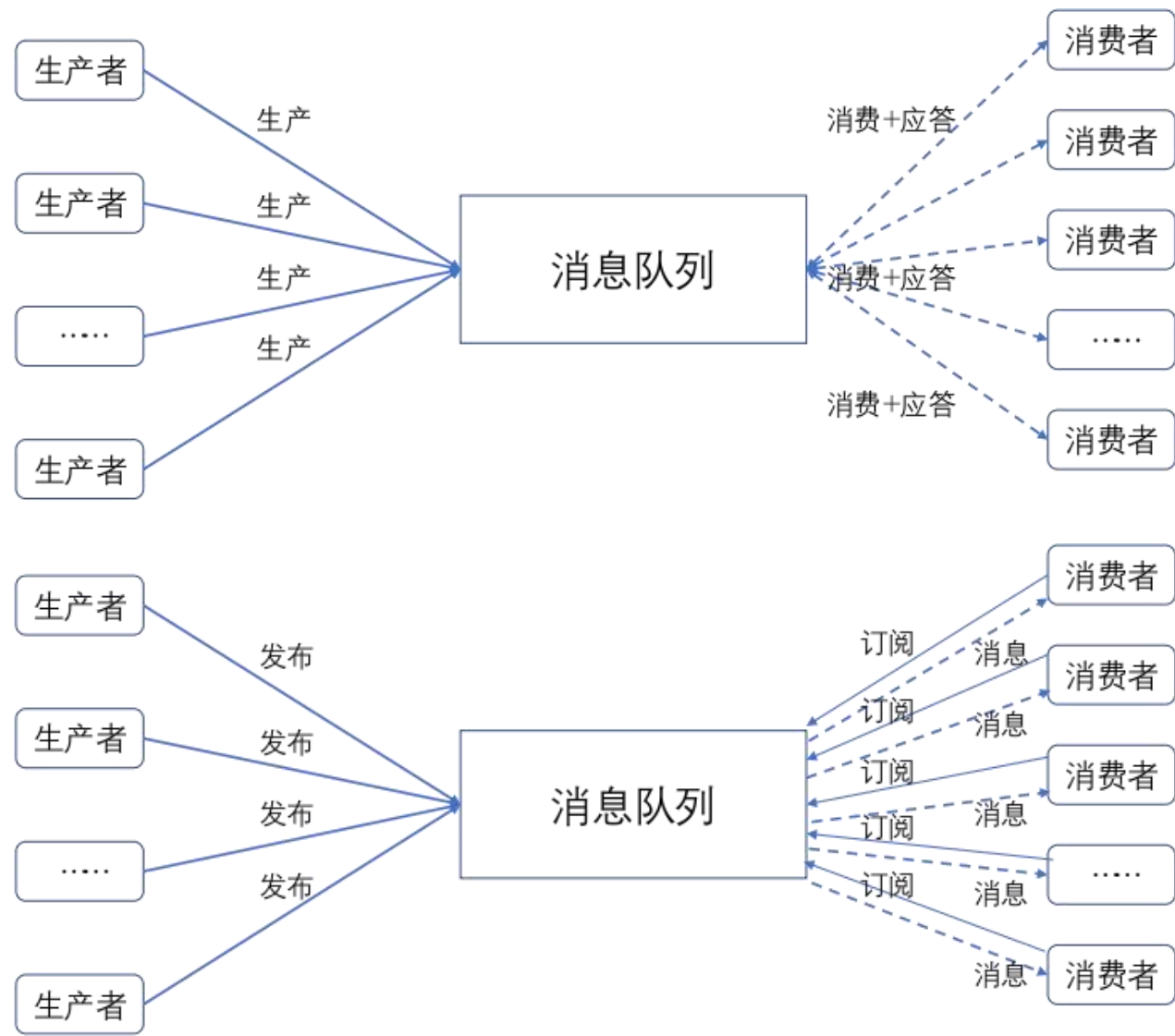
单调队列

- 一种元素具有严格单调性的队列，满足两个约束：
 1. 队列元素从头到尾的严格单调性。
 2. 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



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

本章主要介绍了两个应用广泛的数据结构，本质上均为**限制访问端口**的线性表

- 栈：插入和删除都限制在线性表的一端，形成**后进先出**的结构特点
 - ✓ 适用于具有递归特性的应用问题
- 队列：限制删除只在队首、插入只在队尾；形成**先进先出**的结构特点
 - ✓ 满足**先来先服务**的公平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
- 满足特定性质的栈与队列
 - ✓ 单调栈
 - ✓ 单调队列
- 消息队列

谢谢观看